Understanding Off-chip Memory Contention of Parallel Programs in Multicore Systems

Bogdan Marius TUDOR, Yong Meng TEO
Department of Computer Science
National University of Singapore
13 Computing Drive, Singapore 117417
[bogdanma, teoym]@comp.nus.edu.sg

Simon SEE
NVIDIA
Nordic European Center
3 International Business Park
#01-20A, Singapore 609927

***Abstract*—Memory contention is an important performance issue in current multicore architectures. In this paper, we focus on understanding how off-chip memory contention affects the performance of parallel applications. Using measurements conducted on state-of-the-art multicore systems, we observed that off-chip memory traffic is not always bursty, as it was previously reported in literature. Burstiness depends on the problem size. Small problem sizes lead to bursty memory traffic, and generate small off-chip contention. In contrast, when large program sizes cause memory contention, the memory traffic is non-bursty. Based on these observations, we propose an analytical model that relates the growth of memory contention to the number of active cores and to the problem size, for both uniform (UMA) and non-uniform memory access (NUMA) systems. Our model differs from measurements on average by less than 14%. Contention for off-chip memory grows exponentially with the number of active cores, but adding additional memory controllers reduces the memory contention. For programs such as the pentadiagonal solver SP from NPB benchmark, with a large matrix of 162^3 elements (input size C), our analysis shows that memory contention increases the total number of processor cycles to execute the program by more than ten times on a machine with 24 cores.**

I. Introduction

Over the past decade, multicore systems have become the backbone of parallel processing. In multicore systems, the processors consist of multiple parallel execution units called cores. However, sharing of important resources such as caches and memory bandwidth leads to competition and resource contention among cores. Furthermore, the number of cores is increasing with each technology generation, but the memory bandwidth is increasing at a much slower rate, because of wire delays and power dissipations, among others [22]. Thus, off-chip memory bandwidth available per core is not keeping pace with the increase in the number of cores. Another trend is that memory capacity available per dollar continues to grow according to Moore's Law. As long as these technology trends continue, a significant performance challenge in exploiting multicore systems revolves around contending for off-chip memory [16], [26].

One cause of memory contention is the increasing mismatch between memory and processor performance. A well studied performance issue before the shift to multicore is *the latency* of accessing the main memory [12]. But in current multicore systems, sharing off-chip memory introduces the new performance problem of *contention for memory bandwidth*. Performance of a program degrades when multiple cores compete for off-chip memory bandwidth. Increasing the number of active cores exerts demand for off-chip memory, and results in longer memory access time and increases the total number of cycles to execute a program. With lower memory cost, larger memory capacity allows for an increase in the size of the problems that can be executed, increasing the demand for off-chip memory bandwidth and leading to contention.

The objective of this paper is to advance the understanding of memory contention in parallel programs. To this effect, our first contribution are a series of observations based on measurements of memory request traffic in high performance computing applications and real world programs. Using three state-of-the-art multicore systems with 8, 24 and 48 cores, we show that memory traffic is not always highly bursty, as previously reported in the literature [13], [16]. We show that there are two types of relationships between problem size, burstiness and memory contention: (i) small problem sizes generate small contention but off-chip requests are highly bursty, and (ii) large problem sizes have much less bursty memory traffic but can lead to very large memory contention. Based on these observations, our second contribution is an analytical queueing model for answering two key questions in both UMA (uniform memory access) and NUMA (non-uniform memory access) multicore systems:

1) What is the impact of the number of active cores on memory contention?
2) What is the impact of the problem size on memory contention?

The proposed model is validated against measurements on systems with up to 48 cores. Because it is based on observations derived from measurements, our model achieves good accuracy for programs with large contention, with average relative error between 5-14% on three large multicore systems.

The rest of the paper is organized as follows. In section 2, we discuss related work. Section 3 focuses on key observation on memory contention in large parallel programs drawn from measurement experiments. In section 4 we present our analytical memory, with assumpations and modeling details. Section 5 discusses the validation of our model against measurement for both UMA and NUMA memory systems, and the impact of varying the number of cores and the problem size on memory contention. Our concluding remarks are in section 6.

II. Related Work

Contention for shared resources in multicore systems has received significant attention in the research community. In general, studies of off-chip resource contention fall in two main directions: reducing off-chip memory accesses [4], [5], [7], [15], [19], [21], [28], and improving the performance of off-chip requests [13], [16], [17], [18]. In reducing off-chip memory accesses, a major target for optimization is the last-level of cache memory. Partitioning of shared caches has been proposed as a technique to reduce the number of cache misses [21], [15], [19], [24]. Utility-based cache partitioning [21] uses specialized hardware to determine the miss rate of co-scheduled parallel programs, and partitions the available cache memory to reduce the overall miss rate. But, in software-based cache partitioning [4], operating systems page coloring is used to map the physical memory requests of a program to a reserved part of the cache. In cache-aware applications, co-scheduling is exploited to optimize cache miss fairness among different programs [5] or overall system performance [28]. Herdrich et al. [7] proposes throttling the speed of the cores to generate an imbalanced number of cache misses for relieving contention in applications with different memory access intensity. There are many approaches to improve the performance of off-chip requests. Memory bandwidth partitioning [10], [13], [16], [17], [18] has been proposed for optimizing different performance criteria. Kim et al. propose ATLAS [13], a memory controller scheduler that prioritizes threads with least-attained service levels to improve the overall performance of co-scheduled threads. The Fair Queue Memory System [18] ensures that co-scheduled threads receive a predetermined fraction of the memory bandwidth regardless of other threads memory requirements. Liu et al. [16] studied and modeled the interaction between cache and bandwidth partitioning with the goal of optimizing the overall performance of co-scheduled threads.

While there are many studies to reduce memory contention, there are few general models that directly link the performance of parallel applications to resource contention, number of active cores, problem size and the patterns of memory access.

Using high performance applications, Hood et al. [8] propose a model to determine the performance impact of shared-resource contention such as cache, bus, memory controllers and processor interconnects. Their differential performance analysis approach measurs the performance for different configuration scenarios. However, their approach does not apply to predictive performance analysis, nor does it take in consideration the problem size and burstiness patterns.

Sancho et al. [23] study the relationship between memory bandwidth and peformance of parallel programs when the number of memory channels of each memory controller is changed. The approach, based on measurements of memory bandwidth and parallel processing rate show that increasing the number of cores exerts higher memory demand and has diminishing results on performance due to memory bandwidth saturation. Their focus is to understand which configuration offers the best memory bandwidth. Our observations from experiments using larger number of core count are largely in line with their, but we link directly the peformance of parallel applications with the cycles incurred when using different number of cores. Furthermore, we complement our observations with observations of memory burstiness patterns and an analytical model that enables predictive performance analysis.

Liu et al. propose a general analytical model for understanding the effect of bandwidth fraction on individual thread performance [16]. Based on the cache miss ratio, their model determines the CPI performance of co-scheduled threads, and the slowdown of co-scheduling groups of threads relative to scheduling each thread individually. However, it is unclear how the fraction of last-level cache misses is determined when different number of threads are scheduled together. Their model also does not explore changes in problem size, in particular large problem size that is typical for parallel programs. In contrast, our memory model is designed to relate contention to the last-level cache misses, problem size and number of cores.

In our previous work [26] we proposed a combined analytical model for memory contention and data dependency in high performance applications. The objective of the model is to detemine the speedup of parallel applications and to determine the number of cores that maximizes speedup. In this paper we extend the coverage of the model to programs with low and high memory contention, focusing also on understanding the burstiness patterns of memory requests, and the impact of burstiness on memory contention.

III. Memory Contention in Multicore Systems

This section presents our observations on the memory contention in large multicore systems with different memory architectures. We first show what are effects of memory contention on parallel programs when the number of active cores and the problem size change. Next, we study the nature of memory contention by profiling the patterns of memory access.

A. Experimental Setup

Two benchmarks that are representative of parallel computing are used in this paper. The NPB 3.3 benchmark [2] that implements HPC dwarfs [1] using OpenMP 2.5 is selected because these dwarfs scale in terms of both problem size and number of threads, and covers a wide degree of memory contention. The PARSEC 2.1 programs implemented using pthreads represents real-world parallel workloads [3].

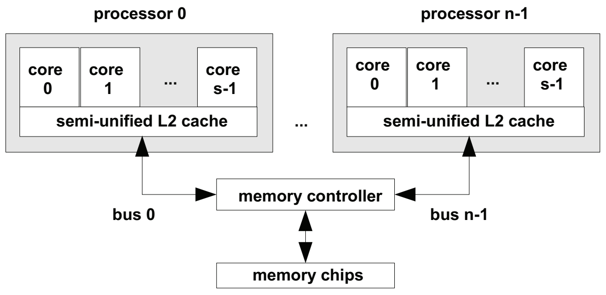
All programs were compiled with GCC 4.3 for 64 bit executables using full optimizations (`-O3`). We have profiled

Name	Parallel kernel
EP	Embarrassingly parallel: low data dependency, low memory
FT	Spectral methods: fast Fourier transform
IS	Parallel sorting: bucket sort on integers
CG	Sparse linear algebra: data with many 0 values
SP	Structured grid: pentadiagonal solver
x264	Video encoding using H264 codec

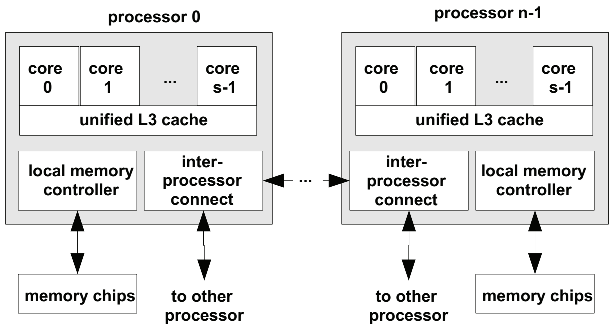
TABLE I
FIVE NPB 3.3 AND ONE PARSEC 2.1 PARALLEL PROGRAMS

six programs from NPB and four PARSEC applications, but because of space limitations we show a subset, in Table I, that best illustrate the different cases of memory contention. The operating system for all experiments is 64 bit Linux 2.6.35.

The multicore systems used in experiments have two main types of memory architectures, UMA and NUMA, as shown in Fig. 1. In UMA systems, multiple cores are connected to



(a) n processors with UMA interconnect



(b) n processors with NUMA interconnect

Fig. 1. Architectures of Multiprocessor Multicore Systems

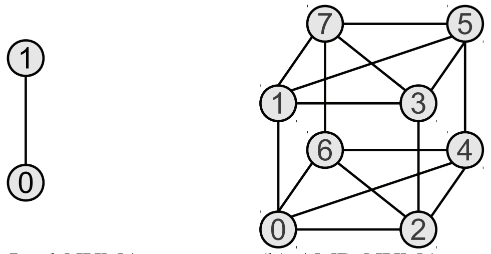
a common memory controller through private buses. The last-level cache in UMA is considered semi-unified. These reflect UMA microarchitectures such as Intel Clovertown and Intel Harpertown. Since all the cores share one memory controller, contention occurs when memory requests exceed the capacity of the memory controller. In contrast, each multicore processor in a NUMA system accesses its own memory through its dedicated local memory controller. A core accesses memory owned by another processor through its inter-processor connection network. Current microarchitectures based on NUMA systems microarchitecture include Intel Nehalem and AMD K10.

The multicore systems used for the experiments are:

1) **Intel UMA (8 cores)**: Dual quad-core Intel Xeon E5320 processors, 8 MB L2 cache, one memory controller with 4 GB dual-channel DDR2 RAM.
2) **Intel NUMA (24 cores)**: Dual six-core Intel Xeon X5650 processors, 12 MB L3 cache, two hardware threads per core, two memory controllers, 12 GB triple-channel DDR3 RAM.
3) **AMD NUMA (48 cores)**: Quad twelve-core AMD Opteron 6172 processors, 10 MB L3 cache, eight memory controllers (two controllers per processor), 64 GB dual-channel DDR3 RAM.

For Intel NUMA, we consider the two hardware threads of each physical core as logical cores, because the objective of this study is off-chip memory requests. Each of the two hardware threads issue memory requests independently, so from the perspective of the memory accesses, the physical core with two hardware threads appears as two cores. Therefore, we consider Intel NUMA as having 24 cores.

The interconnect networks for the NUMA systems is shown in Fig. 2. Intel NUMA has two memory controllers directly interconnected, therefore there are two latencies for accessing the memory – direct and one hop. AMD NUMA has four processors, each with two memory controllers. The eight memory controllers are interconnected through a partial mesh, and there are three latencies of accessing the memory – direct, one hop and two hops.



(a) Intel NUMA (b) AMD NUMA

Fig. 2. Memory Interconnect of NUMA Systems

The program was partitioned into a fixed number of threads. The number of cores was varied from one to maximum number of cores of the machine using a *fill-processor-first* policy. For Intel NUMA, memory controller 0 was used until all cores from processor 0 were active, and then memory controller 1 was activated. For AMD NUMA, the memory controllers belonging to the same processor were activated simultaneously, in the following order: 0 and 1, then also 2 and 3, then also 4 and 5, and finally 6 and 7. Each experiment was conducted five times and average values are reported. To minimize the variability of the results, we fixed the CPU affinity for each thread using `sched_setaffinity` system call. The NUMA policy was applied using `numactl`. We used PAPI 4.1.2.1 on NUMA and PAPI 3.7.0 on UMA to access the processor hardware counters. We measure the following counters: `PAPI_TOT_CYC` for the number of cycles, `PAPI_TOT_INS` for the number of instruction, `PAPI_RES_STL` for stall cycles, `PAPI_L2_TCM` for the number of cache misses, `LLC_MISSES` on Intel NUMA and `L3_CACHE_MISSES` on AMD NUMA for L3 misses. The work cycles were determined as the difference between all cycles and stall cycles. We used `papiex` tool to measure the hardware counters of the profiled applications only, without interference from background processes and operating system. To assure that the memory bandwidth is not shared with any other process, we turned off all non-essential processes and we run the profiled application with the highest priority allowed for non-root processes. We used the LIKWID toolkit to determine the mapping between logical core ids and the physical topology. [25].

B. Measurement Experiments and Observations

Our measurement experiments focus on understanding how the execution of a program is affected by the off-chip memory traffic. We first perform a set of measurements on the number of cycles required to execute the programs when using different number of cores. Second we measure the patterns of memory traffic to understand the nature of the memory contention.

1) Memory Contention vs Number of Cores: For each program, we measure the *total number of cycles* required to execute the program across all the active cores, including initialization and cleanup, as well as the number of stall and work cycles, and the total number of last-level cache misses. Mainstream processor cores are based on superscalar and deeply-pipelined microarchitectures. Thus, in each cycle a core can execute multiple integer and floating point operations and issue multiple memory requests. If the operands of an instruction are available in the registers, the execution of the instruction can proceed. Otherwise, the core stores the instruction in the *instruction dispatch queue* until the operands are fetched from the first-level cache. If the data is not available in the first-level cache, then it attempts to fetch it from the subsequent levels of cache. If the data is not found in cache, the core issues a *memory request* to the main memory. Due to the long latencies of accessing higher levels of cache or the main memory, instructions can be stopped for several hundreds of cycles [12]. If the entire dispatch queue is filled with instructions waiting for data, no instructions can proceed and the core is stalled waiting for memory. If no operations are completed during a cycle, it is called a *stall cycle*. In contrast, if at least one instructions is completed during the cycle then is termed a *work cycle*. Next we discuss our observations.

Tabel II shows the the normalized increase in the total number of cycles for five HPC dwarfs with small (W) and large (C) problem size[1]. The increase in the number of cycles is defined as the difference between the total cycles incurred using n cores and one core, normalized to the number of cycles on one core. We present the normalized increase for n equal to half and all cores of the systems (i.e. 4 and 8 on

Program	**Size**	**Normalized Increase in Number of Cycles**					
		Intel UMA		**Intel NUMA**		**AMD NUMA**	
		#Cores		**#Cores**		**#Cores**	
		n=4	**n=8**	**n=12**	**n=24**	**n=24**	**n=48**
EP	W	0.00	0.00	0.03	0.57	0.01	0.59
IS	W	0.10	0.57	0.33	0.33	0.21	0.44
FT	W	0.32	0.58	0.18	0.34	0.11	0.23
CG	W	0.01	0.04	0.10	0.43	0.11	0.13
SP	W	0.32	0.58	0.10	0.50	0.13	0.21
EP	C	0.00	0.00	0.01	0.54	0.06	0.55
IS	C	0.07	0.56	0.26	0.85	0.40	0.70
FT	C	0.70	1.80	1.62	3.94	0.39	0.46
CG	C	0.91	2.41	1.43	3.31	0.83	1.91
SP	C	3.34	7.05	6.55	11.59	4.69	9.84

TABLE II
NORMALIZED INCREASE IN NUMBER OF CYCLES FOR SMALL (W) AND LARGE (C) PROBLEM SIZE IN HPC DWARFS

Intel UMA, 12 and 24 on Intel NUMA, 24 and 48 on AMD NUMA). Because FT.C working set size exceeds 4 GB and leads to swapping in our Intel UMA system, we use class B as large problem size for program FT on Intel UMA. Overall, on all three systems the increase in number of cycles is more pronounced for higher number of active cores.

We identified two types of behavior with respect to the number of active cores:

1) Programs with small problem size or working sets which are cached effectively generate low number of off-chip requests. This leads to a negligible growth in number of cycles when the number of active cores increase.
2) Programs with large problem sizes generate high number of off-chip memory requests which lead to a significant growth in the number of cycles when the number of cores increase.

We discuss in detail these two patterns using a representative HPC program. Program CG is a parallel application that approximates the largest eigenvalues for a large and sparse matrix. We use a small and a large problem size [2]:

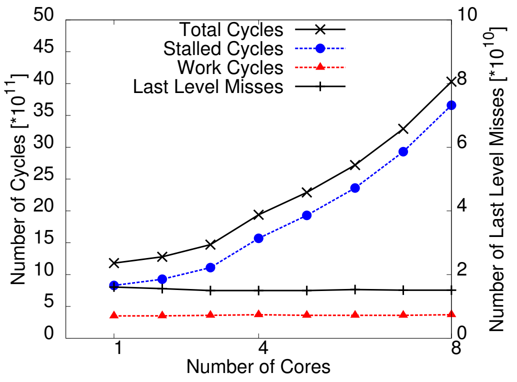
1) Class W consists of a matrix with $7,000^2$ elements;
2) Class C consists of a matrix with $150,000^2$ elements.

CG is representative for all HPC applications and is chosen because it represents a case with moderate memory contention (SP and FT have higher contention, IS, EP and all PARSEC programs have lower contention).

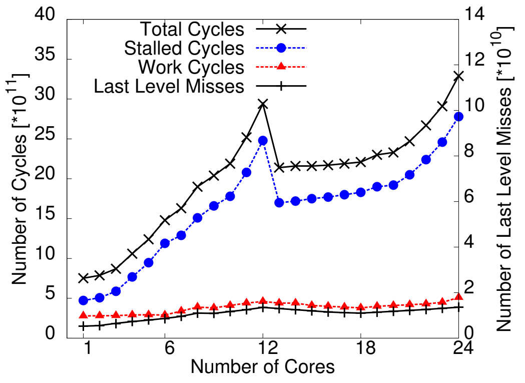
Table II shows for CG the growth of the number of cycles when using all the cores of systems (8 cores on Intel UMA, 24 cores on Intel NUMA, 48 cores on AMD NUMA), relative the the baseline value of one core. Small problem size W generates very small increase in number of cycles, even on large number of cores. In contrast, problem size C shows a very large growth in number of cycles, on all three systems.

Next we focus the discussion on the more interesting case of large problem size. Fig. 3 shows the effects of increasing the number of active cores on the total number of cycles, stalled cycles, work cycles and last level cache misses for CG.C. On all systems, there are three observations when the number of active cores is increased:

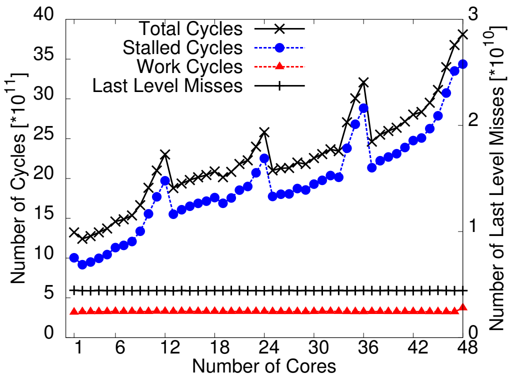
[1]Problem sizes are denoted by letters, and are according to NPB benchmark specification. Notation CG.C means program CG problem size C.



(a) Intel UMA: Xeon E5320



(b) Intel NUMA: Xeon X5650



(c) AMD NUMA: Opteron 6172

Fig. 3. CG.C: Varying the Number of Cores

1) The number of total cycles increases non-uniformly.
2) The growth in number of total cycles is due to an increase in number of stall cycles.
3) The number of work cycles and the number of last level cache misses grow insignificantly.

For problem size C, the patterns of growth depend on the architecture and the number of memory controllers. For Intel UMA we observe two sustained growth intervals, the first from one to four cores, the second from five to eight. This corresponds to a per-processor pattern of growth. Similarly, on Intel NUMA, the growth on the first processor (1 to 12 cores) is similar in shape with the growth from 13 to 24 cores. However, when the second processor is activated (from 13 cores onward), there is a small decrease in memory contention which results from the added memory bandwidth of the second memory controller. On AMD NUMA, there are four intervals of growth, each corresponding to activating two memory controllers from a processor.

As the number of cores is increased, we observe that the number of work cycles remains roughly constant because the critical path of the program is dominated by instruction waiting for operands fetched from memory. Instruction execution is interleaved almost fully with fetching operands. Thus, the increase in the total number of cycles is dominated by waiting for memory requests or stall cycles. This can be clearly seen in Fig. 3, as the shape of growth of the stall cycles closely follows the shape of growth of total cycles.

Another interesting observation is that the total number of last-level cache misses, L2 for UMA and L3 for NUMA, changes unsignificantly when the number of active cores is increased. Because we fixed the number of threads, and varied only the number of cores, the total number of instructions also remains constant for a given problem size. This confirms that the increase in the total number of cycles is the result of contention for off-chip memory requests, rather than an increase in the number of memory requests or an increase in the number of instructions executed.

2) Burstyness of Memory Traffic: To understand the nature of memory contention, we profiled the memory access patterns. Using a very fine grained sampler we have developed, we measure the number of last-level cache misses that occur every five microseconds. This allows us to measure the burstiness of memory accesses over time. The sample size of five microseconds gives very good resolution of the lifetime of the applications, but has minimal impact on intrusiveness. The difference between the number of last level cache misses incurred when using the profiler and when running without profiler is less than 3%.

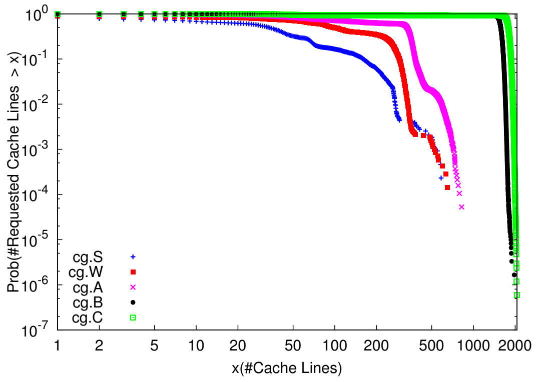
The second objective of our experiments is to analyze the relationship between problem size and the burstiness of memory traffic. To study this, we measure the burstiness of last-level cache misses over time. Table III shows the input size for both problems. Figure 4 shows the burstiness of off-chip

Program and Size	**Problem Size Description**
CG.S	matrix of size $1,400^2$
CG.W	matrix of size $7,000^2$
CG.A	matrix of size $14,000^2$
CG.B	matrix of size $75,000^2$
CG.C	matrix of size $150,000^2$
x264.simsmall	8 frames at 640 x 360
x264.simmedium	32 frames at 640 x 360
x264.simlarge	128 frames at 640 x 360
x264.native	512 frames at 1,920 x 1,080

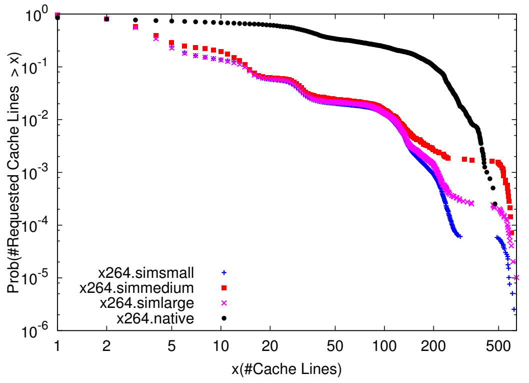
TABLE III
PROBLEM SIZE DESCRIPTION FOR CG AND X264

memory traffic for programs CG and x264, each for a selection of problem size ranging from small and large. Program CG determines the largest eigenvalues of a sparse matrix, while x264 performs H.264 video encoding for different frame numbers and resolutions.

Fig. 4 shows the burstiness of the memory traffic for both programs, on Intel NUMA using 24 threads and 24 cores. The graph in log-log scale plots $P(Burst\ Size > x)$, the probability that the memory burst size exceeds the number of cache lines x, for different sizes of cache lines. The plot



(a) HPC Dwarf: CG



(b) PARSEC: x264

Fig. 4. Burstiness of Off-Chip Memory Traffic

shows that the size of memory requests varies widely, ranging from four to seven orders of magnitude from small to large program sizes. However, the small (S and W for CG, sim-* for x264) and large problem sizes (B and C for CG, native for x264) behave quite differently. In small problem size, for both programs the long tail property of the distribution of burst size is prominent. For bursts larger than 50 cache lines, $log\ P(BurstSize > x)$ decreases linearly with $log\ x$ with the log of burst size in approximatly a diagonal straight line. This confirms that the traffic is highly bursty, which is in line with previous observations about the nature of memory traffic [13]. However, as the problem size increases, the deviation from a decreasing diagonal line becomes more clear: and for large problem sizes B and C in program CG the long tail property is absent. This means that memory traffic for program CG is not significantly bursty. The intuitive explaination behind this observation is that large problem size B and C the memory bandwidth is saturated and therefore there are no significant time intervals without memory requests. The same trend of decreasing burstiness when problem size increases was observed for all programs with significant memory contention (CG, FT and SP). The results on the other two systems, Intel UMA and AMD NUMA are roughly similar.

In conclusion, our experiments show two types of memory contention behavior with respect to problem size:

1) Small problem sizes lead to small contention for off-chip resources but result in highly bursty traffic.
2) Large problem sizes can lead to non-bursty memory traffic but results in large off-chip memory contention among cores.

Motivated by these results, we next propose an analytical model for programs with large memory contention.

IV. Proposed Analytical Model

The aim of our proposed analytical model is to relate the behavior of off-chip memory contention to the number of cores and the problem size in multicore systems. From small problem size, the contention is close to zero. As such, we focus our model on large problem sizes that generate significant contention.

Let $C(n)$ denotes the total number of cycles required to execute a program on n homogeneous cores. $C(n)$ can be expressed as

$$C(n) = W(n) + B(n) + M(n) \tag{1}$$

where $W(n)$ is the total number of work cycles, $B(n)$ is the total number of stall cycles incurred for non off-chip memory contention such as cache hits, pipelines hazards, branch mispredictions and uncontented memory accesses, and $M(n)$ is stall cycles due to off-chip memory contention. For sequential or single-thread execution, there is no off-chip resource contention among cores, and thus $M(1)$ is zero. Since the number of work cycles, $W(n)$, does not grow with the number of cores, as discussed in section III-B, we hypothesize that $B(n)$ does not grow with the number of cores. Because intuitively cache hits, pipeline hazards and branch mispredictions are independent of off-chip resource contention. Therefore, it does not matter on how many cores these stall cycles are divided, since the total number remains the same. Thus,

$$M(n) = C(n) - B(n) - W(n) = C(n) - C(1) \tag{2}$$

Definition 1 (Degree of Memory Contention). *The degree of memory contention for a program running on n homogeneous cores, $\omega(n)$, is defined as the total number of stall cycles incurred due to memory contention over the total number of cycles required to execute the program without contention:*

$$\omega(n) = \frac{M(n)}{C(1)} \tag{3}$$

$$\omega(n) = \frac{C(n) - C(1)}{C(1)} \tag{4}$$

When $\omega(n)$ is zero, the program has no memory contention, and values greater than zero measures the degree of contention. Positive memory contention due to the effect of the cache is exposed when $\omega(n)$ is smaller than zero.

To model a system with multiple processors, we apply hierarchical decomposition in two steps. Firstly, we model the increased in the number of cycles within one processor. As observed from our experimental analysis, the memory traffic for large problem size is non-bursty. Therefore, we can apply a M/M/1 queuing model [11]. Next, we model the effect of scaling to multiple processors.

For one processor with n cores, assume arrival requests from different cores are independent and identically distributed. Let $C_{req}(n)$ denote the average number of CPU cycles required to service one off-chip memory request. Let λ denote the arrival rate of memory requests to the memory controller, and μ the service rate of the memory controller. In UMA, the memory controller is shared among processors and in NUMA each processors has its own memory controllers. From the M/M/1 model,

$$C_{req}(n) = \frac{1}{\mu - \lambda} \tag{5}$$

Let L be the arrival rate of memory requests generated by one core. When n cores are active, $\lambda = nL$. If $r(n)$ denotes the number of last-level cache misses,

$$C(n) = r(n)C_{req}(n) = \frac{r(n)}{\mu - nL} \tag{6}$$

Equation 6 reveals that the number of cycles increases with the number of active cores in a processor. Furthermore, in programs with larger memory requirements and where memory request arrival rate is high, the number of stall cycles increases.

To model multiple processors, we present a model based on its memory interconnection architectures. In UMA, each processor issues memory requests through its dedicated bus to access off-chip memory. Thus, the queuing time for different buses is independent, and the main contention is for the shared-memory controller. In a dual processors system with n_1 cores active in the first processor, and n_2 in the second processor, the total number of cycles is

$$C_{UMA}(n) = C(n_1) + C(n_2) + \Delta C \tag{7}$$

where ΔC accounts for the difference in the number of cycles due to the increase in the load on the memory controller.

Assume a *fill-processor-first* policy for increasing the number of cores. When c cores are active on the first processor and one core on the second processor, $\Delta C = C(c+1) - C(c)$. Generalizing, if c cores are active in one processors, and $n - c$ in the next processors, the number of cycles required to execute a program on n cores is

$$C_{UMA}(n) = C(c) + C(n - c) + \Delta C \tag{8}$$

In NUMA systems, there are two main types of off-chip memory requests. Local memory requests proceed to the local memory controller, while requests to remote memory belonging to other processors are delivered through the memory interconnection network. Remote memory requests are slower because these requests travel travel across interconnection networks to reach the memory. In multiprocessor systems running programs with large memory requirements, the stall cycles due to memory requests are larger than the work cycles. Since memory request stall cycles dominates, work cycles can be interleaved and the number of cycles for a program on NUMA is

$$C_{NUMA}(n) = r(n)\big(C_{req}(n) + \delta(n)\big) \tag{9}$$

where $\delta(n)$ is the number of additional stall cycles incurred by a remote memory request. Assume memory affinity is homogeneous among threads. Given n cores, with c cores allocated on one processor and with $n-c$ on the next processor, memory accesses can be divided proportionally between the two processors with $\frac{c}{n}$ on the first and $\frac{n-c}{n}$ on the second. Thus, $C_{NUMA}(n)$ is the sum of the cycles incurred due to local and remote memory accesses:

$$C_{NUMA}(n) = r(n)\Big(\frac{c}{n}C_{req}(c) + \frac{n-c}{n}\delta(n)\Big) \tag{10}$$

and $C_{NUMA}(n)$ can be expressed as

$$C_{NUMA}(n) = C(c) + r(n)\rho(n - c) \tag{11}$$

where $\rho = \frac{\delta(n)}{n}$, is the average number of stall cycles spent on remote memory requests per core. This assumes that remote memory requests are proportionally distributed among the n cores. For a system with multiple memory latencies (such as AMD NUMA), ρ is a average weighted to the number of memory requests to each of the remote memories.

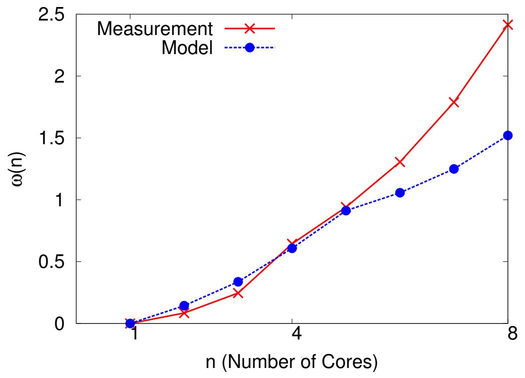
To derive $C(n)$ in equation 6, we apply linear regression to obtain the parameters μ and L. This requires two measurement runs to determine the total number of cycles for the program executed on one core, and another run using two or more cores. For multiple processor systems, $C_{UMA}(n)$ in equation 8 and $C_{NUMA}(n)$ in equation 11 are also derived from linear regression of ΔC and ρ, respectively. Subsequently, degree of memory contention, $\omega(n)$, as in equation 4 is obtained.

V. Analysis

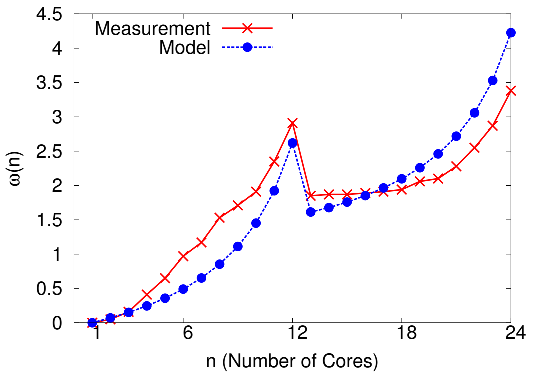
In this section, we apply our analytical model to study the effects of increasing the number of cores and the problem size on memory contention. Due to space constraint, we discuss the validation of our model against measurement together with the analysis.

We discuss the validation and accuracy of our model against measurements before analyzing the effects of varying the number of cores. We show detailed validation results on a program with large contention, CG.C and another with small contention, EP.C. We also provide summary validation results for all applications.

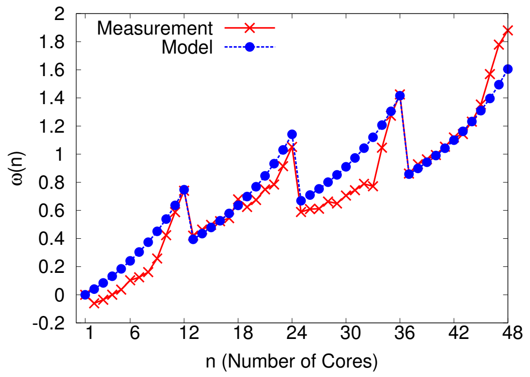
Fig. 5 shows the comparison between modeled and measured degree of memory contention for program CG.C using a fill-processor-first-policy. The average relative error across all measured and predicted model results is 5-14% on all three systems. For Intel UMA we use three measured values of $C(n)$ to apply the model: $C(1)$, $C(4)$ and $C(5)$ and achieve average accuracy of 6%. For AMD NUMA, we use five measured values as inputs: $C(1)$, $C(12)$, $C(13)$, $C(25)$ and $C(37)$ and achieve the best accuracy with error less than 5% across all problems with large contention. For AMD NUMA, we could use three values, $C(1)$, $C(12)$, $C(13)$ and assume that all interconnects are homogeneous, but this degrades the prediction accuracy up to 25% average relative error, because



(a) Intel UMA: Xeon E5320



(b) Intel NUMA: Xeon X5650



(c) AMD NUMA: Opteron 6172

Fig. 5. High Contention: Effects of Increasing the Number of Cores on CG.C

using three inputs would assume that the remote memory latencies are homogeneous, which is not valid on our AMD NUMA system. On Intel NUMA, we use four measured values of $C(n)$ for the regression: $C(1)$, $C(2)$, $C(12)$ and $C(13)$ and the model reaches accuracy of 11% average error. Using only three values for regression – $C(1)$, $C(12)$ and $C(13)$ on Intel NUMA slightly increases the inaccuracy of the prediction to an average of 14% across the profiled applications. There are two main sources of is caused by two factors: (i) variability of measurement values and (ii) oversubscription effects. To counter the variability of measurement values due to the operating system scheduler moving threads across NUMA domains, we binded each thread to a specific core. This has reduced the variability of the results, but has introduced negative caching effects between the threads that share the same core. Because we fix the number of threads, but vary the number of cores, there will be more than one thread executing on each core. Due to oversubscription, there are load imbalances between the threads assigned to the same core. This leads to variability in the measured number of cycles incurred by the threads among different runs of the programs. The effects are more pronounced when the oversubscription factor (ratio of threads to cores) is large [9]. The patterns of growth observed on the other programs with large contention (FT and SP) were similar to those observed on CG.

Programs with low contention do not cause a significant increase in number of cycles when the number of active cores increases. Fig. 6 shows the modeled and measured values of contention for EP.C.

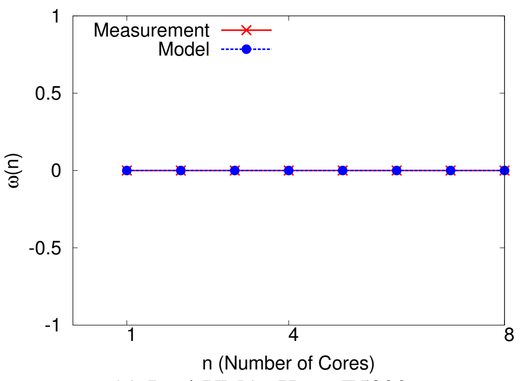
For UMA, memory contention is negligible because demand for memory that arises from more cores can be met by the cache and off-chip memory resources. However, the NUMA architectures shows two interesting trends. The effect of positive memory contention ($\omega(n) < 0$) is observed with less than 11 cores on EP.C running on Intel NUMA because adding cores also increases memory resources ($L1$ and $L2$ cache). Beyond one processor, memory contention increases to 50%, which is not captured by our model. This is caused by an increase in number of last level cache misses, from $1,800$ misses on one core to $31,000,000$ on 24 cores. Our model assumes the number of work cycles and last level misses constant. This assumptions holds for programs with large memory contention, but may not be for programs with low contention, such as EP. Furthermore, the increase in degree of contention is correlated with the latency of memory accesses.

Next we show summary results for all applications. The goodness-of-fit for determining the linearity of $\frac{1}{C(n)}$, as shown in table IV for $n = 1$ to 4 on Intel UMA, $n = 1$ to 12 on Intel NUMA and AMD NUMA, further confirms the accuracy of our model. There is a correlation between the goodness of fit R^2 and the degree of memory contention. Programs EP.C and x264.native, which show the smallest degree of contention also have lower colinearity. This confirms that the M/M/1 queueing model does not explain their behavior very well, because they are bursty. R^2 is close to 1 (i.e. perfect colinearity of $1/C(n)$) when the memory overhead is high. The accuracy of the M/M/1 model for describing the behavior of programs with large memory contention further confirms the non-bursty nature of programs with large memory contention.

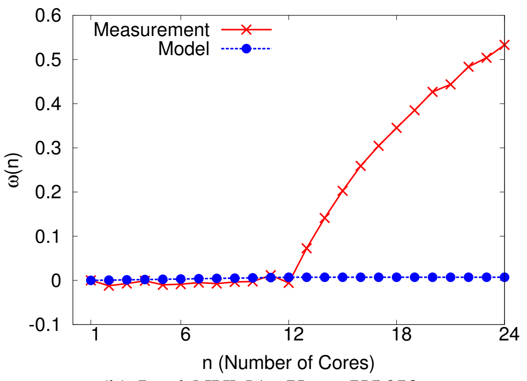
System	Goodness-of-fit, R^2, for Programs					
	EP.C	IS.C	FT.B	CG.C	SP.C	x264.native
Intel UMA	0.86	0.97	1.00	0.96	0.97	0.87
Intel NUMA	0.91	0.98	0.99	0.94	0.96	0.85
AMD NUMA	0.90	0.99	1.00	0.97	0.99	0.81

TABLE IV
COLINEARITY GOODNESS-OF-FIT FOR PROGRAMS

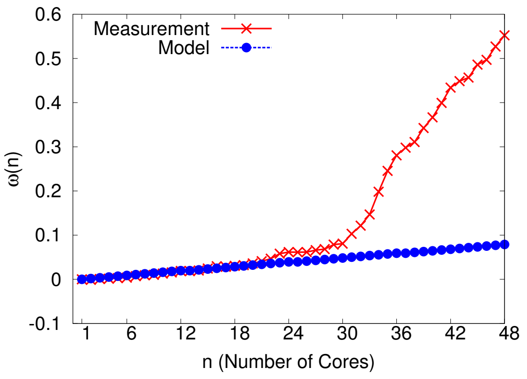
We present an analysis of the growth of memory contention for CG.C. The program exhibits high degree of memory



(a) Intel UMA: Xeon E5320



(b) Intel NUMA: Xeon X5650



(c) AMD NUMA: Opteron 6172

Fig. 6. Low Contention: Effects on Increasing the Number of Cores on EP.C

contention of 1.8 to 3.3 times as compared with a sequential execution. On Intel UMA, the contention closely follows how many cores are used in each processor. For one to four cores, the increase in $\omega(n)$ is due to contention of the shared bus, since all cores within one processor share the same memory bus. From four to five cores, the increase in contention is small, since memory requests by the fifth core, which is allocated in a new processor, uses the bus of the new processor. When the buses and the memory controllers in both processors reach maximum load, contention is most severe as can be seen from increasing the number of cores from seven to eight. On both Intel NUMA and AMD NUMA, the degree of memory contention is smaller than on UMA for similar number of cores. However, the pattern of growth still have a per-processor shape. On Intel UMA, from one to twelve cores, $\omega(n)$ increases non-linearily, which shows that the memory controller of the first processor becomes saturated. When the thirteenth core is activated, in processor two, the memory controller of processor two takes over a fraction of the memory requests from processor one controller, reducing the contention. This is why there is a sharp decrease in $\omega(n)$ from twelve to thirteen. There are other reasons that lead to better NUMA performance such as the larger cache size, faster bus speed and larger memory bandwidth. Overall, the programs that show a larger degree of memory contention on UMA also manifest large contention on NUMA.

Memory contention can be broadly characterized as low, as shown in Fig. 6 and high, as in Fig. 5. However the mapping between problem size and degree of contention is not bijective. Low problem size results in low contention for all programs analyzed by us. This is due to the size of the working set which is of comparable size to the caches of the system. However, for large problem size, there are two cases. In the first case, EP.C and x264.native have large working set (920 MB for EP, 400 MB for x264), much larger than the cache of the system, yet do not result in large contention. This is because their pattern of accessing the memory results in low number of cache misses and therefore their performance does not depend significantly on the memory bandwidth. In contrast, the second case, of CG, FT, and SP, their large problem size also translates in large contention. The program with the largest observed contention, the pentadiagonal-solver SP access memories along all dimensions of a 3D space. Such complex data access patterns leads to large number of cache misses. This results in SP.C having the largest values of contention, with $\omega(n)$ reaching 7.1 on eight cores on Intel UMA and 11.6 on 24 cores on Intel NUMA.

VI. Conclusions

This paper presents an analytical model for understanding memory contention of shared-memory programs in both uniform and non-uniform-memory access multicore systems. Our model is inspired by a series of observations derived from experiments on state-of-the-art UMA and NUMA systems using 8, 24 and 48 cores. In constrast with previously reported behavior of memory requests, which assumed that memory traffic is always bursty, we discovered that the burstiness of memory traffic depends on the problem size. Programs with large sizes and high memory requirements lead to large memory contention factors but have non-bursty memory traffic. Based on the observations, we proposed an analytical queueing model for programs with large contention in both UMA and NUMA multiprocessor systems. Our model is validated against measurements using two representative parallel program benchmarks, NPB and PARSEC. While the model is simple and based on a high-level abstraction of multicore systems, it has high accuracy and differs from measurements by 5-14% for problems with large contention, in the range of problem sizes and number of cores used in the experiments.

Our analysis addresses the impact of increasing the number of cores and the problem size. In a multiprocessor system, increasing the number of cores generally also increases memory contention, as anticipated. However, contention increases very slightly for small problem sizes, with maximum observed increase of around 50% on EP.C when using 48 cores on a quad-processor AMD NUMA system. For programs with large memory requirements, memory contention among cores becomes significant as the number of active cores is increased, with a peak of more than 1000% increase in program SP.C on 24 cores on an Intel NUMA system. The burstiness of memory traffic is also affected by the problem size. Small problems benefit the most from the cache memory, and access main memory in seldom bursts. When problem size increases, if the patterns of cache access result in significant cache misses, we observed more frequent and more sustained accesses to main memory, which result in non-bursty traffic.

There are two main limitations in our model, namely, high-level model abstraction and its decrease in accuracy for programs with low degrees of contention. To account for contention at the next level of system details, the model can be extended, at the expense of higher modeling cost, to factor in bus speed and bandwidth, memory size and bandwidth, number of memory channels, service-discipline of memory controllers, among others. The second limitation is our decrease in accuracy for programs with low working sets and low memory requirements. Some of the observations regarding the independence of last level cache misses and work cycles to the number of active cores may not hold for programs with small working sets. When applied to such programs, such as x264 or EP, the assumption of memory requests being independent and identically distributed no longer holds, leading to lower accuracy of our M/M/1-based model. However, the usefulness of the model increases for programs with large memory requirements, and for this case the model has good accuracy.

References

[1] K. Asanovic et al., The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] D. H. Bailey et al., The NAS Parallel Benchmarks – Summary and Preliminary Results, *Proc. of ACM/IEEE Conference on Supercomputing*, pages 158–165, Albuquerque, USA, 1991.

[3] C. Bienia et al., The PARSEC Benchmark Suite: Characterization and Architectural Implications, *Proc. of 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, Toronto, Canada, 2008.

[4] S. Cho, L. Jin, Managing Distributed, Shared L2 Caches through OS-level Page Allocation, *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Orlando, USA, 2006.

[5] A. Fedorova, M. Seltzer, M. D. Smith, Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler, *Proc. of 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, Brasov, Romania, 2007.

[6] E. Haritan et al., Multicore Design Is the Challenge! What Is the Solution? *Proc. of 45th Annual Design Automation Conference*, pages 128–130, Annaheim, USA, 2008.

[7] A. Herdrich et al., Rate-based QoS Techniques for Cache/Memory in CMP Platforms, *Proc. of 23rd International Conference on Supercomputing*, pages 479–488, Yorktown Heights, USA, 2009.

[8] R. Hood et al., Performance Impact of Resource Contention in Multicore Systems, *Proc. of 24th International Symposium on Parallel & Distributed Sytems*, Atlanta, USA, 2010.

[9] C. Iancu et al., Oversubscription on Multicore Processors, *Proc of 24th International Symposium on Parallel & Distributed Processing*, Atlanta, USA, 2010.

[10] E. Ipek et al., Self-optimizing Memory Controllers: A Reinforcement Learning Approach, *Proc. of 35th Annual International Symposium on Computer Architecture*, pages 39–50, Beijing, China, 2008.

[11] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[12] T. Karkhanis, J. E. Smith, A Day In the Life of a Data Cache Miss, *Proc. of 2nd Workshop on Memory Performance Issues*, Anchorage, USA, 2002.

[13] Y. Kim et al., ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers, *Proc. of 16th International Symposium on High Performance Computer Architecture*, Bangalore, India, 2010.

[14] W. E. Leland et al., On the Self-similar Nature of Ethernet Traffic (Extended Version), *IEEE/ACM Transactions on Networking*, 2:1–15, 1994.

[15] C. Liu, A. Sivasubramaniam, and M. Kandemir, Organizing the Last Line of Defense Before Hitting the Memory Wall for CMP, *Proc. of 10th International Symposium on High Performance Computer Architecture*, Madrid, Spain, 2004.

[16] F. Liu et al., Understanding how Off-chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance, *Proc. of 16th International Symposium on High Performance Computer Architecture*, Bangalore, India, 2010.

[17] O. Mutlu, T. Moscibroda, Parallelism-aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems, *Proc. of 35th Annual International Symposium on Computer Architecture*, pages 63–74, Beijing, China, 2008.

[18] K. J. Nesbit et al., Fair Queuing Memory Systems, *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Orlando, USA, 2006.

[19] K. J. Nesbit, J. Laudon, J. E. Smith, Virtual Private Caches, *Proc. of 34th Annual International Symposium on Computer Architecture*, pages 57–68, San Diego, USA, 2007.

[20] K. Park, W. Willinger, *Self-Similar Network Traffic and Performance Evaluation*, John Wiley & Sons, Inc., New York, USA, 1st Edition, 2000.

[21] M. K. Qureshi, Y. N. Patt, Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches, *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Orlando, USA, 2006.

[22] B. M. Rogers et al., Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling, *Proc. of 36th Annual International Symposium on Computer Architecture*, pages 371–382, Austin, USA, 2009.

[23] J. C. Sancho, D .Kerbyson, M. Lang, Analyzing the Trade-off between Multiple Memory Controllers and Memory Channels on Multi-core Processor Performance *Prof. of Workshop on Large-Scale Parallel Processing*, Atlanta, USA, 2010.

[24] V. Suhendra, T. Mitra, Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores, *Proc. of 45th Annual Design Automation Conference*, pages 300–303, Annaheim, USA, 2008.

[25] J. Treibig, G. Hager, G. Wellein, LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, *Proc. of 1st International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego, USA, 2010.

[26] B. M. Tudor, Y. M. Teo, A Practical Approach for Performance Analysis of Shared-Memory Programs, *Proc. of 25nd International Parallel & Distributed Processing Symposium*, Anchorage, USA, 2011.

[27] Y. Xie, G. Loh, Dynamic Classification of Program Memory Behaviors in CMPs, *Proc. of 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Beijing, China, 2008.

[28] S. Zhuravlev, S. Blagodurov, A. Fedorova, Addressing Shared Resource Contention in Multicore Processors via Scheduling, *Proc. of 15th Edition of Architectural Support for Programming Languages and Operating Systems*, pages 129–142, Pittsburgh, USA, 2010.