Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches

Moinuddin K. Qureshi Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{moin, patt}@hps.utexas.edu

Abstract

This paper investigates the problem of partitioning a shared cache between multiple concurrently executing applications. The commonly used LRU policy implicitly partitions a shared cache on a demand basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand. However, a higher demand for cache resources does not always correlate with a higher performance from additional cache resources. It is beneficial for performance to invest cache resources in the application that benefits more from the cache resources rather than in the application that has more demand for the cache resources.

This paper proposes utility-based cache partitioning (UCP)*, a low-overhead, runtime mechanism that partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. The proposed mechanism monitors each application at runtime using a novel, cost-effective, hardware circuit that requires less than 2kB of storage. The information collected by the monitoring circuits is used by a partitioning algorithm to decide the amount of cache resources allocated to each application. Our evaluation, with 20 multiprogrammed workloads, shows that UCP improves performance of a dual-core system by up to 23% and on average 11% over LRU-based cache partitioning.*

1. Introduction

Modern processors contain multiple cores which enables them to concurrently execute multiple applications (or threads) on a single chip. As the number of cores on a chip increases, the pressure on the memory system to sustain the memory requirements of all the concurrently executing applications (or threads) increases. One of the keys to obtaining high performance from multicore architectures is to manage the largest level on-chip cache efficiently so that off-chip accesses are reduced. This paper investigates the problem of partitioning the shared largest-level on-chip cache among multiple competing applications.

Traditional design for on-chip cache uses the LRU (or an approximation of LRU) policy for replacement decisions. The LRU policy implicitly partitions a shared cache among the competing applications on a demand[1] basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand. However, the benefit (in terms of reduction in misses or improvement in performance) that an application gets from cache resources may not directly correlate with its demand for cache resource. For example, a streaming application can access a large number of unique cache blocks but these blocks are unlikely to be reused again if the working set of the application is greater than the cache size. Although such an application has a high demand, devoting a large amount of cache will not improve its performance. Thus, it makes sense to partition the cache based on how much the application is likely to benefit from the cache rather than the application's demand for the cache.

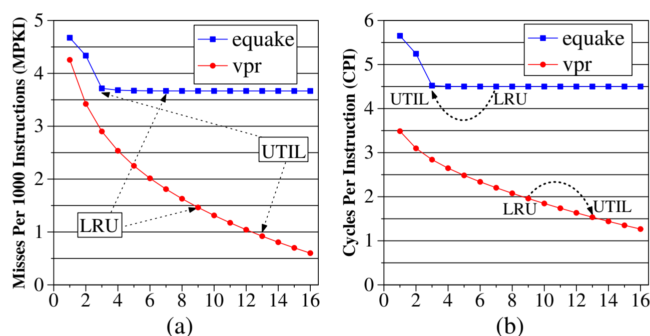

Figure 1. (a) MPKI and (b) CPI as cache size is varied when vpr and equake are executed separately. The horizontal axis shows the number of ways allocated from a 16-way 1MB cache (remaining ways are turned off).

We explain the problem with LRU-based partitioning with a numerical example. Figure 1(a) shows the number of misses for two SPEC benchmarks, vpr and equake, as the cache size is varied, when each one is run separately. We vary the cache size by changing the number of ways and keeping the number of sets constant. The baseline L2 cache in our experiments is 16-way, 1MB in size and contains 1024 sets (other parameters of the experiment are described in Section 4). For vpr, the number of misses reduce monotonically as the cache size is increased from 1 way to 16 ways. For equake, the number of misses decrease as the number of allocated ways increase from 1 to 3, but increasing the cache size by more than 3 ways does not decrease misses. Thus, equake has no benefit or *utility* for cache resources in excess of three ways.

When vpr and equake are run together on a dual-core system, sharing the baseline 1MB 16-way cache, the LRU policy allocates, on average, 7 ways to equake and 9 ways to vpr. If cache partitioning was based on utility (UTIL) of cache resources, then equake would get only 3 ways and vpr would get the remaining 13 ways. Decreasing the cache resources devoted to equake from 7 ways to 3 ways does not increase its misses but increasing the cache re-

[1] Demand is determined by the number of unique cache blocks accessed in a given interval [4]. Consider two applications A and B sharing a fully-associative cache containing N blocks. Then with LRU replacement, the number of cache blocks that each application receives is decided by the number of unique blocks accessed by each application in the last N unique accesses to the cache. If U_A is the number of unique blocks accessed by application A in the last N unique accesses to the cache, then application A will receive U_A cache blocks out of the N blocks in the cache.

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

sources devoted to vpr from 9 ways to 13 ways reduces its misses. As shown in Figure 1(b), partitioning the cache based on utility information can potentially reduce the CPI of vpr from 2 to 1.5 without affecting the CPI of equake, improving the overall performance of the dual-core system.

To partition the cache based on application's utility for the cache resource, we propose *Utility-Based Cache Partitioning (UCP)*. An important component of UCP is the monitoring circuits that can obtain the information about utility of cache resource for all the competing applications at runtime. For the UCP scheme to be practical, it is important that the utility monitoring (UMON) circuits are not hardware-intensive or power-hungry. Section 3 describes a novel, low-overhead, UMON circuit that requires a storage overhead of only 1920B (less than 0.2% area of the baseline 1MB cache). The information collected by UMON is used by a partitioning algorithm to decide the amount of cache allocated to each competing application. Our evaluation in Section 5 shows that UCP outperforms LRU, improving the performance of a dual-core system by up to 23%, and on average 11%.

The number of possible partitions increases exponentially with the number of applications sharing the cache. It becomes impractical for the partitioning algorithm to find the best partition by evaluating every possible partition, when a large number of applications share a highly associative cache. In Section 6, we propose the *Lookahead Algorithm* as a scalable alternative to evaluating all the possible partitions for partitioning decisions.

2. Motivation and Background

Caches improve performance by reducing the number of main memory accesses. Thus, the utility of cache resources for an application can be directly correlated to the change in the number of misses or improvement in performance of the application when the cache size is varied. Figure 2 shows the misses and CPI for some of the SPEC benchmarks.

The utility of cache resource varies widely across applications. The 15 applications are classified into three categories based on how much each of them benefits as the cache size is increased from 1 way to 16 ways (keeping the number of sets constant). The first row contains benchmarks that do not benefit significantly as the cache size is increased from 1 way to 16 ways. We say such applications have *low* utility. These benchmarks either have a large number of compulsory misses (e.g. gap) or have a data set larger than the cache size[2] (e.g. mcf). Benchmarks in the second row continue to benefit as the cache size is increased from 1 way to 16 ways. We say such applications have *high* utility. Benchmarks in the third row benefit significantly as the cache size is increased from 1 way to 8 ways. These benchmarks have a small working set that fits in a small size cache, therefore, giving them more than 8 ways does not significantly improve their performance. We say such applications have *saturating* utility.

If two applications having low utility (e.g. mcf and applu) are executed together, then their performance is not sensitive to the amount of cache available to each application. Similarly, when two applications of saturating utility are executed together, then the cache can support the working set of both applications. However, when an application with saturating utility is run with an application with low utility then the cache may not hold the working set of the application with saturating utility. Similarly, when an application with high utility is run with any other application, its performance is highly sensitive to the amount of cache available to it. In such cases, it is important to partition the cache judiciously by taking utility information into account.

Figure 2 shows that in most cases,[3] reduction in misses correlates with reduction in CPI. Thus, we can use the information about reduction in misses to make cache partitioning decisions. To include utility information in partitioning decisions, we provide a quantitative definition of utility for cache resources for a given application. Since cache is allocated only on a way basis in our studies, we define utility on a way granularity. If $miss_a$ and $miss_b$ are the number of misses that an application incurs when it receives a and b ways respectively ($a < b$), then the utility (U_a^b) of increasing the number of ways from a to b is:

$$U_a^b = miss_a - miss_b \tag{1}$$

Section 3 describes cost-effective monitoring circuits that can estimate the utility (U) information for an application at run-time, along with the framework, the partitioning algorithm, and the replacement scheme for UCP.

3. Utility-Based Cache Partitioning

3.1. Framework

Figure 3 shows the framework to support UCP between two applications that execute together on a dual-core system. One of the two applications execute on CORE1 and the other on CORE2. Each core is assigned a utility monitoring (UMON) circuit that tracks the utility information of the application executing on it. The UMON circuit is separated from the shared cache, which allows the UMON circuit to obtain utility information about an application for all the ways in the cache, independent of the contention from the application executing on the other core. The partitioning algorithm uses the information collected by the UMON to decide the number of ways to allocate to each core. The replacement engine of the shared cache is augmented to support the partitions allocated by the partitioning algorithm.

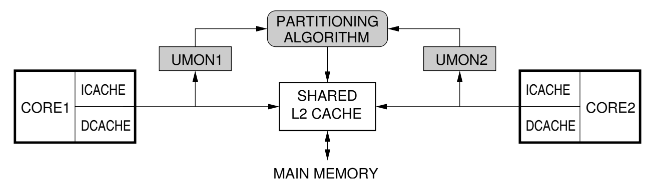

Figure 3. Framework for Utility-Based Cache Partitioning. Newly added structures are shaded. (Figure not to scale)

[2] Applications with low utility can show a large reduction in misses when the cache size is increased such that the dataset fits in the cache. For example, Figure 11 shows that the MPKI of art does not decrease when the cache size is increased from 1 way to 8 ways (0.5MB). However, increasing the size to 24 ways (1.5MB) reduces MPKI by a factor of 5. In such cases, the curve of MPKI vs. cache size resembles a step function.

[3] When eight ways are allocated to swim, it sees a huge reduction in misses. However, this reduction in misses does not translate into a substantial reduction in CPI. This happens because a set of accesses with high memory-level parallelism (MLP) now fits in the cache which reduces the average MLP and increases the average mlp-based cost[12] of each miss.

IEEE COMPUTER SOCIETY

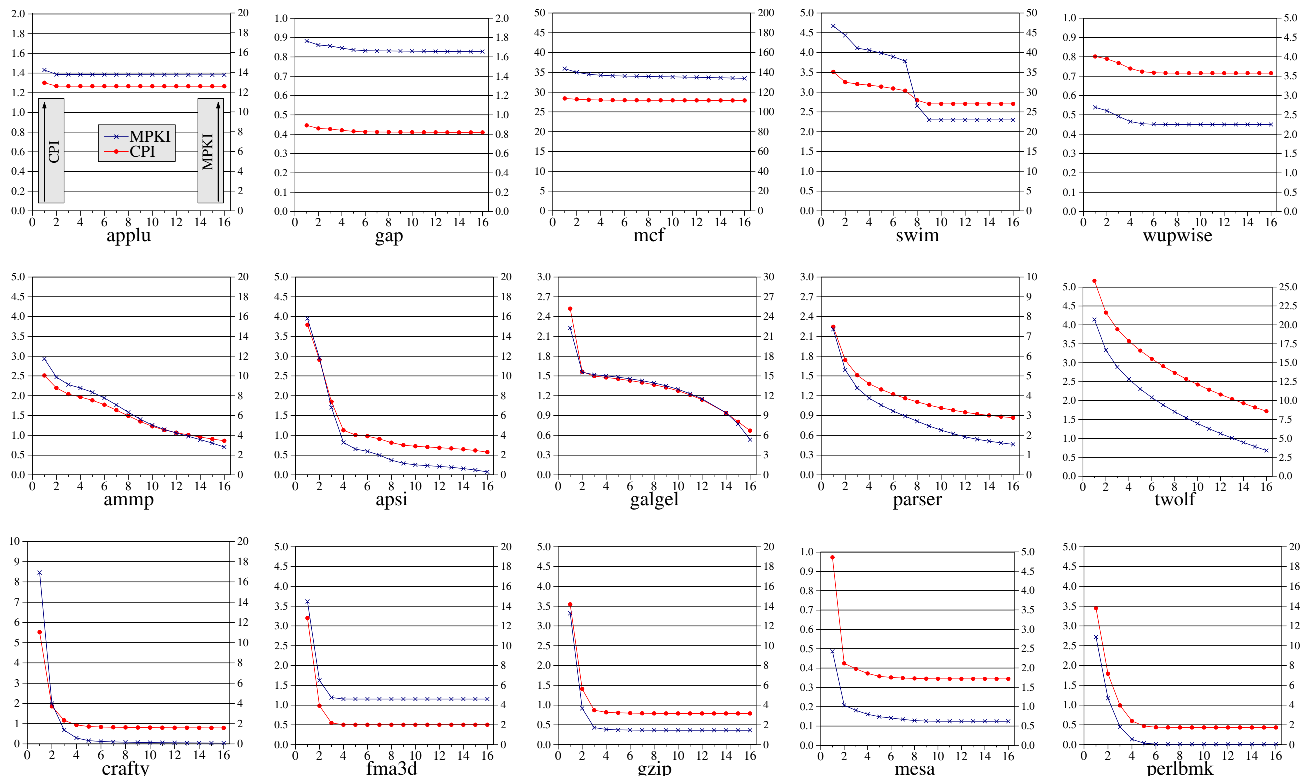

Figure 2. MPKI and CPI for SPEC benchmarks as the cache size is varied. The horizontal axis shows the number of ways allocated from a 16-way 1MB cache (the remaining ways are turned off). The graphs for vpr and equake are shown in Figure 1, and the graph for art is shown in Figure 11.

3.2. Utility Monitors (UMON)

Monitoring the utility information of an application requires a mechanism that tracks the number of misses for all possible number of ways. To compute the utility information for the baseline 16-way cache, the monitoring circuit is required to track misses for all the sixteen cases, ranging from when only 1 way is allocated to the application to when all 16 ways are allocated to the application. A straight-forward, but expensive, method to obtain this information is to have sixteen tag directories, each having the same number of sets as the shared cache, but each having a different number of ways ranging from 1 way to 16 ways (note that data lines are not required to estimate hit-miss information). Although this scheme can track utility information for any replacement scheme implemented in the shared cache, the hardware overhead of multiple directories makes this scheme impractical. Fortunately, the baseline LRU policy obeys the stack property [10], which means that an access that hits in a LRU managed cache containing N ways is guaranteed to also hit if the cache had more than N ways (the number of sets being constant). This means even with a single tag directory containing sixteen ways, it is possible to compute the hit-miss information about all the cases when the cache contains from one way through sixteen ways. To see how the stack algorithm provides utility information, consider the example of a four way set-associative cache shown in Figure 4(a).

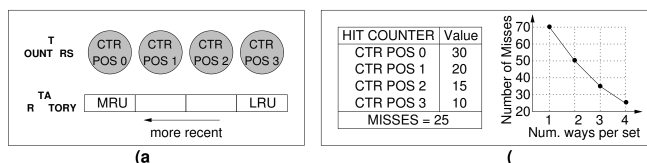

Figure 4. (a) Hit counters for each recency position. (b) Example of how utility information can be tracked with stack property.

Each set has four counters for obtaining the hit counts for each of the four recency positions ranging from MRU to LRU. The position next to MRU in the recency position is referred as *position 1* and the next position as *position 2*. If a cache access results in a hit, the counter corresponding to the hit-causing recency position is incremented. The counters then represent the number of misses saved by each recency position. Figure 4(b) shows an example in which out of the 100 accesses to the cache, 25 miss, 30 hit in MRU, 20 hit in position 1, 15 hit in position 2, and the remaining 10 hit in the LRU position. Then, if the cache size is reduced from four ways to three ways, the misses increase from 25 to 35. Further reducing the cache size to two ways, increases the number of misses to 50. And with only one way the cache incurs 70 misses. Thus, given information about misses in a cache that has a large number of ways, it is possible to obtain the information about misses for a cache with smaller number of ways.

The UMON circuit tracks the utility of each way using an Auxiliary Tag Directory (ATD) and hit counters. The ATD has the same associativity as the main tag directory of the shared cache and uses the LRU policy for replacement decisions. Figure 5 (a) shows a UMON that contains the hit counters for each set in the cache. We call this organization as *UMON-local*. Although, UMON-local can perform partitioning on a per-set basis, it requires a huge overhead because of the extra tag entry and hit counter for each line in the cache. The hardware overhead of the hit counters in UMON-local can be reduced by having one group of hit counters for all the sets in the cache. This configuration, shown in Figure 5(b), is called *UMON-global*. UMON-global enforces a uniform partition for all the sets in the cache. Compared to UMON-local, UMON-global reduces the number of hit counters required to implement UMON by a factor of number of sets in the cache. However, the number of tag entries required to implement UMON still remains equal to the number of lines in the cache.

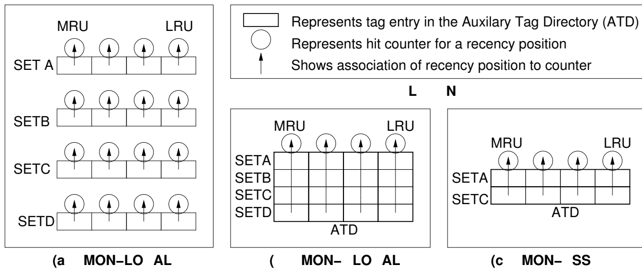

Figure 5. (a) UMON-local (b) UMON-global (c) UMON implemented with dynamic set sampling (DSS).

3.3. Reducing Storage Overhead Using DSS

The number of UMON circuits in the system is equal to the number of cores. For the UMON circuit to be practical, it is important that it requires low hardware overhead. UMON-global requires an extra tag entry for each line in the cache. If each tag entry is 4 bytes then the UMON overhead per cache line is 8 bytes for a two-core system and 16 bytes for a four-core system. Considering that the baseline cache is 64 byte in size, the overhead of UMON-global is still substantial. To reduce the overhead of UMON, we use *Dynamic Set Sampling (DSS)* [12]. The key idea behind DSS is that the behavior of the cache can be approximated by sampling only a few sets. We can use DSS to approximate the hit counter information of UMON-global by sampling few sets in the cache. Figure 5(c) shows the UMON circuit with Dynamic Set Sampling (UMON-DSS). The ATD in UMON-DSS contains ATD entries only for two sets A and C instead of all the four sets in the cache. An important question is that how many sampled sets are required for UMON-DSS to approximate the performance of UMON-global? We derive analytical bounds[4] for UMON-DSS in the next section and in Section 5.4, we compare the performance of UMON-DSS with UMON-global.

[4]DSS was used in [12] to choose between two replacement policies. Thus, it was used to approximated a global decision which had a binary value (one of the two replacement policy) by using the binary decisions obtained on the sampled sets. We are interested in approximating the global partitioning decision which is a discrete value (how many ways to allocate) by using the hit counter information of the sampled sets. Therefore the bounds derived in [12] are not applicable to our mechanism.

3.4. Analytical Model for Dynamic Set Sampling

Let there be two applications A and B competing for a cache containing S sets. Let $a(i)$ denote the number of ways that application A receives for a given set i, if the partitioning is done on a per-set basis. Then if $a(i)$ does not vary across sets then even with a single set UMON-DSS can approximate UMON-global. However, $a(i)$ may vary across sets. The number of ways allocated to application A by UMON-global (u_g) can be approximated as the average of all $a(i)$, assuming that UMON-global gives equal importance to all the sets. Thus,

$$u_g = \sum_{i=1}^{S} a(i)/S \quad (2)$$

Let n be the number of randomly selected sets sampled by UMON-DSS. Let u_s be the number of ways allocated to application A by UMON-DSS. We are interested in bounding the value of $|u_s - u_g|$ to some threshold ϵ. If σ^2 is the variance in the values of $a(i)$ across all the sets, then by Chebyshev's inequality [15]:

$$P(|u_s - u_g| \geq \epsilon) \leq \sigma^2/(n \cdot \epsilon^2) \quad (3)$$

For bounding u_s to within one way of u_g, $\epsilon = 1$.

$$P(u_s \textit{ is at least one way from } u_g) \leq \sigma^2/n \quad (4)$$

$$P(u_s \textit{ is within one way from } u_g) > 1 - (\sigma^2/n) \quad (5)$$

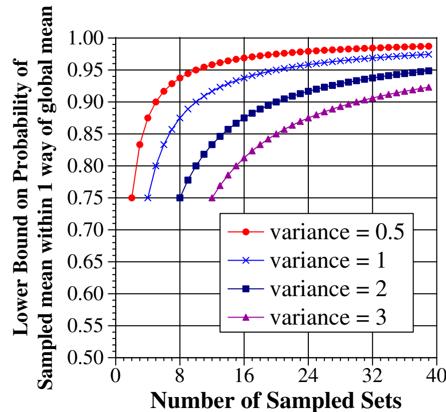

Figure 6. Bounds on Number of Sampled Sets

As Chebyshev's inequality considers only variance without making any assumption about the distribution of the data, the bounds obtained from Chebyshev's inequality are pessimistic[5][15]. Figure 6 shows the lower bound provided by Chebyshev's inequality as the number of sampled sets is varied, for different values of variance. For most of the workloads studied, the value of variance (σ^2) is less than 3, indicating that even with the pessimistic bounds, as few as 32 sets are sufficient for UMON-DSS to approximate UMON-global. We compare UMON-DSS to UMON-global in Section 5.4. Unless stated otherwise, we use 32 sets for UMON-DSS. The sampled sets for UMON-DSS are chosen using the simple static policy [12], which means set 0 and every 33rd set is selected. For the remainder of the paper UMON by default means UMON-DSS.

[5]In general, much tighter bounds can be obtained if the mean and the distribution of the sampled data are known [15].

IEEE COMPUTER SOCIETY

3.5. Partitioning Algorithm

The partitioning algorithm reads the hit counters from all the UMON circuits of each of the competing applications. The partitioning algorithm tries to minimize the total number of misses incurred by all the applications. The utility information in the hit counters directly correlates with the reduction in misses for a given application when given a fixed number of ways. Thus, reducing the most number of misses is equivalent to maximizing the combined utility. If A and B are two applications with utility functions *UA* and *UB* respectively, then for partitioning decisions, the combined utility (U_{tot}) of A and B is computed for all possible partitions for the baseline 16-way cache:

$$U_{tot}(a) = UA_1^i + UB_1^{(16-i)} \;\; ... \, For \; i = \; 1 \; to \, (16-1) \quad (6)$$

The partition that gives the maximum value for U_{tot} is selected. In our studies, we guarantee that the partitioning algorithm gives at least one way to each application. We invoke the partitioning algorithm once every five million cycles (a design choice based on simulation results). After each partitioning interval, the hit counters in all UMONs are halved. This allows the UMON to retain past information while giving importance to recent information.

3.6. Changes to Replacement Policy

To incorporate the decisions made by the partitioning algorithm, the baseline LRU policy is augmented to enable way partitioning [2][18][6]. To implement way partitioning, we add a bit to the tag-store entry of each block to identify the core which installed the block in the cache. On a cache miss, the replacement engine counts the number of cache blocks that belong to the miss-causing application in the set. If this number is less than the number of blocks allocated to the application, then the LRU block among all the blocks that do not belong to the application is evicted. Otherwise, the LRU block among all the blocks of the miss-causing application is evicted.

If the number of ways allocated to an application is increased by the partitioning algorithm, then these added ways are consumed by the application only on cache misses. This gradual change of partitions allows the cache to retain the cache blocks till they are required by the application that is allocated the cache space.

4. Experimental Methodology

4.1. Configuration

Table 1 shows the parameters of the baseline configuration used in our experiments. We use an in-house simulator that models the alpha ISA. The processor core is 8-wide issue, out-of-order, with 128-entry reservation station. The first-level instruction cache and data cache are private to the processor core. The processor parameters are kept constant in our study. This allows us to use a fast event-driven processor model to reduce simulation time. Because our study deals with the memory system we model the memory system in detail. DRAM bank conflicts and bus queuing delays are modeled. The baseline L2 cache is shared among all the processor cores and uses LRU replacement. Thus, the L2 cache gets partitioned among all the competing cores on a demand basis.

Table 1. Base configuration.

Component	Configuration
Processor core	8 wide, out-of-order, with 128 entry reservation station; 64 kB hybrid branch predictor with 4k-entry BTB minimum branch misprediction penalty of 15 cycles.
	L1 Icache and Dcache :16kB, 64B line-size, 4-way, LRU. The L1 caches are private to each core.
Unified Shared L2 Cache	1MB, 64B line-size, 16-way with LRU replacement, 15-cycle hit, 32-entry MSHR, 128-entry store buffer. L2 cache is shared among all the cores
Memory	32 DRAM banks; 400-cycle access latency; bank conflicts modeled; maximum 32 outstanding requests
Bus	16B-wide split-transaction bus at 4:1 frequency ratio. queueing delays modeled

4.2. Metrics

There are several metrics to quantify the performance of a system in which multiple applications execute concurrently. We discuss the three metrics commonly used in the literature: weighted speedup, sum of IPCs, and harmonic mean of normalized IPCs. Let IPC_i be the IPC of the ith application when it concurrently executes with other applications and $SingleIPC_i$ be the IPC of the same application when it executes in isolation. Then, for a system in which N threads execute concurrently, the three metrics are given by:

$$Weighted\,Speedup = \sum (IPC_i / SingleIPC_i) \quad (7)$$

$$IPC_{sum} = \sum IPC_i \quad (8)$$

$$IPC_{norm_hmean} = N / \sum (SingleIPC_i / IPC_i) \quad (9)$$

The *Weighted Speedup* metric indicates reduction in execution time. The IPC_{sum} metric indicates the throughput of the system but it can be unfair to a low IPC application. The IPC_{norm_hmean} metric balances both fairness and performance [9]. We will use *Weighted Speedup* as the metric for quantifying the performance of multicore configurations throughout the paper. Evaluation with the IPC_{sum} and IPC_{norm_hmean} metric will also be discussed for some of the key results in the paper.

4.3. Benchmarks

We use benchmarks from the SPEC CPU2000 suite for our studies. A representative slice of 250M instructions is obtained for each benchmark using a tool that we developed using the Simpoint methodology [11]. Two separate benchmarks are combined to form one multiprogrammed workload that can be run on a dual-core system. To include a wide variety of multiprogrammed workload in our study, we classify the multiprogrammed workloads into five categories. Workloads with *Weighted Speedup* for the baseline configuration between 1 and 1.2 are classified as *Type A*, between 1.2 and 1.4 as *Type B*, between 1.4 and 1.6 as *Type C*, between 1.6 and 1.8 as *Type D*, and between 1.8 and 2 as *Type E*. A suite containing 20 workloads is created by using four workloads from each of the five categories.

Simulation for a dual-core system is continued until both benchmarks in the multiprogrammed workload execute at least 250M instructions each. If a benchmark finishes the stipulated 250M instruction before the other benchmark finishes 250M instruction, it is restarted so that the two benchmarks continue

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

to compete for the L2 cache throughout the simulation. Table 2 shows the classification based on baseline weighted speedup (BaseWS), Misses Per 1000 Instruction (MPKI) and Cycles Per Instruction (CPI) for the baseline dual-core configuration for all the 20 workloads. The benchmark names for ammp (amp), swim (swm), perlbmk (perl), and wupwise (wup) are abbreviated.

Table 2. Workload Summary

Category (BaseWS)	Workload Bmk1-Bmk2	MPKI Bmk1	MPKI Bmk2	CPI Bmk1	CPI Bmk2
TYPE A (1.0-1.2)	galgel-vpr	11.84	8.41	1.25	2.55
	galgel-twolf	11.46	11.44	1.20	3.51
	amp-galgel	6.91	10.62	1.74	1.21
	apsi-galgel	3.08	10.82	1.14	1.19
TYPE B (1.2-1.4)	twolf-vpr	8.76	6.22	2.81	2.06
	apsi-twolf	2.05	7.51	0.93	2.61
	amp-art	6.73	43.73	1.72	4.90
	apsi-art	2.91	43.12	1.12	4.76
TYPE C (1.4-1.6)	apsi-swm	2.71	22.98	1.05	2.89
	amp-applu	6.71	13.76	1.69	1.28
	swm-twolf	22.98	10.64	2.73	3.26
	art-parser	42.75	3.48	4.52	1.33
TYPE D (1.6-1.8)	equake-vpr	18.33	5.74	4.57	1.97
	vpr-wup	5.40	2.25	1.89	0.72
	gzip-twolf	1.61	5.36	0.84	2.17
	art-crafty	41.10	0.63	4.33	0.96
TYPE E (1.8-2.0)	fma3d-swm	4.62	23.53	0.51	2.94
	mcf-applu	134	13.76	28.5	1.27
	gap-mesa	1.66	0.62	0.41	0.35
	crafty-perl	0.14	0.04	0.81	0.44

5. Results and Analysis

5.1. Performance on Weighted Speedup Metric

We compare the performance of UCP to two partitioning schemes: LRU and Half-and-Half. The Half-and-Half scheme statically partitions the cache equally among the two competing applications. The disadvantage of the Half-and-Half scheme over LRU is that it cannot change the partition in response to the varying demands of competing applications. However, it also has the advantage of performance isolation, which means that the performance of an application does not degrade substantially when it executes concurrently with a badly behaving application. Figure 7 shows the weighted speedup of the three partitioning schemes. The bar labeled *gmean* represents the geometric mean of the individual weighted-speedup of all the 20 workloads.

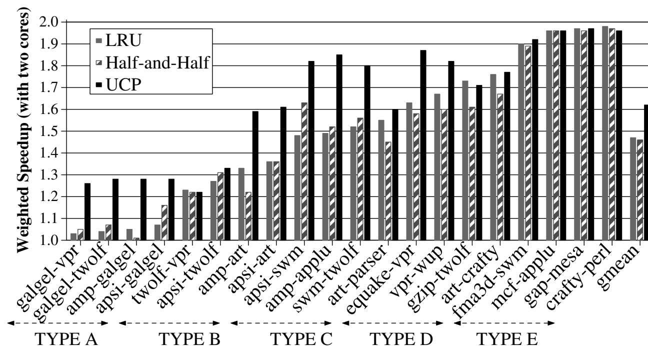

Figure 7. Performance of LRU, Half-and-Half, and UCP.

LRU performs better than Half-and-Half for some workloads and for some Half-and-Half performs better than LRU. For most workloads, UCP outperforms the best-performing scheme out of the other two schemes. On average, UCP improves performance by 10.96% over the baseline LRU policy, increasing the geometric mean weighted speedup from 1.46 to 1.62.

The Type A category contains workloads where both benchmarks in the workload have high utility and high demand for the L2 cache. Therefore, the baseline LRU policy has a value of weighed speedup that is almost half of the ideal value of 2. Partitioning the cache based on utility, rather than demand, improves performance noticeably. For example, UCP increases the weighted speedup for the workload galgel-twolf from 1.04 to 1.28.

The Type C category contains workloads where one benchmark has high utility and the other has low utility. In such cases, UCP allocates most of the cache resource to the application with high utility, thus improving the overall performance. For example, for amp-applu, UCP allocates 14 or more ways out of the 16 ways to amp, improving the weighted speedup from 1.49 to 1.83.

When both benchmarks in the workload have low utility (e.g. mcf-applu), the performance of each benchmark in the workload is not sensitive to the amount of cache available to it, so the weighted speedup is close to ideal. Similarly, if the cache can accommodate the working set of both benchmarks in the workload, the weighted speedup for that workload is close to ideal. Such workloads are included in the Type E category. As the weighted speedup of these workloads is close to the ideal, UCP does not change performance significantly.

For twolf-vpr, crafty-perl, and gzip-twolf , UCP reduces performance marginally compared to LRU. This happens because UCP allocates partitions once every partition interval (5M cycles in our experiments), so it is unable to respond to the phase changes that occur at a finer granularity. On the other hand, LRU can respond to such fine-grained change in behavior of the applications by changing the partitions potentially at every access. The LRU policy also has the advantage of doing the partitioning on a per-set basis depending on the demand on the individual set. On the other hand, the proposed UCP policy globally allocates a uniform partition for all the sets in the cache, sacrificing fine-grained, per-set, control for reduced overhead.

5.2. Performance on Throughput Metric

Figure 8 compares the performance of the baseline LRU policy to the proposed UCP policy for the throughput metric, $IPCsum$. To show the change in the IPC of the individual benchmark of each workload, the graph is drawn as a stacked bar graph. The IPC of the first benchmark that appears in the name of the workload is labeled as *IPC-Benchmark1*. The IPC of the other benchmark is labeled as *IPC-Benchmark2*. For example, for the workload galgel-vpr, IPC-Benchmark1 shows the IPC of galgel and IPC-Benchmark2 shows the IPC of vpr. The bar labeled *hmean* represents the harmonic mean of the IPC_{sum} of all the 20 workloads.

For 15 out of the 20 workloads, UCP improves the IPC_{sum} compared to the LRU policy. UCP can improve performance by improving the IPC of one benchmark in the workload without affecting the IPC of the other benchmark in the workload. Examples of such workloads are apsi-swm and equake-vpr. UCP can also improve the aggregate IPC by marginally reducing the IPC of

0-7695-2732-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

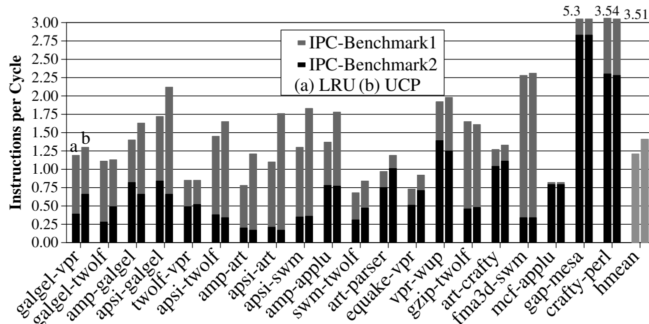

Figure 8. LRU (left bar) vs. UCP (right bar) on throughput metric.

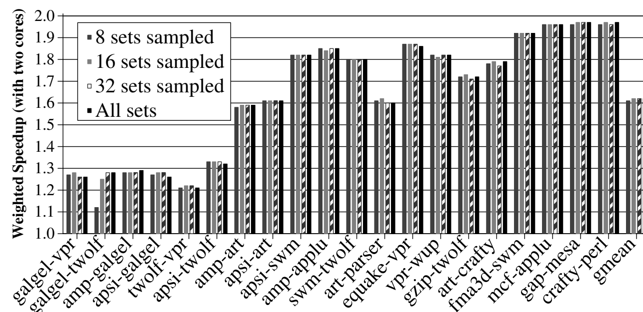

Figure 10. Effect of Number of Sampled Sets on UCP.

one benchmark and significantly improving the IPC for the other benchmark. Examples include apsi-galgel and amp-art. For the IPC_{sum} metric, UCP reduces performance on two workloads, gzip-twolf and crafty-perl. On average, UCP improves the performance on the throughput metric by 16.8%, increasing the harmonic mean IPC_{sum} of the system from 1.21 to 1.41.

5.3. Evaluation on Fairness Metric

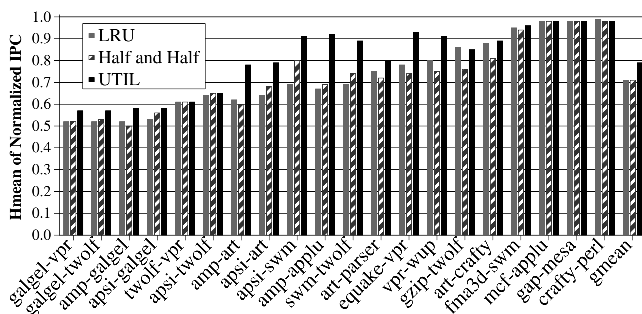

Figure 9. LRU, Half-and-Half, and UCP on fairness metric.

A dynamic partitioning mechanism may improve the overall performance of the system at the expense of severely degrading the performance of one of the applications. The harmonic mean of the normalized IPCs is shown to consider both fairness and performance [9]. Figure 9 shows the performance of LRU, Half-and-Half, and UCP for this metric. The bar labeled gmean is the geometric mean over all the 20 workloads. UCP improves the average on this metric by 11% increasing the gmean from 0.71 to 0.79. Note that more improvement in this metric can be obtained by modifying the partitioning algorithm to directly favor fairness.

5.4. Effect of Varying the Number of Sampled Sets

We use 32 sets for each of the UMON circuit in the default UCP configuration. This section analyzes the sensitivity of the UCP mechanism to the number of sampled sets in the UMON. Figure 10 compares the performance of four UCP configurations: the first samples 8 sets, the second samples 16 sets, the third is the default UCP configuration with 32 sampled sets, and the last is the UMON-global configuration which contains all the sets.

For all workloads, the default UCP configuration with 32 sampled sets performs similar to UMON-global (All sets). The performance of the workload galgel-twolf reduces if the number of sampled sets is reduced to 8. For other workloads, the performance of UCP is relatively insensitive to the number of sampled sets (for sampled sets ≥ 8). This is consistent with the lower bounds derived with the analytical model presented in Section 3.4. This result is particularly useful result as it means that default UCP configuration with only 32 sets performs similar to the UMON-global configuration without requiring the huge hardware overhead associated with the UMON-global configuration. This reduced overhead makes the UCP scheme practical. The next section quantifies the hardware overhead of UCP.

5.5. Hardware Overhead of UCP

The major source of hardware overhead of UCP is the UMON circuit. Table 3 details the storage overhead of UMON containing 32 sampled sets, assuming a 40-bit physical address space. Each UMON requires 1920 B of storage overhead (less than 0.2% of the area of the baseline 1MB cache), indicating that for the baseline dual-core configuration UCP requires less than 0.4% of storage overhead for implementing the UMON circuits. The low overhead for UMON means that the UCP scheme is cost-effective even if the number of core increases (e.g. UMON overhead of less than 1% with four cores). The storage overhead of UMON can further be reduced by using partial tags in the ATD. In addition to the storage bits, each UMON also contains an adder for incrementing the hit counters and a shifter to halve the hit counters after each partitioning interval.

Table 3. Storage Overhead of a UMON circuit with 32 Sets

Size of each ATD entry (1 valid bit + 24-bit tag + 4-bit LRU)	29 bits
Total number of ATD entries per sampled set (1/way * 16)	16
ATD overhead per sampled set (29 bits/way * 16 ways)	58 B
Total ATD overhead (32 sampled sets * 58 B/set)	1856 B
Overhead of hit counters (16 counters * 4B each)	64 B
Total UMON overhead (1856B + 64B)	1920 B
Area of baseline L2 cache (64kB tags + 1MB data)	1088 kB
% increase in L2 area due to 1 UMON (1920B/1088kB)	**0.17%**

Implementing way-partitioning on a dual-core system requires a bit in each tag-store entry to identify which of the two cores installed the line in the cache. The partitioning algorithm contains a comparator circuit and requires negligible storage. Note that none of the structures or operations required by UCP is in the critical path, resource-intensive, complex, or power hungry.

0-7695-2732-9/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

6. Scalable Partitioning Algorithm

We assumed that the partitioning algorithm is able to find the partition of maximum utility by computing the combined utility of all the applications for every possible partition. This is not a problem when there are only two applications, as an *N-way* cache can be way-partitioned among two applications in only *N+1* ways. However, the number of possible partitions increases exponentially as the number of competing applications, making it impractical to evaluate every possible partition. For example, a 32-way cache can be shared by four applications in $6,545$ ways, and by 8 applications in $15,380,937$ ways. Finding an optimal solution to the partitioning problem has been shown to be NP-hard [14]. In this section we develop a partitioning algorithm that has a worst-case time complexity of $N^2/2$.

6.1. Background

Our algorithm is derived from the greedy algorithm proposed in [16]. The *greedy algorithm* is shown in Algorithm 1.

Algorithm 1 Greedy Algorithm

```
balance = N          /* Num blocks to be allocated */
allocations[i] = 0 for each competing application i
    while(balance) do:
        foreach application i, do: /* get utility for next 1 block */
            alloc = allocations[i]
            Unext[i] = get_util_value(i, alloc, alloc+1)
        winner = application with maximum value of Unext
        allocations[winner]++
        balance = balance-1
return allocations

get_util_value(p, a, b):
    U = change in misses for application p when the number
    of blocks assigned to it increases from a to b
    return U
```

In each iteration, one block[6] is assigned to the application that has the maximum utility for that block. The iteration continues till all the blocks are assigned. This algorithm is shown to be optimal if the utility curves for all the competing applications are convex [16]. However, when the utility curves are non-convex, the greedy algorithm can have pathological behavior. Figure 11 shows example of two benchmarks, art and galgel, that has non-convex utility curve. Art shows no reduction in misses until it is assigned at least 8 blocks and after that it shows huge reduction in misses. As the greedy algorithm considers the gain from only the immediate one block it will not assign any blocks to art (unless the utility of that block for even the other application is zero). To address this shortcoming of the greedy algorithm, Suh et. al [18] propose to also invoke the greedy algorithm for each combination of the non-convex points of all applications. However, the number of times the greedy algorithm is invoked increases with the number of combinations on non-convex points of all the applications. Figure 11 shows that an application (galgel) can have as many as

[6] We use the term blocks instead of ways because the greedy algorithm was used in [16] to decide the number of cache blocks that each application receives in a fully associative cache. However, the explanation can also be thought of as assigning ways in a set-associative cache.

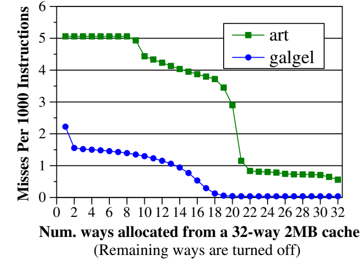

Figure 11. Benchmarks with non-convex utility curves

15 non convex points, indicating that the number of combinations of non-convex points of all the competing applications can be very large. To avoid the time complexity, [18] suggests that the greedy algorithm be invoked only for some number of randomly chosen combination of non-convex points. However, for a given number of trials, the likelihood that randomization will yield the optimum partition reduces as the number of combinations increase.

6.2. The Lookahead Algorithm

We define *marginal utility (MU)* as the utility U per unit cache resource. If $miss_a$ and $miss_b$ are the number of misses that an application incurs when it receives a and b blocks respectively, then the marginal utility, MU_a^b of increasing the blocks from a to b is defined as:

$$MU_a^b = (miss_a - miss_b)/(b-a) \; = \; U_a^b/(b-a) \qquad (10)$$

The basic problem with the greedy algorithm is that it considers the marginal utility of only the immediate block, and thus fails to see potentially high gains after the first block if there is no gain from the first block. If the algorithm could also take into account the gains from far ahead, then it could make better partitioning decisions. We propose the *Lookahead Algorithm*, which considers the marginal utility for all possible number of blocks that the application can receive. The pseudo code for the Lookahead algorithm is shown in Algorithm 2.

Algorithm 2 Lookahead Algorithm

```
balance = N          /* Num blocks to be allocated */
allocations[i] = 0 for each competing application i
    while(balance) do:
        foreach application i, do: /* get max marginal utility */
            alloc = allocations[i]
            max_mu[i] = get_max_mu(i, alloc, balance)
            blocks_req[i] = min blocks to get max_mu[i] for i
        winner = application with maximum value of max_mu
        allocations[winner] += blocks_req[winner]
        balance -= blocks_req[winner]
return allocations

get_max_mu(p, alloc, balance):
    max_mu = 0
    for(ii=1; ii<=balance; ii++) do:
        mu = get_mu_value(p, alloc, alloc+ii)
        if( mu > max_mu ) max_mu = mu
    return max_mu

get_mu_value(p, a, b):
    U = change in misses for application p when the number
    of blocks assigned to it increases from a to b
    return U/(b-a)
```


0-7695-2732-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

In each iteration, the maximum marginal utility *(max_mu)* and the minimum number of blocks at which the max_mu occurs is calculated for each application. The application with highest value for max_mu is assigned the number of blocks it needs to obtain max_mu. Ties for highest value of max_mu are broken arbitrarily. The iterations are repeated until all blocks are assigned. The lookahead algorithm can assign a different number of blocks in each iteration and is guaranteed to terminate as at least one block is assigned in each iteration. For applications with convex utility function, the maximum value of marginal utility occurs for the first block. Therefore, if all the applications have convex utility function, then the lookahead algorithm behaves identical to the greedy algorithm, which is proved to be optimal for convex functions.

The step for obtaining the value of max_mu for each of the application is executed in parallel by the UMON circuits. Calculating the max_mu for an application if it could get up to N blocks takes N operations of add-divide-compare each. As the blocks are allocated, the number of blocks that an application can receive in an iteration reduces. In the worst case only one block is allocated in every iteration. Then, even in the worst case, the time required for the lookahead algorithm to allocate N blocks is: $N + (N-1) + (N-2) + ... + 1 = N(N-1)/2 \approx N^2/2$ operations. In our studies, cache is assigned on a way granularity instead of a block granularity. Therefore, the value of N is equal to the associativity of cache. Thus, for partitioning a 32-way cache the lookahead algorithm will require a maximum time of 512 operations (recall that we perform partitioning once every 5M cycles). In our experiments, we ensure that both the greedy algorithm and the lookahead algorithm allocates at least one way to each of the competing applications.

6.3. Result for Partitioning Algorithms

We evaluate the partitioning algorithms on a quad-core system in which four applications share a 2MB 32-way cache. As there are four cores, the ideal value for weighted speedup is 4. Figure 12 shows the weighted speedup for the LRU policy, and the UCP policy with the three partitioning algorithms - greedy, lookahead, and EvalAll. The *EvalAll* algorithm evaluates all the possible partitions to find the best partition. The greedy algorithm works well when all the benchmarks in the workload have convex utility curves (mix1) or when the cache is big enough to support the working set of majority of the benchmarks in the workload (mix2). However, for workloads that contain benchmarks with non convex utility curves (mix3 and mix4), the greedy algorithm does not perform as well as the EvalAll algorithm. The lookahead algorithm performs similar to the EvalAll algorithm without requiring the associated time complexity.

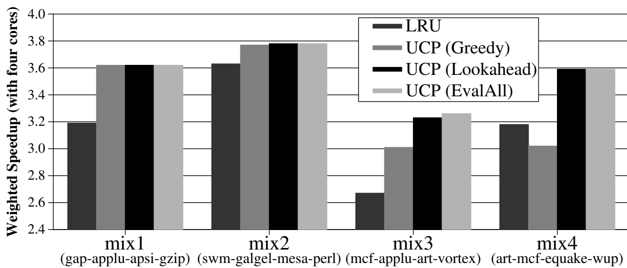

Figure 12. Comparison of Partitioning Algorithms

7. Related Work

7.1. Related Work in Cache Partitioning

Stone et al. [16] investigated optimal (static) partitioning of cache memory between two or more applications when the information about change in misses for varying cache size is available for each of the competing application. However, such information is hard to obtain statically for all applications as it may depend on the input set of the application. The objective of our study is to dynamically partition the cache by computing this information at runtime. Moreover, dynamic partitioning can adapt to the time-varying phase behavior of the competing applications, which makes it possible for dynamic partitioning to out perform even the best static partitioning [13].

Dynamic partitioning of shared cache was first investigated by Suh et al. [17][18]. [18] describes a low-overhead scheme that uses recency position of the hits for the lines in the cache to estimate the utility of the cache for each application. However, obtaining the utility information from main cache has the following shortcomings: (1) The number of lines in each set for which the utility information can be obtained for a given application is also dependent on the other application. (2) The recency position at which the application gets a hit is also affected by the other application, which means that the utility information computed for an application is dependent on (and polluted by) the concurrently executing application. UCP avoids these problems by separating the monitoring circuit from the main cache so that the utility information of the application is independent of other concurrently executing applications. Figure 13 compares UCP to a scheme that uses in-cache information for estimating utility information.

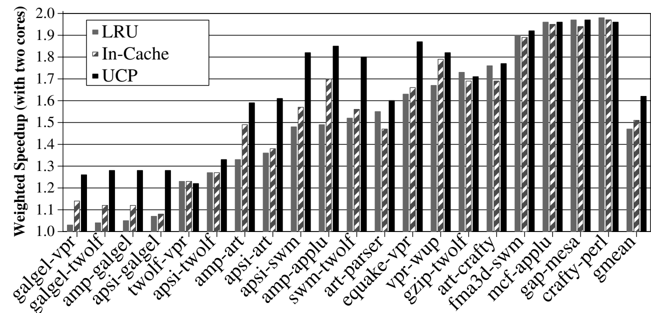

Figure 13. UCP vs. an In-cache monitoring scheme.

The in-cache scheme provides 4% average improvement compared to the 11% average improvement of UCP. Thus, seperating the monitoring circuit from the main cache is important to obtain high performance from dynamic partitioning. However, doing this by having extra tags for each cache line incurs prohibitive hardware overhead. Our proposal makes it practical to compute the utility information for an application, independent of other competing applications, without requiring huge hardware overhead.

Mechanisms to facilitate static and dynamic partitioning of cache is described in detail in [6]. Recently, Hsu et al. [5] studied different policies, including a utilitarian policy, for partitioning a shared cache between competing applications. However, they analyzed these policies using best offline parameters and mechanisms for obtaining these parameters at runtime is left for future work.

0-7695-2732-9/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

7.2. Related Work in Cache Organization

Liu et al. [8] investigated cache organizations for CMPs. They proposed *Shared Processor-Based Split L2* Caches, in which the number of private banks allocated to each competing application is decided statically using profile information. However, it may be impractical to profile all the applications that execute concurrently. Our mechanism avoids profiling by computing the utility information at run-time using cost-effective utility monitors.

Recent proposals [3][1] have looked at dynamic mechanisms to obtain the hit latency of a private cache while approximating the capacity benefits of a shared cache. Our work differs from these in that it focuses on increasing the capacity benefits of a shared cache. It can be combined with these proposals to obtain both improved capacity and improved latency from a cache organization.

7.3. Related Work in Memory Allocation

In the operating systems domain, Zhou et al. [19] looked at page allocation for competing applications using *miss ratio curve*. The objective of both their study and our study is the same, however, their study deals with the allocation of physical memory, which is fully associative, whereas, our study deals with the allocation of on-chip caches, which are set-associative. The hardware solution proposed in [19] stores an extra tag entry for each page in a separate hardware structure for each competing application. While this may be cost-effective in paging domain (approximately 4B per 4kB page), keeping multiple tags for each cache line for on-chip caches is hardware-intensive and power-hungry. For example, if four applications share a cache and each tag-entry is 4B, then the storage required per cache line is 16B (which is a 25% overhead for a 64B cache line), rendering the scheme impractical for on-chip caches. Fortunately, on-chip caches are set-associative which makes them amenable to dynamic set sampling (DSS). Our mechanism uses DSS to propose a cost-effective partitioning framework which requires less than 1% storage overhead.

8. Concluding Remarks

Traditional designs for a shared cache use LRU replacement which partitions the cache among competing applications on a demand basis. The application that accesses more unique lines in a given interval gets more cache than an application that accesses fewer unique lines in that interval. However, the benefit (reduction in misses) that applications get for a given amount of cache resources may not correlate with the demand. This paper proposes *Utility-Based Cache Partitioning (UCP)* to divide the cache among competing applications based on the benefit (utility) of cache resource for each application and makes the following contributions:

1. It proposes a low hardware overhead, utility monitoring circuit to estimate the utility of the cache resources for each application. Our evaluation shows that UCP outperforms LRU on dual-core system by up to 23% and on average 11%, while requiring less than 1% storage overhead.
2. It proposes the *Lookahead Algorithm*, as a scalable alternative to evaluating every possible partition for partitioning decisions when there are a large number of applications sharing a highly associative cache.

Although we evaluated UCP only for CMP processors, ideas presented in this paper can easily be extended to SMT processors. We considered the problem of cache partitioning among the demand streams of competing applications. The UMON circuits can be extended to compute utility information for prefetched data, which can help in partitioning the cache among multiple demand and prefetch streams. The UMON circuits can also be modified to estimate CPI, which can help in providing quality of service guarantees. The proposed framework can also be used to implement execution-time fairness [7] without requiring any profile information. This paper investigated UCP only for multiprogrammed workloads. For multithreaded workloads, UCP can take into account both the variation in utility of private and shared data, as well as the variation in utility of private data of competing threads. Exploring these extensions is a part of our future work.

Acknowledgments

Special thanks to Aamer Jaleel for continued discussion and feedback throughout this work. We also thank Ala Alameldeen, Lee Baugh, Ravi Iyer, Aashish Phansalkar, Srikant Srinivasan, Craig Zilles, the HPS group, and the anonymous reviewers for their comments and feedback. This work was supported by gifts from IBM, Intel, and the Cockrell Foundation. Moinuddin Qureshi was supported by an IBM PhD fellowship during this work.

References

[1] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA-33*, 2006.

[2] D. Chiou. *Extending the reach of microprocessors: column and curious caching.* PhD thesis, Massachusetts Institute of Technology.

[3] Z. Chishti et al. Optimizing replication, communication, and capacity allocation in CMPs. In *ISCA-32*, 2005.

[4] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5), 1968.

[5] L. R. Hsu et al. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT-15*, 2006.

[6] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS-18*, 2004.

[7] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT-13*, 2004.

[8] C. Liu et al. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA-10*, 2004.

[9] K. Luo et al. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.

[10] R. L. Mattson et al. Evaluation techniques in storage hierarchies. *IBM Journal of Research and Development*, 9, 1970.

[11] E. Perelman et al. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[12] M. K. Qureshi et al. A case for MLP-aware cache replacement. In *ISCA-33*, 2006.

[13] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning. Technical report, TR-HPS-2006-009. University of Texas, Austin, 2006.

[14] R. Rajkumar et al. A resource allocation model for QoS management. In *the 18th IEEE Real-Time Systems Symposium*, 1997.

[15] S. Ross. *A First Course in Probability*. Prentice Hall, 2001.

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

[17] G. E. Suh et al. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA-8*, 2002.

[18] G. E. Suh et al. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1), 2004.

[19] P. Zhou et al. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS XI*, 2004.

IEEE COMPUTER SOCIETY