Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand

SUNDEEP NARRAVULA, PAVAN BALAJI, KARTHIKEYAN VAIDYANATHAN, SAVITHA KRISHNAMOORTHY, JIESHENG WU AND DHABALESWAR K. PANDA

Technical Report
OSU-CISRC-11/03-TR65

Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand[*]

S. Narravula P. Balaji K. Vaidyanathan S. Krishnamoorthy J. Wu D. K. Panda

Computer and Information Science,
The Ohio State University,
2015 Neil Avenue,
Columbus, OH-43210
{narravul, balaji, vaidyana, savitha, wuj, panda}@cis.ohio-state.edu

Abstract

It has been well acknowledged in the research community that in order to provide or design a data-center environment which is efficient and offers high performance, one of the critical issues that needs to be addressed is the effective reuse of cache content stored away from the origin server. In the current web, many cache eviction policies and uncachable resources are driven by two server application goals: Cache Coherence and Cache Consistency. The problem of how to provide consistent caching for dynamic content (Active Caches) has been well studied and researchers have proposed several weak as well as strong consistency algorithms. However, the problem of maintaining cache coherence has not been studied as much. In this paper, we propose an architecture for achieving strong cache coherence for multi-tier data-centers over InfiniBand using the previously proposed client-polling mechanism. The architecture as such could be used with any protocol layer; we have also proposed some optimizations to the algorithm to take advantage of the advanced features provided by InfiniBand. We evaluate this architecture using three protocol platforms: (i) TCP/IP over InfiniBand (IPoIB), (ii) Sockets Direct Protocol over InfiniBand (SDP) and (iii) the native InfiniBand Verbs layer (VAPI) and compare it with the performance of the no-caching based coherence mechanism. Our experimental results show that the InfiniBand-Optimized architecture can achieve an improvement of nearly an order of magnitude compared to the throughput achieved by the TCP/IP based architecture (over IPoIB), the SDP based architecture and the no-cache based coherence scheme.

We also propose a shared cache state based adaptive push-pull architecture for propagating updates. Again, we evaluate this using three protocol platforms: TCP/IP, SDP and VAPI. Our experimental results show that even with a simplistic scheme to make the data-push decision, all the three implementations of the adaptive push-pull model (TCP/IP based, SDP based and VAPI based) are able to achieve an efficient trade-off between the number of cache misses and the number of wasted updates.

Keywords: *Data-Center, Caching, InfiniBand, Cache Coherence*

1 Introduction

With the increasing adoption of the Internet as the primary means of electronic interaction and communication, E-portal and E-commerce, web servers which are highly scalable, highly available and high performance, have become critical for companies to reach, attract, and keep customers. Multi-tier Data-centers have become a central requirement to providing such services. Figure 1 depicts a typical multi-tier data-center environment. The first tier consists of front-end servers such as the proxy servers that provide web, messaging and various other services to clients on a network. The middle tier usually comprises of application servers that handle transaction processing and implement data-center business logic. The back-end tier consists of database servers that hold a persistent state of the databases and other data repositories. As mentioned in [23], a fourth tier emerges in today's data-center environment: a communication service tier between the network and the front-end server farm for providing edge services such as load balancing, security, caching, and others.

With the ever increasing on-line businesses and services and the growing popularity of personalized Internet ser-

[*] This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052, #CCR-0204429, and #CCR-0311542

vices, dynamic content is becoming increasingly common [11, 28, 24]. This includes documents that change upon every access, documents that are results of queries, documents that embody client-specific information, and many others. Large-scale dynamic workloads pose interesting challenges in building the next-generation data-centers [28, 23, 14, 25]. Significant computation and communication may be required to generate and deliver dynamic content. Performance and scalability issues need to be addressed for such workloads.

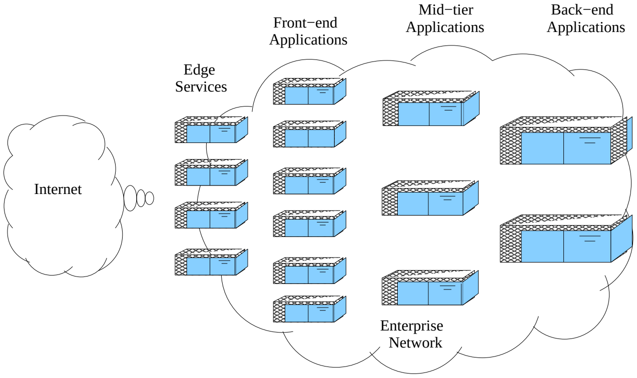

Figure 1. A Typical Multi-Tier Data-Center (Courtesy CSP Architecture design)

Reducing computation and communication overhead is crucial to improving the performance and scalability of data-centers. Caching content at various tiers of a multi-tier data-center is a well known method to reduce the computation and communication overheads. However, caching dynamic content, typically known as *Active Caching* [11], has its own challenges. Issues such as cache consistency and cache coherence become more prominent for dynamically generated data. In the state-of-art data-center environment, these issues are handled based on the type of data being cached. For dynamic data, for which relaxed consistency or coherency is permissible, several methods like TTL [15], Adaptive TTL [12], and Invalidation [17] have been proposed. However, for data like stock quotes or airline reservation, where old quotes or old airline availability values are not acceptable, strong consistency and coherency is essential.

Providing strong consistency and coherency is a necessity for *Active Caching* in many web applications, such as on-line banking and transaction processing. In the current data-center environment, two popular approaches are used. The first approach is pre-expiring all entities (forcing data to be refetched from the origin server on every request). This scheme is similar to a no-cache scheme. The second approach, known as *Client-Polling*, requires the front-end nodes to inquire from the back-end server if its cache entry is valid on every cache hit. Both approaches are very costly, increasing the client response time and the processing overhead at the back-end servers. The costs are mainly associated with the high CPU overhead in the traditional network protocols due to memory copy, context switches, and interrupts [23, 14, 6]. Further, the involvement of both sides for communication (two-sided communication) results in, performance of these approaches heavily relying on the CPU load on both communication sides. For example, a busy back-end server can slow down the communication required to maintain strong cache coherence significantly.

The InfiniBand Architecture (IBA) [1, 2] is envisioned as the default interconnect for the future data-center environments. It is targeted for both Inter-Processor Communication (IPC) and I/O. Therefore, a single IBA interconnect can be used to meet different purposes. This significantly eases network management in data-center servers. In addition, IBA is designed to achieve low latency and high-bandwidth with low CPU overhead. It also provides rich features to greatly improve RAS (Reliability, Availability, and Scalability) of the data-center servers. IBA relies on two key features, namely *User-level Networking* and *Remote Direct Memory Access* (RDMA). User-level Networking allows applications to directly and safely access the network interface without going through the Operating System. RDMA allows the network interface to transfer data between local and remote memory buffers without any interation with the Operating System or processor intervention by using DMA engines. These two features have been leveraged in designing high performance message passing systems [18] and cluster file systems [27].

In this paper, we focus on leveraging these two features to support strong coherency for caching dynamic content in the data-center environment. In particular, we study mechanisms to take advantage of InfiniBand's features to provide strong cache consistency and coherency with low overhead and to provide scalable dynamic content caching (*Active Caching*).

This work contains several research contributions. Primarily, it takes the first step toward understanding the role of the InfiniBand architecture in next-generation data-centers. The main contributions are:

1. We propose an architecture for achieving strong cache coherence for multi-tier data-centers. This architecture requires minimal changes to legacy data-center applications. It could as such be used with any protocol layer; at the same time, it allows us to take advantage of the advanced features provided by InfiniBand to further improve performance and scalability of caching in the data-center environment.
2. We implement the proposed architecture using three protocol platforms: TCP/IP over InfiniBand (IPoIB), Sockets Direct Protocol over InfiniBand (SDP) and the

native InfiniBand Verbs layer (VAPI). We evaluate this architecture with all three implementations and compare it with the performance of the no-caching based coherence mechanism. Our experimental results show that the InfiniBand-Optimized architecture can achieve an improvement of nearly an order of magnitude compared to the throughput achieved by the TCP/IP based architecture, SDP based architecture and the no-cache based coherence scheme.

3. Our results also show that one-sided operations such as the RDMA operations can provide better performance robustness to load in the data-center environment compared to two-sided protocols such as TCP/IP and SDP over the same IBA network. Performance of *Active Caching* with strong coherency based on the RDMA communication mechanism is mostly resilient and well-conditioned to the load on the application servers. This feature becomes more important because of the unpredictability of load in a typical data-center environment which supports large-scale dynamic services.

4. InfiniBand provides opportunities to revise the design and implementation of many subsystems, protocols, and communication mechanisms in the data-center environment. The rich features provided by IBA offer a flexible design space and tremendous optimization potential.

The rest of the paper is organized as follows. Section 2 describes the background and related work. In Section 3, we detail the design and challenges of our approach. Section 4 presents a shared cache state based hybrid Push-Pull model which we propose as an extension to improve the performance of cache coherent data-centers. The experimental results are presented in Section 5. We draw our conclusions and discuss possible future work in Section 6.

2 Background

In this section, we provide some background work previously done in three broad directions: (1) Various schemes proposed by researchers to allow *bounded staleness* to the accessed documents, maintaining strong consistency, etc., (2) InfiniBand Architecture and the features it provides and (3) the Sockets Direct Protocol over InfiniBand (SDP).

2.1 Web Cache Consistency and Coherence

It has been well acknowledged in the research community that in order to provide or design a data-center environment which is efficient and offers high performance, one of the critical issues that needs to be addressed is the effective reuse of cache content stored away from the origin server. This has been strongly backed up by researchers who have come up with several approaches to cache more and more data at the various tiers of a multi-tier data-center. Traditionally, frequently accessed static content was cached at the front tiers to allow users a quicker access to these documents. In the past few years, researchers have come up with approaches of caching certain dynamic content at the front tiers as well [11].

In the current web, many cache eviction events and uncachable resources are driven by two server application goals: First, providing clients with a *recent* or *coherent* view of the state of the application (i.e., information that is not too old); Secondly, providing clients with a *self-consistent* view of the application's state as it changes (i.e., once the client has been told that something has happened, that client should never be told anything to the contrary).

The web does not behave like a distributed file system (DFS) or distributed shared memory (DSM) system; among the dissimilarities are: (1) the lack of a *write* semantic in common use - while the HTTP protocol does include a *PUT* event which is in some ways comparable to a write, it is rarely used. The most common write-like operation is *POST* which can have completely arbitrary semantics and scope. This generality implies, in the general case, an inability to *batch* user induced updates. (2) The complexity of addressing particular content - URLs or *web addresses* do not in fact address units of contents *per se*, but rather address generic objects (*resources*) which produce content using completely opaque processes. (3) The absence of any protocol-layer persistent state or notion of *transactions* to identify related, batched or macro-operations. These issues are further illuminated by Mogul in [21].

In a DSM or DFS world, the mapping from write events to eventual changes in the *canonical* system state is clearly defined. In the web, non-safe requests from users can have arbitrary application-defined semantics with arbitrary scopes of affect completely unknowable from the parameters of a request, or even from the properties of a response. For this reason, the definitions of consistency and coherence used in the DFS/DSM literature do not fit the needs of systems like the data-center; instead, we use definitions more akin to those in the distributed database literature.

Depending on the type of data being considered, it is necessary to provide certain guarantees with respect to the view of the data that each node in the data-center and the users get. These constraints on the view of data vary depending on the application requiring the data.

Consistency: Cache consistency refers to a property of the responses produced by a single logical cache, such that no response served from the cache will reflect older state of the server than that reflected by previously served responses, i.e., a consistent cache provides its clients with non-decreasing views of the server's state.

Coherence: Cache coherence refers to the average *staleness* of the documents present in the cache, i.e., the time elapsed between the current time and the time of the last update of the document in the back-end. A cache is said to be strong coherent if its average *staleness* is *zero*, i.e., a client would get the same response whether a request is answered from cache or from the back-end.

2.1.1 Web Cache Consistency

In a multi-tier data-center environment many nodes can access data at the same time (*concurrency*). Data consistency provides each user with a consistent view of the data, including all visible (committed) changes made by the user's own updates and the updates of other users. That is, either all the nodes see a completed update or no node sees an update. Hence, for strong consistency, stale view of data is permissible, but partially updated view is not.

Several different levels of consistency are used based on the nature of data being used and its consistency requirements. For example, for a web site that reports football scores, it may be acceptable for one user to see a score, different from the scores as seen by some other users, within some frame of time. There are a number of methods to implement this kind of weak or lazy consistency models.

The *Time-to-Live (TTL)* approach, also known as the Δ-*consistency* approach, proposed with the HTTP/1.1 specification, is a popular weak consistency (and weak coherence) model currently being used. This approach associates a *TTL* period with each cached document. On a request for this document from the client, the front-end node is allowed to reply back from their cache as long as they are within this *TTL* period, i.e., before the *TTL* period expires. This guarantees that document cannot be more *stale* than that specified by the *TTL* period, i.e., this approach guarantees that staleness of the documents is bounded by the *TTL* value specified.

Researchers have proposed several variations of the *TTL* approach including *Adaptive TTL* [12] and *MONARCH* [19] to allow either dynamically varying *TTL* values (as in *Adaptive TTL*) or document category based *TTL* classification (as in *MONARCH*). There has also been considerable amount of work on *Strong Consistency* algorithms [10, 9].

2.1.2 Web Cache Coherence

Typically, when a request arrives at the proxy node, the cache is first checked to determine whether the file was previously requested and cached. If it is, it is considered a cache hit and the user is served with the cached file. Otherwise the request is forwarded to its corresponding server in the back-end of the data-center.

The maximal hit ratio in proxy caches is about 50% [24]. Majority of the cache misses are primarily due to the dynamic nature of web requests. Caching dynamic content pages is much more challenging than static content because the cached object is related to data at the back-end tiers. This data may be updated, thus invalidating the cached object and resulting in a cache miss. The problem of how to provide consistent caching for dynamic content has been well studied and researchers have proposed several weak as well as strong cache consistency algorithms [10, 9, 28]. However, the problem of maintaining cache coherence has not been studied as much.

There are two popularly used coherency models in the current web: *immediate or strong coherence* and *bounded staleness*.

The *bounded staleness* approach is similar to the previously discussed *TTL* based approach. Though this approach is efficient with respect to the number of cache hits, etc., it only provides a weak cache coherence model. On the other hand, *immediate coherence* provides a strong cache coherence.

With *immediate coherence*, caches are forbidden from returning a response other than that which would be returned were the origin server contacted. This guarantees semantic transparency, provides *Strong Cache Coherence*, and as a side-effect also guarantees *Strong Cache Consistency*. There are two widely used approaches to support *immediate coherence*. The first approach is pre-expiring all entities (forcing all caches to re-validate with the origin server on every request). This scheme is similar to a no-cache scheme. The second approach, known as *client-polling*, requires the front-end nodes to inquire from the back-end server if its cache is valid on every cache hit. This cuts down on the cost of transferring the file to the front end on every request even in cases when it had not been updated.

The no-caching approach to maintain *immediate coherence* has several disadvantages:

- Each request has to be processed at the home node tier, ruling out any caching at the other tiers
- The propagation of these requests to the back-end home node over traditional protocols can be very expensive
- For data which does not change frequently, the amount of computation and communication overhead incurred to maintain strong coherence could be very high, requiring more resources

These disadvantages are overcome to some extent by the *client-polling* mechanism. In this approach, the proxy server, on getting a request, checks its local cache for the availability of the required document. If it is not found, the request is forwarded to the appropriate application server in the inner tier and there is no cache coherence issue involved at this tier. If the data is found in the cache, the proxy server checks the *coherence status* of the cached object by contacting the back-end server(s). If there were updates made to the dependent data, the cached document is discarded and the request is forwarded to the application server tier for processing. The updated object is now cached for future use. Even though this method involves contacting the back-end for every request, it benefits from the fact that the actual data processing and data transfer is only required when the data is updated at the back-end. This scheme can potentially have significant benefits when the back-end data is not updated very frequently. However, this scheme has its own set of disadvantages, mainly based on the traditional networking protocols:

- Every data document is typically associated with a home-node in the data-center back-end. Frequent accesses to a document can result in all the front-end nodes sending in *coherence status* requests to the same nodes potentially forming a *hot-spot* at this node

- Traditional protocols require the back-end nodes to be interrupted for every cache validation event generated by the front-end

In this paper, we focus on this model of cache coherence and analyze the various impacts of the advanced features provided by InfiniBand on this.

2.2 InfiniBand Architecture

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). IBA defines a semantic interface called as Verbs for the consumer applications to communicate with the HCAs.

IBA mainly aims 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 HCA. The specifications for Verbs includes a queue-based interface, known as a Queue Pair (QP), to issue requests to the HCA. Figure 2 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 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 transfered (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 request, no kind of completion event is returned to the user. However, depending on the implementation, the driver cleans up the the Work Queue Request from the appropriate Queue Pair on completion.

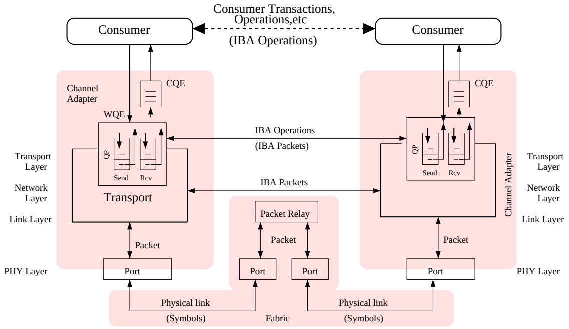

Figure 2. InfiniBand Architecture (Courtesy InfiniBand Specifications)

2.2.1 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. Failure to post a receive descriptor on the remote node results in the message being dropped and if the connection is reliable, it might even result in the breaking of the connection.

In memory semantics, Remote Direct Memory Access (RDMA) operations are used. These operations are transparent at the remote end since they do not require a receive descriptor to be posted. In this semantics, the send request itself contains both the virtual address for the local transmit buffer as well as that for the receive buffer on the remote end.

Most entries in the WQR are common for both the Send-Receive model as well as the RDMA model, except an additional remote buffer virtual address which has to be specified for RDMA operations.

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.

RDMA operations have two notable advantages for us to design and implement strong cache coherence. First, it is one-sided communication, that is completely transparent to the peer side. Therefore, the initiator can initiate RDMA operations at its own will. Eliminating involvement of the peer side can overcome the communication performance degradation due to CPU workload of the peer side. This also avoids any interrupt of the peer side processing. Second, RDMA operations provide a "shared-memory illusion". This eases status sharing in caching. In the following section, we describe the design details.

2.3 Sockets Direct Protocol

Sockets Direct Protocol (SDP) is an IBA specific protocol defined by the Software Working Group (SWG) of the InfiniBand Trade Association [4]. The design of SDP is mainly based on two architectural goals:

- Maintain traditional sockets SOCK_STREAM semantics as commonly implemented over TCP/IP. Issues include graceful closing of connections, ability to use TCP port space, IP addressing (IPv4, IPv6), Connecting/Accepting connect model, Out-of-Band data (OOB) and support for common socket options

- Support for byte-streaming over a message passing protocol, including kernel bypass data transfers and zero-copy data transfers

The SDP specifications focuses specifically on the wire protocol, finite state machine and packet semantics. Operating system issues, etc can be implementation specific. It is to be noted that SDP supports only SOCK_STREAM or Streaming sockets semantics and not SOCK_DGRAM (datagram) or other socket semantics. There has also been some previous work on such high performance sockets interfaces over Gigabit Ethernet [7] and VIA over GigaNet cLAN [16, 22, 8]. In this paper, we use the SDP interface over InfiniBand in order to allow the comparison of the different protocol stacks over InfiniBand.

2.3.1 SDP Overview

SDP's Upper Layer Protocol (ULP) interface is a byte-stream that is layered on top of InfiniBand's Reliable Connection (RC) message-oriented transfer model. The mapping of the byte stream protocol to InfiniBand message-oriented semantics was designed to enable ULP data to be transfered by one of two methods: through intermediate private buffers (Bcopy) or directly between ULP buffers (Zcopy).

A mix of InfiniBand Send and RDMA mechanisms are used to transfer ULP data. Zcopy uses RDMA reads or writes, transferring data between RDMA buffers (which typically belong to the ULP). Bcopy uses InfiniBand sends, transferring data between send and receive private buffers.

SDP has two types of buffers:

Private Buffers: Used for transmission of all SDP messages and ULP data that is to be copied into the receive ULP buffer. The Bcopy data transfer mechanism is used for this traffic.

RDMA Buffers: Used when performing Zcopy data transfer. ULP data is intended to be RDMAed directly from the Data Source's ULP buffer to the Data Sink's ULP buffer.

An implementation dependent parameter defined as the Bcopy Threshold is used to abstractly define the results of the policy decision. For the Bcopy implementation, SDP relies on a flow control mechanism similar to the TCP Sliding Window protocol, i.e., the sender keeps sending data till the window is full. When the application reads data from the socket buffer, the data sink sends a control message back to the data source updating its window size.

Figure 3 shows SDP in relation to the other Architecture layers in InfiniBand.

SDP specifications also specify two additional control messages known as "Buffer Availability Notification" messages.

Sink Avail Message: If the data sink has already posted a receive buffer and the data source has not sent the data message yet, the data sink does the following steps: (1) Registers the receive user-buffer (for large message reads) and (2) Sends a "Sink Avail" message containing the receive buffer handle to the source. The Data Source on a data transmit call, uses this receive buffer handle to directly RDMA write the data into the receive buffer.

Source Avail Message: If the data source has already posted a send buffer and the available SDP window is not large enough to contain the buffer, it does the following two steps: (1) Registers the transmit user-buffer (for large message sends) and (2) Send a "Source Avail" message containing the transmit buffer handle to the data sink. The Data Sink on a data receive call, uses this transmit buffer handle to directly RDMA read the data into the receive buffer.

2.3.2 SDP Implementation

The current implementation of SDP follows most of the specifications provided above. There are two major deviations from the specifications in this implementation.

- **Buffer Availability Notification:** The current implementation does not support "Source Avail" and "Sink

Avail” messages.

- **Zcopy implementation:** The current implementation does not support “Zcopy”. All data transfer is done through the Bcopy mechanism. This limitation can also be considered as part of the previous limitation, since they are always used together.

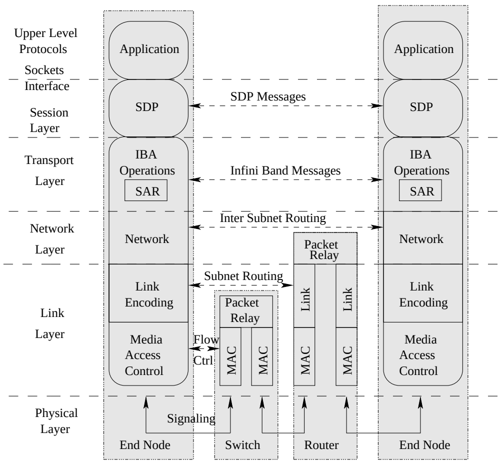

Figure 3. Sockets Direct Protocol Architecture (Courtesy: InfiniBand Specifications Volume I)

3 Providing Strong Cache Coherence

In this section, we describe the architecture we use to support strong cache coherence over InfiniBand. We first provide the basic design of the architecture for any generic protocol. Next, we point out several optimizations possible in the design using the various features provided by InfiniBand.

3.1 Basic Design

As mentioned earlier, there are two popular approaches to ensure cache coherence: *Client-Polling* and *No-Caching*. In this paper, we focus on the *Client-Polling* approach to demonstrate the potential benefits of InfiniBand in supporting strong cache coherence.

While the HTTP specification allows a cache-coherent client-polling architecture (by specifying a TTL value of NULL and using the `‘‘get-if-modified-since’’` HTTP request to perform the polling operation), it has several issues: (1) This scheme is specific to sockets and cannot be used with other programming interfaces such as InfiniBand’s native Verbs layers (e.g.: VAPI), (2) In cases where persistent connections are not possible (HTTP/1.0 based requests, secure transactions, etc), connection setup time between the nodes in the data-center environment tends to take up a significant portion of the client response time, especially for small documents.

In the light of these issues, we present an alternative architecture to perform *Client-Polling*. Figure 4 demonstrates the basic coherency architecture used in this paper. The main idea of this architecture is to introduce *external helper modules* that work along with the various servers in the data-center environment to ensure cache coherence. All issues related to cache coherence are handled by these modules and are obscured from the data-center servers. It is to be noted that the data-center servers require very minimal changes to be compatible with these modules.

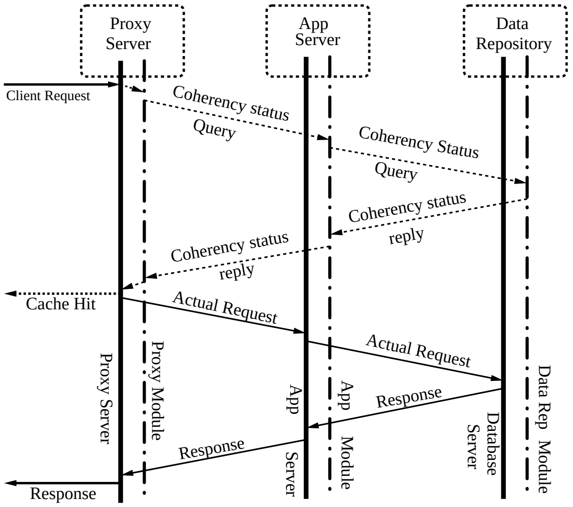

Figure 4. Strong Cache Coherence Protocol

The design consists of a module on each physical node in the data-center environment associated with the server running on the node, i.e., each proxy node has a proxy module, each application server node has an associated application module, etc. The proxy module assists the proxy server with validation of the cache on every request. The application module, on the other hand, deals with a number of things including (a) Keeping track of all updates on the documents it owns, (b) Locking appropriate files to allow a multiple-reader-single-writer based access priority to files, (c) Updating the appropriate documents during update requests, (d) Providing the proxy module with the appropriate version number of the requested file, etc.

Figure 5 demonstrates the functionality of the different modules and their interactions.

Proxy Module: On every request, the proxy server contacts the proxy module through Inter Process Communication (IPC) to validate the cached object(s) associated with the request. The proxy module does the actual verification

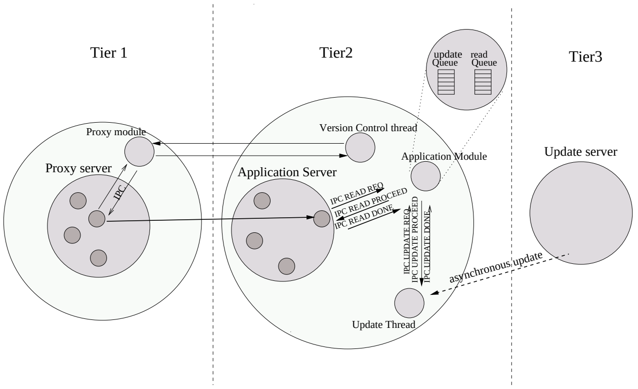

Figure 5. Interaction between Data-Center Servers and Modules

of the document with the application module on the appropriate application server. If the cached value is valid, the proxy server is allowed to proceed by replying to the client's request from cache. If the cache is invalid, the proxy module simply deletes the corresponding cache entry and allows the proxy server to proceed. Since the document is now not in cache, the proxy server contacts the appropriate application server for the document. This ensures that the cache remains coherent.

Application Module: The application module is slightly more complicated than the proxy module. It uses multiple threads to allow both updates and read accesses on the documents in a multiple-reader-single-writer based access pattern. This is handled by having a separate thread for handling updates (refered to as the *update thread* here on). The main thread blocks for IPC requests from both the application server and the update thread. The application server requests to read a file while an update thread requests to update a file. The main thread of the application module, maintains two queues to ensure that the file is not accessed by a writer (update thread) while the application server is reading it (to transmit it to the proxy server) and vice-versa.

On receiving a request from the proxy, the application server contacts the application module through an IPC call requesting for access to the required document (IPC_READ_REQUEST). If there are no ongoing updates to the document, the application module sends back an IPC message giving it access to the document (IPC_READ_PROCEED), and queues the request ID in its *Read Queue*. Once the application server is done with reading the document, it sends the application module another IPC message informing it about the end of the access to the document (IPC_READ_DONE). The application module, then deletes the corresponding entry from its *Read Queue*.

When a document is to be updated (either due to an update server interaction or an update query from the user), the update request is handled by the *update thread*. On getting an update request, the update thread initiates an IPC message to the application module (IPC_UPDATE_REQUEST). The application module on seeing this, checks its *Read Queue*. If the *Read Queue* is empty, it immediately sends an IPC message (IPC_UPDATE_PROCEED) to the update thread and queues the request ID in its *Update Queue*. On the other hand, if the *Read Queue* is not empty, the update request is still queued in the *Update Queue*, but the IPC_UPDATE_PROCEED message is not sent back to the update thread (forcing it to hold the update), until the *Read Queue* becomes empty. In either case, no further read-requests from the application server are allowed to proceed; instead the application module queues them in its *Update Queue*, after the update request. Once the update thread has completed the update, it sends an IPC_UPDATE_DONE message to the update module. At this time, the application module deletes the update request entry from its *Update Queue*, sends IPC_READ_PROCEED messages for every read request queued in the *Update Queue* and queues these read requests in the *Read Queue*, to indicate that these are the current readers of the document.

It is to be noted that if the *Update Queue* is not empty, the first request queued will be an update request and all other requests in the queue will be read requests. Further, if the *Read Queue* is empty, the update is currently in progress.

Table 1 tries to summarize this information.

3.2 Strong Coherency Model over InfiniBand

In this section, we point out several optimizations possible in the design described, using the advanced features provided by InfiniBand. In Section 5 we provide the performance achieved by the InfiniBand-optimized architecture.

As described earlier, on every request the proxy module needs to validate the cache corresponding to the document requested. In traditional protocols such as TCP/IP, this requires the proxy module to send a version request message to the *version thread*[1], followed by the *version thread* explicitly sending the version number back to the proxy module. This involves the overhead of the TCP/IP protocol stack for the communication in both directions. Several researchers have provided solutions such as SDP to get rid of the overhead associated with the TCP/IP protocol stack while maintaining the sockets API. However, the more important concern in this case is the processing required at the version thread (e.g. searching for the index of the requested file and returning the current version number).

Application servers typically tend to perform several computation intensive tasks including executing CGI scripts,

[1] Version Thread is a separate thread spawned by the application module to handle version requests from the proxy module

Table 1. IPC message rules

IPC_TYPE	Read Queue State	Update Queue State	Rule
IPC_READ_REQUEST	Empty	Empty	1. Send IPC_READ_PROCEED to proxy 2. Enqueue Read Request in Read Queue
IPC_READ_REQUEST	Not Empty	Empty	1. Send IPC_READ_PROCEED to proxy 2. Enqueue Read Request in Read Queue
IPC_READ_REQUEST	Empty	Not Empty	1. Enqueue Read Request in Update Queue
IPC_READ_REQUEST	Not Empty	Not Empty	Enqueue the Read Request in the Update Queue
IPC_READ_DONE	Empty	Not Empty	Erroneous State. Not Possible.
IPC_READ_DONE	Not Empty	Empty	1. Dequeue one entry from Read Queue.
IPC_READ_DONE	Not Empty	Not Empty	1. Dequeue one entry from Read Queue 2. If Read Queue is *now* empty, Send IPC_UPDATE_PROCEED to head of Update Queue
IPC_UPDATE_REQUEST	Empty	Empty	1. Enqueue Update Request in Update Queue 2. Send IPC_UPDATE_PROCEED
IPC_UPDATE_REQUEST	Empty	Not Empty	Erroneous state. Not Possible
IPC_UPDATE_REQUEST	Not Empty	Empty	1. Enqueue Update Request in Update Queue
IPC_UPDATE_REQUEST	Not Empty	Not Empty	Erroneous State. Not possible
IPC_UPDATE_DONE	Empty	Empty	Erroneous State. Not possible
IPC_UPDATE_DONE	Empty	Not Empty	1. Dequeue Update Request from Update Queue 2. For all Read Requests in Update Queue: - Dequeue Read Requests from Update Queue - Send IPC_READ_PROCEED - Enqueue in Read Queue
IPC_UPDATE_DONE	Not Empty	Not Empty	Erroneous State. Not Possible.

Java applets, etc. This results in a tremendously high CPU requirement for the main application server itself. Allowing an additional version thread to satisfy version requests from the proxy modules results in a high CPU usage for the module itself. Additionally, the large amount of computation carried out on the node by the application server results in significant degradation in performance for the version thread and other application modules running on the node. This results in a delay in the version verification leading to an overall degradation of the system performance.

In this scenario, it would be of great benefit to have a one-sided communication operation where the proxy module can directly check the current version number without interrupting the version thread. InfiniBand provides the RDMA read operation which allows the initiator node to directly read data from the remote node's memory. This feature of InfiniBand makes it an ideal choice for this scenario. In our implementation, we rely on the RDMA read operation for the proxy module to get information about the current version number of the required file.

Figure 6 demonstrates the InfiniBand-Optimized coherency architecture.

3.3 Potential Benefits

Using RDMA operations to design and implement client polling scheme in data-center servers over InfiniBand has several potential benefits.

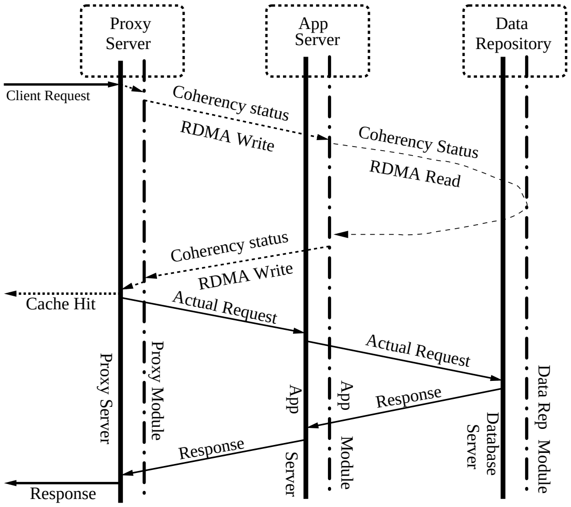

Figure 6. Strong Cache Coherency Protocol: InfiniBand based Optimizations

Improving response latency: RDMA operations over InfiniBand provide very low latency of about 5.5μs and a high bandwidth up to 840Mbytes per second. Protocol communication overhead to provide strong coherence is minimal. This can improve response latency.

Increasing system throughput: RDMA operations have very low CPU overhead in both sides. This leaves more CPU free for the data center nodes to perform other processing, particularly on the back-end servers. This benefit becomes more attractive when a large amount of dynamic content is generated and significant computation is needed in the data-center nodes. Therefore, clients can benefit from active caching with strong coherence guarantee at little cost. The system throughput can be improved significantly in many cases.

Enhanced robustness to load: The load of data center servers with support of dynamic web services is very bursty and unpredictable [24, 26]. Performance of protocols to maintain strong cache coherency over traditional network protocols can be degraded significantly when the server load is high. This is because both sides should get involved in communication and afford considerable CPU to perform communication operations. However, for protocols based on RDMA operations, the peer side is transparent to and nearly out of the communication procedure. Little overhead is paid on the peer server side. Thus, the performance of dynamic content caching with strong coherence based on RDMA operations is mostly resilient and well-conditioned to load.

4 Data Pull Model Vs Data Push Model

Based on the dynamics of the data, it might either be beneficial for the proxies to pull data from the inner tiers (by forwarding the requests to application servers, etc) whenever it detects an update or to have the back-end node push the data to the outer tiers as soon as an update takes place. Each of these approaches has its own advantages and disadvantages [5].

4.1 Client-driven mechanism (Data Pull Model)

In the Data Pull Model, the back-end nodes do not play any active role in updating the outer tiers' caches in case of an update, i.e., the back-end nodes are only passively involved in cache updations. On every request, the proxy checks to see if the data is present in cache (and valid in case of strong coherency). If the data is not present in cache (or is invalid), the proxies forwards the request to the inner tiers and fetches the requested document. Thus, data is only fetched to the outer tiers when requested by the client. The advantage of this approach is that the document is fetched into cache only when it is requested or required. The front tier is completely in control of the cache state and can decide the appropriate eviction policy to use based on the requests arriving, etc. Since the front tiers serve all the requests coming in from the clients, the eviction policies based on the request information and client access patterns can be expected to be the most reliable. On the negative side, this model forces the fetching of the data from the inner tiers to fall in the critical path of the client's response time.

4.2 Server-driven mechanism (Data Push Model)

In the Data Push model, the back-end nodes play an active role in updating the outer tiers' caches in case of an update. As soon as an update takes place, the back-end nodes sends a notification to the outer tiers. The notification can either be through invalidation or the server can propagate the updated file. The disadvantage of this model is that the data fetch still falls in the critical path of the client's response time. In this paper, we focus on the data propagation based push model.

The advantage of this approach is that the fetch of the data does not fall in the critical path of the client's response time, so it can be expected to be low. The main disadvantage of this model is the cache state modification which accompanies the transfer of data. Each server in the system has limited resources. So, there is a limit to what the server can store in its cache before it has to evict someone else for space. And at any given instant of time the server will keep only the files that will most likely be reused in its cache. The inner tiers are not aware of the requests coming in at the proxy. This forces them to make caching decisions based on very limited information on the requests coming in and the client access patterns. In other words, the view of the cache state as seen by the outer tiers and the inner tiers can be completely different. So, pushing data into the cache of the proxy without being aware of all the details may potentially increase the number of cache misses by budging out frequently used data from the cache to fit in the updated data, cause thrashing of the cache entries and could result in significant hampering of the overall performance of the system.

4.3 Hybrid Mechanism (Adaptive)

Generally, the push model is preferred when there are strict consistency requirements and the pull model for decreased server loads when lazy consistency is acceptable. It is to be noted, however, that for cache coherent systems (similar to the one we are considering), functionally either of the schemes could used. The only concern in this case would be the performance provided by each of the schemes.

Due to the advantages and disadvantages associated with each of the Push and the Pull models, researchers have come up with a number of hybrid approaches where an application can choose between push and pull model on the fly [13], with a view to take the best of both the approaches. However, all these approaches are still based on the inner-tiers' view of the outer-tiers' cache state and lack a global view of the cache state.

In this paper, we propose a cache state implementation logically shared between the multiple tiers of the data-center. The design consists of a cache state maintained at each tier in the data center environment except the last tier. For simplicity, we explain only the shared state maintained between the proxy and application server. Since every request goes through the proxy module, the proxy module can maintain a cache state by having a history of every request accessed and also dynamically re-assign weights to these requested files. A typical cache state at proxy module has a table of the requested files, its timestamp, frequency and an associated weight.

When a document gets updated, the update server notifies the change to the application module. In the implementation, we have separate threads listening for requests from update server (*Update Thread*), application server (*Application Module Main Thread*) and also from proxy server (*Version Thread*). These threads communicate with each other via IPCs. After the module gets an update notification, it first checks the cache state of the proxy server, either by contacting the *Proxy Module* (for TCP/IP) or by using RDMA read (for InfiniBand) and gets the current state of the proxy's cache. The decision to push the updated data item to proxy nodes is made based on the current state. After pushing the data to the proxy server, all subsequent requests to that file are in cache.

On getting client requests at the proxy module, the proxy module checks with the application module for the version number and if the file is not modified, the server can serve the request from cache. If an update is in progress, the version numbers at proxy and application module would be different and as explained before the requests will be forwarded to the application server. This design largely helps the application module to adaptively push the data under dynamically varying conditions at the proxy. Figure 7 illustrates the proposed shared cache architecture.

Implementing such a shared state over sockets is very expensive and adds a lot of overhead at the proxy server. However, for protocols based on RDMA operations there is almost zero overhead associated with the peer node and less communication overhead. In this design, we have used the RDMA operations provided by InfiniBand.

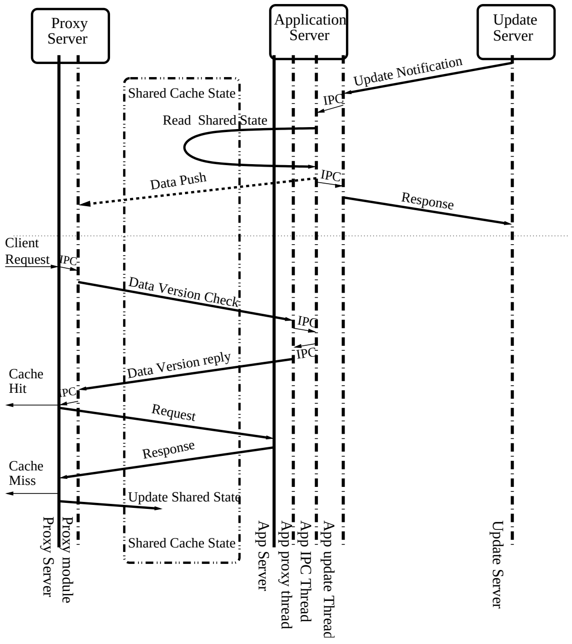

Figure 7. Shared Cache State based Adaptive Push-Pull Model)

5 Experimental Results

In this section, we provide three sets of results. First we show the micro-benchmark level performance given by the native Verbs layer over InfiniBand (VAPI), the Sockets Direct Protocol over InfiniBand (SDP) and that given by the kernel TCP/IP stack over InfiniBand (IPoIB). Next, we analyze the performance of a cache-coherent 2-tier data-center environment. Cache coherence is achieved using the *Client-Polling* based approach in the architecture described in Section 3. Lastly, we compare the performance of our adaptive push-pull model based on shared cache state with that of the native push and pull models.

All our experiments used the following experimental testbed. A cluster system consisting of 8 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 L2 cache and a 400 MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 Dual-Port 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.2.0-build-001. The adapter firmware version is fw-23108-rel-1_18_0000. We used the Linux 2.4.7-10 kernel.

5.1 Micro-benchmarks

In this section, we compare the ideal case performance achievable by IPoIB and InfiniBand VAPI using a number of micro-benchmark tests.

Figure 8a shows the one-way latency achieved by IPoIB, VAPI Send-Receive, RDMA Write, RDMA Read and SDP for various message sizes. Send-Receive achieves a latency of around 7.5μs for 4 byte messages compared to a 30μs achieved by IPoIB, 27μs achieved by SDP and 5.5μs and 10.5μs achieved by RDMA Write and RDMA Read, respectively. Further, with increasing message sizes, the difference between the latency achieved by native VAPI, SDP and IPoIB tends to increase.

Figure 8b shows the uni-directional bandwidth achieved by IPoIB, VAPI Send-Receive and RDMA communication models and SDP. VAPI Send-Receive and both RDMA models perform comparably with a peak throughput of up to 840Mbytes/s compared to the 169Mbytes/s achieved by IPoIB and 500Mbytes/s achieved by SDP. We see that VAPI is able to transfer data at a much higher rate as compared to IPoIB and SDP. This improvement in both the latency and the bandwidth for VAPI compared to the other protocols is mainly attributed to the zero-copy communication in all VAPI communication models.

5.2 Strong Cache Coherence

In this section, we analyze the performance of a cache-coherent 2-tier data-center environment consisting of three proxy nodes and one application server running Apache-1.3.12. Cache coherency was achieved using the *Client-Polling* based approach described in Section 3. We used three client nodes, each running three threads, to fire requests to the proxy servers.

Three kinds of traces were used for the results. The first trace consists of a single 8Kbyte file. This trace shows the ideal case performance achievable with the highest possibility of cache hits, except when the document is updated at the back-end. The second trace consists of 20 files of sizes varying from 200bytes to 1Mbytes. The access frequencies for these files follow a Zipf distribution [29]. The third trace is a 20000 request subset of the WorldCup trace [3]. For all experiments, accessed documents were randomly updated by a separate update server with a delay of one second between the updates.

The HTTP client was implemented as a multi-threaded parallel application with each thread independently firing requests at the proxy servers. Each thread could either be executed on the same physical node or on a different physical nodes. The architecture and execution model is similar to the WebStone workload generator [20].

As mentioned earlier, application servers are typically compute intensive mainly due to their support to several compute intensive applications such as CGI script execution, Java applets, etc. This typically spawns several compute threads on the application server node using up the CPU resources. To emulate this kind of behavior, we run a number of compute threads on the application server in our experiments.

Figure 9a shows the client response time for the first trace (consisting of a single 8Kbyte file). The x-axis shows the number of compute threads running on the application server node. The figure shows an evaluation of the proposed architecture implemented using IPoIB, SDP and VAPI and compares it with the response time obtained in the absence of a caching mechanism. We can see that the proposed architecture performs equally well for all three (IPoIB, SDP and VAPI) for a low number of compute threads; All three achieve an improvement of a factor of 1.5 over the no-cache case. This shows that two-sided communication is not a huge bottleneck in the module as such when the application server is not heavily loaded.

As the number of compute threads increases, we see a considerable degradation in the performance in the no-cache case as well as the Socket-based implementations using IPoIB and SDP. The degradation in the no-cache case is quite expected, since all the requests for documents are forwarded to the back-end. Having a high compute load on the back-end would slow down the application server's replies to the proxy requests.

The degradation in the performance for the Client-Polling architecture with IPoIB and SDP is attributed to the two sided communication of these protocols and the context switches taking place due to the large number of threads. This results in a significant amount of time being spent by the application modules just to get access to the system CPU. It is to be noted that the version thread needs to get access to the system CPU on every request in order to reply back to the proxy module's version number requests.

On the other hand, the Client-Polling architecture with VAPI does not show any significant drop in performance. This is attributed to the one-sided RDMA operations supported by InfiniBand. For example, the version number retrieval from the version thread is done by the proxy module using a RDMA Read. That is, the version thread does not have to get access to the system CPU; the proxy thread can retrieve the version number information for the requested document without any involvement of the version thread.

Figure 9b shows the throughput achieved by the data-center for the proposed architecture with IPoIB, SDP, VAPI and the no-cache cases. Again, we observe that the architecture performs equally well for both Socket based implementations (IPoIB and SDP) as well as VAPI for a low number of compute threads with an improvement of a factor of 1.67 compared to the no-cache case. As the number of threads

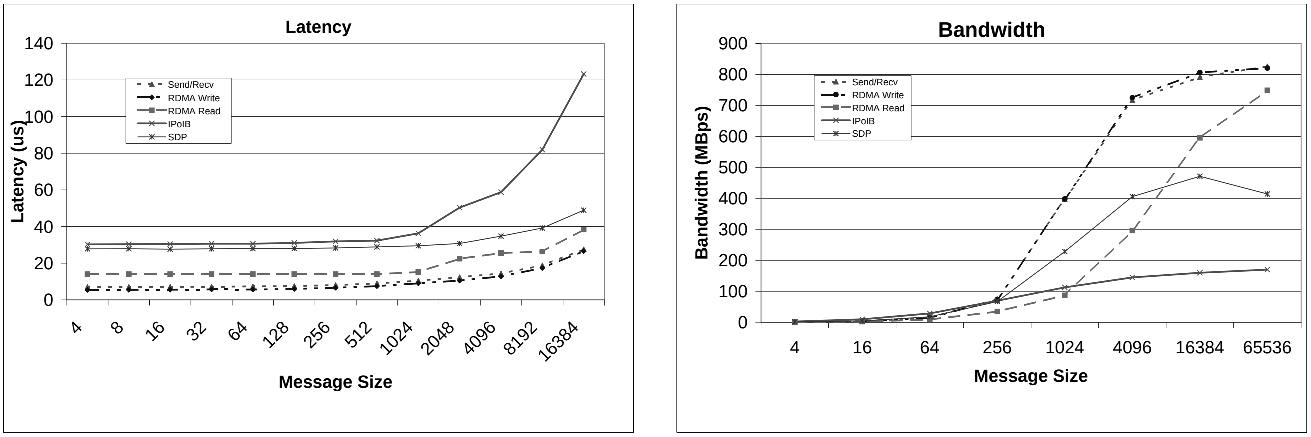

Figure 8. Micro-Benchmarks: (a) Latency, (b) Bandwidth

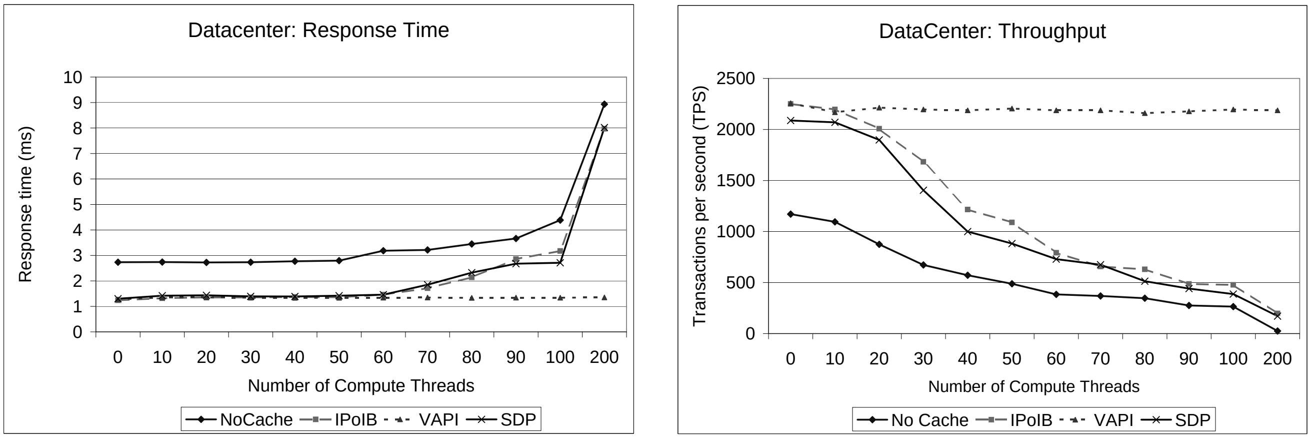

Figure 9. Strong Cache Coherence in Data-Centers; Performance Analysis: (a) Client Response Time, (b) Request Throughput

increases, we see a significant drop in the performance for both IPoIB and SDP based client-polling implementations as well as the no-cache case, unlike the VAPI-based client-polling model, which remains almost unchanged. This is attributed to the same reason as that in the response time test, i.e., no-cache and Socket based client-polling mechanisms (IPoIB and SDP) rely on a remote process to assist them. With a large number of compute threads already competing for the CPU, the wait time for this remote process to acquire the CPU can be quite high, resulting in this degradation of performance. To demonstrate this, we look at the component wise break-up of the response time.

Figure 10a shows the component wise break-up of the response time observed by the client for each stage in the request and the response paths, using our proposed architecture on IPoIB, SDP and VAPI, when the backend has no compute threads and is thus not loaded. In the response time breakup, the legends *Module Processing*, and *Backend Version Check* are specific to our architecture. We can see that these components together add up to less than 10% of the total time. This shows that the computation and communication costs of the module as such do not add too much overhead on the client's response time.

Figure 10b on the other hand, shows the component wise break-up of the response time with a heavily loaded back-end server (with 200 compute threads). In this case, the module overhead increases significantly for IPoIB and SDP, comprising almost 70% of the response time seen by the client, while the VAPI module overhead remains unchanged by the increase in load. This indifference is attributed to the one-sided communication used by VAPI (RDMA Read) to perform a version check at the backend. This shows that for two-sided protocols such as IPoIB and SDP, the main overhead is the context switch time associated with the multiple applications running on the application server which skews this time (by adding significant wait times to the modules for acquiring the CPU).

Figures 11a and 11b show the throughput obtained by the data-center environment using the Zipf based trace (Trace2) and a subset of the world-cup trace (Trace3) respectively. We see similar observations, as the previous trace, for these traces. This shows that the proposed scheme is robust across workload formats and is not specific to a given kind of workload.

5.3 Adaptive Push-Pull Model

In this section, we analyze the performance of our shared cache state based adaptive push-pull architecture in two aspects: (a) The number of wasted updates and (b) The number of cache misses. We compare the proposed architecture with the pure Data Push based and Data Pull based models.

The number of wasted updates refers to scenarios where two successive updates do not have any request for the document in between them. This essentially means that the first update was not used for any client request. Though not completely precise, this metric is expected to capture the number of unneccessary data pushes performed by the back-end server.

The number of cache misses is a more direct metric which captures the actual impact of the cache replacement policy (based on a given model) on the response time observed by the client.

The main idea of our proposed architecture is to have a shared cache state which would present a global picture of the front-tiers' cache state to the back-end nodes. Based on this information, any amount of intelligence can be added into the back-end nodes to develop efficient data push-pull hybrids, i.e., the web-server can decide based on the cache state whether it would push the data to the front tiers or if it would wait for the front-tiers to fetch it from the back-end on a request.

As mentioned earlier, for this implementation, we have used a simple scheme in which the front-end nodes keep track of their current list of *most recently accessed documents*. The back-end nodes either contact the *proxy module* (for IPoIB or SDP) or perform a RDMA Read on the *proxy module's* memory (for VAPI) to get the required information about the cache. If the currently updated file is within this list of the *most recently accessed documents*, the data is pushed to the proxy module. Otherwise, the application server just waits for the proxy to access the updated document on a client request.

Figure 12a shows the comparison of the number of wasted data pushes occurring in the pull model, push model and hybrid models where the back-end only updates certain percentange of recently accessed files. Figure 12b shows the comparison of the cache misses in each model. Figures 13 and 14 show a similar analysis for SDP and VAPI respectively. It is to be noted that in the *Pull Model*, the back-end nodes do not push any data to the front-end nodes, so there are no wasted updates. Similarly, the *Push Model* always pushes data to all proxy nodes who have an version of the document. Though this does not guarantee that there will be no cache misses (for example, compulsory misses if the proxy does not have any older version of the document in its cache, there would be a cache miss), it would minimize the number of cache misses.

It can be seen that the *Pull Model* does not incur any wasted updates, but suffers from a high cache miss rate. Similarly, the *Push Model* incurs minimal cache misses, but suffers from a large number of wasted updates. Further we notice that even with a simplistic scheme to make the data-push decision, the hybrid scheme is able to achieve an efficient trade-off between the number of cache misses and the number of wasted updates. We are currently working on

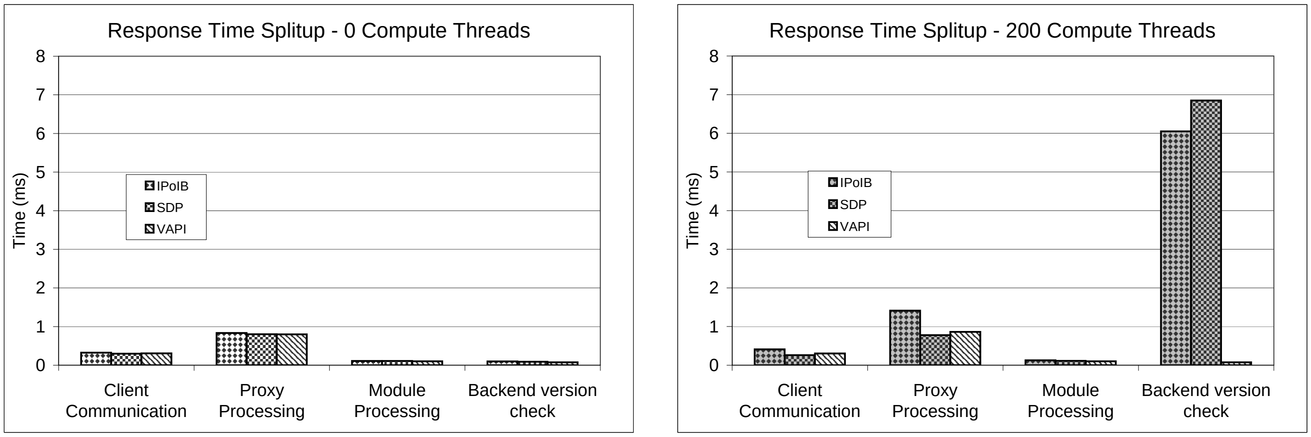

Figure 10. Data-Center Response Time Breakup: (a) 0 Compute Threads, (b) 200 Compute Threads

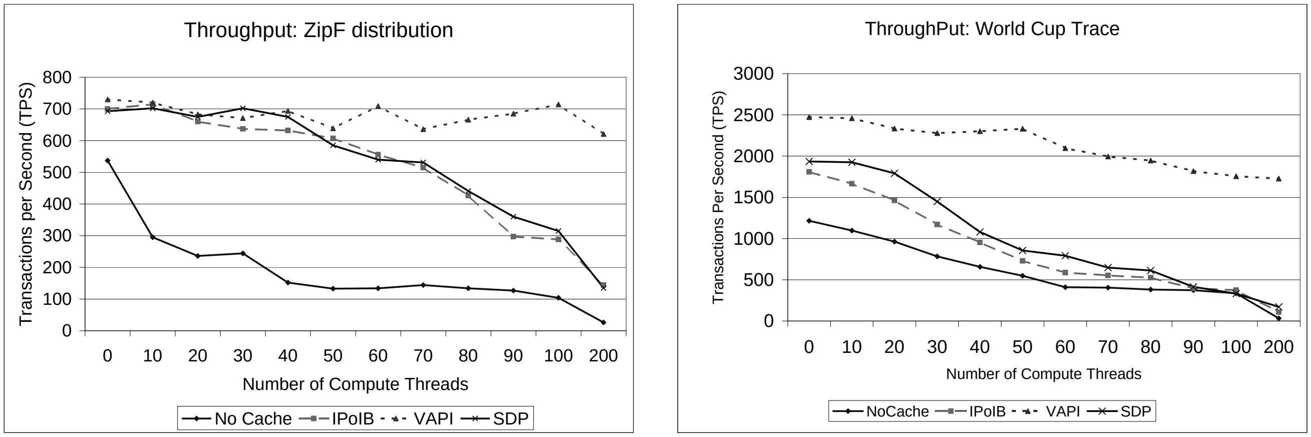

Figure 11. Data-Center Throughput: (a) Zipf Distribution, (b) WorldCup Trace

integrating more advanced schemes into the current framework and expect to get a significantly better performance in both the number of wasted updates as well as the number of cache misses.

6 Conclusions and Future Work

Data-centers are central to providing high performance, highly scalable, and highly available web services. Reducing computation and communication overhead is crucial to improve performance and scalability of data-centers. Caching content at various tiers of a multi-tier data-center is a well known method to reduce the computation and communication overhead. In the current web, many cache policies and uncachable resources are driven by two server application goals: Cache Coherence and Cache Consistency. The problem of how to provide consistent caching for dynamic content has been well studied and researchers have proposed several weak as well as strong consistency algorithms. However, the problem of maintaining cache coherence has not been studied as much.

In this paper, we proposed an architecture for achieving strong cache coherence based on the client-polling mechanism for multi-tier data-centers over InfiniBand. The architecture as such could be used with any protocol layer; we also proposed optimizations to better implement it over InfiniBand by taking advantage of RDMA operations. We evaluated this architecture using three protocol platforms: (i) TCP/IP over InfiniBand (IPoIB), (ii) Sockets Direct Protocol over InfiniBand (SDP) and (iii) the native InfiniBand Verbs layer (VAPI) and compared it with the performance of the no-caching based coherence mechanism. Our experimental results show that the optimized architecture over InfiniBand can achieve an improvement of nearly an order of magnitude for the throughput achieved by the TCP/IP based architecture, the SDP based architecture and the no-cache based coherence scheme. The results also demonstrate that the implementation based on RDMA communication mechanism can offer better performance robustness to load of the data-center servers.

We also proposed an adaptive push-pull architecture based on shared cache states for propagating updates. Again, we evaluated this architecture over TCP/IP over InfiniBand (IPoIB), Sockets Direct Protocol over InfiniBand (SDP) and VAPI. Our experimental results show that even with a simplistic scheme to make the data-push decision, all the implementations of the adaptive push-pull model (over TCP/IP, over SDP and over VAPI) are able to achieve an efficient trade-off between the number of cache misses and the number of wasted updates.

As a future work, we propose to combine InfiniBand RDMA and Atomic operations to efficiently support load balancing and virtualization in the data-center environment.

7 Acknowledgments

We would like to thank NAPS Srinivas, Amith Mamidala, Gopalakrishnan Santhanaraman and Sayantan Sur for all the technical support they extended during the course of the project.

References

[1] InfiniBand Trade Association. http://www.infinibandta.com.

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

[3] The Internet Traffic Archive. http://ita.ee.lbl.gov/html/traces.html.

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

[5] Hossein Sheikh Attar and Yaya Yang. Strong Cache Consistency for Dynamic Web Applications.

[6] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, March 10-12 2004.

[7] P. Balaji, P. Shivam, P. Wyckoff, and D.K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Cluster Computing*, September 2002.

[8] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *the Proceedings of the IEEE International Conference on High Performance Distributed Computing (HPDC 2003)*, June 2003.

[9] Adam D. Bradley and Azer Bestavros. Basis Token Consistency: Extending and Evaluating a Novel Web Consistency Algorithm. In *the Proceedings of Workshop on Caching, Coherence, and Consistency (WC3)*, New York City, 2002.

[10] Adam D. Bradley and Azer Bestavros. Basis token consistency: Supporting strong web cache consistency. In *the Proceedings of the Global Internet Worshop*, Taipei, November 2002.

[11] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the Web. In *Middleware Conference*, 1998.

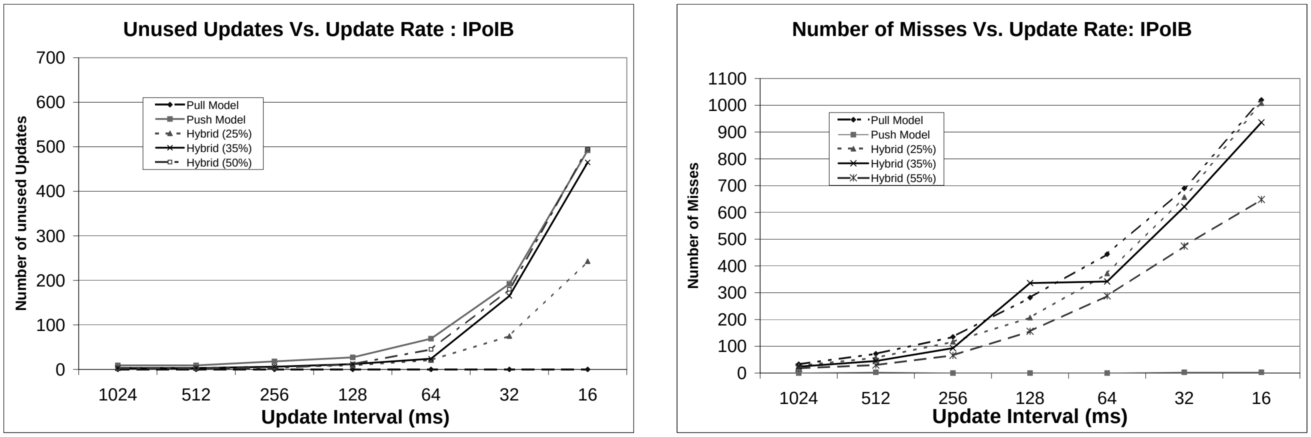

Figure 12. Push Models for IPoIB: (a) Number of Wasted Updates for Different Models, (b) Number of Misses for Different Models

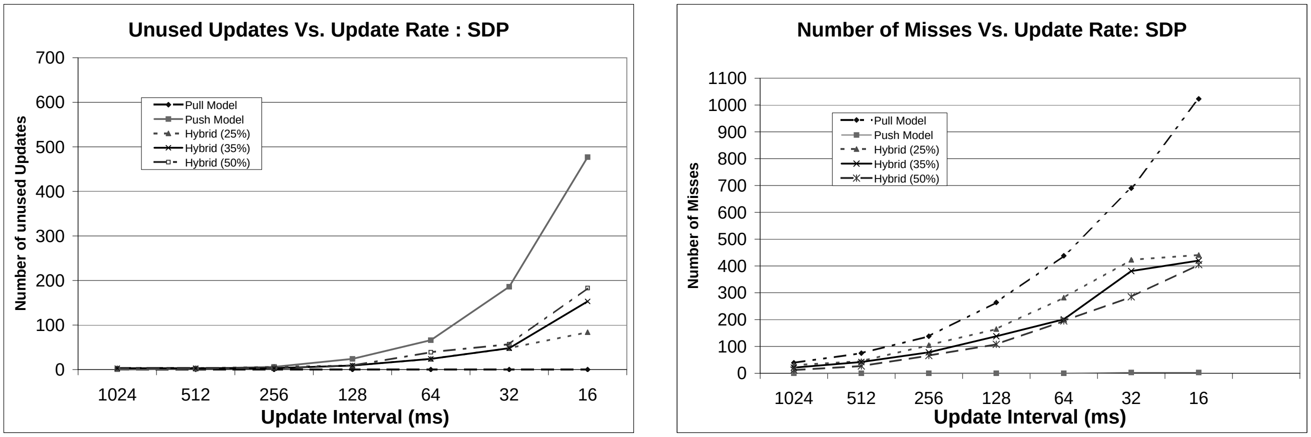

Figure 13. Push Models for SDP: (a) Number of Wasted Updates for Different Models, (b) Number of Misses for Different Models

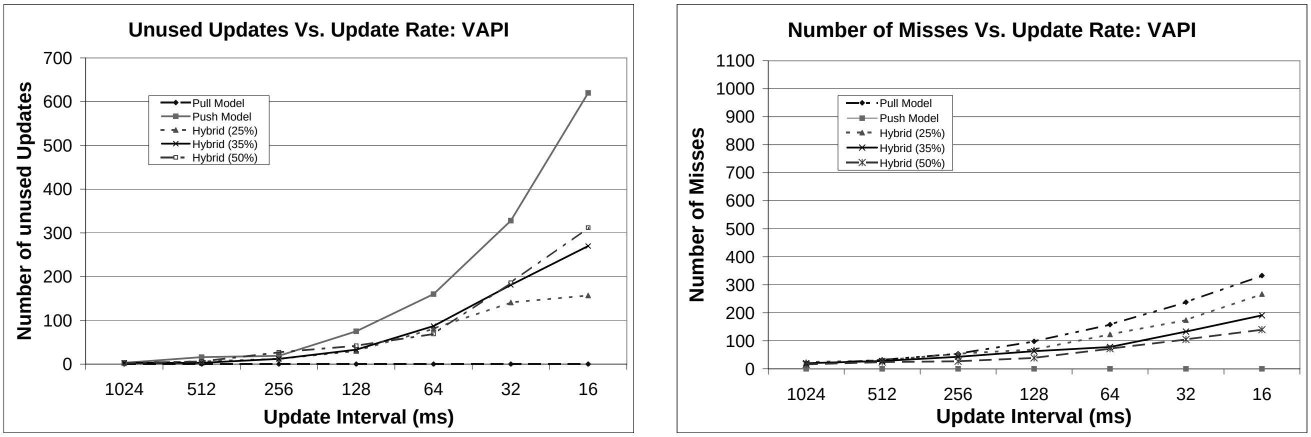

Figure 14. Push Models for VAPI: (a) Number of Wasted Updates for Different Models, (b) Number of Misses for Different Models

[12] 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.

[13] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant J. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *World Wide Web*, pages 265–274, 2001.

[14] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-Level Communication in Cluster-Based Servers. In *the 8th IEEE International Symposium on High-Performance Computer Architecture (HPCA 8)*, Feb. 2002.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP 1.1. RFC 2616. June, 1999.

[16] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *Proceedings of Cluster Computing*, 2001.

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

[18] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing*, June 2003.

[19] Mikhail Mikhailov and Craig E. Wills. Evaluating a New Approach to Strong Web Cache Consistency with Snapshots of Collected Content. In *WWW2003, ACM*, 2003.

[20] Inc Mindcraft. http://www.mindcraft.com/webstone.

[21] Jeffrey C. Mogul. Clarifying the fundamentals of HTTP. In *the Proceedings of WWW-2002*, Honolulu, HI, May 2002.

[22] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *Proceedings of CANPC workshop*, 1999.

[23] Hemal V. Shah, Dave B. Minturn, Annie Foong, Gary L. McAlpine, Rajesh S. Madukkarumukumana, and Greg 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.

[24] Weisong Shi, Eli Collins, and Vijay Karamcheti. Modeling Object Characteristics of Dynamic Web Content. *Special Issue on scalable Internet services and architecture of Journal of Parallel and Distributed Computing (JPDC)*, Sept. 2003.

[25] Mellanox Technologies. InfiniBand and TCP in the Data-Center.

[26] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Oct. 2001.

[27] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[28] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Web Cache Consistency. *ACM Transactions on Internet Technology, 2:3,*, August. 2002.

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