Scalable Working Set Estimation Method For Chip Multicores Using Tagged Bloom Filter And Its Application

Aparna Mandke Bharadwaj Amrutur Y. N. Srikant

IISc-CSA-TR-2011-3
http://archive.csa.iisc.ernet.in/TR/2011/3/

Computer Science and Automation
Indian Institute of Science, India

September 2011

Abstract

In chip multicore platforms (CMPs), leakage power consumption of large on-chip caches has already become a major power consuming component of the memory subsystem. Leakage power can be saved by switching off over-allocated ways in associative cache. However, the state-of-the-art heuristics such as average memory latency or cache miss rate fail to achieve near-optimal energy savings. This is either due to dispersed nature of large caches or they are not fast enough to respond to changes in working set size (WSS), especially in case of over-provisioning of cache. Hence, we first propose a new kind of bloom filter, which we call it as a "tagged bloom filter (TBF)". We implement TBF implicitly in last level cache on a scalable tiled chip multicore platform. TBF is then used to estimate WSS of an application and switch-off over-allocated cache ways in Static and Dynamic Nonuniform Cache Architecture (SNUCA, DNUCA) accordingly. In our implementation of adaptable way SNUCA and DNUCA caches, associativity decision is taken locally by each L2 controller, making it scalable with the number of tiles present on CMP. It gives average of 22% and 23% more EDP savings than average memory latency and cache miss rate heuristics on SNUCA, respectively.

1 Introduction

Due to advances in technology, the number of cores and on-chip cache size on CMPs has increased. As a result, leakage power consumption of on-chip cache has already become a major contributing component of the memory subsystem. Leakage power consumption of cache can be reduced by switching-off cache ways of an associative cache. Power gains obtained by switching off over-allocated cache depends on the accuracy of WSS estimation, how fast heuristic can respond to changes in WSS and overhead incurred by the method. Power gains will be sub-optimal if WSS is estimated conservatively. On the contrary, application will incur more cache misses if WSS is under-estimated, and execution time of an application may degrade significantly if heuristic fails to respond to changes in WSS quickly. Hence, in this paper, we attempt to solve WSS estimation problem in the context of applications running on a scalable tiled CMPs with large distributed NUCA caches. Previous attempts either estimate WSS for applications running on uniprocessor [8] or use some other heuristic as an indirect estimation of WSS [21, 4]. In this paper, we present a novel accurate WSS estimation method which has negligible hardware. It can be applied to partition cache on virtualized systems[13] or to save cache power by switching-off cache ways [4, 3]. Here, we demonstrate application of WSS estimation method to switch-off cache ways of NUCA.

For power and performance reasons, a large cache is partitioned into multiple banks which are connected using a switched-network[12]. The predetermined bits from a memory address determine the location of “home L2 bank”, where data is cached. Such cache offers non-uniform access latency to various cores on CMP. Hence, the architecture is referred to as *Static Non-Uniform Cache Architecture (SNUCA)*. Kim et al. [12] proposed dynamic NUCA (DNUCA) access policy to reduce access latency incurred due to distributed cache. In DNUCA, the whole address space is mapped onto a column. The predetermined bits from a memory address, determine the row in which data is cached. All L2 banks in a row form a bankset and data can be cached in any of these L2 slices. On L1 miss, data is first searched in the nearest L2 bank and then in rest of L2 banks in that row before reading it into the nearest L2 bank from memory. Data is gradually migrated towards a nearer bank on consecutive accesses.

1.1 Motivation

Since cores incur variable latencies due to dispersed nature of NUCA, typically used heuristic such as average memory latency[10] is inappropriate to predict cache requirement. Hence, Bardine et al.[4] used ratio of number of hits to the farthest cache way to the nearest cache way in DNUCA to predict cache requirement. As explained earlier, in DNUCA, less frequently used cache lines subsequently migrate to farther cache ways. They primarily studied DNUCA cache with one or two cores in which case, both the cores are on the same side of the shared bus and cache is on its other side. This heuristic works well for smaller CMPs where cache and cores are on either side of interconnect. It *fails* to scale for CMPs with many cores. Similarly, other typically used heuristic, such as, cache miss rate[21] is not fast enough to respond to changes in WSS. These heuristics require interconnect access to take cache associativity decision, which may cause additional delay and traffic on the interconnect. The main drawback of these heuristics is that their values are compared against values evaluated in the *previous* time slot, and not against their corresponding values if the whole cache is allocated. Hence, we propose a method to estimate WSS of an application(s) executing on a large scalable CMP.

Considering trend of increasing number of cores, we study a tiled CMP such as in Tilera processors (Fig. 1). In this architecture, tiles are replicated and connected through an on-chip switched-network (NoC). Each tile has a core, a private L1 instruction and data cache, a slice of L2 cache and a

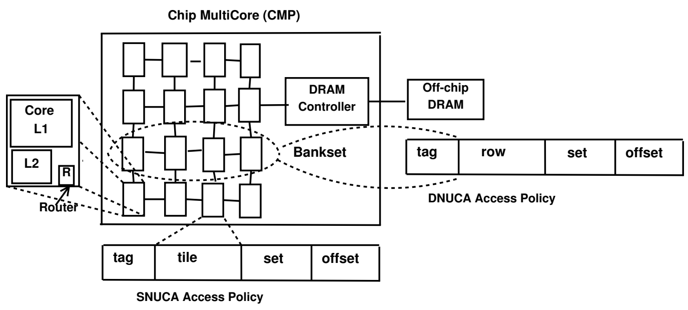


Figure 1: Tiled CMP used for experimentation

router. In our implementation, L2 cache is distributed across all tiles and it is shared by all cores. To maintain cache coherence between private L1 caches, a directory is present in each tile. These tiles are inter-connected via 2D-mesh NoC and per-tile router. Fig. 1 also shows that, *tile* bits from memory address determine L2 slice where data is cached in SNUCA and *row* bits determine the row in which data is cached in DNUCA.

1.2 Our Contributions

A new type of bloom filter called as "tagged bloom filter" implemented implicitly in cache and its use to accurately estimate WSS of an application(s), executing on large CMP. It has negligible hardware overhead of 0.1%.

A scalable implementation of SNUCA and DNUCA with adaptive cache associativity. Each L2 cache controller estimates cache usage of its L2 slice independently without accessing NoC and adjusts cache associativity accordingly. This makes it scalable with increasing number of tiles. We refer these implementations as SNUCA.TBF and DNUCA.TBF in rest of the paper.

DNUCA.TBF achieves 40% higher EDP[1] savings compared to DNUCA. Similarly, SNUCA.TBF obtains 45% higher EDP savings compared to SNUCA. We review related work in section 2 and describe TBF WSS estimation method in section 3. The scalable algorithm to evaluate associativity of L2 cache is explained in section 4. Experimental setup and results are presented in Section 5 and section 6, respectively. Section 7 concludes the paper.

[1]Energy-Delay product (EDP) represents a trade-off between energy consumed by an application and its execution time.

2 Related Work

WSS Estimation Methods: Dhodapkar et al. [8] proposed to estimate WSS signature of an application, executing on uniprocessor, by setting a bit obtained by hashing a cache line address into a bit vector of 1024 bits maintained in the core. Their approach estimates WSS accurately up-to size of few tens to hundreds of kilobytes which is not sufficient for applications executing on CMP. Their approach under-estimates large WSS due to aliasing problem. Moreover, its efficiency depends on the hash function and the *empirical* proportionality constant which estimates WSS from the number of ones set in the bit vector. TBF WSS estimation method overcomes drawback by maintaining unique indentifier (tag) for an address. R. Koller et al.[13] proposed a cacheGrabber which increases cache utilization using non-intrusive cache prefetch instructions till a performance drop in the simultaneously executing application and cacheGrabber is observed. This method is used to estimate cache miss rate (CMR) curves. CMR works very well to partition cache among applications executing on a cache constrained platform. However, our experiments show that CMR is poorly correlated to WSS on platforms, where cache is not a constraint. Yang et al. [21] use CMR to selectively use cache associativity or the number of cache sets at run-time. D. Albonesi [3] use offline profiled information to adjust cache associativity, which cannot respond to run-time changes in cache requirement. [10] uses average memory latency to adjust the number of cores sharing L2 cache.

Power Optmized Cache Architectures: Decay cache[11] estimates *dead time* of a cache line and switches it off after its estimated last use. It operates at granularity of a cache line and has very high hardware overhead. Flautner et al.[9] proposed *drowsy cache* in which less frequently used cache lines are put into drowsy mode. In drowsy mode, supply voltage of a cache line is reduced to dissipate lesser leakage power without losing data. The drowsy cache technique achieves large energy gains with negligible performance degradation. However, decreasing the supply voltage makes cache lines susceptible to soft-error faults[7]. D. Albonesi[3] proposed to disable over-allocated ways of a set-associative cache to reduce its *dynamic* power consumption. Bardine et. al [4] vary associativity of DNUCA cache to reduce its *static* power consumption on a one or dual core platform.

3 TBF Working Set Size Estimation

We define working set of an application as the unique L1/primary cache lines accessed in a monitoring period. Cardinality of this set is working set size

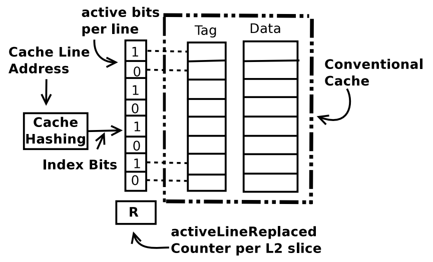


Figure 2: Tagged bloom filter working set estimation method

(WSS) or to be more precise, *cache working set size.* Formally, if $l_1, l_2, l_3....l_n$ are unique cache line addresses (CLAs) accessed by an application, then the working set S is,

$$S = \{l_1, l_2, l_3....l_n\} \qquad WSS = |S| = n \tag{1}$$

In bloom filters[6], element to be inserted in a set, is hashed and corresp-nding bit is set in a bit-vector. The number of bits set indicates the number of elements inserted in the set. However, aliasing causes under-estimation of the number of elements in the set. To overcome aliasing problem to some extent, we propose to maintain unique tags to correctly identify an element already inserted in the set. We refer this type of bloom filter as tagged bloom filter. It can be implemented *implicitly* in caches by maintaining an active bit per cache line and a replacement counter per L2 slice. Tag bits of a cache line form tags in the TBF. When an address is accessed in cache, tag bits are compared. If it is a cache hit, then its active bit is set. However, if it a miss and active bit is set, then cache replaces the existing address and loads a new address in its place. Tag bits of that address are updated automatically by the cache controller. However, in this case, to track the previous accessed address, we increment a replacement counter, called as activeLineReplaced counter. This counter counts the number cache lines replaced with active bit set. With TBF method, WSS is calculated as:

$$WSS = A + R \tag{2}$$

where, A is the number of cache lines with active bit set and R is the number of cache lines replaced with active bit set, i.e. value of the activeLineReplaced counter. TBF overcomes Dhodapkar's method's[8] drawback of under-estimating WSS in following ways:

- Tags maintained along with cache line avoid aliasing problem by identifying cache line addresses uniquely.

- The replacement counter counts lines replaced with active bit set.

TBF over-estimates WSS if the same cache line is accessed and replaced multiple times in the same monitoring period. However, this scenario is rare considering large cache sizes and large cache associativity (16-32) of last level caches (LLCs). Aggregate WSS of all threads executing on CMP can be obtained by implementing TBF in the shared LLC. And WSS of an individual thread can be obtained by implementing TBF in the primary cache.

For cache of 512KB in size and 64B cache line, 8K active bits are required which is just 0.1% overhead. Our experimental analysis shows that a 32 bit saturating counter is sufficient for our monitoring period, as we reset active bits and activeLineReplaced counter at the beginning of every monitoring period. *TBF adds minimal overhead in terms of hardware area and power consumption.*

3.1 Determination of monitoring period

If the monitoring period is too small, fewer active bits would be set. This would be misinterpreted as lesser WSS. On the contrary, if the monitoring period is too high, it would estimate higher WSS, failing to switch-off excess L2 slices. Hence, determination of accurate monitoring period is very important. To estimate it, we measured the number of clock cycles between two consecutive accesses made to the same address by any thread in an application. We call it as *reuse time.* For all applications that we consider, an average of 96% accesses have reuse time less than 2M cycles. If a cache line is not accessed in the last 2M clock cycles then most likely, it will not be accessed in next 2M clock cycles as well and hence it is not considered in working set. Conservatively, we use monitoring period of 4M clock cycles.

4 Variable Way SNUCA and DNUCA

NoCs on CMP have become more complex and add significant latency. Hence algorithm used to vary associativity of cache should use locally available information so as to avoid additional delays and traffic on NoC. The L2 cache controller in each tile executes Algorithm 1 to evaluate associativity of its L2 slice independently without accessing NoC. Because of this our algorithm scales with the number of tiles present on CMP. Unlike average memory access latency (AAL) and CMR heuristics, cache associativity of each L2 slice is determined according to its usage. This enables to achieve higher energy savings with minimal performance degradation.

Algorithm 1 Evaluate associativity of each L2 slice

```
1: Get the number of active Lines(A) of an L2 slice
2: Get the number of activeLineReplaced(R) of an L2 slice
3: $EWSS = A + R$
4: $bank\_size = total\_cache\_size/max\_associativity$
5: $assoc_1 = ceil(EWSS/bank\_size) + 1$
6: $r = ceil(R/total\_number\_of\_sets\_in\_a\_tile)$
7: if $r > T_1$ && $assoc_1 < (current\_associativity + 2)$ then
8:     $assoc_1 = current\_associativity + 2$
9: end if
10: $assoc = max(max\_associativity, assoc_1)$
```

The number of active bits set (A) and the number of active lines replaced (R) in the previous monitoring period are used to estimate WSS ($EWSS$)(lines 1-3). Line 5 evaluates associativity of the L2 slice. Minimum associativity of 2 is assigned to avoid frequent conflict misses. The number of active cache line replacements per set is calculated as R divided by the total number of sets in the L2 slice (line 6). Here, we assume that the active line replacements are distributed uniformly over all the sets. If the number of replacements per set is greater than the threshold T_1, then associativity is increased by 2. Higher value of T_1, will reduce associativity aggressively, causing more cache misses and DRAM accesses. We conservatively set $T_1 = 0$ in all our experiments i.e we increase associativity even if there is one active line replacement per set.

When associativity is decreased (increased), extra associativity of all sets in the L2 slice is switched off (on). The number of ways to be switched off (on) is equal to the difference between the new associativity and associativity in the current time slot. Modified cache lines in the banks to be switched off, are written back and clean lines are invalidated. Apart from determining WSS, active bit serve another purpose. Cache lines with active bit set, are migrated in cache banks that would remain powered on and some other cache line from that bank is replaced instead.

5 Experimental Configuration

5.1 Applications used in Experiments

We evaluate multi-threaded workloads with one-to-one mapping between threads and cores (Table 1)[2]. This assumption is in line with other work

[2]Rest of the PARSEC benchmarks either use OpenMP APIs or libraries which are not supported by SESC compiler. Hence remaining benchmarks cannot be compiled using SESC compiler.

Name	Description, WSS(L/M/S)
Alpbench Benchmark [14]	
MPGEnc	Encodes 15 Frames of size 640x336, M
MPGDec	Decodes 15 Frames of size 640x336, M
Splash2 Benchmark[20]	
Cholesky	blocked sparse matrix factorization on tk29, L
FFT	FFT on 1M points, M
LU (noncontinuous)	1024x1024 LU matrix factorization, S
Radix	Radix sort on 1M keys, M
FMM	simulate interaction of 16K bodies system, M
Water_spatial	simulation of 512 water molecules, M
Water_nsquared	simulation of 512 water molecules, M
Barnes	Barnes-Hut method on 16K bodies, M
Ocean (continuous)	512x512 grid points, L
PARSEC Benchmark[5]	
Blackscholes	SimLarge i/p, Financial Domain, S
Swaptions	SimLarge i/p, Financial Domain, M
Fluidanimate	SimMedium i/p, Animation, L
H.264 Encoder	SimLarge i/p, Media Domain, M

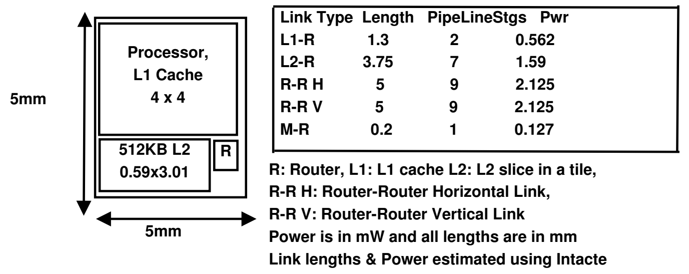
Table 1: Table shows applications used for study and their WSS information(L:Large, M:Medium, S:Small).

done for CMP platforms. We have skipped initial serial portion and simulate only parallel section in all the test cases. We execute 1B instructions, unless otherwise mentioned. We test all workloads with 16 threads.

5.2 Experimental Setup And Methodology

We model all the system components with reasonable accuracy in our framework. We use SESC[18] to simulate a core, Ruby component from GEMS[15] to simulate the cache hierarchy and interconnects. DRAMSim is used to model the offchip DRAM. DRAMSim[19] uses MICRON [2] power model to estimate power consumed in DRAM accesses. Intacte [17] is used to estimate low level parameters of the interconnect such as the number of repeaters, wire width, wire length, degree of pipelining and power consumed by the interconnect. Power consumed by the cache components is estimated using CACTI 6.0.

In order to estimate the latency (in cycles) of a certain wire, we estimate



Link Type	Length	PipeLineStgs	Pwr
L1-R	1.3	2	0.562
L2-R	3.75	7	1.59
R-R H	5	9	2.125
R-R V	5	9	2.125
M-R	0.2	1	0.127

R: Router, L1: L1 cache L2: L2 slice in a tile,
R-R H: Router-Router Horizontal Link,
R-R V: Router-Router Vertical Link
Power is in mW and all lengths are in mm
Link lengths & Power estimated using Intacte

Figure 3: Floorplan of a tile with 512KB L2 slice.

area of all components in a tile and then create the floorplan which is shown in Fig. 3. We make following assumptions to determine area of various components at 32nm technology and 3GHz frequency:

- core : This is estimated based on the area of Intel Nehalem core[1]
- cache : The L1 cache is of size 32KB whose area is very small and is included in the processor area. The area occupied by the L2 cache is obtained using CACTI 6.0. We assume directory information is stored along with each L2 slice. We conservatively assume area of per-tile directory to be negligible. If directory area is considered then interconnect lengths will increase which is more beneficial for the remap policy.
- router : The area of the router is assumed to be quite negligible at 32nm.

Fig. 3 also shows wire lengths and their power consumption. The latency of a link in clock cycles is equal to the number of its pipeline stages. To obtain power consumption of NoC, we compute the link activity and coupling factors of all links, caused due to the messages sent over NoC.

5.3 Simulation Procedure

Table 2 gives the system configuration used in our experiments. The flow chart in Fig. 4 shows the experimental procedure. It includes computing the area of tile components, computing link lengths and low level link parameters using Intacte and then performing simulation. Our simulator estimates the activity and coupling factors of all the links. Intacte determines power dissipated in NoC using these activity factors. Power consumed by the off-chip DRAM and on-chip cache is estimated using DRAMSim (MICRON) and CACTI power models, respectively.

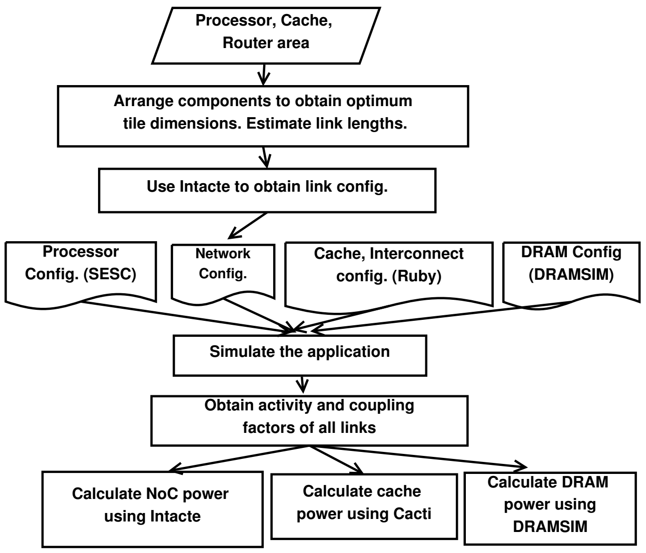

Processor, Cache, Router area
Arrange components to obtain optimum tile dimensions. Estimate link lengths.
Use Intacte to obtain link config.
Processor Config. (SESC)
Network Config.
Cache, Interconnect config. (Ruby)
DRAM Config (DRAMSIM)
Simulate the application
Obtain activity and coupling factors of all links
Calculate NoC power using Intacte
Calculate cache power using Cacti
Calculate DRAM power using DRAMSIM

Figure 4: **Experimental Procedure**

Core	out-of-order execution, 3GHz frequency, issue/fetch/retire width of 4
L1 Cache	32KB, 2 way, 64 bytes cache line size, access latency of 2 cycles (estimated using CACTI[16]), private, cache coherence using MOESI protocol
L2 Cache	512KB/tile, 16 way, 64B line size, 4 subbanks per slice, 3 cy. latency (estimated using CACTI), noninclusive, shared and distributed across all tiles
Directory	Tag bits of L2 cache line include full bitmap for L1 sharers. A separate table of 3000 entries maintains dir info. for cache lines not cached in L2 but only in L1s.
Interconnect	16 bits flit size, 4x4 2D MESH, deterministic routing, 4 virtual channels/port, credit based flow control, router queues with length of 10 buffers
Off-chip DRAM	4GB, DDR2, 667MHz freq, 2 channels of 8B in width, 8 banks 16K rows, 1K columns, close page row management policy

Table 2: System configuration used in experiments

6 Results

6.1 Accuracy of TBF

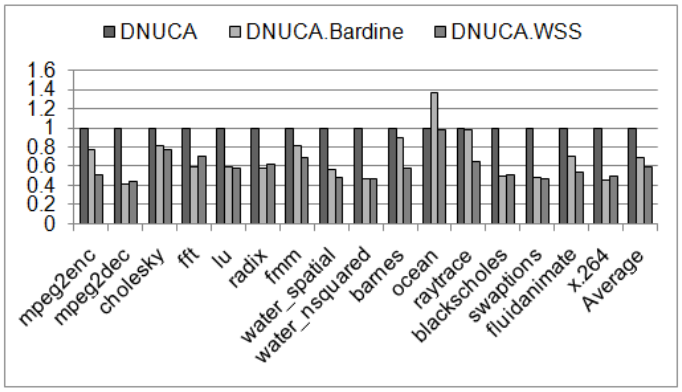
To determine accuracy of TBF, we determine *actual* WSS (AWSS) by counting unique number of CLAs accessed by all cores. First column shows cor-

App.	Correlation to AWSS			EWSS/ AWSS
	TBF	Avg. Mem. Latency	MissRatio	
mpegenc	0.98	-0.16	0.03	1.15
mpegdec	0.96	-0.37	-0.55	1.12
cholesky	0.95	0.37	0.24	1.28
fft	0.99	0.64	0.73	1.85
lu (noncontinuous)	0.95	-0.65	0.96	1.23
radix	0.99	-0.92	0.68	1.69
fmm	0.98	0.33	0.72	1.08
water_spatial	0.99	0.52	0.17	0.96
water_nsquared	0.99	0.35	0.44	0.98
barnes	0.99	0.07	0.27	1.04
ocean (continuous)	0.99	-0.69	-0.47	1.7
raytrace	0.99	-0.81	-0.85	1.55
blackscholes	0.99	-0.47	0.04	0.96
swaptions	0.76	0.27	0.27	0.91
fluidanimate	0.96	0.64	0.77	1.41
x.264 Encoder	0.98	0.46	0.4	0.83
average	0.97	-0.03	0.24	1.23

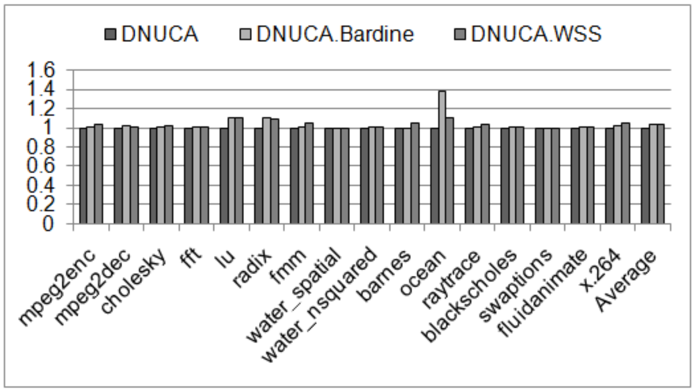
Table 3: Table shows correlation of various hueristics to actual WSS (AWSS). It also shows ratio of estimated WSS and actual WSS (AWSS).

relation between WSS estimated by TBF and AWSS determined after every 4M clock cycles. For all applications, two values show good correlation. Table 3 also shows correlation between average memory access latency (AAL) and AWSS determined after every 4M clock cycles. These two values are very poorly correlated, which is quite intuitive. We determine NoC link latencies using Intacte and use very realistic values of latencies. Hence, AAL also depends on time spent in NoC traversal, which makes it not suitable for larger NUCA caches. Cache miss ratio (CMR) also shows poor correlation to WSS, which is not intuitive. We measure *aggregate* AWSS at L1 caches. Majority of the accesses are L1 hits. Hence, higher AWSS does not imply more L2 cache misses due over-provisioning of L2. In this case, on reducing L2 associativity aggressively will invalidate active cache lines from L1 caches, degrading execution time of an application. Whereas, TBF sets active bit in case of L1 replacements, changes in sharers' list or change in access permission, which enables it to estimate WSS accurately.

Table 3 also shows ratio of average WSS estimated by TBF to average of AWSS. In majority of applications TBF is greater than AWSS. However, over-estimation is not huge, as the ratio is close to 1. Over-estimation in applications like fft is due to uneven use of sets. Some sets show more cache line replacements than rest. Thus the same cache line gets accessed and replaced multiple times in a monitoring period, causing re-counting by the replacement counter. When an L2 cache line with L1 sharers is replaced, replacement counter is incremented, even if its active bit is not set. This



(a) Smaller EDP value is better



(b) Smaller Execution Time is better

Figure 5: Quantitative comparison of our variable way DNUCA implementation with Bardine's hueristic

avoids unnecessary L1 cache invalidations. x.264 shows under-estimation of WSS. This is because, in x264, most of accesses are L1 cache hits during when L2 cache line is not accessed. On an average, the ratio of estimated WSS to actual WSS is 1.23.

6.2 Comparison with Bardine's hueristic

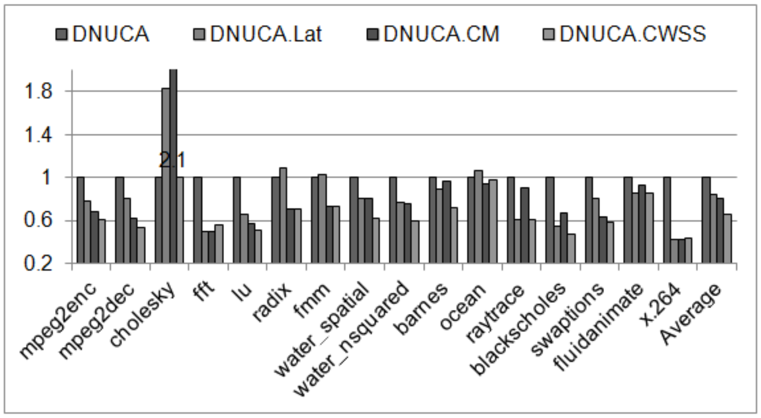
As explained earlier, Bardine et al. [4] use ratio of number of hits to the farthest cache line to hits to nearest cache line as a heuristic to change cache associativity in DNUCA. We refer to this ratio as *hit ratio.* This heuristic works in case of DNUCA as frequently used cache lines gradually move towards cores. However, in tiled architecture cache lines nearer to one core are far for other cores. *Hence, Bardine's method can be applied only up to* *four threads scheduled on cores in the first column in Fig. 1.* We compare DNUCA.TBF to Bardine's implementation by executing workloads with four threads scheduled in the first column. Fig. 5(a) compares EDP savings obtained with TBF to that obtained with Bardine's heuristic. DNUCA.TBF gives average of 9.5% higher EDP savings than DNUCA.Bardine. This is because cache associativity is gradually adjusted in Bardine's method, whereas, TBF estimates WSS and adjust cache associativity to the correct value immediately from the next monitoring period. Accurate estimation of WSS, enables to achieve higher EDP savings with negligible performance degradation, except in case of ocean. Ocean has smaller WSS initially and it increases suddenly after some time. In Bardine's method, associativity is reduced if hit ratio is smaller than that obtained in previous monitoring slot[3]. As associativity reduces to very small value of four, hit ratio does not increase on increase in WSS. Hence, Bardine's method fails to allocate more cache, degrading execution time by 37%. On the contrary, our method sets active bit when cache line migrates to farther slice and replacement counter counts the number of replaced active cache lines. This indicates sudden increase in WSS and our method allocates more cache lines, giving maximum execution time degradation of 9.7% in ocean. However, in all rest of applications is lesser than 5% 5(b). In this implementation, to apply Bardine's hueristic we migrate L2 cache lines to farther L2 slices on replacements. This causes counting of active bits multiple times in a monitoring slot. This introduces inaccuracies in TBF WSS estimation. However, in tiled architecture, a cache line is sent to offchip DRAM on replacement. Hence, very negligible performance impact is seen in DNUCA.TBF and SNUCA.TBF on tiled architecture. Apart from giving average of 9.5% higher EDP savings than DNUCA.Bardine with negligible performance degradation, DNUCA.TBF is scalable with the number of cores, unlike DNUCA.Bardine.

6.3 Scalability of CWSS method

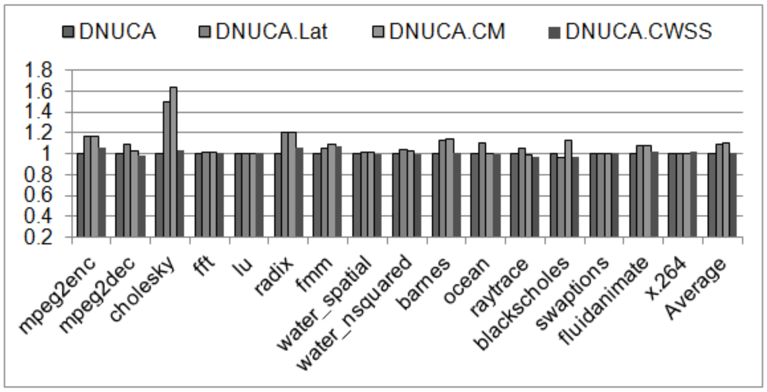
To demonstrate scalability of TBF, we execute applications with sixteen threads on a sixteen tiled CMP with DNUCA. We compare DNUCA.TBF quantitatively against EDP savings obtained with AAL and CMR as heuristics.

DNUCA.Lat, DNUCA.CM: In these configurations, AAL and CMR are used as a heuristic to vary cache associativity. We calculate these values by aggregating accesses made by all threads. The cache associativity of all L2 slices is reduced if AAL/CMR is lesser than that of the previous time slot

[3]Threshold values are chosen as per [4]



(a) Smaller EDP value is better



(b) Smaller Execution Time is better

Figure 6: Graphs in 6(a) and 6(b) show normalized EDP and Execution time obtained with various power efficient DNUCA configurations

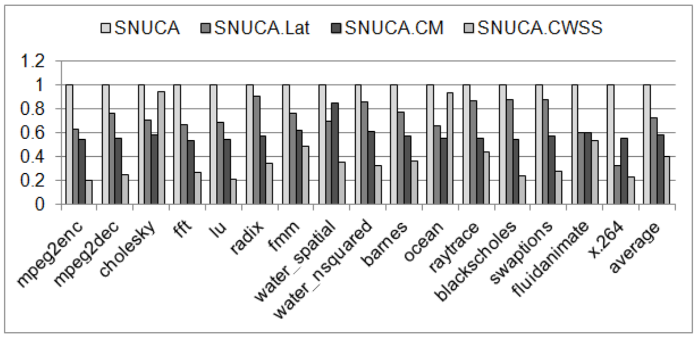


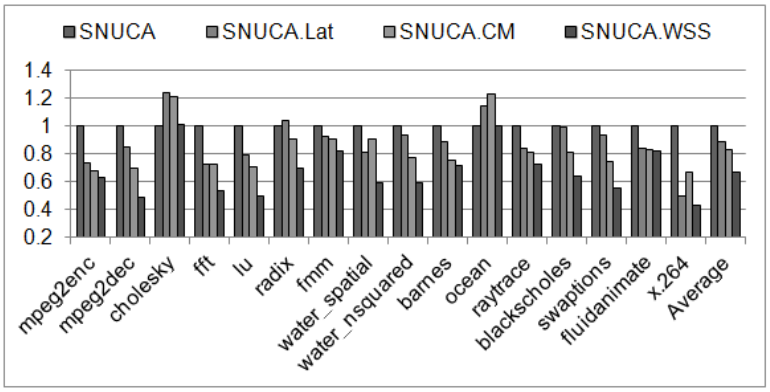
Figure 7: Average L2 slice usage

by 10%. The associativity of all L2 slices is increased if values are greater by 10% than that in the previous slot, else, associativity remains same. It should be noted that NoC access is essential to evaluate AAL or CMR. In these configurations, decision is taken by one L2 controller and then conveyed to rest of the controllers which makes these heuristics *unscalable*. Apart from this, all L2 slices use the *same* cache associativity, irrespective of their usage. **DNUCA.TBF**: It executes TBF estimation method after every 4M clock cycles and with threshold T_1 (see Algorithm 1) set to zero.

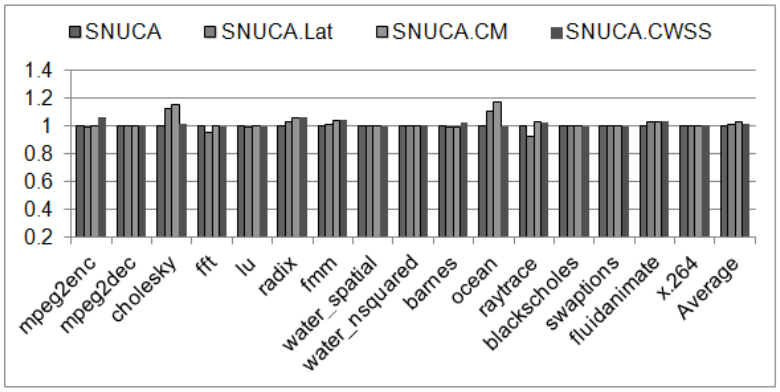
Fig. 6(a) plots EDP normalized with respect to (w.r.t.) that obtained with reference DNUCA. DNUCA.TBF estimates cache usage accurately of each L2 slice and assigns associativity accordingly. As a result, on an average it achieves EDP savings of 35% over the reference. On the contrary, DNUCA.Lat and DNUCA.CM achieve only average of 15% and 19% EDP savings over DNUCA. As shown in section 6.1, AAL and CMR show very poor correlation to WSS, especially when WSS of an application is much smaller than cache size. EDP savings obtained depend on how quickly heuristic can respond to changes in WSS. Unlike TBF, CMR and AAL estimate WSS indirectly. Hence, in case of cholesky, CMR and AAL show 63% and 50% degradation in execution time (Fig. 6(b)). This degrades EDP by 210% and 182%. Due to initial smaller WSS, cache associativity is decreased. On increase in WSS, cache allocation is slowly increased in the following monitoring periods. CMR/AAL is compared against values obtained in previous slots and not against corresponding values in the reference execution with the whole cache allocated. *Hence, these heuristics fail to quickly respond to cache needs of an application. Whereas, TBF measures the number of cache lines accessed in a monitoring period, due to which it responds quickly to application's cache needs.* Hence, DNUCA.TBF shows execution time degradation of only 3.3% in cholesky. On average, AAL and CMR show 8.8% and 9.6% degradation in execution time. On the contrary, TBF shows less than 1% degradation.

6.4 Applicability of CWSS to SNUCA

TBF can also be applied to SNUCA. Fig. 8(a) and Fig. 8(b) compare EDP savings and execution time obtained with TBF against AAL and CMR heuristics. SNUCA.TBF shows 21% and 16% higher EDP savings than that obtained with CMR and AAL heuristics. Large EDP savings in TBF are due to lesser number of L2 slices allocated by TBF. Fig. 7 shows average number of L2 slices allocated by each of these hueristics. TBF allocates average of 33% and 19% lesser number of L2 slices than that allocated by AAL and CMR hueristics. Despite lower usage of L2 slices, TBF does not show



(a) Smaller EDP value is better



(b) Smaller Execution Time is better

Figure 8: Graphs in 8(a) and 8(b) show normalized EDP and Execution time obtained with various implementations of power efficient SNUCA

performance degradation. In case of cholesky, AAL and CMR show 18% and 20% degradation in execution time. Thus showing 22% and 20% degradation in EDP, whereas, TBF accurately estimates large cache requirement and allocates the whole cache. The maximum execution time degradation shown by TBF is of 5% in radix. This is becase of sudden increase in WSS. TBF increases cache associativity in the next monitoring period. Even if execution time degrades by 5%, TBF gives 54% EDP saving.

6.5 Sensitivity to Monitoring Period

Fig. 9 shows normalized EDP gains for monitoring period of 3.5, 4 and 4.5 million cycles. CWSS is not very sensitive to the monitoring period. Therefore, SNUCA.Way and DNUCA.Way achieve approximately same EDP

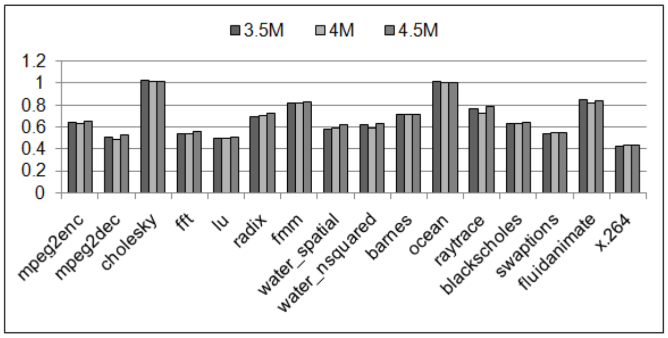


Figure 9: Sensitivity to monitoring period

savings for all applications on varying monitoring period.

7 Conclusions

We propose a new kind of bloom filter called as a "tagged bloom filter". We implement it implicitly in cache with negligible hardware overhead of 0.1% of cache size. Unlike previous approaches, it measures large or small WSS accurately. We use estimated WSS to switch-off over-allocated associativity of SNUCA and DNUCA cache, which is shared and distributed on a tiled CMP. Each L2 cache controller determines cache associativity of its L2 slice, with locally available information. Apart from avoiding additional delays and traffic on NoC, this method gives finer and independent control over associativity of each L2 slice. Unlike previous implementations of adaptive way DNUCA[4], our implementation is scalable with the number of cores present on CMP and achieves higher EDP savings. Due to accurate estimation of WSS, SNUCA.TBF and DNUCA.TBF achieve 54% and 35% EDP savings over their reference SNUCA and DNUCA platforms, respectively, with negligible degradation in execution time.

Unlike average memory access latency and cache miss rate, our method responds to WSS changes quickly and achieves 23% and 22% higher EDP savings on SNUCA than that obtained with AAL and CMR as heuristics.

References

[1] Intel Nehalem.

[2] Micron DRAM power data sheet.

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

[4] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete. Way adaptable DNUCA caches. *Int. J. High Perform. Syst. Archit.*, 2010.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 1970.

[7] B. Calhoun and A. Chandrakasan. Static noise margin variation for subthreshold sram 65-nm CMOS. In *IEEE Journal on Solid State Circuits*, 2006.

[8] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, 2002.

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

[10] M. Hammoud, S. Cho, and R. Melhem. Dynamic cache clustering for chip multiprocessors. In *ICS*, 2009.

[11] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*, 2001.

[12] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*, 2002.

[13] R. Koller, A. Verma, and R. Rangaswami. Estimating application cache requirement for provisioning caches in virtualized systems. In *MASCOTS*, 2011.

[14] M. lap Li, R. Sasanka, S. V. Adve, Y. kuang Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *IEEE ISWC*, 2005.

[15] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacets general execution-driven multiprocessor simulator (gems) toolset. 2005.

[16] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A tool to model large caches. 2009.

[17] R. Nagpal, A. Madan, A. Bhardwaj, and Y. N. Srikant. Intacte: an interconnect area, delay, and energy estimation tool for microarchitectural explorations. In *CASES*, 2007.

[18] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, 2005. http://sesc.sourceforge.net.

[19] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. 2005.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[21] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. HPCA '02.