Optimized Distributed Data Sharing Substrate in Multi-Core Commodity Clusters: A Comprehensive Study with Applications*

K. Vaidyanathan P. Lai S. Narravula D. K. Panda
Department of Computer Science and Engineering
The Ohio State University
{vaidyana, laipi, narravul, panda}@cse.ohio-state.edu

Abstract

Distributed applications tend to have a complex design due to issues such as concurrency, synchronization and communication. Researchers in the past have proposed simpler abstractions to hide these complexities. However, many of the proposed techniques use messaging protocols which incur high overhead and are not very scalable. To address these limitations, in our previous work [20], we proposed an efficient Distributed Data Sharing Substrate (DDSS) using the features of high-speed networks. In this paper, we propose several design optimizations for DDSS in multi-core systems such as the combination of shared memory and message queues for inter-process communication, dedicated thread for communication progress and for onloading DDSS operations such as get and put. Our micro-benchmark results not only show a very low latency in DDSS operations but also demonstrate the scalability of DDSS with increasing number of processes. Application evaluations with R-Tree and B-Tree query processing and distributed STORM shows an improvement of up to 56%, 45% and 44%, respectively, as compared to traditional implementations. Evaluations with application checkpointing using DDSS demonstrate the scalability with increasing number of checkpointing applications. Further, in our evaluations, we demonstrate the portability of DDSS across multiple modern interconnects including InfiniBand and iWARP-capable 10-Gigabit Ethernet networks (applicable for both LAN/WAN environments).

1 Introduction

Cluster systems consisting of commodity off-the-shelf hardware components are becoming increasingly attractive as platforms for distributed applications, primarily due to their high performance-to-cost ratio. Today, several distributed applications such as STORM [4], database query processing [3, 2] and services [18] such as caching, load-balancing and resource monitoring have been developed and deployed in such systems to not only enable sharing the datasets that these applications generate but also to improve the performance and scalability.

Unfortunately, these applications tend to have a complex design since they explicitly try to deal with issues such as concurrency, communication and synchronization. Further, for the sake of availability and performance, programmers use ad-hoc messaging protocols for communication and synchronization. As mentioned in [17], the code devoted to these protocols accounts for a significant fraction of overall application size and complexity. Researchers in the past [6, 9, 16, 13] have proposed a simpler way to build these applications that hides these complexities through data sharing primitives such as read/write, begin/end transactions and notifications. However, these mechanisms were built on top of communication protocols like TCP/IP that is known to have overheads such as protocol processing, memory copies, context-switches and remote CPU involvement.

On the other hand, System Area Networks (SAN) such as InfiniBand (IBA) [7] and iWARP [14] capable 10-Gigabit Ethernet (10-GigE) adapters [1] are currently gaining momentum in both LAN/WAN environments. Besides high performance, these modern interconnects also provide a range of novel features such as Remote Direct Memory Access (RDMA), Protocol Offload and several others. Accordingly, in our previous work [20], we proposed a Distributed Data Sharing Substrate (DDSS) using these features to address the limitations of TCP/IP-based data sharing primitives.

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-

*This research is supported in part by DOE grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342 and #CNS-0509452; grants from Intel, Mellanox, Cisco systems, Linux Networx and Sun Microsystems; and equipment donations from Intel, Mellanox, AMD, Apple, Appro, Dell, Microway, PathScale, IBM, SilverStorm and Sun Microsystems.

threading (SMT), also known as hyper-threading, to support large number of concurrent thread executions in the system. While the main idea of these systems is to provide multiple processing elements to function in parallel, it also opens up new ways to design and optimize existing middleware such as DDSS. Further, as the number of cores and threads multiply, one or more of these cores can also be dedicated to perform specialized services.

In this paper, we propose several design optimizations for DDSS in multi-core systems such as the combination of shared memory and message queues for inter-process communication, dedicated thread for communication progress and for onloading other DDSS operations such as *get* and *put*. Our micro-benchmark results not only show a very low latency in DDSS operations but also demonstrate the scalability of DDSS with increasing number of processes. Application evaluations with R-Tree and B-Tree query processing and distributed STORM shows an improvement of up to 56%, 45% and 44%, respectively, as compared to traditional implementations. Evaluations with application checkpointing demonstrate the scalability of DDSS. Further, in our evaluations, we demonstrate 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 show a lot of potential benefits for performing these services on dedicated cores.

The remaining part of the paper is organized as follows: Section 2 provides a background on high-speed interconnects and the functionalities of DDSS. In Section 3, we propose our design optimizations in a multi-core system. Section 4 deals with the evaluation of DDSS using micro-benchmarks and applications. In Section 5, we discuss some of the issues involved in dedicating the functionalities of DDSS to multiple cores and we conclude the paper in Section 6.

2 Background

In this section, we provide a brief background on high-speed networks and the features of DDSS.

2.1 High-Speed Interconnects

As mentioned earlier, there are several high-speed interconnects such as InfiniBand, 10-Gigabit Ethernet, Myrinet and Quadrics that are currently deployed in large-scale clusters. The InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clusters within a LAN environment offering very low latency and high bandwidth while 10-Gigabit Ethernet adapters offer high-performance for both LAN/WAN environments. Apart from high-performance, both the adapters support several advanced features such as remote memory operations (performing a remote read/write operation without interrupting the remote CPU), protocol offload and several others. Recently OpenFabrics [12] has been proposed as the standard interface which allows a common API that is portable across multiple interconnects such as IBA and 10-GigE. We use this standard interface and demonstrate the portability on multiple interconnects.

2.2 Distributed Data Sharing Substrate

The basic idea of DDSS [20] is to allow efficient sharing of information across the cluster by creating a logical shared memory region. DDSS supports operations such as a *get* operation to read a shared data segment, a *put* operation to write onto a shared data segment using the advanced features of high-speed interconnects. Figure 1 shows a simple distributed data sharing scenario with several processes (back-end servers) writing and several front-end servers reading certain information from the shared environment simultaneously. DDSS also supports several other features such as basic locking services, several coherence models, versioning of data and timestamps using the advanced features of high-speed interconnects. Applications can create and destroy memory segments using *allocate* and *release* operations. As shown in the figure, DDSS uses System V message queues for sending and receiving messages (shown as *ipc_send* and *ipc_recv* in Figure 1). The readers are encouraged to refer to our previous work [20] for more details on the design of DDSS.

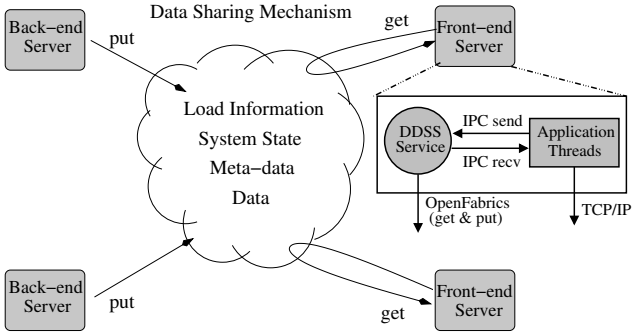


Figure 1. DDSS

3 Design Optimizations

In this section, we first present our existing Message Queue-based DDSS (MQ-DDSS) design. Next, we present two design optimizations for multi-core systems, namely, (i) Request/Message Queue-based DDSS (RMQ-DDSS) and (ii) Request/Completion Queue-based DDSS (RCQ-DDSS).

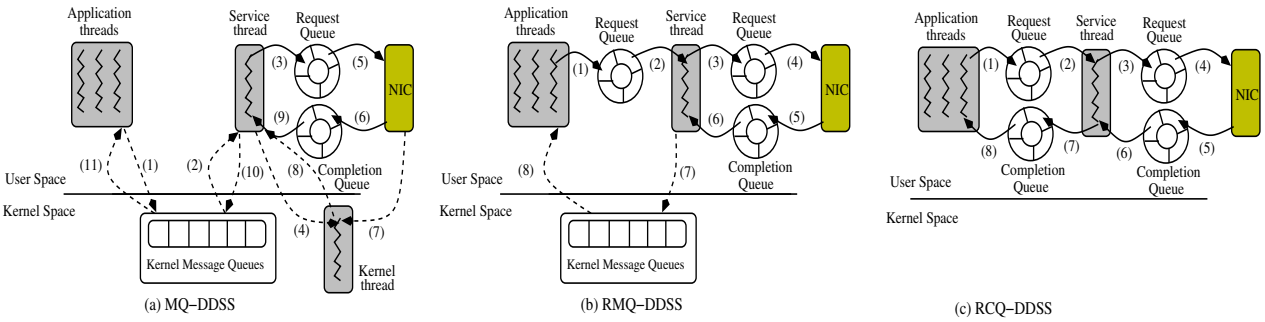


Figure 2. Design Optimizations in DDSS

3.1 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 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 2(a)) and thus may lead to degradation in performance.

3.2 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 2(b)) for inter-process communication. In this approach, the reply messages still follow the 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.

3.3 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 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 [15]. 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.

4 Experimental Results

In this section, 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 [12], OFED 1.2 release.

4.1 Micro-benchmarks

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

4.1.1 DDSS Latency

First, we measure the performance of inter-process communication (IPC) using the different approaches mentioned in Section 3. 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 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 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 3(c).

4.1.2 DDSS Scalability

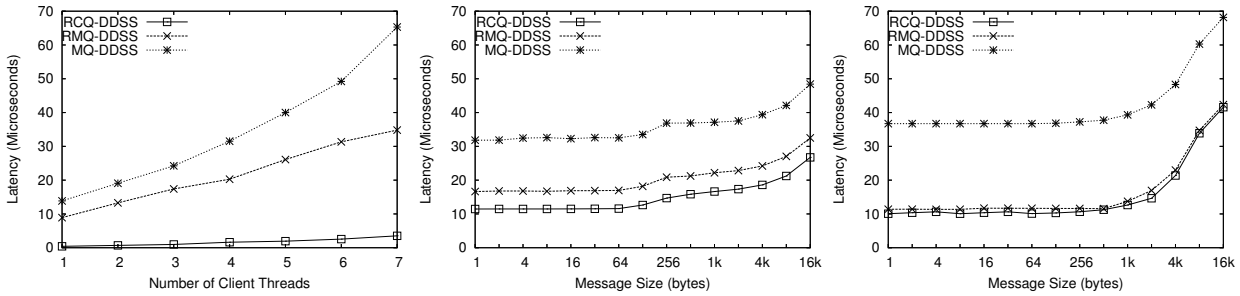
Here, we measure the scalability of DDSS with increasing number of processes performing the DDSS operations over IBA. First, we stress the inter-process communication and show the performance of the three approaches, as shown in Figure 4(a). We observe that, for very large number of client threads (up 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 (shown as DDSS-(non-distributed) in the Figure 4(b)). We compare its performance by distributed data across different nodes in the cluster and show its scalability. As shown in Figure 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.

4.2 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.

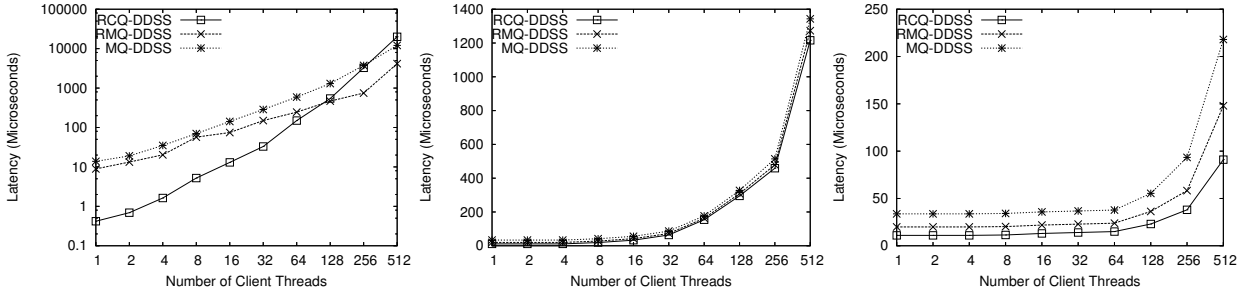
4.2.1 R-Tree Query Processing

R-Tree [10] 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, bioinformatics and computer vision. In our experiments, we use an R-tree query processing application, developed by Berkeley [3] 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



(a) IPC Latency (b) Performance of *get()* in IBA (c) Performance of *get()* in 10-GigE

Figure 3. DDSS Latency



(a) IPC Latency (b) Performance of *get()* for single key (c) Performance of *get()* for distributed keys

Figure 4. DDSS Scalability

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 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.

4.2.2 B-Tree Query Processing

B-Tree [8] 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 [3] 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 1 shows the overall execution time of a B-Tree application with varying percentage of queries. As shown in table1, 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.

4.2.3 Distributed STORM

STORM [11, 4] 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 [20], 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 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.

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 5. The performance benefits achieved in an R-Tree query processing application is shown in Figure 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 5(b) and 5(c).

4.2.4 Application Checkpointing

Here, we present our evaluations with an application checkpointing benchmark [20] 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%).

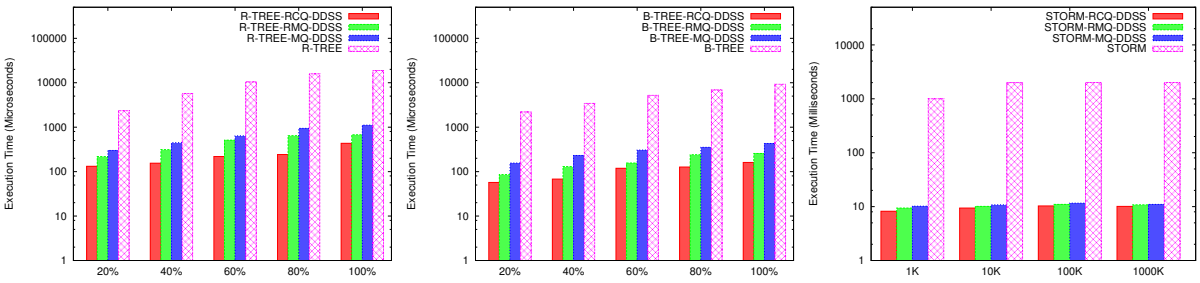
Figure 6(a) shows the performance of checkpoint applications with increasing number of processes within the same node. We observe that the performance of the RCQ-DDSS approach scales with increasing number of application processes for up to 16. However, for large number of processes up to 64, we see that the RMQ-DDSS approach performs better which confirms the results shown in Section 4.1.2. Figure 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(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 4.1.2. 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(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 4.1.2.

4.2.5 DDSS Services on Additional Cores

Several existing services such as resource monitoring, caching, distributed lock manager can be built on top of DDSS. 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 [18] 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 different experimental testbed (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) as we did not see any appreciable difference using the 8-core testbed. In the 48-node experimental testbed, we use one CPU for both the proxy server and the DDSS service and blocked the other CPU with dummy computations. Figure 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 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 and different monitoring granularities and see similar trends. These results can be found in [19].

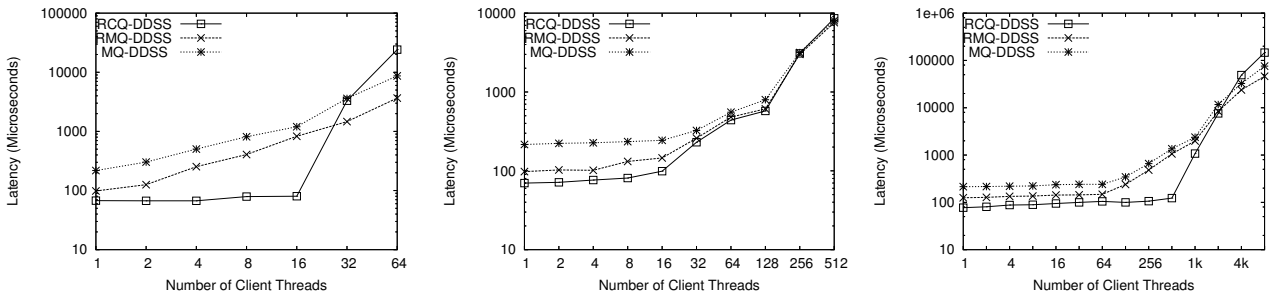
Table 1. Application Performance

Application		No. of DDSS Operations	Average 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



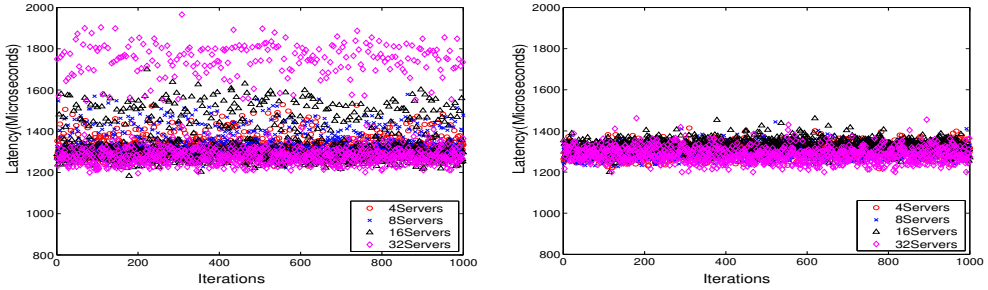
(a) R-Tree Query Processing (b) B-Tree Query Processing (c) Meta-data in Distributed STORM

Figure 5. Data Sharing Performance in Applications



(a) Clients within a single node (non-distributed) (b) Clients across different nodes (non-distributed) (c) Clients across different nodes (distributed)

Figure 6. Checkpoint Application Performance



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

Figure 7. Performance impact of DDSS on Web Servers

5 Discussion and Related Work

Modern processors are seeing a steep increase in the number of cores available in the system [5]. 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 [16], Khazana [9], InterAct [13] and Sinfonia [6]. 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 [20]. 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 Conclusions

In this paper, we presented design optimizations in DDSS for multi-core systems 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.

7 Acknowledgements

We would like to thank Sivaramakrishnan Narayanan for providing us with the several significant details about the Distributed STORM application.

References

[1] Chelsio communications. http://www.chelsio.com/.

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

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

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

[5] Intel corporation. http://www.vnunet.com/vnunet/news/2165072/intel-unveils-tera-scale, 2006.

[6] 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.

[7] I. T. Association. http://www.infinibandta.org.

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

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

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

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

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

[13] 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.

[14] A. Romanow and S. Bailey. An Overview of RDMA over IP. In *International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet)*, 2003.

[15] M. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell, L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, and N. P. Jouppi. High-performance Ethernet-based Communications for Future Multi-core Processors. In *Super Computing*, 2007.

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

[17] C. Tang, D. Chen, S. Dwarkadas, and M. Scott. Integrating Remote Invocation and Distributed Shared State. In *the Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[18] 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.

[19] 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.

[20] 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.