Low-Latency Caching for Cloud-Based Web Applications

Adam Wolfe Gordon and Paul Lu
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada
{awolfe,paullu}@cs.ualberta.ca

ABSTRACT

Many Web applications are now hosted in elastic cloud environments where the unit of resource allocation is a virtual machine (VM) instance; entire VMs are added or removed to scale up or scale down. A variety of techniques can reduce the latency of communication between VMs co-located on the same server in, say, a private cloud. For example, paravirtualized network mechanisms (e.g., vhost and virtio in Linux KVM) can optimize the number of protection boundary crossings. Inter-VM shared memory can further reduce boundary crossings after setting up a shared region.

We present the design, implementation, and an evaluation of Nahanni memcached, a port of the well-known memcached that uses inter-VM shared memory instead of a virtual network for cache reads. As a widely deployed cache for back-end datastores and databases, memcached's latency is important to the performance of many well-known web sites (e.g., Facebook, Twitter) and cloud platforms (e.g., Google's App Engine). Although using shared-memory IPC is a well-known strategy, the recent introduction of the ivshmem inter-VM shared memory mechanism (also known as Nahanni) to Linux KVM makes the strategy practical for virtual machines. Using the Yahoo Cloud Serving Benchmark, we confirm the intuition that Nahanni memcached can reduce the latency of cache read operations by up to 86%, and that given reasonable hit rates, this can reduce the total latency of read-related operations for a workload by up to 45% compared to standard memcached. When using the experimental paravirtualized vhost networking mechanism in Linux KVM, Nahanni memcached offers a smaller, but still significant, advantage of 29%.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design, Performance

Keywords

Cloud computing, memcached, latency, virtual machines, web applications

1. INTRODUCTION

Web applications often have unpredictable load patterns, and cloud computing allows developers to automatically scale their services by allocating resources on-demand as traffic grows. In infrastructure-as-a-service (IaaS) systems such as Amazon Elastic Compute Cloud (EC2) and private clouds built using Eucalyptus [20] or OpenStack [4], resources are allocated in the form of virtual machines (VMs). A number of platform-as-a-service (PaaS) offerings (e.g., Google App Engine (GAE) [1], Microsoft Azure [6], AppScale [10]) provide application programming interfaces (APIs) that allow developers to build and deploy web applications that will be automatically scaled based on demand.

Furthermore, many web applications are backed by a datastore, such as an SQL database or a cloud-scale NoSQL storage engine. For example, Amazon offers SimpleDB, and GAE includes a component called `Datastore` that is backed by BigTable [9]. Because datastore access can be expensive, contributing significantly to web application latency, it is common to use an in-memory cache.

In IaaS environments, application-level caching is usually the responsibility of the application developer, who may choose to deploy a caching service such as memcached [2] using the virtual machines in which their web application is deployed. In PaaS environments, a caching mechanism may be provided in the API (e.g., the `memcache` API in GAE and AppScale). This API could be backed by memcached (as it is in AppScale) or a similar system.

We present a modified version of the memcached server that uses inter-VM shared memory to store cached data, while maintaining compatibility with standard memcached clients. Additionally, we present a memcached client library for C, Java, and Python that can take advantage of the shared memory design. We demonstrate that our implementation can reduce the latency of cache reads by up to 86%, and given reasonable cache hit rates can reduce the total latency of datastore reads by up to 45% compared to using standard memcached, depending on the workload. We also show that our shared-memory-based technique can be faster than sophisticated paravirtualized network devices, such as virtio and vhost in Linux KVM.

2. MOTIVATION

The advantage of the cloud computing model over physical servers or static VM deployment is flexibility. In a cloud system, services can have more resources allocated by adding more VMs on-the-fly, and can have resources removed by shutting down these VMs. In the context of public clouds such as EC2, this has allowed computing time and storage to be sold inexpensively, since infrastructure can be efficiently shared among many users [13].

These advantages are not limited to public clouds. Using open-source systems such as Eucalyptus and OpenStack, an organization can provide a cloud environment to internal users, enabling better utilization of existing resources. These private and hybrid cloud systems [26] offer additional advantages compared to public clouds. For example, hosting a private cloud helps to mitigate security concerns [23, 14] induced by sharing physical resources with other, potentially hostile, customers. We argue that as the popularity of the cloud computing model continues to increase, so will the popularity of private clouds.

Further, we argue that these private cloud deployments will tend to encompass fewer resources than public clouds, and that as the core counts of commodity servers continue to increase, these resources will be concentrated in fewer and fewer individual servers. Thus, it will be increasingly common for multiple VMs running a web application in such a cloud to be co-located on a single physical host. Multiple physical servers will still be required for many scales, but for the cases where VMs are co-located, the latency of interprocess communication (IPC) to, say, memcached can be significantly reduced using shared-memory IPC.

We also note that there is currently active research on network-aware placement and migration of virtual machines [16, 21, 18]. With such mechanisms in place, a web application VM and a memcached VM that communicate heavily (i.e., to store and fetch cached data) would likely be co-located, and thus be able to use shared memory for caching.

3. BACKGROUND AND RELATED WORK

Our work builds upon two software projects: memcached (an open-source project originally developed by LiveJournal), and the Nahanni inter-VM shared memory system (an open-source project that is included in the Linux KVM mainline).

Memcached [2] is a distributed in-memory cache designed for use in web applications. The server acts as a simple key-value store, where both keys and values are arbitrary sequences of bytes. Clients communicate with one or more servers using IP sockets, and the servers keep cached data entirely in memory. By design, the server has few features: replication and distribution, for example, are not provided by the server but are commonly provided by client libraries.

Memcached can be thought of as an optimization layer between the database and the application, albeit one that is managed by the application itself. Facebook, for example, employs memcached as one of several caching layers [19], using it to cache frequently-accessed user data that is stored in MySQL clusters [27]. Twitter is another known deployment of memcached [2, 5].

There are two performance benefits to using memcached: reduced latency when data can be found in the cache instead of being re-generated, and improved database scalability due to reduced request volume. Memcached is preferable to purely application-level caching because it can increase the total capacity of the cache by distributing the cache across multiple servers, and it allows multiple web servers to access the same cache.

Nahanni [3, 17] is a mechanism in QEMU/KVM v0.13 that allows a POSIX shared memory region on a host to be mapped into the address space of one or more VMs running on the same host. Consequently, the co-located VMs and the host can share data and perform IPC through the shared-memory region. Nahanni is implemented as a virtual PCI device with an on-board memory region that can be mapped using a special driver in the guest operating system.

To use the shared memory region, an application opens the device as a file, then uses `mmap` to map the memory into its address space. This initialization procedure is nearly identical to that of standard POSIX shared memory, in which the memory region is opened with the `shm_open` system call, then mapped with `mmap`. The only difference with Nahanni is that the device file is opened using the `open` system call, and the `mmap` argument indicating the offset into the file must be changed to a particular value. Once an application has initialized Nahanni, the shared memory region can be used just like any other memory in the application.

In related work, of course, the idea of using shared-memory IPC is not new. For example, the fbufs mechanism goes back to 1993 and, more recently, a variety of Xen-based projects have used shared memory to speed up sockets-based IPC (e.g., XenSocket [29]). The Nahanni memcached approach is different in that it shares data across VMs (not just intra-VM domains, like fbufs) and the pointer-based hash table of memcached can be placed in shared memory and directly accessed by the clients, avoiding the potential bottleneck of the memcached server as well as network latency. We note that an evaluation of memcached bottlenecks and the scalability of Nahanni memcached are subjects of future work.

Nahanni memcached shares many similarities with the global shared buffer cache in Disco [8]. However, whereas in Disco the shared memory region was used to implicitly cache filesystem buffers for homogeneous operating systems, memcached allows applications running on any operating system, and even on other servers, to explicitly cache arbitrary data.

Automatic, consistency-preserving caching of datastore results [28] and transactionally-consistent caching [22] are active areas of research. The goal in such consistency-preserving caching is to enable automatic caching, particularly for applications where transactional consistency is vital. However, work on cache consistency does not address the latency of caching, nor is it aimed at memcached applications, which often gain performance by tolerating some degree of staleness. Work on memcached performance [25] has focused primarily on scaling memcached servers for large deployments, and does not take advantage of the structure of cloud environments to reduce latency.

Something similar to Nahanni memcached was implemented for a cloud environment on the IBM Blue Gene/P (BG/P) [7]. Unlike Nahanni memcached, the BG/P hardware has physically distributed memory and a fast RDMA mechanism. But, like Nahanni memcached, the BG/P work attempts to reduce the latency of cache look-ups using RDMA instead of normal network mechanisms.

4. NAHANNI MEMCACHED

For Nahanni memcached, we modified the memcached server to keep cached data in a shared-memory region which can be accessed directly by VMs on the same host using the Nahanni mechanism (Figure 1). To a standard memcached client, the server acts exactly like a standard memcached server. But, we have implemented a client library in C, with Java and Python wrappers, to provide a standard memcached API, but fetch items directly from the shared cache memory pool (Figure 1(b)), instead of using TCP/IP sockets over a virtualized network (Figure 1(a)), for lower latency. Omitted in both figures are connections to the memcached server from other clients — web applications running in VMs on other hosts, which use the network for all communication with either version of the server.

4.1 Item Storage and Lookup

Because the standard memcached server allocates a large pool of memory, then handles item allocation internally, minimal changes were required to have memcached store cached data in shared memory. Instead of `malloc`ing a large region of memory for the cache allocator, the server `mmap`s the shared memory into its address space. Having the cached items in shared memory only solves half the problem: the client still needs the ability to find them.

Since cached data tends to be small (memcached limits values to 1 MB by default), the time to read a value is usually dominated by the round-trip latency to the server and the server's processing time. Therefore, it is highly desirable to avoid network communication with the server entirely. We accomplish this by keeping the hashtables in the shared memory region along with the data.

We chose, for reasons of performance and simplicity, to avoid pointer swizzling by mapping the shared memory region to the same address in the server and all clients. Mapping to a particular address is accomplished using the `MAP_FIXED` flag to `mmap` when the server and the library initialize the shared memory region, as described in Section 3. By mapping the memory to a high address and doing the mapping early, before much memory has been mapped or allocated, we avoid collisions that would cause `mmap` to fail. Of course, if the mapping does fail, we can fall back to communicating over the network. In a future version of the Nahanni memcached library, we could implement pointer swizzling as a faster fallback.

Because our client accesses cached data directly in the server's memory, it is possible that an item will be removed (due to expiry or capacity) between when the client acquires a pointer to it and when it reads the data. To avoid potential race conditions, we take advantage of the fact that memcached uses locking and reference counting to enable its multithreaded design. The client locks the cache before looking up an item, then increments the reference count when an item is found. The reference count is not decremented until the client is finished copying or consuming the cached data, preventing the item from being deleted while in use.

4.2 Item Expiration

Memcached's lazy expiration causes a complication in our design. Because the client library looks up items directly in the server's memory, it will find expired items that the server would not return over the network. However, memcached does not store items with absolute expiry times; instead, it uses relative times based on when the memcached server was started. In order for the client to behave correctly, it must be able to recognize expired items in the cache.

To accomplish this, we add a message to the one-time handshake between the client and the server that occurs when a connection is initiated. The client sends a special request message to the server, which responds with the real time at which the server was started. The client records this time, which it uses to convert the relative expiry times stored with the cached data into real expiry times.

4.3 Locality Discovery

In a simple environment involving only one host, memcached clients may be able to assume that they are running on the same physical host as the memcached server, and thus can use shared memory to fetch values. However, most cloud environments employ multiple servers, each potentially running a memcached server, a datastore node, and a number of application VMs. In this case, the memcached client must discover whether each memcached server is co-located with it. We call this process *locality discovery*.

We have designed and implemented a simple mechanism for locality discovery in Nahanni memcached. On startup, the server writes a random 64-bit value called the *cookie* into shared memory. After connecting to the server, the client sends a special locality discovery message. A standard memcached server will reply to this message with an error, but the Nahanni memcached server responds with a packet containing the cookie's address and value. The client checks whether the received value matches the cookie stored at the received address to determine whether the server is local.

5. EVALUATION

We evaluated the performance of Nahanni memcached using the Yahoo Cloud Serving Benchmark (YCSB) [11]. YCSB is intended to benchmark cloud-scale datastore systems by providing configurable workloads that can model the loads provided by web applications. The *core workload* provided by YCSB can be configured with a target throughput; the distribution of operations: read, insert, update, and scan; and the statistical distributions to use for selecting which record to read or update and how many records to scan. The YCSB framework reports latency statistics for each operation.

We have implemented a YCSB workload (the *cache workload*) that caches the values it reads. The cache workload allows the user to specify one or more memcached servers; the validity time for cached data; and whether cached data should be invalidated on update operations. The cache workload reports latency statistics for cache reads and cache writes, and the hit rate of the cache, in addition to the datastore latency statistics reported by YCSB. Because YCSB is written in Java, the cache workload uses the Java wrapper for our Nahanni memcached client library.

In order to evaluate the latency advantages provided by caching in general, and Nahanni memcached specifically, we define the *average total read latency* (ATRL) as the total time spent on read-related operations (datastore reads, cache reads, and cache writes), divided by the total number of read operations performed. Most reads will be much

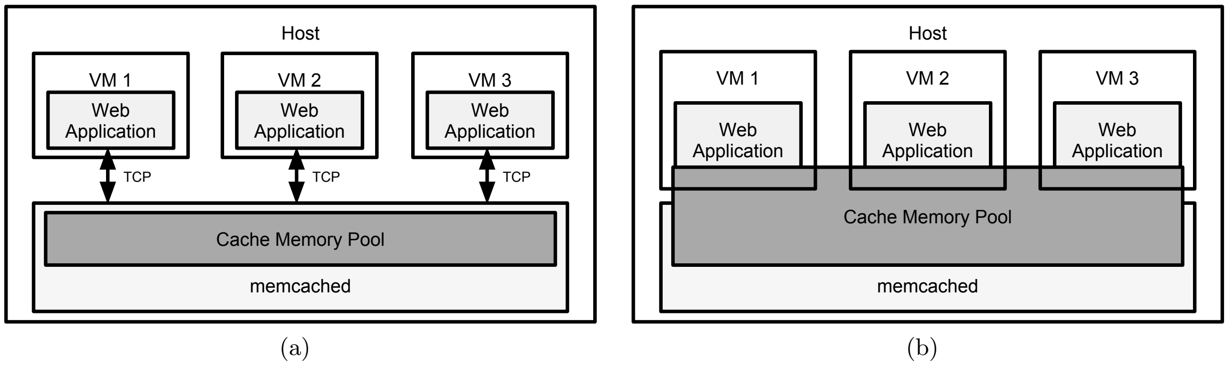


Figure 1: A cloud-based web application using (a) standard memcached and (b) Nahanni memcached

faster (on a cache hit) or much slower (on a cache miss) than the ATRL, but this measurement is useful in comparing the overall latency of reads, including the latency of read-related operations, in a system.

For our benchmarks, we used an Intel Xeon X5550 (2.67 GHz) server with two sockets, a total of eight cores (16 hyperthreads), and 48 GB of RAM. We ran the Cassandra [15] datastore and memcached natively (not in a VM) and ran YCSB in VMs. Each VM was given four virtual CPUs and 4 GB of RAM. Cassandra was chosen because it is a typical example of a cloud-scale NoSQL datastore, and is open-source. We used a 4 GB memory pool for memcached in both the standard and Nahanni configurations. Except where otherwise specified, the VMs were connected to the host and each other by a Virtual Distributed Ethernet (VDE) [12] network, and use the paravirtualized virtio [24] network interface provided by QEMU/KVM.

We used three different sets of parameters with our cache workload in YCSB (Table 1). For brevity, we refer to each set of parameters as a workload, even though they use the same workload component in YCSB. We used the same sets of parameters the YCSB authors [11] used to evaluate datastores, choosing the three workloads that best reflect scenarios in which web applications would use memcached.

We loaded Cassandra with one million 1 KB records, and ran each workload with three YCSB clients in separate VMs running at 2,000 operations per second each, for a total of 6,000 operations per second. Each client executed 12 million operations, for a total of 36 million operations. Note that although the read heavy and read latest workloads perform write operations to Cassandra, we do not report the latency of Cassandra writes, as this latency is not affected by caching.

5.1 Read Heavy

For the read heavy workload (Figure 2(a)) standard memcached provides a 17% improvement (i.e., 1.81 vs. 2.17) over the non-caching case (i.e., using only Cassandra) with a 71% cache hit rate. Nahanni memcached further improves the ATRL by 41% (i.e., 1.07 vs. 1.81) compared to standard memcached, improving the cache read latency by 81% (i.e., 0.15 vs. 0.79).

The read only workload displays similar results and time breakdowns (but with 38% ATRL reduction), with the same cache hit rate, therefore we omit the graph for the read only workload for space reasons.

Note that the read heavy workload involves update operations. Although our YCSB cache workload supports cache invalidation, we left it disabled. Applications that use memcached for mutable data are usually tolerant to some degree of staleness, since cache invalidation must be handled by the application. Also note that the current implementation of Nahanni memcached still uses sockets-based messages to send updates to memcached, which is why the CACHE_WRITE portions of the graphs (i.e., 0.27 vs. 0.25 per read) are similar. Tackling the latency of cache writes is future work.

5.2 Read Latest

The cache hit rate for the read latest workload (Figure 2(c)) is 81%, which is higher than for the read heavy workload (71%) because the latest distribution favors the newest records heavily, more than the Zipfian distribution. This also allows Cassandra's internal cache to be more effective, thus decreasing the average datastore read latency. The high cache hit rate allows standard memcached to improve the ATRL by 32% (i.e., 1.3 vs. 1.91). Nahanni memcached further improves the ATRL by 45% (i.e., 0.72 vs. 1.3) compared to standard memcached, due to an 86% (i.e., 0.1 vs. 0.73) reduction in cache read latency.

5.3 Paravirtualized Networks

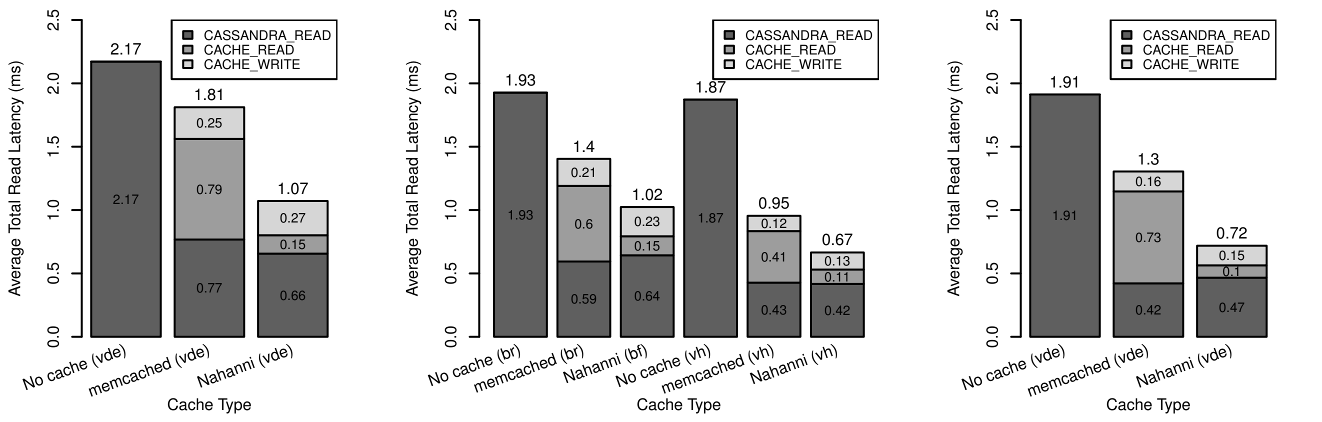
Our experiments in Figure 2(a) and Figure 2(c) used VDE [12] to provide the virtual network. VDE has the advantage that a non-root user can start new VM instances with no administrator intervention. However, if one is willing to provide root or sudo permissions to accounts that start up VM instances (we are wary of such privilege levels, but not everyone in the Linux KVM community is similarly concerned), it is possible to configure higher-performance network setups.

One standard mechanism for virtualized networking is to use a bridged tap device for each VM. This requires per-VM setup from the administrator or sudo permissions for the user running the VM. The experimental vhost mechanism in Linux KVM extends the bridging mechanism and further improves performance by reducing the number of protection domain crossings required for network I/O. We display results from the read heavy workload using bridged (br) and vhost (vh) network setups in Figure 2(b).

Our main conclusion is that the Nahanni memcached configurations still have the lowest latencies, although the specific amount of improvement might be more modest: using

Workload	Operations	Record Selection	Application Scenario
Read only	100% Read	Zipfian	Static, externally-generated user information
Read heavy	95% Read, 5% Update	Zipfian	Photo tagging; reads to view tags, updates to add tags
Read latest	95% Read, 5% Insert	Latest	Status updates; users mostly view the newest statuses

Table 1: Workload parameters for evaluation



(a) Read heavy with VDE (b) Read heavy with bridging (br), vhost (vh) (c) Read latest with VDE

Figure 2: YCSB Benchmark: Breakdown of Average Total Read Latency (ATRL). Bar segments represent the contribution of Cassandra read, memcached cache read, and memcached cache write to the ATRL. "No cache" uses only Cassandra. "Nahanni" uses the network for Cassandra reads and cache writes.

bridging without vhost, Nahanni is 27% faster (Figure 2(b), read heavy workload, 1.02 vs. 1.4) than using standard memcached. And, using both bridging and vhost (vh), Nahanni is 29% faster (Figure 2(b), read heavy workload, 0.67 vs. 0.95). After all, bridged networking and vhost *also* improve the performance of Nahanni memcached since sockets are still being used for updates and for IPC with Cassandra when we miss in the cache.

6. DISCUSSION AND FUTURE WORK

6.1 Security and Protection

Some natural concerns when sharing memory between processes and VMs are security and protection: Can a buggy or malicious memcached client corrupt the shared data? Although it is not yet implemented, the Nahanni shared-memory region can be made read-only on a per-VM instance basis. The host OS mechanism for implementing the shared memory is a memory-mapped file, which supports per-VM-mapping configurations, and the files also support authentication and authorization using the normal file system permissions.

Of course, if a client has read-only access to the memory, then our implementation of Nahanni memcached would have to replace the current load-store based synchronization with something else, such as the event-based synchronization mechanism already in Nahanni. A proper implementation and evaluation of this alternate design is the subject of future work.

6.2 Isolation and Migration

Ironically, one of the main motivations for using VMs is to better isolate one VM from another VM, and shared-memory IPC breaks that isolation. However, we note that full isolation is not always the most important reason to use VMs (e.g., convenient resource allocation in clouds is another common motivation for VMs), and the same dichotomy exists between the isolation of processes versus the sharing of memory between processes (e.g., using System V or POSIX mechanisms) in host operating systems.

In the end, if isolation is more important than performance, then the existing sockets-based IPC for memcached can be used. And, our client library will (in the future) support a simple switch that optionally selects either shared-memory IPC or sockets IPC to memcached, depending on user preference, and the dynamic location of the memcached server (e.g., if clients or servers are ever migrated off the same physical node).

6.3 Platform Integration

Nahanni memcached is currently suitable for use in an IaaS environment with Nahanni shared memory support. That is, it could deployed by a developer in VMs along with a web application. However, many applications are deployed in PaaS environments where memcached is provided as a service.

One area of future work is to integrate Nahanni memcached into a PaaS environment so that existing applications using a particular API can take advantage of the reduced latency provided by shared-memory caching. We plan to integrate Nahanni memcached into AppScale, an open-source framework that emulates the GAE API. This will allow us to evaluate the benefits of shared-memory caching in a real web application. In preparation for this integration, we have already implemented Nahanni-aware memcached client libraries for Java and Python, the languages supported by AppScale.

6.4 Direct Cache Writes

In effective uses of memcached the application looks up each item in the cache before reading it from the datastore, and experiences a high cache hit rate. In such an application, cache reads are much more common than other operations, since cache writes are only required on cache misses. Therefore, we targeted read latency in our design and our current implementation requires the memcached client to communicate with the server over the network for all operations other than reads. However, since the client has direct access to the cached data, it is possible to implement direct writes in the client.

Allowing client-side modifications to the cache will help solve another problem: expiration. In our current implementation, the client recognizes and ignores expired cache data, but does not remove it from the cache. This leads to the server performing an expensive cleanup operation when the cache gets full. With client-side writes implemented, it would be a natural extension to allow the client to remove expired items from the cache when they are found, as the server does.

7. CONCLUDING REMARKS

In the context of different IPC techniques for accessing key-value stores in web applications, we have presented Nahanni memcached, a memcached server using an inter-VM shared memory IPC mechanism paired with wrapper functions (i.e., C, Java, Python) that implement a standard memcached API. Arguably, on elastic cloud systems, there will be co-located VMs of the same web application on the same physical server that can take advantage of shared-memory IPC between the VMs.

Using the Yahoo Cloud Serving Benchmark (YCSB), we have empirically shown that Nahanni memcached, used with VDE networking, can improve the total read-related latency for a workload by up to 45% (i.e., read latest workload) compared to standard memcached, resulting from reductions in cache read latency of up to 86%. When combined with state-of-the-art paravirtualized network mechanisms, such as vhost, Nahanni memcached can still reduce the total read-related latency by up to 29%.

8. ACKNOWLEDGEMENTS

Our work was made possible by Cam Macdonell's work on the Nahanni shared memory system for Linux KVM [3, 17]. This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

9. REFERENCES

[1] Google App Engine. `http://code.google.com/appengine/`.

[2] Memcached - a distributed memory object caching system. `http://memcached.org/`.

[3] Nahanni. `http://gitorious.org/nahanni`.

[4] Openstack open source cloud computing software. `http://openstack.org/`.

[5] Twitter engineering: Memcached SPOF mystery. `http://engineering.twitter.com/2010/04/memcached-spof-mystery.html`.

[6] Windows azure. `http://www.microsoft.com/windowsazure/`.

[7] J. Appavoo, A. Waterland, D. Da Silva, V. Uhlig, B. Rosenburg, E. Van Hensbergen, J. Stoess, R. Wisniewski, and U. Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 385–394, Chicago, Ill, 2010. ACM Press.

[8] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM SIGOPS Operating Systems Review*, 31(5):143–156, Dec. 1997.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[10] N. Chohan, C. Bunch, and S. Pang. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *First International Conference on Cloud Computing*, Munich, 2009.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[12] R. Davoli. VDE: Virtual Distributed Ethernet. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220. IEEE, 2005.

[13] R. Grossman. The Case for Cloud Computing. *IT Professional*, 11(2):23–27, Mar. 2009.

[14] F. J. Krautheim. Private virtual infrastructure for cloud computing. In *Hot Topics in Cloud Computing (HotCloud 2009)*. USENIX Association, June 2009.

[15] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35, Apr. 2010.

[16] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz. Topology-aware resource allocation for data-intensive workloads. In *Proceedings of the first ACM Asia-Pacific workshop on systems - APSys '10*, New Delhi, India, Aug. 2010. ACM Press.

[17] C. Macdonell. *Fast Shared Memory-based Inter-Virtual Machine Communications*. PhD thesis, University of Alberta, 2011.

[18] X. Meng, V. Pappas, and L. Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, Mar. 2010.

[19] L. Nealan. Caching & performance: Lessons from Facebook. OSCon 2008, `http://www.scribd.com/doc/4069180/Caching-Performance-Lessons-from-Facebook`, July 2008.

[20] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE, May 2009.

[21] J. T. Piao and J. Yan. A Network-aware Virtual

Machine Placement and Migration Approach in Cloud Computing. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 87–92. IEEE, Nov. 2010.

[22] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2010.

[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 199–212, Chicago, Ill, Nov. 2009. ACM Press.

[24] R. Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[25] P. Saab. Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919`, 2008.

[26] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, Sept. 2009.

[27] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10)*, pages 1013–1020, Indianapolis, Indiana, June 2010. ACM Press.

[28] N. Tolia and M. Satyanarayanan. Consistency-preserving caching of dynamic database content. In *International World Wide Web Conference*, pages 311–320, Banff, Alberta, 2007.

[29] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. Xensocket: a high-throughput interdomain transport for virtual machines. In *MIDDLEWARE2007: Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, pages 184–203, Berlin, Heidelberg, 2007. Springer-Verlag.