Architecture for Caching Responses with Multiple Dynamic Dependencies in Multi-Tier Data-Centers over InfiniBand[*]

S. Narravula P. Balaji K. Vaidyanathan H. -W. Jin D. K. Panda
Department of Computer Science and Engineering
The Ohio State University
{narravul, balaji, vaidyana, jinhy, panda}@cse.ohio-state.edu

Abstract

It has been well acknowledged in the research community that in order to 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. However, for caching dynamically changing content (e.g., content involved in online banking, Internet auctions, etc.), consistency and coherency issues need to be addressed. In addition, most current real world requests have multiple dynamic dependencies, i.e., these requests might depend on multiple data objects. Further, these requests are not entirely independent; several requests might have common dependencies. While there have been previous research solutions on maintaining coherent caches for dynamic content, these solutions have several shortcomings including inability to adapt to server load or handle multiple dynamic dependencies. In this paper, we propose a load resilient architecture using one sided operations supported by several high performance interconnects such as InfiniBand, while maintaining multiple dynamic dependencies per response. Our experimental results show that our schemes to tackle the multi-dependency issue efficiently and significantly outperform the existing approaches. Further, our results demonstrate that the proposed load resilient architecture can possibly improve the performance of loaded data-centers by over an order of magnitude.

Keywords: *Multi-Tier Data-Center, InfiniBand, Caching, Dynamic Content Caching, Coherency*

1 Introduction

The unprecedented growth of Internet has deeply infiltrated all of today's society. With more and more people using the Internet for a wide range of purposes, Internet use has become an absolute necessity for businesses to survive and grow. Electronic communications, e-commerce, online services, etc. have become ubiquitous and are growing in complexity in terms of both raw data content and processing required. Consequently, high performance and scalable web-servers have become critical tools to deliver these requirements.

On the other hand, Cluster systems have become the main system architecture for a number of environments mainly due to their high performance-to-cost ratio. In the past, they had replaced mainstream supercomputers as a cost-effective alternative in a number of scientific domains. During the last few years, research and industry communities have been proposing and implementing several high performance communication systems to address some of the problems associated with the traditional networking protocols for cluster-based systems. InfiniBand Architecture (IBA) [2] has been recently standardized by the industry in the light of next generation high-end clusters designs.

IBA is envisioned as the default interconnect for several environments in the near future. IBA provides two key features, namely *User-level Networking* and *One-Sided Communication Operations*. User-level Networking allows applications to directly and safely access the network interface without going through the operating system. One-sided communication allows the network interface to transfer data between local and remote memory buffers without any interaction with the operating system and the processor. It also provides features for performing network based atomic operations on the remote memory regions. These can be leveraged in providing efficient support for multiple environments [15, 22].

Based on these two trends, several researchers [21, 1] have proposed the feasibility and potential of cluster-based multi-tier data-centers to become the fundamental instruments to providing these Internet services.

Figure 1 shows a typical cluster-based multi-tier data-center. The various nodes in the typical data-center are logically partitioned to provide various related services including web and messaging services, transaction processing, business logic, databases, etc. End user requests are processes as a collective effort by these nodes.

These services include online services like personalized services, e-commerce based services, etc. which have recently increased several folds in volume. Scenarios like online banking, auctions, etc. are constantly adding to the complexity of content being served on the Internet. The responses generated for these can change depending on the request and are typically known as dynamic or active content. Multi-tier data-centers process these complex requests by breaking-up the request processing into several stages with each data-center tier handling a different stage of request processing. With the current processing needs and growth trends in mind, the scalability of data-centers has become an important issue.

Traditionally, caching has been an important technique to improve scalability and performance of data-centers. However, simple caching methods are clearly not applicable for dynamic content caching. Documents of dynamic nature are typically generated by processing one or more data objects stored in the back-end database, i.e., these documents are dependent on several persistent data objects. These persistent data objects can also be a part of multiple dynamic documents. So in effect these documents and data objects have several many to many mappings between them. Thus, any change to one individual object can potentially affect the validity of multiple cached requests.

In our previous work [17], we have presented a simple architecture that supports strong cache coherency for proxy caches. However, [17] presents a simplistic scheme for strong cache coherency

[*]This research is supported in part by NSF grants #CCR-0204429 and #EIA-9986052.

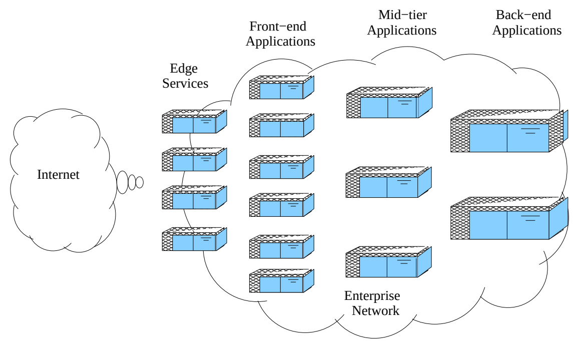


Figure 1. A Typical Multi-Tier Data-Center (Courtesy CSP Architecture design [21])

which only deals with a file level granularity for coherency, i.e., each update affects an object which can be a part of one or more cached requests. However, most data-centers allow and support more complex web documents comprising of multiple dynamic objects. These additional issues necessitate more intricate protocols to enable dynamic content caching and make the design of strongly coherent caches extremely challenging. Further, since an updated object can potentially be a part of multiple documents across several servers, superior server coordination protocols take a central role in these designs.

In this paper, we present a complete architecture to support strong cache coherency for dynamic content caches. Our architecture is designed to handle caching of responses composed of multiple dynamic dependencies. We propose a complete architecture to handle two issues: (i) caching documents with multiple dependencies and (ii) being resilient to load on servers. We also study the effect of varying dependencies on these cached responses. Our experimental results show more than 20 times improvement for the overall data-center throughput using our caching techniques. Also, our design can sustain high performance for overall data-center requests while maintaining strong coherency with multiple object dependencies even under heavy load.

The rest of the paper is organized as follows: Section 2 gives a brief background on multi-tier data-centers and web cache coherency and consistency. Section 3 elaborates the design of the basic architecture. The experimental results are presented in section 4. Section 5 briefly covers related research work, followed by conclusions and future work.

2 Background

In this section, we provide a brief background of Multi-Tier Data-Centers and Web-Cache Coherency and Consistency.

2.1 Multi-Tier Data-Centers

A typical data-center architecture consists of multiple tightly interacting layers known as tiers. Each tier can contain multiple physical nodes. Figure 1 shows a typical Multi-Tier Data-Center. Requests from clients are load-balanced by the edge services tier on to the nodes in the proxy tier. This tier mainly does caching of content generated by the other back-end tiers. The other functionalities of this tier can include data-center security and balancing the request load sent to the back-end based on certain pre-defined algorithms.

The second tier consists of two kinds of servers. First, those which host static content such as documents, images, music files and others which do not change with time. These servers are typically referred to as web-servers. Second, those which compute results based on the query itself and return the computed data in the form of a static document to the users. These servers, referred to as application servers, usually handle compute intensive queries which involve transaction processing and implement the data-center business logic.

The last tier consists of database servers. These servers hold a persistent state of the databases and data repositories. These servers could either be compute intensive or I/O intensive based on the query format. Queries involving non key searches can be more I/O intensive requiring large data fetches into memory. For more complex queries, such as those which involve joins or sorting of tables, these servers can be more compute intensive.

2.2 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 [9]. 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. So, either every client sees an update or no client sees that particular update.

Researchers have proposed several different schemes providing several different levels of consistency. TTL [12], Adaptive TTL [10] and MONARCH [16], present schemes for lazy or delayed consistency. Schemes for strong cache consistency are detailed in [6, 9, 17].

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.

On lines similar to web-cache consistency, two popular coherence models are used: *immediate* or *strong coherence* and *bounded staleness*. With *strong coherence*, caches are forbidden from returning a response other than that which would be returned were the origin server contacted. Since, in effect, the origin server is contacted for each request, as a side effect, *Strong Cache Coherency* also guarantees *Strong Cache Consistency*.

2.2.1 Maintaining Cache Coherence

Strong cache coherency can be maintained by two popularly used methods: (i) No Cache scheme and (ii) Client Polling

No Cache Scheme: In this scheme no caching is performed in the data-center. The no-cache based scheme has several disadvantages. Firstly, each request has to be processed at the home node tier, ruling out any caching at the other tiers. Secondly, propagation of these requests to the back-end nodes over traditional protocols can be very expensive and lastly, 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.

Client Polling: 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 benefi ts 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 signifi cant benefi ts when the back-end data is not updated very frequently.

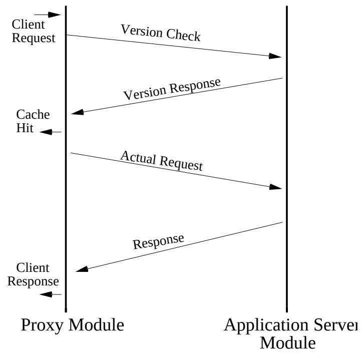


Figure 2. Strong Cache Coherence Protocol

However, this scheme also has signifi cant disadvantages: (i) 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, (ii) Traditional protocols require the back-end nodes to be interrupted for every cache validation event generated by the front-end, and (iii) The agents on the back-end nodes need to keep track of validity of the cache entities for further references by forward caches. In particular, coordination of back-end nodes is needed to propagate the information regarding object updates. In our previous work [17], we have shown that very effi cient client polling by proxies can be performed with the use of one sided operations, addressing the fi rst two disadvantages listed above very effectively.

In this paper, we focus on the issues and challenges posed by the need to maintain cache validity at the back-end with object level granularity. So we present an effi cient client polling architecture that can provide strong cache coherence with object level granularity using the advanced features of Infi niBand.

3 Design and Implementation Details

In this section, we describe all aspects of our design. We detail each of the requirements along with the corresponding design solution. We broadly divide this section into three parts: (i) Section 3.1: The basic protocol for the cache coherency, (ii) Section 3.2: Application server interaction and (iii) Section 3.3: The study of the effect of multiple dependencies on cached documents.

Cachable Requests: Data-center serving dynamic data, usually have HTTP requests that may be reads (select based queries to the database) or writes (update or insert based queries to the database). While read based queries are cachable, writes cannot be cached at the proxies. Since, caching the popular documents gives the maximum benefi ts, its a popular practice to cache these. Most simple caching schemes work on this principle. Similarly, in our design, a certain number of top most frequent requests are marked down for caching. Naturally, caching more requests leads to better performance but requires higher system resources. The actual number of requests that are cached are chosen based on the availability of resources. Based on these constraints, for each request the proxy server decides if the request is cachable. And if it is cachable, the proxy decides if caching that particular request is benefi cial enough. Signifi cant amount of research has been done on cache replacement policies [11, 13, 19]. Our work is complimentary to these and can leverage those benefi ts easily.

External Module Based Design: Traditional Data-Center applications have been developed over a long period of time. It is highly cumbersome and infeasible to make major re-designing of these applications for possible benefi ts. In view of this restriction, we use external helper modules to enable caching in our architecture. This approach requires minimal changes to the existing applications. We use the native Infi niBand user-level communication protocol Verbs API (VAPI) for our all module internode communications. Figure 3 shows the typical setup of each node in our design. For all cache related operations, the data-center applications contact the external module running on the same node. This module in-turn contacts other modules in the system as required and replies back to the application.

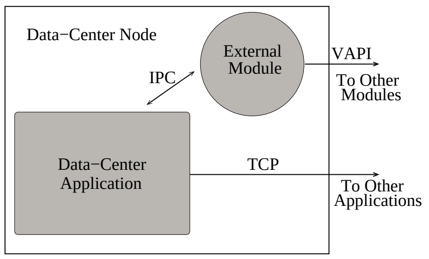


Figure 3. External Module based Design

3.1 Caching Documents with Multiple Dependencies

In our approach we divide the entire operation into two parts based on the tier functionalities. Proxy servers that maintain the cache need to validate the cache entity for each request. The application servers need to maintain the current version of the cached entity for the proxies to perform validations.

3.1.1 RDMA based Strong Cache Coherence

Caches in our design are located on the proxy server nodes. On each request, the primary issues for a proxy are as follows: (i) Is this request cachable? (ii) Is the response currently cached? and (iii) Is this cached response valid?

These services are provided to the proxy server by our module running on the proxy node. The apache proxy server is installed with a small handler that contacts the local module with an *IPC-Verify* message and waits on the IPC queue for a response. The module responds with a *use cache* or *do not use cache* depending on the choices. If the request is not cachable or if the cache is not present or invalid, the module responds with *do not use cache*. And if the request is cachable, cache is present and valid, the module responds with *use cache*.

The module verifies the validity of the cached entry by contacting the home node application server module which keeps track of the current version for this particular cache file. InfiniBand's one sided operation (RDMA Read) is used to obtain the current version from the shared version table on the home node application server thereby avoiding interrupts at that application server. Figure 4 shows the basic protocol. The information on cachability and presence of cache are available locally on each proxy.

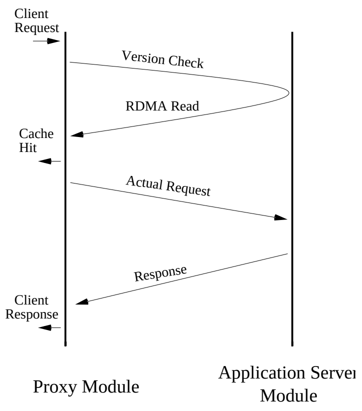


Figure 4. RDMA based Strong Cache Coherence

Each proxy maintains a version number for each of the cached entries. The same cache files also have a current version number maintained on the home node application server modules. When necessary, the application server modules increment their cache version numbers. For each proxy verify message, if the application server cache file version number and the proxy's local cache file version number match, then it implies that the cache file is current and can be used to serve the request. This basic protocol was proposed in our previous work [17].

3.1.2 Multi-Dependency Maintenance

All cache misses from the proxies are serviced by application servers. Since all accesses to the database need to be routed through an application server and since an application server (unlike a proxy) has the capability to analyze and process database level queries, we handle all coherency issues at this tier.

An application server module needs to cater to two cases: (i) version reads from the proxy server and (ii) version updates from local and other application servers. The main work of the application server module lies in updating the shared version table readable by the proxy server modules based on the updates that occur to the data in the system.

As mentioned, a single cached request contains multiple dynamic objects which can get updated. For any version updates to take place, it is necessary to know the following: (i) Which updates affect which dynamic objects? and (ii) Which dynamic objects affect which cache files? Since, typically dynamic objects are generated as results of queries to a database, knowledge of the database records that a query depends on is sufficient to answer the above.

There are three cases that arise in this context: (i) The application server understands the database schema, constraints, each query and its response thereby knowing all the dependencies of a given request, (ii) Each query response contains enough information (e.g. list of all database keys) to find out the dependencies or (iii) The application server is incapable of gauging any dependencies (possibly for cases with very complex database constraints). The first two cases can be handled in the same manner by the application server module since the dependencies for all requests can be obtained. The third case needs a different method. We present the following two sample schemes to handle these cases. It is to be noted that these schemes are merely simple schemes to show proof of concept. These can be further optimized or be replaced by complex schemes to handle these cases.

Scheme - Invalidate All: For cases where the application servers are incapable of getting any dependency information, the application servers modules can invalidate the entire cache for any update to the system. This makes sure that no update is hidden from the clients. But this also leads to a significant number of false invalidations. However, the worst performance by this scheme is lower bounded by the base case with no caching.

Scheme - Dependency List: In cases where all the dependencies of the required queries are known, the application server module maintains a list of dependencies for each cached request (for which it is a home node) along with the version table. In case the application server module is notified of any update to the system, it checks these lists for any dependencies matching the update. All cache files that have at least one updated dependency are then invalidated by incrementing the version number on the shared version table. This scheme is very efficient in terms of the number of false invalidations but involves slightly higher overhead as compared to the *Invalidate All* scheme.

3.2 Protocol for Coherent Invalidations

In addition to the issues seen above, requests comprising multiple dynamic objects in them involve additional issues. For example, two different cache files with different home nodes might have a common dependency. So, any update to this dependency needs to be sent to both these home nodes. Similarly, the application server modules need to communicate all updates with all other application server modules. And the update can be forwarded to the database for execution only after all the application server modules invalidate all the dependent cache files.

Figure 5 shows the interaction between the application servers and the database for each update. As shown, the application server on getting an update, broadcasts the same to all the other application server modules. These modules then perform their local invalidations depending on the scheme chosen (*Invalidate All* or *Dependency List* search). After the invalidations, the modules send an acknowledgment to the original server, which forwards the request to the database and continues with the rest as normal.

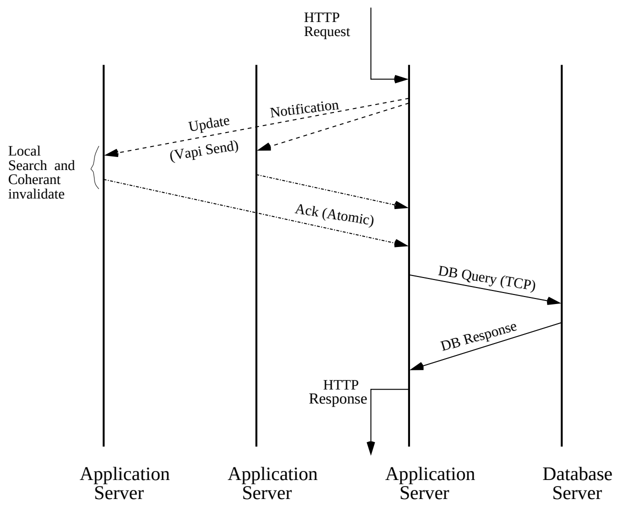


Figure 5. Protocol for Coherent Invalidations

In our design, we use *VAPI-SEND/VAPI-RECEIVE* for the initial broadcasts. The acknowledgments are accumulated by the original process using *VAPI-ATOMIC-FETCH-AND-ADD* : poll : yield cycle. For each application server module, an acknowledgment collection variable is defined and set to zero for each update. All the other application server modules perform a *VAPI-ATOMIC-FETCH-AND-ADD* incrementing the value of this variable by one. The original server module checks this ack collection variable to see if all the remaining modules have performed this operation. If the value of the ack collection variable is less than the number of other servers, then the original application server module process yields the CPU to other processes in the system using the system call *sched_yield()*. This kind of polling cycle makes sure that the module process does not waste any CPU resources that other processes on that node could have used.

3.3 Impact of Multiple Dependencies on Caching

We have seen that there is significant complexity in managing multiple dependencies per cache file on strong coherency caching. In addition, having multiple dependencies also affect the overall cache. Typically, caching is expected to yield maximum benefits when the number of updates is low and the benefits of caching are linked with the number of updates in all calculations.

However, the actual value that affects caching is the number of invalidations that occur to the cached entries. The main difference between the number of invalidations and the number of updates to an object is the magnification factor for updates. This magnification factor represents the average number of dependencies per cached entry. Hence, the cache effectiveness is dependent on the product of *system update rate* and *average dependency magnification factor*.

In our design each application server module maintains its own set of cache file versions and the corresponding dependency lists. So, for each update, the number of messages between the application servers is not affected by this magnification factor. Each application server module is just notified once for each update, and all the invalidations on that node are taken care of locally by the corresponding module. However, the overall cache hit ratio remains directly affected by this factor.

4 Experimental Results

In this section, we describe our experimental testbed and a set of relevant micro-benchmark results followed by overall data-center results.

Experimental Testbed: For all our experiments we used two clusters whose descriptions are as follows:

Cluster 1: 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 and 512 MB of main memory. We used the RedHat 9.0 Linux distribution.

Cluster 2: A cluster system consisting of 8 nodes built around SuperMicro SUPER X5DL8-GG motherboards with ServerWorks GC LE chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 3.0 GHz processors with a 512 kB L2 cache and a 533 MHz front side bus and 512 MB of main memory. We used the RedHat 9.0 Linux distribution.

The following interconnect was used to connect all the nodes in Clusters 1 and 2.

Interconnect: InfiniBand network with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 twenty-four 4x Port completely non-blocking InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-3.2-rc17. The adapter firmware version is fw-23108-rel-3_00_0002. The IPoIB driver for the InfiniBand adapters was provided by Mellanox Incorporation as a part of the Golden CD release 0.5.0.

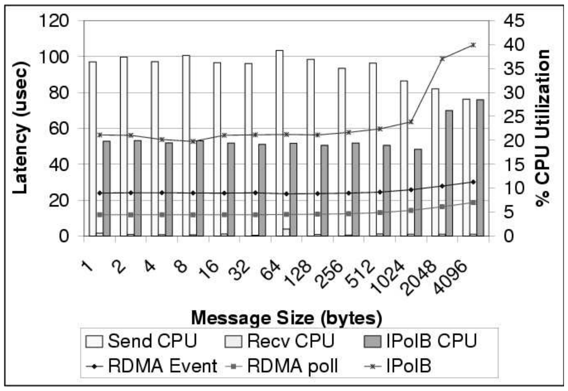
Cluster 1 was used for all the client programs and Cluster 2 was used for the data-center servers. In our experiments, we used apache servers 2.0.48 as proxy servers, apache 2.0.48 with PHP 4.3.7 as application servers and mysql 4.1 as the database. Our system was configured with *five* proxy servers, *two* application servers and *one* database server.

4.1 Micro-benchmarks

We show the basic micro-benchmarks that characterize our experimental testbed. We present the latency, bandwidth and CPU utilizations for the communication primitives used in our design. Figure 6 shows the performance achieved by VAPI RDMA read and TCP/IP over InfiniBand (IPoIB).

The latency achieved by the VAPI RDMA Read communication model and IPoIB (round-trip latency) for various message sizes is shown in Figure 6a. RDMA Read, using the polling based approach, achieves a latency of 11.89μs for 1 byte messages compared to the 53.8μs achieved by IPoIB. The event based approach, however, achieves a latency of 23.97μs. Further, with increasing message sizes, the difference between the latency achieved by VAPI and IPoIB tends to increase significantly. The figure also shows the CPU utilized by RDMA Read (notification based) and IPoIB. The receiver side CPU utilization for RDMA as observed is negligible and close to zero, i.e., with RDMA, the initiator can read or write data from the remote node without requiring any interaction with the remote host CPU. In our experiments, we benefit more from the one sided nature of RDMA and not just due to the raw performance improvement of RDMA over IPoIB.

Figure 6b shows the uni-directional bandwidth achieved by RDMA Read and IPoIB. RDMA Read is able to achieve a peak bandwidth of 839.1 MBps as compared to a 231 MBps achieved



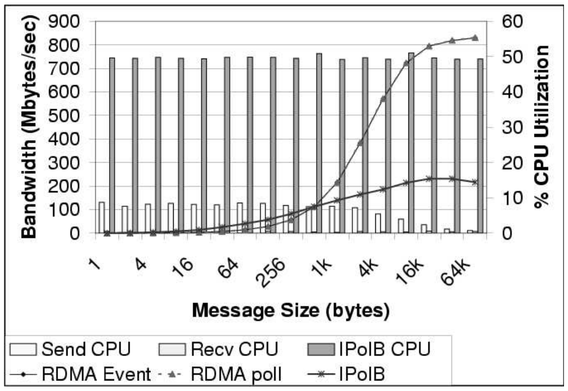


Figure 6. Micro-Benchmarks for RDMA Read and IPoIB: (a) Latency and (b) Bandwidth

by IPoIB. Again, the CPU utilization for RDMA is negligible on the receiver side.

In Figure 7, we present performance results showing the impact of the loaded conditions in the data-center environment on the performance of RDMA Read and IPoIB on Cluster 2. We emulate the loaded conditions in the data-center environment by performing background computation and communication operations on the server while the read/write test is performed by the proxy server to the loaded server. This environment emulates a typical cluster-based multiple data-center environment where multiple server nodes communicate periodically and exchange messages, while the proxy, which is not as heavily loaded, attempts to get the version information from the heavily loaded machines. Figure 7 shows that the performance of IPoIB degrades significantly with the increase in the background load. On the other hand, one-sided communication operations such as RDMA show absolutely no degradation in the performance. These results show the capability of one-sided communication primitives in the data-center environment.

4.2 Coherent Active Caching

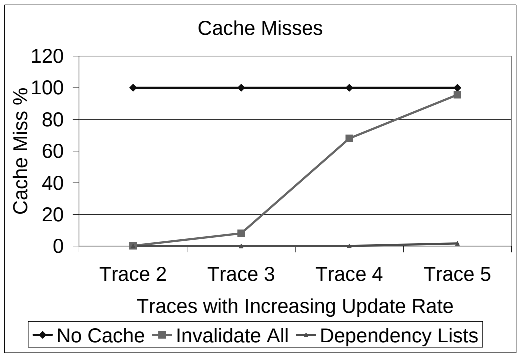


Figure 9. Cache Misses With Increasing Update Rate

In this section, we present the basic performance benefits achieved by the strong coherency dynamic content caching as compared to a traditional data-center which does not have any such support. To measure and make these comparisons, we use the following traces: (i) *Trace 1:* Trace with 100% reads, (ii) *Trace 2* - *Trace 5:* Traces with update rates increasing in the order of milliseconds to seconds. (iii) *Trace 6:* Zipf like trace [8, 23] with medium update rate.

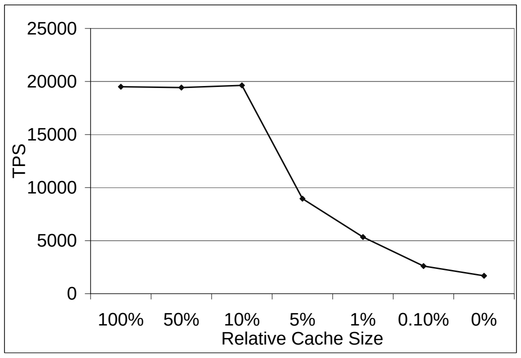


Figure 10. Effect of Cache Size for Trace 6

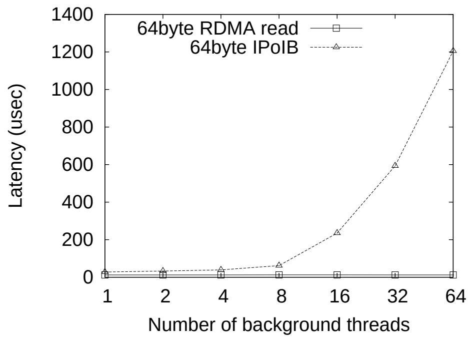
4.2.1 Overall Performance and Analysis

In this section, we present results for overall data-center performance with dynamic content caching. We also analyze these results based on the actual number of cache misses and cache hits. All results presented in this section are taken in steady state (i.e. eliminating the effects of cold cache misses, if any). We noncached data-center throughput of about 1700 TPS for *Trace 1* and 15000 TPS for a fully cached data-center. These values roughly represent the minimum and maximum throughput achievable for our setup.

Figure 8a compares the throughput achieved by dynamic content caching schemes and the base case with no caching. We observe that the caching schemes always perform better than the *no cache* schemes. The best case observed is about 8.8 times better than the *no cache* case.

Figure 8a also shows the two schemes (*Invalidate All* and *Dependency Lists*) for the traces 2 - 5. We observe that *Invalidate All* scheme drops in performance as the update rate increases. This is due to the false invalidations that occur in the *Invalidate All* scheme. On the other hand, we observe that *Dependency Lists* scheme is capable of sustaining performance even for higher update rates. The latter sustains a performance of about 14000 TPS for our setup. Figure 8b shows the response time results for the above three cases. We observe similar trends in these results as above. *No Cache* case has a response time of about 4 milliseconds where as the best response time for dynamic content caching schemes is about 1.2 milliseconds.

Figure 9 shows the cache misses that occur in each of the runs in the throughput test. The *No Cache* scheme obviously has 100% cache misses and represents the worst case scenario. We clearly observe that the cache misses for *Invalidate All* scheme increase



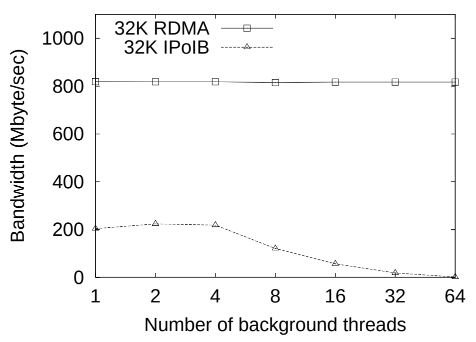
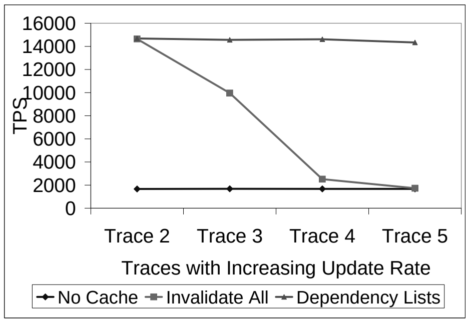


Figure 7. Performance of IPoIB and RDMA Read with background threads: (a) Latency and (b) Bandwidth



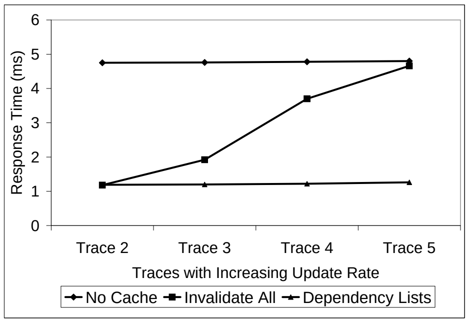


Figure 8. Performance of Data-Center with Increasing Update Rate: (a) Throughput and (b) Response Time

drastically with increasing update rate, leading to the drop in performance. Data-center scenarios in which application servers cannot extract dependencies from the requests can take advantage of our dynamic content caching architecture for lower update rates. For higher update rates, *Invalidate All* performs slightly better than or almost equal to the performance of *No Cache* case. The difference in the number of cache misses between *Invalidate All* and *Dependency Lists* is the number of false invalidations occurring in the system for the *Invalidate All* scheme.

Selective Caching: As mentioned earlier, in real scenarios only a few popular files are cached. In Figure 10, we present the results of an experiment showing the overall data-center performance for varying cache sizes. We used *Trace 6* for this experiment. We observe that even for very small cache sizes the performance is significantly higher than the *No Cache* case. The throughput achieved by caching 10% of the files is close to the maximum achievable. Hence, data-centers with any amount of resources can benefit from our schemes.

Effect of Varying Dependencies on Overall Performance: Figure 11 shows the effect of increasing the number of dependencies on the overall performance. The throughput drops significantly with the increase in the average number of dependencies per cache file. This is because the number of coherent cache invalidations per update request increase with the average number of dependencies tending toward *Invalidate All* in the worst case. We see that as the ratio of object updates to file invalidations representing the dependency factor increases to 64 in Figure 11 the throughput achieved drops by about a factor of 3.5.

4.2.2 Effect on Load

In this section, we study the effect of increased back-end server load on the data-center aggregate performance. In this experiment, we emulate artificial load as described in section 4.1. We use *Trace 5* (trace with higher update rate) to show the results for the *Dependency Lists* scheme. Figure 12 shows the results.

We observe that our design can sustain high performance even under heavy back-end load. Further, the factor of benefit for the *Dependency Lists* scheme to the *No Cache* scheme increases from about 8.5 times to 21.9 times with load. This clearly shows that our approach is much more resilient to back-end load than the *No Cache* scheme. In addition, since loaded back-end servers can support the proxy caching with negligible overhead, our approach can scale to bigger data-centers with significantly higher number of caching servers. The results in [17] show that these benefits are largely due to the one-sided communication in the basic client polling protocols.

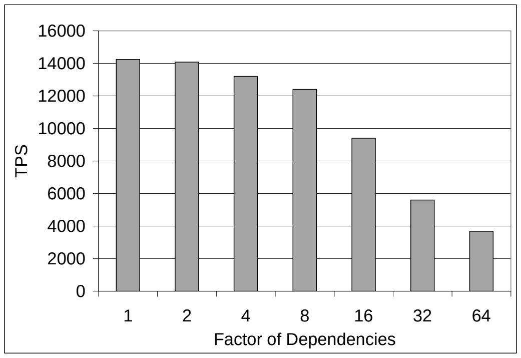


Figure 11. Effect of Varying Dependencies on Overall Performance

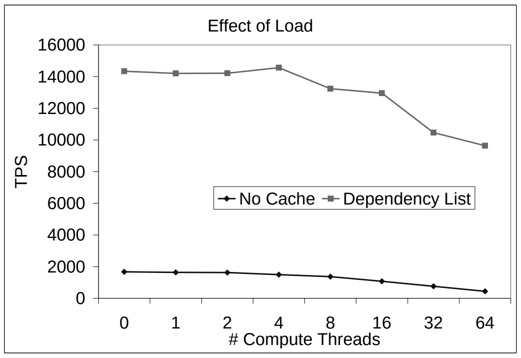


Figure 12. Effect of Load

5 Related Work

Several researchers have focused on the aspect of dynamic content caching. Popular approaches like TTL [12], Adaptive TTL [10], MONARCH [16], etc. deal with lazy consistency and coherency caches. These cannot handle strong cache coherency. Approaches like [6, 7] deal with strong cache consistency. Other approaches like *get-if-modified-since* specifi ed in [12] can handle coherent changes to the original source fi le, but are not designed to handle highly dynamic data. These essentially will cause the fi le to be re-created for each request negating the benefi ts of caching. Solutions proposed in [18] handle the strong cache coherency but use two-sided communication and are less resilient to load.

Shah, Kim, Balaji, et. al., have done signifi cant research in User Level High Performance Sockets implementations [20, 14, 4, 5, 3]. In one of our previous works [3], we had evaluated the capabilities of such a pseudo-sockets layer over Infi niBand in the data-center environment. However, as we had observed in [17], the two-sided nature of Sockets API becomes an inherent bottleneck due to the high load conditions common in data-center environments. Due to this, we focused on the one-sided nature of Infi niBand to develop our external modules. Further, the existing data-center framework (Apache, PHP, etc.,) is still based on the sockets API and can benefi t from such high-performance sockets implementations. Thus, these approaches can be used in a complimentary manner with our reconfi gurability technique to make better utilization of system resources and provide high performance in a data-center environment.

6 Conclusions

Caching has come to be an important tool in the scalability of multi-tier data-centers. However, for content used in online banking, Internet auctions, e-commerce, etc. issues like cache coherency and consistency requirements and the dynamics of data involved can preempt a majority of caching schemes. Caching of dynamic content with strong cache coherency and consistency has been studied by few researchers and is a subject of high interest. Web responses that involve multiple dynamic dependencies add additional constraints to the designing of strongly coherent caches.

In this paper, we have presented a load resilient architecture that supports caching of dynamic requests with multiple dynamic dependencies in multi-tier data-centers. Our architecture is designed to support existing data-center applications with minimal modifi cations. We have used one sided operations like RDMA and Remote Atomics in our design to enable load resilient caching. We have performed our experiments using native Infi niBand Verbs Layer (VAPI) for all protocol communications. Further, we have presented two schemes *Invalidate All* and *Dependency Lists* to suite the needs and capabilities of different data-centers. Our experimental results show that in all cases the usage of our schemes yield better performance as compared to *No Cache* case. Under loaded conditions, our architecture can sustain high performance better than the *No Cache* case, and in some cases being more than an order of magnitude better. The results also demonstrate that our design can scale well with increasing number of nodes and increasing system load.

While our scheme is well suited for small to medium clusters, for large scale clusters, we propose to utilize Infi niBand's one-sided operations and hardware based multicast operations to design advanced caching capabilities with better performance.

References

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

[2] Infi niband Trade Association. http://www.infi nibandta.org.

[3] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over Infi niBand in Clusters: Is it Benefi cial? In *ISPASS* 2004.

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

[5] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *HPDC*, 2003.

[6] A. D. Bradley and A. Bestavros. Basis Token Consistency: Extending and Evaluating a Novel Web Consistency Algorithm. In *WC3*, 2002.

[7] A. D. Bradley and A. Bestavros. Basis token consistency: Supporting strong web cache consistency. In *Global Internet Worshop*, 2002.

[8] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

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

[10] Michele Colajanni and Philip S. Yu. Adaptive ttl schemes for load balancing of distributed web servers. *SIGMETRICS*, 1997.

[11] John Dilley, Martin Arlitt, and Stephane Perret. Enhancement and validation of the Squid cache replacement policy. In *WCW*, 1999.

[12] 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.

[13] Terence P. Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey K. MacKie-Mason. Biased replacement policies for Web caches: Differential quality-of-service and aggregate user value. In *WCW*, 1999.

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

[15] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over Infi niBand. In *SC*, 2003.

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

[17] Sundeep Narravula, Pavan Balaji, Karthikeyan Vaidyanathan, Savitha Krishnamoorthy, Jiesheng Wu, and Dhabaleswar K. Panda. Supporting strong cache coherency for active caches in multi-tier data-centers over infi niband. In *SAN*, 2004.

[18] Jin Zhang Pei Cao and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Distributed Systems Platforms and Open Distributed Processing*, 2002.

[19] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. Technical Report RN/98/13, UCL-CS, 1998.

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

[21] 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 *USITS*, 2001.

[22] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over Infi niBand: Design and Performance Evaluation. In *ICPP*, 2003.

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