Filling the Pipe: A Guide to Optimising Memcache Performance on SolarFlare® Hardware

Solarflare Communications Inc.

Technical Report Number: SF-110694-TC

1st July 2013

Contents

1 Introduction 2

2 A Primer on Modern x86 Architecture 2
2.1 Packages, Cores, Dies and Hyperthreads . . . 2
2.2 Non-Uniform Memory Access . . . 2
2.3 Network Interrupt Management . . . 2

3 Optimising Memcache Performance: A Quickstart Guide 3
3.1 Optimization Principles . . . 3
3.2 Optimising Memcache Server Performance . . . 3
3.3 Optimising Memcache Client Performance . . . 7
3.4 Benchmark Client-Server Performance . . . 7

4 Maximising Memcache Performance 8
4.1 Use The Optimum Server Version . . . 8
4.2 Optimize Server-to-NUMA Node Mapping . . . 9
4.3 Consider Multiserver Configurations . . . 9
4.4 Optimize The Interrupt Service Configuration . . . 9
4.4.1 Determine The Optimum Number of Interrupt Service CPUs . . . 9
4.4.2 Localize Interrupt Service CPUs . . . 10
4.4.3 Shield Interrupt Service CPUs . . . 10
4.5 Localize Memory Allocation . . . 11
4.6 Client Side Optimizations . . . 11
4.6.1 Multiget . . . 12
4.6.2 UDP Get . . . 12
4.6.3 OpenOnload - Stack Per Thread . . . 12
4.7 Consider Hyperthreading . . . 12
4.8 Further Exploration . . . 13

5 Conclusion 13

A Hardware Specification & Software Versions 14
A.1 Hardware Specification . . . 14
A.2 Software Versions . . . 14

1 Introduction

This document outlines the general principles and specific steps that need to be carried out in order to optimise Memcache performance on Solarflare hardware in Linux environments. While this document focuses on Memcache the principles and mechanisms outlined should be useful for any application dominated by request-response style traffic.

2 A Primer on Modern x86 Architecture

To effectively tune Memcache performance it is useful to have a synoptic understanding of the physical and logical structure of modern x86 architecture in regards to network communication. This section outlines the basic hardware properties affecting Memcache performance. Readers familiar with the structure of modern servers can skip to Section 3.

2.1 Packages, Cores, Dies and Hyperthreads

Modern CPUs are shipped in physical units called *packages* – the piece of metal with pins that plugs into a socket on the motherboard. Packages are composed of multiple *cores* each of which is an independent execution unit that runs programs. A package is usually composed of one or more dies. Dies are the physical silicon on which the transistors composing cores reside. The advantage of clustering cores on a die is that they can share common resources, e.g. the memory controller or Last Level Cache. Modern packages are usually comprised of a single die.

Hyperthreading leverages the superscalar nature of modern processors capable of executing multiple instructions operating on separate data in parallel by making a core appear as multiple (usually 2) logical processors. As hyperthreading is transparent from the software perspective it is an efficient mechanism for increasing parallelism. Its effectiveness is, however, dependent on the complementariness of programs being run on related logical processors.

2.2 Non-Uniform Memory Access

As illustrated in Figure 1, every package contains a local memory controller accessible to all cores. The combination of a package and its attached memory is commonly referred to as a *Non-Uniform Memory Access (NUMA) node*. Packages are interconnected via a specialised (QuickPath or HyperTransport) CPU interconnect.

While memory is globally addressable (i.e. every processor can access all memory in the system) it is slower to access non-local (remote) memory as this requires traversal over the CPU interconnect. It thus stands to reason that memory should be physically equally distributed across all packages in the system to ensure optimum application performance.

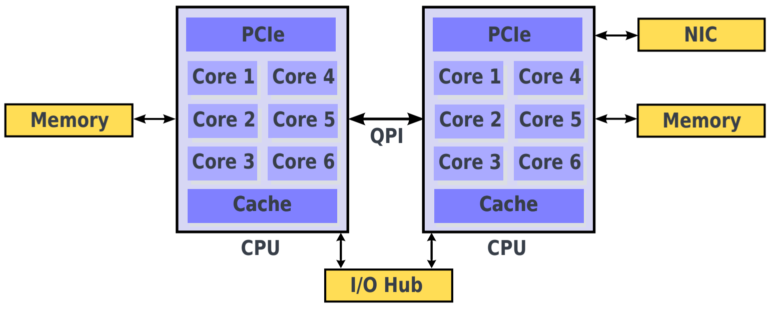

Figure 1: Block diagram of typical modern NUMA system
This diagram, also represents the structure of the example system used in this report (Appendix A)

2.3 Network Interrupt Management

NICs employ interrupts in two dimensions: Upon receiving data from the network the NIC informs the host via a *receive interrupt* causing it to carry out basic data demultiplexing functionality. Similarly, the NIC sends the host a *transmit interrupt* when data queued for transmission has been dispatched.

Servicing interrupts can be CPU intensive, especially for traffic comprised of small, frequent exchanges. Slow CPUs servicing interrupts suffer from livelock in which they are unable to make progress on application work due to heavy interrupt load. Even with adequate CPU, servicing interrupts can lead to decreased application performance due to (data) cache pollution.

To mitigate the CPU availability problem modern NICs can spread interrupts over multiple CPUs. Solving the cache pollution problem, however, is less straightforward and is commonly addressed by dedicating a number of CPUs to just servicing interrupts.

3 Optimising Memcache Performance: A Quickstart Guide

This section provides a step-by-step guide for configuring your server for optimum Memcache performance with Solarflare equipment. Readers interested in additional tuning options are advised to consult Section 4.

3.1 Optimization Principles

Memcache is a multithreaded application capable of scaling up performance by increasing the number of worker threads. Based on the properties highlighted in Section 2 it follows that optimum performance is dependent on ensuring (*i*) an adequate number of application threads are servicing client requests, (*ii*) application threads don't access memory across NUMA nodes, (*iii*) there are an adequate number of CPUs to service network interrupts and (*iv*) CPU (data) cache pollution is minimised.

Building on these principles, we advocate a configuration in which a separate instance of Memcache is run on every NUMA node on the server and clients are configured to use them all (easily achieved given the first step in storing or accessing data is hashing the key across available servers).

3.2 Optimising Memcache Server Performance

This section outlines the commands needed to optimise Memcache 1.4.15 server performance on our example system – a dual socket 12 core E5-2620 machine (Figure 1) with 32GB RAM running Redhat Enterprise Linux 6.2. Please adapt them depending on your setup. Readers interested in a detailed description of the example hardware specification and software versions used in this section are advised to consult Appendix A.

The steps required to maximise server performance are as follows:

1. **Disable all non-essential services.**

   Specifically, be sure to disable `irqbalance`.

```
[root@mc-svr ~]# for s in acpid cgconfig irqbalance atd auditd \
                    avahi-daemon cpuspeed crond haldaemon \
                    iscsid kdump ksm ksmtuned libvirt-guests \
                    libvirtd lvm2-monitor mdmonitor microcode_ctl \
                    openct pcscd portreserve sysstat
                 do
                  /etc/init.d/$s stop
                  /sbin/chkconfig --del $s
                 done
```

2. **Disable Hyperthreading**

```
[root@mc-svr ~]# for c in \
               $(cat /sys/devices/system/cpu/cpu*/\
                 topology/thread_siblings_list \
                 | cut -f2 -d ',' | sort -n | uniq)
    do
        echo 0 > /sys/devices/system/cpu/cpu$c/online
    done
```

3. **Disable Swapping and Turn On Hugepage Support**

```
#turn off swapping
[root@mc-svr ~]# swapoff -a

#non-essential.  OK if it fails.
[root@mc-svr ~]# echo 128 > /proc/sys/vm/nr_hugepages
```

4. **Determine The Number of NUMA nodes and the CPU-to-Numa Node Mapping**

```
[root@mc-svr ~]# numactl --hardware | grep node | grep cpus
node 0 cpus: 0 1 2 3 4 5
node 1 cpus: 6 7 8 9 10 11
```

   Based on these results we would run two instances of Memcache on this machine, with the threads for instance one spread over Numa Node 0 (CPUs 0–5) and the second Numa Node 1 (CPUs 6–11).

5. **Create A System Tasks Shield**

   Shields are used to restrict processes to preselected CPU and Memory nodes. They are useful for isolating CPUs to prevent arbitrary processes running on them resulting in a reduction in the amount of available CPU time and increased (data) cache pollution. Shields are implemented via the `cpusets(7)` mechanism and may be defined, deleted and modified with the `cset` command.

   We recommend constraining all boot and non-essential programs to a shield on CPU 0 thereby preventing arbitrary process execution on other CPUs. This can be achieved as follows:

   (a) Create A Shield Limited to CPU 0 for System Tasks

```
[root@mc-svr ~]# cset set --cpu=0 system-tasks
cset: --> created cpuset "system-tasks"
```

   (b) Move All Existing Programs Into The System Task Shield

```
[root@mc-svr ~]# cset proc --move / system-tasks --kthread
cset: moving all tasks from root to /system-tasks
cset: moving 54 userspace tasks to /system-tasks
cset: moving 0 kernel threads to: /system-tasks
[==================================================]%
cset: done
```

   At this point all existing programs and Kernel threads will always run on CPU 0, effectively shielding CPUs 1–11.

6. **Configure Interrupt Servicing CPUs**

   As explained in Section 2.3, servicing interrupts reduces application CPU time and results in cache pollution. For this reason, Solarflare recommends interrupt servicing should be carried out by a small number of CPUs dedicated to the task. The number of CPUs required for interrupt servicing is dependent on the network stack. OpenOnload does not heavily rely on interrupts. Users deploying it are not required to carry out any further configuration in this step and may proceed directly to the next step.

   Users deploying the Kernel networking stack are recommended to dedicate at least 4[a] CPUs per 10G port, evenly distributed across CPUs in all nodes.

   Dedicating CPUs to interrupt servicing consists of the following steps:

   (a) Configure the Solarflare driver to spread interrupts over 4 CPUs

```
[root@mc-svr ~]# echo "options sfc rss_cpus=4" >> /etc/modprobe.d/sfc.conf
```

   (b) Reload the Solarflare driver to initiate the effect

```
[root@mc-svr ~]# modprobe -r sfc && modprobe sfc
```

[a]The exact number for a particular configuration is empirically determinable as explained in Section 4.4.1

(c) Reconfigure networking for the Solarflare interface

```
[root@mc-svr ~]# ifconfig eth2 <ip-address>
```

Alternatively, you can restart the networking service (`/etc/init.d/network restart`).

(d) Determine the interrupt lines claimed by the Solarflare NIC

```
[root@mc-svr ~]# cat /proc/interrupts | grep eth2 | cut -f1 -d':'
 128
 129
 130
 131
```

As expected, 4 interrupt lines are claimed to enable the NIC to spread interrupts over 4 cores.

(e) Assign the NIC-claimed interrupt lines to the appropriate CPUs.
The 4 interrupt lines are evenly distributed across CPUs in NUMA nodes 0 (CPUs 0,1) and 1 (CPUs 6,7) respectively.

```
[root@mc-svr ~]# echo 0 > /proc/irq/128/smp_affinity_list
[root@mc-svr ~]# echo 1 > /proc/irq/129/smp_affinity_list
[root@mc-svr ~]# echo 6 > /proc/irq/130/smp_affinity_list
[root@mc-svr ~]# echo 7 > /proc/irq/131/smp_affinity_list
```

Note that CPU 0 is being reused to service interrupts *and* run all existing programs and kernel threads. This is acceptable as we do not expect the CPU load of existing processes to be high. Doing so allows us to utilise the CPU more efficiently.

(f) Create An Interrupt Service Shield
Creating an optional "interrupts" shield is useful for creating a simple administrative record of interrupt servicing CPUs.

```
[root@mc-svr ~]# cset set --cpu=0,1,6,7 interrupts
cset: --> created cpuset "interrupts"
```

7. **Optimize Interrupt Delivery**

Turning off adaptive receive interrupt delivery reduces latency and improves performance:

```
[root@mc-svr ~]# ethtool -C eth2 adaptive-rx off
```

For users deploying the Kernel stack, turning off receive interrupt coalescing further improves performance:

```
[root@mc-svr ~]# ethtool -C eth2 rx-usecs 0
```

8. **Create Memcache Application Shields**

Shields are also useful for segregating Memcache instances within and across NUMA nodes. Solarflare recommends creating a shield for every separate instance of Memcache. The CPU and Memory nodes included in every shield depend on how the resources of the system are allocated to instances. In our example server we aim for two Memcache instances, one on each NUMA node. This is achieved as follows:

```
[root@mc-svr ~]# cset set --cpu=2-5 --mem=0 mc-node0
cset: --> created cpuset "mc-node0"
[root@mc-svr ~]# cset set --cpu=8-11 --mem=1 mc-node1
cset: --> created cpuset "mc-node1"
```

The following points are of note:

(a) By restricting Shield `mc-node0` to CPUs 2–5 and Memory Node 0 (all in NUMA Node 0) we guarantee processes running in the shield will always only access hardware resources on NUMA Node 0. Similarly, by restricting `mc-node1` to CPUs 8–11 and Memory Node 1 we guarantee processes running in the `mc-node1` shield will always only access resources on NUMA Node 1.

(b) We restrict `mc-node0` to CPUs 2–5 because CPUs 0,1 are allocated to system tasks and interrupts. Similarly, `mc-node1` is restricted to CPUs 8–11 because CPUs 6,7 are allocated to interrupts. As OpenOnload is not reliant on interrupts (Step 6), users deploying it may reassign the interrupt CPUs to the respective Memcache shields.

(c) It is possible to create multiple Memcache shields on the same NUMA Node and it may be useful to do so if multiple servers are required for performance reasons.

9. **Export OpenOnload Environment Variables**

   Users deploying OpenOnload should set the following environment variables:

```
export EF_POLL_USEC=1000
export EF_FDS_MT_SAFE=0
export EF_EPOLL_SPIN=1
export EF_POLL_SPIN=1
export EF_SELECT_SPIN=1
export EF_STACK_PER_THREAD=1
```

   Users who want to debug/verify connections are accelerated should set the `EF_NO_FAIL` environment variable (this option disallows conventional unaccelerated sockets).

```
 export EF_NO_FAIL=0
```

10. **Start Memcache Server Instances In Shields**

   The final step involves starting Memcache server instances in the Memcache Shields created in Step 8 via `cset`:

   Start instances in `mc-node0` and `mc-node1` respectively:

```
[root@mc-svr ~]# cset proc --set=mc-node0 --exec -- \
/home/rs/mp/tmp/memcached-1.4.15/memcached -m 4096 -t 4 -p 11212 -u rs &

[root@mc-svr ~]# cset proc --set=mc-node1 --exec -- \
/home/rs/mp/tmp/memcached-1.4.15/memcached -m 4096 -t 4 -p 11213 -u rs &
```

   The following points are of note:

   (a) As recommended for our example server configuration, we create one Memcache instance per Memcache Application Shields (Step 8). The `cset` tool is used to start the instances in the correct shields. The command `cset proc --set=<set-name> --exec -- <cmd>` executes the command `<cmd>` in the set `<set-name>`.

   (b) The following command-line options passed to Memcache are noteworthy:

      i. `-m <size>`: Sets memory cache size. In our example configuration every instance has a 4GB Memory cache.

      ii. `-t <nr-of-threads>`: Indicates the number of Memcache worker threads to start. While the optimum number of worker threads is empirically determinable (see Section 4.1) and is dependent on hardware, on our example server we use 3 when running on the OpenOnload network stack (for a total of 6 threads across both servers) and 4 when running on the Kernel stack (for a total of 8 threads across both servers).

      iii. `-p <port-num>`: Indicates the TCP listening port number of the server. Note that every server instance requires a separate listening port number.

      iv. `-u <username>`: Indicates which identity should be assumed when the server is started. This option is only necessary if you start the Memcache server instances as `root` and must identify an existing user on the system.

      v. `-k`: Indicates all Memcache memory should be locked down to prevent it being swapped. This option is only necessary if you have not turned off swapping as recommended in Step 3.

   (c) If running on OpenOnload, remember to prepend the `memcached` command with the `onload` script (i.e. `cset proc ... --exec -- onload /home/rs/mp/tmp/memcached-1.4.15/memcached ...`).

11. **Pin Worker Threads**

While it is not strictly necessary, it may be additionally beneficial to pin every Memcache worker thread onto a single CPU to further improve performance by minimising cache pollution.

To correctly carry out worker thread pinning it is necessary to understand how threads are spawned in Memcache. A Memcache instance started with `n` threads will spawn `n + 1` threads of which the first `n` are worker threads and the last is a maintenance thread used for hash table expansion under high load factor.

To pin worker threads it is necessary to determine their thread ID's (TIDs). To list only all worker thread IDs for a particular instance of Memcache use the command pipe:

```
ps -p <memcache-process-id> -o tid= -L | sort -n | tail -n +2 | head -n -1
```

Where `<memcache-process-id>` is the Process Identifier (PID) of the Memcache instance.

For example, substituting the PID of the Memcache instance started in `mc-node0` earlier in the step, worker thread TIDs are obtained as follows:

```
[root@mc-svr ~]# ps -p 23443 -o tid= -L | sort -n | tail -n +2 | head -n -1
23444
23445
23446
23447
```

TIDs are pinned to single CPUs via the `taskset -pc <cpu-id> <tid>` command where `<cpu-id>` identifies the target CPU and `<tid>` identifies the TID. For example, to pin the worker threads of the Memcache instance running in the `mc-node0` shield to CPUs 2–5 respectively, run the following series of commands:

```
[root@mc-svr ~]# taskset -pc 2 23444
pid 23444's current affinity list: 2-5
pid 23444's new affinity list: 2
[root@mc-svr ~]# taskset -pc 3 23445
pid 23445's current affinity list: 2-5
pid 23445's new affinity list: 3
[root@mc-svr ~]# taskset -pc 4 23446
pid 23446's current affinity list: 2-5
pid 23446's new affinity list: 4
[root@mc-svr ~]# taskset -pc 5 23447
pid 23447's current affinity list: 2-5
pid 23447's new affinity list: 5
```

By carrying out a similar set of steps on the Memcache instance running in the `mc-node1` shield, we can obtain and pin its worker threads to CPUs 8–11 respectively.

3.3 Optimising Memcache Client Performance

While the exact configuration of the client is dependent on the client application, we advocate adhering to the optimisation principles outlined in Section 3.1. Broadly replicating the steps provided in Section 3.2 with the slight modification of running multiple client applications in segregated NUMA-local shields should be sufficient to guarantee optimum performance. Finally, if running on OpenOnload, remember to prepend client binaries with the `onload` script.

3.4 Benchmark Client-Server Performance

We recommend benchmarking server performance via the `memslap` benchmark described in Appendix A.

In particular, we benchmark performance via the following command:

```
[root@mc-cnt ~]# /home/rs/mp/build/bin/memslap -s mc-svr:11212,mc-svr:11213 \
-T 8 -c 256 -B -t 30s -S 1s
```

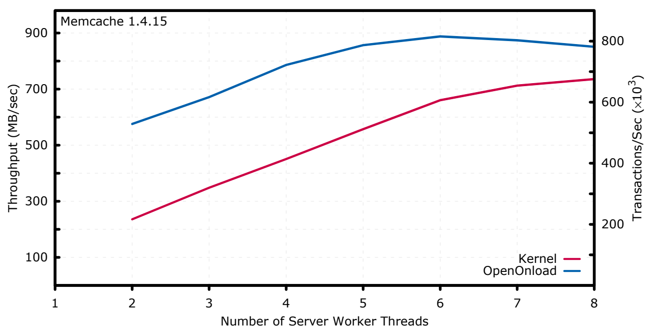

Figure 2: Performance of 2 Server Instances (1 Per NUMA Node)

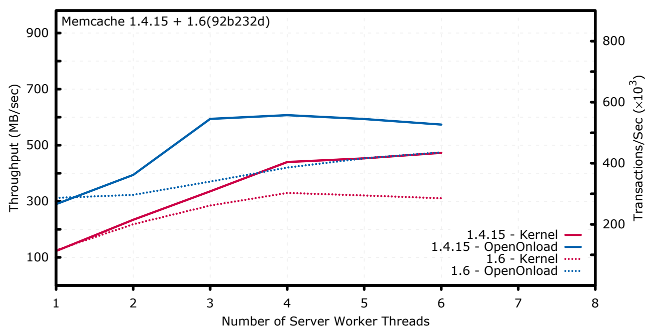

Figure 3: Performance of Memcache 1.4.15 vs 1.6 (1.6 Git Commit ID:92b232d)

The following command-line options are noteworthy:

1. `-s`: Lists the servers to use. Based on our example configuration we are connecting to two separate Memcache instances running on ports `11212` and `11213` of our example server (`mc-svr`).
2. `-T 8`: Instructs the benchmark to spawn 8 client threads.
3. `-c 256`: Instructs the benchmark to keep 256 concurrent connections open per-thread.
4. `-B`: Instructs the benchmark to use the binary Memcache protocol
5. `-t 30s`: Instructs the benchmark to run for 30 seconds.
6. `-S 1s`: Instructs the benchmark to output statistics every 1 second.

Figure 2 illustrates Memcache performance for our example server with varying numbers of server threads based on the server configuration explained in Section 3.2 (2 Memcache instances, one per-NUMA node). As illustrated, OpenOnload has much higher performance than the Kernel stack, peaking at 6 worker threads (3 per server-instance) before performance begins to drop. Conversely, Kernel performance increases linearly for up to 8 worker threads. However, it is always slower than OpenOnload.

4 Maximising Memcache Performance

This section quantifies the major hardware and software issues that may influence Memcache performance in production systems. It is useful for readers looking to micro-optimise server performance and debug performance problems.

4.1 Use The Optimum Server Version

Our experience with the Memcache codebase indicates that performance can significantly improve *or* reduce between successive versions iterations. Given that the two supported branches (1.4 and 1.6), are continually being modified and that there are a number of variants of the core codebase, we recommend empirically comparing all potential options when making a deployment decision.

Even in the standard 1.4 and 1.6 branches performance can vary significantly depending on configuration. Figure 3 compares single server performance of Memcache 1.4.15 against Memcache 1.6 (HEAD: 92b232d) as of 01.05.2013 running on a single NUMA node. As the results show, 1.4.15 performs significantly better (peaking at 600 MB/sec) than Memcache 1.6 (peaking at 480 MB/sec). Furthermore, the 1.4.15 results show, running on OpenOnload, it is not beneficial to allocate more than 3 worker threads to an instance as the performance is constant. However, 1.6 has slightly better single threaded performance with OpenOnload.

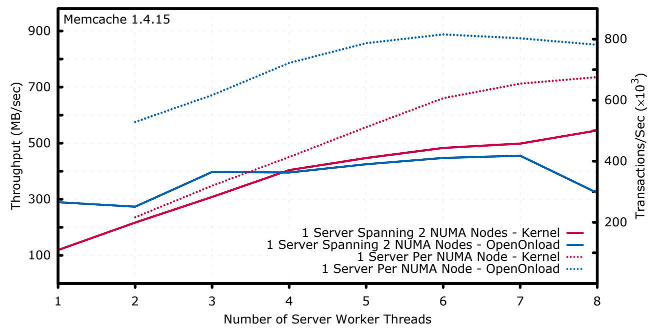

Figure 4: Performance of 1 Server Spanning 2 NUMA Nodes vs 1 Server Per NUMA Node

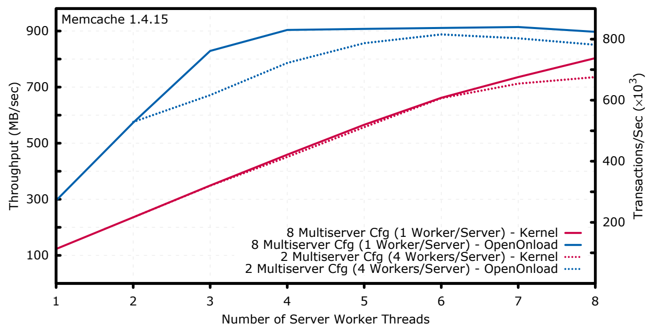

Figure 5: Performance of 8 Multiserver vs 2 Multiserver Configuration (8 Worker Threads Total)

4.2 Optimize Server-to-NUMA Node Mapping

As outlined in Section 3.2, Solarflare recommends running a separate Memcache instance on every NUMA node. The performance benefits of this configuration are highlighted in Figure 4. Both the Kernel and OpenOnload show an increase in performance with an increasing number of worker threads when separate instances are executed on every NUMA node.

In contrast, a single server spanning two NUMA nodes performs poorly even with a small number of worker threads; comparing Figure 3 and Figure 4 we notice that, in the OpenOnload case, when the server spans two NUMA nodes performance flatlines at 400MB/sec with 3 worker threads as opposed to 600MB/sec in the single node case.

4.3 Consider Multiserver Configurations

At the cost of increased (client and server) setup and configuration complexity, it is possible to increase performance even further than for the configuration outlined in Section 3.2. By running multiple servers, (Multiservers), each with a single worker thread we eliminate internal lock contention completely. An example of this principle is provided in Figure 5 where up to 8 multiple servers are started, each comprised of a single worker thread.

As the figure shows, in this configuration OpenOnload exhibits a steady rate of performance improvement with increasing number of servers for up to 3 servers and is able to saturate the link with 4 CPUs. Similarly, the Kernel is able to maintain a steady rate of performance improvement with increasing numbers of CPUs for up to 8 servers compared to the 2 server case (in which the rate of performance improvement begins to decrease after 6 worker threads).

4.4 Optimize The Interrupt Service Configuration

4.4.1 Determine The Optimum Number of Interrupt Service CPUs

As explained in Section 2.3, performance can be limited by the number of CPUs servicing interrupts. However, dedicating CPUs purely to servicing interrupts is expensive as the CPU is rendered unusable for other tasks. It is important, therefore, to maximise performance without wasting CPUs.

Determining the optimum number of CPUs required to service interrupts can be a tricky task. In particular, it is important to determine whether performance flatlines are due to application or network processing overheads. In general, we recommend increasing the number of CPUs when application traffic is composed of short, frequent exchanges ("chatty" traffic) or when interrupt coalescing is reduced (or disabled) to improve network latency. We also recommend runtime monitoring of interrupt service CPUs (with `mpstat`, `top` or similar tools) to verify they are not saturated.

It is possible to empirically determine the minimum number of CPUs needed for interrupts for a particular hardware and software configuration by monotonically increasing the number of CPUs servicing interrupts when a flatline in performance is detected to the point where no significant difference in performance is noted with additional CPUs. Note however that we advocate increasing the number of server instances(as indicated in Section 4.3) *before* increasing the number of CPUs to verify the application is not the bottleneck.

An example of empirical determination on the example server is illustrated in Figure 6 and Figure 7 which shows Kernel and OpenOnload Memcache performance as the number of CPUs servicing interrupts is increased. Using the server configuration specified in Section 4.1 we notice Kernel networking performance (Figure 6) is interrupt limited with 1, 2 and 3 CPUs, flatlining

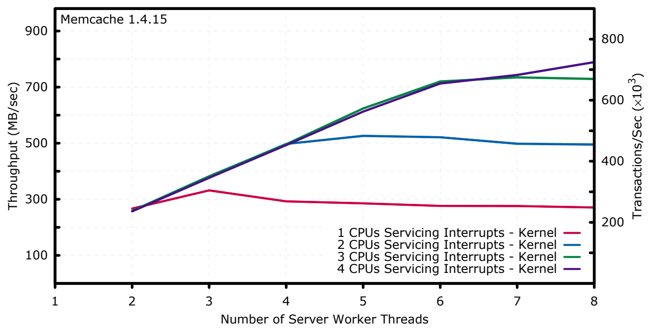

Figure 6: Kernel Performance With Increasing Numbers of Interrupt Service CPUs

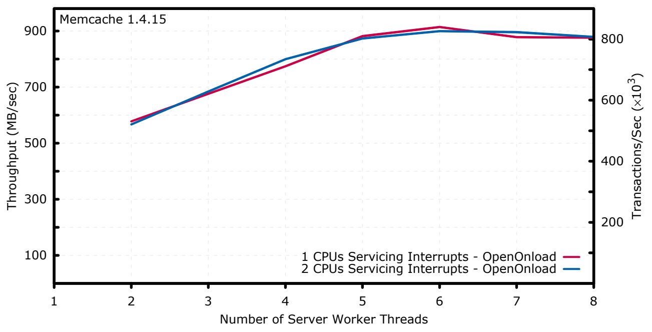

Figure 7: OpenOnload Performance With Increasing Numbers of Interrupt Service CPUs

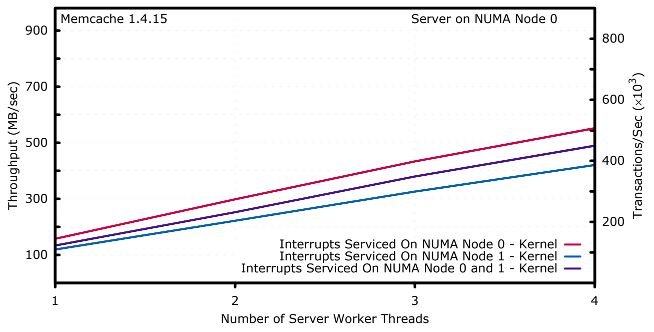

Figure 8: Performance As Interrupt Servicing Is Varied Across NUMA Nodes

at 3, 4 and 6 worker threads respectively. Using 4 CPUs however allows the system to scale in proportion to the active number of worker threads. In contrast, OpenOnload performance (Figure 7) shows no performance increase as interrupt service CPUs are increased. We can thus conclude that OpenOnload is not interrupt limited.

4.4.2 Localize Interrupt Service CPUs

In configurations where Memcache servers are restricted to a single NUMA node (e.g. it is common to multiplex Memcache servers with I/O bound applications) we recommend localising interrupt servicing to cores on the same package as those hosting Memcache worker threads in order to maximise performance.

Figure 8 illustrates the effect of this optimisation based on our example configuration. As the results show, for a single server comprising of up to 4 worker threads running on NUMA Node 0, the best performance is achieved by co-locating two interrupt service CPUs on Node 0. Spreading interrupt servicing on CPUs contained in Node 0 and 1 reduces performance compared to the Node 0 case. Finally, only servicing interrupts on CPUs on a remote node leads to the worst performance. The performance differential between configurations is proportional to the number of active worker threads – for 4 worker threads the best configuration outperforms the worst by 31%.

4.4.3 Shield Interrupt Service CPUs

As outlined in Section 3.2 (Step 6), we recommend running interrupt service CPUs in a separate shield to prevent applications being scheduled on them thereby leading to reduced performance resulting from a decrease in the amount of CPU available and an increase in (data) cache pollution. While it may be intuitive to co-locate interrupts and the Memcache server where load is very low (Unshielded Interrupt Servicing) or where latency is important, our experience has shown that, in practice, Unshielded configurations usually lead to worse performance than Shielded configurations.

An experimental result comparing Shielded and Unshielded Interrupt configurations for the Kernel and OpenOnload networking stacks is provided in Figure 9 and Figure 10. In this comparison a single server, comprising of a single worker thread, is started on NUMA Node 0. Furthermore, all interrupts are serviced on CPUs on Node 0. Clients are configured to carry out

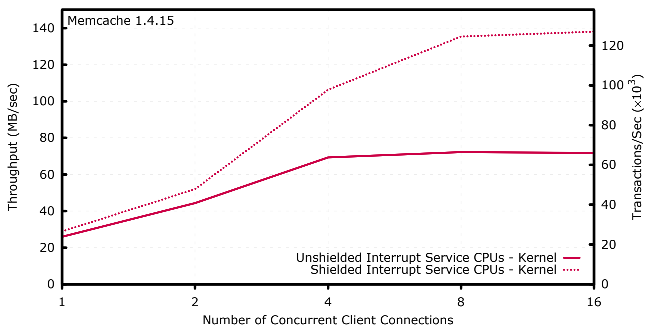

Figure 9: Kernel Performance With Shielded vs Unshielded Interrupt Servicing

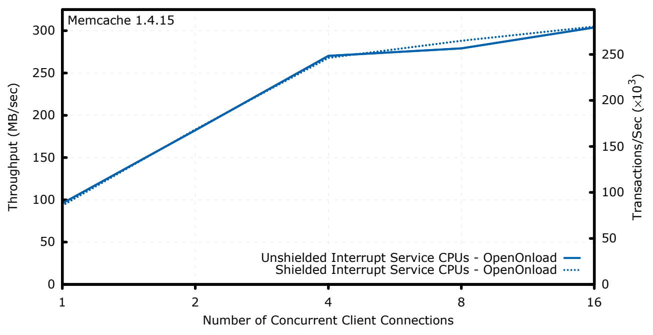

Figure 10: OpenOnload Performance With Shielded vs Unshielded Interrupt Servicing

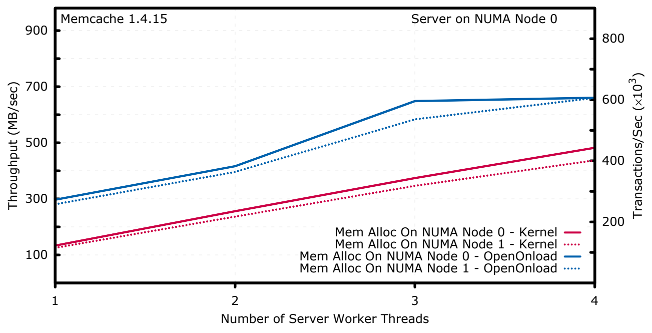

Figure 11: Performance With Local vs Remote Memory Allocation

ping-pong requests to the server, always maintaining a single outstanding request. We vary load from 1–16 clients, doubling the amount of clients at every iteration.

For the Kernel stack (Figure 9), the results show that, at best, Unshielded interrupt servicing performs approximately equally to the shielded case. As the number of clients (and, by extension, load) increases, the performance of the unshielded case flatlines at approximately 70 MB/sec while the performance of the shielded case continues to increase until server CPU saturation occurs at 138 MB/sec. In contrast, due to the low reliance of OpenOnload on interrupts, performance is approximately the same for both shielded and unshielded configurations (Figure 10).

4.5 Localize Memory Allocation

As explained in Section 2.2, accessing non-local (remote) memory on a NUMA node is slower than accessing local memory. Figure 11 illustrates the performance impact of remote memory access on Memcache running on the Kernel and OpenOnload network stacks. A single server is configured to allocate all cache memory on Node 0 or Node 1 respectively (achieved via the `numactl --membind` command). The results show that, with the Kernel stack, the performance differential between localised and remote allocation grows linearly with an increasing number of threads, from 7 MB/sec with 1 worker thread to 44 MB/sec with 4 worker threads.

In contrast, with OpenOnload, the performance differential for localised and remote allocation increases in proportion to the number of active worker threads: 16 MB/sec for 1 thread, 21 MB/sec for 2 threads and 65 MB/sec for 3 threads. In this scenario, adding an additional (4th) worker thread in the remote allocation configuration increases the amount of CPU available to the server, allowing it to perform to the upper performance bound (659 MB/sec) imposed by server design (Section 4.1).

4.6 Client Side Optimizations

In this section we outline a number of client-side optimisations that can improve overall Memcache server performance.

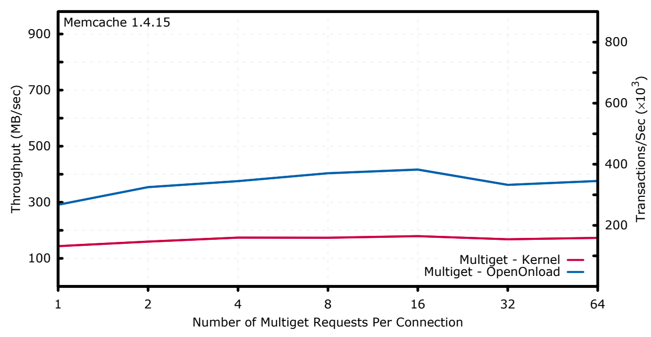

Figure 12: Performance With Increasing Multiget Requests Per Connection

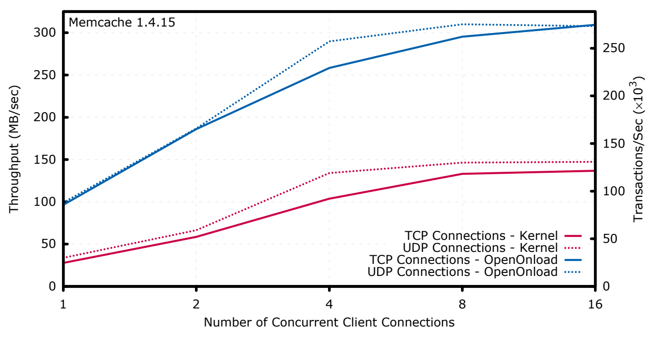

Figure 13: Performance Of TCP vs UDP Get Requests

4.6.1 Multiget

Memcached supports *multiget* requests, a performance optimisation that operates on the principle that throughput can be increased by amortising the network connection cost over several Memcache `get` requests. Clients connect to the server and issue a get request for several keys to which the server responds in-order.

There are significant performance gains to be realised if client logic can be modified to support multiget requests. Figure 12 illustrates single threaded server performance as the client is configured to issue increasing numbers of multiget requests per connection. There is a linear increase in performance with increasing numbers of multiget requests up until 16 multiget requests per connection after which performance decreases. At peak, the 16-multiget configuration results in a 25% increase over baseline (1 get per request) performance for the Kernel stack and a 43% increase for the OpenOnload stack.

4.6.2 UDP Get

Clients accessing light-to-moderately loaded servers can improve throughput by issuing UDP based get requests. Figure 13 illustrates the difference between TCP and UDP based retrieval performance on a single worker thread based server running on NUMA Node 0. In this configuration, the client is configured to issue an increasing number of concurrent retrieval requests over both TCP and UDP transport.

For the Kernel stack, UDP performance is *always* (if marginally) better than TCP performance with the biggest difference in relative performance observed with 4 concurrent requests where UDP provides 30% higher throughput than TCP. Similarly, with 4 concurrent requests OpenOnload provides 12% higher throughput over UDP compared to TCP. As client load increases and performance is dominated by application processing overhead the advantage of UDP based transport decreases.

4.6.3 OpenOnload - Stack Per Thread

OpenOnload supports the `EF_STACK_PER_THREAD` configuration option which, when set, instructs the stack to demarcate internal networking data structures on a per-thread basis. We recommend this option is always set in multithreaded client applications where threads create a large number of active sockets as the benefits gained by reducing data structure sharing between threads can be substantial, especially at high load.

The quantitative benefits of per-thread stacks are clearly illustrated in Figure 14 which shows that, for a 2 server Memcache 1.4.15 configuration, disabling per-thread stacks limits client performance to approximately 50% of maximum link throughput.

4.7 Consider Hyperthreading

As outlined in Section 2.1, hyperthreading is a useful mechanism for increasing CPU utilisation by increasing parallelism in a software-transparent manner. While the exact effects of enabling hyperthreading for a given workload are hard to quantify without empirical experimentation, we recommend that readers concerned with maximising CPU utilisation evaluate hyperthreading as a potential means for increasing hardware utilisation.

Figure 15 provides an example of the potential benefits of running a hyperthreaded configuration. In this experiment we compare the 2 server configuration (One Per NUMA Node) described in Section 3.4 and Section 4.2 against one in which we

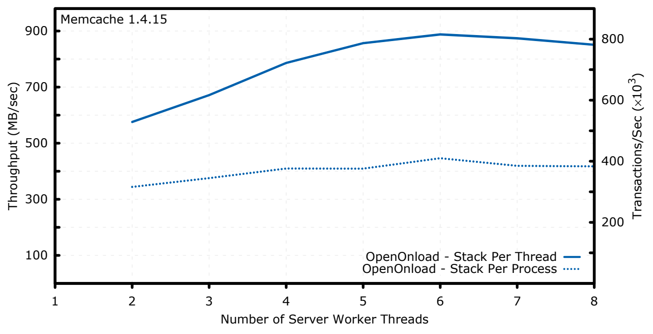

Figure 14: OpenOnload Performance Of Stack Per Thread vs. Stack Per Process Configurations

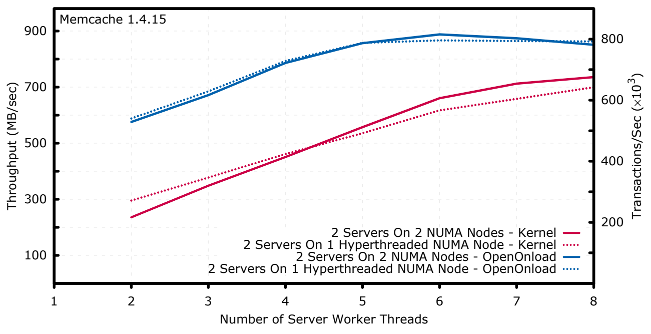

Figure 15: Performance Of Non-Hyperthreaded vs Hyperthreaded CPUs

enable hyperthreading (resulting in 12 logical processors per NUMA node). We configure the first 4 CPUs. on Node 0 to service interrupts and run 2 servers (each of up to 4 threads) on the remaining 8 CPUs.

As the results show, with the Kernel stack hyperthreaded performance is marginally better than non-hyperthreaded performance with low numbers of worker threads. This can be attributed to the fact that all interrupts are locally serviced on Node 0. As the number of threads increases hyperthreaded performance slightly trails non-hyperthreaded performance, resulting in a maximum 4.8% reduction in performance for 8 threads spread over 2 servers. OpenOnload however shows approximately equivalent performance in both configurations. With both stacks, running in hyperthreaded mode results in a 100% increase in the amount of available (free) CPU on the system.

4.8 Further Exploration

For brevity, a number of recent software mechanisms designed to increase the efficiency of network processing have been omitted. Readers interested in exploring further performance improvements are advised to research Receive Packet Steering [1], Receive Flow Steering [2] and TCP Fastopen [3].

5 Conclusion

This document has outlined the general principles and specific steps that need to be carried out in order to optimise Memcache server performance on Solarflare hardware in Linux environments and provided insight into the effects of tunable parameters on throughput performance. Solarflare welcomes any clarifications, corrections or improvements to this technical report on support@solarflare.com.

A Hardware Specification & Software Versions

The results obtained in this technical report are based on the hardware specification and software versions detailed below:

A.1 Hardware Specification

Server and load generation client are identically specified:

Component	Description	Comments
System	Dell PowerEdge C8220	
CPU	2 × Intel(R) Xeon(R) CPU E5-2620 @ 2.00GHz	For a total of 12 cores
RAM	8 × Hynix 4GB PC3-10600 DDR3-1333MHz ECC	For a total of 32GB equally distributed between both CPUs.
NIC	1 × Solarflare Communications SFC9020 Solarstorm SFN5322F-R1	One port used, directly connected to peer via SFP+ Direct Attach.

Table 1: Test setup: Hardware Specification

A.2 Software Versions

Server and load generation client run identical software versions:

Software Package	Version	Comments
Linux distribution	Red Hat Enterprise Linux Server release 6.2 (Santiago)	64 bit version
Kernel	3.2.43	Stock kernel, configuration derived from configuration of installed Red Hat Kernel
OpenOnload	201210-u1	OpenOnload TCP/IP stack
Memcache 1.4.15	1.4.15	1.4 server branch representative
Memcache 1.6	Git commit id: 92b232d	1.6 server branch representative
Libevent	2.0.21	Event library against which Memcache is linked
Memslap	1.0	Load generation client
Cpuset	1.5.5	

Table 2: Test setup: Software Versions

References

[1] J. Corbet, "Receive packet steering." `http://lwn.net/Articles/362339/`, November 2009. Retrieved 12th June 2013.

[2] J. Edge, "Receive flow steering." `http://lwn.net/Articles/382428/`, April 2010. Retrieved 12th June 2013.

[3] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "TCP fast open," in *Proceedings of the Seventh Conference On Emerging Networking Experiments and Technologies*, CoNEXT '11, (New York, NY, USA), pp. 21:1–21:12, ACM, 2011.