DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over Modern Interconnects

K. VAIDYANATHAN, S. NARRAVULA AND D. K. PANDA

Technical Report
Ohio State University (OSU-CISRC-1/06-TR06)

DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over Modern Interconnects*†

K. Vaidyanathan S. Narravula D. K. Panda

Dept. of Computer Science and Engineering
The Ohio State University
{vaidyana, narravul, panda}@cse.ohio-state.edu

Abstract

Information-sharing is a key aspect of distributed applications such as database servers, application servers, web servers, etc., in cluster-based web data-centers. Information-sharing also assists services such as load-balancing, dynamic caching, reconfiguration, etc. In the past, information-sharing has been implemented using ad-hoc messaging protocols which often incur high overheads and are not very scalable. This paper presents a new design for a scalable and a low-overhead *distributed data sharing substrate* (DDSS). DDSS is designed to perform efficient data and memory management and supports a variety of coherence models by leveraging the features of modern interconnects like one-sided communication and atomic operations. It 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 emerging networks like InfiniBand and iWARP-capable Ammasso networks through micro-benchmarks and data-center services such as reconfiguration and active caching not only show an order of magnitude performance improvement over traditional implementations but also show the load resilient nature of the substrate. Application-level evaluations with Distributed STORM using DataCutter achieves close to 19% performance improvement over traditional implementation, while evaluations with check-pointing application suggest that DDSS is scalable and has a low overhead.

1 Introduction

Distributed applications in the fields of nuclear research, biomedical informatics, satellite weather image analysis etc., are increasingly getting deployed in cluster environments due to their high computing demands. Advances in technology have facilitated the storing and sharing of the large

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #CNS-0403342 and#CNS-0509452; grants from Intel, Mellanox, Sun Microsystems and Linux Networx; and equipment donations from Intel, Mellanox and Silverstorm.

†We would like to thank Sivaramakrishnan Narayanan for providing us with the several significant details about the Distributed STORM application and helping us tremendously in our evaluations.

datasets that these applications generate, typically through a web interface forming web data-centers [25, 15, 10, 24]. A web-based data-center environment (illustrated in Figure 1) comprises of multiple tiers; the first tier consists of front-end servers such as the proxy servers that provide web, messaging and services like caching and load balancing to clients; the middle tier comprises of application servers that handle transaction processing and implement business logic, while the back-end tier consists of database servers that hold a persistent state of the databases and other data repositories. In addition to hosting these distributed applications, current data-centers also need ef-

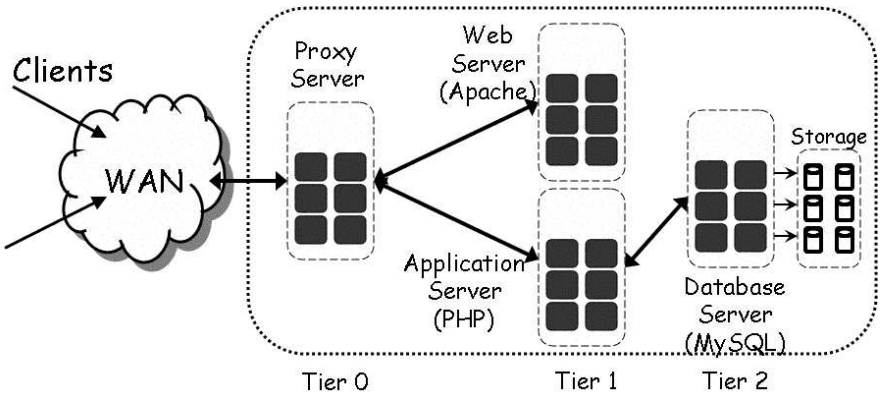

Figure 1: Web-based data-centers

ficient and scalable support for intelligent services like dynamic caching of documents, resource management, load-balancing, etc. Apart from communication and synchronization, these applications and services exchange some key information at multiple sites (e.g, versioning and timestamps of cached copies, coherency and consistency information, current system load). However, for the sake of availability, high-performance and low-latency, programmers use ad-hoc messaging protocols for maintaining this shared information. Unfortunately, as mentioned in [27], the code devoted to these protocols accounts for a significant fraction of overall application size and complexity. As system sizes increase, this fraction is likely to increase and cause significant overheads. Further, the performance of these protocols can degrade in the presence of load imbalances.

On the other hand, System Area Network (SAN) technology has been making rapid progress during the recent years. SAN interconnects such as InfiniBand (IBA) [2] and 10-Gigabit Ethernet (10GigE) [17, 14] have been introduced and are currently gaining momentum for designing high-end computing systems and data-centers. Besides high performance, these modern interconnects provide a range of novel features and their support in hardware, e.g., Remote Direct Memory Access (RDMA), Remote Atomic Operations, Offloaded Protocol support and several others. Re-

cently OpenFabrics [22] has been proposed as the standard interface that allows for a generic implementation to be portable over several modern interconnects such as IBA, 10-GigE including iWARP-capable [26] networks such as Ammasso [1] both in LAN and WAN environments.

In this paper, we design and develop a low-overhead distributed data sharing substrate (DDSS) that allows efficient sharing of data among independently deployed servers in data-centers by leveraging the features of the SAN interconnects. DDSS is designed to perform efficient data and memory management and supports a variety of coherence models by leveraging the features of modern interconnects like one-sided communication and atomic operations. Specifically, DDSS offers several coherency models ranging from null coherency to strict coherency model. In addition, the substrate provides basic features for locking mechanisms by utilizing the atomic operations, several data management and data distribution techniques, etc.

Experimental evaluations with IBA and iWARP-capable Ammasso networks through micro-benchmarks and data-center services such as reconfiguration and active caching not only show an order of magnitude performance improvement over traditional implementations but also show the load resilient nature of the substrate. Application-level evaluations with Distributed STORM using DataCutter achieves close to 19% performance improvement over traditional implementation, while evaluations with check-pointing application suggest that DDSS is scalable and has a low overhead. The proposed substrate is implemented over the OpenFabrics standard interface and hence is portable across multiple modern interconnects.

The rest of the paper is organized as follows. Section 2 provides a brief background on the capabilities of modern interconnects. The basic requirements of several applications and services for DDSS are mentioned in Section 3 and design goals are mentioned in Section 4. The design and implementation of DDSS is discussed in Section 5. Section 6 presents the experimental results. Related work is given in Section 7 and Section 8 presents our conclusions and future work.

2 Background

This section describes the two salient features of modern interconnects: (i) Remote Direct Memory Access (RDMA) and (ii) Atomic Operations, that are used in designing DDSS.

2.1 Remote Direct Memory Access

Modern interconnects such as InfiniBand (IBA), 10-Gigabit Ethernet, etc., provide two types of communication semantics: channel semantics (send/receive communication model) and memory

semantics (one-sided communication model). In channel semantics, every send request has a corresponding receive request at the remote end. On the other hand, memory semantics follows a one-sided communication model. Here, Remote Direct Memory Access (RDMA) operations are used, which allow the initiating node to directly access the memory of the remote-node without the involvement of the remote-side CPU. RDMA operations are allowed only on pinned memory locations thus securing the remote node from accessing any arbitrary memory location. 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 memory, while in an RDMA Read operation, the initiator reads data from the remote node's memory.

2.2 Atomic Operations over IBA

In addition to RDMA, the reliable communication classes over IBA optionally include atomic operations [3] which can be performed directly on remote memory without the involvement of the remote CPU. Atomic operations are posted as descriptors similar to any other type of communication. However, the operation is completely handled by the network adapter. 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, while the Compare-and-Swap compares two 64-bit values and swaps the remote value with the data provided if the comparison succeeds.

OpenFabrics [22] interface has been proposed recently as a standard for several modern interconnects. We propose to use this interface for implementing DDSS to ensure portability across different interconnects.

3 Constraints of Data-Center Applications

This section describes the necessary characteristics and features to be provided by the DDSS both in the context of applications as well as services that are deployed in data-centers.

3.1 Requirements of Data-Center Servers

Existing data-center applications such as Apache, PHP, MySQL, DB2, 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 based on current system load, request patterns, etc. To provide fault-tolerance, check-pointing applications that save the program state at regular intervals are also

extensively used. Many of these mechanisms are performed at multiple sites in a cooperative fashion. Unfortunately, these applications have been implemented in an ad-hoc manner using two-sided communication protocols such as TCP/IP, which makes the sharing of state information between applications difficult and inefficient. Clearly, all the applications mentioned above can be greatly benefited by an efficient run-time substrate that can support their different needs efficiently. Since communication and synchronization are an inherent part of these applications, support for basic operations to read, write and synchronize are critical requirements from the DDSS. Further, as the nodes in a data-center environment may experience fluctuating load conditions depending on the traffic pattern of the incoming requests, the DDSS needs to be resilient and robust to changing system loads.

3.2 Requirements of Higher-level Data-Center Services:

Higher-level data-center services are intelligent services that are critical for the efficient functioning of data-centers. Services such as active caching [20] and cooperative caching [21] deal with efficient and load-resilient caching techniques for both static and dynamic content, while the active resource adaptation service deals with scalable management of various system resources. Other services such as resource monitoring actively monitors the resource usage and helps other higher-level services in identifying the bottleneck resources and alleviating such bottlenecks as they occur.

All these services require sharing of some state information. For example, caching services require the need for maintaining versions of cached copies of data and locking mechanisms for supporting cache coherency and consistency. Other services such as active resource adaptation require the need for advanced locking mechanism in order to move nodes serving one website to another in a transparent manner and needs simple mechanisms for data sharing. Resource monitoring services, on the other hand, require efficient, low overhead access to the load information on the nodes. The DDSS has to be designed in a manner that meets all of the above requirements.

4 Design Goals of DDSS

To effectively manage information-sharing in a data-center environment, the DDSS must understand in totality, the properties and the needs of data-center applications and services and must cater to these in an efficient manner.

Caching dynamic content at various tiers of a multi-tier data-center is a well known method to reduce the computation and communication overheads. Since the cached data is stored at mul-

tiple sites for caching purposes, there is a need to maintain cache coherency and consistency. Current data-centers support methods like Adaptive TTL [13], invalidation schemes [18] and also strong cache coherence schemes [20] for online transactions. Broadly, to accommodate the diverse coherency requirements of data-center applications and services, DDSS supports a range of coherency models.

The six basic coherency models [12] to be supported are: 1) *S*trict Coherence, which always obtains the most recent version and excludes concurrent writes and reads. Database transactions require strict coherence to support atomicity. 2) *W*rite Coherence, which always obtains the most recent version and excludes concurrent writes. Resource monitoring services [29] need such a coherence model so that the server can update the system load and other load-balancers can read this system information concurrently. 3) *R*ead Coherence is similar to write coherence except that it excludes concurrent readers. Services such as reconfiguration [6] are usually performed at many nodes and such services constantly monitor the system load information and dynamically move applications to serve other websites in order to maximize the resource utilization. Though all nodes perform the same function, such services can benefit from a read coherence model to avoid two load balancers looking at the same load information and performing a reconfiguration. 4) *N*ull Coherence, which always accepts the currently 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. This is similar to the adaptive TTL model mentioned previously.

Secondly, to meet the consistency needs of data-center applications, the DDSS 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 to 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 cpu-intensive and hence require the data to be maintained at remote nodes distributed over the cluster.

Apart from the above, DDSS should also meet the following needs. Due to the presence of multiple threads of each of these applications at each node in the data-center environment, the

DDSS should support the access, update and deletion of the shared data by all threads in a transparent manner. Services such as resource adaptation and monitoring are characterized by frequent reading of the system load on various nodes in the data-center. In order to efficiently support reading of this distributed state information, the DDSS 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, as mentioned in Section 3, the DDSS must be designed to be robust and resilient to load imbalances and should have minimal overheads and provide high performance access to data. Finally, the DDSS must provide an interface that clearly defines the mechanism to allocate, read, write and synchronize the data being managed in order for such services and applications to utilize the DDSS efficiently.

5 Proposed DDSS Framework and Implementation Issues

This section describes the proposed framework and its implementation details. Specifically, we present the mechanisms through which we achieve our design goals in an efficient manner.

The basic idea of DDSS is to allow efficient sharing of information across the cluster by creating a logical shared memory region. It supports two basic operations, *get* operation to read the shared data segment and *put* operation to write onto the shared data segment. Figure 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 simultaneously.

Figure 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. All master processes waits for the lock to be acquired for updating since the shared data is currently being read by multiple slave servers.

In order to efficiently implement distributed data sharing, several components need to be built. Figure 3 shows the various components of DDSS that help in satisfying the needs of the current and next generation data-center applications. Broadly, in the figure, all the colored boxes are the components which exist today. The white boxes are the ones which need to be designed to efficiently support next-generation data-center applications. In this paper, we concentrate on the boxes with the *dashed lines* by providing either complete or partial solutions. In this section, we describe how these components take advantage of advanced networks in providing efficient services.

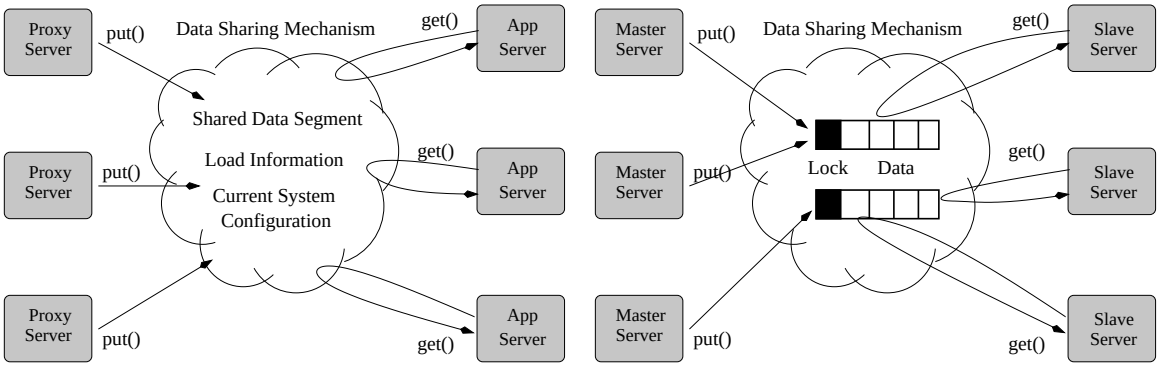

Figure 2: DDSS using the proposed Framework (a) Non Coherent Distributed Data Sharing Mechanism (b) Coherent Distributed Data Sharing Mechanism

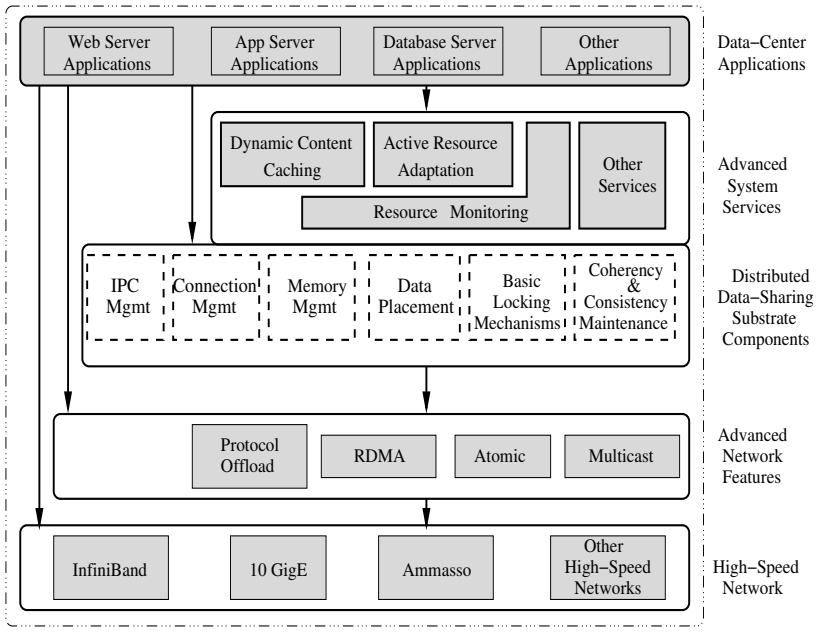

Figure 3: Proposed DDSS Framework

5.1 Connection Management

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 also allows for new connections to be established and existing connections to be terminated. All the nodes in the system are assigned specific ranks and applications interested in communicating with other nodes can either explicitly mention the node name or the rank.

5.2 Memory Management and Data Access

Similar to other data sharing mechanisms, memory management is an important issue for maintaining distributed data sharing. In our implementation, each node allocates a large pool of memory to be shared with DDSS. We perform the allocation and free operations inside this distributed pool of memory region. One way to implement the memory allocation in DDSS is to inform all the nodes about an allocation. However, as we mentioned earlier, since we focus on providing a low-overhead DDSS, informing all the nodes may lead to large latencies for memory allocation. Another approach to achieve this 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 are chosen in a round-robin fashion for every allocation so that the burden of maintaining the meta-data information is well distributed. The nodes also maintain a list of free blocks available within the pool memory. During a *release_ss()* operation for releasing the free block, we inform the designated remote node. In 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 and send it to requesting node. For the *release_ss()* operation, we can also use the advanced *IBV_WR_RDMA_WRITE_WITH_IMM* feature set offered by modern interconnects. This design allows for better progress as this event generates an interrupt at the remote node and the event is immediately handled. We plan to implement this feature as a part of future work. To gain access to DDSS, end-applications need to create an allocation unit and use *get(), put()* operations for reading and writing to the data sharing segment. High-speed networks provide one-sided remote memory operations (like RDMA read and RDMA write) that allow access to remote memory without interrupting the remote node. In our design, we use these operations to perform the read and write. All the applications and services

mentioned in Figure 3 will need this interface in order access/update the shared data.

5.3 Coherency and Consistency Maintenance

As mentioned earlier, we support six different coherence models. We implement the different coherence models by utilizing the one-side RDMA operations 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 DDSS 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 *IBV_WR_ATOMIC_FETCH_AND_ADD* 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. We plan to provide several consistency models as a part of future work. Proxy servers, applications servers and services such as dynamic content caching and reconfiguration utilize this component extensively.

5.4 Basic Locking Mechanisms

Basic locking mechanisms are provided using the atomic operations offered by modern interconnects. In our implementation of DDSS, every allocation unit is associated with a 64-bit integer which serves as a lock for acquiring access to the shared data. 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 interface automatically unlocks the shared segment after successful completion of *get()* and *put()* operations. Several services such as caching and reconfiguration can

utilize this component for providing efficient and scalable services.

5.5 Data Placement Techniques

Though DDSS hides the placement of shared data segments, it also exposes specific interfaces to the application to explicitly mention the location of the shared data segment (e.g. local or remote node). For the remote node case, the interface also allows the application to choose a particular node. As mentioned earlier, resource monitoring services can utilize this component to provide fine-grained services.

5.6 IPC Management

In order to support multiple user processes or threads in the system to access the DDSS, we optionally provide a run-time daemon to handle the requests of the multiple processes in the system. We use shared memory channels and semaphores for communication and synchronization purposes between the user process and the daemon. The daemon typically establishes connections with other data sharing daemons and forms the distributed data sharing framework. Any service which is multi-threaded or the presence of multiple services need to utilize this component for efficient communication.

5.7 Distributed Sharing Mechanism

In our implementation, every 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 data-center environments. This is particularly useful in reducing the contention at the NIC in the case where all the shared segment resides in one single node and several nodes needs access different data segment 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.

5.8 Locks Vs Data Sharing

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 distributed data sharing mechanism, 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.

Table 1: DDSS Interface

DDSS Operation	Description
int allocate_ss(nbytes, type, identifier)	allocate a memory block of size nbytes in the shared state
int release_ss(key)	free the shared data segment
int get(key, data, nbytes, timestamp, stack)	read nbytes of memory from the shared state and place it in data
int put(key, data, nbytes, timestamp, stack)	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

5.9 Interface

Table 1 shows the current interface of DDSS that is available to the end-user applications and services. The interface essentially supports six main operations for gaining access to DDSS: *allocate_ss(), get(), put(), releast_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 framework to reuse the memory chunk for future allocations. *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, DDSS also supports asynchronous operations such as *async_get(), async_put(), wait_ss()* and additional locking operations such as *try_lock()* operation for a wide range of applications.

6 Experimental Results

In this section, we evaluate DDSS with a set of microbenchmarks to understand the performance, scalability and associated overheads. Later, we analyze the applicability of DDSS with data-center services such as reconfiguration and active caching. Also, we analyze the performance and scalability of DDSS with applications such as Distributed STORM and check-pointing. We evaluate our DDSS framework on two interconnects IBA and Ammasso using the OpenFabrics implementation. 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.

6.1 Microbenchmark

In this section, we present the basic performance of DDSS. We also report the overhead and scalability of using DDSS.

6.1.1 Measuring Access Latency

The latency test is conducted in a ping-pong fashion and the latency is derived from round-trip time. For the 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. Similarly, we perform the *get()* operation and report the latency of *get()* operation. Figure 4a shows the latencies of different coherence models achieved by using the *put()* operation of DDSS 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. We observed that the overhead of communicating with the daemon process is close to 10-12μs which explains the large latencies we see 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 both write and strict coherency models use atomic operations to acquire the lock before updating the shared data segment. 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. Also, as the message size increases, we observe that the latency increases for all coherence models.

We see similar trends for *get()* operation as shown in Figure 4b. However, we observe that the latency of Version-based coherence model does not change with varying message sizes. The reason being, the current version maintained at the site requesting for the *get()* operation matches the version maintained at the remote end throughout the benchmark run. We see similar trends in the performance of latencies using OpenFabrics over Ammasso as shown in Figure 5. As mentioned earlier, the communication through the daemon process adds some amount of overhead. However,

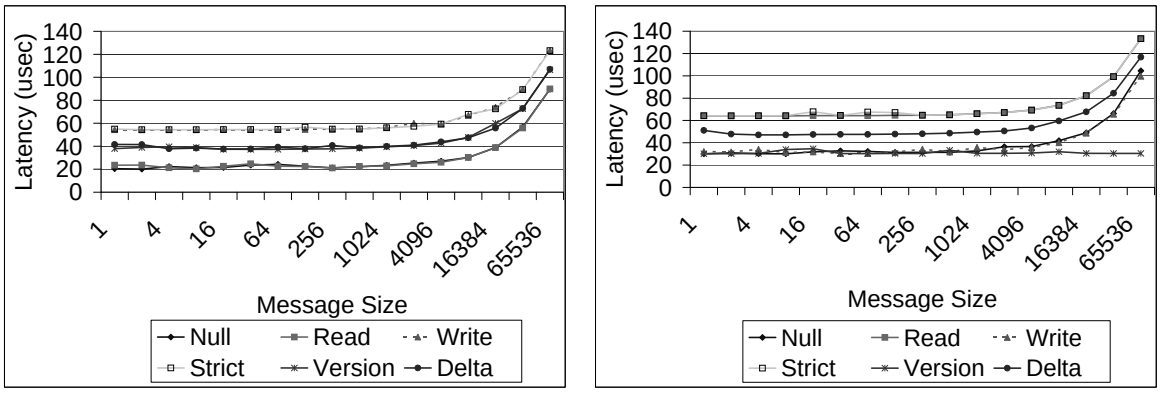

Figure 4: Basic Performance using OpenFabrics over IBA: (a) *put()* operation (b) *get()* operation

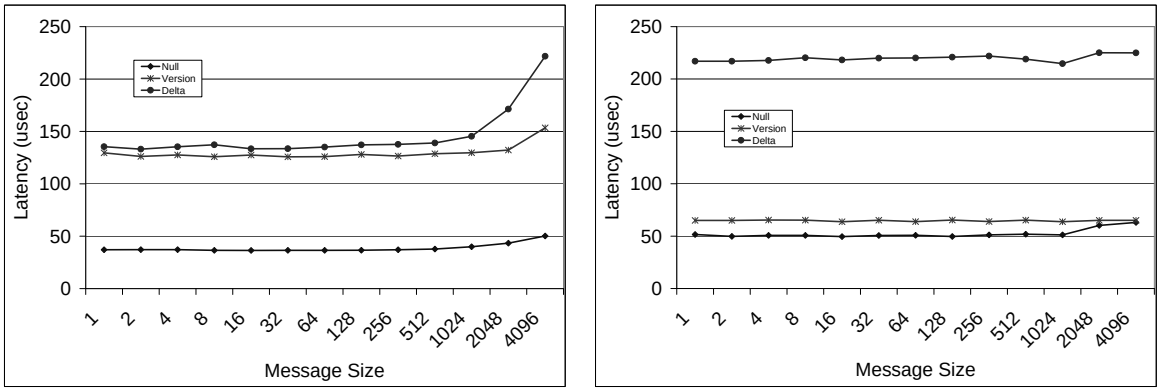

Figure 5: Basic Performance using OpenFabrics over Ammasso: (a) *put()* operation (b) *get()* operation

if we access DDSS without the daemon process (assuming only one service per node), the overhead is very minimal.

6.1.2 Measuring Substrate Overhead

One of the critical issues to address on supporting DDSS is to minimize the overhead of the middle-ware layer for applications. Table 2 reports the overhead of the middle-ware layer for different scenarios. We measure the overhead for different configurations (i) Direct scheme allows application to directly communicate with underlying network through DDSS library, (ii) Thread-based scheme allows application to communicate through a daemon process for accessing DDSS and (iii) Thread-based asynchronous scheme is same as thread-based scheme except that applications use asynchronous calls. We see that the overhead is less than a microsecond through the direct scheme. If the run-time 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 DDSS daemon consumes close to 10μs. If the

Table 2: Overhead of DDSS

DDSS-Model	Protocol Overhead	Synchronization Time	Total Overhead
Direct	0.35	20	20.35
Threading Support	10	20	30
Threading + Asynchronous Operations	12	20	32

application uses asynchronous operations (thread-based asynchronous scheme), this overhead can be significantly reduced for large message transfers. However, in the worst case scenario, for small message sizes and scheduling of asynchronous operations followed by a wait operation can lead to an overhead of 12μs. The average synchronization time observed in all the schemes is around 20μ.

6.1.3 Substrate Scalability with Access Contention

In order to test the scalability of DDSS, we designed several experiments accessing the shared data segments with increasing number of clients. In the first experiment as shown in Figure 6a, every client accesses different portions of the shared data segment on the same node. The graphs clearly indicate that DDSS using OpenFabrics over IBA is highly scalable for such scenarios and increasing number of clients does not seem to affect the performance. The primary reason being DDSS is based on one-sided operations on the shared segment and the remote CPU is not disturbed.

However, if multiple clients access the same shared data segment residing on a particular node, we observe some amount of performance degradation based on the contention-level as shown in Figure 6b. In this experiment, every client performs several *get()* operation or a *put()* operation based on a contention percentage. If the contention percentage is 10% and if total number of operations including both *get()* and *put()* is 1000, then the total number of *put()* operation performed would be 100. We report the performance of various coherence models for this experiment. Due to requirement of acquiring the locks before accessing the shared data segment for strict and write coherence model, we expect the *get()* and *put()* operations to have a longer waiting time in acquiring the locks. As shown in the figure, we observe that for relatively lesser contention-levels of up to 40%, the performance of *get()* and *put()* operations do 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 model mainly due to the waiting time for acquiring the lock.

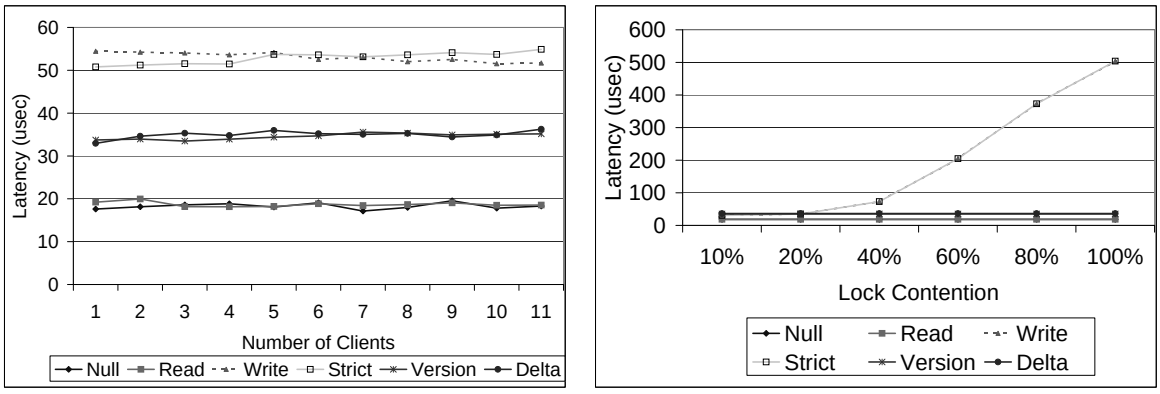

Figure 6: Performance of *put()* operations using OpenFabrics over IBA (a) Increasing Clients accessing different portions (b) Contention accessing the same shared segment

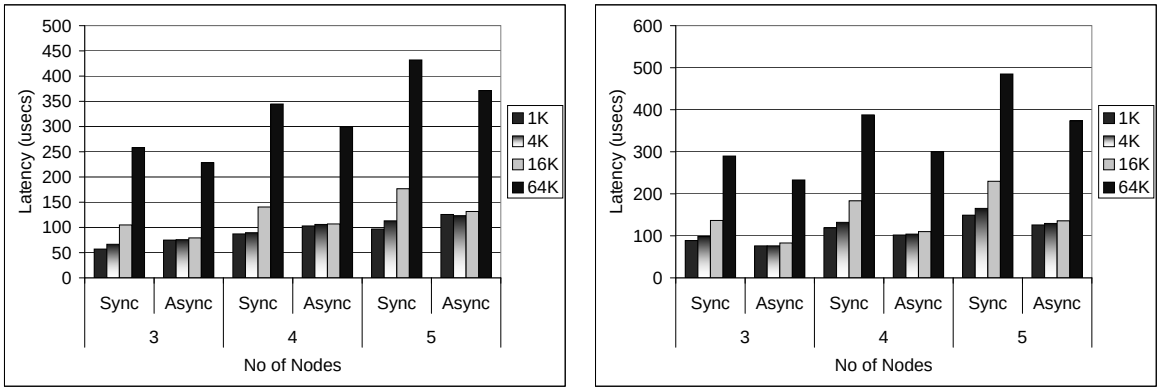

Figure 7: Performance of *get()* and *put()* operations

6.2 Impact of Asynchronous Operations

We have also measured the performance of asynchronous operations using OpenFabrics on IBA as shown in Figure 7. As expected, large message asynchronous transfers siginificantly reduces the overhead of DDSS.

6.3 Data-Center Service Evaluation

In this section, we evaluate our DDSS with data-center services in terms of performance, scalability and loaded conditions in a data-center environment.

6.3.1 Dynamic Reconfiguration

Multi-tier data-centers are logically broken down into several tiers or sub-clusters which handle different aspects of the data-center functionality. The increase in such services and partitions results in a growing fragmentation of the resources available and ultimately in the degradation of the performance provided by the data-center. Active resource adaptation and reconfiguration

alleviates this problem of wastage of resources by dynamically mapping applications to resources available inside the data-center.

In our previous work [7, 6] we have shown the strong potential of using the advanced features of high-speed networks in designing reconfiguration techniques. In this section, we use this technique to illustrate the overhead of using DDSS for such a service in comparison with implementations using native protocols. We modified our code base to use the DDSS and compared it with the previous implementation. Also, we simulate the loaded conditions of a real data-center scenario by firing client requests to the respective servers. Reconfiguration service dynamically moves the nodes inside the data-center based on server load and the website that is currently being served. If the load on the servers are significant and fluctuating, we expect the number of reconfiguration (moving nodes from one web-site to another) to increase. Also, the service needs to make sure that the loaded servers and free servers are locked before performing the switching and informing other nodes of the recent change. We use the direct scheme of DDSS for this implementation.

As shown in Figure 8a, 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 DDSS 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 data-center 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. The figure clearly shows that the contention for acquiring locks on loaded servers does not affect the total reconfiguration time showing the scalable nature of this service.

6.3.2 Strong Cache Coherence

In our previous work [20], we have shown the strong potential of using the features of modern interconnects such as IBA in alleviating the issues of providing strong cache coherence with traditional implementations. In this section, we show the load resilient nature of the one-sided feature in providing such strong cache coherency using DDSS over Ammasso. OpenFabrics implemen-

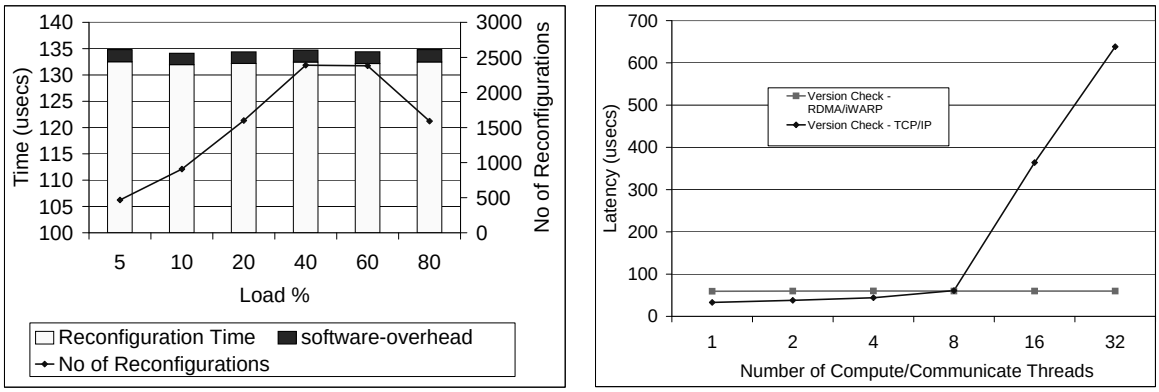

Figure 8: Software Overhead on Data-Center Services (a) Active Resource Adaptation using OpenFabrics over IBA (b) Dynamic Content Caching using OpenFabrics over Ammasso

tation over Ammasso is iWARP-compliant and hence the communication can also go over WAN. As shown in Figure 8b, we observe that as we increase the number of server compute threads, the time taken to check for the version number increases exponentially for a two-sided communication protocol such as TCP/IP. However, since DDSS is based on one-sided operations (RDMA over iWARP in this case), we observe that the time taken for version check remains constant for increasing number of compute threads.

6.4 Application-level Evaluation

In this section, we evaluate the performance of DDSS with applications such as check-pointing and Distributed STORM with DataCutter.

6.4.1 STORM with DataCutter

STORM [19, 4] is a middle-ware 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. It is implemented using DataCutter [8] which is designed to enable exploration and analysis of scientific datasets. DataCutter library provides a set of services on which application developers can implement more application-specific services. We concentrate only on the applicability of DDSS with respect to STORM application. As mentioned above, STORM selects the data of interest and transfers this data to the compute nodes. In distributed environments, it is also common to have several STORM applications running which can act on same or different datasets serving the queries of different clients. Due to the fact that the same dataset is processed by multiple STORM nodes and multiple compute nodes, DDSS can

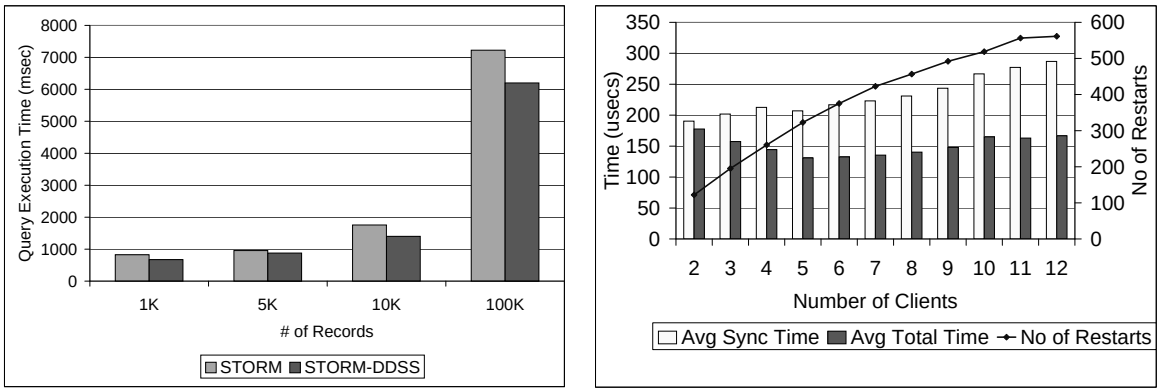

Figure 9: Application Performance using OpenFabrics over IBA (a) Distributed STORM application (b) Check-pointing

help in sharing this dataset in a cluster environment so that multiple nodes can get direct access to this shared data. DDSS can also benefit applications exchanging several meta data information frequently. We realize that the meta data management of STORM is significant and all the compute nodes access STORM to get meta data frequently. However, we are currently facing some minor porting issues in the implementation. We will include these results in the final version of the paper.

In our experiment, we modified the STORM application code to use DDSS in maintaining the dataset so that all nodes can 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 DDSS. Due to the fact that larger datasets showed inconsistent values, we perform the experiments on small datasets and we completely flush the file system cache in order to show the benefits of maintaining this dataset on other nodes memory. As shown in Figure 9a, we observe that the performance of STORM is improved by around 19% for 1K, 10K and 100K record dataset sizes using DDSS in comparison with the traditional implementation.

6.4.2 Check-pointing

We use a check-pointing benchmark to show the scalability and the performance of using DDSS. 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 take a consistent check-point and informing all other processes to revert back to the consistent check-point at other random intervals. In this experiment, we show the scalability of this application with increasing number of processes. In Figure 9b, we observe that the average time taken for check-pointing is only around 150μs for increasing number of processes. Since this value remains constant around

this range for increasing number of clients and increasing number of application restarts, we can conclude that the application scales well using DDSS. Also, we see that the average application restart time to reach a consistent checkpoint increases with the increase in the number of clients. This is expected as each process needs to get the current checkpoint version from all other processes in order to decide the most recent consistent checkpoint.

In summary, we notice that DDSS enhances several services like reconfiguration and active caching and applications such as distributed STORM and check-pointing by improving both performance and scalability.

7 Related Work

There has been several distributed data 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 real data-center environment on very large scale clusters.

Several run-time data sharing models such as InterWeave [11, 28], Khazana [9], InterAct [23] offer many benefits to applications in terms of relaxed coherency and consistency protocols. Friedman [16] and Amza et. al [5] have shown ways of combining consistency models. Khazana [9] also proposes the use of several consistency models. InterWeave [11, 28] allows various coherence models allowing users to define application-specific coherence models. Many of these models are implemented based on 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 data-center environment under heavy loaded conditions. Also, none of distributed data 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 included in the OpenFabrics standard, DDSS can also work well in WAN environments and can still benefit applications using the advanced features offered by modern networks.

8 Conclusion and Future Work

In this paper, we proposed and evaluated a low-overhead distributed data sharing substrate for cluster-based data-center environment targeting the applications and services that are traditionally hosted in these environments. 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 overheads and provide high performance by leveraging the features of modern interconnects like one-sided communication 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 data-center services such as reconfiguration and active caching not only show an order of magnitude performance improvement over traditional implementations but also show the load resilient nature of the substrate. Application-level evaluations with Distributed STORM using DataCutter achieves close to 19% performance improvement over traditional implementation, while evaluations with check-pointing application suggest that DDSS is scalable and has a low overhead.

We plan to enhance the distributed data sharing substrate by supporting advanced locking mechanisms and study the performance benefits in a wide range of services and applications such as meta-data management and access and storage of BTree data structure in database servers, advanced caching techniques and several others.

References

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

[2] Infiniband trade association. http://www.infinibandta.org.

[3] InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 1.0. http://www.infinibandta.com.

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

[5] 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.

[6] P. Balaji, S. Narravula, K. Vaidyanathan, H. W. Jin, and Dhabaleswar K. Panda. On the Provision of Prioritization and Soft QoS in Dynamically Reconfigurable Shared Data-Centers over InfiniBand. In *the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.

[7] 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 (RDMA): Applications, Implementations, and Technologies (RAIT)*, San Diego, CA, Sep 20 2004.

[8] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed Processing of Very Large Datasets with DataCutter. *Parallel Computing*, October 2001.

[9] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Services. In *ICDCS*, May 1998.

[10] 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.

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

[12] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M. Scott. Exploiting high-level coherence information to optimize distributed shared state. In *Proc. of the 9th ACM Symp. on Principles and Practice of Parallel Programming*, June 2003.

[13] Michele Colajanni and Philip S. Yu. Adaptive ttl schemes for load balancing of distributed web servers. *SIGMETRICS Perform. Eval. Rev.*, 25(2):36–42, 1997.

[14] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *Proceedings of the IEEE International Conference on Supercomputing*, Phoenix, Arizona, November 2003.

[15] 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.

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

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

[18] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol. IETF Internet Draft, November 2000. .

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

[20] 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 *SAN*, 2004.

[21] 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 *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2005.

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

[23] 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*, May 1998.

[24] 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.

[25] 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 *the Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages pages 61–72, San Francisco, CA, March 2001.

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

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

[28] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Efficient Distributed Shared State for Heterogeneous Machine Architectures. In *ICDCS*, May 2003.

[29] 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 (RDMA): Applications, Implementations, and Technologies (RAIT)*, Barcelona, Spain, 2006.