HIGH PERFORMANCE AND SCALABLE SOFT SHARED STATE FOR NEXT-GENERATION DATACENTERS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Karthikeyan Vaidyanathan, M.Sc (Tech)

* * * * *

The Ohio State University

2008

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. F. Qin

Approved by

Adviser

Graduate Program in
Computer Science and
Engineering

ABSTRACT

In the past decade, with the increasing adoption of Internet as the primary means of electronic interaction and communication, web-based datacenters have become a central requirement for providing online services. Today, several applications and services have been deployed in such datacenters in a variety of environments including e-commerce, medical informatics, genomics, etc. Most of these applications and services share significant state information that are critical for the efficient functioning of the datacenter. However, existing mechanisms for sharing the state information are both inefficient in terms of performance and scalability, and non-resilient to loaded conditions in the datacenter. In addition, existing mechanisms do not take complete advantage of the features of emerging technologies which are gaining momentum in current datacenters.

This dissertation presents an efficient soft state sharing substrate that leverages the features of emerging technologies such as high-speed networks, Intel's I/OAT and multicore architectures to address the limitations mentioned above. Specifically, the dissertation targets three important aspects: (i) designing efficient state sharing components using the features of emerging technologies, (ii) understanding the interactions between the proposed components and (iii) analyzing the impact of the proposed components and their interactions with datacenter applications and services in terms of performance, scalability and resiliency.

Our evaluations with the soft state sharing substrate not only show an order of magnitude performance improvement over traditional implementations but also demonstrate the resiliency to loaded conditions in the datacenter. Evaluations with several datacenter applications also suggest that the substrate is scalable and has a low-overhead. The proposed substrate is portable across multiple modern interconnects such as InfiniBand, iWARP-capable networks like 10-Gigabit Ethernet both in LAN and WAN environments. In addition, our designs provide advanced capabilities such as one-sided communication, asynchronous memory copy operations, etc., even on systems without high-speed networks and I/OAT. Thus, our proposed designs, optimizations and evaluations demonstrate that the substrate is quite promising in tackling the state sharing issues with current and next-generation datacenters.

Dedicated to my parents, Amma and Appa;

to my siblings, Raju and Suba;

to my wife, Swaroopa

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for his support, patience and encouragement throughout the duration of my Ph.D. study. I'm greatly indebted to him for the time and efforts which he spent for my progress, especially during the tough periods in my dissertation.

I would also like to thank my committee members, Prof. P. Sadayappan and Prof. F. Qin for their valuable guidance and suggestions. I'm grateful to Dr. M. Schlansker, Dr. N. Jouppi, Dr. J. Mudigonda and Dr. N. Binkert for their support and guidance during my summer internships.

I'm thankful to my colleague Sundeep Narravula and Dr. Hyun-Wook Jin for the innumerable discussions and collaborations relating to this research. I would like to thank all my senior NOWLAB members for their patience and guidance: Dr. Pavan Balaji, Dr. Jiesheng Wu, Dr. Jiuxing Liu and Dr. Weikuan Yu. I'm lucky to have collaborated closely with my colleagues: Sundeep, Ping, Wei and Lei. I would also like to thank all my colleagues: Sandy, Gopal, Amith, Sayantan, Abhinav, Wei, Lei, Matt, Ping, Savitha, Qi, Hari, Ouyang, KGK, Tejus and Greg, for their discussions on technical and non-technical issues, as well as their friendship.

During all these years, I met many people at Ohio State, some of whom are very close friends, and I'm thankful for all their love and support: Sandeep Rao, Ajay, Shankar, Deepa, Vijay, Chubs and Nithya. I'm also thankful to my friends who

helped me during the tough periods in my dissertation: Damak, Prasanth gangavelli, Shanthi, Khandoo, Manoj, Gara and Nidhi.

Last, but definitely not the least, I would like to thank my wife, Swaroopa (Naga Vydyanathan), for her understanding and love during my dissertation. She has spent several sleepless nights proof reading all my papers and her support and encouragement was in the end what made this dissertation possible. Finally, I would like to thank my family members: amma, appa, raju, manni, suba, sai athimber, mano, ashwin, gayathri, aditi, jayashree, my wife's amma, appa, thatha, paatti, brother and cousins (aravind and srikanth). They all receive my deepest gratitude and love for their dedication and the many many years of support that provided the foundation for this work. I would not have had made it this far without their love and support.

VITA

September 1, 1978	Born - New Delhi, India.
August 1996 - July 2001	M. Sc (Tech) Information Systems, M. Sc (Tech) General Studies, Birla Institute of Technology and Sciences (BITS), Pilani, India.
August 2001 - July 2002	Research Associate, INSEAD, France.
August 2002 - December 2005	Graduate Teaching/Research Associate, The Ohio State University.
June 2006 - September 2006	Summer Intern, HP Labs, Palo Alto, CA.
June 2007 - September 2007	Summer Intern, HP Labs, Palo Alto, CA.
January 2006 - Present	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

K. Vaidyanathan, P. Lai, S. Narravula and D. K. Panda, "Optimized Distributed Data Sharing Substrate for Multi-Core Commodity Clusters: A Comprehensive Study with Applications", IEEE International Symposium on Cluster Computing and the Grid, May 2008.

P. Lai, S. Narravula, K. Vaidyanathan and D. K. Panda, "Advanced RDMA-based Admission Control for Modern Data-Centers", IEEE International Symposium on Cluster Computing and the Grid, May 2008.

K. Vaidyanathan, L. Chai, W. Huang and D. K. Panda, "Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT", IEEE International Conference on Cluster Computing, September 2007.

S. Narravula, A. Mamidala, A. Vishnu, K. Vaidyanathan and D. K. Panda, "High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations", IEEE International Symposium on Cluster Computing and the Grid (CCGrid), May 2007.

K. Vaidyanathan and D. K. Panda, "Benefits of I/O Acceleration Technology (I/OAT) in Clusters", IEEE International Symposium on Performance Analysis of Systems and Software, April 2007.

K. Vaidyanathan, W. Huang, L. Chai and D. K. Panda, "Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT", International Workshop on Communication Architecture for Clusters, March 2007.

K. Vaidyanathan, S. Narravula and D. K. Panda, "DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over Modern Interconnects", IEEE International Symposium on High Performance Computing, December 2006.

H. -W. Jin, S. Narravula, K. Vaidyanathan and D. K. Panda, "NemC: A Network Emulator for Cluster-of-Clusters", IEEE International Conference on Communication and Networks, October 2006.

K. Vaidyanathan, H. -W. Jin and D. K. Panda, "Exploiting RDMA operations for Providing Efficient Fine-Grained Resource Monitoring in Cluster-based Servers", Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations and Technologies, September 2006.

S. Narravula, H. -W. Jin, K. Vaidyanathan and D. K. Panda, "Designing Efficient Cooperative Caching Schemes for Multi-Tier Data-Centers over RDMA-enabled Networks", IEEE International Symposium on Cluster Computing and the Grid, May 2006.

P. Balaji, H. -W. Jin, K. Vaidyanathan and D. K. Panda, "Supporting iWARP Compatibility and Features for Regular Network Adapters", Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations and Technologies, September 2005.

H. -W. Jin, S. Narravula, G. Brown, K. Vaidyanathan, P. Balaji and D. K. Panda, "Performance Evaluation of RDMA over IP: A Case Study with the Ammasso Gigabit Ethernet NIC", Workshop on High Performance Interconnects for Distributed Computing, July 2005.

S. Narravula, P. Balaji, K. Vaidyanathan, H. -W. Jin and D. K. Panda, "Architecture for Caching Responses with Multiple Dynamic Dependencies in Multi-Tier Data-Centers over InfiniBand", IEEE International Symposium on Cluster Computing and the Grid, May 2005.

P. Balaji, S. Narravula, K. Vaidyanathan, H. -W. Jin and D. K. Panda, "On the Provision of Prioritization and Soft QoS in Dynamically Reconfigurable Shared Data-Centers over InfiniBand", IEEE International Symposium on Performance Analysis of Systems and Software, March 2005.

K. Vaidyanathan, P. Balaji, H. -W. Jin and D. K. Panda, "Workload-driven Analysis of File Systems in Shared Multi-Tier Data-Centers over InfiniBand", Workshop on Computer Architecture Evaluation using Commercial Workloads, February 2005.

P. Balaji, K. Vaidyanathan, S. Narravula, K. Savitha, H. -W. Jin and D. K. Panda, "Exploiting Remote Memory Operations to Design Efficient Reconfiguration for Shared Data-Centers over InfiniBand", Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations and Technologies, September 2004.

P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu and D. K. Panda, "Sockets Direct Procotol over InfiniBand in Clusters: Is it Beneficial?", IEEE International Symposium on Performance Analysis of Systems and Software, March 2004.

S. Narravula, P. Balaji, K. Vaidyanathan, S. Krishnamoorthy, J. Wu and D. K. Panda, "Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand", Workshop on System Area Networks, February 2004.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Prof. D. K. Panda
Software Systems	Prof. P. Sadayappan
Computer Networks	Prof. D. Xuan

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Over the past several years, there has been an incredible growth of applications in the fields of e-commerce [2, 8], financial services [17, 21, 26, 7], search engines [9, 57, 25, 15, 4], medical informatics [24], etc. Indeed, it is almost impossible for us to imagine our daily life without these applications (e.g., accessing a web page, buying a book from an online book store). Typically, these applications are hosted through a web-based datacenter [85, 55] (e.g., Google [9], Amazon [2], IBM [11], HP [10]). Such datacenters are not only becoming extremely common today but are also increasing exponentially in size, currently ranging to thousands of nodes. With technology trends, the ability to store and share the datasets that these applications generate is also increasing. Accordingly, several distributed applications such as STORM [24], database query processing [23, 16], Apache [54] and services such as load-balancing [87], resource monitoring [78], reconfiguration [31], caching [71] and several others have been developed and deployed in such environments.

Typically, these applications and services exchange key information at multiple sites (e.g., system state, versioning and timestamps of data/meta-data, current system load, locks currently held, coherency and consistency information and several

others). As demonstrated by recent literature [28, 71], managing this shared information becomes critical for the efficient functioning of the datacenter. However, for the sake of availability and performance, programmers typically use ad-hoc messaging protocols for maintaining this shared information, which is both complex and cumbersome, as mentioned by Tang et. al [88]. Researchers [28, 42, 89, 79] in the past have proposed simpler state sharing abstractions to hide these complexities. However, these abstractions do not address issues such as load resiliency, low-overhead access and efficient resource utilization which are becoming critical in current datacenters to increase the performance and scalability. In addition, these abstractions do not take complete advantage of features of emerging technologies which are currently gaining momentum in next-generation datacenters.

On the other hand, the System Area Network (SAN) technology is making rapid advances during the recent years. SAN interconnects such as InfiniBand (IBA) [61] and 10-Gigabit Ethernet [60] not only provide high performance but also provide a range of novel features such as Remote Direct Memory Access (RDMA), protocol offload and several others. Emerging technology such as Intel's I/O Acceleration Technology (I/OAT) provides an asynchronous DMA engine to offload the memory copy operations. Multicore architectures provide several processing elements within a node which further opens up new ways to improve the datacenter performance and scalability. This dissertation presents an efficient soft state sharing substrate that leverages the features of these emerging technologies to address the limitations mentioned above. Specifically, the dissertation targets three important aspects: (i) designing efficient state sharing components using the features of emerging technologies, (ii) understanding the interactions between the proposed components and (iii)

analyzing the impact of the proposed components and its interactions with several applications and services. Applications and services can take advantage of the proposed state sharing substrate with minimal effort.

The rest of this Chapter is organized as follows. First we provide an overview of datacenter and emerging technologies in datacenter environments. Next, we present the limitations associated with existing state sharing mechanisms in datacenter applications and services. Following that, we present the problem statement. Finally, we provide an overview of this dissertation.

1.1 Overview of Datacenters

Figure 1.1 shows the common components involved in designing a web-based datacenter. Requests from clients (over Wide Area Network (WAN)) first pass through a load balancer which attempts to spread the requests across multiple front-end proxies (Tier 0). These proxies determine if the request can be satisfied by a static (time invariant) content web server or if the request requires more complex dynamic content generation. The proxies also usually do some amount of caching of both static and dynamic content. Tier 1 is generally the most complex as it is responsible for all application-specific processing such as performing an online purchase or building a query to filter some data. At the back end of the processing stack (Tier 2) is the data repository/ database server with the associated storage. This is the prime repository of all the content that is delivered or manipulated.

Figure 1.2(a) illustrates the overall strategy for processing workload through the datacenter tiers [85]. For accesses/queries involving static (time invariant) data, it is beneficial to use caching schemes (with appropriate proxy and cache servers) to

satisfy maximum of the requests in Tier 0. Those requests which can not be satisfied in Tier 0 are forwarded to the next tiers. In these situations, the application and database servers may need to generate/access new data to serve this request. As the requests move inside the datacenter tiers, more penalty is involved in terms of time. As a result of query processing in the application and database tiers, the load on these tiers naturally increases.

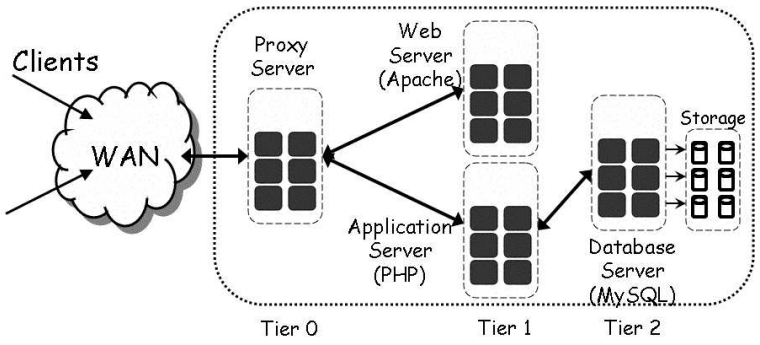


Figure 1.1: Datacenter Architecture

1.1.1 Datacenter Applications

With technology trends, several applications are developed and deployed to enable sharing the datasets generated in scientific and commercial environments in these datacenters. Web servers typically host applications such as Apache [54], Squid [52] to process the static client requests and perform higher-level functionalities such as load-balancing, caching, etc. Application servers process CGI, Java or PHP scripts using Apache [54], RMI [90] and several other applications. Database servers host applications such as MySQL [16], DB2 [84] which internally use many database query

processing service such as R-Tree [59], B-Trees [32], etc. In biomedical informatics, applications such as STORM [24] are currently deployed to perform different data queries and transfer the data from storage nodes to the front-end nodes. To provide fault-tolerance, datacenters also use check-pointing applications [97] to monitor and replay transactions in case of a failure. Apart from these, there are several other applications in the fields of search engines, financial services, etc., that are deployed in current datacenters.

1.1.2 Datacenter Services

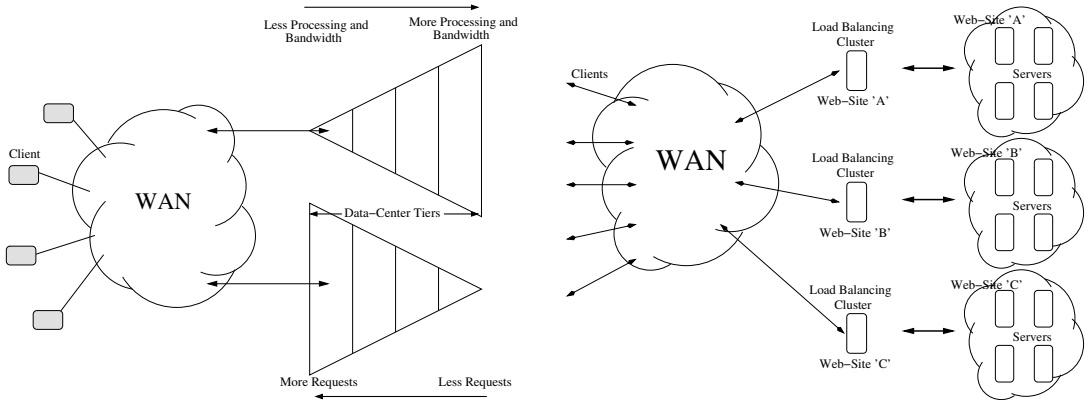
In addition to the applications, datacenters also require intelligent support for services for efficient functioning of the datacenter. Services such as active resource adaptation service including reconfiguration [31] and admission control [64] deal with scalable management of various system resources. Other services such as resource monitoring [94] actively monitors the resource usage of system resources and helps higher-level services in identifying the bottleneck resources and alleviating such bottlenecks as they occur. Caching services such as active caching [71] and cooperative caching [72] deal with efficient caching techniques for both static and dynamic (time variant) content.

1.1.3 Shared Datacenter Environments

A clustered datacenter environment essentially tries to utilize the benefits of a cluster environment (e.g., high performance-to-cost ratio) to provide the services requested in a datacenter environment (e.g., web hosting, transaction processing).

Researchers have also proposed and configured datacenters to provide multiple independent services, such as hosting multiple web-sites, forming what is known as shared datacenters.

Figure 1.2(b) shows a higher level layout of a shared datacenter architecture hosting multiple web-sites. External clients request documents or services from the datacenter over the WAN/Internet through load-balancers using higher level protocols such as *HTTP*. The load-balancers on the other hand serve the purpose of exposing a single IP address to all the clients while maintaining a list of several internal IP addresses to which they forward the incoming requests based on a pre-defined algorithm (e.g., round-robin).



(a) Communication and Computation Issues

(b) Shared Datacenter Environment

Figure 1.2: Datacenter Issues

1.2 Overview of Emerging Technologies

In this section, we first provide a brief background on high-performance networks and its capabilities. Next, we present the capabilities of I/O Acceleration Technology. Finally, we discuss the features of multicore architectures.

1.2.1 High Performance Networks

Several high-performance networks such as InfiniBand [61], 10-Gigabit Ethernet [60], etc., mainly aim at reducing the system processing overhead by decreasing the number of copies associated with a message transfer and removing the kernel from the critical message passing path. This is achieved by providing the consumer applications direct and protected access to the network.

The InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. In a typical IBA cluster, switched serial links connect the processing nodes and the I/O nodes. The compute nodes are connected to the IBA fabric by means of Host Channel Adapters (HCAs). IP is one type of traffic that could use this interconnect. IPoIB provides standardized IP encapsulation over IBA fabrics as defined by the IETF Internet Area IPoIB working group [14]. IBA defines a semantic interface called as Verbs or Open Fabrics for the consumer applications to communicate with the HCAs. The specification for Verbs includes a queue-based interface, known as a Queue Pair (QP), to issue requests to the HCA. Figure 1.3 illustrates the InfiniBand Architecture model.

Each Queue Pair is a communication endpoint. A Queue Pair (QP) consists of the send queue and the receive queue. Two QPs on different nodes can be connected

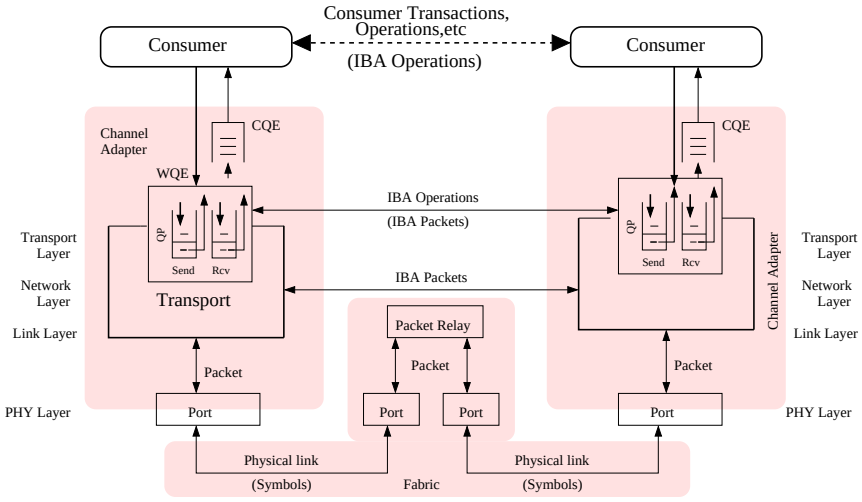


Figure 1.3: InfiniBand Architecture (Courtesy InfiniBand Specifications)

to each other to form a logical bi-directional communication channel. An application can have multiple QPs. Communication requests are initiated by posting Work Queue Requests (WQRs) to these queues. Each WQR is associated with one or more pre-registered buffers from which data is either transferred (for a send WQR) or received (receive WQR). The application can either choose the request to be a Signaled (SG) request or an Un-Signaled request (USG). When the HCA completes the processing of a signaled request, it places an entry called as the Completion Queue Entry (CQE) in the Completion Queue (CQ). The consumer application can poll on the CQ associated with the work request to check for completion. There is also the feature of triggering event handlers whenever a completion occurs. For un-signaled requests, no kind of completion event is returned to the user. However, depending on the implementation, the driver cleans up the Work Queue Request from the appropriate Queue Pair on completion.

RDMA Communication Model: IBA supports two types of communication semantics: channel semantics (send-receive communication model) and memory semantics (RDMA communication model). In channel semantics, every send request has a corresponding receive request at the remote end. Thus there is one-to-one correspondence between every send and receive operation. In memory semantics, RDMA operations are used. These operations are transparent at the remote end since they do not require the remote end to involve in the communication. Therefore, an RDMA operation has to specify both the memory address for the local buffer as well as that for the remote buffer. There are two kinds of RDMA operations: RDMA Write and RDMA Read. In an RDMA write operation, the initiator directly writes data into the remote node's user buffer. Similarly, in an RDMA Read operation, the initiator reads data from the remote node's user buffer.

Atomic Operations Over IBA: In addition to RDMA, the reliable communication classes also optionally provide atomic operations directly against the memory at the end node. Atomic operations are posted as descriptors as in any other type of communication. However, the operation is completely handled by the HCA. This feature is currently available only in IBA. The atomic operations supported are Fetch-and-Add and Compare-and-Swap, both on 64-bit data. The Fetch-and-Add operation performs an atomic addition at the remote end. The Compare-and-Swap is used to compare two 64-bit values and swap the remote value with the data provided if the comparison succeeds. Atomics are effectively a variation of RDMA: a combined write and read RDMA, carrying the data involved as immediate data. Two different levels of atomicity are optionally supported: atomic with respect to other operations on

a target channel adapter; and atomic with respect to all memory operation of the target host and all channel adapters on that host.

1.2.2 I/O Acceleration Technology (I/OAT)

I/O Acceleration Technology introduced by Intel mainly targeted the datacenter environment in an attempt to reduce the server-side overheads in protocol processing of the TCP/IP stack. In this context, Intel's I/OAT [58, 68, 81] introduced an Asynchronous DMA Copy Engine (ADCE) to offload the data copy operation. ADCE is implemented as a PCI-enumerated device in the chipset and has multiple independent DMA channels with direct access to main memory. When the processor requests a block memory operation from the engine, it can then asynchronously perform the bulk data transfer with no host processor intervention. When the engine completes a copy, it can optionally generate an interrupt.

DMA Engine: Figure 1.4 illustrates the basic architecture of a copy execution using a CPU vs using a DMA copy engine. As mentioned in [103], utilizing a copy engine for bulk data transfer offers several benefits:

1. *Reduction in CPU Resources and Better Performance:* Memory copies are usually implemented as a series of load and store instructions through registers. Data is fetched onto the cache and then onto the registers. Typically, the CPU performs the copy by register size which is 32 or 64 bit long. On the other hand, using a copy engine, memory copies can be done at a faster rate (close to block sizes) since it directly operates with main memory. Further, the load and store instructions used in CPU-based copies may end up occupying the CPU resources, limiting the CPU to

not look far ahead in the instruction window. Copy engines can help in freeing up CPU resources so that other useful instructions can be executed.

2. *Computation-Memory Copy Overlap:* Since the memory-to-memory copy operation can be performed without host CPU intervention using an asynchronous copy engine, we can achieve better overlap with memory copies. This is similar to DMA operation where data is transferred directly between the memory and device which is commonly used by networks such as InfiniBand, 10-Gigabit Ethernet, etc.

3. *Avoiding Cache Pollution Effects:* Large memory copies can pollute the cache significantly. Unless the source or destination buffers are needed by the application, allocating this buffer in the cache may result in polluting the cache as it can evict other valuable resources in the cache. As mentioned in [103], cluster applications such as web servers do not touch the data immediately even after completing the memory copy. Using a copy engine in this case, results in avoiding any cache pollution as it can directly perform the copy without getting the data onto the cache.

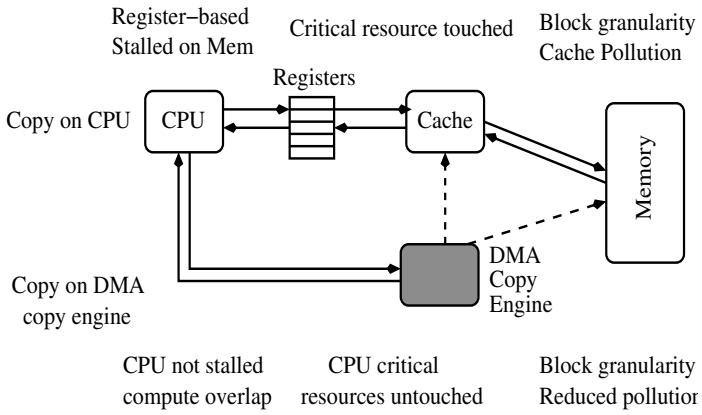


Figure 1.4: Copy execution on CPU vs Copy Engines [103]

Though ADCE offers several benefits, the following issues need to be taken care of. First, the memory controller uses physical addresses, so a single transfer cannot span discontinuous physical pages. Hence, memory operations should be broken up into individual page transfers. Secondly, memory copies whose source and destination overlap should be carefully handled. Applications need to schedule such operations in an appropriate order so as to preserve the semantics of the operation. Lastly, the copy engine must maintain cache coherence immediately after data transfer. Data movement performed by the memory controller should not ignore the data stored in the processor cache, potentially requiring a cache coherence transaction on the bus.

1.2.3 Multicore Architectures

Emerging trends in processor technology has led to Multicore Processors (also known as Chip-level Multiprocessing or CMP) which provide large number of cores on a single node thus increasing the processing capability of current-generation systems. Dual-core architectures (two cores per die) are widely available from various industry leaders including Intel, AMD, Sun (with up to 8 cores) and IBM. The negligible cost associated with placing an extra processing core on the same die has allowed these architectures to increase the capabilities of applications significantly. Quad-core (four cores per die) from Intel and AMD are also available currently. Recently, Intel has announced that it will be introducing an 80-core die [51] within the next five years. Other industries are expected to follow this trend. The emergence of several number of cores within a single node further opens up news way for dedicating one or more of these cores for performing specialized functions.

1.3 Limitations of Existing State Sharing Mechanisms in Datacenters

In this section, we present the limitations of state sharing mechanisms in datacenter applications and services.

Existing datacenter applications such as Apache [54], MySQL [16], STORM [24], etc., implement their own data management mechanisms for state sharing and synchronization. Applications like database servers communicate and synchronize frequently with other database servers to satisfy the coherency and consistency requirements of the data being managed. Web servers implement complex load-balancing mechanisms at multiple sites based on current system load on the back-end nodes, request patterns, etc. Active resource adaptation service implement locking mechanisms to reconfigure nodes serving one website to another in a transparent manner. Resource monitoring services, on the other hand, frequently acquire the load information on remote nodes to assist higher-level services such as reconfiguration, admission control, etc. Unfortunately, the mechanisms used to share the state information in these applications and services have been developed in an ad-hoc manner using two-sided communication protocols such as TCP/IP, which makes the sharing of state information between applications extremely inefficient. Such protocols are known to have high latency, low bandwidth and high CPU utilization limiting the maximum capacity (in terms of requests the datacenter can handle per unit time) of datacenters. Further, the processing of the TCP/IP stack includes additional overheads such as multiple memory copies, context switches, interrupts, etc. In addition, these applications and services interact with other datacenter system software using System V IPC mechanisms for both communication and synchronization. Such mechanisms

are also known for multiple memory copies, context switches, heavy operating system involvement and involves the host CPU for protocol processing.

Apart from communication and synchronization, datacenter applications and services exchange key information at multiple sites (e.g, versioning and timestamps of cached copies, coherency and consistency information, current system load). However, as mentioned earlier, programmers use ad-hoc messaging protocols for maintaining this shared information. Unfortunately, as mentioned in [88], the code devoted to these protocols account for a significant fraction of overall application size and complexity. As system sizes increase, this fraction is likely to increase and cause significant overheads.

Furthermore, many of the datacenter services periodically monitor the resources used in the cluster and use this information to make various decisions including whether a request should be admitted, what resources should be allotted to the request, adapt resources provided to different classes of requests, etc. Though these approaches are generic and applicable for all environments, the main drawback with them is that they rely on coarse-grained monitoring of resources in order to avoid the overheads associated with fine-grained monitoring. Accordingly, they base their techniques on the assumption that the resource usage is consistent through the monitoring granularity (which is in the order of seconds in most cases). On the other hand, as demonstrated by recent literature [46], the resource usage of requests is becoming increasingly divergent making this assumption no longer valid.

Finally, existing datacenter applications are multi-threaded. Each thread uses a request-process-response model and multiple threads handle multiple requests simultaneously. Thus, the inefficiencies of protocol overheads using TCP/IP and System

V IPC mechanisms magnify when several threads (typically up to 256 to 512 threads per node) perform communication and synchronization operations simultaneously.

1.4 Problem Statement

To address the limitations mentioned in Section 1.3, it is necessary to design an efficient state sharing mechanism that provides high-performance and scalability to datacenter applications and services. Further, as these datacenters are increasingly relying on modern system and network architectures including multicore systems, high-speed interconnects and I/O acceleration technologies, it is important that the state sharing mechanisms be able to exploit these added features to improve the datacenter performance and scalability. In addition, it is also important that the state sharing mechanisms utilize the features of these technologies and translate them to advanced datacenter capabilities such as load resiliency, low-overhead and fine-grained access, data/lock distribution and several others for applications and services. Finally, since multiple datacenter system software maintain and manage the same state information at several tiers, the complexity of the applications can be significantly reduced if we can decouple the state sharing component from applications and manage the state sharing independently.

In this dissertation, we aim to address the above mentioned issues by designing an efficient state sharing substrate. Specifically, we will address the following questions:

- Can we leverage the features of high performance networks in designing an efficient state sharing mechanism that addresses the limitations associated with communication protocols such as TCP/IP? - Current datacenters use TCP/IP

communication protocols to exchange significant state information. Such protocols involve multiple memory copies, context switches and interrupts which not only reduce the performance but also heavily involve the host CPU for protocol processing. It would be beneficial to exploit the advanced features of modern networks to alleviate such issues in state sharing with datacenter applications and services.

- Can the state sharing mechanism address the limitations of inter-process communication protocols using the advanced features of I/OAT? - Datacenters also employ System V IPC mechanisms to communicate and synchronize between processes on the same node. However, such mechanisms not only include multiple memory copies but also involve the host CPU, operating system in terms of context switches, protocol processing which can affect the performance and scalability of datacenter. It remains to be investigated whether offloading such inter-process communication operations using I/OAT can be beneficial to datacenter environments.

- How can multicore architectures help in improving existing state sharing mechanisms? - As mentioned earlier, there is a growing need for communication protocols that do not involve the application for protocol processing, memory copy operations, etc. One approach to solve this issue is to offload these operations to a dedicated external device such as the network adapter or the DMA engine. If modern systems have multiple processing elements, a completely radical approach is whether such operations can be dedicated to one or more of

the processing elements. Such an approach can not only provide advance capabilities to datacenters independent of any networking and I/O technologies but can also demonstrate how other datacenter-specific operations can be onloaded to multiple processing elements.

- Can we design and analyze the interactions of multiple emerging technologies for state sharing in datacenters? - Modern systems have one or more of the technologies such as high-speed networks, I/OAT and multicore architectures. The interactions of these technologies such as the impact of multicore systems with state sharing using high-performance networks, memory copies using DMA engines and the impact of DMA engine on network processing remains to be studied. It would be beneficial to explore such possibilities and further enhance the state sharing mechanism in datacenter applications and services.

1.5 Dissertation Overview

Keeping the characteristics and the needs of state sharing in datacenters in mind, we propose an efficient state sharing substrate leveraging the novel features of emerging technologies such as high-speed networks, I/OAT and multicore architectures. Figure 1.5 shows the various components of our proposed state sharing substrate. Broadly, in the figure, all the white boxes are the components which exist today. The dark boxes are the ones which need to be designed to efficiently support state sharing in next-generation datacenters. Specifically, we propose to design the following components and evaluate its impact with datacenter applications and services:

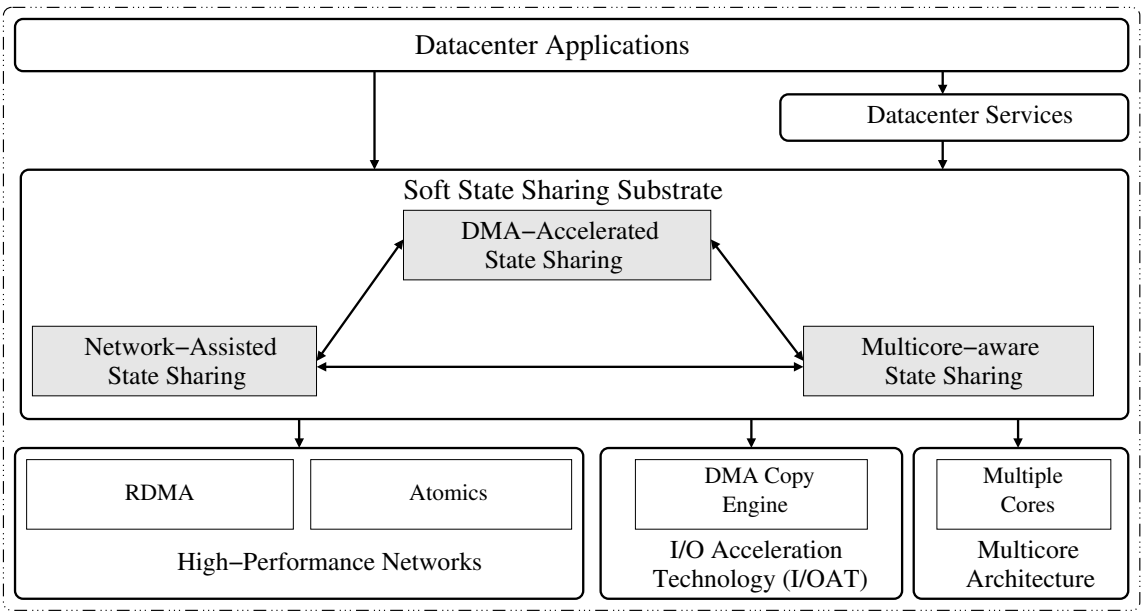


Figure 1.5: Proposed State Sharing Substrate

- **Network-Assisted State Sharing using high-speed networks:** To avoid the bottlenecks associated with TCP/IP based communication such as multiple memory copies, high CPU utilization, we have designed and developed a network-assisted state sharing component using the features of high-speed networks such as RDMA and atomic operations. The proposed component has been demonstrated to work with multiple interconnects such as InfiniBand, 10-Gigabit Ethernet and including iWARP-enabled networks such as Ammasso. Detailed designs and evaluations of this component is discussed in Chapter 2. Figure 1.6 shows the basic idea of a network-assisted state sharing substrate with several processes (proxy servers) writing and several application servers reading certain information simultaneously.

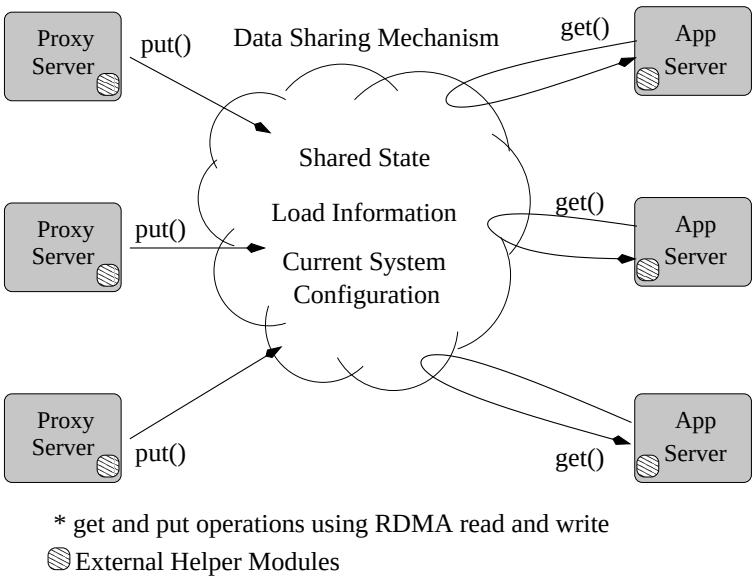
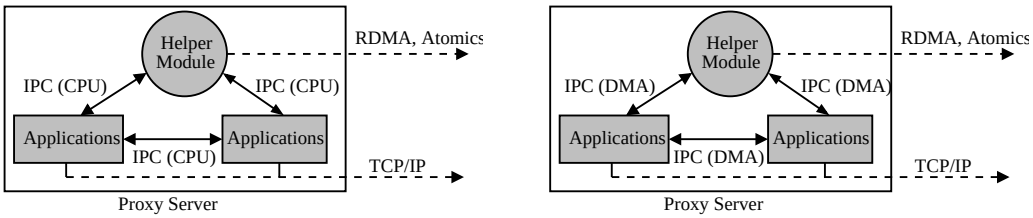


Figure 1.6: Network-Assisted State Sharing Mechanism

- **DMA-Accelerated State Sharing using I/OAT:** We have designed and developed efficient state sharing mechanisms that not only reduce the number of memory copies involved in System V IPC mechanisms but also transparently overlap the computation with memory copy operations with reduced CPU utilization using an asynchronous DMA copy engine. We discuss the design challenges and our solutions in Chapter 3. Figure 1.7 shows the CPU-based and DMA-based IPC communication mechanism that is used to share information across applications and helper module within one node.

- **Multicore-aware State Sharing using Multicore Architectures:** Following an onloading approach, we have designed and developed efficient memory copy routines, communication and synchronization mechanisms in state sharing for large-scale multicore environments. This component is discussed in detail



(a) IPC Communication using CPU (b) IPC Communication using DMA

Figure 1.7: DMA-Accelerated State Sharing Mechanism

in Chapter 4. In this approach, as shown in Figure 1.8, the communication related operations are completely dedicated to one or more processing elements and the proposed solution provides advanced capabilities that is independent of the high-speed networking and I/O technologies.

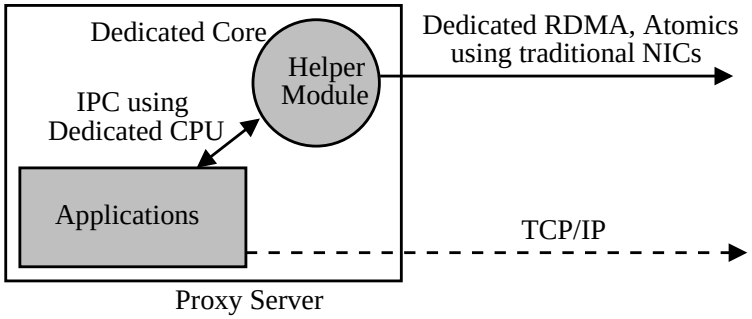


Figure 1.8: Multicore-aware State Sharing Mechanism

- **Multicore-aware, DMA-Accelerated State Sharing:** We have further designed and analyzed the different combinations of these two technologies in

enhancing the System V IPC communication in state sharing, as shown in Figure 1.9. Detailed designs and evaluations of this component are discussed in Chapter 5.

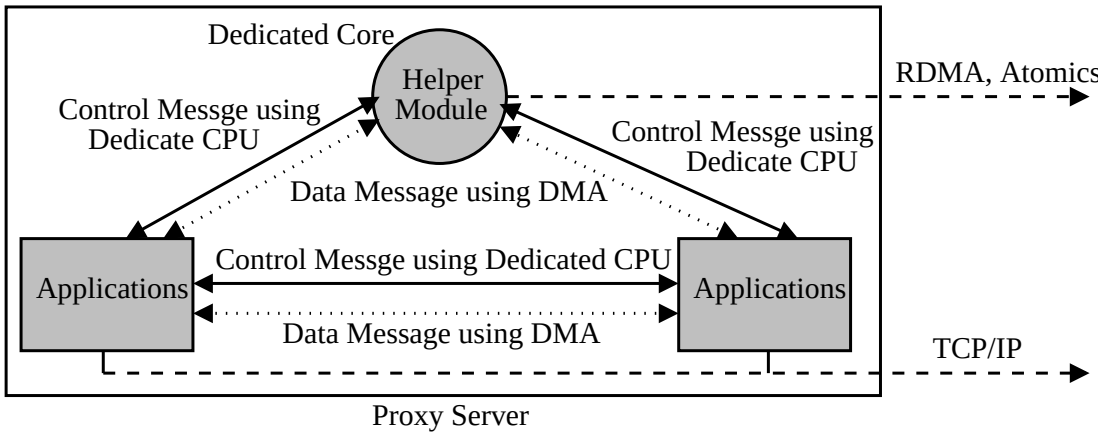
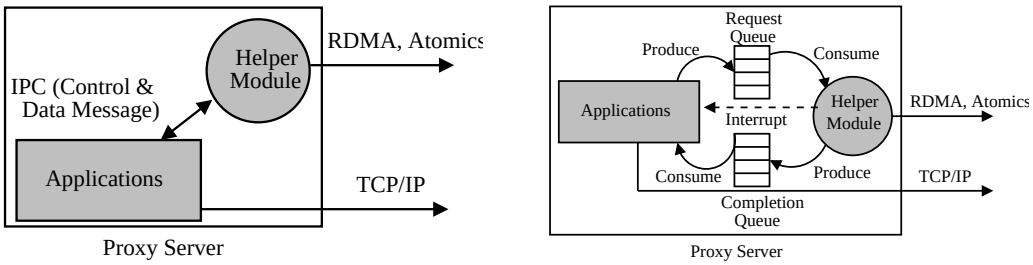


Figure 1.9: Multicore-aware, DMA-Accelerated State Sharing Mechanism

- **Multicore-aware, Network-Assisted State Sharing:** The proposed network-assisted state sharing can be further enhanced with the emergence of multicore systems by using a combination of request and response queues for communication as shown in Figure 1.10. We have proposed several different approaches in utilizing the capabilities of multicore systems and discuss our detailed designs and optimizations in Chapter 6.

- **DMA-Accelerated, Network-Assisted State Sharing:** Apart from enhancing the System V IPC communication, the asynchronous DMA engine can also enhance the network communication by offloading the copy operations from



(a) Communication using IPC Mechanism

(b) Communication using Request/Response Queues

Figure 1.10: Multicore-aware, Network-Assisted State Sharing Mechanism

the host CPU to the DMA engine. We show the benefits of such an approach using detailed evaluations in Chapter 7.

While several of these technologies and services provide features and high-performance, it is important to understand how these benefits translate to application improvement. We perform detailed analysis with several datacenter applications such as Apache, STORM, database query processing using workloads such as RUBiS auction benchmark [20], TPC-W e-commerce workload [34], Zipf-like workload [104] and understand the benefits in terms of response time, transactions processed, resource utilization and several others. In addition, we demonstrate the capability of our proposed state sharing substrate with datacenter services such as resource monitoring and reconfiguration. The performance benefits achieved with these datacenter applications and services are discussed in detail under each of the proposed state sharing chapters. In Chapter 8, we highlight several scenarios and system environments where

each of the proposed state sharing components can be applicable. Conclusions and future work are discussed in Chapter 9.

CHAPTER 2

NETWORK-ASSISTED STATE SHARING USING HIGH-SPEED NETWORKS

In this Chapter, we utilize the advanced features of high-performance networks for designing an efficient state sharing substrate. Figure 2.1 shows the various components of network-assisted state sharing substrate. Broadly, in the figure, we focus on the colored boxes for designing efficient network-assisted state sharing components and understanding its benefits with datacenter applications and services. The dark colored boxes show the features and technologies that we utilize and the light colored boxes show the proposed components and datacenter system software evaluated.

2.1 Background and Related Work

As mentioned in Chapter 1, existing datacenter applications such as Apache, MySQL, etc., implement their own data management mechanisms for state sharing and synchronization. Web servers implement complex load-balancing mechanisms based on current system load, request patterns, etc. To provide fault-tolerance, check-pointing applications save the program state at regular intervals for reaching a consistent state. Many of these mechanisms are performed at multiple sites in a cooperative fashion. Since communication and synchronization are an inherent part

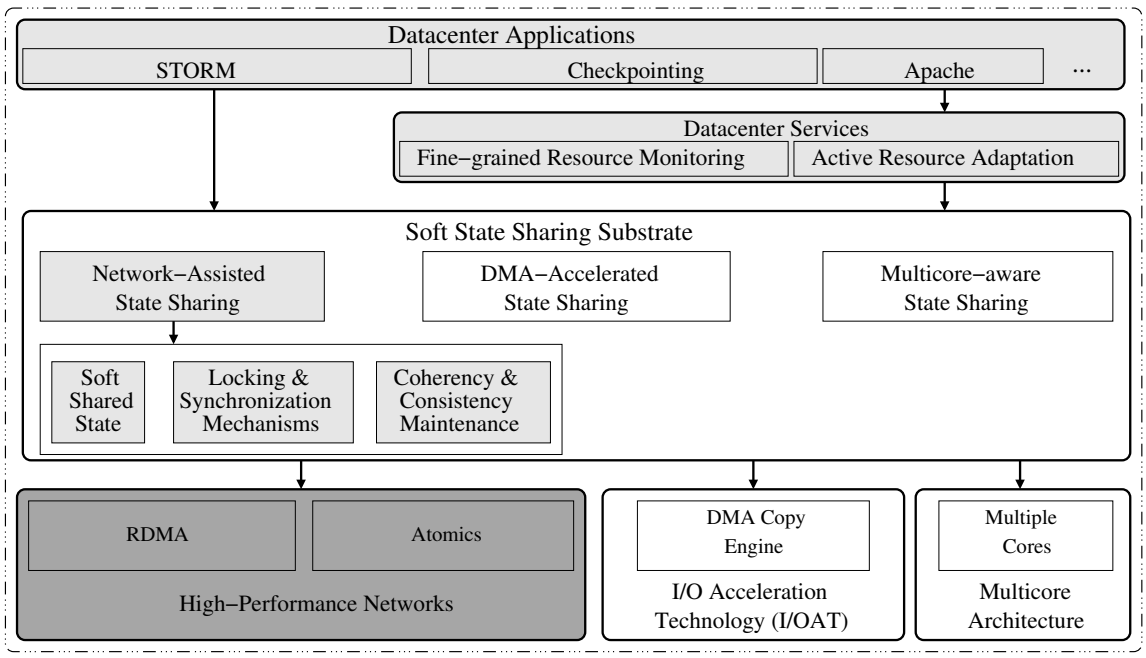


Figure 2.1: Network-Assisted State Sharing Components

of these applications, support for basic operations to read, write and synchronize are critical requirements of the distributed state sharing substrate. Further, as the nodes in a datacenter environment experience fluctuating CPU load conditions the substrate needs to be resilient and robust to changing system loads.

Higher-level datacenter services are intelligent services that are critical for the efficient functioning of datacenters. Such services require sharing of some state information. For example, caching services such as active caching [71] and cooperative caching [102, 72] require the need for maintaining versions of cached copies of data and locking mechanisms for supporting cache coherency and consistency. Active resource adaptation service requires the need for advanced locking mechanism in order to reconfigure nodes serving one website to another in a transparent manner and

needs simple mechanism for data sharing. Resource monitoring services, on the other hand, require efficient, low overhead access to the load information on the nodes. The substrate has to be designed in a manner that meets all of the above requirements.

Broadly, to accommodate the diverse coherence requirements of datacenter applications services, the substrate should support a range of coherency models [48]. Secondly, to meet the consistency needs of datacenter applications, the substrate should support versioning of cached data and ensure that requests from applications at multiple sites view the data in a consistent manner. Thirdly, services such as resource monitoring require the state information be maintained locally due to the fact that the data is updated frequently. On the other hand, services such as caching and resource adaptation can be compute intensive and hence require the data to be maintained at remote nodes distributed over the cluster.

Apart from the above, the substrate should also meet the following needs. Due to the presence of multiple threads on each of these applications at each node in the datacenter environment, the substrate should support the access, update and deletion of the shared data by all threads. Services such as resource adaptation and monitoring are characterized by frequent reading of the system load on various nodes in the datacenter. In order to efficiently support reading of this distributed state information, the substrate must provide asynchronous interfaces for reading and writing of shared information and provide the relevant wait operations for detecting the completions of such events. Further, the substrate must be designed to be robust and resilient to load imbalances and should have minimal overheads and provide a low access latency to data. Finally, the substrate must provide an interface that

clearly defines the mechanisms to allocate, read, write and synchronize the data being managed in order for such services and applications to utilize the substrate efficiently.

Related Work: There has been several state sharing models proposed in the past for a variety of environments. The most important feature that distinguishes DDSS from previous work is the ability to take advantage of several features of high-performance networks, its applicability and portability with several high-performance networks, its exploitation of relaxed coherence protocols and its minimal overhead. Further, our work is mainly targeted for datacenter environments on very large scale clusters.

Several run-time data sharing models such as InterWeave [47, 89], Khazana [42], InterAct [79] offer many benefits to applications in terms of relaxed coherency and consistency protocols. Friedman [56] and Amza et. al [30] have shown ways of combining consistency models. Khazana [42] also proposes the use of several consistency models. InterWeave [47, 89] allows various coherence models allowing users to define application-specific coherence models. Many of these models are implemented based on the traditional two-sided communication model targeting the WAN environment addressing issues such as heterogeneity, endianness, etc. Such two-sided communication protocols have been shown to have significant overheads in a real cluster-based datacenter environment under heavy loaded conditions. Also, none of state sharing models take advantage of high-performance networks for communication, synchronization and supporting efficient locking mechanisms. Though many of the features of high-performance networks are applicable only in a cluster environment, with the advent of advanced protocols such as iWARP [86] included in the OpenFabrics [77]

standard, DDSS can also work well in WAN environments and can still benefit applications using the advanced features offered by modern networks.

Several researchers have also proposed the feasibility and potential of cluster-based servers [55, 82] for scalability and availability of resource-intensive distributed applications. In the past, researchers have proposed coarse-grained monitoring approaches [78, 55] in order to avoid the overheads associated with fine-grained monitoring for such environments. Researchers have proposed and evaluated various load balancing policies using load information for cluster-based network services [41, 87]. Our proposed scheme is applicable to all the load balancing schemes that use monitoring information. In addition, some of the existing load-balancing schemes can be enhanced further using the detailed system information provided by our scheme. Several others have focused on the design of adaptive systems that can react to changing workloads in the context of web servers [40, 65, 80]. Our schemes are also applicable in these environments which use monitoring information for reconfiguration of resources.

2.2 Proposed Design

The basic idea of the state sharing substrate [97] is to allow efficient sharing of information across the cluster by creating a logical shared memory region. The substrate supports two basic operations, *get* operation to read the shared data segment and *put* operation to write onto the shared data segment. Figure 2.2a shows a simple distributed data sharing scenario with several processes (proxy servers) writing and several application servers reading certain information from the shared environment

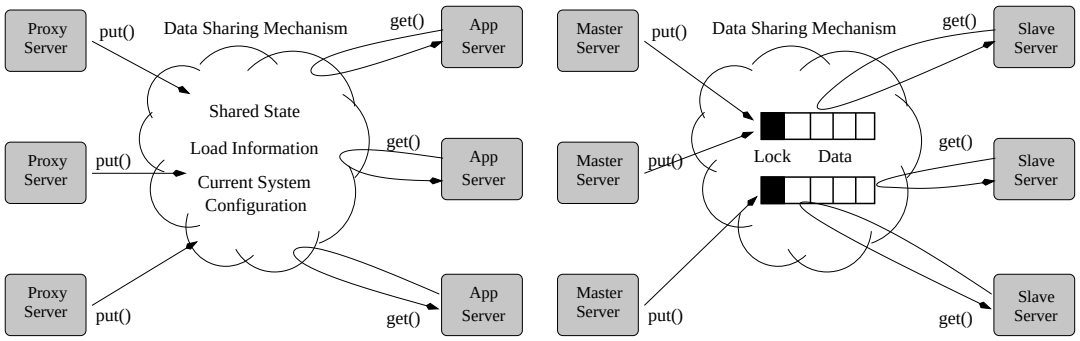


Figure 2.2: Network-Assisted State Sharing using the proposed Framework (a) Non Coherent State Sharing Mechanism (b) Coherent State Sharing Mechanism

simultaneously. Figure 2.2b shows a mechanism where coherency becomes a requirement. In this figure, we have a set of master and slave servers accessing different portions of the shared data. Each master process waits for the lock to be acquired for updating if the shared data is currently being read by multiple slave servers.

2.2.1 Soft Shared State using RDMA and Atomics

In the following sections, we refer to our proposed substrate as a distributed data sharing substrate (DDSS). Each node in the system allocates a large pool of memory to be shared with the shared state. We perform the allocation and release operations inside this distributed memory pool. One way to implement the memory allocation is to inform all the nodes about an allocation. However, informing all the nodes may lead to large latencies. Another approach is to assign one node for each allocation (similar to home-node based approach but the node can maintain only the metadata and the actual data can be present elsewhere). This approach reduces the allocation latency. The nodes maintain a list of free blocks available within the memory pool. During

a *release_ss()* operation, we inform the designated remote node for that allocation. During the next allocation, the remote node searches through the free block list and informs the free block which can fit the allocation unit. While searching for the free block, for high-performance, we get the first-fit free block which can accommodate the allocation unit. High-speed networks, as mentioned in Section 1.2.1, provide one-sided operations (like RDMA read and RDMA write) that allow access to remote memory without interrupting the remote node. In our implementation, we use these operations to perform the read and write. All the applications and services mentioned in Figure 2.3 will need this interface in order access/update the shared data.

Though the substrate hides the placement of shared data segments, it also exposes interfaces to the application to explicitly mention the location of the shared data segment (e.g. local or remote node). For the remote nodes, the interface also allows the application to choose a specific node. In our implementation, each time a data segment is allocated, the next data segment is automatically allocated on a different node. This design allows the shared data segments to get well-distributed among the nodes in the system and accordingly help in distributing the load in accessing the shared data segments for datacenter environments. This is particularly useful in reducing the contention at the NIC in the case where all the shared segments reside in one single node and several nodes need to access different data segments residing on the same node. In addition, distributed shared segments also help in improving the performance for applications which use asynchronous operations on multiple segments distributed over the network.

In order to support multiple user processes or threads in a system to access the substrate, we optionally provide a run-time daemon to handle the requests from multiple processes. We use shared memory channels and semaphores for communication and synchronization purposes between the user process and the daemon. The daemon establishes connections with other data sharing daemons and forms the state sharing framework. Any service which is multi-threaded or the presence of multiple services need to utilize this component for efficient communication. Connection management takes care of establishing connections to all the nodes participating in either accessing or sharing its address space with other nodes in the system. It allows for new connections to be established and existing connections to be terminated.

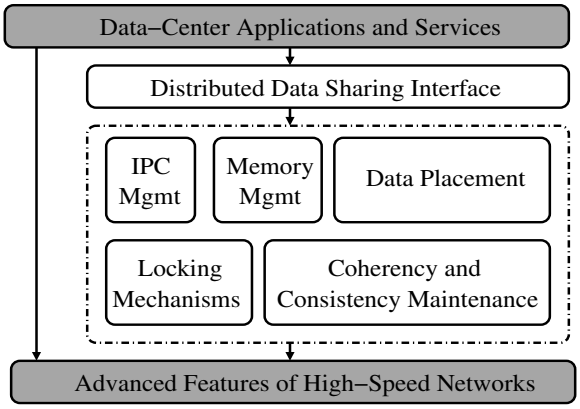


Figure 2.3: Network-Assisted State Sharing Framework

2.2.2 Efficient Locking and Synchronization Mechanisms

Locking and synchronization mechanisms are provided using the atomic operations which is completely handled by modern network adapters. The atomic operations

such as Fetch-and-Add and Compare-and-Swap operate on 64-bit data. The Fetch-and-Add operation performs an atomic addition at a remote node, while the Compare-and-Swap compares two 64-bit values and swaps the remote value with the data provided if the comparison succeeds. In our implementation, every allocation unit is associated with a 64-bit data which serves as a lock to access the shared data and we use the Compare-and-Swap atomic operation for acquiring and checking the status of locks. As mentioned earlier, modern interconnects such as IBA provide one-sided atomic operations which can be used for implementing basic locking mechanisms. In our implementation, we perform atomic compare and swap operations to check for the lock status and in acquiring the locks. If the locks are implicit based on the coherence model, then the substrate automatically unlocks the shared segment after successful completion of *get()* and *put()* operations. Each shared data segment has an associated lock. Though we maintain the lock for each shared segment, the design allows for maintaining these locks separately. Similar to the data in state sharing substrate, the locks can also be distributed which can help in reducing the contention at the NIC if too many processes try to acquire different locks on the same node.

2.2.3 Coherency and Consistency Maintenance using Atomics

Broadly, to accommodate the diverse coherency requirements of datacenter applications and services, the substrate supports a range of coherency models. The six basic coherency models [48] to be supported are: 1) *S*trict Coherence, which obtains the most recent version and excludes concurrent writes and reads. Database transactions require strict coherence to support atomicity. 2) *W*rite Coherence, which obtains the most recent version and excludes concurrent writes. Resource monitoring

services [94] need such a coherence model so that the server can update the system load and other load-balancers can read this information concurrently. 3) *R*ead Coherence is similar to write coherence except that it excludes concurrent readers. Services such as reconfiguration [31] are usually performed at many nodes and such services dynamically move applications to serve other websites to maximize the resource utilization. Though all nodes perform the same function, such services can benefit from a read coherence model to avoid two nodes looking at the same system information and performing a reconfiguration. 4) *N*ull Coherence, which accepts the current cached version. Proxy servers that perform caching on data that does not change in time usually require such a coherence model. 5) *D*elta coherence guarantees that the data is no more than x versions stale. This model is particularly useful if a writer has currently locked the shared segment and there are several readers waiting to the read the shared segment. 6) *T*emporal Coherence guarantees that the data is no more than t time units stale.

We implement these models by utilizing the RDMA and atomic operations of advanced networks. However, for networks which lack atomic operations, we can easily build software-based solutions using the send/receive communication model. In the case of Null coherence model, since there is no explicit requirement of any locks, applications can directly read and write on the shared data segment. For strict, read, write coherence models, we maintain locks and *get()* and *put()* operations internally acquire locks to the substrate before accessing or modifying the shared data. The locks are acquired and released only when the application does not currently hold the lock for a particular shared segment. In the case of version-based coherence model, we maintain a 64-bit integer and use the InfiniBand's atomic operation to update the

version for every *put()* operation. For *get()* operation, we perform the actual data transfer only if the current version does not match with the version maintained at the remote end. In delta coherence model, we split the shared segment into memory hierarchies and support up to x versions. Accordingly, applications can ask for up to x previous versions of the data using the *get()* and *put()* interface. Basic consistency is achieved through maintaining versions of the shared segment and applications can get a consistent view of the shared data segment by reading the most recently updated version.

Proposed State Sharing Interface: Table 2.1 shows the current interface that is available to the end-user applications or services. The interface essentially supports six main operations for gaining access to the substrate: *allocate_ss(), get(), put(), release_ss(), acquire_lock_ss(), release_lock_ss() operations*. The *allocate_ss()* operation allows the application to allocate a chunk of memory in the shared state. This function returns a unique shared state key which can be shared among other nodes in the system for accessing the shared data. *get() and put()* operations allow applications to read and write data to the shared state and *release_ss()* operation allows the shared state substrate to reuse the memory chunk for future allocations. The *acquire_lock_ss() and release_lock_ss()* operations allow end-user application to gain exclusive access to the data to support user-defined coherency and consistency requirements. In addition, we also support asynchronous operations such as *async_get(), async_put(), wait_ss()* and additional locking operations such as *try_lock()* operation to support a wide range of applications to use such features.

The substrate is built as a library which can be easily integrated into distributed applications such as checkpointing, DataCutter [24], web servers, database servers,

Operation	Description
int allocate_ss(nbytes, type, ...)	allocate a block of size nbytes in the shared state
int release_ss(key)	free the shared data segment
int get(key, data, nbytes, ...)	read nbytes from the shared state and place it in data
int put(key, data, nbytes, ...)	write nbytes of memory to the shared state from data
int acquire_lock_ss(key)	lock the shared data segment
int release_lock_ss(key)	unlock the shared data segment

Table 2.1: State Sharing Interface

etc. For applications such as DataCutter, several data sharing components can be replaced directly using the substrate. Further, for easy sharing of keys, i.e., the key to an allocated data segment, the substrate allows special identifiers to be specified while creating the data sharing segment. Applications can create the data sharing segment using this identifier and the substrate will make sure that only one process creates the data segment and the remaining processes will get a handle to this data segment. For applications such as web servers and database servers, the substrate can be integrated as a dynamic module and all other modules can make use of the interface appropriately. In our earlier work, cooperative caching [72], we have demonstrated the capabilities of high-performance networks for datacenters with respect to utilizing the remote memory and support caching of varying file sizes. The substrate can also be utilized in such environments.

2.2.4 Fine-Grained Resource Monitoring Services

The basic idea of fine-grained resource monitoring in a datacenter environment is to capture the dynamic resource usage of the hosted applications. Fine-grained

resource monitoring can be implemented using two approaches: back-end based monitoring and front-end based monitoring. In the former, the back-end informs the front-end node on detecting a high load. In the latter, the front-end node periodically sends a request to a resource monitoring process in the back-end to retrieve the load information. It is to be noted that when the back-end server gets a network packet from the front-end, the kernel treats it as a high priority packet and tries to schedule the resource monitoring process as early as possible. However, in the back-end resource monitoring scheme, the monitoring process sleeps for a given time interval and calculates the load information, thus decreasing its priority to be scheduled. Since the load reporting interval resolution highly depends on the operating system scheduling timer resolution, the scheduling of this back-end monitoring process is vital for sending the load responses in a timely manner. For fine-grained resource monitoring since there is a need for an immediate reply and small reporting interval resolution, front-end based resource monitoring is preferred. Previous work [87] also suggests that a front-end based approach is better than back-end based approach for fine-grained services. For these reasons, we focus on front-end based resource monitoring.

In the following sections, we present existing sockets-based implementations and our proposed design alternatives [94] based on the state sharing substrate. We refer to the *get()* operation in the state sharing substrate as *RDMA read* in the following sections. Broadly, two ways of designing the front-end based resource monitoring for sockets and RDMA exist: (i) Asynchronous and (ii) Synchronous. In an asynchronous approach, the load calculating phase (i.e., reading the resource usage information and calculating the current load on the back-end) and load requesting phase (i.e.,

requesting for load information from the front-end) are independent. On the other hand, in a synchronous approach, the back-end calculates the current load information for every request received from the front-end node.

Asynchronous Resource Monitoring using Sockets (Socket-Async): In this approach, we have two processes, one running on the front-end server and the other running on the back-end server. The back-end server process consists of two threads; a load calculating thread that calculates the load information periodically and a load reporting thread that responds to load requests from the front-end servers. The sequence of steps in asynchronous resource monitoring using sockets is shown in Figure 2.4(a). In the first step (Step 1), the load calculating thread reads */proc*. To access */proc*, a trap occurs because of file I/O in Step 2, during which the kernel calculates the system information. In Step 3, */proc* sends the monitoring information to the thread and in Step 4, the thread copies this information to a known memory location. Once this task is completed, the load calculating thread sleeps for a specific time interval T and repeats this process again. In parallel, the front-end monitoring process periodically sends a request for load information to the load reporting thread (Step a). The load reporting thread receives this request, reads the load information from the known memory location (Step b) and sends it to the front-end monitoring process (Step c).

Synchronous Resource Monitoring using Sockets (Socket-Sync): This approach is very similar to the asynchronous approach, except that there is no requirement for two threads in the back-end. As shown in Figure 2.4(b), when the front-end monitoring process sends a load request (Step 1), the back-end monitoring process calculates the load information by reading the */proc* file system (Steps 2, 3 and 4)

and reports this load information to the front-end monitoring process (Step 5). Thus, there is no requirement for a separate thread to calculate the load information for every time interval T.

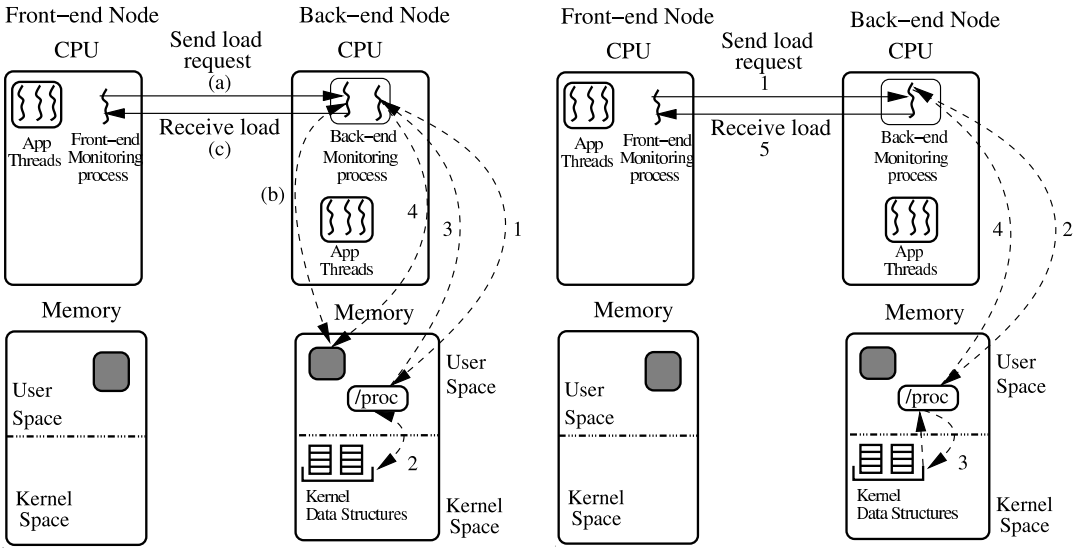


Figure 2.4: Resource Monitoring Mechanism using Sockets: (a) Asynchronous and (b) Synchronous

We propose two design alternatives in performing fine-grained resource monitoring using the state sharing substrate.

Asynchronous Resource Monitoring using RDMA: In this approach, as shown in Figure 2.5(a), we use two different kinds of monitoring processes running on front-end and back-end server. The back-end monitoring process handles connection management and creates registered memory regions in the user space. The front-end monitoring process periodically performs RDMA read operations (similar to *get()* operations using the resource sharing substrate) on the registered memory regions

to retrieve updated load information. We use the same mechanism as Socket-Async scheme for calculating the load information. The back-end monitoring process constantly calculates the relevant load information after every time T interval from */proc* and copies the load information onto the registered memory region.

Synchronous Resource Monitoring using RDMA: In theory, as RDMA operations are one-sided, it is not possible to have a synchronous resource monitoring approach using RDMA. However, in practice, we can achieve the accuracy of synchronous resource monitoring if the front-end node can obtain the most up-to-date load information from the kernel memory of the back-end for every request. To enable this, we register the necessary kernel data structures that hold the resource usage information to the state sharing substrate, and allow the front-end monitoring process to directly retrieve this information using RDMA read (similar to *get()* operations using the state sharing substrate) as shown in Figure 2.5(b). Such a design has two major advantages: (i) it removes the need for an extra process in the back-end server and (ii) it can exploit the detailed resource usage information in kernel space to report accurate load information.

In this approach, we use a Linux kernel module for handling connection management, address exchange and memory registration of specific memory regions for the front-end monitoring process to perform RDMA read operation. After the initialization phase, the kernel is no longer disturbed. As shown in Figure 2.5(b), the load information is directly obtained from the kernel space, reading the kernel data structures using RDMA read operations (Step a).

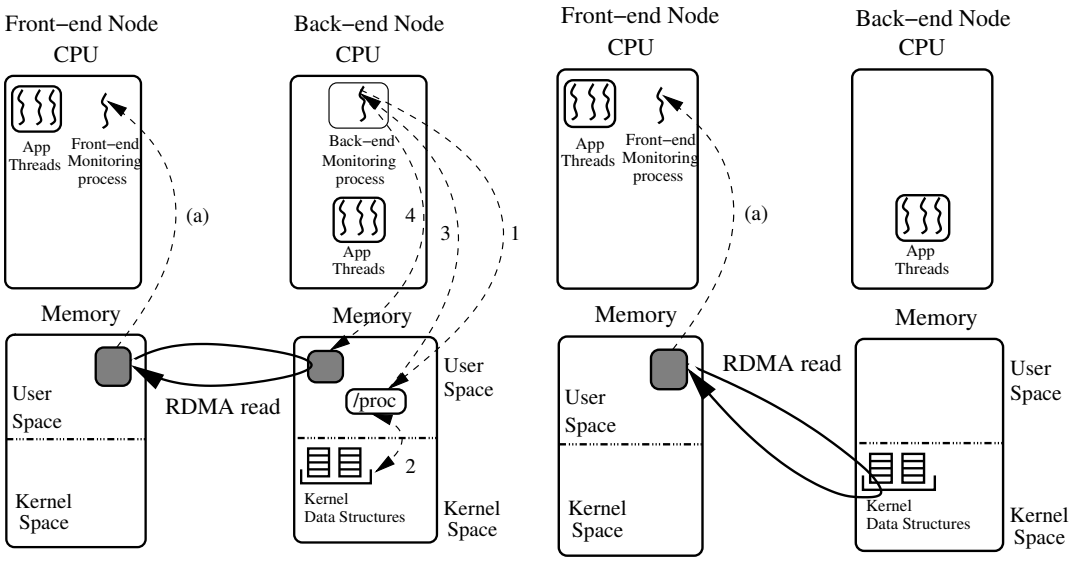


Figure 2.5: Resource Monitoring Mechanism using State Sharing: (a) Asynchronous and (b) Synchronous

Potential Benefits of RDMA-Sync: Using the RDMA-Sync scheme to design and implement fine-grained resource monitoring has several potential benefits as described below.

Getting accurate load information: Due to the asynchronous nature of Socket-Async and RDMA-Async schemes, there is a delay between the time at which the back-end monitoring process updates the load information and the time at which the front-end monitoring process reads this load information. For example, if we assume that the load information is updated every T *ms* at the back-end server, the load information seen by the front-end monitoring process can be up to T *ms* old. In the Socket-Sync scheme, if the server nodes are heavily loaded, the back-end monitoring process can compete for CPU with other threads in the system. This can result in huge delays in reporting the current load information to the front-end monitoring

process. However, regardless of the back-end server load, the RDMA-Sync scheme can report accurate load information since the front-end monitoring process directly retrieves the load information from kernel data structures without interrupting the CPU. As a result, the RDMA-Sync scheme can quickly and accurately detect the load and can help to avoid overloaded conditions in several environments.

Utilizing detailed system information: While all other monitoring schemes operate at the user space, the RDMA-Sync scheme operates at the kernel space. This provides several opportunities to access portions of the kernel memory which may be useful for providing system-level services. Some of them are directly exposed via */proc* interface while others like *irq_stat, dq_stat, and aven_run* are not. Though the other schemes can access these kernel data structures using a kernel module, later in the experimental section, we show some unique benefits of the RDMA-Sync scheme.

No extra thread for remote resource monitoring: All monitoring schemes except the RDMA-Sync scheme require a separate thread on the back-end server to calculate the load of the back-end node periodically. While this operation may not occupy considerable CPU, in a highly loaded server environment, it certainly competes for processor cycles. This can result in huge delays in updating the load information. Also, if the incoming traffic arrives in bursts, such delays may lead to poor reconfiguration and process migration since delayed load information can give a wrong picture of current load status of the back-end servers. However, in the RDMA-Sync scheme, there is no extra thread required to calculate the load information thus avoiding all the issues mentioned above.

Enhanced robustness to load: Performance of system-level services over traditional network protocols can be degraded significantly if there is a high load in the back-end.

This is because both sides should get involved in communication and it is possible that the back-end monitoring process capturing the load on the back-end may never get the CPU for a long time. However, for protocols based on RDMA operations, the peer side is totally transparent to the communication procedure. Thus, the latency of both RDMA-Sync and RDMA-Async schemes is resilient and well-conditioned to load.

2.2.5 Active Resource Adaptation Services

Request patterns seen over a period of time, by a shared datacenter 1.1, may vary significantly in terms of the ratio of requests for each co-hosted web-site. For example, interesting documents or dynamic web-pages becoming available and unavailable might trigger traffic in bursts for some web-site at some time and for some other web-site at a different time. This naturally changes the resource requirements of a particular co-hosted web site from time to time. The basic idea of resource adaptation (used interchangeably with dynamic resource reconfiguration) is to utilize the idle nodes of the system to satisfy the dynamically varying resource requirements of each of the individual co-hosted web-sites in the shared datacenter. Dynamic reconfigurability of the system requires some extent of functional equivalence between the nodes of the datacenter. We provide this equivalence by enabling software homogeneity such that each node is capable of belonging to any web-site in the shared datacenter. Depending on current demands (e.g., due to a burst of a certain kind of requests), nodes reconfigure themselves to support these requests.

Support for Existing Applications: A number of applications exist that allow highly efficient user request processing. These have been developed over a span of

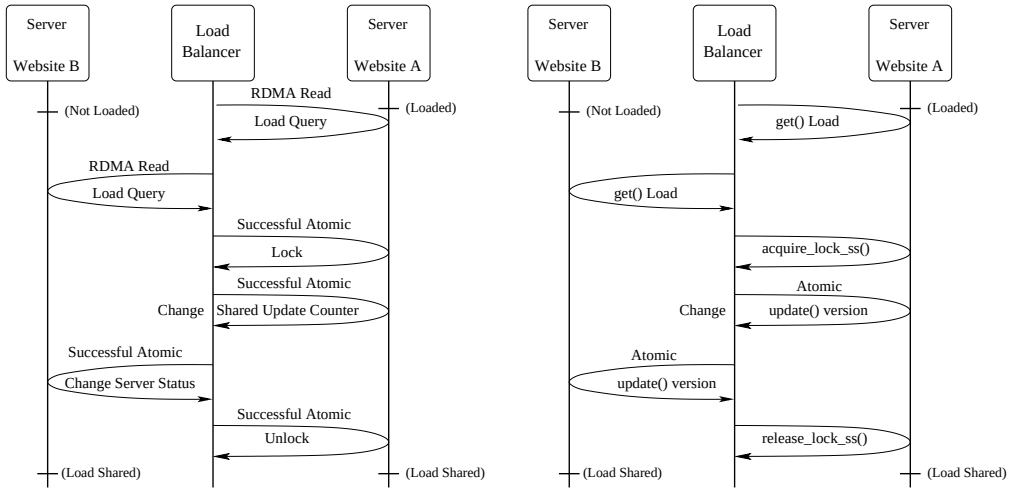
several years and modifying them to allow dynamic reconfigurability is impractical. To avoid making these cumbersome changes to the applications, our design makes use of *external helper modules* which works along with the applications to provide effective dynamic reconfiguration. Tasks related to system load monitoring, maintaining global state information, reconfiguration, etc. are handled by these helper modules in an application transparent manner. These modules, running on each node in the shared datacenter, reconfigure nodes in the datacenter depending on current request and load patterns. They start, stop and use the run-time configuration files of the datacenter applications to reflect these changes. The servers on the other hand, just continue with the request processing, unmindful of the changes made by the modules.

Load-Balancer Based Reconfiguration: Two different approaches could be taken for reconfiguring the nodes: Server-based reconfiguration and Load-balancer based reconfiguration. In server-based reconfiguration, when a particular server detects a significant load on itself, it tries to reconfigure a relatively free node that is currently serving some other web-site content. Though intuitively the loaded server itself is the best node to perform the reconfiguration (based on its closeness to the required load information), performing reconfiguration on this node adds a significant amount of load to an already loaded server. Due to this reason, reconfiguration does not happen in a timely manner and the overall performance is affected adversely. On the other hand, in a load-balancer based reconfiguration, the edge servers (functioning as load-balancers) detect the load on the servers, find a free server to alleviate the load on the loaded server and perform the reconfiguration themselves. Since the shared information like load, server state, etc. is closer to the servers, this approach incurs the cost of requiring more network transactions for its operations.

As mentioned in Chapter 1, by their very nature, the server nodes are compute intensive. Execution of CGI-Scripts, business-logic, servlets, database processing, etc. are typically very taxing on the server CPUs. So, the helper modules can potentially be starved for CPU on these servers. Though in theory the helper modules on the servers can be used to share the load information through explicit two-sided communication, in practice [71], such communication does not perform well. InfiniBand, on the other hand, provides one-sided remote memory operations (like RDMA and Remote Atomics) that allow access to remote memory without interrupting the remote node. In our design, we use these operations to perform load-balancer based server reconfiguration in a server transparent manner. Since the load-balancer is performing the reconfiguration with no interruptions to the server CPUs, this RDMA based design is highly resilient to server load. Figure 2.6(a) shows the RDMA based protocol used by Dynamic Reconfigurability. As shown in the figure, the entire cluster management and dynamic reconfiguration is performed by the lightly loaded load-balancer nodes without disturbing the server nodes using the RDMA and remote atomic operations provided by InfiniBand. Further details about the other design issues can be found in [31]. Figure 2.6(b) shows how this datacenter service can be easily written over the state sharing substrate.

2.3 Experimental Results

We evaluate the proposed substrate with a set of microbenchmarks to understand the performance, scalability and associated overheads. Later, we analyze the applicability of the substrate with applications such as Distributed STORM and checkpointing and services such as resource monitoring and reconfiguration. We evaluate



(a) Dynamic Reconfigurability using RDMA [31]

(b) Dynamic Reconfigurability using DDSS

Figure 2.6: Dynamic Reconfigurability Schemes

our framework on two interconnects IBA [61] and Ammasso [3] using the OpenFabrics implementation [77]. While designing the state sharing substrate, the iWARP [86] implementation of OpenFabrics over Ammasso was available only at the kernel space. We wrote a wrapper for user applications which in turn calls the kernel module to fire appropriate iWARP functions. Our experimental testbed consists of a 12 node cluster with dual Intel Xeon 3.4 GHz CPU-based EM64T systems. Each node is equipped with 1 GB of DDR400 memory. The nodes were connected with MT25128 Mellanox HCAs (SDK v1.8.0) connected through a InfiniScale MT43132 24-port completely non-blocking switch. For Ammasso experiments, we use two node dual Intel Xeon 3.0 GHz processors with a 512 KB L2 cache and a 533 MHz front side bus and 512

MB of main memory. First, we evaluate the access latency and the overhead of our proposed state sharing substrate.

2.3.1 State Sharing Latency

The latency test is conducted in a ping-pong fashion and the latency is derived from round-trip time. For measuring the latency of *put()* operation, we run the test performing several *put()* operations on the same shared segment and average it over the number of iterations. Figure 2.7(a) shows the latencies of different coherence models by using the *put()* operation of the substrate using OpenFabrics over IBA through a daemon process. We observe that the 1-byte latency achieved by null and read coherence model is only 20μs and 23μs, respectively. Note that the overhead of communicating with the daemon process is close to 10-12μs and hence, we see large latencies with null and read coherence models. For write and strict coherency model, the latencies are 54.3μs and 54.8μs, respectively. This is due to the fact that both write and strict coherency models, apart from the RDMA operation, also use atomic operations to acquire the lock before updating the shared data. Version-based and delta coherence models report a latency of 37μs and 41μs, respectively, since they both need to update the version status maintained at the remote node using atomic operations. Also, as the message size increases, we observe that the latency increases for all coherence models. We see similar trends for *get()* operations with the basic 1-byte latency of get being 25μs. Figure 2.7(b) shows the performance of *get()* operation with several clients accessing different portions from a single node. We observe that the substrate is highly scalable in such scenarios and the performance is not affected for increasing number of clients. Figure 2.7(c) shows the performance of *put()*

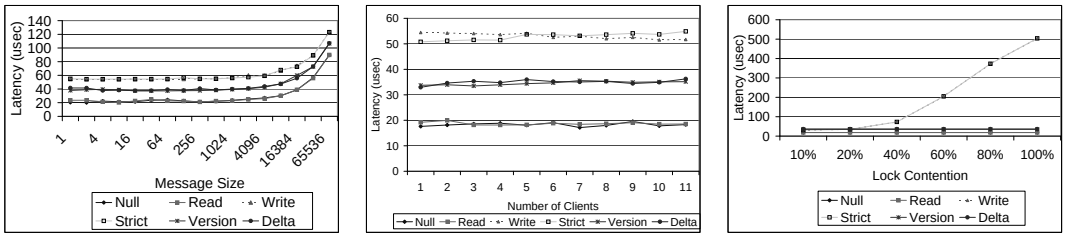


Figure 2.7: Basic Performance using OpenFabrics over IBA: (a) *put()* operation, (b) Increasing Clients accessing different portions (*get()*) and (c) Contention accessing the same shared segment (*put()*)

operation with several clients accessing the same portion from a single node. Here, we observe that for relatively lesser contention-levels of up to 40%, the performance of the *put()* operation does not seem to be affected. However, for contention-levels more than 40%, the performance of clients degrades significantly in the case of strict and write coherence models mainly due to the waiting time for acquiring the lock.

2.3.2 State Sharing Overhead

One of the critical issues to address on supporting state sharing substrate is to minimize the overhead of the middleware layer for applications. We measure the overhead for different configurations, namely: (i) a direct scheme that allows application to directly communicate with underlying network through the substrate, (ii) a thread-based scheme that allows applications to communicate through a daemon process for accessing the substrate and (iii) a thread-based asynchronous scheme that allows applications to use asynchronous operations using the substrate. We see that the overhead is less than a microsecond (0.35μs) through the direct scheme. If the runtime system needs to support multiple threads, we observe that the overhead jumps

to 10μs using the thread-based scheme. The reason being the overhead of round-trip communication between the application thread and the substrate daemon using System V IPC communication comes close to 10μs. If the application uses asynchronous operations (thread-based asynchronous scheme), this overhead can be significantly reduced for large message transfers. However, in the worst case, for small message sizes, this scheme can lead to an overhead of 12μs. The average synchronization time observed in all the schemes is around 20μs.

2.3.3 Performance with Datacenter Applications

We redesign existing datacenter applications such as STORM and checkpointing applications and show the improvement achieved in using our substrate.

STORM with DataCutter: STORM [24] is a middleware service layer developed by the Department of Biomedical Informatics at The Ohio State University. It is designed to support SQL-like select queries on datasets primarily to select the data of interest and transfer the data from storage nodes to compute nodes for processing in a cluster computing environment. In such environments, it is common to have several STORM applications running which can act on same or different datasets serving the queries of different clients. If the same dataset is processed by multiple STORM nodes and multiple compute nodes, the substrate can help in sharing this dataset in a cluster environment so that multiple nodes can get direct access to this shared data. In our experiment, we modified the STORM application code to use the substrate in maintaining the dataset so that all nodes have direct access to the shared information. We vary the dataset size in terms of number of records and show the performance of STORM with and without the substrate. Since larger

datasets showed inconsistent values, we performed the experiments on small datasets and we flush the file system cache to show the benefits of maintaining this dataset on other nodes memory. As shown in Figure 2.8(a), we observe that the performance of STORM is improved by around 19% for 1K, 10K and 100K record dataset sizes using the substrate in comparison with the traditional implementation.

Application Check-pointing: We use a check-pointing benchmark to show the scalability and performance of using the substrate. In this experiment, every process attempts to checkpoint a particular application at random time intervals. Also, every process simulates the application restart, by attempting to reach a consistent check-point and informing all other processes to revert back to the consistent check-point at other random intervals. In Figure 2.8(b), we observe that the average time taken for check-pointing is only around 150μs for increasing number of processes. As this value remains almost constant with increasing number of clients and application restarts, it suggests that the application scales well using the substrate. Also, we see that the average application restart time to reach a consistent checkpoint increases with increasing clients. This is expected as each process needs to get the current checkpoint version from all other processes to decide the most recent consistent checkpoint.

2.3.4 Performance with Resource Monitoring Services

In this section, we evaluate our proposed resource monitoring schemes as mentioned in Section 2.2.4. For all our experiments we use the following two system configurations: A cluster system consisting of 8 server nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 KB

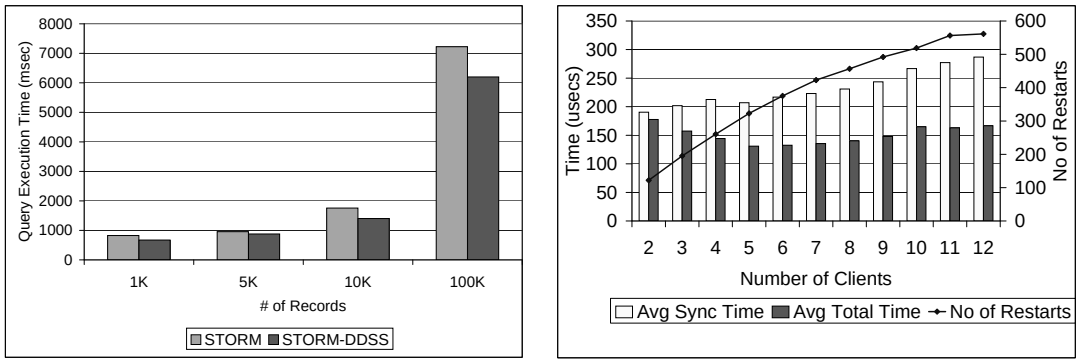


Figure 2.8: Application Performance over IBA: (a) Distributed STORM application and (b) Application Check-pointing

L2 cache and a 400 MHz front side bus and 1 GB of main memory. We use the RedHat 9.0 Linux distribution using an InfiniBand network with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 twenty-four 4x Port completely non-blocking InfiniBand Switch. The IPoIB driver for the InfiniBand adapters was provided by Voltaire Incorporation [6]. The version of the driver used was 2.0.5_10.

We use 8 client nodes with two Intel Xeon 3.0 GHz processors which include 64-bit 133 MHz PCI-X interfaces, a 533 MHz front side bus and 2 GB memory. We use the RedHat 9.0 Linux distribution. Apache 2.0.48, PHP 4.3.1 and MySQL 4.0.12 were used in our experiments. Requests from the clients were generated using eight threads on each node. We use a polling time T of 50ms for resource monitoring schemes in all the experiments unless otherwise explicitly specified.

Here, we evaluate the four schemes mentioned in Section 2.2.4 in terms of latency, granularity, accuracy of load information obtained and potential for extracting detailed system load information.

Latency of Resource Monitoring: In this section, we present the performance impact on the monitoring latency of the four schemes with loaded conditions in the cluster-based servers. We emulate the loaded conditions by performing background computation and communication operations on the server while the front-end monitoring process monitors the back-end server load. This environment emulates a typical shared server environment where multiple server nodes communicate periodically and exchange messages, while the front-end node, which is not as heavily loaded, attempts to get the load information from the monitoring process on the heavily loaded servers.

The performance comparison of Socket-Async, Socket-Sync, RDMA-Async and RDMA-Sync schemes for this experiment is shown in Figure 2.9. We observe that the monitoring latency of both Socket-Async and Socket-Sync schemes increases linearly with the increase in the background load. On the other hand, the monitoring latency of RDMA-Async and RDMA-Sync schemes which use one-side communication, stays the same without getting affected by the background load. These results show the capability of one-sided communication primitives in a cluster-based server environment.

Granularity of Resource Monitoring: We present the impact on the performance of running applications with respect to increasing granularity of resource monitoring of all four schemes. In this experiment, application performs basic floating-point operations and reports the time taken. We report the average application delay normalized to the application execution time for each of the schemes as we vary the granularity from 1 ms to 1024 ms as shown in Figure 2.10. We observe that the application performance degrades significantly when Socket-Async, Socket-Sync and RDMA-Async schemes are running in the background at smaller granularity such as 1

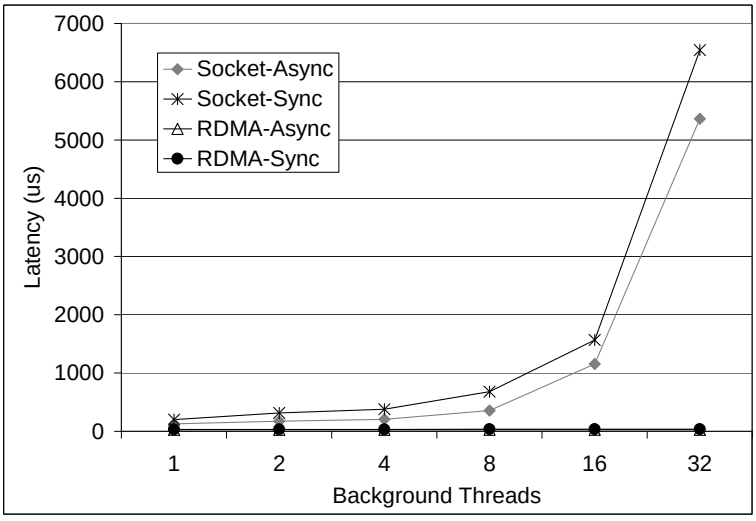


Figure 2.9: Latency of Socket-Async, Socket-Sync, RDMA-Async, RDMA-Sync schemes with increasing background threads

ms and 4 ms. Since the Socket-Async scheme uses two threads for resource monitoring in the back-end server, we find that this scheme affects the application performance significantly in comparison with other schemes. In RDMA-Async scheme, due to the presence of the back-end monitoring process, we see that the application performance degradation is lesser in comparison with the two-sided Socket-Sync scheme. However, we find that there is no performance degradation with the RDMA-Sync scheme due to the fact that there are no processes running on the back-end server to affect the application performance.

Accuracy of Resource Information: In this experiment, we analyze the accuracy of the load information obtained from the four schemes. In order to be uniform across all four schemes, we design the experiment in the following way. We run all four schemes simultaneously and monitor the load information. In addition, a kernel module on the back-end server periodically reports the actual load information

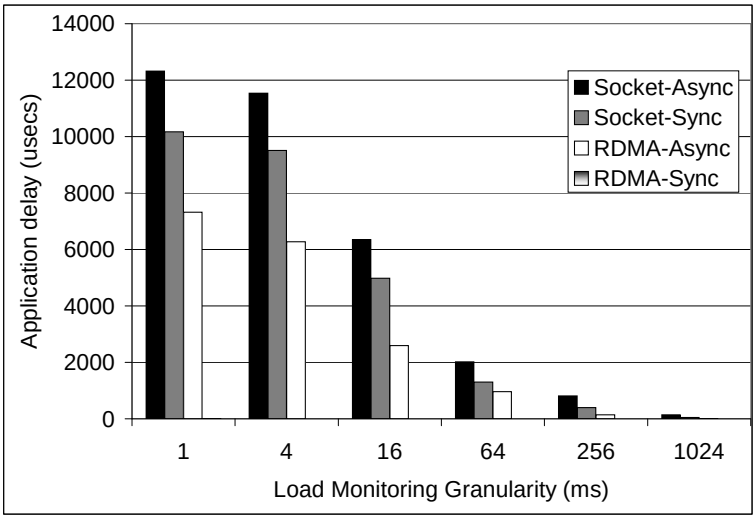


Figure 2.10: Impact on application performance with Socket-Async, Socket-Sync, RDMA-Async and RDMA-Sync schemes

at a finer granularity. To emulate loaded conditions, we fired client requests to be processed at the back-end server. We compare these numbers against the load information reported by the kernel module and plot the deviation between these two values.

Figure 2.11(a) shows the deviation of the number of threads running on the server with respect to the numbers reported by all four schemes. We see that all four schemes report the same values initially since there was no load on the server. However, as the load on the server increases, we see that Socket-Async, Socket-Sync and RDMA-Async schemes show deviations with respect to the number of threads reported by the kernel module. On the other hand, the RDMA-Sync scheme consistently reports no deviation. Further, we observe that both Socket-Async and Socket-Sync schemes show large deviations when the back-end server is heavily loaded. Since sockets is a two-sided communication protocol, as the load on the back-end server increases, the

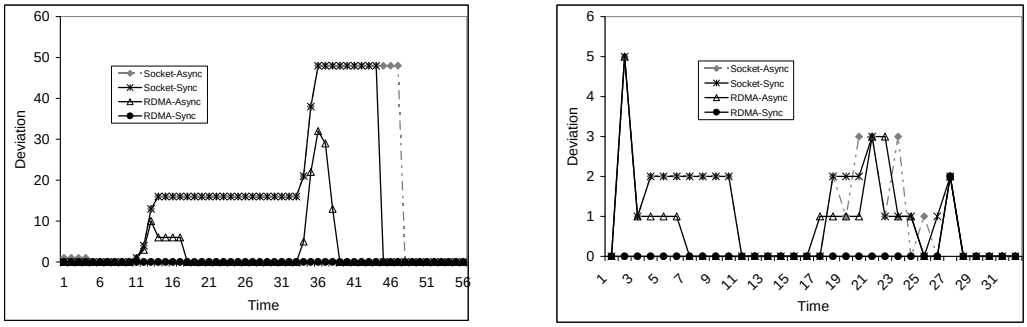


Figure 2.11: Accuracy of Load information: (a) Number of threads running on the server and (b) Load on the CPU

latency for capturing the number of threads running also increases leading to such inaccuracies.

Figure 2.11(b) shows the accuracy of the CPU load information reported by all four schemes in comparison with the actual CPU load. We perform the experiment in a similar way as explained above. We find that Socket-Async, Socket-Sync and RDMA-Async schemes show deviations in comparison with the actual load while RDMA-Sync scheme reports very few deviations. Since CPU load fluctuates more rapidly in comparison with the number of threads in the system, we see that RDMA-Async scheme also reports inaccurate load information. Socket-Async and Socket-Sync schemes, due to the reasons mentioned above, report stale CPU load values leading to large deviations.

Detailed System Information: In this experiment, we evaluate our four schemes in terms of their ability to obtain detailed system information with finer granularity. To explore this feature, we measure the number of interrupts pending on CPUs of the servers.

For our evaluation, we use the *irq_stat* kernel data structure, which maintains the number of software, hardware and bottom-half pending interrupts on each of the CPUs. We use all four schemes to report these values to the front-end monitoring process. Since the data structure is available at the kernel space, we use a kernel module to expose this to user-space, so that Socket-Async, Socket-Async and RDMA-Async schemes can report this information.

We see that the Socket-Async, Socket-Sync and RDMA-Async schemes, as shown in Figures 2.12(a), 2.12(b) and 2.13(a) report less and infrequent interrupts in comparison with the RDMA-Sync scheme as shown in Figure 2.13(b). As mentioned above, an user process triggers the kernel module to report the interrupt information for these three schemes. However, if there are pending interrupts on the CPUs, the operating system would give a higher priority to schedule the interrupts rather than a user process. Furthermore, in a uni-processor kernel, the operating system may complete all the interrupt handling and then pass the control to the user process. However, since there is no such requirement for the RDMA-Sync scheme, we observe that this scheme reports interrupt information more accurately. Interestingly, the RDMA-Sync scheme reports more interrupts (in terms of the number of interrupts) in comparison with the other three schemes. Moreover, the number of interrupts reported on the second CPU by the RDMA-Sync scheme is consistently higher in comparison with the numbers reported by all other schemes.

Performance with RuBiS and Zipf Workloads

Next, we study the benefits of our resource monitoring schemes in a datacenter environment. Following that, we compare the performance of the proposed fine-grained resource monitoring schemes against the traditional approaches.

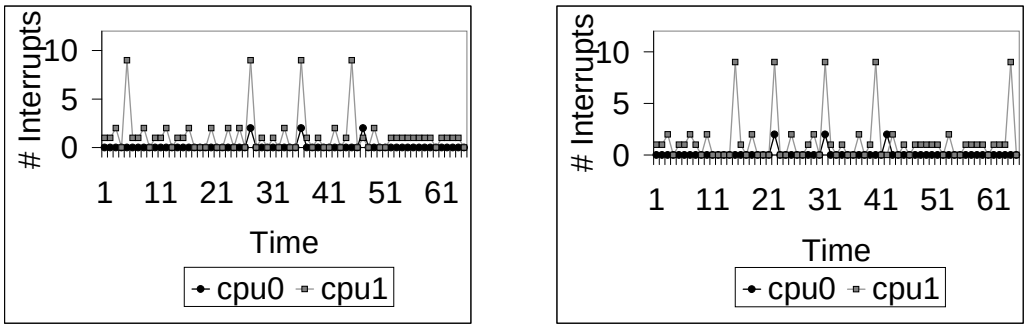


Figure 2.12: Number of Interrupts reported on two CPUs: (a) Socket-Async and (b) Socket-Sync

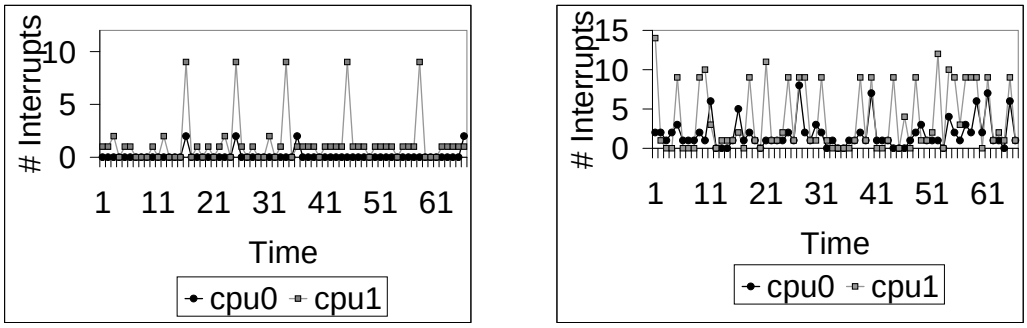


Figure 2.13: Number of Interrupts reported on two CPUs: (a) RDMA-Async and (b) RDMA-Sync

Cluster-based Server with RUBiS: We evaluate our schemes in a cluster-based server environment using a RUBiS auction benchmark [20, 29] developed by Rice University. The benchmark simulates a workload similar to a typical e-commerce website. It implements the typical functionality of auction sites such as selling, browsing and bidding. We modified the client emulator to fire requests to multiple servers and evaluate the resource monitoring schemes. In order to understand the benefits of detailed system information, we added an e-RDMA-Sync scheme that utilizes system load and also the pending interrupts on the CPUs for choosing the least-loaded servers. All other schemes use only system load for choosing the least-loaded servers. Tables 2.2 and 2.3 report the average and maximum response time of several queries of the RUBiS benchmark. We find that, both RDMA-Sync and e-RDMA-Sync schemes perform consistently better than the other three schemes. Since the maximum response time is considerably low for the RDMA-Sync scheme in comparison with Socket-Async, Socket-Sync and RDMA-Async schemes, it validates the fact that completely removing the need for another process on the server brings down the maximum response time. The improvement we realize for queries like BrowseRegions, Browse is close to 90% for RDMA-Sync and e-RDMA-Sync schemes. Other queries like BrowseCategoriesInRegions and SearchCategoriesInRegion show benefits up to 80% for both RDMA-Sync and e-RDMA-Sync schemes. In addition, we also observe that there is considerable improvement in the average response time of the queries for RDMA-Sync and e-RDMA-Sync schemes in comparison with other schemes. Also, we see that the e-RDMA-Sync scheme consistently performs better than the RDMA-Sync scheme showing the benefits of using detailed system information for performing effective and fine-grained services.

Query	Average Response Time				
	Socket Async	Socket Sync	RDMA Async	RDMA Sync	e-RDMA Sync
Home	3	3	3	3	2
Browse	3	3	3	3	2
BrowseReg	6	6	6	5	5
BrowseCatgry	17	17	18	16	14
SearchItems	4	4	4	4	3
PutBidAuth	3	3	3	3	2
Sell	4	4	3	2	2
About Me	3	3	3	3	2

Table 2.2: Average Response Time with RUBiS Benchmark

Query	Maximum Response Time				
	Socket Async	Socket Sync	RDMA Async	RDMA Sync	e-RDMA Sync
Home	416	274	36	33	31
Browse	495	348	197	81	45
BrowseReg	392	206	249	41	32
BrowseCatgry	265	278	210	74	66
SearchItems	150	180	78	43	32
PutBidAuth	99	231	38	30	20
Sell	373	264	19	21	21
About Me	178	220	32	35	32

Table 2.3: Maximum Response Time with RUBiS Benchmark

Cluster-based Server with RUBiS and Zipf Trace: In order to show the maximum potential of fine-grained resource monitoring, we design an experiment where cluster-based servers host two web services. We use a Zipf trace with varying α value. According to Zipf law, the relative probability of a request for the ith most popular document is proportional to $1/i^{\alpha}$, where α determines the randomness of file accesses. Higher the α value, higher is the temporal locality of the document accessed.

The experiment is designed in the following manner. We run the RUBiS benchmark and Zipf traces simultaneously and use all five schemes namely Socket-Async, Socket-Sync, RDMA-Async, RDMA-Sync and e-RDMA-Sync schemes for resource monitoring. We fix the RUBiS benchmark and vary the α value for the Zipf trace from 0.25 to 0.9. As mentioned earlier, higher α values mean the workload has a high temporal locality. We report the total throughput improvement in comparison with the Socket-Async scheme for each of these traces separately as shown in Figure 2.14. We can observe that in the case of Zipf trace with α value 0.25, both RDMA-Sync and e-RDMA-Sync schemes achieve a performance improvement of up to 28% and 35% respectively. For smaller α values, we see a considerable performance improvement. This is due to the fact that there are lots of requests with different resource requirements and these requests are forwarded to appropriate servers in a timely manner. As α value increases, the number of requests with different resource requirements decreases resulting in an increase in the temporal locality of the documents. Hence the load on all the servers are already well distributed leading to lesser performance gains.

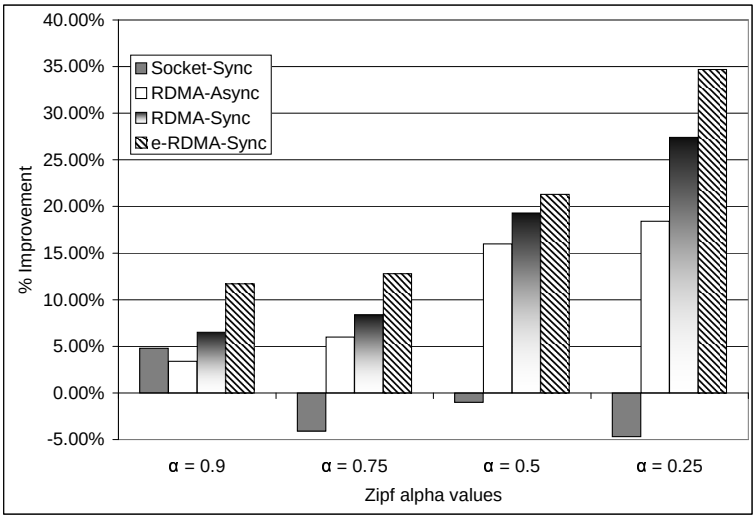


Figure 2.14: Throughput Improvement of Socket-Sync, RDMA-Async, RDMA-Sync and e-RDMA-Sync schemes compared to Socket-Async scheme with RUBiS and Zipf Trace

Performance with Coarse-grained vs Fine-grained Monitoring

In order to understand the impact of granularity of resource monitoring with applications, we evaluate the performance for different load fetching granularity from 64 *m*secs to 4096 *m*secs. Figure 2.15 shows the throughput performance of a RUBiS benchmark and Zipf trace with α value 0.5 running simultaneously for different load fetching granularity. We see that the throughput increases for decreasing granularity for the RDMA-Sync scheme. With granularity 1024 *m* secs, all four schemes report comparable performance. As the granularity decreases to 64 *m*secs, we see a performance degradation for Socket-Sync and Socket-Async schemes. Thus, traditional resource monitoring approaches based on sockets cannot be used for fine-grained resource monitoring. On the other hand, we see an improvement close to 25% for the RDMA-Sync scheme compared to the rest of the schemes when the granularity is

64 *m*secs showing the performance benefits of fine-grained resource monitoring with applications. Thus, our results indicate that fine-grained monitoring can significantly improve the overall utilization of the system and accordingly lead to up to 25% improvement in the number of requests the cluster system can admit.

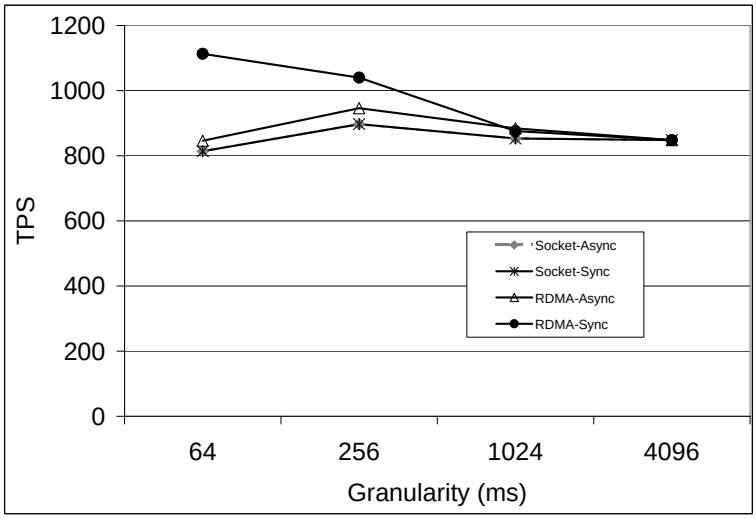


Figure 2.15: Fine-grained vs Coarse-grained Monitoring

2.3.5 Performance with Active Resource Adaptation Services

In this section, we evaluate the proposed active resource adaptation (reconfiguration) service, as mentioned in Section 2.2.5. We evaluate the service on two interconnects IBA and Ammasso using the OpenFabrics implementation. As mentioned earlier, the iWARP implementation of OpenFabrics over Ammasso was available only at the kernel space. We wrote a wrapper for user applications which in turn calls the kernel module to fire appropriate iWARP functions. Our experimental testbed

consists of a 12 node cluster with dual Intel Xeon 3.4 GHz CPU-based EM64T systems. Each node is equipped with 1 GB of DDR400 memory. The nodes were connected with MT25128 Mellanox HCAs (SDK v1.8.0) connected through a InfiniScale MT43132 24-port completely non-blocking switch. For Ammasso experiments we use two node dual Intel Xeon 3.0 GHz processors with a 512 KB L2 cache and a 533 MHz front side bus and 512 MB of main memory.

As shown in Figure 2.16a, we see that the average reconfiguration time is only 133μs for increasing loaded servers. The x-axes indicates the number of servers that are currently heavily loaded. The substrate overhead is only around 3μs and more importantly, as the number of loaded servers increases, we see no change in the reconfiguration time. This indicates that the service is highly resilient to the loaded conditions in the datacenter environment. Further, we observe that the number of reconfigurations increase linearly as the number of loaded servers increase from 5% to 40%. Increasing the loaded servers further does not seem to affect the reconfiguration time and when this reaches 80%, the number of reconfigurations decreases mainly due to insufficient number of free servers for performing the reconfiguration. Also, for increasing number of reconfigurations, several servers get locked and unlocked in order to perform efficient reconfiguration. Figure 2.16 also shows that the contention for acquiring locks on loaded servers does not affect the total reconfiguration time showing the scalable nature of this service. Figure 2.16b shows the performance of TCP and DDSS substrate for increasing back-end load on the server. We see that the performance of TCP degrades significantly with increasing back-end load, whereas the performance of DDSS remains unaffected with increasing back-end load. This demonstrates the load-resilient capability of DDSS in datacenter environments.

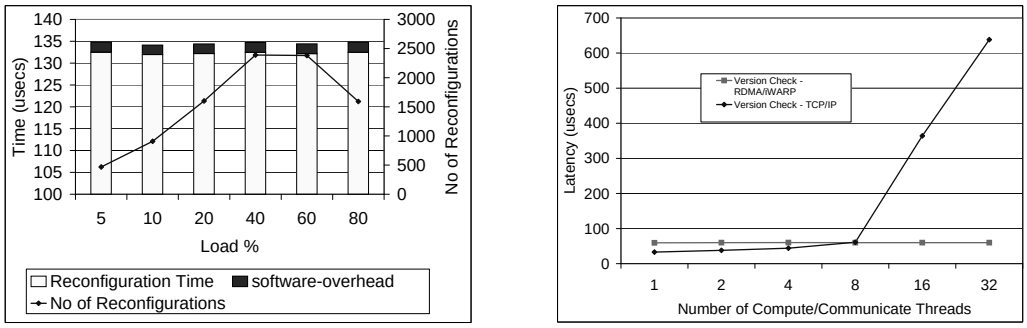


Figure 2.16: Software Overhead on Datacenter Services (a) Active Resource Adaptation using OpenFabrics over IBA (b) Comparison of TCP and DDSS performance using OpenFabrics over Ammasso

2.4 Summary

In this Chapter, we proposed and evaluated a low-overhead network-assisted state sharing substrate for cluster-based datacenter environment. Traditional implementations of data sharing using ad-hoc messaging protocols often incur high overheads and are not very scalable. The proposed substrate is designed to minimize these overheads and provide high performance by leveraging the features of modern interconnects like RDMA and atomic operations. The substrate performs efficient data and memory management and supports a variety of coherence models. The substrate is implemented over the OpenFabrics standard interface and hence is portable across multiple modern interconnects including iWARP-capable networks both in LAN and WAN environments. Experimental evaluations with IBA and iWARP-capable Ammasso networks through micro-benchmarks and datacenter services not only showed an order of magnitude performance improvement over traditional implementations but also showed the load resilient nature of the substrate. Further, our evaluations

with fine-grained resource monitoring services show that our approach can significantly improve the overall utilization of the system and accordingly lead to up to 25% improvement in the number of requests the cluster system can admit. Application-level evaluations with Distributed STORM using DataCutter achieved close to 19% performance improvement over traditional implementation, while evaluations with check-pointing application suggest that the state sharing substrate is scalable and has a low overhead.

CHAPTER 3

DMA-ACCELERATED STATE SHARING USING INTEL'S I/OAT

In this Chapter, we utilize the advanced features of I/OAT for designing efficient state sharing substrate. Figure 3.1 shows the various components of DMA-accelerated state sharing substrate. Broadly, in the figure, we focus on the colored boxes for designing efficient DMA-accelerated state sharing components and understanding its benefits with datacenter applications. The dark colored boxes show the features and technologies that we utilize and the light colored boxes show the proposed components and datacenter system software evaluated.

3.1 Background and Related Work

Apart from communicating across the nodes in a datacenter, several datacenter applications and services also communicate across different threads running in the same system. For example, several application servers communicate between themselves (may exchange shared data) for query processing. Several web server threads communicate with services such as caching and resource monitoring for cached content or back-end system information using System V IPC communication. Due to limitations in allocating large pool of shared memory, the resource or the data that

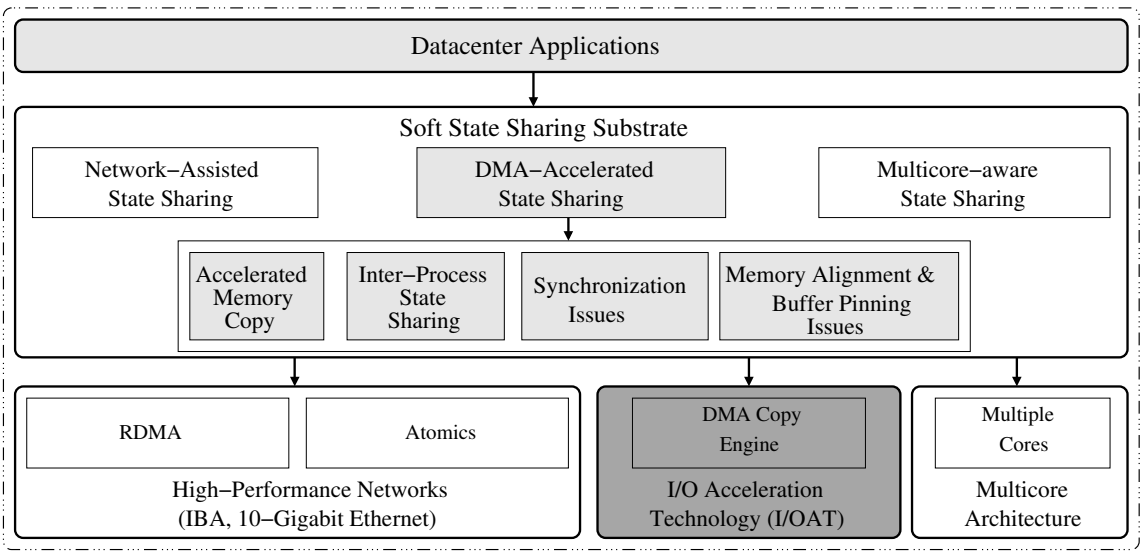


Figure 3.1: DMA-Accelerated State Sharing Components

is shared is also copied several times while communicating between the datacenter threads. The limited memory bandwidth is often addressed as the major performance degradation factor for many of these applications. Several memory block operations such as copy, compare, move, etc., are performed by the host CPU leading to an inefficient use of the host compute cycles. In addition, such operations also affect the caching hierarchy since the host CPU fetches the data onto cache, thereby, evicting some other valuable resources in cache. The problem gets even worse with the introduction of multicore systems since several cores can concurrently access the memory leading to memory contention issues, CPU stalling issues, etc. Due to several of the issues mentioned above, the ability to overlap computation and memory operation as a memory latency-hiding mechanism becomes critical for masking the gap between processor and memory performance.

To alleviate such issues, in this Chapter, we propose a mechanism which utilizes a DMA engine for accelerating the copy operation and overlaps computation with memory copy operation by using Intel's I/OAT.

Related Work: Researchers have proposed several solutions for asynchronous memory operations in the past. User-level DMA [69, 35] deal with providing asynchronous DMA explicitly at the user space. Zhao et. al [103] talk about hardware support for handling bulk data movement. Calhoun's thesis [39] proposes the need for dedicated memory controller copy engine and centralized handling of memory operations to improve performance. However, many of these solutions are simulation-based. Ciaccio [49] proposed the use of self-connected network devices for offloading memory copies. Though this approach can provide an asynchronous memory copy feature, it has a lot of performance-related issues. I/OAT [58] offers an asynchronous DMA copy engine (ADCE) which improves the copy performance with very little startup costs.

3.2 Design and Implementation Issues

In this section, we propose our design [93] for accelerating the memory copy and IPC operations using an asynchronous DMA engine offered by I/OAT.

3.2.1 Accelerated Memory Copy using Asynchronous DMA Engine

As mentioned in Section 1.2.2, Intel's I/OAT offers an asynchronous DMA engine to offload the memory copy operation from the host CPU. Currently, Intel supports several interfaces in kernel space for copying data from a source page/buffer to a destination page/buffer. These interfaces are asynchronous and the copy is not guaranteed

to be completed when the function returns. These interfaces return a non-negative cookie value on success, which is used to check for completion of a particular memory operation. It is necessary to wait on another function to wait for the copies to complete.

A memory copy operation typically involves three operands: (i) a source address, (ii) a destination address and (iii) number of bytes to be copied. For user-space applications, the source and destination addresses are virtual addresses. However, as mentioned in Section 1.2.2, the DMA copy engine can only understand physical addresses. The first step in performing the copy is to translate the virtual address to physical addresses. For various reasons related to security and protection, this is done at the kernel space. Once we get the physical address, we also need to make sure that the physical pages that are mapped to the user application does not get swapped onto the disk while the copy engine performs the data transfer. Hence, we need to lock the pages in memory before initiating the DMA and unlock the pages after the completion of the copy operation, if required. We use the *get_user_pages()* function in the kernel space to lock the user pages.

Operation	Description
adma_copy(src, dst, len)	Blocking copy routine
adma_icopy(src, dst, len)	Non-blocking copy routine
adma_check_copy(cookie)	(Non-blocking) check for completion
adma_wait_copy(cookie)	(Blocking) wait for completion

Table 3.1: Basic ADCE Interface

In order for the datacenter applications to use the copy engine, we propose the addition of the following interfaces, as shown in Table 3.1. The *adma_icopy* operation helps in initiating the copy and returns a cookie which can be used later to check for completion while the *adma_copy_check* operation helps in checking if the corresponding memory operation has completed. The *adma_copy_wait* operation waits for the corresponding memory operation to complete and the *adma_copy* operation is a blocking version which uses the copy engine and does not return until the copy finishes. The basic design of our proposed approach is shown in Figure 3.2.

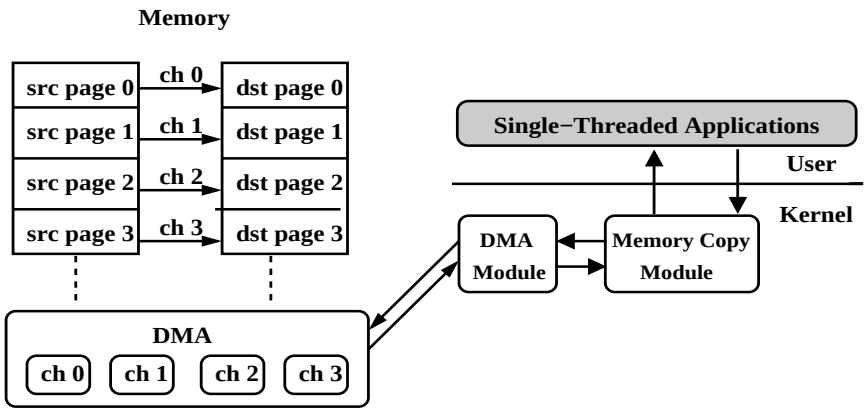


Figure 3.2: Memory Copy Design using DMA Engine

3.2.2 Inter-Process State Sharing using Asynchronous DMA Engine

In addition to offloaded memory operations within a single process, applications also require support for exchanging messages across different processes in a single node. As shown in Figure 3.3, there are many ways of performing inter process communication. The most common way followed is the user space shared memory

based approach. In this approach, processes A and B create a shared memory region. Process A copies the source data onto the shared memory and process B copies this shared memory segment to its destination. Clearly, this approach involves an extra copy. Several HPC middle-ware layers such as MPI use this approach [44]. As mentioned in Section 1.2.2, this approach also occupies some CPU resources. Another approach is the NIC-based loop back approach wherein network device can DMA the data from the source to the destination. The third approach [62] is to map the user buffer in kernel space and use the standard copy operation in kernel to avoid an extra copy incurred by user space shared memory approach. In addition, System V IPC communication can also be utilized to communicate across different processes. However, this mechanism not only involves several memory copies but also involves the operating system for context-switches, interrupts, etc., leading to significant overheads. We propose an approach that utilizes the DMA copy engine to perform the copy. Such an approach does not incur any extra copies, not touch many CPU critical resources and also avoids any cache pollution effects.

We support the following user interface, as shown in Table 3.2, for applications to exchange messages across different processes. The *adma_read* and *adma_write* operations read and write data onto another process and *adma_iread* and *adma_iwrite* operations initiate the data transfer. However, due to the presence of two different processes, synchronization becomes a bottleneck for performance. Data transfer cannot be initiated unless both the processes have posted their buffers for data transfer. The *adma_icheck* operation checks whether the memory operation has completed and the *adma_wait* operation waits till the memory operation completes.

Operation	Description
adma_iread(fd, addr, len)	Non-blocking read routine
adma_iwrite(fd, addr, len)	Non-blocking write routine
adma_read(fd, addr, len)	Blocking read routine
adma_write(fd, addr, len)	Blocking write routine
adma_check(cookie)	(Non-blocking) check for read/write completion
adma_wait(cookie)	(Blocking) Wait for read/write completion

Table 3.2: ADCE Interface for IPC

Figure 3.4 shows the mechanism by which we support IPCs. Our design can be easily integrated with the pipe or socket semantics. Currently, we support the socket semantics for establishing the connection between different processes. Once the connection is made, processes can use the set of interfaces mentioned above for utilizing the copy engine. Let us consider two connected processes (A and B). If process A needs to send data to process B, process A makes a request to the kernel (Step 1). In step 2, the kernel locks the user page and adds the entry to a list of cached virtual to physical mappings. The kernel then makes an entry to a list of pending read and write requests. At this time, if process B posts its read buffer (Step 4), the kernel locks the user page and caches the page mapping (Step 5). The kernel searches the list to find the matching write request (Step 6). Since the write buffer is already posted, it initiates the DMA copy (Step 7). Process A waits for the completion of operation (Step 8) by issuing a request to the kernel. The kernel first makes sure that the corresponding buffers are posted by waiting on a semaphore (Step 9a). This

semaphore is initially in a locked state and released when both the read and write buffers match. Steps 10-11 are similar to Steps 8-9.

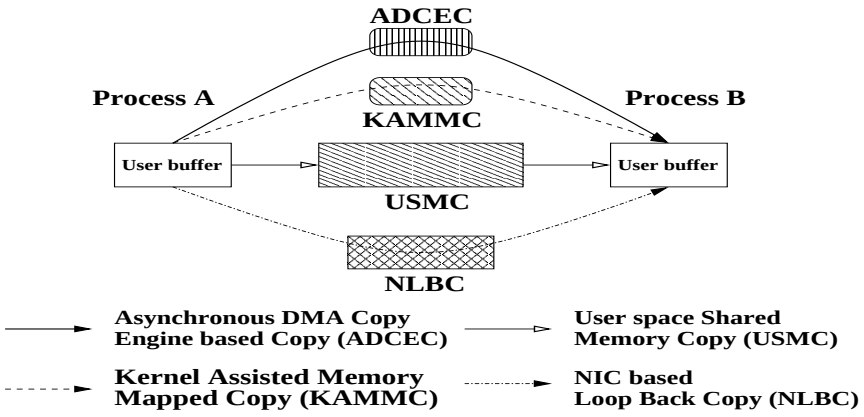


Figure 3.3: Different IPC Mechanisms

3.2.3 Handling IPC Synchronization Issues

Since we have two different processes performing communication using the copy engine, synchronization becomes a critical issue before initiating the DMA transaction. For example, consider processes A and B wanting to communicate a buffer of size 1 KB. We need to handle the following cases making sure that latency, progress and CPU utilization do not get affected significantly. Case 1: Process A posts the write buffer and waits for the operation to finish. Then process B posts an *adma_iread* operation. Case 2: Process B posts the read buffer and waits for the operation to finish. Then process A posts an *adma_iwrite* operation. Case 3: Both processes A and B post their respective buffers before performing the wait operation. To address these cases, we use a binary semaphore in our implementation. For Case 1, we queue the request posted by Process A during the write request and we allow the process

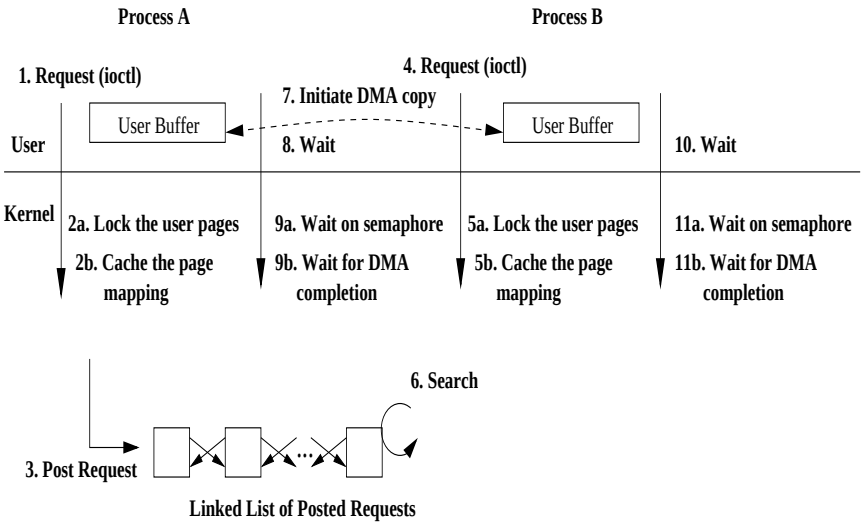


Figure 3.4: IPC using DMA copy engine

to wait on the semaphore during the wait operation. When Process B posts a read buffer, the DMA is initiated and immediately process A is woken up by releasing the semaphore. Process A then waits on the DMA copy to finish and the control is given back to the user process. For Case 2, a similar approach is followed except that Process A wakes up process B after process A posts the corresponding write buffer. In Case 3, both processes A and B see a matching request posted and thus do not wait on any semaphore and directly check for DMA completion. All three cases avoid unnecessary polling and the control is released immediately after the buffers are posted so that DMA completion is checked immediately leading to better notification.

3.2.4 Handling Memory Alignment and Buffer Pinning Issues

Another issue is the memory alignment problem associated with source and destination buffers. Since the copy engine operates with main memory, the performance

of the copy operation can be enhanced if the memory is page-aligned. For example, lets say that the source address starts at offset 0 and the destination address at 2K. If we assume the page size to be 4 KB, then we can only schedule a maximum of 2 KB copy since the copy length is required to be within the page-boundary, leading to 2000 such operations if we assume a 4 MB data transfer. On the other hand, if the addresses were page-aligned, we only need 1000 such operations. In the worst case, we may end up issuing copies for very small messages (<100 bytes) for several iterations. Clearly, by making the addresses page-aligned, we can save on the number of copy operations and more importantly avoid issuing very small data transfers using the copy engine.

As mentioned in Section 1.2.2, the copy engine deals with physical addresses as it directly operates on main memory. To avoid swapping of user pages to the disk during a copy operation, it is mandatory that the kernel locks the user buffers before initiating the DMA copy and releases the user buffers once the copy completes. Usually this locking/unlocking cost is quite large, in the order of μs contributing significantly to the total time required for data transfer. To reduce this cost, we lock the buffers initially and do not release the locked buffers even after the completion of data transfer. For subsequent data transfers, if the same user buffer is reused, we can avoid the locking costs and directly use the physical address that maps to the virtual address. However, the kernel module needs to be aware of any changes in memory usage in applications such as allocate and release, and appropriately release these buffers in the kernel.

Several applications can use the DMA engine simultaneously. Hence, it is possible that a small memory operation is queued behind several large memory operations.

Due to the fact that we have several DMA channels, scheduling these memory operations on appropriate channels becomes a challenging task. Currently, we use a simple approach of using the channels in a round-robin manner and schedule the memory operations.

Apart from these issues, due to the other features mentioned in Section 1.2.2, I/OAT can also directly improve the performance of the datacenter both in terms of CPU utilization and the number of requests a datacenter can admit. More details can be found in [98].

3.3 Experimental Evaluation

We ran our experiments on a dual dual-core Intel 3.46 GHz processors and 2 MB L2 cache system with SuperMicro X7DB8+ motherboards which include 64-bit 133 MHz PCI-X interfaces. The machine is connected with an Intel PRO 1000Mbit adapter. We used the Linux RedHat AS 4 operating system and kernel version 2.6.9-30.

3.3.1 Memory Copy Performance

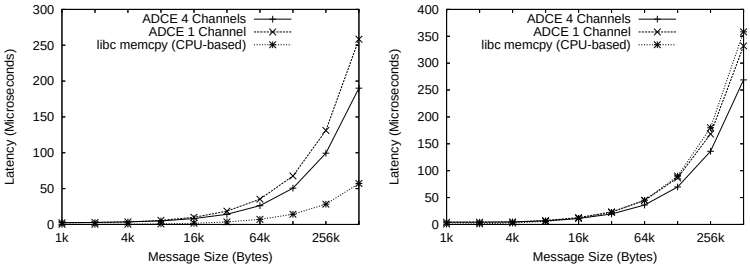
First, we compare the performance of the copy engine with that of traditional CPU-based copy and show its benefits in terms of performance and overlap efficiency. For the CPU-based copy, we use the standard *memcpy* utility.

Latency and Bandwidth Performance: Figure 3.5(a) shows the performance of copy latency using CPU and ADCE (DMA-accelerated state sharing approach) for small message sizes. In this experiment, both source and destination buffers fit in the cache. For the CPU-based copy operation, we measure the *memcpy* operation of *libc* library and average it over several iterations. This is indicated as *libc memcpy (CPU-based)* line in the figure. For the ADCE approach, we use the *adma_icopy*

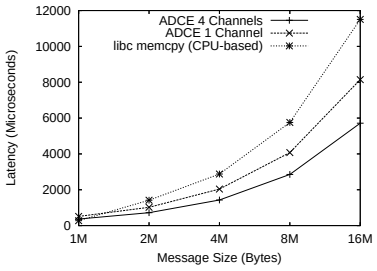
operation followed by the *adma_wait* operation and measure the time to finish both the operations.

As shown in Figure 3.5(a), we see that the CPU-based approach performs well for all message sizes. This is mainly due to the cache size which is 2 MB. Since both source and destination buffers can fit in the cache, the CPU-based approach performs better. Figure 3.5(b) show the performance of copy latency for small messages when source and destination buffers are not in the cache. In this experiment, we use two 64 MB buffers as source and destination. After every copy operation, we move the source and destination pointers by the message size, so that memory copy always uses different buffers. We repeat this for a large number of iterations and ensure that the buffers are not in the cache. As observed in the figure, we see that the ADCE approach using four channels performs better from 16 KB. As mentioned earlier, since we are using buffers that are not in the cache, for the ADCE approach, we also incur penalties with huge pinning costs for every copy operation. As a result, we see that the performance is little worse compared to the previous experiment where the buffers are in the cache. Also, the performance of the ADCE approach with one channel gets better after 256 KB message size. However, as shown in Figure 3.5(c), we see that the performance of the ADCE approach for large message sizes is significantly better than the CPU-based approach. For 4 MB message size, we observe that the ADCE approach with four channels results in 50% improvement in latency as compared to the CPU-based approach. Also, we observe that the ADCE approach using four channels achieves better latency compared to the ADCE approach with one channel.

The bandwidth performance of copy operation is shown in Figure 3.6. In this experiment, we post a window of *adma_icopy* operations (128 in our case) and wait for



(a) Small Message hot-cache (b) Small Message cold-cache



(c) Large Message hot-cache

Figure 3.5: Memory Copy Latency Performance

these memory operations to finish. We repeat this experiment for several iterations and report the bandwidth. For the CPU-based approach, we use the *libc memcpy* instead of the *adma_icopy* operation. As shown in Figure 3.6, for message sizes till 1 MB, the CPU-based approach yields a maximum bandwidth of 9189 MB/s. This is mainly due to the caching effect since the copy happens inside the cache. For message sizes greater than 1 MB, we observe a huge drop in bandwidth for CPU-based approach achieving close to 1443 MB/s. However, the ADCE approach with four channels achieves a peak bandwidth of 2912 MB/s, almost double the bandwidth achieved by the CPU-based approach. The ADCE approach with one channel achieves close 2048 MB/s.

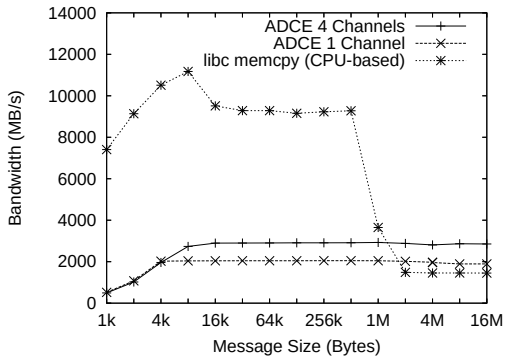


Figure 3.6: Memory Copy Bandwidth

Cache Pollution Effects: In this section, we measure the effect of cache pollution with applications. We design the experiment in the following way. We perform a large memory copy operation and perform a column-wise access of a small memory buffer which can fit in the cache. Figure 3.7 shows the access time for various memory sizes. We measure the access time without the memory copy and report it as *access*

w/o copy and for remaining cases, we perform the memory copy using CPU and the ADCE approach. As shown in figure, the access time after performing the copy using the ADCE approach does not change with the normal access time. However, the CPU-based approach increases the access time by 30% due to cache eviction. Since the ADCE approach operates directly on main memory, it avoids cache pollution effects. As a result, the access latency after using the ADCE approach does not change. However, the CPU-based approach evicts some of the entries in the cache resulting in an increase in access time latency.

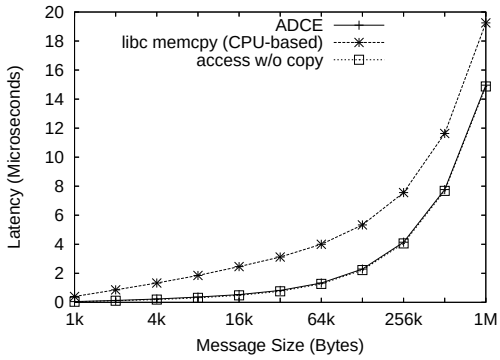


Figure 3.7: Cache Pollution Effects

3.3.2 Overlap Capability

In this section, we evaluate the ability of the ADCE approach to effectively overlap memory copy process and computation. To carry this evaluation, we design an overlap benchmark. For a certain message size, the benchmark first estimates the latency of blocking memory copy T_{copy} (*adma_icopy* operation immediately followed by *adma_wait* operation). To test the overlap efficiency, the benchmark initiates an

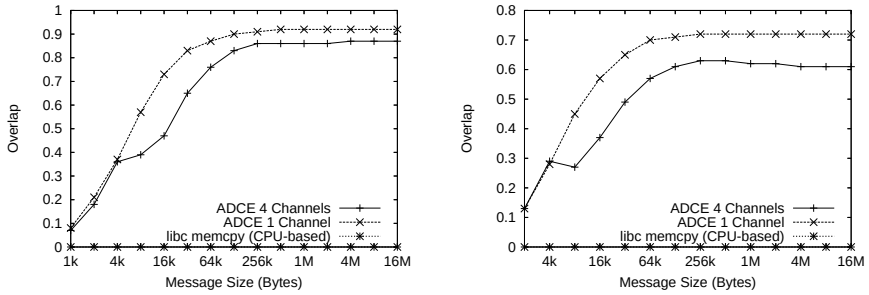
asynchronous memory copy (*adma_icopy*) followed by a certain amount of computation which at least takes time $T_{compute} > T_{copy}$, and finally waits for the completion (*adma_wait*). The total time is recorded as T_{total}. If the memory copy is totally overlapped by computation, we should have $T_{total} = T_{compute}$. If the memory copy is not overlapped, we should have $T_{total} = T_{copy} + T_{compute}$. The actual measured value will be in between, and we define overlap as:

Overlap = (T_{copy} + $T_{compute}$ - T_{total}) / T_{copy}

Based on the above definition, the value of *overlap* will be between 0 (non-overlap) and 1 (totally overlapped). A value close to 1 indicates a higher overlap efficiency. Figure 3.8(a) illustrates the overlap efficiency we measured. As we can see, the CPU-based copy using *memcpy* is blocking, thus we always get an overlap efficiency of 0. By using the ADCE approach for large size memory copies, we are able to achieve up to 0.92 (92%) and 0.87 (87%) overlap using one and four channels, respectively. For the ADCE approach with four channels, we check the completion across four channels and thus it results in lesser overlap compared to the ADCE approach with one channel case. For smaller sizes, the overlap efficiency is small due to DMA startup overheads. We see similar trend in overlap efficiency when the source and destination buffers are not in the cache as shown in Figure 3.8(b). However, the actual percentages seen are much lower. We explain the reason for such lower percentages in the section below.

3.3.3 Asynchronous Memory Copy Overheads

In order to understand the low overlap efficiency observed in the previous section, we measure the split-up/overhead of the ADCE approach. Figure 3.9 shows the split-up overhead of the ADCE approach using four channels. In this experiment, we



(a) Hot-Cache (b) Cold-Cache

Figure 3.8: Computation-Memory Copy Overlap

ran the copy latency test with source and destination buffers not in the cache and measure the overhead of user/kernel transition, pinning of user buffer, DMA startup and completion. We observe that the pinning cost occupies a significant fraction of the total overhead. For small message sizes, we see that all four overheads contribute equally towards the latency and there is very little room for overlap. For larger message sizes, we see that the pinning cost and DMA startup cost occupies 30% and 7%, respectively. The remaining time is overlapped with the computation (62%).

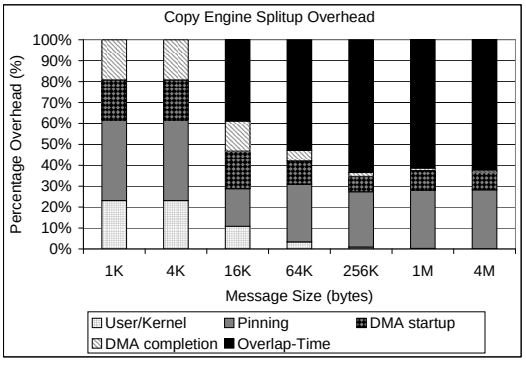
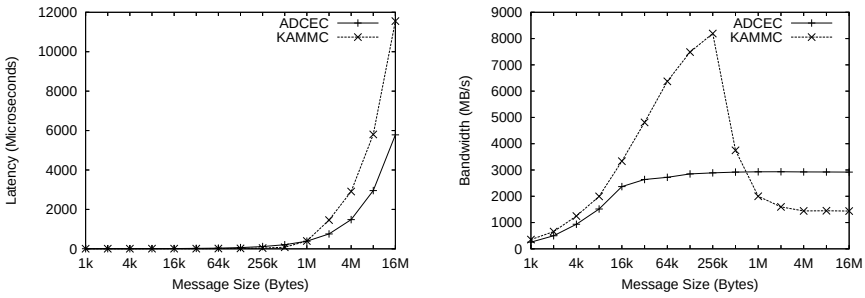


Figure 3.9: Asynchronous Memory Copy Overheads

3.3.4 Inter-Process State Sharing Performance

Figure 3.10(a) shows the IPC latency for ADCE based copy (ADCEC) and Kernel-assisted memory mapped based copy (KAMMC). For 4 MB message size, we see that the ADCEC approach achieves close to 2954 μs whereas the KAMMC approach achieves close to 5803 μs. Further, for increasing message sizes, the performance of the ADCEC approach is much better than the KAMMC approach.

Figure 3.10(b) shows the IPC bandwidth with ADCEC approach and KAMMC approaches. Since the buffers can fit in the cache, we observe that the performance of the KAMMC approach is better than the ADCEC approach till 256 KB message size achieving close to 8191 MB/s. However, for message sizes greater than 1 MB, we see that the ADCEC approach achieves 2932 MB/s whereas the KAMMC approach achieves only 1438 MB/s.



(a) Latency (b) Bandwidth

Figure 3.10: Inter-Process Communication Performance

3.4 Summary

Intel's I/O Acceleration Technology offers an asynchronous memory copy engine in kernel space that alleviates copy overheads such as CPU stalling, small register-size data movements, etc. In this Chapter, we proposed a set of designs for asynchronous memory operations in user space for both single process (as an offloaded *memcpy()*) and IPC using the copy engine. We analyzed our design based on overlap efficiency, performance and cache utilization. Our microbenchmark results showed that using the copy engine for performing memory copies can achieve close to 87% overlap with computation. Further, the copy latency of bulk memory data transfers is improved by 50%.

CHAPTER 4

MULTICORE-AWARE STATE SHARING USING MULTICORE ARCHITECTURES

In this Chapter, we utilize the advanced features of multi-core systems for designing efficient state sharing substrate. Figure 4.1 shows the various components of multicore-aware state sharing substrate. Broadly, in the figure, we focus on the colored boxes for designing efficient multicore-aware state sharing components and understanding its benefits with applications. The dark colored boxes show the features and technologies that we utilize and the light colored boxes show the proposed components and datacenter system software evaluated.

4.1 Background and Motivation

Many high-performance networks such as InfiniBand [61], 10-Gigabit Ethernet [60], Quadrics [18], Myrinet [36] and JNIC [83] support Remote Direct Memory Access (RDMA) [19] to provide high-performance and scalability to applications. While there are multiple RDMA standards, in this Chapter, we use the generic term to denote one-sided inter-node memory access. Unlike two-sided sends and receives, one-sided operations access remote memory without requiring the remote application's participation. RDMA often combines one-sided execution with OS-bypass to

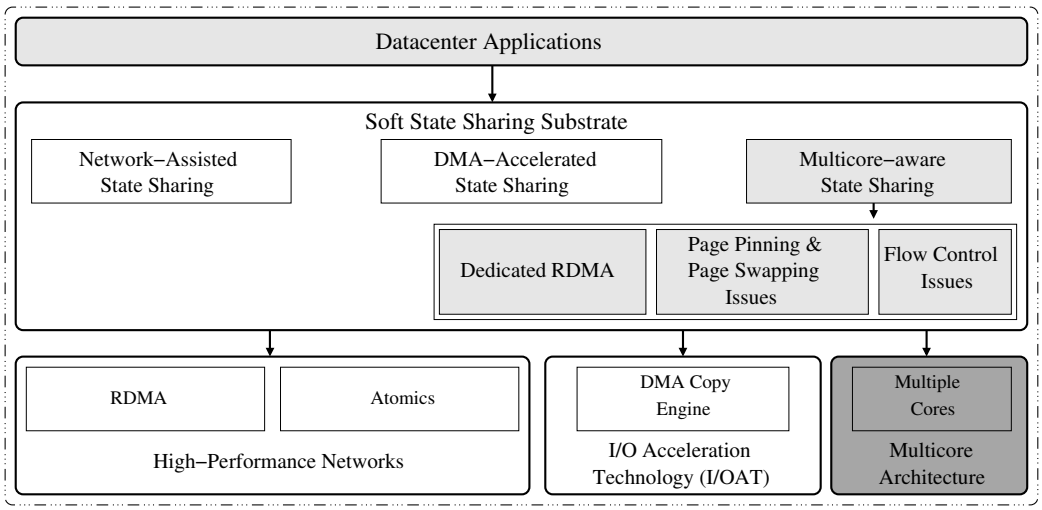


Figure 4.1: Multicore-aware State Sharing Components

achieve low latency and high bandwidth. This provides network-centric application primitives that achieve three major objectives: data is delivered without expensive software-based copies; concurrency is improved when one-sided access occurs without remote application cooperation; application processing is reduced when asynchronous remote processing is moved away from the application. RDMA defines two primitives: a *put* writes to, and a *get* reads from remote memory.

In the high-performance computing domain, the utility of RDMA is already proven [73, 74]. MPI is the most popular standard for parallel computing and networks such as InfiniBand, Quadrics, etc., use RDMA to accelerate MPI's two-sided messaging. More recently, MPI also includes direct support for one-sided communications that exploits optimized RDMA. RDMA's get and put operations are natural communication primitives for Partitioned Global Address Space (PGAS) Languages, such as Unified Parallel C (UPC) [5] and Co-Array Fortran [75], that incorporate benefits from message passing's scalability and shared memory's ease-of-programming.

RDMA also offers potential in datacenter environments [85]. The distributed applications [91] hosted in these environments such as web and application servers, file systems, caching and resource management services can significantly benefit from RDMA for achieving datacenter-wide scalability. Researchers have proposed several lower-level mechanisms such as Sinfonia [28], Khazana [42], DDSS [97] to build efficient datacenter sub-systems including cluster file system, distributed lock manager, and databases. These mechanisms typically deal with memory-based objects and manipulate these objects frequently. Thus, it is important to provide efficient distributed manipulation of memory-based objects using *get* and *put* operations to increase the performance and scalability.

While existing Remote Direct Memory Access (RDMA) provides a foundation, a closer inspection indicates that today's RDMA is not suitable for many of these environments. Firstly, existing RDMA implementations do not preserve all of the benefits of virtual memory to applications such as the illusion of using more memory than that is physically present and the protection capabilities for memory regions that are shared among user programs. Secondly, the memory regions used for RDMA are typically managed by users in an independent manner. Multiple users making independent decisions can lead to starvation of resources and robustness issues (e.g., a system crash due to unavailable pages). Networks such as Quadrics address some of these limitations by using complex NIC hardware that maintains the page tables and frequently interacts with the operating system. In this Chapter, we address these limitations by proposing a software-centric onload approach.

Next, we present the capabilities of the JNIC architecture and the registration issues with RDMA.

4.1.1 JNIC Architecture

The Intel/HP Joint Network Interface Controller (JNIC) [83] prototype models in-datacenter communications over Ethernet. Figure 4.2 shows a JNIC system consisting of front-side-bus-attached NIC hardware and optimized software using a dedicated kernel helper. The prototype hardware is an FPGA-based Gigabit Ethernet NIC that plugs into an Intel Xeon socket, allowing communication over the front side bus. A reliable communications (VNIC) layer implements JNIC-to-JNIC communication using the TCP protocol. The VNIC layer presents virtual NIC device interfaces to user or kernel tasks. Messages are sent by multiplexing message requests from VNIC clients to the reliable communications layer. Messages are received when the reliable communications layer receives a message and delivered to the appropriate receiving destination VNIC. The VNIC implements copy-free transport using physically addressed DMA. At the time of VNIC-layer registration, VNIC source and target buffers are physically locked/pinned regardless of when they are needed for future RDMA. For more details on JNIC architecture and VNIC layer, the readers are encouraged to refer to [83].

In this Chapter, we describe the architecture of a working onload-style RDMA implementation. This architecture is carefully crafted to ensure forward progress even when client applications compete for limited resources. To support multiple client RDMA operations on regions of arbitrary size, a number of key tasks are executed. Tasks include: flow control, region segmentation, page pinning, initiating copy-free transport, and page unpinning.

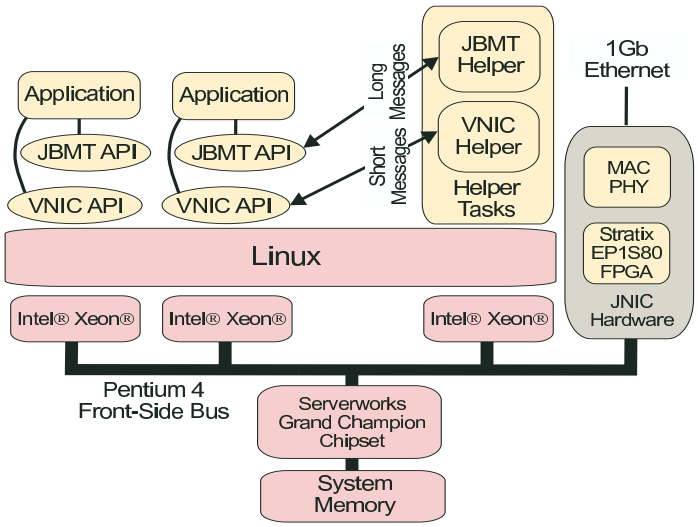


Figure 4.2: JNIC Prototype [83]

4.1.2 RDMA Registration

RDMA typically involves two steps: registration and the actual *put* or *get*. The registration process can be broadly classified into pinning-based registration [61, 60] and hardware-assisted registration [18, 76]. In this section, we present the details on these two registration strategies and its associated issues.

Pinning-based Registration: In this method, to register a buffer, a task makes a system call into a kernel component of the RDMA service. The kernel initializes the control data and creates a handle that encodes buffer access rights. The kernel then swaps in and pins (locks) all the buffer pages. After a successful buffer pinning, the kernel component asks the NIC to install a mapping between the handle and the pinned physical pages and waits for an acknowledgment. The buffer handle is then passed back to the user task after receiving the acknowledgment, which in turn sends the buffer handle to remote tasks to be used in subsequent *get* (or *put*) requests.

Similarly, during a de-registration operation, the kernel component asks the NIC to remove the mapping between the handle and the pinned physical pages and waits for an acknowledgment. The kernel then unlocks all the buffer pages associated with the handle and removes the corresponding entries. Due to the page pinning restriction in the registration phase, this approach not only limits the amount of buffers registered to available physical memory but can also waste the physical memory, if it is currently not utilized. Finally, allowing users to pin physical memory compromises robustness. One buggy application affects all others by monopolizing physical memory. Further, the cost of this registration and de-registration is typically expensive [67, 100] in networks such as InfiniBand.

Hardware-assisted Registration: In this approach (e.g. Quadrics [18]), the NIC combines a hardware TLB and tweaks in operating system's virtual memory support to allow the NIC to pin pages, initiate page faults and track changes in the application's page table. In this approach, there is no restriction on what memory is DMA-able, potentially the entire virtual memory can be made available for RDMA operations. However, pushing the responsibility of pinning and managing the page tables to the NIC comes with increased hardware cost and complexity [33, 101]. Further, the hardware requires frequent updates as changes are made to the internal virtual memory subsystems.

4.2 Related Work

Modern processors are seeing a steep increase in the number of cores available in the system [51]. As the number of cores increases, the choice to dedicate one or more cores to perform specialized functions will become more common. In this work,

we proposed a mechanism to onload the RDMA tasks to a dedicated kernel helper thread for such systems. Researchers [61, 18, 60, 76, 66, 81] have proposed several mechanisms in designing RDMA operations and onloading techniques. In this work, we combined the existing dynamic page pinning and onloading technique to perform RDMA related tasks. Unlike conventional RDMA, buffer pinning is kernel-managed to allow the system to have better control over critical resources in our proposed design. Our architecture preserves the benefits of virtual memory in a robust and well controlled environment. Further, the presence of a kernel helper thread avoids replication of page table entries and provides faster access to kernel page tables and page pinning, and simplifies the NIC hardware as compared to existing approaches.

In the following sections, we describe an architecture that eliminates problems associated with user pinning and hardware-based registration by combining dynamic page pinning [66] (the ability to pin a small set of pages and make progress in a pipelined manner) and onloading [81] (the ability to perform the tasks on the host processor).

4.3 Design and Implementation Issues

In this section, we present the design goals and details of our proposed onloaded service [99].

Goals: To address the limitations mentioned above, our primary goal is to allow RDMA operations on unpinned virtual memory with simplified NIC hardware. Thus, we need a mechanism that supports page pinning only when necessary in a pipelined manner but at the same time, the pinning process should be rapid and should not slow down RDMA. In addition, we need a mechanism that jointly pins the pages

on both sending and receiving side to allow a copy-free hardware transport and a mechanism that guarantees forward progress even when available memory is limited.

In the following sections, we present the detailed design of RDMA service that meets our design goals.

4.3.1 State Sharing using Onloading

The basic idea of our design is to exploit the abundant compute cycles of future many-core processors to perform the tasks involved in *get* and *put* operations. We use a dedicated kernel helper thread to perform just-in-time physical page pinning, access rights enforcement, copy-free data transport, guaranteeing progress even when the pages are not resident in memory and handling flow control. We refer to the RDMA service as JNIC's Bulk Message Transport (JBMT) and present the details in performing a JBMT *get* operation in the rest of the section.

Figure 4.3 shows the overall architecture of JBMT that provides copy-free autonomous message delivery. As shown in the figure, the JBMT kernel helper thread receives requests and generates responses through virtualized JBMT command and completion ports. In this approach, a source buffer is registered by the server. JBMT registration provides access control for the source region and it does not pin the memory pages. As a result, it does not consume valuable resources which are otherwise required to manage the pinned memory regions. On a successful registration request, a token is generated by JBMT. The resulting registration token is sent to the receiver (client) using conventional VNIC messaging, which can be used for data transfer. During a get operation, the receiver specifies the source region token, an offset into the source region, a target region pointer, and a transfer length. Legal transfers are

confined to source buffers for which access rights were granted in prior source buffer registrations.

After checking the access rights, JBMT decomposes a large message into smaller dynamically pinned sections. First, a local section (a small portion of a large *get* operation) is pinned. The local section can be any subset of the target region requested by the *get* operation as dictated by memory availability. A request for a local section is then sent to the server-side remote interface. After checking for available resources on the server, the server attempts to pin a remote section for data transfer. The remote section is again possibly a subset of the requested local section length. The kernel helper thread on the server attempts to transmit the remote section to the receiver as Ethernet-frame-size VNIC transfers. During this data transfer both the local and remote sections are pinned/locked and hence, we can directly perform DMA operations to transfer the data. After a remote section is transferred, it is unpinned and additional remote sections are pinned and transmitted until the entire requested local section has been received. After the local section is completed, it is unpinned and the next local section is pinned and the process continues until the entire buffer is transferred. Upon successful transfer of the entire buffer, a completion event is deposited in the JBMT port's completion queue, depending on application's request.

Figure 4.4 shows the detailed steps involved during a *get* operation. As shown in the figure, a *get* request is submitted as a message request into a local node command queue for processing by the dedicated JBMT kernel helper thread. This command queue (request and response) is a memory mapped queue that is shared between the application and the dedicated JBMT kernel helper thread. The helper thread detects the *get* request by polling on this command queue and the local target buffer handle

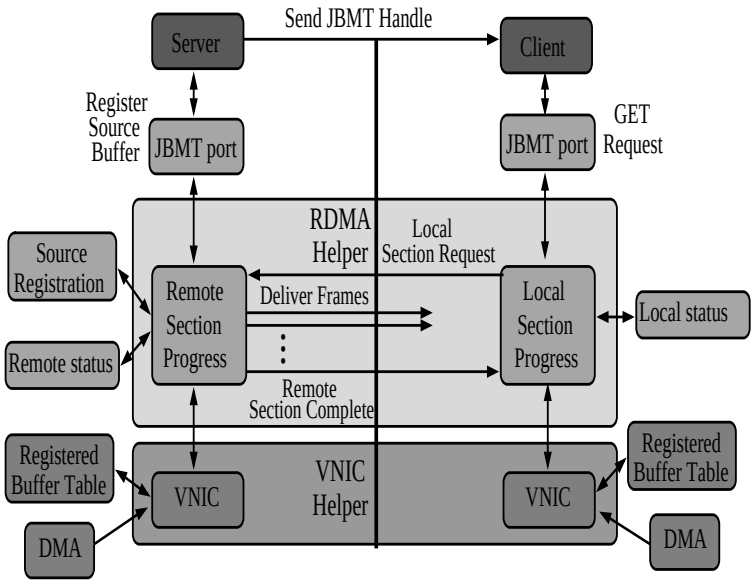


Figure 4.3: JBMT Architecture

is checked to ensure that virtual buffer access is valid. In JBMT registration, the buffers can be of arbitrary size and can exceed the size of physical memory. Hence, this requires that virtual user buffers are segmented into smaller pinned sections and processed sequentially. The *get* operation processes the target buffer handle to determine the appropriate target virtual address for data transfer. As mentioned above, the JBMT helper thread requests that the underlying VNIC layer register and pin a portion of a user buffer and return the successfully registered physical section size. This lower-layer registration pins sections of a larger virtual buffer temporarily and on demand. Further, if the pages are not resident in memory, the helper thread invokes a request to bring the swapped pages to memory and the details of how this is handled will be discussed in Section 4.3.2. Allowing the lower layer to determine the extent of physical pinning facilitates rapid progress when resources are plentiful and

ensures forward progress when resources are scarce. When resources are plentiful, a large physical section minimizes pinning costs. When resources are scarce, progress is ensured when only the head of the buffer is physically registered and as little as a single page is pinned.

Once the maximum size of the local section is determined, a local section request is sent to the server-side remote interface and similar steps are followed on server-side to determine the minimum remote section. This architecture is carefully crafted to ensure forward progress on both local and remote nodes. Upon completion of remote and local sections, the status of the corresponding request is updated, the locked pages are unpinned and proceeds to the next local section.

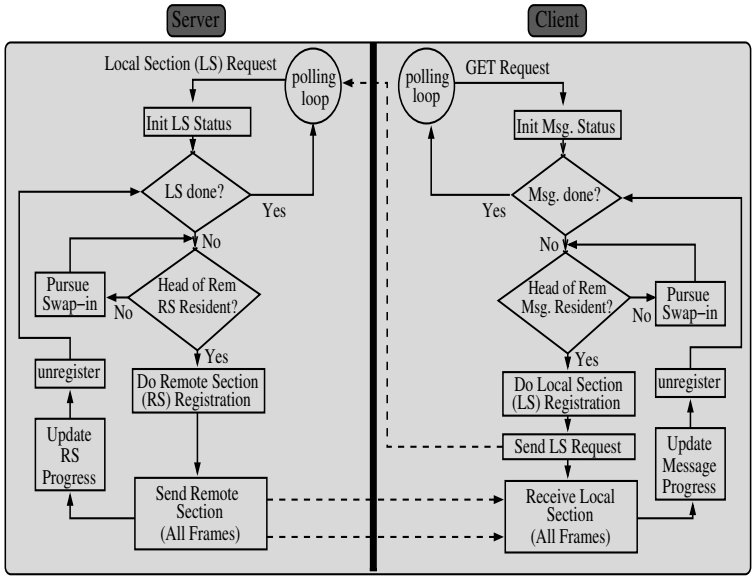


Figure 4.4: *get* Operation in JBMT

In the following sections, we present the details of how we handle page pinning while guaranteeing progress when pages are not in memory, flow control and page swapping.

4.3.2 Handling Page Pinning and Page Swapping Issues

Page pinning consists of two actions. First, pinning requires that the head of a requested buffer is physically resident. Thus, if the request buffer pages are not physically resident, JBMT kernel helper thread stimulates the swap-in of needed pages. Second, page pinning prevents future swap out of memory pages by marking the kernel tables. When pages are missing, a separate thread of execution swaps the needed pages. When no pages are resident, the resulting physical section has size zero, and DMA is delayed. JBMT keeps track of the pending RDMA tasks and periodically attempts to lock a non-empty physical section, if the head of the particular section is currently resident in memory.

In operating systems like Linux, the kernel helper function that helps to lock/pin the user pages is typically a blocking call. In other words, if the user pages are not resident in memory, the helper function usually does not return until the pages are swapped in and a lock is acquired on these pages. As a result, the dedicated kernel helper thread may block for a very long time if it attempts to directly lock the buffers that are currently not resident in memory. This would also result in blocking JBMT operations submitted by other tasks that are currently in-flight and would significantly affect the performance of any other JBMT client tasks. To avoid this scenario, we use an asynchronous page fault handler thread to handle page fault requests from JBMT kernel helper thread. The basic design of this thread is to accept a sequence

of page fault requests and make progress on these requests. This thread attempts to bring in pages from disk for a small portion of the accessed user buffer. The thread only touches the pages and does not pin or lock the pages in physical memory during a page fault miss. This can also be modified so that the first page of the buffer can be locked/pinned and the remaining pages can just be swapped in. Since the JBMT helper thread periodically checks if pages are resident in memory and immediately locks the pages if it is, locking the first page may not be necessary. Also, to process multiple page fault requests at the same time, we spawn several asynchronous page fault handler threads and the JBMT service chooses these threads in a round-robin manner to submit page fault requests.

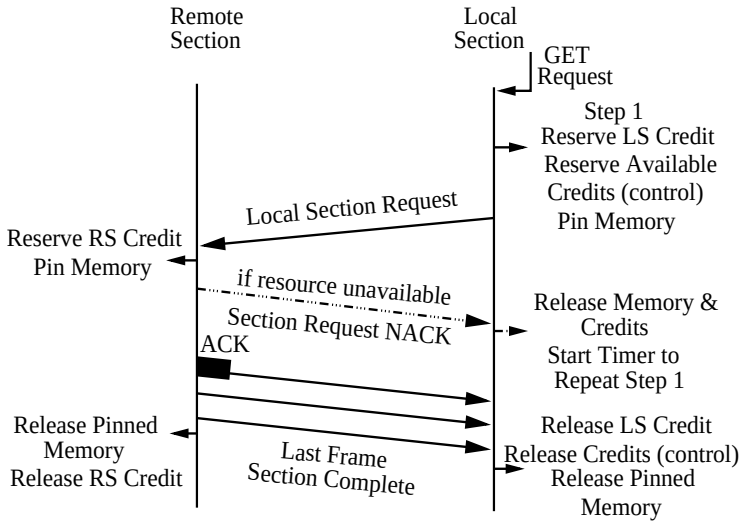


Figure 4.5: Flow Control in JBMT

4.3.3 Handling Flow Control Issues

Though the actual data is delivered through DMA, JBMT sends control commands to remote-node to perform the appropriate DMA data transfers. Thus, JBMT layer needs to ensure that there is space on the remote-side to accept control commands and thus requires flow-control mechanisms to prevent scenarios such as too many JBMT requests from a single receiver, too many local section requests and several others. As shown in Figure 4.5, we use a credit-based flow control in our design. Every JBMT *get* operation reserves a local credit in order to send a local section request (shown as Step 1 in the figure). Further, it also reserves enough control credits for the remote node to send the control commands in performing the DMA transfer for the requested local section. After the needed credits are successfully acquired, the client-side sends a local section request to the remote node. Similarly, the remote node attempts to acquire a remote section credit. This limit is applied to prevent a remote section to service a large number of local section requests. During the failure of this event, the remote-node sends appropriate NACK messages to release the credits/pages on the local-side and repeat Step 1 at a later point in time using a timer. However, if enough remote section credits are available, the remote node starts delivering the frames to the local node. After completion, the local and remote node releases the credits to process future local and remote section requests.

To summarize, our architecture provides substantial enhancement to traditional RDMA by onloading the RDMA tasks such as page pinning, DMA startup/completion, flow control, etc. Our architecture provides unlimited access to objects in unpinned virtual memory, simplifies the NIC hardware and supports more control in managing memory pages.

4.4 Experimental Results

In this section, we analyze the performance of our proposed RDMA service. Our experimental test bed consists of two nodes with two 3 GHz Xeon "Gallatin" processors with a 512 KB cache and 512 MB memory. The FPGA-based 1-Gigabit Ethernet NIC is connected via the FSB. In our prototype implementation, we design the *put* as a remote *get* operation and defer a more efficient *put* implementation for future work.

4.4.1 Latency and Bandwidth

In this experiment, we show the latency of a JBMT *get* operation. We perform the benchmark in the following way. To measure the *get* latency, we initiate a JBMT *get* operation and poll on the memory to wait for completion. After completion, we post the next JBMT *get* operation. We repeat this for several iterations and report the average, as shown in Figure 4.6. As mentioned earlier, a JBMT *get* operation involves processing local and remote sections (shown as JBMT Processing Time in the figure), a local section request (shown as VNIC Control Message Time in the figure) and the actual data transfer (shown as VNIC Data Transfer Time in the figure). We see that the latency of *get* operation for a 1 byte message is 19 μs. Further, we observe that the *JBMT Processing Time* and the VNIC Control Message Time occupies only 3 μs and 7 μs, respectively. Also, we see that the *JBMT Processing Time* does not increase with increasing message sizes. However, *JBMT Processing Time* will start varying when the buffers span over multiple pages and multiple local and remote section sizes, especially for very large *get* operations. There are existing caching techniques [100, 76] which can be used to further alleviate this overhead.

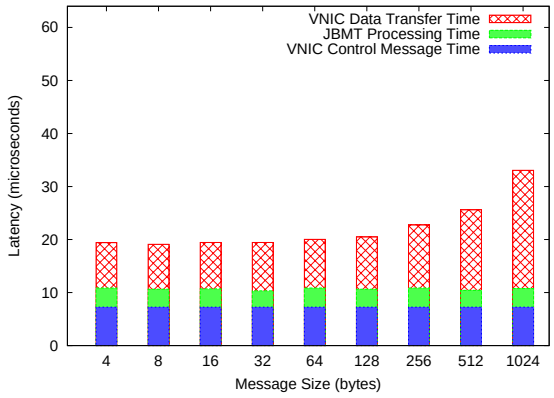


Figure 4.6: Latency of *get* operation

Next, we present the bandwidth performance of the *get* operation. To measure the *get* bandwidth, we post a window of *get* operations. After every *get* completion, we post another *get* operation and repeat this for several iterations and measure the *get* bandwidth. Figure 4.7 shows the bandwidth performance of *get* operation. We see that the JBMT *get* can achieve a peak bandwidth of up to 112 MB/s for very large messages, thus almost saturating the link bandwidth. Hence, it demonstrates that performing page pinning during a JBMT *get* operation does not significantly affect the bandwidth performance. However, for very small messages, we see that the JBMT *get* shows poor bandwidth due to several factors including latencies required for Ethernet transmission, needed page pinning in the critical path, and limitations of our prototype.

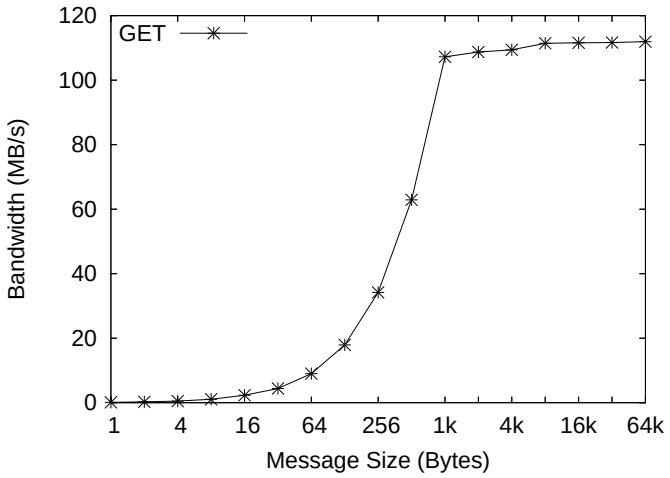


Figure 4.7: Bandwidth of *get* operation

4.4.2 Cost breakdown of Onloaded State Sharing

First, we measure the registration and de-registration cost in JBMT. This registration is different from the VNIC-layer registration which pins/locks the memory pages. JBMT registrations do not pin any pages. It only creates a local handle which can be used by peer nodes for a future JBMT operation. We perform several JBMT registrations of a particular message size and report the average latency in performing the JBMT registration. The JBMT de-registration cost is measured in a similar way. Table 4.1 reports the cost of registration and de-registration operations in JBMT. Both registration and de-registration costs remain constant irrespective of the message size of less than 2 μs. Due to the fact that pages are not pinned and no page translations are maintained in the NIC, the registration and de-registration operations remain constant and inexpensive.

	Registration (usecs)	De-Registration (usecs)
Any Msg. Size	1.32	0.25

Table 4.1: Registration Cost

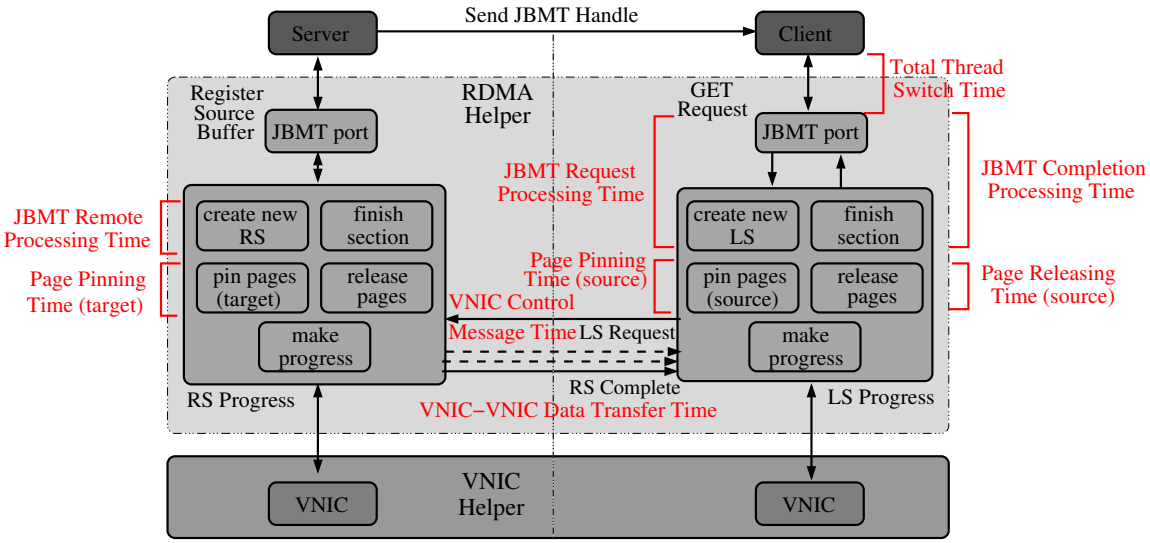


Figure 4.8: Timing Measurements of JBMT *get*

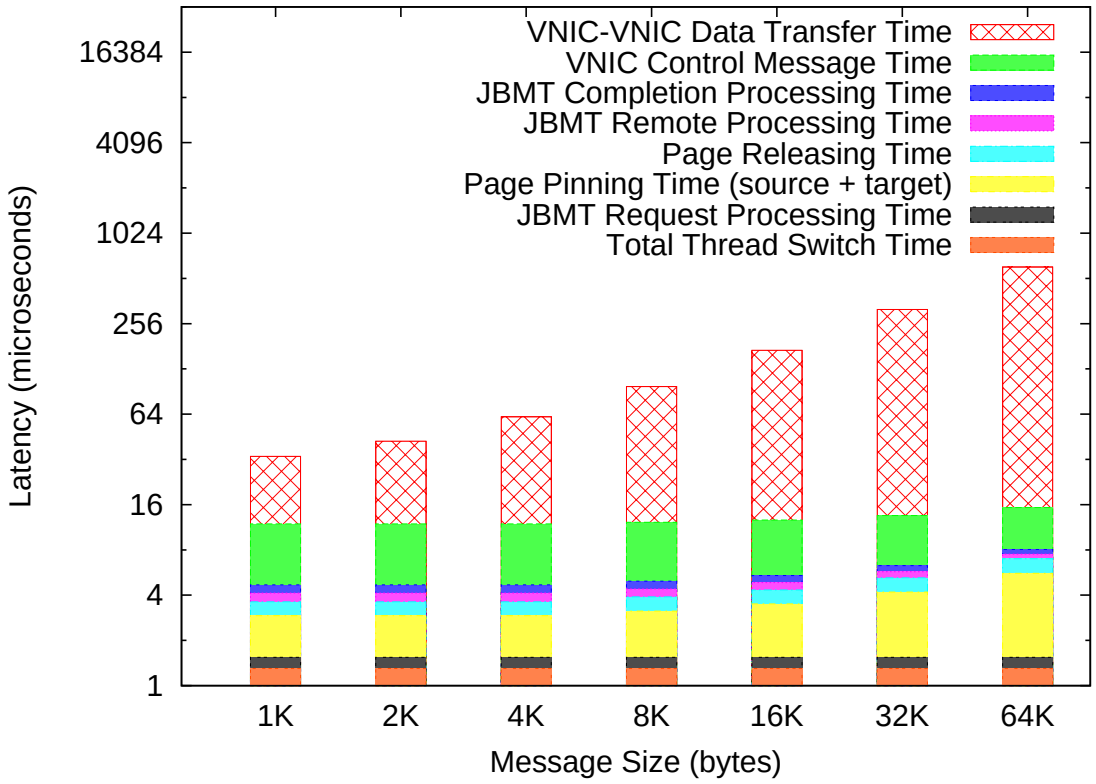


Figure 4.9: Cost Breakdown of JBMT *get*

To further analyze the JBMT *get* operation in detail, we measure the cost of several steps involved in JBMT *get* operation and report its overhead in Figures 4.8 and 4.9. The detailed tasks during a *get* operation (shown in red color) is shown in Figure 4.8. The *Total Thread Switch Time* indicates the time taken to switch from the application that initiates the *get* operation to the JBMT kernel helper thread that listens for such requests. The *JBMT Request Processing Time* refers to the time spent before initiating a local section request and the *JBMT Completion Processing Time* refers to the time spent after receiving all the remote frames and the post processing of a local section. The *Page Pinning Time* and *Page Release Time* refer to the time spent in kernel for locking and releasing the user pages. The *VNIC Control Message Time* refers to the time spent for sending a local section request to the remote side. The *JBMT Remote Processing Time* refers to the time spent by the remote side in initiating the remote section frames. The *VNIC-VNIC Data Transfer Time* indicates the time spent in sending the data using the underlying VNIC layer.

In Figure 4.9, we observe the time spent by each of these operations for various message sizes. We see that the time spent by *Total Thread Switch Time*, *JBMT Request, Remote and Completion Processing Time* is much less when compared to other components in a JBMT *get* operation. However, as mentioned earlier, due to our prototype hardware, the time spent by these operations is still considered quite high and there is room for improving this further. Further, we see that *Page Pinning Time*, *Page Release Time* and *VNIC-VNIC Data Transfer Time* increases with increasing message sizes. As message size increases, the buffers span over multiple pages which automatically increases the page pinning/unpinning costs. Also, since the message sizes are less than the maximum allowed local and remote section sizes

(1 MB), we observe that the overhead of *JBMT Request Processing Time*, *JBMT Completion Processing Time* and *JBMT Remote Processing Time* does not increase with increasing message size. The overhead of these operations is expected to increase as the number of local/remote sections increases. However, this overhead will be significantly less than the overall time taken to perform the *get* operation.

4.5 Summary

While the proposed state sharing mechanism using RDMA implementations do provide efficient one-sided, inter-node remote memory access, they do not preserve all of the benefits of virtual memory. Further, the memory regions are either managed directly by users (requiring user control over system critical resources) or by using complex NIC hardware. In this Chapter, we addressed these limitations by using a software-centric onloading approach to perform tasks such as page pinning, DMA startup/completion, page releasing, flow control and page fault handling. Our architecture provides access to objects in unpinned virtual memory, simplifies the NIC hardware, and ensures robust system behavior by managing memory within trusted kernel code. The salient features and main contributions of the proposed approach are:

1. Our approach exploits the abundant compute cycles in future many-core processors to perform RDMA tasks. Our experimental results are measured on a working prototype and demonstrate a low-overhead for performing needed operations in the critical path.

2. Unlike many existing networks, our design preserves the key capabilities that are provided by virtual memory. The design allows access to more virtual memory

than is physically present and supports access protection for client applications. This is especially important in complex multiple program environments associated with commercial computing.

3. Our design utilizes a kernel helper thread to manage memory pages leading to a robust and well controlled environment for managing the virtual memory subsystem. This compares to existing approaches which require that users manage memory pages or that pages are managed through complex NIC hardware.

4. Our design simplifies the NIC hardware by onloading RDMA tasks such as page pinning, DMA startup/completion, page unpinning, handling page faults and flow control to the kernel helper thread. Further, the presence of a kernel helper thread avoids replication of page table entries and provides faster access to page tables and helps in easier maintenance. In addition, there are no changes/updates required in NIC hardware as changes are made to the internal virtual memory subsystems.

Further, due to the presence of a kernel helper thread in our design, application-specific tasks such as distributed queue insertions/modifications/deletions, locking operations and several other memory-based operations can be onloaded, thus providing opportunities to revise the design and implementation of many subsystems in multi-program datacenter environments.

CHAPTER 5

MULTICORE-AWARE, DMA-ACCELERATED STATE SHARING

In this section, we utilize the advanced features of multicore systems and I/OAT for designing efficient state sharing substrate. Figure 5.1 shows the various components of the multicore-aware, DMA-accelerated state sharing substrate. Broadly, in the figure, we focus on the colored boxes for designing efficient multicore-aware, DMA-accelerated state sharing components and understanding its benefits with datacenter applications. The dark colored boxes show the features and technologies that we utilize and the light colored boxes show the proposed components and datacenter system software evaluated.

5.1 Background and Related Work

As mentioned in the previous section, several datacenter applications and services use memory copies extensively. The memory copy operation can be accelerated with the help of a copy engine. However, this requires an on-chip or an off-chip DMA engine that can perform the copy.

As mentioned in Chapter 3, ADCE can be utilized to perform accelerated copy operations in several datacenter environments. Figure 5.2(a) shows the latency of

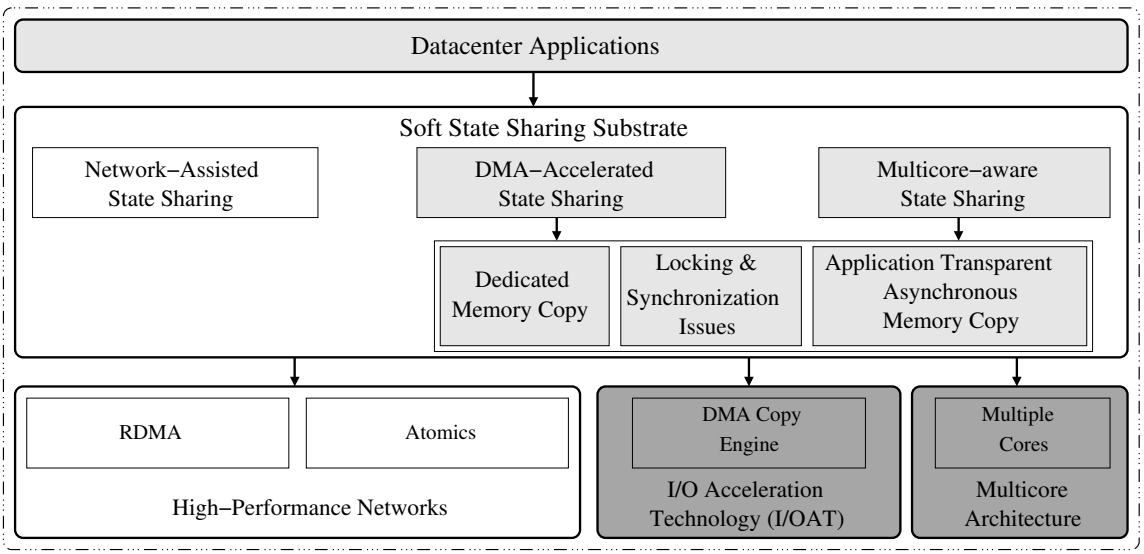
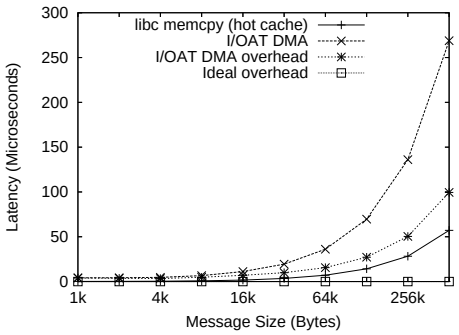


Figure 5.1: Multicore-aware, DMA-Accelerated State Sharing Components

memory copy operation using the I/OAT's DMA copy engine and the associated overheads for different message sizes as mentioned in [93]. Since the copy engine is known to give better performance for large memory copies [93], we focus only on small and medium message sizes. Also, we report the performance of traditional *libc memcpy* when the application buffers are resident in the cache (referred as *libc memcpy (hot-cache)* in the figure). As shown in Figure 5.2(a), we observe that the traditional *libc memcpy* outperforms the I/OAT DMA engine's performance if the application buffers are in the cache. Further, we observe that the DMA startup overhead associated with the copy engine is much higher than the memory copy time (*libc memcpy hot-cache*), thus removing the benefits of asynchronous memory copy provided by these copy engines.

Researchers in the past have looked at different ways of providing memory copy operations as shown in Figure 5.2(b). For single-core systems with no hardware copy



(a) Memory Copy Latency

	No I/OAT	I/OAT
Single Core	SCNI Single–Core with No I/OAT	SCI Single–Core with I/OAT
Multiple Cores	MCNI Multi–Core with No I/OAT	MCI Multi–Core with I/OAT

(b) Different Mechanisms for Memory Copies

Figure 5.2: Motivation for Using Asynchronous Memory Copy Operations

engine support, traditional *libc memcpy* is used for memory copies. We refer to this scheme as SCNI (Single-Core with No I/OAT). However, if the system has an I/OAT support [68], applications can *offload* the memory copy to the hardware copy engine. We refer to this scheme as SCI (Single-Core with I/OAT). As multicore systems are emerging, it opens up new ways to design and implement memory copy operations. Currently, there is no study that has explored the impact of multicore systems in designing efficient memory copy operations. We take on this challenge and introduce two new schemes as shown in white boxes. In the first scheme, MCI (Multi-Core with I/OAT), we *offload* the memory copy operation to the copy engine and *onload* the startup overheads to a dedicated core. For systems without any hardware copy engine support, we propose a second scheme, MCNI (Multi-Core with No I/OAT) that *onloads* the memory copy operation to a dedicate core.

Though the SCI scheme offers several benefits such as performance improvement, cache pollution avoidance, overlap capability, it has the following overheads.

Copy Engine Overheads: As mentioned earlier, in order to perform memory copy operation using a copy engine, we need to post a descriptor to a channel specifying the source and destination buffer and the size of the data transfer. Due to the presence of multiple channels in the copy engine, the cost of posting the descriptor increases since this operation cannot be overlapped. After the copy operation is initiated, we also need to check for the completion of memory copy operation across all the channels. Though the hardware copy engine provides a mechanism to avoid this cost by sending an interrupt after the completion, this may not be suitable for latency-sensitive applications.

Page Locking Overheads: Further, due to the fact that the hardware copy engine can understand only physical addresses, it is mandatory that the application buffers are locked/pinned while the copy operation is in progress. Similarly, after the completion of copy operation, the locked application buffers can be released. The locking of application buffers is done in kernel space with the help of *get_user_pages* function which is a costly kernel function. Especially if the application buffer is not in memory, this overhead can be quite huge since the page has to be brought from the disk. Later, in the experimental section, we show that this locking overhead occupies a significant fraction of the total overhead limiting the overlap capability of the copy engine.

Context Switch Overheads: Due to the fact that the copy engine is accessible only in kernel space, a context switch occurs for every copy engine related operation performed by the user application. This cost is especially huge if multiple applications

try to access the copy engine at the same time while the copy operation is still in progress resulting in several context switch penalties.

Synchronization Overheads: As mentioned earlier, several applications can access the hardware copy engine simultaneously and hence the copy engine resources need to be locked for protection. Though spin locks are used in the SCI scheme, several user applications compete for locks to gain access to the copy engine.

While the SCI scheme helps user applications to *offload* memory copy operations, several critical operations still remain in the critical path. In the following section, we propose a novel scheme to alleviate these overheads to achieve maximum overlap between memory copy operation and computation.

Related Work: Researchers have proposed several solutions for asynchronous memory operations in the past. Zhao et al [103] talk about hardware support for handling bulk data movement. Calhoun's thesis [39] proposes the need for dedicated memory controller copy engine and centralized handling of memory operations to improve performance. However, many of these solutions are simulation-based. Ciaccio [49] proposed the use of self-connected network devices for offloading memory copies. Though this approach can provide an asynchronous memory copy feature, it has a lot of performance-related issues. I/OAT [1] offers an asynchronous copy engine which improves the copy performance with very little startup costs. In this paper, we use this hardware for supporting asynchronous memory operations.

Regarding intra-node communication, Buntinas et. al. [38] and Chai et. al. [43] have discussed shared memory based approaches and optimizations. Jin et. al. have proposed a kernel assisted design in [63], which is similar to the MCNI scheme discussed in this paper. However, the scheme proposed in this paper is more general

in that it can be applied not only to datacenter environments but also to high-performance computing environments. Besides, our scheme dedicates the copy operation to another core thus providing complete overlap of copy operation with computation.

5.2 Design and Implementation Issues

In this section, we propose two design alternatives [92] to address the limitations mentioned above.

Multi-Core with No I/OAT (MCNI) Scheme: In order to provide asynchronous memory copy operations for systems without the copy engine support, we propose a MCNI scheme (Multi-Core systems with No I/OAT) that *onloads* the memory copy operation to another processor or a core in the system. This scheme is similar to the MCI scheme. In this scheme, as shown in Figure 5.3, we dedicate a kernel thread to handle all memory copy operations, thus relieving the main application thread to perform computation.

5.2.1 Dedicated Memory Copy using Multicore Systems

The MCNI scheme takes help from the operating system kernel to perform memory copy operations. To perform direct memory copy operations, the kernel thread should have access to the physical pages pointed by the application's virtual address to perform copy operations to/from one process or to another process. This is done through memory mapping mechanism that maps a part of other processes address space into the address space of the kernel. After the memory mapping process, the kernel thread can directly access the mapped area.

For memory mapping, we use the *vmap* kernel helper function provided by the operating system. The *vmap* function essentially maps a set of physical pages into contiguous virtual memory region. In order to get the physical pages of the application buffer, we use the *get_user_pages* which not only locks the application buffer but also returns the set of mapped physical pages. After mapping this memory region, the kernel thread can access this memory region as its own and perform the memory copy operation. We use the *vunmap* kernel helper function to remove the mapping of the physical pages to release any unnecessary memory regions.

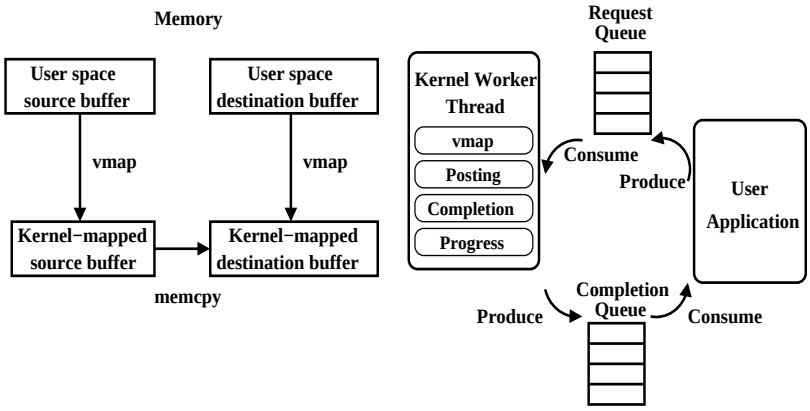


Figure 5.3: Asynchronous Memory Copy Operations using MCNI

5.2.2 Dedicated Memory Copy Using Multicore Systems and DMA Engine

Multi-Core with I/OAT (MCI) Scheme: In order to alleviate the overheads mentioned above, we propose a scheme that takes advantage of the copy engine and multicore systems. Specifically, we *offload* the copy operation to the hardware copy

engine and *onload* the tasks that fall in the critical path to another core or a processor so that applications can exploit complete overlap of memory operation with computation.

Figure 5.4 shows the various components of the proposed scheme. Since the copy engine is accessible only in the kernel space, we dedicate a kernel thread to handle all copy engine related tasks and allow user applications to communicate with the kernel thread to perform the copy operation. The kernel thread also maintains a list of incomplete requests and attempts to make progress for these initiated requests. Apart from servicing multiple user applications, the dedicated kernel thread also handles tasks such as locking the application buffers, posting the descriptors for each user request on appropriate channels, checking for device completions, releasing the locked buffers after completion events. Since the critical tasks are onloaded to this kernel thread, the user application is free to execute other computation or even execute other memory copy operations while the copy operation is still in progress thus allowing almost total overlap of memory copy operation and computation.

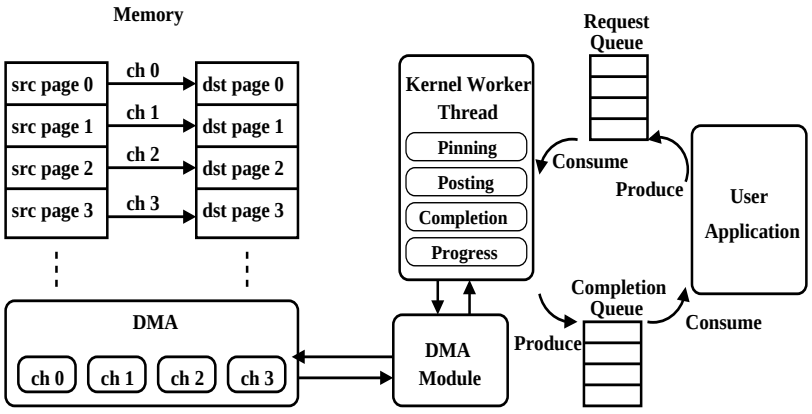


Figure 5.4: Asynchronous Memory Copy Operations using MCI

5.2.3 Avoiding Context Switch Overhead

In order to avoid the context switch overhead between the application process and the kernel thread, we use an approach that has already been proposed by [83]. We memory map a region from a user space to kernel space using a *vmap* kernel helper function so that both the application and the kernel thread can access this common memory region at the same time. We divide this memory region as a set of request and response queues. We use the request and response queues to communicate between the application and the kernel thread. The request queue is used to submit memory copy requests by the application. The dedicated kernel thread constantly looks at the request queue to process new copy requests. Similarly, the response queue is used to notify the completion of copy operations by the kernel thread. The applications constantly look at the response queue for completion notifications.

5.2.4 Handling Locking and Synchronization Issues

As mentioned earlier, the dedicated kernel thread can also help in handling pinning and unpinning memory regions for performing efficient asynchronous memory copy operations. Further, in the SCI scheme, since the kernel module exposes a set of interfaces for applications, several kernel instances can be spawned if multiple applications need to access the copy engine. This increases the requirement of locking the shared resources and careful management for supporting concurrency. However, in the MCI scheme, since the dedicated kernel thread handles all tasks for multiple applications, it avoids the need for locking the resources and becomes easier for managing the shared resources.

5.2.5 Application-Transparent Asynchronous Memory Copy

The main idea of application-transparent asynchronism is to avoid blocking the application while the memory copy operation is still in progress. With the asynchronous memory copy interfaces, the application can explicitly initiate the copy operation and wait for its completion using another function. However, several applications are written with the blocking routine (*memcpy*), which assumes that the data is copied once the function finishes. Further, the semantics of the *memcpy* operation assumes that the buffer is free to be used after the completion of *memcpy* operation. To transparently provide asynchronous capabilities for such operations, two goals need to be met: (i) the interface should not change; the application can still use the blocking *memcpy* routine and (ii) the application can assume that the blocking semantics, i.e., once the control returns to the application, it can read or write the buffer. In our approach, we memory-protect the user buffer (thus disallow the application from accessing it) and perform the copy operations. After the copy operation finishes, we release the memory protection so that the applications can access both the source and destination buffers. Since in a *memcpy* operation the source does not get modified, we allow read accesses to the source buffer. However, the destination address gets modified during a copy operation and hence we do not allow accesses to this memory region during the copy operation. We further optimize the performance by checking for copy completion on successive memory copy operation calls and release the protection. If the application does not modify the destination buffer for sufficiently long time, then the application will realize only the page protection time, which is much lesser compared to the copy operation for large message transfers.

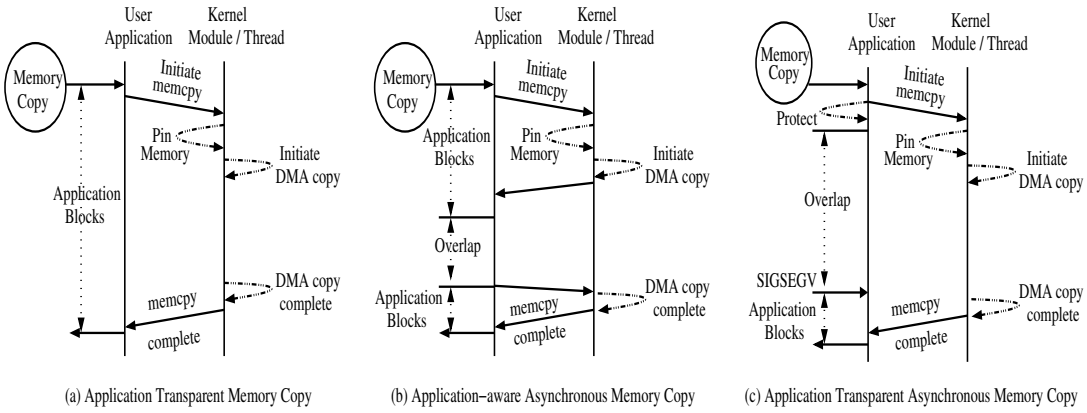


Figure 5.5: Different Mechanisms for Asynchronous Memory Copy Operations

Figure 5.5 illustrates the designs of asynchronous memory copy operation. As shown in Figure 5.5(a), though the memory copy is performed by the DMA engine or by a dedicated core, the application blocks for every memory copy to finish before performing any other operation. Figure 5.5(b) shows the impact of an application-aware asynchronous memory copy operation. In this case, the application has to be modified to take advantage of asynchronous feature. Figure 5.5(c) shows the design of the application-transparent asynchronous memory copy operation approach. This approach can be adopted along with the previous schemes. In this approach, as shown in the Figure 5.5(c), we memory protect buffers before initiating the data transfer and return the control to the application. If the application attempts to access the destination buffer or modify the source buffer, a page fault is generated due to page protection. This results in a SIGSEGV signal for the application which is handled by our helper module. In this case, we block for all pending memory copy operations to complete and release the protection appropriately. This can be further optimized if we divide a large memory copy operation into smaller chunks and perform the page

protection and release for these smaller chunks. This can potentially allow better overlap of memory copy operation with computation. However, if the application touches the destination buffer or modifies the source buffer very frequently (thus generating the *page fault* very frequently), it may lead to very less overlap of memory copy operation with computation.

Potential Benefits of SCNI, SCI, MCI and MCNI Schemes: While all four schemes address different design goals, in this section, we try to address the benefits and issues with each of these schemes in providing asynchronous memory copy operations to applications. The SCNI scheme simply uses *libc memcpy* which does not provide any overlap to user applications as shown in Figure 5.6(a). However, the SCI scheme shown in Figure 5.6(b), allows applications to take advantage of overlap capability after initiating the memory copy operation. In this scheme, the user application waits for the kernel module to lock the application buffers and post the descriptors across the DMA channels before performing the computation, thus limiting the overlap capability. On the other hand, in the MCI scheme as shown in Figure 5.6(c), we see that the user application can proceed to perform computation immediately after submitting the request to the request queue. This transaction also avoids context switch overheads. This scheme provides almost complete overlap of memory copy operation with computation. We expect similar trends for the MCNI scheme as shown in Figure 5.6(d), in overlapping memory copy operation with computation. However, due to huge mapping costs associated with this scheme, the memory copy operation is very likely to take a longer time to finish, thus impacting the performance of large memory copies.

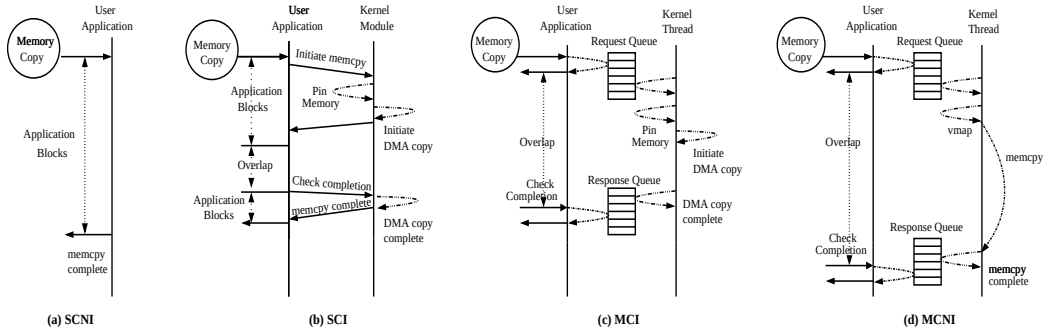


Figure 5.6: Overlap Capability: (a) SCNI, (b) SCI, (c) MCI and (d) MCNI

5.3 Experimental Results

We ran our experiments on a dual dual-core Intel 3.46 GHz processors with 2 MB L2 cache system using SuperMicro X7DB8+ motherboards which include 64-bit 133 MHz PCI-X interfaces. We use the Linux RedHat AS 4 operating system and the kernel version 2.6.20 for all our experiments.

Our experiments are organized as follows. First, we analyze our schemes in terms of performance, overlap capability and the associated overheads. Next, we evaluate the impact of these schemes in applications.

5.3.1 Performance with and without Page Caching

In this section, we evaluate the schemes in terms of latency and bandwidth performance, overlap efficiency and its associated overheads.

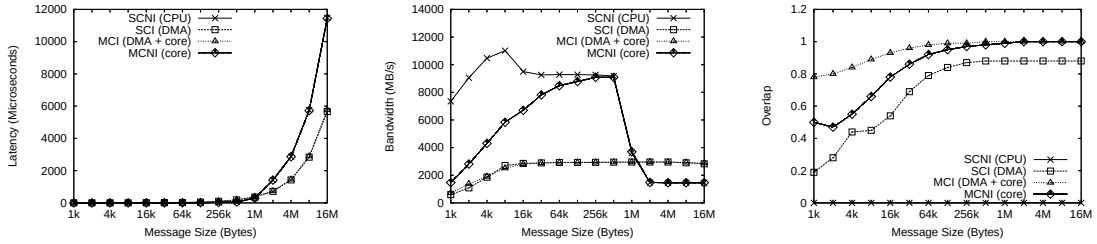
Basic Performance with Page Caching: Figure 5.7 shows the basic performance of memory copy operation using the page caching mechanism. Figure 5.7(a) shows the latency of all four schemes. For the SCNI scheme, we perform several *memcpy* operation using the *libc* library and average it over several iterations. For SCI, MCI and

MCNI schemes, we initiate the memory copy operation and wait for the completion notification before initiating the next copy operation. As shown in Figure 5.7(a), we see that the latency of both SCNI and MCNI schemes for message sizes greater than 2 MB is significantly worse compared to the performance of SCI and MCI schemes. Since the cache size is only 2 MB, both SCNI and MCNI schemes perform the copy operation in memory using the CPU which is limited by small register-size copy operations. However, for SCI and MCI schemes, since the copy operation is performed by the DMA channels directly in memory, it is not limited by the register size. Hence, we see a performance improvement of up to a factor of two for SCI and MCI schemes in comparison with SCNI and MCNI schemes. For message sizes less than 1 MB, since the buffers can fit in cache, the performance of SCNI and MCNI schemes are significantly better than SCI and MCI schemes.

The bandwidth performance of memory copy operation is shown in Figure 5.7(b). In this experiment, we initiate a window of copy operations and wait for these copy operations to finish. We repeat this experiment for several iterations and report the bandwidth. As shown in Figure 5.7(b), we see that the bandwidth performance of SCNI and MCNI schemes for message sizes less than 1 MB is significantly better than the bandwidth performance of SCI and MCI schemes due to caching effects. The peak bandwidth for SCNI and MCNI schemes achieved are 11014 MB/s and 9087 MB/s, respectively. However, for message sizes greater than 2 MB, we see that the bandwidth of SCNI and MCNI schemes drops to 1461 MB/s and 1463 MB/s since the buffers are accessed in memory. On the other hand, SCI and MCI schemes report a peak bandwidth of up to 2958 MB/s and 2954 MB/s, respectively.

To measure the overlap efficiency, we perform the overlap benchmark as mentioned in [93]. First, the benchmark estimates the copy latency (T_{copy}) by performing a blocking version of memory copy operations. Next, the benchmark initiates the asynchronous memory copy followed by a certain amount of computation ($T_{compute} > T_{copy}$) which takes at least the blocking copy latency and finally waits for the copy completion. The total time is recorded as T_{total}. If the memory copy is totally overlapped by computation, we should have $T_{total} = T_{compute}$. If the memory copy is not overlapped, we should have $T_{total} = T_{copy} + T_{compute}$. The actual measured value will be in between, and we define overlap as ($T_{copy} + T_{compute}$ - T_{total}) / T_{copy}. Based on the above definition, the value of *overlap* will be between 0 (non-overlap) and 1 (totally overlapped). A value close to 1 indicates a higher overlap efficiency. Figure 5.7(c) shows the overlap efficiency of all four schemes in performing memory copy operations and computations. For the SCNI scheme, since we only have a blocking version of memory copy (*libc memcpy*), we see that the overlap efficiency is zero. In the SCI scheme, since the copy operation is offloaded, we observe that it can achieve up to 0.88 (88%) overlap efficiency. However, for small message sizes, we see that the overlap efficiency is quite low. On the other hand, we observe that the MCI scheme can achieve up to 1.00 (100%) overlap efficiency for large messages and up to 0.78 (78%) overlap efficiency even for small messages. We also observe that the MCNI scheme achieves up to 1.00 (100%) overlap efficiency for large messages and up to 0.5 (50%) overlap efficiency for small messages.

Performance without Page Caching: In this section, we measure the performance of our schemes without the page caching mechanism.

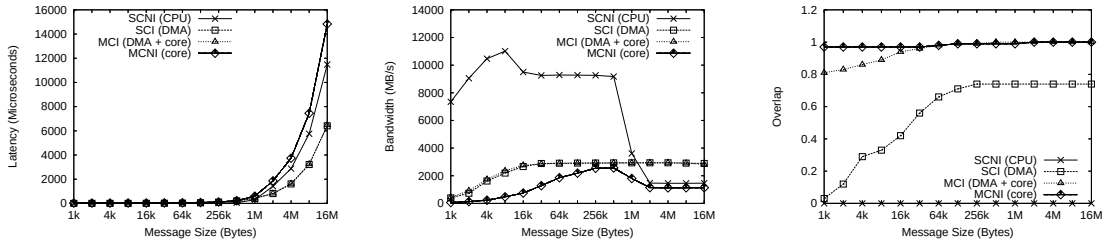


(a) Latency (b) Bandwidth (c) Overlap

Figure 5.7: Micro-Benchmark Performance with Page Caching

Figure 5.8(a) shows the latency of all four schemes without page caching. For the MCNI scheme, we observe that the latency is significantly worse reaching up to 14829 μs for 16 MB message size. Further, we see that SCI and MCI schemes report a latency of 6414 μs and 6468 μs, respectively. As mentioned earlier, since the application buffers are not cached, every memory copy operation using SCI, MCI and MCNI schemes incur a page locking cost, thus increasing the latency. However, the SCNI scheme does not show any degradation since the scheme does not depend on the underlying page caching mechanism. The bandwidth performance without page caching mechanism is shown in the Figure 5.8(b). Since the locking costs can be pipelined with several memory copy operations, we do not observe any degradation in bandwidth for SCI and MCI schemes. However, for the MCNI scheme, due to huge mapping costs, we see a drop in bandwidth. Figure 5.8(c) shows the overlap efficiency of all four schemes without page caching mechanism. For the SCI scheme, since the startup overheads fall in the critical path, we observe that it can achieve

only 0.74 (74%) overlap efficiency for large message transfers. However, we see that both MCNI and MCI schemes show up to 1.00 (100%) overlap efficiency.



(a) Latency (b) Bandwidth (c) Overlap

Figure 5.8: Micro-Benchmark Performance without Page Caching

5.3.2 Split-up Overhead of Different Memory Copy Approaches

To understand the low overlap efficiency observed for small messages in the previous section, we measure the split-up overhead of the three schemes, namely SCI, MCI and MCNI schemes. Figure 5.9 shows the split-up overhead of using memory copy operations with the SCI, MCI and MCNI schemes. For small message sizes, we see that the pinning costs, startup overheads and completion notifications consume considerable amount of time reducing the overlap efficiency for the SCI scheme. Even for large message sizes, we observe that the pinning costs and DMA startup overheads occupy close to 18% and 7%, respectively. However, for MCI and MCNI schemes, we observe that the only overhead is posting the request and checking for completions through memory transactions in the response queue, thus resulting in almost 100% overlap efficiency.

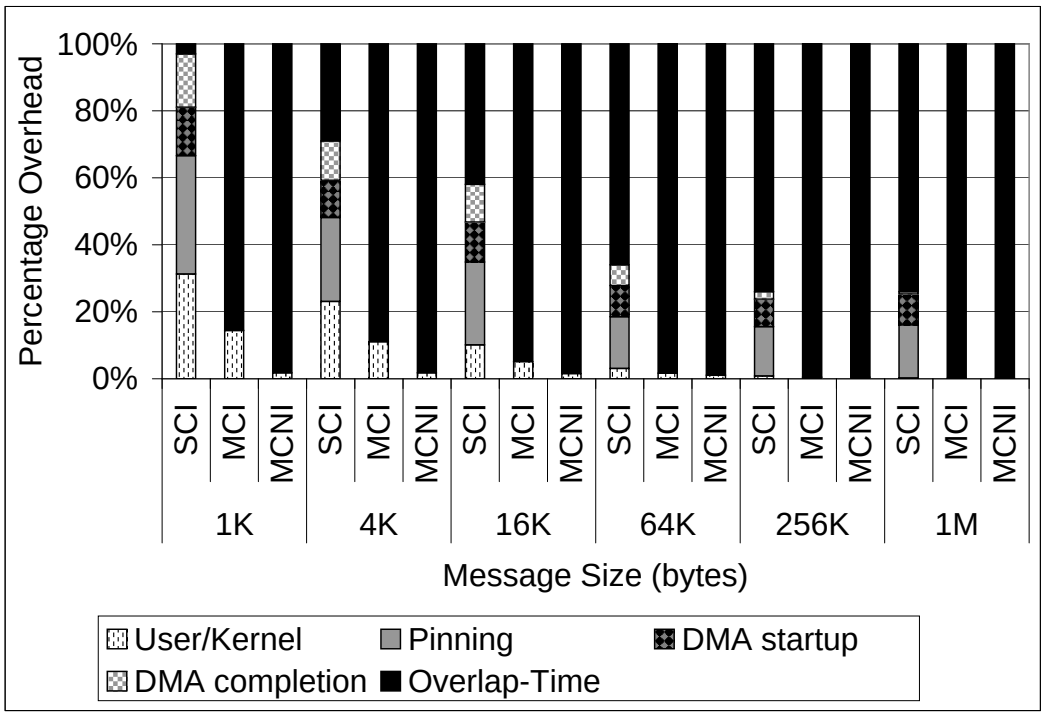


Figure 5.9: Overheads of SCI, MCI and MCNI Schemes

5.3.3 Evaluations with SPEC and Datacenters

In this section, we evaluate the performance of the proposed schemes with *gzip* SPEC CPU2000 benchmark and simulated datacenter services. SPEC CPU2000 benchmark [22] is a set of benchmarks designed to characterize and evaluate the performance of overall system performance such as CPU, memory, etc. In this work, we focus on one such benchmark, *gzip*, which measures the CPU and memory performance. In order to force SPEC CPU2000 benchmarks to use our schemes, we preloaded a library that intercepts all *memcpy* operations. In all our experiments, we forced the benchmarks to use the different schemes if the memory copy size is greater than 64 KB.

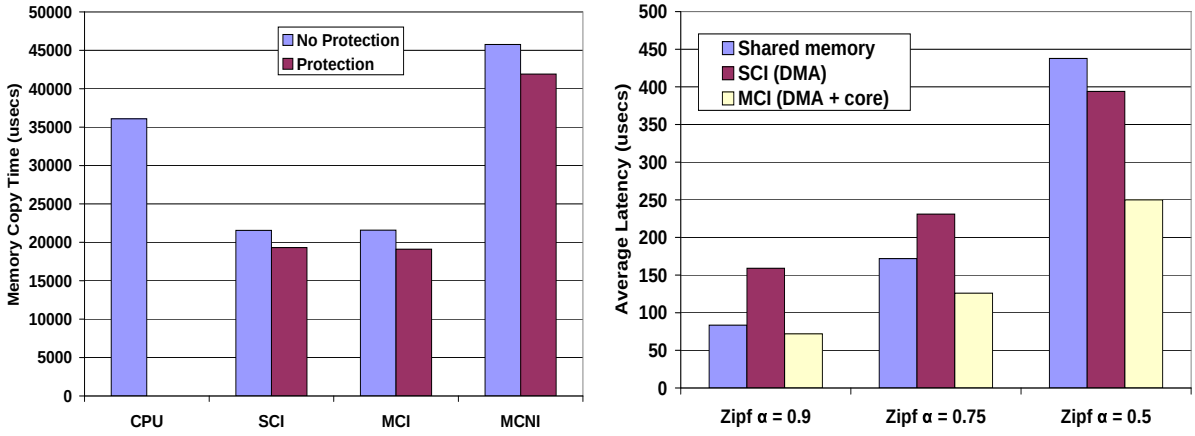
SPEC benchmarks focus on CPU and memory-intensive operations (i.e., memory reads, computations, memory writes) and hence we did not observe any significant improvement in the overall execution time. However, we report the time spent in memory copy operations (greater than 64 KB message size) using the four different schemes during application execution. With the protection approach, we include the time spent in initiating the memory copy and protecting the source and destination buffers and also the time spent in waiting for the memory copy operation to finish when either the source or the destination buffers are touched (i.e. the time spent after receiving a SIGSEGV). Table 5.1 shows the total cost of protecting the source and destination buffers before and after the copy operation and we observe that the total protection cost (this includes four *mprotect* calls) is quite less compared to the total time for the memory copy operation. It is to be noted that this cost is the worst case estimate and it can be further optimized for consecutive memory copy operations involving the same source or the destination buffers and if the buffers are not touched in between these memory copy operations.

Msg. Size	64 KB	256 KB	1 MB	4 MB	16 MB
Cost (usecs)	4.3	7.1	18.6	63.6	227.9

Table 5.1: Memory Protection Overhead

Figure 5.10(a) shows the performance of *gzip* benchmark with all four schemes. As shown in the figure, we observe that both SCI and MCI schemes improve the performance of memory copy time (i.e., memory copies greater than 64 KB) by up to 35% as compared to the SCNI scheme. We profiled the message distribution of *gzip*

benchmark for all message sizes greater than 64 KB. We found that more than 50% of the memory copies greater than 64 KB fall between 1 MB and 2 MB. Due to this reason, we observe that SCI and MCI schemes improve the performance of memory copies. For the MCNI scheme, we observe that the performance does not improve due to large mapping cost overheads. As mentioned in Section 5.2, application-transparent memory copies can further improve the performance if the source and destination buffers are not accessed immediately after a memory copy operation. As shown in Figure 5.10(a), we observe that the performance of *gzip* consistently improves by 10% as compared to the performance of blocking memory copy operations for SCI, MCI and MCNI schemes. This result is quite promising for several applications that use memory copies similar to the *gzip* benchmark.



(a) Memory Copy Time (> 64 KB) in *gzip* SPEC Benchmark

(b) Emulated Data Sharing in Datacenters

Figure 5.10: Application Performance

Next, we evaluate the performance of memory copy operations in a datacenter environment [85]. For efficiently transferring the data from a remote site to the local node in a datacenter environment, a distributed shared state has been proposed in the literature [97]. However, there is no efficient support for transferring the data between a datacenter service and the application threads within a node. We use the asynchronous memory copy operations for supporting data sharing within a node. To emulate the multiple threads copying the shared data scenario, we create a single server and three application threads in a single node. The server initiates a data copy operation to all three application servers and waits for the completion operation. The application threads also wait for the completion of the copy operation. For data copy operation, we use a Zipf distribution with varying α value which is common in several datacenter environments. According to Zipf law, the relative probability of a request for the ith most popular document is proportional to $1/i^{\alpha}$, where α determines the randomness of file accesses. Higher the α value, higher will be the temporal locality of the document accessed. We use file sizes ranging from 500 bytes to 8 MB in the Zipf trace. We emulate the datacenter environment by firing copy requests according the Zipf pattern and measure the average latency after the completion of all copy operations. We report the performance of SCI, MCI and the traditional shared memory (SCNI) schemes for copying the data as shown in Figure 5.10(b). We observe that the performance of the MCI scheme is significantly better for all Zipf traces. The MCI scheme shows up to 37% performance improvement as compared to the SCI scheme for an α value of 0.5. This is mainly due to avoiding context switch overheads and scheduling the memory copy operations without any delay. The shared memory (SCNI) scheme is better for larger α values compared to the SCI scheme,

since majority of the data is transferred through the cache. However, as α value decreases, we see that the performance of the shared memory (SCNI) scheme gets worse. Due to a lot of overheads associated with the SCI scheme, the benefits of the SCI scheme show up only when the application uses large memory copies (smaller α values). The performance of the MCNI scheme with other application threads (all four cores are completely utilized) degraded significantly and hence we did not include this result.

5.4 Summary

In this Chapter, we proposed several approaches to provide complete overlap of memory copy operation with computation by dedicating the critical tasks to a core in a multicore system. In the first scheme, MCI (Multi-Core with I/OAT), we *offloaded* the memory copy operation to the copy engine and *onloaded* the startup overheads associated with the copy engine to a dedicated core. For systems without any hardware copy engine support, we proposed a second scheme, MCNI (Multi-Core with No I/OAT) that *onloaded* the memory copy operation to a dedicate core. We further proposed a mechanism for an application-transparent asynchronous memory copy operation using memory protection. We analyzed our schemes based on overlap efficiency, performance and associated overheads using several micro-benchmarks and applications. Our microbenchmark results showed that memory copy operations using MCI and MCNI schemes can be significantly overlapped (up to 100%) with computation. Evaluations with datacenters using MCI show up to 37% improvement compared to the traditional implementation. Our evaluations with *gzip* SPEC

benchmark using application-transparent asynchronous memory copy show a lot of potential to use such mechanisms in several application domains.

CHAPTER 6

MULTICORE-AWARE, NETWORK-ASSISTED STATE SHARING

In this Chapter, we propose design optimizations in state sharing substrate using the features of both high-speed networks and multicore systems. Figure 6.1 shows the various components of multicore-aware, network-assisted state sharing substrate. Broadly, in the figure, we focus on the colored boxes for designing efficient multicore-aware, network-assisted state sharing components and understanding its benefits with datacenter applications and services. The dark colored boxes show the features and technologies that we utilize and the light colored boxes show the proposed components and datacenter system software evaluated.

6.1 Background and Related Work

As mentioned in the previous sections, we have proposed and developed state sharing using several emerging technologies. Recently, multi-core systems have been gaining popularity due to their low-cost per processing element and are getting deployed in several distributed environments. In addition, many of these systems enable simultaneous multi-threading (SMT), also known as hyper-threading, to support large number of concurrent thread executions in the system. While the main idea of these

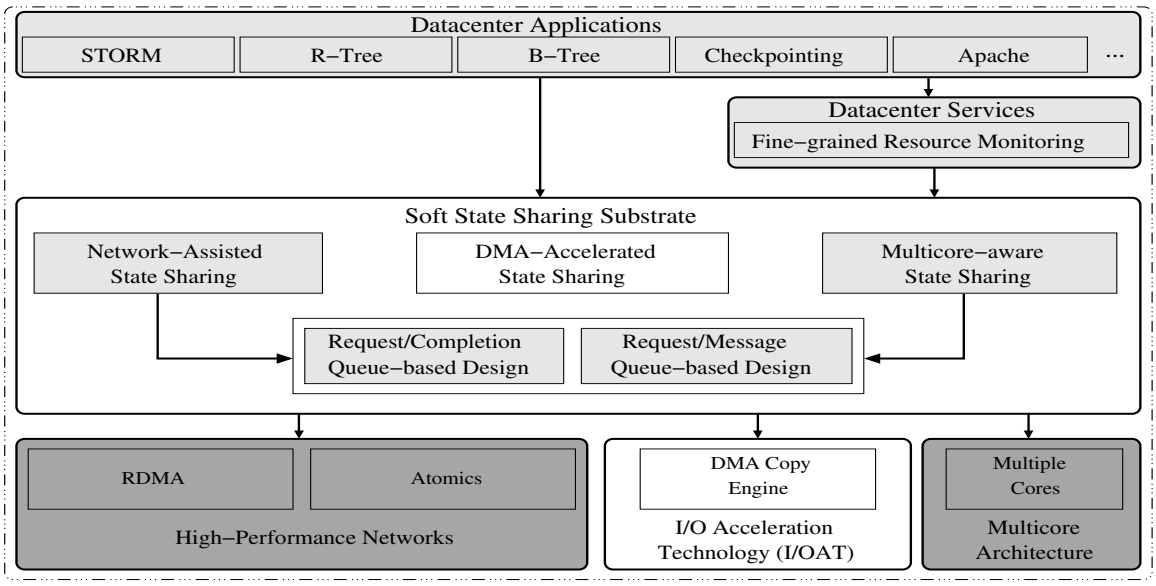


Figure 6.1: Multicore-aware, Network-Assisted State Sharing Framework

systems is to provide multiple processing elements to function in parallel, it also opens up new ways to design and optimize existing middle-ware such as the state sharing substrate. Further, as the number of cores and threads multiply, one or more of these cores can also be dedicated to perform specialized datacenter services. In this context, we propose several design optimizations for the state sharing substrate in multi-core systems and high-speed networks such as the combination of shared memory and message queues for inter-process communication, dedicated thread for communication progress and for onloading other state sharing operations such as *get* and *put*. In the following sections, we refer to our proposed substrate as a distributed data sharing substrate (DDSS).

Related Work: Modern processors are seeing a steep increase in the number of cores available in the system [51]. As the number of cores increases, the choice to

dedicate one or more cores to perform specialized services will become more common. In this paper, we demonstrated the benefits with a resource monitoring service using DDSS. There has been several distributed data sharing models proposed in the past for a variety of environments such as InterWeave [89], Khazana [42], InterAct [79] and Sinfonia [28]. Many of these models are implemented based on the traditional two-sided communication model targeting the WAN environment addressing issues such as heterogeneity, endianness and several others. Such two-sided communication protocols have been shown to have significant overheads in a cluster-based data-center environment under loaded conditions [97]. The most important feature that distinguishes DDSS from these models is the ability to take advantage of several features of multi-core systems and high-performance networks for both LAN/WAN environments, its applicability and portability with several high-speed networks and its minimal overhead.

6.2 Proposed Design Optimizations

In this section, we first present our existing Message Queue-based DDSS (MQ-DDSS) state sharing design as discussed in Chapter 2. Next, we present two design optimizations [96, 95] for multi-core systems, namely, (i) Request/Message Queue-based DDSS (RMQ-DDSS) and (ii) Request/Completion Queue-based DDSS (RCQ-DDSS).

6.2.1 Message Queue-based DDSS (MQ-DDSS)

Existing DDSS service utilizes a kernel-based message queue to store and forward user requests from one application thread to another and notification-based mechanism to respond to network events. For example, in the case of a *get()* operation in

DDSS, the application thread performs an *ipc_send* operation (shown as Step 1 in Figure 6.2(a)) to submit a request. During this operation, the kernel (after a context-switch) copies the user request buffer to a FIFO (First-In-First-Out) message queue. If the corresponding *ipc_recv* (shown as Step 2 in the figure) operation is posted by the service thread, then the kernel copies the request buffer to the service thread's user buffer and sends an interrupt to the service thread to handle the request. Next, the service thread determines the remote node that holds the data and issues an RDMA read operation (Step 4). After the RDMA read operation completes, the network sends an interrupt to the kernel (Step 6 shown in the figure) to signal the completion and submits a completion event in the completion queue (Step 7). The kernel looks at the interrupt service routine and raises another interrupt (Step 8) for the user process in order to signal the completion of the network event. The service thread processes the network completion event (Step 9) and accordingly informs the application thread regarding the status of the operation (Steps 10 and 11). Though this approach does not consume significant CPU, it suffers from the fact that the operating system gets involved in processing the request and reply messages and in handling network events. As a result, this approach leads to several context-switches, interrupts (shown as the dashed line in Figure 6.2(a)) and thus may lead to degradation in performance.

6.2.2 Request and Message Queue-based DDSS (RMQ-DDSS)

One approach to optimize the performance of DDSS is to use a shared memory region as a circular array of buffers (Steps 1 and 2 shown in the Figure 6.2(b)) for inter-process communication. In this approach, the reply messages still follow the

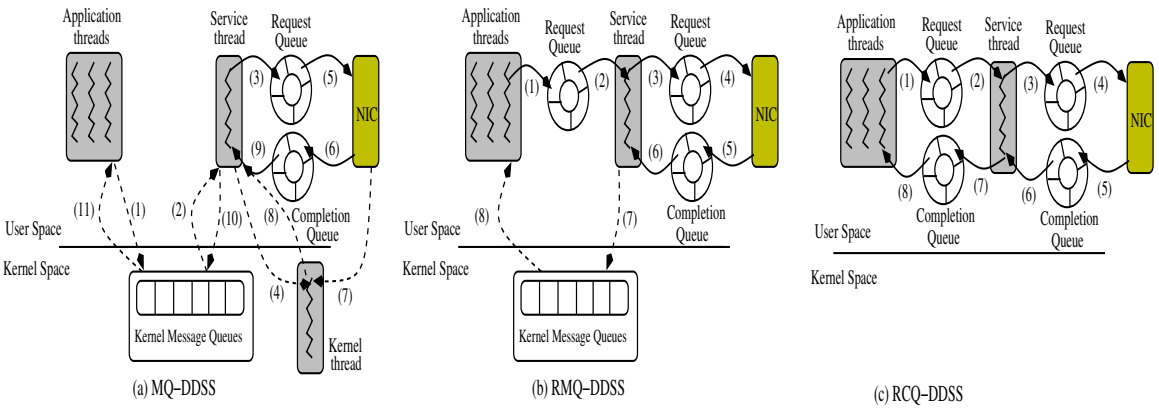


Figure 6.2: Design Optimizations in State Sharing

path of using the kernel-based message queues (Step 7 and 8). Networks such as InfiniBand and iWARP-capable 10-Gigabit Ethernet also allow applications to check for completions through memory mapped completion queues. The DDSS service thread periodically *polls* on this completion queue, thereby avoiding the kernel involvement for processing the network events. However, the service thread cannot *poll* too frequently as it may occupy the entire CPU. We propose an approach through which critical operations such as *get()* and *put()* use a polling-based approach while other operations such as *allocate()* and *release()* use a notification-based mechanism and wait for network events. The performance of *allocate()* and *release()* operations are not most critical since applications typically read and write information frequently. This optimization reduces the kernel involvement significantly.

6.2.3 Request and Completion Queue-based DDSS (RCQ-DDSS)

Another approach to optimize the performance of DDSS is to use a circular array of buffers for both request and reply messages for inter-process communication

as shown in Figure 6.2(c). Applications submit requests using the request circular buffer (Step 1) . The service thread constantly looks for user requests by *polling* at all request buffers (Step 2) and processes each request (Step 3) by issuing the corresponding network operations. The network processes this request (Step 4) and issues a completion (Step 5) in a completion queue. The service thread periodically *polls* on this queue (Step 6) to signal completions to applications threads (Step 7 and 8). It is to be noted that a similar mechanism using memory mapped request and response queues has already been proposed by [83]. This approach completely removes the kernel involvement for both submitting requests and receiving reply messages, thus leading to better performance for several operations in DDSS. However, application threads need to constantly *poll* on the reply buffer to look for a receive completion and this may result in occupying a significant amount of CPU. As application threads in the system increase and if all threads constantly *poll* on the reply buffers, it is very likely that the performance may degrade for systems with limited CPUs or SMTs. However, for systems which support large number of cores or SMTs, this optimization can significantly help improve the performance of the application.

We present various performance results. First, we present the impact of our design optimizations in DDSS at a micro-benchmark level and then we show the performance improvement achieved by applications such as distributed STORM, R-Tree and B-Tree query processing, application checkpointing and resource monitoring services using DDSS.

Our experimental testbed is a 560-core InfiniBand Linux cluster. Each of the 70 compute nodes have dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of 8 cores per node. Each node has a Mellanox MT25208 dual-port Memfree

HCA. For experiments with 10-GigE, we use two Chelsio T3B 10 GigE PCI-Express adapters (firmware version 4.2) connected to two nodes. InfiniBand and 10-Gigabit Ethernet software support is provided through the OpenFabrics/Gen2 stack [77], OFED 1.2 release.

6.3 Basic Performance

In this section, we present the benefits of our design optimizations in DDSS over IBA and 10-GigE.

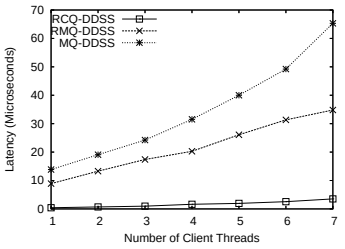
6.3.1 DDSS Latency

First, we measure the performance of inter-process communication (IPC) using the different approaches mentioned in Section 6.2. We design the benchmark in the following way. Each application thread sends an eight byte message through System V message queues or shared memory (using a circular array of buffers) to the service thread. The service thread immediately sends a reply message of eight bytes to the corresponding application thread. Figure 6.3(a) shows the inter-process communication latency with increasing number of application threads for MQ-DDSS, RMQ-DDSS and RCQ-DDSS approaches. We observe that the RCQ-DDSS approach achieves a very low latency of 0.4μsecs while RMQ-DDSS and MQ-DDSS approaches achieve a higher latency of 8.8μsecs and 13μsecs, respectively. This is expected since the RCQ-DDSS approach completely avoids kernel involvement through memory-mapped circular buffers for communication. Also, we see that the performance of RCQ-DDSS approach scales with increasing number of processes as compared to RMQ-DDSS and MQ-DDSS approaches. Next, we measure the performance of DDSS operations including the inter-process communication and the network operations.

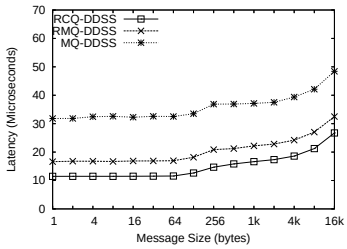
Figure 6.3(b) shows the performance of *get()* operation of DDSS over InfiniBand. We see that the latency of a *get()* operation over InfiniBand using RCQ-DDSS approach is 8.2μsecs while the RMQ-DDSS and MQ-DDSS approaches show a latency of up to 11.3μsecs and 22.6μsecs, respectively. For increasing message sizes, we observe that the latency increases for all three approaches. We see similar trends for a *get()* operation over iWARP-capable 10-Gigabit Ethernet as shown in Figure 6.3(c).

6.3.2 DDSS Scalability

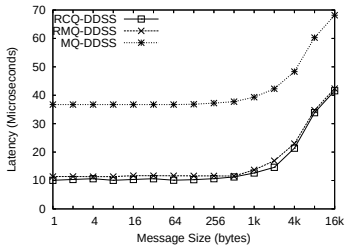
Here, we measure the scalability of DDSS with increasing number of processes performing the DDSS operations over IBA. First, we stress the inter-process communication and show the performance of the three approaches, as shown in Figure 6.4(a). We observe that, for very large number of client threads (up to 512), the RMQ-DDSS approach performs significantly better than RCQ-DDSS and MQ-DDSS approaches. Since the RCQ-DDSS approach uses significant amount of CPU to check for completions, it does not scale well with large number of threads. In the case of MQ-DDSS approach, it generates twice the number of kernel events as compared to the RMQ-DDSS approach and thus it performs worse. Next, we stress the network and show the scalability of DDSS. In this experiment, we allocate the data on a single node and multiple applications from different nodes access different portions of the data simultaneously, as shown in Figure 6.4(b). We compare its performance by distributing the data across different nodes in the cluster and show its scalability. As shown in Figure 6.4(c), we observe that the performance of DDSS scales with increasing number of clients using the distributed approach as compared to the non-distributed approach.



(a) IPC Latency

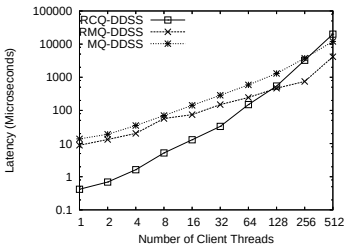


(b) Performance of *get()* in IBA

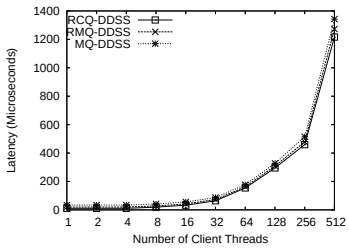


(c) Performance of *get()* in 10-GigE

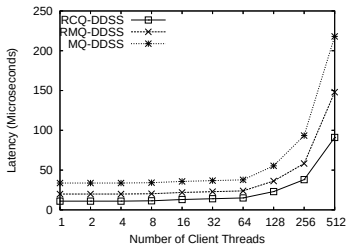
Figure 6.3: DDSS Latency



(a) IPC Latency



(b) Performance of *get()* for single key



(c) Performance of *get()* for distributed keys

Figure 6.4: DDSS Scalability

6.4 Application-level Evaluations

In this section, we present the benefits of DDSS using applications such as R-Tree and B-Tree query processing, distributed STORM, application checkpointing and resource monitoring services over IBA.

6.4.1 R-Tree Query Processing

R-Tree [59] is a hierarchical indexing data structure that is commonly used to index multi-dimensional geographic data (points, lines and polygons) in the fields of databases, bio-informatics and computer vision. In our experiments, we use an R-tree query processing application, developed by Berkeley [23] that uses helper functions such as *read_page* and *write_page* to read and write the indexing information to the disk. We place the indexing information on a network-based file system so that multiple threads can simultaneously access this information for processing different queries. We modify the *read_page* and *write_page* function calls to use the *get()* and *put()* operations of DDSS and place the indexing information on DDSS to show the benefits of accessing this information from remote memory as compared to the disk using the network-based file system. Table 6.1 shows the overall execution time of an R-Tree application with varying percentage of queries (100% query implies that the query accesses all the records in the database while 20% implies accessing only 20% of the records). As shown in the table, we see that all three approaches (RCQ-DDSS, RMQ-DDSS and MQ-DDSS) improve the performance by up to 56% as compared to the traditional approach (No DDSS approach) and the RCQ-DDSS approach shows an improvement of up to 9% and 4% as compared to MQ-DDSS and RMQ-DDSS approaches, respectively. Moreover, for increasing percentage of query accesses, we

see that the performance improvement decreases. This is expected since applications spend more time in computation for large queries as compared to small queries, thus reducing the overall percentage benefit.

6.4.2 B-Tree Query Processing

B-Tree [32] is a data structure that is commonly used in databases and file systems which maintains sorted data and allows operations such as searches, insertions and deletions in logarithmic amortized time. In our experiments, we use a B-Tree query processing application, developed by Berkeley [23] that uses similar helper functions to read and write the indexing information. Similar to the R-Tree application, we place the indexing information in DDSS and compare its performance with accessing the information from the disk using network-based file system. Table 6.1 shows the overall execution time of a B-Tree application with varying percentage of queries. As shown in Table 6.1, we see that all three approaches (RCQ-DDSS, RMQ-DDSS and MQ-DDSS) improve the performance by up to 45% as compared to the traditional approach (No DDSS approach) and the RCQ-DDSS approach shows an improvement of up to 3% and 1% as compared to MQ-DDSS and RMQ-DDSS approaches, respectively.

6.4.3 Distributed STORM

STORM [70, 24] is a middle-ware layer developed by the Department of Biomedical Informatics at The Ohio State University. It is designed to support SQL-like queries on datasets primarily to select the data of interest and transfer the data from storage nodes to compute nodes for data processing. In our previous work [97], we demonstrated the improvement of placing the datasets in DDSS. In this work, we

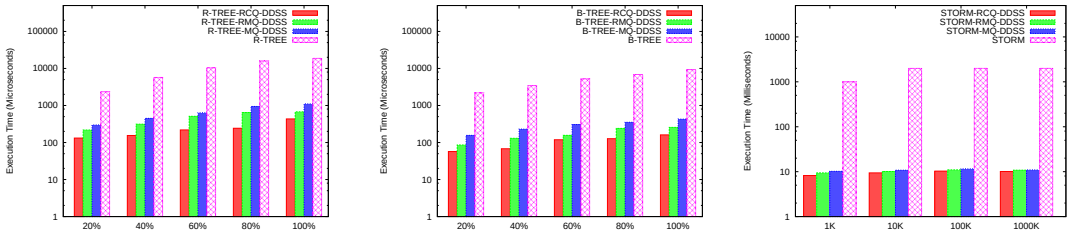
place the meta-data information of STORM in DDSS and show the associated benefits as compared to accessing the meta-data using TCP/IP communication protocol. Table 6.1 shows the overall execution time of STORM with varying record sizes. As shown in the table, we see that all three approaches improve the performance by 44% as compared to the traditional approach (No DDSS approach). STORM establishes multiple connections with the directory server to get the meta-data information and uses socket-based calls to send and receive the data, which is a two-sided communication protocol. Most of the benefits shown is achieved mainly due to avoiding connections (in the order of several milliseconds) and using one-sided communication model.

Application		No. of DDSS Operations	Avg Size (bytes)	Overall Execution Time (milliseconds)			
				No DDSS	MQ-DDSS	RMQ-DDSS	RCQ-DDSS
R-Tree	20%	8	8192	3.917	1.868	1.786	1.700
	40%	23	8192	12.427	7.149	7.013	6.855
	60%	41	8192	25.360	15.609	15.481	15.189
	80%	60	8192	42.457	27.529	27.232	26.830
	100%	74	8192	60.781	43.064	42.587	42.385
B-Tree	20%	8	8192	4.715	2.675	2.605	2.576
	40%	13	8192	8.114	4.906	4.805	4.743
	60%	20	8192	12.242	7.336	7.186	7.149
	80%	26	8192	16.040	9.490	9.425	9.311
	100%	33	8192	20.400	11.534	11.360	11.265
STORM	1K	86	7.6	2250	1260.2	1259.4	1258.2
	10K	177	6.4	4900	1910.8	1910.1	1909.4
	100K	158	6.5	6200	3211	3210.7	3210
	1000K	125	6.4	13100	11110.5	11110.3	11109.6

Table 6.1: Application Performance

To understand the decrease in performance improvement of RCQ-DDSS approach as compared to MQ-DDSS and RMQ-DDSS approaches, we repeat the experiments

and report the time taken by the data sharing component in applications in Figure 6.5. The performance benefits achieved in an R-Tree query processing application is shown in Figure 6.5(a). We see that the performance benefits achieved by the RCQ-DDSS approach as compared to MQ-DDSS and RMQ-DDSS approaches is 56% and 27%, respectively. However, we also observe that the MQ-DDSS approach achieves close to 87% performance improvement as compared to R-Tree query processing without DDSS, with only 13% of the remaining time to be optimized further. As a result, we see marginal improvements in application performance using the RCQ-DDSS approach as compared to MQ-DDSS and RMQ-DDSS approaches. We see similar trends for B-Tree query processing and STORM as shown in Figures 6.5(b) and 6.5(c).



(a) R-Tree Query Processing

(b) B-Tree Query Processing

(c) Meta-data in Distributed STORM

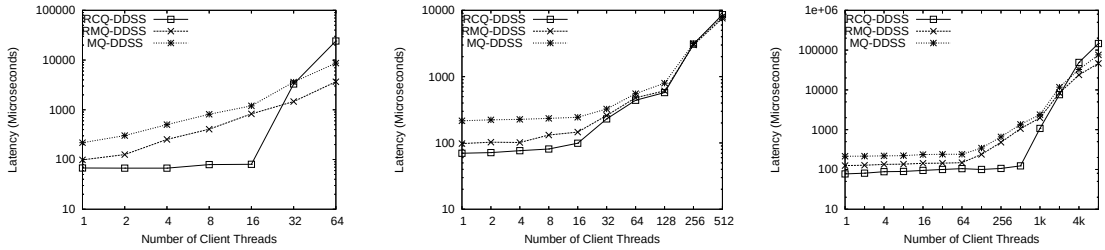
Figure 6.5: State Sharing Performance in Applications

6.4.4 Application Checkpointing

Here, we present our evaluations with an application checkpointing benchmark [97] to demonstrate the scalability of all three approaches. In this experiment, every process checkpoints an application at random time intervals (if the current version

does not match, it restarts to a previous consistent version, else commits an updated version). Also, every process simulates the application restart by taking a consistent checkpoint at other random intervals based on a failure probability 0.001 (0.1%).

Figure 6.6(a) shows the performance of checkpoint applications with increasing number of processes within the same node. We observe that the performance of the RCQ-DDSS approach scales with increasing number of application processes for up to 16. However, for large number of processes up to 64, we see that the RMQ-DDSS approach performs better which confirms the results shown in Section 6.3.2. Figure 6.6(b) shows the performance of checkpoint applications with increasing number of processes on different nodes. Here, we stress the network by assigning the operations in DDSS on one single remote node and all processes perform DDSS operations on this remote node. As shown in Figure 6.6(b), we see that the performance of all three approaches scale for up to 16 processes. However, for processes beyond 16, the performance of all three approaches fail to scale due to network contention. Further, with 512 processes, we observe that the performance of the RCQ-DDSS approach performs significantly worse as compared to RMQ-DDSS and MQ-DDSS approaches which confirms the results shown in Section 6.3.2. Next, we show the performance by distributing the operations in DDSS on all the nodes in the system and show its scalability for up to 8192 processes. As shown in the Figure 6.6(c), we observe that the RCQ-DDSS approach scales for up to 512 processes. However, for very large number of processes up to 8192, we see that the RMQ-DDSS approach performs better as compared to RCQ-DDSS and MQ-DDSS approaches, confirming our earlier observations in Section 6.3.2.



(a) Clients within a single node (non-distributed)

(b) Clients across different nodes (non-distributed)

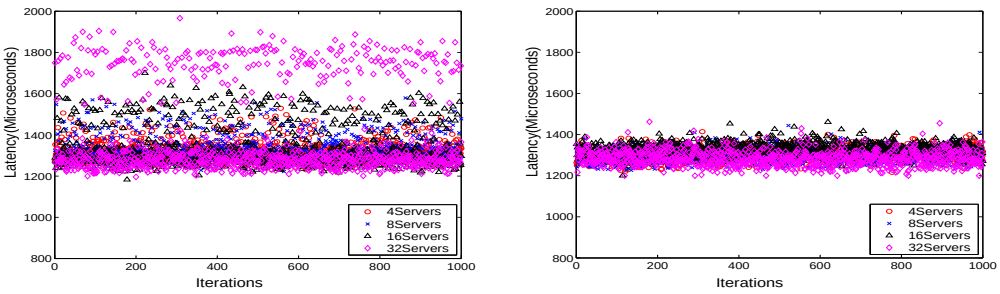
(c) Clients across different nodes (distributed)

Figure 6.6: Checkpoint Application Performance

6.5 Datacenter Services on Dedicated Cores

Several existing services such as resource monitoring, caching, distributed lock manager can be built on top of the state sharing substrate. While these services help improve the performance of many applications, it can also affect the performance of other applications that run concurrently, since it requires some amount of CPU to perform these tasks. To demonstrate this effect, we use a resource monitoring application [94] and build it as a part of a DDSS service and show its impact with a proxy server that directs client requests to web servers for processing HTML requests. The DDSS service periodically monitors the system load on the web servers by issuing a *get()* operation. For our evaluations, we use a cluster system consisting of 48 nodes and each node has two Intel Xeon 2.6 GHz processors with a 512 KB L2 cache and 2 GB of main memory. In the 48-node experimental testbed, we use one CPU for both the proxy server and the resource monitoring service and blocked the other CPU with dummy computations. Figure 6.7(a) shows the response time seen by clients in

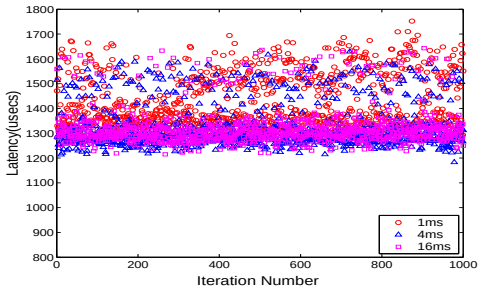
requesting a 16 KB file from the web server. We observe that the client response time fluctuates significantly depending on the number of web servers monitored by the DDSS service. With 32 web servers, we see that the client response time can get affected by almost 50%. Next, we use one CPU for the proxy server and an additional CPU for the DDSS service and show the performance impact in Figure 6.7(b). We observe that the client response time remains unaffected irrespective of the number of servers being monitored by the DDSS service. Accordingly, applications with stringent quality of service requirements can use a dedicated core to perform the additional services and still meet their requirements in an efficient manner. We also varied the experiments in terms of different file size requests, as shown in Figure 6.9 and different monitoring granularities, as shown in Figure 6.8 and see similar trends.



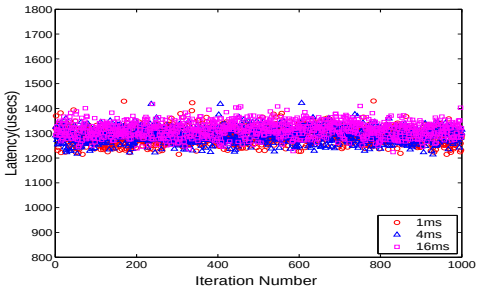
(a) DDSS Service and Web Server on the same core

(b) DDSS Service and Web Server on different cores

Figure 6.7: Performance impact of DDSS on Web Servers: Number of Servers

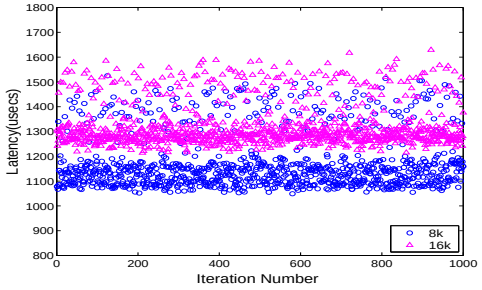


(a) DDSS Service and Web Server on the same core

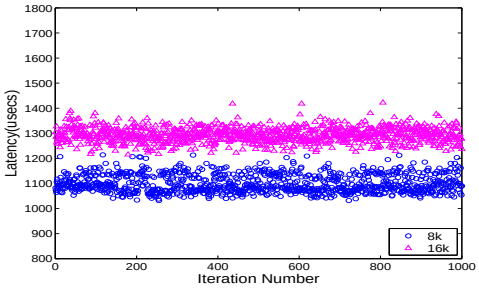


(b) DDSS Service and Web Server on different cores

Figure 6.8: Performance impact of DDSS on Web Servers: Monitoring Granularity



(a) DDSS Service and Web Server on the same core



(b) DDSS Service and Web Server on different cores

Figure 6.9: Performance impact of DDSS on Web Servers: Different File Sizes

6.6 Summary

In this Chapter, we presented design optimizations in DDSS for systems with high-performance networks and multicore architectures and comprehensively evaluated DDSS in terms of performance, scalability and associated overheads using several micro-benchmarks and applications such as Distributed STORM, R-Tree and B-Tree query processing, checkpointing applications and resource monitoring services. Our micro-benchmark results not only showed a very low latency in DDSS operations but also demonstrated the scalability of DDSS with increasing number of processes. Application evaluations with R-Tree and B-Tree query processing and distributed STORM showed an improvement of up to 56%, 45% and 44%, respectively, as compared to traditional implementations. Evaluations with application checkpointing demonstrated the scalability of DDSS. Further, we demonstrated the portability of DDSS across multiple modern interconnects such as InfiniBand and iWARP-capable 10-Gigabit Ethernet networks (applicable for both LAN/WAN environments). In addition, our evaluations using an additional core for DDSS services showed a lot of potential benefits for performing services on dedicated cores.

CHAPTER 7

DMA-ACCELERATED, NETWORK-ASSISTED STATE SHARING

Figure 7.1 shows the various components of DMA-accelerated, network-assisted state sharing substrate. Broadly, in the figure, we focus on the colored boxes for designing efficient DMA-accelerated, network-assisted state sharing components that assist network processing and understanding its benefits with datacenter applications. The dark colored boxes show the features and technologies that we utilize and the light colored boxes show the proposed components and datacenter system software evaluated. Note that the design optimizations and the benefits shown in this section demonstrate the capabilities of an asynchronous DMA engine in accelerating the network communication for Ethernet-based networks.

7.1 Background and Related Work

Several clients request datacenter servers for either the raw or some kind of processed data simultaneously. However, existing servers are becoming increasingly incapable of meeting such sky-rocketing processing demands with high-performance and scalability. These servers rely on TCP/IP for data communication and typically use

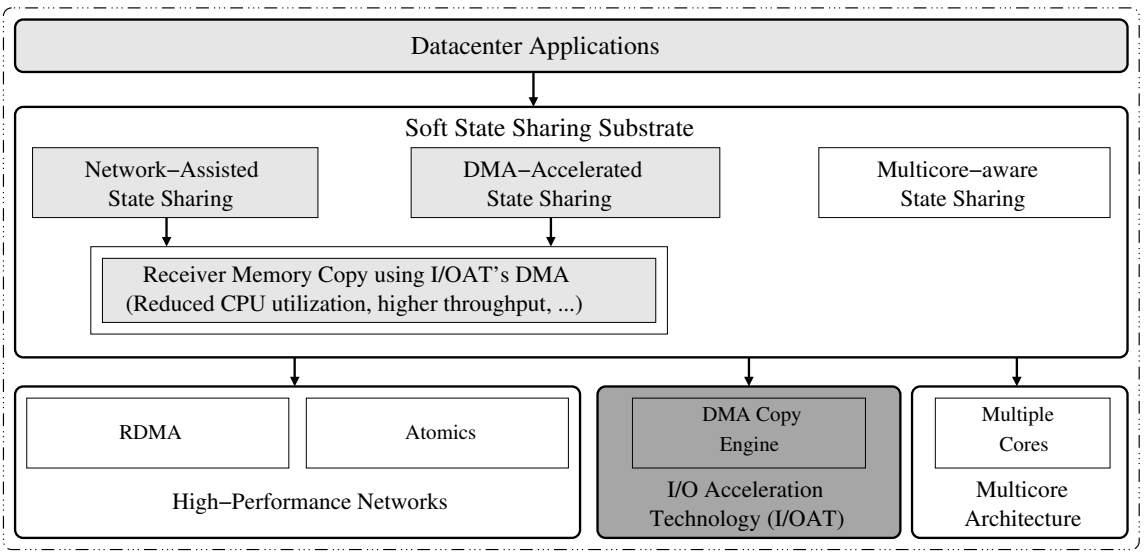


Figure 7.1: DMA-Accelerated, Network-Assisted State Sharing Framework

Gigabit Ethernet networks for cost-effective designs. The host-based TCP/IP protocols on such networks have high CPU utilization and low bandwidth, thereby limiting the maximum capacity (in terms of requests they can handle per unit time). Alternatively, many servers use multiple Gigabit Ethernet networks to cope with the network traffic. However, at multi-Gigabit data rates, packet processing in the TCP/IP stack occupies a significant portion of the system overhead.

Packet processing [50, 53] usually involves manipulating the headers and moving the data through the TCP/IP stack. Though this does not require significant computation, processor time gets wasted due to delays caused by latency of memory accesses and data movement operations. To overcome these overheads, researchers have proposed several techniques [45] such as transport segmentation offload (TSO), jumbo frames, zero-copy data transfer (sendfile()), interrupt coalescing, etc. Unfortunately, many of these techniques are applicable only on the sender side, while the

receiver side continues to remain as a bottleneck in several cases, thus resulting in a huge performance gap between the CPU overheads of sending and receiving packets.

Intel's I/O Acceleration Technology (I/OAT) [1, 13, 12, 58] is a set of features which attempts to alleviate the receiver packet processing overheads. It has three additional features, namely: (i) split headers, (ii) DMA copy offload engine and (iii) multiple receive queues. However, in this work [98], we only evaluate the benefits of DMA copy offload engine feature in network processing.

Related Work: Packet processing [50, 53] usually involves manipulating the headers and moving the data through the TCP/IP stack. Though this does not require significant computation, processor time gets wasted due to delays caused by latency of memory accesses and data movement operations. To overcome these overheads, researchers have proposed several techniques [45] such as transport segmentation offload (TSO), jumbo frames, zero-copy data transfer (sendfile()), interrupt coalescing, etc. Unfortunately, many of these techniques are applicable only on the sender side, while the receiver side continues to remain as a bottleneck in several cases, thus resulting in a huge performance gap between the CPU overheads of sending and receiving packets.

7.2 I/OAT Micro-Benchmark Results

In this section, we compare the ideal case performance benefits achievable by I/OAT as compared to the native sockets implementation (non-I/OAT) using a set of micro-benchmarks. We use two testbeds for all of our experiments. Their descriptions are as follows:

Testbed 1: A system consisting of two nodes built around SuperMicro X7DB8+ motherboards which include 64-bit 133 MHz PCI-X interfaces. Each node has a

dual dual-core Intel 3.46 GHz processor with a 2 MB L2 cache. The machines are connected with three Intel PRO 1000Mbit adapters with two ports each through a 24-port Netgear Gigabit Ethernet switch. We use the Linux RedHat AS 4 operating system and kernel version 2.6.9-30.

Testbed 2: A cluster system consisting of 44 nodes. Each node has a dual Intel Xeon 2.66 GHz processor with 512KB L2 cache and 2GB of main memory.

For the experiments mentioned in Sections 7.3, we use the nodes in Testbed 2 as clients and the nodes in Testbed 1 as servers. For all other experiments, we only use the nodes in Testbed 1. Also, for experiments within Testbed 1, we create a separate VLAN for each network adapter in one node and a corresponding IP address within the same VLAN on the other node to ensure an even distribution of network traffic. In all of our experiments, we define the term relative CPU benefit of I/OAT as follows: if a is the % CPU utilization of I/OAT and b is the % CPU utilization of non-I/OAT, the relative CPU benefit of I/OAT is defined as $(b - a)/b$. For example, if I/OAT occupies 30% CPU and non-I/OAT occupies 60% CPU, the relative CPU benefit of I/OAT is 50%, though the absolute difference in CPU usage is only 30%.

7.2.1 Bandwidth: Performance and Optimizations

Bandwidth and Bi-directional Bandwidth: Figure 7.2(a) shows the bandwidth achieved by I/OAT and non-I/OAT with an increasing number of network ports. We use the standard *ttcp* benchmark [27] for measuring the bandwidth. As the number of ports increase, we expect the bandwidth to increase. As shown in Figure 7.2(a), we see that the bandwidth performance achieved by I/OAT is similar to the performance achieved by non-I/OAT with an increasing number of ports. The maximum

bandwidth achieved is close to 5635 Mbps with six network ports. However, we see a difference in performance with respect to the CPU utilization on the receiver side. We observe that the CPU utilization is lower for I/OAT as compared to non-I/OAT using three network ports and this difference increases as we see the number of ports increase from three to six. For a six port configuration, non-I/OAT occupies close to 37% of the CPU while I/OAT occupies only 29% of the CPU. The relative benefit achieved by I/OAT in this case is close to 21%.

In the bi-directional bandwidth test, we use two machines and 2*N threads on each machine with N threads acting as servers and the other N threads as clients. Each thread on one machine has a connection to exactly one thread on the other machine. The client threads connect to the server threads on the other machine. Thus, 2*N connections are established between these two machines. On each connection, the basic bandwidth test is performed using the *ttcp* benchmark. The aggregate bandwidth achieved by all threads is calculated as the bi-directional bandwidth, as shown in Figure 7.2(b). In our experiments, N is equal to the number of network ports. We see that the maximum bi-directional bandwidth is close to 9600 Mbps. Also, we observe that I/OAT shows an improvement in CPU utilization using only two ports and this improvement increases with an increasing number of ports. With six network ports, non-I/OAT occupies close to 90% of CPU whereas I/OAT occupies only 70% of the CPU. The relative CPU benefits achieved by I/OAT is close to 22%. This trend also suggests that with an addition of one or two network ports to this configuration, non-I/OAT may not give the best network throughput in comparison with I/OAT since non-I/OAT may end up occupying 100% CPU.

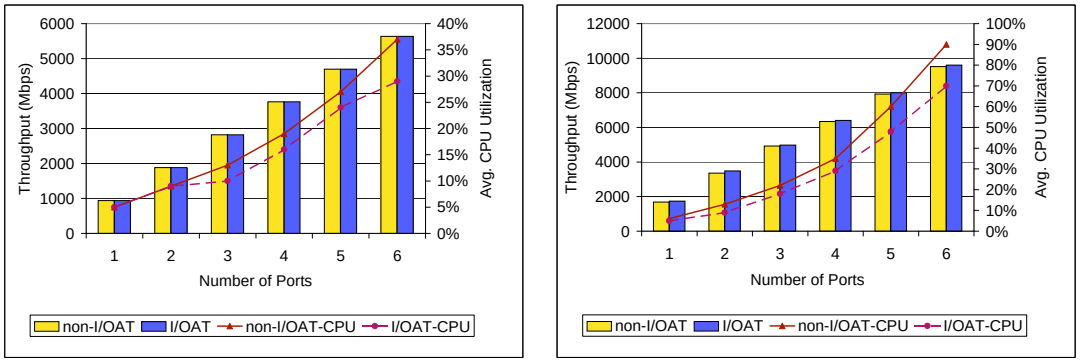


Figure 7.2: Micro-Benchmarks: (a) Bandwidth and (b) Bi-directional Bandwidth

Multi-Stream Bandwidth: The multi-stream bandwidth test is very similar to the bi-directional bandwidth test mentioned above. However, in this experiment, only one machine acts as a server and the other machine as the client. We use two machines and N threads on each machine. Each thread on one machine has a connection to exactly one thread on the other machine. On each connection, the basic bandwidth test is performed. The aggregate bandwidth achieved by all threads is calculated as the multi-stream bandwidth. As shown in Figure 7.3, we observe that the bandwidth achieved by I/OAT is similar to the bandwidth achieved by non-I/OAT for an increasing number of threads. However, when the number of threads increases to 120, we see a degradation in performance of non-I/OAT; whereas, I/OAT consistently shows no degradation in network bandwidth. Further, the CPU utilization for non-I/OAT also increases with an increasing number of threads. With 120 threads in the system, we see that non-I/OAT occupies close to 76% CPU whereas I/OAT only occupies 52% resulting in 24% absolute benefit in CPU utilization. The relative CPU benefits achieved by I/OAT in this case is close to 32%.

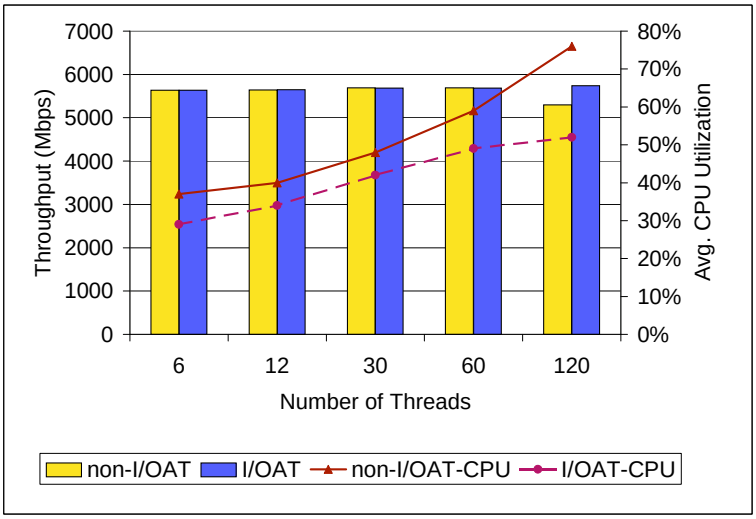


Figure 7.3: Multi-Stream Bandwidth

Bandwidth and Bi-directional Bandwidth with Socket Optimizations: As mentioned in Section 1.2.2, several optimizations on the sender side exist to reduce the packet overhead and also to improve the network performance. In this experiment, we considered three such existing optimizations: (i) Large Socket buffer sizes (100 MB), (ii) Segmentation Offload (TSO) and (iii) Jumbo Frames. In this experiment, we aim to understand the impact of each of these optimizations and observe the improvement, both in terms of throughput and CPU utilization on the receiver-side. Case 1 uses the default socket options without any optimization. In Case 2, we increase the socket buffer size to 100 MB. For Case 3, we further improve the optimization by enabling segmentation offload (TSO) so that the host CPU is relieved from fragmenting large packets. In Case 4, in addition to the previous optimizations, we increase the MTU-size to 2048 bytes so that large packets are sent over the network. In addition to

the sender-side optimizations, in Case 5, we include the interrupt coalescing feature. Performance numbers with various socket optimizations are shown in Figure 7.4.

In the bandwidth test, as shown in Figure 7.4(a), we observe two interesting trends. First, as we increase the socket optimizations, we see an increase in the aggregate bandwidth. Second, we observe that the performance of I/OAT is consistently better than the performance of non-I/OAT. Especially for Case 5, which includes all socket optimizations, the bandwidth achieved by I/OAT is close to 5586 Mbps, whereas non-I/OAT achieves only 5514 Mbps. More importantly, we observe that there is a significant improvement in terms of CPU utilization. As shown in Figure 7.4(a), the relative CPU benefit of I/OAT increases as we increase the socket optimizations. We see similar trends with the bi-directional bandwidth test as shown in Figure 7.4(b).

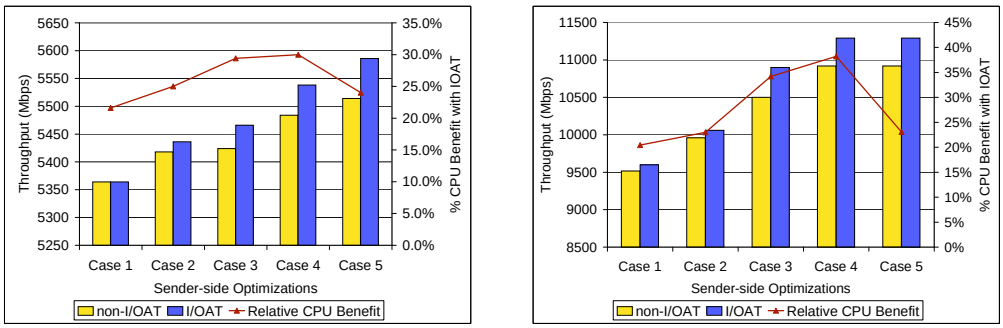


Figure 7.4: Optimizations: (a) Bandwidth and (b) Bi-directional Bandwidth

In summary, we see that the micro-benchmark results show a significant improvement in terms of CPU utilization and network performance for I/OAT. Since I/OAT in Linux has two additional features, the split-header and DMA copy engine, it is also

important to understand the benefits attributed by each of these features individually. In the following section, we conduct several experiments to show the individual benefits.

7.2.2 Benefits of Asynchronous DMA Copy Engine

In this section, we isolate the DMA engine feature of I/OAT and show the benefits of an asynchronous DMA copy engine. We compare the performance of the copy engine with that of traditional CPU-based copy and show its benefits in terms of performance and overlap efficiency. For CPU-based copy, we use the standard *memcpy* utility.

Figure 7.5 compares the cost of performing a copy using the CPU and I/OAT's DMA engine. The *copy-cache* bars denote the performance of CPU-based copy with the source and destination buffers in the cache and the *copy-nocache* bars denote the performance with the source and destination buffers not in the cache. The *DMA-copy* bars denote the total time taken to perform the copy using the copy engine and the *DMA-overhead* bars include the startup overhead in initiating the copy using the DMA engine. The *Overlap* line denotes the percentage of DMA copy time that can be overlapped with other computations. As shown in Figure 7.5, we see that the performance of DMA-based copy approach (*DMA-copy*) is better than the performance of CPU-based copy approach (*copy-nocache*) for message sizes greater than 8 KB. Further, we observe that the percentage of overlap increases with increasing message sizes reaching up to 93% for a 64 KB message size. However, if the source and destination buffers are in the cache, we observe that the performance of the CPU-based copy is much better than the performance of the DMA-based copy approach. Also, since the

DMA-based copy can be overlapped with processing other packets, we incur only the DMA startup overheads. As observed in Figure 7.5, we see that the DMA startup overhead time is much less than the time taken by the CPU-based copy approach. Thus, the DMA copy engine can also be useful even if the source and destination buffers are in cache.

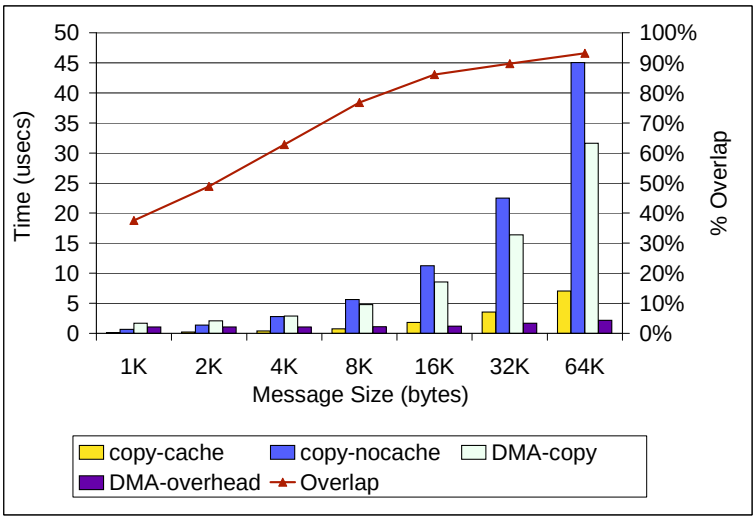


Figure 7.5: Copy Performance: CPU vs DMA

7.3 Datacenter Performance Evaluation

In this section, we analyze the performance of a 2-tier datacenter environment with I/OAT and compare its performance with non-I/OAT. For all experiments in this section, we use the nodes in Testbed 1 for the datacenter tiers. For the client nodes, we use the nodes in Testbed 2 for most of the experiments. We notify the

readers at appropriate points in the remaining sections when other nodes are used as clients.

7.3.1 Evaluation Methodology

As mentioned in Section 1.2.2, I/OAT is a server architecture geared to improve the receiver-side performance. In other words, I/OAT can be deployed in a datacenter environment and client requests coming over the Wide Area Network (WAN) can seamlessly take advantage of I/OAT and get serviced much faster. Further, the communication between the tiers inside the datacenter, such as proxy tier and application tier, can be greatly enhanced using I/OAT, thus improving the overall datacenter performance and scalability.

We set up a two-tier datacenter testbed to determine the performance characteristics of using I/OAT and non-I/OAT. The first tier consists of the front-end proxies. For this, we used the proxy module of Apache 2.2.0. The second tier consists of the web server module of Apache in order to service static requests. The two tiers in the datacenter reside on 1 Gigabit Ethernet network; the clients are connected to the datacenter using a 1 Gigabit Ethernet connection.

Typical datacenter workloads have a wide range of characteristics. Some workloads may vary from high to low temporal locality, following a Zipf-like distribution [37]. Similarly workloads vary from small documents (e.g., on-line book stores, browsing sites, etc.) to large documents (e.g., download sites, etc.). Further, workloads may contain requests for simple cacheable static or time invariant content or more complex dynamic or time variant content via CGI, PHP, and Java servlets with a back-end database. Due to these varying characteristics of workloads, we classify

the workloads into three broad categories: (i) Single-file Micro workloads, (ii) Zipf-like workloads and (iii) Dynamic content workloads. However, in this work, we focus our analysis on the first two categories.

Single-File Micro workloads: These workloads contain only a single file. Several clients request the same file multiple times. These workloads are used to study the basic performance achievable by the datacenter environment without being diluted by other interactions in more complex workloads. We use workloads ranging from 2 KB to 10 KB file sizes since this is considered the average file size for most documents in the Internet.

Zipf-like Workloads: It has been well acknowledged in the community that most workloads for datacenters hosting static content, follow a Zipf-like distribution [37]. According to Zipf law, the relative access probability of a request for the i'th most popular document is proportional to $1/i^{\alpha}$, where α determines the randomness of file accesses. In our experiments, we vary α from 0.95 to 0.5, ranging from high temporal locality for documents to low temporal locality.

We also evaluate the performance achieved inside the datacenter with proxy servers acting as clients and web-servers with I/OAT capability. For example, within a datacenter, the proxy servers forward dynamic content requests to the application servers. The application servers are known to be CPU-intensive due to processing of scripts such as PHP, CGI, servlets, etc. If the application servers have I/OAT capability, the servers can not only accept more number of requests but can also process the pending requests at a faster rate, due to reduced CPU utilization.

For both of the scenarios, we use a testbed with one proxy at the first tier and one web-server at the second tier. Each client fires one request at a time and sends another request after getting a reply from the server.

In this section, we separate our analysis into two categories. First, we analyze the performance benefits of I/OAT with the two workloads. As mentioned above, we use Testbed 2 to fire requests to the proxy server. Due to the I/OAT capability on the server nodes, we expect the performance of the servers to improve.

7.3.2 Analysis with Single File Traces

In this experiment, we use five different traces with varying file size requests. Trace 1 uses an average file size of 2 KB while Traces 2, 3, 4 and 5 use 4 KB, 6 KB, 8 KB and 10 KB file sizes, respectively. Each client has a 10,000 request subset of different files and reports the TPS (Transactions Per Second) achieved after getting the response from the servers for all of the requests. TPS is defined as the total number of transactions serviced per second as seen by the client. Higher TPS values attribute to better server performance.

Figure 7.6a shows the performance of a datacenter with varying trace files ranging from 2 KB to 10 KB. As shown in Figure 7.6a, we see that the throughput reported by I/OAT is consistently better than the throughput reported by non-I/OAT. It is also to be noted that for Trace 2, with average file size of 4 KB, we see a significant throughput improvement for I/OAT. I/OAT reports a TPS which is close to 9,754 whereas non-I/OAT reports only 8,569 TPS resulting in a 14% overall improvement. For other traces, we see around 5-8% TPS improvement with I/OAT.

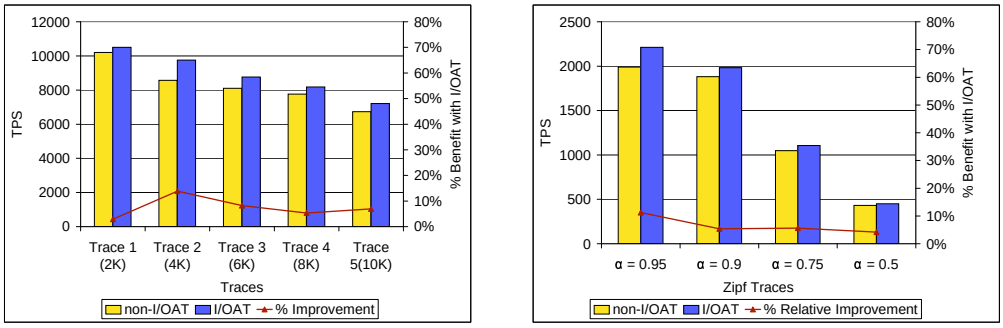


Figure 7.6: Performance of I/OAT and non-I/OAT: (a) Single File Traces and (b) Zipf Trace

7.3.3 Analysis with Zipf File Traces

Next, we evaluate the performance of the datacenter with the zipf-trace. As mentioned earlier, we vary the α values from 0.95 to 0.5 ranging from high temporal locality to low temporal locality. As shown in Figure 7.6b, I/OAT consistently performs equal to or better than non-I/OAT. Further, we note that non-I/OAT achieves close to 1,989 TPS whereas I/OAT achieves close to 2,212 TPS, i.e., I/OAT achieves up to 11% throughput benefit as compared to non-I/OAT. Also, we measured the CPU utilization for these experiments but the improvement was negligible. However, the throughput improves since the server can accept more requests (i.e., reduction in CPU overheads result in greater CPU efficiency).

7.3.4 Analysis with Emulated Clients

Next, we evaluate the performance of I/OAT within a datacenter when both the proxy and the web servers have I/OAT capability. In this experiment, the proxy server acts as a client in sending the requests to the web servers. We use only the nodes in

Testbed 1, as mentioned in Section 7.2. Due to reduced CPU usage in proxy nodes using I/OAT, we expect the clients to fire requests at a much faster rate, resulting in higher throughput. Figure 7.7 shows the performance with I/OAT capability for increasing number of client threads. In this experiment, we fix the file size to 16 KB and vary the number of client threads from 1 to 256. As shown in Figure 7.7, we note that the performance of I/OAT is similar to non-I/OAT from one to sixteen threads. As the number of threads firing the requests increases from 32 to 256, we observe two interesting trends. First, I/OAT throughput performance increases with an increasing number of client threads. With 256 client threads, I/OAT achieves close to 14,996 TPS whereas non-I/OAT achieves only around 12,928, i.e., I/OAT achieves close to 16% throughput improvement as compared to non-I/OAT. It is also to be noted that the CPU utilization of I/OAT is consistently lower than the CPU utilization of non-I/OAT. In this experiment, we only report the CPU utilization on the client node, since our focus is to capture the client-side benefits when clients have I/OAT capability. We observe that the CPU utilization saturates with non-I/OAT with 64 threads and thus the throughput does not improve any further with an increase in the client threads after 64. With I/OAT, we see that the CPU utilization saturates only with 256 client threads, resulting in superior performance. I/OAT not only improves the performance by 16%, but can also support a large number of threads (up to a factor of four as compared to non-I/OAT).

7.4 Summary

I/O Acceleration technology (I/OAT) developed by Intel is a set of features particularly designed to reduce the packet processing overheads on the receiver-side. This

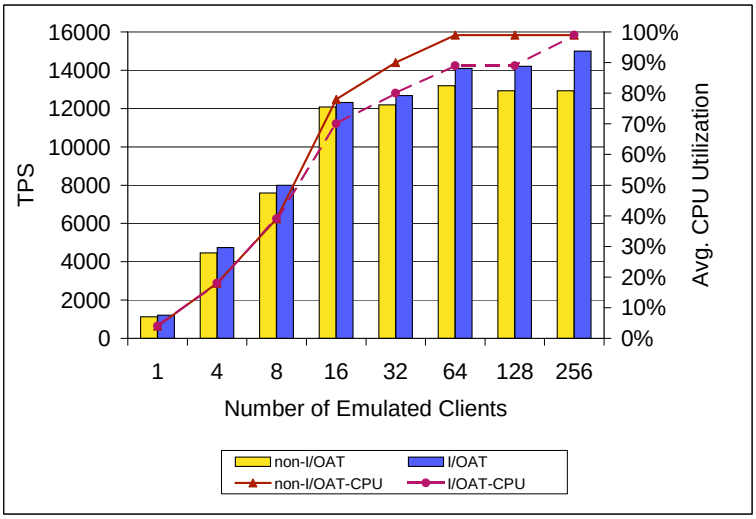


Figure 7.7: Clients with I/OAT capability using a 16 KB file trace

Chapter studies the benefits of I/OAT technology through extensive evaluations of micro-benchmarks as well as evaluations on a multi-tier datacenter environment. Our micro-benchmark results showed that I/OAT results in 38% lower overall CPU utilization in comparison with traditional communication. Due to this reduced CPU utilization, I/OAT delivers better performance and increased network bandwidth. Our experimental results with datacenters reveal that I/OAT can improve the total number of transactions processed by 14%. In addition, I/OAT can sustain a large number of concurrent threads (up to a factor of four as compared to non-I/OAT) in datacenter environments, thus increasing the scalability of the servers.

CHAPTER 8

APPLICABILITY OF STATE SHARING COMPONENTS IN DATACENTER ENVIRONMENTS

In this Chapter, we highlight several scenarios and system environments where each of the state sharing components can be applicable. This can help the end users who build datacenter system software to choose the right components based on the environment and the workload expected in order to get the maximum benefits. Table 8.1 summarizes the different datacenter workload environments and the benefits that can be achieved using the different state sharing components depending on the systems capability. We use the following notations to refer to the different technologies: (i) HPN refers to the network-assisted state sharing component using high-performance networks, (ii) I/OAT refers to the dma-accelerated state sharing component using Intel's I/O Acceleration Technology and (iii) MC refers to the multicore-aware state sharing component using multicore systems. The combinations of these notations such as HPN & I/OAT refers to the dma-accelerated, network-assisted state sharing component, HPN & MC refers to the multicore-aware, network-assisted state sharing component and MC & I/OAT refers to the multicore-aware, dma-accelerated state sharing component.

Workloads	Performance Metric	State Sharing Components					
		HPN	I/OAT	MC	HPN & I/OAT	HPN & MC	MC & I/OAT
Static	Latency	Good	Good	Good	**Best**	**Best**	Good
	Throughput	Good	Good	Bad	**Best**	Good	Bad
	Both	Good	Good	Avg	**Best**	Good	Avg
Dynamic	Latency	Good	Good	Good	**Best**	**Best**	Good
	Throughput	Good	Good	Avg	**Best**	Good	Bad
	Both	Good	Good	Avg	**Best**	Good	Avg
Streaming	Bandwidth	Good	Good	Bad	**Best**	Good	Bad

Table 8.1: Applicability of State Sharing Components in Datacenter Environments

For static datacenter workloads, the files are usually lesser in size, typically around 4 KB to 10 KB. In such workloads, if the requirements are latency-sensitive, we expect that both dma-accelerated, network-assisted state sharing component and multicore-aware, network-assisted state sharing component can give significant benefits to applications. For systems with I/OAT, since the DMA engine accelerates the copy operations of memory operations and saving a few CPU cycles, we may not see a huge latency benefits. However, if the workload is throughput-sensitive that occupies significant CPU, the dma-accelerated state sharing component can help in improving the performance. In such an environment, since applications are in need of CPU, the multicore-aware state sharing and its interactions may show lesser benefits or can even degrade the application performance. Thus, for systems with all three technologies, it is beneficial to use the dma-accelerated, network-assisted state sharing to get the maximum benefits. For workloads which have both latency and throughput as a constraint, we expect that using the dma-accelerated, network-assisted state sharing component can satisfy both requirements and show maximum benefits as compared to the other combinations.

In the case of dynamic workloads, we see similar trends, except that multicore-aware state sharing component and its interactions start playing an important role. As demonstrated by recent literature [71, 72], the role of one-sided operations in dynamic content caching becomes an important factor in improving the performance. Thus, we expect that the interactions of multicore systems can further improve the performance. For streaming workloads, since we deal with large files, typically it is bandwidth or throughput-sensitive. Thus, the dma-accelerated, network-assisted state sharing component is expected to show the maximum benefits. It is to be noted that the expected benefits are only applicable to current generation systems and technologies and can significantly change for next-generation datacenters.

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

With the increasing adoption of Internet as the primary means of electronic interaction and communication, web-based datacenters have become a common requirement for providing online services. Today, several distributed applications in the fields of e-commerce, financial industries, search engines, biomedical informatics, etc., have been developed and deployed in such datacenters. However, these applications tend to have a complex design due to concurrency, synchronization and communication issues. Existing mechanisms proposed to hide these complexities are both inefficient in terms of performance and scalability, and non-resilient to loaded conditions in the datacenter. In addition, existing mechanisms do not take complete advantage of the features of emerging technologies which are gaining momentum in current datacenters.

In this dissertation, we proposed an efficient soft state sharing substrate that hides the above mentioned complexities and addresses the limitations in existing state sharing mechanism by leveraging the features of emerging technologies such as high-speed networks, Intel's I/OAT and multicore architectures. The proposed substrate improves the performance and scalability of several datacenter applications

and services. Figure 9.1 shows the various components designed as a part of this dissertation. In the figure, all the dark colored boxes are the technologies which exist today. The light colored boxes are the ones that are developed as a part of this dissertation for designing a high-performance soft shared state for datacenters.

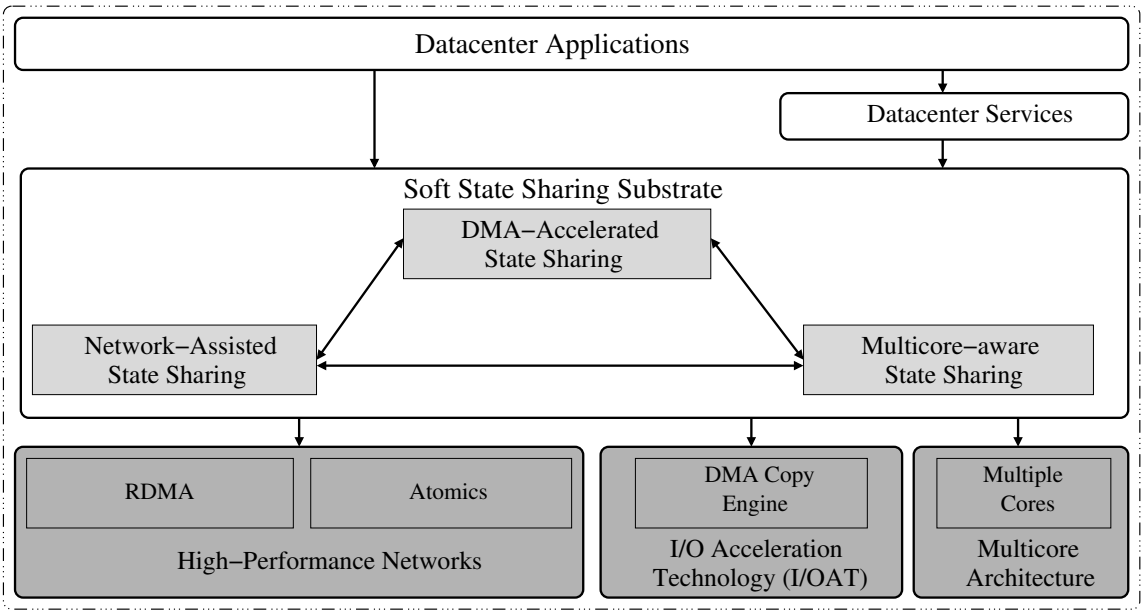


Figure 9.1: State Sharing Components

9.1 Summary of Research Contributions

We summarize the research contributions of this dissertation along three important aspects: (i) designing efficient state sharing components using the features of emerging technologies, (ii) understanding the interactions between the proposed components and (iii) analyzing the impact of the proposed components and its interactions with several datacenter applications and services.

9.1.1 State Sharing Components using Emerging Technologies

We leveraged the features of emerging technologies and proposed three components: (a) network-assisted state sharing using high-speed networks, (b) DMA-accelerated state sharing using Intel's I/O Acceleration Technology and (c) multicore-aware state sharing using multicore architectures.

Network-Assisted State Sharing: In Chapter 2, we utilized the features of high-speed networks such as RDMA and atomic operations to address the limitations associated with state sharing based on TCP/IP and improve the performance and scalability of datacenter applications and services. We demonstrated the portability of the proposed substrate across multiple modern interconnects including iWARP-capable networks both in LAN and WAN environments. Evaluations with multiple networks not only showed an order of magnitude performance improvement but also demonstrated that the proposed substrate is scalable and has a low overhead.

DMA-Accelerated State Sharing: As mentioned in Chapter 3, the proposed substrate leveraged the asynchronous memory copy engine feature of I/OAT to address the limitations associated with System V IPC mechanisms and improve the performance and CPU utilization of memory copy operations in datacenter environments. In addition, the proposed substrate allows the capability of utilizing the system cache intelligently.

Multicore-aware State Sharing: Multicore systems addresses the limitations of existing state sharing mechanisms by dedicating the protocol processing, memory copies to one or more cores. In addition, this approach also addressed other issues such as user-managed memory regions, complex NIC hardware, the notion of physical

memory, etc., by using the software-centric onloading approach to manage the memory regions. The proposed architecture, as mentioned in Chapter 4, provides access to objects in unpinned virtual memory, simplifies the NIC hardware, and ensures robust system behavior.

9.1.2 Interaction of Proposed State Sharing Components

We further enhanced the proposed state sharing components for systems with multiple of these emerging technologies and analyzed the impact with datacenter applications and services.

Multicore-aware, DMA-Accelerated State Sharing: We thoroughly investigated, designed and analyzed the several combinations of memory copy operations using multicore systems and I/OAT in Chapter 5. We further proposed a mechanism for an application-transparent asynchronous state sharing using memory protection. Our evaluations with datacenter services and SPEC benchmarks showed a lot of potential in several application domains.

Multicore-aware, Network-Assisted State Sharing: We presented several design optimizations in state sharing with multi-core systems and high-speed networks in Chapter 6 and demonstrated the benefits in terms of performance, scalability and associated overheads using several applications such as Distributed STORM, R-Tree and B-Tree query processing, checkpointing applications and resource monitoring services. Our evaluations using an additional core for state sharing services showed a lot of potential benefits for performing services on dedicated cores.

Network-Assisted, DMA-Accelerated State Sharing: We demonstrated the benefits of I/OAT in network communication through several receiver side optimizations in Chapter 7. Our evaluations not only increased the performance but also improved the scalability of datacenter applications.

In Chapter 8, we highlighted several scenarios and system environments where each of the state sharing components can be applicable.

9.2 Future Research Directions

Advances in processor and networking technologies continue to provide additional capabilities for emerging datacenters. In this dissertation, we have utilized these capabilities in proposing a state sharing substrate for datacenters. Though we have proposed several components and studied its interactions, there are many interesting research topics to pursue in this area. We present some of the research topics below.

9.2.1 Low-overhead Communication and Synchronization Service

As the number of cores increase within the system up to 64 or 128 cores, a sequential execution of tasks is not going to get the benefits from pipelining, branch predictors, caches, etc. Researchers are moving towards component-based model and specializing the cores for different tasks such as communication, protocol handling, logging, etc. The proposed state sharing service can be extended as a low-overhead communication and synchronization service for efficient interaction of these specialized components and ultimately improve the performance and scalability of the datacenter system software.

9.2.2 Dedicated Datacenter Functionalities

As a part of this dissertation, we have taken a first step in looking at dedicating a core for tasks such as resource monitoring, memory copies and RDMA. Within a few years, we are going to see a large number of cores within a system and researchers are going to think of dedicating cores for specialized functionalities such as communication, synchronization, collective operations and other application-specific operations such as queue insertions, deletions and and modifications. The ideas proposed in this dissertation can be extended to understand, design and analyze the impact of such functionalities in several environments.

9.2.3 Integrated Evaluation

While we have shown the benefits of state sharing for systems with at least two of these technologies, it is also important to analyze the design optimizations and the performance benefits achievable for systems with all three technologies. For example, the DMA-accelerated state sharing component proposed in Chapter 3 can help in accelerating the data copy operations of multicore-aware, network-assisted state sharing proposed in Chapter 6. Thus, it is crucial to evaluate the interactions of these components in an integrated manner.

9.2.4 Redesigning Datacenter System Software

Using the current designs of state sharing, several existing datacenter system software such as distributed lock manager in databases, communication and synchronizations mechanisms in distributed file systems, global memory aggregator, etc., can be built on top of the state sharing service. Several applications and services which require more memory can use the state sharing substrate as a collection of remote

memory within the cluster and use the cluster memory efficiently. This can be extended for other datacenter system software such as distributed databases, group communication services and several others.

9.2.5 Extending State Sharing Designs for HPC Environments

The designs proposed in this dissertation are also applicable to HPC environments. Preliminary designs and evaluations with MPI, a parallel programming model for high-performance computing environment, shows a lot of promise for extending our designs for other MPI operations such as collective communication, asynchronous communication progress, one-sided communication and synchronization and several others.

9.2.6 Impact of State Sharing in Virtualization Environments

Server consolidation using existing virtualization techniques is currently increasing, thus requiring efficient ways of consolidating the system resources in a fine-grained manner for managing the datacenter resources. The proposed state sharing designs can be utilized for monitoring and efficiently sharing the system information in a virtualized environment and can be used for several important designs such as migration, increased/decreased allocation of system resources and availability.

BIBLIOGRAPHY

[1] Accelerating High-Speed Networking with Intel I/O Acceleration Technology. http://www.intel.com/technology/ioacceleration/306517.pdf.

[2] Amazon Now is A Datacenter. http://www.heyotwell.com/heyblog/archives/2006/03/amazon_is_now_a_1.html.

[3] Ammasso, inc. http://www.ammasso.com.

[4] Ask Jeeves . http://www.ask.com.

[5] Berkeley UPC project home page. http://upc.lbl.gov.

[6] Breaking through the Bottleneck. http://www.voltaire.com.

[7] E-trade. http://us.etrade.com/.

[8] eBay imparts datacenter knowlege. http://www.infoworld.com/article/07/08/07/Ebaykeynote_1.html.

[9] Google Data Center. http://www.datacenterknowledge.com/archives/2008/Mar/27/google_data_center_faq.html.

[10] HP Data Center. http://www.hp.com/hpinfo/newsroom/press_kits/2008/datacenter-transformation/.

[11] IBM Data Center. http://www.ibm.com/services/us/index.wss/offering/its/a1000013.

[12] Increasing network speeds: Technology@intel. http://www.intel.com/technology/magazine/communications/intel-ioat-0305.htm.

[13] Intel I/O Acceleration Technology. http://www.intel.com/technology/ioacceleration/306484.pdf.

[14] IP over InfiniBand Working Group. http://www.ietf.org/html.charters/ipoib-charter.html.

[15] MSN. http://www.msn.com/.

[16] MySQL - Open Source Database Server. http://www.mysql.com/.

[17] NASDAQ Stock Market . http://www.nasdaq.com.

[18] Quadrics Supercomputers World Ltd. http://www.quadrics.com/.

[19] RDMA Consortium. http://www.rdmaconsortium.org/home.

[20] RUBiS: Rice University Bidding System. http://rubis.objectweb.org.

[21] Scottrade . http://www.scottrade.com.

[22] SPEC CPU2000 benchmarks. http://www.spec.org/cpu2000/.

[23] The GiST Indexing Project. http://gist.cs.berkeley.edu/.

[24] The STORM Project at OSU BMI. http://storm.bmi.ohio-state.edu/index.php.

[25] Yahoo Data Center. http://www.datacenterknowledge.com/archives/yahoo-index.html.

[26] Yahoo! Finance. http://finance.yahoo.com/.

[27] USNA, TTCP: A test of TCP and UDP Performance, 2001.

[28] M. K. Aguilera, C. Karamanolis, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[29] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikey, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck Characterization of Dynamic Web Site Benchmarks. In *Technical Report TR02-391,Rice University*, 2002.

[30] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd Intl. Symp. on High Performance Computer Architecture,*, 1997.

[31] P. Balaji, K. Vaidyanathan, S. Narravula, K. Savitha, H. W. Jin, and D. K. Panda. Exploiting Remote Memory Operations to Design Efficient Reconfiguration for Shared Data-Centers. In *Workshop on Remote Direct Memory Access: Applications, Implementations, and Technologies (RAIT)*, 2004.

[32] R. Bayer and C. Mccreight. Organization and Maintenance of large ordered indexes. In *Acta Informatica*, 1972.

[33] C. Bell and D. Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *Proceedings of International Workshop on Communication Architecture for Clusters*, 2003.

[34] TPC-W Benchmark. http://www.tpc.org.

[35] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*, 1996.

[36] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[37] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM (1)*, pages 126–134, 1999.

[38] D. Buntinas, G. Mercier, and W. Gropp. The Design and Evaluation of Nemesis, a Scalable Low-Latency Message-Passing Communication Subsystem. In *Proceedings of International Symposium on Cluster Computing and the Grid (CCGrid)*, 2006.

[39] Michael Calhoun. Characterization of Block Memory Operations. In *Masters Thesis, Rice University*, 2006.

[40] J. Carlstrom and R. Rom. Application-Aware Admission Control and Scheduling in Web Servers. In *Proceedings of the IEEE Infocom 2002*, 2002.

[41] E. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming, Snowbird, UT*, 2001.

[42] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Services. In *International Conference on Distributed Computing Systems (ICDCS)*, 1998.

[43] L. Chai, A. Hartono, and D. K. Panda. Designing Efficient MPI Intra-node Communication Support for Modern Computer Architectures. In *IEEE International Conference on Cluster Computing*, 2006.

[44] L. Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *IEEE International Conference on Cluster Computing*, 2006.

[45] J. Chase, A. Gallatin, and K. Yocum. End System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[46] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centres. In *Symposium on Operating Systems Principles*, 2001.

[47] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. InterWeave: A Middleware System for Distributed Shared State. In *LCR*, 2000.

[48] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M. Scott. Exploiting High-level Coherence Information to Optimize Distributed Shared State. In *Proceedings of the 9th ACM Symp. on Principles and Practice of Parallel Programming*, 2003.

[49] Giuseppe Ciaccio. Using a Self-connected Gigabit Ethernet Adapter as a memcpy() Low-Overhead Engine for MPI. In *Euro PVM/MPI*, 2003.

[50] J. R. David D. Clark, Van Jacobson, and H. Salwen. An analysis of TCP processing overhead, 1989.

[51] Intel Corporation. http://www.vnunet.com/vnunet/news/2165072/inte l-unveils-tera-scale, 2006.

[52] Squid: Optimising Web Delivery. URL: http://www.squid-cache.org/.

[53] Annie Foong et al. TCP performance analysis revisited. In *IEEE International Symposium on Performance Analysis of Software and Systems*, 2003.

[54] The Apache Software Foundation. URL: http://www.apache.org/.

[55] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Symposium on Operating Systems Principles*, 1997.

[56] R. Friedman. Implementing Hybrid Consistency with High-Level Synchronization Operations. In *In Proceedings of 13th ACM Symp on Principles of Distributed Computing*, 1993.

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, 2003.

[58] Andrew Gover and Christopher Leech. Accelerating Network Receiver Processing. http://linux.inet.hr/files/ols2005/grover-reprint.pdf.

[59] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings on ACM SIGMOD Conference*, 1984.

[60] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro*, 2004.

[61] Infiniband Trade Association. http://www.infinibandta.org.

[62] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster. In *International Conference on Parallel Processing (ICPP)*, 2005.

[63] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster. In *International Conference on Parallel Processing*, 2005.

[64] P. Lai, S. Narravula, K. Vaidyanathan, and Dhabaleswar K. Panda. Advanced RDMA-based Admission Control for Modern Data-Centers. In *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2008.

[65] S. Lee, J. Lui, and D. Yau. Admission control and dynamic adaptation for a proportionaldelay diffserv-enabled web server. In *Proceedings of SIGMETRICS*, 2002.

[66] Jochen Liedtke, Volkmar Uhlig, Kevin Elphinstone, Trent Jaeger, and Yoonho Park. How To Schedule Unlimited Memory Pinning of Untrusted Processes Or Provisional Ideas about Service-Neutrality. In *Workshop on Hot Topics in Operating Systems*, 1999.

[67] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference*, 2006.

[68] S. Makineni and R. Iyer. Architectural characterization of TCP/IP packet processing on the PentiumM microprocessor. In *High Performance Computer Architecture, HPCA-10*, 2004.

[69] Evangelos P. Markatos and Manolis G. H. Katevenis. User-Level DMA without Operating System Kernel Modification. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture, (HPCA)*, 1997.

[70] S. Narayanan, T. Kurc, U. Catalyurek, and J. Saltz. Database Support for Data-driven Scientific Applications in the Grid. In *Parallel Processing Letters*, 2003.

[71] S. Narravula, P. Balaji, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand. In *Workshop on System Area Networks*, 2004.

[72] S. Narravula, H.-W. Jin, K. Vaidyanathan, and D. K. Panda. Designing Efficient Cooperative Caching Schemes for Multi-Tier Data-Centers over RDMA–enabled Networks. In *Proceedings of International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2006.

[73] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.

[74] J. Nieplocha and J. Ju. ARMCI: A Portable Aggregate Remote Memory Copy Interface. In *IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS)*, 1999.

[75] R. Numrich and J. Reid. Co-array fortran for parallel programming, 1998.

[76] Joon Suan Ong. Network Virtual Memory. PhD. Thesis, The University of British Columbia, 2003.

[77] OpenFabrics Alliance. OpenFabrics. http://www.openfabrics.org/.

[78] O. Othman, J. Balasubramanian, and D.C. Schmidt. The Design of an Adaptive Middleware Load Balancing and Monitoring Service . In *Proceedings of the Third International Workshop on Self-Adaptive Software (IWSAS)*, 2003.

[79] S. Parthasarathy and S. Dwarkadas. InterAct: Virtual Sharing for Interactive Client-Server Application. In *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

[80] G. Porter and R. H. Kaltz. Effective Web Service Loadbalancing through Statistical Monitoring. In *SelfMan 2005, IFIP/IEEE International Workshop on Self-Managed Systems and Services*, 2005.

[81] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. In *IEEE Computer*, 2004.

[82] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Symposium on Operating Systems Principles*, 1999.

[83] M. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell, L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, and N. Binkert and N. P. Jouppi. High-performance ethernet-based communications for future multi-core processors. In *International Conference on High Performance Computing, Networking, Storage and Analysis (Super Computing)*, 2007.

[84] IBM DB2 Database Server. URL: http://www.ibm.com/software/data/db2/.

[85] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, R. S. Madukkarumukumana, and G. J. Regnier. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *USITS '01*.

[86] H. V. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct Data Placement over Reliable Transports, 2002.

[87] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2002.

[88] C. Tang, D. Chen, S. Dwarkadas, and M. Scott. Integrating Remote Invocation and Distributed Shared State, 2004.

[89] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Efficient Distributed Shared State for Heterogeneous Machine Architectures. In *International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[90] Introduction to Java RMI. URL: http://www.javacoffeebreak.com/articles/javarmi/javarmi.html.

[91] K. Vaidyanathan, P. Balaji, S. Narravula, and H. W. Jinand D. K. Panda. Designing Efficient Systems Services and Primitives for Next-Generation Data-Centers. In *NSF Next Generation Software(NGS) Program;*, 2007.

[92] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT. In *IEEE International Conference on Cluster Computing*, 2007.

[93] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *Proceedings of International Workshop on Communication Architecture for Clusters (CAC)*, 2007.

[94] K. Vaidyanathan, H. W. Jin, and D. K. Panda. Exploiting RDMA operations for Providing Efficient Fine-Grained Resource Monitoring in Cluster-based Servers. In *Workshop on Remote Direct Memory Access: Applications, Implementations, and Technologies (RAIT)*, 2006.

[95] K. Vaidyanathan, P. Lai, S. Narravula, and D. K. Panda. Benefits of Dedicating Resource Sharing Services for Data-Centers using Emerging Multi-Core Systems. In *Technical Report OSU-CISRC-8/07-TR53, The Ohio State University*, 2007.

[96] K. Vaidyanathan, P. Lai, S. Narravula, and Dhabaleswar K. Panda. Optimized Distributed Data Sharing Substrate in Multi-Core Commodity Clusters: A Comprehensive Study with Applications. In *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2008.

[97] K. Vaidyanathan, S. Narravula, and D. K. Panda. DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over Modern Interconnects. In *International Conference on High Performance Computing (HiPC)*, 2006.

[98] K. Vaidyanathan and D. K. Panda. Benefits of I/O Acceleration Technology (I/OAT) in Clusters. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[99] K. Vaidyanathan, M. Schlansker, J. Mudigonda, N. Binkert, and D. K. Panda. RDMA Service using Dynamic Page Pinning: An Onloading Approach, 2008.

[100] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *International Conference on Parallel Processing (ICPP)*, 2003.

[101] Pete Wyckoff. Memory Registration Caching Correctness. In *Proceedings of International Symposium on Cluster Computing and the Grid (CCGrid)*, 2005.

[102] Youhui Zhang and Weimin Zheng. User-level Communication based Cooperative Caching. In *ACM Special Interest Group on Operating Systems (SIGOPS)*, 2003.

[103] Li Zhao, Ravi Iyer, Srihari Makineni, Laxmi Bhuyan, and Don Newell. Hardware Support for Bulk Data Movement in Server Platforms. In *Proceedings of International Conference on Computer Design*, 2005.

[104] George Kingsley Zipf. Human Behavior and the Principle of Least Effort. Addison-Wesley Press, 1949.