2-16-2010

COSET: Cooperative Set Last Level Caches

Dongyuan Zhan
University of Nebraska-Lincoln, dzhan@cse.unl.edu

Hong Jiang
University of Nebraska-Lincoln, jiang@cse.unl.edu

Sharad Seth
University of Nebraska-Lincoln, seth@cse.unl.edu

COSET: Cooperative Set Last Level Caches

Dongyuan Zhan, Hong Jiang, Sharad C. Seth
Department of Computer Science & Engineering,
University of Nebraska – Lincoln, Lincoln, NE 68588
{dzhan, jiang, seth}@cse.unl.edu

Abstract

The speed gap between processors and DRAM remains a critical performance bottleneck for contemporary computer systems, which necessitates an effective management of last level caches (LLC) to minimize expensive off-chip accesses. However, because all sets in a conventional set-associative cache design are statically assigned an equal number of blocks, the LLC capacity utilization can drastically diminish when the cache actually exhibits non-uniform capacity demands across the sets. To reveal the wide existence of set-level non-uniformity of capacity demand in real applications, this technical report first establishes an accurate metric for measuring individual sets' capacity demands by developing a group of mathematical models. Then, the report presents a last-level cache design[1] called COSET (COoperative SET) L2 cache that identifies the capacity needs of individual sets based on the new metric, dynamically couples two sets with complementary capacity demands, and enables the set with a higher resource demand to utilize the capacity of its coupled set to reduce conflict misses. Our simulation study on 6 selected SPEC CPU 2000 benchmarks shows that the COSET L2 cache achieves a MPKI of as low as 0.383 and 0.781 on average normalized to the standard LRU cache, outperforming the state-of-the-art approach SBC that has the best and average performance results of 0.585 and 0.867 respectively.

1. Introduction

As the memory wall continues to limit processor performance, judiciously architecting and managing on-chip last level L2 **c**aches continues to play a critical role in bridging the speed gap between processors and memory subsystems. Since the L2 caches are typically built on the conventional set-associative basis, all sets are statically assigned the same number of cache blocks. However, the static set-associativity cannot effectively minimize the overall cache misses due to its inflexibility in adjusting block allocation to the specific needs of individual sets. In several prior studies researchers have identified in workload characterization that L2 accesses can be non-uniformly distributed over all sets, which is also referred to as set-level access non-uniformity. With the implicit assumption that a higher set-level access count implies greater set-level capacity demand, their studies have attempted to diffuse L2 accesses across the entire L2 sets so as to generate equal set-level capacity demands. Seznec's skewed associativity [1], Kharbutli's prime-based cache indexing [2], and Qureshi's V-way cache [3] are all such approaches. Their basic assumption, however, may not always hold true, because there is no direct correlation between a set's access count and its working set size. A simple argument for the inaccuracy of "access count" in measuring set-level capacity demand can be stated as follows: if a set is currently getting a large number of accesses but with the accesses exhibiting good temporal locality, these accesses will only touch a small working set; furthermore, if the working set size is less than the set associativity, all accesses will eventually result in cache hits rather than conflict misses, which is actually worth preserving in rather than evicting from cache.

Therefore, simply alleviating set-level access non-uniformity may not be the best way of reducing conflict misses, because a set's access count is not a direct indicator of its working set size. A recent proposal by Rolan et al. [4] has attempted to measure a set's capacity demand by counting the "saturation level", a metric defined as the difference between a set's miss and hit counts. In their study, if a set has experienced more misses than hits during a period, the set is considered to have a higher saturation level and thus assumed to require more capacity than it currently possesses; otherwise, the set is considered less saturated and its capacity is assumed to be underutilized. With this assumption, they propose the *Set Balancing Cache*[2] (SBC) scheme that associates two sets with complementary saturation levels and enables the set with a higher saturation-level value to place victim blocks in the other set. In [4], the SBC scheme has been evaluated to outperform the V-way [3] and DIP [5] caches. The proposed "saturation level" used by SBC, however, cannot differentiate a set's compulsory misses from its conflict misses, rendering the metric less accurate in measuring the set's real capacity need. For instance, if a set is receiving more misses than hits just because it has excessive compulsory misses, which means that the set is a streaming-like set, extending the set's capacity will not contribute to miss reduction at all.

Although using either a set's "access count" or "saturation level" is not very accurate according to the analysis above, both metrics still lead to more effective set-level cache management approaches than the cache designs with a fixed set-associativity. This leads us to the belief that a set-level cache management approach with higher performance can be designed if a more accurate set-level capacity demand metric can be adopted, which serves as the fundamental objective of this study. Unlike previous studies that mainly focus on the architectural design of L2 caches, this study first develops a group of accurate mathematical models, based on which a new hardware-based metric of set-level capacity demand is proposed. The basic idea behind and novelty of the new metric is to directly measure the con-

[1] Without loss of generality, L2 is assumed to be the on-chip LLC in this study.

[2] In this technical report, the term SBC is dedicated to the *Dynamic Set Balancing Cache* scheme in [4].

flict miss reduction rate for a set as if its capacity were extended (by a shadow set), which sets it apart from all existing set-level capacity demand metrics. Then, we adopt the new metric to evidencing and characterizing the set-level non-uniformity of capacity demand in re programs. Finally, based on the new metric, we propose a new L2 cache design, called the *COoperative SET* (COSET) L2 cache, which dynamically couples two L2 sets with complementary set-level capacity demands and allows the set with a higher need, called a *taker set*, to spill victim blocks to the other set, called a *giver set*. In addition to adopting the novel and more accurate set-level metric, COSET is significantly different from SBC in that COSET adopts a background search algorithm to find a globally optimal giver set for coupling, and incorporates a feedback loop to control the quantity of victim blocks that can be received by the giver set in response to its real-time capacity demand.

The rest of this report is organized as follows. Section 2 introduces the research motivation. Section 3 elaborates on the design issues of our proposed COSET L2 caches. Section 4 shows the experiment setup used for evaluation and Section 5 provides an analysis of the obtained results. Related work is discussed in Section 6 and the technical report concludes with a summary in Section 7.

2. Motivation

This section develops a set of mathematical models to obtain our new metric and guide workload characterization, which motivates the study presented in this technical report.

2.1 Quantification of Set-Level Capacity Demand

We start with defining the notations and terms used in the discussion in Table 1.

Table 1. Glossary of Notation and Terms Used

Symbol	Annotation
N	The total number of sets in an L2 cache
A	The number of blocks (associativity) owned by a set, $0 \le A < \infty$
S	The index of a set, $0 \le S \le \mathrm{N}-1$
I	A fine-grained sampling interval based on workload characterization
$miss\_count(S,I,A)$	The number of misses on set S with A blocks during the sampling interval I
$hit\_count(S,I,A)$	The number of hits on set S with A blocks during the sampling interval I
$blocks\_required(S,I)$	The number of blocks required by set S during the sampling interval I
$A_{threshold}$	A value of associativity large enough to approximate ∞
$A_{baseline}$	The associativity (integral power of 2) of the baseline private L2 cache
M	The number of buckets/sub-ranges (explained in Section 2.1.2) within $[1, A_{threshold}]$
$bucket_j$	The j^{thh} bucket, which is in the sub-range $[\frac{(j-1)\cdot A_{threshold}}{M}+1, \frac{j\cdot A_{threshold}}{M}]$
$SF(S,I,bucket_j)$	A membership function used to indicate if the number of blocks required by set S is categorized into the j^{th} bucket during interval I
$size\_bucket_j(I)$ $1 \le j \le M$	The size of the j^{th} bucket during interval I

2.1.1 Quantifying Set-level Capacity Demand

Since a cache set can be treated as an array of blocks, under a fixed block size, we can use the number of blocks in a set to measure the amount of cache resource possessed by the set. Intuitively, if a set has enough blocks during a given time interval, there will be no capacity or conflict misses on the set, because these two kinds of misses happen only when the set resource is limited. Therefore, if we denote the capacity demand of a particular set during a given time interval as $blocks\_required(S,I)$, where S is the index of the set and I is the time interval of interest, we can define it as the minimum number of blocks required to resolve all capacity and conflict misses for the set during the interval.

We introduce another function, $miss\_count(S,I,A)$, which measures the number of misses on set S during interval I when S has A blocks. Under the LRU replacement policy that has the stack property [6], the following relationship always holds true: $miss\_count(S,I,0) \ge miss\_count(S,I,1) \ge \cdots \ge miss\_count(S,I,\infty)$. From this property, we can also infer that $miss\_count(S,I,A)$ is monotonically non-increasing for the given S and I when only A increases. Ideally, if set S could get an infinite number of blocks ($A=\infty$) during interval I, then there would be no capacity or conflict misses on the set. At the other extreme, if set S had no blocks at all ($A=0$), all accesses to the set during interval I would miss. Consequently, $miss\_count(S,I,\infty)$ is equal to the number of compulsory misses on set S during interval I, while $miss\_count(S,I,0)$ is equivalent to the number of accesses to set S during interval I.

On the other hand, during interval I, if set S's capacity demand is satisfied, which means that set S gets as many blocks as $blocks\_required(S,I)$, then only compulsory misses can happen to set S. Thus, we give a quantitative definition of $blocks\_required(S,I)$ in Formula (1) below.

$$blocks\_required(S,I) = \min A \quad s.t.\ miss\_count(S,I,A) - miss\_count(S,I,\infty) = 0 \tag{1}$$

Since it is impractical to measure $miss\_count(S,I,\infty)$ when the set associativity A is ∞, and also because the function $miss\_count(S,I,A)$ is monotonically non-increasing for the given S and I when only A increases, we can use a sufficiently large but finite number $A_{threshold}$ to approximate ∞. Then, we can use Formula (2) below to quantify the capacity demand of a set.

$$blocks\_required(S,I) = \min A \quad s.t.\ miss\_count(S,I,A) - miss\_count(S,I,A_{threshold}) = 0 \tag{2}$$

Alternatively, since $miss\_count(S,I,0)$ is equivalent to the number of accesses to set S during interval I, the total number of hits on set S that has A blocks during interval I (denoted as hit$_count(S,I,A)$) can be expressed as hit$_count(S,I,A) = miss_count(S,I,0) - miss_count(S,I,A)$. Therefore, Formula (2) can be converted to Formula (3):

$$blocks\_required(S,I) = \min A \quad s.t.\ hit\_count(S,I,A) - hit\_count(S,I,A_{threshold}) = 0 \tag{3}$$

Practically, Formula (3) is more convenient to use than Formula (2), because it is much easier to locate a position in the LRU stack when an access to a set is a hit [5]. Equivalently, $hit\_count(S, I, A)$ is actually the total number of hits on the LRU positions that are less than or equal to A on set S during interval I.

2.1.2 Characterizing Set-Level Non-Uniformity of Capacity Demand

From the aforementioned analysis, we can infer that $blocks\_required(S, I)$ is in the integer range $[1, A_{threshold}]$. Without loss of accuracy, we divide the integer range $[1, A_{threshold}]$ into M sub-ranges (a.k.a., buckets) of equal length $\{bucket_1, bucket_2, \ldots, bucket_M\}$, where $bucket_j = \left[\frac{(j-1)\cdot A_{threshold}}{M} + 1, \frac{j\cdot A_{threshold}}{M}\right]$ for $1 \le j \le M$. Then, for a given interval I, set S is said to be categorized into $bucket_j$ if and only if the value of $blocks\_required(S, I)$ is in the integer range $\left[\frac{(j-1)\cdot A_{threshold}}{M} + 1, \frac{j\cdot A_{threshold}}{M}\right]$. Further, because any two adjacent buckets do not intersect, the value $blocks\_required(S, I)$ will be in one and only one bucket. Therefore, we can differentiate two cache sets in terms of their individual capacity demands if their $blocks\_required(S, I)$ values belong to different buckets. Here, we restrict both $A_{threshold}$ and M to be integral powers of 2.

To identify if set S is categorized into the j^{th} bucket during interval I, we can define a membership function $SF(S, I, bucket_j)$ to indicate if set S has a capacity demand that is in the range of $bucket_j$ during interval I, which is formulated in Formula (4) below:

$$SF(S, I, bucket_j) = \begin{cases} 1, & \text{if } blocks\_required(S, I) \in bucket_j \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

For all of the N sets in an L2 cache, we are interested in knowing how many sets are categorized into each one of the M buckets during the sampling interval I, because any two sets that are categorized into different buckets will show different set-level capacity demands. Here, we normalize the number of sets that are categorized into the j^{th} bucket during interval I by the total number of sets N, define it as *the size of the bucket* for that interval, and denote the value as $size\_bucket_j(I)$. The formal definition of $size\_bucket_j(I)$ is shown in Formula (5) below.

$$size\_bucket_j(I) = \frac{\sum_{S=0}^{N-1} SF(S, I, bucket_j)}{N} \quad (1 \le j \le M) \quad (5)$$

In summary, we can characterize the set-level non-uniformity of capacity demand for all of the N sets in an L2 cache by using Formula (5).

2.2 Methodology of Workload Characterization

We experiment on all 26 SPEC CPU 2000 benchmarks [7] using the *sim-cache* tool of Simplescalar [8], and analyze the set-level capacity demand distributions of their L2 caches. The configurations of L1 and L2 caches are listed in Table 4 in Section 4. Specifically, there are 1024 sets in the L2 cache (N=1024). All of the benchmarks are executed with the *reference* data inputs. For each benchmark, we fast forward the execution by 6 billion cycles and then simulate the caches until 1000 sampling intervals of which each contains 100K L2 accesses are encountered. Therefore, the variable I is in the range [1,1000]. Within a sampling interval I, for an L2 set S, we sample the number of hits on set S at each LRU position A that is less than or equal to $A_{threshold}$, and then find the minimum A (a.k.a. $blocks\_required(S, I)$) such that $hit\_count(S, I, A) = hit\_count(S, I, A_{threshold})$), where $A_{threshold}$ is assumed to be $2A_{baseline}$ in this study.

Since $A_{threshold}$ is assumed to be $2\ A_{baseline}$ ($A_{baseline} = 16$) in this study, we divide the entire range $[1, A_{threshold}]$ into 8 buckets {[1,4], [5,8], …, [29,32]}. Then, for all 1024 sets and 1000 sampling intervals, we can obtain the normalized size of each bucket, $size\_bucket_j(I)$ for $1 \le j \le 8$, which is actually the distribution of set-level capacity demand for all L2 sets during the sampling period.

2.3 Conclusions on Workload Characterization

To summarize, we find that among the 26 SPEC2000 benchmarks, there are 7 applications (*ammp*, *apsi*, *galgel*, *gcc*, *parser*, *twolf*, *vortex*) that show strong set-level non-uniformity of resource demand. Figures 1 - 3 illustrate the distributions of set-level capacity demand for the three applications, among which *ammp* and *vortex* show strong set-level non-uniformity of capacity demand but *applu* does not. In Figure 1 - 3, the 8 legends on the right side of the figure represent the 8 buckets, the *x* axis shows the 1000 sampling intervals, and the *y* axis shows the distribution breakdown for the 8 buckets.

For instance, although both *ammp* and *vortex* have been shown to benefit from additional cache resource in previous research [9], Figures 1 and 2 clearly indicate that both of them exhibit significant set-level non-uniformity of capacity demand. For *ammp*, about 40% sets require only 1 - 4 blocks during the entire sampling period. For *vortex*, from the sampling interval 405 to about 792, about 15% sets require only 1 - 4 blocks, about 9% sets require 5 - 8 blocks, and over 7% sets require 9 - 12 blocks. In contrast, for the streaming application *applu*, almost all sets require only 1 - 4 blocks during the entire sampling period.

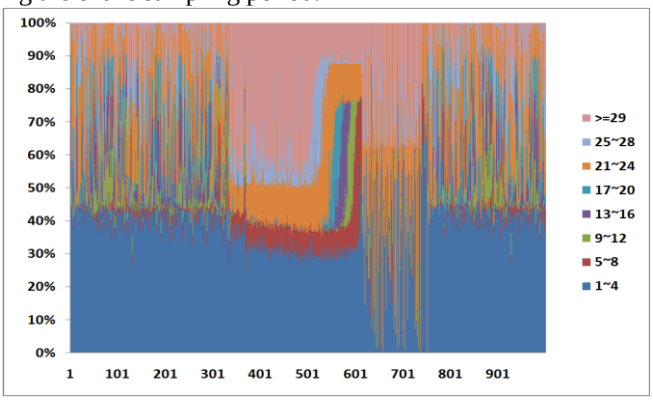

Figure 1. Distribution of Set-level Capacity Demand for *ammp*

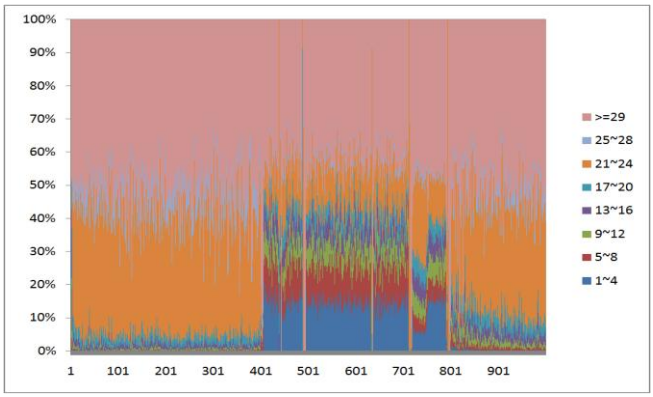

Figure 2. Distribution of Set-level Capacity Demand for *vortex*

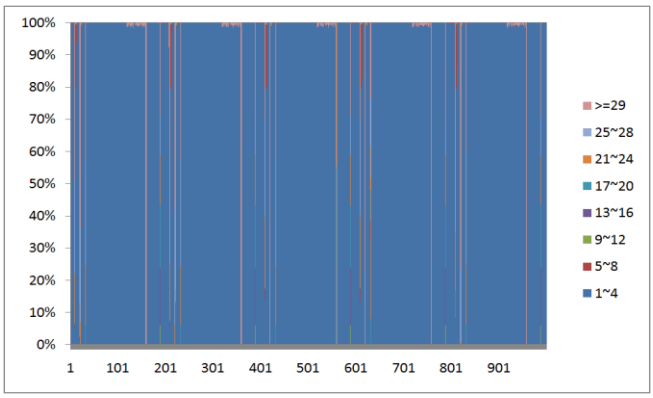

Figure 3. Distribution of Set-level Capacity Demand for *applu*

3. The COSET Architecture

COSET is designed to exploit the fine-grained set-level non-uniformity of capacity demand, clearly evidenced in the above experimental study by us and shown in other studies, to enhance the L2 cache performance. The COSET L2 cache design aims to achieve two critical goals: (1) identifying the capacity demand for each L2 set, and (2) coupling pairs of sets with complementary set-level capacity needs for spilling and receiving to significantly reduce cache miss rate.

Figure 4 provides an architectural view of the COSET L2 cache. The COSET *cache controller* accepts access requests from the upper L1 caches. Then, the controller looks up the requested block in the *tag store* to see if the block is present in the L2 cache. There can be two scenarios if the requested block is on-chip: the block is either in its native set with the same index as indicated in the block's physical address or a cooperative set with a different index. In the latter scenario, the controller needs two accesses to reach the cooperative set so as to find the block. Then, the requested block is forwarded to its native set if it is found or otherwise fetched from DRAM. Meanwhile, the *set-level capacity monitor* is operated to capture the dynamic information of individual sets' capacity needs and feed it back to the cache controller. Based on the information, the controller couples two sets with complementary capacity needs and controls the spilling and receiving between the coupled sets.

The *set-level capacity monitor* consists of as many *shadow sets* as the L2 cache sets in the tag or data stores, and a one-to-one correspondence is maintained between a shadow set and an L2 set with the same index. The shadow set is intended to monitor the capacity demand of the corresponding L2 set for its evicted blocks that will be accessed again. In addition, there is a per-set saturating counter associated with each shadow set. The design and working principles of a shadow L2 set will be elaborated in Section 3.1, and an overhead analysis of this organization appears in Section 3.5.

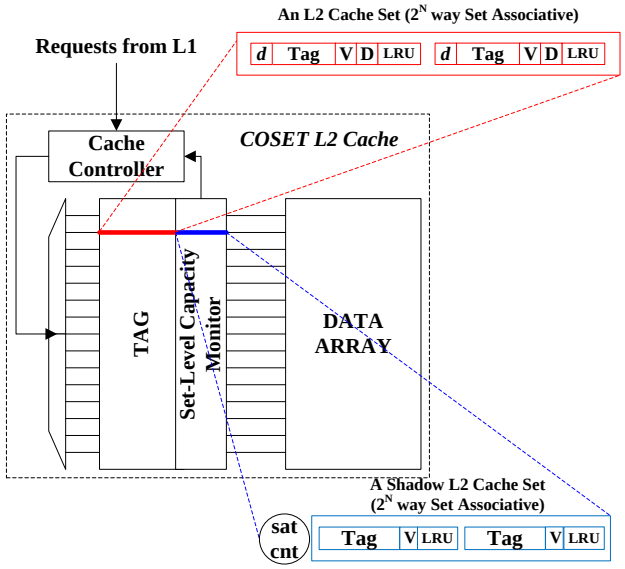

Figure 4. An Architectural View of the COSET L2 Cache[3]

3.1 Identifying Taker and Giver Sets

This subsection details the structure of *set-level capacity monitor* (shown in Figure 5), defines its operations and elaborates how the set-level capacity demand is monitored.

3.1.1 The Set Structures of the Tag and Monitor Stores

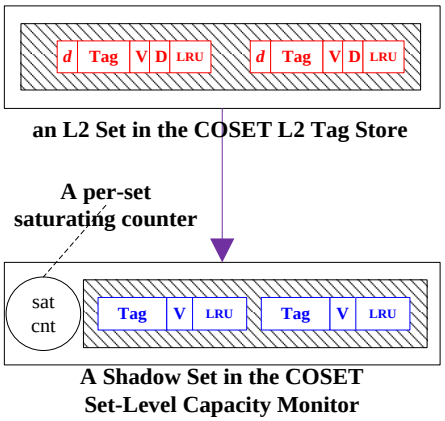

Figure 5. An L2 Set and its Corresponding Shadow Set

Since the new metric directly measures the (conflict) miss rate reduction for a set as if its capacity were extended (via a shadow set), the *Set-Level Capacity Demand Monitor* is critical in realizing this idea because it takes advantage of

[3] The schematic area does not necessarily reflect the physical area.

shadow sets and per-set saturating counters to estimate a set's expected miss rate reduction with the additional VIRTUAL capacity provided by the corresponding shadow set. Figure 5 shows the structures of a set in the tag store and the corresponding shadow set respectively. A tag store entry has all the usual required fields (e.g., the *tag*, *valid*, *dirty* and LRU), except that each entry is augmented with a single bit "*d*" indicating if the block is a native block of the current set ($d = 0$) or a cooperatively cached victim block from another set ($d = 1$). On the other hand, each entry of a shadow set contains such fields as *tag*, *valid* and LRU and is used to record the "shadow" tag field of evicted native blocks from the corresponding LLC set. Then, the set-level capacity demand can be reflected by a per-set *k*-bit saturating counter in the monitor store, as will be detailed in Subsection 3.1.3.

3.1.2 The Operations on Shadow Sets

There are three essential operations on a shadow set: (1) if a native block is evicted from its native L2 set, the corresponding shadow set in the monitor store will retain the tag field of the victim line in one of its entries and set it valid; (2) the shadow set maintains its own independent LRU ranking for all of its valid entries and uses the ranking for replacement; (3) if there is an access miss on a native block in an L2 set, the corresponding shadow set will be looked up to check if the tag field of the requested block is present in a valid shadow set entry. Additionally, it is required that the shadow set entries be strictly exclusive with the native blocks in the corresponding LLC set in terms of their tag fields. Therefore, if a previously evicted block with its tag present in the shadow set is revisited by the owner set, two operations must be performed: (1) the shadow entry that has the target tag needs to be invalidated after the corresponding block enters the real set; (2) a hit on the shadow set is signaled to operate its saturating counter.

3.1.3 Monitoring Set-Level Capacity Demand

If an L2 set and its corresponding shadow set have the same associativity, the L2 and shadow sets implicitly form two buckets as defined in Section 2. Then, we can use the per-set saturating counter to monitor the set-level capacity demand, based on which complementary pairs of set-level takers and givers are identified and coupled for spilling and receiving.

Since an L2 set and its shadow set form two buckets, according to Formula (3), we can use the ratio σ (defined in Formula (6) below) to project the potential performance benefit in terms of hit rate increase if the capacity of the L2 set were to double with respect to the number of cache blocks. If σ is greater than a predefined threshold $1/p$, where p is an integer, we claim that doubling the capacity of the L2 set can lead to an increase in the hit rate by $1/p$. This is because $\sigma > 1/p$ is equivalent to the relationship in Formula (7) below.

$$\sigma = \frac{\#hits\ (\ on\ the\ shadow\ set\ )}{\#hits\ (on\ thhe\ L2\ set) \quad + \quad \#hits\ (\ on\ the\ shadow\ set\ )} \tag{6}$$

$$\#hits(on\ the\ shadow\ set) - 1/p * [\#hits(on\ the\ L2\ set) + \#hits(on\ the\ shadow\ sets)] > 0 \tag{7}$$

To implement this idea, we define operations on a saturating counter as follows (also shown in Figure 6): (1) every hit on the shadow set increments the saturating counter by 1; (2) after every p hits to the private or shadow sets, the saturating counter is decremented by 1. Then, the outcome of the two operations can be reflected by the MSB (*most significant bit*) of the saturating counter. This is shown for an example in Figure 6: if a k-bit saturating counter is initialized to the value $2^{k-1} - 1$, which means that all bits except the MSB of the counter is set to one, a one-valued MSB of the counter indicates that the L2 set has a higher capacity demand than that provided by its local L2 cache, and that doubling its capacity can potentially lead to an increase in hit rate by at least $1/p$.

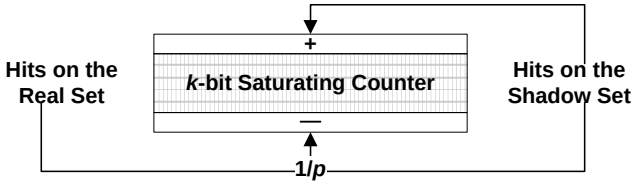

Figure 6. The Operation on a Saturating Counter

3.2 Coupling Two Sets for Spilling & Receiving

As described above, we can differentiate taker and giver sets by simply checking the MSB of each saturating counter. A 1-valued MSB indicates that extending the capacity of the corresponding set is beneficial; hence the set is regarded as a *taker set* that can significantly reduce its conflict misses when its capacity is extended. On the other hand, a 0-valued MSB denotes a *giver set* that needs less blocks than it currently possesses. Thus, COSET L2 cache can couple a taker set and a giver set so that the taker set can utilize part of the giver set's capacity to reduce conflict misses.

In [4], SBC adopts a *Destination Set Selector* (DSS) that keeps records of recently accessed giver sets by using a lightweight hardware heap. When an unassociated taker set gets accessed, SBC provides it with the giver set that has the least saturating level in the DSS structure for association. After the association is established, the SBC stores the association information in an *Association Table*. We argue that the DSS structure may not be appropriate in our COSET L2 cache design that uses a different set-level capacity demand metric from SBC. This is because, in SBC, only when a giver set is recently accessed can the set be considered a candidate for association. However, some sets become giver sets just because they are seldom accessed during a certain period. Such giver sets are not likely to be selected as candidates by DSS in SBC. Therefore, based on the principle of divide and conquer, we use a per-region *finite state machine* (FSM) to exhaustively search the best uncoupled giver set in each cache region that corresponds to a page color [10]. Since this search needs not be in the critical path of the COSET operations, it can be done in the background during light load. Then, a globally optimal uncoupled giver set is selected

among the regional best to satisfy the coupling request. We believe that this background search algorithm can always find a globally optimal uncoupled giver set for coupling, because there are typically tens of cycles' intermissions between two consecutive L2 accesses as a result of L1's filtering effect, and also because consecutive L2 accesses are more likely to be confined to a cache region (corresponding to a page color) as a result of the access locality. More accurate analysis will be carried out in our further research.

Now consider a block is missed in its native L2 set. If the set has previously evicted any blocks to a coupled giver set, as indicated in the corresponding *association table* entry, the COSET cache controller will signal a retrieving request to the coupled set. In response, the block is forwarded to the original native set if it is found in the coupled set; otherwise, it is fetched from the DRAM.

3.3 Spilling and Receiving Control

Unlike SBC that allows a taker set to continuously evict blocks to its coupled set, our COSET cache imposes some restrictions on the spilling and receiving process for any pair of coupled sets. This is because a giver set can be overwhelmed if the eviction from the taker set is too frequent. However, whether or not a giver set is overwhelmed can be easily detected by checking the MSB of its corresponding shadow set. If a previously 0-valued MSB turns 1, it means that the set might have been overwhelmed by another set's spilling or it has changed its role from a giver set to a taker set. The set-level capacity monitor returns such information to the cache controller to form a feedback loop as depicted in Figure 4. With the feedback loop, only when a set has a 0-valued MSB in its corresponding shadow set can it receive victim blocks from its coupled taker set.

3.4 Decoupling Two Sets

The decoupling takes place when a former giver set evicts all foreign blocks (with $d = 1$) within the set. After the decoupling, the two entries in the *association table* will be re-initialized to the two sets' original (native) indices respectively.

3.5 Hardware Overhead Analysis

In our COSET L2 cache design, the *set-level capacity demand monitor* and *association table* account for the vast majority of the hardware overhead in our design. Table 2 lists the length of each storage field in the COSET L2 cache.

Table 2. The Length of Each Field in the COSET Cache with the Configuration in Table 4

Field	**Length**
address length	32 bits
#(cache sets)	1024
set associativity	16
size(data block)	64 byte
length(tag field)	16 bits
v, *d*	1 bit each
LRU field	4 bits
log *p* (the length of the module *p* counter)	3 bits ($p = 8$)
k (= the length of the saturating counter)	4 bits

The overall storage overhand for both monitor store and association table is 4.13% by estimation. However, since the tag field in the shadow set does not affect the semantics of running threads at all, we plan to design a hash function for shadow caches to shorten their tag fields in our future work.

4. Evaluation

In our execution-driven experiment, we use the *sim-cache* tool in SimpleScalar [8] to evaluate the MPKI improvement of our COSET L2 design by using 6 SPEC CPU 2000 programs (*ammp*, *twolf*, *apsi*, *galgel*, *parser* and *vortex*) that show noticeable set-level non-uniformity of capacity demand in the workload characterization in Section 2.3. Since SBC has been evaluated in [4] and shown to outperform other well-known cache schemes such as V-way [3] and DIP [5], we only directly compare the performance of COSET against SBC in this study. Table 3 shows the *sim-cache* configuration used in the evaluation. A more thorough evaluation on and comparison with various state-of-the-art L2 design schemes and emerging workloads will be conducted in our near-future work, which has been planned to incorporate some subtle features capable of dynamic work set analysis into the updated version of the COSET design.

Table 3. The *sim-cache* Configuration

Address Bits	32
L1I/D	2 way, 32KB, 64B lines, write back
L2 Cache	16 way, 1MB, 64B lines, write back

5. Results and Analysis

For each instance of simulation, we fast forward the execution by 500 million instructions to bypass the initialization section of the programs, then warm up the cache modules with another 500 million instructions and finally continue the simulation for 1 billion instructions.

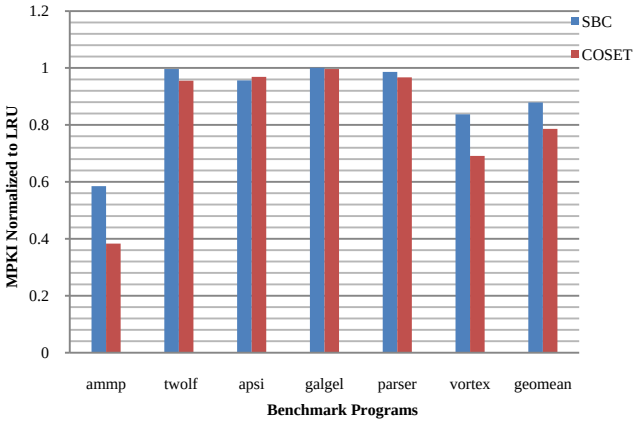

Figure 7. The Operation on a Saturating Counter

Figure 7 illustrates the MPKI (misses per 1K instructions) values of SBC and COSET that are normalized to the standard LRU cache. For the 6 benchmark programs that exhibit obvious set-level non-uniformity of capacity demand, our COSET L2 cache can improve the MPKI measure by 21.9% over the standard LRU while SBC only improves the

MPKI by 13.2%. Specifically, for *ammp*, the normalized MPKI of COSET is 0.383, most significantly outperforming that of SBC's which is 0.585. We can infer from the results that our new metric for capturing the set-level capacity demand and our optimal pairing of complementary taker-giver sets are more effective than the "saturation level" metric and the "*Destination Set Selector*" adopted in SBC.

6. Related Work

On-chip Level 2 caches have received extensive attention from the research community because of their critical role in reducing off-chip DRAM accesses. In the following, we briefly summarize some state-of-the-art research work that is closely related to our technical report.

- **App-Level Alternative Replacement Policies**

It has been proven that the LRU replacement policy mainly favors the programs with good temporal locality but it can make an L2 cache thrash when an application's working set is larger than the L2 capacity. To address this issue, there have been several studies that alter the traditional LRU replacement policy so as to respond to applications' working set sizes. Qureshi has developed the Dynamic Insertion Policy (DIP) [5] to adaptively support good and poor temporal behaviors in response to programs' runtime locality. Based on memory instructions' PC signatures, Liu et al. [11] have devised Dead Block Prediction approaches to identify "dead" blocks that occupy L2 cache capacity without any reuse before eviction. Once identified, the dead blocks can be replaced much earlier than through the LRU policy, making room for new lines requested either on demand or by prefetching. The schemes above focus on designing alternative replacement policies at the application level and are fundamentally different from our work, because ours is dedicated to utilizing the non-uniform distribution of capacity demands at the set level.

- **Page-Level Cache Management**

Page coloring is an OS-based approach for cache management which requires no modification of existing processor hardware. A recent proposal [10] called the *Run-time Operating system Cache-filtering Service* (ROCS) utilizes page coloring in reducing conflict misses in L2 caches. In ROCS, a small L2 cache region that a page can fit in is dedicated as a pollute buffer. ROCS identifies those pages that exhibit high miss rates as pollutants by polling processors' PMUs, then re-colors the pollutant pages and re-map them to the pollute buffer, thus protecting other pages with high hit rates from the pollutants' interference. Although the software-based page coloring schemes are very flexible in implementation, the re-coloring process can be quite expensive at runtime since re-coloring needs to flush off a page's cached blocks and migrate the page from one memory frame to another. Therefore, this software approach is only applicable to the programs with relatively long stable phases that can offset the re-coloring cost. The L2 cache design in our work, on the contrary, is a low-overhead hardware scheme that does not incur time-consuming software-based operations like page-recoloring. Besides, our scheme works at a finer granularity than the page level.

- **Set-Level Approaches**

Several prior studies have explored the non-uniform distribution of set-level accesses. Seznec's skewed associativity [1], Kharbutli's prime-based cache indexing [2], Qureshi's V-way cache [3] are all examples of such approaches. The above schemes use the "access count" as an indicator of set-level capacity demand and attempt to evenly diffuse the accesses among all L2 sets to reduce conflict misses, but it may not be the most efficient way as analyzed in Section 1. Our work tries to utilize the set-level non-uniformity of capacity demand in leveraging the resource allocation between a set with high capacity demand and another that requires low. The previous proposal that is most closely related to our work is the *Set Balancing Cache* (SBC) since both studies adjust resource allocation by spilling victim blocks from one set to another. But the SBC scheme uses a less accurate measurement for set-level capacity demand. Our COSET scheme overcomes the shortcomings and in turn leads to potentially better performance outcomes.

7. Conclusions

This technical report proposes a new metric that can overcome the shortcomings of existing "access count" and "saturation level" metrics in capturing the set-level capacity demand. Based on the idea of directly measuring the increased hit rate by utilizing the *virtual* capacity provided by shadow sets, a novel last-level L2 cache design, which is called COSET (COoperative SET) L2 cache, is proposed to identify the capacity demands of individual sets, dynamically couples two sets with complementary capacity needs, and enables the set with a higher resource demand to utilize the capacity of its coupled set. Our simulation on selected SPEC CPU 2000 benchmarks shows that the COSET L2 cache achieves a normalized MPKI of 0.383 at best and 0.781 on average over the standard LRU configuration, better than SBC's best and average performance results of 0.585 and 0.867 respectively.

8. References

[1] Andr é Seznec. A Case for Two-Way Skewed-Associative Caches. In *Proceedings of the 20th International Symposium on Computer Architecture* (ISCA), pages 169–178, June 1993.

[2] M. Kharbutli, K. Irwin, Y. Solihin, J. Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture* (HPCA), pp 288 – 299, February 2004.

[3] M. K. Qureshi, D. Thompson, Y. Patt. The V-Way Cache: Demand-Based Associativity via Global Replacement. In *Proceedings of the 32nd International Symposium on Computer Architecture* (ISCA), pp 544 – 555, June 2005.

[4] D. Rol án, B. B. Fraguela, R. Doallo. Adaptive Line Placement with the Set Balancing Cache. In *Proceedings of 42nd International Symposium on Microarchitecture*, pp. 529-540, December 2009.

[5] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture* (ISCA), pages 381–391, June 2007.

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

[7] http://www.spec.org/cpu2000/

[8] T. Austin, E. Larson, D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2): 59 – 67, February 2002.

[9] http://terpconnect.umd.edu/~ajaleel/workload/

[10] L. Soares, D. Tam, and M. Stumm. Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level, Software-Only Pollute Buffer. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 258–269, November 2008.

[11] H. Liu, M Ferdman, J. Huh and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In Proceedings of the 41st International Symposium on Microarchitecture, pages 222–233, November 2008.