Supporting Quality of Service in High-Performance Servers *

Yan Solihin, Fei Guo, Seongbeom Kim, Fang Liu
Dept. of Electrical and Computer Engineering
North Carolina State University
{solihin,fguo,skim16,fliu3 @ece.ncsu.edu}

Abstract

This paper describes issues that we have analyzed and technology that we have developed in supporting Quality of Service in high-performance servers. More specifically, we target on-chip cache resource allocation and efficiency needed for guaranteeing certain performance levels on Chip Multi-Processor (CMP) architectures. Both prior and ongoing work are summarized in this paper.

1 Introduction

Chip Multi-Processor (CMP) architecture has become an architecture of choice for major micro-processor makers [19, 11, 16, 13]. With an increasing number of cores per chip in a CMP system, throughput can be improved for multi-threaded applications as well as for multiprogrammed workload. However, since many on-chip resources such as the lowest level on-chip cache and bandwidth to off-chip interconnect are shared by all the processor cores, the performance and throughput of programs running on a CMP depend heavily on how these resources are allocated to them. The absence of resource allocation policies, such as the ability to partition caches and off-chip bandwidth, can lead to unpredictable contention for these resources and result in a large performance variation for many applications [4, 25, 12, 10]. It has been pointed out that the large performance variation may even produce effects of cache thrashing, starvation, and priority inversion, which are problematic to the Operating System design [4, 25, 7]. Without any policies, these problems will become worse in future CMP architectures where the number of cores as well as the number simultaneously running applications are expected to increase.

As a result of this observation, researchers have proposed various policies for managing these contended on-chip resources. For example, Suh et al. proposed cache partitioning policy that minimizes the total number of cache misses [29, 30], while Kim et al. has proposed cache partitioning policy that optimizes for fairness (uniform slowdown) for applications that share the cache [25]. These policies have been shown to improve throughput and/or fairness. However, they treat all applications in the same way without considering how important one application is with respect to another. In many cases, it is useful to convey applications' performance requirement to the hardware as an input that guides the on-chip resource allocation policy. This is especially relevant in the context of *virtual computing*, *utility computing*, and *real-time computing*. In virtual computing, a virtual machine manager or VMM hosts multiple guest Operating Systems, each of which is run for specific purposes ranging from casual to critical computation, therefore it is important for the system to enable the VMM to allocate more resources to guest OSes with critical computation but fewer resources to guest OSes used for casual computation. In a virtual computing environment, programs from different consumers share the same servers provided by a utility computing provider. It would be helpful if the provider is able to distinguish service levels for different customers, and these service levels reflected proportionally in the fraction of allocated resources. These service levels are often referred to as *Quality of Service* (QoS) models. Finally, real-time transactions in a service oriented computing domain require a certain minimum service level to be provided to each task, requiring performance differentiation and performance guarantee [20, 22].

Once QoS support, such as on-chip resource partitioning is there, there are still challenges of scalability of CMP since each processor will get only a fraction of such resources. It has even been pointed out that many-core chips containing tens to hundreds of processor cores can be built. The problem is whether the current memory resources (caches and off-chip bandwidth) can support them since declining share of resources per core would make each core slow and further worsens the contention at lower level memory hierarchy such as the off-chip bandwidth and main memory. Therefore, it is important that each processor core can improve its *resource usage efficiency* by adopting cache policies that are *bandwidth conserving*.

Contributions. The contributions of the project include:

1. The identification, measurement, and modeling of contention for onchip resources, primarily the caches [4, 25, 5, 26].

2. Identification of cache replacement policy as a significant source of cache inefficiency which leads to in-

*This project is supported in part by NSF Grant CNS-0406306

1-4244-0910-1/07/$20.00 ©2007 IEEE

efficient use of off-chip bandwidth, modeling of such inefficiency [8, 9], and new replacement policy and bypassing techniques that reduce such inefficiency [15].

3. Quality of Service models supported by the architecture and the Operating System [14].

4. Tutorial and public release of cache performance modeling toolset [27].

2 Resource Contention Identification, Measurement, and Modeling

Cache Contention Measurement. A certain class of applications, such as *mcf* and *gzip*, are very vulnerable to cache sharing. To illustrate the performance problem due to cache sharing, Figure 1 shows the number of L2 cache misses (1a) and throughput (measured as the number of instructions commmitted per cycle or IPC) (1b) for *mcf* when it runs alone compared to when it is co-scheduled with another thread which runs on a different processor core but sharing the L2 cache. The bars are normalized to the case where *mcf* runs alone. The figure shows that when *mcf* runs together with *mst* or *gzip*, *mcf* does not suffer from many additional misses compared to when it runs alone. However, when it runs together with *art* or *swim*, its number of misses increases to roughly 390% and 160%, respectively, resulting in IPC reduction of 65% and 25%, respectively.

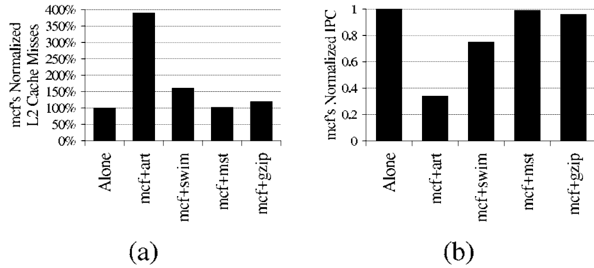

Figure 1. The number of L2 cache misses (a), and IPC (b), for *mcf* when it runs alone compared to when it is co-scheduled with another thread. The L2 cache is 512KB, 8-way associative, and has a 64-byte line size.

Fairness is a critical aspect to optimize because the Operating System (OS) thread scheduler's effectiveness depends on the hardware to provide fairness to all co-scheduled threads. An OS enforces thread priorities by assigning timeslices, i.e., more timeslices to higher priority threads. However, it assumes that in a given timeslice, the resource sharing uniformly impacts the rates of progress of all the co-scheduled threads. Unfortunately, we found that the assumption is often unmet because a thread's ability to compete for cache space is determined by its temporal reuse behavior, which is often very different compared to that of other threads which are co-scheduled with it.

When the OS' assumption of fair hardware is not met, there are at least three problems that can render the OS scheduler ineffective. The first problem is *thread starvation*, which happens when one thread fails in competing for sufficient cache space necessary to make satisfactory forward progress. The second problem is *priority inversion*, where a higher priority thread achieves a slower forward progress than a lower priority thread, despite the attempt by the OS to provide more timeslices to the higher priority thread. This happens when the higher priority thread loses to the lower priority thread (or other threads) in competing for cache space. To make things worse, the operating system is not aware of this problem, and hence cannot correct this situation (by assigning more timeslices to the higher priority thread). The third problem is that the forward progress rate of a thread is highly dependent on the thread mix in a co-schedule. This makes the forward progress rate difficult to characterize or predict, making the system behavior unpredictable. Unfortunately, despite these problems, cache implementations today are thread-blind, producing unfair cache sharing in many cases.

Fair Caching. We define fairness as equal slowdown when the execution time of an application is compared to the case where it runs alone in the system. Let $Tded_i$ denote the execution time of thread i when it runs alone with a dedicated cache, and $Tshr_i$ denote its execution time when it shares the cache with other threads. When there are n threads sharing the cache and assuming that the threads are always co-scheduled for their lifetime, an ideal fairness is achieved when:

$$\frac{Tshr_1}{Tded_1} = \frac{Tshr_2}{Tded_2} = \ldots = \frac{Tshr_n}{Tded_n} \tag{1}$$

which we refer to as the *execution time fairness* criteria. To enforce fairness, we approximate it with metrics that are directly measurable at the shared L2 cache. We evaluate five L2 cache fairness metrics that are directly related to the L2 cache performance and are insensitive to external factors, while at the same time easy to measure. Let $Miss$ and $Missr$ denote the number of misses and miss rates, respectively. For any pair of co-scheduled threads i and j, the following metrics measure the degree of fairness between a thread pair:

$$M_1^{ij} = |X_i - X_j|, \text{ where } X_i = \frac{Miss_shr_i}{Miss_ded_i} \tag{2}$$

$$M_2^{ij} = |X_i - X_j|, \text{ where } X_i = Miss_shr_i \tag{3}$$

$$M_3^{ij} = |X_i - X_j|, \text{ where } X_i = \frac{Missr_shr_i}{Missr_ded_i} \tag{4}$$

$$M_4^{ij} = |X_i - X_j|, \text{ where } X_i = Missr_shr_i \tag{5}$$

$$M_5^{ij} = |X_i - X_j|, \text{ where } X_i = Missr_shr_i - Missr_ded_i \tag{6}$$

We found that M_1 has the highest correlatation with the execution time fairness criteria (94%). By periodically partitioning the L2 cache to optimize for the fairness metric, fairness (measured as the difference in the fairness metrics between two threads that share the L2 cache) improves by 4-5X, while throughput improves by 15%, as shown in Figure 2.

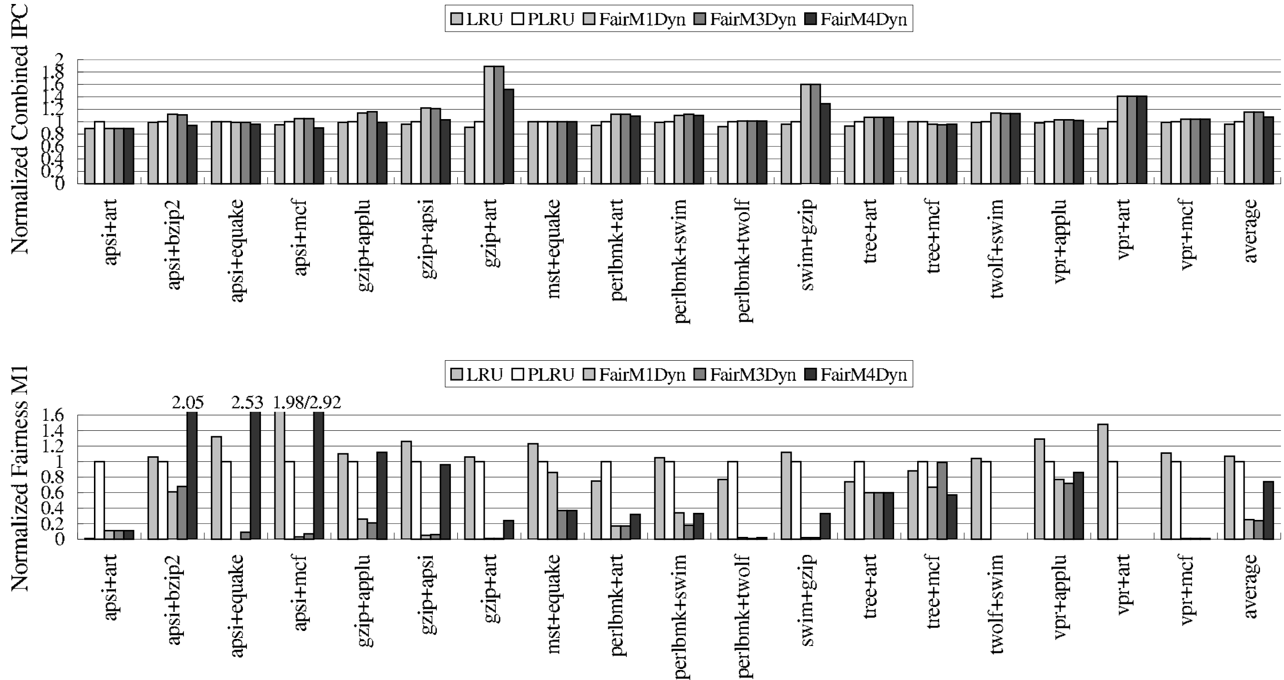

Figure 2. Throughput (top chart) and fairness metric M_1 (bottom chart) of dynamic partitioning algorithms. Lower M_1 values indicate better fairness.

Figure 2 shows the throughput (top chart) and fairness results (bottom chart) of our dynamic partitioning algorithm for the benchmark pairs and their average. Throughput and fairness are represented as the combined Instructions Per Cycle (IPC) and metric M_1, respectively. Each benchmark pair shows the result for four schemes. The first bar (*LRU*) is a non-partitioned shared L2 cache with LRU replacement policy. The second bar (*PLRU*) is a non-partitioned shared L2 cache with pseudo-LRU replacement policy that is more likely to be implemented in highly associative L2 cache in real systems, compared to LRU. The last three bars (*FairM1Dyn*, *FairM3Dyn*, and *FairM4Dyn*) are our L2 dynamic partitioning algorithms that minimize M_1, M_3, and M_4 metrics, respectively. All the bars are normalized to *PLRU*, which is selected as the base case because it is a more realistic L2 cache implementation.

The figure shows that *PLRU* and *LRU* achieve roughly comparable throughput and fairness. *FairM1Dyn* and *FairM3Dyn* improve fairness over *PLRU* significantly, reducing the M_1 metric by a factor of 4 on average (or 75% and 76%, respectively) compared to *PLRU*. This improvement is consistent over all benchmark pairs, except for *FairM3Dyn* on *tree+mcf*. In *tree+mcf*, *PLRU* already achieves almost ideal fairness, and therefore it is difficult to improve much over this. The figure also shows that fairness strongly impacts throughput. By achieving better fairness, both algorithms achieve a significant increase in throughput (15%). The throughput improvement is consistent for almost all benchmark pairs. Nine out of eighteen cases show throughput improvement of more than 10%. In *gzip+art*, the throughput increases by almost two times (87%). The reason for the improved throughput is that by achieving better fairness, we eliminate situations where one of the processor in a 2-core chip is under-utilized due to unfair cache usage, and hence the overal throughput in general improves. The only noticeable throughput decrease is in *apsi+art* and *tree+mcf*, where *FairM1Dyn* reduces the throughput by 11% and 4%, respectively. In those cases, the fairness is improved (M_1 is reduced by 89% and 33%, respectively). This implies that, in some occasions, optimizing fairness may reduce throughput.

3 Modeling and Improving Cache Replacement Policies as Means to Conserve Bandwidth

Modeling Cache Replacement Policy Performance [9, 8]. Due to the increasing gap between CPU and memory speed, cache performance plays an increasingly critical role in determining the overall performance of microprocessor systems. Utility computing servers increasingly use multi-core chips as their building blocks. In these chips, because multiple processors share the L2 and L3 caches, the caches suffer from increased capacity pressure. Cache replacement policy, in addition to cache associativity and block size, is an important factor that affects cache performance for a fixed cache size. The performance variation between different cache replacement policies can be quite significant in many cases.

To illustrate the extent of the performance variation, Figure 3 compares the L2 cache miss rates and normalized execution time of three memory-intensive Spec2000/NAS applications under two replacement policies: Least Recently

Used (LRU) replacement which is the most popular implementation in caches, and random replacement policy which replaces more recently used lines with a higher probability than less recently used lines (Rand-MRUskw). The figure shows that the applications achieve much lower L2 cache miss rates (by up to 67%) and execution time (by up to 67%) with Rand-MRUskw compared to LRU. Although the very high performance variation shown in the figure is not typical across all applications, it clearly shows that replacement policy is a factor that should be taken into account in designing a cache and modeling cache performance. Furthermore, we also found that a majority (10 out of 17) of the Spec2000/NAS applications tested achieve more than 5% lower miss rates under Rand-MRUskw versus under LRU on at least one cache size configuration.

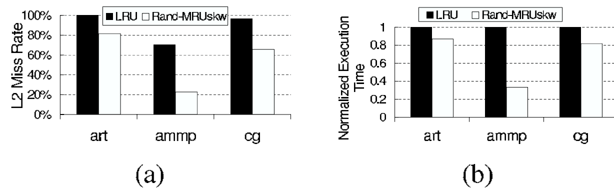

Figure 3. The L2 cache miss rate (a), and execution time (b), for *art, ammp* and *cg* under LRU and Rand-MRUskw replacement policies. The L2 cache is 8-way, 512-KB, and has 64-B block size.

Despite the extent of the performance variation, current analytical cache performance models ignore the impact of cache replacement policies on performance. They assume a certain replacement policy in their theoretical derivation, such as LRU [3, 4, 6, 21, 23, 24, 28], or fully random [1, 2, 17]. Assuming a LRU or random replacement policies greatly simplifies cache performance modeling, and the assumption was relatively valid for small caches which tend to have a low associativity or are direct-mapped. However, modern lower level caches (L2 and L3 caches) are often highly associative and large, increasing the role of replacement policy in determining cache performance.

The contribution of this paper is an analytical cache model that predicts the performance of cache replacement policies. To the best of our knowledge, this is the first attempt to arrive at such a model. The input of our model is the *circular sequence profiling* [4] of each application. The profiling can be easily collected, and requires very little storage. Only one profiling run is required for each application to generate the predicted performance across different replacement policies. The output of the model is the predicted miss rates for a given application. The model is based on probability theory and utilizes Markov processes to compute each cache access' miss probability. The model uses realistic assumptions and relies solely on the statistical properties of each application's access pattern (i.e. it does not employ any heuristics or rules of thumbs). Replacement policies are represented by a *replacement probability function* (RPF) which specifies the probability of a line in different LRU stack position to be replaced on a cache miss. Many replacement policies, such as LRU, Random, Not Most Recently Used (NMRUx), and Skewed Random, can be specified or approximated by an RPF.

Our model allows an in-depth analysis of how an application's behavior impacts its performance under different cache replacement policies. This analysis is difficult to achieve with simulation models, because they are limited by the discrete and possibly narrow behavior patterns available in current benchmark suites. Behaviors that are not represented by current benchmark suites cannot be analyzed. In addition, any two applications usually differ in more than one factor, making it difficult to isolate any particular factor's contribution to performance. The model achieves significant advantages compared with trace simulations in term of running time and storage overhead for studying cache replacement policy performance. The model takes a constant time of less than 0.1 seconds to generate a miss rate estimate on an Intel Xeon 2.0-GHz processor platform, compared to $\Theta(number\ of\ events)$ run time needed by a trace simulation, which is in the order of hours–days for realistic workloads. The model also requires a small and constant storage overhead for the profiling information, compared to $\Theta(number\ of\ events)$ in a trace simulation.

We validate the model by comparing the predicted miss rates of seventeen Spec2000 and NAS benchmark applications against cycle-accurate execution-driven simulations. The model is very accurate, achieving a prediction error (the absolute difference between the predicted and simulated miss rates) of 1.41% on average, and 20% in the worst case. There are only 14 out of 952 validation points in which the prediction errors are larger than 10% across all experiments with different cache sizes, associativities, and four replacement policies. Finally, to illustrate one possible practical use of the model, we present a case study that analyzes the relationship between cache access patterns of an application with its performance under different replacement policies. The case study reveals that temporal reuse patterns of applications play a major part in deciding how they perform under different replacement policies.

Counter-Based Cache Replacement and Bypassing [15]. Recent studies have shown that in highly associative caches such as the L2 cache, the performance gap between the Least Recently Used (LRU) and Belady's theoretical optimal replacement algorithms is large. For example, the number of cache misses using LRU can be up to 197% higher than using the optimal replacement [31, 18]. This suggests that alternative replacement algorithms may be able to improve cache performance significantly over LRU. The need for more efficient cache design is even more important in multi-core chips, which are the building blocks of utility computing servers, in which multiple processors share the caches and increase capacity pressure on the caches.

LRU replacement algorithm tries to accomodate temporal locality by keeping recently used lines away from replacement, in hope that when they are reused, they will still be in the cache. Unfortunately, two things work against LRU replacement. For one thing, each cache line will be eventually replaced after its last use. However, even after

its last use, a line is not immediately replaced because it remains in the cache until it becomes the LRU line. Such "dead" lines unnecessarily reduce the cache capacity available for other lines. The *dead time*, i.e. time between when a line becomes dead and when it is eventually replaced, becomes worse with larger cache associativities since it takes longer for a line that is recently last-used to travel down the LRU stack to become the LRU line. Hence, although a larger cache associativity improves cache performance in general, the performance gap between LRU and the optimal replacement algorithm also increases. Clearly, replacing dead lines promptly after their last use would improve cache performance by making the wasted capacity available for other cache lines that are not dead yet. In order to achieve that, a *dead line prediction* technique is needed to identify and replace dead lines early.

Another factor that works against LRU replacement is that temporal locality may *invert* when there are multiple levels of caches. There are many cache lines that exhibit bursty temporal reuses. This is often due to spatial reuses of different bytes of the same cache line, which tend to occur in burst. Current caches typically have large cache lines (64 or 128 bytes), amplifying this bursty reuse pattern. With a single-level cache, this bursty temporal reuses would be well accomodated by the LRU replacement. However, in multi-level caches, this bursty pattern often manifests at the L1 cache only and is filtered by the L1 cache. To the L2 cache, the line does not appear to have temporal reuse since it is brought into the cache but is not used until it is replaced. Hence, the lines are immediately dead after they are brought into the L2 cache. Such *never-used* lines unnecessarily waste the L2 cache capacity. Ironically, lines with less frequent temporal reuses cannot be filtered by a small L1 cache and such reuses will show up at the L2 cache. Note that for never-used lines, immediately replacing them after they are brought into the L2 cache would only be partially beneficial. A better approach is to identify them and avoid placing them in the L2 cache in the first place, using a technique often referred to as *cache bypassing*.

Our main finding is that a single mechanism can simultaneously achieve both dead line prediction and cache bypassing. Our mechanism relies on counters that keep track of the number of relevant cache events in a cache line's history, and use that to predict the cache line's future behavior. We call our approach *counter-based cache replacement* and *counter-based cache bypassing*. In our approach, each L2 cache line is augmented with an *event counter* that is incremented when an event of interest (such as certain cache accesses) occurs. For replacement decisions, when the counter reaches a *threshold*, the line *expires*, and immediately becomes replaceable. We design and evaluate two alternative algorithms, which differ by the type of events counted and the intervals in which they are counted: *Access Interval Predictor* (AIP) counts the number of accesses to a set in an interval between two consecutive accesses to a particular cache line, while *Live-time Predictor* (LvP) records the number of accesses to a cache line in an interval in which the line resides continuously in the cache. For bypassing decisions, the same event counters can identify never-used lines, and in the future they can be directly placed in the L1 cache without polluting the L2 cache.

Through a detailed simulation evaluation, AIP and LvP speed up 10 out of 21 Spec2000 applications that we tested by up to 41%, or 11% on average without slowing down the remaining eleven applications by more than 1%. Both AIP and LvP outperform other dead line predictors in terms of *coverage* (fraction of replacements initiated by the predictors), *accuracy* (fraction of replacements that agree with the theoretical optimal replacement), and *IPC improvement* using comparable hardware cost compared to other dead line predictors. Furthermore, bypassing can be added to AIP and LvP without additional hardware, which further improves the average speedup to 13%. Both AIP and LvP only incur small overheads: each cache line is augmented with 21 bits to store prediction information, equivalent to 4.1% storage overhead for a 64B line. In addition, a simple 40-KB prediction table is added between the L2 cache and its lower level memory components. The prediction table is only accessed on an L2 cache miss and thus its access is overlapped with the L2 cache miss latency.

4 Hardware and OS Support for Quality of Service in Servers

(This is an ongoing work. Preliminary results are published in the Proceedings of the *Workshop on Design, Architecture and Simulation of Chip Multi-Processors* (**dasCMP**) in December 2006 [14].)

We will look at the implications of various QoS models on the performance of the system and the ability of the system to provide performance guarantee. In particular, the ability of the system to provide performance guarantee depends on whether the system has an ability to compare its spare computation capacity with the the capacity requested by a incoming job. Without this ability, QoS cannot be ensured. Furthermore, the ability of the system to guarantee QoS also has a direct impact on its throuhgput. For example, the system may be very conservative in allocating more resources than needed for a job to satisfy its QoS requests, however doing this would limit the number of jobs that run on the system and in turn reduce overall throughput. Therefore, we plan to study QoS models, admission control of jobs, and how to maximize throughput while satisfying the QoS requests of each job.

We have revised Linux OS to interface with QoS-capable hardware and have constructed a full system processor simulation based on Simics. We are implementing major functionalities of the system and are starting to analyze the first set of results.

5 Cache Modeling Tool

Together with IBM researchers, we are holding a tutorial "Practical Cache Performance Modeling for Computer Architects" at the *13th International Symposium on High-Performance Computer Architecture*, Phoenix, Arizona, Feb 11, 2007. The tutorial presenters would be Yan

Solihin, Thomas Puzak, and Phil Emma. We are releasing the cache performance modeling tool that we have developed to the public, and in the tutorial we plan to demonstrate how to use the tool.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.

[2] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software*, pages 20–27, 2004.

[3] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proc.of the 17th Intl. Conf. on Supercomputing*, pages 150–159, 2003.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multiprocessor Architecture. In *Proc. of the 11th Intl. Symp. on High Performance Computer Architecture*, pages 340–351, 2005.

[5] D. Chandra, S. Kim, and Y. Solihin. Predicting the Impact of Cache Contention on a Chip Multiprocessor Architecture. In *IBM T.J. Watson Conf. on Interaction between Architecture, Circuits, and Compilers*, 2004.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *Proc.of the 11th international conference on Supercomputing*, pages 317–324, 1997.

[7] R. Goodwins. Does hyperthreading hurt server performance? *http://news.com.com/Does+hyperthreading+hurt +server+performance/2100-1006-3-5965435.html*, 2005.

[8] F. Guo and Y. Solihin. A prediction model for alternative cache replacement. In *Proc. of the IBM Watson Conf. on Interaction between Architecture, Circuits, and Compilers*, 2005.

[9] F. Guo and Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. In *Proc. of ACM SIGMETRICS/Performance 2006 Joint Intl. Conf. on Measurement and Modeling of Computer System*, 2006.

[10] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *Proc. of the 15th international conference on Parallel architectures and compilation techniques*, pages 13–22, 2006.

[11] Intel Corporation. Intel dual-core processors:the first step in the multi-core revolution. *http://www.intel.com/technology/computing/dual-core/*, 2006.

[12] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proc. of the 18th annual international conference on Supercomputing*, pages 257–266, 2004.

[13] R. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[14] H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From Chaos to QoS: Case Studies in CMP Resource Management. In *Proc. of the Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2006.

[15] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *Proc. of the Intl. Conf. on Computer Design*, 2005.

[16] K. Krewell. Best servers of 2004. *http://www.mdronline.com/*, 2005.

[17] R. E. Ladner, J. D. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Proc. of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 613–622, 1999.

[18] W.-F. Lin and S. Reinhardt. Predicting Last-Touch References under Optimal Replacement. *University of Michigan Tech. Rep. CSE-TR-447-02*, 2002.

[19] K. Olukotun and L. Hammond. The future of microprocessor. *ACM Queue*, 3(7), 2005.

[20] M. P. Papazoglou and D. Georgakopoulos. Seviced-oriented computing:introduction. *Communications of the ACM*, 46(10):24–28, 2003.

[21] R. W. Quong. Expected i-cache miss rates via the gap model. In *Proc. of Intl. Symp. on Computer Architecture*, pages 372–383, 1994.

[22] P. Ranganathan and N. Jouppi. Enterprise it trends and implications for architecture research. In *Proc. of the 11th international Symp. on High Performance Computer Architecture*, pages 253–256, 2005.

[23] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, 2002.

[24] J. P. Singh, H. S. Stone, and D. F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Trans. on Computers*, 41(7):811–825, 1992.

[25] S.Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning on a Chip Multiprocessor Architecture. In *Proc.of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2004.

[26] S.Kim, D. Chandra, and Y. Solihin. Fair Caching on a Chip Multiprocessor Architecture. In *IBM T.J. Watson Conf. on Interaction between Architecture, Circuits, and Compilers*, 2004.

[27] Y. Solihin, T. Puzak, and P. Emma. Practical Cache Performance Modeling for Computer Architects. *Tutorial at the 13th Intl. Symp. on High-Performance Computer Architecture*, 2007.

[28] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proc. of the 15th international conference on Supercomputing*, pages 1–12, 2001.

[29] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proc. of Intl. Symp. on High Performance Computer Architecture*, 2002.

[30] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 2002.

[31] W. Wong and J.-L. Baer. Modified LRU Policies for Improving Second-Level Cache Behavior. In *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2000.