Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IMPACT FACTOR: 5.258

IJCSMC, Vol. 5, Issue. 3, March 2016, pg.256 – 264

An Analytical Model for Optimum Off-Chip Memory Bandwidth Partitioning in Multi-core Architectures

Miss. Priyanka V P[1]
[1]M.Tech Student, Department of Computer Science and Engineering,
New Horizon College of Engineering, Bangalore, Karnataka, India
[1] priyankavp38@gmail.com

Miss. K. Pramilarani[2]
[2]Senior Assistant Professor, Department of Computer Science and Engineering,
New Horizon College of Engineering, Bangalore, Karnataka, India
[2] pramiselva@yahoo.co.in

Abstract--- Multi core architectures have recently become a mainstream computing platform. These architectures along with excellent compute power, also facilitates sharing of expensive resources such as the last level cache and off-chip memory bandwidth between the cores. Such sharing has produced interesting contention-related performance phenomena, in which system throughput, as well as individual thread performance is highly volatile, depending on the mix of applications that share the resources and on how the resources are partitioned between the applications. If the sharing of these resources is made optimal, then the performance volatility of individual threads running in different cores can be reduced and obviously the overall system performance can be increased. Thus partitioning of resources shared by multi-core architectures such as cache and off-chip pin bandwidth, is necessary to a great extent. Applying cache partition alone, definitely improves the system performance. The proposed simple yet powerful analytical model shares the off-chip memory bandwidth in such a way that it have a progressive impact over the performance of the system. The optimal bandwidth sharing in proposed model is achieved by exploiting the advantage of token bucket algorithm. The memory controller is modified in such a way that apart from normal memory request queuing, it also implements a token bucket system which controls memory requests from every core. The results are calculated based on weighted speedup which is a factor for measuring throughput in multi core architectures.

I. INTRODUCTION

Resource sharing is a very important factor in multicore architectures for the considerable increase in overall throughput. But the sharing of expensive resources has produced an interesting contention-related performance phenomenon, in which system throughput, as well as individual thread performance is highly volatile, depending on the mix of applications that share the resources and on how the resources are partitioned between the applications. The growing number of cores on a chip increases the degree of sharing of last level cache and off-chip bandwidth. If the sharing of these resources are made optimal, then the performance volatility of individual threads running in different cores can be reduced and obviously the overall system performance can be increased. Thus partitioning of resources shared by multi-core architectures such as cache and off-chip pin bandwidth, is necessary to a great extent. The main objective of the work is to provide optimum partitioning of memory bus bandwidth between multiple cores of the multiprocessor. This kind of partitioning is obtained by implementing a token bucket system in between L2 cache and the memory bus. The memory controller not only processes various memory requests but also control bandwidth utilization between memory requests. The memory requests from different cores enter the token bucket system where it takes a token generated in periodic intervals. No of buckets present in the token bucket system would be equal to the number of cores present in the CMP. Requests from every core enter to their corresponding bucket and then supplied with a token from token generator. The generation of tokens is proportional to the amount of bus bandwidth. Thus restricting any single core from dictating the entire bus. With this partitioning scheme every core gets its own dynamic share of bandwidth according to the current traffic condition. Also the bandwidth utilization is very efficient especially when the bus bandwidth is low. The proposed model is designed in simics simulator where a existing processor model is taken and its memory controller is modified such that it controls memory request traffic to bus bandwidth apart from normal memory request processing. Coherency details between applications running in the modified processor are ignored for simplicity of design.

The remainder of this paper is organized as follows: Section II reviews the work related to partitioning of resources including various analysis work which projects the importance of bandwidth partitioning, Section III describes the proposed work which is to make an optimal partitioning of off-chip memory bandwidth, Section IV describes the overall design and implementation of the proposed memory bandwidth partitioning model, Section V discusses the result and analysis of the performance enhancement acquired by use of optimal bandwidth partitioning. Section VI concludes the paper.

II. **RELATED WORK**

Off-Chip Memory Bandwidth Partitioning in CMP: How memory bandwidth utilization in last level can affect system performance has become an important factor in the current situation of multicore architecture where memory bandwidth is considerably low. Recent CMPs allow cores to share expensive resources, such as the last level cache and off-chip pin bandwidth. To improve system performance and reduce the performance volatility of individual threads, last level cache and off-chip bandwidth partitioning schemes have been proposed. While how cache partitioning affects system performance is well understood, little is understood regarding how bandwidth partitioning affects system performance, and how bandwidth and cache partitioning interact with one another. Liu et al [1] proposes a effective analysis for bandwidth partitioning based on queuing model which sketches the major factors having an impact over off-chip memory bandwidth in multiprocessor environment.

Effective Management of DRAM Bandwidth: Technology trends are leading to increasing number of cores on chip. All these cores inherently share the DRAM bandwidth. The on-chip cache resources are limited and in many situations, cannot hold the working set of the threads running on all these cores. This situation makes DRAM bandwidth a critical shared resource. Existing DRAM bandwidth management schemes provide support for enforcing bandwidth shares but have problems like starvation, complexity, and unpredictable DRAM access latency. Rafique et al [2] propose a DRAM bandwidth management scheme with two key features. First, the scheme avoids unexpected long latencies or starvation of memory requests. It also allows OS to select the right combination of performance and strength of bandwidth share enforcement. Second, it provides a feedback-driven policy that adoptively tunes the bandwidth shares to achieve desired average latencies for memory accesses. This feature is useful under high contention and can be used to provide performance level support for critical applications or to support service level agreements for enterprise computing data centers.

Cache Sharing and Partitioning: Fairness is a critical issue because the Operating System (OS) thread scheduler"s effectiveness depends on the hardware to provide fair cache sharing to co-scheduled threads. Without such hardware, serious problems, such as thread starvation and priority inversion, can arise and render the OS scheduler ineffective. Nesbit et al [3] proposes a model of cache partitioning where a concept of virtual private cache is used. Suh et al [4] [5] presents a detailed study of fairness in cache sharing between threads in a chip multiprocessor (CMP) architecture where partitioning is done with a base of memory-aware scheduling of various types of requests from hardware context. Prior work in CMP architectures has only studied throughput optimization techniques for a shared cache. The issue of fairness in cache sharing, and its relation to throughput, has not been studied. Kim et al [6] found that optimizing fairness usually increases throughput, while maximizing throughput does not necessarily improve fairness. Using a set of co-scheduled pairs of benchmarks, on average these algorithms improve fairness by a factor of 4, while increasing the throughput by 15%, compared to a non-partitioned shared cache. This model gives a cache sharing methodology which is static. Qureshi et al [7] proposes a low-overhead, runtime mechanism that partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. Unlike the cache partitioning schemes mentioned above, Chang et al [8] proposes a cache partitioning model which resolves cache contention with multiple time sharing partitions. Time-sharing cache resources among partitions allows each trashing thread‟s capacity allocations, while improving fairness by giving different partitions equal chance to execute. Also cache sharing impacts threads nonuniformly, where some threads may be slowed down significantly, while others are not. This may cause severe performance problems such as sub-optimal throughput, cache thrashing, and thread starvation for threads that fail to occupy sufficient cache space to make good progress Chandra et al [9] proposes three performance models that predict the impact of cache sharing on co-scheduled threads.

Interactions Between Compression and Prefetching in Chip Multiprocessors: In chip multiprocessors (CMPs), multiple cores compete for shared resources such as on-chip caches and off-chip pin bandwidth. Stride-based hardware prefetching increases demand for these resources, causing contention that can degrade performance (up to 35% for one of our benchmarks). Alameldeen et al [10] propose a simple adaptive prefetching mechanism that uses cache compressions extra tags to detect useless and harmful prefetches. Furthermore, in the central result of this paper, it is shown that compression and prefetching interact in a strong positive way, resulting in combined performance improvement of 10-51% .

Prefetch-Aware DRAM Controllers: Existing DRAM controllers employ rigid, non-adaptive scheduling and buffer management policies when servicing prefetch requests. Some controllers treat prefetch requests the same as demand requests, others always prioritize demand requests over prefetch requests. However, none of these rigid policies result in the best performance because they do not take into account the usefulness of prefetch requests. If prefetch requests are useless, treating prefetches and demands equally can lead to significant performance loss and extra bandwidth consumption. In contrast, if prefetch requests are useful, prioritizing demands over prefetches can hurt performance by reducing DRAM throughput and delaying the service of useful requests. Lee et al [11] proposes a new low-cost memory controller, called Prefetch-Aware DRAM Controller (PADC), that aims to maximize the benefit of useful prefetches and minimize the harm caused by useless prefetches. To accomplish this, PADC estimates the usefulness of prefetch requests and dynamically adapts its scheduling and buffer management policies based on the estimates. The key idea is to 1) adaptively prioritize between demand and prefetch requests, and 2) drop useless prefetches to free up memory system resources, based on the accuracy of the prefetcher. Evaluation shows that PADC significantly outperforms previous memory controllers with rigid prefetch handling policies on both single- and multi-core systems with a variety of prefetching algorithms. Across a wide range of multiprogrammed SPEC CPU 2000/2006 workloads, it is shown to improve system performance by 8.2%on a 4-core system and by 9.9%on an 8-core system while reducing DRAM bandwidth consumption by 10.7% and 9.4% respectively.

Coordinated Cache and Bandwidth Partitioning: Bitirgen et al. [12] proposed a global resource manager that uses Artificial Neural Networks (ANNs) to manage the allocation of shared cache capacity and off-chip bandwidth. The study showed that coordinated cache and bandwidth partitioning can outperform unmanaged cache and bandwidth partitioning. As ANN is a black box optimizer, fundamental insights of how cache and bandwidth partitioning interact remain undiscovered.

III. **BANDWIDTH PARTITIONING MODEL**

The architectural design for optimum memory bandwidth partitioning in a CMP environment involves in modifying the exiting architecture of the 21174 single-chip memory controller [13] which forms the interface between memory bus and 2 core Alpha CMP. By bandwidth partitioning, it is specifically referred to allocating shares (fractions) of off-chip bandwidth between different cores, without changing the memory controller scheduling policy. Note that naively allocating fixed fractions of off-chip bandwidth to different cores is risky because it can degrade, instead of improve, system performance, for two reasons. First, if a thread is allocated more fraction of bandwidth than it needs, the allocated bandwidth will be underutilized, while at the same time other threads may be penalized due to insufficient bandwidth allocation. Secondly, bandwidth usage is often bursty, hence a fixed partition can significantly hurt performance during bursts of memory requests, and suffer from fragmentation during a period of sparse memory requests. To allocate fractions of off-chip bandwidth to different cores while tolerating the bursty nature of memory requests, a bandwidth partitioning scheme is implemented using a token bucket algorithm, a bandwidth partitioning algorithm borrowed from computer networking [14]. The memory controller is modified in such way that form normal processing of memory requests from different cores, it also restricts any single core from dictating the entire memory bus. The memory controller sequences the memory requests with the help of a sequencer module, connected to any number of DIMMs with any number of DRAMs each, which reserves and dispatches memory requests of any type to be placed in the system bus. The sequencer queues and dequeues memory requests in a first come first serve manner. The memory controller also includes mechanism for prioritizing long waiting requests in order to avoid starvation over a long period of time. The performance enhancement acquired by the proposed system is measured in terms of weighted Speedup (WS) which is sum of individual CPI slowdowns [15].

Token Bucket System: The token bucket model in the chip- multiprocessor environment is placed inside memory controller in such a way that memory requests enters sequencer only with the matching token. A separate context-aware token generator is placed which generates and distributes tokens to different buckets at rates proportional to fractions of bandwidth allocated to every core of a CMP. Unused tokens are accumulated in the buckets and are consumed in the future during a period of bursty memory requests. The extent of burstiness can be determined by the depth of the bucket. When tokens for a core are discarded because the bucket is full, they are lost forever, hence the core will never reach the allocated bandwidth fraction. This model also restricts any single core from dictating the entire bandwidth at any instant which results in starvation for the requests from the other hardware-context. The proposed model is designed as below.

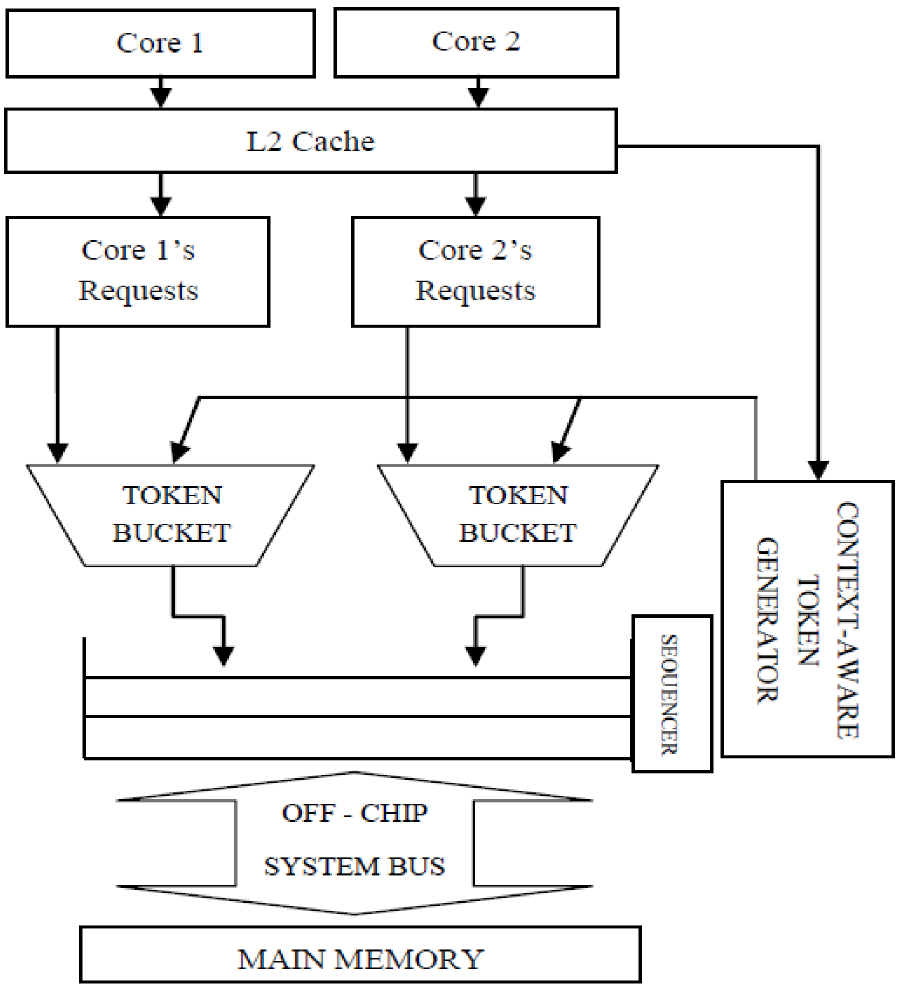

Fig. 1. Off-Chip Bandwidth Partitioning with Token Bucket Scheme.

Since the nature of applications running in the CMP environment is much more uncertain, predicting the fraction of bandwidth to every core is impossible and unstable which may lead to degradation of overall performance of the multiprocessor. Thus the dynamic off-chip memory bandwidth partitioning is facilitated by means of the Context-Aware Token Generator which distributes tokens to every bucket according to the need of every core present in the CMP. As mentioned earlier the fraction of bandwidth allocated to a core depends on the „need of bandwidth". This knowledge of „need of a bandwidth" for every core is presented to the context-aware token generator by monitoring the miss frequency of a particular thread continuously [1]. In this case, now the token generator implementation has two functionalities, one to distribute tokens to the buckets and to monitor the miss frequency ratio of a particular thread.

The bandwidth fraction to be allocated to any core directly depends on the L2 miss frequency of the particular thread [1].

TABLE I

M i	Thread i‟s L2 cache miss rate
Ai	Thread i‟s L2 cache access frequency (no of access per second)
MFi	Thread i‟s miss frequency
Bi	Fraction of Bandwidth allocated to Thread i.

PARAMETERS USED IN OUR MODEL.

The miss frequency can be calculated as the product of L2 cache miss rate and L2 cache access frequency [1].

$$MFi = M\ i.\ Ai$$

The Bandwidth fraction for any particular thread is decided by the context-aware token generator by analyzing the behavior of various threads in L2 cache level. The ratio between the miss frequencies of two different threads gives rise to the fraction of bandwidth that is to be allocated to a particular thread. The ratio decides what thread have to be biased for a particular fraction of bandwidth [1]. (i.e. a thread with greater miss frequency will get greater bandwidth share when compared to the other thread.).

$$Bi = MFi / MFj$$

The context-aware token generator collects the miss rate and access frequency of L2 cache and uses it to calculate the miss frequency of all threads running in the CMP. A table registry is maintained for every thread running, where it stores the miss frequency of every thread running separately. A comparison is made in regular intervals normally in ticks or cycles, for every miss frequency values. The ratio between the miss frequency of two threads decides which thread have to be facilitated with a little extra plus bandwidth. The fraction of bandwidth allocated is logical and not physical which is impossible too. For any thread, to have a large fraction of bandwidth, the tokens supplied to the corresponding bucket should be large when compared to the other bucket. This in turn facilitates large admission of memory requests of type, either read or write, for some amount time until the miss frequency ratio is favorable to that particular thread which is in domination with respect to other thread. The context-aware token generator triggers the token distribution system in such a way that when the miss frequency is greater than 1, the fraction of bandwidth allocated more to „i" that other thread. But when the ratio is less than that of 1, thread „j" gets an upper hand according to the „*Bi*" expression above. Obviously the bias for maximum bandwidth fraction allocation towards any particular thread is not constant and is also uncertain to a great extent. The relation between the bandwidth fraction and the miss frequency ratio value is presented below.

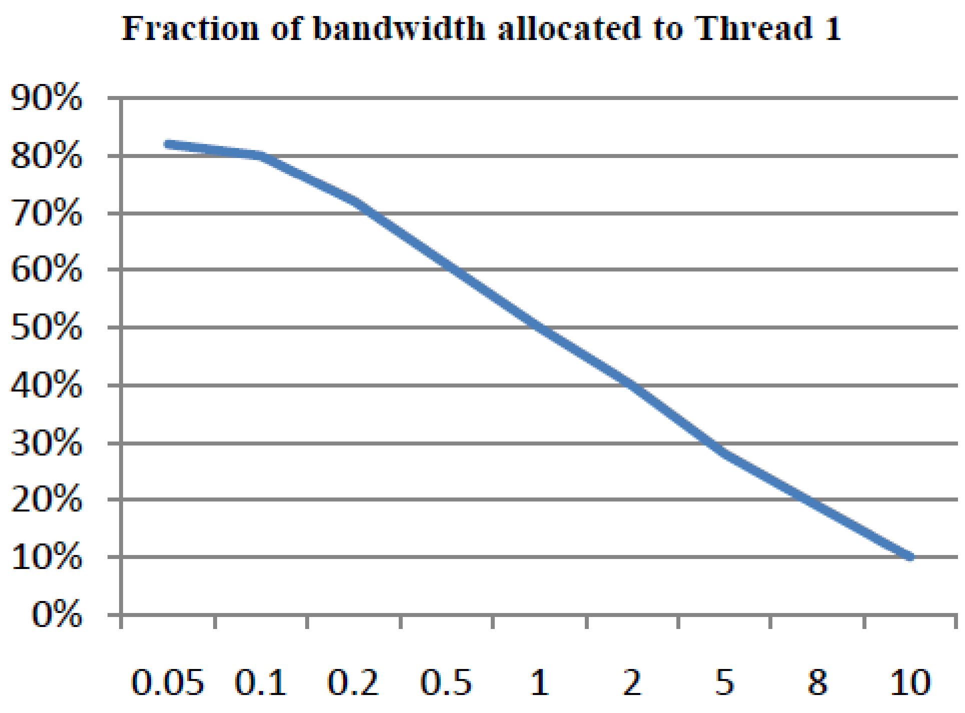

Fig.2. Fraction of bandwidth allocated to thread 1 as a function of miss frequency ratio of thread 2 to thread 1. (X-axis= miss frequency ratio; Y-axis = Bandwidth fraction allocated to thread 1).

From the above graph, it is very clear that the bandwidth fraction allocated to both cores are equal when the miss frequency ratio is 1 or both have equal miss frequency values. Also the proposed model is likely to constraint the thread with large miss frequency value. Even though it causes some slight degradation impact, our model mainly concentrates on avoiding the dictatorship of any particular core or thread and is facilitated very well, ignoring the negligible impact of performance degradation caused.
Next section describes how the proposed model is implemented.

IV. **IMPLEMENTATION**

The proposed model is designed in an event-driven full system simulator based on gem5 to model a 2-core Alpha CMP. Each core has a scalar out of order issue pipeline with a 3 GHz clock frequency. Each core has a private L1 instruction cache and L1 data cache with 16KB size, 2-way associativity, and 2-cycle access latency. The L2 cache is shared among two cores, and has 2MB size, 16-way associativity, and 8-cycle access latency. All caches have a block size of 64-byte, implement write-back policy, and use the LRU replacement policy. The bus to off-chip memory is a split transaction bus, with a peak bandwidth varied from 1.6GB/s to 25.6GB/s (with a base case of 6.4 GByte/s, in line with a DDR2-800 SDRAM channel). Variable bandwidth implementation is done just to study the impact of different range of bandwidth of the system bus. The main memory is 2GB with 240-cycle access latency. The simulator ignores the impact of page mapping by assuming each application is allocated contiguous physical memory. The CMP runs Linux OS.

The token bucket system is implemented in memory controller which forms the interface between the CPU and the main memory. Requests from each core enter to their corresponding token bucket which is then supplied with a token from a token generated by the context-aware token generator. The context aware token generator maintains a simple data structures which is a table registry to get the miss frequency values of each and every core of the CMP. The context-awareness character is acquired by the token generator, simply by calculating the miss frequency ratio of prescribed threads. According to the value of the ratio calculated, the context-aware token generator distributes the token with varying amount for the deserving token bucket, in other words to the thread with large miss frequency value. Our memory controller is designed in such a way that no memory requests enter the system bus without a matching token with it. By this way every CPU is controlled from dictating the entire memory bus bandwidth. Next section presents the visualization of the effectiveness of our proposed model.

V. **RESULT**

In order to visualize the effectiveness of or proposed model, we run several benchmark programs which are more regressed and parallel, in order put our CMP environment into complete work. **PARSEC Benchmark Suite:** For the above said purpose, we chose PARSEC benchmark suit, to construct the workload, which consisted of several data-parallel programs with which we can explore the entire functionality of the modeled CMP. The benchmarks in PARSEC use a variety of parallelization methods including pthreads, Intel TBB, and OpenMP. We use the pthreads version of all benchmarks except for freqmine, which does not have a pthreads implementation and in that case we use the OpenMP version. The applications are divided into three phases: an initial serial phase, a parallel phase, and a final serial phase. The parallel phase is called the region of interest (ROI). The parsec programs that are run in our CMP environment are Blackscholes, Bodytrack, Dedup, Ferret, Fluidanimate, Freqmine, Streamcluster, Swaptions. The set of instructions to be actually run is grouped into five inputsets which are Test, Simdev, Simsmall, Simmedium, Simlarge, and Native. We actually chose Simmedium inputset which simulates real input for a multiprocessor to work on. These programs are run separately in the modeled Alpha CMP with 2 cores and the performance statistics in terms of Weighted Speedup (WS) is taken accordingly. The performance enhancement provided by our bandwidth partitioning model is shown as below.

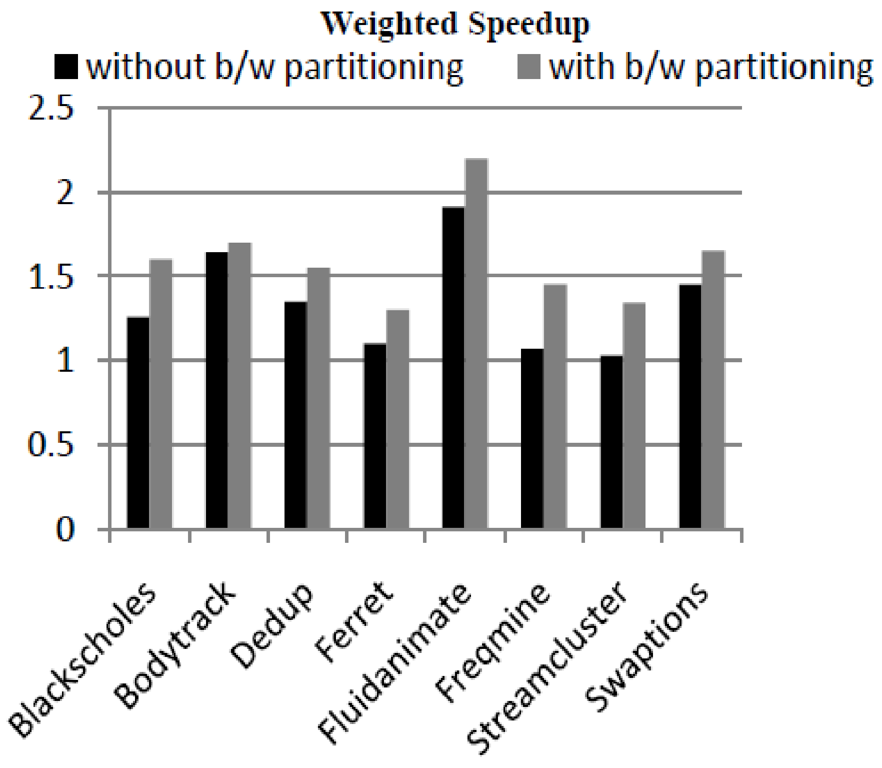

Fig. 3. Weighted Speedup of 8 applications running on a 2-core ALPHA CMP with 2 MB L2 cache.

Obviously the run time of every applications run in the modeled CMP also increases reflecting a simple visualization of performance enhancement given by our model. The run-time enhancement is visualized as below.

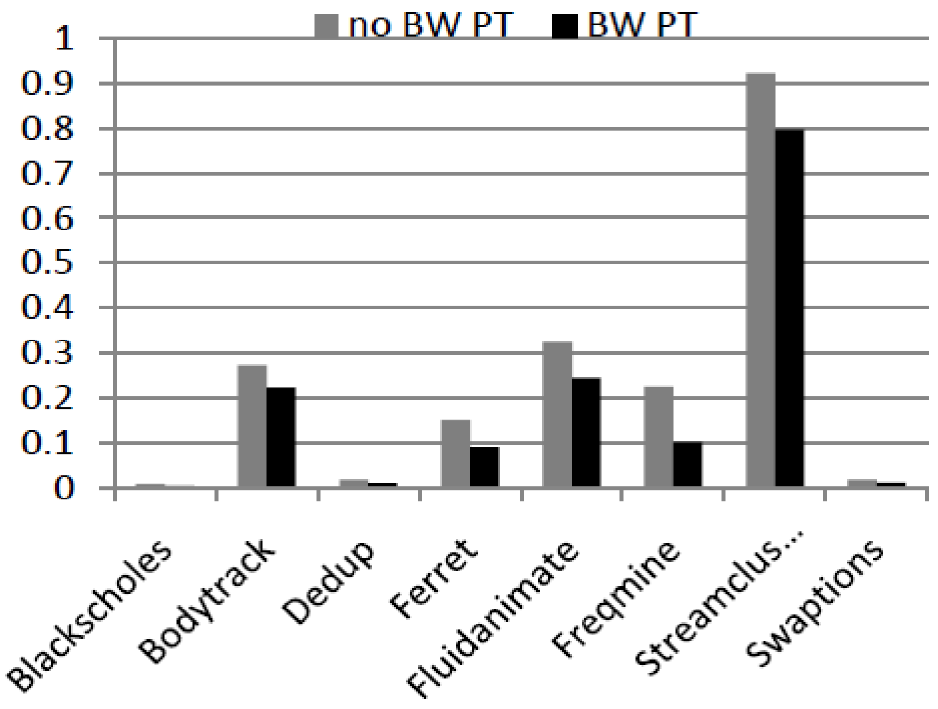

Fig. 4. Run-time visualization of 8 applications running with and without bandwidth partitioning model.

The constraint of bandwidth fraction for some threads in operation is exibhited in applications like Dedup, where there is no significant improvement or enhancement in Weighted Speedup. This forms a limitation to our model where any thread with extreme miss frequency variation with that of other thread is constrained for bandwidth fraction. This is done in order to see to that just for cause of higher miss frequency, any single core or thread cannot be allowed to dictate the entire bandwidth making the other thread to starve.

VI. CONCLUSIONS

The goal of this project is to improve overall system throughput by partitioning the memory bandwidth between multiple cores of the multiprocessor. A simple yet powerful analytical model is proposed that has achieved the goal. Constructing and studying the model leads to several subtle, yet surprising, observations. It is found that scarcer the off-chip bandwidth becomes, the more bandwidth partitioning can improve system performance significantly. Bandwidth partitioning can only improve system performance when miss frequencies between threads differ significantly. Also when the magnitude of miss frequency for a particular thread is very high, the deserved fraction of bandwidth is not given entirely to any particular core, the bandwidth allocation here is constrained in order to avoid the dictatorship of a particular thread or a core.

REFERENCES

[1] F. Liu, X. Jiang, and Y. Solihin, “Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance,” in *Proc.of the High Performance Computing and Applications (HPCA)* 2010.
[2] N. Rafique, W. Lim, and M. Thottethodi, “Effective Management of DRAM Bandwidth in Multicore Processors,” in *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques(PACT)*, 2007.
[3] K. Nesbit, J.Laudon, and J. Smith, “Virtual Private Caches,” in *Proc.of the 34th International Symposium on Computer Architecture (ISCA), 2007*, pp. 57–68.
[4] G. Suh, S. Devadas, and L. Rudolph, “A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning,” in *Proc. Of International Symposium on High Performance Computer Architecture(HPCA),* 2002, pp. 117–126.
[5] G. Suh, L. Rudolph, and S. Devadas, “Dynamic Cache Partitioning of Shared Cache Memory,” *The Journal of Supercomputing*, vol. 28, no. 7–26, 2004.
[6] S.Kim, D. Chandra, and Y. Solihin, “Fair Cache Sharing and Partitioning on a Chip Multiprocessor Architecture,” in *Proc.of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.
[7] M. Qureshi and Y. Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” in *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 423–432.
[8] J. Chang and G. S. Sohi, “Cooperative Cache Partitioning for Chip Multiprocessors,” in *Proc. of International Conference on Supercomputing*, 2007, pp. 242–252.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting the Impact of Inter-Thread Cache Contention on a Chip Multiprocessor Architecture," in *Proc. of the 11th International Symposium on High Performance Computer Architecture. IEEE Computer Society, 2005,* pp. 340–351.
[10] A.R. Alameldeen and D.A. Wood, "Interactions Between Compression and Prefetching in Chip Multiprocessors," in *Proc. of the 13th International Symposium on High-Performance Computer Architecture(HPCA),* 2007.
[11] C. Lee, O. Mutlu, V. Narasiman, and Y. Patt, "Prefetch-Aware DRAM Controller," in *Proc. of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.
[12] R. Bitirgen, E. Ipek, and J. Martinez, "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach," in *Proc. of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.
[13] Reinhard C. Schumann, "Design of the 21174 Memory Controller for DIGITAL Personal Workstations," *Digital Technical Journal,* Vol. 9, No. 2, 1997.
[14] A. S. Tanenbaum, Computer Networks. Prentice-Hall, 1996.
[15] A. Snavely and D.M. Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreading Processor," in *Proc. of 19th International Conference on Architecture Support for programming Language and Operating Systems(ASPLOS),* 2000, pp. 234–224.