An Adaptive Bloom Filter Cache Partitioning Scheme for Multicore Architectures

Konstantinos Nikas
School of Electrical and Computer Engineering
National Technical University of Athens
knikas@cslab.ece.ntua.gr

Matthew Horsnell
School of Computer Science
University of Manchester
horsnelm@cs.man.ac.uk

Jim Garside
School of Computer Science
University of Manchester
jdg@cs.man.ac.uk

***Abstract*—This paper investigates the problem of partitioning the last-level shared cache of multicore architectures. Contention for such a shared resource has been shown to severely degrade performance when running multiple applications. As architectures incorporate more cores, multiple application workloads become increasingly attractive, further exacerbating contention at the last-level cache. Today, cache replacement policies, extensively studied for uniprocessor systems, are being employed within new multicore architectures with little, if any, adaptation. However the parameters in these new systems are likely to be different. The *least recently used (LRU)* policy, for example, which is widely accepted as the best replacement policy in uniprocessor caches, often results in poor resource sharing in a multicore system, signalling the importance of reevaluating the effectiveness of these policies in the new architectures.**

This paper proposes *Adaptive Bloom Filter Cache Partitioning (ABFCP)*, a low-cost, dynamic cache partitioning mechanism capable of better resource sharing at the last-level cache than LRU, improving the performance of an eight-core system on average by 5.92% over the LRU policy. Moreover, the proposed scheme provides the equivalent performance benefits that could be gained from almost a 50% increase in the last-level cache and shows increasing benefit as the number of cores rises.

***Index Terms*—Cache partitioning, Bloom filters, Multicore architectures.**

I. Introduction

Improvements in silicon process technology have facilitated the integration of multiple cores into modern processors and it is anticipated that the number of cores on a single chip will continue to increase in the future. Multiple application workloads, attractive for utilising multicore processors, put significant pressure on the memory system. This motivates the need for more efficient use of the last-level shared cache in order to minimize the expensive, in terms of both latency and power, requests to off-chip memory. This paper investigates the problem of partitioning the last-level cache in the presence of multiple concurrently executing applications.

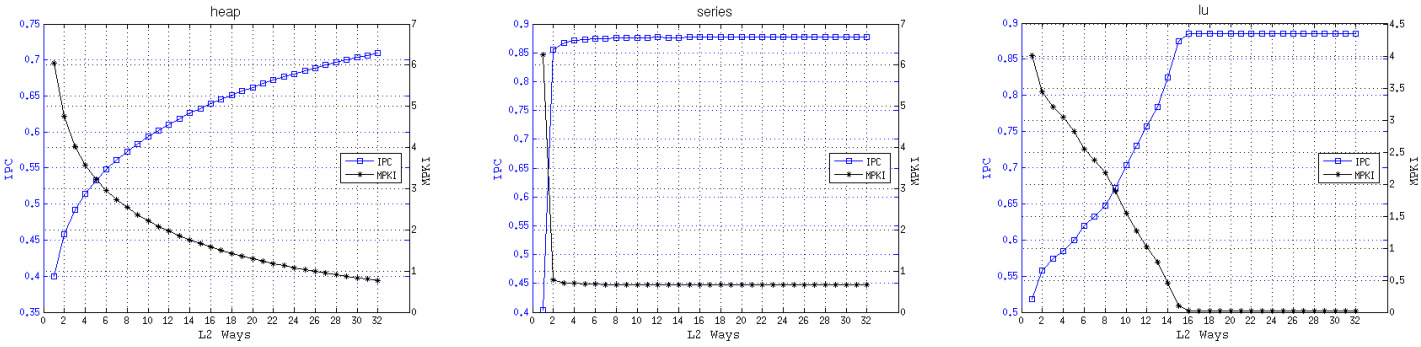
A key factor to the efficiency of a shared cache is the line replacement policy. The majority of CMP systems today [10], [12], [13] employ the *least recently used* (LRU) policy, widely accepted as the best line replacement policy in uniprocessor caches, or approximations thereof. The LRU policy, in the context of a shared cache, implicitly partitions the cache by allocating more space to the application with the highest demand, i.e. many accesses to different entries. However, not all applications benefit from exploiting additional cache resources. A typical example is a streaming application, where data is fetched into the cache, processed and then is unlikely to be reused. A replacement policy, such as LRU, will naïvely keep allocating resources to a streaming application, even though there are no performance benefits. Consequently, other concurrently running applications are deprived of resources that they could have efficiently exploited.

In a multicore system several applications will be running in parallel and the benefit to each one of obtaining additional cache resources will vary. Applications also exhibit distinct phases during execution, each with varying cache requirements. Therefore a dynamic partitioning of the last-level shared cache, allocating cache resources in a way that will maximise the overall system performance is attractive. In this paper we propose the *Adaptive Bloom Filter Cache Partitioning* (ABFCP) scheme which strives to achieve dynamic beneficial allocation. The last-level shared cache is instrumented, using a combination of Bloom filters and counters, to determine how the concurrent applications will benefit from additional cache resources. Based on this information, the cache is repartitioned periodically and the information is reset. This scheme, which incurs minimal hardware overhead, around 5.5% over the area of a 4MB, 32-way associative L2 cache, improves performance over an LRU cache on average by 5.92% for an eight-core system. Moreover, it provides the equivalent performance benefits that could be gained from almost a 50% increase in the last-level cache and shows increasing benefit as the number of cores rises.

This paper is structured as follows: Section II outlines the motivation for this work and Section III describes related work. In Section IV the proposed ABFCP scheme is described. Section V outlines our experimental methodology and Section VI presents an evaluation of the proposed scheme. Finally, Section VII concludes.

II. Motivation and Background

An increase in cache space affects the performance of each application differently. The amount of performance gain in response to additional cache space can be described as an application's cache utility. As presented by Qureshi and Patt [15], this cache utility metric can be used to classify



(a) High Utility (b) Low Utility (c) Saturating Utility

Fig. 1. Examples of different cache utility profiles.

applications into three categories; *high*, *low* and *saturating*. Fig. 1 presents an example of this classification using three benchmarks from the JavaGrande suite [17]. The cache size is varied by changing the number of ways and keeping the number of sets constant.

High utility applications, such as heap, continue to benefit significantly from increases in cache resources as shown in Fig. 1(a). As more ways, each 128KB, are added to the cache, heap's instructions per cycle (IPC) increases while its misses per thousand instructions (MPKI) are reduced. On the other hand, *low utility* applications, such as series shown in Fig. 1(b), gain little or no performance gains from increasing the cache space as the number of their misses remains stable. Finally, *saturating utility* applications, such as lu shown in Fig. 1(c), benefit from an initial increase in cache resources, then gain no performance beyond a given cache size, where their misses drop almost to zero.

When multiple applications execute concurrently in a multi-core system there is an opportunity to exploit the differences in their cache utility. To illustrate this, the following experiment is performed. A dual-core system, where the two processors share a 4MB, 32-way associative L2 cache, is used to execute heap and sor from the JavaGrande benchmark suite [17] in parallel (other parameters of the simulated system are described in Section V). The former is an application of high utility, while the latter is an application of low utility. The cache is statically partitioned each time between the two applications by allowing each core to use a specific subset of the available cache ways.

Fig. 2 compares the IPC of each application and the total IPC, labelled IPCSum, to those achieved when the two cores share the cache using the LRU replacement policy. The overall system performance is clearly affected by altering the number of ways that each application is allowed to replace lines within. Because, in general, heap has a lower IPC than sor, allocating the majority of the L2 cache ways to sor causes IPCSum to increase by 10% compared to the LRU policy.

This static partitioning, however, has two main drawbacks. Firstly, to partition the cache correctly, the system must be aware of each application's profile, as it is possible to degrade

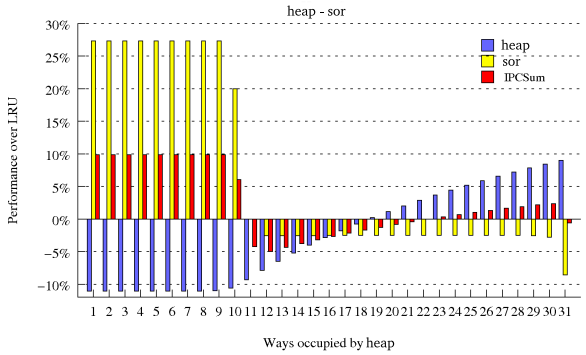


Fig. 2. Effect of statically partitioning a 4MB (32-way associative) cache when executing heap (high utility) and sor (low utility).

the performance. This is illustrated in this example, when sor is allowed to use only 12 out of the 32 available cache ways and IPCSum is 5% worse compared to the performance achieved when LRU is used. A second disadvantage is that the partitions remain the same during the execution, even though applications are known to have distinct phases of behaviour. These provide a strong motivation to develop a scheme that can partition a shared cache dynamically, based on an execution profile acquired at run-time, with the ability to re-partition the cache should the phase of an application change.

III. Related Work

Different partitioning schemes for shared CMP caches have been developed. The majority of them attempt to exploit the stack property of the LRU policy [11], according to which an access that hits in an LRU managed N-way associative cache is guaranteed to also hit if the cache had more than N ways, provided that the number of sets remains constant. Suh *et al.* [18] first described a mechanism that counts the cache hits to the different recency positions, ranging from MRU to LRU, on a per process basis. The proposed partitioning algorithm, invoked every 5 million cycles, uses these counters to estimate gains (losses) for each process in case it is assigned more (less) cache space and repartitions the cache in an attempt to maximise the overall performance by reducing off-chip misses.

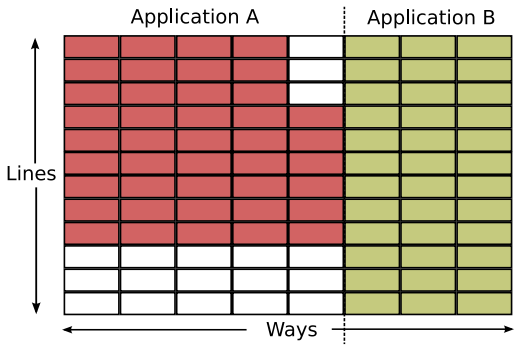


Fig. 3. Example of non-optimal partitioning

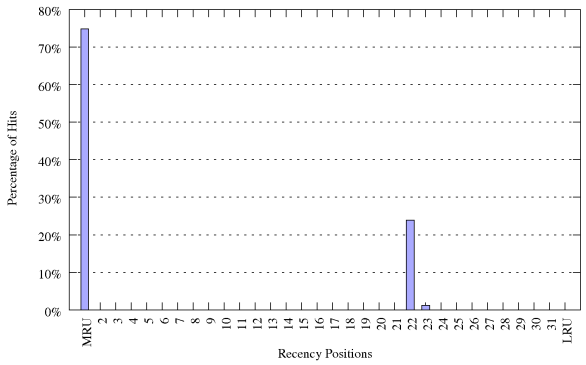


Fig. 4. LRU stack profile for sor

Recently Qureshi and Patt [15] introduced "Utility-based Cache Partitioning" (UCP) by implementing utility monitors (UMONs), which shadow the tag array of the L2 cache for each processor and essentially emulate a private L2 cache per core. Counters kept for accesses to each way in the UMONs are read every 5 million cycles by the partitioning algorithm, which then allocates cache ways to the running applications attempting to minimise the total number of misses for the next period. To keep the hardware overhead low, each UMON monitors only a small subset of the cache sets and enforces the same partition for the whole cache.

Dybdahl *et al*. [5] proposed "Cache-Partitioning Aware Replacement Policy" (CPARP). They add a *shadow* register together with the *shadow hit* and *LRU hit* counters for each processor in each cache set. The shadow register stores the last evicted tag and thus can identify misses that would have been hits had the processor been allowed to occupy one more cache way. Every time such a miss is identified, the shadow hit counter is incremented, while the other counter tracks the accesses that hit in the LRU position. Every 2,000 misses the maximum value of the shadow hit counters is compared to the minimum value of the LRU hit counters. If it is found to be greater, then the allocation of the process associated with the shadow counter, i.e. the process with the greatest desire for one more way, is increased while the allocation of the process associated with the LRU counter, i.e. the process with the smallest loss if it is deprived of one way, is decreased.

The UCP scheme is able to repartition the cache by allocating any amount of ways to each core, whereas CPARP only allows ± 1 way changes. On the other hand, CPARP, by monitoring the shadow misses in each cache set, is able to enforce a different partition for each set while UCP enforces the same partition for the whole cache. This means that CPARP can adapt better to the requirements of each processor, as it is known that applications could use only a subset of the cache sets as shown in Fig. 3. In this example, by enforcing the same partition for the whole cache, Application B has been deprived of valuable resources.

However, CPARP is not able to handle efficiently workloads with non-convex miss rate curves. For example, consider sor, whose LRU stack profile is shown in Fig. 4. According to this profile, 75% of the hits occur on the MRU position, 24% on the 22nd and 1% on the 23rd recency position. According to the stack property of the LRU policy, if sor is allowed to occupy less than 22 ways then the hits that occur on the 22nd and 23rd position will become misses. CPARP is only able to detect shadow misses, which means that it will be able to identify these misses and consider assigning more cache resources to sor, if and only if sor occupies 21 ways. On the other hand, UCP, by using the UMONs, which monitor all the cache ways, and a lookahead partitioning algorithm, is able to identify the characteristics of sor and make the appropriate partitioning decisions.

Based on these observations we have designed a novel cache partitioning scheme that attempts to combine the advantages of both CPARP and UCP. More specifically, it attempts to identify the characteristics of the running applications accurately like UCP, while maintaining the flexibility of CPARP that will allow to dynamically partition the cache on a set granularity.

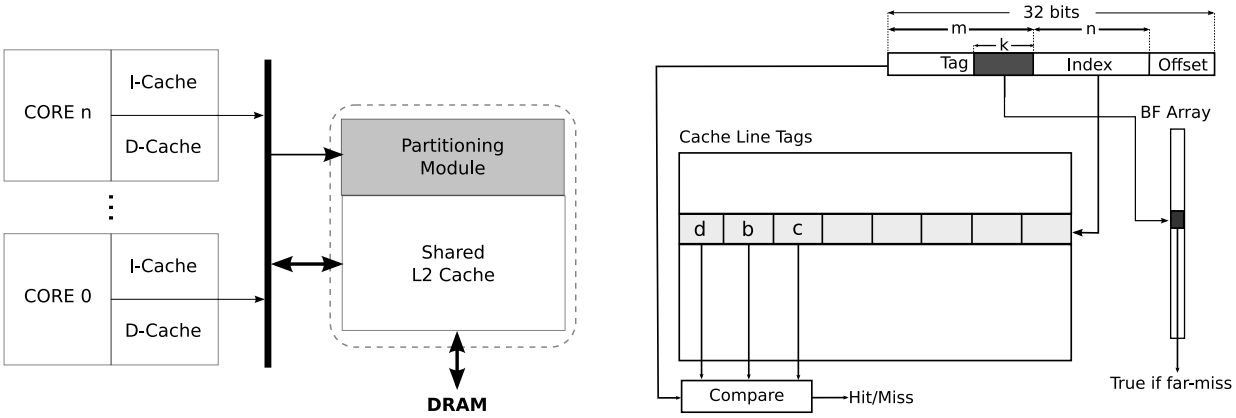
Finally, partitioning, as a mechanism for reducing memory pressure, has also been proposed at the memory controller level [16] or to provide quality of service [20]. Additionally, by exposing cache usage information the OS can guide thread scheduling [3], [6] to improve cache sharing.

IV. Adaptive Bloom Filter Cache Partitioning

A. Overview

This section presents the Adaptive Bloom Filter Cache Partitioning (ABFCP) scheme, a novel low-cost cache partitioning scheme based on the concept of Bloom filters [2]. A Bloom filter is a structure for maintaining probabilistic set membership. It trades off a small chance of false positives for a very compact representation. The Bloom filters used in this scheme are based on the "Partial-Address Bloom Filter" developed by Peir *et al*. [14].

Fig. 5(a) illustrates a system employing the ABFCP scheme. A partitioning module is attached to the L2 shared cache and monitors all core accesses. The partitioning module needs to track the actual cache occupancy of each application running in each core, and therefore a *core ID* field is added alongside the address tag of each cache line. Whenever a core brings a line into the cache, its ID is stored in the *core ID* field. This



(a) Overview. (b) Cache monitoring components.

Fig. 5. The Adaptive Bloom Filter Cache Partitioning scheme

field is necessary in order to keep a record of the misses and hits for each core, which is subsequently used to repartition the cache.

B. Tracking misses and hits

The partitioning module tracks both misses and hits on a per core basis. However, there is no need to track all misses in general, but only those that may have been hits had the core been allowed to use more cache ways. This subset of misses is defined as "far-misses" and is monitored by Bloom filters.

More specifically, a Bloom Filter Array (BFA) with 2^k bits is added to each set for each core. When a tag is rejected from the cache, its k least significant bits are used to index a bit of the BFA, which is then set. On a cache miss, the appropriate entry of the BFA is looked up using the k least significant bits of the requested tag. If the array bit is *set*, then a far-miss has been detected. Consider the example shown in Fig. 5(b), where an application is only allowed to occupy three cache ways, which initially hold the tags *a, b* and *c* with *a* being the LRU. When *d* is brought into the cache, *a* is replaced and its k least significant bits are used to set the appropriate bit in the BFA associated with the core executing the application. If tag *a* is requested again, then it is resolved as a miss and a lookup in the BFA reveals that the bit indexed by the k least significant bits of *a* is set, thus identifying a far-miss.

When the BFA bit is found to be clear, it is certain that the requested tag was not stored in the cache in the past. On the contrary, if the bit is found to be set, there is a possibility of a *false positive*, due to the problem known as "*aliasing*". As only k bits of the tag are used to index the BFA, it is possible for more than one tag to map to the same array bit. This means, that the system may detect a far-miss for an address that was never brought into the cache. Moreover, in the system described in Fig. 5(b), the order in which the BFA bits are set is not recorded, and so, there is no information on the order in which lines were rejected from the cache. At the same time, the number of true entries in the BFA could be higher than the associativity of the cache. Therefore, when a far-miss is detected, it is not possible to deduce how many more ways the core should have been allocated for that miss to become a hit. That information could have been acquired if the Bloom filter was extended with a set of counters and registers. However, as these extensions would increase the hardware overhead and the complexity of the system, they were rejected. For these reasons, far-misses are defined as misses that *may* have become hits, had the application been allocated more cache ways in general. In practice, they are used to decide whether a core's allocation should be increased by one way.

The cache partitioning mechanism also tracks hits to the LRU position per set for each core. In total, two counters per core are used for each cache set:

$\mathbf{C_{far\text{-}miss}}$: incremented when the BFA detects a far-miss.
$\mathbf{C_{LRU}}$: incremented when a cache access hits on the LRU entry owned by the core.

Exploiting the stack property of the LRU, these counters provide estimates of performance gains or losses possible by changing a core's allocation. More specifically, according to this property, if the allocation of a core was reduced by one way, then the hits in the LRU position would become misses. Therefore, for this case the value of the C_{LRU} counter estimates the performance loss, as shown in (1). As mentioned previously, the $C_{far-miss}$ counter could include far-misses that would become hits if the core's allocation was increased by more than 1 way. Therefore a factor, $a = 1 - \frac{ways_{occupied}}{associativity}$, is introduced to scale the value of the $C_{far-miss}$ counter and the performance gain in this case is shown in (2).

$$lose_1 = C_{LRU} \quad (1)$$

$$gain_1 = a \times C_{far-miss} \quad (2)$$

Finally, it should be noted that the update and lookup of the Bloom filters and the counters are not on the critical path, as the cache response to a processor request does not depend on them. Therefore, the cache access times should remain the same.

TABLE I
NUMBER OF POSSIBLE PARTITIONS

Processor Cores	Possible Partitions
2	3
4	19
8	1107
16	5196627

C. Partitioning Algorithm

At the end of each monitoring period the partitioning algorithm is executed. As the total number of cache ways does not change, the possible partitions that need to be evaluated can be deduced from (3), where Δx_i is the change in the allocation of core i and N is the number of cores.

$$\sum_{i=1}^{N} \Delta x_i = 0, \qquad \Delta x_i = \{-1, 0, 1\} \tag{3}$$

The number of solutions of (3) is shown in Table I for different numbers of cores. It is obvious that the algorithm does not scale efficiently, as for 16 cores the partitioning module needs to evaluate over 5 million different partitions. Even for 8 cores, the comparison of 1107 partitions is not possible, as it needs to be done for every cache set. Therefore, a linear algorithm that selects the best partition or a good approximation thereof has been developed and is listed in Algorithm 1.

Algorithm 1 Linear Partitioning Algorithm

```
cores = N
for core i = 0 to N do
    gain_1[i] = a × C_far−miss
    lose_1[i] = C_LRU
end for
order gain_1, lose_1 from min to max
while max gain_1 > min lose_1 do
    increase allocation[i] by 1, decrease allocation[j] by 1
    remove i and j from gain_1, lose_1
    cores -= 2
    if cores ≤ 1 then
        return allocation;
    end if
end while
return allocation;
```

The algorithm reads the hit and miss counters of each core. Then, in each iteration, the maximum gain value is compared against the minimum loss value. If the former is greater, then the allocation of the core associated with that $C_{far-miss}$ counter is increased by one way, while the core associated with that C_{LRU} counter is deprived of one cache way. The process is repeated until no cores are left to be considered or the maximum gain value is smaller than the minimum loss value. In the worst case $N/2$ comparisons need to be performed, where N the number of cores. Therefore, the complexity of this algorithm is O(N).

TABLE II
BASE CONFIGURATION.

Processor cores	2, 4, 8 configurations single issue in-order, 5 stage pipeline 32-bit RISC ISA
Level 1 caches	Private instruction and data caches 32KB, 4-way set-associative, 32B line size 1 cycle access latency
Unified shared Level 2 cache	4MB, 32-way set-associative, 32B line size 16 cycle access latency
Memory	Maximum 32 outstanding requests 100 cycle access latency

Finally, the system guarantees that each core is allocated at least one cache way in each set to avoid problems like thread starvation. The partitioning algorithm is executed every one million cycles and at the end of the repartitioning phase the counters and the Bloom filters are reset.

D. Changes to Replacement Policy

To enforce the decisions made by the partitioning algorithm the baseline LRU policy is modified to enable way partitioning [4], [9], [18]. On a cache miss, the replacement mechanism counts the number of cache blocks in the accessed set that belong to the miss invoking core. If this number is equal to or greater than the number imposed by the partitioning algorithm then the LRU block belonging to that core is rejected. Otherwise, the LRU block of an over-allocated core is evicted. If the number of ways allocated to a core is increased, then the new ways are consumed only on cache misses. This lazy reallocation allows the cache to retain the previously stored cache blocks until the space that they occupy is actually needed.

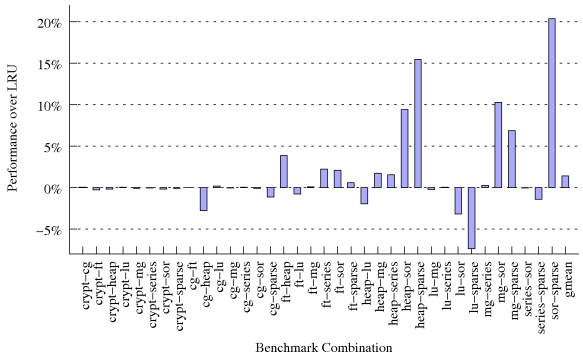
V. EXPERIMENTAL METHODOLOGY

We use an in house cycle-level simulator, modelling the JAMAICA chip multiprocessor architecture [8], [19]. The instruction set is based on the 32-bit Alpha ISA. The simulation platform models the caches, buses and memory in sufficient detail to account for contention, queue blocking and access delays. Table II shows the parameters of the baseline configuration used for simulation.

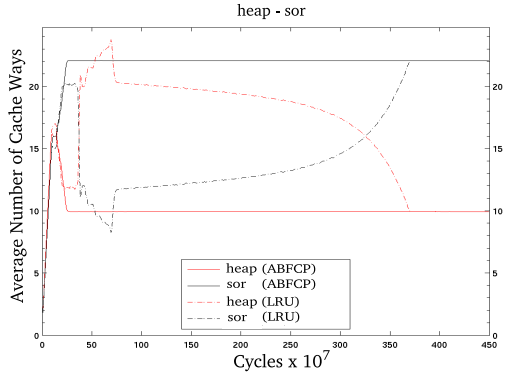
For each evaluation we use a set of nine Java benchmarks: `ft`, `mg` and `cg` are taken from the NAS parallel benchmark suite [7] while `heap`, `sor`, `sparse`, `series`, `crypt` and `lu` are taken from the JavaGrande benchmark suite [17]. During simulation, multiprogrammed workloads are run within a Java virtual machine, ported from the Jikes RVM [1], ensuring each single benchmark is executed within a single processor core in the simulator. The benchmarks chosen exhibit varying cache utility profiles, with three benchmarks fitting into each of the categories, high-utility (`ft`, `heap`, `mg`), low-utility (`sor`, `sparse`, `series`), and saturating-utility (`crypt`, `cg`, `lu`), as described in Section II.

VI. RESULTS AND ANALYSIS

We evaluate ABFCP for a dual, a quad and an eight-core system with a shared 4MB, 32-way associative L2 cache. Fig.

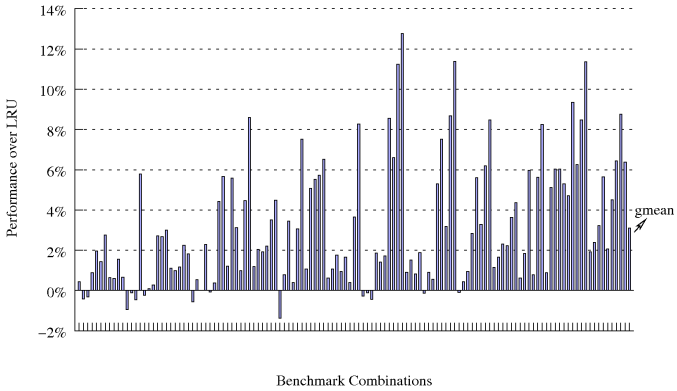


(a) Performance over LRU for a dual core system

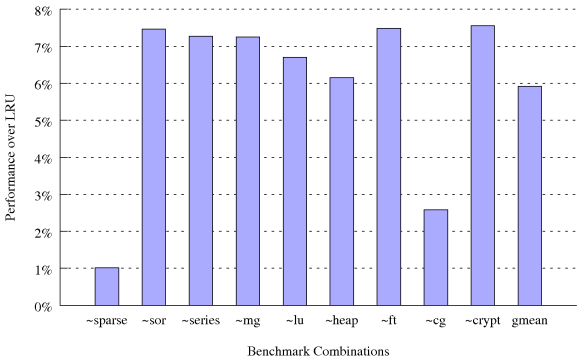


(b) Average cache occupancy for heap and sparse run on a dual core system

Fig. 6. Results for using ABFCP in a dual-core system.



(a) Performance over LRU for a quad core system



(b) Performance over LRU for an eight-core system

Fig. 7. Results for a quad and an eight-core system

6(a) compares the performance of the proposed scheme to the baseline LRU policy for the throughput metric, *IPCSum*. The bar labeled *gmean* represents the geometric mean of the *IPCSum* for the 36 benchmark combinations. The maximum improvement of 20.3% occurs for the *sor-sparse* combination while the maximum degradation of 7.3% occurs for the *lu-sparse* combination. In general the performance degrades by over 1% for only 5 cases. On average, the proposed scheme improves the performance only by 1.41%, as for many combinations ABFCP achieves similar performance to LRU.

To gain a better insight on the effects of using the ABFCP scheme, Fig. 6(b) shows the average number of ways allocated to heap and sor when executed in parallel for both ABFCP and the baseline LRU policy. In the system employing the LRU policy, sor uses more cache ways in average than heap for the first 500 million cycles. At this point however, heap is allocated up to almost 24 out of the 32 available cache ways. As the execution moves forward, sor starts acquiring more and more cache resources until it occupies 22 ways in average. On the other hand, ABFCP is able to identify the different requirements of each application and almost from the start allocates 22 cache ways to sor. This is in accordance with sor's profile shown in Fig. 4 as well as with the observations from the static cache partitioning results presented in Fig. 2.

As it was mentioned before, ABFCP achieves the same performance with baseline LRU for many cases for the dual-core system. We believe that this is due to the cache being big enough to accommodate the working sets of both competing applications. To test this claim the proposed scheme is evaluated for a quad and an eight-core system and the results are presented in Fig. 7 for all the available benchmark combinations (126 and 9 combinations respectively). In Fig. 7(a) it is obvious that the proposed scheme improves the system performance for the majority of the simulated benchmark combinations. The performance is worse than LRU for only 14 out of the 126 combinations and only in one case the degradation is over 1% (-1.4%). The geometric mean is 1.031, which means that the proposed scheme improves the performance of a quad-core system by 3.1% on average. Fig. 7(b) compares ABFCP with baseline LRU for an eight-core system. In this case the performance is improved by 5.92% on average. The x-axis of Fig. 7(b) shows which of the 9 benchmarks was not included in the executed mix of 8 workloads.

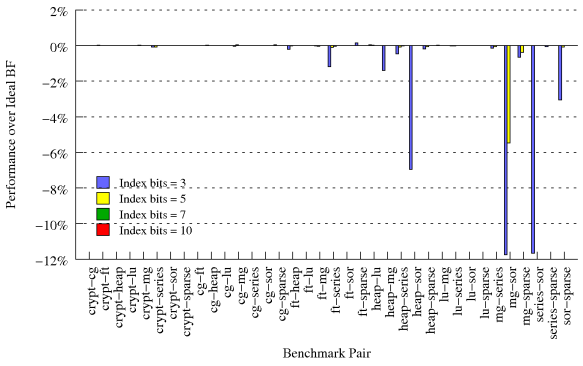


Fig. 8. Effect of different BF array sizes for a dual-core system

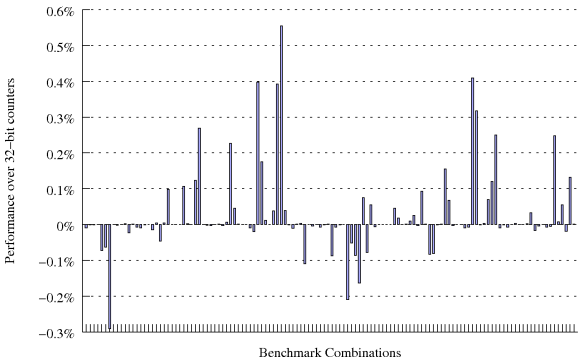


Fig. 9. Comparison of 8 and 32-bit counters in a quad-core system

A. Effect of Varying the Bloom filter Size

The L2 cache used in the previous simulations is 4MB, 32-way associative and each cache block can hold eight words. Assuming a 32-bit physical address space, the length of each tag is 15 bits. All the previous simulations were performed assuming that only the 5 least significant bits of the tag were used to index the Bloom filter arrays. If all the 15 tag bits were used, then the size of each array would be $2^{15} = 32Kb$. This means that for the whole cache the overhead for using the Bloom filters would be:

$$4096\ sets \times 32\ Kb/(sets \times processor) = 16\ MB/processor \quad (4)$$

Of course this overhead renders the implementation of the scheme impossible. However as it was explained in section IV-B, due to their simplicity, the filters' information on far-misses is not expected to be strictly accurate and factor a has been introduced to scale down the value of the $C_{far-miss}$ counters. Therefore, the system should be able to tolerate a few extra filter errors which will be introduced if the BF arrays are shrunk. Fig. 8 compares the performance of a dual core system when 3, 5, 7 and 10 bits are used to index the BF arrays to the case where 'ideal' filters indexed by all 15 bits of the tags are used. Reducing the length of the index from 15 to 10 or 7 bits has no effect on the performance of the system, while a 3-bit index seems to be too small, for a few combinations at least. Based on these results the 5-bit index can be selected, as it causes degradation only for 5 out of the 36 benchmark combinations and only 'noticeable', i.e. greater than 1%, degradation in one case. The size of each BF array is therefore reduced to $2^5 = 32bits$. Consequently, the hardware overhead for the whole cache is reduced to :

$$4096\ sets \times 32\ b/(sets \times processor) = 16\ KB/processor \quad (5)$$

B. Hardware Overhead of ABFCP

Each processor uses one BF array and two counters for each cache set. As the counters are reset every one million cycles, their length can be set to 20 bits. However, all the previous simulations have been performed assuming that it can be reduced further to 8 bits. Fig. 9 shows the effect of using 8-bit instead of 32-bit counters in the ABFCP scheme for a quad-core system. The geometric mean of the results for a quad-core system, shown in Fig. 9, is equal to 1.0002, which means that the reduction of the counters' length has no significant effect on the performance of the cache partitioning scheme. Similar conclusion can be drawn for dual and eight eight-core systems.

TABLE III
STORAGE OVERHEAD PER PROCESSOR

BF arrays (4096 sets * 32bits)	16KB
Counters (4096 sets * 2 counters * 8 bits)	8KB
Total overhead	24 KB
Area of L2 cache (240KB tags + 4MB data)	4336KB
% increase in area	0.55%

In section VI-A the hardware overhead of each Bloom filter was analysed and reduced from $32KB$ to $32b$. The storage overhead per processor for a 4MB, 32-way associative cache, assuming a 32-bit physical address space, is presented in Table III. Each processor requires $24KB$ of storage overhead or 0.55% of the area of the baseline 4MB cache. At the same time, ABFCP requires a processor ID for each cache entry. The length of each ID is $log_2 N$ bits, where N is the number of processors.

For an eight-core system sharing the 4MB, 32-way associative L2 cache, the overhead of the IDs will be $4096 \times 32 \times 3b = 48KB$, an increase of 1.1%. Therefore the total storage overhead for an eight-core system is $8 \times 24 + 48 = 240KB$, an increase of 5.5% over the L2 area. In addition to the storage bits, adders are needed to increment the counters of each processor and the partitioning algorithm requires a comparator circuit. So the true overhead will be slightly greater than indicated here. However this is still proportionately small.

C. Comparison to LRU

To evaluate the effectiveness of ABFCP as a solution to increasing the system performance, it needs to be compared against increasing the cache size, which is the usual approach taken by system designers. Therefore, an eight-core system sharing a 4MB, 32-way associative L2 cache and using the ABFCP scheme was compared to other eight-core systems sharing bigger L2 caches employing the baseline LRU policy.

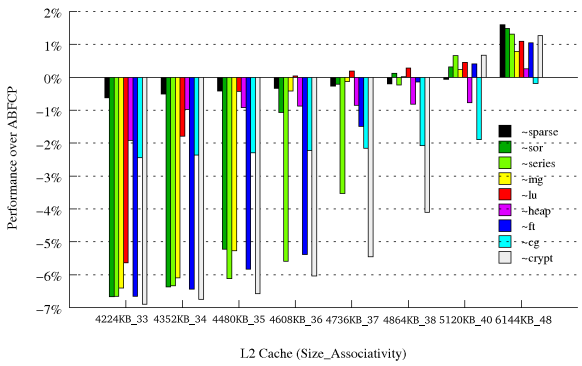


Fig. 10. Comparison of ABFCP against bigger caches employing the LRU policy

The results are presented in Fig. 10.

The L2 cache is increased by adding extra ways. Each way adds $128KB$ of data and $7.5KB$ of tags, an overhead of $135.5KB$. The storage overhead of ABFCP was previously calculated for an eight-core system and found to be $240KB$, slightly less than adding 2 ways to the baseline cache. Fig. 10 shows that a 4352KB, 34-way associative cache using the LRU policy performs worse than the baseline 4MB, 32-way associative cache combined with the ABFCP scheme. The geometric mean for that case is 0.957, which means that the smaller cache with ABFCP performs better on average by 4.3%. Moreover, Fig. 10 reveals that for LRU to perform better than ABFCP for almost all the simulated benchmark combinations, the baseline cache needs to be increased to 6MB, an increase of 50%. And even in that case, the average improvement of LRU over ABFCP is only 0.95%. Therefore, it appears that ABFCP offers a cost effective solution to improving the overall system performance.

VII. Conclusion

Cache design has been extensively studied for uniprocessors and many design choices have migrated to the new multicore architectures. A typical example is the employment of the LRU replacement policy, or approximations thereof, which partitions the cache among competing applications on a demand basis. However, it is possible that this partitioning results in sub-optimal sharing of the cache, as happens when one of the competing processes is a streaming application. This paper proposes *Adaptive Bloom Filter Cache Partitioning* (ABFCP), a novel low cost scheme that identifies the cache requirements of each running application and divides the cache between them attempting to improve the overall system performance. Our evaluation shows that for an eight-core system sharing a 4MB, 32-way associative L2 cache ABFCP outperforms LRU on average by 5.92% while requiring 5.5% storage overhead. Moreover, for LRU to achieve similar performance to ABFCP, the cache has to be increased by around 50%, making ABFCP an attractive solution.

Future work includes a detailed evaluation of ABFCP against other cache partitioning schemes, like UCP and CPARP. Moreover, ABFCP was evaluated here for multicore architectures executing multiprogrammed workloads. Future research will target multithreaded workloads for CMP systems with SMT cores. Additionally, as power is becoming an increasingly significant constraint in computer design, it is necessary to evaluate the effect of the cache partitioning mechanism on the power consumption of the system.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, and others. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[2] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[4] D. Chiou, L. Rudolph, and S. Devadas. Dynamic cache partitioning via columnization. In *Proc. of Design Automation Conference*, Los Angeles, 2000.

[5] H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proc. of 2006 ACM Conference on High Performance Computing*, pages 22–34, 2006.

[6] A. Fedorova. *Operating system scheduling for chip multiprocessor architectures.* PhD thesis, Harvard University, 2006.

[7] M. A. Frumkin, M. Schultz, H. Jin, and J. Yan. Implementation of the NAS Parallel Benchmarks in Java. Technical report, NASA Advanced Supercomputing Division, 10 2002.

[8] M. J. Horsnell. *A chip multi-cluster architecture with locality aware task distribution.* PhD thesis, School of Computer Science, The University of Manchester, 2007.

[9] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proc. of the 18th annual international conference on Supercomputing*, pages 257 – 266, 2004.

[10] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power 5 chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March 2004.

[11] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[12] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, March 2005.

[13] A. Mendelson, J. Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer, and A. Kumar. CMP implementation in systems based on the Intel Core Duo Processor. *Intel Technology Journal*, 10(2):99–107, May 2006.

[14] J. K. Peir, S. C Lai, S. L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proc. of the 16th international conference on Supercomputing*, pages 189–198, New York, New York, USA, 2002.

[15] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high performance runtime mechanism to partition shared caches. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Orlando, Florida, USA, 2006.

[16] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 245–258, 2007.

[17] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Proc. of the 2001 ACM/IEEE conference on Supercomputing*, pages 8–17, Denver, Colorado, 2001.

[18] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28:7–26, 2004.

[19] G. Wright. *A single-chip multiprocessor architecture with hardware thread support.* PhD thesis, Department of Computer Science, The University of Manchester, 2001.

[20] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 339–352, 2007.