Managing Interference for Latency-sensitive Workloads

Jacob Leverich and Christos Kozyrakis
Computer Science Department
Stanford University
{leverich, kozyraki}@stanford.edu

ABSTRACT

Providing quality-of-service guarantees for latency-critical services and increasing the average utilization of servers are two important challenges for warehouse-scale datacenters. Unfortunately, these two goals are in conflict as latency guarantees break down as more workloads are placed on a server. As a result, servers mostly operate at low utilization (10% to 50%) [8], leading to increased capital and operational expenses for the datacenter.

This work focuses on what it takes to run servers at 100% utilization while providing QoS guarantees on latency-critical workloads. We use memcached as a canonical latency-critical service. Starting with a carefully optimized baseline configuration, we characterize the contributors to its latency and how load relates to performance and latency QoS violations. Next, we quantify the sensitivity of memcached to hardware and software sources of interference and discuss how existing isolation and scheduling mechanism can mitigate the impact.

We demonstrate how to place additional workload on dedicated memcached servers, raising the server utilization to nearly 100% without impacting the memcached QoS at low and high loads. We also demonstrate how to place a memcached service on servers dedicated to other applications in order to benefit from underutilized memory resources. Depending on the base load in this case, we raise server utilization by up to 60%. More important, we get significant memcached throughput with QoS guarantees without impacting the original workload. Finally, we discuss opportunities for further isolation and interference management mechanisms.

1. INTRODUCTION

Warehouse-scale datacenters (DCs) host tens of thousands of servers and consume tens of MWatts. These facilities support popular online services such as search, social networking, webmail, video streaming, enterprise tools, online maps, automatic translation, and cloud computing and storage platforms [18]. We have come to expect that these services provide us with instantaneous, personalized, and contextual access to terabytes of data.

This work focuses on the apparent conflict between two important challenges for warehouse-scale datacenters [7]: (1) the need to maintain high quality-of-service (QoS) for latency sensitive workloads, and (2) the desire to improve resource efficiency. Low latency is important for complex, user-facing services. For instance, updating a social networking news feed involves queries for the user's connections and his/her recent status updates; ranking, filtering, and formatting these updates; retrieving related media files; selecting and formatting relevant advertisements and recommendations; etc. Since tens of servers are involved in each user query, low average latency is not sufficient. The requirement is for low tail latency (e.g., low 95th or 99th percentile) so that latency variability does not impact a significant percentage of user requests.

High resource-usage efficiency is important because it is directly linked to total cost of ownership (TCO) and scalability. DC servers are typically utilized at 20% or below [8], with individual hardware components utilized at even lower levels due to the difficulty of balancing resources within a server [21]. Low utilization is particularly wasteful because servers are not energy proportional and consume a significant percentage of peak power even at low utilization. Increasing average server utilization by an integer factor (e.g., 2x or 3x) reduces the capital and operational expenses for a certain level of service or provides the headroom needed to support new services or additional users.

Unfortunately, assigning more work to each server in order to raise utilization typically leads to higher latency and higher variability. Latency-critical requests may be queued for milliseconds if other tasks occupy processing cores. But even if cores are available, the latency-critical requests may underperform due to interference and contention on shared resources such as caches, memory, storage, and networking. Hence, it is common for latency-critical services to be deployed on dedicated machines, which are underutilized for long periods of time.

The goal of this paper is to analyze the interference between latency-critical services and other workloads in modern servers and suggest ways to manage interference so that utilization can be increased without an impact on tail latency. We focus on memcached, a framework for distributed caching on DRAM [2], as the latency-critical service. Memcached is widely used by several companies in order to overcome the high latency of disk storage or database queries. Companies like Facebook have thousands of dedicated servers for memcached and place strict requirements on tail latency. Memcached involves processing cores, caches, memory, and networking, hence it is a good proxy for other low-latency services.

We start with a carefully optimized configuration of memcached in order to ensure that underutilization or interference patterns are not a side-effect of a suboptimal setup. We answer two questions. First, how can we place other load on memcached servers during periods of low utilization without affecting the throughput and latency of memcached? Second, how can we place a memcached service on any DC server, so that we make good use of any underutilized memory resources, all without affecting the primary workload and while providing good QoS guarantees for memcached? In the process of answering these questions, we characterize hardware and software interference sources for low-latency workloads, analyze the available mechanisms for enforcing isolation in modern servers, and comment on proposed isolation mechanisms that future systems could support. Our main contributions are:

1. We present a carefully tuned baseline deployment of memcached for a 2-socket commodity server that can process 864K queries per second (QPS) with a 500μsec 95th percentile latency (QoS). We describe how to best scale memcached with available resources and explain where and how latency QoS violations manifest under load.
2. We quantify the sensitivity of memcached to hardware and software sources of interference. We conclude that while memcached is susceptible to interference on execution resources, shared caches, and memory channels, careful OS scheduling and interrupt processing based on network flow affinity can help achieve high QoS even in the presence of interference.
3. We demonstrate how to configure the system in order to place additional workloads on a dedicated memcached server operating at either low or high load. We raise the server utilization nearly 100% without impacting the throughput or latency QoS of memcached.
4. We demonstrate how to deploy a memcached service in order to utilize any available memory on a DC server dedicated to other workloads. Depending on the resources used by the other workloads, we can increase the server utilization by up to 60% and get 80K to 600K QPS with high QoS guarantees from memcached without a significant performance impact for the other applications.
5. We show that there are further opportunities for hardware and software mechanisms that provide isolation between workloads and mitigate interference.

The rest of this paper is organized as follows. Section 2 reviews our baseline memcached configuration and how it was optimized. Section 3 presents the analysis of hardware and software interference on memcached. Section 4 discusses how we can increase utilization on DC servers without impacting memcached QoS. Finally, Section 6 presents related work and Section 7 concludes the paper.

2. MEMCACHED: A CANONICAL LOW-LATENCY SERVICE

Memcached is a distributed key-value store, commonly used as a query cache between front-end web-application servers and back-end database servers. It exposes a simple protocol to allow clients to set and retrieve objects based on a lookup key. Originally developed at LiveJournal, memcached is famously deployed by Facebook to alleviate load on MySQL servers [4]. Unlike persistent storage systems, memcached behaves as an opportunistic cache. It is assigned a memory capacity limit when deployed; when this limit is reached, new objects replace old objects according to some replacement policy.

In a datacenter context, memcached is deployed as a distributed hash-table service across 10s to 1,000s of servers. Clients direct requests to specific servers according to a mapping between object keys and servers. Although early adopters used simple mapping functions (i.e. modulo arithmetic), modern deployments use consistent hashing [20] to ensure even object distribution and to minimize object migration when servers are added or removed from a memcached pool. Often, clients make requests for a large set of keys all at once (called a *multi-get*). These multi-get requests get divided and sent in parallel to the servers where its keys reside. The responses are later aggregated by the client.

Internally, memcached is implemented on top of `libevent` [33], which itself uses Linux's `epoll` mechanism to achieve high connection scalability. It is a canonical example of an *event-driven* daemon [11, 15], where a given memcached thread or process manages multiple clients and connections. This is in contrast to other daemons, such as the Apache Web Server, which often assign processes or threads to a single connection. Event-driven services can handle huge numbers of simultaneous connections without the overhead of constantly context switching, or having to time-share a processor.

We choose to study memcached for several reasons. First, memcached and its variants are widely used in many online services. Given the pressure for low latency access to increasing volumes of data, the importance and DC footprint of such services will only increase. Second, memcached achieves response times that are orders of magnitude faster than those expected of traditional enterprise benchmarks, like SPECWeb2005 or SPECMail2009 [1] which require 95th percentile latency on the order of several seconds. Thus, memcached is susceptible to interference not typically seen with these workloads. Third, memcached is open-source, easy to deploy, and its source code is quite concise (roughly 9,000 source lines of code). Thus, it is tractable to experiment with its configuration, understand its data-flow, and characterize the contributors to its response latency. Finally, memcached exhibits similarities to a growing category datacenter workloads (REDIS, Voldemort, RAMCloud [30]) for which no experimental evaluation of interference or QoS has been published. Lessons learned from memcached should apply to these workloads as well.

2.1 Quality of Service

Response time is a vital metric for the health of online services and is distinct from peak performance in terms of queries per second (QPS). For instance, Mayer reported anecdotally that experiments at Google found increases to delay as small as 500msec in rendering Google Search results caused a drop in revenue as high as 20% [24]. Most "Web 2.0" benchmarks incorporate QoS requirements and report a performance score based on the maximum QPS which meets some QoS criteria (i.e. 95% of responses take less than 5 seconds). Barroso [7] argues that tail latency (e.g., the 95th percentile) is more important than average latency when evaluating online services. If 1 in 100 requests takes >1 second, the average response time can still be quite low. However, if a user request is processed by 100 servers, as it is often the case, there is a 63% chance that some server will take >1 second to reply.

The QoS constraints for services like memcached are extremely tight and are measured in microseconds [35]. First, given high-end networking and DRAM-based storage, such low latencies are feasible. Second, since dozens of back-to-back requests may be required to construct the reply to a user request, the time budget for each request is quite low. For the purposes of this study, we select 500μsec 95th-percentile response time as our QoS constraint. The average latency is typically quite lower. In our discussions with operators of large-scale services that utilize memcached, this QoS constraint was considered appropriate. In the rest of the paper, QoS-constrained QPS (QoS-QPS) refers to the throughput reached with this QoS constraint, while peak QPS refers to peak throughput without any latency consideration.

2.2 Experimental Setup

We use a commodity Westmere-based system as our memcached server. This system is representative of the 2-socket servers that have emerged as the price/performance sweet-spot for DC deployments [18, 21]. Memcached servers are typically configured for high memory capacity but do not include the most powerful processor chips available at the time [27]. Our system has two Intel

Xeon L5640 processors (2.27GHz, 6 cores, 2 hardware threads per core) and is provisioned with 48GB of DDR3 DRAM evenly spread across memory channels. It runs Ubuntu 11.10 and the Linux-3.0.0-22 kernel. We use memcached-1.4.9 and libevent-2.1. We disable frequency and voltage scaling, and limit the system to the C1 power management state. We disable C6, as the wakeup latency from this state (~180μsec) is almost 10 times longer than the latency of a typical memcached request, which prevents us from getting clean latency measurements in Sec. 3.1. Meisner details the applicability of core-parking and other power management modes in datacenter contexts [26].

To amortize the server's cost and better utilize its resources, we used an Intel 10Gb Ethernet NIC based on the 82599EB chip and the IXGBE driver version 3.9.15. This is an enterprise-grade NIC with support for features like multiple RX/TX queues, LRO (Large Receive Offload), TSO (TCP Segmentation Offload), and RSS (Receive-Side Scaling). It also includes Intel's Flow Director, a configurable hash table to steer packets to different queues. Without RSS, a single core would receive all of the interrupts and quickly become overwhelmed at 10GbE rates. With RSS, as each packet arrives, it's TCP/IP *5-tuple* (protocol, src, dst, src port, dst port) is hashed and steered to a RX/TX queue. Each queue of the NIC is assigned a different MSI-X interrupt address, and the I/O APIC can route these to different cores. Each core gets its own private RX/TX queue pair. This distributes interrupt handling and TCP/IP connection handling across all cores in the system. The IXGBE driver provides an additional feature called ATR, which establishes "flow-to-core" affinity (which we'll call flow affinity from now on). The driver dynamically updates the Flow Director hash table in order steer packets towards the processor running the user-application that will ultimately handle them [32, 38]. It inspects outgoing packets on each TX queue to see if new flows (unique 5-tuples) are present. When it sees a new flow, it inserts a rule into the Flow Director hash table to steer return packets to the RX queue associated with the TX queue. Thus, return packets from a remote host interrupt the same core that is transmitting on this same flow. As we will see in later sections, flow affinity turns out to be a critical performance isolation feature.

2.3 Load Generator

To perform detailed analysis of memcached, we required a tool which could (1) fully saturate a memcached server, (2) issue requests at a controlled rate, (3) accurately measure 95th percentile response time, and (4) include minimal client-side jitter. We found no existing tool that satisfied (1) or (2), and clumsy attempts at (3). As such, we developed our own in-house memcached load generator called *mutilate*. Mutilate is implemented using the same *libevent* library that drives memcached. It is multi-threaded but uses essentially no locking except for barriers to begin and end measurement runs. Each thread is allocated its own `event_base` and handles its own connections. Each thread can service a large number of connections and supports connecting to multiple servers simultaneously. Most importantly Mutilate is distributed and supports a "swarm-mode", where remote servers can be enlisted to generate load on a set of memcached servers. This enables us effortlessly saturate a memcached server. Furthermore, by enlisting a client dedicated to taking latency measurements, we experience little measurement jitter at the client-side.

Given a target QPS for the memcached server, a fraction is assigned to each connection in a Mutilate swarm, and requests are generated at that rate according to an exponential distribution. Connections are established and warmed up before taking latency measurements, as memcached connections are cached by most real-world clients. We determine "peak QPS" by running connections flat-out (no inter-transmission time), and we determine QoS-constrained QPS by binary searching on target QPS and inspecting 95th-% latency. We short-circuit this search when the span between "high" and "low" is smaller than 2%. While relatively stable, we occasionally see error as high as 5% when performing this process.

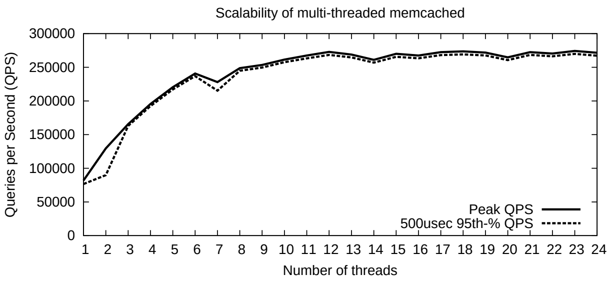

Figure 1: Peak and QoS-constrained QPS for one memcached process configured with 1 through 24 threads.

Mutilate can be configured to vary key-size, value-size, dataset size, key popularity (uniformly random or Zipf), degree of request pipelining, number of connections, and request distribution (set vs. get). In our experimentation, we found memcached to be relatively insensitive to dataset size so long as it greatly exceeded the size of the L3 caches. Interestingly, we found memcached insensitive to changes in the distribution of key popularity; we show in Section 3.1 that the vast majority of request latency is due to the Linux kernel and socket handling, and not the actual memcached code.

To eliminate ambiguity in latency measurements, we decided to use fixed key and value sizes (30 and 200 bytes respectively). The sizes we chose are based on Atikoglu's analysis of Facebook's memcached installation [4]. The principle problem with variable value sizes is that spurious large requests (i.e. 100's of KB) not only take a long time to service, but delay the processing of subsequent smaller requests. This queueing delay is an example of quality-of-service degrading interference that memcached can induce on *itself*, without outside influence. As this paper is more about understanding how low-latency workloads interact with external work than about fixing problems with memcached in particular, we focus on small, fixed-size values.

Our experimental platform is as follows. The measurement client is connected to the memcached server by 10Gb Ethernet via a Pronto 3290 switch (4x 10GbE, 48x 1GbE). 29 other client machines (comprised of a mix of Xeon 54xx- and 56xx-based servers) are connected to the switch via a mix of 10Gb and 1Gb links, for a total of 272 client hardware threads. Each client thread makes one connection per memcached process running on the server (as discussed in the next section). In essence, we treat this setup as if there were 272 front-end application servers communicating with one back-end memcached server. In reality, there would be a large number of memcached servers. Since hashing is used by clients to distribute requests across servers, back-end servers are reasonably load-balanced and we can limit our study to a single server.

2.4 Memcached Scalability

Before co-locating memcached with other workloads, it is important to carefully configure memcached itself. Otherwise, any conclusions about its resource usage and the likelihood of interference could be misguided.

Memcached scales poorly with multiple threads. Figure 1 shows the QPS achieved by one memcached process as we scale it to use

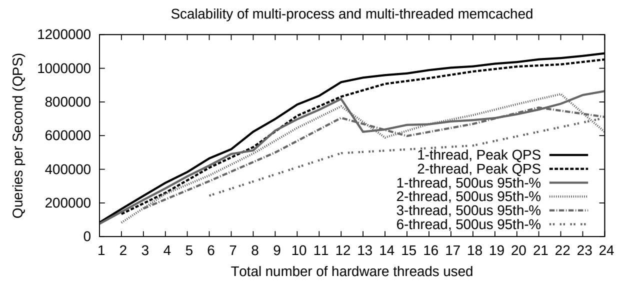

Figure 2: Peak and QoS-constrained QPS for multi-process memcached configured with 1 to 6 threads per process.

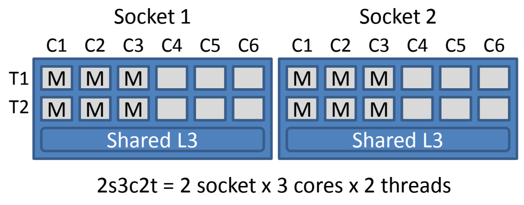

Figure 3: Example memcached configuration with 12 processes (2s3c2t). We refer to each layout by specifying XsYsZt where X, Y, and Z are the number of sockets, cores, and threads a workload is spread across, respectively.

24 threads. Threads are first assigned to all cores of each socket before utilizing hyperthreads. QPS plateaus at around 8 threads and both peak and QoS-constrained QPS saturate at roughly 250K. A similar scalability bottleneck was reported by Berezecki et al. in their evaluation of memcached on a Tilera TILEPro64 platform [9].

The solution to achieve high performance with memcached is to use multiple processes, each managing a separate portion of the server's memory and a separate shard of the overall key space [10]. Figure 2 shows the scalability of memcached when multiple processes are deployed and each process uses 1, 2, 3, or 6 threads. We scale memcached from one process up to however many processes are required to use every hyperthread in the server. For example, in the 6-thread case, we run 4 instances of memcached concurrently. Each process listens to a separate port. With multiple processes, memcached scales perfectly to 12 hardware threads and, as expected, linearly but at a slower rate when hyperthreads are used. The change in slope at 13 hardware threads is an artifact of how we assign processes to hardware threads. At 13 hardware threads, one of the threads is sharing an execution pipeline with another thread. For peak QPS, this simply reduces the effective contribution of the 13th thread. For QoS-constrained QPS, it results in a marked dip, since the 2 processes sharing a core (15% of 13 total processes) process requests slower than the other 11 processes. This slow-down is obvious in measurements of 95th-% latency, but is diminished when looking at the average. The effect disappears when all of the cores are load-balanced (i.e. at 24 hardware threads). A uniformly linear scalability trend is observed when work is packed onto cores rather than spread across cores.

Overall, the 24-process 500μsec QPS with 1-thread per process is 864K, whereas the 12-process QPS is 818K (a 5% drop). The 24-process peak QPS is 1,088K QPS, which is the highest reported QPS for non-pipelined memcached requests that we are aware of. Note that this is a relatively low-end Westmere system. We have measured QPS in excess of 1.3M on a Xeon X5670, and even crest above *3.0M QPS* when dozens of requests are pipelined over each connection. In any event, we believe that this captures the performance that multi-threaded memcached will achieve once its scalability bottlenecks are addressed (e.g., lock contention). Multi-threaded memcached fundamentally behaves very similar to multi-process memcached. Connections are statically assigned to threads, so from a queueing theory perspective each thread behaves like a separate M/M/1 queue, rather than n threads behaving like a single M/M/n queue. Finally, any concern regarding load-balance across memcached processes already exists across servers when considering large-scale memcached pools with thousands of servers.

The NIC features are critical to achieving this level of performance. In particular, multiple RX/TX queues and *receive-side scaling* (RSS) are vital to distribute interrupt and TCP/IP processing across hardware threads. Without RSS, we could not exceed 500K packets per second. We should note that there can be an apparent benefit to disabling the NIC's flow affinity when memcached is deployed on a fraction of the server's hardware threads. By steering interrupts away from the hardware threads that are running memcached, higher QPS can be achieved than when they handle interrupts and TCP/IP processing in addition to memcached. No such opportunity exists when memcached is deployed across all hardware threads, as the interrupt workload harms *some* memcached processes and causes a drop in 95th-% latency. Moreover, the interrupt handling on the non-memcached hardware threads impacts any other workloads that might be running on them. Thus, it is best with respect to performance isolation and 95th-% latency to have the NIC's flow affinity enabled and to steer interrupts towards the hardware threads that are running memcached. Flow affinity is enabled in all subsequent experiments.

2.5 Rightsizing Memcached

So far, we assumed that memcached can use all resources in a server. However, when we attempt to run memcached on DC servers that already support some other load, a smaller number of resources will be available. The question we must ultimately answer is, "given a subset of the resources on a server, what's the best way to deploy memcached"? Figure 3 introduces the notation that we use for this exploration. A configuration specifies how many sockets, cores, and threads memcached is distributed across. Table 1 presents the results of this analysis. The 24-process configuration is shown for reference, but there are really no options here: you should utilize every hardware thread when aiming for peak performance.

When only one socket or half of each socket is available (i.e. 12 hardware threads), we again find that it best to use all the available hardware threads. Unsurprisingly, it is also best to spread memcached out across sockets than to consolidate it on a single one (see 1s6c* vs. 2s3c*). By spreading it out across sockets, we reduce the degree to which memcached contends for the L3 cache amongst its siblings. However, this effect diminishes as the number of memcached processes becomes small (N=4 and 2). All of this suggests two simple rules for deploying memcached in resource constrained environments: (1) *spread it out across sockets*, and (2) *use both hardware threads of each core it occupies*.

We also investigated two other crucial issues with regard to how memcached is deployed: (1) the impact of forcing memcached to timeshare with itself, and (2) the impact of pinning memcached to cores. As seen for 24/2s6c1t in Table 1, QoS-constrained QPS drops significantly when memcached is forced to timeshare with itself, even though Peak QPS is only moderately affected. This makes sense, as a given memcached process may have to wait on the order of several milliseconds for its turn to use the processor when timesharing is initiated, even though the processor can still

N	Conf.	QoS-QPS	Peak QPS	Util %
24	2s6c2t	**864,509**	1,088,891	90.4
12	2s6c1t	716,524	897,934	53.5
12	1s6c2t	**425,457**	559,960	57.6
6	1s6c1t	408,557	451,989	33.1
12	2s3c2t	**445,176**	598,139	46.8
6	2s3c1t	412,610	472,122	32.5
4	2s2c1t	266,517	327,710	19.0
4	1s4c1t	263,236	317,372	25.1
2	2s1c1t	145,785	161,045	8.8
2	1s2c1t	146,667	158,840	7.8
24	2s6c1t†	590,303	907,681	44.8
24	2s3c2t†	342,245	580,649	24.8
24	1s6c2t†	309,903	540,112	19.7
12	1s6c1t†	223,615	449,245	36.6
24	2s6c2t⋆	518,774	1,049,392	82.5
12	2s6c1t⋆	453,031	887,575	59.4
12	2s3c2t⋆	262,737	584,258	49.0
12	1s6c2t⋆	272,499	548,107	56.6

Table 1: QoS-constrained and peak QPS for different strategies for deploying memcached to partially utilize a server. N specifies the number of memcached processes deployed. Conf. indicates the layout used for placing these processes (see Fig. 3). † indicates that this configuration requires memcached to time-share with itself, and ⋆ that processes are not pinned to distinct hardware threads. Util% is average CPU utilization as reported by Linux in `/proc/stat`. Green rows highlight the benefit to using both hardware threads of a core, and red cells highlight severe QoS regressions.

maintain high throughput. The conclusion here is that *it is better to deploy fewer memcached processes than force them to timeshare*. This conclusion will drive the way we co-locate memcached with other applications on a server.

Finally, in the course of these experiments, we observed significant variation in our results, particularly for multi-threaded memcached configuration. We were able to mitigate this variation by explicitly pinning each memcached process to specific hardware threads. We found that in the absence of explicit pinning, the Linux scheduler would frequently migrate two memcached processes to the same hardware thread and leave other hardware threads idle. This behavior occurs when memcached is run at less than peak QPS, such that the hardware thread is not 100% utilized. The consequence is that QoS-constrained QPS once again suffers due to time sharing. This presents an interesting conundrum when the system is underutilized. From the perspective of energy-proportionality, it is best to consolidate work onto fewer cores and power-gate the rest of them. However, if this leads to time-sharing on the remaining cores, it has a significant impact on QoS. We revisit this point in Section 5. In all of our subsequent experiments, we pin memcached processes to specific cores using `taskset`.

3. LATENCY AND INTERFERENCE

In this section, we seek to understand what contributes to the latency of memcached requests, and how interference manifests itself in terms of this latency and throughput degradation.

3.1 Life of a Memcached Request

There is substantially more to executing a memcached query than just looking up a value in a hash-table. In order to gain a more detailed understanding, we now trace the life-cycle of a memcached request in Linux. Figure 4 depicts the basic steps involved.

Who	What	Unloaded	Ctx Sw.	Loaded	L3 int.
Server	RX	0.9us	0.8us	1us	1us
	TCP/IP	4.7us	4.4us	4us	4us
	EPoll	3.9us	3.1us	2778us	3780us
	queue	2.4us	2.3us	3074us	4545us
	Read	2.5us	2.1us	5us	7us
	execute	2.5us	2.0us	2us	4us
	Write⋆	4.6us	3.9us	4us	5us
	Total	**21.5us**	**18.7us**	**5872us**	**8349us**
Client	End-to-end	49.8us	47.0us	6011us	8460us
	TX-to-RX	36us	30us	-	-
Switch	RX-to-TX	3us	-	-	-

Table 2: Latency breakdown of an average request when the server process is unloaded, when it is context-switching with another process, when it is fully loaded, and fully loaded and subject to heavy L3 cache interference. Client "TX-to-RX" is the time from when the client has posted a packet to its NIC until it receives a reply, and "End-to-end" is the time reported by Mutilate. Switch latency is estimated by connecting the client directly to the server and measuring the difference in latency. ⋆For brevity, we include TCP/IP and TX time in Write.

(1) A client initiates a request by constructing a query and calling the `write()` system call. We omit the TX details as we are not so interested in the client. (2) The request is then sent to the server's NIC via two cables and a switch, where upon the hardware thread running memcached receives an interrupt (3), by virtue of the NIC's flow affinity. Linux quickly acknowledges the interrupt, constructs a `struct skbuff`, and calls `netif_receive_skb` in *softIRQ* context (4). After identifying the packet as an IP packet, *ip_rcv* is called (5), and after TCP/IP processing is complete, *tcp_rcv_established* is called (6). At this point, the memcached process responsible for handling this packet has been identified and marked as runnable. Since there is no more packet processing work to be done, the kernel calls `schedule` to resume normal execution (7). Assuming memcached is asleep waiting on an `epoll_wait` system call, it will immediately complete the epoll and is now aware that there has been activity on a socket (8). If memcached is not asleep at this point, it is still processing requests from the last time that `epoll_wait` returned. Thus, when the server is busy, it can take a while for memcached to even be aware that new requests have arrived. After returning from `epoll_wait`, it will eventually call `read` on this socket (9), after which memcached finally has a buffer containing the memcached request. If `epoll` returns a large number of ready file descriptors, it executes them one by one and it may take a long time for memcached to actually call `read` on any particular socket. We call this "queue" time. After executing the request (by looking up the key in its object hash-table), memcached constructs a reply and `write`'s to the socket (9). Now TCP/IP processing is performed (11) and the packet is sent to the NIC (12). The remainder of the request's life-cycle at the client-side plays out similar to how the RX occurred at the server-side.

Using SystemTap [12], we have instrumented key points in the Linux kernel to estimate how long each step of this process takes. By inspecting the arguments passed to kernel functions and system calls, we are able to create accurate mappings between `skbuff`s, file descriptors, and sockets. Using this information, we can track the latency of *individual requests* as they work their way through

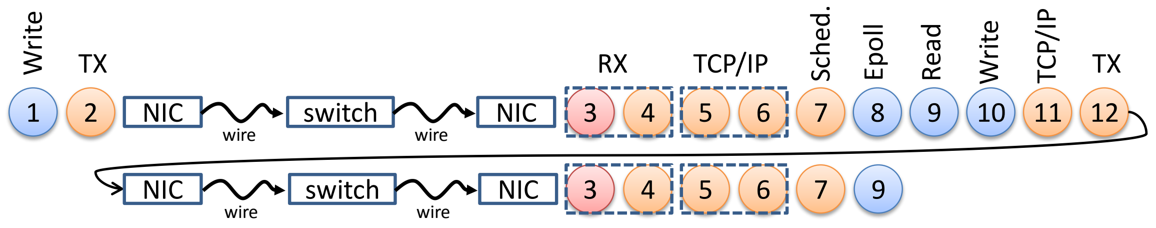

Figure 4: Life of a memcached request. Blue = system call, yellow = kernel activity, red = interrupt handler.

the kernel, even though hundreds of requests may be outstanding at any given time. This per-request latency attribution gives us far more insight into the exact nature of request latency than a regular run-time profile. Naturally, each packet observes variations in the contributions to its specific latency, so we take averages over millions of requests to observe the trend. We take measurements both for an unloaded case (where only one request is outstanding), a context switching case (where a cpu-bound task is running and the OS must context-switch to memcached after receiving a packet), a loaded case (where memcached is handling requests from all 272 clients at peak throughput), and an interference case (where we subject the loaded memcached to heavy L3 cache interference).

Table 2 presents the latency breakdown by condensing measurements into key periods: driver RX time, TCP/IP processing, waiting for Epoll to return (which includes process scheduling and context switching if memcached isn't already running), queueing delay, read system-call time, execution time, and write system-call time. In the unloaded case there are no surprises: TCP/IP processing, scheduling, and Epoll take a plurality of time. What is surprising is that in the context switching case, not only does it not seem to incur any overhead to context switch to memcached after receiving a request, *it actually goes slightly faster*. We limit the CPU to the C1 C-state, so this is not C2 or C6 transition overhead. We suspect that this overhead has to do with the C1 state, or the bookkeeping that Linux does when discovering that a CPU is idle. In any case, *context-switching is not itself a large contributor to memcached latency*.

Distinct from the unloaded case, our measurements of the loaded case gives us a key insight: *the vast majority of the latency when memcached is overloaded is queueing delay*. This queueing delay manifests itself in the measurement of "queue" time, but also "Epoll" time. When overloaded, `epoll_wait` is returning on average ˜200 ready file descriptors. Thus, it will take a while to get to any one packet. Second, since so many requests are being received by memcached at once, it will take a long time to process them all and call `epoll_wait` again. This shows up in the long "Epoll" time measured for subsequent packets. When subjected to interference (for instance, L3 interference), the moderate growth in the time it takes to process each individual request (Read, execute) results in a substantial increase in this queueing delay.

In all of the subsequent experiments, the queueing delay is generally the single biggest contributor to latency, second only to the latency caused by OS-mediated time-sharing. When interference causes memcached to process requests more slowly, it translates into longer queueing time. Given that memcached implements a common event-based design pattern, we expect that this form of queueing delay is a widespread phenomenon. In Section 5, we discuss alternative design patterns that could mitigate this queueing delay. At any rate, we hope the reader now appreciates how a millisecond request latency manifests from handling 20us requests.

3.2 Interference Analysis

μbench	Sh.	% QoS-QPS drop			Interference level		
		Sm	Med	Lg	Sm	Med	Lg
CPU	ST	-	-	-46.3	-	-	-
CPU.nice	ST	-	-	-2.6	-	-	-
CPU.fifo	ST	-	-	-2.1	-	-	-
CPU	SC	-	-	-31.0	-	-	-
L1D	SC	-26.3	-29.4	-40.0	8K	16K	32K
L1I	SC	-42.2	-37.6	-39.7	12K	24K	96K
L2	SC	-44.1	-46.9	-51.8	96K	160K	288K
L3	SS	-8.3	-20.8	-37.9	3M	6M	12M
Network	DS	-1.3	-1.4	-3.6	1Gb	3Gb	5Gb
Intr	SS	-	-	-4.1	-	-	-
Intr	ST	-	-	-48.8	-	-	-
MMAP	DS	-	-	-3.8	-	-	-

Table 3: Single-process interference results. The memory hierarchy and network microbenchmarks are run at several interference levels (small, medium, large). For memory workloads, the level indicates the capacity that is being contended for (e.g., size of the array being looped over). The network level refers to how many Gbps of traffic are being generated by D-ITG.

This section presents the results of running a range of microbenchmarks concurrent with memcached, each targeted to stress a particular shared resource. We divide this analysis into two parts. First, we explore interference that a single memcached process might be subject to at the *core*-level, were it to share to resources of a single core with another workload. Second, we scale up these experiments and explore the interference that multi-process memcached might be subject to at the *socket*- and system-level.

We use sysbench-0.4.12 as our CPU microbenchmark. To stress levels of the memory hierarchy, we extend it with a memory-stride mode (inspired by the canonical homework exercise found in Hennessy and Patterson [17]), a mode with large instruction sequences to stress the instruction cache, and a mode which `mmaps`, faults on, and `munmaps` a page. Since sysbench is multi-threaded, this MMAP mode can be used to cause a torrent of TLB shootdowns, and also stresses the kernel's memory allocator. We stress Linux's general interrupt handling by having two threads pinned to separate cores spinning on a pair of `futexes`. Each iteration generates a Rescheduling Interprocessor Interrupt (IPI), and invokes the scheduler. We can generate over 100,000 interrupts per second per core with this microbenchmark. We stress the network with TCP traffic generated by D-ITG [5].

We run these benchmarks concurrently with memcached in three different situations: (ST) running on the same hardware thread as memcached, (SC) running on a different hardware thread in the same core, (SS) running on a different core in the same socket, and (DS) running on a different socket altogether.

3.2.1 *Single-process Interference*

Table 3 presents measurements of the amount of interference single-process memcached can be subject to due to antagonists time-sharing a hardware thread, using a sibling hardware thread, sharing different levels of the cache hierarchy, and due to miscellaneous OS activity. All slowdowns are reported as percent drop in QoS-constrained QPS relative to single-process memcached running on its own (77K QPS).

As hinted at in Section 2.5, memcached is terribly sensitive to time-sharing a hardware thread. The several millisecond wait that it experiences between execution periods harms both QoS-constrained QPS and average latency (CPU/ST). However, the QoS-QPS drop is only 46%, which illustrates that *so long as requests arrive for memcached at a slow enough rate, it does not suffer long delays waiting for its turn to use the processor*. It is thus possible to achieve good latency at low QPS when memcached is co-located on the same hardware-thread as another workload. On the other hand, merely *attempting* to use memcached at higher QPS results in poor latency as OS-mediated time-sharing kicks in. This property is a natural result of the design of Linux's CFS scheduler. So long as memcached is not using more than its "fair-share" of a hardware thread, it remains at the head of CFS's red-black tree run queue. When Linux finishes processing an incoming packet, marks memcached as runnable, and calls `schedule()`, it selects memcached, even if another process was running before the packet arrived. It is also noteworthy that the observable latency of memcached jumps abruptly when time-sharing is initiated. For instance, below 46% QPS, we see latency in the 200-300 μsec range, where as above 46% QPS we see latency in the 3-6 millisecond range.

Nevertheless, by adjusting the relative priority of memcached with respect to other processes (CPU.nice/ST and CPU.fifo/ST), or making use of the POSIX real-time schedulers in Linux (`SCHED_FIFO` or `SCHED_RR`), we can largely protect memcached from having to time-share. We make use of this property in Section 4.1. Note that since memcached is continually yielding to the kernel in the course of its execution, its relatively safe to use the real-time schedulers, unlike the case of CPU-bound workloads.

Memcached suffers significantly from sharing a core with another hardware thread. Just sharing the pipeline causes a 31% QoS-QPS drop (CPU/SC). Additionally sharing the L1 data (L1D/SC), L1 instruction (L1I/SC), or L2 caches (L2/SC) results in further significant slowdowns (up to 52% with L2/SC). The drop for the L1 data cache seems to be sensitive to the amount of interference being caused (26% at 8K vs. 40% at 32K). However, memcached seems to be intolerant of even modest instruction cache interference. When memcached instead time-shares a hardware thread with these cache antagonists, we don't find a big performance drop relative to the CPU antagonist. Thus, the overhead of refilling the TLB and caches after context-switching is not so significant.

At the socket-level, memcached is sensitive to L3 cache interference (L3/SS). Note that in this experiment, no core resources are being contended for; only socket-wide resources. The degree of slowdown scales with the amount of contention present at the L3. In Section 4.2, we show that this L3 contention presents difficult challenges co-locating memcached with other workloads.

Unexpectedly, we found that network interference is not a substantial source of interference for single-process memcached (Network/DS). Even when 5Gb/s of traffic is arriving at the NIC, only minor slowdown is observed by memcached. In this experiment, memcached is running on one thread of one socket and a collection of processes streaming TCP data from a remote host are running on the other socket. The upshot is that the NIC's flow affinity does a good job of steering interrupts to the cores that are actually handling the streams, and do not impact the core running memcached.

μbench	Sh.	% QoS-QPS drop			Interference level		
		Sm	Med	Lg	Sm	Med	Lg
L3	SS	-34.4	-42.8	-55.1	6M	12M	24M
Mem	DS	-	-	-12.9	-	-	24M
Network	DS	2.9	-1.4	-2.4	1G	3G	5G

Table 4: Multi-process interference results. (SS) means memcached in the 2s3c2t configuration, sharing the cache with the L3 antagonist. (DS) means memcached in the 1s6c2t configuration, not sharing the any caches.

Finally, we find that miscellaneous OS interference does not cause significant problems. Specifically, constantly exercising the kernel's memory allocator (MMAP/DS) does not significantly impact memcached, not does a constant stream of IPIs to the same socket (Intr/SS), such as TLB shootdowns or rescheduling interrupts. A constant stream of rescheduling interrupts to the same hardware thread does impact performance, but little more than simply time-sharing (Intr/ST).

The main conclusions of this analysis are: (1) Memcached can run on the same hardware thread as other workloads, but only at reduced QPS (approximately 50%). (2) Memcached is sensitive to workloads running on sibling hardware threads, especially if they are cache intensive. (3) Memcached is also very sensitive to L3 cache contention. (4) It is largely insensitive to miscellaneous interrupts and OS memory management activity on different cores.

3.2.2 *Multi-process Interference*

The results presented in the previous section are incomplete, since a single memcached-process does not put maximum demand on per-socket resources, like L3 capacity or memory bandwidth. To explore how contention for these resources affects memcached, we repeat the L3, memory, and network experiments while running 12 memcached processes. To ensure that the results are not obscured by hardware multithreading or time-sharing, we pin the memcached processes either to a different socket from the interference (DS, using 1s6c2t as the configuration), or across half the cores of the same sockets (SS, using 2s3c2t as the configuration) as the interference. Table 4 presents the results of these experiments. QoS-QPS drop is relative to memcached running on its own in a given configuration, as presented in Table 1.

Memcached is even more sensitive to L3 cache interference when it is scaled up (L3/SS, 55% vs. 38% in the single-process case). We verified that this drop is not experienced when the L3 antagonist is running on a different socket. Additionally, memcached shows moderate sensitivity (12.9% QoS-QPS drop) to memory bandwidth interference. Finally, as in the single-process case, it shows little sensitivity to network interference on different cores.

In summary, we can state the following: (1) memcached is definitely sensitive to L3 contention, and moderately sensitive to main memory contention. Absent mechanisms to isolate this interference, co-locating it with other workloads requires careful consideration, as we will show in Section 4.2. (2) Even at scale, memcached is largely insensitive to network interference on different cores. Thus, flow-to-core affinity is a very effective isolation mechanism.

4. MANAGING INTERFERENCE

We now distill the knowledge about memcached's sensitivity to interference into practical advice for how to deploy it on shared servers in order to increase utilization. We consider two sharing

Type	Benchmarks
(i)	400.perlbench, 410.bwaves, 416.gamess
(f)	401.bzip2, 403.gcc, 434.zeusmp
(t)	450.soplex, 470.lbm, 471.omnetpp
(s)	429.mcf, 433.milc, 459.GemsFDTD

Table 5: SPEC benchmarks used as interfering workloads in Section 4. See the text for the definition of Type.

scenarios. The first scenario, which we call the "Facebook" scenario, is where memcached is the primary workload on cluster with thousands of servers. Suppose the cluster operator observes that the cluster is often underutilized. Can other work be co-located with memcached without impacting memcached? The second scenario, which we call the "Google" scenario, is where other workloads are the first-class citizen on a cluster. Again, suppose the operator observes that the cluster is not using its full compute resources and that that a significant amount of memory is unused. Can memcached run on these servers so that the free memory in the cluster can be utilized for some purposes (e.g., distributed caching)? Can we achieve both good quality of service for memcached and minimally impact the throughput of the original workload?

Methodology: To evaluate these scenarios, we use an assorted mix of SPEC CPU2006 benchmarks as the interfering or original workload, given in Table 5. We adopt Jaleel's methodology to divide SPEC benchmarks into four categories based on cache behavior: insensitive (i), friendly (f), fitting (t), and streaming(s) [19]. "Insensitive" have few L2 misses. "Friendly" benchmarks gradually benefit from increasing cache capacity. "Fitting" benchmarks abruptly fit into L3 caches of 1MB in size, and "Streaming" benchmarks thrash large L3 caches. Since memcached is mostly insensitive to high rates of interrupts and OS activity on different cores (Section 3.2), SPEC benchmarks that stress the memory hierarchy are sufficient to provoke the interference we are concerned with.

4.1 The Facebook Scenario

As memcached is the first-class citizen in the "Facebook" scenario, we assume that it is configured and deployed in order to achieve maximum performance. Thus, we assume that memcached is deployed with 24-processes in the 2s6c2t configuration from Table 1. We consider this configuration at four different loads: 25%, 50%, 75%, and 100% of the QoS-constrained QPS that memcached achieves on its own. This represents the various levels of utilization the memcached cluster may be at any point (e.g., 25% on an average day and 100% when a major event occurs). At each load point, we measure the interference caused by running mixes of from 1 to 24 of the SPEC benchmarks from each category alongside memcached. We construct these mixes by selecting one of each type of SPEC benchmark in round-robin order. Thus, with a mix of 24 benchmarks, we are running every benchmark in Table 5 twice. Since the memcached processes are pinned across hardware threads, *any* interfering load put on this server will have to context-switch or time-share with memcached.

Figure 5 plots the measured 95th-percentile latency of memcached across every point in this scenario, as well as the IPC that the interfering SPEC workload achieves (normalized to it running by itself). The baseline IPC of the SPEC workload ranges from 1.2 at 1 process to 13.6 at 24 processes.

At 25% QPS, the interference does not cause undue degradation on memcached. Memcached can still achieve latency far better than 500μsec. Moreover, the interfering workload sustains a significant fraction of its baseline IPC (around 60%). Thus, *at low utilizations, a significant fraction of idle cycles can be used without impacting memcached quality of service*. This result is a natural extension of the latency analysis related in Section 3.1; context-switching itself has little impact on memcached latency, and at 25% QPS, memcached is not using enough CPU for the scheduler to decide it needs to initiate time-sharing.

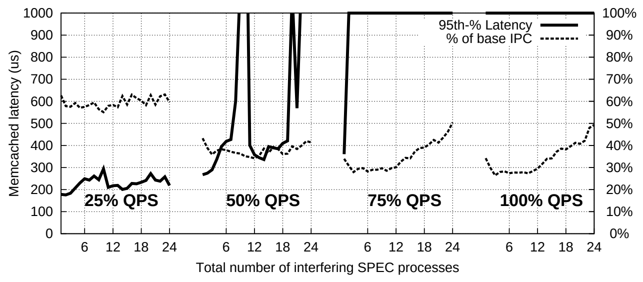

Figure 5: Baseline results of co-locating SPEC processes with memcached in the "Facebook" scenario. The measured 95th-% latency and normalized IPC of the SPEC workload is shown for 1 to 24 concurrent SPEC workloads at 4 different QPS rates: 25%, 50%, 75%, and 100% of the top QoS-QPS this server achieves running memcached alone.

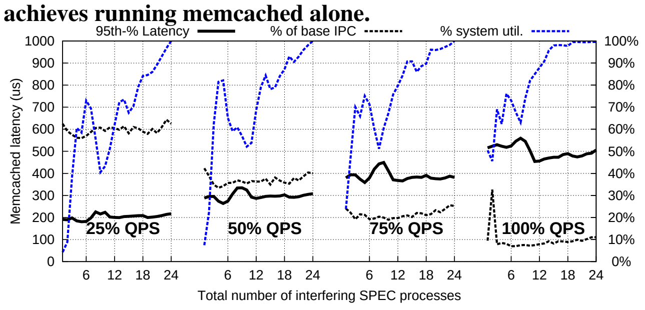

Figure 6: Results of co-locating SPEC processes with memcached in the "Facebook" scenario when using Linux's `SCHED_FIFO` scheduler.

Unfortunately, the picture is not so pretty at 50% QPS and above. For several instances at 50% QPS, memcached latency spikes up as high as 1ms. Above 50% QPS, latency is virtually always above 1ms, and sometimes crests above 10ms. Not plotted is the unsurprising fact that memcached is unable to sustain 75% or 100% QPS when there is a large amount of interference. The cause of this problem is obvious: starting around 50% QPS, memcached begins to use more than its fair-share of CPU time relative to other processes, and is victimized by the Linux scheduler. The spike in latency is sudden and abrupt. However, as we discussed in Section 3.2, this behavior is easy to mitigate by configuring the Linux scheduler to prefer memcached over the interfering workload.

Figure 6 again plots latency and IPC for memcached and the SPEC workloads. In this case, however, we've set the scheduling policy of memcached to `SCHED_FIFO` using `chrt`. Thus, the Linux scheduler will always prefer memcached over the interfering workload, and never force memcached to time-share. In nearly every load and interference scenario, memcached is able to maintain a fast 95th-percentile latency, with the exception of small violations (as high as 559us) at *100%* of the baseline QoS-constrained QPS. In each load scenario, the SPEC workload is still able to extract a meaningful fraction of its baseline IPC (10% to 60%). This means that we can extract additional value from every lightly or heavily loaded memcached server in a datacenter. Moreover, we can bring the apparent utilization of the server (as reported by `/proc/stat`) up to just shy of 100%. We obtain nearly identical results when using NICE levels as opposed to the `SCHED_FIFO` scheduler, though with slightly more violations at 100% QPS. The results at 75% and below are unchanged. We also found little variation when repeating this experiment over each category of SPEC workload individually (i.e. 24 copies of the (s)treaming benchmarks), causing at worst a 584us QoS violation at 100% QoS-QPS with streaming-only workloads.

Overall, we report the surprising result that properly configuring the NIC and simply setting memcached to have a significantly higher priority than interfering workloads is sufficient to protect it from QoS violations. The utilization of the memcached servers can be raised to nearly 100% with little ill-effect. This represents a major opportunity for DC operators to improve TCO by combining memcached servers with other work such as analytics. To realize this opportunity, the operator must reserve a small fraction of the server's memory for the other workloads. This is already the case in many servers, where the limiting parameter for memcached is QoS-constrained QPS, as opposed to capacity. We will discuss some further caveats in Sections 4.3 and 5.

4.2 The Google scenario

In the "Google" scenario, we assume that there is a server running some workload and its operator would like to co-locate memcached with it in order to expose any available memory for other uses. Memcached is a second-class citizen now and that there is a new QoS constraint: the operator is not willing to sacrifice more than 10% of the IPC of the primary workload in order to run memcached. We still require that memcached achieve better than 500μsec 95th-% latency in order for it to offer a worthwhile service. Thus, the problem of co-location memcached with the other workload requires us to find a configuration of memcached that degrades the other workload less than 10% and achieves reasonably high QoS-constrained QPS to be worth deploying. Like the "Facebook" scenario, we consider several different utilizations that a server might be at: 25% (3 cores occupied), 50% (6 cores), 75% (9 cores), 100% (12 cores), and "101%" (all 24 hardware threads). At the 50% point, we consider both the case where all of the cores on one socket are occupied, and also where half the cores on each socket are occupied.

Table 6 reports optimal memcached configurations at each utilization point given the new combined QoS constraint. Refer to the table caption for a description of the metrics. We find these configurations by first trimming the configuration space down to reasonable configurations, guided by the lessons we learned from Sections 2 and 3. The two most important rules are: (1) Do not deploy memcached processes across asymmetric execution resources (i.e. don't share hardware threads with the SPEC workload on some cores and not on others); as we saw in Figure 2, this can lead to a significant hit to 95th-% latency. (2) It does not make sense to deploy memcached in any configuration where it must time-share with itself (Section 2.5). After culling the configuration space, we find the QoS-constrained QPS that impacts the SPEC workload's IPC less than 10%. An ideal co-location configuration would be one which caused no IPC drop to the SPEC workload, achieved 100% of the QoS-constrained QPS that memcached could get on its own with the available resources (CQPS), and increased the server's utilization to 100%.

First, when the server is 25% utilized (3 cores running SPEC), we find that an 18-process memcached configuration, using 3 cores on one socket and all 6 cores on the other socket, achieves nearly ideal QoS-constrained QPS (-0.2% drop relative to it running on its own) while only inducing a 5% IPC drop for the SPEC workload. Thus, substantial utility can be gained from this server without disturbing its primary workload. The two 50% utilization points (6 cores), as well as the 75% (9 core) point, convey similar findings as at 25%. First, the nominal interference caused by memcached operating at 500μsec is insufficient to cause more than 10% IPC drop. In the case of 1s6c1t (SPEC occupying one socket and memcached occupying the other), even WC-IPC% is quite low. In each of these cases, the OS reported CPU utilization has been greatly increased.

We should point out that it is remarkable that memcached induces as little interference as we have observed. Flow affinity plays no small part in enabling this level of isolation, as it is dynamically steering interrupts (and consequently TCP/IP packet processing) towards the cores that are actually running memcached. Specifically, we find that no CPU time at all is spent in interrupt handing on the cores running the SPEC workloads. This is a very a positive result for flow affinity in general, and Intel's Flow Director and IXGBE driver in particular.

However, there are a couple of caveats. First, as seen in the -32.8% WC-IPC drop, a sudden spike in memcached traffic could cause a substantial slowdown to the SPEC workload. Second, when the heterogeneous SPEC workload is replaced with a SPEC workload composed of only streaming benchmarks, memcached can only handle considerably lower QPS without causing a 10% slowdown. It is evident from the 95th-% latency measurement that memcached itself is not overloaded. The real issue here is contention for the L3 cache and memory bandwidth. This contention can be mitigated in three manners: rate-limiting queries (as we've shown here), hardware resource partitioning [36, 29] (see Section 5), or holistic cluster scheduling [23]. Note that in the 50% utilization cases (2s3c1t and 1s6c1t), the best QoS-QPS from the mixed SPEC workload (fist) and the streaming SPEC workload (s) are reversed. 2s3c2t is the best configuration for memcached given the mixed workload (9% higher QoS-QPS), but 1s6c2t is the best configuration given the streaming workload (60% higher!). Thus, the optimal distribution of workloads across sockets can depend on the memory characteristics of the primary workload as well as memcached. Job placement decisions by a cluster scheduler could take this knowledge into account. Section 5 discusses these options in more depth.

Up until now, the memcached configuration has been straightforward and followed the general trends observed in Section 2.5. However, the situation becomes murky when considering a server at 100% or "101"% utilization. At N=12, where memcached is using the sibling hardware threads of the SPEC workload, and N=24, where memcached is context-switching with the SPEC workload, memcached's QPS must be severely curtailed to avoid excessive IPC drop, regardless of the type of SPEC workload (note the WC-IPC in the N=24/M=24 case). In the N=24/M=24 case, attempting to use process priorities (i.e. NICE) only makes the situation worse for memcached. We find that the best memcached performance for a given IPC drop is found by deploying it in a minimalist configuration (in both cases, with 4 processes). Such small configurations achieve marginally higher QoS-constrained QPS and suffer small worst-case IPC drops at peak QPS. Interestingly, in both cases memcached achieves approximately 9% of the baseline best QoS-QPS that this server can offer while disturbing its co-located workload less than 10%. The good news is that this performance is roughly proportional to the amount of hardware resources used. The downside to such a minimal configuration is that it is not *scalable*. If the server utilization drops from 100% to 25%, the 4-process memcached cannot automatically scale up to fill this new slack. A potential solution is to instantiate new memcached threads or processes and reassign connections in order to scale-up.

Primary Workload			**Memcached**							**%Utilization**		
N	**Conf**	**Mix**	**M**	**MConf**	**-IPC%**	**QoS-QPS**	**95th**	**-%CQPS**	**-%BQPS**	**Before**	**After**	**WC-IPC%**
3	1s3c1t	fis	18	**2s3,6c2t**	-5.0	592K	500	-0.2	-31.6	13.3	72.2	-32.8
3	1s3c1t	s	18	2s3,6c2t	-9.7	374K	338	-37.0	-56.8	12.9	76.0	-46.3
6	1s6c1t	fist	12	**1s6c2t**	-1.9	406K	494	-4.5	-53.0	26.7	81.1	-3.7
6	1s6c1t	fist	6	1s6c1t	-0.9	387K	495	-5.3	-55.2	26.7	59.0	-1.2
6	1s6c1t	s	12	1s6c2t	-10.0	278K	291	-34.6	-67.8	27.1	80.5	-26.0
6	2s3c1t	fist	12	**2s3c2t**	-6.7	443K	479	-0.4	-48.7	29.2	76.6	-23.9
6	2s3c1t	fist	6	2s3c1t	-3.8	343K	396	-16.8	-60.3	29.2	56.8	-7.5
6	2s3c1t	s	12	2s3c2t	-9.2	173K	296	-61.2	-80.0	27.2	40.2	-25.8
9	2s36c1t	fist	6	**1s3c2t**	-3.4	154K	493	-40.0	-82.2	42.7	57.3	-1.0
9	2s36c1t	s	6	1s3c2t	-8.7	127K	261	-50.6	-85.3	42.8	68.7	-14.8
12	2s6c1t	fist	4	**2s2c1t**	-7.1	81K	320	-69.6	-90.6	50.5	60.7	-20.0
12	2s6c1t	fist	12	2s6c1t	-8.0	59K	170	-91.8	-93.2	50.0	60.6	-38.2
24	2s6c2t	fist	4	**2s1c2t**	-3.2	77K	479	-52.2	-91.1	100.0	99.4	-13.7
24	2s6c2t	fist	24	2s6c2t	-7.5	24K	158	-97.2	-97.2	100.0	99.8	-95.6

Table 6: Results of co-location experiments in the "Google" scenario. N is the number of SPEC benchmarks occupying the server, chosen from the types of benchmarks given by "Mix". "Conf" is the configuration used for the SPEC workloads. M is in the number of memcached processes packed with the SPEC workloads, and "MConf" specifies their configuration. The best memcached configuration in each scenario is highlighted. Percent drop in IPC is relative to the SPEC workload running alone in the given configuration. QPS and 95th are the achieved QoS-constrained QPS and measured 95th-% latency in this scenario. "-%CQPS" is the percent slowdown of QPS relative to M-process memcached running in this configuration on its own, and "-%BQPS" is the percent slowdown relative to 24-process memcached running alone, which is the fastest configuration for this server. "Utilization" is average utilization as reported by the Linux in `/proc/stat`, and is reported for the SPEC workload alone ("Before") and for when it is co-located with memcached ("After"). "WC-IPC%" is the "worst-case" IPC drop that SPEC experiences when memcached is run at maximum QPS (i.e. fully loaded with no QoS constraint). Gray rows show optimal configurations for a given set of resources, and other rows highlight performance regressions.

4.3 Summary

Overall, our results show that there is huge opportunity to make use of underutilized servers, even when co-locating work with low-latency services. To summarize the lessons learned, the following is a list of *DO*s and *DON'T*s for deploying memcached.

DO use multiple memcached processes to scale memcached performance until scalability bottlenecks in the multi-threaded memcached are addressed. (Sec. 2.4)

DON'T deploy memcached in asymmetric configurations where some threads or processes will have higher performance than others. The slowest ones will determine the 95th-% latency. (Sec. 2.4)

DO use all hardware threads of cores available to memcached. A marginal increase in QoS-constrained performance is observed. (Sec. 2.5, 4.2)

DO set memcached's scheduling priority very high. As soon as memcached is forced to enter OS-mediated time-sharing with another process, it no longer provides acceptable quality of service, even if it can service queries at the requested rate. (Sec. 4.1)

DO rate-limit requests to memcached when interference for other workloads is intolerable. This rate-limiting could be internal to memcached (i.e. sleep when QPS is too high) or external (i.e. rate-limit at the clients, in the networking stack, or at the OS scheduler). The need for such rate-limiting would likely be reduced if this server had effective cache or memory bandwidth partitioning features (Sec. 4.2). See Section 5 for discussion of future work.

DO consider the memory characteristics of a server's primary workload when deciding whether or not to deploy memcached across sockets [23]. In the absence of last-level cache pressure, it is best to spread memcached across sockets. In the opposite case, it is best to consolidate memcached on sockets. (Sec. 4.2)

DO use flow affinity. Steering interrupts to the core actually responsible for the traffic protects both memcached and other workloads from interference due to network traffic. (Sec. 4.2)

DO deploy memcached in a minimalist configuration if you want a small, but guaranteed, QoS-constrained QPS on servers heavily loaded with other, more important work. (Sec. 4.2)

5. OPPORTUNITIES & LIMITATIONS

It should be clear from our results that there is no fundamental reason that services with tight QoS constraints cannot be co-located with other load. There are, naturally, several opportunities for future work (i.e. technical hurdles), and none seem insurmountable.

First, last-level cache and memory bandwidth contention are clear obstacles to co-locating memcached with some workloads. We demonstrated how reducing the QPS rate of memcached avoids excessive interference for these workloads, but obviously hardware cache partitioning is another avenue to explore. Our results in Section 3.2.2 are indicative of how memcached's performance scales as a function of L3 cache capacity. Hardware cache [36, 34] and memory bandwidth partitioning [29, 28] are clearly applicable here. As with any resource partitioning, the hardware must provide the partitioning mechanism, whereas the software can guide the policy.

In the absence of hardware partitioning, software-enforced rate-limiting could be an effective alternative to manage interference. This rate-limiting could be implemented in a number of ways, including internal changes to memcached (delay or reject requests), memcached clients (feedback directed flow-control), in the network stack (metering or dropping when packet rates become excessive).

This rate-limiting could even be accomplished by the Linux process scheduler. In particular, the CFS Bandwidth Control support added to Linux 3.2 in January 2012 (submitted by engineers from Google, coincidentally [3]), could be useful here. With scheduler bandwidth control, processes with low-latency requirements could be run with very high scheduling priorities (to avoid time-sharing in the short-term) while still granting long-term priority to other workloads by descheduling memcached if it has been too busy over a long period of time. Such a strategy trades-off minor, but frequent, QoS violations for major infrequent ones.

Second, flow affinity contributes mightily to the management of interference. However, in its current implementation in the IXGBE driver, it only applies for TCP traffic. It is worth investigating strategies for steering UDP traffic. Moreover, we do not yet understand the full opportunity presented by multi-queue NICs with configurable packet steering. Further isolation of extreme network traffic scenarios may be possible. In particular, it appears likely that multiple low-latency services can be co-located without loss of QoS, so long as their network traffic is steered to distinct cores. For services with latency requirements well under 500μsec, other NIC issues (i.e. jumbo frames, interrupt coalescing, etc.) will become more important.

Third, as queueing delay is the biggest contributor to memcached latency under heavy load, it makes sense to address this problem directly. For instance, several categories of memcached requests do not have low-latency requirements: set requests, get requests for large values, requests off the critical path for rendering end-user pages, etc. It makes sense to de-prioritize these categories of traffic to reduce their influence on queueing delay. `libevent` has some facilities for specifying socket priority, but it is not used by memcached, and the issue presents itself more precisely at the request level than the socket level.

Fourth, we have observed severe performance regressions due to the Linux scheduler when we do not pin memcached processes to distinct cores. It frequently migrates the processes to run on the same hardware thread and never migrates them back, most likely because the thread isn't running at 100% utilization. Though not noticeable on human timescales, it has significant impact on latency at microsecond timescales due to time-sharing. This problem should be addressed in the Linux kernel, and may benefit a wide variety of latency-sensitive tasks.

Finally, we've shown how to co-locate memcached with other workloads, achieving significant performance (from 77K QPS on a fully loaded machine to 592K QPS on a lightly loaded machine) with good quality of service. The downside is that such a service deployed across a cluster would have heterogeneous performance characteristics: the performance of each server would depend on what other work running on that server. This problem is similar to the challenge of dealing with heterogeneous mixes of servers (e.g., Xeons vs Atoms). A solution to managing heterogeneous datacenter resources greatly complement the value of our results.

Limitations

There are certain limitations to this study. We do not study the UDP version of the memcached protocol, and this has implications for the effectiveness of flow affinity. On the other hand, the UDP protocol seems to be in little use outside of Facebook, and research suggests that affinity for UDP is feasible [38]. Likewise we do not consider memcached's alternative binary protocol, but using this different protocol would only reduce the memcached-portion of latency in processing requests.

We do not study the co-location of several latency sensitive tasks, and instead only consider co-locating one latency sensitive task (memcached) with throughput oriented tasks. This is certainly a topic for future study. Finally, we considered one latency-critical workload at one quality of service constraint (95th-% below 500μsec). While we expect that many of the lessons from this work would be directly applicable, one could expect other services (like REDIS, HBase, etc.) to have some idiosyncrasies. Furthermore, it is interesting to investigate if it is possible to co-locate ultra low-latency services such as RAMCloud [30].

6. RELATED WORK

Barroso and Hölzle alerted the community to the total cost of ownership consequences of underutilized servers [8], and there has been an outpouring of research into improving the energy-efficiency of underutilized servers [25, 37, 26]. Rather than addressing the problem of energy-proportionality, this paper resorts to "first principles" and addresses under-utilization head-on by considering the quality of service concerns that lead to it.

Meisner et al. provided a detailed analysis of the impact power-saving modes have on performance and latency [26]. Our work differs in two key manners: we are studying latency in the presence of *interference*, not power saving goals, and we study a quality of service target an order of magnitude smaller than what they consider. In any event, our goals are the same: to reduce the total cost of ownership for large datacenters.

Considerable attention has been given to the challenges of hardware resource partitioning on CMPs, both for caches [36, 34] and memory channels [28, 29]. We believe our study strengthens the case for these techniques as our work shows how (1) memory hierarchy contention can result in major quality of service issues, even for servers that are apparently underutilized, and (2) we show how interference contributes to queueing delays on the order of several milliseconds. Distinct from past work, we evaluate our workloads on a metric (QoS-QPS) that is materially different from IPC.

Recent work by Mars and Tang [23, 22] addresses the challenge of finding suitable jobs to co-locate on DC servers. Mars' work could be applied to guide the memcached configuration decisions studied in Section 4.2. Unlike our work, these studies did not pay any particular attention to network isolation, interrupt handling, or OS scheduling issues. Moreover, these previous works focus on throughput metrics, whereas we focus on latency metrics.

Diwaker studied performance isolation for virtual machines and addressed issues concerning device drivers [16]. There has also been considerable work on fair scheduling of OS processes [31, 13] and groups of processes [6]. All of these works operate at the time-scale of time-slices and do not address the causes of QoS violations for applications with tighter latency constraints.

Berezecki et al. evaluated memcached on a 64-core Tilera TILEPro64 system [9]. Similar to our paper, they study memcached QPS given a tight latency bound (1ms). They find that the Tilera system offers better performance and power efficiency than a commodity server platform. The high performance they achieve on the Tilera is partly due to rearchitecting memcached to scale to 64-cores. Inspired by their findings, we deploy memcached in a multi-process configuration and achieve performance *several times* faster than they did on a comparable Xeon platform or a single-node TILEPro64.

Managing the interrupt, flow, and core affinity of multi-queue network interfaces has received increased attention recently. Foong details how affinity affects performance of the TCP/IP stack in Linux [14]. Pesterev et al. propose a new OS abstraction called Affinity-Accept to arrange for and manage TCP flow affinity using Intel 82599 NICs, and discuss the design and drawbacks to Intel's Flow Director [32]. Wu et al. address the problem of Intel's Flow Director causing packet reordering. While all of these works describe how affinity greatly improves the efficiency of TCP/IP processesing, none addresses the potential that flow-affinity has for performance isolation on shared datacenter servers.

7. CONCLUSIONS

We studied the apparent conflict between two important challenges for warehouse-scale datacenters: the need for QoS guarantees for latency-critical workloads and the desire to improve resource efficiency by increasing average server utilization. Starting with a carefully optimized baseline for a latency sensitive workload (memcached), we quantified how load relates to latency QoS violations, how software and hardware sources of interference affect performance and latency guarantees, and how existing scheduling and isolation mechanism can mitigate these problems. We used this information to target two scenarios of underutilized servers. First, we showed how to place additional work on dedicated memcached servers, raising their utilization to nearly 100% without impacting memcached throughput or latency QoS. Second, we demonstrated how to place a memcached service on a server dedicated to other workloads that has underutilized memory resources. We can get significant throughput for memcached with good QoS guarantees without impacting the performance of the original workload. While there are several further opportunities for additional hardware and software mechanisms for isolation, we find that it is already possible to co-locate work with memcached servers without harming quality of service. Given the common event-based design pattern that memcached follows, its likely that these results extend to several other latency-sensitive services as well.

8. REFERENCES

[1] Specmail2009. `http://www.spec.org/mail2009/`.

[2] memcached. `http://memcached.org/`, July 2012.

[3] [rfc] cpu hard limits. `https://lkml.org/lkml/2009/6/4/24`, July 2012.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. SIGMETRICS, 2012.

[5] S. Avallone, S. Guadagno, D. Emma, A. Pescap, and G. Ventre. D-ITG: Distributed internet traffic generator. *Intl. Conf on Quantitative Evaluation of Systems*, 2004.

[6] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. OSDI, 1999.

[7] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *Proc. of the 38th Intl. symposium on Computer architecture*, ISCA '11, 2011.

[8] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec. 2007.

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. *International Green Computing Conference and Workshops*, 0:1–8, 2011.

[10] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proc. of the 9th USENIX Conf. on OSDI*, 2010.

[11] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. EW 10, 2002.

[12] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen. Architecture of systemtap: a linux trace/probe tool, 2005.

[13] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techn.*, PACT '07.

[14] A. Foong, J. Fung, D. Newell, S. Abraham, P. Irelan, and A. Lopez-Estrada. Architectural characterization of processor affinity in network processing. In *IEEE Intl. Sym. on Performance Analysis of Systems and Software*, 2005.

[15] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *In Proceedings of 6th Annual Linux Symposium*, 2004.

[16] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. Middleware '06, 2006.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[18] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[19] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. of the 17th Intl. Conf. on Parallel architectures and compilation techniques*, 2008.

[20] D. Karger et al. Web caching with consistent hashing. WWW '99, 1999.

[21] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30(4):8–19, July 2010.

[22] J. Mars, L. Tang, and R. Hundt. Heterogeneity in "homogeneous"; warehouse-scale computers: A performance opportunity. *IEEE Comput. Archit. Lett.*, 10(2):29–32, July 2011.

[23] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. of The 44th IEEE/ACM Intl. Symp. on Microarchitecture*, 2011.

[24] M. Mayer. What google knows. In *Web 2.0 Summit*, November 2006.

[25] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of the 14th international ASPLOS*, ASPLOS '09, 2009.

[26] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. *SIGARCH Comput. Archit. News*, 39(3).

[27] A. Michael, H. Li, and P. Sart. Open compute project. HotChips 23 Tutorial, August 2011.

[28] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. of the 40th IEEE/ACM Intl. Symposium on Microarchitecture*, 2007.

[29] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proc. of the 39th IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 39, 2006.

[30] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[31] C. S. Pabla. Completely fair scheduler. *Linux J.*, 2009(184).

[32] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, 2012.

[33] N. Provos. libevent. `http://libevent.org/`.

[34] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, 2006.

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

[36] D. Sanchez and C. Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. *IEEE Micro's Top Picks from the Computer Architecture Conferences*, 32(3), May-June 2012.

[37] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering energy proportionality with non energy-proportional systems: optimizing the ensemble. In *Proceedings of the 2008 conference on Power aware computing and systems*, 2008.

[38] W. Wu, P. DeMar, and M. Crawford. A transport-friendly nic for multicore/multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 23:607–615, 2012.