Set Utilization Based Dynamic Shared Cache Partitioning

Peter Deayton, Chung-Ping Chung
Department of Computer Science
National Chiao-Tung University
Hsinchu, Taiwan, ROC
{pdeayton, cpchung}@cs.nctu.edu.tw

***Abstract*— As the number of processors sharing a cache increases, conflict misses due to interference amongst competing processes have an increasing impact on the individual performance of processes. Cache partitioning is a method of allocating a cache between concurrently executing processes in order to counteract the effects of inter-process conflicts. However, cache partitioning methods commonly divide a shared cache into private partitions dedicated to a single processor, which can lead to underutilized portions of the cache when set accesses are non-uniform.**

Our proposed method compliments these cache partitioning algorithms by creating an additional shared partition able to be shared amongst all processors. Underutilized areas of the cache are identified by a monitoring circuit and used for the shared partition. Detection of underutilization is based on the number of unique set accesses for a given allocated way. For a 16-way set associative cache, the implementation of our method requires 64 bytes of storage overhead per core in addition to that needed for the method that determines the sizes of the private partitions. For the tested system, our method is able to improve performance over the traditional LRU policy for a number of selected benchmark sets by an average of 1.4% and up to 13.3% for a two core system and an average of 1.4% and up to 7.8% for a four core system, and is able to improve the performance of a conventional cache partitioning method (Utility-Based Cache Partitioning) by an average of 0.1% and up to 0.5% for both a two and four core systems.

Keywords- cache partitioning; shared cache; set utilization; chip multi-processor

I. INTRODUCTION

With multi-core processors now the norm, the number of processors simultaneously sharing a shared cache has increased. This can result in an increase in the number of inter-process conflict misses within the shared cache and hence result in poor overall system performance. When using the LRU replacement policy this poor performance can be exacerbated due to the LRU replacement policy's demand approach to cache block selection, in which applications that have a high demand for unique cache blocks and poor temporal locality (for example, those that stream data) are allocated a larger portion of the cache than applications that have a lower demand for unique cache blocks but stronger temporal locality.

Cache partitioning is a method designed to avoid this destructive interference between applications by restricting the amount of cache applications can use. Generally, cache partitioning methods assign each application a dedicated private partition, in effect dividing the shared cache into a number of private caches. Whilst eliminating inter-process conflict misses, the effective cache capacity for each process is decreased, potentially increasing the number of capacity misses. Partition sizes are often adjusted dynamically based on a cost function with an objective such as improving the miss rate, CPI or fairness. The unit for partitioning can vary, with granularities ranging from line to way to set based. The focus of this paper is on improving cache partitioning methods using a way granularity that solely allocate private partitions. Our goal is to partition a cache in such a way as to gain the benefits of both a shared cache (decreased capacity misses) and private caches (decreased inter-process conflict misses).

Our proposed method takes note of the fact that not all blocks in a cache are used uniformly, particularly during a given repartitioning period since it is relatively short (five million cycles in some methods). After a processor has been allocated its private partition, information from a monitoring circuit is used to determine which ways in the allocated partition are underutilized. As the chance of inter-process conflict misses is low in these underutilized areas, they are then shared with the other processors. We determine through simulation that for the given benchmark sets the best performance is obtained if a way is shared when less than 15% of the sets within the way are underutilized. When used in conjunction with a way-based cache partitioning method that monitors the stack distance information of individual processors, our method requires an additional storage overhead of 64 bytes per core for a 16-way set associative cache with 32 sets monitored regardless of cache size.

Results show that for both two and four core systems improvements in performance can be made over both having no partitioning scheme and conventional cache partitioning into private partitions. Our method is able to improve performance over having no partitioning scheme for a number of selected benchmark sets by an average of 1.4% and up to 13.3% for a two core system and an average of 1.4% and up to 7.8% for a four core system. Performance is also improved over a conventional cache partitioning method (Utility-based Cache Partitioning) by an average of 0.1% and up to 0.5% for both two core and four core systems. Importantly, although the average improvement in performance is fairly modest, our proposed method is able to improve both the best and worst case performance of the conventional cache partitioning scheme.

This paper contributes a method for partitioning a cache into private and shared partitions based on the number of unique sets accessed, a measure we are unaware of being used in other related work. The method is targeted at systems where the number of cores is lower than the associativity of the cache, enabling each core to be allocated a minimum of one way each. As a result, the scalability of the proposed method is limited as the number of cores sharing a cache is increasing faster than the associativity of the cache.

The paper is divided into six sections. The first serves as an introduction to the topic and research. The proposed method is then detailed, with the rationale and description of the algorithm specified. Next, the experimental methodology is shown and afterward results are shown and analyzed. Related work is then covered and compared with our method and lastly the conclusion is presented.

II. Set Utilization Based Dynamic Cache Partitioning

A. Rationale

To determine how to better partition a cache, we analyze the cache access patterns of an application over time. Given a way-based partitioning method and the LRU replacement policy, looking at the LRU stack distance hit counts of an application is useful in determining what the effect of allocating a certain number of ways to an application would be [1]. Fig. 1 shows the average number of hits per stack distance position in a 1MB 16-way set associative L2 cache per five million simulated cycles over one billion simulated cycles for the gzip benchmark from the SPEC CPU2000 benchmark suite. The simulation parameters used are further outlined in Table I in Section IV. Five million cycles were chosen as it is the suggested repartitioning period as discussed in [2] and [3]. For this benchmark, as the number of ways allocated (increasing stack position) increases, the number of hits steadily decreases, i.e. there is a diminishing benefit from an increase in cache capacity. However this figure does not describe accurately the distribution of the hits within the cache. Fig. 2 shows the average number of hits per stack distance position for each set in the cache for the same parameters. As can be seen, for a given stack distance position not all sets are accessed uniformly, an observation also noted in work related to cache set mapping functions [4]. This presents an opportunity for improvement over a partitioning scheme that allocates only private partitions.

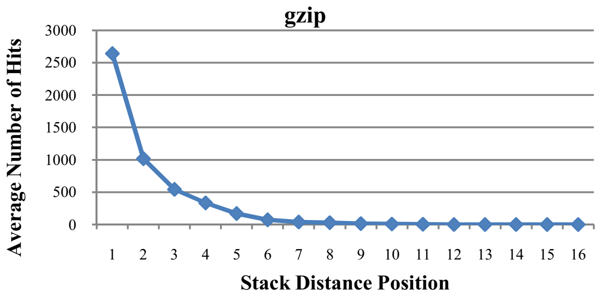

Figure 1. Average number of hits per stack distance position in a 1MB 16-way set associative L2 cache per five million simulated cycles over one billion simulated cycles for the gzip benchmark

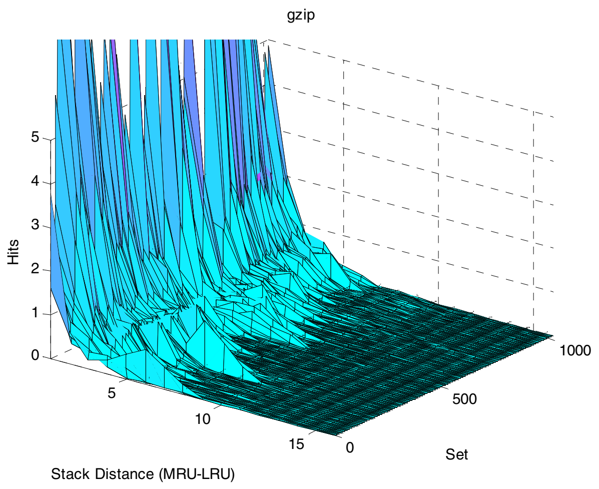

Figure 2. Average number of hits per stack distance position for each set in the cache

Fig. 3 shows the average percentage of total sets accessed per stack distance position over five million cycles for a number of selected SPEC CPU2000 benchmarks. We find the utilization of cache sets varies greatly between both applications and stack distance positions. If two processes share a way and the utilization of a given process's stack position for that way is low, the chance of accesses between processes conflicting is low. Sharing this way should be able to decrease the number of capacity misses and have little to no effect on the number of inter-process conflict misses of processes sharing the way. Our method is based upon this principle. If a way allocated to a process is underutilized, it is shared with other processes. This creates two types of partitions - private and shared.

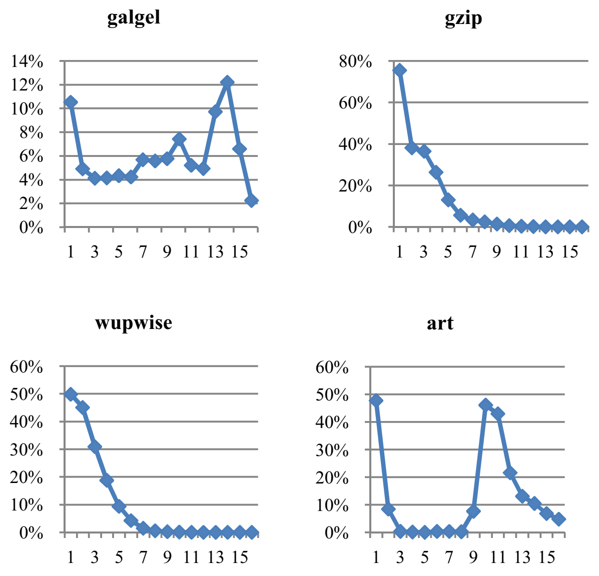

Figure 3. Average cache set utilization for the gzip, galgel, wupwise, and art benchmarks. The y-axis represents the average percentage of sets used.

B. Partitioning Algorithm

The overall framework of the proposed method for a dual core system is shown in Fig. 4. Each processor has private L1 instruction and data caches that are connected to a shared L2 cache. Additionally, each processor is connected to a monitoring circuit that is used to gain hit information for each processor as if the L2 cache were private to it. As such it is not necessary to contain the actual cache block data, only the tags to know if a cache access would hit or not (referred to as an Auxiliary Tag Directory (ATD) in [3]). The private partitioning algorithm allocates all the ways in the L2 cache between each processor. The shared partitioning algorithm then takes this partition size information along with information on cache utilization to determine which allocated ways can be shared. For systems with more than two cores, the L1 caches are connected to the shared L2 cache and an additional hit monitor which is connected to the partitioning algorithms.

1) Private Partition Size Determination

In general, any way-based partitioning algorithm that completely allocates all ways in a cache can be used to determine the private partition sizes. In our experiments, we use the Utility-based Cache Partitioning (UCP) partitioning algorithm to determine the sizes of the private partitions. This partitioning algorithm aims to maximize the total reduction in cache misses. Further details on the algorithm can be found in [3]. Although the UCP method is used, our method is complimentary to other partitioning algorithms with alternate cost functions (for example based on fairness [5]).

2) Shared Partition Size Determination

The shared partitioning algorithm determines which allocated ways are amenable to sharing. The first step is to detect the usage of a stack distance in a set. To do this the per core hit monitors needed for private partition determination are modified to add an additional used bit for each stack distance position in the set. The number of sets used is then the total number of used bits set for a given stack distance. If the total number of sets used is below a threshold, the way is considered underutilized.

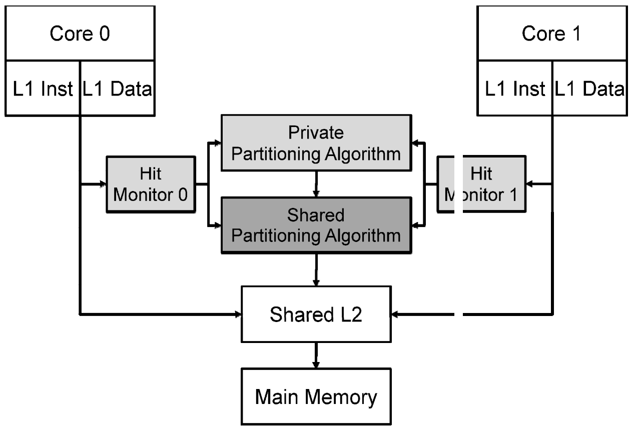

Figure 4. Design overview of proposed method for a dual core system, showing additional structures required.

3) Sharing Granularity of Shared Partition

Once an allocated way has been determined as underutilized and to be shared, the question arises of which processes to share the way with. Two options are presented.

The first is that the newly shared way can be shared amongst all other processes, termed the ShareAll algorithm. This provides a potential decrease in capacity misses amongst all processes, however at the expense of a potential increase in inter-process conflict misses, with the likelihood of inter-process conflict misses increasing as the number of processors sharing the cache increases. The implementation for this method is straightforward and only requires the total size of the shared partition to be tracked (this can be further optimized to only track the sizes of the private partitions).

Pseudo code for this method is shown below in Algorithm 1. For each private partition, the utilization of each allocated way is compared to a threshold for sharing. If it is below the threshold, the way is shared, otherwise no further ways from that partition are shared. Allocated ways are shared in order from least recently used to most recently used. This is due to the LRU replacement policy obeying the stack property, meaning accesses that hit in a private partition will not hit in the shared partition.

Algorithm 1 ShareAll algorithm

```
foreach core do:
  foreach allocated_way from LRU to MRU do:
    if(number_of_used_sets < threshold)
      private_partition[core] -= 1
      shared_partition += 1
    else
      skip this core
```

The second method is to share the to-be-shared way with a selected number of other processors, termed the ShareSubset algorithm. The implementation of this method is more complex and complicates the determination of which block the replacement policy should evict. The problem of determining if a block belonging to a process is in the shared partition is a form of constraint satisfaction problem (CSP) and NP hard. As the replacement policy must be able to choose a block to replace by the time a block is fetched from the next level in the memory hierarchy (in our case main memory, but possibly another cache), the latency must not be too long. Therefore instead of more complicated and slower algorithms, we use an approximation. Cores are allowed to share underutilized ways whilst the combined utilization is below a threshold. The replacement policy then tracks the total number of ways a core can use, evicting a block from any core that exceeds its allocation (further details provided in below in the Augmented LRU Policy section).

Pseudo code for this method is shown below in Algorithm 2. Similar to the ShareAll method, the utilization of each allocated way from the LRU allocated way to the MRU allocated way is compared to the threshold. If underutilized, the way is transferred to the shared partition, with the number of ways in the shared partition that the originating core is allowed to use increased. Then ways from other cores that can be shared in this position are ordered in terms of reduction in misses. The newly shared way is shared with other cores until the total utilization reaches a threshold, after which the chance of conflicts is established as too great. This method requires storage of both the private partition sizes and number of blocks that can be used in the shared partition for each connected core.

Algorithm 2 ShareSubset algorithm

```
foreach core do:
  foreach allocated_way from LRU to MRU do:
    if(number_of_used_sets < threshold)
      private_partition[core] -= 1
      shared_partition[core] += 1
      corelist = get list of cores ordered in benefit from
                 additional way
      foreach toCore in coreList do:
        if(combined_usage < threshold):
          shared_partition[toCore] += 1
    else
      skip this core
```

4) *Augmented LRU Policy*

As with other way-based cache partitioning methods, the standard LRU policy is also augmented to support our method. Each cache line has an additional tag representing which processor it belongs to (one bit for a cache shared between two processors). This is used to determine how many blocks are allocated in a set to each processor. On a cache miss the LRU block needs to be chosen whilst keeping the partitioning constraints. Roughly speaking, if any processors have too many blocks (greater than the combined size of the private and shared partitions) the LRU block from them is chosen for eviction. If not, then the LRU block of the shared partition is chosen for eviction. If there is no shared partition (i.e. the cache utilization for each way is high), then the LRU block of the private partition is then chosen. In addition, a lazy repartitioning method [3] is used to evict cache blocks on demand rather than evicting all blocks belonging to a processor that are over its allocated limit.

C. *Storage Overhead*

The additional hardware requirements are minimal given a private partitioning algorithm that already uses an auxiliary tag directory. Using the UCP method as an example, for a 32-bit system it requires an additional storage overhead of 1920 bytes for a 1MB 16-way cache shared between two cores with 32 sets monitored. Our method adds an additional bit for each stack position monitored, and with 32*16 blocks monitored, 64 additional bytes are needed for storage per core, a 6.67% increase over the UCP method or 0.006% of the total cache size per core. For additional cores, this number is multiplied by the total number of cores.

D. *Effect on Latency*

The effect of the proposed method on the latency of the system has two components - that of the partition determination, and that of the enforcement of the partition constraints.

Although partition determination is not on the critical path of a cache access, it is limited by the repartitioning period as it must complete before the partitions are changed again. As the repartitioning algorithm is composed mainly of additions, subtractions, and comparisons, the latency of the algorithm is predicted to be within the repartitioning period.

Partition enforcement is done through the augmented LRU policy, and thus falls on the critical path of a cache access. For any cache access (be it a hit or miss), the latency will increase slightly due to the additional core tag comparison required, but which most likely can be masked within the existing latency or requiring only one extra cycle. On a cache miss, the latency for choosing the block to evict is increased, however this computation can be performed whilst the new block is being fetched from the next level of the memory hierarchy, hiding the increased latency. Therefore overall, the proposed method should have little if any effect on the latency of the cache.

III. EXPERIMENTAL METHODOLOGY

A. *Baseline Configuration*

Whilst any number of cores can be connected to the shared cache, using a way based partitioning method limits the actual number of cores that can be sensibly connected. As a minimum of one way is allocated to each processor and the number of ways is usually limited for hardware cost reasons to 16 or 32, a reasonable maximum number of cores would be 4 to 8. For this reason our experiments only simulate systems with two and four cores.

The Simics system simulator [6] was used to simulate a system running Solaris 10 on an UltraSPARC III processor. Table I shows the configuration parameters used.

Either two or four processors were used, with each of the benchmarks bound to a single processor using pbind. Benchmarks were run in parallel and fast forwarded to the beginning of their main loops. The caches were then warmed up for 500M cycles, and simulation of results began for 1B cycles.

TABLE I. BASELINE CONFIGURATION USED

Processor	2/4 processors, single threaded
Private L1 Instruction Caches	16KB, 64 byte block size, 4-way set associative, 3 cycle access latency
Private L1 Data Caches	16KB, 64 byte block size, 4-way set associative, 3 cycle access latency
Shared L2 Unified Cache	1MB/2MB, 64 byte block size, 16-way set associative, 15 cycle access latency
Memory	4GB, 200 cycle access latency

B. *Benchmarks*

A number of benchmarks were chosen from the SPEC CPU2000 benchmark suite [7], all using the reference input sets. As partitioning algorithms are designed to counter the poor performance of the LRU replacement policy for applications with a large number of misses and that have a limited need for cache (utility), the benchmarks have been divided into four categories along two axes - total number of misses and utility. Benchmarks with an average of more than

TABLE II. CLASSIFICATION OF BENCHMARKS

	Low Utility	High Utility
High Number of Misses	swim, wupwise, equake	mcf,art, vpr, twolf
Low Number of Misses	gzip, crafty, ammp, fma3d	parser, galgel

500 misses over five million cycles (arbitrarily chosen) are classified as having a high number of misses. Classification into high and low utility is based on stack distance position hit curves. Benchmarks where the number of hits for higher stack distances is less than 10 are classified as low utility, as these positions are relatively unused and represent receiving no benefit from being allocated more cache. The benchmark classifications are shown in Table II.

A number of benchmark sets for two and four cores systems were then selected based on this categorization. 10 benchmarks for the two core system and 11 benchmarks for the four core system comprising a combination of each of the categories were chosen shown in Table III.

It is predicted that any benchmark sets containing benchmarks with a high number of misses and low utility will greatly benefit from cache partitioning. It is also predicted that our proposed method will outperform the UCP method particularly for benchmark sets containing benchmarks with low average set usage.

TABLE III. BENCHMARK SETS USED FOR SIMULATION

Two Core Benchmark Sets	Four Core Benchmark Sets
art-ammp	ammp-fma3d-parser-galgel
crafty-fma3d	ammp-fma3d-vpr-art
equake-galgel	art-twolf-parser-galgel
gzip-parser	gzip-crafty-ammp-fma3d
mcf-art	mcf-art-vpr-twolf
mcf-parser	parser-parser-galgel-galgel
parser-galgel	swim-equake-gzip-crafty
swim-crafty	swim-vpr-gzip-parser
swim-mcf	swim-wupwise-equake-wupwise
wupwise-equake	swim-wupwise-mcf-twolf
	swim-wupwise-parser-galgel

C. Metrics

We use the normalized weighted speedup to evaluate the performance of our proposed method. The weighted speedup measures the relative difference in the number of instructions executed per cycle for an application running concurrently compared to when it is run in isolation. This value is then normalized against the performance of the no-partitioning policy (LRU). The formula for calculating the weighted speedup is shown below in (1).

$$Weighted\ Speedup = \sum IPC_i \,/\, SingleIPC_i \qquad (1)$$

IV. RESULTS AND ANALYSIS

A. Two Core System

The performance of our proposed method compared with a no-partitioning method (using the standard LRU replacement policy for eviction decisions) and the UCP method (using only private partitions) for a two core system in shown in Fig. 5. For the case in which there are only two cores in the system, the ShareAll and ShareSubset algorithms are still valid. The ShareAll algorithm shares an allocated way in a private partition if the set utilization of that way is below a threshold (sharing is based only on the utilization by the owning core). The ShareSubset algorithm shares an allocated way in a private partition if the set utilization of that way combined with the set utilization of the way that it would be shared with from the other core is below a threshold (sharing is based on combined utilization).

We find our proposed method increases performance against the no-partitioning case by an average of 1.4% over our selected benchmark sets, and up to 13.3% for the case of the mcf-art benchmark set. This is in comparison to the UCP method, which increases performance relative to the no-partitioning case by an average 1.3% over the selected benchmark sets and up to 12.8% for the mcf-art benchmark set. Both variations of our proposed method are able to improve performance of the UCP method. The relative low average increase in performance is a result of 8 out of the 10 benchmark sets differing in performance by less than 1% between no-partitioning and cache partitioning. This indicates cache partitioning may not be particularly effective in a two core system for most combinations of applications.

Of note is for benchmark sets where the UCP method performs worse than the no-partitioning method (wupwise-equake, swim-crafty, crafty-fma3d), our proposed method is able to increase the performance, bringing it closer to the performance of the no-partitioning method. Interestingly, for the parser-galgel benchmark set, the performance of our proposed method is higher than the no-partitioning and UCP methods. Our proposed method is able to take advantage of the low set utilization of the fma3d, swim and crafty benchmarks to provide a larger capacity for the benchmarks.

In comparison to the UCP method, our proposed method provides an average of 0.1% better performance and up to 0.5% better performance comparative to the no-partitioning case. This small increase likely represents a diminishing return in further improving the UCP method.

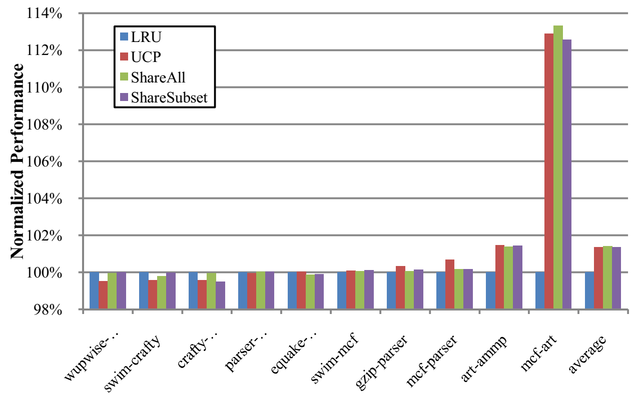

Figure 5. Performance of no partitioning, UCP, and proposed method for two core system

Also, we find the difference in performance between the two variations of our proposed method (ShareAll and ShareSubset) to be very small (0.06% in favor of the ShareAll algorithm). Therefore, a hardware implementation of the proposed method would be able to use the less costly ShareAll algorithm with a negligible effect on performance.

B. Four Core System

The performance of the proposed method for a four core system is shown in Fig. 6. The performance of a no-partitioning (LRU), UCP, and the two variations of our proposed method (ShareAll and ShareSubset) are shown. ShareAll implements the algorithm described in Algorithm 1 and shares an allocated way in a private partition with all other cores in the system if the number of utilized sets is below a threshold. The ShareSubset variation implements the algorithm described in Algorithm 2 and allows each core a different proportion of the shared partition.

We find our proposed method increases performance in comparison to the no-partitioning case by 1.4% on average for the selected benchmark sets and up to 7.8% for the art-twolf-parser-galgel benchmark set. In comparison, the UCP method increases performance relative to the no-partitioning case by 1.3% on average for the selected benchmark sets and up to 7.7% for the art-twolf-parser-galgel benchmark set. The average performance is similar to the two core case, in that 7 out of the 11 benchmark sets have a difference in performance of less than 1% between the methods. The two variations of our proposed method are again very similar in performance, however this time the ShareSubset variation offers slightly better average performance (by 0.06%).

Results across the benchmark sets are similar in pattern to those of the two core system, with the proposed method able to increase both the worst case and best case performance of the UCP method, with the difference being up to 0.5% in the case of the ammp-fma3d-parser-galgel benchmark set. In addition, our method is able to out-perform both a no-partitioning method and the UCP method in 6 out of the 11 benchmark sets and out-perform the UCP method in 9 out of the 11 benchmark sets, indicating our method may be more beneficial as the number of cores increases.

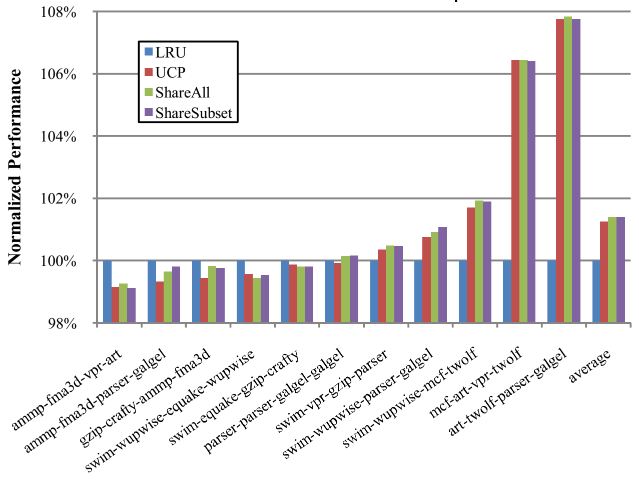

Figure 6. Performance of no partitioning, UCP, and proposed method for four core system

C. Set Considered Used Threshold

It is not necessary for the used bit to be set given a single access to that stack distance position. A counter can instead be used to have a threshold at which a stack distance position in a set is considered used or not. Sensitivity analysis of this parameter was conducted using the average weighted speedup of the benchmark sets. As the threshold for a stack distance position being considered used or not increases, overall performance deteriorates. The best performance is for the case when the threshold is 1, or whether the stack distance position has been used at all for that set. This is fortunate since it means the overhead of the stack distance position usage bits will not be large and only one bit for each block is needed in the monitored sets.

D. Way Considered Utilized Threshold

The number of unique sets used given a stack distance position before that stack distance position is considered sufficiently utilized is an additional parameter that can be considered. The threshold for the number of unique sets used before which a way is shared is adjustable and sensitivity analysis of this parameter is shown in Fig. 7. Unique set usage thresholds were chosen based on a percentage of the total number of sets, with results from 0% to 100% in 1% intervals of the total sets used compared. Performance for each of the methods was normalized against the no-partition four core case.

Sharing a processor's allocated way based on 15% utilization shows the best results. The lower the threshold, the closer the performance to that of the UCP method. This is to be expected as the chance of sharing a way will be low, hence the shared partition is non-existent and the UCP derived partitions are exclusively used. When the threshold increases, the performance more closely matches that of the non-partitioned case. This also is expected, as the probability of a processor's way being shared is higher and hence more likely to have a larger shared partition, equivalent to the non-partitioned case. The ShareSubset method does not match that of the non-partitioned case when the threshold is 100%

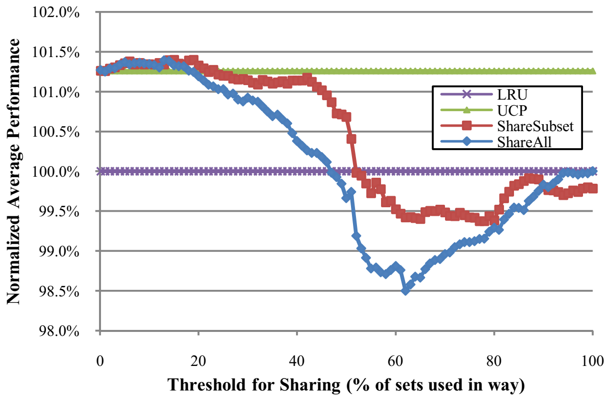

Figure 7. Performance of proposed method for varying thresholds

as it uses the combined utilization and each core can potentially utilize 100% of the sets in the way. Of note is the performance degradation as the threshold increases above 40% of the total sets used. The performance quickly becomes worse than the no-partitioning case and then improves to match that of the no-partitioning case. This is due to the suboptimal restriction of cache capacity amongst the applications.

V. RELATED WORK

Cache partitioning as a research topic saw an increase in interest with the rise of chip multi-processors. A number of different methods have been proposed, with a large proportion using way-based partitioning.

Dynamic Partitioning of Shared Cache Memory [2] is a way-based partitioning method to dynamically reduce the total number of misses for simultaneously executing processes. Cache miss information for each process is collected through stack distance counters (termed marginal gain counters) and a greedy algorithm used to determine a new partition size. Of note is the rollback mechanism, where the performance of the current and previous partition sizes are compared and the better one chosen for the next partition size. Limitations of this method is include the fact that separate hit counters are not kept for each core making miss prediction less accurate and the limited scalability to four or more cores.

Utility Based Cache Partitioning [3], the example method used for determining the private partitions in this paper, allocates ways amongst the cores based on maximizing the reduction in misses. This is computed through stack distance counters and alternate tag directories enabling the effect of various cache partition sizes to be determined simultaneously. The method however is not able to adequately adjust to situations where having no explicit partitioning policy performs well (applications with a low number of inter-process conflict misses running concurrently).

Cooperative Cache Partitioning [8] is another way based partitioning method designed to deal with thrashing threads. It uses Multiple Time-sharing Partitions to share a large partition between multiple thrashing threads, giving each thread the entire partition for a portion of the repartitioning period. This combined with the Cooperative Caching [9] method provides an improvement in performance, particularly Quality of Service. This scheme is also compatible with our proposed method and we anticipate additional improvements in performance if used together.

Adaptive Shared/Private NUCA Cache Partitioning [10] is a method similar to our proposed method that divides a cache into shared and private partitions. The difference lies in the method for determining the size of the partitions. In this method shadow tags are used, however only one way is reallocated per repartitioning period, meaning the method is unable to adjust quickly to changes in working sets unlike our method which can make larger changes in partition sizes. Additionally, cache misses are used as the determinant for when to repartition the cache, meaning applications with a large number of cache misses yet no change in their working sets will cause unnecessary repartitioning.

Recent work has noted the poor scalability of having separate monitors for each core and methods have been proposed including In-Cache Estimation Monitors [11] and set-dueling [12] to eliminate the need for separate monitors. A number of sets in the cache are dedicated to a particular core from which the monitored statistics can be gathered. These methods improve in effectiveness as the cache size increases while associativity remains constant, as there are a larger number of sets and less reduction in effective cache capacity per core. These methods are compatible with our proposed method and can also be adjusted to help in the monitoring of set usage, helping reduce the overhead of our proposed method.

To the best of our knowledge, our method is the only cache partitioning method that explicitly takes into account the non-uniform set usage of applications in partitioning decisions.

VI. CONCLUSION

Previous cache partitioning methods ignored the effect of the non-uniformity of cache set accesses upon the effectiveness of partitioning decisions. Our method uses this non-uniformity of set accesses as the basis for our improvement of cache partitioning schemes with partitions that are private to each core. A shared partition is created with underutilized portions of the private partitions, gaining the benefits of increased capacity while minimizing the chance of destructive cache interference through inter-process conflict misses. Our results show modest improvements in performance for both two and four core systems over having no partitioning scheme and using private partitions only. As our method applies to way-based partitioning, scalability to a larger number of cores with a relatively low associativity shared cache remains a problem. There are still further opportunities in dealing with non-uniform cache set accesses to improve the performance of cache partitioning (for example alternate address mapping functions) and will be the focus of our future work.

REFERENCES

[1] R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal* , vol.9, no.2, pp.78-117, 1970.

[2] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," The Journal of Supercomputing, vol. 28, no. 1, pp. 7-26, Apr. 2004.

[3] M. K. Qureshi, and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on* , pp.423-432, Dec. 2006.

[4] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," In *Proceedings of the 11th international conference on Supercomputing (ICS '97)*, pp. 76-83, 1997.

[5] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on* , pp.111-122, 2004.

[6] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg; F. Larsson, A. Moestedt, B. Werner, "Simics: A full system simulation platform," *Computer* , vol.35, no.2, pp.50-58, 2002.

[7] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the New Millennium," *Computer* , vol.33, no.7, pp.28-35, Jul 2000.

[8] J. Chang, and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," In *Proceedings of the 21st annual international conference on Supercomputing (ICS '07)*, pp.242-252, 2007.

[9] J. Chang; G. S. Sohi, "Cooperative caching for chip multiprocessors," *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on* , pp.264-276, 2006.

[10] H. Dybdahl, and P. Stenstrom, "An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors," In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pp.2-12, 2007.

[11] Y. Xie, and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches,". In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*, pp. 174-183, 2009.

[12] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08)*, pp. 208-219, 2008.