RECAP: Region-Aware Cache Partitioning

Karthik T. Sundararajan
Synopsys Inc
Mountain View, California, USA
tsk@synopsys.com

Timothy M. Jones
Computer Laboratory
University of Cambridge, UK
timothy.jones@cl.cam.ac.uk

Nigel P. Topham
School of Informatics
University of Edinburgh, UK
npt@inf.ed.ac.uk

***Abstract*—In recent years, high performance computing systems have obtained more processing cores and share a last level cache (LLC). However, as their number grows, the core-to-way ratio in the LLC increases, presenting problems to existing cache partitioning techniques which require more ways than cores. Furthermore, effective energy management of the LLC becomes increasingly important due to its size. This paper proposes a Region Aware Cache Partitioning (RECAP), an LLC energy-saving scheme for high-performance, many-core processors. RECAP partitions the data within the cache into shared and private regions. Applications only access the ways containing the data that they require, realising dynamic energy savings. Any ways that are not within the shared or private regions can be turned off to save static energy. We evaluate our scheme using an 8-core CMP running multi-programmed workloads and show that it achieves 17% dynamic and 13% static energy savings in the shared LLC with a 15% performance gain. Across our multi-threaded applications, we achieve 17% dynamic and 41% static energy savings with no impact on performance.**

I. Introduction

In the last ten years, last level cache (LLC) partitioning for performance has received significant interest [21], [22], [25], [26], the idea being to give each core a share of the cache resources according to their needs. These schemes have the potential to realise significant performance increases, yet for the most part they do not consider LLC energy saving. Furthermore, to work effectively they often require the number of cache ways to be more than the number of sharing cores.

This work addresses these issues through a new cache partitioning scheme: RECAP. Instead of partitioning based on the requirements of each core, we partition based on the *ownership of the data* that is being accessed — shared or private. Consider Figure 1 which shows the number of accesses to shared and private data within the LLC for the Parsec benchmark suite, and the amount of shared and private data resident in the cache for the system described in Section III. The overwhelming majority (80%) of accesses request shared data, yet the vast majority of data in the cache is actually private (93%). Other researchers have seen similar statistics [4].

Our Region-Aware Cache Partitioning architecture (RECAP) is shown in Figure 2. We identify and separate private and shared data, justifying the former to the left-hand side of the cache and the latter to the right. When an access requires private data, only the ways on the left of the cache need to be activated, and vice-versa for shared requests. This way alignment of data allows us to realise significant dynamic energy savings and also allows us to support a core to cache way ratio of 1:1. Furthermore, we monitor the cache usage of each core in the system and dynamically restrict

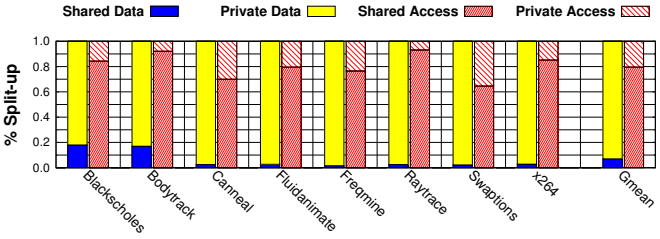


Fig. 1: Data types and accesses to a shared last level cache.

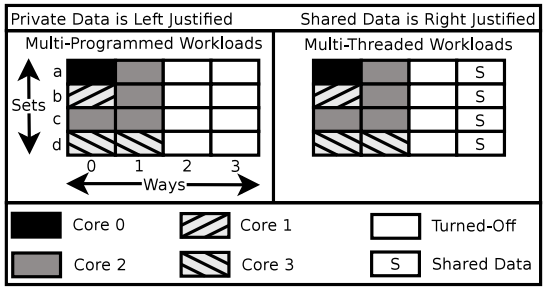


Fig. 2: Example data layouts in RECAP.

the size of the private region for each one. This means that cores executing workloads with high LLC miss rates can be effectively contained, reducing their interference with other cores. Ways in the centre of the cache then become unused and can be turned off for static energy savings. To our knowledge, we are the first to propose separate partitioned cache regions for private and shared data.

Using our technique in an 8-core system with an 8-way cache brings dynamic energy savings of 17% and static energy savings of 13% with no performance loss across a range of multi-programmed workloads. For multi-threaded applications, RECAP achieves 17% dynamic and 41% static energy savings at the same level of performance. Furthermore, we introduce a simple scheme whereby all cores help in flushing dirty data back to main memory when a core no longer requires a cache way, and show that this is 85% faster and consumes 98% less energy than a basic technique that relies on one core alone.

The rest of this paper is structured as follows. Section II presents RECAP, then Section III describes the experimental methodology used in this paper. Section IV evaluates our approach on an 8-core systems and analyses our results. Section V describes the related work then, finally, Section VI presents our conclusions.

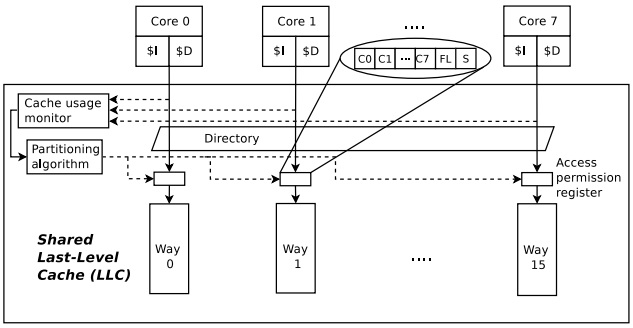


Fig. 3: An overview of the RECAP architecture. Cache use by each core is monitored and periodic partitioning decisions are made. The ability of each core to read or write into each way is granted through access permission registers (APRs).

Algorithm 1: Determining cache requirements.

T_{Ways} = N; /* Total number of cache ways in LLC*/
blocks_req[c] = 0; /* For each competing core, c */
foreach *core* c **do**
    max_mu[c] = get_max_mu(c, T_{Ways});
    blocks_req[c] = min blocks to get max_mu[c] for core c;
end
return blocks_req;

get_max_mu(c, T_{Ways}):
max_mu = 0;
for $j = 1$; $j <= T_{Ways}$; $j++$ **do**
    U = change in misses for core c when moving from 0 to j ways;
    mu = U/j;
    if $mu > max\_mu$ **then**
        max_mu = mu;
    end
end
return max_mu;

II. The RECAP Architecture

Figure 3 gives an overview of RECAP. All accesses from the higher-level caches pass through the directory on their way to the LLC. Over an interval containing a fixed number of instructions, these accesses are monitored by the cache usage logic and are then used to make partitioning decisions. These are written into the access permission registers (APRs) that sit before the cache ways, to control which cores have access to each way. Data that is private to a particular thread is kept left-justified in the cache (i.e., in the low-numbered ways) whereas shared data is kept right-justified.

Our cache architecture is orthogonal to existing cache partitioning schemes [17], [23], [25] because these techniques consider cache partitioning only when the number of ways is greater than the number of cores. In RECAP, we consider cache partitioning whatever the core-to-way ratio is. Furthermore, our approach can be applied to any LLC configuration, from banked NUCAs to multiple, independent LLCs.

A. Usage Monitoring

Our cache architecture builds on prior work to determine the optimal number of ways for each core. We use utility monitors [17] to track the accesses by each core to characterise each thread's use of the cache. These monitors have been used by other partitioning techniques [23], [25]. However, the operating system could also provide this information [18].

To determine partitions we use Algorithm 1. In our scenario we are not concerned with allocating ways to each core individually, so we only need to find the number of ways for each core to achieve its highest utilisation (max_mu). We assume all threads have equal priority and therefore give each core a number of ways based on the performance benefits it can realise from them. If threads were to have differing priorities we could incorporate this information into Algorithm 1 by increasing the number of ways that high-priority threads receive and decreasing the number allocated to low-priority threads.

B. Cache Partitioning Control

To control each core's accesses to the LLC, and to enforce way-aligned data, we introduce an access permission register (APR) to each way. The layout of these registers is shown in Figure 3. Each APR has one bit per core to indicate whether a core has access permission to the associated way. There is an additional shared bit to indicate a way shared by all cores and a further flush bit which is used when contracting a core's cache allocation (explained in Section II-C).

For a given cache way, when a core's bit is set in the corresponding APR, it has permission to both read and write in that way. When it is unset, then the core cannot access that way at all. However, during contraction, the flush bit is set and this allows all threads to read from that way, regardless of their core bit in the APR. This leads to three possible modes of operation, with read access to a way controlled by *APR[id]* || *APR[fl]*, and write permission by *APR[id]*, where *id* is the core's id and *fl* is the flush bit.

Since the number of cache ways may be equal to or less than the number of cores, more than one core will have its APR bit set for certain ways. Thus RECAP not provide data isolation, in contrast to other schemes [23], utilising the cache more effectively.

The APR register enforces the cache partitioning by only allowing cores to access certain ways. At the same time, this enables dynamic energy savings because cores only need to access the ways where they have permission. If the APR for a way is totally clear (i.e., all core, shared and flush bits unset), then the way can be turned off, realising static energy savings.

C. Cache Reconfiguration

Initially all cores share one cache way (way 0) so all core bits in the APR for this way are set, enabling full access to all threads. When a core requires more cache ways than it currently has, we enter an expansion mode for that thread. On the other hand, when a core needs fewer ways than it currently has access to, we enter a contraction mode.

a) Expansion Mode: Expanding the number of ways that a core can access is simply a matter of setting the appropriate bit in the APR for each way that the core requires. If any of the ways were previously turned off, then they are enabled once more.

b) Contraction Mode: When a core requires fewer ways than it currently has, we must flush dirty data back to main

memory before resetting the core's APR bit for the ways it no longer needs. Therefore, we only prevent write access to those ways and flush dirty data back to main memory as each set is accessed. However, to avoid excessive data being written back to DRAM, we only flush a line if it is not in one of the L1 caches. To keep track of the sets that need to be flushed (if they contain dirty data) we provide a bit vector for each way that is initially clear, with each bit being set according to the sets that are accessed. Once this contraction bit vector is totally set, no more dirty data from the contracting core exists in that way and all permissions can be removed.

One downside with this approach is that it can take many cycles for all sets to be accessed, which decreases the effectiveness of our partitioning scheme and can lead to fewer energy savings (e.g., because it takes longer for unneeded ways to be turned off). To aid the flushing process, we provide an extra bit in the APR, called the *flush bit*. When this is set, all cores access the corresponding way to flush data back to main memory from the set they are consulting, and set the relevant bit in the contraction bit vector. So instead of one core being responsible for all the flushing, all cores contribute to it, significantly speeding up the contraction process. Whenever a new reconfiguration occurs during the takeover process, we reset the bit vector and start the whole process again.

Within RECAP there are no restrictions on when cores can expand and contract at the end of each interval. Both contraction by one set of cores and expansion by another set can happen in parallel with no impact on correctness.

D. Separating Private and Shared Data

One key contribution of our work, as highlighted in Section I, is the ability to realise significant energy savings simply based on the ownership of the data being accessed — private or shared. As such, we must be able to identify these different data ownerships and deal with the situations where they change from one category to the other.

All accesses to the LLC first pass through the directory. We use the directory information to determine whether the required data is private or shared. Data that is in exclusive or modified states is considered private whereas data that is in a shared state is shared.

When a core accesses the LLC, it only searches in the ways that contain the ownership of data that it wants. I.e., when looking for private data it only searches the private ways that it has access to (with its core bit set in the APR). When searching for shared data it only looks in the shared ways (those that have the shared bit set in the APR). When a directory does not have an entry for a particular data item, the core searches its ways in both the private and shared regions.

RECAP uses the default LRU replacement policy to evict data on a miss, requiring only minor changes to the baseline LRU scheme. Upon a core's access, the cache ways that have that core's bit set in their APR are enabled and the LRU policy is applied to the enabled ways to find the victim cache block.

E. Data Category Transitioning

There are two situations where we have to alter the category of the data in the LLC. First, if the data is in the shared region but a core wants to write to it, we simply invalidate it in the LLC as usual. If the core then evicts that data from its L1 cache (e.g., through a capacity miss), then the data will be written into the private region of the LLC. If, on the other hand, another core requests read access to that data, it will be written back into the LLC's shared side. In this scenario there are no reads nor writes into main memory.

Second, if the data is in the private region, but another core wants to read it, then the data must be placed into the shared region. If the data is in modified state then we invalidate the line in the LLC, obtain the most up-to-date copy from the L1 that owns it and place that in the shared region. If the data is in exclusive state then we first send the block back the requesting core, then invalidate the existing line (flushing dirty data, if necessary) and refetch from main memory, writing into the shared region. This incurs an overhead, but is more simple than implementing logic to move the line between two regions. Therefore we only involve DRAM when moving from exclusive to shared state. In practice, this happens infrequently and thus overheads are small.

F. Private Data Expansion

One downside to our approach so far is that cache requirements are estimated on a per-core basis, but private data partitions are overlapped in the LLC, which could cause contention and limit our benefits. To address this we perform selective expansion of private data regions after each reconfiguration of the ways containing private data. When a partitioning decision is made, the private data partitions for each core are set. After one interval, each partition is expanded by a single way, if one is available, and another interval is executed. If the larger partition size achieves 10% more performance for its core then we expand again. If not, then we roll back by contracting out of the way. This is repeated until there are no more ways to expand into, or a larger partition does not give at least 10% performance increase. This expansion of private data partitions allows us to overlap the partitions without unnecessarily impacting performance.

G. Overheads of RECAP

We use the same hardware as Qureshi and Patt [17] to monitor cache usage (including dynamic set sampling), as is required by other partitioning schemes [23], [25]. We also require extra bits for the APRs (1 bit per core, flush and shared bits) and contraction bit vectors (1 vector per core each containing 1 bit per set). For an 8-core system with an 8MB, 8-way cache, this comes to a little over 128K bits.

The RECAP cache consumes more power than a standard last level cache, because it has the extra circuitry for monitoring and partitioning. All power overheads are included in our simulated results in Section IV. However, the gain of our approach are enough to prevail over these overheads.

In terms of performance, there is an extra latency for cache accesses due to serialising the directory lookup. Finally, we only flush dirty data back to DRAM during contraction and when it is not present in one of the L1 caches. These have negligible impact and are all modelled in our evaluation.

Parameters	Configuration
Processor	8-Cores, 1 thread/core, in-order
Operating System	Linux-2.6.28.smp
L1 ICache	32kB, 64B lines, 4-way, 3 cycle lat.
L1 DCache	32kB, 64B lines, 4-way, 3 cycle lat.
Shared L2	8MB, 64B lines, 8-way, 15 cycle lat.
Coherence	MOESI CMP directory

TABLE I: System configuration.

Group	Benchmarks
1T-1	Mcf, Astar, Calc., Povray, Sjeng, Xalan, DealII, H264.
1T-2	Art, Bzip2, Gobmk, Namd, Omn., Perl., Gromacs, Xalan
1T-3	Equake, Omn., Astar, Gromacs, Gobmk, H264., Sjeng, Bzip2
2T-1	Sphinx3, Art, Gcc, Bzip2, DealII, Gobmk, Xalan, Povray
2T-2	Soplex, Lbm, Astar, Gobmk, H264., Namd, Sjeng, Perl.
2T-3	Mcf, Milc, Omn., Gcc, Xalan, Sjeng, Calc., Gromacs
3T-1	Mcf, Lbm, Milc, Bzip2, Namd, Povray, Sjeng, H264.
3T-2	Equake, Sphinx3, Libq., Omn., Sjeng, Gromacs, Astar, Gobmk
3T-3	Lbm, Libq, Soplex, Astar, Calc., Bzip2, Sjeng, H264.
4T-1	Equake, Sphinx3, Soplex, Lbm, Calc., Gromacs, Gcc, Gobmk
4T-2	Mcf, Milc, Libq., Art, Astar, Bzip2, Perl., Xalan
4T-3	Art, Equake, Sphinx3, Libq., Gcc, Omn., Namd, Povray
5T-1	Mcf, Lbm, Milc, Soplex, Libq., Gcc, Sjeng, Xalan
5T-2	Equake, Mcf, Lbm, Sphinx3, Art, Astar, Povray, Calc.
5T-3	Milc, Equake, Sphinx3, Libq., Lbm, Omn., DealII, Namd
6T-1	Mcf, Equake, Lbm, Libq., Art, Milc, Gcc, Sjeng
6T-2	Equake, Sphinx3, Mcf, Art, Soplex, Lbm, Astar, Namd
6T-3	Art, Milc, Lbm, Libq., Soplex, Equake, Gromacs, H264.
7T-1	Mcf, Lbm, Milc, Soplex, Libq., Art, Sphinx3, Povray
7T-2	Equake, Sphinx3, Art, Libq., Lbm, Milc, Soplex, Bzip2
7T-3	Art, Equake, Mcf, Soplex, Sphinx3, Lbm, Libq., Gcc
Parsec	Blackscholes, Bodytrack, Canneal, Fluidanimate, Freqmine
	Raytrace, Swaptions, X264

TABLE II: SPEC2006 combinations using reference inputs and Parsec workloads, all using sim-large inputs.

III. Experimental Methodology

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

A. Simulator

We implemented our partitioned cache architecture in gem5 [3]. Table I shows the configuration of the system. We simulated an x86-based in-order processor with 8 cores running an SMP-enabled Linux kernel 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. Our cache configurations are similar to those used by others [13], [15]. To obtain energy information we used Cacti [24] at 45nm. Finally, as in prior work [17], we assumed a 5 million cycle phase interval for monitoring and partitioning decisions.

B. Workloads

We ran Parsec benchmarks [2] for our multi-threaded applications and a random mix of benchmarks from SPEC CPU2006 [20] for our multi-programmed workloads. Tables II show the applications. To choose our SPEC, mixes we first grouped benchmarks based on their misses per kilo instructions (MPKI) values, denoting those with a value greater than 5 as *thrashing* applications. We then randomly selected mixes of 8 benchmarks containing 1 through 7 thrashing applications. In Table II, 3T-1 refers to the first group containing 3 thrashing applications. Programs are ordered so that the thrashing applications occur first.

We used the *sim-large* inputs for all Parsec applications, simulating the parallel region in full. For the SPEC workloads we used the *reference* inputs. We first fast-forwarded each application by 20 billion instructions, warmed-up the caches and branch predictor for 500 million instructions and then simulated for at least 1 billion further instructions, per application, as is common practice [6], [25]. Statistics are reported for 1 billion instructions per benchmark, but all applications continued running until the last program in the mix had reached this instruction count, to keep contending for cache resources.

C. Comparison Approaches

One significant problem with implementing comparison approaches is that existing schemes for cache partitioning only work when there are more cache ways than there are cores sharing that cache. RECAP is flexible and scalable, so is independent of the core to cache way ratio. Therefore, we have implemented two comparison techniques. The first scheme, *Thrasher Caging (TC)*, is a policy for containing thrashing workloads [26]. The second, *TA-DRRIP*, is a state-of-the-art cache replacement policy targeting high-performance [9], but only works well when there are more ways than cores. In our experiments, we used 32 set dueling monitors [16] with $\epsilon = 1/32$.

IV. Evaluation

We evaluate RECAP by showing its performance when the number of ways is the same as the number of cores. Other cache partitioning schemes cannot cope with this scenario, since they assume that there are more ways than cores within the LLC. We are able to implement this in RECAP because we do not enforce a fixed and separate partition to each core, but a porous partitioning technique that allows the entire region space to be used by all threads that share the partition. RECAP works on a serially-accessed LLC, as is the norm, so dynamic energy savings come from the tags only.

A. Performance and Energy

Figures 4-5 show that RECAP achieves higher performance than an LRU scheme (15% better for the SPEC application mixes using an 8-way cache) while realising large reductions in energy. The dynamic energy consumption of RECAP is significantly less than the baseline for the SPEC application mixes (83% of the dynamic and 87% of the static energy consumption, on average for an 8-way cache). The results for the multi-threaded workloads are similar, with an average consumption of 83% dynamic and 59% static energy. These show that RECAP is able to realise significant energy reduction without causing slowdown. Comparing RECAP to TA-DRRIP, we can see that RECAP achieves significant dynamic and static energy savings that cannot be realised by TA-DRRIP, since it only targets performance.

However, groups *2T-3*, *4T-3* and *6T-3* disappointingly achieve no static energy savings with an 8-way cache. All mixes contain an application that benefits from a large number

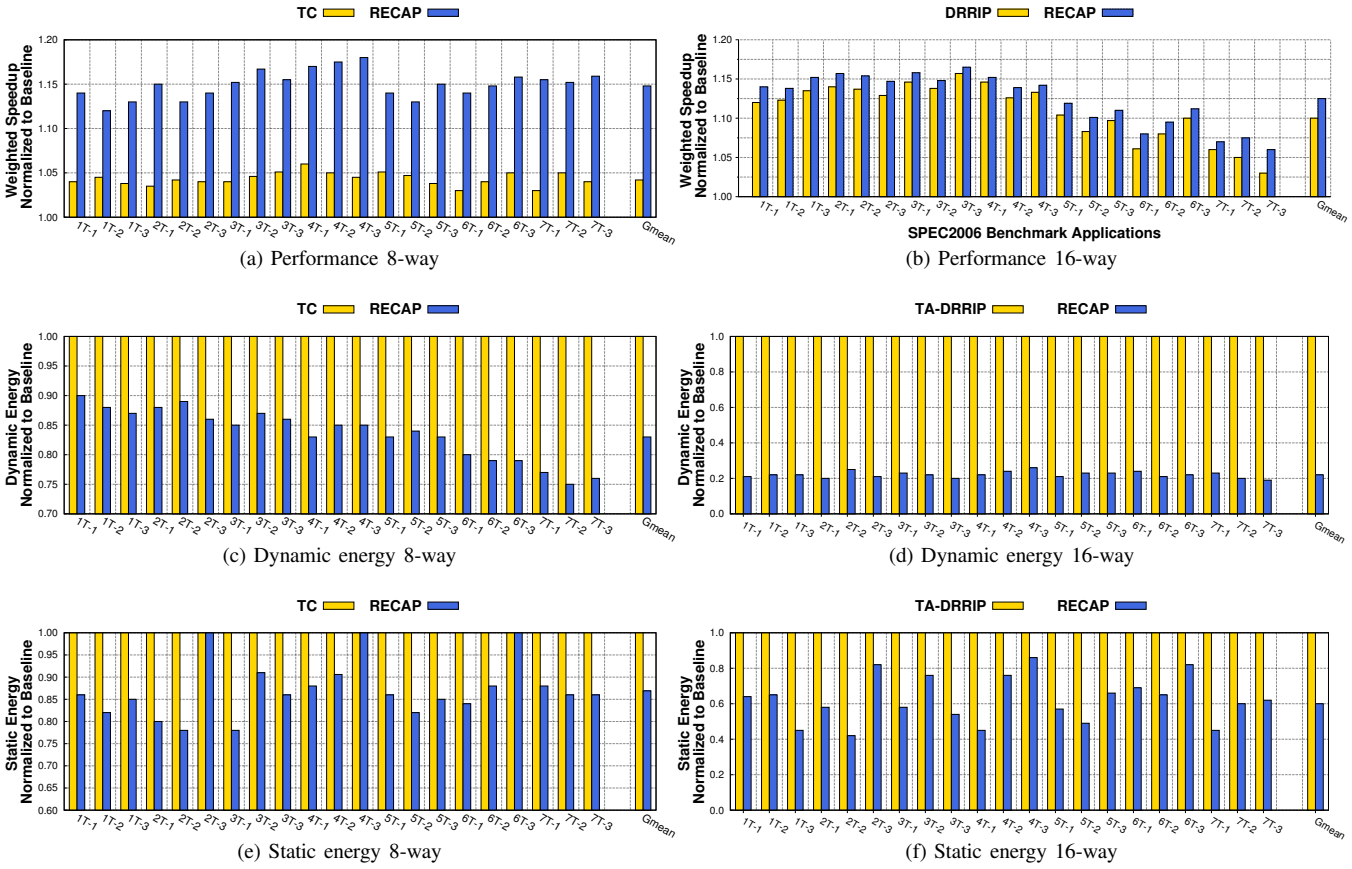


Fig. 4: Performance and energy consumption of SPEC 2006 groups on an 8-core system.

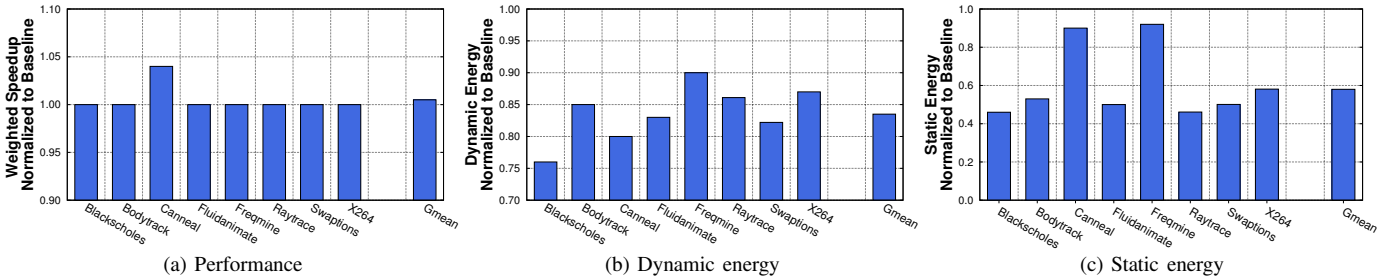
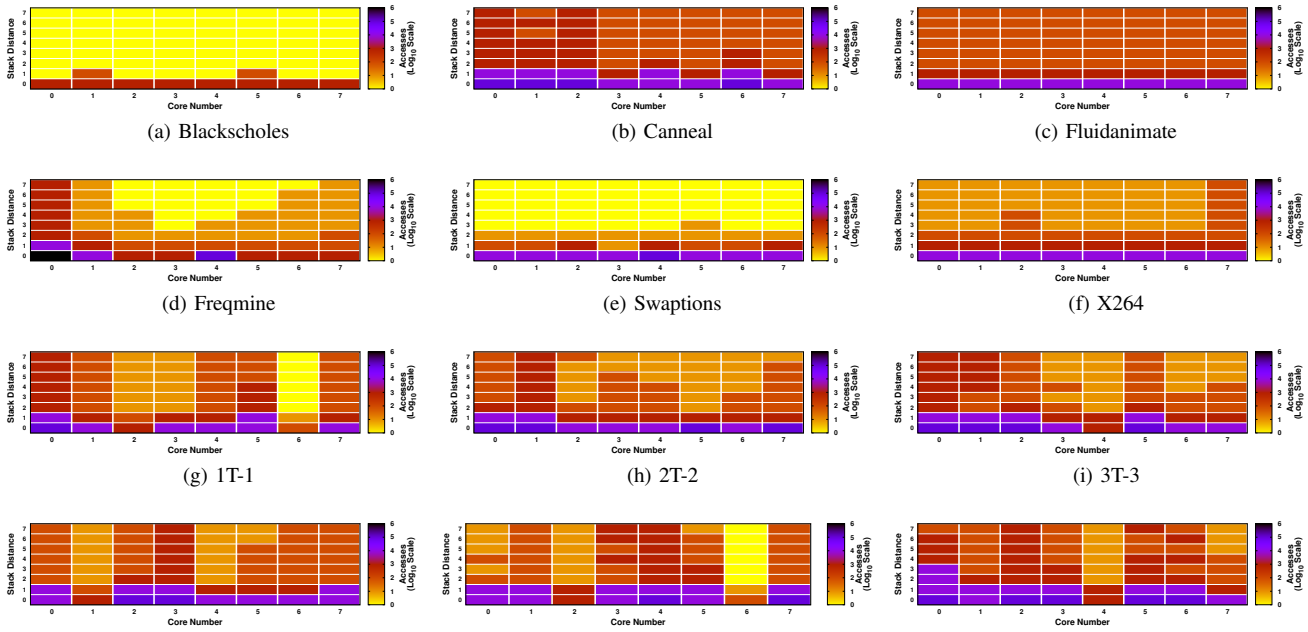


Fig. 5: Performance and energy consumption of Parsec workloads on an 8-core system with an 8-way cache.

of cache ways. In *2T-3* it is *Xalan*, which requires 7 ways, in *4T-3* it is *Povary*, which requires 6, and in *6T-3* it is *H264* and *Gromacs*, which both require 4 ways. Due to contention in the left-hand side of the cache (i.e., where the private data is justified, since there is no shared data across these applications), these workloads expand to the full 8 ways, as described in Section II-F, preventing any ways from being turned off. This effect can be seen, to an extent, in other mixes containing *Xalan* (*1T-1*, *4T-2*, *5T-1*), *Povray* (*7T-1*), *Gromacs* (*3T-2*, *4T-1*) and *H264* (*3T-3*). With a 16-way cache the effect is less pronounced, resulting in fewer static energy savings than other groups achieve. However, there are still significant energy savings achieved across all groups of benchmarks, showing the ability of RECAP to realise energy savings even in the presence of workloads with large cache requirements.

In the 8-way scenario, Thrasher Caging is unable to realise the significant performance increases achieved by RECAP. This is because it only partitions the cache into two regions — one for the thrashing threads and another for the rest. The non-thrashing threads could still benefit from a limit on the cache resources they are allowed, but this does not happen. Instead there is contention and additional performance benefits are not realised. RECAP does not suffer from this because it considers each thread separately and allows different way restrictions for each one. This then allows it to achieve the speedups and energy savings shown.



(a) Blackscholes (b) Canneal (c) Fluidanimate

(d) Freqmine (e) Swaptions (f) X264

(g) 1T-1 (h) 2T-2 (i) 3T-3

(j) 4T-1 (k) 5T-3 (l) 7T-1

Fig. 6: Heat maps showing frequency of ways accessed by each core for representative workloads for an 8-way cache.

B. Analysis

Figure 6 analyses these results by showing heat maps corresponding to the stack distance [12] for each core. The darker the colour, the greater the number of accesses by that core to that distance. In the SPEC application mix, programs are assigned to ways in the same order as in Table II.

For *Blackscholes* it is easy to see that the majority of the accesses are concentrated in the MRU or second position. This means that *Blackscholes* can survive with only 2 last-level cache ways, and explains the energy savings shown in Figure 5. Turning to *Canneal* and *Freqmine*, it is clear to see that threads 0 - 2 require the most ways for the former, and thread 0 for the latter, which leads to fewer static energy savings.

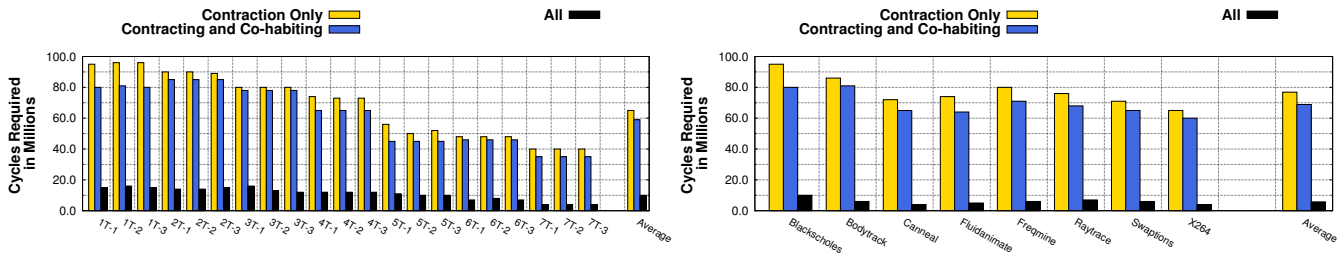
For some benchmarks (e.g., *Blackscholes*, *Fluidanimate*, and *Swaptions*), the access patterns are similar across all cores. However, for other applications (e.g., *Canneal* and *X264*), there are two distinct groups of threads that have similar access patterns within each group but entirely different patterns across groups. RECAP can achieve large dynamic and static energy savings when all cores have similar cache requirements with a small stack distance. However, when the cores have varying cache access patterns, RECAP can realise large dynamic energy savings (by restricting the cores with small stack distance requirements), but is limited in the static energy savings it can achieve (because the cores with large stack distance patterns occupy more ways). In general, the dynamic energy savings achieved by RECAP are influenced by the number of ways allocated to the cores that make the most accesses. Static energy savings, however, are influenced by the cores that require the most amount of cache space.

In the heatmaps for the SPEC application mixes the colours are darker across the stack distances, meaning that, in general, each core requires more ways to keep high performance. This explains the generally lower energy savings seen in Figure 4c and Figure 4d for the SPEC application mixes compared to Parsec, although since the majority of accesses are made to stack distances 0 and 1, RECAP can still realise significant savings. One application, in particular, makes few accesses to the last-level cache and requires a small stack distance when it does so. This is *DealII*, which is scheduled on core 6 in *1T-1* and core 6 in *5T-3* and is a significant out lier in this analysis of this set of benchmarks.

Considering the 8-way static energy savings, from Figure 4 again, we can see reasons for the large savings achieved by *2T-2* and *7T-1*. In *2T-2*, the six non-thrashing applications (towards the right-hand side of the heatmap) mostly have low stack distance requirements. Hence there is an opportunity to turn off ways. For *7T-1*, the single non-thrashing program on core 7 also mainly accesses stack distances up to five ways, which again allows RECAP to turn parts of the cache off.

C. Cycles Required for Contraction

Section II-C described how a core contracts out of ways that it no longer has access to, through the use of the flush bit within the APR. We now evaluate the effectiveness of this bit by comparing three different approaches to contraction. Figure 7 shows the results using an 8-way cache. In the first scheme, labelled *Contracting Only*, only the contracting core flushes dirty data back to main memory when it leaves a way. The second approach, *Contracting & Cohabiting*, corresponds to the case where data is flushed by the contracting core and any other cores that already have access to that way. Finally, the scheme labelled *All* is the technique used in RECAP where all cores participate in flushing dirty data back into main memory. Contraction is only completed when the group of cores involved in flushing have accessed all sets between them.



(a) SPEC2006

(b) Parsec

Fig. 7: Number of cycles required to contract out of a way for different flushing schemes.

It is clear from Figure 7 that allowing all cores to flush dirty data provides a significant reduction in the time taken to contract out of a way. On average for the SPEC application mixes, only 10m cycles are required, compared with 65m for the first scheme and 59m for the second, meaning that it is 85% faster than simply allowing the contracting core to flush data. For Parsec, the corresponding values are 13m, 77m and 69m. In addition, further experiments show that this is also more energy efficient, requiring only 2% of the dynamic energy required when the contracting core or both contracting and co-habiting cores perform the flushing. Although the dynamic energy consumption is larger initially (because additional cores have access to the contracting way), it is for such a small amount of time compared with the two other schemes that these overheads are subsumed by the benefits of fast contraction. This clearly shows that simply adding a flush bit to the APR is effective when contracting out of a way.

D. Data Category Transitioning

Figure 8 shows the number of data blocks that transition category from private to shared or vice-versa. We show only Parsec benchmarks because there are no transitions with the SPEC application mixes. As described in Section II-D, data transitions from shared to private do not affect performance but transitions from private to shared require us to invalidate the line in the private region in the last level cache, then refetch into the shared region from main memory.

The two benchmarks with the smallest number of transitions from private to shared are *Blackscholes* and *Raytrace*, with 1m and 1.5m transitions respectively. All other benchmarks have 10m transitions, or more (up to 30m for *Freqmine*). Blackscholes, in particular, is known to communicate rarely with other threads in the application, leading to a low number of transitions [1].

On average there are 8m transitions from private to shared. However, this is small compared to the overall number of LLC accesses (which is 170m per core on average), so the additional power overheads from refetching data are easily subsumed by our large energy savings.

E. Summary

RECAP achieves 17% dynamic and 13% static energy savings with no slowdown across twenty-one mixes of SPEC CPU2006 applications on an 8-core system with an 8-way LLC. Across Parsec benchmarks it averages 17% dynamic and 41% static energy savings, again with no slowdown. In addition, using a flush bit in each APR provides fast and efficient contraction out of ways when a core's cache requirements shrinks. When all cores contribute to the flushing, contraction can be realised 85% faster than when the contracting core is the only core to flush dirty data back to memory.

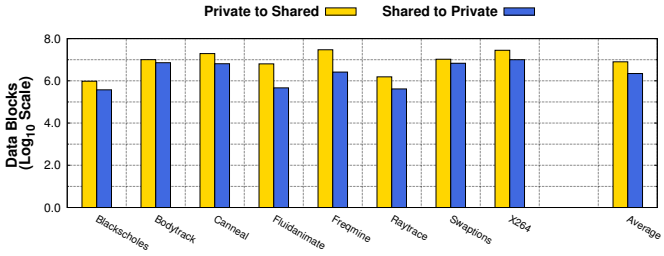


Fig. 8: Number of data blocks transitioning from private to shared and from shared to private.

V. Related Work

There is a rich body of work in the literature concerning cache partitioning, especially in the recent past. Qureshi and Patt [17] proposed utility-based cache partitioning (UCP) that monitors each core's LRU stack property [12] as a proxy for cache usage by using a low-overhead auxiliary tag directory. Partitioning is performed periodically based on the cache utility curves that are generated from these monitors. We use this cache monitoring scheme in RECAP to track accesses by each core. However, the UCP partitioning algorithm does not work when the core count is greater than or equal to the number of ways in the LLC. In contrast, our method works effectively making our approach scalable to many cores.

Xie and Loh [25] and Jaleel et al. [10] obtain performance benefits by modifying the replacement policy. DRRIP uses set dueling monitors to keep track of competing static replacement policies and regularly selects the one with the lowest number of misses. These schemes target performance, whereas our work is orthogonal to these approaches, as we partition for energy saving as well. In a similar vein, Jaleel et al. [8] proposed a scheduling aware cache replacement algorithm. This scheme requires two or more LLCs and more cache ways than cores in the system, whereas our work can be used whatever the memory hierarchy configuration.

Applications that thrash the cache can be detrimental to

the performance of other workloads and the thrasher caging scheme [26] identifies and isolates these programs through partitioning. This allows non-thrashing, co-habiting workloads to obtain the benefits of an unmanaged cache, while not limiting the performance of the thrashing applications. Fine-grained partitioning using an efficient hashing function has been proposed by Sanchez and Kozyrakis [19]. Rather than taking ways from other cores, this scheme provides data isolation through the use of a small unpartitioned area for competing cores to increase their original partitions. Again, this works well in highly-associative caches, whereas RECAP can be used whatever the core-to-way ratio.

Muralidhara et al. [14] proposed cache partitioning for parallel applications targeting performance by monitoring each thread's cache requirements and allocating ways based on this. However this does not distinguish between private and shared data, missing significant opportunities for energy saving.

The R-NUCA [7] scheme identifies private and shared data at a page level. This enables data to be moved close to the cores that request it most often, increasing performance from the non-uniform cache architecture. Cuesta et al. [4] also identified shared and private data at a page level, bypassing the directory lookup for private data. In comparison, our scheme identifies private and shared data at a cache line granularity and uses this information to realise energy savings. RECAP is orthogonal to both schemes and can be implemented together with either.

Sundararajan et al. [23] proposed Cooperative Partitioning, which uses way-alignment to restrict cores to a subset of all cache ways for energy efficiency. However, their approach allows only one core to own each way, and there must be more ways than cores, whereas RECAP does not have these restrictions. Further, this only targets multi-programmed workloads, whereas our approach handles any application mix.

Way-guarding [5] has also been proposed as a mechanism to reduce dynamic energy by accessing fewer ways, especially when there are more cache ways than cores. In contrast, our approach can realise both static and dynamic energy savings, while using significantly less hardware. Finally, dynamic and static power can be saved by a power-aware partitioning algorithm using a drowsy cache implementation [11]. Drowsy caches can also be implemented alongside RECAP to provide further energy reduction.

VI. Conclusion

We have proposed RECAP, a region-aware cache partitioning scheme for both multi-programmed and multi-threaded applications running within a shared last-level CMP cache. Our approach maintains high performance while saving significant dynamic and static energy. This is achieved by partitioning the cached data into private and shared regions, and only allowing cores to access the ways containing the data that they seek. Unused ways can be turned off for static energy savings. We have evaluated RECAP on an 8-core system with an 8-way LLC, showing that it achieves 17% dynamic and 13% static energy savings with no performance loss, showing how LLC partitioning can still occur when the way-to-core ratio is low.

Acknowledgements

This work was supported by the UK's Royal Academy of Engineering and EPSRC. It has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF — http://www.ecdf.ed.ac.uk/). The ECDF is partially supported by the eDIKT initiative (http://www.edikt.org.uk). The authors are members of HiPEAC.

References

[1] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of Splash-2 and Parsec," in *IISWC*, 2009.

[2] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[3] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, May 2011.

[4] B. A. Cuesta *et al.*, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *ISCA*, 2011.

[5] M. Ghosh *et al.*, "Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches," in *ISLPED*, 2009.

[6] F. Guo *et al.*, "A framework for providing quality of service in chip multi-processors," in *MICRO*, 2007.

[7] N. Hardavellas *et al.*, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.

[8] A. Jaleel *et al.*, "Cruise: Cache replacement and utility-aware scheduling," in *ASPLOS*, 2012.

[9] A. Jaleel *et al.*, "Adaptive insertion policies for managing shared caches," in *PACT*, 2008.

[10] A. Jaleel *et al.*, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ISCA*, 2010.

[11] K. Kedzierski *et al.*, "Power and performance aware reconfigurable cache for CMPs," in *IFMT '10*, 2010.

[12] R. L. Mattson *et al.*, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, 1970.

[13] J. Merino, V. Puente, and J. Gregorio, "Esp-nuca: A low-cost adaptive non-uniform cache architecture," in *HPCA*, 2010.

[14] S. Muralidhara, M. Kandemir, and P. Raghavan, "Intra-application cache partitioning," in *IPDPS*, 2010.

[15] S. H. Pugsley *et al.*, "Swel: hardware cache coherence protocols to map shared data onto shared caches," in *PACT*, 2010.

[16] M. K. Qureshi *et al.*, "Adaptive insertion policies for high performance caching," in *ISCA*, 2007.

[17] 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.

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

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

[20] SPEC Corporation, "SPEC CPU2006," http://www.spec.org/cpu2006/.

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

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

[23] K. T. Sundararajan *et al.*, "Cooperative partitioning: Energy-efficient cache partitioning for high-performance cmps," in *HPCA*, 2012.

[24] S. Thoziyoor *et al.*, "Cacti 5.1. Technical Report HPL-2008-20," *HP Laboratories Palo Alto*, 2008.

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

[26] Y. Xie and G. H. Loh, "Scalable shared-cache management by containing thrashing workloads," in *HiPEAC*, 2010.