

A Server Performance Model for Static Web Workloads

K. Kant and C.R.M. Sundaram
Server Architecture Lab
Intel Corporation
{kkant|schintha}@co.intel.com

Abstract

This paper describes a queuing network model for a multiprocessor system running a static web workload such as SPECweb96. The model includes architectural details of the web server in terms of multilevel cache hierarchy, processor bus, memory pipeline, PCI-bus based I/O subsystem, and bypass I/O-memory path for DMA transfers. The model is based on detailed measurements from a *baseline* system and a few of its variants. The model operates at the web-transaction level, and does not explicitly model the CPU core or the caching hierarchy. Yet, the model predicts the performance impact of low level features such as number of processors, processor speeds, cache sizes and latencies, memory latencies, higher level caches, sector prefetching, etc. The model shows an excellent match with measured results. Because of many features that are difficult to handle analytically, the default solution technique is simulation. However, the paper also proposes a simple hybrid approach that can significantly speed up the solution without affecting the accuracy appreciably. The model has also been extended to handle clusters of symmetric multiprocessor systems with both centralized and distributed memories.

1 Introduction

This paper describes a queuing network model for a multiprocessor system running a static web workload. The model includes architectural details of the web server in terms of multilevel cache hierarchy, processor bus, memory pipeline, PCI-bus based I/O subsystem, and bypass I/O-memory path for DMA transfers. The multiprocessor configurations covered are (a) bus-based symmetric multiprocessor (SMP), (b) cluster of multiple SMP nodes with snoop filtering, and (c) distributed memory clusters using crossbars and point to point links. The model currently handles only HTTP 1.0 GET requests for static contents and has no explicit notion of embedded requests. Both native and proxy server modeling is

supported, and so is incomplete caching of the static file-set. Logging of HTTP requests is modeled explicitly.

The model could be used to predict the performance of static web workloads, such as SPECweb96 benchmark, on a variety of web server configurations with different processor speeds, cache size and latencies, chipset features, I/O subsystems, and memory sizes. SPECweb96 [2] is a simple benchmark to evaluate the performance of static GET requests on web servers. It specifies a static set of “directories” stored on a web server, each containing 36 files of predefined sizes. The primary performance metric is completed requests per second or ops/sec. Because of a static file-set, all directories are typically cached unless the memory requirements exceed maximum supportable memory. Further information on SPECweb96, along with a simple technique to determine its bandwidth requirements is discussed in [8].

The model operates at the web-transaction level, and does not explicitly model the CPU core or the caching hierarchy. Yet, to be useful, the model must be able to predict the impact of very low level features such as number of processors, cache size and latency, memory access latency, coherence traffic, etc. To handle this task, the crux of the model is in taking the low level parameters (e.g., cache miss ratios, memory transactions per web transaction, etc.) from measurements of a baseline system, and scaling them appropriately to apply to the model at hand (the current model). In some cases, the current model has features that are not present in the baseline system (e.g., level-3 cache, larger cacheline size, snoop filter, deferred transactions, inadequate file cache to store the entire file set, etc.), yet the model must be able to account for them. In the absence of data from a baseline system, these aspects are dealt with in a heuristic manner.

The model is actually more general than needed for SPECweb96. In addition to admitting more general request size, response size, and sleep time distributions, it also includes the capability of forwarding the request to another web-server for the requested “file”. That is, the model encompasses both a native server and a proxy server. In the following, these aspects of the model have

been pointed out where relevant. Also, while the architectural details in the model are rooted in Intel based web-servers, it is easy to modify them to correspond to other traditional architectures.

The outline of the rest of the paper is as follows. Section 2 describes the queuing model for an SMP system in terms of the resources and activities that are explicitly represented in the model. Section 3 discusses some issues in model calibration and solution.

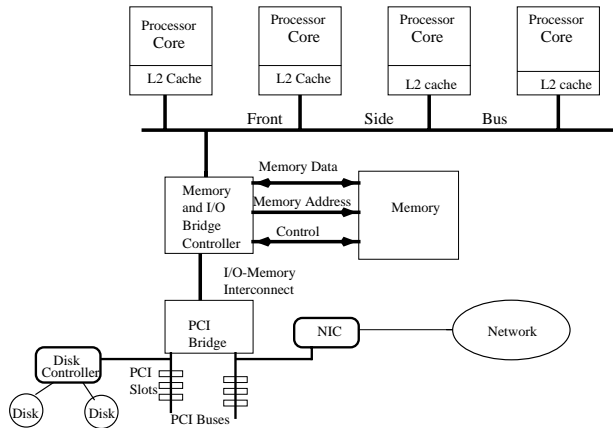


Figure 1: Overall server architecture

2 Basic Queuing Network Model

2.1 Architecture of Web Server

Figure 1 shows a highlevel diagram of the system architecture that is assumed in this study. As the figure shows, a number of processors (each with its own L1 and L2 caches) interconnected through a front side bus are connected to NIC's, memory, and I/O subsystem through a bunch of chipset components (MIOC, I/O-memory interconnect, and PCI-bridge). The FSB is assumed to be a pipelined snooping bus that enforces the transactions to complete in the same order in which they were initiated. It is assumed that the cache coherence among the processors is implemented through the popular MESI (Modify-Exclusive-Shared-Invalidate) coherence protocol [5]. Such a protocol results in two additional sources of traffic on the processor bus:

1. Invalidation of potential entries in other caches when a processor attempts to gain exclusive access over data currently in shared mode in its cache.
2. Attempt by a processor to access data in modified state in another processor's cache. This results in

cache-to-cache data transfer and writing of modified data to memory (called implicit writeback, in order distinguish it from capacity related eviction of modified data known as explicit writeback).

The MIOC (memory and I/O controller) accepts data from the host processors, to be routed either to memory or to one of the I/O devices through PCI-bus, I/O-memory interconnect, and PCI-bridge. It accepts data from the main memory, to be routed to either one of the host processors or to one of the I/O devices. Also, it accepts data from one of the I/O devices to be routed to either one of the host processors or to the main memory. MIOC is also responsible for generating the appropriate control signals to control the data transfer to and from the memory. As implemented in most popular chipsets, there is a bound on the number of read or write transactions that can be outstanding.

2.2 Queuing Network Model

The model operates at the level of web transactions (GET requests). The model is driven by a set of identical client machines, each of which models a fixed number of client processes. Each client process generates a transaction that goes to the server and eventually returns with the server response. The process then sleeps for some time, and repeats the process. Because of space constraints, the model will be sketched here only briefly. In most instances of performance projection, it is assumed that the I/O subsystem (both network and disks) have adequate capacity as needed, and the objective is to determine the maximum throughput that the processor-memory subsystem can provide. For this reason, the I/O part of the model is rather simplistic. In particular, we represent NICs via an array of delays stations and the network via a single-server (with appropriate modeling of collisions in case of half-duplex ethernet). By default, it is assumed that a network send is a zero-copy operation, but network receives require single kernel to user space copy. Both send and receive involve CPU usage to represent network stack processing and interrupt handling.

Client behavior is simply represented by a delay station in series with a single-server. The delay station represents the sleep time T_{sleep} , which for SPECweb96 is given by $\frac{N_c N_p}{2\lambda_d} + 0.0005$, where N_c is the number of client machines, N_p the number of processes per client, and λ_d is the design throughput.

The I/O transfers over the PCI bus are modeled at the level of PCI bursts in order to account for the impact of limited queue slots in the PCI bridge and retries of in-bound read operations. The interconnect between memory and I/O controller (MIOC) and PCI bridge, if any,

is also represented explicitly, but only via a single-server whose service time is computed by accounting for the appropriate protocol overhead. The data transfer over the PCI bus to the PCI-bridge and from PCI-bridge to the memory are modeled separately (along with the implicit assumption of an appropriate flow-control between the two transfers).

Writing of HTTP logs is modeled explicitly. Data is accumulated in memory buffers until some K ($=16$) disk blocks worth of data has accumulated. The disk write demon then writes these blocks as a single sequential operation. The disk subsystem consists of a delay station (representing seek/rotational delays) in series with a multi-server (representing RAID transfer time). Disk write involves CPU usage (for caching, file system handling, DMA setup, etc.) and one memory to memory copy (HTTP buffers to I/O buffer space). The model includes the capability of reading requested files from the disk if the available memory is inadequate to fully cache the entire file-set. Disk reads occur in units of single disk blocks, and involve one memory to memory copy (I/O buffer space to file cache). For speed reasons, file cache is not simulated explicitly; instead, the amount of data to be retrieved from the disk in cases of partial caching is calculated during the calibration phase.

The front-side bus (FSB) is represented by two single-server queuing stations, ABUS (for address bus) and DBUS (for data bus). The effect of bus cycles when the data bus cannot be used due to conflicts (i.e., *dead cycles*) is accounted for indirectly in computing DBUS service time. Bus arbitration is not modeled explicitly; instead, a single delay station *post-L2* with a fixed delay represents the duration from L2-miss until the end of bus arbitration.

Memory is modeled as a 4-stage pipeline. For memory reads, the 4 stages are: mem1 (address decode), mem2 (RAS), mem3(CAS and data read), and mem4 (pushing data onto DBUS). The same four stages are used for memory writes, except for some changes in their interpretations. The stations mem1, mem2, and mem4 are modeled as pure delay type, and thus lumped into a single station in the model called “othr”. mem3 has as many servers as the number of parallel memory channels (or channels groups) capable of producing full cachelines in parallel. From a modeling perspective, the memory bandwidth is controlled by mem3, and no-load latency by “othr”.

Each CPU in an SMP system is represented separately as a single-server queuing station, with CPU service time representing the time until the next memory transaction issue. That is, execution out of processor caches is not explicitly modeled. Similarly, the internals of the CPU

(e.g., parallel execution units) are not modeled explicitly; instead, the impact of architectural details of the core enters into the model only via the core *cycles per instruction* (CPI) as discussed later.

The maximum number of pending bus transactions is modeled in a somewhat simplistic manner. In particular, no per-processor limitation on the number of memory transactions is enforced, since the chip-set in-order queue (IOQ) limitation is usually dominant. The current model is structured as follows: Only the memory reads are placed in the IOQ, which is called the read token queue (RTQ) in the model. Memory writes directly go to a write token queue (WTQ), which represents the pending memory write buffer. In reality, a memory write can go to the write buffer only after any reads from the same location that are already in progress have completed.¹ The model uses RTQ/WTQ in such a way so as to accurately model the CPU stalls when actual transaction queues are full. In particular, the model forces all transactions to go to memory pipeline directly from the CPU node, so that if a transaction completing at the CPU is unable to get adequate tokens, it immediately freezes further processing by that CPU.

In an SMP system, each memory transaction must go through a “snoop phase” to determine if the requested cacheline is held by another processor in modified state (known as a “HITM” condition). In case of a HITM, the modified line must be written back to the memory (known as *implicit writeback*) and also provided to the requesting processor (i.e., cache to cache transfer over the bus). In the model, HITM condition follows a Bernoulli trial based on the computed HITM probability.

Certain chipsets do *sector prefetching*, i.e., automatically fetch the second part of double-sized cacheline on a miss in the highest cache level. This is represented in the model via background traffic *bg_traf*, whose routing through the memory/bus subsystem is identical to that of regular memory transactions.

In queuing theoretic terms, each client process is a customer belonging to a distinct *category* (or *chain*) that goes through various *service stations* of the model. Each category involves several *phases*, each corresponding to the major activities in transaction processing such as request receive, computation, disk read/write, response send, etc., as well as coherency related bus traffic such as invalidations and writing back of modified cache data to the memory. Most phases, in turn, involve looping a few times to accomplish the task. For example, sending a file out is done one packet at a time. We identify each

¹If split transactions are supported, the IOQ itself doesn’t hold the transaction for very long; instead, the ordering must be enforced on the responses, which means that the deferred queue is the critical queue to model.

pass as a *subphase*. Furthermore, certain subphases involve multiple *passes* through the memory pipeline (e.g., packet reception at the socket layer followed by kernel to user space copying). If subphases are defined purely from operational considerations, we run into the difficulty that the service times in the memory pipeline could be very different for various phases, whereas in reality, most bus transactions transfer only one cacheline and hence would have almost constant service times. This problem is a result of trying to model very low-level features (e.g., FSB/memory transfers) in a high level (e.g., web transaction level) model. We handle this difficulty by considering memory references in chunks of certain number of cachelines. The number of loop iterations through the memory pipeline is then set appropriately.

The basic queuing model has been extended to include both split bus transactions and support for SMP clusters, however space does not permit a full discussion of these features. The most important aspect of the cluster model is the estimation of snooping/invalidation frequency for the remote busses and the rate of “back invalidations”, i.e., bus invalidations forced due to conflicts in the snoop filter. The model also supports both centralized memory (i.e., a multiported memory shared among all nodes of the cluster) and distributed memory (i.e., per node memory accessible from other nodes as well). In the case of distributed memory, up to 3 nodes may be involved in a memory transaction (e.g., node1 requesting access to a cacheline belonging to node2 which is cached in modified state by node3). The most difficult part of these extensions is the enumeration of all potential paths and computation of their delays during the initialization phase, so that impact of various features on the average number of cycles to complete an instruction (CPI) can be computed.

3 Model Calibration and Solution

The model is calibrated based on the detailed CPU counter and timer measurements and data from the performance monitoring tools of the operating system. This data is obtained both for the “baseline system” and a few variants thereof (e.g., baseline case with different cache-sizes, cache latencies, and number of processors). The baseline data is used to determine the following per op parameters (a) kernel and user CPU time, (b) memory reads, invalidations, and implicit/explicit writebacks, (c) cache references at L1 and L2 levels (code fetches, code fetch misses, data reads, data read misses, writes, write misses), and (d) overall CPI (cycles/instruction), overall MPI (misses/instruction), read memory latency, and path length (instructions/op).

These parameter values are then converted to apply to the modeled (or “current”) system based on a fairly complex set of calculations that use baseline and current system parameters and a number of scaling factors derived from the baseline variants. The major steps in these calculations are

1. Scaling of cache miss ratios from baseline to current model. This scaling accounts for three possible differences between the baseline and current systems: (a) different throughput, which affects the working set size, (b) different cache size, which affects capacity misses, and (3) different number of processors, which affects coherence misses. The same basic scaling applies to all caching levels.
2. Path length estimation. Path lengths are again scaled from baseline to the current system. Two effects are accounted for here: (a) spin-lock (or multiprocessor) penalty as a function of number of processors, and (b) effect of different IOs/op in the current system.
3. Scaling of cache reference rates, and computation of MPIs in the current model for all cache levels (L1, L2, and L3, if any). This is done using the baseline references, baseline miss ratios, current miss ratios determined above, and new path length computed above.
4. Estimation of overall CPI disregarding the queuing and blocking delays (since those delays are represented explicitly in the model). This provides scaling of per op CPU service times. The overall CPI is obtained using the “core CPI” (defined as the CPI under the assumption of infinite L1 cache), MPIs, and “blocking factors” (defined as the fraction of non-core access latency that is visible for execution within the core under low-load conditions). The fundamental assumptions here are that both the core CPI and blocking factors remain constant for a given architectural lineage.
5. Scaling with respect to cacheline size, and estimation of impact of sector prefetching, if any. This is done by obtaining a crude estimate of the locality of access for the workload. If address traces are available, this step can also be done using a cache simulator.
6. Estimation of coherence traffic (local bus invalidations, local implicit/explicit writebacks, remote snoops, remote invalidations, remote writebacks, back invalidations, etc.). This is done using a Markovian model of the MESI protocol discussed in cite [7].

7. Estimation of memory references and service times per op for each phase and at each service station of the model.

The space does not permit a detailed discussion of these aspects. The details can be found in [9].

Following the calibration, the entire model is a closed queuing network with following features: (a) general service times, (b) blocking due to limited queue sizes for memory and PCI operations, (c) multiple chains and classes, (d) priority scheduling of chip-set transactions, etc. These features rule out exact analytic solution; therefore, the default solution method is simulation following the initialization phase that determines station service times and various routing probabilities. Simulation is very expensive even for a single run; unfortunately, multiple runs are typically needed. The problem is that most steps in the initialization phase depend on the *design op-count*, which is an input parameter to the model. The corresponding *achieved op-count* is determined by solving the model, and must be very close to (and ideally no larger than) the design op-count. This naturally leads to an iterative process, which is made even harder because of the deficiencies of the simulation package used (SES workbench).

In order to minimize simulation iteration, a quick analytic solution is obtained assuming separability, where the queuing delays are based on open M/G/1 formula, and blocking delays are based on an Erlang-B type of formula. Except when resources other than CPU are the primary bottleneck, this usually gives throughputs within $\pm 5\%$. Based on such a solution, the model automatically determines the starting op count for simulation.

For a more elaborate analytic solution, the most difficult but indispensable aspect of the model is blocking due to limited sizes of RTQ and WTQ. (In contrast, although priority scheduling of chip-set transactions is difficult to handle, its impact is small and can be ignored.) Several techniques available in the literature were tried for this, starting with the well known mean value analysis (MVA) based techniques (see [6], chap 8). Unfortunately, the accuracy of these techniques was rather poor, which precluded an all-analytic solution. Consequently, analytic solution of only the FBUS-memory subsystem is considered here. In this regard, we propose a MVA based analytic approximation to obtain a flow-equivalent server approximation for the memory-bus subsystem [6]. The rest of the model is still solved by simulation. This approach cuts down simulation time by a factor of 2-3, while retaining good accuracy.

The analytic solution treats FBUS and memory portions of the model as a closed queueing network model, and computes throughputs as a function of number of

transactions present in FBUS-memory subsystem. This is done for various classes of traffic such as reads and writes initiated by some of the I/O devices or NIC's over the PCI-bus, or reads, writes, and invalidations initiated by host processors. Subsequently, the FBUS-memory subsystem is replaced by a multi-chain flow-equivalent server [6]. Blocking delays experienced by a transaction to acquire tokens at chipset RTQ (Read Token Queue) and WTQ (Write Token Queue) and at the processor are estimated through simulation.

3.1 Model Validation

The model validation has two facets: (a) comparison between simulation model results and measurements, and (b) comparison between the pure simulation and hybrid model results. Let us first concentrate on (a). For a faithful validation, the measurements included here were actually obtained by a different group of people with a different setup than those used in calibrating the model. Table 1 presents relative throughputs obtained through measurement, simulation, and hybrid simulation/analytic techniques for various configurations. The given values are relative to the design throughput for that case. The listed processor speeds are relative to a 300 MHz processor.

It is seen that the simulation throughput and analytical throughput matches measured throughput fairly well except for the 300 MHz, 1P case where the measured results are a bit old and the system was not quite as well optimized as for the other cases. It is important to note in this regard that both simulated and measured values contain uncertainties that would make a few percent difference inconsequential. On the measurement side, we often found that the best achievable throughput over a 10-15 second interval is not sustainable over longer periods, and small changes in the setup related to the NIC, driver, TCP/IP, file-caching, interrupt handling, etc. can alter throughput in the range of a few percent or more. More substantial configuration changes (e.g., a different version of web-server, enabling of 9 KB ethernet frames, etc.) can have a much more substantial impact on the performance. Since the measured values are used for model calibration, this uncertainty is transferred to the model as well. Furthermore, simulations necessarily include uncertainties related to the randomness in the model. In view of this, the difference of a few percent is really "noise" and perhaps doesn't have any significance.

Next, we compare results in columns "simulated" against "analytic" in Table 1. It is seen that the match is excellent in all cases. The "analytic" case, however, requires less than 1/2 of the simulation time currently in spite of on-the-fly flow-equivalent aggregation that does

Table 1: Comparison between Simulation, Measurement, and Analytic Results

Rel. speed	No of procs	L2 cache size	Simulation		Relative Throughput		
			CPU util	tot CPI	Simulated	Measured	Analytic
1.00	1	512	99.71	3.10	1.025	1.114	0.996
1.00	1	1024	99.88	2.99	0.999	1.075	0.996
1.00	1	2048	99.99	2.99	1.000	1.057	0.996
1.33	1	512	99.99	3.19	0.991	1.036	0.975
1.33	1	1024	100.0	3.12	0.987	1.078	0.976
1.33	1	2048	99.99	3.08	0.988	1.009	0.970
1.67	1	512	99.91	3.43	1.005	1.046	1.013
1.67	1	1024	99.79	3.29	0.986	1.010	0.992
1.67	1	2048	99.94	3.26	0.999	0.987	0.980
1.00	2	1024	99.34	3.41	1.001	1.048	1.017
1.67	2	2048	98.98	3.87	0.992	1.023	1.005
1.67	3	2048	97.46	4.13	1.000	0.955	1.007
1.00	4	1024	96.55	3.73	1.014	1.029	1.026
1.00	4	2048	96.76	3.64	1.018	1.014	1.021
1.33	4	512	97.16	4.34	1.034	1.048	1.041
1.33	4	1024	96.89	4.06	0.967	0.962	0.982
1.33	4	2048	96.53	3.94	1.017	1.024	1.031
1.67	4	512	96.29	4.81	1.019	1.010	1.020
1.67	4	1024	96.33	4.46	1.019	1.026	1.028
1.67	4	2048	96.37	4.25	1.012	1.005	1.019

not retain any results for future use. With a more sophisticated implementation that avoids repeated computation for the same population level, it should be possible to cut-down the time even further.

References

- [1] "PCI System Architecture", Third Edition, Mindshare Inc., 1995.
- [2] "An explanation of the SPECweb96 benchmark", available at www.specbench.org/osg/web96.
- [3] "SES Workbench", Scientific and Engineering Software Inc., Austin, TX.
- [4] A. Agrawal, M. Horowitz, and J. Hennessy, "An Analytical Cache Model", ACM transactions on Computer Systems, Vol 7, No 2, May 1989, pp 184-215.
- [5] D.E. Culler and J.P. Singh, *Parallel Computer Architecture – A hardware/software approach*, Morgan Kaufmann, 1999.
- [6] K. Kant, "Introduction to Computer System Performance Modeling", McGraw Hill 1992.
- [7] K. Kant, "Estimation of Invalidation and Writeback Rates in Multiple Processor Systems", Submitted for publication.
- [8] K. Kant and Y. Won, "Server Capacity Planning for Web Traffic Workload", IEEE transactions on knowledge and data engineering, Oct 1999.
- [9] K. Kant and S. Chinthamani, "A Server Performance Model for Static Web Workloads", Technical Report, Jan 2000.