# Many-Core Key-Value Store

Mateusz Berezecki
*Facebook*
*mateuszb@fb.com*

Eitan Frachtenberg
*Facebook*
*etc@fb.com*

Mike Paleczny
*Facebook*
*mpal@fb.com*

Kenneth Steele
*Tilera*
*ken@tilera.com*

***Abstract*—Scaling data centers to handle task-parallel workloads requires balancing the cost of hardware, operations, and power. Low-power, low-core-count servers reduce costs in one of these dimensions, but may require additional nodes to provide the required quality of service or increase costs by under-utilizing memory and other resources.**

**We show that the throughput, response time, and power consumption of a high-core-count processor operating at a low clock rate and very low power consumption can perform well when compared to a platform using faster but fewer commodity cores. Specific measurements are made for a key-value store, Memcached, using a variety of systems based on three different processors: the 4-core Intel Xeon L5520, 8-core AMD Opteron 6128 HE, and 64-core Tilera TILEPro64.**

***Keywords*-Low-power architectures; Memcached; Key-Value store; Many-core processors**

## I. Introduction

Key-value (KV) stores play an important role in many large websites. Examples include: Dynamo at Amazon [1]; Redis at Github, Digg, and Blizzard Interactive [1]; Memcached at Facebook, Zynga and Twitter [2], [3]; and Voldemort at Linkedin[2]. All these systems store ordered $(key, value)$ pairs and are, in essence, a distributed hash table.

A common use case for these systems is as a layer in the data-retrieval hierarchy: a cache for expensive-to-obtain values, indexed by unique keys. These values can represent any data that is cheaper or faster to cache than re-obtain, such as commonly accessed results of database queries or the results of complex computations that require temporary storage and distribution.

Because of their role in data-retrieval performance, KV stores attempt to keep much of the data in main memory, to avoid expensive I/O operations [4], [5]. Some systems, such as Redis or Memcached, keep data exclusively in main memory. In addition, KV stores are generally network-enabled, permitting the sharing of information across the machine boundary and offering the functionality of large-scale distributed shared memory without the need for specialized hardware.

This sharing aspect is critical for large-scale web sites, where the sheer data size and number of queries on it far exceed the capacity of any single server. Such large-data workloads can be I/O intensive and have no obvious access patterns that would foster prefetching. Caching and sharing the data among many front-end servers allows system architects to plan for simple, linear scaling, adding more KV stores to the cluster as the data grows. At Facebook, we have used this property grow larger and larger clusters, and scaled Memcached accordingly[3].

But as these clusters grow larger, their associated operating cost grows commensurably. The largest component of this cost, electricity, stems from the need to power more processors, RAM, disks, etc. Lang [6] and Andersen [4] place the cost of powering servers in the data center at up to $50\%$ of the three-year total ownership cost (TCO). Even at lower rates, this cost component is substantial, especially as data centers grow larger and larger every year [7].

One of the proposed solutions to this mounting cost is the use of so-called "wimpy" nodes with low-power CPUs to power KV stores [4]. Although these processors, with their relatively slow clock speeds, are inappropriate for many demanding workloads [6], KV stores can present a cost-effective exception because even a slow CPU can provide adequate performance for the typical KV operations, especially when including network latency.

In this paper, we focus on a different architecture: the Tilera TILEPro64 64-core CPU [8], [9], [10], [11], [12]. This architecture is very interesting for a Memcached workload in particular (and KV stores in general), because it combines the low-power consumption of slower clock speeds with the increased throughput of many independent cores (described in detail in Sections II and III). As mentioned above, previous work has mostly concentrated on mapping KV stores to low-core-count "wimpy" nodes (such as the Intel Atom), trading off low aggregate power consumption for a larger total node count [4]. This trade-off can mean higher costs for hardware, system administration, and fault management of very large clusters. The main contribution of this paper is to demonstrate a low-power KV storage

[1]http://redis.io
[2]http://project-voldemort.com

[3]For example, see http://facebook.com/note.php?note_id=39391378919 for a discussion of Facebook's scale and performance issues with Memcached.

978-1-4577-1221-0/11/$26.00 © 2011 IEEE

solution that offers better performance/Watt than comparable commodity solutions, without increasing the overall server count and associated operating cost. A secondary contribution is the detailed description of the adjustments required to the Memcached software in order to take advantage of the many-core Tilera architecture (Sec. IV). The third main contribution is a thorough experimental evaluation of the Tilera TILEPro64 system under various workload variations, and an exploration of its power and performance characteristics as a KV store, compared to typical x86-based Memcached servers (Sec. V).

## II. Memcached Architecture

Memcached[4] is a simple, open-source software package that exposes data in RAM to clients over the network. As data size grows in the application, more RAM can be added to a server, or more servers can be added to the network. In the latter case, servers do not communicate among themselves—only clients communicate with servers. Clients use consistent hashing [13] to select a unique server per $key$, requiring only the knowledge of the total number of servers and their IP addresses. This technique presents the entire aggregate data in the servers as a unified distributed hash table, keeps servers completely independent, and facilitates scaling as data size grows.

Memcached's interface provides all the basic primitives that hash tables provide—insertion, deletion, and lookup/retrieval—as well as more complex operations built atop them. The complete interface includes the following operations:

- STORE: stores $(key, value)$ in the table.
- ADD: adds $(key, value)$ to the table iff the lookup for $key$ fails.
- DELETE: deletes $(key, value)$ from the table based on $key$.
- REPLACE: replaces $(key, value_1)$ with $(key, value_2)$ based on $(key, value_2)$.
- CAS: atomic compare-and-swap of $(key, value_1)$ with $(key, value_2)$.
- GET: retrieves either $(key, value)$ or a set of $(key_i, value_i)$ pairs based on $key$ or $\{key_i$ s.t. $i = 1...k\}$.

The first four operations are write operations (destructive) and follow the same code path as for STORE (Fig. 1). Write operations are always transmitted over the TCP protocol to ensure retries in case of a communication error. STORE requests that exceed the server's memory capacity incur a cache eviction based on the least-recently-used (LRU) algorithm.

GET requests follow a similar code path (Fig. 2). If the key to be retrieved is actually stored in the table (a *hit*), the $(key, value)$ pair is returned to the client. Otherwise (a *miss*), the server notifies the client of the missing key. One notable difference from the write path, however, is that clients can opt to use the faster but less-reliable UDP protocol for GET requests.

[4]http://memcached.org/

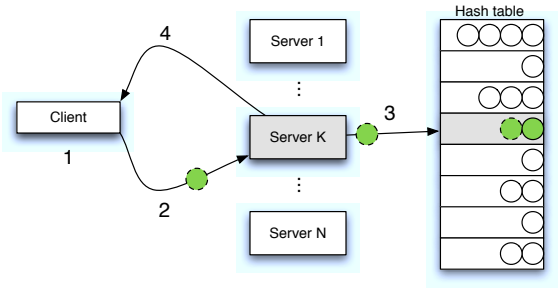



Figure 1: Write path: The client selects a server (1) by computing $k_1 = consistent\_hash_1(key)$ and sends (2) it the $(key, value)$ pair. The server then calculates $k_2 = hash_2(key) \mod M$ using a different hash function and stores (3) the entire $(key, value)$ pair in the appropriate slot $k_2$ in the hash table, using chaining in case of conflicts. Finally, the server acknowledges (4) the operation to the client.

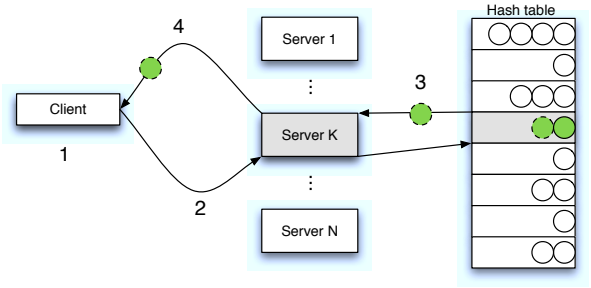



Figure 2: Read path: The client selects a server (1) by computing $k_1 = consistent\_hash_1(key)$ and sends (2) it the $key$. The server calculates $k_2 = hash_2(key) \mod M$ and looks up (3) a $(key, value)$ pair in the appropriate slot $k_2$ in the hash table (and walks the chain of items if there were any collisions). Finally, the server returns (4) the $(key, value)$ to the client or notifies it of the missing record.

It is worth noting that GET operations can take multiple keys as an argument. In this case, Memcached returns all the KV pairs that were successfully retrieved from the table. The benefit of this approach is that it allows aggregating multiple GET requests in a single network packet, reducing network traffic and latencies. But to be effective, this feature requires that servers hold a relatively large amount of RAM, so that servers are more likely to have multiple keys of interest in each request. (Another reason for large RAM per server is to amortize the RAM acquisition and operating costs over fewer servers.) Because some clients make extensive use of this feature, "wimpy" nodes are not a practical proposition for them, since they typically support relatively small amounts of RAM per server.

## III. TILEPro64 Architecture

Tile processors are a class of general-purpose and power-efficient many-core processors from Tilera using switched,

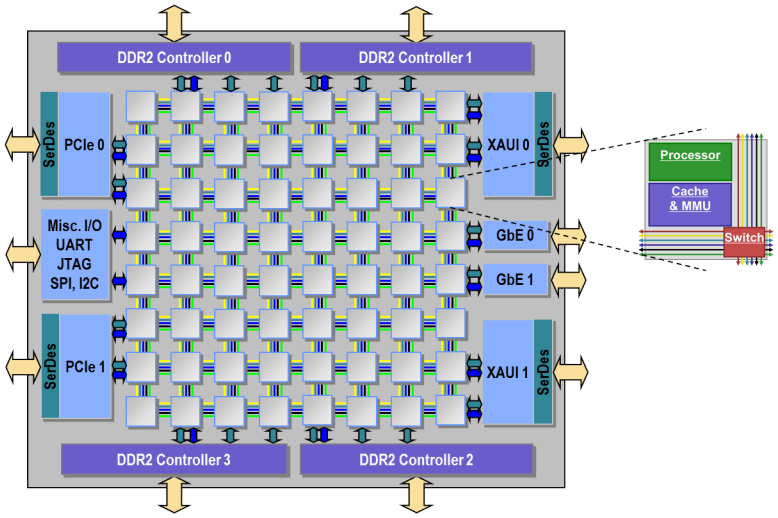



Figure 3: High-level overview of the Tilera TILEPro64 architecture. The processor is an 8x8 grid of cores. Each of the cores has a 3-wide VLIW CPU, a total of 88KB of cache, MMU and six network switches, each a full 5 port 32-bit-wide crossbar. I/O devices and memory controllers connect around the edge of the mesh network.

on-chip mesh interconnects and coherent caches. The TILEPro64 is Tilera's second generation many-core processor chip, comprising 64 power efficient general-purpose cores connected by six 8x8 mesh networks. The mesh networks also connect the on-chip Ethernet, PCIe, and memory controllers. Cache coherence across the cores, the memory, and I/O allows for standard shared memory programming. The six mesh networks efficiently move data between cores, I/O and memory over the shortest number of hops. Packets on the networks are dynamically routed based on a two-word header, analogous to the IP and port in network packets, except the networks are loss-less. Three of the networks are under hardware control and manage memory movement and cache coherence. The other three networks are allocated to software. One is used for I/O and operating system control. The remaining two are available to applications in user space, allowing low-latency, low-overhead, direct communication between processing cores, using a user-level API to read and write register-mapped network registers.

Each processing core, shown as the small gray boxes in Fig. 3, comprises a 32-bit 5-stage VLIW pipeline with 64 registers, L1 instruction and data caches, L2 combined data and instruction cache, and switches for the six mesh networks. The 64KB L2 caches from each of the cores form a distributed L3 cache accessible by any core and I/O device. The short pipeline depth reduces power and the penalty for a branch prediction miss to two cycles. Static branch prediction and in-order execution further reduce area and power required. Translation look-aside buffers support virtual memory and allow full memory protection. The chip can address up to 64GB of memory using four on-chip DDR2 memory controllers (greater than the 4GB addressable by a single Linux process). Each memory controller reorders memory read and write operations to the DIMMs to optimize memory utilization. Cache coherence is maintained by each cache-line having a "home" core. Upon a miss in its local L2 cache, a core needing that cache-line goes to the home core's L2 cache to read the cache-line into its local L2 cache. Two dedicated mesh networks manage the movements of data and coherence traffic in order to speed the cache coherence communication across the chip. To enable cache coherence, the home core also maintains a directory of cores sharing the cache line, removing the need for bus-snooping cache coherency protocols, which are power-hungry and do not scale to many cores. Because the L3 cache leverages the L2 cache at each core, it is extremely power efficient while providing additional cache resources. Figure 3 shows the I/O devices, 10G and 1GB Ethernet, and PCI-e, connecting to the edge of the mesh network. This allows direct writing of received packets into on-chip caches for processing and vice-versa for sending.

## IV. Execution Model

Although TILEPro64 has a different architecture and instruction set than the standard x86-based server, it provides a familiar software development environment with Linux, gcc, autotools, etc. Consequently, only a few software tweaks to basic architecture-specific functionality suffice to successfully compile and execute Memcached on a TILEPro64 system. However, this naïve port does not perform well and can hold relatively little data. The problem lies with Memcached's share-all multithreaded execution model (Fig. 4). In a standard version of Memcached, one thread acts as the event demultiplexer, monitoring network sockets for incoming traffic and dispatching event notifications to one of the $N$ other threads. These threads execute incoming

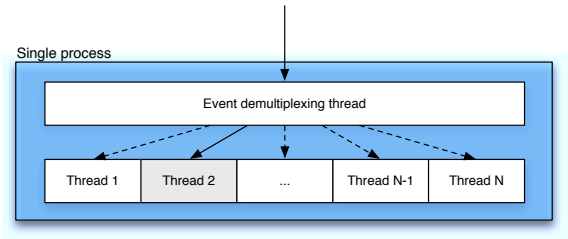



Figure 4: Execution model of standard version of Memcached.

requests and return the responses directly to the client. Synchronization and serialization are enforced with locks around key data structures, as well as a global hash table lock that serializes all accesses to the hash table. This serialization limits the scalability and benefits we can expect with many cores.

Moreover, recall that TILEPro64 has a 32-bit instruction set, limiting each process' address space to $2^{32}$ bytes (4GB) of data. As discussed in Sec. II, packing larger amounts of data in a single node holds both a cost advantage (by reducing the overall number of servers) and a performance advantage (by batching multiple get requests together).

However, the physical memory limit on the TILEPro64 is currently 64GB, allowing different processes to address more than 4GB in aggregate. The larger physical address width suggests a solution to the problem of the 32-bit address space: extend the multithreading execution model with multiple independent processes, each having its own address space. Figure 5 shows the extended model with new processes and roles. First, two hypervisor cores handle I/O ingress and egress to the on-chip network interface, spreading I/O interrupts to the appropriate CPUs and generating DMA commands. The servicing of I/O interrupts and network layer processing (such as TCP/UDP) is now owned by $K$ dedicated cores, managed by the kernel and generating data for user sockets. These requests arrive to the main Memcached process as before, which contains a demultiplexing thread and $N$ worker threads. Note, however, that these worker threads are statically allocated to handle either TCP or UDP requests, and each thread is running on exactly one exclusive core. These threads do not contain KV data. Rather, they communicate with $M$ distinct processes, each containing a shard of the KV data table in its own dedicated address space, as follows:

When a worker thread receives a request, it identifies the process that owns the relevant table shard with modulo arithmetic. It then writes the pertinent request information in a small region of memory that is shared between the thread and the process, and had been previously allocated from the global pool using a memory space attribute. The worker thread then notifies the shard process of the request via a message over the on-chip user-level network to the

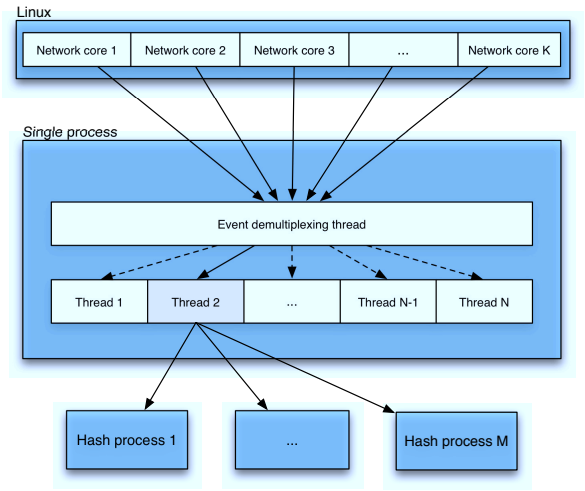



Figure 5: Execution model of Memcached on TILEPro64.

shard process (using a low-latency software interface). On a STORE request, the shard process copies the data into its private memory. For a GET operation, the shard process copies the requested value from its private hash table memory to the shared memory, to be returned to the requesting thread via the user-level network. For multi-GET operations, the thread merges all the values from the different shards into on the response packet.

This execution model solves the problem of the 32-bit address space limitations and that of a global table lock. Partitioning the table data allows each shard to comfortably reside within the 32-bit address space. Owning each table shard by a single process also means that all requests to mutate it are serialized and therefore require no locking protection. In a sense, this model adds data parallelism to what was purely task-parallel.

## V. Experimental Evaluation and Discussion

This section explores the performance of the modified Memcached on the TILEPro64 processor, and compares it to the baseline version on commodity x86-based servers. We start with a detailed description of the methodology, metrics, and hardware used, to allow accurate reproduction of these results. We then establish several workload and configuration choices by exploring the parameter space and its effect on performance on the TILEPro64. Having selected these parameters, we continue with a performance comparison to the x86-based server and a discussion of the differences. Finally, we add power to the analysis and look at the complete picture of performance per Watt at large scale.

### A. Methodology and Metrics

We measure the performance of these servers by configuring them to run Memcached (only), using the following command line on x86:

```
memcached -p 11211 -U 11211 \
  -u nobody -m <memory size>
```

and on the TILEPro64:

```
memcached -p 11211 -U 11211 -u root \
  -m <memory size> -t $tcp -V $part
```

for the Tilera system, where $tcp and $part are variables representing how many TCP and hash partitions are requested (with the remaining number of cores allocated to UDP threads). On a separate client machine we use the open-source *mcblaster* tool to stimulate the system under test and report the measured performance. A single run of mcblaster consists of two phases. During the initialization phase, mcblaster stores data in Memcached by sending requests at a fixed rate $\lambda_1$, the argument to -W. This phase runs for 100 seconds (initialization requests are sent using the TCP protocol), storing $1,000,000$ 32-byte objects, followed by 5 seconds of idle time, with the command line:

```
  mcblaster -z 32 -p 11211 -W 50000 -d 100 \
-k 1000000 -c 10 -r 10
  <hostname>
```

During the subsequent phase, mcblaster sends query packets requesting randomly-ordered keys initialized in the previous phase and measures their response time using the command line:

```
  mcblaster -z 32 -p 11211 -d 120 -k 1000000 \
-W 0 -c 20 -r $rate
  <hostname>
```

where $rate is a variable representing offered request rate.

We concentrate on two metrics of responsiveness and throughput. The first is the median response time (latency) of GET requests at a fixed offered load. The second, complementary, metric is the *capacity* of the system, defined as the approximate highest offered load (in transactions per second, or TPS) at which the mean response time remains under 1msec. Although this threshold is arbitrary, it is in the same order of magnitude of cluster-wide communications and well below the human perception level. We do not measure multi-GET requests because they exhibit the same read TPS performance as individual GET requests. Finally, we also measure the power usage of the various systems using Yokogawa WT210 power meter, measuring the wall power directly.

### B. Hardware Configuration

The TILEPro64 S2Q system comprises a total of eight nodes, but we will focus our initial evaluation on a single node for a fairer comparison to independent commodity nodes. In practice, all nodes have independent processors, memory, and networking, so the aggregate performance of multiple nodes scales linearly, and can be extrapolated from a single node's performance (we verified this assumption experimentally).

Our load-generating host contains a dual-socket quad-core Intel Xeon L5520 processor clocked at 2.27GHz, with 72GB of ECC DDR3 memory. It is also equipped with an Intel 82576 Gigabit ET Dual Port Server Adapter network interface controller that can handle transaction rates of over 500,000 packets/sec.

We used these systems in our evaluation:

- 1U server with single/dual socket quad-core Intel Xeon L5520 processor clocked at 2.27GHz (65W TDP) and a varying number of ECC DDR3 8GB 1.35V DIMMs.
- 1U server single/dual socket octa-core AMD Opteron 6128 HE processor clocked at 2.0GHz (85W TDP) and a varying number of ECC DDR3 RAM DIMMs.
- Tilera S2Q[5]: a 2U server built by Quanta Computer containing eight TILEPro64 processors clocked at 866MHz, for a total of 512 cores. The system uses two power supplies (PSUs), each supporting two trays, which in turn each hold two TILEPro64 nodes. Each node holds 32GB of ECC DDR2 memory, a BMC, two GbE ports (we used one of these for this study), and two 10 Gigabit XAUI Ethernet ports.

We chose these low-power processors because they deliver a good compromise between performance and power compared to purely performance-oriented processors.

The Xeon server used the Intel 82576 GbE network controller. We turned hyperthreading off since it empirically shows little performance benefit for Memcached, while incurring additional power cost. The Opteron server used the Intel 82580 GbE controller. Both network controllers can handle packet rates well above our requirements for this study.

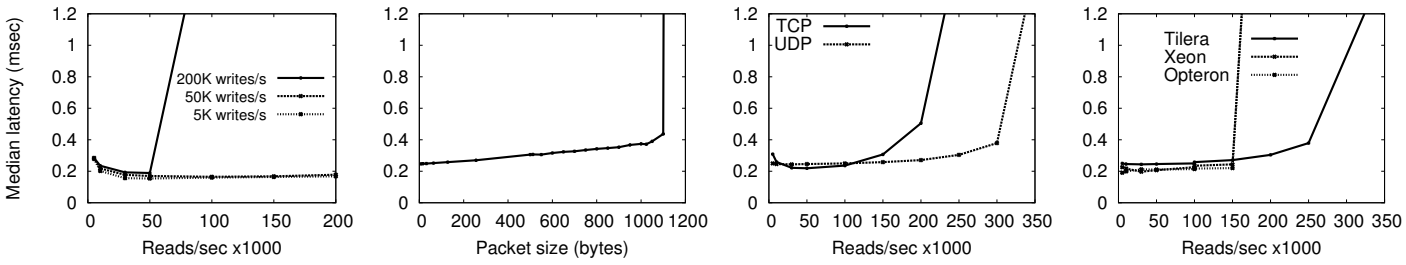
In all of our tests we used Memcached version 1.2.3h, running under CentOS Linux with kernel version 2.6.33.

### C. Parameter Space Exploration

We begin our exploration by determining the workload to use, specifically the mix between GET and STORE requests. In real-world scenarios we often observe that read requests far outnumber write requests. We therefore typically concentrate only on measuring read rates, and assume that the effect of writes on read performance is negligible for realistic workloads. To verify this assumption we conducted the following experiment: We set write rates at three levels: 5,000, 30,000, and 200,000 writes/sec, and varied read rates from 0 to 300,000 reads/sec. The packet size (including headers) was fixed at 64 bytes, as will be explained shortly. Fig. 6(a) shows that latency does not change significantly with increasing read rates for the lowest two curves. We observe a small change in the top curve, corresponding to 200,000 writes/sec—an unrealistically high rate. This data confirms that moderate write rates have little effect on read performance, so we avoid write requests in all subsequent experiments, after the initialization phase.

We similarly tested the effect of packet size (essentially $value$ size) on read performance. Packet sizes are limited

[5]tilera.com/solutions/cloud_computing




(a) (b) (c) (d)

Figure 6: Median response time as a function of (a) workload mix; (b) payload size; (c) protocol; (d) architecture (32GB).

to the system's MTU when using the UDP protocol for Memcached, which in our system is 1,500 bytes. To test this parameter, we fixed the read rate at $\lambda_{size} = 100,000$ TPS and varied the payload size from 8 to 1,200 bytes. The results are presented in Fig. 6(b). The latency spike at the right is caused by the network's bandwidth limitation: sending 1,200-byte sized packets at the rate of $\lambda_{size}$ translates to a data rate of 960Mbps, very close to the theoretical maximum of the 1Gbps channel. Because packet size hardly affects read performance across most of the range, and because we typically observe sizes of less than 100 bytes in real workloads, we set the packet size to 64 bytes in all experiments.

Another influential parameter is the transmission protocol for read requests (we always write over TCP). Comparing the two protocols, as shown in Fig. 6(c), shows a clear throughput advantage to the UDP protocol. This advantage is readily explained by the fact that TCP is a transaction-based protocol and as such, it has a higher overhead associated with a large number of small packets. We therefore limit most of our experiments to UDP reads, although we will revisit this aspect for the next parameter.

Last, but definitely not least, is the static core allocation to roles. During our experiments, we observed that different core allocations among the 60 available cores (4 are reserved for Linux), have substantial impact on performance. We systematically evaluated over 100 different allocations, testing for correlations and insights (including partial allocations that left some cores unallocated). To conserve space, we reproduce here only a small number of these results, enough to support the following conclusions:

- The number of hash table processes determines the node's total table size, since each process owns an independent shard. But allocating cores beyond the memory requirements (in our case, 6 cores for a total of 24GB, leaving room for Linux and other processes), does not improve performance (Fig. 7(a),(b)).
- The amount of networking cores does affect performance, but only up to a point (Fig. 7(a),(c),(d)). Above 12 networking cores, performance does not improve significantly, regardless of the number of UDP/TCP cores.
- TCP cores have little effect on UDP read performance, and do not contribute much after the initialization phase. They do affect TCP read capacity, which for allocations (g) and (h), for example, is 215,000 and 118,000 TPS respectively, so we empirically determined 12 cores to be a reasonable allocation for occasional writes.
- Symmetrically, UDP cores play a role for UDP read capacity, so we allocate all available cores to UDP, once the previous requirements have been met (Fig. 7(e)–(f)).

These experiments served to identify the most appropriate configuration for the performance comparisons: measuring UDP read capacity for 64-byte packets with no interfering writes, at a core allocation of 8 network workers, 6 hash table processes, 12 TCP cores, and 34 UDP cores (Fig. 7 (a)).

### D. Performance Comparison

Fig. 6(d) shows the median response time for the three architectures under increasing load. The data exposes the difference between processors optimized for single-threaded performance vs. multi-threaded throughput. The x86-based processors, with faster clock speeds and deeper pipelines, complete individual GET requests $\approx$ 20% faster than the TILEPro64 across most load points. (Much of the response time is related to memory and network performance, where the differences are less pronounced.) This performance advantage is not qualitatively meaningful, because these latency levels are all under the 1msec capacity threshold, providing adequate responsiveness. On the other hand, fewer cores, combined with centralized table and network management, translate to a lower saturation point and significantly reduced throughput for the x86-based servers.

This claim is corroborated when we analyze the scalability of Memcached as we vary the number of cores (Fig. 8). Here, the serialization in Memcached and the network stack prevents the x86-based architectures from scaling to even relatively few cores. The figure clearly shows that even within a single socket and with just 4 cores, performance scales poorly and cannot take advantage of additional

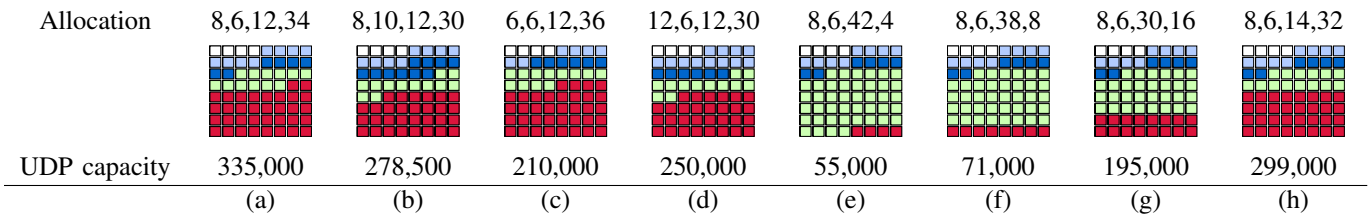



Figure 7: Read capacity at various core allocations. the numbers in each sequence represent the cores allocated to network workers (light blue), hash table (dark blue), TCP (green), and UDP (red), respectively. Linux always runs on 4 cores (white).

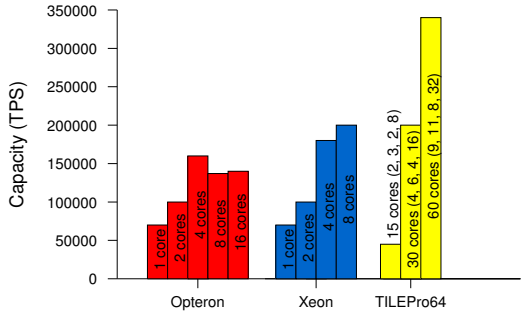



Figure 8: Scalability of different architectures under increasing number of cores. For x86, we simply change the number of Memcached threads with the -t parameter, since threads are pinned to individual cores. For TILEPro64, we turn off a number of cores and reallocate threads as in Fig. 7.

| Configuration | RAM (GB) | Capacity (TPS) | Power (Watt) |
|---|---|---|---|
| 1 × TILEPro64 (one node) | 32 | 335,000 | 90 |
| 2 × TILEPro64 (one PCB) | 64 | 670,000 | 138 |
| 4 × TILEPro64 (one PSU) | 128 | 1,340,000 | 231 |
| Single Opteron | 32 | 165,000 | 115 |
| Single Opteron | 64 | 165,000 | 121 |
| Dual Opteron | 32 | 160,000 | 165 |
| Dual Opteron | 64 | 160,000 | 182 |
| Single Xeon | 32 | 165,000 | 93 |
| Single Xeon | 64 | 188,000 | 100 |
| Dual Xeon | 32 | 200,000 | 132 |
| Dual Xeon | 64 | 200,000 | 140 |

Table I: Power and capacity at different configurations. Performance differences at the single-socket level likely stem from imbalanced memory channels.

threads. In fact, we must limit the thread count to 4 on the Opteron to maximize its performance. On the other hand, the TILEPro64 implementation can easily take advantage of (and actually requires) more cores for higher performance. Another aspect of this scaling shows in Fig. 7(e)–(f),(a), where UDP capacity roughly grows with the number of UDP cores. We do not know where this scaling would start to taper off, and will follow up with experiments on the 100-core TILE-Gx when it becomes available.

The sub-linear scaling on x86 suggests there is room for improvement in the Memcached software even for commodity servers. This is not a novel claim [14]. But it drives the point that a different parallel architecture and execution model can scale much better.

### *E. Power*

Table I Shows the capacity of each system as we increase the amount of memory, CPUs, or nodes, as appropriate. It also shows the measured wall power drawn by each system while running at capacity throughput. Node-for-node, the TILEPro64 delivers higher performance than the x86-based servers at comparable power. But the S2Q server also aggregates some components over several logical servers to save power, such as: fans, BMC, and PSU. In a large data center environment with many Memcached servers, this feature can be very useful. Let us extrapolate these power and performance numbers to 256GB worth of data, the maximum amount in a single S2Q appliance (extrapolating further involves mere multiplication).

As a comparison basis, we could populate the x86-based servers with many more DIMMs (up to a theoretical 384GB in the Opteron's case, or twice that if using 16GB DIMMs). But there are two operational limitations that render this choice impractical. First, the throughput requirement of the server grows with the amount of data and can easily exceed the processor or network interface capacity in a single commodity server. Second, placing this much data in a single server is risky: all servers fail eventually, and rebuilding the KV store for so much data, key by key, is prohibitively slow. So in practice, we rarely place much more than 64GB of table data in a single failure domain. (In the S2Q case, CPUs, RAM, BMC, and NICs are independent at the 32GB level; motherboard are independent and hot-swappable at the 64GB level; and only the PSU is shared among 128GB worth of data.)

Table II shows power and performance results for these configurations. Not only is the S2Q capable of higher throughput per node than the x86-based servers, it also achieves it at lower power.

The TILEPro64 is limited, however, by the total amount of memory per node, which means we would need more nodes than x86-based ones to fill large data requirements. To compare to a full S2Q box with 256GB, we can an-

| Architecture | Nodes | Capacity | Power | TPS / Watt |
|---|---|---|---|---|
| TILEPro64 | 8 (1 S2Q) | 2,680,000 | 462 | 5,801 |
| Opteron | 4 | 660,000 | 484 | 1,363 |
| Xeon | 4 | 752,000 | 400 | 1,880 |

Table II: Extrapolated power and capacity to 256GB.

alyze a number of combinations of x86-based nodes that represent different performance and risk trade-offs. But if we are looking for the most efficient choice—in terms of throughput/Watt—then the best x86-based configurations in Table I have one socket with 64GB. Extrapolating these configurations to 256GB yields the performance in Table II.

Even compared to the most efficient Xeon configuration, the TILEPro shows a clear advantage in performance/Watt, and is still potentially twice as dense a solution in the rack (2U vs. 4U for 256GB).

## VI. Conclusions and Future Work

Low-power many-core processors are well suited to KV-store workloads with large amounts of data. Despite their low clock speeds, these architectures can perform on-par or better than comparably powered low-core-count x86 server processors. Our experiments show that a tuned version of Memcached on the 64-core Tilera TILEPro64 can yield at least 67% higher throughput than low-power x86 servers at comparable latency. When taking power and node integration into account as well, a TILEPro64-based S2Q server with 8 processors handles at least three times as many transactions per second per Watt as the x86-based servers with the same memory footprint.

The main reasons for this performance are the elimination or parallelization of serializing bottlenecks using the on-chip network; and the allocation of different cores to different functions such as kernel networking stack and application modules. This technique can be very useful across architectures, particularly as the number of cores increases. In our study, the TILEPro64 exhibits near-linear throughput scaling with the number of cores, up to 48 UDP cores. One interesting direction to take for future research would be to reevaluate performance and scalability on the upcoming 64-bit 100-core TILE-Gx processor, which supports 40 bits of physical address.

Another interesting direction is to transfer the core techniques learned in this study to other KV stores, port them to TILEPro64 and measure their performance. Similarly, we could try to apply the same model to x86 processors using multiple processes with their own table shard and no locks. But this would require a very fast communication mechanism (bypassing main memory) that does not use global serialization such as memory barriers.

## Acknowledgments

We would like to thank Pamela Adrian, Ihab Bishara, Giovanni Coglitore, and Victor Li for their help and support of this paper.

## References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2007, pp. 205–220.

[2] B. Fitzpatrick, "Distributed caching with Memcached," *Linux Journal*, vol. 2004, no. 124, p. 5, August 2004.

[3] J. Petrovic, "Using Memcached for Data Distribution in Industrial Environment," in *Proceedings of the 3rd International Conference on Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 368–372.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A Fast Array of Wimpy Nodes," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2009, pp. 1–14.

[5] K. Lim, P. Ranganathan, C. Jichuan, P. Chandrakant, M. Trevor, and R. Steven, "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments," *Proceedings of 35th International Symposium on Computer Architecture*, June 2008.

[6] W. Lang, J. M. Patel, and S. Shankar, "Wimpy Node Clusters: What about non-wimpy workloads?" in *Proceedings of the 6th International Workshop on Data Management on New Hardware*. New York, NY, USA: ACM, 2010, pp. 47–55.

[7] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," *SIGCOMM Computuer Communication Review*, vol. 39, no. 1, pp. 68–73, December 2008.

[8] B. Taylor, M., J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The RAW Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs." *IEEE Micro*, April 2002.

[9] B. Taylor, M., W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the RAW Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams." *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[10] S. Bell, B. Edwards, and J. Amann, "Tile64-processor: A 64-core SoC with Mesh Interconnect," *Solid-State Circuits*, pp. 88–598, 2008.

[11] A. Agarwal, "The Tile Processor: A 64-core Multicore for Embedded Processing," *Proceedings of 11th workshop on High Performance Embedded Computing*, 2007.

[12] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff, "Tile Processor: Embedded Multicore for Networking and Multimedia." *Proceedings of 19th Symposium on High Performance Chips*, August 2007.

[13] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web Caching with Consistent Hashing," in *Proceedings of the 8th International Conference on World Wide Web*, New York, NY, USA, 1999, pp. 1203–1213.

[14] N. Gunther, S. Subramanyam, and S. Parvu, "Hidden scalability gotchas in Memcached and friends," June 2010.