A Bandwidth-aware Memory-subsystem Resource Management using Non-invasive Resource Profilers for Large CMP Systems

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

***Abstract*—By integrating multiple cores in a single chip, Chip Multiprocessors (CMP) provide an attractive approach to improve both system throughput and efficiency. This integration allows the sharing of on-chip resources which may lead to destructive interference between the executing workloads. Memory-subsystem is an important shared resource that contributes significantly to the overall throughput and power consumption. In order to prevent destructive interference, the cache capacity and memory bandwidth requirements of the last level cache have to be controlled. While previously proposed schemes focus on resource sharing within a chip, we explore additional possibilities both inside and outside a single chip. We propose a dynamic memory-subsystem resource management scheme that considers both cache capacity and memory bandwidth contention in large multi-chip CMP systems. Our approach uses low overhead, non-invasive resource profilers that are based on Mattson's stack distance algorithm to project each core's resource requirements and guide our cache partitioning algorithms. Our bandwidth-aware algorithm seeks for throughput optimizations among multiple chips by migrating workloads from the most resource-overcommitted chips to the ones with more available resources. Use of bandwidth as a criterion results in an overall 18% reduction in memory bandwidth along with a 7.9% reduction in miss rate, compared to existing resource management schemes. Using a cycle-accurate full system simulator, our approach achieved an average improvement of 8.5% on throughput.**

I. Introduction

Chip Multiprocessors (CMP) have become an attractive architecture to leverage system integration by providing capabilities on a single die that would have previously occupied many chips across multiple small systems [16]. To achieve high efficiency and throughput, CMPs heavily rely on sharing resources among multiple cores. Under high capacity pressure, conventional greedy resource sharing policies lead to destructive interference [3], unfairness [15] and eventually lack of Quality-of-Service (QoS) [6], [15], i.e., lacking the ability to guarantee a certain performance or fairness level. In addition, CMPs are widely deployed in large server systems. These large systems typically utilize virtualization where many independent small and/or low utilization servers are consolidated [23]. Under such environment the resource sharing problem is extended from the chip-level to the system-level, and therefore system-level resources, like main memory capacity and bandwidth, have to be considered for the appropriate sharing policies. Consequently, to design the most effective future systems for such computing resources, effective resource management policies are critical not only in mitigating chip-level contention, but also in improving system-wide performance and fairness.

A great deal of research has recently been proposed on harnessing the shared resources at a single CMP chip level. The primary focus has been to control contention on the resources that most affect performance and execution consistency, that is the shared *last-level cache* (LLC) capacity [6], [8], [9], [15], [19], [20], [22], and the *main memory bandwidth* [21]. To address LLC contention, the proposed techniques partition the LLC cache capacity and allocate a specific portion to each core or execution thread. Such private partitions provide interference isolation and therefore can guarantee a controllable level of fairness and QoS. There are both static [8], [15] and dynamic partitioning algorithms proposed [9], [19], [20], [22] that use profiled workload information to make a decision on cache capacity assignment for each core/thread. On the other hand, to address the memory bandwidth contention problem, *fair queuing* network principles [21] have been proposed. Those techniques are based on assigning higher priorities on bandwidth use either to the most important applications (for QoS) or to the tasks that are close to missing a deadline (for fair use). In addition, machine learning techniques have also been employed in managing multiple interacting resources in a coordinated fashion [2]. Such methods require a hardware implemented artificial neural network to dynamically configure the appropriate partitions for different resources.

Prior work monitoring schemes can be classified into *trial-and-error* and *prediction-based* configuration exploration. *Trial-and-error* systems are based on observing the behavior of the system under various configurations while *predictive* systems are able to concurrently infer how the system will perform under many configurations. As the state space of the system grows, *predictive* schemes have inherent scalability advantages. Examples of representative *predictive* schemes that have been proposed in the past are the one from Zhou et al. [29] for main memory and Qureshi et al. [22] for LLC caches. Both are based on Mattson's stack distance algorithm [17] and are able to provide miss rate predictions of multiple LLC configurations in parallel. While these proposals provide reasonable speed-ups by solving the memory capacity assignment, they neglect the role of memory bandwidth constraints and system-level optimizations opportunities.

Virtualization systems are most effective when multiple CMP processors are aggregated into a large compute resource.

In such environments, optimization opportunities exist in the ability to dispatch/migrate jobs targeting full system utilization. Limiting optimizations to a single chip can produce sub-optimal solutions. Therefore, while previous methods have shown good gains within a single chip, we explore additional possibilities in the context of large multi-chip CMP systems.

In this paper, we propose a dynamic, bandwidth-aware, memory-subsystem resource management scheme that provides improvements beyond previous single chip solutions. Our solution aims at a global resource management scheme that takes into account: *a*) the need of low overhead, predictive non-invasive profilers that are able to capture the temporal behavior of applications and therefore guide our resource management algorithm, *b*) the limited available main memory bandwidth per chip and the performance degradation of the overall system when executing applications with over-commited memory bandwidth requirements, and finally *c*) the hierarchical nature of large CMP computer systems. In particular, the contributions of this paper are as follows:

1) We propose a low overhead, non-invasive, hardware profiler implementation based on Mattson's stack distance algorithm (MSA) [17] that can effectively project memory bandwidth requirements of each core in addition to cache capacity requirements that previous papers have proposed [22]. Our implementation requires approximately 1.4% of the size of an 8MB LLC and introduces an *Effective error* of only 3.6% in bandwidth and 1.3% in cache capacity profiling. This MSA profiler can guide our fine-grained dynamic resource management scheme and allow us to make projections of capacity and bandwidth requirements under different LLC partition allocations.
2) We propose a system-wide optimization of resource allocation and job scheduling. Our scheme aims to achieve overall system throughput optimization by identifying over-utilized chips, in terms of memory bandwidth and/or cache capacity requirements. For those chips, a set of job migrations is able to balance the utilization of resources across the whole platform leading to improved throughput. Based on our experiments, our bandwidth-aware scheme is able to achieve a reduction of 18% reduction in memory bandwidth along with a 7.9% reduction in miss rate and an average 8.5% increase in IPC, compared to single chip optimization policies.

To the best of our knowledge, this is the first proposed scheme that systematically combines memory bandwidth and cache capacity requirements for memory-subsystem resource management in large multi-chip CMP systems, looking for optimizations within and beyond a single chip.

The rest of the paper is organized as follows. Section II states our baseline multi-chip CMP design. In Section III we elaborate on our proposed *Bandwidth-aware* cache partitioning scheme. Section IV describes the simulation and evaluation methodology and reports our experimental results. Finally, Section V presents the related work in the literature.

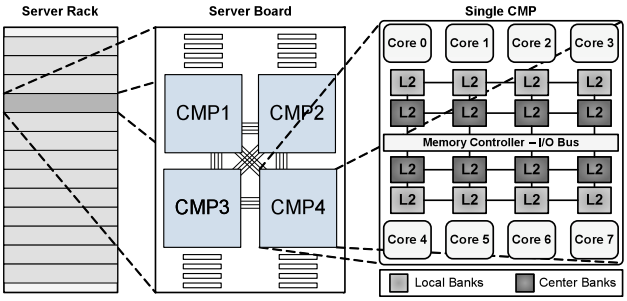


Fig. 1. Baseline CMP system

TABLE I. Single-Chip CMP parameters

Memory Subsystem		Core Characteristics	
L1 Data & Inst. Cache	64 KB, 2-way associative, 3 cycles access time, 64 Bytes block size, Pseudo-LRU	Clock Frequency	4 GHz
L2 Cache	8 MB (16 x 512KB banks), 8-way associative, 10-70 cycles bank access, 64 Bytes block size, Pseudo-LRU	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		

II. Baseline System Architecture

Fig. 1 demonstrates an example of server construction in a large CMP environment where machine racks are filled with high density machines. In such systems, the servers are typically comprised of 2 to 4 packaged chips, with each chip being a CMP. In our baseline system, each CMP features 8 cores sharing a DNUCA-like [1] cache design as our LLC. Work can be partitioned and migrated as needed, based on scheduling and load balancing algorithms.

The rightmost part of Fig. 1 (*Single CMP*) demonstrates our baseline CMP system. The selected LLC contains 16 physical banks with each cache bank configured as an 8-way set associative cache. The overall LLC capacity was set to 8MB. An alternative logical interpretation of the cache architecture is as a 128-way equivalent cache separated in sixteen 8-way set associative cache banks. The eight cache banks physically located next to a core are called *Local banks* and the rest are the *Center banks*. Table I includes the basic system parameters selected for our baseline system.

III. Bandwidth-aware Resource Management

A. Applications' Resource Profiling

In order to dynamically profile the memory-subsystem resource requirements of each core, we implemented a prediction scheme that is able to estimate both cache misses and memory bandwidth requirements. This prediction scheme is

contrasted against typical profiling models in that we estimate resource behavior of all possible configurations concurrently, as opposed to profiling the currently configured resources. Our prediction model is based on Mattson's stack distance algorithm (MSA), which was initially proposed by Mattson et al. [17] for reducing the simulation time of trace-driven caches. More recently, hardware-based MSA algorithms have been used for CMP system resource management [22], [29].

To predict memory access behavior in addition to cache miss rates we extend previously proposed hardware solutions [22], [29]. Specifically, our scheme predicts the memory write traffic produced by the eviction of modified lines in a write-back cache. In the following subsections, we first present a brief description of the baseline MSA algorithm for profiling LLC misses followed by the description of the additional structures needed to predict the memory bandwidth requirements.

1) MSA-based Profiler for LLC Misses: 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 $Counter_1$ 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 the $Counter_i$ counter. Finally, if the access ends up being a miss, we increment the $Counter_{K+1}$. The *Hit Counter* of Fig. 2 demonstrates such a MSA profile for *bzip2* of SPEC CPU2006 suite [26] 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. The graph of Fig. 2 can change accordingly to each application's spatial and temporal locality.

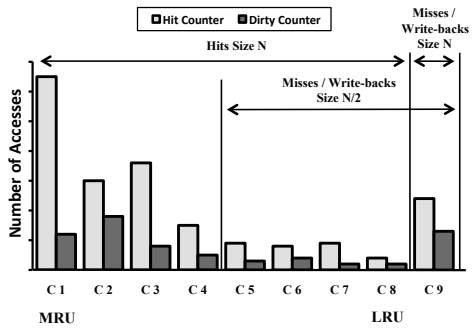


Fig. 2. Example of misses and bandwidth MSA histograms of *bzip2*

2) MSA-based Profiler for Memory Bandwidth: In addition to capacity misses, we augmented the MSA circuit to estimate the required memory bandwidth. There are two components that must be addressed: *a*) read bandwidth due to *cache fills*, and *b*) write bandwidth due to cache evictions (dirty *write-backs* to memory). The bandwidth needed for fetching data from main memory required by a cache miss (*cache fills*) can be derived from the previously described MSA circuit. The number of misses estimated by MSA is proportional to the needed memory bandwidth to fetch the missed data with each miss representing a cache-line of bandwidth.

To project the *write-back* bandwidth, we exploit the following property. In our MSA structure, hits to dirty lines indicate a write-back operation if the cache capacity allocation is smaller than its stack distance. Essentially, the store data must have been written back to memory and re-fetched since the cache lacked the capacity to hold the dirty data. This process is complicated in that only one write-back per store should be accounted for. To track these possible write-backs to memory we propose the following structure.

For each cache line tracked in the MSA structure we add a *Dirty Bit* and a *Dirty Stack Distance* indicator (register). In addition, we add a dirty line access counter, named *Dirty Counter* for each cache way. The *Dirty Stack Distance* is used to track the largest stack distance at which a dirty line has been accessed. We cannot simply update the *DirtyCounter* access counter on each hit of a dirty line, since this will give multiple counts for one store. In addition, we cannot reset the *Dirty Bit* on an access, since a future access to the same line at a greater stack distance must be accounted for in the dirty access counters. Essentially, we must track the greatest stack distance that each store is referenced. Pseudocode 1 describes how we update the *Dirty Bits*, *Dirty Counters* and *Dirty Stack Distance* registers.

Pseudocode 1 Write-back dirty access hardware description

```
if (access is a hit) {/* Handling Dirty Counters & Dirty Stack Distance */
    hit_distance <= stack distance of hit
    hit_dirty <= dirty bit set in hit entry
    dirty_stack_distance <= dirty distance value in hit entry
    if (hit_dirty and(hit_distance > dirty_stack_distance)) {
            DirtyCounter[dirty_stack_distance] - -;
            DirtyCounter[hit_distance] ++;
            dirty_stack_distance = hit_distance;
    }
} else{ /* access is a miss */
        /* Hanlde deallocation of evicted line */
        DirtyCounter[dirty_stack_distance] - -;
        K+1_Counter++;
        /* Handle new line allocation */
        dirty_bit = 0
}
/* Handling a store operation */
if (is_store) {
        dirty_bit = 1
        dirty_stack_distance = 0
}
```

The dirty bit is only reset when the line is evicted from the cache. At eviction time an additional counter ($K+1$ counter of Fig. 2) tracks the number of write-back operations. This number gives the write-back rate for a cache size corresponding to the maximum capacity tracked. The projection of write-back rates can then be made from the *DirtyCounters*. For each cache size projection, the sum of all counters larger than the

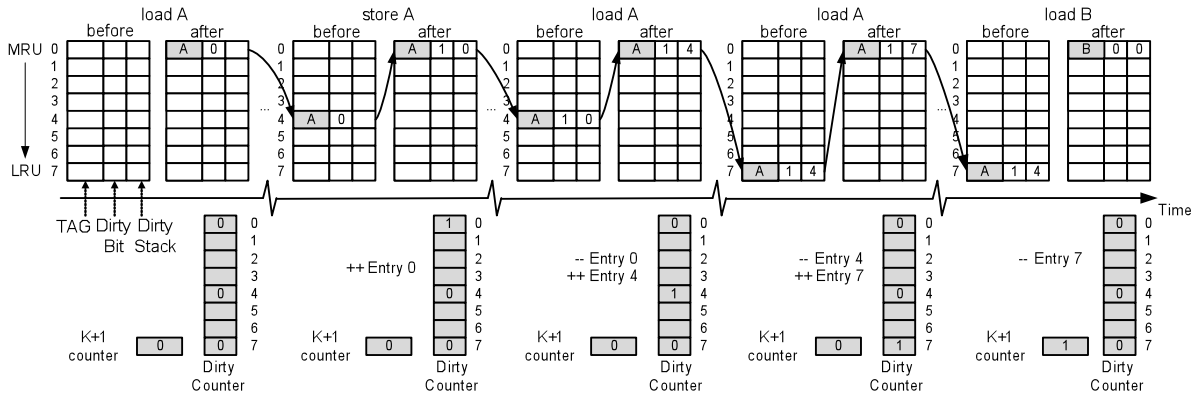


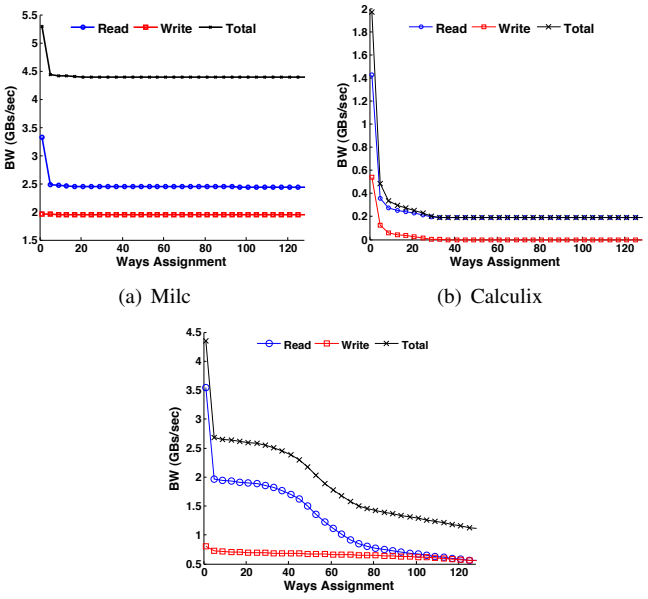
Fig. 3. Example of MSA-based write-back bandwidth profiling

allocated size indicates the number of write-back operations sent to main memory.

Fig. 3 demonstrates how we track the maximum stack distance which a given store is referenced. In the example, we start with a store to a previously allocated clean line. This will set the *Dirty Bit* to 1, set the *Dirty Stack Distance* to 0, and increment the entry '0' of the histogram. Essentially, if the cache had no free capacity, the store operation would immediately produce a write to memory. Following the store operation, we show two load hits of the MSA structure. The first hit is at a stack distance of 4. As such any cache that is smaller than 4 ways would have not been able to contain the store data. We then move the accounting for the store from entry 0 to entry 4 of the histogram. As we detect a load that hits entry 7, we must move the accounting to entry 7. If the line is evicted, this line results in a write-back for any of the captured sizes. The key insight shown here is that each store will result in exactly one net increment across the histogram and *K+1 Counter* (*K+1 Counter* of Fig. 2). Increments to larger stack distances leading to the *K+1 Counter* are more powerful in that avoiding the write-back to memory requires a larger cache assignment.

Overall, our MSA histograms allow us to make a prediction for both the number of misses and the required memory bandwidth dependent on the number of ways that a core is assigned. An example of such an MSA profile is shown in Fig. 2 where both MSA histograms and dirty evictions are shown.

3) Examples of Real Applications' Histograms: Fig. 4 demonstrates the three most representative categories of memory bandwidth requirements that we observed examining SPEC CPU 2006 [26] suite. As we explained before, the read bandwidth is highly correlated with cache miss rate and therefore previous techniques indirectly account for it. From the figure, *milc* features a write rate almost equal to the read rate. In this workload, complex matrix data structures are modified at each iteration, producing a high fraction of modified data. In contrast, the *calculix* benchmark uses cache blocking of matrix multiplications and dot product operations to condense and contain store data within the cache. As such, the memory traffic becomes read only beyond the blocking size. As another example of the workload variation we included *gcc*. In the *gcc* workload smaller caches produce a more read dominated pattern, while larger caches become write dominated. This behavior is due to the code generation aspect. As the cache grows, data tables within the compiler become cache-contained, leaving only the write-back traffic of the generated code.



(a) Milc (b) Calculix

(c) Gcc

Fig. 4. Histogram examples of SPEC CPU2006

B. Intra-Chip Partitioning Algorithm

Our *Intra-chip* cache capacity assignment policy is based on the concept of *Marginal-Utility* [5]. 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 for us). The amount of utility relative to the resource is defined as the *Marginal-Utility*. Specifically, the *Marginal-Utility* for n additional elements when c of them have already been used is defined as:

$$Marg.\ Utility(n) = \frac{Utility\_Rate(c+n) - Utility\_Rate(c)}{n} \quad (1)$$

The MSA histogram provides a direct way to compute the *Marginal-Utility* for a given workload across a range of possible cache allocations. 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 assigned capacity. Our greedy algorithm is shown in Algorithm 1. The general form of the algorithm appears in [27] and later on modified by Qureshi et al. in [22] for cache partitioning allocation.

Algorithm 1 Marginal-utility allocation algorithm

```
/* Initial */
num_ways_free = 128
best_reduction = 0
/* Repeat until all ways have been allocated to a core */
while (num_ways_free) {
   for core = 0 to num_of_cores {
        /* Repeat for the remaining un-allocated ways */
        for assign_num = 1 to num_ways_free {
          local_reduction = (MSA_hits(bank_assigned[core]
                +assign_num) - MSA_hits(bank_assigned[core]))
                /assign_num;
          /* keep the best reduction so far */
          if (local_reduction > best_reduction) {
               best_reduction = local_reduction;
               save(core, assign_num);
          }
        }
   }
   retrieve(best_core,best_assign_num);
   num_ways_free -= best_assign_num;
   bank_assigned[best_core]+= best_assign_num;
}
```

Using Algorithm 1 as our guide we have implemented a detailed **Intra-chip partitioning algorithm**. The algorithm takes into consideration the ideal capacity assigned by Algorithm 1 and the distributed nature of our DNUCA LLC design, to assign specific cache-ways to each core. The algorithm is based on a set of heuristics that are applied on the initial ideal assignments to decide on the partitions placement. The algorithm was proposed in [12] for a single CMP chip. The heuristics, that are analyzed in more details in [12], are the following:

1) *Center* cache banks are completely assigned to a specific core. This enables efficient aggregations of multiple cache banks.
2) If a core is assigned less than 8 ways (the size of a cache bank), all the ways are assigned to a *Local* bank close to it to keep a low access latency for the core.
3) *Local* cache banks can only be shared with an adjacent core in order to provide low latency and minimal network loads for data transfers.

To enforce the selected cache partitions, we modified the typical design of a cache bank to support a fine-grain, cache-way partitioning scheme as was proposed in [10]. According to this scheme, each way of a cache bank can only belong to one specific core. When a core suffers a cache miss, a modified LRU replacement mechanism is used to select the least recently used cache block that belongs to that specific core, for replacement. Our approach is compared against the scheme from Qureshi et al. in [22] enhanced for our CMP DNUCA LLC, which from now on we call UCP+ (Utility-based Cache Partitioning).

Each iteration of Algorithm 1 requires up to the number of available cache ways computations. As the cache ways are allocated, the number of ways a core can receive in each iteration reduces accordingly. Assuming the worst-case assignment of ways to be one cache way per iteration of the algorithm, its computational complexity is equal to $N + (N-1) + ... + 1 = N * (N-1)/2 = O(N^2/2)$, where N is the maximum allowable number of cache ways we can assign to a core. Moreover, since we assign cache ways in a granularity of 8-ways for most of the cases, for an equivalent of 128 ways LLC the $N \approx 128ways/8 \approx 16$. Therefore the complexity of estimating the cache partitions in our case is significantly less than the one needed by the Utility-Based algorithm of Qureshi et al. [22] for a large CMP system like our baseline. Finally, such complexity is less significant because of the low frequency of these computations. As our evaluation section shows, we use epochs of 100M cycles to re-evaluate the LLC partitions and the evaluation can be done off-line, minimizing the performance impact.

C. Inter-Chip Partitioning Algorithm

Our **Inter-chip partitioning algorithm** utilizes the non-uniform marginal utilities of *Intra-chip partitioning algorithm* and the bandwidth requirements of each core to find an efficient workload schedule on the various available chips within our system. Given a random workload assignment among many CMP chips in a system, some chips are expected to feature higher contention of cache capacity and memory bandwidth than others. To mitigate this problem, we propose a global partitioning algorithm that aims at lowering this system level contention. Our approach first uses a global *Marginal-Utility* assignment to guide migrations of work between the chips in the system in an effort to relieve cache capacity contention. This optimization step is combined with memory bandwidth over-commit detection, which can potentially create additional migrations. Each migration step is evaluated against a heuristic so that the migration overhead is bounded with respect to the expected execution speed gains. In the following we describe the two basic steps of the algorithm.

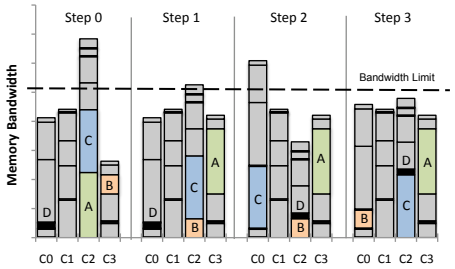
1) Cache Capacity: First, we use the *Marginal-Utility* allocation of Algorithm 1 to estimate an optimal resource assignment for each core (*ideal*), assuming that each core can freely use all of the available cache capacity in any chip. This gives us an optimal resource assignment per core and allows us to estimate the distance of the *Intra-chip partitioning algorithm* assignment from the ideal one per core. Having this information, we use Algorithm 2 to perform workload swaps between chips following a greedy approach. In line 1

of the algorithm, we estimate the distance, in number of cache ways, of each core's capacity assignment from its *ideal* one. Based on that we find the core with the worst cache-ways assignment in each chip. Lines 3 and 4 find the chip and core that has the biggest number of surplus ways in the system, respectively. The surplus ways are the ways assigned to a core that do not significantly contribute to the miss rate of the core's workload, based on our MSA profiler and their *Marginal-Utility* analysis, and therefore can be reassigned with small performance cost. Our greedy approach swaps (migrates) the workloads of the worst and best identified cores (line 5) and re-estimates the overall miss rate of the current assignment. The whole process is repeated until the threshold based on migration cost is reached (line 7).

Algorithm 2 Inter-Chip Capacity-based Workload Swapping

```
   do {
      for-each core in system-cores {
1        estimate core partition efficiency over ideal
2        find worst-core ∈ system-core with largest deficiency
      }
      for-each chip in system-chips {
3        find best-chip ∈ system-chips with largest surplus of ways
4        find best-core in best-chip with lowest capacity requirement
      }
5     swap workloads of best-core with worst-core
6     estimate overall_miss_rate
7  } while (overall_miss_rate − previous_miss_rate < threshold)
```

2) Memory Bandwidth: Using the proposed MSA-based profiler we can predict the bandwidth requirements of a given cache capacity assignment of each workload. An efficient assignment of cache partitions by the *Intra-chip partitioning* algorithm may end up having a very high memory bandwidth requirement per chip. Such an assignment can therefore saturate the available bandwidth. Note that the latency due to bandwidth over-commit is non linear, typically following an exponential relation with the network utilization. Our migration based **Memory Bandwidth Over-commit** algorithm attempts to find combinations of workloads with high/low bandwidth requirements. The workload with higher bandwidth demands are shifted from over-committed chips to under-committed chips. The swapping workloads must have similar cache capacity assignments, within a small percentage difference (10% in our case), in order to allow such swapping. This is necessary to guarantee that the migrations caused by the *memory bandwidth over-commit* algorithm do not contradict the assignments of the *Intra-chip cache partitioning* algorithm. The process is repeated up to the point there is either no bandwidth over-commited chips or additional swapping does not offer any reduction in bandwidth usage. Fig. 5(a) shows an example of how *Memory Bandwidth Over-commit* algorithm works on a four-chips system case with each chip having 8 randomly selected workloads executing on it from SPEC CPU2006 suite [26]. The Y-axis shows the stacked memory bandwidth requirements per chip as it was projected by our MSA-based profiler. The figure includes four steps of the algorithm (steps *Step 0* to *Step 3*). In the initial step *Step 0*, *Chip 2* (C2) is assigned a set of workloads which memory



(a) Memory Bandwidth Over-commit algorithm illustration

Step #	Workload($Chip_{Initial} \rightarrow Chip_{Final}$)
Step 0	Initial step, no swaps
Step 1	A($C2 \rightarrow C3$), B($C3 \rightarrow C2$)
Step 2	C($C2 \rightarrow C0$), D($C0 \rightarrow C2$)
Step 3	C($C0 \rightarrow C2$), B($C2 \rightarrow C0$)

(b) Workload swaps per step

Fig. 5. Four steps example of *Memory Bandwidth Over-commit* algorithm for four chips (C0 to C3)

bandwidth requirements exceed the available one. If this set of workloads is executed on a single chip, the memory bandwidth contention will slow down the chip, affecting the overall throughput. The selected workloads are: lbm (A), $calculix$ (B), $bwaves$ (C) and $zeusmp$ (D). Note that $zeusmp$ (D) has very small memory bandwidth requirements. Table 5(b) shows the migrations that the algorithm performs in order to find a solution that satisfies the maximum bandwidth constraint. As we can see from *Step 3*, the algorithm terminates with an assignment that meets the bandwidth restriction for every chip.

3) Computational Overhead of Inter-chip algorithm: The *Inter-chip* algorithm is based on both Algorithm 1 and Algorithm 2 with the first being more computational demanding. Therefore, the computational complexity is bounded by the one of the *Marginal-Utility* algorithm which according to Section III-B is equal to $O(N^2/2)$. In addition, Algorithm 2 investigates migration only among the chips that are over-utilizing the memory bandwidth and the number of steps is limited by a threshold. As in the case of the *Intra-chip* algorithm, such computational complexity is less significant because of the low frequency of computation. The evaluation take place infrequently and can be done off-line, minimizing the performance impact.

D. Overall Dynamic Scheme

Fig. 6 shows an outline of our framework for the proposed hierarchical bandwidth-aware resource management scheme. The dark shaded modules indicate our additions over a typical large CMP system as the one described in Section II. Looking at the single-chip level, each core has a dedicated *Cache Profiling* circuit that tracks its shared resource (cache capacity and memory bandwidth) requirements (description in Section III-A). This profiling circuit is independent of the cache subsystem and is able to non-invasively monitor the behavior of an application running on a core. Each *Cache Profiling*

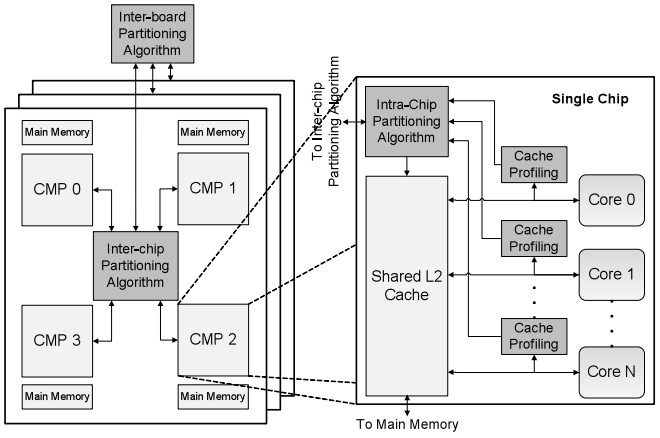


Fig. 6. Proposed Bandwidth-aware Resource Management framework

circuit is unaware of the workloads executing on other cores and assumes that the whole cache is available to the monitoring core. The overall proposed dynamic scheme is based on the notion of epochs. During an epoch, our MSA-based profiling circuit constantly monitors the behavior of each core in every CMP of the overall system. When an epoch ends, the profiled data of each core are passed to the *marginal-utility* algorithm to find the ideal cache capacity assignments for each core in every individual CMP-chip. Those partitions are then given to the *Intra-chip cache partitioning* algorithm for assigning specific cache-ways to each core using the heuristics described before. When each core has a specific cache partition assigned to it, we look at a higher than a single chip level for further optimizations in both cache capacity and memory bandwidth usage. To accomplish that we use the *cache capacity* algorithm of the *Inter-chip partitioning scheme* to find better cache allocations among multiple chips. Following that, we use the *Memory bandwidth* part of the same scheme to identify and solve bandwidth over-committed chip. In the end, we perform the necessary workload migrations and update the assigned cache partitions to each chip. The whole process is repeated at the end of the next epoch.

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 [25] as the full system functional simulator extended with Gems toolset [18] to simulate a cycle-accurate 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, inter-core network using a MOESI cache coherence protocol. Throughout the paper, the frequency of evaluating and reallocating the L2 cache partitions on a simple chip was set to a 100M cycle epoch.

We use SPEC CPU2006 [26] scientific benchmark suite, compiled to SPARC ISA with peak compilation options, to evaluate 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 for a slice of 200M instructions after cache warm up. Table I includes the basic system parameters that were used.

In the remaining of the evaluation section we first demonstrate the accuracy and overhead of our MSA profilers followed by the evaluation and comparison of our proposed scheme over one of the most promising cache partitioning algorithms in the literature, the *Utility-based Cache Partitioning* [22], extended to match our baseline system (*UCP+*).

A. MSA-profiler Implementation Overhead and Accuracy

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* and *dirty evictions* counter pairs of Fig. 2. Additional overhead is introduced by the implementation of the *Hit Counters*, *Dirty Counters* and *Dirty Stack Distance* registers 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.

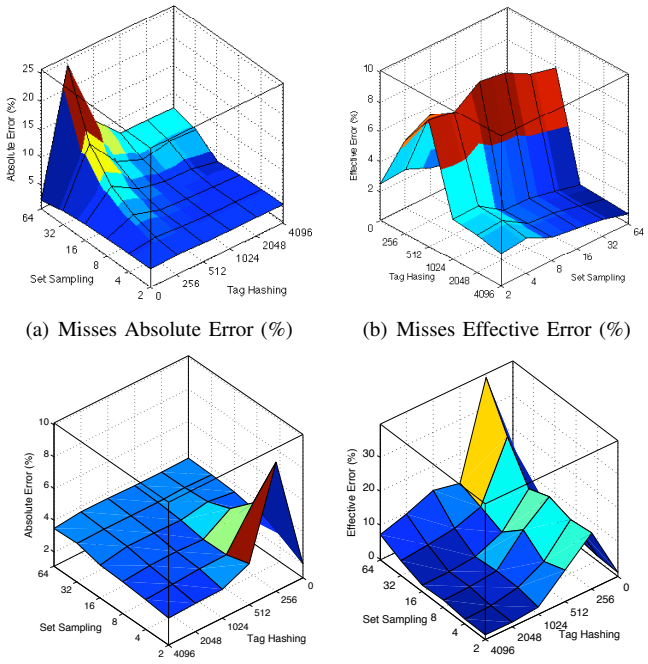
A naive implementation would require a complete copy of the cache block tags for each cache set in each one of MSA profilers, which is prohibitively high. The overhead can be greatly reduced using: *a*) *partial hashed tags* [13], *b*) *set sampling* [14], and *c*) *maximum assignable capacity* reduction techniques. With *partial hashed tags* one can use less than full tags to identify the cache blocks assigned at each counter pair thus reducing the storage overhead. Hashing is necessary to reducing the aliasing problem of using less than full tags. *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 counter pairs are reduced to the maximum number of assignable 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.

Fig. 7 demonstrates the errors introduced by *partial hashed tags* and *set sampling* techniques in our MSA profilers, for both misses and bandwidth, compared to their unrealistic full implementations. Our analysis target is to find a configuration that can significantly reduce the necessary overhead to a realistic implementation with an acceptable error rate. The errors estimated as an average error over the analysis of the whole SPEC CPU2006 suite for a slice of 100M instructions per benchmark using our detailed Gems implementation of our scheme. The full implementation profilers are able to accurately monitor the requirements and do not introduce any

errors since they keep all the necessary information from our simulator. We provide two space exploration error analysis, *Absolute error* and *Effective error*. The first one represents the aliasing error of the actual raw data in the counters for all configurations. On the other hand, the *Effective error* is estimated over the information that our algorithms use to estimate the Marginal-Utility of each cache way and reflects the error that can lead to a different assignment decision for *Intra-chip* algorithm and workload migration for *Inter-chip* algorithm. *Set sampling* was selected to change from 1-in-2 up to 1-in-64 samples per cache set and *partial hashed tags* changed from 0 (no partial tags) to 4096 (using only 12 bits of address tags). To mitigate the aliasing problem, the *partial hashed tags* use a randomly created network of XOR gates to hash the partial tags. The XOR tree overhead is very small in comparison to the necessary number of counters and only one copy per MSA-profiler is necessary. There are two rules for choosing a configuration and minimize the hardware overhead: The hardware overhead is a) proportional to the size of *partial hashed tags* , and b) inversely proportional to the *set sampling*. Therefore, we ideally want to keep a small number of tag bits and use a large number of *set sampling*.

From Fig. 7 we can clearly see that as the *set sampling* number increases so does the error rate. In addition, an increased number of *partial hashed tags* can significantly improve the error rates especially for the case of *Effective error*. For most cases a small number of *partial hashed tags* introduce a significant error which indicated that we need to choose a big number of bits in our tags. Fig. 7(a) and Fig. 7(c) show that our estimation of misses is more sensitive to the selected configuration that the memory bandwidth use, introducing a variation in the absolute error. On the other hand, Fig. 7(b) and Fig. 7(d) demonstrate the opposite trend for the effective error. This is a strong indication that our algorithm's decisions are more sensitive to the bandwidth use and we should choose a configuration that favors those decisions more. Furthermore, for high number of partial tags and set sampling, the effective errors are significantly smaller than the absolute errors which allow us to be more elastic on the final selected configuration, improving our overhead.

Taking all the previous trends into consideration we chose to use 11 bit *partial hashed tags* (Tag hashing 2048) combined with 1-in-32 *set sampling*. Such a configuration produced average *Absolute error* rates of 6.4% for misses and 5% for bandwidth. On the other hand, the *Effective error* was estimated to be 1.3% for misses and 3.6% for bandwidth compared to the profiling accuracy obtained using a full tag implementation. Such error bounds are inline with other set-sampling based monitor schemes like UMON [22] and CacheScouts [30]. The first one concludes that 1-in-32 *set-sampling* is enough for their profiler and the latter reports error rates of 6% for their 1-in-128 set sampling of cache occupancy for scientific workloads. To further improve overhead, our *Intra-chip partitioning* assignment algorithm limits each core to a maximum of 9/16 of the total cache capacity and therefore the *lru_pointer* and the *Dirty Stack Distance* register sizes



(a) Misses Absolute Error (%) (b) Misses Effective Error (%)

(c) Bandwidth Absolute Error (%) (d) Bandwidth Effective Error (%)

Fig. 7. MSA absolute and effective accuracy for different implementations

were set to 6 bits. The Hit and $Dirty$ counters size was set to 32 bits to avoid overflows during an epoch. Finally, we have implemented the LRU stack distance of the MSA as a single linked list with head and tail pointers. The cost for such a structure is included in Table II. Overall, the implementation overhead is estimated to be 117 kbits per profiler, which is approximately 1.4% of our 8MB LLC cache design assuming 8 profilers.

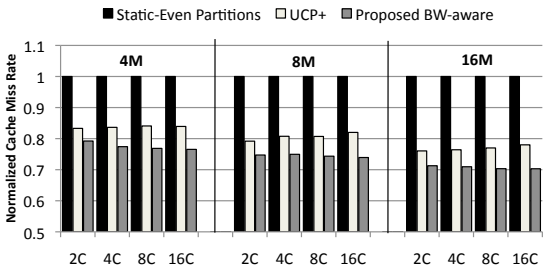
TABLE II. Overhead of the proposed MSA profiler

Structure Name	Overhead Equation	Overhead
Tags Size	$Tag\_width * Cache\_ways * (Sets/Set\_sampling)$	54 kbits
LRU Stack Distance	$((LRU\_pointer * Cache\_ways) + Head/Tail) * (Sets/Set\_sampling)$	27 kbits
Dirty Bits	$Cache\_ways * (Sets/Set\_sampling)$	2.25 kbits
Dirty Counters	$Cache\_ways * Counter\_size$	4.5 kbits
Dirty Stack Distance Registers	$Cache\_ways * Register\_size$	27 kbits
Hit Counters	$Cache\_ways * Counter\_size$	2.25 kbits

B. Bandwidth-aware Resource Management Scheme

Analysis of computer systems in virtualization environments is an important 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 CPU2006 integer and floating point benchmark suites. Even with

this limitation the possible combinations of the 29 workloads on our eight core target machine is very large and equal to $C(num\_workloads + num\_cores$ - $1, num\_cores) = C(29 + 8\text{-}1, 8)$ ≈30 million possible workload combinations. Based on this very large state space, we employ a Monte Carlo based approach for the evaluation of our proposed scheme. The cache miss rates for the overall system are estimated from projecting miss rates based on MSA data collected from our detailed system simulations. We used this method, to evaluate our proposed scheme over a range of cache sizes and number of chips in the system. The evaluation method was executed for 1000 random workload assignments for each configuration and includes the following steps: a) Collect MSA histograms from detailed simulations for all workloads, b) Randomly select (with repetition) a workload for each core in the simulated system and c) Execute stand-alone *Intra-chip* algorithm (*UCP+*) and the proposed *Bandwidth-aware* algorithm (*BW-aware*), that is *UCP+* combined with *Inter-chip* algorithms together. In the following we first report the gains in LLC misses followed by the gains in bandwidth use.



(a) Relative miss rate for different number of chips (2 to 16 chips) and sizes of LLC (4 to 16MB)

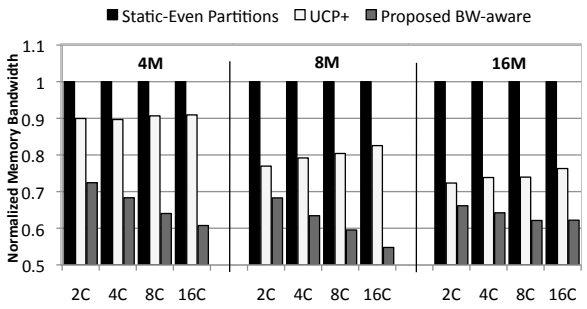
	Cache Size per Chip		
# of Chips	**4MB**	**8MB**	**16MB**
2 Chips	4.9%	5.6%	6.2%
4 Chips	7.1%	7.4%	7.5%
8 Chips	8.5%	7.8%	8.6%
16 Chips	8.7%	9.8%	9.9%

(b) Avg. relative miss rate reduction of proposed BW-aware over UCP+

Fig. 8. Relative miss rate

1) LLC misses: Fig. 8 shows the normalized cache miss rate results from 2 to 16 CMP chips and LLC cache sizes of 4, 8 and 16MB. The first bar of Fig. 8(a) (*Static-Even Partitions*) represents the miss rate for the static even partitions where each core is statically allocated 1/8 of the cache capacity. The next two bars show the relative miss rate of the *UCP+* and *BW-aware* partitioning schemes, respectively. As we stated before, the *UCP+* is our implementation of the *Utility-based Cache Partitioning* algorithm proposed by Qureshi et al. [22] extended with a set of heuristics to support an 8-core CMP-DNUCA system like our baseline one. According to the figure, the average reduction provided by *BW-aware* is 25.7% over the simple static-even partitions scheme. This is a significant reduction considering the relatively small 1.4% storage overhead of our MSA-based profiling structures. In addition, Fig. 8(b) shows the additional miss rate reductions provided by *BW-aware* algorithm over the *UCP+*'s scheme. From the figure, we can see an average LLC misses reduction of 7.9% taking into consideration all the configurations. Therefore, the *BW-aware* scheme provides significant additional miss rate reductions with no significant additional hardware over *UCP+*. As the number of chips in the system increases, the *BW-aware* algorithm shows improvement in the miss rate up to 8 chips after which we observe diminishing returns. This indicates that large SMP systems could contain the *BW-aware* algorithm within its shared memory space and therefore avoid more complex partition migrations across multiple servers. Finally, the miss rate improvements are consistent across a reasonable range of cache sizes, demonstrating a small improvement as the LLC size increases. The increased size allows more surplus of cache ways (see Algorithm 2) and therefore allows the *Inter-chip* algorithm to find more candidates for swapping that can potentially lead to a better mapping of the workloads to the available chips.

2) Memory Bandwidth: To show the memory bandwidth improvements we focus on the reduction of the average worst-case chip memory bandwidth in the system during an epoch. As our algorithm migrates bandwidth from the highly to lightly loaded chips in the system, an average reduction of bandwidth across all the chips is not relevant in determining our performance. Instead we care for the cases that chips feature long memory latencies due to overcommitted memory bandwidth requirements. Fig. 9(a) includes the bandwidth reductions achieved by using *UCP+* and our proposed scheme over the static-even partitions case. The *UCP+* algorithm reductions are only due to the reductions in the miss rate achieved by the Marginal-Utility-based algorithm using our MSA-profilers information and on average is equal to 19% over the simple static case with even partitions. The addition of the *Inter-chip* algorithm in our proposed *BW-aware* scheme increases this reduction to an overall 36%. The additional gains that our migration-based scheme can provide over the *UCP+* are shown in more details in the table of Fig. 9(b). Overall, our scheme is able to provide an additional 18% reduction in bandwidth over *UCP+*. As expected, the bandwidth reduction is greater in the smaller cache configurations. Essentially, the higher miss rates due to smaller caches results in greater contention which provides greater opportunities for improvement. In addition, as the number of chips in the system increases, we are able to find more opportunities for migrations as such the memory bandwidth reductions increase significantly from 2 to 16 chips. The system size required for good optimizations is comparable to that required by the cache capacity optimizations, that is 8 to 16 chips. One artifact of using average worst-case bandwidth is the upward trend of the *UCP+*. As the number of chips in the simulation increases, the expected worst-case bandwidth increases due to the random nature of the workload selection.

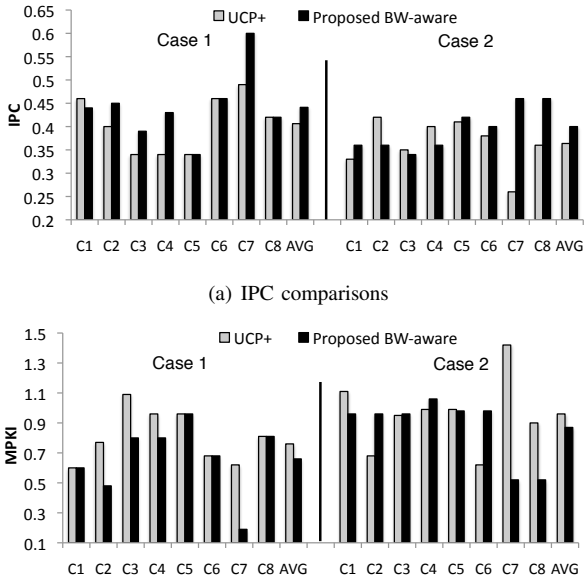


(a) Relative memory bandwidth for different chip configurations

	Cache Size per Chip		
# of Chips	**4MB**	**8MB**	**16MB**
2 Chips	17.5%	8.6%	6.1%
4 Chips	21.3%	15.5%	19.6%
8 Chips	26.6%	20.8%	11.8%
16 Chips	30.1%	27.8%	14.1%

(b) Avg. bandwidth use reduction of BW-aware over UCP+

Fig. 9. Relative bandwidth use



(a) IPC comparisons

(b) MPKI comparisons

Fig. 10. Evaluation of two case studies

C. Detailed Simulation Case Study

We chose a set of experiments of the previous simulations to validate the effectiveness on a full system simulation based on Simics [25] and Gems [18]. We simulated an 8-chip system with each chip being an 8-core CMP. We selected the experiments randomly to validate our proposed scheme and see its effectiveness on overall performance. We chose two sets of 64 random assignments and run two experiments on these sets. The first experiment uses only the *UCP+* approach which is equivalent to our *Intra-chip partitioning* algorithm. For the second experiment we apply the multi-chip *Inter-chip partitioning* algorithm (*BW-aware*) on each set in addition to *UCP+*.

Figs. 10(a) and 10(b) show the IPC and MPKI (misses per kilo instructions) comparison of *UCP+* and *BW-aware* for each chip for the first and second experiment sets, respectively. The first set demonstrated an average improvement of 8.6% in IPC and 15.3% in MPKI across all chips, mainly due to bandwidth improvements on Chip 4 and Chip 7. The main difference between the assignments of Chip 4 and Chip 7 for the cases of *UCP+* and *BW-aware* was moving *bwaves* and *mcf* and replacing them with *povray* and *calculix*. Both *bwaves* and *mcf* are, according to our MSA-based profiling, two of the most demanding memory bandwidth benchmarks in the whole suite and compared to them *povray* and *calculix* have very low bandwidth requirements.

The second set of experiments showed an average improvement of 8.5% in IPC and 11% in MPKI. The main contributor of this reduction are the workload migrations on Chip 7. According to our profiling, Chip 7 was initially over-utilizing the memory bandwidth creating a significant number of misses. By applying the *Inter-chip* algorithm, *bwaves* and *gcc* workloads where swapped with *zeusmp* from Chip 2 and *gamess* from Chip 6. Both *zeusmp* and *gamess* benchmarks need a small percentage of memory bandwidth while *bwaves* and *gcc* have both high memory bandwidth demands. Therefore, such migrations enabled both the high-demanded benchmarks to use the previously available bandwidth in Chip 2 and 6, and the previously saturated Chip 7, to allocate the capacity and bandwidth to the rest of its workloads. Both of these experiments validate the effectiveness of our algorithm in improving both the cache miss rate and memory bandwidth requirements in over-commited systems, and show IPC and MPKI improvements close to the one estimated in the previous section using the Monte-carlo approach.

V. Related Work

Kim et al. [15] highlighted the importance of enforcing fairness in CMP caching and proposed a set of fairness metrics for optimization. However, their policies are analyzed using only best static offline parameters and mechanisms. Suh et al. [28] propose a dynamic L2 cache partitioning technique that allocates the L2 cache resource at the granularity of cache ways to improve system throughput. They employ a greedy heuristic to allocate cache ways to the application that has the highest incremental benefit from the assignment. Qureshi et al. [22] improves Suh's allocation policy by using cache utility monitor (UMON) to estimate the utility of assigning additional cache ways to an application. Both of them target at improving the throughput of a single CMP system through reductions in cache misses. Chang et al. [4] proposed time-sharing cache partitions, which translates the problem of fairness to the problem of scheduling in a time-sharing system. Jiang et al. [11] demonstrate that co-scheduling on k-core ($k > 2$) is NP-hard, and propose a greedy algorithm that schedules jobs in the order of their sensitivity to cache contentions. Our approach extends these papers by proposing a systematic

approach that considers main memory traffic in addition to cache miss rate. In addition we seek system-level solutions beyond a single chip in the context of large CMP systems. Iyer et al. [10] proposed a framework for enforcing QoS characteristics in a system based on trial-and-error approaches to fit the QoS targets. That work was later on extended by Zhao et al. [30] with a set of counters, named CacheScouts, that monitored their QoS characteristics in a system and made resource management decisions accordingly. We use non-invasive MSA profilers that can predict resource requirements for all different cache capacity assignments in parallel for both misses and bandwidth use. Such prediction is important for reconfigurable schemes in order to perform expensive reconfiguration operations only when the benefit is predicted to be important enough.

VI. 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. Our analysis shows that bandwidth-aware optimizations at the system level provide significant improvements over optimizations limited to a single chip with very small additional hardware overhead. Our proposed *Bandwidth-aware* partitioning scheme is able to provide on average, 25.7% reduction in misses and 36% reduction in worst-case bandwidth use compared to static-even LLC partitioning schemes. A comparison with a state-of-art cache partitioning scheme showed an average 18% reduction in memory bandwidth along with 7.9% reduction in the LLC miss rate. In addition, our detailed simulation case studies show that our resource management scheme can achieve an average reduction of 8.5% in IPC and 13% in MPKI on a 8-chip CMP system. These improvements over existing techniques can justify the hardware overhead of our proposed MSA-based profilers which was estimated to be 1.4% of our baseline 8MB LLC.

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, IBM, Semiconductor Research Consortium and Lockheed Martin. 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, IBM, Semiconductor Research Consortium or Lockheed Martin.

References

[1] B. M. Beckmann and David A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches", Proceedings of the 37th International Symposium on Microarchitecture, 2004

[2] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach", In International Symposium on Microarchitecture 2008, Pages 318–329, 2008

[3] D. Chandra, et al., "Predicting inter-thread cache contention on a chip multiprocessor architecture", In Proc. 11th International Symposium on High Performance Computer Architecture, 2005

[4] J. Chang, et al., "Cooperative cache partitioning for chip multiprocessors", In ICS '07: Proceedings of the 21st annual international conference on Supercomputing, 2007

[5] Wieser, Friedrich von, "Der naturliche Werth [Natural Value]", Book I, Chapter V, 1889

[6] F. Guo, et al., "From Chaos to QoS: Case Studies in CMP Resource Management", dasCMP/Micro, Dec 2006

[7] R. Ho, K. W. Mai, and M. A. Horowitz. "The Future of Wires", Proceedings of the IEEE, 89(4):490-504, Apr. 2001

[8] 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

[9] 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

[10] R. Iyer, et al. "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms", 18th Annual International Conference on Supercomputing (ICS'04), July 2004

[11] Y. Jiang, et al. , "Analysis and approximation of optimal co-scheduling on chip multiprocessor", International Conference on Parallel Architectures and Compilation Techniques (PACT) 2008, pages 220-229, 2008

[12] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware Dynamic Cache Partitioning for Multicore Architectures," The 38th International Conference on Parallel Processing (ICPP), 2009

[13] R. Kessler, R. Jooss, A. Lebeck and M. D. Hill ,"Inexpensive implementations of set-associativity", In Proceedings of the 16th Annual International Symposium on Computer Architecture, 1989

[14] 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.

[15] S. Kim, et al., "Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", 13th Int PACT, 2004

[16] K. Krewell, "Best Servers of 2004: Multicore is Norm", Microprocessor Report, www.mpronline.com, Jan 2005

[17] R. L. Mattson. et al. "Evaluation techniques for storage hierarchies". IBM Systems Journal, 9(2):78-117, 1970

[18] Milo M. K. Martin, et al. "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", Computer Architecture News (CAN), September 2005

[19] 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

[20] 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

[21] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems", In MICRO 39, 2006

[22] M. K. Qureshi and Yale N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", MICRO 2006

[23] 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

[24] 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, 1999

[25] Simics Microarchitect's Toolset, http://www.virtutech.com/

[26] SPEC CPU2006 Benchmark Suit, http://www.spec.org

[27] H. S. Stone et al. "Optimal partitioning of cache memory", IEEE Transactions on Computers, 41(9), 1992.

[28] G. E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-aware Scheduling and Partitioning", In Proceedings of the 8th International Symposium on High-Performance Computer Architecture, 2002

[29] P. Zhou, et al. "Dynamic Tracking of Page Miss Ratio Curve for Memory Management", Architectural Support for Programming Languages and Operating Systems, 2004

[30] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *PACT '07 pages 339–352, 2007.*