2009 15th International Conference on Parallel and Distributed Systems

Modeling Data Access Contention in Multicore Architectures

Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA
e-mail: sun@iit.edu

Surendra Byna
NEC Laboratories America Inc.
Princeton, NJ, USA
e-mail: sbyna@nec-labs.com

Don Holmgren
Computing Division
Fermi National Accelerator Lab
Batavia, IL, USA
e-mail: djholm@fnal.gov

Abstract— **Multicore processors are now part of mainstream computing. However, data access contention among multiple cores is a significant performance bottleneck in utilizing these processors. Typically, memory hierarchies in multicore architectures use shared last level cache or shared memory. As multiple cores concurrently send requests to access data from these shared memory hierarchy levels, their capacity to serve all the requests is overwhelming and causes performance bottlenecks. In this paper, we introduce simple analytical models for predicting the occurrence of data access contention and provide a guideline for choosing optimal number of cores in running an application without causing data access contention. We verify our models by comparing the predicted optimal number of cores without causing data contention with the measured value in running MIMD Lattice Computation (MILC) application. The proposed analytical models are accurate and promising in guiding data access optimizations to improve multicore utilization.**

Keywords-multicore processors; data access contention; performance prediction; modeling

I. INTRODUCTION

The historical trend in processor technology had been increasing processor speeds as the number of transistors increased. Processor speeds were increasing rapidly as feature sizes shrunk and the number of transistors increased in accordance with Moore's law. However, higher power consumption and heat generation by faster processors have driven chipmakers instead towards packaging multiple simple processor cores into a single chip and reducing speeds. The trend of scaling the number of cores per chip is expected to continue from the 2 to 6 core processors currently available on the market to many more in the future.

While multicore processors offer computing power to improve the performance of applications significantly by providing hardware support for running multiple threads concurrently, it is mainly in the hands of developers to utilize them efficiently. The biggest challenge with multicore processors has been utilizing their computing power. Thread or task level parallelism (TLP) and data-level parallelism (DLP) are the two popular models to utilize their computing power. There are many obstacles in programming for multicore processors using these models.

A critical hurdle in achieving full utilization of multicore processors is data access latency. It has been shown by studies that concurrent multicore processing often achieves much lower performance than expected; sometimes it is even worse than single-core processors due to data access delay [13]. Data access latency has even been a problem with single-core processors as processors are much faster than memory. With the emergence of multicore processors, a more severe problem arises with data access due to bandwidth limitation of shared resources in the memory hierarchy. In a multicore system, multiple cores compete for shared resources (e.g. memory bandwidth, cache memories, and TLB's), which leads to performance degradation if the data requirements are more than what the shared resources can provide.

Identifying data access delay bottlenecks is a valuable step in helping developers to understand their application performance on multicore processors. Modeling data access latency on multicore processors is a way to identify these bottlenecks. In this study, we introduce basic performance models for predicting data access latency, the number of cores to use in running an application without contention, and the impact of data access optimizations. Using the number of data access requests that come to a shared memory, and based on its available bandwidth, we develop a simple analytical model to predict the amount of data transferred between a shared level in memory hierarchy and the level above it. We use this model as a foundation, and two additional models that are designed for user and system level performance optimizations. We then model the impact of optimizations on an application using data prefetching optimization. Using these models, we predict and verify the number of cores for running an application without causing data access contention. These models are designed for different data access and computation patterns in multicores. While the models introduced in this study are simple, they can be used effectively to provide developers an idea of scaling behavior of their application. There exist many models to predict the number of data access requests to memory accurately [7, 5, 2, 11, 1]. The number of data access requests by these models can be directly used in our models to identify contention. The objective of our study is to focus the spotlight on contention caused by shared resources and identifying it using analytical models.

1521-9097/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPADS.2009.121

The paper is organized as follows. In section 2, we provide a brief introduction to memory hierarchy in multicore architectures and to data access contention. In section 3, we introduce our analytical models to predict data access contention and its corollaries. Section 4 presents experimental results that verify our model with measured performance results in running MILC application, and section 5 discusses related work in multicore data access delay modeling. We conclude and provide future directions of our work in section 6.

II. Data Access Contention

Memory hierarchy in processors is an important design feature. To bridge the performance gap between processor and memory, multiple levels of cache memories have been introduced. In the ideal scenario, the data being accessed by a processor should be available in processor registers or in a cache memory that is closer to the processor to avoid stall-time. If data is not available near a processor core, the core has to stall waiting for the data, and computation capabilities of the core are wasted. While multicore processors are designed with the goal of amortizing this stall time by switching the core to run another application, this goal cannot often be realized when running scientific applications where processors (compute nodes) are dedicated to run only one application.

From existing multicore architectures, we can observe two types of memory hierarchies. One has a private cache hierarchy, and the other has a shared outer-most cache. In the private cache architecture, each core has its own cache memories (L1 and L2). In the shared outer-most cache architecture, multiple cores share the lowest level of cache. Figure 1 illustrates two types of two level cache hierarchies in processors, with four cores. In processors, such as IBM POWER multicore processors [9], the last level on-chip cache is split into multiple banks and shared by two or more processor cores. In some processors, there are two levels of private cache with a shared off-chip L3 cache, and in others more than one level of cache memory is shared. It can be concluded that in all types of multicore memory hierarchy, a level is shared among multiple cores, either closer to or farther from the cores.

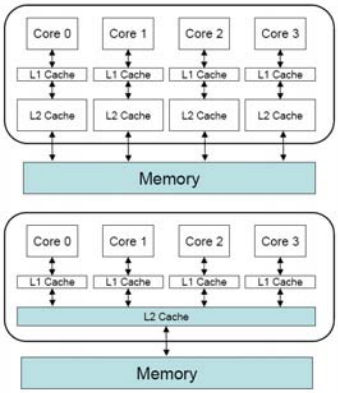

Figure 1. Memory Hierarchy in Multicore.

Sharing a level of memory hierarchy is required as there are benefits of sharing same data by multiple cores. For instance, when the same data is being accessed by multiple cores, having a shared cache requires data transfer only once from DRAM. Multiple private caches require twice the number of data transfers. When multiple cores are processing different sets of data, however, the shared resource becomes a performance bottleneck, if the bandwidth of the shared level is not high enough to support the multiple cores. For instance, if main memory is shared and its bandwidth is not enough to satisfy requests from all the cores of a processor concurrently, then the memory becomes overwhelmed with requests for data and becomes a performance bottleneck. This has been already experienced in some of current processors [13].

Data access contention in multicore processors is application dependent. When an application has its data access workload distributed over time and does not cause bursts of simultaneous accesses, there is no contention. If the number of concurrent data accesses to a shared level of memory hierarchy is higher than the shared level can handle, then that level becomes a source of contention. But, how can one know that there exists data access contention in running an application? To answer that question, in this study we develop analytical models, first to calculate the number of accesses to a level of memory hierarchy, and then to find out whether a shared resource would become a source of contention. Throughout this paper, the term *contention* refers to *data access contention*.

III. Predicting Contention in Multicore Processors

To derive the contention model, we assume the following memory hierarchy in a multicore architecture. We assume two levels of private cache (private L1 and L2), and we assume that the data bus between L2 cache and main memory is shared. The shared memory becomes a bottleneck if there is contention among multiple cores. We use this bottleneck as the target example throughout this paper to illustrate how the contention can be modeled and analyzed. In the presence of a shared L3 cache, as in POWER processors [9], the data bus between the L2 and L3 caches can also be the subject of contention analysis.

We assume that the application under consideration uses one-stage divide-and-conquer problem partitioning, which is common in scientific computing [3, 6]. In this pattern, there are two separate phases in the application: a computing phase and a data access phase. During the computing phase, multiple threads perform computing, and during the data access phase, multiple threads try to access data concurrently. We assume that there is no data dependency among multiple threads. We also assume that each thread is running on its own core (or hardware threads), a typical OS scheduling method on multicore processors.

Assume that n cores are used for computing, and that the achievable speed of each core is $P_{sustained}$ MIPS (million instructions per second). The total number of instructions in each iteration is I, and the ratio between total memory

access instructions and total instructions is R. Then, the computation time of the application running on n cores is:

$$T_{comp} = \frac{I \times (1-R)}{P_{sustained} \times n} \quad (1)$$

Now, let us calculate memory access time in running one iteration with I instructions and the ratio of memory accesses are R. The total number of memory accesses from all cores for one iteration is $I \times R$, despite the number of cores. While the total number of memory accesses is the same for single core or multicore processor per iteration, in a multicore processor, there will be simultaneous accesses from multiple cores, which could result in cause contention. Let us assume that the L1 cache hit ratio of the iteration be H_{L1}, the L2 cache hit ratio be H_{L2}, the L2 cache line be L_{cache}, and assume that each memory instruction accesses one word of L_{word} bytes of data. We define *spatial reuse factor* (F), to represent the spatial locality characteristic of an application. It is well known that according to the principle of locality, instead of fetching just the requested element, a full cache line is fetched into cache memory. If data elements in the fetched cache line are also used by the processor core, then the spatial reuse factor is higher, otherwise, it is lower. The value of F ranges from 0 (in the case where strides between successive accesses is larger than L_{cache}) to 1 (in the case of all contiguous access, where all of the fetched cache line is used). If each instruction requires one word, the actual amount of data accessed for one memory instruction on average then is:

$$F \times L_{word} + (1-F) \times L_{cache} \quad (2)$$

Therefore, the actual data transferred on the bus between L2 cache and the memory is: L =

$$I \times R \times (1-H_{L1}) \times (1-H_{L2}) \times [F \times L_{word} + (1-F) \times L_{cache}] \quad (3)$$

If the sustained bandwidth of the bus between L2 and main memory is $B_{sustained}$, (in bytes/sec) then, the data access latency between L2 and main memory is:

$$T_{comm} = I \times R \times (1-H_{L1}) \times (1-H_{L2}) \times [F \times L_{word} + (1-F) \times L_{cache}] \; / \; B_{sustained} \quad (4)$$

Without cache optimizations, such as data prefetching, loop transformations, etc., the total execution time of one iteration is the summation of the computing time and data access time at each iteration.

$$T = T_{comp} + T_{comm}. \quad (5)$$

A. Finding the Number of Cores without Contention

When an application is run on a multicore processor, it is important to choose an appropriate number of cores without causing contention. Otherwise, many cores will be idle waiting for overwhelmed memory, which becomes a performance bottleneck due to contention. We now show how a user can choose an appropriate number of cores for computing.

The performance of multicore parallel processing can be given in terms of utilization or in terms of execution time. If we assume that the cache hit ratios are unchanged with the number of cores, then the data access time from equation (3) is independent of the number of cores and the computing time decreases with the number of cores. When the number of cores used for computing increases, the data access time quickly becomes the dominant factor of the execution time. If it is required to maintain a certain computing/data access ratio and to maintain the utilization at a certain level, we have to limit the number of cores used for computing. For instance, if we want the computing time to be more than 90% of the execution time, then $T_{comp} > 9T_{comm}$. From equations (1) and (2), we can derive that:

$$n < \frac{I \times (1-R)}{9P_{sustained} \times T_{comm}} \quad (6)$$

This equation gives a guideline to compute the number of cores in a multicore processor to be used for computing in order to keep computing time as a dominant factor.

If the performance requirement is given in terms of execution time, we get different inequalities. For instance, if we require that the execution time of each iteration must be less than C, i.e. T < C, then from equations (1)-(5), we have:

$$T_{comp} + T_{comm} < C \;\Rightarrow\; T_{comp} < C - T_{comm}$$

$$\frac{I \times (1-R)}{P_{sustained} \times n} < C - T_{comm}$$

$$n > \frac{I \times (1-R)}{P_{sustained} \times (c - T_{comm})} \quad (7)$$

This equation is helpful in calculating the appropriate number of cores for achieving the goal of execution time of running an iteration of an application. Note that T_{comp} is a non-negative real number. Hence, an implied assumption of in the inequality equation (7) is that $C - T_{comm.} > 0$.

B. Impact of Data Prefetching Optimization

System level modeling provides performance analysis so that performance optimization can be evaluated. For instance, the impact of optimizations can be identified when optimizations, such as data prefetching and loop transformations, are implemented to mitigate data access delay. Following the same example used above, here we illustrate the model of prefetching. This model is useful for both system level and user level optimizations.

Data prefetching is considered to be effective in improving memory performance. A prefetching engine predicts future memory references, and fetches the required data early enough to avoid cache misses. Prefetching can reduce execution time considerably if it is performed correctly.

Let us first assume that we have no-cost perfect prefetching, i.e. the prefetching mechanism incurs no cost for predicting future references, and the computing cost is overlapped perfectly with data access cost by accurately predicting and prefetching data in time. With this assumption the data access time is negligible and the total execution time is: $T = T_{comp}$, *if* $T_{comp} \geq T_{comm}$. (8)

In other words, when T_{comp} is greater than or equal to T_{comm}, there is no CPU stall. We can find the number of cores to satisfy this $T_{comp} \geq T_{comm}$ condition. From equations (1) and (6), we find that the condition is

$$n \leq \frac{I \times (1-R)}{P_{sustained} \times T_{comm}}. \quad (9)$$

Now, let us relax the perfect prefetching assumption. In practice, no-cost perfect prefetching is impossible to achieve. Prefetching may take some computing power away for data access prediction, thus, in general, we have

$$T'_{comp} = \frac{I \times (1-R)}{P'_{sustained} \times n} \quad (10)$$

where $P'_{sustained} < P_{sustained}$ and $T'_{comp} > T_{comp}$. In equation (8) we assumed that the accuracy is 100%, which is also not practical. When the accuracy of prefetching is considered, it becomes even more complex. There are two types of inaccuracies: *miss fetch* and *fetch miss*. *Miss fetch* fetches some data blocks which are not needed; and *fetch miss* fails to fetch some data which is needed. Miss fetch leads to unnecessary data accesses and increases data transfer (it also may lead to cache pollution; for the sake of simplicity we do not consider the pollution factor here). Fetch miss leads to processor stall for accessing data.

Let the miss fetch ratio be *S,* the ratio between the total data access instructions and the *miss fetch* prefetching instructions. The value of *S* ranges from 0, in the case there is no miss fetch, to 1, in the case of all prefetches are miss fetched. Therefore, the data access time is: $T'_{comm} = I \times R \times [(1-H_{L1}) \times (1-H_{L2}) + S] \times [F \times L_{word} + (1-F) \times L_{cache}] / B_{sustained}$ (11)

T'_{comm} can be overlapped with computing if $T'_{comp} \geq T'_{comm}$. Assume that the fetch miss ratio is α, which is the ratio between the total data access instruction and fetch miss instructions. The execution time with optimization is:

$T' = T'_{comp} + \alpha \ T_{comm}$ (12)

where T_{comm} is given by equation (4).

Now, the number of cores in a multicore processor to be used for computing in order to keep computing time as a dominant factor with prefetching is:

$$n \leq \frac{I \times (1-R)}{9P'_{sustained} \times \alpha T_{comm}} \quad (13)$$

The equation for calculating the appropriate number of cores in achieving the goal of execution time for running an iteration of an application with prefetching becomes:

$$n > \frac{I \times (1-R)}{P'_{sustained} \times (c - \alpha T_{comm})} \quad (14)$$

Comparing the performance between no-cost perfect prefetching and practical prefetching with cost, it can be observed that their performance could be quite different from equations (9) and (13).

IV. Experimental Results

We analyzed the performance of the MIMD Lattice Computation (MILC) application [8] to show selecting optimal number of cores without causing data access contention. MILC is a community code of the lattice quantum chromo-dynamics (LQCD) community and is a grand challenge application identified by US Department of Energy. It represents the QCD applications from high energy physics. Due to its importance to the scientific community, many performance optimizations have been applied to make MILC to run faster, but the success on multicore processors has been limited. LQCD has very poor temporal locality. Its computing/communication ratio is low. The current optimizations of MILC include no write allocation (write directly to lower level cache, which is the L2 cache) and prefetching. The no write allocation approach is helpful in single-core processors, but is not beneficial in a multicore system due to the L2 and L3 data access bottleneck. Prefetching in MILC has been achieved manually through exhaustive testing.

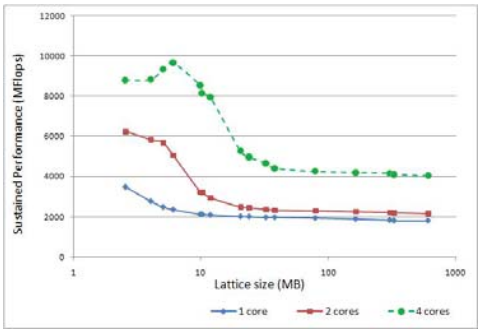

Figure 2. Performance of MILC on Multicore.

Figure 2 shows an example of poor scalability in running the MILC code, on an Intel quad-core processor. As the lattice size increases, sustained performance by using 2-cores or 4-cores scales poorly. It can also be observed that the sustained performance of each case degrades with larger lattice sizes. The main reason for this poor scaling and degrading performance is insufficient memory bandwidth. The severity of this problem is critical as the number of cores in multicore architectures announced is much higher than the two or four in existing processors.

Let us now delve into the details of MILC to understand its data access. The main computation kernel of MILC is a conjugate gradient iteration. The primary data structures of MILC are *su3* vectors, which are 3x1 complex arrays, and *su3* matrices, which are 3x3 complex matrices. MILC uses 4-D lattices. On each lattice site, there are 8 *su3* matrices connecting to the + and - neighbors in x, y, z, and t. Each lattice site also has a *su3* vector. When performing a conjugate gradient solution, all the sites are accessed in each iteration, and each neighboring *su3* vector is multiplied by the corresponding *su3* matrix.

To verify our analytical models in predicting the optimal number of cores on a processor without contention, we retrieved performance results from MILC running on a system with quad-socket quad-core 2.4 GHz Xeon (Harpertown) processors. These Xeon processors support Hyper-threading technology. The CPU has 64 KB L1 data cache, 64 KB L1 instruction cache, and 8 MB L2 cache, with a 1066MHz system bus. The server chipset, Intel 975X ExpressI, was released to support this processor in servers; its dual DDR channels operate to provide up to 10.7GB/s of bandwidth and 8GB memory addressability for faster system response.

On the test system, the sustained performance from the experiments is 130 MFlops/s when accessing

strided/mapped data, where the spatial reuse factor (F) is nearly (but larger than) 0. The sustained performance is 710 MFlops/s when accessing sequential data, where the value of F is 1.

In MILC code, one matrix-vector multiplication operation with single-precision performs 96 bytes of reading, 24 bytes of writing (i.e. total 120 bytes), and 66 floating point operations. Since each memory instruction of floating-point computing accesses 8 bytes of data, the MILC application needs 15 (i.e. 120/8) floating-point memory access instructions for one matrix-vector multiplication. The 66 Flops can be translated into 66 instructions assuming that one instruction is needed to finish one floating-point operation. The ratio between the total number of memory access instructions and the total number of instructions, (i.e. R in our multicore performance models) is: $R = \frac{15}{15+66} \approx 0.19$.

For one stage divide-and-conquer model, computing time decreases with the number of cores while data access time is independent of the number of cores used. It is intuitive to expect that the best utilization of multiple cores should be achieved at the smallest n such that $T'_{comp} \geq T'_{comm}$. From equation (9), $T'_{comp} \geq T'_{comm}$ is true if

$$n \leq \frac{I \times (1-R)}{P'_{sustained} \times T'_{comm}} . \qquad (15)$$

In MILC, prefetching is implemented at the software level manually, which requires no prediction and thus the prediction overhead involved is zero. The loop code, where prefetching is implemented, has high prefetching distance ahead of actual DRAM requests due to cache misses. Hence, the number of requests to DRAM due to cache misses for data that has already been prefetched is negligible, i.e. $P'_{sustained} \approx P_{sustained}$ and $T'_{comm} \approx T_{comm}$.

That is, equation (15) becomes $n_{opt} \leq \frac{I \times (1-R)}{P_{sustained} \times T_{comm}}$, where n_{opt} is the optimal number of cores without causing data access contention.

From equation (11), we can derive T_{comm}.

$$n_{opt} \approx \frac{I \times (1-R) \times B_{sustained}}{P_{sustained} \times L}$$

$$= \frac{I \times (1-R) \times B_{sustained}}{P_{sustained} \times I \times R \times [(1-H_{L1}) \times (1-H_{L2})] \times [F \times L_{word} + (1-F) \times L_{cache}]}$$

The measured cache hit rates of L1 and L2 in running MILC is 90% and 80%, respectively. Using these numbers, the optimum number of cores without causing data contention is:

$$n_{opt} \approx \frac{(1-0.19) \times 604.8}{1799.97 \times 0.19 \times [(1-0.9) \times (1-0.8)] \times [1 \times 8 + (1-1) \times 64]} = 8$$

By using our model, the estimated number of cores without contention (n_{opt}) is 8.

We analyzed the performance results of running MILC on 1, 2, 4, 8, and 16 cores to verify our estimation. Table 1 shows the performance of MILC, where communication is performed via MPI using the internal memory bus. It shows the performance per core and the total performance in MFlops. From the Table, it can be seen that the performance of MILC reaches its peak when the number of cores is 8, which matches the model prediction. This verifies that our prediction model is accurate in predicting the optimal number of cores without causing contention. The optimal number of cores without contention can be used in estimating the impact of data access optimizations on performance improvement and scalability studies. For instance, improving cache performance, i.e. reducing cache miss rate, and increasing memory bandwidth are helpful in increasing n_{opt}.

TABLE I. PERFORMANCE OF MILC (SUSTAINED BANDWIDTH = 604.842MB/S)

Number of cores	1	2	4	8	16
Performance per core (MFlops)	1799.97	1074.52	1010.04	894.29	429.79
Total Performance (MFlops)	1799.97	2149.04	4040.16	**7154.32**	6876.64

A. Usage of the Analytical Model

The models introduced in this study are functional and can be effectively utilized in tuning applications. It has been intentionally kept simple to be effective. The values of the parameters used in predicting the occurrence of contention may be obtained by simple analysis and by the use of profiling applications. The number of memory instructions, total instructions, and cache miss rates can be directly obtained using hardware counters [12]. Sustained memory bandwidth can be measured using the Stream benchmark [4]. The achievable speed of each core is $P_{sustained}$ in MIPS (million instructions per second) can be calculated from the ratio of the number of instructions finished and the execution time of the application or iteration of a loop. These two values are available by simple analysis of hardware counters and profiling. The performance parameters, such hit ratio and spatial reuse factor, are important to the accuracy of the modeling. For most users, who need a guideline to determine the number of cores to use, these parameters can be found for an estimated performance guideline. The model can be extended to obtain accuracy critical results by considering various factors of cache memory, such as cache utilization, cache configuration including associativity, replacement policies, cache conflicts, memory bank parallelism, etc. In case of overlapping computing and data access phases, an overlapping factor can be introduced in equation (5) and in equations derived from it. A model with all these extensions is being considered in our future work.

The analytical model presented in this study is useful in guiding memory hierarchy optimizations and in analyzing the impact of these optimizations. Memory performance optimizations for multicore processors can be broadly

categorized as architectural optimizations, such as improving memory bandwidth, and application-level optimizations, such as reducing cache misses by loop transformations and data prefetching. Our model represents all these optimizations. For instance, improving memory bandwidth, i.e. increasing the value of $B_{sustained,}$ (MB/s) in equation (4), affects the occurrence of data access contention of a memory hierarchy for running an application. Reducing memory access traffic either by improving data access reordering or by other cache optimizations, reduces the number of cache misses and in turn reduces the number of memory access instructions. All of these parameters are included in the model to be able to guide the impact of optimizations on memory performance. The model is comprehensive in covering diverse parameters of memory performance for multicore processors and yet requires less time in calculating the impact of architectural and application level enhancements. This is an attractive feature for utilizing the model in automatic optimization tools that require faster models for tuning applications for certain hardware architectures.

V. Related Work

Performance prediction models have been developed to predict the number of cache misses to help programmers in estimating the cost of memory access. Much of this research effort has been spent in developing accurate cache performance models. Jacob [7] extracts address traces from the code, which requires execution of the program, and consumes a lot of time if an optimization has to be applied. Chatterjee et al. [2] and Ghosh et al. [5] study exact analysis of cache misses, which are very complex. Mao and Saavedra [11] present an analytical model for predicting the effectiveness of data prefetching. This model provides equations for predicting execution time with prefetching. However, it is limited to software-controlled prefetching and needs a major overhaul to model the behavior of new optimizations such as pre-execution based prefetching, accuracy of prediction based prefetching, source and destination of prefetching, and contention of shared resources in multicore processors.

All the above data access latency prediction modes are complex and lack generality. They are bounded to a small set of algorithms. With the goal of simpler and faster models in order to predict performance while searching for various optimizations, we developed an analytical model [1]. However, this model is limited to single core processors. Our model introduced in this study predicts the appropriate number of cores to be used for running an application. It considers the traffic generated by cache misses as its model parameter in identifying the contention. We used a simple method in our study to show that even without accurate information of cache misses the model can serve the purpose of choosing the appropriate number of cores. There are several memory access cost prediction tools available, which can be used for improving the confidence in prediction further.

Suleman et al. [14] introduced a system that measures the amount of utilized bandwidth and schedules threads until the bus bandwidth is saturated. Maury et al. [10] focus on prediction accuracy data access performance by using data collected form 18 counters of 8 events at system level. Both these methods are based on measurement using runtime system support. We use an analytical model to predict what number cores should be used, where cache miss information is used as a model parameter. Also, our study is based on simple analysis and used at the application level, which is helpful for programming as well as scheduling workload on multicore processors.

Shalf et al. [13] have estimated execution time of applications in a dual-core processor by equally splitting memory bandwidth between two cores. In this way, the memory access time for a specific amount of data transfer in an application doubles and that is added to the computation time. This model is straightforward and does not consider any cache parameters or the impact of optimizations. The Roofline model [15] defines arithmetic intensity (AI) of a kernel as the ratio of floating point operations to bytes of DRAM communication and estimates performance by multiplying AI with peak DRAM bandwidth. The Roofline model gives an estimation of achievable performance. The purpose of our model is more specific to identify memory contention and to provide means to predict the optimal number of cores to be used so that memory contention can be avoided. Our analytical models can be helpful in estimating the effect of data access optimizations. Accuracy of our model is verified with experimental results.

VI. Conclusion

Multicore architectures are becoming the norm of high-performance computing by providing many cores and hardware resources for running multiple threads. However, utilization of their computing power has been very low for many scientific applications. Data access contention is a critical hurdle in utilizing multicore processors. Identifying the existence of contention in an application is the primary step in applying optimizations. In this study, we introduced an analytical model to provide guidelines for identifying data access contention, for finding the limit of the number of cores without causing contention, and for showing the effect of prefetching optimization on contention.

Our experimental analysis proves that our models are accurate in predicting data access contention and in finding the number of cores that should be used for achieving performance without contention. These models are helpful for multicore application developers in assessing their applications and for identifying contention on a target system. We intend to extend this study further to predict memory contention at runtime and to re-schedule threads on multicore based compute nodes on-the-fly.

Acknowledgment

This work is supported in part by United States Department of Energy (DoE) SciDAC program under the contract No. DOE DE-FC02-06 ER41442 and by National Science Foundation (NSF) ITR program under NSF-CCF-0621435.

REFERENCES

[1] S. Byna, X.-H. Sun, W. Gropp and R. Thakur, "Predicting the Memory-Access Cost Based on Data Access Patterns", in the proceedings of IEEE International Conference on Cluster Computing, September 2004.

[2] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact Analysis of the Cache Behavior of Nested Loops", Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, Snowbird, UT, June 2001.

[3] D. E. Culler, J. P. Singh, and A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach," Morgan Kaufmann Publishers, 1999.

[4] Dongarra, J., Moore, S., Mucci, P., Seymour, K., You, H. "Accurate Cache and TLB Characterization Using hardware Counters," Proceedings of ICCS 2004 (to appear), Krakow Poland, June 6-9, 2004.

[5] S. Ghosh, M. Martonosi, S. Malik, "Cache miss equations: a compiler framework for analyzing and tuning memory behavior", ACM Transactions on Programming Languages and Systems (TOPLAS), v.21 n.4, pp.703-746, July 1999.

[6] K. Hwang and Z. Xu, "Scalable Parallel Computing: Technology, Architecture, Programming," McGraw-Hill, 1998.

[7] B.L. Jacob, "An analytical model for designing memory hierarchies", IEEE Transaction on Computers, Volume 45, pp. 83-105, 1996.

[8] US Lattice Chromo-dynamics, http://www.usqcd.org/

[9] H.Q. Le, W.J. Starke, J.S. Fields, et al., "IBM POWER6 Microarchitecture", IBM Journal of Research and Development, vol. 51, no. 6, November 2007.

[10] M. Cutis-Maury, F. Blagojevic, C. Antonopoulos, and D. Nikolopoulos, "Prediction Based Power-Performance Adaptation of Multithreaded Scientific Code," IEEE Transactions on Parallel and Distributed Systems, October 2008.

[11] W. Mao and R. H. Saavedra, "The Limits and Effectiveness of Data Prefetching on Scalable Multiprocessors", in Performance Evaluation, vol. 27-28, pp. 209 – 229, Oct 1996.

[12] The STREAM Benchmark: Computer Memory Bandwidth, http://www.streambench.org/

[13] J. Shalf, "Memory Subsystem Performance and QuadCore Predictions", Presentation at NERSC User Group Meeting, September 17, 2007 <http://www.nersc.gov/about/NUG/meeting_info/Sep07/charts/Shalf-NUG2006_QuadCore.pdf>.

[14] A. Suleman, M. K.Qureshi, Y. N. Patt, "Feedback Driven Threading: Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPS," ASPLOS'08, Seattle, Washington, USA.

[15] Williams S, Patterson D, Oliker L, Shalf J, and Yelick K 2008 The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures Hot Chips 20: Stanford University, Stanford, California, August 24–26, 2008.