Power and Performance Aware Reconfigurable Cache for CMPs

Kamil Kędzierski
Technical University of Catalonia and Barcelona Supercomputing Center
Barcelona, Spain
kkedzier@ac.upc.edu

Francisco J. Cazorla
Spanish National Research Council and Barcelona Supercomputing Center
Barcelona, Spain
francisco.cazorla@bsc.es

Roberto Gioiosa
Barcelona Supercomputing Center
Barcelona, Spain
roberto.gioiosa@bsc.es

Alper Buyuktosunoglu
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
alperb@us.ibm.com

Mateo Valero
Technical University of Catalonia and Barcelona Supercomputing Center
Barcelona, Spain
mateo@ac.upc.edu

ABSTRACT

This paper investigates the problem of partitioning a shared cache among threads executing in a Chip Multi-Processor (CMP). We propose *Reconfigurable Cache for CMPs* (ReCaC), a low-overhead run-time mechanism that dynamically partitions the cache based on the phase behavior of threads. Unlike the previously proposed performance aware partitioning approaches, ReCaC targets to use a minimum number of ways to trade-off between power and performance. ReCaC dynamically reverts back to a performance centric cache partitioning scheme if the power savings are not achievable.

Our results show that in a 2-core architecture ReCaC saves on average 51.5% power in a L2 cache, which corresponds to 11.5% power savings of the processor chip and the memory. The overall processor energy efficiency is improved by up to 13.6%, achieving on average 8.5%. ReCaC proves to be scalable, saving on average 10.8% and 12.5% of the processor chip and memory power in 4- and 8-core architectures, respectively.

1. INTRODUCTION

The high cost of extracting instruction level parallelism (ILP) from single thread leads to a thread-level parallelism (TLP) as an effective strategy to improve processor performance. One of the most common TLP paradigms are chip multiprocessors (CMP). In most CMPs the last level cache (LLC) is shared between threads, as this approach guarantees more flexible and dynamic allocation policies. The growth in the number of cores increases the activity in shared resources, which not only affects the performance, but also increases the final power and design complexity. As such, the complexity and power dissipation of the shared resources continue to be key performance limiting factor. The LLC, the L2 cache in our baseline processor setup, has been identified as one of the major sources of contention between threads. As a consequence, researchers have proposed several *partitioning* algorithms, both at hardware [22, 26] and software [10, 20] level, to improve a target metric like total throughput, or per-thread performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IFMT '10, 19-JUN-2010, Saint-Malo, France
Copyright 2010 ACM 978-1-4503-0008-7/10/06 ...$10.00.

Previous cache partitioning algorithms gather runtime profile information through *Auxiliary Tag Directories* (ATD) [27], a separate tag directories with the same associativity, size, and replacement policy as the L2 cache tag directory. The goal of the ATD is to track the L2 cache behavior as if each thread was running in isolation. To do so, each thread accesses solely one separate ATD structure. During the execution, the ATD counts the number of misses and observes positions of hits in LRU stack. This allows to estimate the relationship between the number of cache ways assigned to a thread and its performance [27]. To reduce the high hardware cost of the tag directory replication, Qureshi et al. [27] proposed a *sampled* ATD (sATD) that stores only a subset of the sets used in the ATD. However, this solution still requires thousands of bytes per each core to precisely profile running threads.

Recently researches attempted to eliminate the need of external, area costly profiling structures. Qureshi et al. [24, 25] and Jaleel et al. [18] used a Set Dueling (SD) mechanism, with different cache replacement schemes applied to some sets. The replacement policy with the best results in those sets determines the winning policy for the remaining sets. This solution needs only several bytes per core [18]. Unfortunately, SD scheme does not provide full profile information that estimates how each cache way assigned to a given thread influences the performance. Moreover, this scheme requires modification of the L2 cache internal design, as some sets have implemented different replacement policies. SD also faces the scalability problem, when the number of the replacement policies to be checked increases, as in case of some cache partitioning algorithms number of possible replacement scenarios can exceed number of sets [27].

We propose an *embedded Auxiliary Tag Directories* (eATD), a complexity-effective hardware monitoring logic for shared caches in multi-core architectures. eATD is a flexible framework controlled by the Operating System (OS). The OS specifies which part

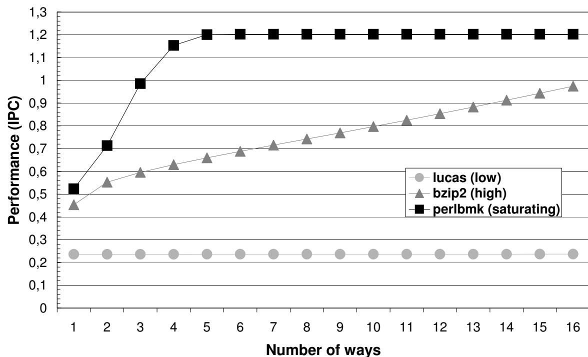


Figure 1: An example of three benchmarks belonging to low (lucas), high (bzip2) and saturating (perlbmk) cache utility groups.

of the cache serves as a monitoring logic, removing the need of the extra tag storage. In particular, the OS specifies which sets serve as the monitoring ones for a given core. These sets are accessed only by one thread, which allows us to monitor the behavior of this thread as it was running in isolation, and properly gather full profile. To ensure that only specified thread accesses its monitoring set, we force the accesses of the remaining threads to be remapped to other non-monitoring sets. This is done before accessing the cache by an *Index Remapping Logic* (IRL), so that the cache structure remains unchanged.

While the eATD approach seems a reasonable choice for both performance and power oriented research, in this paper we focus on the latter. We propose to use the area saved by the low-overhead monitoring logic to implement a *selective* drowsy mode. In this scheme, each line in a cache may be individually set into a low-power, drowsy mode [14]. The information stored in a line in this state is not lost. However, before the line is accessed, it needs to be *woken up* and brought back to the normal power mode. This introduces additional *drowsy latencies* affecting the overall performance. As a result, a partitioning scheme with power only as an optimization goal can decrease performance. There is a need for a power *and* performance aware cache partitioning policy that would benefit from the low-power mode and at the same time preserve the system performance.

To this end, we present ReCaC: *Reconfigurable Cache for CMPs*, a new L2 cache design for CMP architectures. The ReCaC augments L2 cache with 1) low-overhead monitoring logic (eATD), 2) selective drowsy mode and 3) novel power and performance aware cache partitioning policy. It targets to use a minimum cache space to trade-off power and performance. The ReCaC limits drowsy latencies cost, reducing the unnecessary line rouses. The partitioning policy is controlled by the Operating System that dynamically sets the *allowed performance degradation* factor (APD). The ReCaC assigns such a number of ways to a given application that it guarantees the highest power gains under a given APD. When the OS does not specify any particular APD, the ReCaC reduces the cache resources only if it does not affect performance. If the power savings are not achievable under the current APD, the ReCaC dynamically reverts back to the performance centric cache partitioning scheme.

There are two primary contributions of this work. First, we present and evaluate a low-overhead L2 cache profiling logic that can be easily implemented in the current processors and targets the area-constrained designs. Second, we present ReCaC, a new L2 cache scheme designed to improve energy efficiency within the power-performance trade-off specified by the OS. It is a general framework developed for the power-constrained designs.

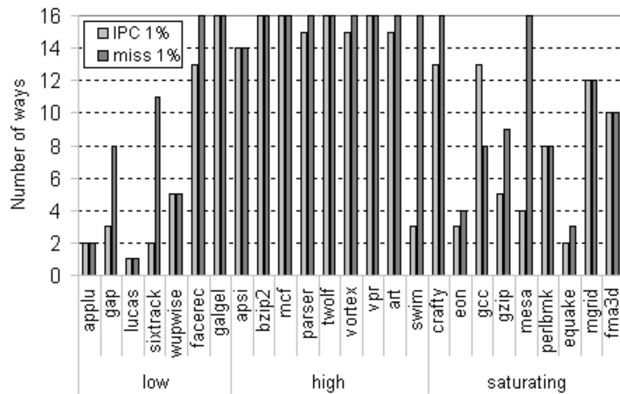


Figure 2: The minimum number of ways required to ensure on maximum 1% performance degradation (IPC) and miss rate increase (miss) for a 2MB 16-way L2 cache.

The rest of this paper is structured as follows. Section 2 motivates our research. We discuss our methodology in Section 3. Section 4 introduces the embedded monitoring logic and presents its results. Section 5 describes the ReCaC design, and Section 6 describes its results. We study related work in Section 7 and finally conclude in Section 8.

2. MOTIVATION

Increasing the amount of cache resources assigned to a thread does not always lead to performance improvement [27], since the utilization of the L2 cache varies widely across benchmarks. Applications can be classified into three groups [26] depending on the benefit they experience when they receive more cache space (ways). 1) *Low* utility applications do not benefit from the extra number of ways. Either they fit in the L1 data cache and do not make extensive use of the L2, or they do not fit in the L2 cache and frequently miss regardless of the ways they receive. 2) *High* utility benchmarks improve their performance when they receive more L2 cache ways. 3) *Saturating* utility applications improve their performance if they receive more L2 cache ways until a point when their performance saturates. Figure 1 shows the IPC curve of three benchmarks belonging to low, high, and saturating utility groups when varying the number of ways used by the benchmarks.

Previous studies [27] show that the cache partitioning algorithms lead to a very limited performance improvement with workloads composed of low and saturating utility benchmarks. In our view, these benchmarks create an opportunity for power savings. A dynamic cache partitioning mechanism can recognize the cases with limited performance gains and employ power optimizations.

Figure 2 shows the minimum number of ways that each benchmark has to use in order to keep the performance degradation below 1% (light-grey bars) and miss rate increase below 1% (dark-grey bars). Some benchmarks suffer a small performance degradation when they use only a subset of ways. For example, *gap* suffers less than 1% IPC degradation when assigned only 3 ways. In general, the best candidates for power savings are the low and the saturating

(a) Baseline processor configuration

Processor setup
CORE: 8 wide, out-of-order, 98 entry reservation station branch predictor: select best from bimodal & gshare BTB: 1KB, 4-way; min penalty - 3 cycles
L1 cache: Icache: 64KB, 2-way, 128B line, LRU, 11 cycles miss penalty Dcache: 32KB, 2-way, 128B line, LRU, 9 cycles miss penalty
L2 cache: Unified: 2MB, 16-way, 128B line size, 250 cycles miss penalty, *MinMisses* policy

(b) Workload summary

Benchmarks acronyms and types			
a-applu *(low)*	h-facerec *(low)*	o-mcf *(high)*	w-swim *(high)*
b-apsi *(high)*	i-fma3d *(sat)*	p-mesa *(sat)*	x-vortex *(high)*
c-art *(high)*	j-galgel *(low)*	r-mgrid *(sat)*	y-vpr *(high)*
d-bzip2 *(high)*	k-gap *(low)*	s-parser *(high)*	z-wupwise *(low)*
e-crafty *(sat)*	l-gcc *(sat)*	t-perlbmk *(sat)*	
f-eon *(sat)*	m-gzip *(sat)*	u-sixtrack *(low)*	
g-equake *(sat)*	n-lucas *(low)*	v-twolf *(high)*	
Workloads evaluated			
type	**2 threads**	**4 threads**	**8 threads**
low-low	ak; nu; hz; jh	aknu; njhz	aknuhzjh
high-high	vy; bv; bc; bw; wv; cs; bd; os; vx; yc	bdos, svxy; wbyc; vybc; wvcs	abosvwyc
sat-sat	et; ef; lm; pt; gr	eflm; pflm; ptgr	eflmptgr
low-sat	kp; ae; kf; nl; um; zp; ht; jg; hr	aekf; nlum; zpht; jghr; kpet	aekfnlum; zphtjghr
low-high	jy; jv; bj; yz; oa; ab; kd; no; us; zc	abkd; nous; hvzc; jyvb	abkdnous; noushvzc; jyvbcwsz
high-sat	mv; ce; ie; be; df; ol; sm; vp; xt; yg; cr	bedf; olsm; vpxt; ygcr	bedfolsm; vpxtgcr
low-high-sat		yzme; iwoa	meioakpt

Table 1: Processor setup and workload summary.

utility applications. For these cases, some cache ways do not contribute significantly to the final performance and can be maintained in the low-power drowsy mode [14].

In our research we conservatively assume that a X% miss rate increase translates into X% performance degradation. When the OS establishes a given $APD = X\%$ for a thread T0, the ReCaC varies the cache allocation so that the T0 does not experience an increase in its L2 miss rate higher than X%.

3. METHODOLOGY

3.1 Processor and Memory Configuration

We use an enhanced version of a detailed cycle-accurate simulator, Turandot from IBM [16, 23], the *Parallel Turandot CMP* (PTCMP) [12]. The power statistics are acquired from the IBM's PowerTimer methodology [6] and HotSpot 2.0 [17,28] (for leakage power) integrated with Turandot [7]. The methodology used in this research to model cache power follows the one presented in [12]. We have implemented drowsy cache models according to [14].

Table 1(a) shows our baseline processor configuration. We model 2-, 4- and 8-core CMP processors with 1 thread executing in each core. Both instruction and data first level caches are private to each core, while the L2 cache is shared between all cores. The processor configuration remains constant for all the experiments.

We also take into account the power overhead of accessing off-chip memory. We assume that the power cost of a memory access is 150 times higher than an access to L2 [8]. Given that this value depends on the particular processor chip and memory architecture, in Section 6.2 we make sensitivity analysis for different architectures.

3.2 Metrics and Benchmarks

We use three performance metrics: IPC throughput defined as sum of the N threads IPCs $\sum_{i=1}^{N} IPC_i$, weighted speedup [29] defined as $\sum_{i=1}^{N} IPC_i^{CMP}/IPC_i^{isolation}$ and harmonic mean [21] defined as $N/(\sum_{i=1}^{N} IPC_i^{isolation}/IPC_i^{CMP})$. In order to evaluate the energy efficiency of the entire processor and memory we use CPI^2 *x Power* and CPI^3 *x Power* metrics. It has been proved in [6] that these metrics correspond to *Energy x Delay* [15] and *Energy x* $Delay^2$ [6] products, respectively. We compare the energy-efficiency of the baseline system and our proposal and we present relative values.

We use the SPEC CPU 2000 suite [1] to evaluate our proposal. We combine the benchmarks into 49 two-thread workloads, 25 four-thread workloads and 11 eight-thread workloads with randomly selected benchmarks. Table 1(b) depicts workload summary. We generate traces using SimPoint methodology. We stop the simulation when the slowest thread commits 100 million instructions.

4. EMBEDDED MONITORING LOGIC

4.1 Hardware Requirements

Our profiling logic proposal, *embedded Auxiliary Tag Directories*, uses part of the L2 cache for monitoring purposes, leaving the rest for a normal execution. We distinguish three types of the L2 cache sets: 1) *monitoring* sets, that are accessed only by one thread; 2) *normal* sets, corresponding to the default cache sets; and 3) *remapped* sets, that, in addition to the data of the direct access, store part of the data moved from the accesses to the monitoring sets. We add the *Index Remapping Logic* that remaps some accesses from the monitoring sets to the remapped sets. The structure of the L2 cache remains unchanged, since we add the additional hardware before accessing the cache.

For example, let us assume that we run two threads, T0 and T1, in a 2-core architecture with a shared L2 cache. Also, we define the cache set 0 as a monitoring set for core 0 (running T0) and the cache set 1 as a monitoring set for core 1 (running T1), as Table 2(a) depicts. In order to do so, we remap all accesses of the thread T0 from the set 1 to the set 2, and all accesses of the T1 from the set 0 to the set 3. We call the sets 0 and 1 *monitoring* sets (mon), the sets 2 and 3 *remapped* (rmp) sets, and the sets 4 to 7 *normal* (nor) sets.

A remap logic prevents threads T0 and T1 from using the monitoring sets of each other, namely sets 1 and 0, respectively. This ensures that a given thread executes in a single-threaded mode in its monitoring set, and allows us to estimate how each way given to a thread affects its L2 miss rate. Therefore, we are able to plot L2 miss rate of each core as a function of the number of assigned ways to a given core.

The remap logic identifies whether a given thread accesses a normal/remapped set or accesses a monitoring set. We select the *K* least significant bits from the address index (*K=3* in the example). We call these *K* bits a *current pattern* and drive it as an input to the IRL. IRL is a combinational logic that may replace the *current pattern* bits with a *new pattern*. In the depicted example, if we want to remap an access of T0 from the set 1 to the set 2, we set the *new pattern* as '010' for the *current pattern* '001' for the thread T0 (see Table 2(a)). Since the *current pattern* is built only with the least

(a) Rationale and example of the eATD

Example set mapping for a 2-core configuration				
set number		eATD		set
dec	bin	T0	T1	type
0	000	0	-	mon
1	001	-	1	mon
2	010	2 & 1	2	rmp
3	011	3	3 & 0	rmp
4	100	4	4	nor
5	101	5	5	nor
6	110	6	6	nor
7	111	7	7	nor

Example signals for Index Remapping Logic			
thread	IRL & R-table	current pattern	
		000	001
T0	new pattern	000	010
	R-bit	0	1
T1	new pattern	011	001
	R-bit	1	0

(b) Comparison of the sATD and eATD storage overhead

sATD, 64 monitoring sets		
size of each ATD entry	47 tag bits + 1 valid bit + 4-bit LRU	52 bits
one monitoring set size	52 bits x 16-ways	104 B
total size	64 monitoring sets x 2 threads	**13312 B**
eATD, 64 monitoring sets		
Index Remapping Logic	2 x current pattern reg + 4 x new pattern reg + 4 x R-bit	3.5 Bytes
R-table size	64 rmp sets x 16-ways x 2 threads	256 B
total size	IRL + R-table size	**259.5 B**

Table 2: The eATD rationale and storage overhead.

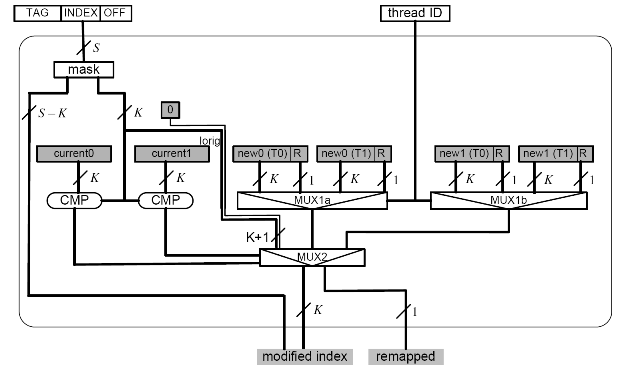
significant bits, the effect of the bit replacement is comparable to adding a fixed number to the set number. Therefore, accesses from T0 will be remapped from the set 1 to 2, 9 to 10, 17 to 18, etc. Analogously, accesses from T1 will be remapped from the set 0 to 3, 8 to 11, 16 to 19, etc. Such a design does not require set decoder modifications, as the total number of *set* bits remains unchanged.

In the eATD approach we have an aliasing problem when a direct access to a given set has the same tag as a remapped access to that set. To solve the problem, we use a bit that indicates whether a given line in a set stores the information from a direct or from a remapped access. We call this bit a *remapped* bit (*R-bit*). For each access to the cache, the IRL logic sets *R = 1* if the access is remapped and *R = 0* for a direct access. If there is a cache miss, the R-bit is saved as a part of the tag.

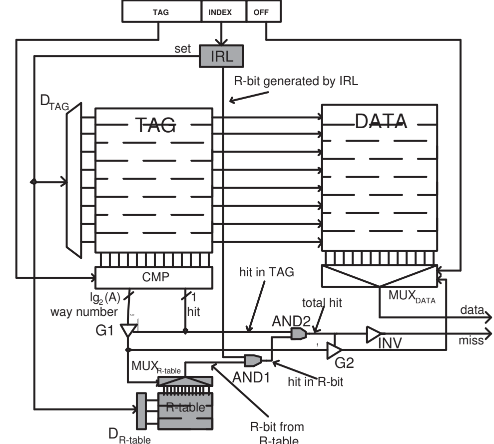
Figure 3(a) shows the logic required to implement the IRL for a 2-core CMP. The logic comprises 3 multiplexors, 2 comparators and 3 registers. The *mask register* is ANDed with the index of the address to select the *K* least-significant bits from the set of *S* index bits. We compare the *K* least significant bits with the current patterns. The remaining *S-K* most significant bits are unchanged and delivered to the IRL output. The *Current Pattern registers* store the current patterns that have to be identified, and the *New Pattern registers* store the new patterns. The *R registers* store the remapped (R) bits corresponding to the new patterns.

Figure 3(b) shows the integration of the IRL logic in the L2 cache. Instead of saving one R-bit per line in the L2 cache, we store the R-bits in a separate structure called the *R-Table*. The R-table saves the *R-bits* of the *remapped* sets, one bit per each line. We access the R-table in parallel with the tag directory. The area of the R-table is $A \times T_{rmp}$ bits, where A is the L2 cache associativity and T_{rmp} is the maximum number of *remapped* sets, for which the eATD is designed.

Mask pattern, *current pattern*, *new pattern* and *R registers* are exposed to the OS to control the index remapping performed by the IRL. In a given example, if the OS decides that the core 0 (running



(a) IRL implementation for a dual-core architecture



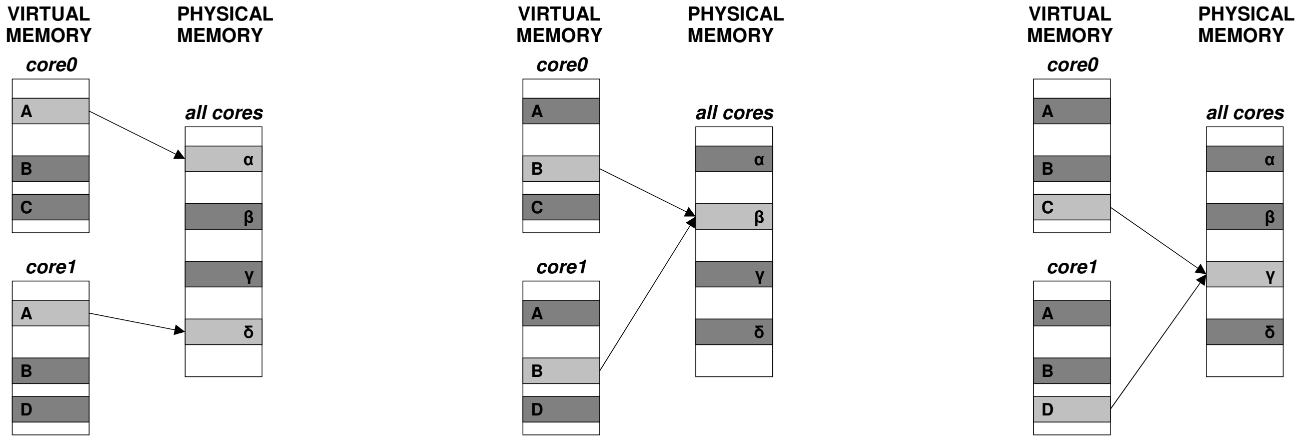
(b) Integration of IRL and R-table in the L2 structure

Figure 3: IRL implementation and integration into the L2.

T0) is off, it can cancel remapping for the T0 by setting IRL's registers. This increases the amount of the *normal* sets for T1. The number of rows in the R-table, however, is budgeted for the case when all the cores are used - the R-table stores the R-bits for the *maximum* number of remapped sets, which supports the case when the OS decides to use all cores. In Section 4.3 we show what is the best number of monitoring/remapped sets for 2, 4 and 8-core CMPs in our baseline configuration. The OS changes index remapping on the context switch. The TLB and L2 cache flush at the time cleans the previous remapping scenario.

The physical structure of the L2 cache remains unchanged. The *hardware support* elements for the eATD are marked in grey in Figure 3(b) and consists of the IRL, the R-table, and gates AND1, AND2. We add one additional clock cycle to every cache access, for the IRL to perform index remapping and $MUX_{R-table}$, AND1, AND2 gates to select and check appropriate R-bit for a hit. We access the R-table in parallel with the tag directory. By adding one additional cycle for the eATD hardware support, we eliminate the need to modify the L2 cache set decoder.

Table 2(b) compares the area overhead of the eATD and sATD [27] in the 64-bit architecture, where both use 64 monitoring sets in 16-way L2 cache. The sATD experience 6656 bytes overhead, while the eATD overhead is only 131.5 bytes. The area cost of the eATD is independent of the tag width, in contrast to the sATD design [27].



(a) Two same virtual addresses mapped to two different physical addresses.

(b) Two same virtual addresses mapped to the same physical address.

(c) Two different virtual addresses mapped to the same physical address.

Figure 4: Examples of the virtual to physical address mapping. We use virtual addresses in private L1 caches and physical addresses in shared L2 cache.

accessing address	p-IRL generates			set searched	value searched	profiling		where should miss?		update LRU stack on HIT?		update LRU stack on MISS?	
	TAG	INDEX	R	T0 & T1	T0 & T1	T0	T1	T0	T1	T0	T1	T0	T1
FF0	FF FF	0 3	0 1	0 (mon for T0) 3 (rmp for T1)	FF (TAG) FF (TAG) & R=1	yes N/A	no N/A	in set 0	in set 3	yes yes	no yes	yes N/A	N/A yes
FF1	FF FF	1 2	0 1	1 (mon for T1) 2 (rmp for T0)	FF (TAG) FF (TAG) & R=1	no N/A	yes N/A	in set 2	in set 1	no yes	yes yes	N/A yes	yes N/A
FF2	FF	2	0	2 (rmp for T0)	FF (TAG) & R=0	N/A	N/A	in set 2	in set 2	yes	yes	yes	yes
FF3	FF	3	0	3 (rmp for T1)	FF (TAG) & R=0	N/A	N/A	in set 3	in set 3	yes	yes	yes	yes
FF4	FF	4	0	4 (nor set)	FF (TAG)	N/A	N/A	in set 4	in set 4	yes	yes	yes	yes
FF5	FF	5	0	5 (nor set)	FF (TAG)	N/A	N/A	in set 5	in set 5	yes	yes	yes	yes
FF6	FF	6	0	6 (nor set)	FF (TAG)	N/A	N/A	in set 6	in set 6	yes	yes	yes	yes
FF7	FF	7	0	7 (nor set)	FF (TAG)	N/A	N/A	in set 7	in set 7	yes	yes	yes	yes

Table 3: Searching the data when running parallel applications. The examples correspond to Table 2(a). *N/A* and *R* stand for *Not Applicable* and *R-bit*, respectively.

4.2 Parallel Applications

4.2.1 Data Sharing

The OS assigns memory addresses (pages) to user processes. In case of parallel applications, the process may use several threads that share the address space, each executing on a different core in a CMP architecture. This implies that cores may not necessarily work on disjoint address spaces. If the cores share data in the monitoring sets, and eATD constrains each core to a separate L2 cache area, then there would be multiple copies of the same data. In this scenario the correctness can be violated if either of the copy becomes dirty.

We investigate this problem in greater detail in Figure 4. Let us assume that core 0 and core 1 are using virtual memory addresses as depicted at Figure 4(a). There are addresses *A*, *B* and *C* within virtual address space used by the core 0. Similarly, there are addresses *A*, *B* and *D* within virtual address space used by the core 1. If the OS sets the virtual to physical address mapping as in Figure 4(a), two same virtual addresses for two cores are mapped to two different physical addresses. In this case, virtual address *A* of the core 0 is mapped to physical address α, and virtual address *A* of the core 1 is mapped to the physical address δ. We find such a mapping in non-parallel SPEC CPU 2000 [1] benchmarks. In this case eATD depicted in Section 4.1 is guaranteed with separate physical address space per each core, and remapping some data from the monitoring sets does not influence the final correctness.

However, for the parallel applications there are some data shared between the cores. Figure 4(b) depicts the situation when two same virtual addresses *B* of the core 0 and core 1 are mapped to the same physical address β. Similarly, Figure 4(c) shows the case when two different virtual addresses, *C* for the core 0 and *D* for the core 1, are mapped to the same physical address γ. In both cases, if the shared physical data, β or γ, is by default stored in one of the monitoring set, the core which accesses are remapped from a given monitoring set cannot find the data in L2 cache. For example, let us assume that core 0 (running thread T0) attempts to access address *FF0*, where *FF* are the tag bits and *0* are the index bits. Further assume that we use set mapping as Table 2(a) depicts. Whenever core 0 issues FF0 address to access L2 cache, it accesses its own monitoring set, namely set 0. However, if the FF0 data is shared among the cores, and core 1 tries to access the data, it will search in the remapped set 1 instead. As a result, even if the FF0 data is present in set 0 (and has been written by core 0), core 1 will detect the FF0 to be a miss, since it is not found in the remapped set 1.

4.2.2 Parallel eATD

The data sharing problem affects only monitoring sets, as part of the data these sets would normally store is moved to the remapped sets. To this end, we propose *parallel eATD* that searches both monitor-

ing and corresponding remapped sets. For the example given in Table 2(a), whenever accessing set 0 (monitoring set for core 0), all the cores look for a hit in sets 0 and 3. Similarly, whenever accessing set 1, all the cores search for a hit in sets 1 and 2. For example, when looking for addresses FF0, a *parallel Index Remapping Logic* (p-IRL) generates two addresses to be searched: FF0 to access set 0 (tag bits FF, and 0 determines set number) and FF0 with R-bit equal 1 to search set 1. Let us note that lines in remapped set 1 with R-bit equal 1 store the data of the set 0, that would be accessed by the address FF0 for the case there is no eATD implemented. The lines in set 1 with R-bit equal 0 store the data of the set 1 (no remapping), that would be accessed by the address FF1 (tag bits FF, and index bits 1) for the case with no eATD implemented.

When accessing monitoring set, parallel eATD searches for the data always in one monitoring and one remapped set, regardless of the number of running cores. This is due to the fact that we remap all the data from the monitoring set (for all the cores that are not owners of the given monitoring set) to only one remapped set. Therefore, we roughly double the access pressure on the remapped sets. This allows us to ensure that only two sets need to be searched in the worst case, regardless of the number of running cores.

Table 3 depicts details of the data searching for the parallel applications when accessing monitoring, remapping and normal sets for the example set mapping in Table 2(a). When looking for the data stored at address FF0 that by default maps to the monitoring set 0, we search for a hit both in the set 0 and in the corresponding remapped set 3 (with R-bit equal 1). Both cores are allowed to hit in any of the sets. However, we do not allow core 1 to miss in the set 0, in order to protect the profile information gathered on the base of a behavior of the core 0 in the set 0. For the same reason we do not update LRU stack whenever core 1 hits in the set 0. However, we always update LRU stack in case of a miss (core 0 can miss in the set 0 and core 1 in the set 3). When accessing remapped sets (addresses FF2 and FF3 in Table 3) we search only corresponding remapped sets with R-bits set to 0 (sets 2 and 3, respectively). When accessing normal sets (addresses FF4, FF5, FF6 and FF7) we search normal sets (sets 4, 5, 6 and 7, respectively), as in the baseline L2 cache configuration.

Table 3 also depicts the values generated by the p-IRL. Whenever accessing the monitoring set, p-IRL generates two accesses to the L2 cache. For example for FF0 address, p-IRL generates addresses FF0 with $R = 0$ and FF3 with $R = 1$, as shown in Table 3. For every access p-IRL compares accessing address with all *current patterns*. If the last K bits of the access match any of the current patterns, p-IRL recognizes that one of the monitoring sets is to be accessed (without distinguishing which is the accessing core and the owner core of the given monitoring set). In that case p-IRL generates two accesses to L2 cache, one built with current pattern (FF0 in our example), and the other built with new pattern (FF3 in our example).

Figure 5 shows simplified view of the L2 cache and parallel IRL integration. If p-IRL detects access to any remapped or normal set, it does not modify the cache access and puts the address in $entry0$ of the p-IRL queue. Whenever p-IRL detects access to any monitoring set, it generates two accesses and puts them in $entry0$ and $entry1$ of the p-IRL queue. Two accesses are served in order, which increases the latency when accessing monitoring sets by a factor of 2. As we show in Section 4.3, we achieve the best results having 32 monitoring sets regardless of the number of running cores.

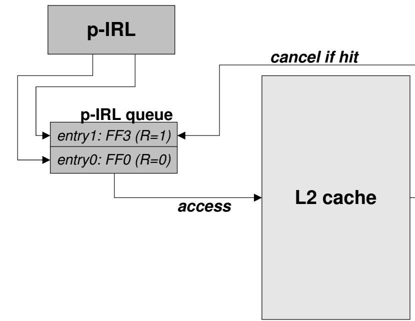


Figure 5: Simplified view of the L2 cache and parallel IRL integration. Example values in p-IRL queue entries correspond to address FF0 (tag bits FF, index bits 0) cache access and set mapping as in Table 2(a).

In this case, space occupied by the monitoring sets corresponds to 3.125% of the total cache space, in our baseline L2 cache configuration with 1024 sets. If we assume uniform data distribution across the sets and each set to be accessed with the same frequency, only around 3% of accesses are penalized with double access latency. We believe this has a negligible performance impact. Furthermore, Figure 5 shows that if $entry0$ of the p-IRL queue already hits in L2 cache, $entry1$ can be cancelled to decrease access latency.

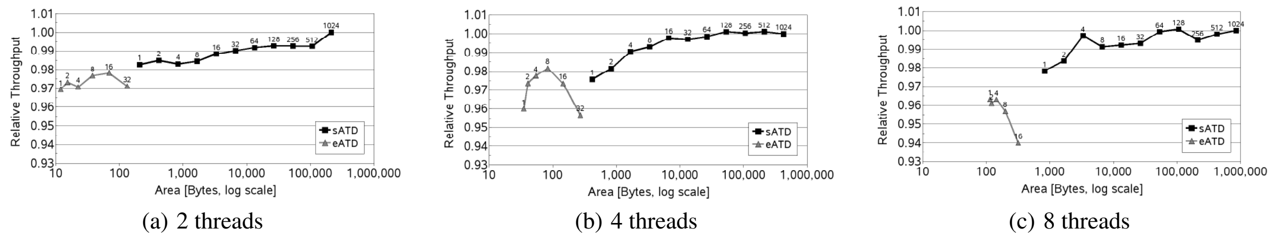
To conclude, if we detect that a given physical access to the L2 cache points to a monitoring set, we search both a given monitoring set and a corresponding remapped set to which part of the content of the monitoring set was moved. If the access points to a remapped or normal set, we search only a given remapped or normal set, respectively.

In this paper we evaluate our proposal with non-parallel SPEC CPU 2000 [1] benchmarks that use separate physical address spaces with virtual to physical address mapping as depicted at Figure 4(a). For this reason we use non-parallel eATD implementation depicted in Section 4.1 for the rest of the paper.

4.3 Embedded Monitoring Logic Results

Cache access latency. When using eATD as a profiling logic, we need one additional clock cycle to perform index remapping for each L2 cache access. Our results show that increased access latency causes negligible effect on the performance for both the LRU and the *minMisses* [27] replacement policies. The average performance degradation stays always below 0.5% for both policies, when varying the number of cores from 1 (LRU) up to 8.

Embedded ATD. Figure 6 shows the area and the performance results when using the eATD or the sATD as a profiling logic. In both cases we use the *minMisses* as the replacement policy [27]. In our baseline L2 cache configuration with 1024 sets, the eATD can support up to 128, 64, 16 *monitoring* sets for 2, 4, 8-core CMPs, respectively. The fewer the monitoring sets, the more cache space is shared between the cores and less cache space works in single-threaded mode. Moreover, small number of the monitoring sets implies small number of the remapped sets, and small data trashing due to the additional remapped accesses. However, as we reduce the number of the monitoring sets, the accuracy of the predictions done by the profiling logic drops, which negatively affects the per-



(a) 2 threads (b) 4 threads (c) 8 threads

Figure 6: Performance versus the area results for the eATD and sATD for 2MB 16-way L2 cache. For each case we depict the number of the monitoring sets per core.

formance. As a result, the performance follows a concave function, reaching its maximum for 16, 8 and 4 monitoring sets per core for 2, 4 and 8-core CMP, respectively. The results (not shown here) indicate the same trend for weighted speedup and harmonic mean metrics. Therefore, we achieve the highest performance when there are 32 monitoring sets in total, regardless of the number of running cores.

Figure 6 also shows that the area of the monitoring logic in the eATD is significantly lower than in the sATD with small impact on performance. We compare the eATD configuration achieving the highest performance with the baseline sATD with 32 monitoring sets [27]. The eATD requires total area of 69, 83.5 and 144 bytes for 2, 4 and 8-core architectures, respectively. Therefore, eATD reduces the area by a factor of 96, 159 and 185, respectively. The throughput degradation is 1.2%, 1.6% and 3.1%, respectively.

No remapping. Figure 7 shows the throughput of the NR architecture, where instead of remapping some cache accesses, we force them to miss in the L2 cache. We compare eATD and NR schemes with 12 and 11 clock cycles L2 cache access latency, respectively. The results are normalized to the eATD case with 16 monitoring sets. We observe that higher number of the monitoring sets decreases the throughput of the NR scheme, as the higher number of the L2 cache accesses are forced to miss. The NR architecture achieves the best results when it uses 2 monitoring sets - in this case performance is 0.6% lower with respect to the best eATD configuration.

5. POWER-AWARE CACHE DESIGN

5.1 Algorithm

The cache partitioning algorithm used by the ReCaC involves two steps.

Performance step. First (lines between 1 and 7 in Algorithm 1), the algorithm finds a cache partition that improves the performance. It uses the *minMisses* policy [27]. However, the ReCaC is flexible to use any other performance-aware policy. The minMisses algorithm assigns ways to the running threads so that it minimizes the overall number of misses. The *get_next_partition* procedure (line 2) generates all possible cache partitions of the cache among the running threads. For example, if there are two threads T0 and T1 accessing a 6-way L2 cache, it generates the following partitions: {1,5}[1], {2,4}, {3,3}, {4,2}, {5,1}. The algorithm assigns to each

[1]First digit defines number of ways assigned to the thread T0, and second digit defines number of ways assigned to the thread T1.

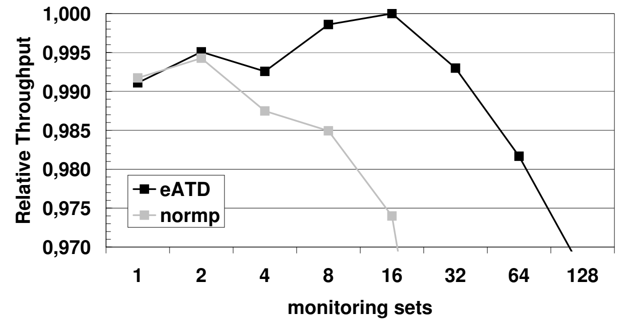


Figure 7: Performance results for eATD (eATD) and embedded monitoring logic without remapping (normp). ***eATD*** **and** ***normp*** **have 12 and 11 clock cycles L2 cache access latency, respectively.**

thread between 1 and $A - N$ ways, where A is the cache associativity and N is the number of threads. Each way is guaranteed to be assigned to only one thread, and each thread is guaranteed to be assigned to at least one way. Let us assume that after the first step the minMisses scheme finds the partition {1,5}to have the lowest miss rate.

Algorithm 1 ReCaC cache partitioning algorithm

```
 1: performance step: run minMisses
 2: while get_next_partition do
 3:   if lowest_miss_rate > current_miss_rate then
 4:     minMisses = current_partition;
 5:     lowest_miss_rate = current_miss_rate;
 6:   end if
 7: end while
 8: power step: optimize for power
 9: while get_next_partition do
10:   if performance_cost < APD_registers then
11:     if energy_gains > energy_cost then
12:       if best_gains < current_gains then
13:         best_partition = current_partition;
14:         best_gains = current_gains;
15:       end if
16:     end if
17:   end if
18: end while
19: return best_partition;
```

Power step. Second (lines between 8 and 18), the algorithm searches for the partition with the highest net energy gains with respect to the *minMisses partition* found in the previous step. In the above example, the policy evaluates the partitions {1,1}, {1,2}, {1,3} and {1,4} and compares their power consumption and per-

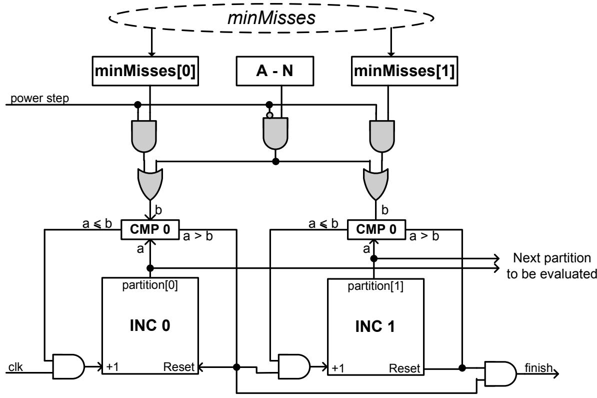


Figure 8: Simplified view of the hardware implementation of *get_next_partition()* function from Algorithm 1. Shaded logic corresponds to the additional hardware for the ReCaC functionality.

formance to the {1,5} selected by the minMisses. It can apply the minMisses partition, if it is the most energy effective scenario for given threads. The best candidate is the one that guarantees 1) the performance degradation lower than the allowed performance degradation (APD) threshold specified by the OS (line 10) 2) the energy gains higher than energy cost due to the miss rate increase and energy-costly off-chip accesses to the main memory (line 11). To compute the energy cost of the additional memory accesses for the miss rate increase (reported by the profiling logic) we multiply estimated number of misses by the energy of each access to the main memory. We assume that the average memory access energy is constant in a given architecture.

To decrease the complexity of our proposal and to avoid the analysis of which lines should be set to the drowsy mode, we set all cache lines to the low power mode on every interval boundary. Therefore, some lines in which during the next interval either a hit occurs or they have been allowed to serve a miss, will be waken up. Once a line is waken up, it remains awake during the rest of the interval, so that the performance cost of unnecessary switching a line between low and normal power mode (and introducing drowsy latencies) is limited.

5.2 Complexity Analysis

Figure 8 shows a simplified view of the *get_next_partition* hardware implementation in a 2-core CMP. The logic contains two incrementers (INC0 for the thread T0 and INC1 for the thread T1), each generating $log_2(A-N)$-bit number of ways assigned to a corresponding thread on the $partition$ output. The incrementers have as an input *increase* and *reset*, marked as $+1$ and $Reset$, respectively.

When the signal $power_step = 0$, the hardware generates partitions to be checked in the *performance step*. It assigns to each thread between 1 and $A-N$ ways, starting from {1,1} partition up to {A,A}. The minMisses policy ignores the invalid partitions, when the total number of ways assigned to both threads does not sum up to A. As the performance step finishes, signal $finish = 1$ and the minMisses policy stores the partition with the lowest total number of misses in minMisses registers. Next, when the signal $power_step = 1$, the hardware generates partitions to be checked in the *power step*. Similarly, it assigns to each thread between 1

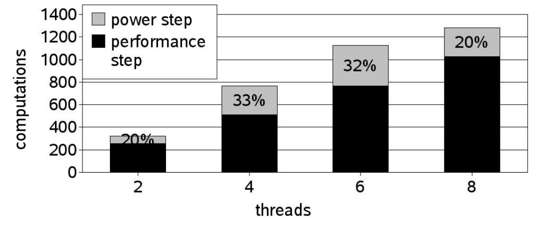


Figure 9: Complexity analysis of the ReCaC algorithm

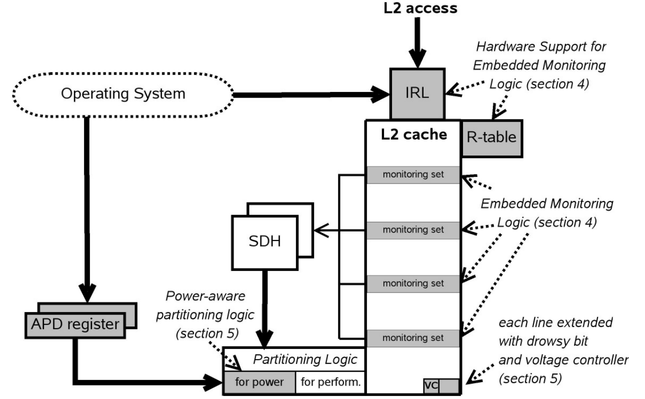


Figure 10: ReCaC architecture.

and value stored in the corresponding $minMisses$ register. Likewise, signal $finish = 1$ informs that power step finishes. Figure 8 shows an easy implementation of the ReCaC power step functionality, marked with grey, into the existing performance step. For clarity reasons, we do not show the initialization logic (each thread is given at least one way), the clock propagation tree and the logic disabling a generation of the invalid partitions.

$$C = C_{perf} + C_{power} = \frac{A^2}{2} \times N + (\frac{A}{N})^N \quad (1)$$

Equation 1 shows a formula of the algorithm complexity for *A*-associative L2 cache shared by *N* cores. It is a sum of *performance* (C_{perf}) and *power* (C_{power}) step complexities. We define the total *complexity C* as the number of unique partitions to be checked when searching for the best candidate. We assume that the power consumption and performance of each partition can be evaluated in one clock cycle[2]. Therefore, the complexity transforms into the number of clock cycles required to find the best partition. Figure 9 plots the complexity ratio between the power and performance steps for a 16-way L2 cache. The power step is a smaller fraction of the whole. It corresponds to 20% of the total complexity for 2 and 8 threads.

5.3 How to Use ReCaC

ReCaC, depicted in Figure 10, is controlled by the Operating System. On a context switch, the OS can define new performance goals - it decides the Allowed Performance Degradation (APD) factor. For example, for a given power budget, the OS can decide to allow the ReCaC to decrease performance and save power (as the OS does with DVFS). For multimedia applications, the OS deter-

[2]Our results show negligible performance change when we vary the evaluation time of each partition from 1 to 5 clock cycles.

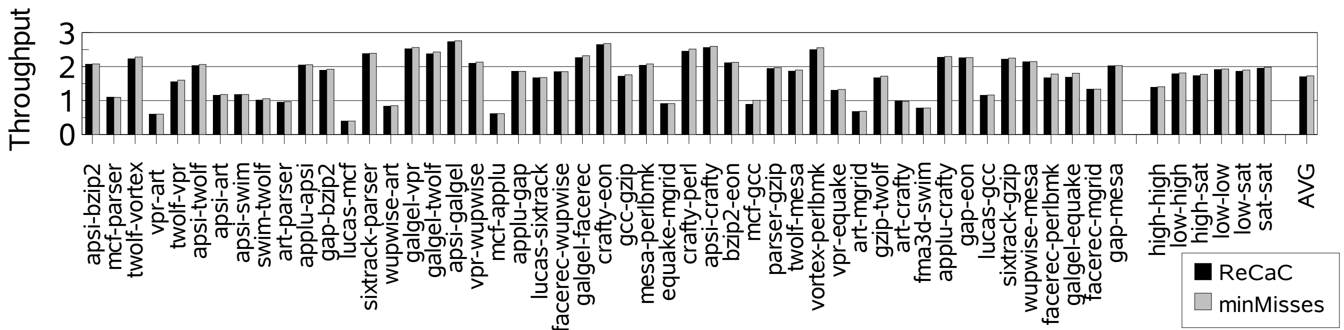
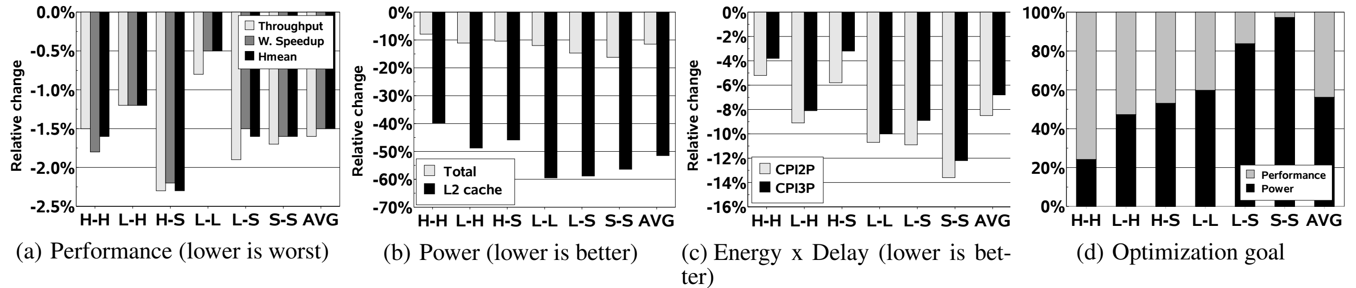


Figure 11: Performance for ReCaC and baseline design. As a baseline we use minMisses policy guided with sATD.



(a) Performance (lower is worst)
(b) Power (lower is better)
(c) Energy x Delay (lower is better)
(d) Optimization goal

Figure 12: ReCaC results. H-H, L-H, H-S, L-L, L-S and S-S stands for *high-high*, *low-high*, *high-sat*, *low-low*, *low-sat* and *sat-sat* workloads, respectively.

mines how much an application can be slowed down to meet performance goals reducing power consumption. Likewise, setting the APD of threads in the unbalanced High-Performance Computing applications can save power without affecting the overall application performance. For a high value of the APD, the ReCaC has more possibilities to save power, as it can adjust less L2 cache ways to the executing threads and maintain a higher amount of cache in the drowsy state. The OS informs the ReCaC about the maximum degradation allowed for each running thread via special purpose registers, the *APD registers*. These registers, one per core, specify tolerable performance degradation. The partitioning policy adjusts its behavior to the current OS guidelines. In this way the OS provides the Quality of Service (QoS) inside the L2 cache, since the power-performance trade-off is set individually per each core. We leave QoS studies in the ReCaC for our future research.

6. RECAC RESULTS AND ANALYSIS

Unless stated differently, we perform cache partitioning every 750 thousand clock cycles, and assume that the power consumption in the drowsy mode is 20% of the normal mode power [14]. We model the power cost of the drowsy-to-normal and normal-to-drowsy transitions [14] and take into account the energy wasted due to the additional stall cycles for the increased miss rate. We use performance oriented *minMisses* policy drived with the sATD in our baseline architecture. The energy cost of the off-chip access is 150 times higher than the L2 cache access [8]. The baseline architecture does not contain the low-power drowsy mode.

6.1 Case Study

We analyze different combinations of two threads executing in a 2-core processor. We breakdown the results into 6 groups based on the type of threads in each workload: *high-high*, *low-high*, *high-sat*, *low-low*, *low-sat* and *sat-sat*. Figure 11 compares the throughput of the ReCaC (driven by the eATD) and the baseline *minMisses* partitioning scheme (using sATD), when we set the APD to 1% in both cases for 2 running threads. The ReCaC introduces the throughput degradation of 1.6% with respect to the baseline. We observe similar trends with weighted speedup (1.5% degradation) and harmonic mean (1.5%). Further improvements in the harmonic mean can be obtained when the partitioning policy is modified to favor fairness.

Figure 12(a) shows the performance degradation for each workload type. When both threads are low utility, the throughput reduction is only 0.8%, as these benchmarks are less sensitive to the amount of assigned cache space. We observe similar trend with weighted speedup and harmonic mean (0.5% reduction in both cases). It makes low utilization benchmarks good candidates for power savings. The L2 cache power is decreased by 59.5%, which translates into 11.5% of the total power savings, as Figure 12(b) depicts. Figure 12(c) shows that the final energy-efficiency measured with CPI^2P and CPI^3P metrics is improved by 10.7% and 10%, respectively. Overall, in 59.8% of the partitioning decisions the ReCaC finds the opportunity for the power savings, as depicted at L-L case in Figure 12(d).

Saturating (*sat*) cache utilization benchmarks are even better candidates for power optimizations - the ReCaC finds power savings opportunities in 97.3% cases. It saves 16.2% of the overall processor and memory power, as a consequence of small working sets of these benchmarks. ReCaC improves the energy efficiency by 13.6% and 12.2%, with CPI^2P and CPI^3P, respectively.

On the contrary, *high* cache utilization applications are the worst candidates for power optimizations in the L2 cache. In this case, every way not assigned to the benchmark causes a noticeable per-

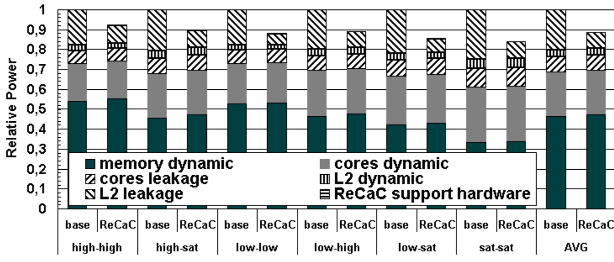


Figure 13: Power breakdown in ReCaC and baseline designs.

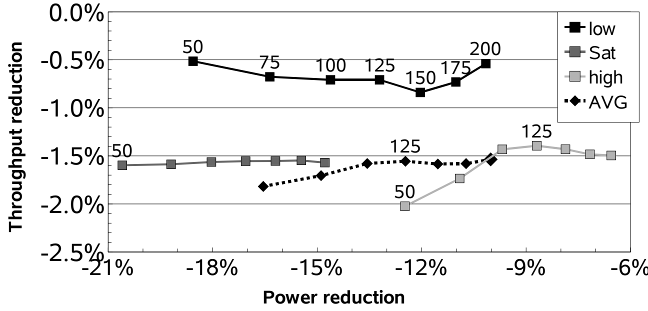


Figure 14: ReCaC sensitivity studies.

formance reduction. The ReCaC takes this fact into account and optimizes for performance rather than for power. As a result, performance is not significantly affected (throughput decrease is similar to *sat* group), but, as expected, the total chip and memory power savings of 7.9% (39.8% in L2 cache) are the smallest among all the groups. If there is only one high utility thread, the power savings are higher - 11.1% and 10.4% total power savings for low-high and high-sat workloads (48.8% and 45.9% in L2 cache), respectively.

Figure 13 shows the power consumption per processor component for both the baseline and ReCaC architectures. The ReCaC saves power in the L2 cache on average by 51.5%, with an increase of 1.7% in the main memory power, due to a higher miss rate. The leakage and dynamic power cost of the eATD (IRL, R-table, APD registers) is negligible - 0.003% of the baseline L2 cache design. Similarly, the transition power of the lines switching between the drowsy-normal-drowsy modes corresponds to 0.042%, and the total leakage of voltage controllers (VC) to 0.54% of the baseline L2 cache. Therefore, the power consumption of the eATD, the lines transition power and the VC's leakage marked as *ReCaC support hardware* in Figure 13 corresponds to 0.59% of the baseline L2 cache power, which transforms into 0.14% of the total baseline system power (chip and memory). Power consumption of the ReCaC support hardware is not visible in Figure 13, as it is a negligible factor in the whole system power.

6.2 Sensitivity and Scalability Analysis

Sampling interval. We vary the sampling interval from 100 thousand to 20 million cycles. Our results show that the cache partitioning every 750 thousand cycles gives the most energy-efficient scenario. If partitions are assigned more often, the ReCaC is too sensitive to temporal changes in the program behavior. On the contrary, with infrequent partitioning, the ReCaC does not follow all the program phase changes.

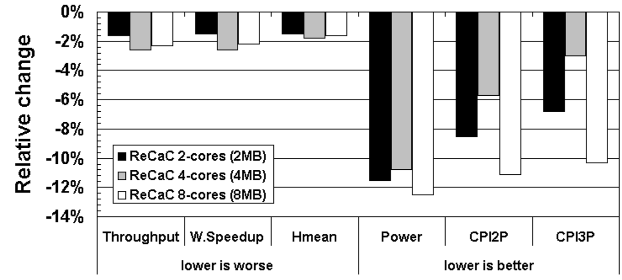


Figure 15: ReCaC scalability.

Off-chip access energy. The energy required to access off-chip data depends on the memory size and design technology. Figure 14 analyzes the power savings that ReCaC obtains when we vary *the memory access vs. the L2 cache access* energy ratio. When the energy cost of the off-chip access increases, the ReCaC finds less situations when it is worth to set power as an optimization goal, and achieves lower power savings with similar performance. For ratios smaller than 125 (the energy required to access main memory is 125 times higher than accessing L2 cache), the ReCaC uses more aggressive power optimizations, what causes slight increase in the average performance degradation for the high utility benchmarks.

Scalability. Figure 15 depicts the ReCaC results when we vary the number of cores. We observe that with 4 cores the ReCaC power savings drop to 10.8%, as an effect of a higher relative access pressure on each cache way - in all the cases we use 16-way L2 cache. The performance cost increases to 2.6% (throughput), since the ReCaC uses aggressive power optimization policy. However, as the number of cores further increases to 8 cores (8MB, 16-way L2 cache), performance decrease stabilizes (2.3% degradation in throughput) and power savings increase to 12.5%. Since in the current CMP increase in the number of cores is followed by bigger shared caches (with similar associativity), we believe that ReCaC proves to be a good candidate for future architectures.

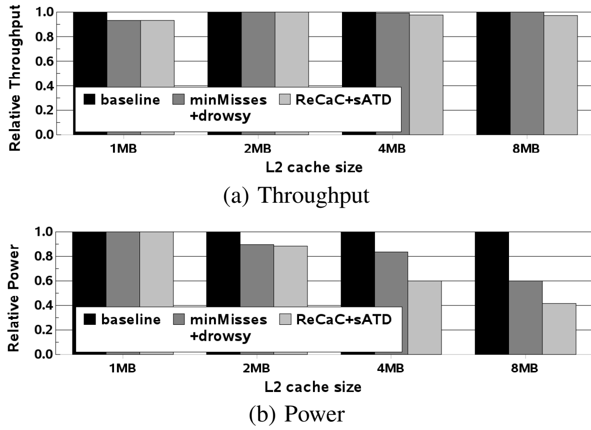
6.3 The Effect of the Power-Aware Cache Partitioning

Thus far we compared two architectures: 1) the **baseline** configuration with the minMisses [27] policy and the costly sampled ATD (sATD) profiling logic. This scheme focuses to improve the final performance. 2) The **ReCaC** uses a low cost embedded ATD (eATD), together with the low-power drowsy mode [14] applied to each L2 cache line. It seeks the power optimization, and in case it is not feasible, ReCaC reverts back to the performance-centric cache partitioning algorithm.

In this section we compare two architectures with the same profiling logic (sATD), applying the low power drowsy mode with the same wake up latency (1 clock cycle):

1) **minMisses+drowsy**: minMisses policy applied together with the low power drowsy mode. In this case, we set all the L2 cache lines to the drowsy mode on every sampling interval boundary. The minMisses algorithm optimizes for the highest performance.

2) **ReCaC+sATD**: ReCaC power and performance aware policy, guided by the separate sATD profiling logic instead of the eATD.



(a) Throughput

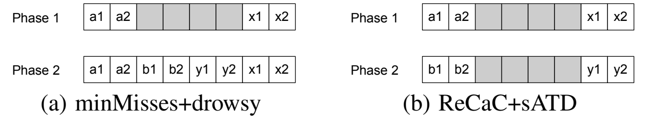
(b) Power

Figure 16: Relative power and throughput for the baseline (minMisses without drowsy), minMisses+drowsy and ReCaC+sATD architectures.

We analyze performance and power of both architectures varying the L2 cache size from 1MB to 8MB in Figure 16(a) with respect to the baseline. For all the cases we use 16-way L2 cache. Our results show that the throughput in both scenarios remains similar. For 1MB L2 cache both architectures that use drowsy mode, minMisses+drowsy and ReCaC+sATD, decrease throughput by up to 7%. It is the performance cost of the additional drowsy latencies, when each line in the low power mode needs to be first put to the normal mode before it may be accessed. We observe this effect on small caches, where the L2 cache access pressure is high. We achieve similar results with the harmonic mean and weighted speedup metrics.

Figure 16(b) plots the relative power of the evaluated architectures. We do not observe significant power reduction for 1MB L2 cache size augmented with the selective drowsy mode, due to the high cache access pressure that forces the lines to be awake. However, if the cache size increases to 2MB, ReCaC+sATD and minMisses+drowsy configurations reduce the power by 11.5% and 10.2%, respectively. When we increase the cache size to 4MB and 8MB, the ReCaC+sATD wins significant power savings over minMisses+drowsy architecture. For the 8MB L2 cache it saves 58.7%, whereas minMisses+drowsy saves 40.2% of the total power (in a 2-core CMP with 8MB L2 cache the L2 leakage becomes the dominant power component in the total system power). The minMisses+drowsy savings are smaller, since this scheme tends to wake all the lines across the LRU stack. As a result, we observe an interesting effect: the lines in the sets are accessed with the similar frequency, however, the size of the stack distance of a benchmark is smaller than the cache associativity. It causes the current working set of a given benchmark to *move* across the cache ways, unnecessarily waking up the lines from the low power mode.

We further discuss this phenomena in Figure 17. Let us assume that there are two threads, T0 and T1, accessing an 8-way L2 cache. The cache partitioning algorithm applies a new cache partition every 1 million clock cycles. Further, let us assume that both threads experience a phase change every 500 thousand clock cycles. The thread T0 uses the addresses {a1, a2} during the first 500 thousand clock cycles (phase 1) and {b1, b2} during the following 500 thousand clock cycles (phase 2). Similarly, the thread T1 has {x1, x2} as a working set during the first 500 thousand clock cycles (phase



(a) minMisses+drowsy

(b) ReCaC+sATD

Figure 17: Cache content at the end of each phase for minMisses and ReCaC cache partitioning algorithms. Shaded boxes correspond to cache lines in the low power drowsy mode.

1) and {y1, y2} during the following 500 thousand clock cycles (phase 2). Therefore, in each benchmark phase the stack distance for both threads equals 2, as the benchmarks use two lines in each phase. However, the minMisses policy [27] splits the entire cache among the running threads. The policy assigns 4 ways to the thread T0 and 4 ways to the thread T1, as Figure 17(a) shows. In case of the minMisses+drowsy architecture, whenever there is a phase change, the old working set is not removed from the cache. It is a property of the LRU replacement policy, where the data is removed from the cache only if it is degraded to the LRU position. On the contrary, as Figure 17(b) depicts, the ReCaC assigns only 2 ways to the thread T0 and 2 ways to the thread T1. On the phase change, the thread T0 already owns 2 lines in a set {a1 and a2}. Therefore, it replaces these lines with the {b1, b2}. Similarly, the thread T1 replaces lines {x1, x2} with {y1, y2}. As a result, the total amount of the lines in the low-power drowsy mode is higher than in case of the minMisses+drowsy policy.

7. RELATED WORK

Chiou, et al. [9], proposed one of the first static partitioning of the L2 cache using a profiling information. Suh, et al. [30], use a dynamic cache partitioning with Stack Distance Histogram. Finally, Qureshi, et al. [27], propose the cheapest so far solution in terms of hardware. Recently, new techniques have been proposed to improve performance over the LRU scheme [18, 25]. All these schemes, however, do not seek to optimize power when partitioning a cache.

Balasubramonian, et al. [4, 5], implement a cache and TLB with a variable latency in a single-core architecture. The approach improves performance and energy by dynamically balancing hit and miss latency intolerance during the program execution. We instead track the thread(s) miss rate and partition a cache with a fixed latency. In [4] algorithms detect benchmark phase changes. Dhodapkar, et al. [11], propose *working sets signatures* to limit complex and time consuming tuning process in reconfigurable caches. When a given *set signature* is recognized, the last optimal partition is applied. This solution, however, requires a significantly higher hardware cost with respect to the ReCaC.

Albonesi, et al. [3], and Dropsho, et al. [13], propose a dynamic extension of the static selective caches [2] for a single thread. The authors use a reconfigurable cache design called an *accounting cache*. The cache access protocol is extended, incorporating the primary and the secondary accesses. In our approach, we use less hardware demanding cache partitioning with negligible performance degradation in the CMP architectures. Yang, et al. [31], compare various design choices for resizable caches and propose a selective-sets-and-ways cache organization. The proposal, however, targets the L1 caches.

Kim, et al. [19], propose an improvement of the drowsy caches [14] that reduces the leakage power of the L1 caches using a dynamic

voltage scaling and a cache sub-bank prediction for the single core case. The size of the simplest predictor evaluated in the paper is 4KB, for 32KB direct-mapped L1 cache, 32-byte line size and a 40-bit address. The ReCaC proposes a significantly reduced hardware in the CMP architecture, with an overhead of tens of bytes.

8. CONCLUSIONS

This paper targets two design approaches. For the area-constrained architectures we propose the eATD, a novel and low-cost profiling logic for the L2 caches. Our results show that the area is reduced from tens of Kilo bytes to tens of bytes, with a negligible additional power consumption. The performance degradation is of 1.2%, 1.6% and 3.1% for 2-, 4- and 8-core CMP, respectively.

When power, instead of area, is the first order design issue, we propose the ReCaC: *Reconfigurable Cache for CMPs*. The ReCaC is a L2 cache extended with 1) low-overhead monitoring logic (eATD), 2) selective drowsy mode and 3) novel power and performance aware cache partitioning policy. The ReCaC is an OS guided framework that targets to use a minimum number of ways to trade-off power and performance. It dynamically adapts to the benchmark behavior. Further, it reverts back to the performance centric cache partitioning scheme in case the power savings are not possible for the threads that actively use the entire L2 cache. Our results show that in the 2-core architecture the ReCaC saves on average 51.5% power in L2 cache, which corresponds to 11.5% power savings of the processor chip and the memory. The impact on performance is 1.6% throughput reduction. The overall processor energy efficiency is improved by 13.6%. The ReCaC proves to be scalable, since it saves 10.8% and 12.5% overall power in the 4- and 8-core CMP, respectively. The energy-efficiency is improved on average by 11.1% in the 8-core architecture.

The ReCaC creates an easy to manage framework for the OS-guided Quality of Service, since the power-performance trade-off of each thread can be set individually. We leave the QoS research for our future work.

Acknowledgements

This work was supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625 and grant AP-2005-3776, by the HiPEAC Network of Excellence (IST-004408) and a Collaboration Agreement between IBM and BSC with funds from IBM Research and IBM Deep Computing organizations. The authors are grateful to Pradip Bose and Chen-Yong Cher from the IBM T. J. Watson Research Center, and the reviewers for their valuable comments.

9. REFERENCES

[1] http://www.specbench.org/.

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

[3] D. Albonesi, R. Balasubramonian, S.G.Dropsbo, S. Dwarkadas, F. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, and A. Buyuktosunoglu. Dynamically tuning processor resources with adaptive processing. *IEEE Computer, Special Issue on Power-Aware Computing*, 2003.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Transactions on Computers*, 2003.

[6] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20, 2000.

[7] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield. New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors. *IBM J. of Res. and Dev.*, 46(5/6), 2008.

[8] Y. Chen. Hundreds of cores: Scaling to tera-scale architecture. In *Intel Developer Forum*, 2006.

[9] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *DAC*, 2000.

[10] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO*, 2006.

[11] A. Dhodapkar and J. Smith. Managing multiconfiguration hardware via dynamic working set analysis. In *ISCA*, 2002.

[12] J. Donald and M. Martonosi. Power efficiency for variation-tolerant multicore processors. In *ISLPED*, 2006.

[13] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *PACT*, 2002.

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

[15] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE J. Solid-State Circuits*, 31, 1996.

[16] Z. Hu, D. Brooks, V. Zyuban, , and P. Bose. Microarchitecture-level power-performance simulators: Modeling, validation and impact on design. tutorial. In *MICRO*, 2003.

[17] W. Huang, M. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *DAC*, 2004.

[18] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for managing shared caches on CMPs. In *PACT*, 2008.

[19] N. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches. leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *MICRO*, 2002.

[20] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multi-core cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, 2008.

[21] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *ISPASS*, 2001.

[22] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. MLP-aware dynamic cache partitioning. In *HiPEAC*, 2008.

[23] M. Moudgill, J. Wellman, and J. H. Moreno. Environment for PowerPC microarchitecture exploration. In *IEEE Micro*, volume 19, 1999.

[24] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for high-performance caching. In *ISCA*, 2007.

[25] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Set-dueling controlled adaptive insertion for high-performance caching. In *IEEE MICRO*, 2008.

[26] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, 2006.

[27] 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.

[28] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *ISCA*, 2003.

[29] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

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

[31] S.-H. Yang, Michael D. Powell, B. Falsafi, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA*, 2002.