9-2009

Exploiting Set-Level Non-Uniformity of Capacity Demand to Enhance CMP Cooperative Caching

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

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

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

Exploiting Set-Level Non-Uniformity of Capacity Demand to Enhance CMP Cooperative Caching*

Dongyuan Zhan, Hong Jiang, Sharad C. Seth
Dept. of CSE, University of Nebraska – Lincoln
Email: {dzhan, jiang, seth}@cse.unl.edu

Abstract

As the Memory Wall remains a bottleneck for Chip Multiprocessors (CMP), the effective management of CMP last level caches becomes of paramount importance in minimizing expensive off-chip memory accesses. For the CMPs with private last level caches, *Cooperative Caching* (CC) has been proposed to enable capacity sharing among private caches by spilling an evicted block from one cache to another. But this eviction-driven CC does not necessarily promote cache performance since it implicitly favors the applications full of block evictions regardless of their real capacity demand. The recent *Dynamic Spill-Receive* (DSR) paradigm improves cooperative caching by prioritizing applications with higher benefit from extra capacity in spilling blocks. However, the DSR paradigm only exploits the coarse-grained application-level difference in capacity demand, making it less effective as the non-uniformity exists at a much finer level.

This paper (i) highlights the observation of cache set-level non-uniformity of capacity demand, and (ii) presents a novel L2 cache design, named SNUG (***S**et-level **N**on-**U**niformity identifier and **G**rouper*), to exploit the fine-grained non-uniformity to further enhance the effectiveness of cooperative caching. By utilizing a per-set shadow tag array and saturating counter, SNUG can identify whether a set should either spill or receive blocks; by using an *index-bit flipping* scheme, SNUG can group peer sets for spilling and receiving in an flexible way, capturing more opportunities for cooperative caching. We evaluate our design through extensive execution-driven simulations on Quald-core CMP systems. Our results show that for 6 classes of workload combinations our SNUG cache can improve the CMP throughput by up to 22.3%, with an average of 13.9%, over the baseline configuration, while the state-of-the-art DSR scheme can only achieve an improvement by up to 14.5% and 8.4% on average.

1. Introduction

As Chip MultiProcessors (CMP) are becoming predominant in processor chip manufacture, computer architects are challenged to architect CMPs for their full performance potentials. One of the key research issues is to reduce the high cost of CMPs' off-chip memory accesses that are generally determined by three factors: access latency, bandwidth and the number of off-chip accesses. While there are techniques such as 3D memory stacking [1], prefetching [2] and optical I/Os [3] that can help reduce (or hide) the long latency and increase the bandwidth of DRAM accesses, on-chip *Last Level Caches* (LLC) play an irreplaceable role in reducing the number of DRAM accesses by keeping as much data as possible on-chip for future references, which necessitates a very effective management of CMP LLCs.

In CMPs, two typical cache organizations are available for architecting the on-chip LLCs. The entire LLC can be either shared among all cores by address interleaving, called shared LLC, or statically partitioned with each cache slice privately used by a core, called private LLC. In this paper, for simplicity and without loss of generality, the CMP LLCs are assumed to be L2 caches, and L2S is short for the shared L2 organization while L2P is short for the private L2 organization. The L2S organization provides a natural way of capacity sharing since the entire L2 capacity is accessible to all cores. But the interleaved addressing can result in too many remote L2 accesses that are penalized by the *Non-Uniform Cache Access* (NUCA) time [4]. The L2P organization, on the other hand, provides good data proximity to the requesting cores, since each core only

* This paper has been submitted to IPDPS 2010.

places its own cache blocks in the local private L2 cache, but with the drawback of limited L2 capacity to each core. Recent studies [5, 6] have advocated the L2P cache organization since it has a lower L2 access latency, lower requirement for on-chip interconnects, better performance isolation and easier support for resource management. Moreover, L2P has been evaluated to outperform L2S when the CMP core count scales [6].

However, due to the limited cache capacity accessible to each core, the miss rate of L2P can be higher than L2S when a core's cache resource requirement exceeds its local private L2 capacity. The rigid constraint that a core can only access its private cache prevents cores from sharing their L2 capacity. To break this barrier, Chang and Sohi [7] proposed the mechanism of *Cooperative Caching* (CC) to allow cross-chip data transfers between different "private" L2 caches and enable capacity sharing by entitling each cache to utilize the capacity of others as victim caches. But in their proposal, cooperative caching is performed regardless of the performance implication: whenever a block is evicted from its own private cache, cooperative caching attempts to retain the block in one of the peer L2 caches, whether or not spilling the block to a peer cache will help the overall performance. For instance, a streaming application can actually always prevail in cooperative caching since it continuously replaces cache blocks; but having its victim blocks cooperatively cached will not benefit its performance at all. Instead, retaining its victim blocks can adversely hurt other L2 caches' performance, since cooperative caching comes at the cost of occupying other caches' capacity.

To overcome the shortcoming of cooperative caching, Qureshi [8] has recently proposed the *Dynamic Spill-Receive* (DSR) paradigm to regulate block spilling and receiving in response to different applications' cache resource demand. In the DSR paradigm, applications are classified into two categories: taker applications and giver applications. Taker applications can have their performance improved with additional cache capacity, while giver applications can contribute part of their cache capacity to others with little performance degradation. When taker and giver applications are co-scheduled on a CMP, taker applications' L2 caches can spill victim blocks to those of giver applications, but not vice versa. While this application-level approach is shown to improve the overall performance when the non-uniformity of cache resource demand explicitly exists at the application-level, it becomes less effective for applications of which such non-uniformity exists at a finer granularity as demonstrated in this paper.

The objective of this paper is to establish that the non-uniformity exists at the cache set-level in capacity demand and then exploit this non-uniformity to further enhance the effectiveness of cooperative caching. The key insight of this work is that differentiating the cache resource demand only at the application level is insufficient for enhancing cooperative caching when performance-sensitive non-uniformity of capacity demand does not surface to the application level but instead exists at the cache set level. This paper then presents a novel L2 cache design, called ***S**et-level **N**on-**U**niformity identifier and **G**rouper* (or SNUG), which identifies and flexibly groups cache sets with complementary capacity demand for cooperative caching. Evaluation results show that the SNUG cache design can significantly boost the effectiveness of cooperative caching.

The main contributions of this work are:

- The key observation on the cache set-level non-uniformity of capacity demand.
- A novel L2 cache design (SNUG) that exploits the set-level non-uniformity to significantly enhance the performance of cooperative caching.
- The key conclusion and performance results through extensive execution-driven simulations.

The rest of the paper is organized as follows. Section 2 introduces the research motivation based on the evidence of cache set-level non-uniformity of capacity demand. Section 3 elaborates the design issues of our proposed SNUG 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 paper concludes with a summary in Section 7.

2. Motivation

Previous studies [8, 9] have revealed that applications have diverse requirement for cache resource. They tried to utilize the application-level difference in resource demand to optimize the usage of CMP L2 caches for multi-programmed workloads. Distinct from previous work, however, we take further steps to evidence the existence of non-uniformity of capacity demand at the cache set level. To accomplish this goal, we need to first develop a group of mathematical models that accurately quantify a cache set's requirement for capacity. With the models, we can characterize the cache set-level non-uniformity of capacity demand. Finally, we argue that this fine-grained non-uniformity can be utilized to further optimize the utilization of CMP L2 caches for multi-programmed workloads, achieving better performance than the state-of-the-art application-level approaches.

Table 1. Glossary of Notation and Terms Used

Symbol	Annotation
N	the total number of sets in an L2 cache
A	#blocks (associativity) owned by a set, $0 \le A < \infty$
S	the index of a set, $0 \le S \le N-1$
I	a fine-grained sampling interval for 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
$block\_required(S,I)$	the number of blocks required by the set S during the sampling interval I
$A_{threshold}$	a value of asssociativity 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 of $[1, A_{threshold}]$
$bucket_j$	The jth bucket, which is the sub-range $[\frac{(j-1)\cdot A_{threshold}}{M}+1, \frac{j\cdot A_{threshold}}{M}]$, where $1 \le j \le M$
$SF(S,I,bucket_j)$	a membership function used to indicate if the number of blocks required by set S is categorized into the jth bucket during interval I
$size\_bucket_j(I)$ $1 \le j \le M$	The size of the jth bucket during interval I

2.1 Quantification of Set-Level Capacity Demand

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

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 specific 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 certain time interval as $block\_required(S,I)$, where S is the index of the set and I is the interested time interval, we can define it as the minimum number of blocks required to resolve all capacity and conflict misses for the set.

We introduce another function, $miss\_count(S,I,A)$ which means 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 [20], 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 $block\_required(S,I)$, then only compulsory misses can happen to set *S*. Thus, we give a quantitative definition of $block\_required(S,I)$ in Formula (1).

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

Because 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 finite number $A_{threshold}$ that is large enough to approximate ∞. Then, we can use Formula (2) to quantify the capacity demand of a set.

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

Alternatively, since $miss\_count(S,I,0)$ is equivalent to the number of accesses to set *S* during interval I, the total hits on set *S* during interval I when the set has A blocks (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 rewritten as follows:

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

Practically, Formula (3) is more convenient 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 [21]. 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 analysis above, we can infer that $block\_required(S,I)$ 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 $\{bucket_1, bucket_2, \dots, bucket_M\}$, 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 $block\_required(S,I)$ 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 have no intersection, the value $block\_required(S,I)$ will be in one and only one bucket's range. **Therefore, we can differentiate two cache sets in terms of their capacity demand if their $block\_required(S,I)$ values belong to different buckets.** Here, we restrict that both $A_{threshold}$ and *M* are integral power 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 capacity demand that is in the range of $bucket_j$ during interval *I*, which is formulated in (4):

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

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 show different set-level capacity demand. Here, we normalize the number of sets that are categorized into the j^{th} bucket during time 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).

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

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

2.2 Methodology of Characterization

We experiment on all 26 SPEC2000 benchmarks [10] using the *sim-cache* tool of *Simplescalar* [11], 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. $block\_required(S,I)$) such that $hit\_count(S,I,A) = hit\_count(S,I,A_{threshold})$, where $A_{threshold}$ is assumed to be double $A_{baseline}$ in this paper.

Since $A_{threshold}$ is assumed to be double of $A_{baseline}$ ($A_{baseline} = 16$) in this paper, we divide the entire range $[1, A_{threshold}]$ into 8 buckets {[1,4], [5,8], ⋯, [29,32]}. Then, for all of the 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 of the L2 sets during the entire sampling period.

2.3 Characterization Conclusions

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 distribution of set-level capacity demand for 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 [12], Figure 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 whole sampling period.

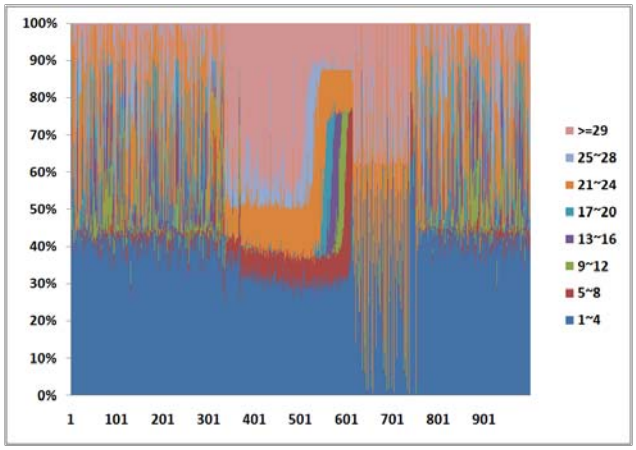

Figure 1. Distribution of Set-level Capacity Demand for *ammp*

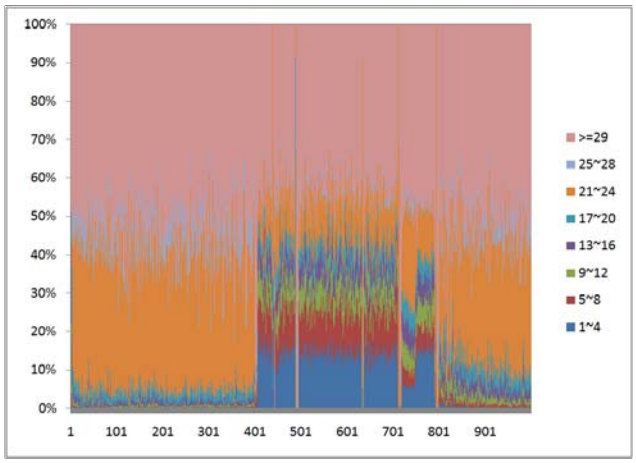

Figure 2. Distribution of Set-level Capacity Demand for *vortex*

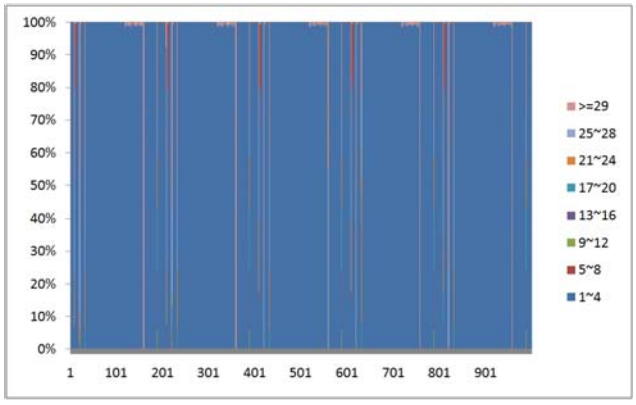

Figure 3. Distribution of Set-level Capacity Demand for *applu*

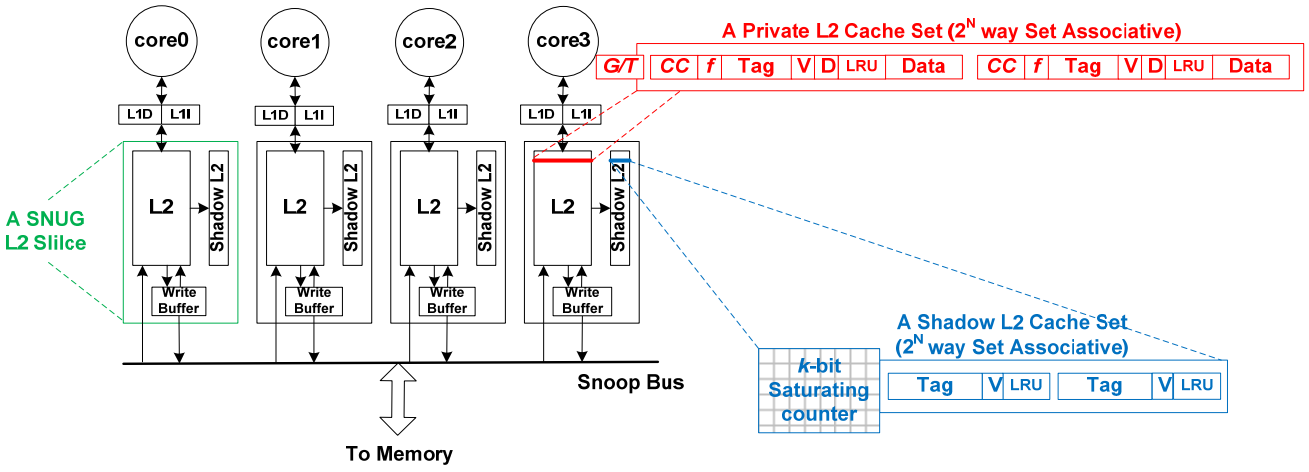

Figure 4. The SNUG L2 Cache Organization for a Quad-Core CMP

3. The SNUG Architecture

SNUG is designed to exploit the fine-grained set-level non-uniformity of capacity demand to enhance the performance of cooperative caching. It aims to accomplish two specific goals: identifying the capacity demand for each L2 set, and grouping peer sets (from different cores) that have complementary set-level capacity demand for flexible cooperative caching.

Figure 4 illustrates a high-level view of a Quad-core CMP with SNUG. Each core has a split private L1 instruction/data cache and a SNUG slice consisting of a private L2 cache capable of cooperative caching, a shadow L2 cache that is used to monitor the set-level capacity demand in the L2 cache, and an L2 write-back buffer that frees the private L2 cache from write back stalls and supports direct data read from the buffer [13]. Within a SNUG L2 slice, the shadow L2 cache has the same number of sets as the L2 cache, and a one-to-one correspondence is maintained between two sets that have the same index in the L2 cache and its shadow cache. The shadow set is intended to monitor the capacity demand of the L2 set for evicted blocks that are accessed again. As Figure 4 shows, a shadow set block has all the usual fields as an L2 set block except for the data field. 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 3.2, and a detailed overhead analysis of this organization appears in Section 3.4 showing that the SNUG overhead falls in the range of 2-6%.

During program execution, the SNUG operation alternates between two stages, as shown in Figure 5. The first stage is used to identify the status of each L2 set as either a giver (G) or taker (T) using the per-set capacity demand monitor. Then, at the beginning the second stage, the dynamic status of L2 sets is used for regrouping them for spilling and receiving. Each two-stage cycle defines a sampling period: Stage I determines the G/T status of each set after a sampling epoch of 5 million cycles, then Stage II follows for 100 million cycles until the start of the next sampling period. A novel *index-bit flipping* scheme determines the constraints observed in grouping the sets. Typically, Stage I is much shorter than Stage II and the total time for the two is shorter than a program-phase during which the program shows relatively stable set-level capacity demand.

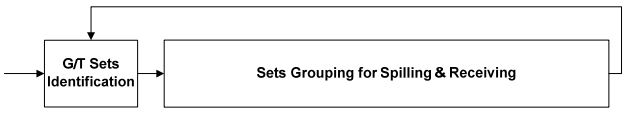

Figure 5. The G/T Sets Identification and Grouping Stages

3.1 Identifying Taker and Giver Sets

In this part, we first explain the structures of the L2 sets and the shadow sets (shown in Figure 4) and how they are updated. Then we describe a HW scheme for measuring the set-level capacity demand and identifying the giver/taker status of each set based on the measurement.

3.1.1 The Structures of Private & Shadow L2 Sets

In an L2 cache, shown in Figure 4, besides the typical fields such as tag, valid, dirty, LRU and data, each cache line is augmented with a *CC* bit that indicates whether this cache line is owned by the local processor core (when *CC*=0) or it is cooperatively cached (when *CC*=1). Another bit *f* is used in the *index-bit flipping* scheme, only takes effect when the *CC* bit is set. If the *f* bit is one, it means that the line is cooperatively cached with the last bit of its original set index flipped. There is also a *G/T* bit corresponding to each L2 set, which is used to indicate whether the set is a giver (when *G/T*=0) or taker (when *G/T*=1) set. The *G/T* bits for all L2 sets form a G/T vector, each entry of which is addressable independent of addressing the L2 sets.

Each entry in a shadow set has a tag field, a valid bit and LRU bits. The shadow set retains the "shadows", namely the tag fields, of *locally* evicted lines from the corresponding private L2 set: when an L2 set needs to replace a line by the LRU policy, and the victim line is owned by the local processor core, the shadow set will retain the tag field of the victim line in one of its entries and set it valid. Additionally, the shadow L2 set maintains its own independent LRU ranking for all of its valid entries and uses it for replacement. We require that the shadow set entries be strictly exclusive with the local lines in the corresponding L2 set in terms of their tag fields. Therefore, if a formerly evicted block with its tag present in the shadow set is revisited by the local core, two actions 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.

3.1.2 Monitoring Set-Level Capacity Demand

If an L2 set and its corresponding shadow set have the same associativity, the private 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 set-level takers and givers are identified and grouped for cooperative caching.

Since an L2 set and its shadow set form two *buckets*, according to Formula (3), we can use the ratio $\sigma = \frac{\#hits\ (\ on\ the\ shadow\ set\ )}{\#hits\ (on\ the\ L2\ set)\ \ +\ \ \#hits\ (\ on\ the\ shadow\ sadow\ set\ )}$ to measure the potential performance benefit in terms of hit rate increase if the capacity of the L2 set is doubled in terms of 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

"$\#hits(on\ the\ shadow\ set) - 1/p *$
$[\#hits(on\ the\ L2\ set) + \#hits(on\ the\ shadow\ sets)] > 0$".

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 7: 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 lead to an increase in hit rate by at least $1/p$.

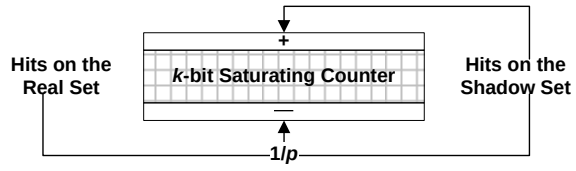

Figure 6. The Operation on a Saturating Counter

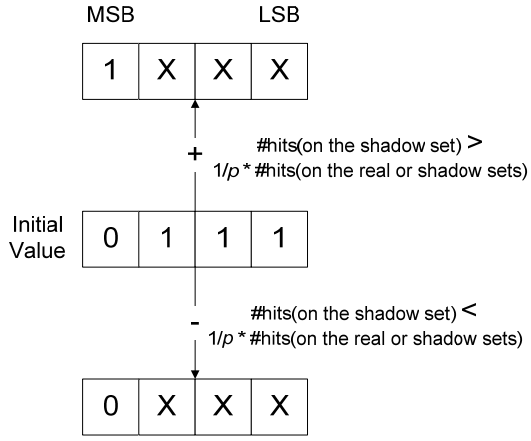

Figure 7. An illustration of the operations on a 4-bit Saturating Counter

3.1.3 G/T Sets Identification

As described above, we can differentiate taker and giver sets by just checking the MSB of the saturating counter of each set. A one value of the MSB indicates that extending the capacity of the set is beneficial, hence the set should be regarded as a taker and entitled to spill blocks in cooperative caching; otherwise, the set is defined as a giver and receives spilled blocks from its peer taker set. Thus, the MSB of the saturating counter can be directly used to update the corresponding entry of the G/T vector.

3.2 Grouping Sets for Spilling & Receiving

After the *G/T Sets Identification* stage, the SNUG caches enter the *Sets Grouping* Stage to group different cores' sets with complementary capacity demand to perform block spilling and receiving. The simplest grouping strategy is to group different cores' sets with the same index, as is done in ordinary CC or DSR. But this naïve approach only allows the sets with the same index to form a receiving & spilling group. Ideally, we would like to group taker and giver sets based just on their capacity demand and supply, independent of their index values. However, this would lead to significant hardware complexity, as the information of how sets are globally grouped would need to be stored and retrieved for each private L2 set. Hence, we propose an *index-bit flipping Scheme* that flexibly groups sets with complementary capacity demand for spilling and receiving at the low hardware complexity of one *f* bit per cache line.

The *index-bit flipping* scheme works as follows. In an L2 cache, when a taker set needs to spill a local cache line, the L2 cache will put a CC spilling request together with the address of the spilled line on the interconnection bus. By snooping on the bus, other peer caches can detect the CC request as well as the address of the spilled block. Each peer cache will look up its own *G/T* vector to find the *G/T* information of the two adjacent entries that have the same index as the CC-spilling block but with the last index bit being don't-care. There can be three cases as shown in Figure 8. In Case 1, if the set with exactly the same index is a giver set in the peer L2 cache, then the peer L2 cache will attempt to retain the spilled block in its set with exactly the same index. In Case 2, if the set with exactly the same index in the peer L2 cache is a taker set while the other set with the last index bit different is a giver set, then this giver set will attempt to retain the spilled block. In Case 3, if the corresponding two adjacent sets are both taker sets, then this peer L2 cache will not respond to the CC request. Any peer cache that first responds to the CC request on the interconnection bus will get the spilled block. Based on whether the block is cooperatively cached in the set with exactly the same index or with the last index bit flipped, the *f* bit of the cooperatively cached block will be set to zero (if the last index bit is not flipped) or set to one (if the last index bit is flipped).

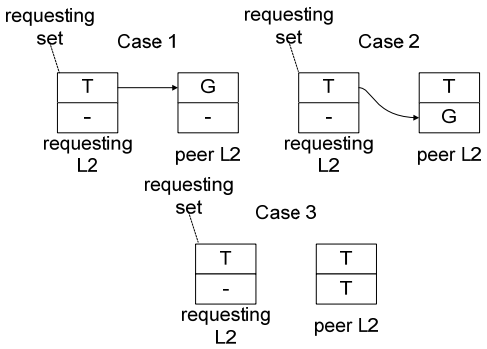

Figure 8. The *index-bit flipping* scheme

Now suppose a block is missed in its local L2 cache, the cache will signal a retrieving request for the block with

its address on the snoop bus. After a peer cache detects the request, it will first lookup its *G/T* vector for the information of the two adjacent *G/T* bits that have the same index as the block address but with the ending bit being don't-care. If the G/T bit with exactly the same index, or otherwise with only the last index bit different, indicates a giver set, then the L2 cache will try to find the block in the corresponding giver set; if both of the adjacent peer sets are indicated as taker sets, then it means that the block being retrieved can't be located in this L2 cache. This leads to at most one unambiguous search for the block in a peer L2 cache. Because the cooperatively cached block can only be located in a giver set of at most one peer L2 cache, then the peer cache that has the cooperatively cached block will directly forward the block to the requesting L2 cache. At the same time, the peer cache will invalidate its cooperatively cached copy of the block to free space for other blocks. If no peer caches respond to the retrieving request, the requested block is not on chip and will need to be fetched from the main memory.

3.3 Maintaining Cache Coherence

In the SNUG cache design, we use two restrictions to maintain coherence between different L2 caches. First, only when a locally evicted block is clean can it be cooperatively cached in a peer L2 cache. If the block is dirty, it will be directly put in the local L2 write buffer. Second, if a peer cache forwards a cooperatively cached block to the original owner cache of the block, the copy of the block in the peer cache needs to be invalidated.

Table 2. The length of each field in the SNUG cache design by using the cache configuration in Table 4

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

3.4 Space & Time Overhead Analysis

Since the SNUG caches require the per-set capacity demand monitor, the shadow sets and saturating counters will account for the major hardware overhead in our design. Then, the storage overhead of the SNUG cache can be calculated by using Formula (6)

$$storage\ overead = \frac{storage\ of\ a\ shadow\ set}{storage\ of\ a\ shadow\ set + storage\ of\ an\ L2\ set} \quad (6)$$

Table 2 shows the length of each storage field in the SNUG design if we use the cache configurations in Table 4. Under such a cache configuration, the storage overhead of the SNUG cache design is only 3.9% by Formula (6), which is reasonably low when we consider the abundant silicon resources available as a result of technology scaling.

However, many processors, such as SUN's UltraSPARC-III [22], use 64-bit wide memory addresses. A longer memory address leads to a longer tag field in the shadow set that introduces more hardware overhead, although typically some leading bits of the memory address are unused (e.g., the leading 20 and 23 bits of the virtual address and physical addresses are unused in UltraSPARC-III respectively). We can offset the hardware overhead by adopting larger cache block size while keeping the cache capacity fixed. Table 3 shows the hardware overhead of different memory address and cache line size combinations for a 1MB private L2 cache.

In terms of the time overhead, in our SNUG cache implementation, we experimentally observed that a combination of 5 million cycles for the *G/T Sets Identifying* Stage and another 100 million cycles for the *Sets Grouping* Stage produces a good performance outcome, which is adopted in Section 5. During the 5 million cycles for *G/T Sets Identification*, the cache can still accept retrieving request but no spilling request from others. At the end of this stage, each L2 cache maintained a new G/T vector, and continues to use the set-level G/T information in grouping sets for spilling or retrieving blocks.

Table 3. The hardware overhead of different memory address and block size combinations

	32-bit address	64-bit address
64B/cache line	3.9%	5.8% (assuming only 44 address bits are used)
128B/cache line	2.1%	3.1%

4. Evaluation Methodology

To evaluate SNUG against other last-level cache management schemes available in the literature, we simulated a combination of workloads, consisting of 12 programs from the SPEC2000 benchmark suite on quad-core systems. In this section, we describe the configuration of our simulation system and the workload combinations.

Table 4. The configuration of the PolyScalar simulator

Out-of-Order Core Configuration		Mem Hierarchy Configuration	
Processors	4	L1 Lat	1 Cycle
Issue/Commit	8/8	L1I/D	4 way, 32KB, 64B lines
I-Fetch Queue	8	L1D	write back
LSQ Size	64	L2 Lat	10 cycles locally
RUU Size	128	Each L2 Slice	16 way, 1MB, 64B lines, write back
ALU/FPU/Mult/Div	4/4/1/1		
Branch Predictor	2-Level, 1024 Entry, History Length 10	Snoop Bus	16B-wide split transactional bus, 4:1 speed ratio, 1 cycle for arbitration
BTB Size	512 Sets 4 way	DRAM Lat	300 Cycles
Branch Penalty	3 Cycles	L2 Write Buffer	FIFO, Mergeable, 16 entries*64B/entry, support direct read
RAS Entries	8		
Address bits	32		

4.1 Simulation Configuration

In our experiment, we use the cycle-accurate PolyScalar [14], a multi-core simulator with detailed memory hierarchy model and SimpleScalar out-of-order cores [11]. We implement and evaluate five L2 cache organizations, L2P, L2S, CC (Best), DSR, and SNUG. According to [7], one of the spill-probabilities 0%, 25%, 50%, 75% and 100% that produces the best performance is selected as CC (Best) for a given workload. Table 4 lists the configuration shared by the five L2 schemes above. The difference between the L2 schemes is the remote L2 access latency: for L2P, CC and DSR, we assume the remote L2 access latency is 30 cycles, while the remote latency for SNUG is assumed to be 40 cycles to include the additional delay of looking up the G/T vector of each L2 cache.

For the purpose of thorough comparison, three standard metrics (shown in Table 5) are used to quantify the performance [8]: throughput that is the sum of IPCs (instructions per cycle) evaluates the utilization of a system; average weighted speedup indicates reduction in execution time; fair speedup balances both performance and fairness.

Table 5. Performance Metrics

Metric	Definition (N is the core count)
Throughput	$\text{Throughput(Scheme)} = \sum_{i=1}^{N} IPC_i(Sceme)$
Average Weighted Speedup [15]	$\text{AWS(Scheme)} = \frac{1}{N} \times \sum_{i=1}^{N} \frac{IPC_i(Sceme)}{IPC_i(Baseline)}$
Fair Speedup [16]	$\text{FS(Scheme)} = N / \sum_{i=1}^{N} \frac{IPC_i(Baseline)}{IPC_i(Sceme)}$

4.2 Workload Combinations

Table 6 classifies the 12 SPEC CPU2000 benchmarks used in our studies. Our evaluation takes into account 6 different classes of workload combinations described in Table 7. Specifically, workload combination class C1 and C2 are both stress tests, which means that the four co-scheduled applications from C1 or C2 are all identical, but with the assumption that there can be only capacity sharing among the co-scheduled applications, excluding any data sharing. The purpose of the stress tests is to see how different L2 cache designs can respond to applications' set-level capacity demand, since the identical co-scheduled applications have the same capacity demand at both

application and set levels. Within a class in C3 - C6, all of the co-scheduled applications are different, and at least two applications showing set-level non-uniformity of capacity demand are chosen in each workload combination,

Table 6. Workload Classification & Selection

Application Type	Workload Class	Application-Level Capacity Demand	Applications
showing set-level non-uniformity of capacity demand	A	> 1MB	ammp, parser, vortex
	B	< 1MB	apsi, gcc
showing set-level uniformity of capacity demand	C	> 1MB	vpr, art, mcf, bzip2
	D	< 1MB	gzip, swim, mesa

Table 7. Workload Combination Classes & Characteristics

C1	4 identical applications from class A without data sharing (stress test)
C2	4 identical applications from class C without data sharing (stress test)
C3	(2 different applications from class A) + (2 different applications from class C)
C4	(2 different applications from class A) + (1 application from class B)+ (1 application from class C)
C5	(2 different applications from class A) + (2 different applications from class D)
C6	(2 different applications from class A) + (1 application from class B) + (1 application from class D)

Table 8 shows that 21 workload combinations that are categorized into the 6 different classes respectively.

5. Result Analysis

For each instance of simulation, we forward the execution by 6 billion cycles to bypass the initialization section of the programs, and then execute each workload combination with detailed out-of-order core model and different cache schemes for additional 3 billion cycles. In the results analysis, the numbers reported for a class of workload combinations are the geometric means calculated for all of the workload combinations in a given class.

Table 8. The Configurations of Different Workload Combinations

C1	4 ammp	C3	(ammp+parser)+(bzip2+mcf)	C5	(ammp+parser)+(swim+mesa)
	4 parser		(parser+vortex)+(mcf+art)		(parser+vortex)+(mesa+gzip)
	4 vertex		(vortex+ammp)+(art+vpr)		(vortex+ammp)+(swim+gzip)
C2	4 vpr	C4	(ammp+parser)+(apsi)+(bzip2)	C6	(vortex+ammp)+(apsi)+(gzip)
	4 bzip2		(parser+vortex)+(gcc)+(mcf)		(parser+vortex)+(gcc)+(mesa)
	4 mcf		(vortex+ammp)+(apsi)+(art)		(ammp+parser)+(apsi)+(swim)
	4 art		(ammp+parser)+(gcc)+(vpr)		(vortex+ammp)+(gcc)+(mesa)

Figure 9 shows the throughput results of the L2S, CC(Best), DSR and SNUG schemes normalized to L2P (1.0). In class C1 that is the stress test, because all of the applications have an application-level capacity demand of over 1MB and also exhibit the set-level non-uniformity of capacity demand, the SNUG cache organization can take advantage of the complementary capacity demands of interleaved taker and giver sets by the *index-bit flipping* scheme and then capture more opportunities for cooperative caching. Therefore, SNUG achieves a throughput improvement over the baseline L2P cache by 22.3% in class C1, bettering the performance gain of CC(Best) by 3.5% and that of DSR by 6.9%. In C2, DSR achieves a throughput improvement over the baseline by 2.3%, and performs slightly better than CC(best) (- 0.5 % performance degradation) and SNUG (- 0.2% performance degradation), because DSR can assign some of the identical applications as taker applications while assigning others as giver applications to achieve biased performance improvement. In C3, C4, C5 and C6, SNUG outperforms all the other cache schemes. Overall, on average, SNUG can improve

the Quad-core CMP throughput by 13.9% for all of the 6 classes of workload combinations, in contrast to 8.4% of DSR (the second best).

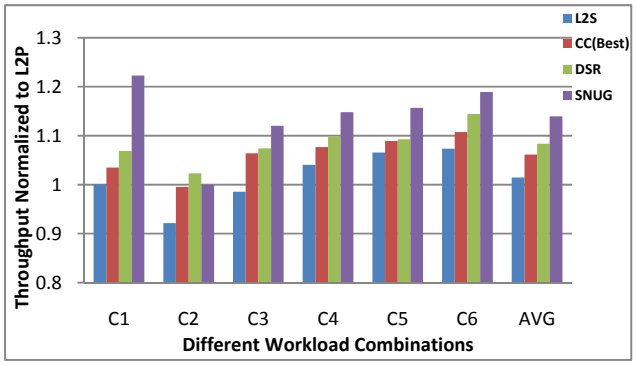

Figure 9. Performance on Throughput Metric

Because the throughput metric is not fair to the application with a lower absolute IPC, we also use the metric of Average Weighted Speedup to consider the change of relative IPC (the absolute IPC of a scheme over that of the baseline) of the applications. From Figure 10, it can be concluded that SNUG can also improve the *Average Weighted Speedup* by 13.0%, while DSR, CC(Best) and L2S improves it by 9.9%, 7.0%, and 2.5%, respectively .

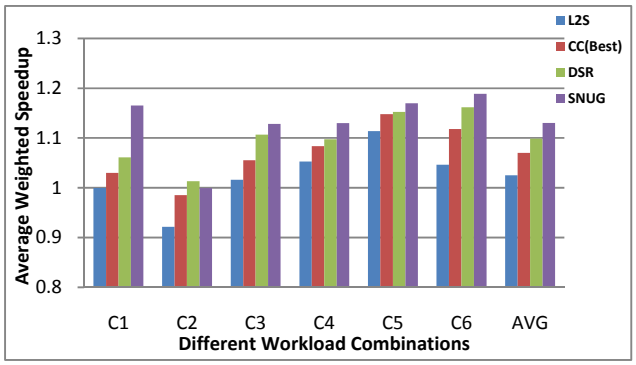

Figure 10. Performance on Average Weighted Speedup

Figure 11 demonstrates the performance on the Fair Speedup metric (the harmonic mean of programs' relative IPCs) that balances both performance and fairness for different classes of workload combinations as well as different L2 cache schemes. On average, the SNUG scheme improves the performance by 10.4%, better than L2S(-1.5% degradation), CC(Best) (4.2%) and DSR (6.3%).

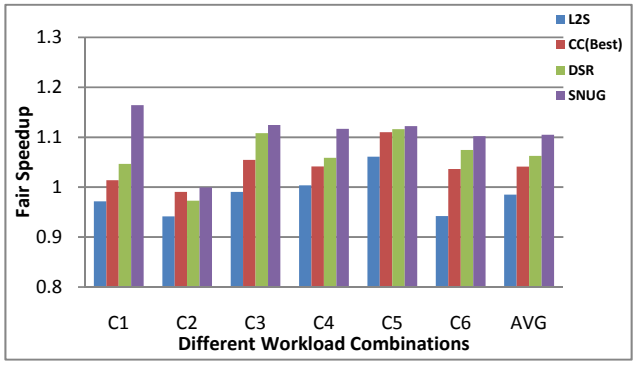

Figure 11. Performance on Fair Speedup

6. Related Work

Previous Research on Set-Level Non-Uniformity: Set-level non-uniformity of cache resource demand is a research focus in the unicore processor cache management. Hash functions based on prime modulus were proposed to equalize the number of misses over all cache sets [17], and the V-Way cache architecture [18] was proposed to vary the associativity of the last level cache on a per-set basis in response to the demands of the program. SNUG differs significantly from these earlier schemes, which aimed at alleviating set-level non-uniformity of cache demand, in that it exploits the set-level non-uniformity in cooperative caching to improve performance.

Improvement on Cooperative Caching: Since Chang and Sohi [7] introduced the concept of CMP cooperative caching, there have been several proposals to improve the original CC scheme from various angles. *Adaptive Selective Replication* (ASR) [19] relaxes the restriction on replicated blocks in cooperative caching. By dynamically replicating shared read-only blocks in multiple private cache partitions, ASR can hide the latency of cross-chip transfer of these commonly used blocks for multithreaded workloads. *Distributed Cooperative Caching* [5] aims to resolve the performance bottleneck imposed by the centralized coherence engine in cooperative caching. It simply utilizes the distributed directory caches rather than a centralized one as the cooperative caching engine to improve the scalability and power efficiency of cooperative caching. In another proposal [6], Eisley et al. enable

on-chip network to propagate the information of invalid blocks so that victim blocks can be spilled to other caches with abundant invalid blocks. As the contributions focus on such factors as scalability and power efficiency of cooperative caching, their results can be considered orthogonal to our scheme on improving the effectiveness of cooperative caching.

7. Conclusion

Although cooperative caching allows CMP private L2 caches to share their capacity, its effectiveness is limited by its eviction-driven spilling and receiving. The *Dynamic Spill and Receive* (DSR) technique improves cooperative caching by taking into account the differences in capacity demand that appear at the application-level. DSR is less effective when such differences manifest themselves at the cache set level but not at the application level. Our investigations reveal that this situation is common, motivating our proposal of *Set-level Non-Uniformity identifier and Grouper* (SNUG) scheme that can exploit the fine-grained non-uniformity via cooperative caching to improve the system performance. Experiments show that for six classes of workload combinations our SNUG cache can improve the Quad-Core CMP throughput by 22.3% at best and by 13.9% on average over the baseline configuration, outperforming the state-of-the-art DSR scheme that can only achieve an improvement by up to 14.5 % and 8.4 % on average. Our future work would attempt to extend SNUG to multi-threaded workloads, and to both intra- and inter-cache accesses.

References

[1] G. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proceedings of the 35th International Symposium on Computer Architecture* (ISCA), pp 453 – 464, June 2008.

[2] C. J. Lee, O. Multu, V. Narasiman. Prefetch-Aware DRAM Controllers. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pp 200 – 209, November 2008.

[3] L. Young. Optical I/O Technology for Tera-Scale Computing. In *Proceedings of the International Solid-State Circuits Conference* (ISSCC), pp 468 – 470, February 2009.

[4] C. Kim, D. Burger, S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pp 211 – 222, October 2002.

[5] E. Herrero, J. Gonzalez, R. Canal. Distributed Cooperative Caching. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (PACT), pp 134 – 143, October 2008.

[6] N. Eisley, L. Peh, L. Shang. Leveraging On-Chip Networks for Data Cache Migration in Chip Multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (PACT), pp 197 – 207, October 2008.

[7] J. Chang, G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture* (ISCA), pp 264 – 276, June 2006.

[8] M. K. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture* (HPCA), pp 45 – 54, February 2009.

[9] A. Jaleel, W. Hasenplaugh, et al. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (PACT), pp 208 – 219, October 2008.

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

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

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

[13] K. Skadron, D. W. Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture* (HPCA), pp 144 – 155, February 1997.

[14] http://www.cs.ucsb.edu/~franklin/PolyScalar/Home.htm

[15] D. M. Tullsen and J. A. Brown. Handling long-latency loads in asimultaneous multithreading processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pp 318 – 327, December 2001.

[16] K. Luo, J. Gummaraju, M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (ISPASS), pp 164 – 171, November 2001.

[17] 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.

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

[19] B. M. Beckmann, M. R. Marty, D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pp 443 – 454, December 2006.

[20] 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.

[21] M. K. Qureshi, Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pp 423 – 432, December 2006.

[22] T. Horel and G. Lauterbach. UltraSPARC III: DesigningThird Generation 64-Bit Performance. IEEE Micro, 19(3): 73 - 85, May/June 1999.