Predicting the Effect of Memory Contention in Multi-Core Computers Using Analytic Performance Models

Shouvik Bardhan and Daniel A. Menascé, *Fellow, IEEE*

Abstract—Analyzing and predicting the performance of applications that run on multi-core computers is essential. This paper demonstrates experimentally that memory contention resulting from multiple cores accessing shared memory resources can become a significant component (i.e., over 50 percent) of an application's execution time. The paper develops single- and multi-class analytic performance models for predicting the effect of memory contention on a job's execution time. The models consider local and remote memory as in NUMA architectures. Model validation was done using a micro-benchmark and programs from HBench, UnixBench, and SPECCPU2006 running on machines with 4, 12, and 16 cores. The paper shows how to derive the model parameters and demonstrates that there is a significant difference in predicted values when memory contention is ignored. For example, a model that ignores memory contention predicts an average execution time about four times smaller than the experimental value for a concurrency level of 18 while the model with memory contention predicts a value that is 90 percent of the experimental value for the same concurrency level.

Index Terms—Multi-core computers, analytic models, Mean Value Analysis, experimentation, queuing theory

1 Introduction

Scaling up the speed of servers and personal computers by increasing the clock speed of their CPUs became ineffective around 10 or so years ago. Increased heat due to current leakage and other manufacturing-related issues caused the top clock speed to flatten out at around 4 GHz. Chip designers decided instead to increase the number of processing elements on a chip. Two or more CPU cores in a single chip allow the peak performance of CPUs to follow Moore's Law. However, the memory subsystem now had to support the increased number of accesses generated by the multiple cores. Thus, memory access became the bottleneck and prevented the peak speed of the CPU cores to be achieved. The two main sources of performance inefficiency in a multi-core architecture are: (1) The overhead of cache-coherency management. (2) Memory bandwidth, memory latency and contention for memory access among cores [7].

This paper significantly extends a short paper by the same authors that presented a preliminary single-class model for multi-core computers [2]. The main contributions of this paper are: (1) a method to quantify the slowdown due to access to shared memory bus and (2) simple and efficient predictive approximate single and multi-class analytic queuing models that consider contention due to shared memory elements (e.g., shared caches, bus, and memory banks) on Symmetric Multi Processing (SMP) and Non Uniform Memory Access (NUMA) multi-core machines.

• *The authors are with the Department of Computer Science, George Mason University, Fairfax, VA 22030. E-mail: {sbardhan, menasce}@gmu.edu.*

Manuscript received 20 Nov. 2013; revised 2 Sept. 2014; accepted 1 Oct. 2014. Date of publication 2 Oct. 2014; date of current version 10 July 2015. Recommended for acceptance by R. F. DeMara. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2014.2361511

Several papers [28], [29], [30] studied the effects of memory contention on multi-processor systems. Our work differs from previous work in that it provides a simpler model whose parameters are easy to obtain in practice and allows for performance prediction of multiple applications of different kind (i.e., the multiple class case). Our model was validated with experimentation on several multi-core machines. Other approaches to assessing the performance characteristics of multi-core systems include simulation [16] and benchmarking [17]. More related work is covered in Section 8.

The rest of this paper is organized as follows. Section 2 shows experimental results that demonstrate the performance impact of contention for shared memory bus in multi-core computers. The next section presents a single-class analytic model used to predict the performance of applications running on SMP and NUMA multi-core machines. Section 4 describes how memory access time can be measured. The following section presents results that validate the single-class model with experimental data. Section 6 extends the model to multiple classes of jobs and Section 7 discusses the experimental validation of the multi-class model. The next section discusses related work and Section 9 presents concluding remarks and future work.

2 Effects of Memory Contention on Multi-Core Computers

Modern CPU cores have complex memory hierarchies [9]. Level 1 (L1) cache (sometimes separately for data and instructions) is the closest to the CPU and has the fastest access, L2 comes after L1 away from the CPU and there may even be an L3 cache. Programs that execute on multi-core machines compete for access to shared memory

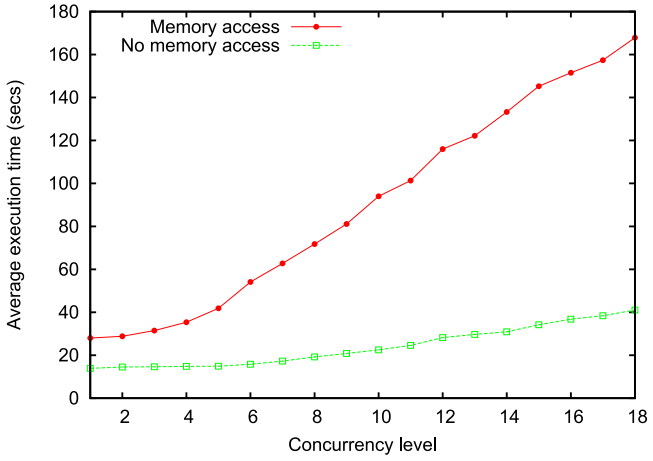


Fig. 1. Effect of memory contention on a 12-core Westmere EP machine.

components (e.g., data banks, front side bus, and shared caches). This contention generates a queuing effect that cannot be ignored when predicting the performance of concurrent applications running on such machines. The effect of contention for access to memory can be illustrated in Fig. 1, which depicts the average execution time versus the concurrency level (i.e., number of concurrent programs in execution) for two versions of a C program running on a 12-core Intel Westmere-EP machine. Similar results were observed for the same programs on a four-core Intel Sandy Bridge machine.

The characteristics of the two versions of the C program are summarized in rows 1, 2, 4, and 5 of Table 1. For the Westmere-EP machine case (depicted in Fig. 1), the program performs 700 allocations and deallocations of 50-MB chunks of main memory. In one case (bottom curve of the graph), the allocated memory is never accessed by the program. In the other (top curve), each allocated memory address is accessed once by the program. In all cases, the programs do not execute any disk I/O and perform sequential memory access. We noted an even more dramatic effect of memory contention on the execution times of concurrently running programs when memory access is random.

The graphs in Fig. 1 show that contention for memory plays a significant role on a program's execution time in multi-core machines. See also [12]. For the 12-core machine, when there is only one copy of the program in execution (i.e., concurrency level equal to one), the difference in execution time is 14.1 sec, which can be attributed to accessing 7 GB of main memory. As the concurrency level increases from 1 to 12, the difference in execution time increases to 87.8 sec. That means that an additional 73.7 sec (i.e., 87.8 - 14.1) are incurred due to mainly memory (and possibly other) contention.

After the concurrency level exceeds the number of cores, the difference in execution time continues to increase due to contention for cores. For example, for a concurrency level of 18, the difference in average execution time is 126 sec, of which 14.1 sec are due to memory access time, 73.7 sec due to contention for memory access, and 80.1 sec due to contention for cores. Thus, over 40 percent of the execution time can be attributed to memory contention. For a four-core Sandy Bridge machine, more than 16 percent of the execution time is due to memory contention (graph not shown).

Therefore, models that predict the execution time of applications in multi-core machines must take memory contention into account. These models are presented and validated in the rest of this paper.

3 A Two-Layer Analytic Model For Multi-Core Computers

This section presents a simple single class (i.e., all jobs are equivalent in terms of their resource consumption) queuing network (QN) based model that captures memory contention. Section 6 extends the model to multiple classes.

3.1 Basic Model

Readers are assumed to be familiar with queuing network models [15]. We consider that there are m cores and n instances of a job concurrently being executed (i.e., the single class case). We extend the model to the case in which there are concurrent instances of different job types (i.e., the multiclass case) later in the paper. We consider that jobs use the CPU and perform I/O and are not bound to any of the m cores; they are assigned to any available core when dispatched by the operating system.

Fig. 2 shows a pictorial description of our analytic queuing model of multi-core systems. The model has two layers. The bottom one is the memory contention model and consists of three devices:

TABLE 1
Specification of the Experiments Ran Using the Micro Benchmark (MBench)

	Architecture	Code Parameters	Amount of Memory	Disk I/O	Concurrency	PMU Collected
1	Sandy Bridge	Memory Allocation	700 allocations of 10 MB	None	1 to 8	None
2	Sandy Bridge	Memory Allocation and Access	700 allocations of 10 MB	None	1 to 8	Backend stall cycles for concurrency 1
3	Sandy Bridge	Memory Allocation, Access, and I/O	700 allocations of 10 MB	several hundred MB file R/W with cold disk cache	1 to 8	Backend stall cycles for concurrency 1
4	Westmere-EP	Memory Allocation	700 allocations of 50 MB	None	1 to 18	None
5	Westmere-EP	Memory Allocation and Access	700 allocations of 50 MB	None	1 to 18	Backend stall cycles for concurrency 1
6	Westmere-EP	Memory Allocation, Access, and I/O	700 allocations of 50 MB	several hundred MB file R/W with cold disk cache	1 to 18	Backend stall cycles for concurrency 1

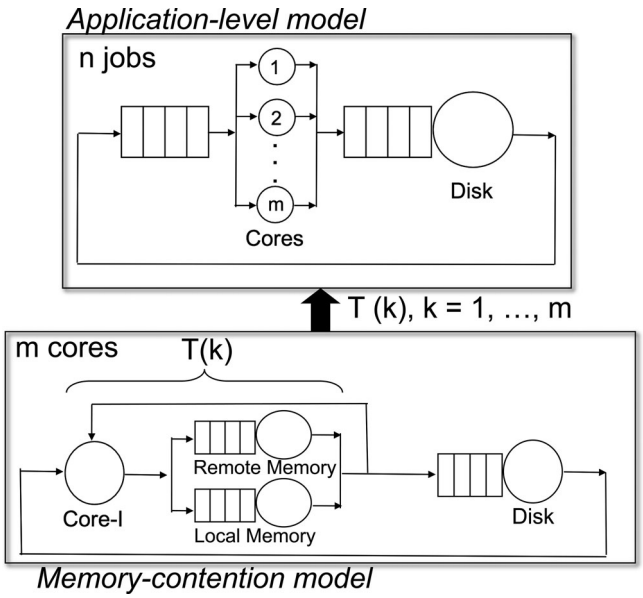


Fig. 2. Two-layer analytic model of a multi-core computer.

- A delay device called *Core-I* that represents the total time, $D_{\text{Core-I}}$, spent by a core executing instructions while not waiting for memory access (for data or instructions). We call $D_{\text{Core-I}}$ the service demand at the *Core-I* device.
- A load independent queuing device called *Memory* that represents the system memory elements shared by all cores (e.g., L3 cache, memory bus, and memory banks). For now let us consider only one combined memory queue. The service demand at this device is denoted by D_{Mem}. The contention for memory (i.e., queuing for this device) is caused by the fact that several cores are concurrently executing instructions that require access to memory. For the basic model, only the local memory of Fig. 2 should be considered (see Section 3.2 for the NUMA extension).
- A load independent queuing device that represents I/O on a disk. Let D_d be the service demand on the disk.

The memory contention model is a closed queuing network model with a population equal to k, where $k = 1, \ldots, m$. If there are $n > m$ jobs in execution, there cannot be more than m active cores (i.e., executing instructions or waiting for memory). This model can be solved using Mean Value Analysis (MVA) [18] and yields as a result the value $T(k) = D_{\text{Core-I}} + R'_{\text{Mem}}(k)$ for $k = 1, \ldots, m$, where $R'_{\text{Mem}}(k)$ is the memory residence time (i.e., total time waiting to access memory + total time accessing memory). $T(k)$ is then the average time spent by a job either using a core, waiting to access memory, or accessing memory, when there are k jobs in execution concurrently.

The service demands for *Core-I*, *Memory*, and *Disk* are computed by running a single instance of the application multiple times to obtain average values. The disk service demand is computed as follows using the Service Demand and Utilization Laws [15]:

$$D_d = \frac{X_d \times S_d}{X_0(1)}, \tag{1}$$

where X_d, S_d, and $X_0(1)$ are the disk throughput (in I/Os per second), average disk service time (in seconds), and average system throughput for a concurrency level of one, respectively. The values of X_d and S_d can be easily obtained by running `iostat` or any equivalent utility on a Unix/Linux system. $X_0(1)$ is just the ratio between the number of instances of the program executed by the duration of the experiment.

The values of $D_{\text{Core-I}}$ and D_{Mem} are computed as follows. First, as we run the program with a concurrency level of one, we must collect the average job elapsed time, $E(1)$, and the percent of "stalled cycles backend," *PercStalledBackendCycles*, which can be obtained by running `perf` as described in Section 4. PercStalledBackendCycles is the percentage of clock cycles that cannot be used by the processor because it is waiting for instructions or data to come from the memory subsystem using a shared bus.

Since with a concurrency level of one there is no I/O contention, we can write that

$$T(1) = D_{\text{Core-I}} + D_{\text{Mem}} = E(1) - D_d. \tag{2}$$

We can also write that

$$D_{\text{Mem}} = T(1) \times \text{PercStalledBackendCycles}. \tag{3}$$

Combining Eqs. (2) and (3) we get

$$D_{\text{Core-I}} = (E(1) - D_d) \times (1 - \text{PercStalledBackendCycles}). \tag{4}$$

The values of $T(k)(k = 1, \ldots, m)$, are then used in the application-level model. We now provide a detailed explanation of the algorithm used to solve the combined memory-contention and application level models (see pseudo code in Algorithm 1). Consider the following notation:

- $R'_i(k)$: residence time, i.e., total time spent by a job at device i (waiting to use or using the device) when the concurrency level is equal to k.
- $\overline{n}_i(k)$: average number of jobs at device i when the concurrency level is equal to k. This includes jobs waiting to use the device and using the device.
- $R_0(k)$: application response time, in seconds, when the concurrency level is equal to k.
- $X_0(k)$: throughput, in jobs/sec, when the concurrency level is equal to k.

The goal of the algorithm is to compute the execution time and the throughput of a program when the concurrency level is equal to n. Lines 1 to 12 in Algorithm 1 use regular single-class MVA equations [18] to solve the memory-contention level model. As a result, one obtains $T(m)$, $R'_d(m)$, $R_0(m)$, and $X_0(m)$. The solution of the application level model starts at line 15. The concurrency level varies from $m + 1$ to n. The solution of this model follows the single class MVA equations with a few modifications described in what follows. Line 18 is the regular MVA equation for the residence time at the disk. Note that when $j = m + 1$, this equation needs $\overline{n}_d(m)$ obtained in the memory-contention model when $k = m$. Line 19 uses the Bard-Schweitzer (B-S) approximation [19] to compute the average queue length, $\overline{n}_d(j)$, at the disk. If the disk demand is greater than zero (line 20), the job throughput $X_0(j)$ is computed as $\overline{n}_d(j)/R'_d(j)$ by combining

Little's Law [14] with the Forced Flow Law [15]. Otherwise, $X_0(j)$ is computed in line 22 as follows.

Algorithm 1. Multi-Core Analytic Model - Single Class

```
    Input: m, n, D_d, D_Core-I, D_Mem
    /* Compute T(k), k = 1, ..., m */
    Initialize: n̄_d(0) ← 0; n̄_Mem(0) ← 0
    for k = 1 → m do
5:      R'_d(k) ← D_d[1 + n̄_d(k − 1)]
        R'_Mem(k) ← D_Mem[1 + n̄_Mem(k − 1)]
        T(k) ← D_Core-I + R'_Mem(k)
        X_0(k) ← k/[T(k) + R'_d(k)]
        n̄_d(k) ← X_0(k)R'_d(k)
10:     n̄_Mem(k) ← X_0(k)R'_Mem(k)
    end for
    R_0(m) ← T(m) + R'_d(m)
    /* Set R and X in case n = m */
    R ← R_0(m); X ← X_0(m)
15: /* Compute the execution time R and throughput X
         for the application */
    for j = m + 1 → n do
        R'_d(j) ← D_d[1 + n̄_d(j − 1)]
        n̄_d(j) ← (j/(j−1)) × n̄_d(j − 1) /* B-S approximation */
        if D_d > 0 then
20:         X_0(j) ← n̄_d(j)/R'_d(j)
        else
            X_0(j) ← m/T(m) /* See Eq. (6) */
        end if
        n̄_CPU(j) ← j − n̄_d(j)
25:     R'_CPU(j) ← n̄_CPU(j)/X_0(j)
        R_0(j) ← R'_CPU(j) + R'_d(j)
    end for
    R ← R_0(n); X ← X_0(n)
```

According to Little's Law [14], the average number, $\overline{n}_{\text{Core}}(j)$, of jobs using one of the m cores, is equal to $X_0(j) \times T(m)$ for $j = m+1, \ldots, n$. But, when $j > m$, all m cores are busy all the time and $\overline{n}_{\text{Core}}(j) = m$. Thus,

$$\overline{n}_{\text{Core}}(j) = m = X_0(j) \times T(m) \tag{5}$$

which implies that

$$X_0(j) = \frac{m}{T(m)} \tag{6}$$

for the case in which there is no (or negligible) I/O.

In line 24, the average queue length at the CPU can be obtained by subtracting the average queue length at the disk from the concurrency level. Then, in line 25, Little's Law and the Forced Flow Law are used to compute the residence time at the CPU. The response time $R_0(j)$ is just the sum of the residence times at the CPU and disk (line 26).

It is instructive to note that if we were to model situations in which processes are bound to cores, we would only need to modify the application-level model as follows. Instead of having a single queue for all the cores, we would need to have a dedicated queue in front of each core. The resulting application-level queuing network is a standard queuing network model that can be solved using well known methods and does not require the Bard-Schweitzer approximation explained in the preceding paragraphs.

This section presented a two-level single-class algorithm that considers the time spent at the cores, waiting to use the cores (contention for cores), accessing memory, waiting to access memory (memory contention), performing I/O, and waiting to access I/O devices (I/O contention). While we only showed one disk in our model, it would be trivial to consider any number of disks by adding the corresponding MVA residence time and queue length equations, as would be recognized by those who are familiar with queuing network models.

While we have not considered burstiness in memory access, Eager et al. developed and validated a model that uses Approximate MVA to deal with this situation [33].

3.2 Non Uniform Memory Access Extension

The NUMA architecture was designed to mitigate the scalability limits of the Symmetric Multi Processing architecture [9]. With SMP, all memory accesses are performed on the same shared memory bus. This performs well for a small number of CPU cores, but congestion on the shared bus limits its scalability on machines with dozens of CPU cores, common on modern machines. NUMA alleviates these bottlenecks by limiting the number of CPU cores on any one memory bus, and connecting the various nodes by means of a high speed interconnect. In a NUMA architecture, memory access time depends on the location of the memory relative to the processor core. As an example, the NUMA machine used for conducting the experiments for this paper has 12 CPU cores and 8 GB of memory equally divided between two nodes of 4 GB of memory each. For six out of the 12 cores, 4 GB of memory are considered local and the other 4 GB remote.

To extend our basic model to NUMA architectures, we need to consider both memory queues shown in Fig. 2: local and remote. These two load independent devices have service demands that we denote by $D_{\text{Mem-Local}}$ and $D_{\text{Mem-Remote}}$ and can be written as $D_{\text{Mem-Local}} = N_{\text{L}} \times T_{\text{L}}$ and $D_{\text{Mem-Remote}} = N_{\text{R}} \times T_{\text{R}}$, where N_{L}, T_{L}, N_{R}, and T_{R} denote the average number of accesses to local memory, access time to local memory, number of accesses to remote memory, and access time to remote memory, respectively. Let us define the following ratios: $K_N = N_{\text{L}}/N_{\text{R}}$ and $K_T = T_{\text{R}}/T_{\text{L}}$. Thus, $D_{\text{Mem-Local}}/D_{\text{Mem-Remote}} = K_N/K_T$.

A process can potentially have its data segment either on a local or a remote memory node. Despite the long history of Linux, NUMA awareness is a relatively new addition to Linux kernels. Starting from 2.6.X, the Linux kernel has adapted to larger memory and number of cores. The latest Linux kernels make every attempt to keep a process on the node where its memory is initially allocated. Memory intensive programs, when migrated between cores can have its data on both local and remote memory segments. Linux uses the "local node first" scheduling policy; our experiments have shown that on a fully utilized machine, a process is scheduled to a non-local node between 15 to 25 percent of the time. Remote memory latency is substantially higher than local memory latency (i.e., $K_T \gg 1$).

TABLE 2
Specifications of the Intel Sandy Bridge and Westmere-EP Machines Used in the Experiments

System	Intel Sandy Bridge	Intel Westmere-EP
Processor	I5-2520M	XEON X5660
Family/Model	6/42	6/44
Speed	2.5 GHZ	2.8 GHz
Memory	6 GB	48 GB
Cores	4	12 and 16
NUMA	No	Yes
Operating System	Ubuntu 12.04	RHEL 2.6.32
On-chip cache	L2-2 x 256 KB, L3-3 MB	L2-6 x 256 KB, L3-12 MB

Our measurements of the NUMA effect using a test program and also the `numactl` utility showed a slowdown of approximately four times when the accessed memory was remote to the executing core (i.e., $K_T = 4$). Based on the observations above regarding Linux scheduling, we considered that a program executes 20 percent of the time on a remote node (i.e., the number of accesses to local memory is four times the number of accesses to a remote memory, or equivalently $K_N = 4$). Then $K_N/K_T = 4/4 = 1$ and $D_{\text{Mem-Local}} = D_{\text{Mem-Remote}}$. Table 5 shows the parameters obtained by running several programs on a NUMA-based Intel XEON machine. The columns represent the parameters for two different sets of experiments consisting of three classes of jobs.

4 Measuring Memory Access Time

We conducted experiments on Intel machines with two different architectures: an Intel Sandy Bridge Micro-Architecture based system (four cores) and an Intel Westmere-EP Micro-Architecture based system (12 and 16 cores). See Table 2 for their characteristics.

We developed a micro-benchmark (MBench) in C and used it to gather experimental data (see pseudo code in Algorithm 2). Others have developed micro-benchmarks for similar reasons [11]. The program parameters determine if the code actually accesses the allocated memory and whether it performs disk I/O. This allowed us to control and quantify the behavior of the code and measure the performance effect of memory access and I/O. See Table 1 for the various combinations of experiments with the micro-benchmark.

Algorithm 2. Micro-Benchmark (MBench) Pseudo Code

```
Input: memAccessRequired, diskWriteRequired
Input: MEM_SIZE, REPEAT
for i = 0 → REPEAT do
  if diskWriteRequired then
5:   performDiskIO;
  end if
  byte * ptr_mem ← malloc(MEM_SIZE)
  hold_ptr ← ptr_mem;
  for i = 0 → MEM_SIZE do
10:   Access the byte pointed to by ptr_mem
    if memAccessRequired then
      ptr_mem++
    end if
  end for
15:  free hold_ptr
 end for
 printTimingInfo;
```

Our goal was to estimate the memory service demand, D_{Mem}, i.e., the amount of time spent by a job accessing memory when not competing with other jobs for shared memory elements. To that end, we chose to measure the time taken by a program to access memory with the help of the hardware based Performance Monitoring Unit (PMU) built into modern Intel processors (AMD and Alpha processors have similar functionality; however, we have not used these in the experiments discussed in this paper). The hardware counters available inside the CPU provide a very precise picture of CPU resource utilization and there are literally hundreds of events to choose from [10], [13].

Eranian [5] gives a very detailed description of Intel processor performance counters and indicates that these counters are the key to understanding issues with the memory subsystem. He shows how to collect memory-related metrics from today's hardware using the performance counters with the help of the Linux tool `perfmon2` that his team developed. His paper also discusses the nuances of Non Uniform Memory Access based machines.

Our experiments are based on a Linux utility called `perf` [23], based on `perfmon2`, that allows us to read the PMU counters. Fig. 3 shows a representative output for a `perf stat` command. We concentrate on the line containing the counter "stalled-cycles-backend" and note that a total of 27.33 percent of the total cycles were due to stalls for this particular execution. Back end stalls have two main sources: memory sub-system stalls and execution stalls.

```
Performance counter stats for './MBM YES NO 7':

      36483.335874 task-clock                #    0.998 CPUs utilized
               43 context-switches          #    0.000 M/sec
                0 CPU-migrations            #    0.000 M/sec
          476,999 page-faults               #    0.013 M/sec
  112,341,248,111 cycles                    #    3.079 GHz                      [83.33%]
   72,222,333,153 stalled-cycles-frontend   #   64.29% frontend cycles idle     [83.33%]
   30,700,100,241 stalled-cycles-backend    #   27.33% backend  cycles idle     [66.67%]
  114,989,289,020 instructions              #    1.02  insns per cycle
                                            #    0.63  stalled cycles per insn [83.34%]
   14,576,388,567 branches                  #  399.536 M/sec                    [83.33%]
          952,160 branch-misses             #    0.01% of all branches          [83.33%]

       36.554921783 seconds time elapsed
```

Fig. 3. Output of measurements with `perf`.

TABLE 3
Input Parameters for the MBench Models Obtained by Running the Programs on the Intel Sandy Bridge and Westmere-EP Machines

Parameter	Disk Access		No Disk Access	
	Intel Sandy Bridge	Intel Westmere-EP	Intel Sandy Bridge	Intel Westmere-EP
$D_{\text{Core-I}}$	10.94 sec	20.16 sec	26.26 sec	19.70 sec
D_{Mem}	8.67 sec	8.88 sec	11.8 sec	8.31 sec
PercStalledBackendCycles	34%	30%	31%	30%
D_d	8.93 sec	0.8 sec	0	0

These resource stalls happen at the back-end of the pipeline when a resource cannot be allocated and the cost to access memory is a big part of the cost incurred. Similarly, there are quite a few events such as L1 instruction cache miss and instruction buffer full, that contribute to front end stall cycles. Instructions that involve Load and Store from the L1, L2, L3 caches or from memory result in wasted cycles. The CPU of modern processors contain specific hardware counters to measure the number of wasted cycles. We have considered the total stalls value as an indicator of the time spent performing memory access and this value is reflected as part of the "stalled-cycles-backend" metric. It is to be noted that the amount of prefetching over the entire execution of the program is captured by the value of the metric 'PercStalledBackendCycles'. Even though a program may exhibit different memory access patterns during the lifetime of its execution, our model considers the average value during the execution of the program. Our experimental results confirm that such an approach yields accurate results that can be obtained by following a simple and feasible measurement procedure.

The most important registers and counters on Intel processor microarchitectures are kept consistent across all architectures (we used Sandy Bridge and Westemere-EP). That makes it easy to use a simple C-language structure to setup the event select register used by the Linux tool `perf`. For the backend stalled cycles, the value of interest is an event value of 0xb1 and umask value of 0x3f in that structure [10].

We execute our micro-benchmark program `mbm` with parameters specified in `[params]` using the command `perf stat ./mbm [params]` which gives us the stall values and various other predetermined events for this program for a job concurrency level of 1. The value of backend stall cycles provided in the output is used in determining the time spent by the program to access memory (see Eq. (3)). The next section shows the results of using the model presented in the previous section to predict the application performance for higher concurrency levels.

5 Single Class Results

This section reports the results obtained by running MBench with various parameters (memory allocation with memory access and no disk access, memory allocation and access and disk access), on the Sandy Bridge and Westmere-EP machines as described in Table 1 (rows 2, 3, 5, and 6). We also present the results of running programs from well-known Linux benchmarks (Hbench [25] and UnixBench [27]) as well as from SPECCPU2006. The results show that considering memory contention in the model significantly improves its accuracy.

MBench always allocates a big chunk of memory. However, an input parameter to the program determines if it accesses the memory or simply allocates and frees it. This way, we are able to see the effect of cache misses and DRAM access. It is very evident from the results that the portion of the time performing memory access (and the eventual contention experienced by the program due to shared memory elements) is appreciably higher when the program actively accesses the allocated memory.

The input parameters obtained by running the programs are provided in Table 3. The first two columns represent the case of memory and disk access and the last two columns contain input parameters for the case of no disk access.

All results depicted in the figures that follow in this paper show three graphs: (1) Experimental results obtained by running real applications on multi-core machines. These curves represent average values for several runs and are accompanied by 95 percent confidence interval bars. (2) Results obtained through the model that considers memory contention, as presented in this paper. (3) Results obtained by using a queuing network model such as the application-level model at the top of Fig. 2. Such a model only considers contention for cores but does not capture contention for memory. The solution to the model without memory contention is obtained by using MVA (for the single-class case) and AMVA (for the multi-class case) [15] in addition with Seidmann's approximation [20] for multi-server queues (representing the multiple cores in this case). Moreover, at the top of each graph we indicate what contention elements are present, which program was used in the experiment, and the number of cores of the machine used. For example, the top of Fig. 4 displays "CPU/Memory contention - MicroBench (four cores)" to indicate that there is only contention for cores and memory but no disk activity and that our microbenchmark was ran on a four-core machine. The top of Fig. 6 displays "CPU/Memory/Disk contention - MicroBench (4cores)" to indicate that the experiments also include disk activity.

Fig. 4 shows three curves for the Sandy Bridge experiments with no disk access and memory access. These curves show the average execution time of the program vs. the concurrency level. The lower curve represents the results obtained with an analytic model that does not consider memory contention. Because this model does not consider memory contention and there is no disk contention, the average execution time does not vary until the concurrency

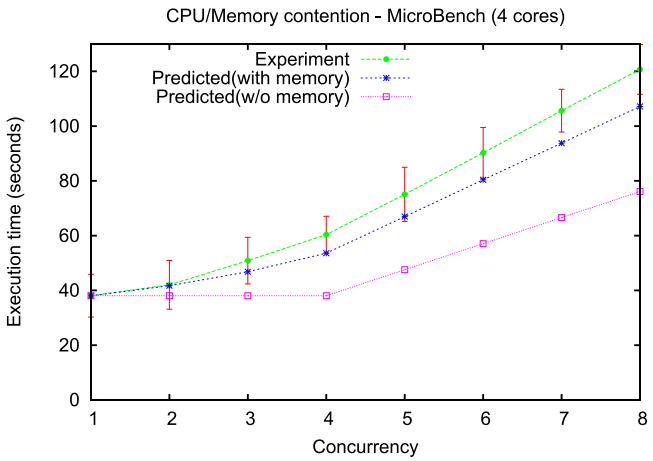


Fig. 4. Average execution time vs. concurrency level for MBench without disk I/O on Sandy Bridge (4 cores).

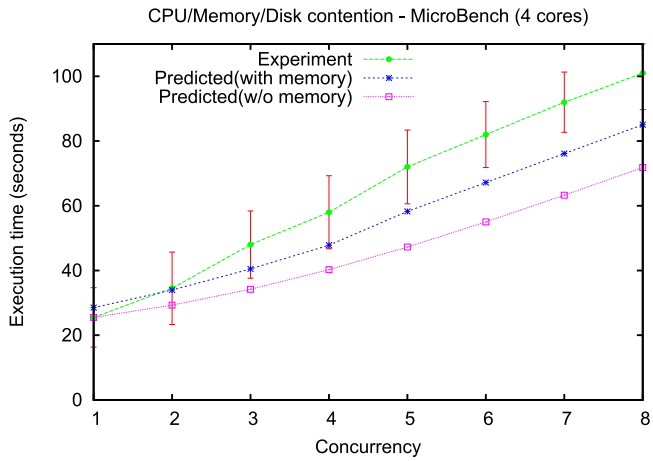


Fig. 6. Average execution time vs. concurrency level for MBench with disk I/O on Sandy Bridge (4 cores).

level exceeds the number of cores (4 in this case). After that point, the execution time increases due to contention for cores. The curve above that shows the results obtained with our proposed model that considers memory contention. As it can be seen, the execution time increases from concurrency 1 to 4 due to memory contention and continues to increase after that due to core contention. The curve above that shows the results obtained from running experiments. As it can be seen, our model tracks very closely the experimental results. For example, for a concurrency level equal to 4, the relative percent error (i.e., 100 × (experiment - model) / experiment) between the experimental results and the model with memory contention is 11.2 percent, while the percent relative error between the experimental results and the model that does not consider memory contention is 37 percent for that concurrency level. For a concurrency level of 8, the model with memory contention predicts an average execution time that is 89 percent of the experimental value while the model without memory contention predicts a value that is 63 percent of the experimental value.

Similar results can be observed for the experiments carried out with the Westmere-EP machine as illustrated in Fig. 5. As it can be seen, the model with memory contention tracks very closely the experimental results and exhibits a maximum absolute percent relative error of 14.1 percent for all

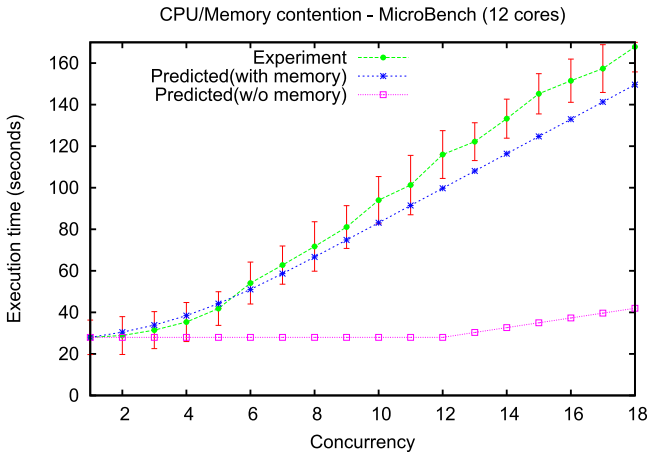


Fig. 5. Average execution time vs. concurrency level for MBench without disk I/O on Westmere (12 cores) .

concurrency levels. The model that does not consider memory contention exhibits very large errors. For example, for a concurrency level of 18, the model without memory contention predicts an average execution time that is only 25 percent of the experimental value, while the model with memory contention predicts a value that is 90 percent of the experimental value. The errors in the case of the Westmere-EP machine (as compared to Sandy Bridge) are more pronounced because in the Westmere-EP case we have 12 cores competing for memory and the program in this case accesses 35 GB of memory as opposed to 7 GB. On the Westmere-EP machine, we experimented with and without hyper-threading enabled and the results are statistically equivalent.

The next two figures illustrate a situation where the program also performs I/O. These situations correspond to rows 3 and 6 in Table 1 for the Sandy Bridge and Westmere-EP machines, respectively. Fig. 6 shows three curves for the Sandy Bridge machine: experiment (top), model with memory contention (middle) and model without memory contention (bottom). Again, the model with memory contention tracks the experimental results much better. In this case, since there is disk contention, the model without memory contention shows an increase in the average response time even in the range from 1 to 4. However, the model with memory contention also predicts memory contention besides disk contention. The percent relative error for a concurrency level of 4 for the memory contention model is 17.6 percent while that of the model without memory contention is 30.6 percent.

The situation for the Westmere-EP case depicted in Fig. 7 shows a very good correlation for the model with memory contention: a 12.5 percent percent relative error at concurrency level of 12 and 14 percent for concurrency level equal to 18. The model without memory contention exhibits a percent relative error of 71 percent percent at a concurrency level of 12 and an error of 76 percent at a concurrency level of 18. This means that the model without memory contention shows a value more than four times smaller than the experimental value for a concurrency level of 18 while the model with memory contention computes a value that is 86 percent of the experimental value.

We now describe validation results obtained by running programs of the UnixBench [27] and HBench [25]

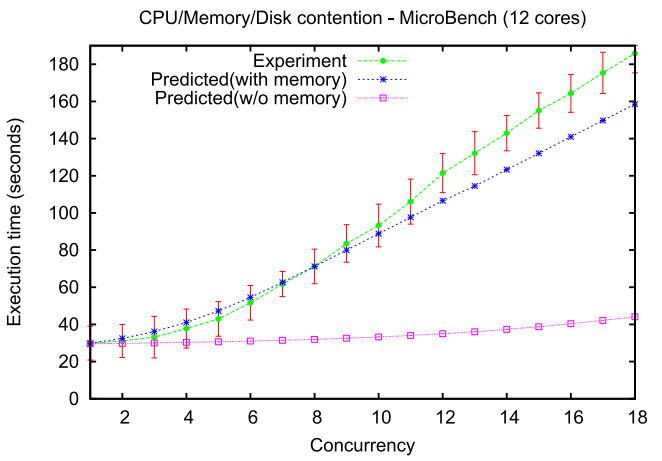


Fig. 7. Average execution time vs. concurrency level for MBench with disk I/O on Westmere (12 cores).

benchmarks. HBench has its roots in lmbench [26], which is another popular memory bandwidth and latency benchmark useful in comparing Unix/Linux system performance. We focus first on HBench, a suite of portable benchmark programs, which along with memory read, write and copy also performs numerous other tests such as network speed. We focused on the memory read, write, and copy functions. For each of these functions, HBench performs tests on many different sizes of memory starting from a few kilobytes all the way up to several megabytes. While running the tests and observing the stall values through `perf stat`, it was obvious that some of the programs were experiencing low stall values and some of them high stall values. This is not surprising since the runs with a high memory exposure will have to spend more time doing memory access and, as a result, experience more contention due to memory access when multiple copies of the same program execute concurrently. We collected execution time information along with the stall values for each of these runs. The top graph of Fig. 8 shows curves of average execution time vs. concurrency level on a four-core Sandy Bridge machine running HBench programs. The observed backend stall percentage is 51 percent. The bottom graph shows curves for HBench programs running on a 12-core Westmere machine; the backend stall percentage is 25 percent in this case. As it can be seen, the model with memory contention tracks the experimental results very closely while the model without memory contention deviates significantly from the experimental results.

We now discuss results obtained by running programs from UnixBench, which includes programs that perform multiple types of tests of Unix/Linux systems: DhryStone, Towers of Hanoi (a purely recursive algorithm), Arithmetic, Grep of a large file, graphics, system calls, and numerous others. We discuss in what follows results obtained by running Towers of Hanoi and grep. These programs were chosen (classical benchmark, Unix utilities and a well known recursive algorithm) for representing a good mix of programs with a varied amount of CPU, memory, and I/O service demands.

The graph in Fig. 9 shows multiple instances of the Towers of Hanoi program running on a 16-core Westmere machine. The percentage of backend stalls is large enough

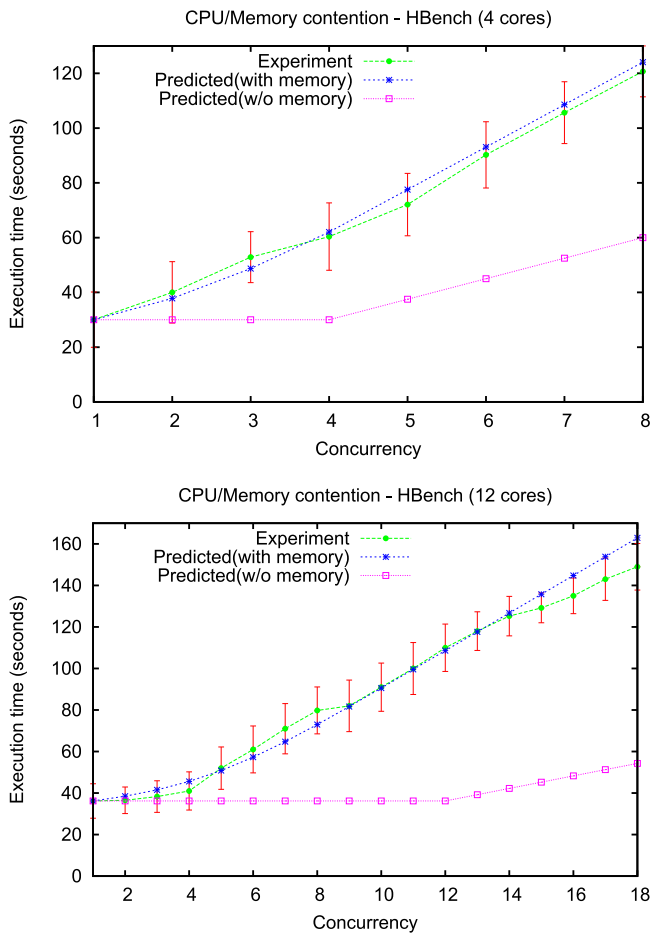


Fig. 8. Average execution time vs. concurrency level for the HBench benchmark. Top: Sandy Bridge (four cores) and 51 percent backend stall. Bottom: Westmere (12 cores) and 25 percent backend stall.

(10 percent) to make the model without memory contention deviate significantly from the experimental results, while the model with memory contention remains much closer to the experimental results. For example, the percent relative error of the model with memory contention is 19.7 percent for the 16-core case and a concurrency level equal to 24 while the same error is 50 percent for the same concurrency level for the model without memory contention.

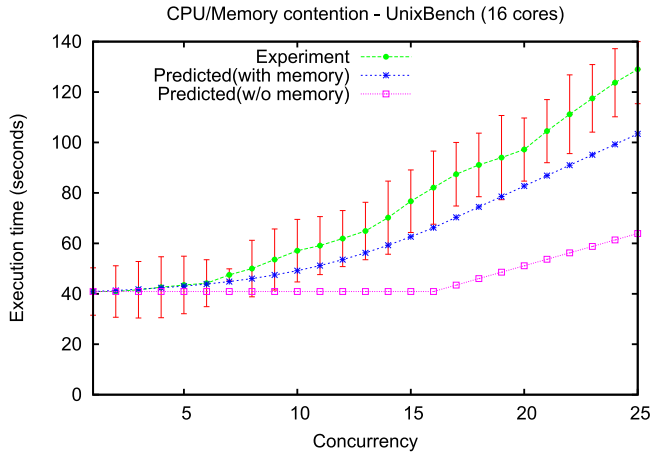


Fig. 9. Average execution time vs. concurrency level for the Unixbench benchmark Towers of Hanoi program. Westmere (16 cores) with 10 percent backend stall

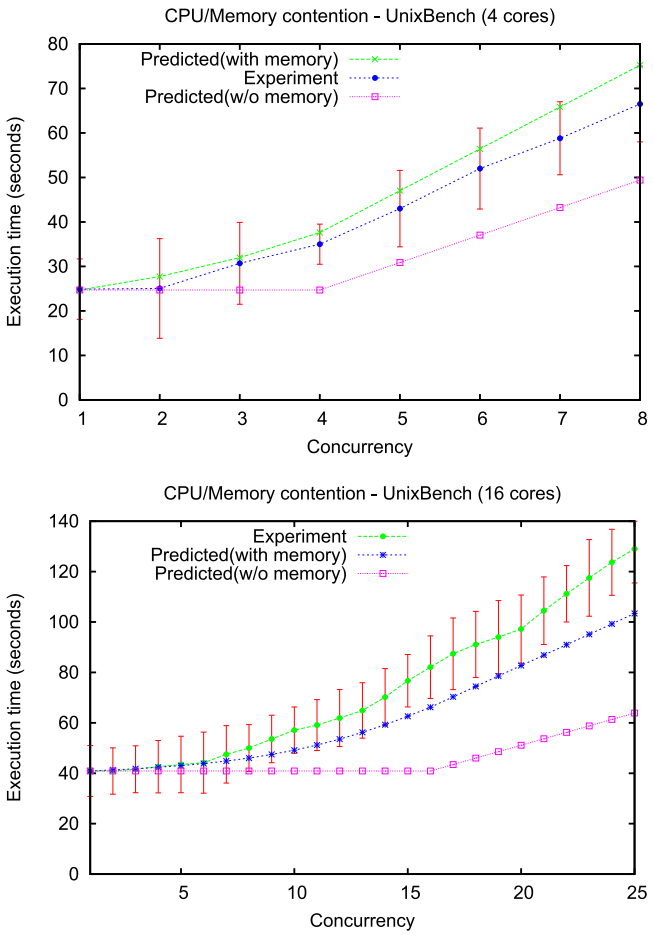


Fig. 10. Average execution time vs. concurrency level for the Unixbench benchmark Grep program. Top: Sandy Bridge (4 cores) with 35 percent backend stall. Bottom: Westmere (16 cores) with 10 percent backend stall.

The top graph of Fig. 10 shows the results for `grep` on a four-core Sandy Bridge machine; the percent stall is 35 percent. One can see that the model with memory contention tracks much better the experimental results than the model without memory contention. The same behavior can be seen in the bottom graph of the figure, which shows the results of running many instances of `grep` on a 16-core Westmere with a 10 percent percent backend stall.

We also ran experiments with the SPECCPU2006 benchmark suite on four-core and 12-core Linux CentOS servers. We chose two benchmarks from the suite: 429.mcf from the Integer benchmarks and 470.lbm from the Floating point benchmarks. Both benchmarks have high memory demand. The experimental results mirrored very closely (error ranges from −5 to 22 percent) the predicted values of our memory contention model. Fig. 11 shows the result of running benchmark 429.mcf on a four-core Sandy Bridge machine.

6 Extending the Analytic Model to Multiple Job Classes

The previous sections discussed our two-layer multi-core performance model from a single class workload perspective. We have also discussed the effects of NUMA-based

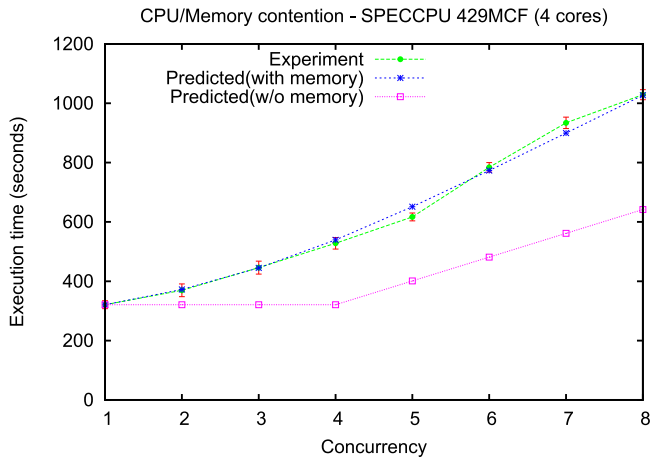


Fig. 11. Average execution time vs. concurrency level for the SPECCPU 429.MCF benchmark; Sandy Bridge (four cores) with 40 percent stall.

machines common today. We now extend the model to multiple classes of jobs (i.e., multiple types of jobs in terms of resource demands) and concentrate on the more general NUMA-based architecture introduced in Section 3.2. Note that the SMP case can be treated by removing the remote memory queue from the memory contention model.

The notation used in the multi-class case model is given in Table 4. The multi-class model extension requires the use of Approximate MVA (AMVA) [19] due to the high and known computational complexity of solving exact MVA for multiple classes.

Algorithm 3 shows the multi-class algorithm for the memory contention model. The loop that starts at line 4 estimates the average number, N_r^{mem}, of class r jobs in the memory contention model. The total number of jobs in the memory contention model cannot exceed m, the number of cores. The question is how this number should be apportioned to each job class. We use an approximation that apportions the total number of class r jobs in the memory contention model based on the proportion of class r jobs in the population $\vec{N}$. For example, suppose that there are three job classes, 12 cores, and that the population vector $\vec{N} = (8, 12, 4)$. Then, $\vec{N}^{\text{mem}} = (8/24 \times \min(12, 24), 12/24 \times \min(12, 24), 4/24 \times \min(12, 24)) = (4, 6, 2)$.

Algorithm 3. Memory Contention Model—Multiple Classes

Input: $m, r, \vec{N}, \vec{D}_{\text{Core-I}}, \vec{D}_{\text{Mem-Local}}, \vec{D}_{\text{Mem-Remote}}, \vec{D}_d$
/* Calculate the elements of a concurrency vector whose values are in proportion to the population vector $\vec{N}$ */
for $r = 1 \rightarrow R$ **do**
5: $N_r^{\text{mem}} \leftarrow N_r / \sum_{s=1}^{R} N_s \times \min\left(\sum_{s=1}^{R} N_s, m\right)$
end for
/* The vector $\vec{T}$, obtained by solving a closed QN model, provides the total time spent by jobs of each class processing instructions, waiting for, and accessing memory */
10: $\vec{T} \leftarrow \text{SolveClosedQNAMVA}(\vec{N}^{\text{mem}}, \vec{D}_{\text{Core-I}}, \vec{D}_{\text{Mem-Local}}, \vec{D}_{\text{Mem-Remote}}, \vec{D}_d)$;

We then use $\vec{N}^{\text{mem}}$ as the population vector to solve a closed QN model using AMVA. The algorithm for solving AMVA is well-known and can be found for example in [15].

TABLE 4
Notation for Multi-Class Model

R	number of job classes
$\vec{N} = (N_1, \ldots, N_r, \ldots, N_R)$	population vector indicating the number of jobs in each class
$\vec{D}_{\text{CORE-I}}$	vector of service demands for each class for device CORE-I
$\vec{D}_{\text{Mem-Local}}$	vector of service demands for each class for device Mem-Local
$\vec{D}_{\text{Mem-Remote}}$	vector of service demands for each class for device Mem-Remote
$\vec{D}_d$	vector of service demands for each class at the disk
$\vec{N}^{\text{mem}} = (N_1^{\text{mem}}, \ldots, N_r^{\text{mem}}, \ldots, N_R^{\text{mem}})$	vector of number of jobs of each class in the memory contention model
$\vec{T} = (T_1, \ldots, T_r, \ldots, T_R)$	vector of execution times per class for the memory contention model
$X_{0,r}(\vec{N})$	throughput of class r jobs for a job population $\vec{N}$
$\overline{n}_{\text{CPU},r}(\vec{N})$	average number of class r jobs using or waiting for cores for a job population $\vec{N}$
$R'_{\text{CPU},r}(\vec{N})$	average class r CPU residence time (i.e., time spent waiting or using a resource) for a job population $\vec{N}$

Line 10 of Algorithm 3 indicates that such a model is solved using as parameters $\vec{N}^{\text{mem}}$ and the service demands for all classes at the cores, local, remote memory, and I/O devices. The solution to this QN network provides the execution times, T_r, of jobs of class r in the memory model. The values in $\vec{T}$ are used in the application level model. It is also worth noting that some of the elements of the vector $\vec{N}^{\text{mem}}$ may not be integers as a result of the computation in line 5 of Algorithm 3. This is not a problem because AMVA allows for non-integer population values.

The multi-class application-level model is shown in Algorithm 4. This algorithm uses the same approximation for apportioning classes to cores as Algorithm 3 and does not require the use of the B-S approximation when there is no or little I/O activity. Then, the average number, $\overline{n}_{\text{CPU,r}}(\vec{N})$, of class r jobs using or waiting for cores is simply the class r population N_r (first statement in the loop). Each class throughput, $X_{0,r}(\vec{N})$, is estimated by applying Little's Law on a per class basis to the set of cores (not including the queue for cores). The average time spent by a class r job using a core (including executing instructions, waiting for, and accessing memory) is given by T_r, obtained by the memory contention model. To apply Little's Law on a per class basis to the set of cores, we estimate the average number of class r jobs using the cores as in Algorithm 3. Finally, we apply Little's Law on a per class basis to the entire system consisting of the queue for cores and the cores themselves to obtain the average class r CPU residence time, $R'_{\text{CPU},r}(\vec{N})$, which represents the execution time of class r jobs for a job population of $\vec{N}$.

Algorithm 4. Application Model - Multiple Classes

for $r = 1 \rightarrow R$ **do**
  $\overline{n}_{\text{CPU,r}}(\vec{N}) \leftarrow N_r$
  /* Apply Little's Law per class to the set of cores */
  $X_{0,r}(\vec{N}) \leftarrow [(N_r \times m)/(\sum_{s=1}^{R} N_s]/T_r$
5: /* Apply Little's Law per class to the entire system */
  $R'_{\text{CPU,r}}(\vec{N}) \leftarrow \overline{n}_{\text{CPU,r}}(\vec{N})/X_{0,r}(\vec{N})$
end for

If the I/O activity is not negligible, then the application level model can be solved using regular AVMA enhanced by the B-S approximation to capture contention for cores. The service demands at the cores is obtained from Algorithm 3. The next section discusses the experimental setup and the results of the multi-class experiments.

7 Multi-Class Model Results

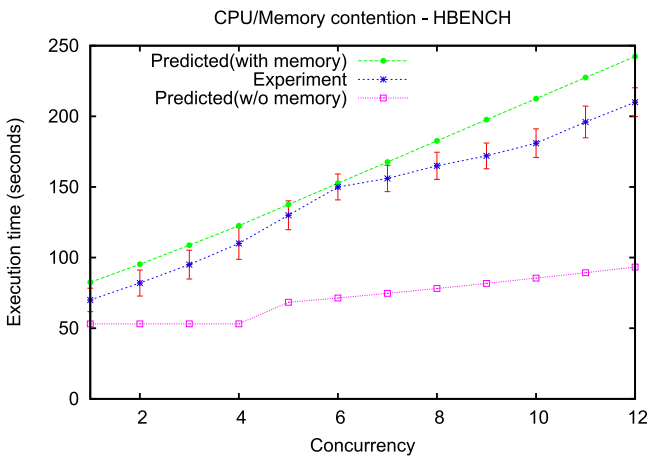
Multi-class experiments were conducted on a 12-core, 8-GB XEON-E5-2620 NUMA server running Linux 2.6.32. We generated two different workloads, each with three job classes derived from UBench, HBench, and our own MBench by modifying the number of times the main loop is executed. This allowed us to generate different service demands for these three programs and obtain two very distinct workloads (see Table 5).

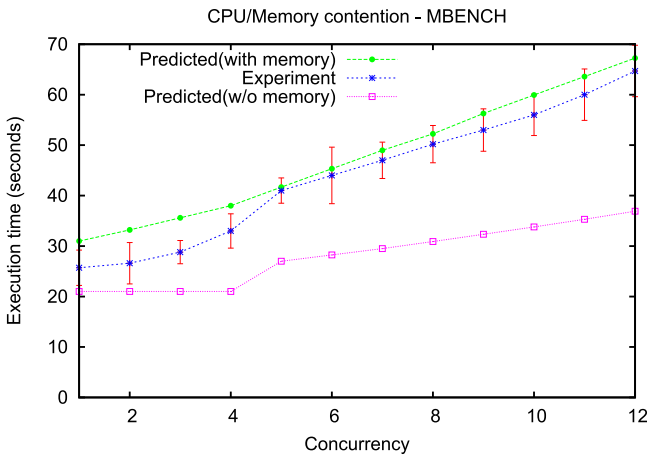
As in Section 4, memory service demands were collected using Linux's `perf`. For each workload, we ran experiments that varied the concurrency level of one of the classes while keeping constant the concurrency level of the other classes.

All figures presented in this section display experimental average execution times (with 95 percent confidence

TABLE 5
Input Parameters Obtained by Running the Programs on NUMA Based Intel XEON Machines

Parameter	Workload 1			Workload 2		
	UBench	MBench	HBench	UBench	MBench	HBench
$D_{\text{Core-I}}$	7.08 sec	14.7 sec	26.55 sec	27.94 sec	25.2 sec	8.65 sec
$D_{\text{Mem-Local}}$	0.1 sec	3.2 sec	13.0 sec	0.43 sec	5.4 sec	2.3 sec
$D_{\text{Mem-Remote}}$	0.1 sec	3.2 sec	13.0 sec	0.43 sec	5.4 sec	2.3 sec
PercStalledBackendCycles	3%	30%	50%	3%	30%	35%





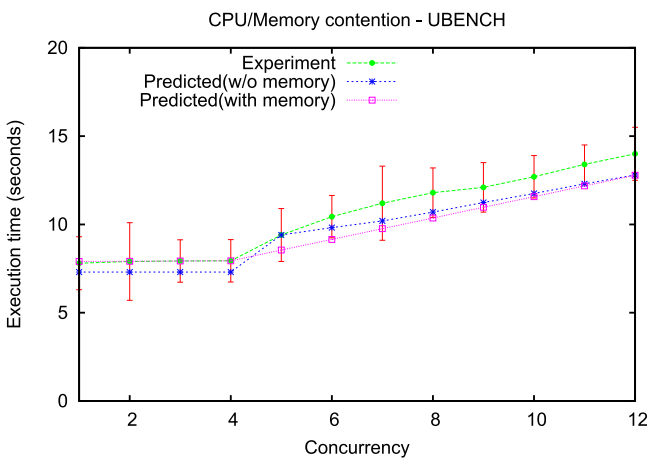
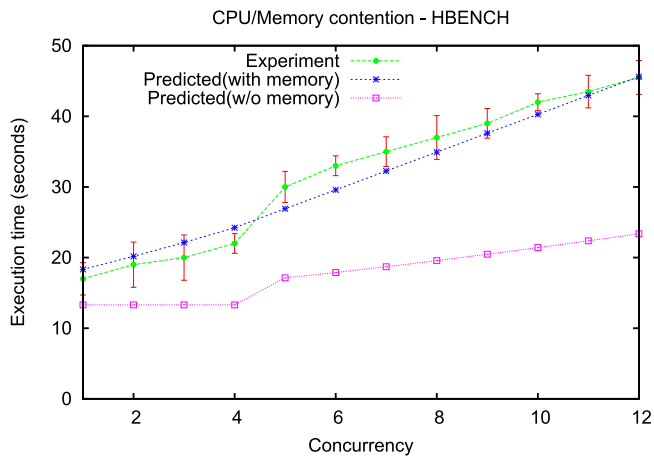
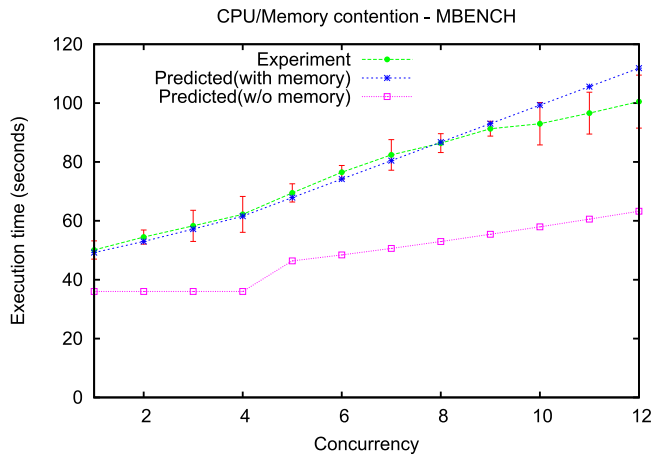


Fig. 12. Average execution time vs. concurrency level for HBench, MBench and UBench for Workload 1. Top: UBench and MBench fixed at concurrency 4. Middle: UBench and HBench fixed at concurrency 4. Bottom: HBench and MBench fixed at concurrency 4.





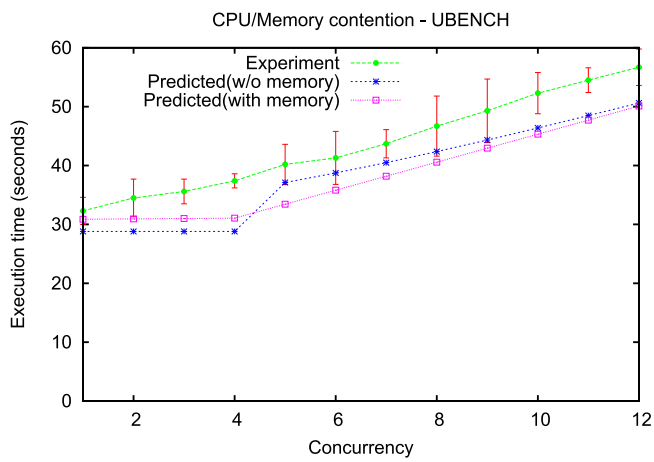


Fig. 13. Average execution time vs. concurrency level for HBench, MBench and UBench for Workload 2. Top: UBench and MBench fixed at concurrency 4. Middle: UBench and HBench fixed at concurrency 4. Bottom: HBench and MBench fixed at concurrency 4.

intervals) as well as the predicted execution times from the models with and without memory contention. Figs. 12 and 13 show similar results. The top (HBench) and middle (MBench) graphs show that the analytic model with memory contention tracks very well the experimental results while the model that does not consider memory contention does a very poor job at estimating the average execution time. For example, for HBench and a concurrency level of 12, the model with memory contention predicts an average execution time of 245 sec and the model without memory contention predicts 93 sec. The average measured execution time is 210 sec $\pm$ 10 sec. So, the model with memory contention predicts an execution time 11 percent above the upper bound of the 95 percent confidence interval for the measured values. The model without memory contention predicts an average execution time that is 46.6 percent of the lower bound of the 95 percent confidence interval for the measured values.

There is no significant difference between the models with and without memory contention for UBench (bottom curves in the graphs of Figs. 12 and 13) because UBench has a very small (i.e., 3 percent) value for

TABLE 6
Input Parameters for the Multi-Class Experiments with Disk Activity on a Four-Core Machine

Parameter	UBench	MBench	HBench
$D_{\text{Core-I}}$	10.8 sec	22.3 sec	71.2 sec
D_{Disk}	0 sec	2.5 sec	4.0 sec
PercStalledBackendCycles	3%	30%	50%

`PercStalledBackendCycles` (see Table 5). It is to be noted that when the total number of jobs is less than the number of cores, there is no contention in the model which does not consider memory contention. However, when the total number of jobs exceeds the number of cores, we use two approximations for the model without memory contention: AMVA as well as BS. This explains the sharp jump in execution time from concurrency 4 to concurrency 5 in the multi-class graphs when there is no disk demand for the "w/o memory" lines in the graphs. At that concurrency level, all three classes will have a concurrency level of four for a total number of jobs equal to 12, the number of cores.

We also ran multi-class experiments with disk activity on a four-core machine with the following job classes: MBench and HBench modified with disk load and an unmodified UBench (no disk activity). See service demands for these experiments in Table 6. Fig. 14 shows the measured average execution times for MBench, the predicted execution times for the models with and without memory contention. As the graph shows, the model with memory contention predicts the average execution time with much better accuracy (the results are within the confidence intervals for the measured values in most cases) than the model that does not consider memory contention (the results are always outside of the confidence intervals).

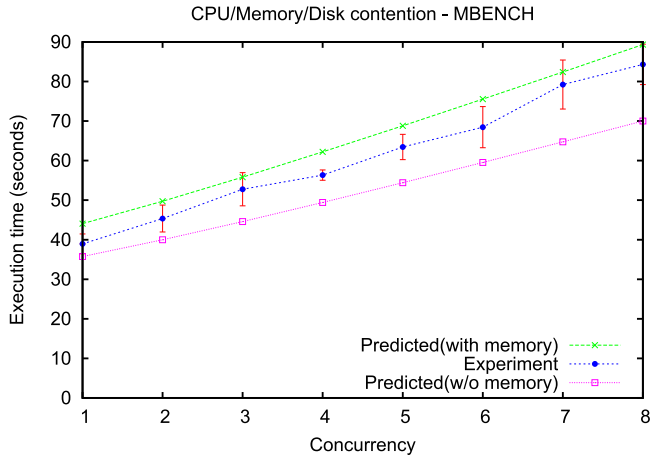


Fig. 14. MBench average execution time vs. Mbench concurrency level. UBench and HBench fixed at concurrency 4.

8 Related Work

Approximate MVA models for closed QNs with multiple-server queuing stations were developed in [1], [20], [22]. These models consider that all resources in a multi-server queuing station have the same processing speed regardless of concurrency level and do not consider the effects of memory contention. Three operational laws for a single queue with parallel servers were derived by Kelly et. al. [8]. These results also do not consider any contention for memory resources.

Fedorova et. al. discuss the possible reasons for contention in multi-core systems and provide new methods for mitigating contention through scheduling algorithms [6]. The authors in [21] use a control theory approach to partition processing cores, shared cache, and off-chip memory bandwidth between concurrently executing applications. A global resource broker is used to manage per application demand. Levesque et. al. discuss the AMD chips and also discuss methods to collect performance data and use that data to analyze the performance impact of multi-core processors [12]. That work uses micro benchmarks and other benchmarks to explore compiler switches for software optimization. In [5], the author argues that performance counters available on all major current processors are crucial to the understanding of hardware performance. The paper describes Linux-based simple monitoring tools that allow one to pinpoint key bottlenecks in applications. The paper in [24] examines several Java applications and concludes that saturating the memory bandwidth of a multi-core system essentially degrades scalability. The authors postulate that Java's garbage collection scheme is detrimental on multi-core platforms.

Chen et. al. [4] study applications running on multi-core machines and examine the question of whether one can maximize resource usage while respecting application performance goals. The authors use queuing theory to predict the scalability of applications on multi-core machines. They do not however include memory contention in their model.

Towsley [28] provided a discrete time Markov model solution to modeling multiple processors and multiple bus/memory configurations. His methodology consisted of three steps: a) Replace the memory systems with a single aggregate queue whose service rate reflects the behavior of the memory system when there is no bus contention; b) Modify this queue to add contention; and c) Solve the resulting two-queue system with CPU and memory. That work, which was done in 1983, has some similarities with our work in the sense that it also collapses the entire cache/memory system into a single queue. However, there are some significant differences: a) Unlike the work in [28] that uses simulation and exact solutions for validation, we use experiments, b) The model in [28] is single class while ours includes a multi-class formulation, c) Our solution has small computational and storage requirements because we rely on AMVA, and d) We use hardware performance counters to parameterize our model, making it more applicable.

In [29], the authors modeled shared bus multiprocessors using MVA and compared model results with trace-driven simulation results. The authors used `awk`, `compress`, `nroff`, and `sort` as benchmark programs. They focus their work on cache misses and collected memory trace data for the benchmark programs to generate bus request statistics. These statistics were then fed into their Customized MVA (CMVA) algorithm. That paper does not specifically mention how to handle multiple classes of jobs and does not compare analytic results with experimental results.

The authors in in [30] provided MVA models for CPU, I/O, and memory. They provided several what-if scenarios and discussed cache coherency. The results are based on simulation and not actual experimentation. That paper presented a detailed study on the I/O portion of the total execution time of a process and also studied the architectural effects of symmetric multi-processing machines on TPC-C (www.tpc.org) benchmarks for a single class environment.

Lastly, in [32] the authors developed and validated an analytic model for evaluating shared-memory systems with ILP processors. Even though this work predates multi-core computers, it shows how application performance can be studied and inferences drawn by using analytical models. One of the main contributions of that paper is a trace-driven simulation method to generate parameters an order of magnitude faster than a detailed execution-driven simulator.

9 Concluding Remarks

Memory contention in multi-core machines may be a significant portion of the execution time of an application, especially for machines with a large number of cores. This paper presented a two-level approximate single- and multi-class queuing network analytic model to predict the execution time of applications running on multi-core machines. The first level captures contention for memory and the other incorporates that contention into an application-level model that captures contention for cores. Our model also considers SMP and NUMA architectures.

We used hardware counters provided by modern Intel processor chips to obtain the parameters for the memory contention model. To that end, we focused on back-end-stall cycles, which provide a good approximation for cycles spent by the CPU due to memory access during the load/store phase of instruction execution. In the future, we plan to research additional CPU hardware counter events to obtain more accurate values for memory-related contention.

We developed a micro-benchmark written in C that allowed us to vary the amount of memory allocation, the amount of memory accessed, and the extent of I/O activity. Linux's heavy use of buffer cache, whereby the OS fetches complete file data if system memory is available, required us to flush the cache buffer before every run. We intend to investigate the effect of this eager fetch of file data on the performance model. We have also ran experiments using well-known Linux benchmarks, HBench, UnixBench and SPECCPU2006 to further validate our model. We plan to validate our model on other Intel, AMD, and Alpha chips and extend testing based on programs written in languages other than C.

Our many experiments showed that ignoring the effect of memory contention when modeling applications with any amount of appreciable memory demand produces erroneous results. The results of our single and multiple-class experiments demonstrate that our model is capable of estimating with an acceptable accuracy the execution time of jobs running on multi-core machines. Moreover, we showed how one can easily obtain input parameters to our model using data readily available in the performance counters of modern processors.

Our experiments used single-threaded applications. We will investigate the impact of multi-threaded applications in future work. This may require using QN models of software contention [15].

Our work may be very relevant for Hadoop-like systems [31] wheret most cluster nodes are multi-core computers. The models presented here afford the possibility to predict the execution time of MapReduce jobs.

References

[1] I. Akyildiz and G. Bolch, "Mean value analysis approximation for multiple server queuing networks," *Perform. Eval. J.*, vol. 8, pp. 77–91, 1988.

[2] S. Bardhan and D. A. Menascé, "Analytic models of applications in multi-core computers," in *Proc. IEEE 21st Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, San Francisco, CA, USA, Aug. 2013, pp. 318–322.

[3] A. Brown and M. Seltzer, "Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture," in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.*, Seattle, WA, USA, Jun. 1997, pp. 214–224.

[4] L. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder, "Achieving application-centric performance targets via consolidation on multi-cores: Myth or reality?" in *Proc. 21st Int. Symp. High-Perform. Parallel Distrib. Comput.*, Delft, The Netherlands, 2012, pp. 37–48.

[5] S. Eranian, "What can performance counters do for memory subsystem analysis?" in *Proc. ACM SIGPLAN Workshop Memory Syst. Perform. Correctness, Conjunction 13th Int. Conf. Archit. Support Programm. Lang. Oper. Syst.*, 2008, pp. 26–30.

[6] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing contention for shared resources on multi-core processors," *Commun. ACM*, vol. 53, no. 2, pp. 49–57, Feb. 2010.

[7] A. Kayi, Y. Yao, T. El-Ghazawi, and G. Newby, "Experimental evaluation of emerging multi-core architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Long Beach, CA, USA, Mar. 2007, pp. 1–6.

[8] T. Kelly, K. Shen, A. Zhang, and C. Stewart, "Operational analysis of parallel servers," in *Proc. 16th Annu. Meeting IEEE Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst.*, Baltimore, MD, USA, Sep., 8–10, 2008, pp. 1–10.

[9] J. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach* (*The Morgan Kaufmann Series in Computer Architecture and Design*), 5th ed. Waltham, MA, USA: Elsevier, 2012.

[10] Intel Corporation. (2014, Sept.). Intel 64 and IA-32 architectures software developer manuals [Online]. Available: www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[11] C. Isci, J. Liu, B. Abali, J.O. Kephart, and J. Kouloheris, "Improving server utilization using fast virtual machine migration," *IBM J. Res. Develop.*, vol. 55, no. 6, pp. 4:1–4:12, Nov./Dec. 2011.

[12] J. Levesque, J. Larkin, M. Foster, J. Glenski, G. Geisler, and S. Whalen, "Understanding and mitigating multi-core performance issues on the AMD Opteron architecture," Nat. Energy Res. Sci. Comput. Center, Oakland, CA, USA, LBNL Tech. Rep. LBNL-62500, 2007.

[13] D. Leventhal, "Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors," *Intel Perform. Anal. Guide*, 2009.

[14] J. D. C. Little,"A proof for the queuing formula: $L = \lambda W$," *Oper. Res.*, vol. 9, no. 3, pp. 383–320, 1961.

[15] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*. Upper Saddle River, NJ, USA: Prentice-Hall, 2004.

[16] O. Ozturk, "Multi-core education through simulation," *IEEE Trans. Educ.*, vol. 54, no. 2, pp. 203–209, May 2011.

[17] L. Peng, J. K. Peir, T. K. Prakash, Y. K. Chen, and D. Koppelman, "Memory performance and scalability of Intel's and AMD's dual-core processors: A case study," in *Proc. IEEE Int. Perform., Comput., Commun. Conf.*, New Orleans, LA, USA, Apr. 2007, pp. 55–64.

[18] M. Reiser and S. Lavenberg,"Mean-value analysis of closed multichain queuing networks," *J. ACM*, vol. 27, no. 2, pp. 313–322, 1980.

[19] P. Schweitzer, "Approximate analysis of multi-class closed networks of queues," in *Proc. Int. Conf. Stochastic Control Optim.*, Amsterdam, Netherlands, 1979.
[20] A. Seidmann, P. J. Schweitzer, and S. S. Oren, "Computerized closed queueing models of flexible manufacturing systems," *Large Scale Syst. J.*, vol. 12, pp. 91–107, 1987.
[21] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das, "Mete: Meeting end-to-end QoS in multi-cores through system-wide resource management," in *Proc. ACM SIGMETRICS Joint Int. Conf. Meas. Model. Comput. Syst.*, San Jose, CA, USA, Jun. 07–11, 2011, pp. 13–24.
[22] R. Suri, S. Sahu, and M. Vernon, "Approximate mean value analysis for closed queuing networks with multiple-server stations," in *Proc. Ind. Eng. Res. Conf.*, May 2007.
[23] From Wikipedia. Perf: Linux profiling with performance counters [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page
[24] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao,"Allocation wall: A limiting factor of Java applications on emerging multi-core platforms," *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 361–376, 2009.
[25] Hbench. (1997). [Online]. Available: http://www.eecs.harvard.edu/vino/perf/hbench/readme.html
[26] LMBench. (1998). [Online]. Available: http://www.bitmover.com/lmbench/
[27] UnixBench. [Online]. Available: https://code.google.com/p/byte-unixbench/
[28] D. Towsley, "An approximate analysis of multiprocessor systems," in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst. ACM*, 1983, pp. 207–213.
[29] M-C. Chiang and G. S. Sohi.,"Experience with mean value analysis model for evaluating shared bus, throughput-oriented multiprocessors," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 19, no. 1, pp. 90–100, 1991.
[30] X. Du, X. Zhang, Y. Dong, and L. Zhang, "Architectural effects of symmetric multiprocessors on TPC-C commercial workload," *J. Parallel Distrib. Comput.*, vol. 61.5, pp. 609–640, 2001.
[31] Hadoop. (2014). Documentation and open source release [Online]. Available: http://hadoop.apache.org/core/
[32] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ILP processors," in *Proc. 25th Int. Symp. Comput. Archit.*, Barcelona, Spain, 1998, pp. 380–391.
[33] D. L. Eager, D. J. Sorin, and M. K. Vernon, "AMVA techniques for high service time variability," in *Proc. Int. Conf. Meas. Model. Comput. Syst.*, Santa Clara, CA, USA, 2000, 217–228.

Shouvik Bardhan received the undergraduate degree from IIT, Kharagpur, India, and the MS degree in CS from The Johns Hopkins University. He is currently working toward the PhD degree in CS at George Mason University, Fairfax, VA. His areas of interest include performance modeling and analysis, big data, and autonomic computing.

Daniel A. Menascé received the PhD degree in CS from UCLA in 1978. He is a university professor of computer science at George Mason University, Fairfax, VA. He received the 2001 A.A. Michelson Award from the Computer Measurement Group. His research interests include autonomic computing, analytic modeling and analysis of computer systems, and software performance engineering. He is a fellow of the IEEE and the ACM.