國立交通大學

電機資訊國際學位學程

碩士論文

快取分區模式效能改進的方法

Techniques to Improve the Performance of Cache Partitioning Schemes

研究生：戴彼德

指導教授：鍾崇斌 教授

中華民國一百零一年三月

快取分區模式效能改進的方法

Techniques to Improve the Performance of Cache Partitioning Schemes

研 究 生：戴彼德　　Student：Peter Deayton

指導教授：鍾崇斌　　Advisor：Chung-Ping Chung

國 立 交 通 大 學
電 機 資 訊 國 際 學 位 學 程
碩 士 論 文

A Thesis
Submitted to EECS International Graduate Program
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master

Mar. 2012

Hsinchu, Taiwan, Republic of China

中 華 民 國 一 百 零 一 年 三 月

快取分區模式效能改進的方法

學生：戴彼德　　指導教授：鍾崇斌　博士

國立交通大學資訊科學與工程研究所碩士班

摘要

當越來越多處理器共享一個快取時，會使得處理器之間對於快取資源的競爭更加劇烈，進而影響個別單一程序的效能。快取分區方法是一個可以降低程序間互相競爭的方法，其通常將快取分割給各個處理器單獨使用，然而此方法在程序不平衡的存取快取時，會造成快取分區利用率不佳的問題。針對此問題，我們提出兩個方法做改善:1. 額外增加一塊共享分區供所有處理器共同使用 2. 針對每一個處理器，給予不同的索引方式。另外，我們針對所有分區，在一般常用於減少失誤的方法下(可變動區塊大小、可變動關聯度、改變替代策略)進行討論。

Techniques to Improve the Performance of Cache Partitioning Schemes

Student：Peter Deayton Advisor：Professor Chung-Ping Chung

Institute of Computer Science and Engineering
National Chiao-Tung University

Abstract

As the number of processors sharing a cache increases, misses due to destructive 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 this inter-process interference. 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 processes access sets in the cache non-uniformly. Two techniques are proposed designed to take advantage of this non-uniformity - the creation of an additional shared partition able to be shared amongst all processors and alternate cache set indexing functions for each core. Also discussed is the application of general miss reduction techniques (variable block size, variable associativity, and changing replacement policies) on a per-partition basis.

Acknowledgements

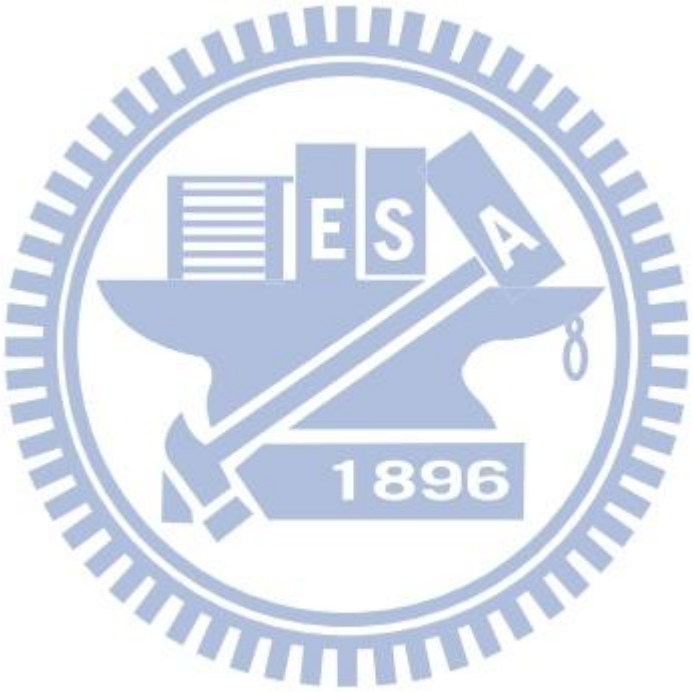
Thank you to my advisor Professor Chung-Ping Chung for his support and advice throughout the course of my degree. Thank you also to my oral defense committee members – Professor Chung-Ta King, Professor Wei-Chung Hsu, and Professor Krishna Kavi for their insightful suggestions for improvements. Finally, a thank you to all members of the Computer Systems Lab without which I would not have enjoyed researching half as much.

Peter Deayton

國立交通大學資訊科學與工程研究所碩士班

中華民國一百零一年三月

Table of Contents



List of Figures

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 [1], IPC [2] or fairness [3]. The unit for partitioning can vary, with granularities ranging from line [4] to way (either through using fixed ways [5] or adjusting the replacement policy [1]) to set based [6]. The focus of this paper is on improving cache partitioning methods using a way granularity that solely allocate private partitions. Our goal is to propose improvements to these private partitioning schemes and determine areas for future research.

We propose two techniques to improve the performance of these cache partitioning

methods, both based on 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).

The first technique (Set Utilization Based Cache Partitioning) is to create a shared partition in addition to the private partitions. 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. The threshold for sharing a way can be determined statically or dynamically and we propose methods for both. 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 and minimal (one cycle) if any increase in cache access time.

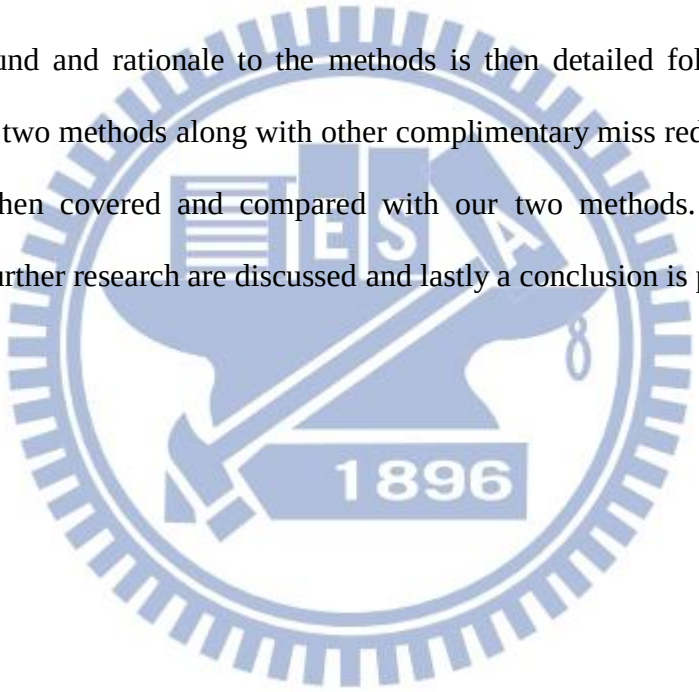
The second technique (Alternate Per Core Indexing Functions) is to apply an alternate cache set indexing function for each core connected to the cache, designed to reduce conflicts between applications that have similar set access patterns. These functions can be selected either randomly or dynamically adjusted by detecting if the cache set accesses are balanced across all sets.

In addition, we discuss modifications required to apply techniques used to reduce the miss rate of a private cache on a per partition basis. These techniques include changing the block size, changing the associativity, and changing the replacement policy.

This paper contributes two methods for improving private cache partitioning schemes. The first, the partitioning of a cache into private and shared partitions based on the number

of unique sets accessed is a measure we are unaware of being used in other related work. The first 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, scalability is limited as the number of cores sharing a cache is increasing faster than the associativity of the cache. The second is the proposed alternate cache indexing functions for each core. While related work generally aims to minimize the number of conflict misses in a cache, the indexing function is identical amongst all cores in a system.

The paper is divided into six sections. The first serves as an introduction to the topic and research. Background and rationale to the methods is then detailed followed by detailed descriptions of the two methods along with other complimentary miss reduction techniques. Related work is then covered and compared with our two methods. Future work and opportunities for further research are discussed and lastly a conclusion is presented.



II. Background

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 [7]. Figure 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. Five million cycles were chosen as it is the suggested repartitioning period as discussed in [1] and [8].

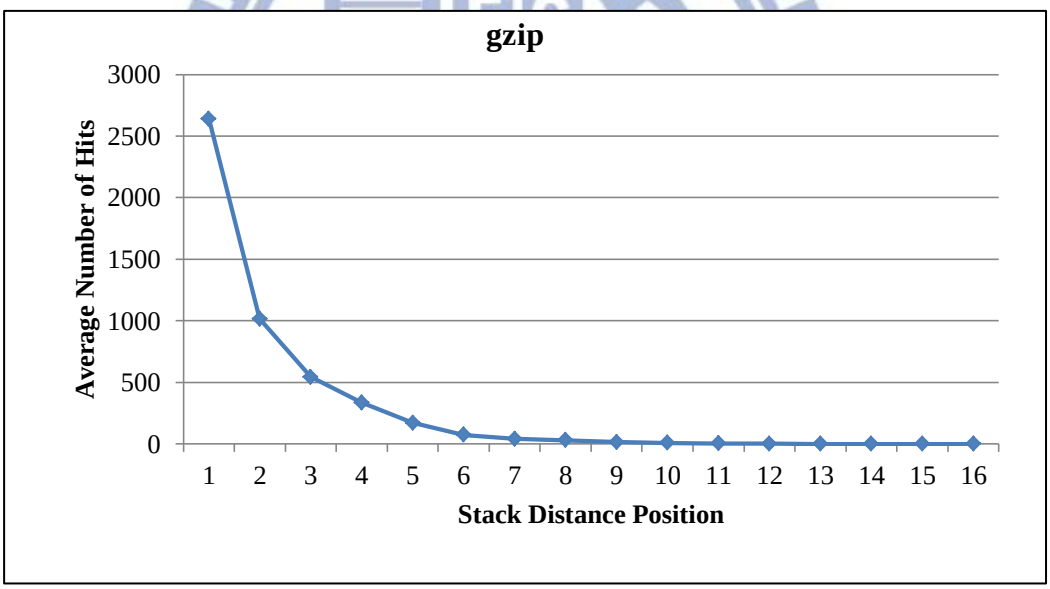


Figure 1: Average number of hits per stack distance position in a 1MB 16-way set associative L2 cache per five million simulated cycles over one billion simulated cycles for the gzip benchmark

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 allocating more ways in the cache to the application. However this figure does not describe

accurately the distribution of the hits within the cache. Figure 2 shows the average number of hits per stack distance position for each set in the cache for the same parameters. We take note of two things. The first is that for a given stack distance position not all sets are accessed uniformly, particularly between stack distances six to eight. If a partitioning scheme that allocates only private partitions allocated the gzip benchmark eight ways, there would be an amount of space wasted. The second thing to note is that set accesses are somewhat clustered, with relatively fewer hits around set 300 compared to set 500. These two observations present opportunities for improvement of a cache partitioning scheme that allocates private partitions only.

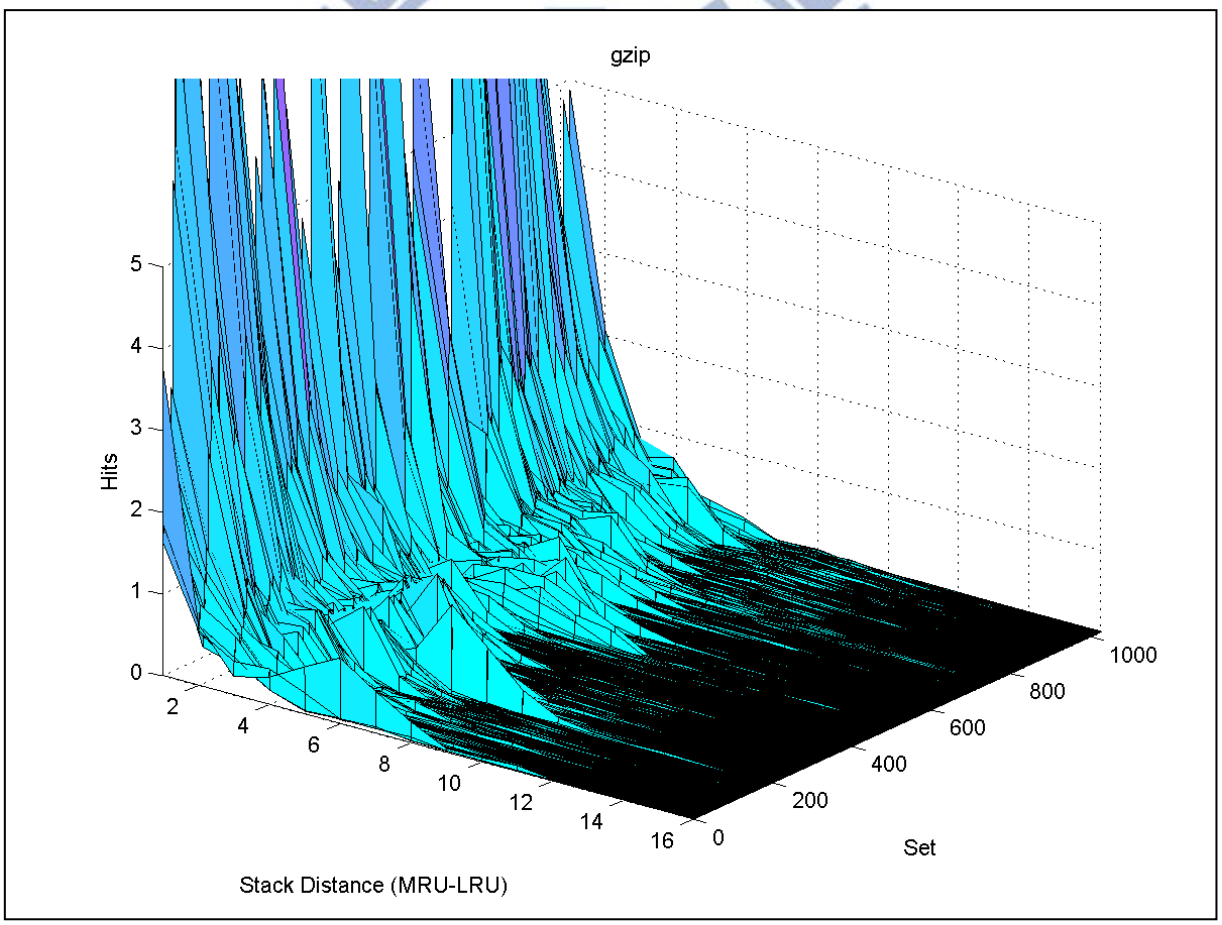


Figure 2: Average number of hits per stack distance position for each set in the cache for the gzip benchmark

To see how prevalent this non-uniformity of set accesses is, we observe the average percentage of total sets accessed per stack distance position for a number of applications. Figure 3 shows this information for a five million cycle period averaged over one billion 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 first 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.

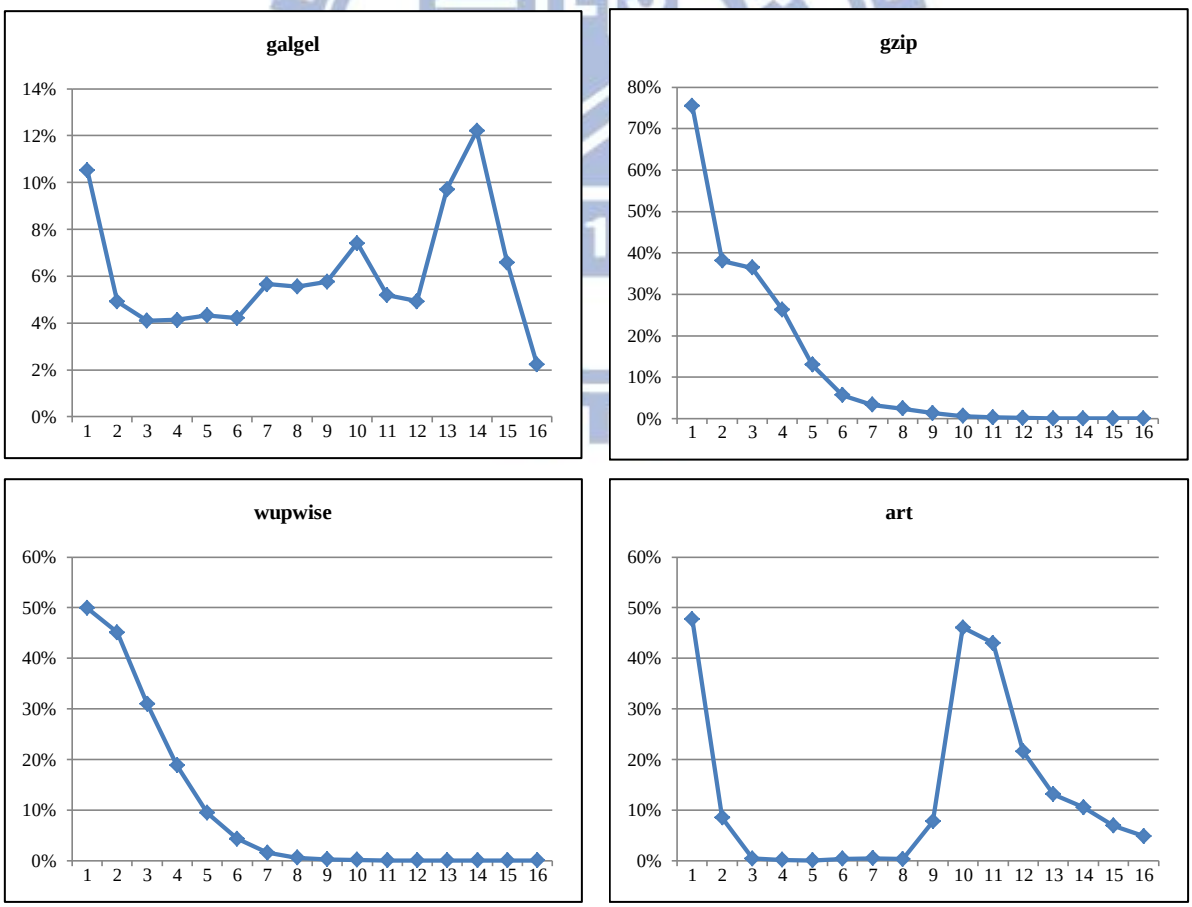


Figure 3: Average cache set utilization for the gzip, galgel, wupwise, and art benchmarks. The y-axis represents the average percentage of sets used.

Although the chance of conflicts between two processes in the shared partition should be low, this may not be the case when the set accesses are non-uniform. If the set accesses are similarly clustered for two applications, inter-process conflict misses will increase. The gap benchmark is an example of these clustered set accesses. Figure 4 shows the average number of hits per stack distance position for each set in the cache.

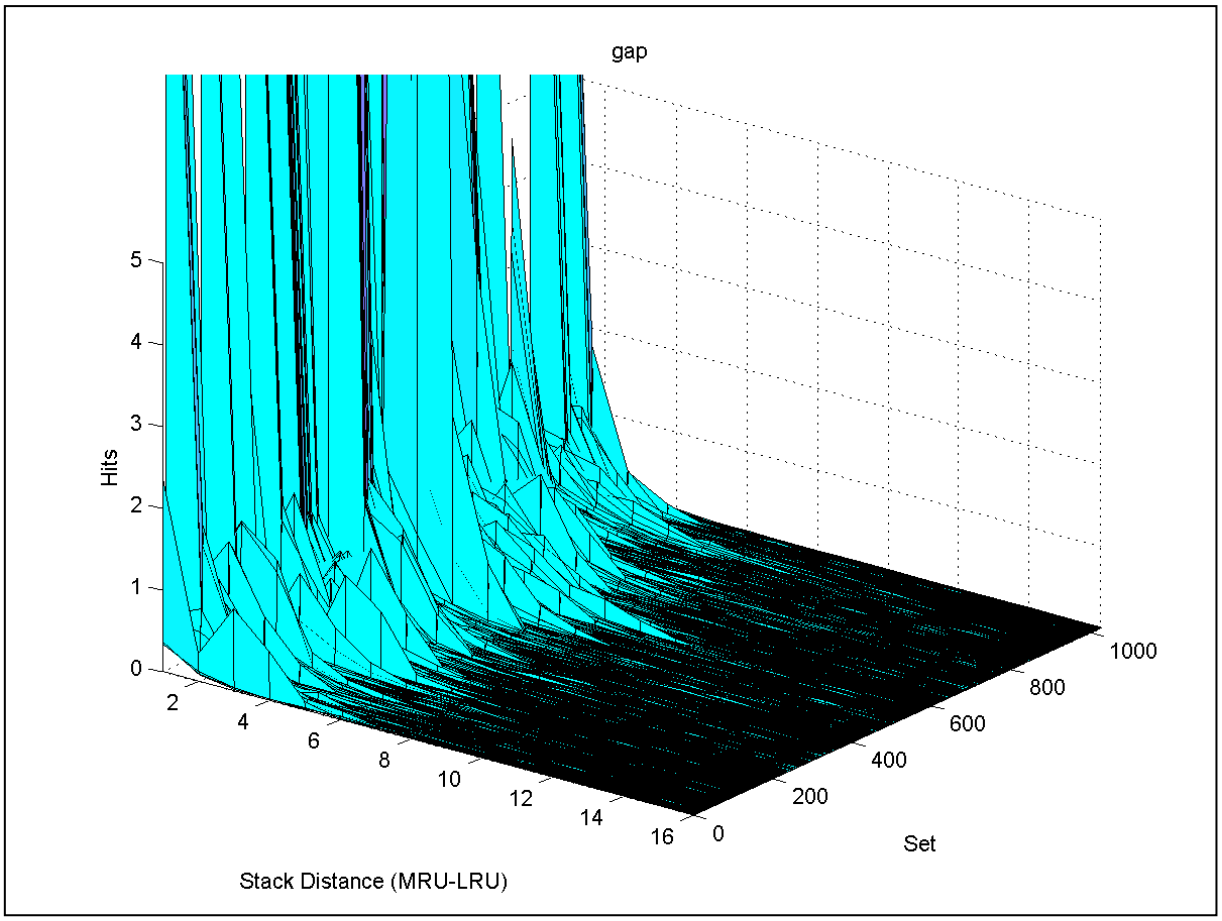
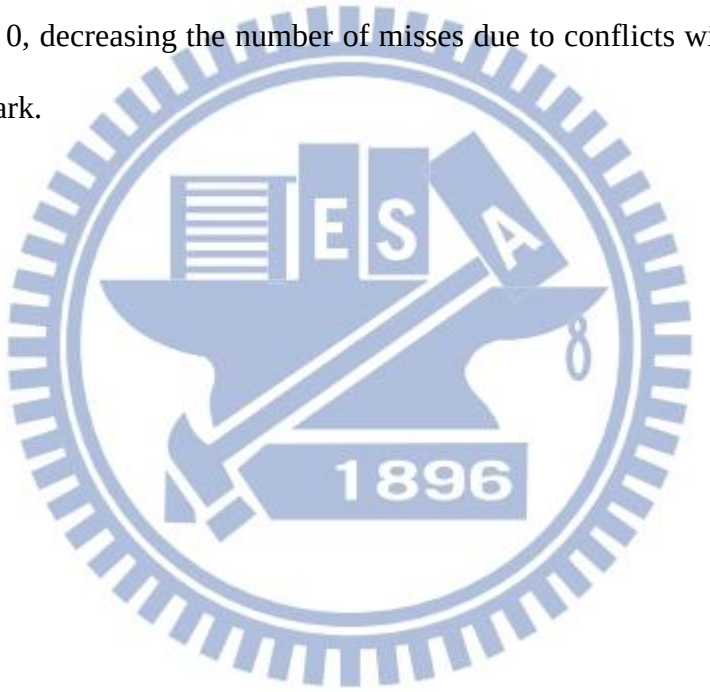


Figure 4: Average number of hits per stack distance position for each set in the cache for the gap benchmark

If two copies of this benchmark were run concurrently with a shared partition, there would be a large number of conflict misses around the sets numbered 500 (this would also be the case when there is no partitioning scheme). To counteract this, individual set index functions for each process can be used. By changing the set index function for one of the gap processes, there should be a reduction in the number of inter-process conflicts. We

propose using two modifications to the traditional cache set indexing function (the address modulo the number of sets) - inverting all index bits, and addition modulo the number of sets on the index. If the index bits were inverted, the higher numbered sets become the lower numbered sets. For two copies of the gap benchmark running concurrently, this would result in a marginal difference in the number of inter-process conflict misses, however may be beneficial for other applications with different access patterns. If an addition were performed on the set index (for example 512 (half the total number of sets)) for one copy of the gap benchmark that is running concurrently, the accesses near set 500 would move to set 0, decreasing the number of misses due to conflicts with the other copy of the gap benchmark.



III. Proposed Modifications

3.1 Set Utilization Based Cache Partitioning

3.1.1 Partitioning Algorithm

The overall framework of the proposed method for a dual core system is shown in Figure 5.

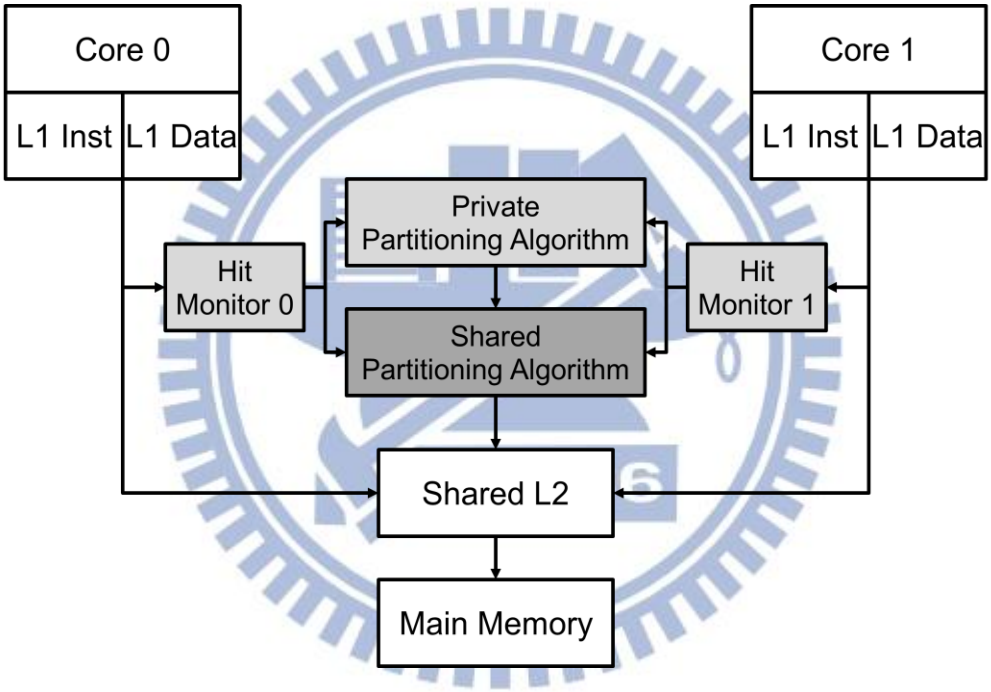


Figure 5: Design overview of proposed shared partitioning method for a dual core system, showing additional structures required.

Each processor has private L1 instruction and data caches that are connected to a shared L2 cache. A dynamic private partitioning scheme will add a monitoring circuit to gather stack distance information and hardware to calculate the partition sizes. Each processor is connected to a monitoring circuit. The monitoring circuit gains hit information for each processor as if the entire L2 cache were private to it. One possible implementation of this monitoring is to create what is termed an Auxiliary Tag Directory (ATD) [8]. The ATD is a

copy of the shared cache containing the tags but does not contain the actual cache block data. The private partitioning algorithm allocates all the ways in the L2 cache between each processor. Note that the ATD is used to gather stack distance information dynamically. A static partitioning method would require a similar structure to perform the same function using memory access profile information of an application. The proposed method adds hardware to calculate the size of the shared partition. The shared partitioning algorithm takes the private partition size information along with the stack distance information and determines 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.

3.1.2 Private Partition Size Determination

In general, any way-based partitioning algorithm (static or dynamic) 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 dynamic partitioning algorithm aims to maximize the total reduction in cache misses. Further details on the algorithm can be found in [8]. Although the UCP method is used, our method is complimentary to other partitioning algorithms with alternate cost functions (for example based on fairness [3]).

3.1.3 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. If the hit monitors monitor the number of hits per stack distance position for each set, no additional hardware is needed. However if there is a global stack distance position hit counter for all sets, additional hardware is required. An additional used bit for each stack distance position for

each set can be added. 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.

3.1.4 Sharing Threshold Determination

The threshold for determining whether a way in a private partition should be shared or not can be determined either offline or dynamically. Offline determination requires simulation of a system running a number of different benchmark sets with different thresholds. The threshold that provides the best average performance can then be chosen. While this method can make the hardware implementation simpler, it cannot adjust to changes in program behavior.

Dynamically determining the sharing threshold will increase the hardware cost, but should be able to provide better performance. The algorithm for determining the sharing threshold is based on observing the past trend in the overall miss rate of the shared cache for different threshold values. Both previous and current miss rates for different thresholds are kept. If the first threshold is lower than the second threshold and has a higher miss rate, this indicates increasing the threshold may be beneficial so the next threshold chosen is higher than the second threshold. Similarly, if the miss rate of the first threshold is less than the second threshold, it indicates decreasing the threshold may be beneficial so the next threshold chosen is lower than the first threshold. This is shown diagrammatically below in Figure 6.

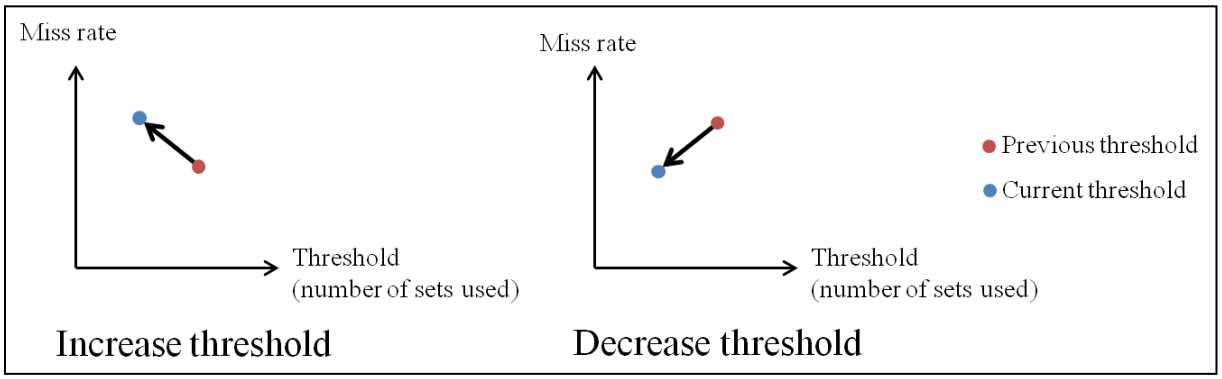


Figure 6: Dynamic sharing threshold determination.

The choice of how much to increase or decrease the sharing threshold is a parameter that can be decided by the implementer. Too fine a granularity will result in slow adjustment to any changes in the behavior of the applications in the shared partition.

3.1.5 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 problem with the first method is that we may not want to share a way with all cores if the *combined* set utilization is too high. It may be better to share the way with a subset of cores. The second method makes two adjustments - the use of combined set utilization in sharing determination, and the sharing of a to-be-shared way with a selected number of other processors. We term this method the ShareSubset algorithm.

The implementation of this method is more complex and complicated as a result of two problems. The first is the selection of cores to share the way with. To optimally assign the cores is a form of the 0-1 knapsack problem and requires a dynamic programming approach to calculate the optimal solution. In order to reduce calculation time, it is possible to use a heuristic and take a greedy approach to core selection. Cores can be ordered based on the optimization goal of the private partitioning algorithm (e.g. benefit gained from an additional way) and selected until a core would exceed the combined set utilization threshold.

The second problem is that as the number of cores sharing the cache increases, the number of partitions increases exponentially (at a rate of $2^{\#cores}-1$). Partition here is

defined as a part of the cache that is only allowed access by a subset of cores. Storing the partition sizes and also determining which blocks belong to each partition is not scalable so it is possible to use an approximation able to scale with a large number of cores. Instead of multiple different combinations of shared partitions, a single shared partition can be created. This single shared partition only allows each core to use a limited number of ways in it. This reduces both partition size storage overhead and simplifies the modifications required to the replacement policy (further details provided in below in the Augmented LRU Policy section).

Pseudo code for this method is shown 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 the optimization goal. The newly shared way is shared with other cores until the

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
```

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.

3.1.6 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 in the shared cache 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. This owner tag is already necessary for a private partitioning method that modifies the replacement policy to enforce the partition sizes. 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 [8] is used to evict cache blocks on demand rather than evicting all blocks belonging to a processor that are over its newly allocated limit.

3.1.7 Storage Overhead

The additional hardware requirements are minimal given a private partitioning algorithm that already uses an auxiliary tag directory. There are two main sources of overhead - the monitoring circuit and the modifications to the shared cache. If stack distance position hit counters per set are available, no additional storage overhead is required for the monitoring circuit. If not, and using the UCP method as an example, for a 32-bit system the UCP

method requires an additional storage overhead of 1920 bytes per core for a 1MB 16-way cache with 32 sets monitored. Our method needs to add 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 and increase of 0.006% of the total cache size per core. For additional cores, this number is multiplied by the total number of cores.

The storage overhead for modifications to the shared cache consists of the additional owner tag needed for each block. For a 16-way set associative cache with 1024 sets shared between two cores, an additional bit is required per block thus the overhead would be 2048 bytes. This overhead is proportional to the number of blocks in the shared cache and the base-two logarithm of the number of cores sharing the cache. This owner tag may originally be required by the private partitioning method so there would be no additional overhead from our proposed shared partitioning method.

3.1.8 Effect on Latency

The effect of the proposed shared partitioning method on the latency of the system has two major sources - 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. This is shown below in Figure 7.

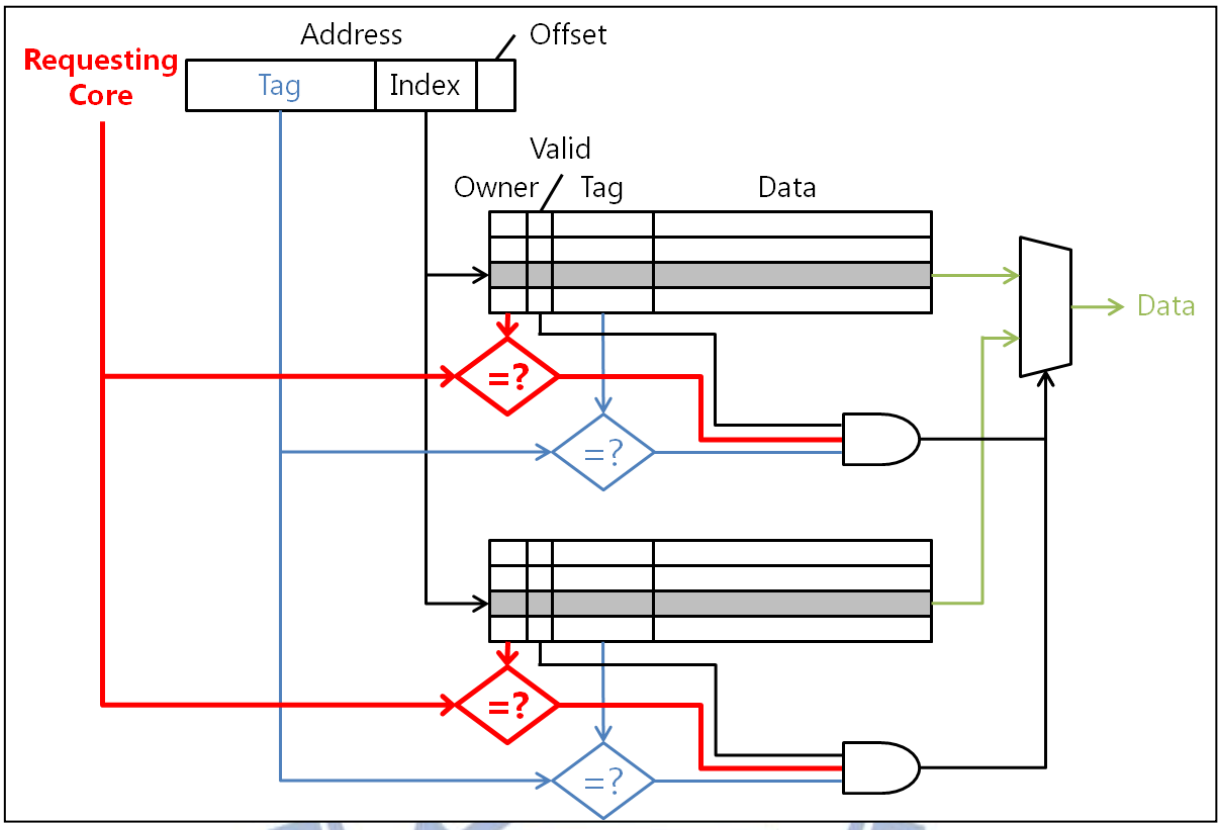


Figure 7: Cache organization when accessing the cache. Red/bolded parts indicate additional structures required by partitioning methods that impact the cache access time.

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.

3.2 Alternate Per Core Indexing Functions

Two modifications to the traditional address mapping function (the address modulo the number of sets) are proposed - inverting all index bits, and addition modulo the number of sets on the index. Each core is able to have its own individual indexing function, in a combination of inverted/non-inverted index bits and addition modulo the number of sets of

the index. These modifications are chosen so as to be low cost and low latency and also to preserve the clustered nature of any accesses. Bit inversion requires one level of NOT gates, while if the addend is chosen judiciously, the addition can be performed on the higher order bits only. For example, if there are 1024 sets, to add 512 only requires inverting the most significant bit of the index. To add 256 would require one XOR gate and one NOT gate for the two most significant bits of the index respectively (a two-bit adder).

One benefit of using a combination of bit inversion and addition is that it is generally scalable to any number of cores, although there may be effects on latency.

3.2.1 Indexing Function Determination

The choice of indexing function for each core is important to ensure performance does not deteriorate compared to the traditional address mapping function. One option is to randomly select index functions for core. This would require a large number of simulations to find a combination of index functions that perform well on average.

Another option is to dynamically adjust the index functions. The goal of the index function is to balance the accesses to cache sets to reduce inter-processor conflict misses. Therefore a measure of balance in the cache is needed. If the cache is out of balance, the index function of a core can be changed to attempt to improve the balance. To measure the balance we introduce two counters per core. Using the stack distance information from the private partitioning method, it is possible to detect the number of unique accesses per set and stack distance position. The counters determine whether the cache accesses are “top heavy” or “bottom heavy”. The counters increment when stack distance positions in higher numbered sets that have not been accessed before are accessed. Similarly they decrement for lower numbered sets. Therefore if an application has a lot of accesses in the top half of the cache, the counter will be positive, while a lot of accesses in the bottom half of the

cache will mean the counter is negative. The counters for two or more cores are then added together. If the result is close to zero, the accesses in the cache are predicted to be balanced and no adjustment of the indexing functions is necessary. If the result is either positive or negative, then an indexing function should be changed and the result compared to zero again. This is repeated until the best balance is found. Note that this may continue for a long time depending on the number of different indexing functions so a limit to the number of combinations searched can be used.

One thing of note is that balancing a private partitioning scheme will have no effect since there is no sharing of the cache (in actuality it may since repartitioning can be lazy resulting in temporary shared parts). This means the balance counters should just be applied to the shared part of the partition. Also, this means that the balance counters are applicable to shared caches with no partitioning (the size of the shared partition is the entire cache).

3.2.2 Dynamic Adjustment of Indexing Functions

A problem encountered when changing the index function for a core is that data placed in sets using the previous indexing function will no longer be able to found. Also, when searching for a block the address will be reconstructed incorrectly and can result in a hit when there should be a miss. One simple solution to this is to invalidate all the blocks using the previous indexing function when changing index functions. This solution increases the number of misses and lowers performance.

A better solution is to keep the data using the old index function in place, and use both index functions when looking for a match. To correctly reconstruct the address, each block can have an additional index function tag to indicate which index function was used. This increases storage overhead and can impact latency through the additional check for the correct index function tag.

If there are a large number of index functions it is impractical to search for blocks using all of them as it will consume too much power. Instead only recently used indexing functions can be searched - perhaps the current and the past or past two index functions.

One other issue that exists and becomes more obvious when not using all the past indexing functions is that data indexed by an indexing function not used anymore is unable to be found even though it is still in the cache. To solve this, when blocks are found using the old index function, they can be moved to the MRU position of the new set, and the LRU block of the new set moved to the LRU position of the old set.

3.2.3 Storage Overhead

If dynamic index function determination is used, an index function tag is required for each block in the cache, so if there are four different index functions and a 16-way set associative cache with 1024 sets (16384 blocks in total) the storage overhead would be 4096 bytes. In addition, balance counters are needed per core. These can be saturating counters so do not need to be large - perhaps between 1-2 bytes meaning 2-4 bytes overhead for the balance counters per core.

3.2.4 Effect on Latency

The alternate indexing function is chosen so as to require a minimal number of additional gates to implement and thus should be within a one cycle time envelope for an access. Also, as the shared cache is most likely second level or higher, even if the access time increases, it can be speculatively accessed during the first level cache access hiding any increased latency.

The determination of the index functions is not on the critical path of a cache access and can thus be performed in parallel. It does however need to provide the new index functions in a timely manner so cannot have too large a latency. When adjusting the index function

for a partition there may a slight delay as a multiplexor chooses the output of the correct index function to pass to the tag and data arrays. This will not affect the critical path as the additional delay will be too small to notice.

3.3 Other Modifications

Since a private partitioning scheme splits the cache into private partitions, only intra-processor misses occur. As such we can apply standard techniques to reduce the miss rate of a cache on a per-partition basis. Such techniques include varying the block size, varying the associativity, and changing the replacement policy.

3.3.1 Per-Partition Variable Block Size

Varying the block size can result in a decrease in compulsory misses for an application, but needs to be balanced with the increase in conflict and capacity misses. A brief overview of various methods to detect the appropriate block size is presented. One method is to have an external monitoring circuit (functionally identical to the cache but not containing data) snooping cache accesses calculating miss rates using different block sizes. Applied on a per partition basis, copies of each of these monitors will be needed for each partition, linearly increasing overhead.

Another option would be to have in cache monitoring, reserving certain sets in the shared cache for particular core and block size. This is not very scalable as a large number of sets will be needed for increasing numbers of cores and possible block sizes.

A third alternative would be to store a number of accessed addresses per core and get the number of hits for that address if the block size were varied. This is lower overhead than making a copy of the cache and able to scale with the number of partitions. However the issue is it does not take into account the effect of replacement policy. One way to deal with

that would be to remove the address from the address list when the block is evicted from the cache. However when the block size monitored is larger than the current block size, the address may be in the cache but could not fit if the block size were larger. For this case the predicted number of hits would be incorrect.

The block size can be adjusted either periodically or based on event. Such an event could include when the cache is repartitioned or when the monitored miss rate differential between the current block size and a monitored block size is greater than a threshold.

The biggest question related to variable block sizes is how they can be implemented, and if the cache can support that implementation on a per-partition basis or it can only work globally across the whole cache. Since the cache size is conceptually fixed (only changed on repartition and not by the variable block mechanism) to change the block size requires either a change in the number of sets or a change in associativity.

Conceptually combining two sets into one set will double the block size. This can be implemented with low cost by fetching multiple sequential blocks per memory request and putting them in different sets. This means block size can be adjusted using integer multiples of the minimum block size. It will however increase the number of memory requests, impacting performance. One issue is that the index function is unaware of the change in number of sets and may index into the middle of a large block. Physically the number of sets in the cache can stay the same, but the index function can be modified to ignore (set to 0) lower order bits when the block size increases, counteracting this issue. If the lower order bits are ignored this means the block size must be a power of two, which it commonly is anyway.

This method for varying the block size is easily adapted to a per partition basis. Each partition is associated with a block size, with the index function changing based on the

partition's block size. This may slightly affect the time to index a function since a lookup is needed. The stored block size is also used to determine how many blocks to sequentially fetch from the next level of the memory hierarchy. When repartitioning and combining partitions with different block sizes, nothing needs to be done as blocks will be overwritten on demand and are still able to be accessed by either core with their different index functions.

Changing the associativity is conceptually similar. The underlying structure of the cache in terms of number of sets and blocks and ways remains the same, it is the unit of addressing that changes. Conceptually, ways are combined to increase the block size. This can be done by again fetching multiple blocks on a miss, however this time the fetched block go in the same set so multiple blocks need to be evicted. Eviction needs extra time to find the correct block to evict and then use this to find the additional blocks that need to be evicted. This can be performed in parallel to evicting the first block and fetching the next block so has no impact on latency. Apart from this, there are no other major modifications needed compared to the original cache structure. Block size per partition is stored and used when fetching and evicting a block. Combining partitions with two different block sizes may cause an issue with choosing which blocks to evict and put in the cache since the number of blocks in a set may not be a power of two. This requires extra logic to deal with the situation.

3.3.2 Per-Partition Variable Associativity

Varying the associativity can result in a change in the number of conflict misses for an application. Methods to detect the desired associativity are similar to that detecting the desired block size. Again, an external monitoring circuit (functionally identical to the cache but not containing data) snooping cache accesses calculating miss rates using different associativities. In-cache monitoring, reserving certain sets in the shared cache for particular

core and associativity. This is not as practical unless the associativity is reduced and not increased from the baseline associativity. Similar to varying the block size, associativity can be adjusted either periodically or based on event.

Since the cache size is conceptually fixed, varying associativity can be achieved by either changing the block size or changing the number of sets. Changing the block size and associativity was described previously - combine ways. This only works to decrease the associativity from the baseline.

Changing the number of sets however, can both increase and decrease associativity. As partitions do not necessarily have an associativity that is a power of two, it is easiest to restrict changes in the number of sets in a partition to halving and doubling. This will double/halve the associativity of the partition respectively. Physically the cache will maintain the same structure, but conceptually the number of sets changes. This requires changes in both the index function and replacement policy to support. When increasing associativity, the number of LRU bits changes so the replacement policy must deal with this (perhaps by randomly choosing the LRU block of one of the sets). When looking for a hit or block to evict, the index function must search two or more sets - done by searching all combinations of the LSBs of the index that change. When decreasing the associativity, blocks that conceptually are in different sets share the same physical set. The replacement policy must deal with this by ensuring the correct blocks are evicted by checking the LSBs of the address tag. The biggest issue is what to do with existing data when increasing or decreasing the number of sets. The easiest way is to invalidate any of the data in the conceptually new portions. The data could be kept, however requires a large overhead to adjust and find data that can exist in new location (able to be found and will not incorrectly cause a hit).

Applying varying associativity by changing the number of sets is able to be applied on a

per-partition basis, as combining partitions with different associativities is similar to changing the associativity. The partitions can first match their associativity, then one partition readjust their associativity back with a different partition size.

3.3.3 Per-Partition Replacement Policies

Changing the replacement policy can result in a decrease in the number of conflict misses, specifically replacement misses. When implementing on a per-partition basis, partitions with different replacement policies can be combined at any time so the replacement policies need to be active concurrently and updating any tags needed for all accesses. On a miss, each replacement policy will select a block to evict and the final block to evict is chosen which replacement policy is currently in use by the partition. This means the replacement policy tags can not be reused increasing hardware cost. Also since all replacement policies are run concurrently for each access, power is increased.

Selection of the replacement policy to use can also be done in a similar fashion to the past two methods - having an external monitor with different replacement policies or an in-cache monitor. Miss rates can be recorded and replacement policy selection based on this.

IV. Related Work

4.1 Cache Partitioning

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 [1] 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 [8], 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 [9] 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 [10] method provides an improvement in performance, particularly Quality of Service. This scheme is also compatible with our proposed shared partitioning method and we anticipate additional improvements in performance if used together.

Adaptive Shared/Private NUCA Cache Partitioning [11] is a method similar to our proposed shared partitioning 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.

The reconfigurable cache mentioned in [12] describes a similar, albeit more simplified method of cache partitioning. They focus more on the hardware requirements and feasibility of implementing cache partitioning. Differences with the proposed cache partitioning method include the use of software vs. hardware for partition size determination, cache scrubbing vs. lazy repartitioning, and the use of L1 vs. a cache explicitly shared between multiple cores.

Peir et al. [13] describe a dynamic partitioning technique for a direct-mapped cache in which partitioning is done by grouping sets (termed a group-associative cache). In addition, underutilized cache blocks are detected based on the recency of their use (to attempt to implement a global LRU scheme) and prefetched data is placed in those blocks in the hope of increasing their utilization. This method of underutilization detection is somewhat similar

to our proposed method, but operates on a direct-mapped cache using what would be classified as a set-based granularity if the associativity were increased.

Adaptive set pinning [14] can be thought of as cache partitioning using a set granularity. Sets are allocated to processors based on the frequency of accessing a particular set, with each set having an owner. This scheme is more scalable than way based partitioning and our proposed method may be able to be extended to complement set pinning by detecting underutilized ways within allocated sets that can be shared.

Recent work has noted the poor scalability of having separate monitors for each core and methods have been proposed including In-Cache Estimation Monitors [15] and set-dueling [16] 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.

4.2 Cache Indexing Functions

Previous research on cache indexing functions has generally not focused on the interaction between processes in a shared cache and has focused on reducing conflict misses within a single process.

Rau [17] discusses the calculation of the index as the address modulo an irreducible polynomial. However Gonzalez et al. [18] show that there is a marginal advantage to choosing a polynomial mapping scheme over their own simpler bitwise XOR mapping.

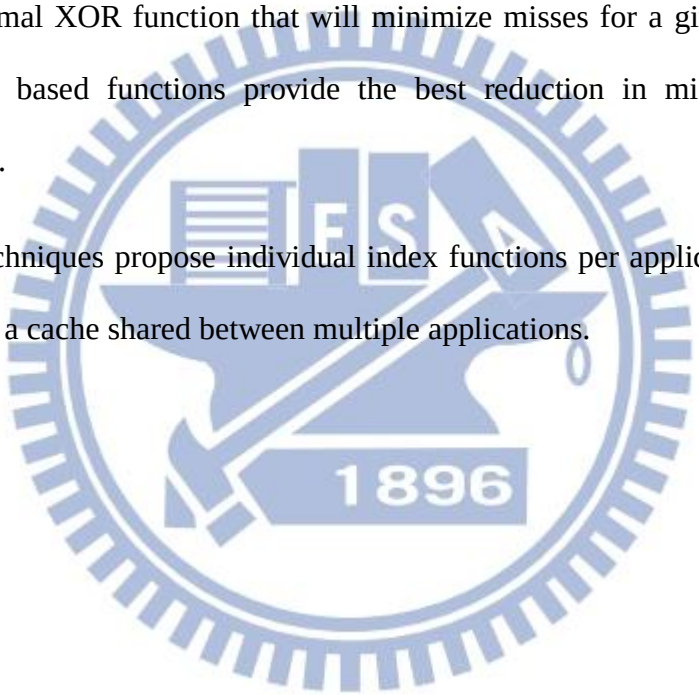
Kharbutli et al. [19] propose a fast implementation of an index function that uses the

address modulo a prime number.While improving performance, they recognize that the hardware cost and increase in delay means it is more suitable for higher level caches.

Zero Cost Indexing for Improved Processor Cache Performance [20] describes a heuristic to select address bits for the index given a program trace. For no cost (just selecting different bits for the index) it is able to reduce the miss rate, however requires a program trace, making it unsuitable in general for other applications.

In a similar manner, Vandierendonck and De Bosschere [21] present an algorithm to determine the optimal XOR function that will minimize misses for a given program trace finding that XOR based functions provide the best reduction in misses of the index functions surveyed.

While these techniques propose individual index functions per application, they do not address the case of a cache shared between multiple applications.



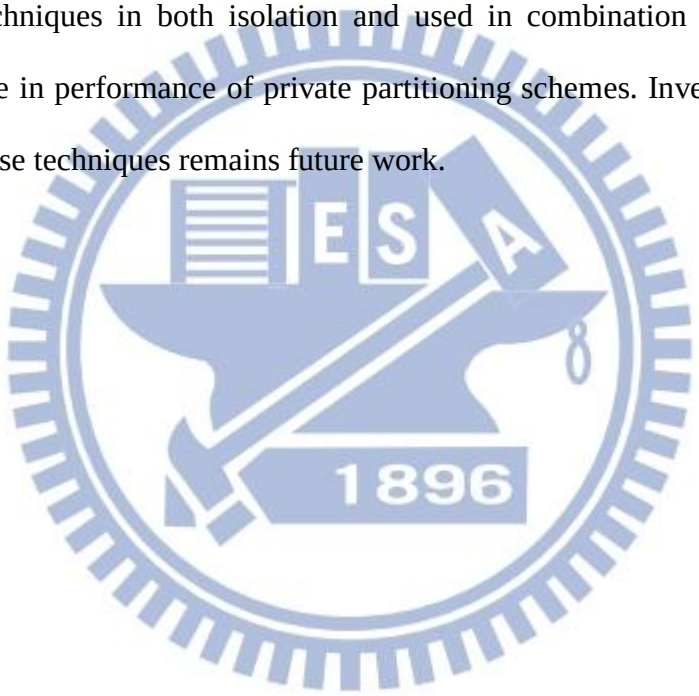
V. Future Work

Results for the individual methods and the methods combined have not been collected yet. The results can be gathered by simulation. Simulation can be performed using a full system simulator that is able to accurately simulate a multi-core processor and related subsystems including the cache and memory system.

One other technique to improve partitioning not described in detail here is partition size prediction. Cache partitioning methods generally collect runtime information on applications executing, making partitioning decisions on this information assuming past behavior will be a good indication of future behavior. As programs generally go through phases during execution in which memory accesses are quite similar, one could detect these phases and associate them with a partition size, similar in principle to a branch history table. In this way, instead of having a fixed repartitioning period, repartitioning would occur upon detection of a change in the execution phase of an application.

VI. Conclusion

Previous cache partitioning methods ignored the effect of the non-uniformity of cache set accesses upon the effectiveness of partitioning decisions. We proposed two methods to take advantage of this non-uniformity - the creation of a shared partition and individual cache indexing functions for each core. Also discussed was how varying block size, varying associativity, and changing the replacement policy can be applied in a private partitioning scheme. These techniques in both isolation and used in combination should be able to provide an increase in performance of private partitioning schemes. Investigation of actual performance of these techniques remains future work.



References

[1] 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.

[2] G. Suo, X. Yang, G. Liu, J. Wu, K. Zeng, B. Zhang, Y. Lin, "IPC-based cache partitioning: An IPC-oriented dynamic shared cache partitioning mechanism," Convergence and Hybrid Information Technology, 2008. ICHIT '08. International Conference on, pp.399-406, Aug. 2008.

[3] 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.

[4] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," In Proceedings of the 38th annual international symposium on Computer architecture (ISCA '11), pp. 57-68, 2011.

[5] D. T. Chiou, Extending the Reach of Microprocessors: Column and Curious Caching, Ph.D. thesis, MIT, 1999.

[6] T. Lee, H. Tsou, "A novel cache mapping scheme for dynamic set-based cache partitioning," Information, Computing and Telecommunication, 2009. YC-ICT '09. IEEE Youth Conference on , pp.459-462, 20-21 Sept. 2009

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] P. Ranganathan, S. Adve, N. Jouppi, "Reconfigurable caches and their application to media processing," Computer Architecture, 2000. Proceedings of the 27th International Symposium on, pp. 214-224, June 2000.

[13] J. Peir, Y. Lee, and W. Hsu, "Capturing dynamic memory reference behavior with adaptive cache topology," In Proceedings of the eighth international conference on Architectural support for programming languages and operating systems (ASPLOS-VIII). pp. 240-250, 1998.

[14] S. Srikantaiah, M. Kandemir, and M. J. Irwin, "Adaptive set pinning: managing shared caches in chip multiprocessors," In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII), pp. 135-144, 2008.

[15] 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.

[16] 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.

[17] B. R. Rau, "Pseudo-randomly interleaved memory," In Proceedings of the 18th International Symposium on Computer Architecture, pp. 74-83, 1991.

[18] 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.

[19] M. Kharbutli, Y. Solihin, and J. Lee, "Eliminating conflict misses using prime number-based cache indexing," IEEE Transactions on Computers, vol. 54, no. 5, pp. 573-586, May 2005.

[20] T. Givargis, "Zero cost indexing for improved processor cache performance," ACM Transactions on Design Automation of Electronic Systems, vol. 11, no. 1, pp. 3-25, Jan. 2006.

[21] H. Vandierendock, and K. De Bosschere, "Constructing optimal XOR-functions to minimize cache conflict misses," ARCS, pp. 261-272, 2008.