Adaptive Spill-Receive for Robust High-Performance Caching in CMPs

Moinuddin K. Qureshi
IBM Research
T. J. Watson Research Center, Yorktown Heights NY
mkquresh@us.ibm.com

Abstract

In a Chip Multi-Processor (CMP) with private caches, the last level cache is statically partitioned between all the cores. This prevents such CMPs from sharing cache capacity in response to the requirement of individual cores. Capacity sharing can be provided in private caches by spilling a line evicted from one cache to another cache. However, naively allowing all caches to spill evicted lines to other caches have limited performance benefit as such spilling does not take into account which cores benefit from extra capacity and which cores can provide extra capacity.

This paper proposes Dynamic Spill-Receive (DSR) *for efficient capacity sharing. In a DSR architecture, each cache uses Set Dueling to learn whether it should act as a "spiller cache" or "receiver cache" for best overall performance. We evaluate DSR for a Quad-core system with 1MB private caches using 495 multi-programmed workloads. DSR improves average throughput by 18% (weighted-speedup by 13% and harmonic-mean fairness metric by 36%) compared to no spilling. DSR requires a total storage overhead of less than two bytes per core, does not require any changes to the existing cache structure, and is scalable to a large number of cores (16 in our evaluation). Furthermore, we propose a simple extension of DSR that provides Quality of Service (QoS) by guaranteeing that the worst-case performance of each application remains similar to that with no spilling, while still providing an average throughput improvement of 17.5%.*

1. Introduction

Chip Multi-Processors (CMP) have become a standard design point for industry. One of the key design decisions in architecting a CMP is to organize the last level cache as either a private cache or a shared cache. Figure 1 shows a four-core CMP with (a) shared cache and (b) private cache. A private cache is an attractive design option as it offers the following advantages over a shared cache. First, reduced cache access latency compared to a shared cache as the cache is located physically closer to the core, reducing wire delays. Second, private caches inherently provide performance isolation so that a badly behaving application cannot hurt the performance of other concurrently executing applications. Third, private caches allow for a tiled architecture as the tag-store and data-store of L2 cache are contained in the same design unit as the core which allows for a scalable design and facilitates power optimizations. Finally, private caches simplify the on-chip interconnect as only the misses in the last level cache access the shared interconnect fabric. Shared caches, on the other hand, can provide capacity sharing but requires a high bandwidth on-chip interconnect as all access to the last level cache have to use the interconnect. For example, Niagara-1 [7] uses a crossbar to interconnect all the 8-cores to the shared L2 cache. A recent study [8] has argued that the area and latency overhead of the shared on-chip interconnect can often offset most of the capacity sharing advantage of the shared cache.

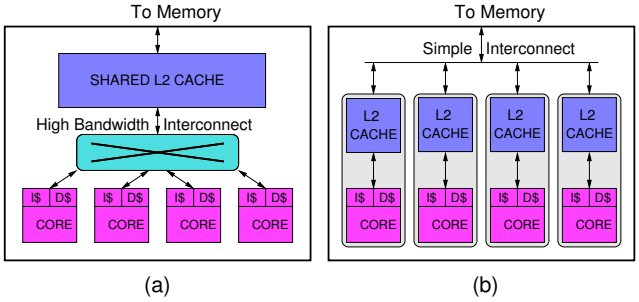

Figure 1. A four-core CMP design with a (a) shared L2 cache (b) private L2 cache.

The main disadvantage of private caches is that the cache capacity is statically partitioned equally among all the cores. This prevents such a cache organization from sharing cache capacity in response to the requirement of individual cores. Therefore, private caches typically incur more misses than a shared cache. Recent studies on efficient private caches, namely Cooperative Caching (CC) [3], use second chance forwarding [5] for capacity sharing. When a cache line is evicted from one of the private caches, CC can store it

978-1-4244-2932-5/08/$25.00 ©2008 IEEE

in another private cache. This transfer of evicted line from one cache to another is called *spilling*. We call the cache that spilled the line as *"spiller cache"* and the cache that received the spilled line *"receiver cache"*. The basic problem with CC is that it performs spilling without knowing if spilling helps or hurts cache performance. All caches act as spiller cache, even if some of the applications do not benefit from extra capacity. Similarly, all caches can receive evicted lines spilled from other cache, even if some of the caches do not have spare capacity. Therefore, the capacity sharing of CC has limited performance improvement. The objective of this paper is to design a practical, low-overhead spilling mechanism for providing robust high-performance capacity sharing for private caches by taking into account the cache requirement of each core. Given that every design changes and added structure requires design effort, verification effort, and testing effort, we would ideally like our mechanism to have no extra structures or design changes, while still being scalable to a large number of cores.

The difference in this work compared to CC is the key insight that a given private cache should either be allowed to get more capacity or be allowed to give away excess capacity but not both at the same time. If the cache can spill as well as receive then the cache tries to get more cache at a remote location by spilling while at the same time provide its own local capacity to store lines of some other caches. Therefore, our design restricts each cache to be either a spiller or a receiver but not both. We propose the *Spill-Receive Architecture* in which each cache is appended with one bit: S/R. When the S/R bit associated with a cache is 1, the cache acts as a spiller cache and when the S/R bit is 0, the cache acts as a receiver cache. With the right configuration of S/R bits, it is straight-forward to design a robust high-performance capacity sharing mechanism.

Whether a cache should be a spiller or a receiver depends not only the given application but also on the other applications concurrently executing in the CMP. For the same application, the best overall performance is obtained when the cache acts as a spiller for some workload mixes and as receivers for others. Therefore, the decision about which caches should be spillers and which should be receivers must be determined at runtime. We propose *Dynamic Spill-Receive (DSR)* cache architecture, in which each cache learns using Set Dueling [10] whether it should act as spiller or receiver for best overall performance. DSR dedicates a few sets (32 in our studies) of the cache to "always-spill" and another few to "always-receive" and uses the policy that gives fewest misses for the remaining sets of the cache. Each cache learns the spill-receive decision independently using a separate Set Dueling mechanism. We show that Set-Dueling based DSR performs similar to apriori knowing the best spill-receive decisions for a given workload using oracle information.

We evaluate DSR on a Quad-core system with 1MB private L2 cache with each core. We use 12 SPEC benchmarks, run all the possible 495 four-threaded combinations, and measure system performance on all the three metrics: throughput, weighted-speedup, and hmean-based fairness. We show that DSR improves average throughput by 18%, weighted speedup by 13% and hmean-based fairness metric by 36%. DSR requires a total storage overhead of less than two bytes per core, does not require changes to existing cache structure, and is scalable to a large number of cores. DSR provides more than double the performance improvement than Cooperative Caching (CC), while obviating the design changes of having extra spill bits required by CC.

For all the 1980 applications examined (495x4), DSR has an IPC degradation of more than 5% compared to no spilling for 1% of the applications. In Section 6, we show that a simple extension of DSR can provide Quality of Service (QoS) by guaranteeing that the worst-case performance of each concurrently executing application remains similar to that with no spilling, while still providing an average throughput improvement of 17.5%.

2. Motivation and Background

In this work, we assume each core in the CMP executes one application. In a CMP, different cores can execute diverse applications concurrently, each application having different memory behavior and varying cache requirement. A private cache statically divides the total cache into equal-size cache units and associates one cache unit with each core. Thus, all cores have uniform cache capacity, albeit at a faster access latency and reduce interconnect requirement than a shared cache. However, applications vary in terms of benefit obtained from cache.

Figure 2 shows the misses per 1000 instructions (MPKI) and Cycles Per Instructions (CPI) for 12 SPEC benchmarks used in our studies. The horizontal axis shows the number of ways allocated from a 32-way 2MB L2 cache. The private L2 cache used in our baseline is 1MB 16-way which is indicated by the Grey dotted line. The benchmarks shown in the top row of Figure 2 have excess cache capacity in the baseline 1MB cache. Their CPI and MPKI do not increase significantly when the cache size is halved. Eon and crafty have a small working set, fma3d and equake are sensitive to cache capacity only up to $\frac{1}{4}th$ MB, and applu and lucas are streaming workloads. These applications can provide their extra cache capacity to other applications that can benefit from more cache capacity. We call these applications "Giver" applications. The benchmarks in the second row of Figure 2 continue to benefit from cache space. Their CPI and MPKI decrease considerably when the cache size is increased from 1MB to 2MB. These applications can benefit by using extra cache capacity. We term these applications as "Taker" applications.

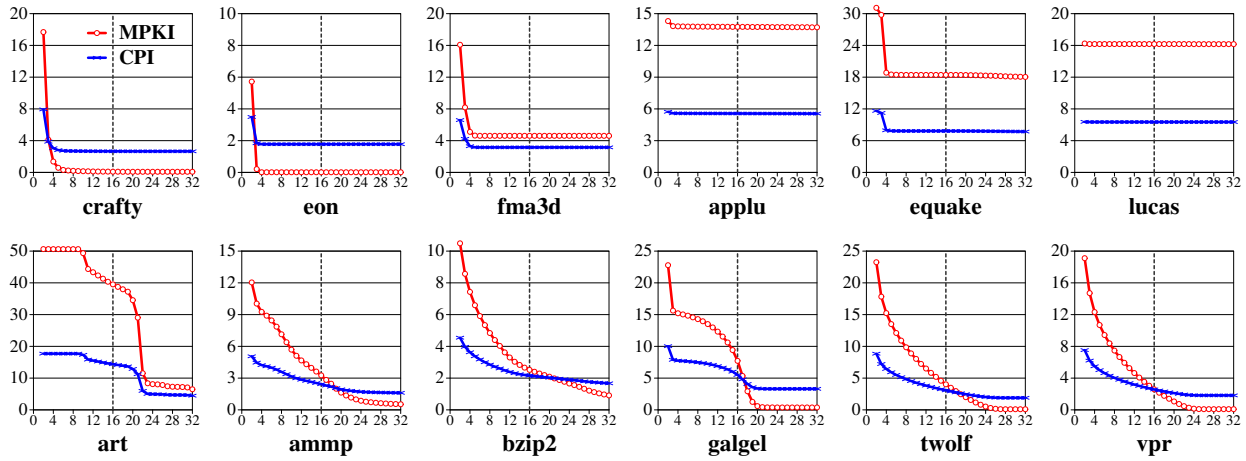

Figure 2. MPKI and CPI for SPEC benchmarks as the cache size is varied. The horizontal axis shows the number of ways allocated from a 32-way 2MB cache (the remaining ways are turned off). The baseline cache is 1MB 16-way: benchmarks in the top row can provide cache capacity and benchmarks in the bottom row benefit significantly from cache capacity more than 1MB.

If all applications in the system are Giver applications then private caches work well. However, when some applications in the system are Takers and other are Givers then cache performance and overall system performance can be improved if the excess cache capacity of Giver applications are provided to the Taker applications. Thus, capacity sharing is important to improve the performance of private caches. Recent studies on efficient private caches, namely Cooperative Caching (CC) [3], use second chance forwarding for capacity sharing. When a line is evicted from one of the private cache, CC can retain that line in another private cache on the same chip. This transfer of evicted line from one cache to another is called *spilling*. We call the cache that spilled the line as *"spiller cache"* and the cache that received the spilled line the *"receiver cache"*. A line can be spilled until it has exceeded a predetermined number of spills. The basic problem with CC is that it performs spilling without knowing if spilling helps or hurts cache performance. For example, both Taker and Giver applications are allowed to spill their evicted lines to neighboring caches. This can be particularly harmful when cache lines of streaming applications (such as applu and lucas) are spilled into caches of Taker applications (say vpr and bzip2). The cache of streaming applications will not benefit from spilling because of poor reuse but receiving the spilled line can hurt performance of applications such as vpr and bzip2. Therefore, the capacity sharing as done in CC has limited performance improvement, and in some cases spilling can in-fact hurt performance.

A key difference in this work compared to CC is the insight that a given private cache should either be allowed to get more capacity or be allowed to give away excess capacity but not both at the same time. If the cache can spill as well as receive then the cache tries to get more cache at a remote location by spilling while at the same time provide its own local capacity to store lines of some other caches. Therefore, our design restricts each cache to be either a spiller or a receiver but not both. Given the information about whether a cache is spiller or receiver, it is straight forward to design an efficient cache sharing scheme for private caches: caches designated as spiller-caches are allowed to spill their evicted lines to receiver-cache. Evicted lines from receiver caches are not spilled to any of the on-chip caches. However, the decision about which cache should be spiller and which should be receiver must be done judiciously otherwise overall system performance can degrade compared to the base case of no spilling. The next section describes our proposed spill-receive architecture and a runtime mechanism to lean the best spill-receive decision for each cache.

3. Design of Dynamic Spill-Receive

3.1. Spill-Receive Architecture

We propose a *Spill-Receive Architecture* in which each cache is appended with one bit: S/R. When the S/R bit associated with a cache is 1, the cache acts as a spiller cache and when the S/R bit is 0, the cache acts as a receiver cache. Thus, the S/R bit classifies each cache in the system as either a spiller or a receiver but not both. This is important because if the cache is trying to spill the line and get higher latency cache space then it should first retain its lower-latency space. Similarly, if the cache is willing to give cache space to other applications, then it should not try to get higher-latency cache space some where else in the system. Figure 3 shows an example of the Spill-Receive architecture for a Quad-core system with private L2 caches. The S/R

bit of caches A and C are set to 1 indicating that these two caches act as spiller caches. Conversely, caches B and D act as receiver caches.

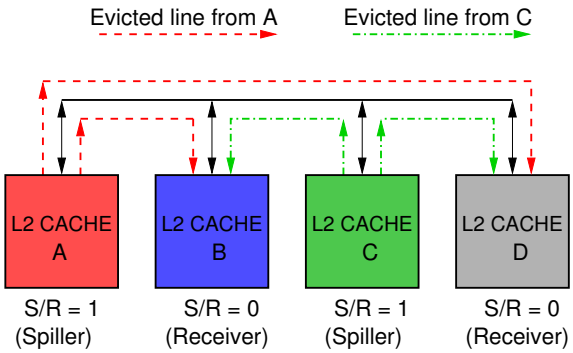

Figure 3. A Spill-Receive architecture for a four-core CMP. Cache A and C spill evicted lines randomly to either cache B or D.

When a core accesses a cache line, it first checks for the cache line in the local cache associated with the core. If there is a miss in the local cache, all the other caches in the system are snooped (this is required in the baseline as well for coherence). If there is a hit in any of the remote caches, the cache line is brought to the local cache and the line evicted from the local cache is transferred to the remote cache. If there is a miss in all the remote caches as well, the line is fetched from memory and installed in the local cache. This line can cause an eviction of another cache line from the local cache. If the local cache is a spiller cache (A and C), the evicted line is spilled to one of the receiver caches in the system. If there are more than one receiver caches in the system then a receiver cache is chosen randomly.[1] This ensures that some of the lines evicted from A are spilled to B and some to D, ensuring good load balancing between receiver caches. Similarly, evicted lines from C are spilled to either B or D. When a line is evicted from any of the receiver caches (B and D) it is simply discarded without trying to spill to any of the caches. Thus, our architecture automatically discards unused spilled lines without the need for any spill bit in the tag-store entry of each cache, as required by CC [3]. When all caches in the system are spiller caches then spilled lines cannot be received in any of the caches, which implicitly disables spilling. Similarly, when all caches in the system are receiver caches then none will spill an evicted line, which explicitly disables spilling.

Given the right configuration of S/R bits for each of the cache, the Spill-Receive architecture can implement an efficient cache sharing mechanism for private caches. Such a scheme can have the latency and bandwidth advantages of private caches and capacity sharing advantages of shared caches. And, it can do so while incurring a negligible hardware overhead (one bit per cache). A vital piece of information in the Spill-Receive architecture is the S/R bit associated with each cache. As the spill-receive decision for an application varies with input set, machine configuration, and behavior of other competing applications, obtaining this information using profiling may be impractical or even impossible. Therefore, we obtain the spill-receive decision at runtime using the recently proposed Set-Dueling [10] technique. We briefly describe Set-Dueling next.

[1] Further optimization of the scheme can be done by tuning what fraction of the spilled lines from a particular spiller cache gets to a particular receiver cache. Or, restricting spills only to nearest neighbor to reduce interconnect latency. We do not consider such optimizations in this paper.

3.2. Set Dueling

Set Dueling is a general mechanism that can choose between competing policies while incurring negligible overhead. Set Dueling leverages the fact that the last-level caches typically have large number (more than thousand) of sets and cache performance can be estimated by sampling a few sets. Figure 4 describes the Set Dueling mechanism to select between two policies P0 and P1. The mechanism uses Set Dueling Monitors (SDM) to estimate the cache performance of each of the two policies. A few sets of the cache are dedicated to always use policy P0, thereby forming SDM-P0. Similarly, another few sets of the cache are dedicated to always use policy P1, thereby forming SDM-P1. The remaining sets are called follower sets, and they follow the better performing policy between P0 and P1. A saturating counter (PSEL) tracks which of the two SDMs have fewer misses: misses in SDM-P0 increments PSEL and misses in SDM-P1 decrements PSEL. Follower sets of the cache use P0 if the Most Significant Bit (MSB) of PSEL is 0 and P1 otherwise.

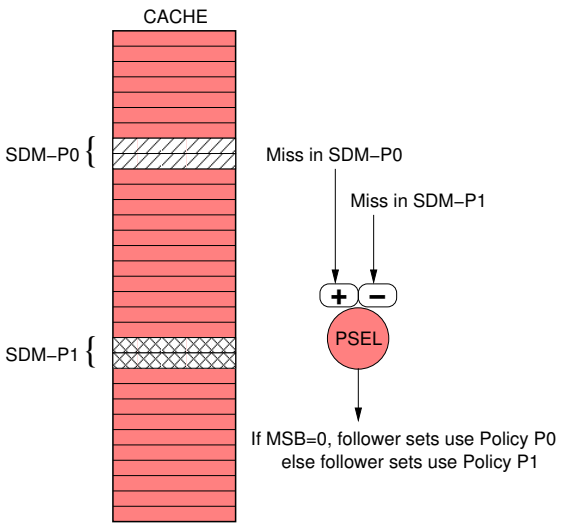

Figure 4. Set Dueling based selection between two policies: P0 and P1

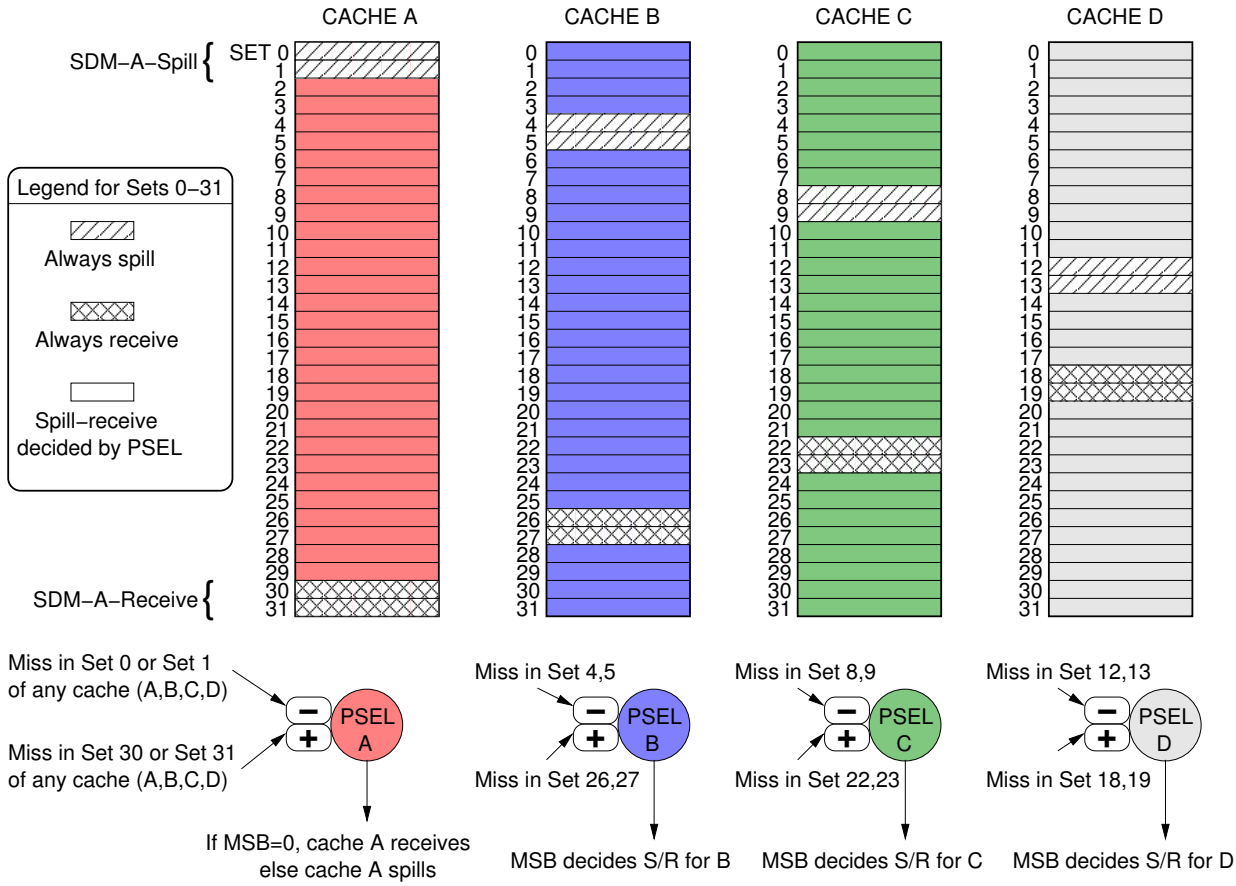

Figure 5. Dynamic Spill-Receive using Set Dueling

3.3. Dynamic Spill-Receive Architecture

We propose *Dynamic Spill-Receive (DSR)* architecture that uses Set Dueling to implement efficient cache sharing in private caches. In DSR, each cache learns whether it should act as a "spiller cache" or "receiver cache" in order to minimize overall cache misses. Figure 5 shows the DSR architecture for a system containing four private caches (cache A to D). For simplicity, we assume each cache consists of 32 sets. Each cache dedicates two sets to estimate the cache performance when it spills and another two sets to estimate the cache performance when it receives spilled lines from other applications. For example, cache A dedicates Set 0 and Set 1 to form SDM-Spill and these sets "always spill" their evicted lines. Cache A also dedicates Sets 30 and 31 to form SDM-Receive and these set can "always receive" spilled lines from other applications. A saturating counter (PSEL-A) keeps track of which of the two policies (spill or receive) when applied to cache A minimizes overall cache misses. A miss in Set 0 or Set 1 of *any* of the cache decrements PSEL-A, whereas, a miss in Set 30 or Set 31 of *any* of the cache increments PSEL-A. Note that the benefit of spilling is obtained only when there is a hit in the remote cache, therefore, it is important to take global cache miss rate into account instead of just local cache miss rate. The most significant bit (MSB) of PSEL-A then indicates which of the two policies (spill/receive) when applied for cache A minimizes overall misses. This policy is used for the remaining 28 sets of cache A.

Thus, the S/R bit for cache A is 0 for sets 30 and 31, is 1 for sets 0 and 1, and is equal to the MSB of PSEL-A for all other sets. Similarly, other three caches (B,C,D) in the system learn which policy (spill/receive) when applied to the cache minimizes overall misses using their individual PSEL counters. The storage overhead incurred by DSR is one PSEL counters per cache. We use a 10 bit PSEL counter in our studies. The last level cache (L2 cache) in our baseline contains 1024 sets. We dedicate 32 sets from each cache to SDM-Spill and another 32-Sets to SDM-Spill.

3.4. Selection of Dedicated Sets

A set dedicated to one of the SDM must not be dedicated to any other SDM of the same cache or any other cache in the system. Therefore, the sets dedicated to SDM-Spill and SDM-Receive for all the caches must be selected in a non-overlapping manner. The sets for SDM can be selected randomly and a separate storage structure can track which

sets belong to SDM-Spill and which sets belong to SDM-Receive for each of the cache. The storage structure can be obviated if the sets dedicated to SDM are selected based on a hash function of the set index of the cache [10][6].

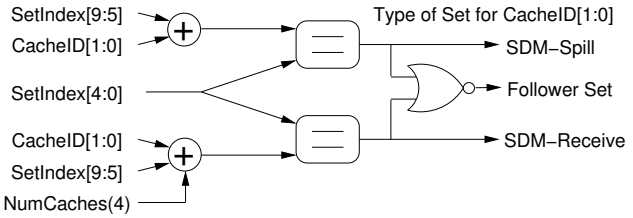

Figure 6. Logic for selecting dedicated sets

The baseline cache contains 1024 sets (indexed by SetIndex[9:0]). Figure 6 describes the logic circuit for selecting 32 sets each for both SDM-Spill and SDM-Receive for each of the four caches. All caches use a separate circuit each driven with corresponding value of CacheID[1:0].

4. Experimental Methodology

4.1. Configuration

We use an in-house CMP simulator for our studies. The baseline configuration is a four-core CMP with the parameters given in Table 1. We use a simple in-order core model so that we can evaluate our proposal within a reasonable simulation time. Unless otherwise specified, the L2 cache in the system is 1MB private per core with a 10 cycle hit latency to the local L2 cache. Cache hits in other L2 caches incur an additional latency of 40 cycles. All caches in the baseline use a uniform linesize of 64B and use LRU replacement policy. Memory is 300 cycles away.

Table 1. Baseline Configuration

System	Four Core CMP
Processor Core	single issue in-order, five stage pipeline
L1 caches (Private)	I and D : 16KB, 64B line-size, 4-way
L2 cache (Private)	1MB, 64B line-size, 16-way LRU repl. 10-cycle local-hits, 40-cycle remote hits
Memory	300-cycle access latency minimum
Off-chip Bus	16B-wide split-transaction, 4:1 speed ratio

4.2. Workloads

Table 2 shows the 12 SPEC CPU2000 benchmarks used in our studies. A representative slice of 250M instructions was obtained for each benchmark using a tool that we developed using the SimPoint methodology. The benchmarks broadly belong to one of the two categories. The first six benchmarks are called *Giver* applications as their CPI does not increases significantly when the cache size is halved from 1MB to $\frac{1}{2}$MB. The other six applications are termed *Taker* applications because their CPI decreases significantly if the cache size is doubled from 1MB to 2MB.

Table 2. Benchmarks Classification (Based on CPI Normalized to 1MB)

"Giver" (G) Applications						
CPI	crafty	eon	fma3d	applu	equake	lucas
$\frac{1}{2}MB$	1.02	1.0	1.0	1.0	1.0	1.0
$1MB$	1.0	1.0	1.0	1.0	1.0	1.0
$2MB$	0.998	1.0	0.997	0.995	0.986	1.0
"Taker" (T) Applications						
CPI	art	ammp	bzip2	galgel	twolf	vpr
$\frac{1}{2}MB$	1.28	1.55	1.33	1.79	1.72	1.73
$1MB$	1.0	1.0	1.0	1.0	1.0	1.0
$2MB$	0.24	0.64	0.74	0.54	0.55	0.66

We form a four-threaded workload by combining four separate benchmarks. We run all possible four-threaded combinations of 12 benchmarks, namely ${}^{12}\mathrm{C}_4$ = 495 workloads. To provide insights in our evaluation, we classify these workloads into five categories depending on how many "Giver" applications (G) and how many "Taker" applications (T) are present in the workload. Table 3 describes this classification. The workloads in $G4T0$ category are unlikely to improve performance with cache sharing since none of the applications in the workload benefit from increased cache capacity. Workloads in $G0T4$ category have very high cache contention and need robust cache sharing.

Table 3. Workload Summary

Type	Description of Workload	Total workloads
G4T0	Four Givers + zero Takers	${}^{6}\mathrm{C}_4 \cdot {}^{6}\mathrm{C}_0 = 15$
G3T1	Three Givers + one Takers	${}^{6}\mathrm{C}_3 \cdot {}^{6}\mathrm{C}_1 = 120$
G2T2	Two Givers + two Takers	${}^{6}\mathrm{C}_2 \cdot {}^{6}\mathrm{C}_2 = 225$
G1T3	One Givers + three Takers	${}^{6}\mathrm{C}_1 \cdot {}^{6}\mathrm{C}_3 = 120$
G0T4	Zero Givers + four Takers	${}^{6}\mathrm{C}_0 \cdot {}^{6}\mathrm{C}_4 = 15$
All	All of the above	$\sum = 495$

All workloads are simulated till each application in the workload executes at-least 250M instructions. When a faster thread finishes its 250M instruction, then it is restarted so that it continues to compete for cache capacity. However, statistics are collected for only the first 250M instruction for each application.

4.3. Metrics

The three metrics commonly used to quantify the aggregate performance of a system in which multiple applications execute concurrently are: Throughput, Weighted Speedup, and Hmean-Fairness. The $Throughput$ metric indicates the utilization of the system but it can be unfair to a low IPC application. The *Weighted Speedup* metric indicates reduction in execution time. The *Hmean-Fairness* metric balances both fairness and performance [9]. We will use all three metrics for key performance comparisons. We also evaluate DSR with regards to Quality-of-Service (QoS) in Section 6. We defer the discussion of QoS to that section.

5. Results and Analysis

We compare the performance of the baseline without spilling to three other configurations: Shared cache of 4MB size, DSR, and Cooperative Caching (CC). The shared cache is 4-way banked with 10 cycle latency for local-bank and 20 cycle latency for remote-bank. A high-bandwidth crossbar interconnect is assumed for the shared cache without penalizing it for the area overhead. The performance of CC is highly dependent on the parameter *spill-probability* [1]. Therefore, for each workload, we evaluate five configurations of CC, each with spill probability of 0%, 25%, 50%, 75% and 100% respectively, and selected the one that gives the best performance for the given workload. We call this $CC(Best)$. In our evaluation, the numbers reported for a suite are the geometric mean averages measured for all of the workloads in a given workload suite.

5.1. Performance on Throughput Metric

Figure 7 shows the throughput of shared cache, DSR, and CC(Best) normalized to the baseline (no spilling). The geometric-mean average for each of the five workload categories and all 495 workloads are shown.

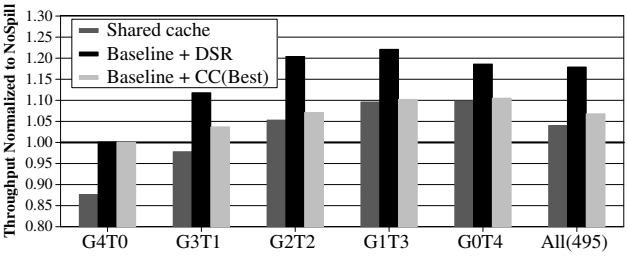

Figure 7. Throughput of shared cache, DSR, and CC over baseline.

G4T0 contains workloads that do not benefit from extra capacity so the increased latency of shared cache hurts throughput. As cache capacity becomes more important, shared cache has better throughput than the baseline. DSR outperforms baseline by 11.8% for G3T1, 20.4% for G2T2, and 22.1% for G1T3. Workloads in G0T4 need more capacity for all applications, still, doing spill-receive intelligently improves the throughput by 18.6%. On average, DSR improves throughput by 18% across all 495 workloads. Comparatively, CC(Best) – even with the best-offline spill parameter – improves throughput by only 7% on average.

Although DSR has 18% more throughput than baseline, it is important that this does not come at the expense of significant degradation in throughput of some workloads. To that end, Figure 8 shows the *S-Curve*[2] of the throughput improvement of DSR over baseline for all 495 workloads.

[2] An S-curve is plotted by sorting the data from lowest to highest. Each point on the graph then represents one data-point from this sorted list.

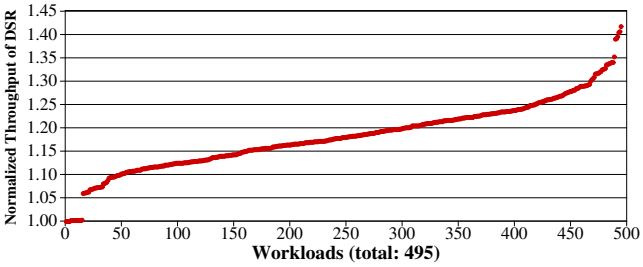

Figure 8. S-Curve for throughput improvement of DSR over baseline (no spilling)

5.2. Performance on Weighted Speedup

The throughput metric gives more weightage to the application with higher IPC, therefore it can be unfair to the slower applications. The Weighted Speedup (WS) metric gives equal weightage to the relative speedup of each application. Figure 9 shows the weighted speedup of shared cache, baseline, DSR, and CC(Best).

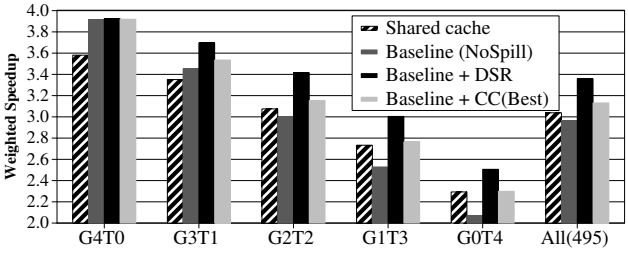

Figure 9. Weighted Speedup of shared cache, baseline, DSR, and CC(Best).

For the workload in G4T0 category, all private cache configurations (baseline, DSR, and CC(Best)) achieve close to ideal weighted speedup, while the higher latency of shared cache causes a degradation. As the number of Taker applications in the workload increases, the contention for cache capacity increases as well and weighted speedup of all scheme starts to decrease. However, DSR consistently has the best performance in all categories that has at-least 1 Taker applications. For the workloads in the G0T4 category, the weighted speedup of the baseline is almost half of the ideal value of 4. DSR tries to minimize overall misses thereby increasing the weighted speedup from 2.07 to 2.5. On average, over all 495 workloads, DSR improves weighted speedup by 13.4% over baseline, and CC(Best) improves weighted speedup by 6%.

We also evaluate the caching schemes using the *"Fair" Speedup* (FS) metric proposed in [2] and found that DSR improves FS by 18% over the baseline. Comparatively, CC(Best) improves average FS by 9%. Although FS uses *harmonic mean* to penalize slowdowns harshly than WS, FS uses a much weaker reference for measuring speedups (1MB per core) compared to the 4MB per core used in WS.

5.3. Performance on Hmean Fairness

The Harmonic-mean based performance metric [9] has been shown to balances both performance and fairness. Figure 10 shows this metric for shared cache, baseline, DSR, and CC(Best). For G4T0 all three private caching schemes have close to ideal value of 1. Shared cache has lower value than 1 because it has higher latency than the 4MB 10-cycle cache used as reference in computing the metric. Partitioning the caches equally does not result in fair allocation compared to the reference of 4MB used in evaluation, therefore, the dynamic allocation of shared cache allows it to outperform baseline for all other categories. On average, for all the 495 workloads, DSR has a value of 0.77, compared to 0.57 for the baseline. Thus, DSR not only improves performance significantly but it also balances fairness well.

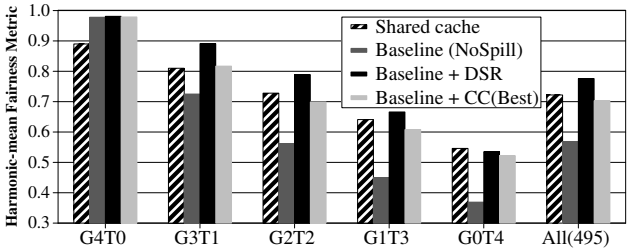

Figure 10. Hmean-Fairness metric

5.4. Comparison with Best-Offline Policy

DSR is a runtime technique that tries to converge to the spill-receive policy that gives the fewest misses. We compare DSR to an offline scheme that executes the workload for all 16 combinations of spill-receive for a four-core system and choose the one that gives best performance. We call this policy *Best-Offline*.[3] Best-Offline may not be practical to use given the extensive offline analysis or even impossible when different applications form a workload at runtime. Nonetheless, it gives us a reference for comparison.

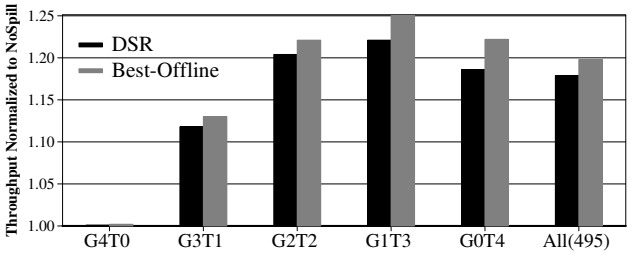

Figure 11. DSR vs. Best-Offline policy

Figure 11 compares the throughput improvement of DSR and Best-Offline (for maximum throughput). Although DSR is a low overhead, practical, runtime mechanism, it still provides 90% of the performance benefit (17.9% vs. 19.8%) of the impractical Best-Offline scheme. The difference is because DSR tries to minimize misses and Best-Offline explicitly chooses maximum throughput.

[3]We also studied another best-offline policy that chooses from a 3-way decision for each core: spill-receive-neither. We executed all $3^4 = 81$ combinations for each workload and selected the one that gave the best performance. The average improvement with the 3-way best-offline policy is similar to that obtained with the binary (spill-receive) best-offline policy.

5.5. Scalability of DSR to Larger Systems

We also evaluate DSR for an 8-core and a 16-core system. We form 100 workloads for each system by randomly combining from the 12 SPEC benchmarks. Figure 12 shows the improvement in throughput of DSR compared to that with no spill for the 8-core and 16-core system respectively.

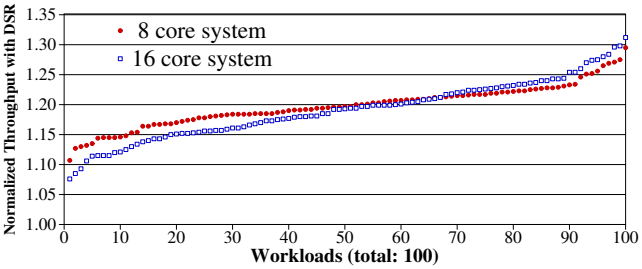

Figure 12. DSR on 8-core and 16-core CMPs

DSR improves throughput on average by 19.7% for the 8-core system and by 19% for the 16-core system, while never degrading the throughput of any of the 200 workloads. Thus, DSR is a scalable, robust, and practical mechanism, given that the improved performance is still obtained with the storage overhead of only one 10-bit counter per core even for the 8-core and 16-core systems.

5.6. Effect on IPC of Each Application

Thus far, we have used metrics that indicate aggregate performance of the workload. We now show the effect of DSR on the performance of individual application within the workload. Figure 14 shows the normalized IPCs of each of the 1980 applications (4x495 workloads) compared to the baseline. For 20 out of 1980 applications, DSR has slowdown of more than 5%. Thus, for 99% of the applications, DSR retains the performance isolation of private caches.

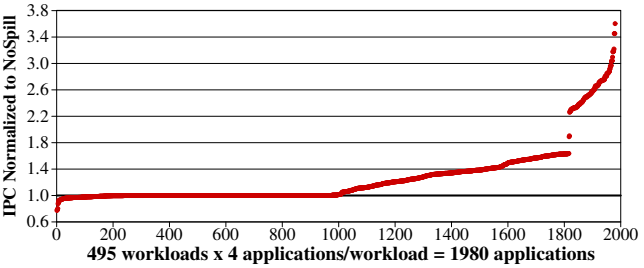

Figure 13. S-Curves for IPC with DSR normalized to the baseline

6. Quality-of-Service Aware DSR

Thus far, we have tried to improve overall performance, even if this means reducing the performance of one of the application if it provides huge improvement in performance of other applications in the workload. However, in some scenarios, such as interactive applications and service level agreements, there is a certain level of guaranteed performance required. Such scenarios require *"Quality of Service (QoS)"*, which means that the worst-case performance of the applications remains similar to (or better than) the baseline case of private caches. With DSR, we reduce the IPC of about 1% of the applications by more than 5%. In some cases this may be unacceptable. DSR must try to improve overall performance while retaining the worst-case performance similar to the baseline. We propose an extension of DSR architecture that facilitates such QoS guarantees. We call this *QoS-Aware DSR*. To implement QoS, we must know the increase in misses with DSR compared to the baseline. We leverage the fact that the spiller sets for each cache do not receive lines from any other cache. Therefore, we can track the misses in spiller sets $(MissInSpillerSets)$ and use it to estimate the misses in the baseline system with no spilling $(MissesWithBaseline)$ using Equation 1.

$$MissesWithBaseline = \frac{NumSetsInCache \cdot MissInSpillerSets}{NumSetsDedicatedToSpillerSets} \quad (1)$$

$$\Delta MissesWithDSR = MissesWithDSR - MissesWithBaseline \quad (2)$$

$$\Delta CyclesWithDSR = AvgMemLatency \cdot \Delta MissesWithDSR \quad (3)$$

$$QoSPenaltyFactor = max(0, \frac{\Delta CyclesWithDSR}{TotalCycles}) \quad (4)$$

The number of misses with DSR $(MissesWithDSR)$ for each cache can be measured at runtime using a counter. The change in misses with DSR $(\Delta MissesWithDSR)$ is given by Equation 2. The change in execution time because of DSR $(\Delta CyclesWithDSR)$ can then be calculated by multiplying the Δ misses with average memory latency (we use a static number for our system but for out-of-order systems a method similar to the one used in [11] can be employed). We calculate $QoSPenaltyFactor$ as percentage increase in execution time due to DSR, which can be calculated using equation 4, given that a cycle counter register can track the $TotalCycles$. Then, instead of incrementing/decrementing the PSEL counters by 1 on each miss, we give more weightage to the misses of an application (say i) that has higher QoSPenaltyFactor using Equation 5.

$$WeightOfMiss_i = 1 + \lambda \cdot QoSPenaltyFactor_i \quad (5)$$

Where λ is a constant (256 in our studies). We calculate QoSPenaltyFactor once every 5 Million cycles, store it in a $QoSPenaltyFactorRegister$, and use it for the next 5 Million cycles. QoS-Aware DSR requires a per-core storage overhead as follows: 3 bytes for MissInSpillerSets, 3 bytes for MissInDSR, 1 byte for QoSPenaltyFactor Register (6.2 fixed point format), and 12 bits for PSEL (10.2 fixed point format). A cycle counter of 4 bytes is shared between all the cores. Thus, the total overhead of enforcing QoS within the DSR architecture is less than 10 bytes per core. If the cycle counter or any of the miss counters overflow, the cycle counter and all the miss counters are halved.

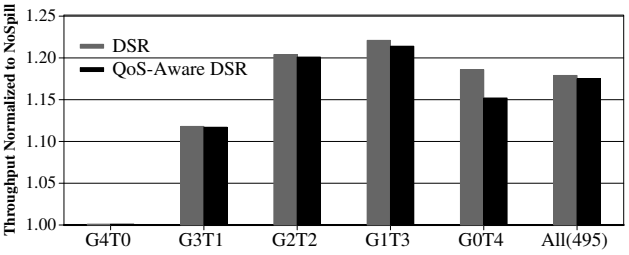

Figure 14. Throughput of QoS-Aware DSR

Figure 14 shows the improvement in throughput with DSR and QoS-Aware DSR. For G0T4 category, QoS constraints reduce throughput improvement form 18% to 15%. On average, the two scheme performs similarly, with DSR providing 18% and QoS-Aware DSR with 17.5%. As G0T4 is the category with the most QoS violations, we show the normalized IPC of each application in this category for DSR and QoS-Aware DSR in Figure 15 (for other categories the two curves are almost identical). QoS-Aware DSR successfully removes the IPC degradation of DSR and all applications have a worst-case IPC similar to the baseline. For all 1980 applications evaluated, 20 out of 1980 applications had an IPC degradation of more than 5% for DSR, whereas, not a single application had that amount of degradation with QoS-Aware DSR. Thus, the DSR architecture can successfully optimize for both QoS and performance.

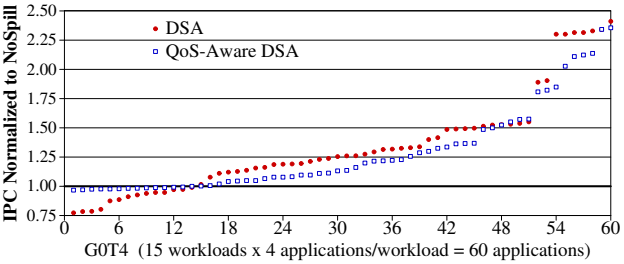

Figure 15. S-Curve of IPC for apps in G0T4 workloads with DSR and QoS-Aware DSR

7. Related Work

Managing Private Caches: When cache lines are concurrently accessed by multiple cores, replication can reduce cache access latency at the expense of reducing the number of unique lines that can be stored on-chip. Several proposals [4][3] have tried to balance this latency vs. capacity trade-off. These schemes contain a replication control parameter that determines the percentage of lines that can be replicated. Adaptive Selective Replication [1] is a dynamic mechanism that learns the best replication parameter at runtime using large tagged structures. The reduced-capacity vs. improved-latency trade-off solved by these proposals is orthogonal to the capacity sharing problem solved by DSR.

Managing Shared Caches: LRU-managed shared cache allocates capacity between competing applications on a demand basis. Cache performance can be improved if cache space allocated to streaming applications can be minimized. TADIP [6] is a simple high-performance scheme that uses Set Dueling and Adaptive Insertion [10] to reduce harmful cache interference in shared caches. However, TADIP is applicable to only shared caches and does not solve the problem of increased latency and high-bandwidth interconnect requirement of a shared cache. Figure 16 shows the throughput improvement of shared cache with TADIP and private cache with DSR. On average, TADIP improves throughput by 15% and DSR by 18%. DSR can also be implemented on a cache that uses DIP [10] for demand lines (spilled lines are always inserted in MRU position). DSR+DIP improves throughput by 23% over NoSpill+LRU and by 17% over NoSpill+DIP (not shown).

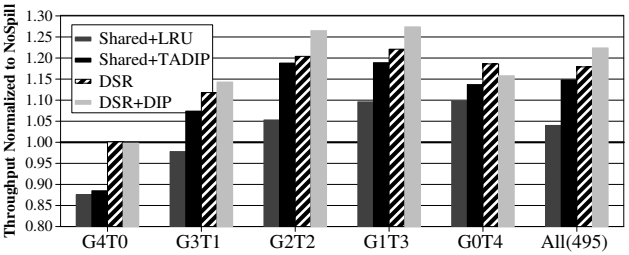

Figure 16. Throughput: TADIP vs. DSR.

Fairness and Quality-of-Service: Cooperative Cache Partitioning (CCP) [2] tries to optimize both fairness and quality of service using time-sharing of cache partitions. MTP needs cache requirement curves for each application. Obtaining such information using profiling may be impractical and using time-sampling requires huge training time. Furthermore, even if MTP has such information, it still requires 2 bits/line (storage: 16KB) to measure cache capacity given to each application. Unlike CCP, DSR does not require time sampling and design changes, and can still enforce QoS. A practical version of MTP can always use DSR (instead of CC) to improve both performance and fairness.

8. Summary

CMPs with private last-level cache suffer from the inability to share cache capacity between cores when the cache requirement of each core varies. Capacity sharing can be implemented in private caches by spilling an evicted line from one cache to another cache. However, previous proposals do spilling without taking into account the cache requirement of each core, which limits the performance improvement obtained with spilling. The goal of this paper is to enable efficient high-performance capacity sharing in private caches by using a practical spill mechanism that takes cache requirement of different cores into account. To that end, this paper makes the following contributions:

1. We propose the *Spill-Receive* architecture, where each cache is allowed to either spill evicted lines or receive spilled lines but not both. This prevents spiller caches from giving their local capacity while they are trying to gain more cache capacity remotely and vice-versa.
2. We propose the *Dynamic Spill-Receive (DSR)* architecture that learns the best spill-receive decision for each cache at runtime. We show that DSR improves average throughput by 18%, weighted speedup by 13% and Hmean fairness by 36% for a 4-core CMP.
3. We propose a simple extension of DSR that guarantees Quality of Service while retaining high performance.

Acknowledgments

The author thanks William Starke, Ravi Nair, Viji Srinivasan, and Trey Cain for their comments and feedback.

References

[1] B. M. Beckmann et al. ASR: Adaptive selective replication for CMP caches. In *MICRO-2006*.
[2] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS-2007*.
[3] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA-2006*.
[4] Z. Chishti et al. Optimizing replication, communication, and capacity allocation in CMPs. In *ISCA-2005*.
[5] M. D. Dahlin et al. Cooperative caching: using remote client memory to improve file system performance. In *OSDI-1994*.
[6] A. Jaleel et al. Adaptive insertion policies for managing shared caches. In *PACT-2008*.
[7] P. Kongetira et al. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.
[8] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *ISCA-2005*.
[9] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS-2001*.
[10] M. K. Qureshi et al. Adaptive insertion policies for high-performance caching. In *ISCA-2007*.
[11] M. K. Qureshi et al. A case for MLP-aware cache replacement. In *ISCA-2006*.