Performance Analysis of Memcached

Vijay Chidambaram
Department of Computer Science
University of Wisconsin Madison
vijayc@cs.wisc.edu

Deepak Ramamurthi
Department of Computer Science
University of Wisconsin Madison
scdeepak@cs.wisc.edu

Abstract

Memcached is an open source, high-performance, distributed memory object caching system. It is widely used by large scale Web 2.0 companies to speed up dynamic web applications by alleviating database load. However, there is little or no academic literature on memcached - It is not understood very well, and its performance has not been analyzed so far. In this paper, we present an analysis of memcached. We identify avenues for optimizing memcached and explore two of them. We analyze the effects on performance for both avenues.

1. Introduction

Memcached[6] is a distributed cache, originally developed at Danga to improve their website performance. Since then, it has evolved into a high-performance object caching system which is widely used by large-scale companies. Facebook has the world's largest memcached deployment - It is used in their photo service to cache the on-disk locations of photos that are requested by users all over the world. LiveJournal uses memcached to alleviate its database load due to hundreds of thousands of users accessing blog entries. Twitter uses memcached to reduce database load when millions of users twitter and access each other's streams.

The power of memcached lies in the fact that it is very easy to scale - To increase performance or to increase the amount of data cached, one just adds nodes. Memcached is very fast since it resides entirely in main memory. This makes it very attractive to companies that might need to scale very quickly.

Inspite of the huge deployments of memcached in several companies all over the world, it remains a poorly understood system. The research community do not understand memcached well - there have been no academic papers on the subject. There are also no publicly available performance studies made by industry. It is very poorly documented inspite of being widely used. Development is carried out by a dozen programmers , and outside this elite circle, knowledge of how memcached works is not widespread.

In this work, we set out to understand memcached, analyze its performance and find possible bottlenecks. Our main contributions in this paper include analysis of memcached's performance (both with respect to the number of clients contacting the server, and the size of the data objects that memcached stores), and identification of opportunities for optimizing memcached. As proof of concept, we explore two such opportunities and report the effects on performance.

We look at tuning the e1000 network driver for memcached - Interrupt blanking, where the system is non-responsive to interrupts for a period of time; Opportunistic polling, which switches between polling and interrupts for I/O based on the load; We look at modifying the number of buffers for transmitting and receiving packets and the delays associated with transmitting and receiving packets.

We also look at providing operating system support for memcached in terms of zero-copying. The Linux kernel has some interesting system calls called splice and vmsplice which implement zero-copy. We modify memcached to use these system calls and document the effect of this modification on performance.

The rest of the paper is organized as follows: Section 2 explains what memcached is, and some of its important properties. In section 3, we motivate our work and enumerate our goals. Section 4 discusses the experimental setup that we used, along with the specific tools used in the work. We analyze the performance of memcached in Section 5. We present the design of our modified memcached in section 6, and explain its implementation in Section 7. We present a comparison of performance between the original memcached and the modified memcached in Section 8, along with an analysis of the performance. We discuss related work in Section 9, and conclude in section 10.

2. Background

Memcached is an *in-memory* key-value store for small chunks of arbitrary data (strings, objects) from results of

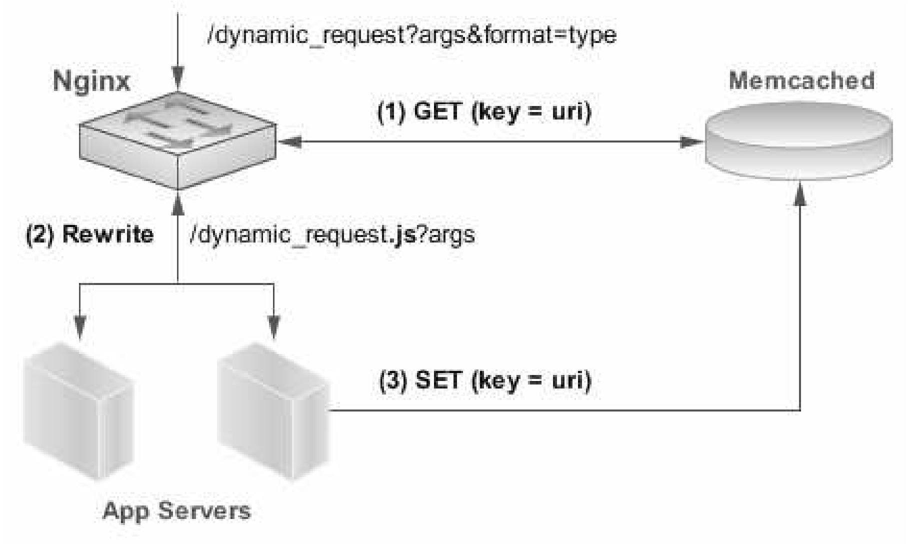

Figure 1. *Memcached in action*. Once values have been cached in the memcached server, GET requests from the Nginx http server can be directly served by memcached, easing load on the app servers. Upon value updates, the app servers SET the values in memcached.

database calls, API calls, or page rendering. Each data item in memcached is associated with a key, which is used to retrieve that particular data item. Memcached uses a simple hash table to stores keys and values. The lookup takes constant time.

Memcached is a *distributed* caching system - There may be several physical machines running memcached, but a unified view is presented to the end user. Memcached uses a technique called consistent hashing to map each key permanently to a single server, among all the servers running memcached. Identifying which server contains a particular key is done at the client side, thereby relieving the servers of this load. This contributes to the high scalability of memcached.

Figure 1 demonstrates the working of memcached. Upon each update, the application servers SET the value in memcached. Note that the value might be stored in any one of the number of machines which are running memcached - But this is oblivious to the application servers. Now when the Nginx http server requires the result of a query, it first asks memcached whether it has that value. If memcached has that value, a trip to the application servers is avoided and the value is returned by it to the http server. This is how memcached helps alleviate database application load.

Instead of using malloc and free, memcached uses *slab allocation*[7] to efficiently allocate space for key-value pairs. Upon start-up, chunks of memory are pre-allocated. Upon request for space for a particular value, the chunk closest in size to it is returned. This is done to avoid fragmentation and to avoid the overhead of finding contiguous blocks of memory.

Memcached uses TCP for communicating with clients. The latest version supports UDP partially, you can send ”set” messages to the server via UDP, but you cannot send ’get’ requests using UDP, since you need to reliably get the response back. It can run in blocking and non-blocking modes. Currently, the maximum size of key-value pairs that can be stored is 1 MB.

Memcached supports multi-threading - the 1.4.3 version which we use runs 4 threads by default. The number of threads run can be controlled by setting an option at the time of starting memcached.

3. Motivation

Memcached is widely deployed by a number of large scale Web 2.0 companies such as Facebook, Flickr, Youtube, Digg, Twitter, LiveJournal and Wordpress. These companies operate at huge scale, typically serving hundreds of thousands of customers per second. These companies use Memcached to alleviate database load and speed up their web applications. This makes memcached one of the most widely deployed distributed system of recent times.

However, there has been little or no academic research about memcached. Due to the small amount of documentation available online about it, the research community does not have a good understanding of it. Our goals in this project are:

1. To gain a deep understanding of memcached
2. To find bottlenecks in memcached performance
3. To optimize memcached

The last goal of optimizing memcached is difficult because several companies, especially Facebook, have invested time and money into optimizing it. The memcached project

Table 1. Specification of server node in experiments

Parameter	Value
Operating System	Fedora Core 6
Linux Kernel Version	2.6.20
Disk size	120 Gb
Frequency	3000 MHz
Memory	4 GB
Processor	Pentium 4
Number of processors	2

leader at Facebook considers that the system is currently so well tuned that adding a single system call will have a considerable impact on throughput.

4. Test Setup

4.1 Hardware Setup

For our experiments, we borrowed machines from the *Wisconsin Advanced Internet Laboratory*. For preliminary experiments, we used 10 machines. For the final experiments, we used 24 machines. The specifications of the server node are shown in Table 1. The frequency of the other nodes varies between 2-3 Ghz, with main memory capacity varying between 1 GB and 3 GB. All nodes in the experiment ran `Fedora Core 6`. All of the nodes were connected to a 1 Gigabit switch. Each node used the `Intel e1000` network driver.

4.2 Memcached version and options

The version of memcached we used is 1.4.3, which was the latest version at the time that we began this work. The command which we used to invoke memcached was:

```
nice -n -20 memcached -m 3000 -p 7000 -u root
```

This invokes memcached at the highest priority, with memory allocation 3000 MB. Memcached runs as the user root, at port 7000.

4.3 Client Library

For client-server communication, we used the `libmemcached 0.34` client library. The reason for choosing this particular client was:

1. Libmemcached is written in C, making it one of the fastest memcached client libraries. Since we use it for performance analysis, we wanted the client library to be as fast as possible so that it does not adversely affect performance.
2. Libmemcached undergoes active development, and is one of the more ”mature” client libraries for memcached.

4.4 Profiler

For profiling, we used `OProfile`[8]. OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information. OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

4.5 Workload

Memcached is a general purpose key-value store - It does not assume anything about the values that it is asked to store. To faithfully mimic this in our experiments, we randomly generated strings containing ASCII characters. In our experiments, the actual value stored in memcached is always random - only the length of the value is controlled during the experiments. Upon startup, memcached is first pre-populated with keys from range 0 to 10000. Then each client randomly picks a number in this range and requests the value for that key. In our workload, all of the GET requests are hits and successfully return a random string.

5. Analysis

We started up the memcached server, prepopulated it with values and let the 9 client machines continuously request values from the server. For the analysis, we used keys and values of size 4 bytes. We ran Oprofile and profiled the entire system, including kernel code. When we first profiled the system, we noticed that memcached was spending a lot of its time inside the kernel, and the network driver. So we switched on additional options in OProfile that enabled us to see *where* inside the kernel and the network driver that memcached was spending time.

5.1 Observations

1. A lot of the time ($\sim$15%) is being spent in the code of the `e1000` network driver. Around 3% of time is spent on waiting for input.
2. Copying data from the user space to the kernel space also takes up around 3% of time. An additional 3% is spent in system calls for switches between the kernel-space and the user-space.
3. Around 2% of the time is spent in handling tcp packets. The profiled results show that all the data arrives in order - the handler for out of order data packets is never called.
4. Around 2% of the time is spent in idle mode - the `mwait_idle` kernel method is used to make the processor sleep, and wakes it up when a write is done in a monitored memory region.
5. Locking among threads also takes up around 2% of the time.

6. Malloc accounts for around 2% of the time during network packet handling. This was initially surprising, since memcached uses a slab allocator instead of malloc/free. However, slab allocation is done only for internally storing data items and not for network packet handling and other functions.
7. The code structure of memcached is very modular - For every function that needs to be mutually exclusive, there is a wrapper function that does locking, unlocking, and then calls the function that actually does the work.
8. The command interpretation of memcached takes around 2% of the time.
9. The hash of memcached takes around 1% amount of time. Given that the hash function is an O(1) operation, this is surprising.

5.2 Opportunities for optimizing memcached

1. Since a lot of time is spent in the network driver, it can be optimized for memcached's operations. This is one of the lines of optimization that we pursue in this work.
2. A lot of time is spent in switching between user and kernel space and copying data between the two. It should be kept in mind that the profiling was done on 1 KB sized values. For bigger sizes such as 256 KB, the amount of time spent for context switches and copying increases a lot. This is the second line of optimization that we pursue in this work.
3. TCP imposes a lot of overhead on memcached, both during connection startup and during data transfer. Performance could be improved if UDP is used instead of TCP. However, this will involve a complete rewrite of memcached's interactions with the network and additional implementation of reliability for get operations of memcached.
4. If memcached's workload is previously known, such as the key-value sizes, the TCP implementation can be optimized for that workload. For example, all the TCP packets can be made to be the same size of the memcached key-value pairs. This would involve considerable amount of work, tweaking the TCP implementation of the system.
5. Consider the actual workhorse functions and the wrapper functions in memcached. Given the amount of times that some functions are called, merging these 2 functions together and avoiding the function call overhead could give some performance improvement.
6. The command interpretation part of memcached could be optimized by replacing the text commands of memcached with a binary protocol. This is being worked on by memcached developers.
7. The hash function of memcached seems to be very complex. It is complex because it provides us certain properties. The hash function could be examined and simplified, cutting away unnecessary properties.

6. Design

From the analysis, we identified that memcached spends a significant amount of time in sending out packets over the network, and in copying information between the user-space and kernel-space. In order to reduce the time spent in network transfer, we looked at tuning the e1000 network driver to be more efficient for memcached. In order to reduce copying information between user and kernel space, we looked at providing operating system support for memcached in the form of zero-copy techniques. Sections 6.1 and 6.2 explain e1000 tuning and operating system support respectively.

6.1 Tuning the e1000 driver

The e1000 supports a feature called *interrupt blanking*[2]. Servicing an interrupt consumes some amount of processing power. Servicing a lot of interrupts all the time increases CPU load drastically. When interrupt blanking is enabled, the processor only responds to interrupts every few microseconds. This reduces the load on the processor. This feature is turned on by default in the e1000. While interrupt blanking reduces CPU load, it adversely affects the throughput since memcached is not responding as fast as it can to the requests. We turned off interrupt blanking or coalescing and measured throughput of the system.

When the system is handling a lot of packets, polling might be a more efficient way to get I/O than interrupts. Our preliminary experiments using polling demonstrated that using 9 clients is not creating enough traffic for polling to be effective. Hence it was not explored further. We also increased the recieve and transmit buffers in TCP to the maximum value allowed by the system.

6.2 Operating System Support

We investigated the option of providing operating system support for memcached. From the profiling, we noticed that there was significant amount of copying of data from user space to kernel space and back. We came across two interesting system calls that were added recently to the Linux kernel - *splice* and *vmsplice*[10]. These system calls allow a user-space process access to a kernel buffer in the form of a pipe. The vmsplice system call transfers data from user-space to a pipe. The splice system call transfers data from one file descriptor to another.

The key idea in both these system calls is that they are *zero copy*[3] - They minimize the amount of copying between user space and kernel space, and also within the kernel space. We expected to get some performance improvement by minimizing the amount of copying currently happening in memcached.

7. Implementation

We describe the hurdles we faced while implementing both the approaches explained in the previous section, and the optimization we made to memcached while implementing them.

7.1 Tuning the e1000 driver

Interrupt blanking in the e1000 driver is related to the `InterruptThrottleRate` parameter of the e1000 driver. The value set for this parameter is the maximum number of interrupts per second that the adapter will generate for incoming packets. We set the value of 0 for this parameter, which denotes that blanking is turned off - As each packet arrives, an interrupt is generated.

The parameter `RxDescriptors` specifies the number of receive buffer descriptors allocated by the driver. Increasing this value allows the driver to buffer more incoming packets, at the expense of increased system memory utilization. We set this parameter to the maximum value allowed.

The paramater `RxIntDelay` specifies the delay imposed on the generation of receive interrupts in units of 1.024 microseconds.Increasing this value adds extra latency to frame reception and can end up decreasing the throughput of TCP traffic. We set this parameter to zero to reduce the delay and increase throughput.

The parameter `TxDescriptors` specifies the number of transmit descriptors allocated by the driver. Increasing this value allows the driver to queue more transmits. Each descriptor is 16 bytes. We set this parameter to the maximum value allowed.

The parameter `TxIntDelay` specifies the delay imposed the generation of transmit interrupts in units of 1.024 microseconds. We set this parameter to zero to reduce the delay and increase throughput.

Setting these parameters was done by changing the options for the e1000 driver in the `/etc/modprobe.conf` file. We also carried out a preliminary evaluation of opportunistic polling. In opportunistic polling, the driver measures the number of packets arriving per second, and switches to interrupts mode under low load and polling in high load. This involved recompiling the e1000 driver with the `NAPI` option enabled.

7.2 Operating System Support

7.2.1 Increasing kernel buffer sizes

The splice and vmsplice system calls are zero-copy and hence should enable us to reduce the amount of data copying happening due to memcached. Splice and vmsplice use a pipe for communication and this pipe is implemented as a kernel buffer in the Linux kernel. This was the first hurdle we ran into while implementing splice support in memcached - The maximum size of a kernel buffer is set inside the kernel and cannot be changed via any system calls. So we modified the Linux kernel to support bigger kernel buffers - This was made difficult because of the complexity of the Linux kernel.

7.2.2 Using splice in memcached

Once we modified the Linux kernel to support kernel buffers of size up-to 256 KB, we needed to modify memcached to use splice[9] and vmsplice. This turned out to a non-trivial task because of the complexity of memcached's state machine and the asynchronous nature of vmsplice and splice. From the profiling, we zeroed in on the part of memcached that is responsible for sending messages over the network.

The amount of data transferred by splice from one file descriptor to another (in our case, from the pipe kernel buffer to the socket buffer) is not guaranteed to be the amount of data in the pipe kernel buffer. This might happen for a number of reasons, such as the socket buffer being temporarily full. In such cases, the splice system call returns the number of bytes actually transferred and sets the global variable `errno` to denote that all the data was not transferred. We needed to identify when this was happening and to call splice again. Note that we do not need to call vmsplice again, as all the data has already been transferred from the user-space buffer to the kernel buffer. This complicated error handling quite a bit and we needed to store separate state to check whether splice transferred all the data or not.

7.3 Optimizations

Once we implemented memcached with splice calls, we found that performance was terrible - There was a delay of about 1 second for each set/get operation. We needed to make some optimizations to memcached to make the performance better. It should be noted that `sendmsg` is already quite optimized and fine-tuned through wide usage, while splice calls are just starting to get used and hence will need to be optimized a lot to deliver the promised performance benefit.

7.3.1 Creating a pool of pipes at startup

Our first optimization was creating a pool of pipe kernel buffers at memcached setup - Because of the large number of connections, pipe creation was slowing down memcached by a considerable amount. Performance improved when pipe creation was moved to memcached startup time, rather than connection startup time. However, throughput of modified memcached was still around 30% lower than that of original memcached. While we were expecting some performance degradation, this was higher than what we expected.

7.3.2 Avoiding lookup costs for pipe-connection information

The second optimization was changing the structure of a memcached connection so as to store information about the kernel buffer pipe allocated to that connection. Previously, we had implemented a table that stored this information and upon connection startup, the table was scanned and a pipe was allocated to that connection and the information was stored in the table. However, the table lookup was reducing performance - Once we stored this information inside each connection, performance of modified memcached was on par with that of original memcached when we stored large key-value pairs in memcached.

8. Evaluation

For analyzing the performance of memcached and to measure the effects of our modifications, we decided to evaluate performance via 2 criteria - how memcached scales with the number of clients connecting to it, and how memcached scales with the size of the data objects stored in it.

Each test was performed at least 10 times, and the average value was computed and is represented in graphs. The standard deviation was also calculated and is shown in the form of yerror bars on the graph.

The maximum data value used in the experiments is 250 KB, since we increased the kernel buffer size in the kernel to a maximum of 256 KB. Increasing the kernel buffer size further caused the kernel to crash.

8.1 Splice vs Sendmsg

In our modification to memcached, we have replaced the usage of sendmsg with system calls of the splice family. For every sendmsg system call, there are now 2 splice family system calls being used - splice and vmsplice. This means that we are trading increased efficiency in copying data for additional system call overhead. Therefore, unless the amount of data copied is significant, and the inefficiency of sendmsg with regards to data balances the system call overhead, splice system calls will perform worse than sendmsg. This should be kept in mind while considering the following results.

8.2 Scalability with respect to number of clients

For the clients scalability test, we borrowed 24 machines from the Wisconsin Advanced Internet Lab (As mentioned in section 4). We varied the number of clients connecting to the server, and measured throughput. To further test scalability, we ran multiple instances of the client code on each of the 23 machines, effectively allowing us to have 46, and 69 clients connecting to the memcached server.

We performed this test with different data sizes - 1 KB, 80 KB, 120 KB and 250 KB. These data sizes were selected from all across the spectrum and emphasize different aspects of the workload. For example, the 1 KB workload has a lot of small packets that shift the focus from copying data to responding to interrupts and sending the packet over the network. On the other hand, the 250 KB workload focusses on copying and handling of data, with interrupt handling taking less importance.

8.2.1 1 KB Workload

The 1 KB workload has the clients bombarding the server with lots of small packets at a high rate. Thus, the capacity of the server to handle interrupts and efficiently send packets over the network comes into play, rather than how efficiently the server copies data back and forth between user and kernel space. Thus, using splice system calls is not expected to give any performance benefit here.

Figure 2 shows the performance of memcached for the 1 KB workload. The performance of original memcached, and our modified version, both with and without disabling interrupt blanking, is shown. As expected, modified memcached which uses splice system calls performs badly when compared to original memcached.

Up-to 10 clients, disabling interrupt blanking works well, consistently outperforming original memcached. However, for any number of clients above 10, the performance degrades. This behavior becomes less mystifying if Figure 3 is considered. The performance of memcached heavily depends on the load that the processor is under. Since the server has two processors, the maximum is 200. Whenever the load becomes equal to or greater than 180, memcached performance degrades very badly.

Looking at Figure 2, memcached with interrupt blanking disabled reaches 180 percentage CPU load at 10 clients. This is exactly the point at which its performance starts degrading. We believe on a more powerful processor, or one with dual cores, the technique of disabling interrupt blanking will scale to much more than 10 clients.

8.2.2 80 KB Workload

For the 80-160 KB workloads, the focus is on a mixture of how the server handles interrupts and how efficiently the data is being copied. Figure 4 shows the performance for this workload. Because of the larger data size, the number of packets sent and processed per second by the server is reduced.

For the technique of disabling interrupt blanking to work, the rate at which interrupts happen must be high. With 80 KB packets, the rate of sending packets is considerably reduced. Morover the number of clients which are connecting to the server is not high. Hence, memcached with disabled interrupt blanking does not outperform original memcached.

Modified memcached still performs worse than original memcached, but the percentage of degradation is now within 1% as compared to the 23% of the 1 KB workload (This value is for 10 clients). The additional copying that must be done for the 80 KB workload enables modified memcached to catch up to original memcached.

8.2.3 120 KB Workload

The 120 KB workload is at the middle of the range between 1 KB and 250 KB, and as such represents the middle-ground between interrupt handling efficiency and data copying efficiency. Figure 5 shows the performance for this workload. Once again, because of the larger size of the data, the rate at which packets arrive at the server decrease and disabling

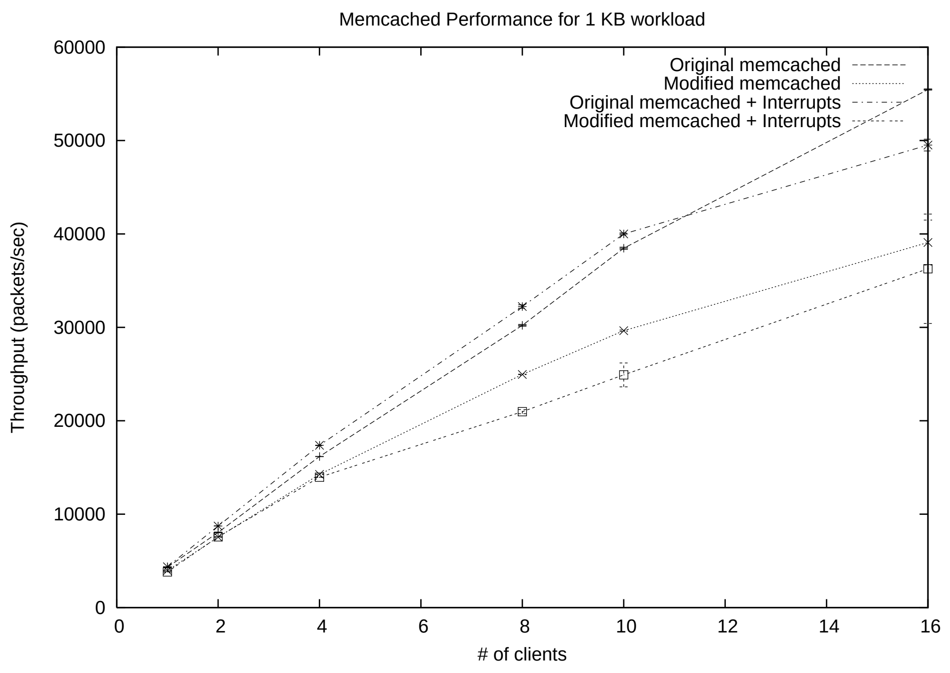

Figure 2. *Performance of memcached for 1 KB workload*. Disabling interrupt blanking improves performance consistently until the number of clients increases beyond 10. Modified memcached performs poorly due to small amount of data copying taking place.

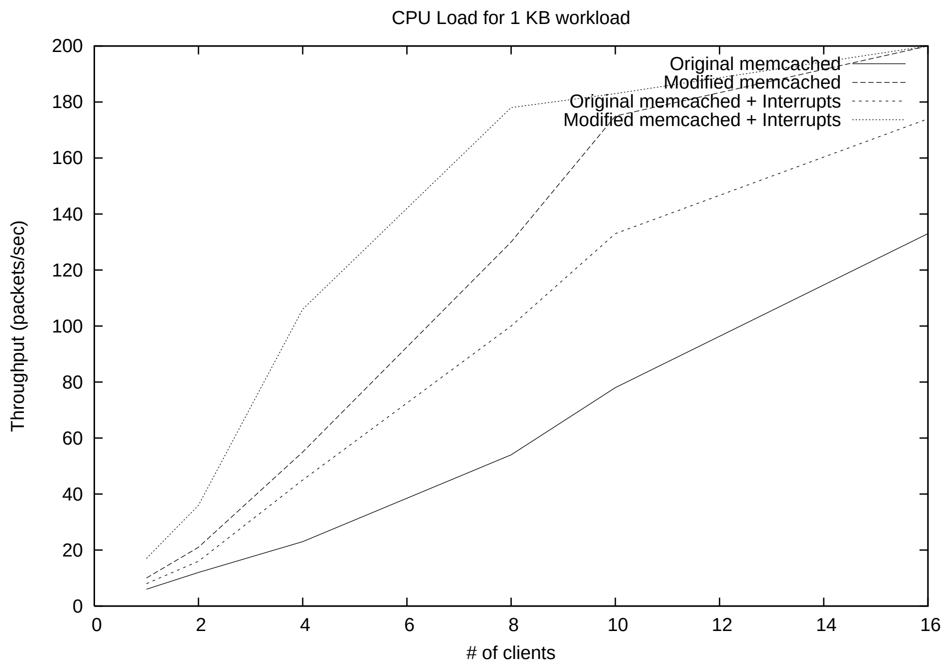

Figure 3. *CPU Load for 1 KB workload*. Original memcached scales beautifully, with the CPU load increasing only slightly as we increase number of clients. When interrupt blanking is disabled, due to the extra CPU load, memcached reaches peak CPU load earlier than normally.

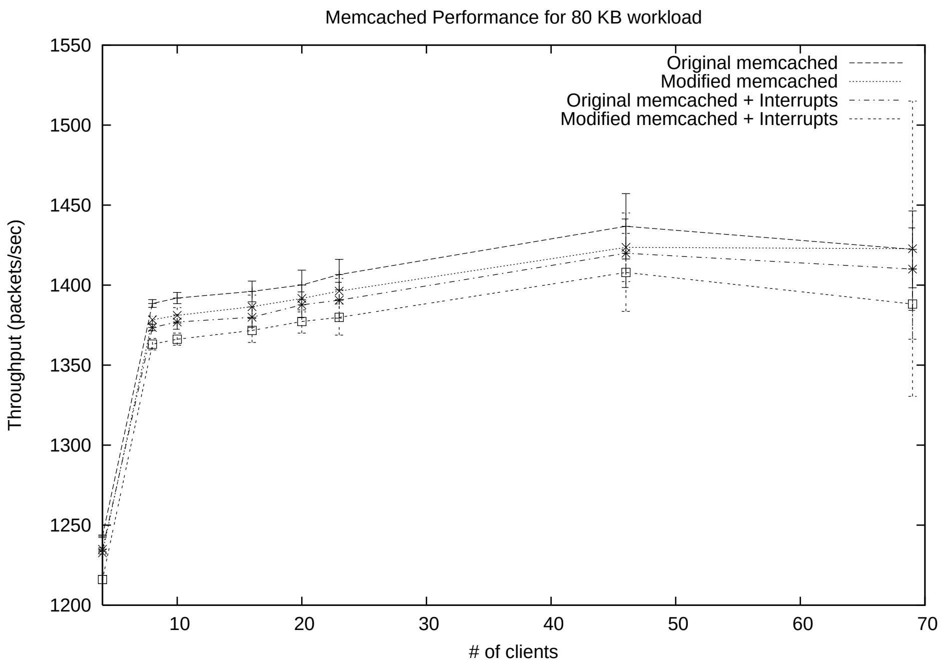

Figure 4. *Performance of memcached for 80 KB workload.* Due to the bigger data size, packet arrival rate reduces and memcached with interrupt blanking disabled performs poorly. Modified memcached catches up-to original memcached, with difference in performance being less than 14 or 1%

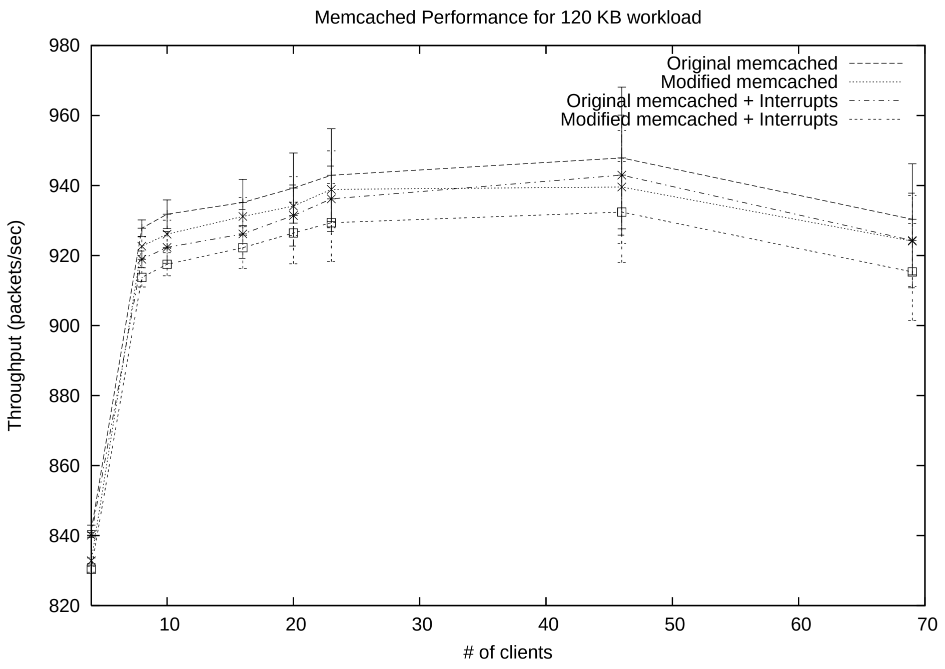

Figure 5. *Performance of memcached for 120 KB workload.* Disabling interrupt blanking does not give performance gains, while modified memcached still performs slightly worse (approx 1%) than original memcached

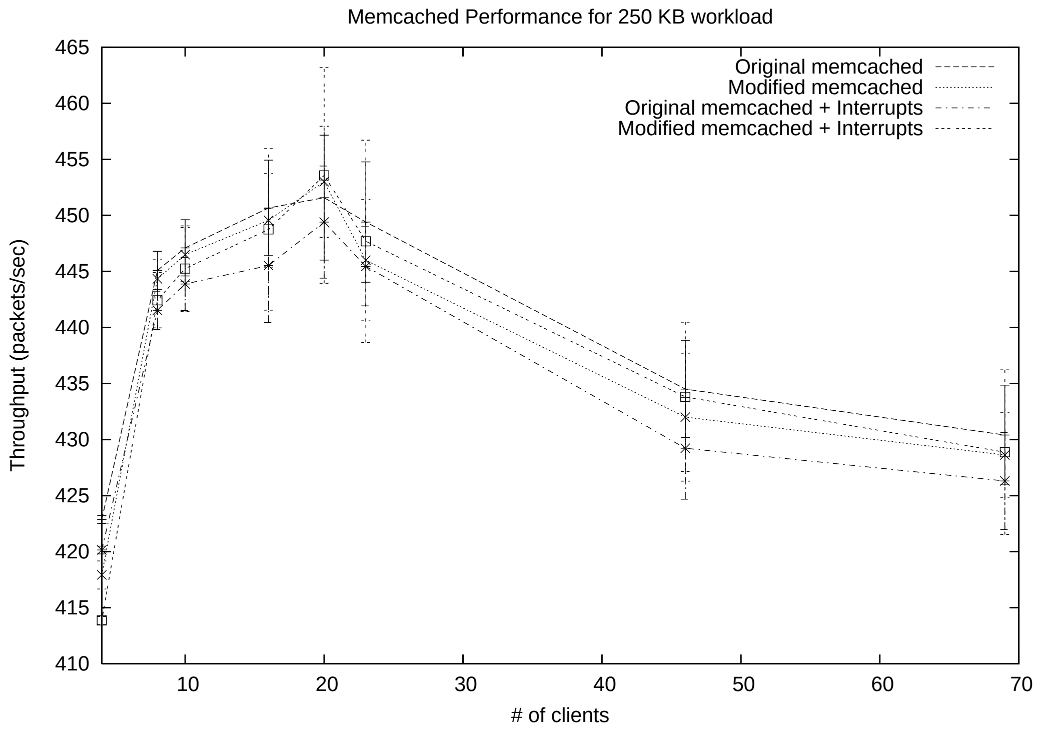

Figure 6. *Performance of memcached for 250 KB workload*. Modified memcached, while getting very close to original memcached's performance, is still unable to do better. Interestingly, modified memcached with interrupt blanking disabled does very well on this workload, performing better than just splice modifications or only disabling interrupt blanking.

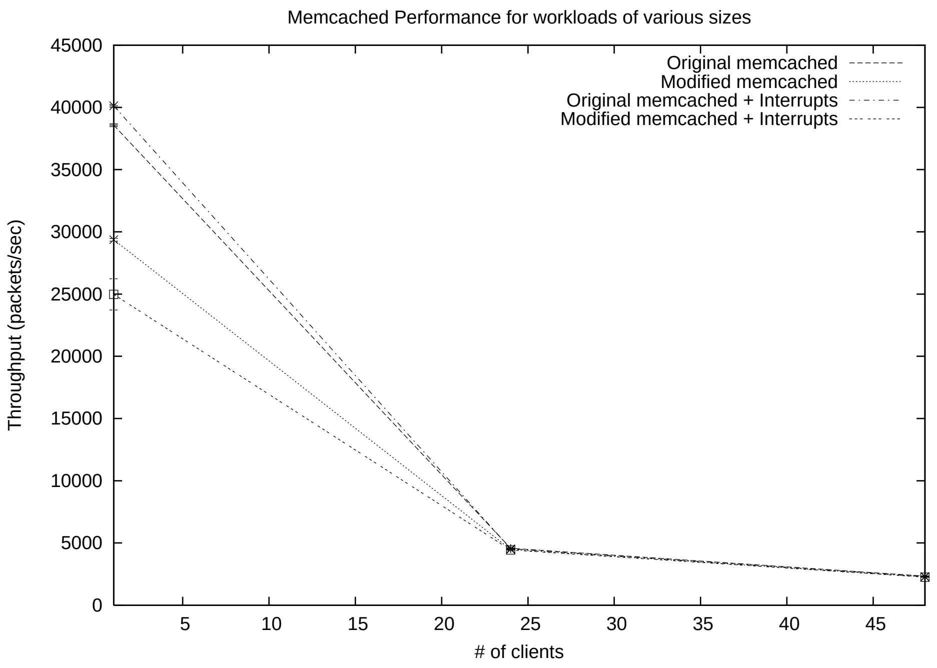

Figure 7. *Performance of memcached for workloads of sizes 1-48 KB*. Disabling interrupt blanking provides benefits when the data size is small, as this leads to more packets arriving at the server per second. Modified memcached's performance is very close to that of original memcached as data size increases.

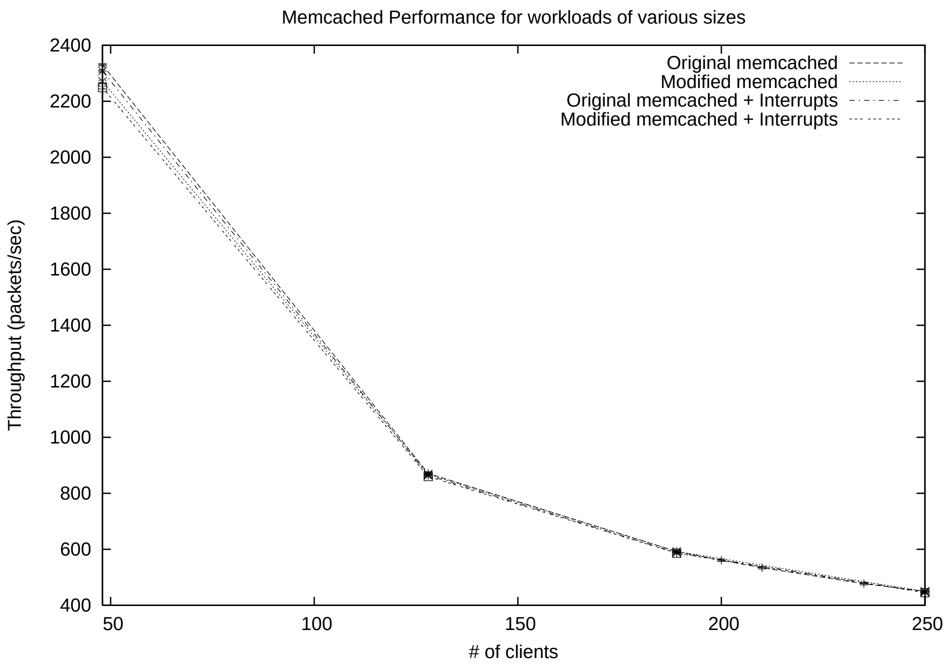

Figure 8. *Performance of memcached for workloads of sizes 48-250 KB.* As data size increases, the difference between the performance of original and modified memcached decreases. Modified memcached's performance is almost on par with that of original memcached.

interrupt blanking does not provide us with a performance gain.

Even though the curve for modified memcached looks well below that of original memcached in the graph, in reality, the performance degradation is within 1% of the original memcached values. Note that sendmsg has been in use for a long time, while splice is a newly introduced system call. Thus, it is not surprising that splice system calls need to be further optimized before providing performance gain.

8.2.4 250 KB workload

The 250 KB workload represents the end of the spectrum, with the focus being on data efficiency rather than interrupt handling. Because of the large data size, the rate of packets arriving at the server is low, ruling out performance gain due to disabling interrupt blanking.

Figure 6 shows the performance for this workload. We notice an anomaly at the 20 client point, where the performance of modified memcached (both with and without interrupts) is better than that of original memcached. Unfortunately, investigating the CPU load did not give us a clue as to why this anomaly occurred.

Looking at the curve for modified memcached with interrupt blanking disabled, we notice an interesting trend. In all of the workloads so far, modified memcached with interrupt blanking disabled has always performed poorly in comparison with the others. For the 250 KB workload, from 8 clients onwards, modified memcached with interrupt blanking disabled outperforms all the other modification techniques we tried namely, only splice, and only interrupt blanking disabled. It is within 1% of the performance of original memcached at all times. This is highly promising and may indicate the combination of our approaches has potential when the data size increases.

8.3 Scalability with respect to data size

For the data scalability test, we used 10 clients and used workloads of sizes varying from 1 KB to 250 KB. The reason that we used clients is that when interrupt blanking is disabled, the CPU load reaches the maximum for 16 clients, leading to performance degradation.

Figures 7 and 8 show the performance of memcached on these workloads. The reason that there are 2 graphs is that throughput values for smaller sizes of the workload is in the thousands, while throughput for the larger sizes of the workload is in hundreds. If both of these graphs were combined, it would be extremely hard to read. Note that the throughput is in terms of packets per second - So while the number of packets decreases as the size of each packet increases, the throughput in terms of bytes is sustained.

Figure 7 shows that for smaller workload sizes, disabling interrupt blanking improves performance. Beyond the 24 KB mark, the modifications and original memcached perform almost the same, thus corroborating the less than 1% difference in performance that was seen in Figures 4, 5 and 6.

From Figure 8, we can infer that as the data size increases, the performance of modified memcached reaches closer and closer to that of original memcached. This is expected since larger data sizes would cause more data copying. The performance of modified memcached is almost on par with the performance of original memcached. Further optimizations are needed for the splice system call family in order to provide performance gain.

9. Related Work

Facebook, having the largest deployment of memcached in the world, have gone to some lengths to optimize it[11]. They have implemented opportunistic polling, which switches to polling under high load and interrupts in low load. They have also replaced TCP with UDP. Both of these approaches are suggested from the profiling in our work. We did a preliminary test with opportunistic polling and concluded that, with our small experimental setup, we would not be able to see the benefit derived from it. Switching from TCP to UDP is a huge task and we would not have been able to complete it in the project deadline.

Our system-wide profiling was inspired by DEC's early work[1] on system-wide profiling. Shanti, an engineer at Sun has done some coarse-grained performance analysis[12] with memcached. However, their study does not explain what key-value pairs were used, how many clients were used and many other crucial details.

John Simons' blog[13] talks about interrupt blanking and how this can be used to increase throughput and reduce latency. We have utilized this technique in our work. [3] is a good resource for understanding zero copy. The splice system call is explained by Linus Torvalds, creator of linux, in this mailing list[14]. It is also discussed in [5] and [4].

10. Conclusion

In this work, we have analyzed the performance of memcached with respect to number of clients and size of key-value pairs. We have profiled memcached and identified possible bottlenecks. We have enumerated opportunities for optimizing memcached and have explored two such opportunities - Interrupt blanking and zero-copying. We have implemented both of these modifications in memcached and have analyzed the effect on performance. From the analysis, we conclude that:

1. The memcached application in user-space is very finely tuned and offers very little scope for further optimization.
2. Memcached offers a lot of opportunities for optimization in the network and kernel components part. An operating system and a kernel driver optimized for memcached is likely to give very high performance gains.
3. Disabling interrupt blanking will provide performance benefits when the number of packets arriving at the server per second are large
4. Theoretically, using splice system calls should provide performance gains. However, splice system calls have been introduced only recently and need more optimizations before they can equal the performance of long-used-and-optimized sendmsg.

References

[1] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[2] Intel. Linux* base driver overview and installation. http://www.intel.com/support/network/sb/cs-009209.htm.

[3] Linux Journal. Zero copy i: User-mode perspective. http://www.linuxjournal.com/article/6345.

[4] LWN.net Mailing List. And what becomes of zero-copy? http://lwn.net/Articles/178682/.

[5] LWN.net. Some new system calls. http://lwn.net/Articles/164887/.

[6] Memcached. Memcached official website. http://memcached.org/.

[7] Trond Norbye. Communictation from trond norbye to anatoly vorobey. http://code.sixapart.com/svn/memcached/trunk/server/doc/memory_man

[8] OProfile. Oprofile. http://oprofile.sourceforge.net.

[9] Splice Wiki Page. Splice (system call). http://en.wikipedia.org/wiki/Splice_(system_call).

[10] Linux Man Pages. Splice man page. http://manpages.courier-mta.org/htmlman2/splice.2.html.

[11] Paul Saab. Facebook blog: Scaling memcached at facebook. http://www.facebook.com/note.php?note_id=39391378919/.

[12] Shanti. Shanti's blog:memcached performance on sun's nehalem system. http://blogs.sun.com/shanti/entry/memcached_on_nehalem1/.

[13] Josh Simons. The navel of narcissus: Solaris tcp latency for hpc. http://blogs.sun.com/simons/entry/solaris_tcp_latency_for_hpc.

[14] Linus Torvalds. Linux: Explaining splice() and tee(). http://kerneltrap.org/node/6505.