Cache Friendliness-aware Management of Shared Last-level Caches for High Performance Multi-core Systems

Dimitris Kaseridis, *Member, IEEE,* Muhammad Faisal Iqbal, *Student Member, IEEE* and Lizy Kurian John, *Fellow, IEEE*

Abstract—To achieve high efficiency and prevent destructive interference among multiple divergent workloads, the last-level cache of Chip Multiprocessors has to be carefully managed. Previously proposed cache management schemes suffer from inefficient cache capacity utilization, by either focusing on improving the absolute number of cache misses or by allocating cache capacity without taking into consideration the applications' memory sharing characteristics. Reduction of the overall number of misses does not always correlate with higher performance as Memory-level Parallelism can hide the latency penalty of a significant number of misses in out-of-order execution. In this work we describe a quasi-partitioning scheme for last-level caches that combines the memory-level parallelism, cache friendliness and interference sensitivity of competing applications, to efficiently manage the shared cache capacity. The proposed scheme improves both system throughput and execution fairness – outperforming previous schemes that are oblivious to applications' memory behavior.

Index Terms—Cache resource management, Last-level caches, Memory-level Parallelism, Chip Multiprocessors.

1 Introduction

As the number of integrated cores in Chip-Multiprocessor (CMP) designs continues to increase, the typically shared last-level cache memory gradually becomes a critical performance bottleneck. In the past, researchers have shown that traditional, unmanaged, shared cache schemes introduce inter-thread destructive interference, leading to performance degradation and lack of ability to enforce fairness and/or Quality of Service (QoS).

To address the problem of contention in shared last-level caches, three main research directions have been proposed in the past: a) *Cache Capacity Partitioning* schemes [4][13][31][8][19], b) *Cache-blocks Dead-time* management schemes [20][8], and c) *Cache Pseudo-partitioning* schemes [31]. The *Cache Partitioning* schemes identify an ideal size of capacity that each concurrently executing thread should be assigned to maximize system throughput and/or execution fairness. Although such schemes can effectively eliminate threads' destructive interference by utilizing isolated partitions, they result in inefficient capacity utilization [8][31]. A big portion of the last-level cache capacity can be underutilized for a significant amount of time. *Cache-blocks Dead-time* management schemes focus on identifying the dead (not any more useful) cache lines and force their early eviction from the cache. As a result, a big percentage of useful lines are retained in the cache, enabling better utilization of its capacity. Unfortunately, since there is no control over the number of cache lines each thread can maintain in the cache, destructive interference among the concurrently executing threads is still present. To counter this problem, *Cache Pseudo-partitioning* schemes [31] have been proposed that provide a combination of both previous schemes; applications are allowed to share part of the available capacity and compete for it, while at the same time, they can exclusively occupy a portion of capacity to protect useful lines from being evicted by other applications in the cache.

An effective cache management scheme should provide both good *capacity utilization* and *interference isolation*. While both *Dead-time* and *Pseudo-partitioning* schemes are able to identify and prevent "thrashing"[1] applications from polluting the cache, they are oblivious to the other two important memory behaviors that we identify in this paper, namely, *Cache Friendly* and *Cache Fitting* behaviors. As our evaluation shows, a scheme that is aware of application's Cache Friendliness (meaning its memory behavior in this work) can provide better capacity utilization while significantly reducing cache interference.

Previously proposed Dead-time and pseudo-partitioning cache management schemes allocate capacity to applications based on heuristics that aim

- *Dimitris Kaseridis is with ARM Holdings, USA. This work was done while he was with The University of Texas at Austin, Austin, TX, 78712, USA. E-mail: kaseridis@mail.utexas.edu.*
- *Muhammad Faisal Iqbal and Lizy Kurian John are with The University of Texas at Austin, Austin, TX, 78712, USA. Emails: faisaliqbal@utexas.edu, ljohn@ece.utexas.edu*

1. Applications with small temporal locality and high cache space demand rate that tend to occupy a big portion of the cache, evicting useful cache lines that belong to other applications.

Digital Object Indentifier 10.1109/TC.2013.18

at the raw reduction of the absolute number of cache misses. In out-of-order execution, each workloads can have a different number of overlapping cache misses (Memory-level Parallelism - MLP) which could drastically change workloads' performance sensitivity to the absolute number of misses. As it has been shown for isolated cache partitioning schemes [18], targeting the reduction of absolute number of misses introduces significant inefficiencies. A significant percentage of cache space can potentially be allocated to applications with high cache demand rate that cannot actually extract any benefit from the dedicated capacity; while low MLP, miss-latency sensitive workloads with small cache space demand rates can suffocate in small cache partitions. As the number of cores per die increases, such wasteful cache management schemes will severely affect performance and scaling efficiency of future high performance, multicore designs.

In this work we describe a *Memory-level parallelism and Cache-Friendliness aware Quasi-partitioning scheme* (MCFQ). Our *Quasi-partitioning* proposal can successfully combine the benefits of previous schemes as it mimics the operation of cache capacity partitioning schemes without actually enforcing the use of isolated partitions; while at the same time, maintaining all the benefits of pseudo-partitioned schemes. Our scheme manages the cache quasi-partitions by taking applications' MLP and memory use behavior into consideration. To minimize the effects of interference among resource competing applications, we utilize a set of heuristics to assign priorities over the use of the available shared cache capacity.

Overall this work makes the following contributions in the area of last-level caches cache management:

- We advocate the use of the Memory-level Parallelism (MLP) information to predict applications' performance sensitivity to last-level cache misses. The MLP information allow us to tune our heuristics by using the final, overall system throughput instead of focusing on raw reductions of cache misses.
- Propose a cache management scheme that allocates cache quasi-partitions based on applications' "Cache-friendliness", meaning their memory behavior when they coexist with other applications in last-level cache. *Friendly, Fitting and Thrashing* applications are treated with the appropriate priorities in our scheme to achieve the best possible use of the last-level cache capacity.
- Propose the use of two new heuristics: "Interference Sensitivity" factor and "Partitions Scaling" scheme to drive our cache management scheme. The first helps us identify how sensitive is an application to cache contention and how much we expect it to hurt the behavior of other co-executing applications with respect to cache use. The second helps us to readjust the quasi-partition sizes in order to bring the average, dynamically achieved cache space occupancy of each application closer to the best estimated one.

2 Motivation

2.1 Memory-level Parallelism (MLP)

In out-of-order execution, Memory-level Parallelism (MLP) of applications can drastically change workload's performance sensitivity to the absolute number of last-level cache misses. An application running on an out-of-order core with a high level of MLP usually clusters cache miss events together, overlapping their miss latency. As a result, the effective performance penalty of each last-level cache miss can vary according to the application's *concurrency factor* (that is MLP) and in general is expected to be smaller than the main memory access latency of a typical miss.

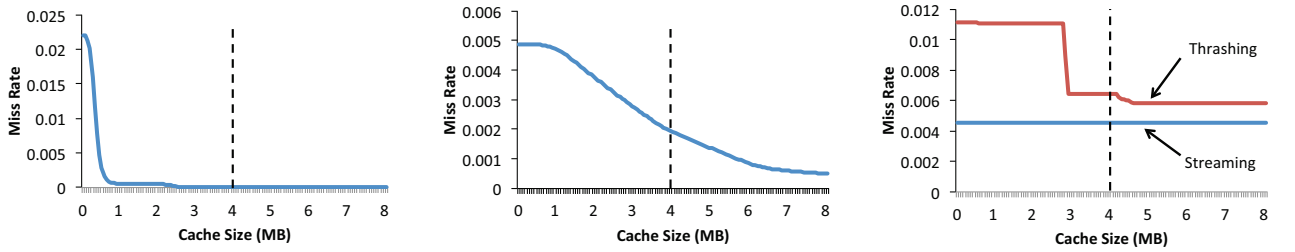
The two extreme cases of MLP are *pointer chasing* code and *vector* code. In a typical pointer chasing code, most of the load instructions are dependent on previous loads and therefore, on average, only one memory access can take place at the same time outside the core. As a result, pointer chasing code has a *concurrency factor* of 1 and reducing the number of last-level cache misses can directly be translated into performance gains.

On the other hand, we typically characterize a piece of code as vector code when a large percentage of the memory accesses are independent of each other and can be executed in parallel and out of order. Vector code has a large concurrency factor that is determined by the maximum number of outstanding memory accesses (or core misses) that the microarchitecture can support. In vector code, many independent load misses can be overlapped with each other, forming clusters of misses. A raw reduction of these overlapping misses may not lead to noticeable performance gains. Each miss cluster introduces a constant latency to reach main-memory and one have to avoid the whole cluster to improve performance. As a result, any cache managing scheme that simply aims at the reduction of the absolute number of cache misses, introduces inefficiencies for overall throughput, and cannot provide any QoS guarantees.

2.2 Cache Friendliness Aware Cache Allocation

Prior work has demonstrated that the capacity of the last-level cache has to be carefully managed in order to prevent destructive interference among multiple divergent workloads [4][31][8][19]. When multiple threads compete for a shared cache, cache capacity should be allocated to the threads that can more efficiently use such resources [6][24][19]. The commonly used LRU replacement policy is unable to identify which applications can effectively use the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.



(a) Cache Fitting Application (dealII) (b) Cache Friendly Application (gcc) (c) Cache Thrashing / Streaming (lbm, milc)

Fig. 1. Categories of applications' miss-rate sensitivity to cache space dedicated to them.

extra capacity and which ones act as cache hogs, polluting the cache. *Cache capacity partitioning* schemes address this problem by allocating specific, isolated, cache partitions to each application, eliminating the potential destructive interference.

Despite the popularity of cache capacity partitioning schemes, previous work [31][8] has shown that they tend to be wasteful as a percentage of the dedicated cache capacity is usually underutilized. To solve this inefficiency, *Cache pseudo-partitioning* schemes [31] break the cache partitions' isolation assumption and allow applications to share cache capacity and therefore compete with other applications for it. Such approaches provide to the cache management scheme the flexibility to "steal" cache capacity from cache-underutilizing applications and temporarily allocate it to the applications that can benefit the most from the extra capacity. As expected, such flexibility is not coming for free. By breaking the partitions' isolation assumption, destructive interference between the cache capacity competing cores is introduced. It is therefore necessary to allocate resources based on both the performance sensitivity of each application to cache space and how well applications compete with each other in order to claim cache capacity.

Fig. 1 illustrates the miss rates of different classes of applications on a 4MB last-level cache, as we increase the cache capacity allocated to them. Based on our analysis on a typical CMP system, we can classify applications into three basic categories:

1) **Cache Fitting Applications:** These applications have a small working set size that can easily fit in a typically sized cache, but at the same time, they are also very sensitive to this allocated cache capacity. As Fig. 1(a) shows, an allocation which is slightly smaller than their ideal size (close to 600KB for dealII) can significantly increase their miss rate, forcing them to behave as cache thrashing applications. It is therefore important for a cache management scheme to allocate enough space to these applications to fit their working sets. If such cache space allocation cannot be guaranteed then it is preferable to reduce the cache resources allocated to them in order to restrict their thrashing behavior from affecting the rest applications.
2) **Cache Friendly Applications:** These applications can efficiently share the cache capacity using an LRU-like replacement policy and can in general benefitted by additional cache space (Fig. 1(b)). In contrast to cache fitting applications, their cache space requirements usually exceed a typically sized cache, especially when they have to share it with other applications. Under a *Pseudo-partitioning* scheme, these applications should have higher priority than fitting applications as they have the potential to benefit from any additional capacity allocated to them through capacity "stealing".
3) **Cache Thrashing/Streaming Applications:** These applications feature poor cache locality due to either their big working set size that cannot fit in a typically sized shared cache, or their streaming behavior that produces a lot of memory traffic with small to almost no reuse. Under a LRU replacement policy, they pollute the shared cache without actually getting any benefit from the occupied cache capacity. When they execute with cache friendly or cache fitting applications, they should be allocated the minimum possible cache space to avoid cache pollution. We refer to both types as *Cache Thrashing* as both generate a thrashing behavior on the cache.

Overall, we can extract three basic rules that a cache capacity management scheme has to follow: a) the cache resources have to be allocated proportionally to the applications' ability to benefit from cache space, b) *Thrashing* applications have to be identified and restricted to a small fraction of the cache to significantly reduce destructive interference, and c) *Fitting* applications are sensitive to the cache space allocated to them and the management scheme has to provide some space allocation guarantees to avoid overall system degradation.

Using the above observations, we propose a quasi-partitioning scheme that allocates cache capacity based on the *Cache Friendliness* of each application. For the case of *Thrashing* applications, *Bimodal Insertion Policy (BIP)* [20] has shown that by inserting the new coming cache blocks of such applications in the LRU position, cache thrashing can be significantly reduced. Our scheme allocates the lowest priority to *Cache*

Thrashing applications, restricting them to the lower 1-2 cache ways of the LRU stack. We treat *Cache Fitting* applications with an intermediate priority level and provide some minimum space allocation guarantees, to allow them to get enough capacity to execute with low miss rate. *Cache Friendly* applications receive the highest priority. For the case of *Cache Friendly* and *Cache Fitting* applications, when multiple concurrently executing applications belong to the same category, the scheme allocates priorities within each category based on their *Interference Sensitivity Factor*, i.e. how friendly each application is to the co-running applications when it has to share the cache (Section 4.2.2).

3 MCFQ PROFILERS

The proposed MCFQ scheme allocates cache capacity based on: a) applications' performance sensitivity to misses (MLP-aware) and b) applications' cache memory behavior (Fitting, Friendly and Thrashing). Both of them are attributes of the applications running on the system that are dynamically changing as applications execute and interact with each other. To monitor the system and collect all the necessary information for our scheme, we need a set of on-line, low-overhead hardware profilers. This section provides a description of the proposed profilers that our scheme requires.

3.1 Applications Cache Demands Profilers

Our on-line cache demands monitoring scheme is based on the principles introduced by Mattson's stack distance algorithm (MSA) [16]. Our implementation is based on previously proposed hardware-based MSA profilers for last-level cache misses [19][32][10].

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, $K+1$ counters are needed, 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 [28] running on an 8-way associative cache. The application in the example shows a good temporal reuse of stored data in the cache as the MRU positions have a significant percentage of the hits over the LRU one. The graph of Fig. 2 can change according to each application's spatial and temporal locality. If we want to estimate the cache miss rate for a cache with half the size of the original one in the system, the "inclusion property" of LRU allows us to assume the measured hits in the $Counter_5$ to $Counter_8$ to be misses. Such an MSA-based profiling allows us to monitor each cores 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.

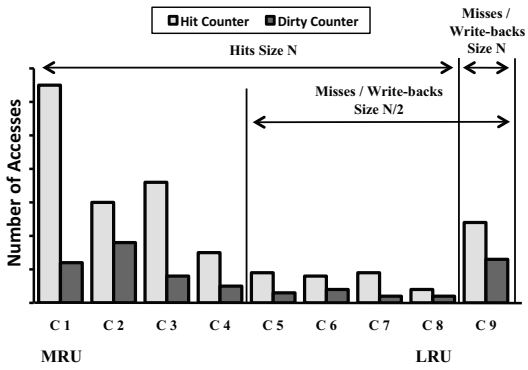


Fig. 2. Example of misses and bandwidth MSA histograms of *bzip2* benchmark [28].

To monitor each core individually, the scheme implements an individual profiler for every core which assumes that the whole cache is dedicated to it. Doing so, we maintain a set of shadow cache tags per core that allows us to monitor what would be each core's cache use if the core had exclusive access to the cache. Based on the positions of the hits in the profiler, we can create a cache miss rate curve that correlates cache misses with allocated cache capacity. The overall profilers' overhead is discussed in Section 3.4.

3.2 Profiler for Concurrency Factor (MLP)

To estimate the average *Concurrency factor* (MLP) of each application running on a single core, we profiled the *Miss Status Holding Registers* (MSHR). The MSHR typically exists in an out-of-order core between the L1 and L2 cache [14]. The purpose of the MSHR is to keep track of all the outstanding L1 misses being serviced by the lower levels of memory hierarchy (i.e. L2 and main memory). Therefore, the number of entries in the MSHR represents the number of the concurrent, long-latency, outstanding memory requests that were sent from the core to the last-level cache or memory, which is equal to the MLP of the application running on the core. To estimate the average MLP, we modified the MSHR by adding two counters: one to hold the aggregated number of outstanding misses in the MSHR, and a second one to hold the overall number of times an L1 miss was added in it. Both counters are updated every time an entry is added (new outstanding miss) or removed (miss served) from the MSHR.

3.3 Identify Applications' Cache Friendliness

To categorize the cache behavior of each application (Fig. 1), we utilize the MSA-based cache miss profiles that the profilers create for each individual core from 3.1. The profiler looks at the estimated Misses Per Kilo Instructions (MPKI) when only one cache-way is allocated to an application ($MPKI_{min.capacity}$), and the MPKI when the whole cache capacity ($MPKI_{max.capacity}$) is given to it. To characterize an application as $Thrashing$ the following has to be true:

$$\frac{MPKI_{max.capacity}}{MPKI_{min.capacity}} > Threshold_{Thrashing} \qquad (1)$$

In the same way, if at any capacity point of the cache miss profile (number of cache-ways dedicated to the core), the MPKI number is estimated smaller than a $Threshold_{Fitting}$ the application is characterized as $Fitting$. The remaining applications are set to $Friendly$.

Both thresholds ($Threshold_{Thrashing}$ and $Threshold_{Fitting}$) are system parameters which can be tuned to the specific system and the level of aggressiveness that is required by the partitioning scheme. For the analysis presented in this chapter, we performed a sensitivity analysis for both parameters to find the best values.

Our analysis, on SPEC CPU2006 benchmarks, showed that a threshold of $Threshold_{Thrashing} = 0.85$ provides the best trade-off for characterizing applications as thrashing/streaming. Bigger values allows applications with thrashing behavior to be characterized as friendly; polluting the cache. On the other hand, smaller values than this threshold treated potentially friendly applications that could be benefitted from additional cache capacity (even marginally) as thrashing applications, restricting them in one or two LRU ways in the cache. Finally the best value for the $Threshold_{Fitting}$ value was found to be 0.005 of the $MPKI_{min.capacity}$. This value was enough to guarantee that when an application achieves less than 0.5% of its initial cache miss rate at any cache allocation dedicated to it, we can treat it as a fitting application.

3.4 Profilers Overhead

The profilers overhead is basically dominated by the overhead of implementing the *Applications Cache Demands Profilers* (Section 3.1) on the hardware. To keep this overhead low, we used *partial hashed tags*, to avoid keeping the full cache tag of each cache access [11], and *set sampling* [12] to only sample a small random number of cache lines. In our implementation we use 11 bit *partial tags* combined with 1-in-32 *set sampling*. This selection of parameters added up only 1.3% error in the estimation of performance when compared to the accuracy we got using a full tag implementation and sample all cache set in the cache, over the whole SPEC CPU2006 [28] suite. The profiler counters' size for keeping the number of misses per cache way, was set to 32 bits to avoid overflows during the monitoring interval. In addition we implemented the circuit to keep the LRU stack distance of the MSA per monitored cache set as a single linked list with head and tail pointers. Overall, the implementation overhead is estimated to be 117 kbits per profiler, which is approximately 1.1% of the size of an 4MB last-level cache design; assuming 4 overall profilers for a 4-core CMP. The estimation of each core's MLP added only 2 counters per core as the MSHR structures necessary for our profilers in included in a typical out of order processor design. Using 32bit counters, the MLP estimation added only 64bits per core in the system. Finally, the circuit of identifying each applications memory behavior come for free as it reuses the applications cache demands profiler circuit described above. The remaining of calculations can take place in software when the routine to collect the profiling data and evaluate the next partition sizes is triggered. Such routine can be implemented as part of the OS scheduler in the kernel, that will collect the data through the use of memory mapped IO registers.

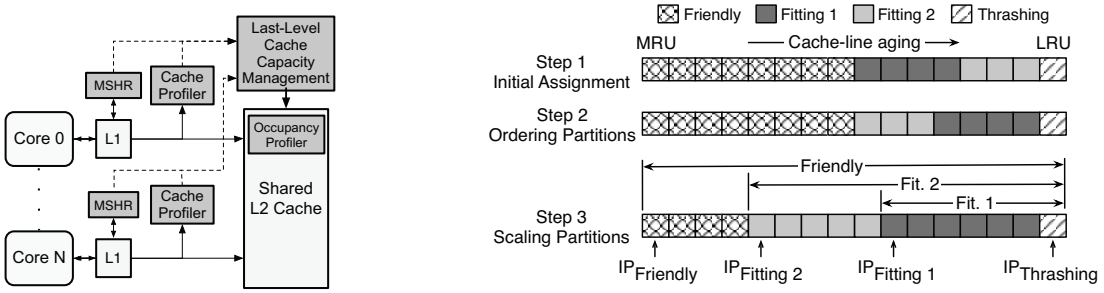
4 Cache-friendliness aware Quasi-Partitioning Scheme

Our scheme uses the MCFQ profilers (section 3) to gather all the information necessary for driving our cache management scheme. Our cache-friendliness aware scheme consists of two logical parts: a) the *MLP-aware Cache capacity allocation* (Section 4.1), and b) the *MCFQ pseudo-partitioning policies* (Section 4.2).

The first part estimates the ideal partition sizes that should be allocated to each core, assuming there will be no overlap of cache partitions. To do so, it uses as its optimization target the expected improvement of the final, observable performance of each application running on a core when a specific cache space is allocated to it. The methodology of estimating the performance sensitivity of applications to cache capacity is analyzed in the same Section 4.2.

The second part (Section 4.2) analyzes the cache pseudo-partitioning policies and two heuristics that we propose to introduce in the cache management scheme to make sure that each core will be able to maintain an average cache capacity occupancy as close to the ideal partition sizes estimated in the first part as possible, while it dynamically shares the cache capacity with the rest of the cores.

Fig. 3 illustrates the proposed last-level cache Quasi-partitioning scheme along with a simplified example. The dark shaded modules in Fig. 3(a), indicate our additions/modifications over a typical CMP system. Each core has a dedicated *Cache Profiling* circuit (Section 3.1) and we augmented the L2 design to add



(a) Overall MCFQ scheme.

(b) MCFQ operation with four applications on a 16-way cache.

Fig. 3. Overall MCFQ scheme on a typical CMP system and an example of allocation.

our *Cache-way Occupancy* profiler (Section 4.2.3). These two profilers along with the MLP statistics that we get from the MSHR (Section 3.2) are the inputs to the *Cache Capacity Management* module that controls the last-level cache.

The overall proposed dynamic scheme is based on the notion of epochs. During an epoch, our profilers constantly monitor the behavior of each core. When an epoch ends, the profiled data of each core is passed to the *Marginal-Utility* algorithm (Section 4.1) to find the ideal cache capacity assignment based on MLP. In parallel, our profilers identify the memory behavior that each application fits (Friendly, Fitting, Thrashing) and based on their category and their cache capacity assignment S_i, we decide the insertion points[2] (IP_i), for each core (Section 4.2). If there are more than one applications in the same category, the *Interference Sensitivity Factor* (Section 4.2.2) is used to assign the priority among them. Finally, the *Partition Scaling Heuristic* (Section 4.2.3) is used to scale the estimated partitions sizes to ensure that our scheme can on average meet the ideal partition sizes estimated by the *Marginal-Utility* algorithm. The whole process is repeated at the end of the next epoch. In our evaluation section we used epochs of 10M instructions.

Fig. 3(b) illustrates a simplified example on a 4-core CMP with a 16-way last-level cache. According to it, we have identified 1 $Friendly$, 2 $Fitting$ and 1 $Thrashing$ application. In the first step, we estimate the partition sizes S_i ($S_{Friendly}$=8, $S_{Fit.1}$=4, $S_{Fit.2}$=3 and $S_{Thrashing}$=1) and we order them giving the highest priority to the $Friendly$ ones and lowest to $Thrashing$. The two $Fitting$ get their initial priority based on their S_i sizes. Overall, the highest priority application inserts new cache line at the MRU position and the rest of IPs are based on the partition sizes from higher to lower priority. Since we have 2 $Fitting$ applications, in $Step$ 2 we estimate their $Sensitivity\ Factors$ and order them from higher to lower. Assuming that $Fitting$ 2 got a higher value, we swap the two applications with the proper change of their IPs. Finally, in $Step$ 3, we use the $Partitions\ Scaling\ Scheme$ to scale their partitions based on their average cache-way occupancies measured on last-level cache during the last epoch. The figure shows the finally selected IPs per application assuming we have actually scaled the partition sizes based on profiled data.

2. Points in cache LRU order that new cache lines are inserted in cache, more information in Section 4.2

4.1 MLP-aware Cache Capacity Allocation

An efficient last-level cache managing scheme must take into consideration the effects of multiple outstanding misses to performance and execution fairness. To model this sensitivity, we separate the execution time spent for each application into two parts: the time spent inside the core ($T_{Perfect\_Cache}$), assuming a prefect cache, and the time consumed outside the core serving cache misses (T_{Misses}). From the two, only T_{Misses} would depend upon the interference due to the resource sharing of the last-level cache capacity with other concurrently executing applications.

To model application's performance in terms of *Cycles-per-Instruction* (CPI) we can use Equation 2 [25]. Assuming a single L2 acting as the last-level cache, the CPI_{Misses} of Equation 2 can be broken down as the CPI when we hit in L2 (CPI_{Hit_L2}) and the CPI due to L2 misses that are directed to the main memory (CPI_{Memory}).

$$CPI = CPI_{Perfect\_Cache} + CPI_{Misses} \quad (2)$$

$$= CPI_{Perfect\_Cache} + CPI_{Hit\_L2} + CPI_{Memory} \quad (3)$$

To express the effects of concurrency among cache misses outside a core, we can extend Equation 3 to include MLP, that is, the average number of *useful* outstanding requests to the memory [2]. To do so, we can define the CPI_{Memory} term as:

$$CPI_{Memory} = \sum_{i=0}^{N_{misses}} \left(\frac{Miss\_Penalty_i}{Memory\_reference} \times \frac{Memory\_references}{Instructions_i} \right) \quad (4)$$

$$= Miss\_Rate_{memory} \times \frac{Miss_{Latency}}{Concurrency\ Factor} \quad (5)$$

As only the CPI_{Memory} term of Equation 3 is affected by last-level misses, Equation 5 is in essence a direct way to estimate the impact of last-level cache misses to final performance. The $Miss_{Latency}$ can be approximated to be equal to the average effective latency of an L2 miss to the main memory and therefore the only information missing from the equation is the $Miss\_Rate_{memory}$ and the *Concurrency Factor* (MLP) of an application. To estimate both terms, we utilize the online profiling mechanisms described in Section 3. The $Miss\_Rate_{memory}$ for every application and every possible cache capacity allocation can be calculated from the *Cache Demands Profilers* (Section 3.1) and the average *Concurrency Factor* can be measured using the *Profiler for Concurrency Factor* (Section 3.2).

Using all the previous information to understand and estimate the applications sensitivity to cache allocation and last-level cache misses, we propose a new *Cache Capacity Allocation Algorithm* to find the ideal cache partition sizes that should be assigned to each core. The algorithm is based on the concept of *Marginal Utility* [3]. This concept originates from economic theory, and has been used in cases where a given amount of resources (in our case cache capacity) provides a certain amount of *Utility* (reduced misses or CPI). The amount of *Utility* (benefit) relative to the used 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:

$$Marg.\ Utility(n) = \frac{Utility\ Rate(c+n) - Utility\ Rate(c)}{n} \quad (6)$$

Our cache miss profilers (Section 3.1) provide 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. The greedy algorithm implementation used in the paper is shown in Algorithm 1 and follows the one initially introduced by Stone et al. [29]. The algorithm estimates the cache capacity partitions assuming that each partition is isolated from each other. These partitions will be our ideal partition sizes that we want to enforce on our shared cache

To take MLP into consideration, the marginal utility is estimated based on the cumulative histograms of the last-level misses and the application's concurrency factor. The *Utility* function of the algorithm represents the effective additional benefit that each cache way can contribute to the final performance of an application. Our *Utility Rate* function, is estimated as the fraction of the cumulative hits at each cache-way over the average concurrency factor of the core for a given instruction window. That is:

Algorithm 1 Marginal-utility allocation algorithm.

```
num_ways_free = 16
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 {
            /* Marginal utility from eq. 6 */
            local_reduction = (MSA_hits(bank_assigned[core]
            +assign_num)−
            MSA_hits(bank_assigned[core]))/assign_num;
            /* keep the best reduction so far with best
            Marginal utility*/
            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;
}
```

$$Utility\_Rate(cache\ way) = \frac{MSA\_hits(cache\ way)}{Average\_Concurrency\_factor} \quad (7)$$

The $CPI_{Perfect\_Cache}$ of Equation 3 can be assumed to remain unchanged for the same application over a small, steady execution phase (we found that a 10M instruction window is a good assumption for the system under study in this paper). Therefore, by modifying the cache allocation of each core, we affect the other two terms in the equation, that is CPI_{Hit_L2} and CPI_{Memory}. Following this approach, our algorithm can effectively allocate capacity to the cores that affect the final observable CPI the most.

4.2 MCFQ Pseudo-partitioning Policies

Using the terminology from [31], a cache management scheme requires three basic pieces of information: a) an *Insertion Policy*; the location (cache-way) where to insert a new cache line in the LRU stack per application/core, named IP_i, b) a *Promotion policy*; the way LRU stack is modified on a cache hit, and c) a *Replacement Policy*: the exit point of cache-lines in case of line eviction.

A partitioning scheme like UCP [19] features one, non-overlapping, isolated partition per core that occupies a number of specific cache-ways, equal to its partition size. The insertion point of each core is set to the top of each partition; the promotion policy is set to move the cache line that got a hit to the MRU position of its partition; and the replacement policy is LRU among the ways that belong to the same partition.

On the other hand, a pseudo-partitioning scheme like PIPP [31] does not have isolated partitions. All

the cache ways are accessible by all the cores, simulating a typical LRU stack. To allow such pseudo-partitions, PIPP sets the insertion point (IP_i) of each core to be at each core's partition size, $IP_i = S_i$; the promotion policy was set to promote cache hit lines by one position based on a P_{prom} probability (otherwise stays unchanged); and the replacement policy was set to LRU evicting from the LRU position of the shared stack. Based on this scheme, there is a shared/unallocated portion of capacity from the largest IP_i to the MRU. This space can be utilized by applications' hot cache lines through line promotions; thus helping the cores with good cache locality.

As our evaluation shows, previous pseudo-partition schemes feature a significant level of performance drop when executing high resource demanding applications due to their Insertion/Promotion polices. Most of the applications can not efficiently utilize the capacity available in the cache in the MRU positions, featuring diminishing returns when allocating this capacity to specific threads, like isolated partitions do. Moreover, the interference that was introduced in the LRU positions found to be quite significant, especially for the Cache Fitting applications.

4.2.1 MCFQ Policies

To overcome these problems, MCFQ estimates the IPs on the basis of a) MLP and ideal partition sizes estimated in Section 4.1, b) applications' cache friendliness, and c) Interference sensitivity factors (Section 4.2.2).

The promotion policy is modified such that a cache line moves to its IP on a hit, i.e, a cache line is never allowed to move beyond its IP. Assuming an LRU replacement policy, the selected values for IPs can be translated into priorities P_i, with higher value for IP meaning higher priority. Our analysis showed that (Section 2.2), we should allocate the higher priority to *Cache Friendly* applications followed by *Cache Fitting* ones.

We restrict *Cache Thrashing* applications at the lowest one cache way of the LRU stack, similar to the TADIP [8] and PIPP [31] schemes. Thus the applications with higher priority have less competition in terms of promotion and demotion of cache lines in the LRU stack.

Our *Partition Scaling* heuristic (Section 4.2.3) scales the estimated partition sizes and IPs to ensure that *Cache Fitting* applications have enough capacity to avoid thrashing the cache. Between applications of the same category, we allocate priorities based on applications' interference sensitivity factor described in 4.2.2. Such approach significantly reduces the cache interference for low priority cache-ways. Since we share lower priority partitions, our *Partition Scaling* scheme makes sure that each application has enough capacity to maintain, on average, a number of cache ways close to the ideal estimated partition size. Therefore, for our case $\sum S_i >$#$cache\_ways$. By doing so, we can allow *capacity stealing* for high priority cores while maintaining a low threshold of capacity allocation to the lower-priority partitions.

4.2.2 Interference Sensitivity Factor Heuristic

While MCFQ has a clear priority scheme between applications with different cache friendliness behavior, we need a way to allocate individual priorities when more than one applications belong to the same category. We base our ordering on how sensitive is an application to cache contention and how much we expect it to hurt the cache behavior of other applications. To calculate the sensitivity of an application, we used the stack-distance profiles from our cache miss profilers (Section 3.1) to estimate the following *Interference Sensitivity Factor*:

$$Interference\ Sensitivity\ Factor = \sum_{i=0}^{\#ways-1} i * hits(i) \quad (8)$$

$Hits(i)$ in the above equation is the actual number of hits from our profiler on the i-th position in the stack (MRU position $i = 0$, LRU $i = \#ways - 1$). The more hits an application has at cache-ways closer to LRU, the more sensitive is the application to the cache contention. Under high contention, the effect of interference in misses is, on average, equivalent to allocating less cache space as a core's lines get evicted with a rate higher than its own demand rate. Hits closer to LRU positions have higher probability to become misses when the cache is shared. Moreover, in a quasi-partitioning scheme with different insertion points per core, the more hits an application has in LRU positions, the more useful cache lines will be evicted by other threads that insert lines at lower insertion points. Therefore, an application with high *Interference Sensitivity Factor* is more likely to see a degradation in performance.

4.2.3 Partition Scaling Heuristic

Since we selectively share a portion of the cache with multiple threads, we need to know what relative percentage of its allocated cache ways (based on its priority) an application actually maintains in the cache. If this number is significantly smaller than its ideal partition size, the application will feature a significant performance degradation. To do so, we added a monitoring scheme that utilizes the cache's core inclusivity bits to measure the average number of cache-ways per core.

Overall, we added two counters for every core the system supports, the $Occupancy\_Counter_i$ and $Cache\_Accesses\_Counter_i$. Whenever there is a cache hit or insertion of a new cache-line triggered by $Core_i$ we update the $Occupancy\_Counter_i$ with the number

of cache ways the $Core_i$ occupies in the set, and increase $Cache\_Accesses\_Counter_i$ by one. Having this information, we can estimate the average number of ways a core occupies.

If the average occupancy is less than 80% of the ideal size estimated by the *Marginal-Algorithm*, we increase the IP_i proportionally in the next epoch to bring the occupancy close to the ideal. This is especially important for *Cache Fitting* applications to avoid them thrashing the cache. If the correction is not feasible, we set the application to insert in the MRU position. Notice that, since we use the average occupancy numbers, the cores can still "steal" useful capacity in a subset of cache-sets.

4.3 MCFQ Inefficiencies

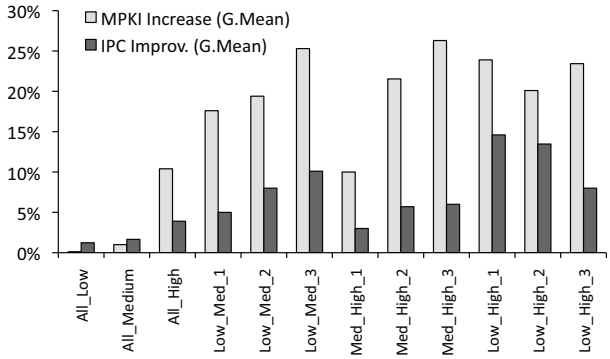
MCFQ policies do not implement a cache line promotion mechanism to allow cache lines from low priority applications to use cache space allocated to higher priority ones. As a result, cache space allocated to high priority applications can be temporarily underutilized in some systems. As we stated before, our scheme targets high performance, server space systems that are typically running applications with big working sets (e.g. databases, commercial/scientific data processing) that tend to use the majority of the cache space allocated to them and therefore the cache space is almost never underutilized. In our evaluation section we analyzed the benefits of promoting cache lines beyond their IP's (Figure 6) and we found such a scheme to provide marginal benefits for the system under study and for a relatively small epoch size (10M in our study). If the proposed scheme is implemented in a less resource demanding environment like personal computers, mobile platforms, etc. a cache line promotion heuristic could potentially benefit the overall cache utilization of the system.

5 Evaluation

To evaluate our scheme, we simulated a 4 and 8 cores CMP system using Simics functional model [27] extended with Gems tool set [17]. Gems provides an out-of-order processor model along with a detailed memory subsystem that includes an interconnection network and a memory controller. The default memory controller was augmented to simulate a *First-Ready, First-Come-First-Served* (FR_FCFS) [23] controller configured to drive a DDR3-1066 DRAM memory. Finally, we implemented a size N stride prefetcher that supports 8 streams per core. Table 1 summarizes the full-system simulation parameters. We used multi-programmed workloads using mixes of SPEC CPU2006 suite [28] that are shown in Table 2 for 4 and 8-cores CMP. The workload mixes combine benchmarks with different cache behaviors ($Friendly$, $Fitting$ and $Thrashing$) and levels of MLP. We used representative 100M instructions Simpoint phases.

Name	Description	Benchmark Set
All_Low	4 Low	gcc, perlbench, h264, astar
All_Medium	4 Medium	dealII, xalancbmk, gobmk, hmmer
All_High	4 High	bzip2, soplex, bwaves, mcf
Low_Medium_1	1 Low – 3 Medium	namd , dealII, sjeng, calculix
Low_Medium_2	2 Low – 2 Medium	sphinx3, GemsFDTD, tonto, povray
Low_Medium_3	3 Low – 1 Medium	sphinx3, h264ref, cactusADM, libquantum
Medium_High_1	1 Medium – 3 High	gobmk, soplex, mcf, leslied3d
Medium_High_2	2 Medium – 2 High	tonto, calculix, bwaves, omnetpp
Medium_High_3	3 Medium – 1 High	dealII, gobmk, hmmer, mcf
Low_Hgh_1	1 Low – 3 High	h264ref, bzip2, omnetpp, soplex
Low_High_2	2 Low – 2 High	astar, gcc, leslie3d, bzip2
Low_High_3	3 Low – 1 High	h264ref, astar, perlbench, omnetpp

(a) Experiments to evaluate MLP-assignment description.



(b) IPC and MPKI comparisons over UCP scheme.

Fig. 4. Evaluation of MLP-aware assignment of cache capacity assuming isolated cache partitioning scheme like UCP [19] on a 4core CMP system.

The MCFQ behavior is compared against three previously proposed schemes: a) a cache partitioning scheme based on isolated partitions: *Utility-based Cache Partitioning* (UCP) [19], b) a cache pseudo-partitioning scheme: *Promotion Insertion Pseudo-Partitioning* (PIPP) [31], and c) a dynamic cache line insertion policy: *Thread-aware Dynamic Insertion Policy* (TADIP) [8]. We implemented all these schemes by modifying Gems' memory model (Ruby). For each experiment we present the throughput, $Throughput = \sum IPC_i$, and fairness, estimated as the harmonic mean of weighted speedup, $Fairness = N/\sum(IPC_{i,alone}/IPC_i)$, where $IPC_{i,alone}$ is the IPC of the i-th application when it was executed stand-alone with exclusive ownership of all the resources [15].

5.1 MLP-aware Capacity Assignment

To evaluate the effectiveness of the MLP-aware cache capacity assignment algorithm (Section 4.1), we extended UCP isolated partitioning scheme using our $Utility\_Rate$ function and compared it against the original UCP. Fig. 4 provides a comparison of our approach against original UCP for 12 specific experiments (Table in Fig. 4(a)) executed on a 4-core CMP system. To use a representative set of benchmarks, we

TABLE 1
Full-system detailed simulation parameters.

Memory Subsystem	L1 D + I Cache	L2 Cache	Main Memory	Memory Controller	Prefetcher
	2-way, 64 KB, 3 cycles access time, 64B block size	16-way, 4 MB, 12 cycles bank access, 64B block size	8 GB, 16 GB/s, DDR3-1066-6-6-6	2 Controllers, 2 Ranks per Controller, 32 Read/Write Entries	H/W stride n, 8 streams / core, prefetching in L2
Core Characteristics	**Clock Frequency**	**Pipeline**	**Reorder Buffer /Scheduler**	**Branch Predictor**	
	4 GHz	30 stages / 4-wide fetch / decode	128/64 Entries	Direct YAGS / indirect 256 entries	

TABLE 2
Multi-programed benchmark sets from SPEC CPU2006 [28] suite for 4 and 8 cores.

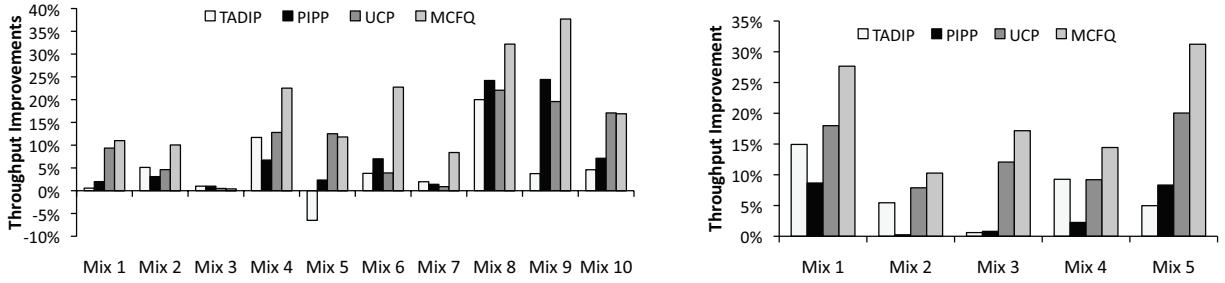
4 Cores		8 Cores	
Benchmark Group	**Benchmarks**	**Benchmark Group**	**Benchmarks**
Mix 1 - All Friendly	soplex, bzip2, h264ref, perlbench	Mix 1- All Friendly	soplex, omnetpp, perlbench, calculix, gromacs, dealII, calculix, gromacs
Mix 2 - All Fitting	xalancbmk, wrf, tonto, gamess		
Mix 3 - All Thrashing	leslie3d, sjeng, bwaves, zeusmp	Mix 2 - All Fitting	xalancbmk, gobmk, wrf, gobmk, hmmer, astar, gamess, hmmer
Mix 4 - 3 Fr.:1 Fit.	omnetpp, bzip2, calculix, astar		
Mix 5 - 2 Fr.:2 Fit.	bzip2, mcf, gobmk, gamess	Mix 3 - 4 Fr.:2 Fit.:2 Thr.	omnetpp, bzip2, gobmk, gromacs, povray, h264ref, lbm, libquantum
Mix 6 - 1 Fr.:3 Fit.	omnetpp, xalancbmk, gamess, wrf		
Mix 7 - 3 Fr./Fit.:1 Thr.	mcf, perlbench, hmmer, bwaves	Mix 4 - 2 Fr.:4 Fit.:2 Thr.	mcf, gobmk, gromacs, hmmer, gamess, tonto, libquantum, milc
Mix 8 - 2 Fr./Fit.:2 Thr.	xalancbmk, dealII, milc, zeusmp		
Mix 9 - 2 Fr.:1 Fit.:1 Thr.	mcf, bzip2, astar, leslie3d	Mix 5 - 2 Fr.:2 Fit.:4 Thr.	omnetpp, soplex, gobmk, gamess, libquantum, milc, zeusmp, milc
Mix 10 - 1 Fr.:2 Fit.:1 Thr.	mcf, gobmk, gamess, libquantum		

segmented SPEC CPU2006 suite in three categories based on their measured MLP. *Low* MLP threshold was set to 2; *Medium* selected between 2 and 4; and higher than MLP of 4 was characterized as *High*. Overall, we included cases for which a) the MLP of all benchmarks is approximately the same (first 3 experiments), b) there is a small difference in MLP between the benchmarks in the set (next 6 experiments), and c) there is a combination of benchmarks with significant big variation of MLP (last 3 experiments).

Fig. 4 includes the IPC improvement and the last-level cache's *Misses Per Thousand Instructions* (MPKI) degradation over simple UCP. Since our target was to improve the final performance and not to reduce the absolute number of misses, in all cases we actually happened to increase the number of misses but, in parallel, improved performance. The selection of benchmarks targeted the extreme cases to show the potentials of the scheme. In normal use, the MPKI is not expected to be so drastically increased. Despite that, the IPC is shown to improve up to 15% with an average (Geometric Mean) improvement close to 8% for our experiments. The benchmarks of the first three cases have comparable MLP and therefore, each application's cache misses are equally important for performance. As a result, the performance of our scheme is very close to UCP's.

In the cases of combining benchmarks with a small difference in MLP, that is *Low-Medium-1* to *Medium-High-3*, there is a significant gain in IPC close to 8% for *Low-Medium* and 6% for *Medium-High* categories. For these cases, the results show that the IPC is improved more when a small number of benchmarks with high MLP executes in the set; cases *Low-Medium-3* and *Medium-High-3*. That is a strong proof that, the high MLP benchmark in the case of simple UCP, was granted a bigger partition than what it should have been assigned. The main reason of such assignment is the use of the absolute number of misses to estimate its *Marginal Utility*. Such bigger partition is not actually contributing to performance, restricting the rest of the lower MLP benchmarks by occupying useful cache capacity. Our approach can recognize the impact of each application to final performance and effectively assign more capacity to the lower MLP benchmarks.

Finally, the biggest performance improvements are found in the last three categories where there is a significant difference between the MLP. The best IPC improvement took place for the case of 3 *High* MLP benchmarks. Our scheme can effectively recognize the importance of the lower MLP benchmark, *h264ref* in that case, and avoid allocating most of cache capacity to the higher MLP benchmarks that generate the majority of cache misses. The last two experiments followed with slightly smaller improvements that are



(a) 4 cores Throughput.

(b) 8 cores Throughput.

Fig. 5. Comparison of throughput improvements of MCFQ, TADIP, PIPP and UCP schemes for 4 and 8 cores over simple LRU scheme.

proportional to the number of benchmarks with a big difference in MLP.

5.2 Performance evaluation on a 4 & 8 cores CMP

Fig. 5(a) includes the throughput improvements TADIP, PIPP, UCP and MCFQ schemes achieved over simple LRU with no advanced cache management scheme, for the benchmark sets listed in Table 2 in the case of a 4-core system. To get the baseline behavior for each category, the first three mixes include applications from the same cache-behavior category; the next three mixes contain combinations of only $Friendly$ and $Fitting$ and finally the last four of them target interesting combinations of all behaviors.

Overall, MCFQ demonstrates significant improvements over the next best scheme of every category. The only cases where MCFQ is comparable to other schemes are Mix 3, Mix 5 and Mix 10. In Mix 3, all benchmarks are $Thrashing$ and therefore all schemes except UCP have chosen to insert new lines in MRU position, so performance is almost the same with LRU. UCP had evenly allocated the cache to all four applications but since the benchmarks are $Thrashing$ the cache, it could not get any real benefit out of it. On the other hand, Mix 5 and 10, include 2 $Fitting$ benchmarks and all schemes expect UCP, did worse than MCFQ because they could not guarantee a minimum number of ways for the $Fitting$ cores to avoid them entering the $Thrashing$ behavior. UCP and MCFQ managed to do equally well by minimizing the interference and providing enough capacity to the $Fitting$ applications. Overall, UCP did well in most cases that include a $Fitting$ application, even better than PIPP, which was initially unexpected.

Looking carefully at the statistics, Fig. 6 shows that for these cases, PIPP effectively used the excess of space in the MRU positions only in $Mixes$ 6, 8 and 9. For the rest of the cases, the space was practically unused, wasting space that UCP effectively used to improve performance. Furthermore, when multiple $Fitting$ applications had almost the same partition size, PIPP's insertion policy put them on the same IP; introducing severe contention and therefore forcing them to work in their $Thrashing$ area. This is clear from the fact that PIPP is better than UCP in Mix 6, 8 and 9 where only one fitting application exists. Despite that, MCFQ was still significantly better than PIPP in the same categories.

Finally, TADIP had an intermediate behavior by achieving comparable results to the second best in a small number of cases. Unfortunately, due to its policy to either insert a new line in the MRU or LRU position, it cannot properly handle $Fitting$ applications; while in Mix 5 it is even worse than simple LRU by 7%. As expected, TADIP gets reasonable performance improvements only in cases with $Thrashing$ applications. Overall, MCFQ for the specific 4-cores cases, managed an average improvement of 19%, 14%, 13% and 10% over LRU, TADIP, PIPP and UCP, respectively.

The 8-cores results in Fig 5(b) can potentially show how well each scheme can scale with the number of cores. Notice that since we have a 16-way last-level cache, each core in a 8-core CMP can get on average 2 ways. Therefore, these benchmark mixes put a lot of pressure on each scheme to keep the most important cache-lines for each application in the cache. As expected, UCP is the second best followed by TADIP. UCP can effectively choose the best isolated partitions size to improve performance while TADIP can handle the high demand rates by forcing the new lines from the applications that hurt performance the most to be allocated at the LRU positions. Both schemes though

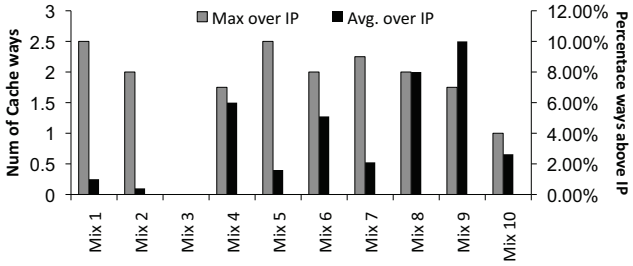
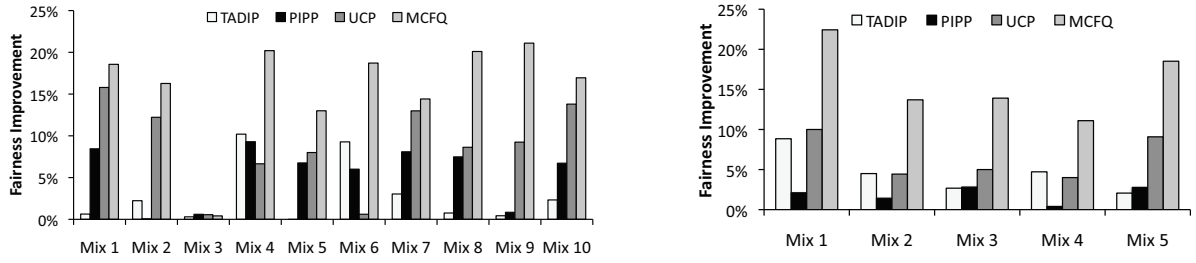


Fig. 6. PIPP's maximum and average number of cache-ways promoted higher than applications' IPs.



(a) 4 cores Fairness.

(b) 8 cores Fairness.

Fig. 7. Comparison of performance fairness/improvements of MCFQ, TADIP, PIPP and UCP for 4 and 8 cores over simple LRU scheme.

got gains by helping only a small number of threads, restricting the rest of applications to only one cache-way; hurting overall system fairness. PIPP showed the worst behavior since there are many applications with similar partition sizes that insert lines in the same IP.

Our statistics show that in most cases the applications were evicting each other's cache-lines, forcing them to a thrashing execution style. Overall, MCFQ showed significant throughput improvements due to its ability to share the capacity efficiently while minimizing the interference. For the 8-cores cases, MCFQ achieved an average improvement of 20%, 13%, 17% and 8% over LRU, TADIP, PIPP, and UCP, respectively.

5.3 Fairness evaluation on a 4 & 8 cores CMP

Fig. 7 provides a comparison of our fairness metric. By comparing Fig. 5(a) and Fig 7(a) one can see that the fairness of MCFQ is improved across the whole set of benchmarks even for the cases that MCFQ had comparable performance gains to other schemes (Mix 5 and 10). In addition, even for the cases that TADIP, PIPP and UCP provide a significant performance improvement over LRU, such improvement is coming from helping only a small number of threads; leading to significantly lower fairness than MCFQ's.

TADIP and PIPP cannot efficiently target fairness because of their greedy philosophy to help the applications that can get the most performance gains and they have no real protection mechanisms to ensure fair execution. UCP on the other hand, could be configured to provide fairness with proper allocation of capacity in its isolated partitions, but such approach would be wasteful and as it has been shown in the past, it hurts performance [6]. Therefore, UCP can either help throughput or fairness. Such behavior can also be confirmed by looking into the 8-core fairness results of Fig. 7(b). TADIP and UCP managed to get their performance gains by helping only a subset of applications. For 8-cores, the worst overall behavior was achieved by PIPP because under high pressure, its insertion point policy fails to help the competing threads, hurting its performance fairness. In addition, PIPP's significant reduction in fairness is an evidence of its inherent difficulty to scale its performance fairly when moving from 4 to 8 cores.

MCFQ, on the other hand, can successfully improve fairness by allocating capacity based on applications' performance sensitivity to cache space and misses. In addition, MCFQ's careful handling of $Fitting$ applications seems to improve fairness for the cases where many Fitting applications coexist (Mix 2, 6, 8 and 10). MCFQ achieved its best fairness performance in the cases where many $Friendly$ applications were competing for capacity. For these cases, the $Interference\ Sensitivity\ Factor$ could effectively reduce interference across the same category of applications and the $Partitions\ Scaling\ Scheme$ was able to fairly scale the capacities allocated to them. Overall, for the 4 core cases, MCFQ achieved an average improvement of 17%, 12%, 14% and 9% over LRU, TADIP, PIPP and UCP, respectively. Finally, for the 8 cores cases, the improvements were found to be 15%, 13%, 12% and 8% over LRU, TADIP, PIPP, and UCP, respectively.

6 Related Work

6.1 Isolated Cache Partitions

To solve the problem of destructive interference in the last-level-caches, researchers have proposed to partition the cache among different threads [19][24][1]. This is commonly done by means of *way-partitioning* i.e, in a set-associative cache each application is given the exclusive ownership of a fixed number of cache-ways. Suh et al. [24] proposed a greedy heuristic to allocate cache-ways proportionally to the incremental benefit that the thread gets from that allocation. Later on, Qureshi et al. [19] proposed Utility Based Cache Partitioning (UCP). They used utility monitors to estimate the utility of assigning additional ways to a thread and allocate ways based on this utility. Kaseridis et al. with their "Bank-aware" proposal [9] applied the basic concept of UCP on realistic CMP implementations with DNUCA-like caches to provide a scaling solution. Since in this paper we do not use a DNUCA cache, UCP and "Bank-aware" proposal are identical and the inclusion of UCP in our analysis

covers both of them. In an another publication from Kaseridis et al. [10] the authors extended Qureshi's work by taking in to consideration both cache capacity and memory bandwidth contention in large multi-chip CMP systems. This paper focuses only on cache capacity within single chip systems but can be extended to multi-chips using a similar to [10] hierarchical scheme. Moreto et al. [18] describes a partitioning scheme that uses the MLP information and it is based on the same principles with our approach. Based on our evaluation, even though using the MLP information can provide more accurate partition sizes, the scheme suffers of the same limitations that isolated partitions do. Our scheme, utilizes applications' memory behavior to create quasi-partitions that can exploit capacity stealing for efficient capacity utilization.

Finally, isolated cache partitions have been used aiming at the reduction of energy consumption. Reddy et al. [22] utilize cache partitioning to reduce power in embedded systems by keeping only part of the cache active at the time that belongs to the active execution thread. Our scheme focuses on dynamic pseudo partitioning of high performance processors with multiple concurrent execution threads for which all cache partitions need to be active in any time. Wang et al. [30], present an energy optimization technique which employs both dynamic reconfiguration of private caches and partitioning of the shared cache space for multicore systems with real-time tasks. Their algorithm, based on static profiling, finds beneficial cache configurations (of private caches) for each task as well as partition factors (of the shared cache) for each core so that the energy consumption is minimized while task deadline is satisfied. In contrast, our scheme is not designed for real time applications and we do not explicitly target energy minimization.

6.2 Dead-time Management

Cache lines with poor temporal locality occupy valuable cache resources without providing any actual benefits (cache hits). To minimize the life time of these lines, Qureshi et al. proposed Dynamic Insertion Policy (DIP) [20]. DIP identifies dead lines and inserts them at LRU position instead of MRU position resulting in their quick eviction. This allows more useful lines to be retained in the cache enabling better utilization of capacity. A subsequent proposal by Jaleel et al. [8] extends DIP to manage dead-time in the multi-core environments. They proposed Thread Aware Dynamic Insertion Policy (TADIP) which can adapt to memory requirements of competing applications.

6.3 Cache Pseudo-Partitioning

A recent proposal *Promotion/Insertion and Pseudo-Partitioning* (PIPP) [31] combines the ideas presented in UCP and TADIP to support a cache pseudo-partitions scheme. PIPP can provide good isolation for highly reused lines but, as we show in our evaluation, due to their insertion policy, still suffers from high destructive interference in the lower parts of the LRU stack. Rafique et al. [21] proposed an OS controlled technique where it is possible for applications to steal lines from other applications if they are not using their allocations effectively. MCFQ, on the other hand, implements quasi-partitioning by assigning different insertion points to applications keeping in view their MLP, cache memory behavior (Friendly, Fitting, Thrashing) and their interference sensitivity. This allows MCFQ to efficiently utilize the capacity while reducing the effects of interference. Herrero et al. in [5] proposed "Elastic Cooperative Caching" for NUCA caches. This scheme detects the dissimilar cache requirements of applications and distributes cache resources accordingly, allowing the creation of partitions that can be private, shared or both based on the demands of applications.

6.4 QoS/Fairness

Researchers have proposed cache partition algorithms that focus on improving fairness and/or Quality of Service (QoS) [4][7][6][13]. Kim et al. [13] highlighted the importance of enforcing fairness in CMP caches and proposed a set of fairness metrics to evaluate fairness optimizations. Chang et al. [1] proposed time-sharing of cache partitions, which transforms the problem of fairness to a problem of scheduling in a time-sharing system. Iyer et al. [7] proposed a framework for enforcing QoS characteristics in a system based on a trial-and-error scheme to fit the QoS targets. That work was later on extended by Zhao et al. [33] with a set of counters, named CacheScouts, that monitored their QoS characteristics in a system and, based on them, made resource management decisions. In this work we were motivated by the CacheScouts work to create our cache occupancy monitoring mechanism. Researchers have used the contention characteristics of applications [34] similar to our *interference-sensitivity* while making applications' co-scheduling decisions, but to our knowledge nobody has previously utilized such metrics for management of caches. Finally, Srikantaiah et al. [26] use formal control theory for dynamically partitioning the shared last level cache in CMPs by optimizing the last level cache space utilization among multiple concurrently executing applications with well defined service level objectives. This paper provides efficient policies for QOS based cache partitioning. We do not focus on QOS and a QoS-driven quasi-partitioning scheme is part of our future research endeavors.

7 Conclusions

In this work we present a last-level cache quasi-partitioning scheme that effectively allocates capacity

to resource competing applications, while minimizing destructive interference. We show that by utilizing applications' memory-level parallelism, MCFQ can predict applications' performance sensitivity to last-level cache misses and therefore, target final system throughput improvements. We demonstrate through cycle-accurate simulation that MCFQ significantly reduces the effects of applications' shared cache interference by categorizing and assigning priorities according to applications' *cache friendliness* behavior (Friendly, Fitting, Thrashing) and their performance sensitivity on sharing cache capacity (*Interference Sensitivity factor*).

REFERENCES

[1] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors", In *International Conference on Supercomputing (ICS)*, pages 242–252, 2007.
[2] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism", In *International Symposium on Computer Architecture*, pages 76–88, 2004.
[3] Wieser, Friedrich von, "Der naturliche Werth [Natural Value]", Book I, Chapter V, 1889.
[4] F. Guo, *et al.*, "From Chaos to QoS: Case Studies in CMP Resource Management", *Comp. Arch. News*, 2007.
[5] E. Herrero, et al., "Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors," In *Proceedings of international symposium on Computer architecture*, pp 419–428, 2010.
[6] L. Hsu, S. Reinhardt, R. Iyer and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource", *In Parallel Architectures and Compilation Techniques (PACT)*, pages 13–22, 2006.
[7] R. Iyer, *et al.* "CQOS: A Framework for Enabling QoS in Shared Caches of CMP Platforms", *International Conference on Supercomputing (ICS)*, pages 257-266, 2004.
[8] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches". In *Parallel Architectures and Compilation Techniques (PACT),* pages 208–219, 2008.
[9] D. Kaseridis *et al.* "Bank-aware Dynamic Cache Partitioning for Multicore Architectures," In *International Conference on Parallel Processing (ICPP)*, pages 18-25, 2009.
[10] D. Kaseridis *et al.* "A Bandwidth-aware Memory-subsystem Resource Management using Non-invasive Resource Profilers for Large CMP Systems," *High Performance Computer Architecture*, pp 93–105, 2010.
[11] R. Kessler, R. Jooss, A. Lebeck, and M. Hill, "Inexpensive implementations of set-associativity", In *International Symposium on Computer Architecture (ISCA)*, pages 131–139, 1989.
[12] 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):664–675, 1994.
[13] S. Kim *et al.*"Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", *In Parallel Architecture and Compilation Techniques (PACT)*, pages 111–122, 2004
[14] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization", *Intl. Symp. on Computer Arch.*, 1981.
[15] K. Luo et al., "Balancing throughput and fairness in smt processors," In ISPASS, 2001.
[16] R. L. Mattson *et al.* "Evaluation techniques for storage hierarchies". IBM Systems Journal, 9(2):78–117, 1970.
[17] Milo M. K. Martin *et al.* "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", *Computer Architecture News (CAN)*, September 2005
[18] M. Moreto *et al*, "MLP-Aware Dynamic Cache Partitioning," In *High Performance Embedded Architectures and Compilers (HIPEAC)*, pp. 337–352, 2008.
[19] M. K. Qureshi and Yale N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", In *Symp. on Microarchitecture (MICRO)*, pages 423-432, 2006.
[20] M. K. Qureshi, *et al.* "Adaptive Insertion Policies for High-Performance Caching", *In Symposium on Computer Architecture (ISCA),*pages 381–391, 2007.
[21] N. Rafique, W. Lim, and M. Hottethodi, "Architectural support for operating system-driven CMP cache management", In *Parallel Architectures and Compilation Techniques* pp. 2–12, 2006.
[22] R. Reddy and P. Petrov, "Cache partitioning for energy-efficient and interference-free embedded multitasking" ACM Trans. on Embeded Computers and Systems 9, 3, Article 16 (March 2010).
[23] S. Rixner *et al.*, "Memory access scheduling," in *International Symposium on Computer Architecture*, 2000.
[24] G. E. *Suh et al.*, "Dynamic partitioning of shared cache memory", *Journal of Supercomputing, 28(1):7-26, 2004.*
[25] J. Shen *et al.*,"Modern Processor Design: Fundamentals of Superscalar Processors", McGraw-Hill Education.
[26] S. Srikantaiah*et al.*. "SHARP control: controlled shared cache management in chip multiprocessors," in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42).
[27] Simics Microarchitect's Toolset, http://www.virtutech.com/
[28] SPEC cpu2006 Benchmark Suit, http://www.spec.org
[29] H. S. Stone *et al.* "Optimal partitioning of cache memory", *IEEE Transactions on Computers*, 41(9), 1992.
[30] W. Wang et al., "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in 48th Design Automation Conference (DAC '11).
[31] Yuejian Xie and Gabriel H. Loh, "PIPP: Promotion/Insertion Pseudo-partitioning of Multicore Shared Caches," *In symposium on Computer architecture (ISCA),* pages 174–183 , 2009.
[32] P. Zhou *et al.* "Dynamic Tracking of Page Miss Ratio Curve for Memory Management", *Architectural Support for Programming Languages and Operating Systems*, pages 177–188, 2004.
[33] L. Zhao *et al.*, "CacheScouts: Fine-grain monitoring of shared caches in CMP platforms", In *Parallel Architecture and Compilation Techniques (PACT),* pages 339–352, 2007.
[34] S. Zhuravlev *et al.*, "Addressing shared resource contention in multicore processors via scheduling", In *Architectural Support for Programming Languages and Operating systems,* 2010.

Dimitris Kaseridis received his B.Sc and M.Sc from the University of Patras, Greece in 2004 and 2005 respectively, and his Ph.D in computer engineering from The University of Texas at Austin in 2011. He is currently a performance architect in ARM, USA working on power efficient and scalable interconnects, last-level caches and memory controller designs. His research interests include computer architecture, performance modeling and analysis.

Muhammad Faisal Iqbal received his bachelor's degree in electronic engineering from Ghulam Ishaq Khan Institute, TOPI, Pakistan in 2003, and his M.E in computer engineering from The University of Texas at Austin in 2009. He is currently a Ph.D candidate at The University of Texas at Austin. His interests include microprocessor architecture, power and performance modeling, workload characterization and low power architectures.

Lizy Kurian John is B. N. Gafford Professor of Electrical Engineering at the University of Texas at Austin. She received her Ph.D in computer engineering from The Pennsylvania State University in 1993. Her research interests include microprocessor architecture, performance and power modeling, workload characterization, and low power architecture. She is an IEEE fellow.