Bank-aware Dynamic Cache Partitioning for Multicore Architectures

Dimitris Kaseridis*, Jeffrey Stuecheli§* and Lizy K. John*
*Department of Electrical and Computer Engineering, The University of Texas at Austin, TX, USA
§IBM Corp., Austin, TX, USA
Emails: kaseridi@ece.utexas.edu, jeffas@us.ibm.com and ljohn@ece.utexas.edu

***Abstract*—As Chip-Multiprocessor systems (CMP) have become the predominant topology for leading microprocessors, critical components of the system are now integrated on a single chip. This enables sharing of computation resources that was not previously possible. In addition, the virtualization of these computational resources exposes the system to a mix of diverse and competing workloads. Cache is a resource of primary concern as it can be dominant in controlling overall throughput. In order to prevent destructive interference between divergent workloads, the last level of cache must be partitioned. In the past, many solutions have been proposed but most of them are assuming either simplified cache hierarchies with no realistic restrictions or complex cache schemes that are difficult to integrate in a real design. To address this problem, we propose a dynamic partitioning strategy based on realistic last level cache designs of CMP processors. We used a cycle accurate, full system simulator based on Simics and Gems to evaluate our partitioning scheme on an 8-core DNUCA CMP system. Results for an 8-core system show that our proposed scheme provides on average a 70% reduction in misses compared to non-partitioned shared caches, and a 25% misses reduction compared to static equally partitioned (private) caches.**

I. Introduction

Chip Multiprocessors (CMP) have gradually become an attractive architecture for leveraging system integration by providing capabilities on a single die that would have previously occupied many chips across multiple small systems [1][2]. This integration has brought abundant on-chip resources that can now be shared in finer granularity among the multiple cores. Such sharing though has introduced chip-level contention and the need of effective resource management policies is more important that ever.

To efficiently exploit these resources, systems require multiple program contexts and virtualization has become a key player in this arena. Many small and/or low utilization servers can now be easily consolidated on a single physical machine [3][4][5], allowing higher utilization of the available resources with significant energy reductions. Such consolidation presents both opportunities and pitfalls to computer architects to best manage these once isolated resources on large CMP designs.

In such virtualization environments, workloads tend to place dissimilar demands on shared resources and therefore, due to resource contention, are much more likely to destructively interfere in an unfair way. Consequently, shared resources' contention become the key performance bottleneck in CMPs [6][7][8][9]. Shared resources include, but are not limited to: main memory bandwidth, main memory capacity, cache capacity, cache bandwidth, memory subsystem interconnection bandwidth and system power.

Among these resources, several studies have identified the shared last-level cache (L2 in our study) of CMPs as a major source of performance loss and execution inconsistency [7][10][11][12][13][14][15]. As a solution, most of the proposed techniques control this contention by partitioning the L2 cache capacity and allocating specific portions of it to each core or execution thread. There are both static [7][13] and dynamic partitioning [10][15][16] schemes available that use workload profiling information to make a decision on cache capacity assignment for each core/thread. All of the above techniques are usually based on high-level system characteristic monitoring since low-level activity based algorithms such as LRU replacement fail to provide a strong barrier among workloads competing for shared resources.

In addition to the cache partitioning need, as wire delays are gradually becoming the most important design factor in cache architectures, designers have successfully used banking techniques [17][18] to mitigate the effects of increasing wire delays for short distances. Banked architectures are now the typical design direction for caches in both industry and academia. Such solutions though are still not efficient enough since wire delays between banks themselves are still an important performance bottleneck. An alternative solution to the wire delay problem, mainly used in academia is the Non-Uniform Cache Architecture (NUCA) designs [9]. NUCA is based on assuming non-uniform access latencies to all cache banks of a large L2 cache. The NUCA model, which was originally proposed for a single core, was later extended to a multicore CMP version named CMP-NUCA by Beckmann et al. [19]. In parallel, industry has also responded to wire delay dominance of on-chip caches with non-uniform structures. These structures have been implemented with a small number of cache levels, rather than large arrays of homogeneous networks of cache blocks as are assumed in academia. Both approaches are logically similar and the differences are more tied to the physical implementation constraints of cache banks and data networks rather than higher-level policy options. As new CMP designs include more cores and cache capacity, a banked L2 cache design is a promising solution that can scale with the number of cores and is able to alleviate wire delay problems.

This work highlights the problem of sharing the last level of cache in CMP systems and motivates the need for low overhead, workload feedback-based hardware/software mechanisms that can scale with the number of cores, for monitoring and controlling the L2 cache capacity partitioning. The need to address the dominating effect of wire delays by taking

into consideration the realistic constraints imposed by banking architectures drove our baseline system structure. Specifically, we propose a *Bank-aware* partitioning strategy for the CMP-DNUCA architecture, consistent with the current industry trends, that is aware of the banking structure of the L2 cache. To evaluate our partitioning scheme, we integrated an 8-core CMP system with a 16-way banked DNUCA L2 cache design, using Simics [20] combined with Gems [21] full system, cycle accurate simulation toolset. In summary, the paper's contributions are the following:

1) We propose a cache partitioning scheme, named *Bank-aware*, for CMP-DNUCA that is aware of the banking structure of the L2 cache. Our simulations showed a 70% reduction in misses compared to non-partitioned shared caches, and a 25% misses reduction compared to static even partitioned (private) caches. Such miss rate reductions result in 43% and 11% reductions in CPI over the non-portioned and static even partitioned schemes, respectively.
2) We demonstrate a detailed implementation of a dynamic cache partitioning algorithm using a non-invasive, low-overhead monitoring scheme based on Mattson's stack distance algorithm [22]. The overall hardware overhead for the proposed cache profiling scheme is equal to 0.4% of our baseline L2 cache design.

The paper is organized as follows. Section 2 summarizes our CMP-baseline design. In Section 3 we elaborate on our proposed *Bank-aware* cache partitioning scheme. Section 4 describes our evaluation methodology and reports our experimental results. Finally, section 5 contains our conclusion remarks.

II. CMP-Baseline

Prior works in industry and academia have proposed quite varied allocation and migration schemes for the memory cache hierarchy. A large amount of work in academia has focused on free form, highly banked, and non-uniform cache structures. This was in response to the expected wire dominant nature of future technologies, where the latency of large monolithic caches would become detrimental to system performance. These proposed free form caches enable great freedom in allocation and migration policies. As an example Huh et al. in [16] proposed a 256 x 64K bank cache. In contrast, industry has thus far typically implemented more traditional structures with fewer than eight cache banks that form specific multi-level caches (compared to more free form NUCA like levels). For example, the recently announced 45nm Intel Nehalem processors has three levels on chip cache (32KB, 256KB, 4-8MB) [23], compared to two levels in the previous design.SUch additional cache levels approach a more NUCA-like cache, formed of homogeneous cache banks.

The industry direction of avoiding highly banked structures can be also explained by recent upgrades to the CACTI 6.0 tool [24]. In this work they demonstrated using detailed modeling of the cache and interconnection subsystem results in generated remarkably different results. As a case study, they evaluated a 32 MB L2 cache [24]. This gave a mix of ideal cache block sizes of 4 MB and 8MB. This landscape drove our baseline system structure. Specifically, we limited the total bank structures on the chip to 1MB cache banks. This was chosen as the smallest reasonable bank size.

Fig. 1 shows our 8-core CMP-NUCA baseline system. Our design uses as the last-level of cache a DNUCA L2 cache with 16 physical banks that provide a total of 16MB of cache capacity. Each cache bank is configured as an 8-way set associative cache. Another way to see the cache is as a 128-way equivalent cache that is separated in 16 cache banks of 8 ways each. The eight cache banks that are physically located next to a core are called *Local banks* and the rest are characterized as *Center banks*. Cores located next to *Local banks* have the minimum access latency but that delay can significantly increase when a core needs to access a *Local bank* physically located next to another core. *Center banks* have, on average, higher access latency than *Local banks* but their distance for each core has smaller variation than *Local banks* and so does the access latency. The access latency to a L2 cache bank varies from 10 up to 70 cycles depending on the physical location of both the core requesting the access and the L2 bank containing the data. A core physical located next to a *Local* cache bank has to wait 10 cycles to access the bank. The maximum possible latency, without significant network contention, is equal to 70 cycles (i.e core 0 to access the *Local bank* next to core 7 since it requires 7 hops). Table I includes the basic system parameters that have been selected for our baseline system.

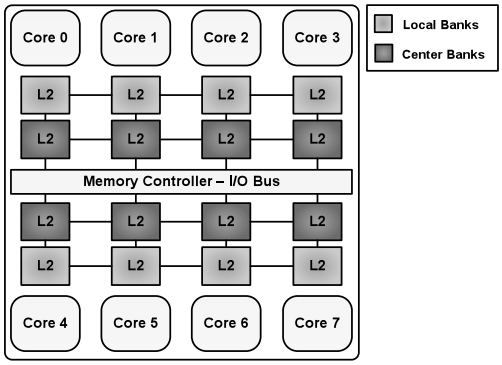

Fig. 1. Baseline CMP system

III. Bank-aware Cache Partitioning

In this section we elaborate on our proposed *Bank-aware* cache partitioning scheme. We start by providing details about our application profiling mechanism followed by our partitioning algorithm for assigning cache capacity to each core. In the end, we describe the cache partitions allocation algorithm for allocating the cache partitions on our CMP-baseline system.

A. Cache Profiling of Applications

In order to dynamically profile the cache requirements of each core, we implemented a cache miss prediction model

TABLE I. Baseline DNUCA-CMP parameters

Memory Subsystem		Core Characteristics	
L1 Data & Inst. Cache	64 KB, 2-way set associative, 3 cycles access time, 64 Bytes cache block size	Clock Frequency	4 GHz
L2 Cache	16 MB (16 x 1MB banks), 8 ways set associative, 10-70 cycles bank access, 64 Bytes cache block size	Pipeline	30 stages / 4-wide fetch / decode
Memory Latency	260 cycles	Reorder Buffer / Scheduler	128/64 Entries
Memory Bandwidth	64 GB/s	Branch Predictor	Direct YAGS / indirect 256 entries
Memory Size	4 GB of DRAM		
Outstanding Requests	16 requests / core		

based on Mattson's stack distance algorithm. Mattson's stack algorithm (MSA) was initially proposed by Mattson et al. in [22] for reducing the simulation time of trace-driven caches by determining the miss ratios of all possible cache sizes with a single pass through the trace. The basic idea of the algorithm was later used for efficient trace-driven simulations of a set associative cache [25]. More recently, hardware-based MSA algorithms have been proposed for CMP system resource management [15][26].

MSA is based on the inclusion property of the commonly used *Least Recently Used* (LRU) cache replacement policy. Specifically, during any sequence of memory accesses, the content of an *N*-sized cache is a subset of the content of any cache larger than *N*. To create a profile for a *K*-way set associative cache we need *K+1* counters, named $Counter_1$ to $Counter_{K+1}$. Every time there is an access to the monitored cache we increment only the counter that corresponds to the LRU stack distance where the access took place. Counters from $Counter_1$ up to $Counter_K$ correspond to the *Most Recently Used* (MRU) up to the LRU position in the stack distance, respectively. If an access touches an address in a cache block that was in the *i*-th position of the LRU stack distance, we increment $Counter_i$ counter. Finally, if the access ends up being a miss, we increment $Counter_{K+1}$. Efficient directories have been implemented exclusively in hardware using set sampling [28] and partial tags [27] in previous work [15].

Fig. 2 demonstrates such a MSA profile for an application running on an 8-way associative cache. The application in the example shows a good temporal reuse of stored data in the cache since the MRU positions have a significant percentage of the hits over the LRU one. Based on the application spatial and temporal locality, the graph of Fig. 2 can change accordingly. Using the inclusion property of the LRU replacement policy

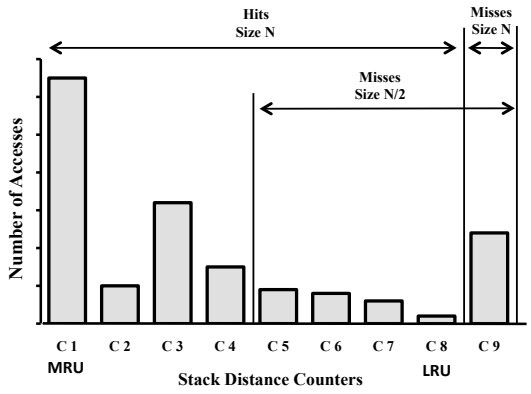

Fig. 2. LRU histograms based on Mattson's stack algorithm

and having a MSA profile of an *N*-sized cache, allow us to make a straight-forward prediction of the misses for every L2 cache with size smaller than *N*. For example, the number of misses that will occur if we make the cache of Fig. 2 half the size, that is using 4 ways instead of 8 ways, would be the previously measured misses plus the hits of the positions 5 up to 8 of the previous case LRU stack distance. For those positions, the LRU replacement policy will replace the stored data to make room for the one in the MRU positions before they are accessed again. Therefore the accesses that were previously recorded as hits would be misses in the 4-way cache case.

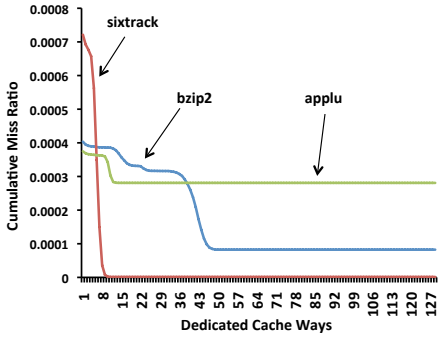

Fig. 3. LRU histograms examples of SPEC CPU2000 benchmarks

Fig. 3 shows the projected cumulative miss ratio of three benchmarks of SPEC CPU2000 benchmark suit [33]. We selected three examples, out of the 26 SPEC CPU2000 workloads that we simulated, as examples of varied behavior within the whole suit. To create the figure, we collected the stack distance profiles of *bzip2*, *sixtrack* and *applu* with each application executing stand-alone on our baseline CMP using just a single core. To collect the profiles we captured the L2 cache accesses of each core and fed it to the MSA histogram profiler described above. The x-axis represents the number of cache ways that are dedicated to each application and the y-axis shows the MSA-based projected cumulative miss rate of each application. *Sixtrack* features a lot of misses with less than six cache ways dedicated to it but after that point, by giving more ways, its misses are close to zero. Therefore, we

can have a good fit of *sixtrack's* cache requirements using only one bank (8 ways). In the same way, *applu* shows a reduction of misses when more than ten ways are dedicated to it, but in this case the miss rate remains flat after more than 10 ways. Therefore, assigning more ways is not beneficial for *applu*. Lastly, *bzip2* shows a behavior somewhere between the two previous cases since additional assigned ways improve miss ratio up to the point where dedicating 45 ways that finally flattens out. Consequently, MSA-based profiling allows us to monitor each core's cache capacity requirements during the execution of an application and based on which we can find the points of cache allocation that can benefit the miss ratio the most.

The hardware overhead of the profiling structure is primarily defined by the implementation of the necessary cache directory tag shadow copy. These cache block tags are necessary for identifying which cache block is assigned at each one of the *hit* counters of Fig. 2 and allow a detailed monitoring of resource requirements on a cache block granularity on our LLC. Additional overhead is introduced by the implementation of the *hit* counters themselves for each cache way, but since those counters are shared over all the available cache-ways, their overhead is significantly lower than the cache block tag information for every set.

A simple with no restrictions implementation would require a complete copy of the cache block tags for each cache set in each one of MSA profilers, which is prohibitively high. The overhead can be greatly reduced using: *a*) *partial tags* [27] *b*) *set sampling* [28] and *c*) *maximum assignable capacity* reduction techniques. With *partial tags* one can use less than full tags to identify the cache blocks assigned at each counter thus reducing the storage overhead. *Set sampling* involves the profiling of a fraction of the available cache sets and therefore it also reduces the number of stored cache tags in the circuit. In addition, the *maximum assignable capacity* approach assumes that the number of cache-ways that can be assigned to each core is less than the overall number of available cache-ways. In that case, the number of counters are reduced to the maximum number of assignable cache ways per core. The first two reduction techniques are subject to aliasing, which introduces errors and affects the overall accuracy of our profiling circuit. In addition, the *maximum assignable capacity* can potentially restrict the effectiveness of our partitioning scheme by not dedicating bigger portions of a cache to a specific core.

TABLE II. Overhead of the proposed MSA profiler

Structure Name	Overhead Equation	Overhead
Partial Tags	$tag\_width * ways * cache\_sets$	54 kbits
LRU Stack Distance Implem.	$((lru\_pointer\_size * ways) + head/tail) * cache\_sets$	27 kbits
Hit Counters	$cache\_ways * hit\_counter\_size$	2.25 kbits

In this paper we propose an implementation based on all of the above methods. Our overhead analysis showed that the use of 12 bit *partial tags* combined with 1-in-32 *set sampling* produced error rates within 5% of the profiling accuracy obtained using a full tag implementation. In addition, our *Bank-aware partitioning* assignment algorithm limits each core to a maximum of 9/16 of the total cache capacity. The hardware overhead of the proposed implementation for every necessary structure is included in Table II. Overall, the implementation overhead is estimated to be 83.25 kbits per cache profiler, which is approximately 0.4% of our 16MB LLC cache design for all the profilers.

B. Bank-aware Assignment of Cache Capacity

Prior work in MSA-based cache partitioning was analyzed on fully configurable caches shared among a small number of CPUs [8][15]. We refer to this type of partitioning algorithm as *Unrestricted*. On the other hand, while Huh et al. in [16] proposed a method for partitioning a CMP-NUCA cache, this relied on a highly banked structure that, as we explained in Section II, features an unrealistic physical implementation. As a solution, we propose a method to partition cache bank structures using a MSA-based profiling mechanism aligned with current industry directions, that is, using a smaller number of higher capacity cache banks. Such configuration limits the granularity of possible partitions and imposes a set of restrictions over the *Unrestricted* techniques proposed in the past. This is rooted in the need to aggregate multiple cache banks into a single partition. In the following we discuss several potential aggregation methods that are shown in Fig. 4. Aggregation possibilities:

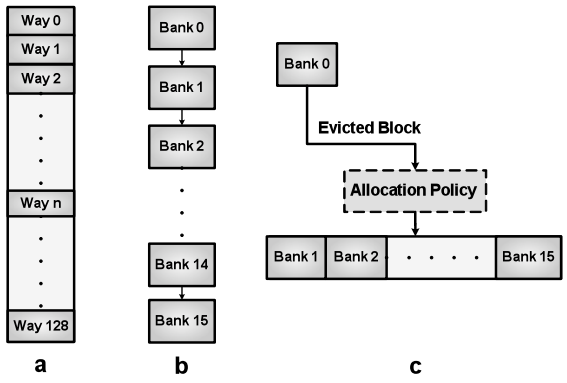

Fig. 4. Cache banks aggregation schemes

1) **Cascade:** In this approach, all cache banks that contain portions assigned to a given core are connected head to tail. To match the MSA LRU strategy (Fig. 4.a), all allocations are placed as MRU at the head of the chain. Each allocation causes a shift down of the LRU. Evictions are passed down the chain from LRU out to MRU position in the next bank until a free spot is located (potentially formed from the cache hit that was moved to the top). This structure is show in Fig. 4.b. This method provides for an LRU policy (assuming the banks are also LRU). The advantage of this method is that one can stitch together arbitrary fractions of banks which will emulate

the MSA very closely. The primary disadvantage is the high migration rate between cache banks.

2) **Address Hash:** A common approach to cache bank aggregation is the use of an *address hash*. Typically this method is used with a power of two number of cache banks, such that lower order address bits can directly select the bank. While systems have also been built with non-power of two hashes, these require complex modulo operations in the hardware hash function. An example would be the IBM POWER4 and POWER5 processors, which hash across three banks [29]. In addition, Gao et al. [30] and Seznec et al. [31] proposed non-power of two hashing functions with increased complexity over simple hashing functions. Irrespective of the number of cache banks aggregated, this method requires symmetry in that each hashed bank must have the same cache capacity. As such this method has some restrictions. Lastly, address hash features low migration rates.
3) **Parallel:** This method is very much like *Address Hash*, except that a line can be stored in any of the cache banks. Allocation is controlled by round robin/random selection. As such, any given line can be stored in any of the cache banks. This forces additional look-up operations in the directory structure (which we implement as partial tags). This is less restrictive than *Address hash* in bank configuration, in that, non-power of 2 aggregations of banks are possible without complex modulo address computations. The migration rate is equivalent to *Address Hash*, however, power is higher due to wider directory look-ups.

Even though *Cascade* provides the greatest flexibility, the migration rates observed in simulation are prohibitively high. Both *Address hash* and *Parallel* provide reasonable solutions to aggregating cache banks. The only restriction is that multiple banks must have the same capacity. We demonstrate in our analysis that the degradation can be mitigated using the structure shown in Fig. 4.c. In this structure we limit the level of cascading to two. The allocation policy can be either *Address Hash* or *Parallel*. In our analysis we use *Parallel*.

These issues present problems in direct application of currently proposed *Unrestricted* cache partitioning schemes [15] and as a solution we propose a *Bank-aware* assignment algorithm. This algorithm is based on progressive control of bank granularity. Essentially, as the capacity assignment increases, small deviations from the ideal assignment are tolerable with respect to overall miss reduction. Based on this observation and the bank aggregation requirements, we propose the following policies:

1) *Center* cache banks are completely assigned to a specific core. This prevents situations where aggregated banks are of different capacities.
2) Any core that is allocated *Center* banks, will receive a full *Local* bank.
3) *Local* cache banks can only be shared with an adjacent core. We only allow per assignment control at *Local* cache banks. In addition, requiring adjacent sharing provides for low latency and minimal network loads for data transfers.

A typical allocation is shown in Fig. 5. From the figure, most of the cores have multiple L2 cache banks allocated to them except core 2 and core 5. Those cores share the capacity of a single L2 bank with core 3 and 4, respectively.

To enforce the selected cache partitions, we modified the typical design of a cache bank to support a vertical, fine-grain, cache-way partitioning scheme, as was proposed in [8]. According to this scheme, each cache-way of a set associative cache can belong to one or more specific cores. When a specific core suffers a cache miss, a modified LRU policy is used to select the least recently used cache block among the ones that belong to that specific core, for replacement. Therefore, only cache-ways that belong to a specific core or set of cores can be accessed and the rest of the cache-ways that belong to other cores are not affected, eliminating the destructive interference between different workloads running on different cores. To reduce the design complexity, all of the sets in a cache bank are vertically partitioned with the same cache-ways assignment and therefore the granularity of assigning a different cache-way partition is a single cache bank.

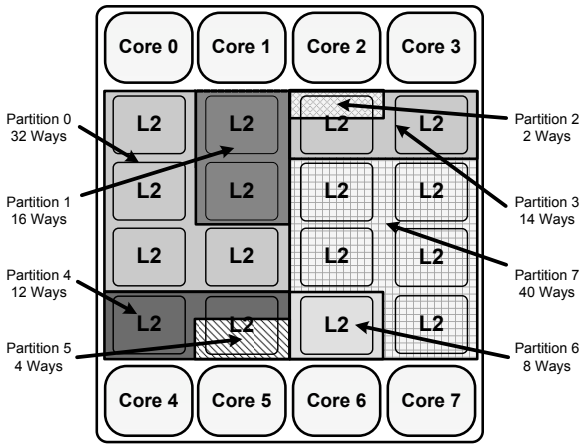

Fig. 5. An example of typical CMP cache partitioning

C. Allocation Algorithm on CMP

In this section we describe in details our *Bank-aware* assignment algorithm. We based the assignment policy on the concept of *Marginal Utility* [32]. This concept originates from economic theory, where a given amount of resources (in our case cache capacity), provides a certain amount of utility (reduced misses). The amount of utility relative to the resource is defined as the *Marginal Utility*. Specifically, *Marginal Utility* is defined as:

$$MarginalUtility(n) = MissRate(c+n) - MissRate(c)/n$$

The MSA histogram provides a direct way to compute the *Marginal Utility* for a given workload across a range of

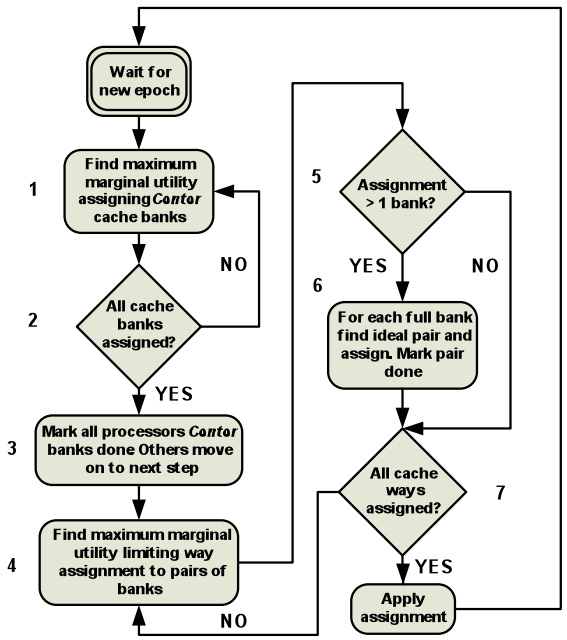

Fig. 6. Cache allocation algorithm flow chart

possible cache allocations. We use this capability to make the *best* use of the limited cache resources. We follow an iterative approach, where at any point we can compare the *Marginal Utility* of all possible allocations of unused capacity. Of these possible allocations, the maximum *Marginal Utility* represents the *best* use of an increment in the assigned capacity.

Our algorithm arrives at a capacity assignment via successive steps determining the maximum *Marginal Utility* for a subset of processors and assignment restrictions. The overall flow is shown in Fig. 6. The first step of the algorithm is to assign the cache-ways in *Center* cache banks. Following that, in Box 1, the maximum *Marginal Utility* is calculated and cache banks are assigned accordingly. For the calculations, we assume that each *Local* bank is assigned to the associated processor. In Box 2, we check if all the banks are assigned, if not, step 1 is repeated. Following Rules 1 and 2, we mark all processors with *Center* banks complete (Box 3). The next steps are used to solve the *Local* cache bank partitions. In Box 4, we once again find the maximum *Marginal Utility*, but assignments are limited to possible pairs of processors (in keeping with Rule 3). In Box 5, we check if the new assignment has caused any processor to overflow into another processors *Local* region. If so, we find the ideal pair with respect to minimal misses. Essentially we defer the pairing as many steps as possible, and make the best pairing choice once it is decided a processor should receive a fraction of an adjacent *Local* bank. Once the pair is assigned, both processors are marked complete. This step is repeated until all the cache ways are assigned.

IV. Evaluation

To evaluate our proposed scheme we utilized a full system simulator, modeling an 8-core SPARCv9 CMP under Solaris 10 OS. Specifically, we used Simics [20] as the full system functional simulator extended with the Gems toolset [21] to simulate an out-of-order processor and memory subsystem. The CMP-NUCA design was implemented in Gems' memory timing model (Ruby) extended with the support of the fine-grain L2 cache partitioning scheme described in Section III. The memory system timing model includes a detailed message-based model of the inter-chip network using a MOESI cache coherence protocol. Throughout the paper, the frequency of evaluating and reallocating the L2 cache partitions was set to a 100M cycle epoch.

We use SPEC CPU2000 [33] scientific benchmark suite, compiled to SPARC ISA with peak configurations, as the workload of our proposed scheme. We fast forward all the benchmarks for 1 billion instructions, and use the next 100M instructions to warm up the CMP-NUCA L2 cache. Each benchmark was simulated in our CMP-NUCA using Gems for a slice of 200M instructions after cache warm up. Table I includes the basic system parameters that were used.

A. Bank-aware vs. Unrestricted Partitioning

The analysis of computer systems in virtualization environments is an open problem. In these systems an arbitrary mix of work may share a server at any given time. The general problem of performance analysis using benchmarks is greatly compounded by the possible combinations of workloads. In order to limit the state space, we base our evaluation on the workloads of the SPEC CPU2000 integer and floating point benchmark suites. Even with this limitation the possible combinations of the 26 workloads on our eight core target machine is very large and equal to:

$$C(num\_workloads + num\_cores \text{ - } 1, num\_cores) = C(26 + 8 \text{ - } 1, 8)$$

This is $\sim$14 million possible workload combinations. Based on this very large state space, we employ a comparative Monte Carlo approach to the evaluation of our assignment algorithm.

Total system miss rates are estimated from projecting miss rates based on MSA data collected from our detailed system simulations. To accurately cover the workload state space would require far too many simulations points than are possible assuming a full simulation of all the cases. Instead, we used the estimated method here, combined with detailed simulations of a more manageable number of workload mixes as a second form of validation. Our comparison methodology is as follows:

1) Collect MSA histograms for a mix of workloads. In our case these are the 26 components from SPEC CPU2000.
2) From these 26 components, we randomly select (with repetition) 8 workloads.
3) Execute both the *Unrestricted* and *Bank-aware* partition algorithms.
4) Compare the MSA predicted miss rates between the two partition algorithms and the case of fixed partitions of 2MB per core.

The above process was executed for 1000 random workload assignments. For each workload set, we compared the MSA

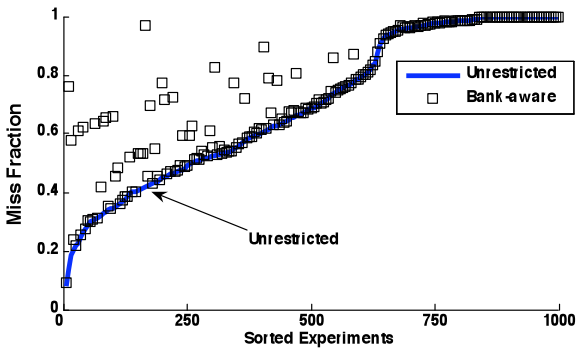

Fig. 7. Relative miss ratio to fixed-share for *Unrestricted*

miss rate based on a fixed even share per core (16 ways for each core), the *Unrestricted* algorithm, and the *Bank Aware* algorithm.

In Fig. 7 we show the miss rate relative to the even partitions for the *Unrestricted* and *Bank-aware* algorithms. A ratio of one represents no reduction in misses compared to static fixed partitioning, while zero indicates all misses are removed with the MSA-based partitioning scheme. We have sorted the 1000 results with respect to the miss rate reduction of the *Unrestricted* scheme. The even partitions and *Unrestricted* essentially form a performance envelope. Ideally, all of the *Bank-aware* assignments would fall on the *Unrestricted* line, which is in general true except some outliners that achieved smaller miss rate reductions than *Unrestricted*. Both figures give an indication of the range of miss rate reductions possible. On average, the miss rate reduction from the *Unrestricted* and *Bank-aware* algorithms are quite comparable. The *Unrestricted* averages a 30% reduction in misses compared to 27% for the *Bank-aware* over the case of even partitions. This result shows that the restrictions placed on the allocation algorithm due to the more realistic implementation of L2 cache do not adversely affect the benefits of our MSA-based dynamic cache partitioning scheme.

TABLE III. Set of 8-core experiments

Exp. Set	Benchmarks & "cache-ways" assignments from core0 to core7 [benchmark(#ways)]
1	apsi(12), galgel(4), gcc(2), mgrid(16), applu(16), mesa(8), facerec(56), gzip(8)
2	crafty(12), gap(4), mcf(24), art(16), equake(8), equake(8), bzip2(48), equake(8)
3	applu(12), galgel(4), art(16), art(16), sixtrack(16), gcc(6), mgrid(40), lucas(16)
4	mgrid(40), mcf(24), art(16), equake(16), gcc(6), equake(10), sixtrack(6), crafty(10)
5	facerec(56), fma3d(8), sixtrack(16), apsi(16), fma3d(6), ammp(10), lucas(6), swim(10)
6	bzip2(48), gcc(8), twolf(16), mesa(24), wupwise(6), applu(10), fma3d(6), ammp(10)
7	swim(8), parser(16), mgrid(40), twolf(16), fma3d(2), parser(14), swim(8), mcf(24)
8	ammp(13), eon(3), swim(11), gap(5), gcc(8), art(16), twolf(56), art(16)

B. Detailed Simulation Results

We randomly chose eight workload sets from the previous simulations to evaluate the proposed partitioning scheme on the 8-core full system shown in Fig. 1. Table III shows the selected workloads along with the cache-ways that were assigned to each core by the *Bank-aware* partitioning scheme. Fig. 8 and Fig. 9 shows the relative miss rate and CPI of *Equal-partitions* and *Bank-aware* partitioning over the simple case of *No-partitions*. *Equal-partitions* is equivalent to assigning private cache partitions of equal size to each core. From the figures, both partitioning schemes show a significant reduction in misses and CPI over the simple *No-partitions* one, which is a strong indication of the need for partitioning the last level of cache. On average, *Bank-aware* shows a 70% and 43% reduction in misses and CPI over *No-partitions*, respectively. Moreover, from Fig. 8, our *Bank-aware* partitioning scheme shows on average a 25% reduction over simple *Equal-partitions*. This reduction is inline with the reduction estimated in our Monte Carlo experiment of the previous section. In addition, Fig. 9 shows that *Bank-aware* partitioning can achieve an 11% reduction in CPI over the *Equal-partitions* scheme. Comparing Fig. 8 and 9, some sets of workloads demonstrate a much higher performance sensitivity to misses than others since a reduction on L2 misses does not always result in an equal size reduction in CPI. For example, in *Set 1* even though we significantly reduced the overall fraction of misses, that reduction in not translated in CPI gain due to the overall small number of misses in that set and the performance characteristics of the applications that feature the highest miss reduction.

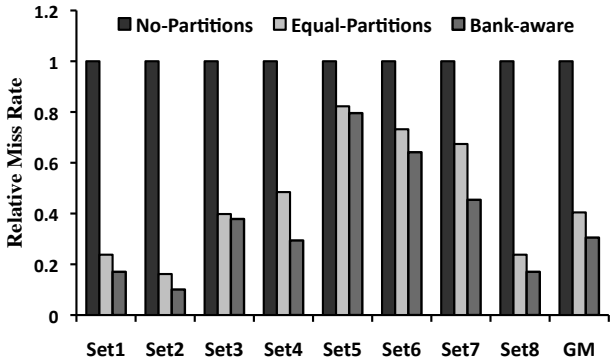

Fig. 8. Relative miss rate of 8-core sets over the no-partitioning scheme

V. Conclusions

Shared resource contention in CMP platforms has been identified as a key performance bottleneck that is expected to become worse as the number of cores on a chip continues to scale to higher numbers. Many solutions have been proposed, but most assume either simplified cache hierarchies with no realistic restrictions or complex cache schemes that are difficult to integrate in a real design. Therefore, both approaches could lead to conclusions that are unrealistic when implemented in a real system. In this paper we highlight the

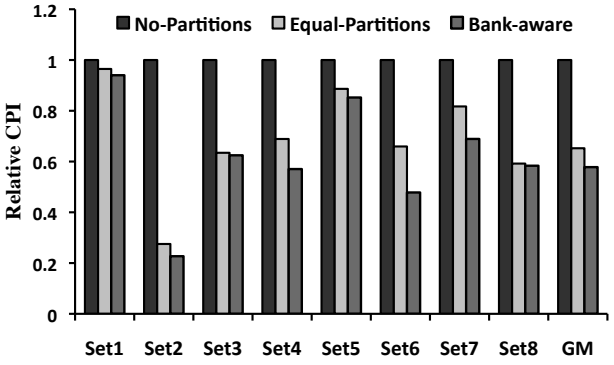

Fig. 9. Relative CPI of 8-core sets over the no-partitioning scheme

problem of sharing the last level of cache in CMP platforms and motivate the need for a low-overhead cache partitioning scheme that is aware of the banking structure of the L2 cache design. Our proposed *Bank-aware* partitioning scheme demonstrates a 70% reduction in misses compared to non-partitioned shared caches, and a 25% miss rate reduction compared to even partitioned (private) caches. Lastly, our proposed scheme managed, on average, the same miss reduction achieved with less realistic proposed *Unrestricted* schemes that are unaware of implementation restrictions.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. This research was supported in part by NSF Award number 0702694 and IBM. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and IBM.

REFERENCES

[1] K. Krewell, "Best Servers of 2004: Multicore is Norm. Microprocessor Report", www.mpronline.com, Jan 2005
[2] K. Olukotun, et al. "The case for a single-chip multiprocessor", In Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct 1996
[3] P. Barham, et al. "Xen and the Art of Virtualization", In Proc. of the ACM Symposium on Operating Systems Principles (SOSP), Oct 2003
[4] P. Ranganathan and N. Jouppi. "Enterprise IT Trends and Implications on Architecture Research", In Proc. of the 11th International Symposium on High Performance Computer Architecture (HPCA), Feb 2005
[5] R. Uhlig, et al., "Intel Virtualization Technology", IEEE Transactions on Computers, 2005
[6] D. Chandra, F. Guo, S. Kim and Y. Solihin, "Predicting inter-thread cache contention on a chip multiprocessor architecture", In Proc. 11th International Symposium on High Performance Computer Architecture (HPCA), 2005
[7] L. Hsu, S. Reinhardt, R. Iyer and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource", International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006
[8] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms", 18th Annual International Conference on Supercomputing (ICS'04), July 2004
[9] C. Kim, D. Burger, S. W. Keckler, "Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches", IEEE Micro 23(6): 99-107, 2003
[10] K. J. Nesbit, et al. , "Multicore Resource Management", IEEE Micro special issue on Interaction of Computer Architecture and Operating Systems in the Manycore Era, May-June 2008
[11] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multiprocessor architecture", In Proc. of the 11th International Symposium on High Performance Computer Architecture (HPCA), 2005
[12] H. Kannan, F. Guo, L. Zhao, et al., "From Chaos to QoS: Case Studies in CMP Resource Management", dasCMP/Micro, Dec 2006
[13] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", 13th Int Conf. on Parallel Architectures & Complication Technique (PACT), 2004
[14] C. Liu, A. Sivasubramaniam, M. Kandemir, "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs" 10th IEEE Symp.on High-Performance Computer Architecture (HPCA), Feb. 2004
[15] M. K. Qureshi and Yale N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", MICRO 2006
[16] J. Huh, et al. "A NUCA Substrate for Flexible CMP Cache Sharing",IEEE Transactions on Parallel and Distributed Systems, Vol. 18, issue. 8, pages 1028-1040, 2007
[17] R. Ho, K. W. Mai, and M. A. Horowitz. "The Future of Wires", Proceedings of the IEEE, 89(4):490-504, Apr. 2001
[18] D. Sylvester and K. Keutzer. "Getting to the Bottom of Deep Submicron II: a Global Wiring Paradigm", In Proceedings of the 1999 International Symposium on Physical Design, pages 193-200, 1999
[19] B. M. Beckmann, et al. "Managing Wire Delay in Large Chip-Multiprocessor Caches", Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37), 2004
[20] Simics Microarchitect's Toolset, http://www.virtutech.com/
[21] Milo M.K. Martin, et al. "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", Computer Architecture News (CAN), September 2005
[22] R. L. Mattson. et al. "Evaluation techniques for storage hierarchies". IBM Systems Journal, 9(2):78-117, 1970
[23] Intel Corporation, "First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)", White Paper, http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf, 2008
[24] N. Muralimanohar, et al. "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0", Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, Pages 3-14, 2007
[25] M. D. Hill and A. J. Smith. "Evaluating associativity in CPU caches". IEEE Transactions on Computers, 38(12), 1989
[26] Pin Zhou, et al. "Dynamic Tracking of Page Miss Ratio Curve for Memory Management", Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2004, pages 177-188, Boston, USA
[27] R. Kessler, R. Jooss, A. Lebeck, and M. Hill, "Inexpensive implementations of set-associativity", 16th Annual International Symposium on Computer Architecture, pages 131139, 1989
[28] R. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches", IEEE Transactions on Computers. 43(6):664675, 1994.
[29] J. M. Tendler, et al. "Power4 system microarchitecture", IBM Journal of Research and Development, 46(1), 2002
[30] Q. S. Gao, "The Chinese Remainder Theorem and the Prime Memory System", International Symposium on Computer Architecture (ISCA) '93, pages 337-340, 1993
[31] Andre Seznec, Jacques Lenfant, "Odd Memory Systems May be Quite Interesting", International Symposium on Computer Architecture (ISCA) '93, pages 341-350, 1993
[32] Wieser, Friedrich von, "Der naturliche Werth [Natural Value]", Book I, Chapter V, 1889
[33] SPEC CPU2000 Benchmark Suit, http://ftp.spec.org/cpu2000/