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.

1 Introduction

With increasing adoption of the Internet as primary means of electronic interaction and communication, E-portal and E-commerce, highly scalable, highly available and high performance web servers, 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 represents a typical multi-tier data-center. The front tiers consist of front-end servers such as proxy servers that provide web, messaging and various other services to clients. The middle tiers usually comprise of application servers that handle transaction processing and implement data-center business logic. The back-end tiers consist of database servers that hold a persistent state of the databases and other data repositories. As mentioned in [16], 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.

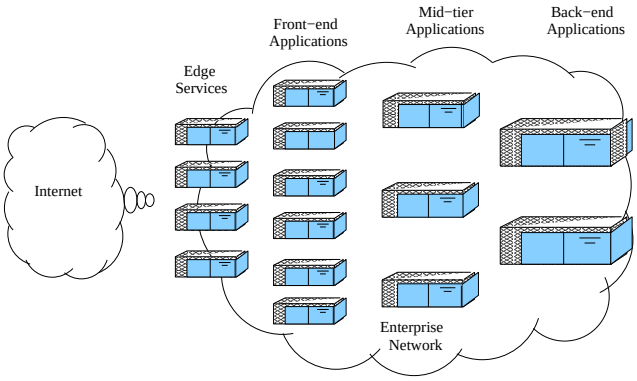

Figure 1. A Typical Multi-Tier Data-Center (Courtesy CSP Architecture design [16])

With ever increasing on-line businesses and services and the growing popularity of personalized Internet services, dynamic content is becoming increasingly common [7, 21, 17]. This includes documents that change upon every access, documents that are query results, documents that embody client-specific information, etc. Large-scale dynamic workloads pose interesting challenges in building the next-generation data-centers [21, 16, 9, 18]. Significant compu-

*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

tation and communication may be required to generate and deliver dynamic content. Performance and scalability issues need to be addressed for such workloads.

Reducing computation and communication overhead is crucial to improving the performance and scalability of data-centers. Caching dynamic content, typically known as *Active Caching* [7] at various tiers of a multi-tier data-center is a well known method to reduce the computation and communication overheads. However, it has its own challenges: issues such as cache consistency and cache coherence become more prominent. 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 [10], Adaptive TTL [8], and Invalidation [11] 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 re-fetched 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 [16, 9, 4]. 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 for 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 interaction with the Operating System or processor intervention by using DMA engines. These two features have been leveraged in designing high performance message passing systems [12] and cluster file systems [20].

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.

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 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) and evaluate their performance compared to that achieved by the no-caching based coherence mechanism.

3. Our experimental results show that the VAPI based architecture can achieve an improvement of nearly an order of magnitude over the throughput achieved by the other implementations of the architecture and the no-cache based coherence scheme. Our results also show that this one-sided communication based architecture is mostly resilient and well-conditioned to the load on the application servers as compared to two-sided protocols such as IPoIB and SDP. 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 several opportunities to revise the design and implementation of many subsystems, protocols, and communication mechanisms in the data-center environment. The rich features of 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. The experimental results are presented in Section 4. We draw our conclusions and discuss possible future work in Section 5.

2 Background

In this section, we briefly describe the various schemes previously proposed by researchers to allow *bounded staleness* to the accessed documents, maintaining strong consistency, etc. Background details about InfiniBand and SDP have been skipped due to space restrictions and can be found in [15].

2.1 Web Cache Consistency and Coherence

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 [7]. 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). 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 based 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* [8] and *MONARCH* [13] 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 [6, 5].

2.1.2 Web Cache Coherence

Typically, when a request reaches the proxy node, the cache is checked for the file. If the file was previously requested and cached, 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% [17]. Majority of the cache misses are primarily due to the dynamic nature of web requests. Caching dynamic content is much more challenging than static content because the cached object is related to data at the back-end tiers. This data may change, thus invalidating the cached object and resulting in a cache miss. The problem providing consistent caching for dynamic content has been well studied and researchers have proposed several weak as well as strong cache consistency algorithms [6, 5, 21]. However, the problem of maintaining cache coherence has not been studied as much.

The two popular coherency models used in the current web are *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.

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
- Propagation of these requests to the back-end nodes 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 also has 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.

3 Providing Strong Cache Coherence

In this section, we describe the architecture we use to support strong cache coherence. 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.

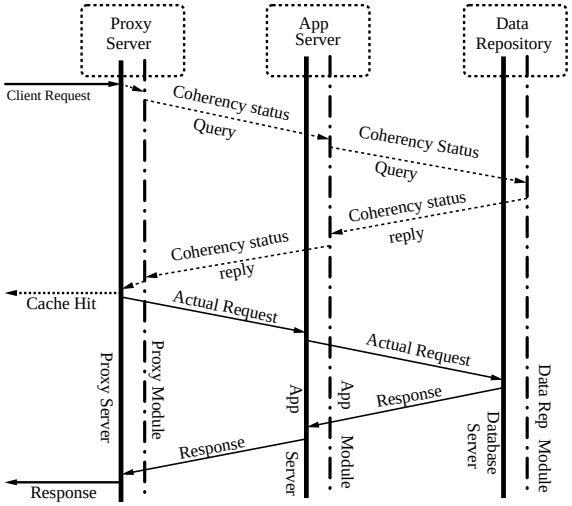

Figure 2. Strong Cache Coherence Protocol

In the light of these issues, we present an alternative architecture to perform *Client-Polling*. Figure 2 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.

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 num-

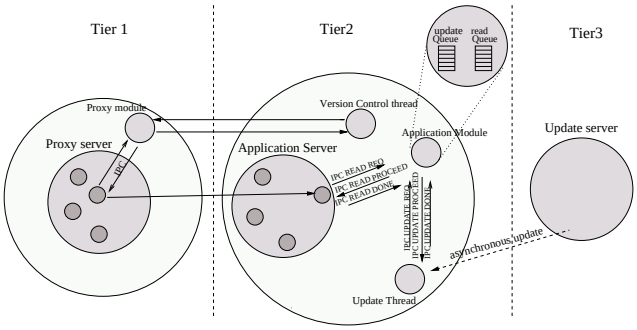

Figure 3. Interaction between Data-Center Servers and Modules

ber 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 3 demonstrates the functionality of the different modules and their interactions.

Proxy Module: On every request, the proxy server contacts the proxy module through IPC to validate the cached object(s) associated with the request. The proxy module does the actual verification 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 4 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).

[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.

Application servers typically tend to perform several computation intensive tasks including executing CGI scripts, 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 4 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.

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 [17, 19]. 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.

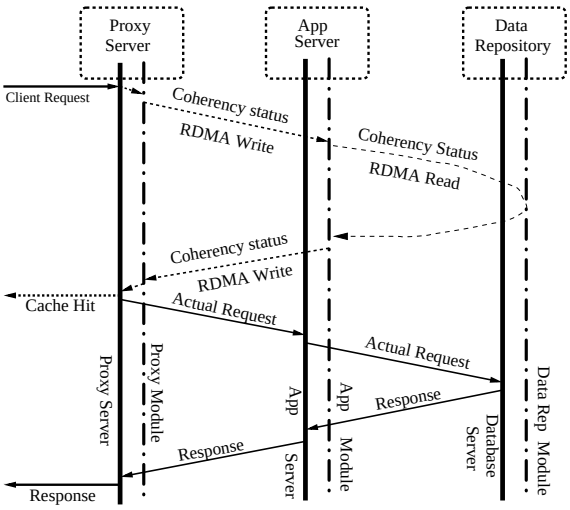

Figure 4. Strong Cache Coherency Protocol: InfiniBand based Optimizations

4 Experimental Results

In this section, we first show the micro-benchmark level performance given by VAPI, SDP and 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.

All our experiments used 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 DualPort 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.

4.1 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 5 a 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 5b 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.

4.2 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 [22]. 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 [14].

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 6a 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

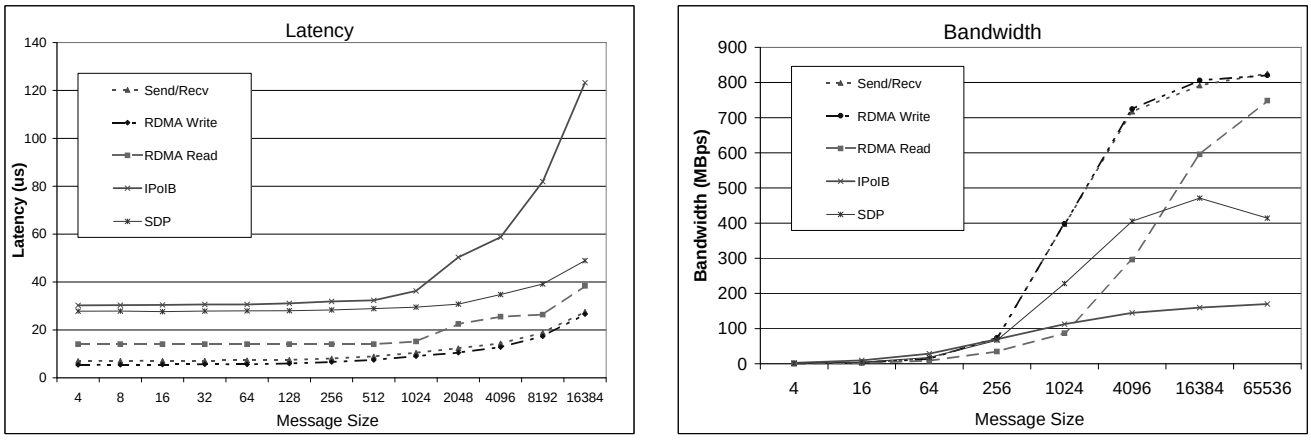

Figure 5. Micro-Benchmarks: (a) Latency, (b) Bandwidth

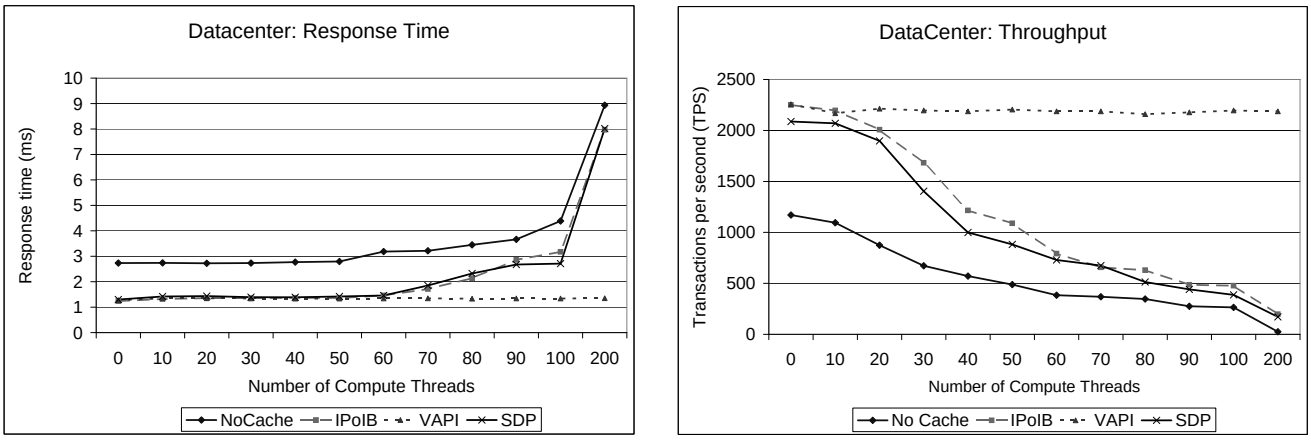

Figure 6. Data-Centers Performance Analysis

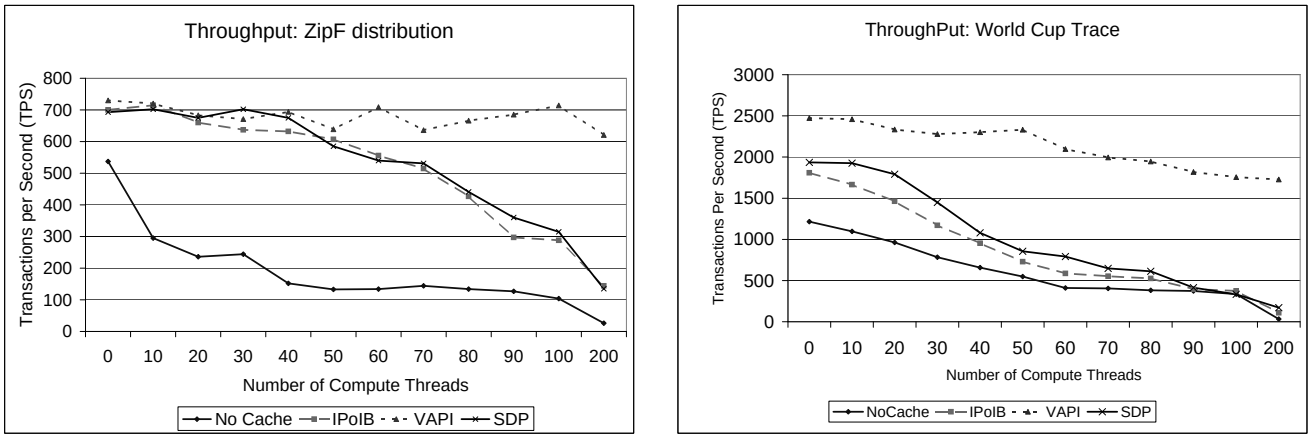

Figure 7. Data-Center Throughput: (a) Zipf Distribution, (b) WorldCup Trace

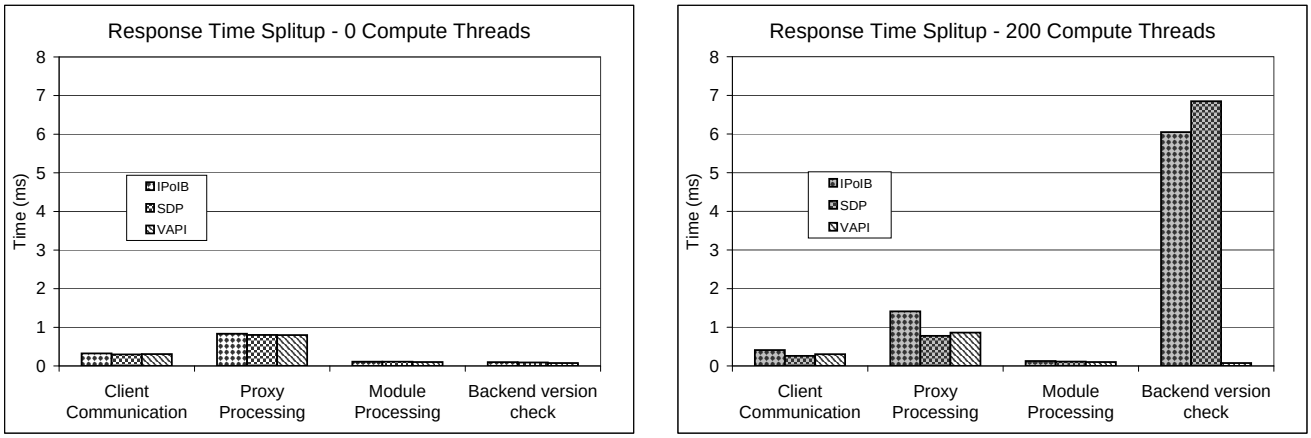

Figure 8. Data-Center Response Time Breakup: (a) 0 Compute Threads, (b) 200 Compute Threads

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. These observations are re-verified by the response time breakup provided in Figure 8.

Figure 6b 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 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. The throughput achieved by the WorldCup trace (Figure 7b) and the trace with Zipf distribution (Figure 7a) also follow the same pattern as above. 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 8a 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 8b 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).

5 Conclusions and Future Work

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 previously proposed client-polling mechanism for multi-tier data-centers. 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 one sided operations such as RDMA. 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 InfiniBand-optimized architecture can achieve an improvement of upto a factor of two for the response time and 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 the load of the data-center servers.

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.

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] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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. Technical Report OSU-CISRC-11/03-TR65, The Ohio State University, 2003.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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