Predicting Memcache Throughput using Simulation and Modeling

Steven Hart, Eitan Frachtenberg, and Mateusz Berezecki
Facebook

Abstract

The current work introduces a method for predicting Memcached throughput on single-core and multi-core processors. The method is based on traces collected from a full system simulator running Memcached. A series of microarchtectural simulators consume these traces and the results are used to produce a CPI model composed of a baseline issue rate, cache miss rates, and branch mispredictions rate. Simple queueing models are used to produce througput predictions with accuracy in the range of 8% to 17%.

Keywords: Memcached, Performance prediction

1. INTRODUCTION

Key-Value Store in general, and Memcached in particular, is an important application at Facebook [12]. Many Memcached servers are used as a large-scale distributed memory cache for slow-to-compute values [13]. Memcached[1] 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 [15] 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. In this paper we focus on read operations (GET requests), because they are the dominant operation in Facebook's workload [13] and represents the focal point of throughput analysis.

This study is movitivated by Memcached's large impact on overall site performance. It is imperative that we understand the factors that affect Memcached performance, as well as develop predictive capabilities for the selection of future server architectures. Hardware performance counters are useful for measuring performance and identifying bottlenecks on machines that are available today; however, even well-established microprocessors may have unidentified flaws in their performance counter subsystems, and the reliability of the performance counters is even lower on pre-release systems. Further, hardware performance counters cannot provide much insight into the performance of our applications on future hardware products for which we have no physical sample of the machine. Performance modeling fills this gap. A model that predicts application performance can be used to predict the performance of systems that may be years from initial release. It is not necessary to have a physical machine in order to model performance.

The main contribution of this paper is a detailed description of the microarchitectural simulation-based methodology we developed to model Memcached performance, and can also be used for similar applications. We have implemented this model and use it to predict cycles-per-instruction (CPI) by composing component models of the baseline issue rate, cache miss rates, and branch mispredictions rate. CPI predictions then lead directly to throughput predictions on sequential and parallel architectures with a high degree of accuracy.

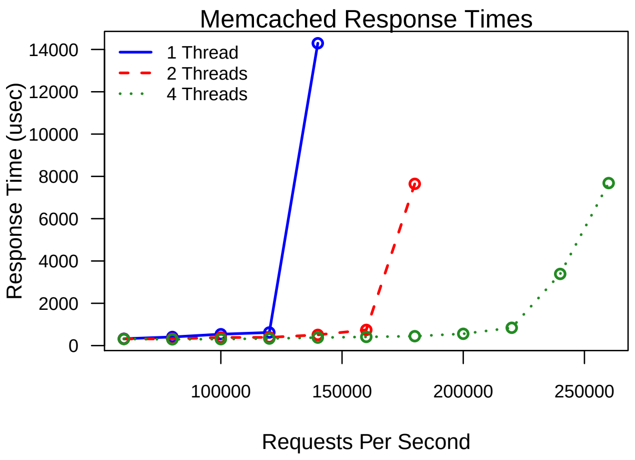

Figure 1. Intel Penryn Memcached Response Times

2. METHODOLOGY

Our first challenge in tackling Memcached performance modeling was how to measure thoughput in a way that allows comparison between different machines and among different levels of parallelism on the same machine. Figure 1 graphs the response times of Memcached running on an Intel Penryn machine under increasing levels of load. It illustrates that Memcached exhibits a clear saturation point after which the response time grows dramatically, and the load at this point is a good means of comparing Memcached performance.

[1] `http://Memcached.org/`

In order to estimate saturation point on a particular platform, the run time of a single Memcached query needs to be determined. Calculating the run time of a single transaction requires three pieces of information: the frequency of the microprocessor, the average number of instructions executed by a single Memcached transaction, and the average rate at which a microprocessor can commit a single instruction (CPI). The frequency of a microprocessor is obtained easily from its published specifications. However, obtaining the frequency is less straight forward for designs that incorporate some sort of dynamic frequency scaling (e.g., Intel's Turbo Boost [7] or AMD's APM [6]). The number of instructions required to service a Memcached transaction depends on the type and size of the transaction, the version of the operating system, and the version of Memcached along, with the versions of any libraries linked to the Memcached binary. In the context of the current study, Memcached transactions are all reads with eight-byte keys and 32-byte values, which are typical of values at Facebook. The Linux kernel version is 2.6.38, and the version of Memcached is 1.2.3h-dev.

The CPI for a microprocessor running Memcached is much harder to estimate, and the process of obtaining this value comprised the greatest portion of effort expended on this project. A microprocessor possesses no inherent CPI. Rather, the value is dependent on the microprocessor's microarchitectural characteristics, its frequency (which determines the latency in cycles of DRAM accesses), and the characteristics of the instructions it executes. We obtain a CPI estimate in three steps. First, a trace (a record of all the instructions executed, all memory locations accessed, and a subset of the processor's architectural state changes) is obtained by running Memcached on a simulated host. Next, the trace is used as input to several programs that simulate various aspects of the processor's microarchitecture. The product of this simulation is the frequency at which performance dependent-events occur (e.g., cache miss rates). Finally, a simple analytical model which assigns a latency in cycles to each of the events is used to produce a final CPI estimate.

We combine the estimate of the processor's CPI with the processor's frequency and a count of instructions per transaction to predict the time required the complete a single transaction. The transaction time is used to produce an estimate of Memcached server's saturation point throughput.

The remainder of this section explains the performance estimation methodology in detail.

2.1. Trace Generation

Simulating Memcached's workload requires knowledge of the instructions that the machine executes, along with the memory locations it accesses. Methods like dynamic instrumentation (e.g., Pin [5]) or a fast interpreter [3]) facilitate the collection of this information. Both techniques allow an execution trace to be collected quickly, but unfortunately, neither can collect traces of privileged code. Since privileged code (including the networking stack) can make up a significant portion of a server application's run time and performance, these tools are not appropriate in this case. The current effort uses full-system simulation to obtain instruction traces. To this end, we ran Memcached on AMD's SimNow [8] simulator, which fully simulates one or more microprocessors along with the rest of the system in software. As the entire system is simulated in software, the execution of privileged instructions, along with all changes to the privileged architectural state, can be accessed. It should be noted here that predictions for all platforms can be generated from a single trace: traces represent the code path that would be executed on any x86 microprocessor. Thus, it is necessary to collect new traces only when new workloads need to be studied.

2.1.1. System Simulation

The SimNow simulator must first be configured with the appropriate simulated hardware: the number of microprocessors and the amount of DRAM must be chosen. In order to run Memcached, We used a single 64-bit Opteron processor and two gigabytes of DRAM. The default network interface is an Intel gigabit Ethernet device, which matches our production Memcached configuration.

Linux and Memcached must next be installed on the SimNow simulator. The simulator is booted with a bootable ISO containing the CentOS 5.2 distribution of Linux. Once CentOS is installed, the Linux kernel is updated with Facebook's 2.6.38 production kernel. Lastly, Memcached is installed. Once all of the software components are configured, the resulting hard disk image is written out to the file system.

A second SimNow instance is configured to offer load to the simulated Memcached server, using a simple load generator [10]. The network that connects the simulated machine offering the load and the Memcached server is also simulated. The program that simulates the network (the *mediator*) includes a DHCP server that assigns IP addresses to the machines on the simulated network.

2.1.2. Trace Collection

Once all of the simulated systems are configured and communicating across the network, *mctester* can be run on the simulated client to offer load to the simulated Memcached server. A trace of Memcached's server state needs to be recorded and written to a file for separate analysis. SimNow provides a callback mechanism (using *analyzers*) to collect this trace file. We implemented a callback analyzer that collects state changes and generates a binary file that is further compressed using the *bzip2* library. The code to record and compress the trace is packaged as a library, which also included an interface for reading traces. The trace reader interface implements a callback for each of the trace records. Clients which wish to read the trace subscribe to the callbacks corresponding to the records that are of interest.

A new trace file is created every 10^{11} instructions. A large number of small trace files is useful for two reasons: First, not all trace files will capture a phase of Memcached's execution that is useful, and small trace files allow useless trace sections to be easily discarded. Second, any simulator which consumes trace files should be multithreaded for performance, and small trace files comprise a natural partitioning of the workload to the simulator's threads. A detailed discussion of multithreading is available in Section 2.2..

2.1.3. Trace Validation

Once a trace is collected, a series of analyses must be conducted to determine whether the trace is a valid proxy for the real-world workload. The goal is to produce a trace whose characteristics match those observed for Memcached running on real hardware.

The primary metric for determining a valid trace is the proportion of user to kernel activity. Our observations of actual Memcached use show that under load from *mctester*, kernel activity (mainly networking) makes up roughly 60% of the system time. It should follow that in the traces, a roughly similar mix of kernel to user instructions should be seen. One characteristic of Memcached using TCP is that when the machine is overloaded, the proportion of kernel instructions decreases dramatically to under 10%, because of excessive memory copying. Other tests include comparing cache misses and branch prediction rates measured on real hardware to those obtained via simulation.

The method of collecting and validating traces is not at all difficult. However, producing a set of valid traces is a matter of trial-and-error, and the process is time consuming. The largest difficulty we encountered was the time dilation between the simulated machine offering load and the simulated Memcached server. When SimNow is running without any *analyzers* installed, it employs a threaded code interpreter to speed up simulation, running at about 60- to 100-million instructions per second (MIPS). However, when an *analyzer* is loaded, SimNow must generate callbacks for events on an instruction-level granularity and must fall back on a much slower decode-and-dispatch method of simulation, slowing down the simulation by an order of 1000x

Since the client is running at ≈ 100 MIPS and the server is running at ≈ 0.1 MIPS, at what rate should load be offered to the server? The initial efforts at trace collection focused on scaling the offered load to the instruction execution rate of a server. In actual practice, however, this method did not yield fruitful results. The traces collected from this effort exhibited very high proportions of user instructions, suggesting that the simulated Memcached server was overloaded.

The alternative, and ultimately successful, route to trace collection discards the idea of a scaled rate altogether. This method is essentially a binary search of the offered load from one to 100 $\text{RPS}_{\text{client}}$, with the goal of the search being the highest $\text{RPS}_{\text{client}}$ that produces a valid trace. Using this method we settled on a trace collected at a rate of 12 $\text{RPS}_{\text{client}}$. The trace obtained by this method reasonably matched the validation criteria in all respects.

2.2. Simulation

A trace contains the record of the instructions that are executed on a generic x86 platform. Traces are consumed by simulators that are configured to represent specific implementations of the x86 architecture (e.g., an Atom, a Xeon, or an Opteron processor). Three major types of simulators are used to produce information that is then consumed by a performance model. Issue rate simulators (Sec. 2.2.1.) estimate the rate at which a processor can consume instructions. Cache simulators (Sec. 2.2.3.) predict miss rates at all the levels of a processor's memory hierarchy. Branch predictor simulators (Sec. 2.2.2.) predict the rate at which a processor mispredicts conditional branches.

2.2.1. Baseline Issue Rate

The issue rate simulator predicts the rate at which instructions flow through the processor, independent of all other stalling events. Issue rate depends on three factors: the native parallelism of the application executed on the processor, the size of the processor's scheduler, and the number of functional units (e.g., load/store units, branch units, ALUs) in the CPU. Prior work [2] has used a model that implements an infinite number of functional units. In this effort, however, we already know or can approximate the functional unit count for the product we wish to predict performance on.

The pipeline of a modern out-of-order processor can be roughly divided into two units: a front end that provides a stream of decoded instructions to a back-end that executes the instructions. At the head of the back-end pipeline is the scheduler. The scheduler stores some instructions and tracks the availability of each instruction's source operands. Out-of-order processors rename instructions' operands from a pool of physical registers in order to eliminate false dependencies that occur due to the limited number of architectural registers. When all of an instruction's source operands are available, it is eligible to be issued to a functional unit for execution. The number of functional units in a processor is limited, and each functional unit is specialized to perform a particular set of operations (e.g., a load, store, integer operations, or floating-point operations).

The issue rate simulator allows the size of the scheduler along with the number and type of functional units to be specified. It assumes infinite decode bandwidth and an infinite number of physical registers. All functional units are assumed to be single-cycle latency (if long-latency, nonpipelined operations are very frequent, their effects can be included in the CPI model described in Section 2.3.). The simulator functions much a like a processor. Instructions are decoded into micro-operations (uops), which are then renamed, stored into the scheduler, and issued as soon as possible.

Uops contain an `end_of_instruction` flag that indicates whether the uop completes an instruction, as is always set in cases where a single instruction translates to a single uop. In other cases, only the last uop's `end_of_instruction` flag is set. Each cycle, the simulator counts the number of uops whose `end_of_instruction` flags are set, and upon completion, uses this count to compute the average issue rate.

2.2.2. Branch Mispredicts

Branch mispredictions tend to be infrequent but incur a large penalty, so significant effort is expended to produce accurate branch predictors. Details about production branch predictors is almost nonexistent, so little time was devoted to producing a variety of branch predictors. Instead, we rely solely on a model of the gShare predictor, trying to match its size to published sources as closely as possible.

2.2.3. Cache Misses

For many server workloads, including Memcached, the processor's memory subsystem has a significant effect on overall performance. It is therefore necessary to model the memory subsystem with a reasonable level of fidelity to the simulated design. Unlike the issue rate and branch mispredicts, the miss rate of the memory subsystem is also impacted when the workload is run on two or more threads.

The memory subsystem consists of caches for instructions and/or data and TLBs. [4]. We developed a cache model that allows individual cache sizes and attributes to be specified and different cache organizations to be configured. For each cache, the user may choose the cache line size, the number of sets, the number of ways, and the replacement policy. Whether the cache is inclusive, exclusive, or a victim cache can be configured as well. Caches up to L3 can be instantiated with any sort of hierarchy. Instruction and data TLBs with up to two levels of hierarchy can be specified.

The cache model does not include a timing component; miss penalties are considered only when estimating overall performance (Sec. 2.3.). The output of the model is a rate of misses per 1,000 instructions (MPKI) for each cache and TLB in the hierarchy. The model differentiates between loads and stores, since store misses are rarely a source of exposed latency and are not considered when modeling performance.

The multiprocessor (MP) cache model is built upon the single processor model's framework, and implements a MESI cache coherence protocol [1]. It provides the same information for each cache as the single processor model. The challenge in simulating multiple threads of execution was not the cache model, but in obtaining multiprocessor traces from SimNow. Numerous attempts to obtain traces from two or more processors on SimNow never yielded a valid trace. Instead, the single processor Memcached trace was broken into multiple segments, and each segment was simulated on a different processor's cache. This technique is straightforward; however, it possesses one flaw that must be overcome. Because all of the traces really come from the same thread they share a stack pointer, and it is likely that stack data will ping-pong between the two processors' caches as stack data is read and written. To avoid this deleterious effect, we identify stack references and hash their physical addresses with a thread id.

2.3. CPI Estimation

We estimate the CPI for Memcached running on a particular processor using a spreadsheet that sums the cycle penalties of events that affect performance. The frequency of events is obtained from simulation studies discussed in Section 2.2.. The size of the penalty is derived from published sources, estimation, or by measurement on platforms that are physically available. The CPI estimate starts with a derived from the issue rate simulator. Penalties for branch mispredicts, cache misses, and misaligned loads are added to the baseline in order to produce the final estimate.

$$CPI_{\text{Total}} = CPI_{\text{Baseline}} + CPI_{\text{Memory}} + CPI_{\text{BranchMisp}} \quad (1)$$

2.3.1. Cache Misses

The frequency of misses to each line in the cache hierarchy is obtained from the cache model. Miss penalties are sometimes published, but are also easy to measure in available systems. LMbench [11] provides a tool called *lat_mem_read* which measures memory latency and TLB latency.

Another source of latency is loads that cross cache line boundaries (misaligned loads). Unlike other architectures, x86 supports accesses that are not aligned on their natural boundaries, including accesses that straddle two cache lines. The exact mechanism by which this is implemented is beyond the scope of this paper; however, performing a misaligned access imparts an additional performance penalty (e.g., 14 cycles on the Atom; 10 cycle on the Penryn Xeon).

2.3.2. Branch Misprediction

Branch misprediction rates are obtained from the branch prediction simulator. To understand the mispredict penalty of a conditional branch, it is useful to review the life of a mispredicted branch. A branch prediction is made at the head of the fetch pipeline. The branch predictor guesses whether the branch is *taken* or *not taken* based on some branch history information. Based on the prediction, the processor fetches instructions in the predicted direction. The branch reaches the scheduler, it must wait until all of it's dependent instructions have executed. Dependent operations clear the scheduler at a drain rate that is determined by the baseline IPC plus the IPC contributions of any blocking operations.

$$IPC_{\text{Scheduler Drain}} = IPC_{\text{Baseline}} + IPC_{\text{Blocking}} \tag{2}$$

If the scheduler size is known, the latency for the branch to execute is calculated as follows:

$$Cycles_{\text{SchedulerLatency}} = \frac{Scheduler\ Size}{IPC_{\text{Scheduler Drain}}} \tag{3}$$

For the purposes of simulation, a blocking operation is counted as a data TLB miss or misses to the L2 cache and lower in the cache hierarchy. When the branch is finally executed, its direction is resolved and the misprediction is detected. Dependent instructions in the scheduler and reorder buffer are killed and the fetch unit is signaled to restart fetching from the correct branch direction. The processor will wait for useful instructions to make their way down the fetch pipeline into the scheduler before execution can resume. Consequently, the value for the branch mispredict penalty is:

$$Cycles_{\text{BranchMisp}} = 2 \times Depth_{\text{FetchPipe}} + Cycles_{\text{SchedulerLatency}} \tag{4}$$

2.4. Estimating Throughput

Having obtained an overall CPI estimate, predicting Memcached throughput for a single thread is a relatively trivial matter of calculating the execution time for a single transaction. The execution time for a transaction is estimated using the CPI, the number of instructions in a single transaction, and the core clock frequency of the microprocessor:

$$TransactionTime_{\mu sec} = CPI \times InstrCount \times \frac{1}{Freq_{mhz}} \tag{5}$$

A count of 9,200 instructions per transactions was derived by counting instructions in Memcached traces, and was validated using performance counters on real hardware. Recalling the discussion in Section 2.1.3., Memcached has two distinct phases of program behavior. In the first phase, Memcached is able to keep pace with the offered load. When load approaches the saturation point, Memcached tansitions to a second phase of behavior in which the connection buffers begin to accumulate transactions awaiting service. In this later phase, Memcached's performance is progressively dominated by calls to *memmove* in the connection buffer handling code. Our goal here is to identify the request rate at which this phase change occurs: the point at which the request rate exceeds Memcached server's ability to sustain that rate.

Predicting multiprocessor performance is more involved, since two additional effects must be modeled: the effect of multiple threads on the performance of the processor microarchitecture and the effect of lock contention in Memcached. Of the microarchiterual features discussed in Sec. 2.2., only cache miss rates are impacted by a multiprocessor workload (the *penalty* of a branch mispredict is affected by the cache miss rate, but the actual rate of mispredicts is assumed to be unchanged by a multiprocessor workload). We employ the simulation technique from Sec. 2.2.3. to quantify this effect.

Memcached (version 1.2.3h-dev) employs a single mutex lock that serializes all accesses to its key-value store. As the number of Memcached threads is increased, contention for this mutex increases as well. Analyzing Memcached traces for code that occurs within the critical section reveals that only about 1.7% of instructions are executed inside a critical section. It cannot be assumed that, because the critical section instruction account for a small proportion overall, the performance of these instructions is equally small. Intuitively, if the instructions in the critical sections are responsible for accessing the key-value store, then these instructions may account for a larger proportion of the memory system performance than their small numbers suggest. In order to test this intuition, the cache simulator was modified to count misses only for accesses that occurred in critical sections. From these miss rates, separate CPI calculations can be made for the serial and parallel portions of the code. The results show that the critical sections, while accounting for only 1.7% of instructions, make up approximately 13% of execution cycles. This analysis is covered in more detail in Sec. 3.2..

3. EXPERIMENTAL EVALUATION

In this section, we apply the analysis techniques discussed in Sec. 2. to two different microarchitectures, Atom and Penryn. The Atom architecture is a low-power, in-order processor that yields impressive performance per Watt on Memcached [9]. The Penryn architecture is a server-class, out-of-order microprocessor that represents the opposite end of the spectrum from the Atom.

3.1. Single Processor Performance

3.1.1. Intel Atom

The Intel Atom N540 runs at a frequency of 1.86 GHz. It has 32 KB, eight-way set-associative instruction cache; a 24 KB, six-way data cache; and 512 KB L2 cache. All cache lines are 64 bytes long. The Atom has fully-associative, 16-entry L1 I- and D-TLBs, and four-way associative, 64-entry, L2 I- and D-TLBs. Other properties are shown in Table 1.

Table 2 shows the CPI-component model of the Atom. Baseline Cycles shows the number of cycles required to is-

Scheduler Size	32 entries
Fetch Pipe	6 stages
L2 Hit Latency	14 cycles
DRAM Latency	166 cycles
L1 TLB Miss	7 cycles
L2 TLB Miss	52 cycles
Branch Mispredict	75 cycles

Table 1. Atom Latencies

Component	Freq/1K Instr	Cycles/1K Instr
Baseline Cycles	1	917
L1 ICache Miss	39.2	545.6
L1 DCache Miss	14.8	188.0
L2 Instr Miss	0.2	38.9
L2 Data Miss	1.4	224.9
L1 ITLB Miss	11.7	82.1
L1 DTLB Miss	21.9	153.5
L2 ITLB Miss	5.3	277.7
L2 DTLB Miss	11.3	585.7
Branch Mispredicts	11.1	834.1
Total		3905.0

Table 2. Atom single-core CPI Components

Scheduler Size	32 entries
Fetch Pipe	8 stages
L2 Hit Latency	15 cycles
DRAM Latency	300 cycles
TLB Miss	7 cycles
TLB Miss	122 cycles
Branch Mispredict	58 cycles

Table 3. Penryn Latencies

Component	Freq/1K Instr	Cycles/1K Instr
Baseline Cycles	1	693
L1 ICache Miss	39.2	587.94
L1 DCache Miss	8.7	124.47
L2 Instr Miss	0.006	1.8
L2 Data Miss	0.45	133.8
L1 ITLB Miss	1.1	24.7
L1 DTLB Miss	21.9	153.5
L2 DTLB Miss	2.89	352.7
Misaligned Load	4.114	41.1
Branch Mispredicts	6.88	403.5
Total		2499.4

Table 4. Penryn single-core CPI Components

sue 1,000 instructions for an in-order machine. Cache miss rates per 1,000 instructions and their corresponding cycle estimates are broken down into each component of the memory system. Note that the branch misprediction rate is the largest component of the model. For the Atom, a branch miss is estimated to be 75 cycles. This large penalty is assessed because 1) the machine has a high baseline penalty because it must issue instructions in order and 2) as an in-order machine, L1 data cache misses are considered blocking events. The sum of cycles per 1,000 instruction is 3,905, or a CPI of 3.905. The time to execute a single Memcached GET transaction is 19.1 μsec, derived as follows:

$$19.1\mu sec = 3.9CPI \times 9,200Instructions \times \frac{1}{1860MHz}\mu sec \tag{6}$$

To obtain a throughput estimate, the Atom server is represented as an M/M/1 queue for which the service time is set to 19.1 μsec. From the queuing model, a maximum throughput of 58,000 requests per second (RPS) is obtained. The actual saturation point measured on the Atom is 51,000 RPS, so the estimated throughput is only 12.1% different from the measured throughput.

3.1.2. Intel Penryn

The Intel Penryn micro-architecture we used (L5420) runs at a core clock frequency of 2.5 GHz. L1 instruction and data caches are both 32KB, eight-way set-associative, with 64-byte lines. The 6MB, 16-way L2 cache is shared by two cores. Instruction addresses are cached in a 128 entry instruction TLB, while the L1 data TLB is has 16 entries and is backed by a 256 entry L2 TLB. Table 3 shows the relevant miss latencies. The CPI model for the Penryn is shown in Table 4. Notice that the Baseline Cycles component is over 200 cycles lower than that of the Atom due to the out-of-order design of the Penryn. The larger L2 cache results in much lower L2 miss rates, and a larger branch predictor results in a lower branch misprediction rate. The CPI model predicts an average instruction latency of 2.49 cycles per instruction, from which a transaction service time of 9.16 μsec is calculated. Plugging this service time into the queuing model gives an estimated throughput of 108,000 RPS. The observed throughput on a real Penryn machine gives a maximum single-core throughput of 128,000 RPS, a discrepancy of only 16%.

3.2. Multiprocessor Performance

These data show that reasonably accurate predictions of Memcached throughput can be obtained using a CPI model on both in-order and out-of-order machines. While these single core results are encouraging, the model must predict multicore performance as well to be useful. This section describes how the single-core model can be expanded to predict multicore performance. The multicore model is validated against the Penryn processor for both 2-core and 4-core workloads.

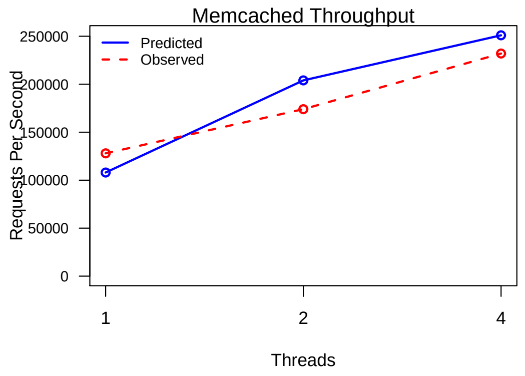

Figure 2. Penryn Scalability

Running Memcached on multiple cores affects the model's estimates of cache miss rates. Using our multiprocessor cache model (Sec. 2.2.3.), new miss rate estimates are generated for the 2-core and 4-core configurations. (Tables 5, 6). Based on these data, CPI is estimated to be 2.48 (8.13 μsec transaction time) and 3.18 (10.43 μsec) for 2 and 4 cores, respectively.

In Sec. 2.4. we learned that ≈ 13% of a transaction's run time is spent in code that is serialized by Memcached's key-value store mutex. This model takes a simple approach to modeling lock contention for Memcached. When running two threads, 87% of each thread's code can be run in parallel with the other thread, while the remaining 13% must be run sequentially. To model the lock contention, we assume the worst case in which every time a thread wishes to obtain the key-value store mutex, it is contending with the second thread for the lock. This assumption adds a 13% performance penalty to the run time of each thread when modeling the dual-thread case. Similarly, when modeling the four-thread case, the worst case scenario is again assumed: when any one thread is trying to obtain the lock, it is contending with the other three threads–incurring a 39% runtime pentalty.

When running two threads, the run time of a single transaction is calculated as 7.47 μsec (based on the CPI estimate from Table 5). A syncronization penalty of 13% is added to the run time, yielding a final run time of 8.14 μsec. An M/M/2 queue is used to model the throughput, resulting in an estimated throughput of 204,000 RPS. The throughput limit of the actual machine is measured as 174,000 RPS, which places the estimate whithin 17% of the measured throughput.

The four-core case is modeled similarly. Based on an estimated CPI of 3.06, the time to complete a single transaction on the Penryn is calculated as 11.26 μsec. Using the worst-case throughput assumption, we assumed that any one thread is contendending with three others. Based on this assumption, a 39% run time pentalty is added to the transaction, producing a final run time estimate of 15.65 μsec. Modeling the four-core machine as an M/M/4 queue in which the service time is 15.65 μsec, an estimated throughput of 251,000 RPS is

Component	Freq/1K Instr	Cycles/1K Instr
Baseline Cycles	1	693
L1 ICache Miss	19.3	288.9
L1 DCache Miss	12.12	162.9
L2 Instr Miss	0.044	13.2
L2 Data Miss	1.261	378.3
L1 ITLB Miss	1.2	27.1
L1 DTLB Miss	21.6	151.5
L1 DTLB Miss	1.9	233.5
Misaligned Load	4.1	41.1
Branch Mispredicts	6.88	375.1
Total		2031.4

Table 5. Penryn 2-core CPI Components

Component	Freq/1K Instr	Cycles/1K Instr
Baseline Cycles	1	693
L1 ICache Miss	25.0	375.0
L1 DCache Miss	2.95	248.1
L2 Instr Miss	0.00	0.0
L2 Data Miss	2.95	885.0
L1 ITLB Miss	0.99	22.9
L1 DTLB Miss	16.3	114.1
L1 DTLB Miss	1.43	175.3
Misaligned Load	4.1	41.1
Branch Mispredicts	6.88	521.2
Total		3059.1

Table 6. Penryn 4-core CPI Components

obtained. On real hardware, Memcached throughput is measured at 232,000 RPS, making this estimate off by only 8%.

4. DISCUSSION AND FUTURE WORK

The results in Section 3. suggest that a CPI component model and a simple concurrency model can predict Memcached performance on both in-order and out-of-order microprocessors in single- and multi-core configurations. Using trace-based simulation means that the long, error-prone job of tuning and validating the simulated workload needs to be performed only once for a particular application. Once traces have been collected, any processor configuration can be simulated using the same set of traces.

Although the throughput predicted by the model is a reasonably good match to the observed performance, there are opportunities to improve the accuracy of the results. The CPI model tends to underestimate performance. As we mentioned earlier, prefetching has a small (≈ 5%) benefit for Memcached performance, and this likely accounts for some of the performance difference. Another possible source of error is that the model does not account for memory-level parallelism (MLP) [16]. The model considers all lower-level cache misses as blocking and counts the full penalty of the miss. In reality, modern microprocessors have some capacity of parallelize misses, with the degree of parallelism depending on the processor's microarchitecture and the characterisitics of the workload. Incorporating a technique models MLP would likely increase the accuracy of the current model [17].

An interesting result of our analysis is that average GET requests only require $\approx 9,200$ instructions to complete from start to finish. This low number corroborates previous claims that Memcached is highly suitable to running on low-power low-speed processors with sacrificing responsiveness [13, 9]. We plan to investigate this suitability further as low-power architectures mature into server applications.

Expanding this modeling technique to other workloads such as HipHop [14] will require more sophisticated modeling techniques in some areas. Although prefetching has a small effect on Memcache performance, other workloads are sensitive to prefetching, so some work in this area would be necessary. Branch prediction is modeled very simply, but as evidenced from the CPI component models, branch misprediction penalties can be a large component of overall CPI. The model would certainly benefit from more choices of branch prediction schemes and more knowledge of exactly what branch predictors are implemented in real hardware products. The simple concurrency model used here appears to be perfectly adequate for a workload that has a single contended lock. However, an application with more locks would certainly require a more sophisticated scheme for modeling concurrency. Finally, this work avoids modeling hardware that implements multithreading. As this feature is present in many products available today, hardware threading should be the subject of future research in this area.

REFERENCES

[1] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel computer Architecture: A Hardware/Sofware Approach*. Morgan Kaufmann Publishers, 1999.

[2] T.S. Karkhanis and J.E. Smith, *A First-Order Superscalar Processor Model*. International Symposium on Computer Archtecture (ISCA), 2004.

[3] B. Cmelik and D. Keppel, *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. 1994.

[4] J. Hennessey and D. Patterson, *Computer Architecture, A Quantitative Approach, 5th Ed.*. Morgan Kaufmann Publishers, 2011.

[5] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood. *Pin: Building Customized Program Anlalysis Tools with Dynamic Instrumentation.* Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005, pp. 190-200.

[6] AMD Corporation. *Software Optimization Guide for AMD Family 15h Processors*. `http://support.amd.com/us/Processor_TechDocs/47414.pdf`.

[7] Intel Corporation. *Intel ® Turbo Boost Technology 2.0: Performance on Demand.* `http://www.intel.com/technology/turboboost/index.htm`

[8] AMD. *AMD SimNow^{TM}Simulator.* `http://developer.amd.com/tools/simnow/pages/default.aspx`.

[9] D.G. Andersen, J. Franklin, A. Phanishayee, and L. Tan. *FAWN: A fast Array of Wimpy Nodes*. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, 2009.

[10] M. Berezecki *mctester*. mateuszb@fb.com.

[11] L McVoy and C. Staelin. *LMbench - Tools for Performance Analysis*. `http://www.bitmover.com/lmbench/`.

[12] R. Nishtala, M. Paleczny, R. McElroy, H. Fugal, T. Tung, M. Kwiatkowski, V. Venkatramani. *Memcached Leases: Protocol Extensions to Improve Cache Consistency and Prevent Thundering Herds*, To Appear, 2012.

[13] M. Berezecki, E. Frachteberg, M. Paleczny, K. Steele. *Performance and Efficiency Evaluation of Memcached on the TILEPro64 Processor* Sustainable Computing, To Appear 2012.

[14] H. Zhao, *HipHop for PHP: Move Fast* `https://www.facebook.com/note.php?note_id=280583813919&id=9445547199`

[15] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, Y. Yerushalmi. *Web Caching with Consistent Hashing*, Proceedings of the Eight International Conference on the World Wide Web, 1999.

[16] Y. Chou, B. Fahs, S. Abraham. *Microarchitecture Optimizations for Expoiting Memory-Level Parallelsim*, Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004.

[17] X. Chen and T. Aamodt. *Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs*, Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, 2008.