Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs

Karthik T. Sundararajan† Vasileios Porpodas† Timothy M. Jones‡
Nigel P. Topham† Björn Franke†

†School of Informatics
University of Edinburgh
{t.s.karthik, v.porpodas}@ed.ac.uk
{npt, bfranke}@inf.ed.ac.uk

‡Computer Laboratory
University of Cambridge
timothy.jones@cl.cam.ac.uk

Abstract

Intelligently partitioning the last-level cache within a chip multiprocessor can bring significant performance improvements. Resources are given to the applications that can benefit most from them, restricting each core to a number of logical cache ways. However, although overall performance is increased, existing schemes fail to consider energy saving when making their partitioning decisions.

This paper presents Cooperative Partitioning, a runtime partitioning scheme that reduces both dynamic and static energy while maintaining high performance. It works by enforcing cached data to be way-aligned, so that a way is owned by a single core at any time. Cores cooperate with each other to migrate ways between themselves after partitioning decisions have been made. Upon access to the cache, a core needs only to consult the ways that it owns to find its data, saving dynamic energy. Unused ways can be power-gated for static energy saving.

We evaluate our approach on two-core and four-core systems, showing that we obtain average dynamic and static energy savings of 35% and 25% compared to a fixed partitioning scheme. In addition, Cooperative Partitioning maintains high performance while transferring ways five times faster than an existing state-of-the-art technique.

1 Introduction

On-chip caches play a significant role in improving the performance of a processor. Within a chip multiprocessor (CMP), a multi-level cache hierarchy is employed, with the last-level cache (LLC) being the largest and often shared among all cores on the chip. Decreasing its energy consumption is important because it is responsible for a significant fraction of the total processor power budget. However, any efficient energy-saving technique should cause minimal or no performance degradation.

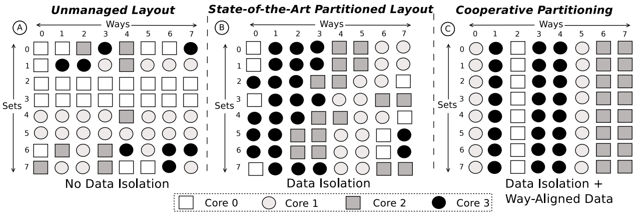

Figure 1. Data allocation across partitioning schemes in shared last-level caches.

Cache energy reduction techniques have been widely studied in the past. Most work has focused on single-core designs. These turn off parts of the cache, to reduce static energy, or predict the ways that will be accessed, to reduce dynamic energy. However, these schemes are not directly applicable to a CMP LLC due to the filtering effects of the higher cache levels making access patterns hard to predict.

In contrast, there has been significant recent work in partitioning a shared LLC for performance [27, 28, 31, 32]. Applications are restricted to a number of logical ways within the cache, giving the most resources to the programs that obtain the most benefit from them. While these techniques can unlock significant performance increases, they do not consider energy saving when partitioning.

In our work we take a novel approach to LLC partitioning by forcing data belonging to each core to be *way-aligned* across all sets. Figure 1 gives an example of how our approach differs from existing partitioning schemes. An unmanaged cache is shown in A, where data belonging to the cores is entirely mixed across sets and ways. In B, a cache partitioning technique has been applied so that the number of ways owned by each core is constant across all sets. However, within the sets, data from each core can reside in any way. Our approach is shown in C. We apply the same partitions as in B, but enforce data way-alignment so that a way is owned entirely by a single core at a time.

The energy savings we can achieve are two-fold. First, dynamic energy can be reduced on each access because a core only needs to consult the ways that it currently owns. Our scheme guarantees that its data will never be found anywhere else in the cache. Second, when a whole way is unused by any core, it can be turned off to save static energy. Implementing our technique in a partitioned architecture combines large energy savings with high performance.

We call our approach *Cooperative Partitioning*, because cores cooperate with each other after partitioning to migrate ways between themselves. Using our technique in a two-core system brings dynamic energy savings of 32% and static energy savings of 25% compared to a fixed partitioning scheme. In a four-core environment, dynamic and static energy savings of 31% and 20% respectively are achieved. In addition, due to our cooperative takeover algorithm for transferring ways, migration is five times faster on average than a state-of-the-art partitioning scheme and flushes less data back to memory.

The rest of this paper is structured as follows. Section 2 describes our cache monitoring scheme and partitioning algorithm and explains the internals of our cache architecture. This section also discusses the overheads associated with the cache reconfiguration. Section 3 describes the experimental methodology, workloads, and metrics used for evaluation. Section 4 evaluates our approach on two-core and four-core systems and Section 5 analyses the reasons behind our results. Section 6 describes the related work and the importance of our cache architecture. Finally, Section 7 presents our conclusions.

2 Cooperative Partitioning

Our cooperative partitioning scheme is split into two distinct parts. The first monitors cache usage and determines the optimal partitions for the running applications. The second enforces the required partitions, enabling static and dynamic energy savings. As is common in last level caches, we assume accesses are serial. Therefore dynamic energy savings come from the tag side only.

An overview of our cache partitioning system is shown in Figure 2. During the first phase, LLC accesses are monitored and, periodically, partitioning decisions are made about the number of ways to allocate to each core according to their cache requirements. In the second phase, this information is used to set the appropriate access permission registers that determine how each core can access each LLC way. During this phase, ways are gradually migrated between cores or turned off.

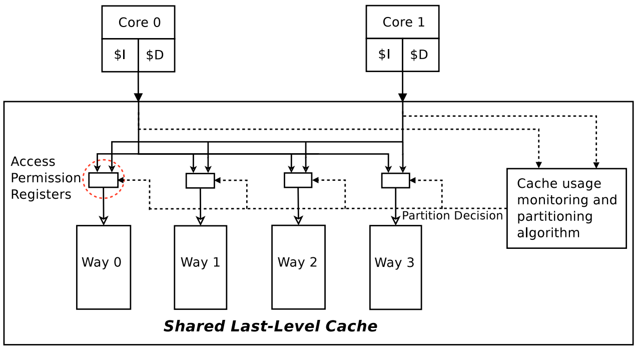

Figure 2. An overview of the cooperative partitioning architecture.

To enforce the required partitioning decisions and enable way-alignment of data, we introduce two new registers for each way called the read access permission (RAP) and write access permission (WAP) registers. These allow specific cores to read and write in each way.

We first describe our cache monitoring scheme, then explain the RAP and WAP registers in more detail. Then we describe how reconfiguration is achieved by transferring ways between cores. We call this cooperative takeover and give an example of its workings. Finally, we describe the overheads associated with our cache architecture.

2.1 Usage Monitoring and Partitioning

Our cache architecture builds on prior work to determine the optimal partitions for the LLC. As in state-of-the-art schemes, we target a cache that is shared among multiprogrammed workloads [2, 5, 14, 20, 32]. Accesses are tracked by utility monitors [20] for computing each application's use of the cache. Other partitioning schemes have also made use of these monitors [32], although they could also be specified through the operating system [21].

We use a modified UCP look-ahead algorithm [20] to determine partitions, shown in Algorithm 1. This contains a threshold value that is used when allocating ways to a core. The threshold controls the decrease in miss-ratio for each application, preventing each core from being awarded additional ways unless it can significantly benefit from them. Therefore, after running the algorithm, there may be ways that are not allocated to any core. These can be turned off for static energy savings with minimal loss of performance.

2.2 Cache Partitioning Control

To control each core's access to the ways, and enforce way-aligned data, we introduce an additional RAP register and WAP register for every way within the cache. Each

Algorithm 1: Cores obtain extra ways when their performance increases above a threshold.

```
balance = N; /* Number of Blocks to be allocated */
allocations[i] = 0; /* For each competing application, i */
prev_max_mu = 0;

while balance do
    foreach application i do
        alloc = allocations[i];
        max_mu[i] = get_max_mu(i, alloc, balance);
        blocks_req[i] = min blocks to get max_mu[i] for i;
    winner = application with maximum value of max_mu;
    /***** Modified implementation starts here *****/
    if |prev_max_mu - max_mu| < (prev_max_mu * T_hold) then
        allocations[winner] += blocks_req[winner];
    balance -= blocks_req[winner];
    prev_max_mu = max_mu;
    /***** End of modifications *****/
return allocations;

get_max_mu(p, alloc, balance):
max_mu = 0;
for j=1; j<=balance; j++ do
    mu = get_mu_value(p, alloc, alloc+j);
    if $mu > max\_mu$ then
        max_mu = mu;
return max_mu;

get_mu_value(p, a, b):
return = (miss_a - miss_b)/(b-a);
```

RAP register has one bit per core to indicate whether that core can read from the associated way. This is used in conjunction with the WAP register for that way. This also has one bit per core and indicates whether the core has permission to write to the way or not.

For each core and way, there are three possible modes of operation. If both registers are set for a particular core, then this core can both read and write in that particular way. Otherwise, if the RAP register is set and the WAP register is unset, then the core has only read permission for accessing that way. If both registers are unset, then the core can neither read nor write that way.

Only one core can have full access (RAP set and WAP set) to a particular way at any given time. In fact, under normal conditions only one core can have any access to the way. However, during a transition period, when reconfiguration is taking place, one core can have full access and another can have read-only access. This lasts until the whole way has been transferred from one core to the other and is discussed in more detail in Section 2.3. Algorithm 2 describes how the RAP and WAP registers are set at the beginning of a transition period.

The RAP and WAP registers serve three purposes. First, they enforce the cache partitioning that is currently in operation by restricting cores' accesses to only the ways that they are allocated. Second, they enable dynamic energy savings because cores only need to access the ways that they have permission for, rather than all ways within the cache. Third, when no cores have access to a particular way (RAP and WAP unset for all cores), then the whole way can be turned off for static energy savings.

Algorithm 2: Setting the RAP and WAP registers to initiate cooperative takeover.

```
$Pre$ = Previous way allocations per core;
$Cur$ = Current way allocations per core;

for $i = 0; i < n; i = i + 1$ do
    if $Pre[i] < Cur[i]$ then /* Core $i$ acts as a recipient */
        receive[i] = $Cur[i] - Pre[i]$; donate[i] = 0;
    else if $Pre[i] > Cur[i]$ then /* Core $i$ acts as a donor */
        donate[i] = $Pre[i] - Cur[i]$; receive[i] = 0;

for $i = 0; i < n; i = i + 1$ do
    for $j = 0; j < n; j = j + 1$ do
        if receive[i] > 0 and donate[j] > 0 then
            if donate[j] > receive[i] then
                donation = receive[i];
            else
                donation = donate[j];
            for $d = 0; d < donation; d = d + 1$ do
                w = Random way owned by core j;
                RAP[w][i] = 1; WAP[w][i] = 1; WAP[w][j] = 0;
                receive[i] -= 1; donate[j] -= 1;

/* Turn ways on or off */
for $i = 0; i < n; i = i + 1$ do
    if donate[i] > 0 then
        for $d = 0; d < donate[i]; d = d + 1$ do
            w = Random way owned by core i;
            WAP[w][i] = 0;
        donate[i] = 0;
    else if receive[i] > 0 then
        for $r = 0; r < receive[i]; r = r + 1$ do
            w = Random way currently off;
            RAP[w][i] = 1; WAP[w][i] = 1;
        receive[i] = 0;
```

Figure 3 shows an example of the RAP and WAP registers before, during and after a transition period. Initially both cores own two ways and the registers are set accordingly. A partitioning decision is then made, that transfers way 2 to core 0. To allow this, core 0 gets read and write access to way 2, and core 1's write permission is revoked. After the transition period, core 0 has full control of the way and core 1's read permission is also withdrawn.

2.3 Cache Reconfiguration

Once the RAP and WAP registers have been set, the cache must be reconfigured to the new partitioning that is required. To achieve this, we introduce a new technique called cooperative takeover. In this scheme, for each way to be transferred, the donor and recipient cores cooperate to quickly flush dirty data back to memory and allow the recipient core to take full ownership of all lines in the way.

Our scheme avoids the cost of immediately flushing data

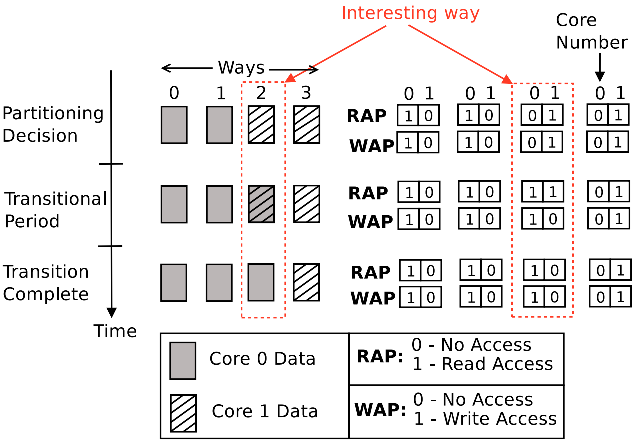

Figure 3. RAP and WAP register changes when transferring way 2 between cores.

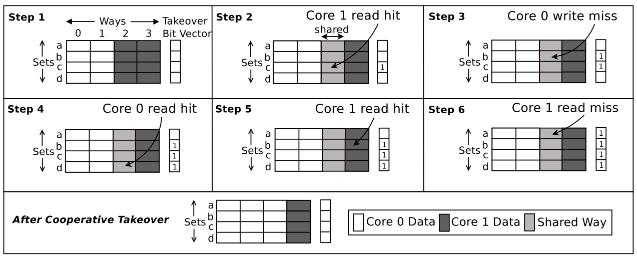

Figure 4. An example of cooperative takeover where core 1 will donate a way to core 0. Whenever either core accesses a set, dirty data is flushed back to memory. Once all sets have been accessed by at least one core, the way can be owned entirely by core 0.

back to memory, but quickly transfers ownership of the whole way, enabling fast realisation of the dynamic energy savings that can be achieved (when the donor core no longer accesses this way). During transitional periods, dynamic energy consumption is higher than normal because multiple cores access the ways that are being transferred. Therefore, we wish to transfer ways as quickly as possible to minimise the length of time that the way is transitioning.

To enable cooperative takeover, the cache is augmented with a takeover bit vector for each core that is the size of the number of sets in the cache (i.e., one bit per set per core). The donor cores' bit vectors are reset at the start of a transition period. Whenever a donor core accesses a particular set, dirty data is flushed back to main memory from the ways that it is transferring. This happens whether it hits or misses on that particular access. At the same time, the bit for that particular cache set in the core's bit vector is set. Additionally, whenever a recipient core accesses a particular set, dirty data is flushed back to memory from the ways that it will be receiving. Again, this occurs whether it hits or misses on that access. In this situation, the bit for that cache set in the donor core's bit vector is set.

The donor core knows that it is donating ways because it has read permission on those ways, but not write permission. The recipient core knows that it will receive certain ways because it will have read and write permission to the ways, but another core will also have read permission. When donating or receiving ways, dirty data is flushed in all ways with read permission on each access.

Bit vectors are reset at the start of a transition period for each donor core that is giving away a way. This could interfere with a prior transition of a different way from a donor core that is still in progress. In this situation the bit vector is still reset and the only result is that the first transition will take longer to complete. However, this situation is rare and we have not seen it in any of our experiments.

2.4 Cooperative Takeover Example

Figure 4 shows an example of cooperative takeover in practice. In this example there are two cores and four cache ways. Initially, the partitioning decision has just been made and two ways are assigned to each core, but core 1 will donate way 2 to core 0. The takeover bit vector for core 1 is totally unset. In the second step, core 1 performs a read that hits in set c in the cache. Its own dirty data from this set in way 2 is flushed back to memory and the takeover bit is set.

Following this, core 0 writes to the cache which misses in set b. In this case, core 1's dirty line from set b, way 2 is flushed and the corresponding takeover bit set as before. When the new line comes in from memory, it can be placed in way 2 instead of replacing an existing line in another way.

Core 0 then has a read hit in set d. In this case the line in way 2 is not dirty so does not need flushing, but the takeover bit is still set. In the fifth step, core 1 has a read hit in set b. However, the line in way 2 is now owned by core 0 and, even though it is dirty, does not need flushing back to memory. Core 1 can see this because the corresponding takeover bit is already set. Finally, core 1 has a read miss in set a. Again, no dirty data needs flushing, and the takeover bit is set. However, when the line comes into the cache, it will replace the data in way 3 (core 1's only way).

At this point, the all takeover bits are set and therefore core 0 takes complete ownership of way 2. This is achieved simply by resetting the bit for core 1 in the RAP register for way 2, meaning that it no longer has read permission for that way (write permission had already been withdrawn).

2.5 Reconfiguration Overheads

There are four types of overhead associated with our cache partitioning and reconfiguration scheme. These are the changes to the replacement policy, hardware overheads of implementation, and the performance and power overheads of carrying out the partitioning.

Replacement Policy We use the scheme proposed in [20] where an extra two bits are added to each tag entry to distinguish data belonging to each core. Algorithm 2 determines which ways will be transferred between cores, then the replacement algorithm links the corresponding ways accordingly [6, 11, 28].

Hardware Overheads As in other schemes [32], we use an existing cache monitoring scheme to track the usage of each set by each core and require the same hardware as this [20]. To implement cooperative takeover, we only require one bit vector for each core for each set, along with RAP and WAP registers for each way. In the 4MB L2 cache that we use for our four-core experiments in Section 4, this comes to a little over 8k bits. Table 1 details the requirements for the two caches that we study.

Performance Overheads When transferring a way from one core to another, we must select blocks from each set to be given to the recipient core. State-of-the-art schemes are free to choose any block within each set; selecting the LRU block is one method [32]. In our approach we must keep the data way-aligned, so do not have the flexibility to choose blocks on a per-set basis. This makes our scheme closer in performance to a random choice of replacement block. However, in practice, this causes a negligible performance loss compared with prior work and is more than offset by our energy savings.

Power Overheads Since the cache has extra circuitry for monitoring and partitioning, it consumes more power than a regular cache. However, our approach can realise considerable savings in dynamic and static energy, which far outweigh the overheads incurred. Nevertheless, all power overheads are included in our simulated results in Section 4.

2.6 Summary

This section has described Cooperative Partitioning, a high-performance, energy-efficient cache partitioning scheme. We introduce RAP and WAP registers to control access to cache ways, keeping data way-aligned and enabling unused ways to be turned off. During transition periods, when ways are being transferred between cores, both donor and recipient cores cooperate to flush dirty data back to memory. This enables the recipient to quickly take ownership of the ways and maximises the time when dynamic energy savings can be realised.

	Two Core		Four Core	
Hardware Description	Details	Bits	Details	Bits
Takeover Bit Vectors	2048 * 2	4096	2048 * 4	8192
RAP	8 * 2	16	16 * 4	64
WAP	8 * 2	16	16 * 4	64
Total		4128		8320

Table 1. Summary of the hardware overheads of our scheme for two-core and four-core systems.

Parameters	Configuration
Processor	4-wide, out-of-order, 7 stage pipeline
ROB	128 entry
LSQ	48 entry
Branch Pred.	Gshare, minimum 10 cycle misprediction penalty
BTB	1024 entry, 4-way set-associative
L1 ICache	32kB, 64B lines, 4-way, 2 cycle lat
L1 DCache	32kB, 64B lines, 4-way, 2 cycle lat
Shared L2	2MB, 64B lines, 8-way, 15 cycle lat (two-core)
	4MB, 64B lines, 16-way, 20 cycle lat (four-core)
MSHR	128 entry
Memory	8 DRAM banks, 400 cycle lat, 64 outstanding reqs

Table 2. System configuration.

3 Experimental Methodology

This section describes the environment used to evaluate our proposed cache architecture.

3.1 Simulator

We implemented our partitioned cache architecture in Marss-x86 [18]. Table 2 shows the configuration of the system. We simulated a 4-wide, x86-based out-of-order processor with a 7 stage pipeline. We modelled both a two-core and a four-core system to fully evaluate the effects of sharing and partitioning the last level cache. All level 1 caches are private and all processors share a common level 2 cache. We model the DRAM conflicts and bus queueing delays and use Cacti [29] at 45nm to get energy information. Finally, we assume a 5 million cycle phase interval for monitoring and partitioning decisions, as in prior work [20].

3.2 Workloads

We ran all C and C++ benchmarks from SPEC CPU2006 [25], which totals 19 applications; FORTRAN benchmarks could not be incorporated into our simulation environment. To select groups to run in parallel, we first arranged them into categories according to their misses per

Group	Benchmark	MPKI	Group	Benchmark	MPKI
High	Gobmk	9	Low	DealII	0.8
	Lbm	20.1		Gromacs	0.32
	Sjeng	9.5		H264ref	0.89
	Soplex	18		Milc	0.96
Medium	Astar	4.8		Namd	0.25
	Bzip2	3.2		Omnetpp	0.26
	Calculix	1.1		Perlbench	0.98
	Gcc	4.92		Povray	0.1
	Libquantum	3.4		Xalan	0.6
	Mcf	4.8			

Table 3. Workload classification based on misses per kilo instructions (MPKI). The High group has MPKI > 5, Medium is $1 <$ MPKI < 5 and Small has MPKI < 1.

Two Core Workloads		Four Core Workloads	
G2-1	Soplex, Namd	G4-1	Gobmk, Gcc, Perl., Xalan
G2-2	Soplex, Milc	G4-2	Sjeng, Lbm, Calculix, Om.
G2-3	Gobmk, H264.	G4-3	DealII, Sjeng, Soplex, Namd
G2-4	Lbm, Povray	G4-4	Soplex, Sjeng, H264., Astar
G2-5	Gobmk, Perl.	G4-5	Lbm, Libq., Gromacs, Mcf
G2-6	Lbm, Bzip2	G4-6	Gobmk, Libq., Namd, Perl.
G2-7	Lbm, Astar	G4-7	Lbm, Sjeng, Povray, Om.
G2-8	Lbm, Soplex	G4-8	Lbm, Soplex, H264., DealII
G2-9	Soplex, DealII	G4-9	Lbm, Xalan, Milc, Soplex
G2-10	Sjeng, Calculix	G4-10	Sjeng, Povray, Milc, Gobmk
G2-11	Sjeng, Xalan	G4-11	Gobmk, Libq., H264., Gromacs
G2-12	Soplex, Gcc	G4-12	Soplex, Astar, Om., Milc
G2-13	Sjeng, Povray	G4-13	Soplex, Gcc, Libq., Xalan
G2-14	Gobmk, Om.	G4-14	Soplex, Bzip2, Astar, Milc

Table 4. Workload groupings.

kilo instructions (MPKI) within the last level cache. Table 3 shows this classification.

We created 14 two-application workloads by randomly selecting benchmarks so that there was at least one highly memory intensive program (MPKI > 5) in each group. The 14 four-application workloads were created by randomly selecting applications so that groups contained at least one highly memory intensive and one mediumly memory intensive program ($1 <$ MPKI < 5). These are shown in table 4.

We ran each benchmark using the reference inputs, after first skipping the initialisation routines that we discovered through source code inspection. Having fast-forwarded through initialisation, we warmed the caches and branch predictor for 5 million cycles. We then simulated for at least 1 billion instructions per application, as is common practice [9, 32]. Statistics are reported for 1 billion instructions per benchmark, but all applications continued running until the last program in the group had reached 1 billion instructions, to keep contending for cache resources.

3.3 Evaluation Metrics

To measure system performance we use weighted speedup. This shows the reduction in execution time for each benchmark compared to its running in isolation (so higher is better).

$$WeightedSpeedup = \sum_{i=1}^{N} \frac{IPC_{shared}[i]}{IPC_{alone}[i]} \tag{1}$$

IPC_{alone} is the IPC of an application when it is running in isolation, IPC_{shared} is the IPC of the same application when it is running in conjunction with other applications, and N refers to the number of concurrent threads.

3.4 Comparison Approaches

To fully evaluate our partitioning scheme, we compare against four different approaches. *Unmanaged* is the baseline case. This corresponds to an LLC with no partitioning at all. Therefore, all cores compete for cache resources and can evict each others' data at any time. The next approach is *Fair Share* which corresponds to a statically-partitioned cache, where all cores have an equal number of ways, regardless of their memory behaviour.

CPE is a state-of-the-art static cache partitioning architecture for energy efficiency [23]. Using profile data, a static partition of the cache is computed. Applications can only access their designated regions of the cache, and these do not change during runtime. This design is the most flexible in terms of partitioning, because both sets and ways are configurable, leading to significant energy savings. In this comparison we extended the architecture to work with dynamic reconfiguration. To do this, we profiled the applications and then used this data to drive the dynamic partitioning at runtime. Although unrealistic, this scheme serves as a useful comparison against an existing energy-focused technique.

UCP is a state-of-the-art dynamic cache partitioning scheme for high performance [20]. We implemented UCP using its look-ahead algorithm to allocate ways to cores. Finally, *Cooperative Partitioning* is our proposed scheme which aims for high performance and large energy savings.

4 Evaluation

We evaluated Cooperative Partitioning in terms of performance and energy consumption. Results are shown for both two-core and four-core systems. Unless otherwise stated, all results are normalised to the Fair Share scheme and the average used is the geometric mean.

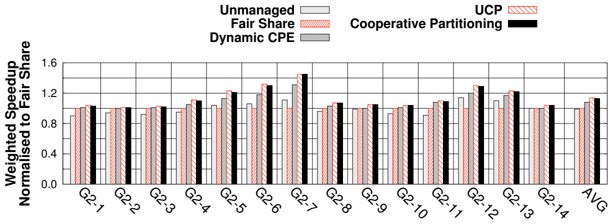

Figure 5. Weighted speedup of two-application workloads.

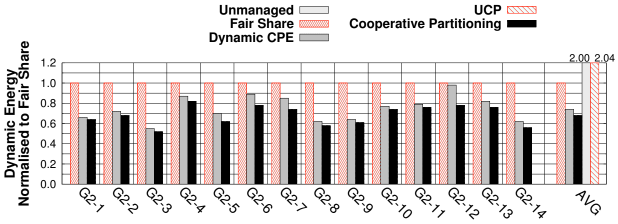

Figure 6. Dynamic energy consumption of the two-application workloads.

4.1 Evaluation of a Two-Core System

Performance Figure 5 shows the weighted speedup of each group of two-application workloads. It is clear that UCP and Cooperative Partitioning consistently have the highest performance across all combination of benchmarks. Further, almost all workloads benefit from some form of partitioning in the LLC. The exceptions are *Group2–5* to *Group2–7*, *Group2–12* and *Group2–13* where the Unmanaged cache performs better than the Fair Share scheme. Applications such as *astar*, *bzip2*, *gcc*, *perlbench* and *povray* benefit significantly from a large amount of cache space, which can be achieved in Unmanaged. In Fair Share, there are fixed boundaries which penalises these programs. Hence Unmanaged achieves a speedup of 14% in *Group2–12* because *gcc* is unconstrained. This motivates the need for a flexible cache partitioning approach, such as UCP or Cooperative Partitioning.

The modified comparison Dynamic CPE algorithm does not perform as well as would be expected, given that it has profile information to guide its partitioning decisions. This is because it has high flushing costs whenever altering the LLC. When workload partitioning changes are infrequent, CPE performs close to UCP and our approach. This is most evident in workloads *Group2–1* to *Group2–3*. On the other hand, when there are frequent changes to the partitions, Dynamic CPE performs worse than UCP and Cooperative Partitioning. For example, in *Group2–7*, CPE achieves a speedup of 1.31, compared with 1.45 for UCP and our scheme, meaning we are 11% faster.

The performance of our approach is close to UCP. On average, we achieve a speedup of 1.13 and UCP achieves 1.14. The reason for this is that we use cooperative takeover of ways and must keep data way-aligned, whereas UCP does not have this restriction. As our results show, in practice this is not a significant issue and we can still get large performance benefits despite this method of partitioning. Further, performance is not the main focus of our approach and we turn our attention to energy consumption in the next section.

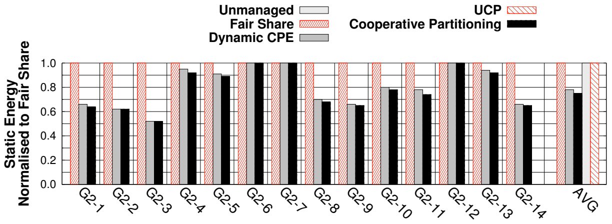

Figure 7. Static energy consumption of the two-application workloads.

Dynamic Energy Figure 6 shows the dynamic energy consumption of each different partitioning scheme. As explained earlier, Unmanaged and UCP do not provide dynamic energy savings as they do not support aligned data (instead each access consults all cache ways). Our approach achieves energy savings of up to 50% compared with the Fair Share scheme. This is because our scheme accesses only 2.9 ways, on average, compared with 8 for the baseline and 4 for Fair Share. The largest savings are achieved by *Group2–3*, the reason being that on average only two ways per access are active.

In *Group2–4*, *Group2–6*, *Group2–7*, *Group2–12* and *Group2–13*, frequent partitioning occurs due to the changing requirements of *astar*, *bzip2*, *gcc* and *povray*. For these workloads, CPE incurs significant overheads from flushing data while partitioning. However, our scheme can cope with these changes and still achieves significant energy savings (between 18% and 26%). On average, Cooperative Partitioning has a dynamic energy consumption of just 68% of the Fair Share scheme, compared to 74% for CPE.

Static Energy Static energy consumption is shown in Figure 7 and we see that again our approach provides significant savings. The Unmanaged, UCP and Fair Share schemes do not reduce static energy because they do not enforce way-aligned data. In Cooperative Partitioning, when workloads under-utilise the cache memory, then the remaining cache ways can be turned off. In CPE, sets and ways can be shut down for static energy savings. As can be

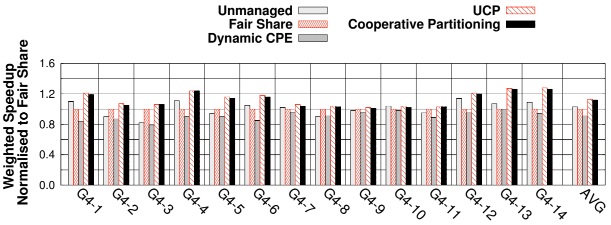

Figure 8. Weighted speedup of the four-application workloads.

seen from Figure 7, in *Group2–2*, static energy savings of 48% are achieved. In this workload, only two ways are required by each application, on average, therefore almost half the cache can be power-gated. However, for *Group2–6*, *Group2–7* and *Group2–12* the cache is used in its entirety, hence no ways can be turned off at all.

The unrealistic Dynamic CPE scheme also saves considerable amounts of energy, although never more than Cooperative Partitioning. On average our approach consumes 75% of the static energy of the Unmanaged, UCP and Fair Share caches, whilst Dynamic CPE consumes 78%. Overall, Cooperative Partitioning consumes less energy than other schemes with performance just 1% away from the best across all comparison approaches.

4.2 Evaluation of a Four-Core System

Performance Figure 8 shows the weighted speedup of our four-application workloads, where it is clear to see that Dynamic CPE performs very poorly. This is due to frequent partitioning changes, leading to significant amounts of flushing in this approach. Dynamic CPE is not scalable across a large number of cores because the number of flushes increases with the number of applications.

Workloads like *Group4–3* have small cache requirements, meaning that there is not a significant amount of performance improvement available beyond the Fair Share scheme. Further, these workloads benefit significantly from partitioning due to thrashing between two applications (*sjeng* and *soplex*) in Unmanaged. On the other hand, workloads like *Group4–13* contain at least one application that requires a large fraction of the cache (i.e., more than a quarter given by Fair Share). In this case the program is *gcc* which obtains 7 ways on average. Fair Share unnecessarily constrains these applications, meaning that other schemes can achieve significant speedups.

As in the two-application workloads, Cooperative Partitioning performs similarly to UCP and is never slower than Fair Share. On average UCP achieves a 1.13 speedup whereas our approach achieves 1.12.

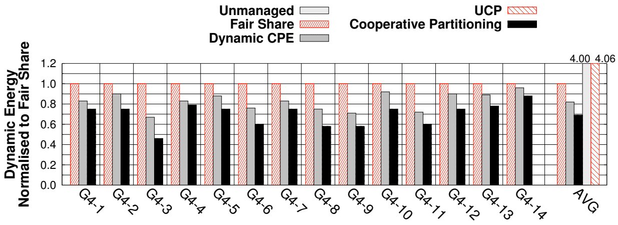

Figure 9. Dynamic energy consumption of the four-application workloads.

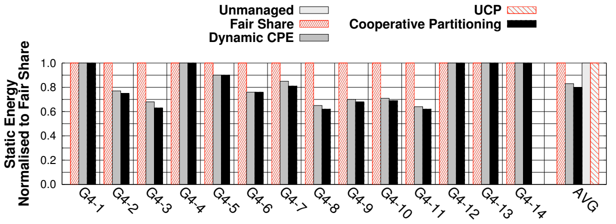

Figure 10. Static energy consumption of the four-application workloads.

Dynamic Energy Figure 9 shows the dynamic energy consumption of these workloads. *Group4–3* obtains the lowest energy consumption that is 46% of the Fair Share scheme. In this workload, two applications get only two ways within the cache and these account for the majority of LLC accesses. Workloads *Group4–4*, *Group4–12* and *Group4–13* have at least one application that benefits from a large LLC. These are *astar* and *gcc*. Therefore it might be reasonable to expect the dynamic energy consumption of these groups to be larger in Cooperative Partitioning than in Fair Share, because these applications are assigned larger cache partitions which consume more energy on each access. However, the energy increases prove to be negligible compared to the consumption from the high MPKI applications that co-execute alongside. The memory intensive applications get assigned to a narrow partition, so consume less energy than in Fair Share. Since these dominate the cache accesses, our scheme ends up with significant dynamic energy savings, even in these cases.

In total, our approach consumes just 69% of the dynamic energy of the Fair Share scheme. In comparison, Dynamic CPE consumes 82%. This is because we access 3.2 ways on average, compared to 4 for Fair Share.

Static Energy Finally, static energy consumption is shown in Figure 10. Here, five workloads completely utilise the cache space, meaning that no ways are turned off. However, in the other groups, large savings are achieved by Cooperative Partitioning, such as *Group4–3*, *Group4–8* and

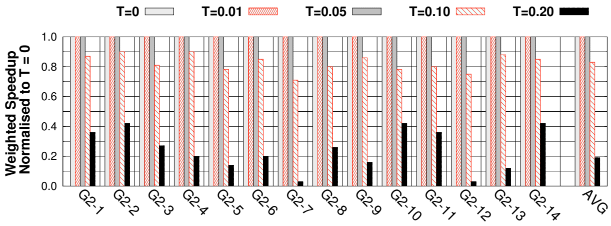

Figure 11. Impact of altering the takeover threshold value on performance.

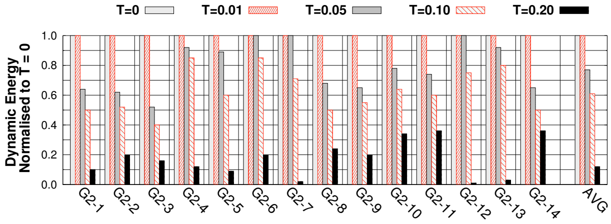

Figure 12. Impact of altering the takeover threshold value on dynamic energy.

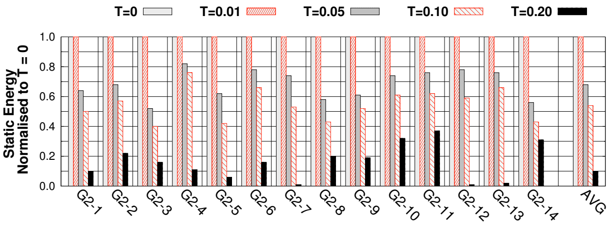

Figure 13. Impact of altering the takeover threshold value on static energy.

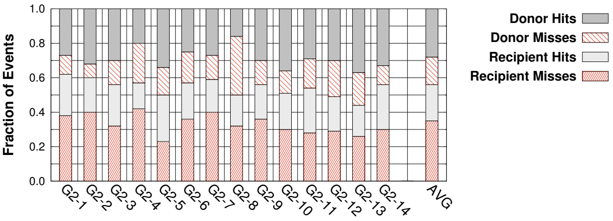

Figure 14. Events that set takeover bits when transferring ways between cores.

Group4–11 where there are 38% savings. This is because these workloads use fewer ways on average compared to other schemes (e.g., two applications have 2 ways and two have 3 ways in *Group4–11*). This leads to an average static energy consumption of 80% of the Fair Share approach.

5 Analysis of Results

Having evaluated cooperative partitioning in terms of both performance and energy consumption, we now analyse the reasons for the benefits seen. We first consider the sensitivity of our algorithm to the turn-off threshold, to show how small values maintain high performance but enable large energy savings. We then show the amount of time taken to transfer ways between cores, which we want to keep as short as possible. To show how cooperation between cores takes place when migrating ways, we show the types of access that set the takeover bit vector, then analyse the LLC to memory bandwidth used when transferring. We conduct this analysis on the two-application workloads only, due to space limitations. In addition, the four-application workloads behave similarly and thus the same conclusions can be applied to them.

5.1 Impact of Takeover Threshold

Figure 11 shows the performance impact of altering the takeover threshold, described in Section 2.3. We explored a range of thresholds, from 0 to 0.2. A threshold value of 0 corresponds to an allocation of ways in the same manner as UCP. Increasing it makes it more difficult for an application to obtain more ways; they are only given out if the application significantly benefits from them. At the other extreme, a threshold value of 1 would mean that no ways were ever allocated to any core.

When a threshold value is 0.05 or less, there is no change in performance compared with a threshold of 0. For a 0.1 threshold, 17% performance loss is incurred. When this is increased to 0.2, all workloads experience large performance losses. The reason is that with such a high value, performance benefits from increasing ways are less than the threshold allows. Therefore, the algorithm falsely prohibits the acquisition of extra ways, leading to poor performance.

On the other hand, large energy savings can be achieved as the threshold value increases. These are shown in Figures 12-13. With a threshold of 0.05, almost all workloads achieve dynamic energy savings (all apart from *Group2–6*, *Group2–7* and *Group2–12*) and all achieve static energy savings. This justifies our use of a 0.05 threshold value for all other experiments, as this provides a good trade-off between high performance and significant energy savings.

5.2 Cooperative Takeover Events

Cooperative Partitioning relies on cooperative takeover to implement its partitioning decisions. Figure 14 shows the breakdown of events that set takeover bits when transferring ways between cores. We show hits and misses by the donor

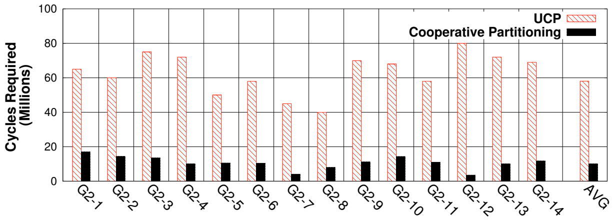

Figure 15. Cycles taken to transfer a way.

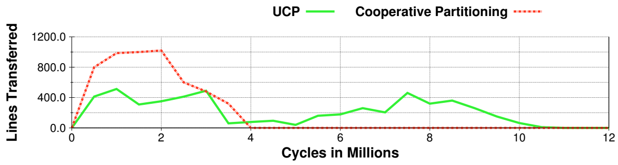

Figure 16. LLC to memory bandwidth usage for flushing data after a partitioning decision.

and recipient cores for each of our fourteen workloads.

In almost all groups, donor hits and recipient misses account for well over half the takeover bits being set. In the majority of cases, these events correspond to approximately two-thirds of the bits being set. The only exception is *Group2–8* where this only happens 48% of the time.

There is an intuitive reason for this finding. The donor core is has enough space in the LLC and, in fact, is giving away one of its ways because it does not need so much room. Therefore most of its accesses will hit in the cache. On the other hand, the recipient core needs more space because its data cannot fit comfortably in the LLC. Therefore, it will miss frequently in the cache until its allocation of ways increases. Hence, these two events are expected to be the most common, and, as Figure 14 shows, in practice they do lead to the majority of takeover bits being set.

5.3 Transition Time

Setting of takeover bits on donor and recipient accesses means that ways can quickly be transferred between cores. To quantify the amount of time this actually takes, consider Figure 15. This shows the average number of cycles for Cooperative Partitioning to transfer each complete way between cores for each workload. For comparison, we show UCP too. Since UCP does not enforce way-aligned data, this value corresponds to the average number of cycles taken to transfer one block from each set.

It is clear that Cooperative Partitioning is significantly faster to transfer ways than UCP. On average, we take 10m cycles whereas UCP takes 58m. The reason for this is that UCP only transfers blocks on a recipient miss. Since these account for just 33% of all accesses to each way during partitioning (as shown in Figure 14), it follows that Cooperative Partitioning is faster. Further, there are some blocks that are infrequently accessed and these take a large number of cycles to cause a recipient miss. This also accounts for the large transition time in UCP.

5.4 Memory Bandwidth Usage

Our final analysis concerns the amount of memory bandwidth used to flush dirty cache blocks back to main memory during a transition. Figure 16 shows how the average number of flushed blocks varies over time once a partitioning decision has been made. Due to the speed that Cooperative Partitioning transfers ways between cores (Section 5.3), we incur a higher cost initially, with a large number of lines being flushed. However, this quickly drops off 4 million cycles after a partitioning decision is made and stays close to 0 until 10 million when the transfer of the way is complete.

For UCP, there is also a peak in the number of lines flushed during the first period (up to 4 million cycles), although its magnitude is considerably smaller than for Cooperative Partitioning. However, after this point there is a steady use of memory bandwidth, rising to another peak at 7.5 million cycles, then down to nearly 0 just after 10 million cycles. As previously stated, UCP does not complete its transfer until 58 million cycles have passed. Therefore UCP has a more constant memory bandwidth usage, whereas Cooperative Partitioning causes a shorter, larger activity burst.

In fact, what is not shown is that UCP has to flush more lines from LLC to memory during a transition than Cooperative Partitioning. This is because UCP only flushes on a miss by the recipient. Before this happens there can be multiple writes by the donor core, making clean blocks dirty. Although this can also happen in Cooperative Partitioning, it is much less likely, since all accesses by the donor will cause the takeover bit vector to be set and the block transferred to the recipient. On average Cooperative Partitioning flushes 5102 lines, whereas UCP flushes 6536.

5.5 Summary

This section has analysed the results of Cooperative Partitioning. We have shown that ways are quickly transferred between cores after a partitioning decision is made, through cooperative takeover. During this transition period, all accesses by the donor and recipient cores help to migrate ways; donor hits and recipient misses accounting for approximately two-thirds of the events. This enables Cooperative caching to transfer ways five times more quickly than occurs in a comparison approach.

6 Related Work

Existing state-of-the-art cache partitioning techniques can be split into two groups.

Partitioning for Performance Cache partitioning has been widely studied in the past with both static and dynamic schemes proposed. Chiou et al. [6] were the first to introduce column caching and also proposed changes that are required by the replacement policy to be aware of partitioning. Suh et al. [27, 28] used the recency position of hits in cache lines to drive dynamic cache partitioning. However, a global monitoring scheme is used to collect data and hence hit statistics of individual applications get polluted by other co-executing programs. Later Qureshi et al. [20] addressed these shortcomings with utility based cache partitioning (UCP) that uses a low-overhead auxiliary tag directory to monitor each core's cache usage through the LRU stack property [15]. Cache utility curves are generated periodically and partitioning performed accordingly. We use the cache monitoring scheme from UCP and compare against the full technique. However, our method for creating partitions and transferring blocks between cores is different to Qureshi's since we focus on energy efficiency.

Xie et al. [32] and Jaleel et al. [13] modified the shared cache replacement policy to provide performance benefits compared to an unmanaged cache. A two-dimensional cache partitioning was proposed by Chang et al. [5]. This allowed both space and time sharing within the cache, meaning that a few processors share a small cache region for particular time interval while the rest share the remaining large region. However, our work is orthogonal to these approaches, as we partition for energy saving.

The thrasher caging scheme [31] identifies workloads that thrash the cache and isolates them through partitioning. This technique obtains the benefits of partitioning for thrashing applications and an unmanaged cache for non-thrashing workloads, targeting performance. Similarly, Sanchez et al. [24] proposed fine-grained partitioning using an efficient hashing function. The scheme provides data isolation, with a small unpartitioned area that can be used by competing cores to increase their original partitions, rather than taking ways from other cores. Again, Cooperative Partitioning can be used in all partitioned areas, thereby offering energy reduction in addition to performance benefits.

There have been proposals to perform set-wise cache partitioning [22, 30]. However, in a dynamic setting, these schemes would require frequent flushing of data due to the varying memory requirements of different phases of the programs. In comparison, our cooperative takeover does not incur immediate flushing costs and is simple to implement since we enforce way-alignment of data.

Bitirgen et al. [2] applied machine learning to efficiently manage the shared cache and off-chip bandwidth for improved performance. They used dynamic voltage scaling to set the optimal per-core voltage level for various configurations. Our scheme can be incorporated into this to offer energy reduction.

There have been various schemes to offer quality of service by assigning priority levels to threads and partitioning accordingly [3, 9, 11, 12, 17]. Chandra et al [4] studied the impact of inter-thread interference by predicting the number of cache evictions that would be introduced by another thread that runs in a CMP system. However, fully-partitioned caches inherently avoid inter-thread interference. Static schemes have determined the optimal partitions for any combination of applications [26] or have been used to set parameters for various management policies [10]. In summary, Cooperative Partitioning is orthogonal to most prior work and can be applied to these schemes to offer energy reduction on top of performance benefits.

Partitioning for Energy Efficiency In terms of energy efficiency, Reddy et al. [23] statically profiled each application to determine their cache requirements. This information was used to compute cache partitions that can be adapted to sets and associativity. However, as the number of workload combination increases, static profiling becomes more impractical. We have adapted this CPE algorithm for a dynamic setting and compare against it in Section 4.

Albonesi [1] proposed a cache design that can vary its size and associativity by enabling or disabling cache ways. Powell et al. [19] developed a gated-Vdd (non-state preserving) technique to reconfigure the cache and turn off unused cache lines. Meng et al. [16] explored the upper limits of reducing leakage power by combining both drowsy [7] and gated-Vdd techniques. However, this work is only a theoretical upper bound on energy saving since it assumes the existence of an ideal prefetcher, which is impossible to provide in practice. We do, however, implement their gated-Vdd technique to turn off unused ways.

Ghosh et al. [8] proposed way-guarding, a mechanism to reduce dynamic energy by accessing fewer ways. Compared with this our scheme also reduces static energy by turning of unused ways, while using much less hardware.

Finally, Kedzierski et al. [14] proposed a power-aware partitioning using a drowsy cache implementation to reduce both dynamic and static power. In contrast, Cooperative Partitioning is a new technique that provides both dynamic and static energy savings and the drowsy scheme can also be implemented in our cache to offer further energy reductions.

7 Conclusion

This paper has proposed Cooperative Partitioning, a novel partitioning scheme for last-level caches in CMPs.

This approach maintains high performance while saving significant dynamic and static energy. It achieves this by enforcing way-aligned data within the cache, and by cooperation between cores when migrating ways between themselves. Evaluation on a two-core system shows savings of 32% dynamic and 25% static energy compared to a fixed partitioning scheme. In a four-core environment, dynamic and static energy savings of 31% and 20% are achieved, with negligible loss of performance. Further, our scheme migrates ways between cores five times more quickly than a state-of-the-art partitioning approach and requires less data to be flushed back to memory.

The energy savings realised by Cooperative Partitioning create additional headroom in the processor's thermal design power. Thus the negligible reduction in performance can be mitigated through higher clock rates for the same number of cores, or increased numbers of active cores on the chip, which we will investigate in future work.

Acknowledgements This work was supported by the UK's Royal Academy of Engineering and EPSRC. We thank Nikolas Ioannou, Luis Fabricio Wanderley Goes and Andrew J. McPherson for their comments and feedback. The authors are members of HiPEAC.

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *MICRO*, 1999.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[3] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Predictable performance in SMT processors. In *CF*, 2004.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting interthread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[5] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA*, 2006.

[6] D. Chiou, S. Devadas, L. Rudolph, and B. S. Angz. Dynamic cache partitioning via columnization. In *DAC*, 2000.

[7] K. Flautner, N. S. Kim, S. M. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *ISCA*, 2002.

[8] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee. Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches. In *ISLPED*, 2009.

[9] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[10] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *PACT*, 2006.

[11] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *ICS*, 2004.

[12] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policy and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.

[13] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA*, 2010.

[14] K. Kedzierski, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. Power and performance aware reconfigurable cache for CMPs. In *IFMT '10*, 2010.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970.

[16] Y. Meng, T. Sherwood, and R. Kastner. On the limits of leakage power reduction in caches. In *HPCA*, 2005.

[17] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA*, 2007.

[18] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A full system simulator for x86 CPUs. In *DAC*, 2011.

[19] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED*, 2000.

[20] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[21] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT*, 2006.

[22] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA*, 2000.

[23] R. Reddy and P. Petrov. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Transactions on Embedded Computing Systems*, 9, 2010.

[24] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *ISCA*, 2011.

[25] SPEC Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/.

[26] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41, 1992.

[27] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, 2002.

[28] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28, 2004.

[29] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20. *HP Laboratories Palo Alto*, 2008.

[30] K. Varadarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *Micro*, 2006.

[31] Y. Xie and G. Loh. Scalable shared-cache management by containing thrashing workloads. In *HiPEAC*, 2010.

[32] Y. Xie and G. H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*, 2009.