Towards a Scalable Working Set Size Estimation Method And Its Application For Chip Multiprocessors

Aparna Mandke Dani, Bharadwaj Amrutur, Y. N. Srikant
aparnam@csa.iisc.ernet.in, amrutur@ece.iisc.ernet.in, srikant@csa.iisc.ernet.in
Indian Institute of Science, Bangalore

Abstract—It is essential to accurately estimate working set size (WSS) of an application for various optimizations such as, to partition cache among virtual machines or to reduce leakage power dissipated in an over-allocated cache by switching it off. However, the state-of-the-art heuristics such as average memory access latency (AMAL) or cache miss ratio (CMR) are poorly correlated to the WSS of an application due to 1) over-sized caches and 2) their dispersed nature. Past studies focus on estimating WSS of an application executing on a uniprocessor platform. Estimating the same for a chip multiprocessor (CMP) with a large dispersed cache is challenging due to the presence of concurrently executing threads/processes. Hence, we propose a scalable, highly accurate method to estimate WSS of an application. We call this method "tagged working set size (TWSS)" estimation method. We demonstrate the use of TWSS to switch-off the over-allocated cache ways in Static and Dynamic Non-Uniform Cache Architectures (SNUCA, DNUCA) on a tiled CMP. In our implementation of adaptable way SNUCA and DNUCA caches, decision of altering associativity is taken by each L2 controller. Hence, this approach scales better with the number of cores present on a CMP. It gives overall (geometric mean) 26% and 19% higher energy-delay product savings compared to AMAL and CMR heuristics on SNUCA, respectively.

Index Terms—Chip multiprocessors, working set size estimation, variable cache associativity

1 Introduction

Due to advances in technology, the number of cores and on-chip cache size on CMPs have been increasing. AMD Opteron 6200 series [1] has up-to 16 cores on a single die with 1MB per core L2 cache and 16MB shared on-chip L3 cache. Intel Nehalem [2] has 8 cores with 24MB on-chip cache. As a result, power consumed by the on-chip cache has already become a major contributing component of the power dissipated in the memory subsystem. Most of the processor families offer various cache sizes. But the dynamic cache resizing is yet to be implemented in a CMP. The on-chip L3 cache occupies nearly 50% of the total chip area in Intel Itanium processor [3]. Hence, providing a flexibility of configuring cache at run-time has now become more important with its growing size on desktop- and server-platforms.

To switch off the over-allocated cache, WSS of an application has to be estimated accurately. Like other researchers [4], we also define WSS as the number of unique cache line addresses accessed by all threads/processes in a given interval. Various heuristics were used in the past to determine the optimal cache configuration. D. Albonesi [5] uses offline method to determine the optimal cache associativity. Researchers have used other parameters, such as, CMR and memory to L1 cache data traffic [6], time interval between two accesses to a cache line [7], CMR, instructions per cycle and branch frequency [8] and AMAL [9] to dynamically reconfigure the cache. However, these parameters are not very appropriate to resize the cache as they form an indirect mechanism for estimating WSS. The power gains obtained by switching off the over-allocated cache depend on the accuracy of WSS estimation method, how fast the controller can respond to changes in WSS and overhead incurred by the estimation method. Power gains are sub-optimal if WSS is estimated conservatively. On the other hand, application incurs more cache misses if WSS is under-estimated. The execution time of an application may degrade significantly, if the controller fails to quickly respond to the changes in WSS. Hence, we propose a new method which accurately estimates WSS of an application(s) executing on a CMP. The estimated WSS can be used for partitioning cache among virtual machines or applications co-scheduled on a CMP. In this paper, we demonstrate its use to switch-off the over-allocated cache. Previous studies [4] estimate WSS for a single threaded application by executing it on a uniprocessor system and use it to optimize leakage power in the primary instruction cache. We demonstrate the use of our WSS estimation method to optimize leakage power dissipated in a large shared L2 cache on a CMP on which threads execute concurrently.

Considering a trend of increasing number of the cores, we study a tiled CMP (Fig. 1). In a tiled architecture, tiles are replicated and connected through an on-chip switched-network (NoC). Each tile has a core, a private split L1 cache, a slice of L2 cache and a router. In our implementation, the L2 cache is distributed across all tiles and it is shared by all cores. To maintain cache

Digital Object Indentifier 10.1109/TC.2012.291

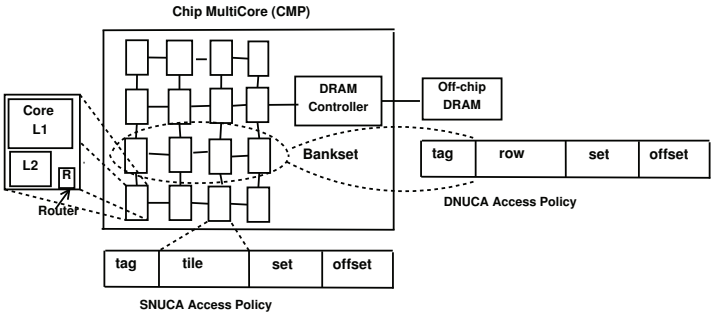


Fig. 1. Tiled CMP used for experimentation

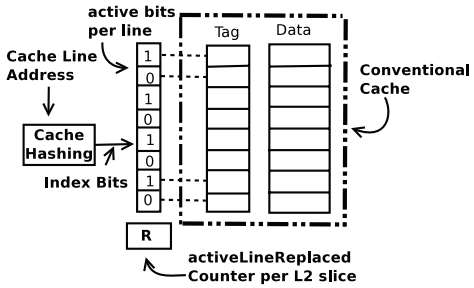


Fig. 2. Tagged working set size estimation method

coherence between the private L1 caches, a directory is present in each tile. These tiles are inter-connected via 2D-mesh NoC and per-tile router.

For power and performance reasons, a large cache on CMPs is partitioned into multiple banks which are connected using NoC [10]. Such a distributed cache offers non-uniform access latency to various cores on a CMP. Hence, it is referred to as "Non-Uniform Cache Architecture (NUCA)". Kim et al. [10] proposed two access policies for the NUCA cache, which are static NUCA (SNUCA) and dynamic NUCA (DNUCA). In SNUCA, the predetermined bits from a memory address statically decide the location of L2 bank, where data is cached. To reduce the cache access latency, 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 (slices in our architecture) in a row form a bank-set. Data can be cached in any of these L2 banks. On an L1 miss, data is first searched in the nearest L2 bank and then in the remaining L2 banks in that row before reading it into the nearest L2 bank. If data is present in a farther bank, then it gradually migrates towards a nearer bank on consecutive accesses. Fig. 1 also shows that, *tile* bits from the memory address determine the L2 slice where data is cached in SNUCA and *row* bits determine the row in which data is cached in DNUCA.

In the case of over-allocated and dispersed NUCA cache, AMAL [9] cannot be applied to predict cache requirement of an application. AMAL includes time required to traverse NoC and also to access off-chip memory on a miss in the L2 cache. In AMAL, time required to traverse NoC dominates over the time required to access off-chip memory when the majority of L2 accesses are hits. Hence, Bardine et al. [11] use ratio of the number of hits to the farthest cache way (which is also the least frequently used in DNUCA) to the nearest cache way in DNUCA to predict the cache requirement. However, they studied DNUCA cache with only one or two cores. In such platforms, both the cores are on the same side of the shared bus and cache is on its other side. This heuristic works well for a smaller CMP where cache and cores are on either side of the interconnect. It fails to scale for a CMP with many cores. Similarly, other commonly used heuristic, such as CMR [12] is not fast enough to respond to the changes in WSS. 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 entire cache is allocated. This might slowly drift cache associativity set by the controller significantly from an ideal associativity value. Conversely, our implementation estimates WSS and adjusts associativity accordingly. Following are our main contributions:

- A proposal for a highly accurate method to estimate WSS of an application(s), executing on a large CMP. It has negligible hardware space overhead of 0.1%.
- A scalable implementation of NUCA with adaptive cache associativity. Each L2 cache controller adjusts associativity of its L2 slice independently without accessing NoC. This makes our way adaptable cache implementation scale with the increasing number of cores. It gives fine control over associativity of each L2 slice. Our implementation achieves overall 37% and 40% energy-delay product (EDP) [1] savings over their reference SNUCA and DNUCA implementations, respectively.
- Implementation of adaptable associative DNUCA cache: TWSS can be applied to a non-tiled or tiled NUCA cache, unlike previous implementations [11]. It achieves 6.3% higher EDP savings than that obtained with Bardine's heuristic [11].

We describe TWSS and our way adaptable algorithm in section 2 and section 3, respectively. Section 4 describes the experimental setup. Results, related work and conclusions are described in section 5, section 6 and section 7, respectively.

2 Working Set Size Estimation

The working set of an application is the unique L1 cache line addresses (CLAs) accessed in an interval or *monitoring interval*. Cardinality of this set is WSS or to be more precise, cache working set size. Formally, if $l_1, l_2, l_3....l_n$ are unique CLAs accessed by an application, then the working set S is,

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

1. EDP represents a trade-off between energy consumed by an application and its execution time.

2.1 Dhodapkar's WSS Estimation Method (DHP)

Dhodapkar et al. [4] hash an instruction address and set the corresponding bit in a 1K bit-vector. The number of bits set in the bit-vector probabilistically determines WSS of an application. If the total number of bits in the bit-vector is N and the number of addresses hashed in the bit-vector is K, then the fraction f of bits set is:

$$f = 1 - (1 - \frac{1}{N})^K \qquad K = \frac{\log(1-f)}{\log(1-\frac{1}{N})} \quad (2)$$

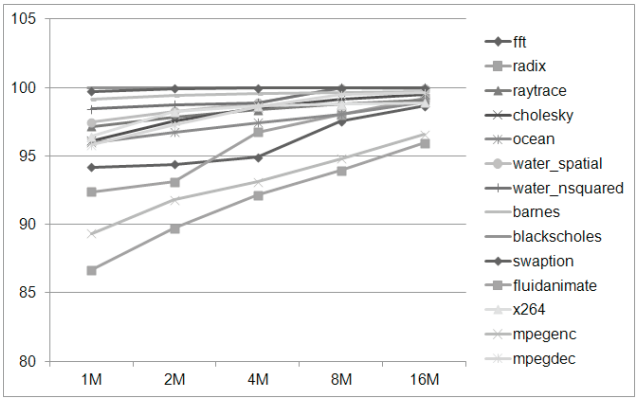
Hence, in DHP, if the fraction of bits set in the bit-vector is known, then WSS can be calculated using Eq. (2). Even though this method has negligible hardware overhead, accuracy of the estimated WSS depends on the choice of hash function. It may under-estimate WSS of an application due to an aliasing problem, i.e. multiple addresses might hash into the same bit. Hence, we propose a new method which overcomes this drawback.

2.2 Tagged WSS Estimation Method (TWSS)

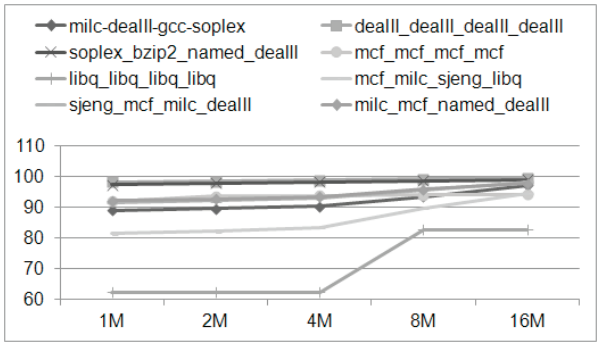
In TWSS, an active bit is maintained for each cache line along with its tag bits as shown in Fig. 2. Tag bits help to uniquely identify the CLA. Thus, TWSS overcomes the aliasing problem seen in DHP. An active bit is set on a cache hit. On a cache miss, the cache controller might evict a cache line. Hence, a counter (activeLineReplaced) is also maintained in each L2 slice to count the number of evicted addresses which have been accessed in the same monitoring interval i.e. the activeLineReplaced counter counts the number of cache lines replaced with their active bits set. The active bits and activeLineReplaced counter are reset at the beginning of every monitoring period. Similar to DHP, WSS is estimated in terms of the number of cache lines. At the end of every monitoring period, WSS is calculated using Eq. (3).

$$WSS = A + R \quad (3)$$

where, A is the number of cache lines with active bits set and R is the number of cache lines replaced with their active bits set, i.e. value of the activeLineReplaced counter. We call our WSS estimation method "tagged working set size (TWSS)" estimation method as tag bits of a cache line uniquely identify addresses accessed and evicted. TWSS method is independent of cache access policies (SNUCA/DNUCA). In the case of DNUCA, when a cache line migrates, its active bit maintained along with its tag, is also moved with it. TWSS over-estimates WSS if the same cache line is accessed and replaced multiple times in a monitoring interval which happens in the case of applications with large WSS only. Please see section 5.7 for the details of the distribution of cache line replacements. Aggregate WSS of all threads executing on a CMP can be obtained by implementing TWSS in the shared last level cache (LLC). Whereas, WSS of an individual thread can be obtained by implementing TWSS in the primary cache. In our case, TWSS is implemented in the LLC as we want to resize it. TWSS



(a) Multi-threaded Workload



(b) Multiprogramming Workload using SPEC 2006

Fig. 3. shows variation in reuse-time on the X axis and percentage of accesses on the Y axis. The majority of accesses have reuse-time less than 4M clock cycles.

can be implemented at any cache level such as at the private L2 cache of a CMP with shared L3 and private L1, L2 caches. However, private caches are generally smaller than the shared lower level caches which might cause more replacements per set. Hence, accuracy of the estimated WSS could be lower i.e. WSS will be over-estimated if the corresponding thread has large WSS, and when it is estimated at the private L2 cache.

2.3 Determination of monitoring interval

Typically monitoring interval is measured in terms of the number of instructions graduated [4], [12]. In the case of a uniprocessor, it is possible to measure the number of committed instructions without accessing NoC. Bardine et al. [11] count the number of requests made to the shared L2 cache while reconfiguring it. However, on a tiled architecture, the L2 cache is shared and dispersed in many tiles. Hence, monitoring the number of cache accesses requires to access NoC. Therefore, we use the monitoring interval in terms of the number of clock cycles. Some studies monitor program phases and reconfigure micro-architectures on detecting changes in a program phase [13]. A program phase consists of multiple (and also dispersed) contiguous time intervals in which the architectural parameters such as branch mispredictions, cache miss rate etc. remain constant. However,

on a large CMP, multiple applications execute in multiple phases simultaneously which greatly increases the over-all possible program phases [14]. This might result in frequent changes in the cache configuration, which further degrades execution time. Hence, we reconfigure cache after a fixed number of clock cycles, i.e monitoring interval.

We reset active bits at the beginning of every monitoring interval. If the monitoring interval is too small, fewer active bits would be set. This would be misinterpreted as smaller WSS. On the contrary, if the monitoring interval is too large, WSS would be over-estimated, failing to switch-off the excess L2 slices. Hence, determination of the accurate monitoring interval is important. To estimate it, we measured the number of clock cycles between two consecutive accesses made to the same CLA by any thread in an application. We refer to the number of cycles between two consecutive accesses as the *reuse-time*. The monitoring interval should be such that the majority of accesses should have reuse-time less than the monitoring interval. With this, if a cache line is not accessed in the previous monitoring interval, then it will not be accessed in the next monitoring interval and such an address can be evicted from the cache. Hence, we profiled the reuse-time of all CLAs for all applications. Please refer to section 4 for details of the experimental setup. Fig. 3(a) shows the percentage of total accesses with reuse-time less than 1M, 2M... clock cycles. In the case of blackscholes, swaption and barnes, 99% of accesses have reuse-time less than 1M clock cycles. This means that the monitoring interval for these applications could be as small as 1M clock cycles. For other applications, if the monitoring interval is increased from 1M to 2M clock cycles, percentage of accesses having reuse-time less than 2M clock cycles increases. This curve tapers after 4M clock cycles. Hence, we use 4M clock cycles as the monitoring interval. In the case of radix, fluidanimate, fft and mpegenc up-to 94% of accesses have reuse-time less than 4M. The execution time might degrade in these applications if monitoring interval smaller than 4M clock cycles is used.

We also evaluated distribution of the reuse-time of all accesses in the case of multiprogramming workloads obtained with SPEC 2006 benchmarks. This is shown in Fig. 3(b). Multiprogramming workloads like mcf-milc-sjeng-libq and libq-libq-libq-libq have only up-to 80% and 60% accesses with reuse-time less than 4M clock cycles, respectively. Remaining 20%-40% of accesses have reuse-time larger than 280M clock cycles. However, these applications have very large WSS. Hence, these cache lines get replaced without accessing them again. On the contrary, mcf-dealII-gcc-soplex has smaller WSS, and 90% of accesses have reuse-time lesser than 4M clock cycles. Hence, this application shows some degradation in execution time with 4M clock cycles as the monitoring interval.

To summarize, after experimenting with a variety of multiprogramming and multithreaded workloads, we conclude that the monitoring interval of 4M clock cycle is suitable for the majority of applications.

Algorithm 1 evaluates 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 > 0 then
8:    assoc_1 = assoc_1 + r
9: end if
10: new_assoc = max(max_associativity, assoc_1)
11: current_assoc = getCurrentAssociativity()
12: assoc = 0.5 * new_assoc + 0.5 * current_assoc
```

2.4 Hardware Implementation and Overhead

For cache of 512KB in size and 64B cache line, 8K active bits are required which is just 0.1% of space overhead. Since we reset active bits and activeLineReplaced counter at the beginning of every monitoring interval, our experimental analysis shows that, a 32-bit saturating counter per L2 slice is sufficient as an activeLineReplaced counter for our monitoring interval. TWSS adds minimal overhead in terms of the hardware area and power consumption. To count the number of active bits set, a simple scan can be done at the beginning of every monitoring interval. This requires 4K clock cycles if scanning is done on both edges of the clock. This time is negligible compared to the monitoring interval value of 4M clock cycles. The other simpler way to count the number of active lines is to maintain a counter and increment it when an active bit changes its value from 0-to-1.

3 Variable Way SNUCA and DNUCA

Interconnect on a CMP has become complex and adds significant latency. Therefore, algorithm used to vary associativity of cache should use locally available information to avoid additional delays and traffic on the NoC. In our implementation of adaptable way NUCA, the L2 cache controller in each tile evaluates associativity of its L2 slice without accessing NoC. Because of this our algorithm scales well with the number of tiles present on a CMP. It achieves higher energy savings with minimal performance degradation as cache associativity of each L2 slice is determined according to its usage.

We use the monitoring interval of 4M clock cycles to estimate the cache requirement. The L2 cache controller in each tile executes Algorithm (1) to evaluate associativity of its L2 slice independently without accessing NoC. The number of active bits set (A) and the number of active lines replaced (R) in the previous monitoring interval are used to estimate WSS (EWSS)(lines 1-3). The WSS is in terms of the number of cache lines. Line 4 evaluates size of the cache bank. We assume the vertical

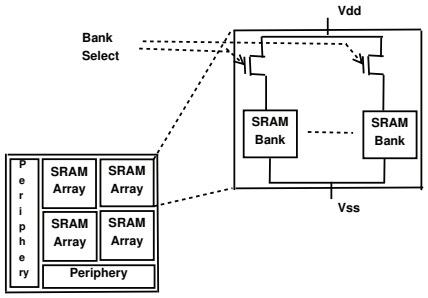


Fig. 4. An SRAM bank is switched off to reduce associativity of all sets in an L2 slice by one

implementation of cache, i.e. a single way of all sets in an L2 slice is implemented in one bank. However, TWSS is generic and can be applied for any other implementation of cache. Line 5 evaluates associativity of the L2 slice from the estimated WSS. Associativity will be reduced (increased) if the new associativity evaluated in line 5 is lesser (greater) than the current associativity. Minimum associativity of 2 is assigned to avoid frequent conflict misses. Line 6 calculates the number of active cache line replacements per set. Fig. 13 shows the distribution of average replacements per set and its standard deviation. Our empirical data shows lesser standard deviation if the average replacements per set is small. Hence, for the nonzero replacements per set, associativity of all sets is increased and the increment is proportional to the number of replacements per set.

As mentioned earlier in the section 2.3, a program is composed of many phases [13]. The monitoring interval of 4M clock cycles might include just a single or multiple program phases. Hence, we gradually vary associativity in the consecutive monitoring periods using a filter as shown in line 12. Filter is a standard term used in the "signal processing" for dampening oscillations. When duration of a program phase of an application is smaller than the monitoring interval, the use of filter reduces fluctuations caused due to phase changes in an application. For a program phase longer than the monitoring interval, associativity calculated in the previous interval will be reused.

3.1 Hardware Implementation Details

We assume that one way of cache is implemented in one bank. Hence, to reduce associativity of all sets by four, four banks are switched off. In other words, the switch-off granularity is at the bank level. The circuit level implementation is shown in Fig. 4. A bank is switched off (on) by switching off (on) the transistor connected in series with the SRAM bank.

To reduce the cache associativity, the L2 controller holds new L1-requests arrived in its queue. After in-progress L1-requests are serviced, clean L2 cache lines from the banks to be switched off are invalidated and modified L2 cache lines are written back to the off-chip memory. Then associativity is reduced and servicing of the held L1-requests is resumed. In the case of increase in cache associativity, L1-requests are held, associativity is increased and then L1-requests are released. Since the incoming L1-requests are put on hold during the L2 cache reconfiguration, the L2 controller might require slightly larger queue to hold incoming L1-requests. However, even if the queue becomes full, it will not cause any erroneous execution as the NoC uses the credit-based flow control. For a fair comparison, we use queues of the same size in all our implementations and the reference execution. Our simulator models time and power overhead caused due to held L1 requests, cache line write-backs and invalidations. We refer to our implementation of adaptable way DNUCA and SNUCA as TWSS.D and TWSS.S, respectively, in the rest of the description.

4 Experimental Setup

We use SESC [15] to simulate a core, Ruby from GEMS [16] to simulate the cache hierarchy and interconnects. DRAMSim [17] is used to model the offchip DRAM. DRAMSim uses MICRON power model [18] to estimate power consumed in DRAM accesses. Intacte [19] 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 and their area is estimated using CACTI 6.0 [20]. Area of the core is estimated based on the area of Intel Nehalem core. In order to estimate the latency (in cycles) of a network link, we estimate area of all components in a tile and create the floor-plan. 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 activity and coupling factors of all links, caused due to the messages sent over the NoC. Table 1 give the system configuration that we simulate.

We evaluate multithreaded workloads with one-to-one mapping between threads and cores (Table 2)[2]. We have carefully chosen applications from splash2 [21], Alpbench [22] and parsec [23] benchmark suites such that they cover programs with different characteristics and domains ranging from high performance computing, financial domain, animation, media and signal processing. We have skipped initial serial portion and simulate only parallel section in all the test cases. We test all workloads with 16 threads and simulate 4 billion instructions. fft, mpegdec and mpegenc are executed till completion as these applications execute less than 4B instructions.

To determine memory intensive and non-memory intensive benchmarks, we follow procedure explained in [24]. Instructions per cycle (IPC) is determined using an ideal L2 cache, which does not generate any L2 cache misses. IPC is also determined using a non-ideal L2

2. Rest of the PARSEC benchmarks either use OpenMP APIs or libraries which are not supported by SESC [15] compiler, hence cannot be compiled using it.

Core	out-of-order execution, 3GHz, issue/fetch/retire width of 4
L1 Cache	32KB, 2-way, 64B cache line size, access latency of 2cycles (estimated using CACTI), private, cache coherence using MOESI protocol
L2 Cache	512KB/tile, 16 way, 64B line size, 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 per tile table 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

TABLE 1
System configuration used in experiments

Name	App. Domain	Description, WSS(L/M/S)
Splash2 Benchmark[21]		
fft	Signal Processing	FFT on 1M points, M
radix	General	Radix sort on 1M keys, M
raytrace	Graphics	Raytrace with car input file, M
cholesky	HPC	blocked sparse matrix factorization on tk29, L
water_spatial	HPC	sim. of 512 water molecules, M
water_nsquared	HPC	sim. of 512 water molecules, M
barnes	HPC	Barnes-Hut method on 16K bodies, M
ocean (continuous)	HPC	512x512 grid points, L
PARSEC Benchmark[23]		
blackscholes	Financial	SimLarge i/p, S
swaptions	Financial	SimLarge i/p, M
fluidanimate	Animation	SimMedium i/p, L
X.264	Media	SimLarge i/p, M
Alpbench Benchmark [22]		
mpegenc mpegdec	Media	Encodes/decodes 15 Frames of size 640x336, M

TABLE 2
shows applications used for study and their WSS information(L:Large, M:Medium, S:Small).

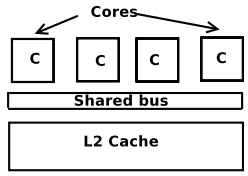


Fig. 5. Experimental configuration used to simulate multiprogramming workloads composed of SPEC 2006 benchmarks

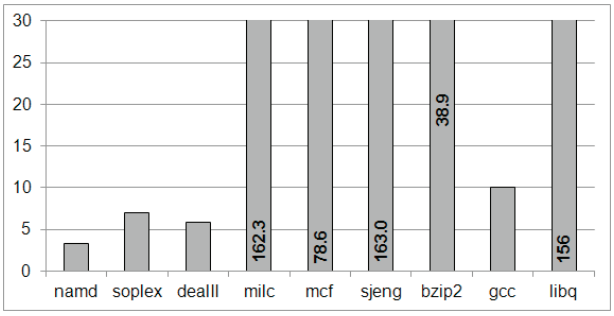


Fig. 6. shows IPC gain with ideal 4MB shared L2 cache over the non-ideal L2 cache

Name	Description, Memory Intensive Property(L/S)
dealII-dealII-dealII-dealII	floating point(FP) insts, LLLL
milc-dealII-gcc-soplex	Combination of integer and FP, HLLL
soplex-bzip2-named-dealII	Combination of integer and FP, HHLL
milc-mcf-named-dealII	Combination of integer and FP, HHLL
sjeng-mcf-milc-dealII	Combination of integer and FP, HHHL
mcf-mcf-mcf-mcf	all integer programs, HHHH
libq-libq-libq-libq	all integer programs, HHHH
mcf-milc-sjeng-libq	Combination of integer and FP, HHHH

TABLE 3
Multiprogramming workload simulated on a quad-core system

cache. Benchmarks which show IPC gain of at least 20% with the ideal L2 cache when compared to that obtained with the non-ideal L2 cache, are considered as memory intensive. The rest of the benchmarks are classified as non-memory intensive. We have a good mix of both memory intensive and non-intensive benchmarks (not shown here due to insufficient space). Applications like ocean, fft, cholesky and raytrace are memory intensive as they have large L2 cache miss rate, where as, blackscholes and swaption have very negligible cache miss rate.

Apart from multithreaded workloads, we also perform experiments with SPEC 2006 benchmarks [25]. These benchmarks allocate large memory and have larger WSS. Due to insufficient virtual memory, malloc syscall fails if sixteen different SPEC 2006 programs are simulated simultaneously on our simulator. This is because, SESC simulator compiles applications for a 32 bit-machine. Hence, we simulate 4 programs on a quad-core CMP with 4MB shared L2 cache. Architecture layout is shown in Fig. 5. This configuration is similar to Intel Xeon [26]. Another reason behind choosing this type of architecture is to show applicability of TWSS to a non-tiled architecture. TWSS shows similar results for 4-tiled architecture with total 4MB of the shared L2 cache.

Similar to multithreaded programs, we determined IPC of SPEC 2006 benchmarks with the ideal and non-ideal L2 cache (Fig. 6). mcf, libq, sjeng, milc and bzip2 programs are memory intensive, whereas, namd, gcc, soplex and dealII are non-memory intensive programs. We have carefully designed combinations of benchmarks with high and low cache miss rates; e.g. a combination of four instances of dealII has all applications with very low L2 miss rate, whereas, a combination of sjeng, milc, mcf and libq has all memory intensive applications. The details of multiprogramming workloads are given in Table 3, where "HHLL" in the description column denotes that it is a combination of two memory intensive and two non-intensive benchmarks. We have simulated 2 billion instructions in all tests.

5 Results

In this section, we first evaluate accuracy of TWSS estimation method. We quantitatively compare it with DHP. We also compare TWSS with other heuristics such as AMAL, CMR and Bardine's [11] proposal of variable

way policy for DNUCA caches. Following are various implementations evaluated in this paper:

TWSS.S/TWSS.D: TWSS.S and TWSS.D determine cache associativity using Algorithm (1) after every 4M clock cycles. "S" and "D" denote SNUCA and DNUCA, respectively.

AMAL.S/CMR.S, AMAL.D/CMR.D: In these implementations, AMAL and CMR are used as heuristics to vary cache associativity. Their values are calculated by considering accesses made by all threads to the last level cache. The cache associativity of all L2 slices is reduced (increased) if AMAL/CMR is lesser (greater) than that in the previous time slot by 10%, otherwise it remains the same. It should be noted that NoC access is essential to evaluate AMAL or CMR as the L2 cache is dispersed in various tiles. It is possible to determine AMAL/CMR per L2 slice and adjust its associativity accordingly. However, these values show negligible variation due to the over-sized cache. Hence, we evaluate AMAL and CMR values by considering accesses made to all L2 slices. In these implementations, decision is taken by one L2 controller and then conveyed to the remaining controllers. Because of this, these implementations do not scale with increasing number of cores. Apart from this, all L2 slices set uniform cache associativity, despite their non-uniform usage.

DHP.srand: In DHP.S, for a fair comparison, we maintain a bit-vector of 8K bits in each L2 slice and use hash function based on srand and rand functions as mentioned in [4]. A bit in the bit-vector is set by hashing CLA missed in the L1 caches or due to write-backs done by them. We evaluate DHP.srand implementation with a variety of commonly used hash functions, such as hash function based on srand and rand, hash function used in swaption benchmark from Parsec benchmarks suites [23] and a prime-modulo hash function (mod of 8209).

5.1 Accuracy of TWSS

Since we want to resize the L2 cache, we implement TWSS in the L2 cache and determine accuracy of TWSS at the L2 cache level. We determine *true/actual* WSS (AWSS) by counting the number of unique CLAs accessed by all L1 caches. This includes read/write requests missed in the L1 cache and write-back requests. Fig. 7 shows overall ratio (in terms of the geometric mean) of the WSS estimated using TWSS to AWSS. It also shows the overall WSS ratio obtained with the DHP method using hash functions mentioned above. All values are compared after every 4M clock cycles. In all applications, the WSS estimated using TWSS is close to 1. Even though applications such as, fft, ocean etc. show non-zero replacements per set (Fig. 13), WSS is not over-estimated. TWSS over-estimates WSS if the same address is accessed and replaced multiple times in a monitoring interval. For all these applications, TWSS estimates WSS accurately within accuracy of 0.001%. We also evaluated TWSS for L2 slice of 256KB for which WSS is marginally over-estimated for ocean and cholesky. This is due to higher number of replacements per set. For L2 slice of 1MB, TWSS does not over-estimate WSS as there are lesser cache line replacements than that obtained with L2 slice of 512KB in size. To summarize, accuracy depends on the cache size and footprint of an application.

In the case of DHP, srand-based hash function estimates WSS accurately or over-estimates it by a small margin for all applications, except for cholesky and ocean. It under-estimates their WSS by 21% and 37%, respectively. With other hash functions, the DHP method under-estimates by a large margin which is detrimental to any cache optimization. Finding a hash function which suits all applications is a very hard problem. The over-estimation in fft with the DHP method is due to its probabilistic nature as shown in Eq. (2).

Fig. 7 shows ratio of the WSS estimated by TWSS to AWSS evaluated at L2 cache. We also compare the WSS estimated by TWSS (which is implemented in the L2 cache) to WSS evaluated at L1 caches. This includes accesses done by all threads. This comparison is given in Table 4(a). In X.264, the WSS is under-estimated by both DHP and TWSS methods, since most of the accesses are L1 cache hits. L2 is accessed on L1 miss or write-backs. X.264 spawns up-to six concurrent threads even when executed with sixteen threads as a command line parameter. However, for the rest of the applications, the WSS estimated by TWSS is close to 1. Our results indicate that TWSS can be used to estimate the aggregate WSS of all applications executing concurrently on a CMP. As per our knowledge, this is the first approach which can estimate the aggregate WSS of multiple threads/processes executing concurrently on a CMP. It can also be used to estimate WSS of an individual thread if TWSS is implemented in the L1 cache. Similarly, it can be implemented at any other cache level.

Hardware complexity of the hash functions: The hash function based on the prime-modulo is easy to implement in hardware but it under-estimates WSS significantly for the majority of applications. The same applies to the hash function used in swaption benchmark. The hash function based on srand and rand function exhibits reasonable accuracy for the majority of applications, except for ocean and cholesky. Hence, we measured the approximate number of instructions executed by the srand and rand functions implemented in the standard "C" library. We compiled these functions using -O3 option for SESC simulator [15] [3]. Altogether these functions execute approximately 13K instructions in 19K clock cycles. Table 4(b) gives the total number of instructions executed to hash addresses accessed at the L2 cache. It also shows the number of clock cycles required to do the same. Clearly, srand-based hash function requires significant number of instructions. This also implies that its hardware implementation is complex. Hence, it is not

3. Pl. check Section 4 for more details on the experimental setup

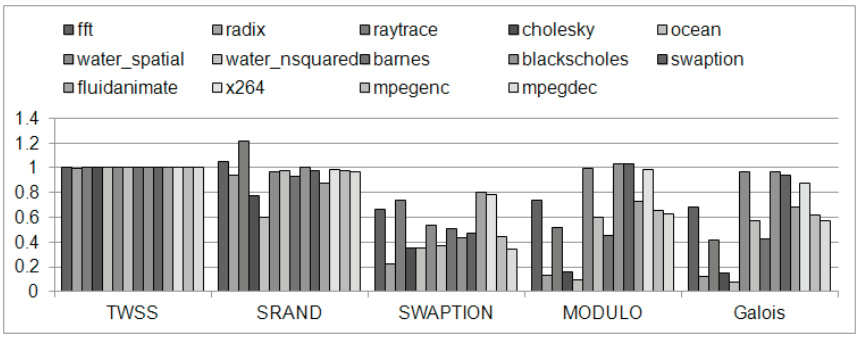


Fig. 7. shows overall ratio (in terms of the geometric mean) of the WSS estimated using TWSS and DHP methods to true/actual WSS (AWSS) evaluated at L2 cache. SRAND, SWAPTION, MODULO and Galois denote various hash functions used in the DHP method.

TABLE 4

(a) shows geometric mean of ratio of the WSS estimated using TWSS and DHP methods to the actual WSS evaluated at L1 cache. SRAND denotes hash function used in DHP.

App.	TWSS	SRAND
fft	1.03	1.03
radix	1.72	1.72
raytrace	1.04	1.55
cholesky	1.02	0.79
water_spatial	0.97	0.98
water_nsquared	0.98	0.98
barnes	1.03	1.03
ocean	1.04	0.63
blackscholes	0.97	0.97
swaptions	0.94	0.94
fluidanimate	1.91	1.19
x.264	0.85	0.87
mpegenc	1.18	1.19
mpegdec	0.98	0.98
average	1.07	1.06

(b) shows the # of committed instructions and # of clock cycles (both in billion) required to evaluate srand and rand functions in the DHP method

App.	Insts	Cycles
fft	545	811
radix	109	162
raytrace	451	672
cholesky	323	481
water_spatial	21	31
water_nsquared	88	131
barnes	1163	1732
ocean	1825	2718
blackscholes	302	450
swaptions	242	360
fluidanimate	75	112
x.264	319	476
mpegenc	217	324
mpegdec	204	304

(c) shows correlation of various heuristics to the actual WSS

App.	TWSS	AMAL	CMR
fft	0.99	0.65	-0.88
radix	0.99	-0.9	-0.93
raytrace	0.99	-0.88	-0.88
cholesky	0.99	0.49	0.58
water_spatial	0.98	0.02	-0.2
water_nsquared	0.99	0.2	0.27
barnes	0.99	0.09	-0.13
ocean	0.99	0.13	-0.22
blackscholes	1	-0.17	0.04
swaptions	0.99	0.63	-0.37
fluidanimate	0.99	0.56	0.43
x.264	0.99	0.67	0.1
mpegenc	0.98	0.04	0.21
mpegdec	0.98	-0.98	-0.79

practical to implement these hash functions in hardware. Though the latency of WSS estimation module is not a bottleneck, it will consume considerable amount of power and increased latency, thereby requiring hardware queues with larger capacity for this module. Hence, next we consider a more feasible hardware implementation of deterministic hash function.

Hash Function based on Linear Feedback Shift Registers (LFSR): LFSR is a shift register whose input bit is a linear function of its previous state. Some of the applications of LFSR include pseudo-random number generators used in stream-ciphers and deterministic input sequence-generators for chip testing. The initial value of an LFSR is called the seed. Since LFSR is deterministic, the next number generated by LFSR depends on the previous state. The bits position that affect the next state are called taps. The rightmost bit of LFSR is called the output bit and the leftmost bit is called the input bit. We evaluate DHP with 32-bits long Galois-LFSR which is commonly used in cryptographic algorithms and for generating cyclic-redundancy-checks in networking [27]. $X^{32} + X^{31} + X^{29} + 1$ describes its characteristic polynomial. This means that the taps are at $32^{nd}, 31^{st}$ and 29^{th} bits position. This hash function requires a bit-vector of 64K in size which is eight times larger than that used by TWSS. Fig. 7 also gives the WSS estimated using Galois-LFSR hash function. This function under-estimates WSS significantly for most of the applications. We also experimented DHP with LFSRs having various Fibonesi polynomials. All of them under-estimate WSS significantly.

5.2 Comparison with AMAL and CMR heuristics

We evaluated AMAL and CMR after every 4M clock cycles. Table 4(c) shows correlation between AMAL and CMR with AWSS. It also shows correlation between the WSS estimated with TWSS to AWSS. AMAL and AWSS are poorly correlated, which is quite intuitive. We use realistic values of NoC link latencies. AMAL depends on time spent in NoC traversal and also on the offchip memory access time. This makes it unsuitable for larger NUCA caches where cache is a surplus. CMR also shows poor correlation to AWSS, which is not intuitive. Due to larger size of cache, most of the accesses are hits in the L2 cache. As a result, larger AWSS does not imply larger CMR. Hence, correlation between CMR and AWSS is also poor. On the other hand, TWSS sets active bits in the case of L1 evictions, changes in sharers'

list or change in access permission, which enables it to accurately estimate WSS.

5.3 Evaluation of Adaptable Way SNUCA

Comparison against DHP: Fig. 8(a) shows EDP savings obtained with TWSS.S, DHP.srand and DHP.galois. DHP.srand and DHP.galois use srand and Galois hash functions mentioned earlier. These reading are normalized w.r.t. that obtained with SNUCA implementation. The EDP savings obtained by TWSS.S and DHP.srand are comparable, except in the case of cholesky and ocean. DHP.srand under-estimates WSS by 21% and 37%, respectively (Fig. 7). As a result, EDP degrades by 28% and 31%, respectively. The degradation in EDP is mainly due to around 20% degradation in the execution time in cholesky and ocean. LFSR with Galois hash function under-estimates WSS of these applications by approximately 83%. Hence, EDP degrades by 1.96X and 14X in cholesky and ocean, respectively. On the other hand, in the case of raytrace, DHP.galois achieves 6% higher EDP savings with 7% degradation in the execution time when compared to TWSS.S. For the remaining applications, TWSS.S and DHP.srand achieve approximately similar EDP savings. It should be noted that EDP gains obtained with the DHP method depend on the accuracy of estimated WSS, which, in turn depends on the choice hash function. It is very difficult to find a hash function which suits all applications.

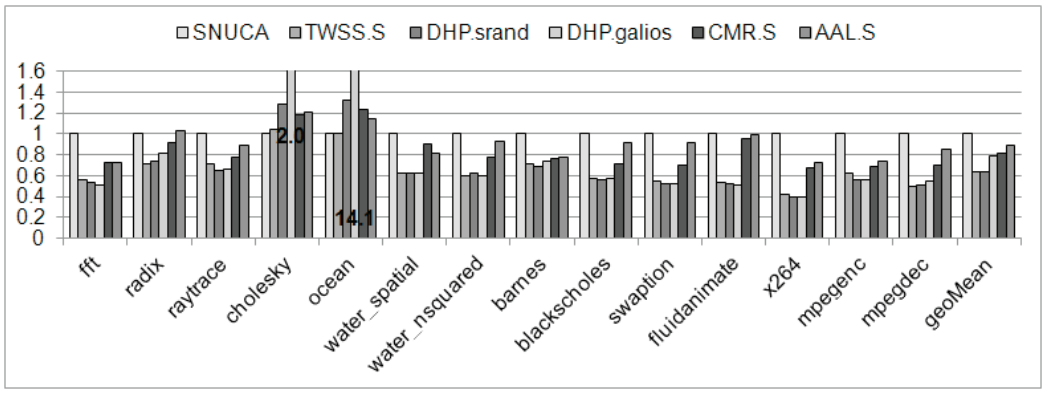
Comparison against AMAL/CMR: Fig. 8 also compares EDP savings and execution time obtained with TWSS.S against that obtained with AMAL and CMR heuristics. Overall, TWSS.S achieves 19% and 26% (geometric average) higher EDP savings than that obtained with CMR.S and AMAL.S. Large EDP savings in TWSS.S are due to lower cache allocation. Despite lower usage of the L2 cache, TWSS.S shows overall negligible execution time degradation of 1.5% (Fig. 8(b)). TWSS estimates cache usage of each L2 slice accurately and adjusts cache associativity appropriately. On the other hand, CMR and AMAL are poorly correlated to WSS in the case of large NUCA caches. Hence, AMAL.S and CMR.S do not switch-off the over-allocated cache in fft, radix etc. applications even if their cache requirement is much lesser. Also, these heuristic values are compared to that obtained in the previous time slot and not with the reference SNUCA execution in which the entire cache is allocated. This causes significant degradation in the execution time in some applications such as in ocean and cholesky. In the case of ocean, both AMAL.S and CMR.S switch off excessive cache ways, causing 14% and 22% degradation in the execution time and EDP, respectively. The maximum execution time degradation caused by TWSS is of 5.4% in mpegenc with 37% savings in EDP. As explained in section 2.3, the degradation is partly due to 93% of accesses in mpegenc have reuse-time lesser than 4M clock cycles. As TWSS.S reduces cache allocation, 7% of accesses that have reuse-time greater than 4M clock cycles cause additional cache misses. The write-back traffic also increases on reducing cache. This causes 5.4% degradation in the execution time.

5.4 Application to DNUCA: Comparison with Bardine's heuristic

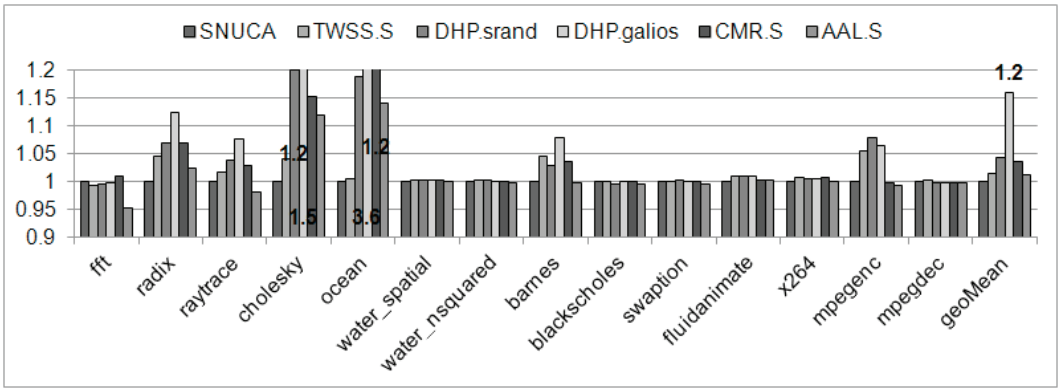
As explained earlier, Bardine et al. [11] use ratio of the number of hits to the farthest cache line to hits to the nearest cache line as a heuristic to change cache associativity in DNUCA. We call this ratio "cacheUsageRatio". This heuristic works in the case of DNUCA because frequently used cache lines gradually move towards the cores. However, in a tiled architecture, cache lines nearer to one core are far for other cores. Hence, Bardine's heuristic can be applied only up to four threads scheduled on cores in the first column of a tiled architecture (see Fig. 1.). Therefore, to compare TWSS.D to Bardine's implementation, we execute all workloads with only four threads scheduled on the cores in the first column. If the cacheUsageRatio is less than a threshold (T_1) then the associativity is reduced by 1. If the cacheUsageRatio is greater than a threshold (T_2) then the associativity is increased by 1. We refer this implementation as "Bardine.D" in the rest of the description. In this implementation of Bardine.D and TWSS.D which schedule four threads only, L2 cache lines are migrated to farther L2 slices in the same bank-set on replacement in its current L2 slice. In TWSS.D, the replacement counter is incremented only in the last L2 slice when a cache line is evicted to the off-chip DRAM.

Fig. 9 compares EDP savings obtained with TWSS.D against that obtained with Bardine's heuristic. TWSS.D gives overall 6.3% higher EDP savings than Bardine.D. This is because, in Bardine.D, cache associativity is gradually adjusted according to the changes in WSS, whereas, TWSS.D estimates WSS accurately and adjusts cache associativity to the correct value immediately from the next monitoring interval. Accurate estimation of WSS achieves higher EDP savings with negligible performance degradation, except in the case of ocean. Ocean has smaller WSS initially and it increases suddenly after a few monitoring intervals. In Bardine.D[4], associativity is reduced if the hit ratio is smaller than that obtained in the previous monitoring interval. As associativity reduces to a very small value of four, hit ratio does not increase when WSS increases. Hence, Bardine.D fails to increase cache associativity, degrading execution time and EDP by 37%. On the contrary, cache lines are migrated to farther slice along with their active bits in TWSS.D and the replacement counter counts the number of replaced active cache lines. Thus, TWSS detects sudden increase in WSS and increases cache associativity. This degrades execution time by 9% in ocean which is much lesser than 37% observed in Bardine.D. For other applications, degradation in execution time is less than 5% in TWSS.D. In addition to giving overall 6.3% higher

4. Threshold values are used as per [11].



(a) Smaller EDP value is better



(b) Smaller Execution Time is better

Fig. 8. Graphs in (a) and (b) show the normalized EDP and execution time obtained with various power optimized SNUCA implementations

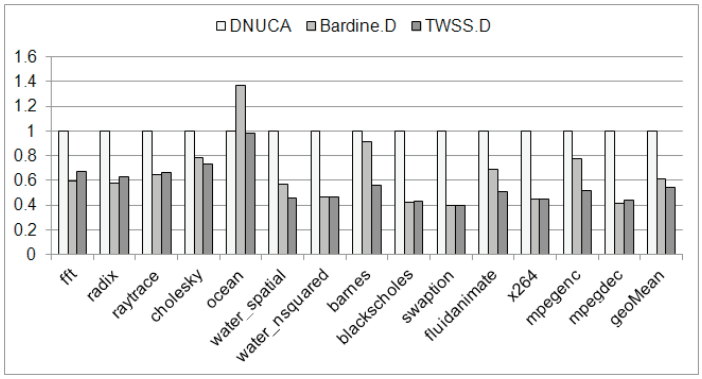


Fig. 9. shows quantitative EDP comparison of our variable way DNUCA implementation with Bardine's heuristic [11]

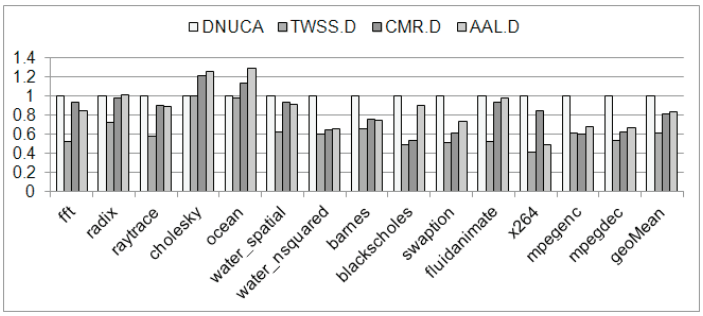


Fig. 10. shows the normalized EDP obtained with various implementations of power efficient DNUCA

EDP savings than that obtained with Bardine.D, TWSS.D is scalable with the number of cores on a CMP, unlike Bardine.D.

5.5 Scalability of TWSS.D

To demonstrate scalability of TWSS, we execute applications with sixteen threads scheduled on a sixteen tiled CMP with DNUCA. We quantitatively compare EDP savings obtained with TWSS.D against that obtained with AMAL and CMR heuristics.

Fig. 10 plots EDP normalized w.r.t. that obtained with the reference DNUCA. TWSS.D estimates cache usage of each L2 slice accurately and assigns associativity accordingly. As a result, it achieves overall 36% of EDP savings over the reference DNUCA. Whereas, AMAL.D and CMR.D achieve only overall 25% and 23% EDP savings over the reference, due to poor correlation to WSS (see Table 4(c)). Previous approaches use these heuristics to reconfigure L1 cache [12], [4]. Here, we use these heuristics to reconfigure L2 cache. As L2 cache is over-allocated, these heuristics do not correlate very well with the WSS estimated at L2 cache. Apart from this, EDP

savings also depend on how quickly heuristic responds to the changes in WSS. Unlike TWSS, CMR and AMAL heuristics estimate WSS indirectly. Hence, in the case of ocean, CMR.D and AMAL.D show 7.7% and 18% degradation in the execution time, respectively. This degrades EDP by 13% and 29%, respectively. Due to initial smaller WSS, cache associativity is decreased. On increase in the WSS, cache allocation is slowly increased in the following monitoring intervals. Whereas, TWSS gives the direct estimation of the cache requirement. Hence, TWSS.D does not show any execution time degradation in ocean. It shows overall very negligible degradation in execution time (geometric average of 1.1%).

5.6 Evaluation with multiprogramming SPEC benchmarks

We evaluated EDP savings obtained by coscheduling four SPEC programs on a quad-core CMP with 4MB shared L2 cache. The experimental details are given in section 4.

Table 5 shows ratio of the WSS estimated by TWSS and DHP method using srand and Galois-LFSR-based hash functions to AWSS. The DHP method with srand-based hash function over-estimates WSS for all applications. On the other hand, it significantly under-estimates with Galois-LFSR hash function. Table 5 also shows correlation of various heuristics to AWSS measured at L2. TWSS is very well correlated to AWSS. However, AMAL and CMR are poorly correlated to AWSS for the majority of applications. It should be noted that, when compared to the multithreaded workloads, multiprogramming workloads show comparatively better correlation between AMAL/CMR and AWSS. To summarize, TWSS accurately estimates WSS, unlike the DHP method. AMAL, CMR and TWSS show 55%, 62% and 96% correlation to the actual WSS.

Fig. 12(a) and Fig. 12(b) plot EDP and execution time normalized w.r.t. the reference execution with SNUCA. TWSS.S achieves 34% higher savings than both CMR.S and AMAL.S. We compare DHP method with srand and Galois-LFSR hash functions. As, Galois underestimates WSS significantly for all workloads, execution time with DHP.galois degrades by 15% and 10% in the case of dealII-dealII-dealII-dealII and mcf-dealII-gcc-soplex workloads. However, savings in leakage power compensate for the increased execution time. As a result, EDP savings do not degrade proportionately in these applications. In the case of mcf-milc-sjeng-libq and libq-libq-libq-libq, significant under-estimation in WSS does not degrade execution time. This is because, in these applications the majority of accesses have very large reuse-distance (Fig. 3(b)). So even if such addresses are kept in the L2 cache, eventually these addresses get replaced without a cache hit. DHP.srand does not underestimate WSS and allocates the similar sized cache as in TWSS.S. Thus TWSS.S and DHP.srand achieve similar EDP savings. These experiments re-confirm that the EDP savings obtained with the DHP method depend on the choice of hash function, whereas, TWSS uses inherent hash function built in the cache which evenly distributes addresses in the cache sets. TWSS over-comes the drawback of aliasing present in the DHP method by uniquely identifying CLA with the help of tag bits maintained with the cache lines.

5.7 Sensitivity to Replacements Distribution

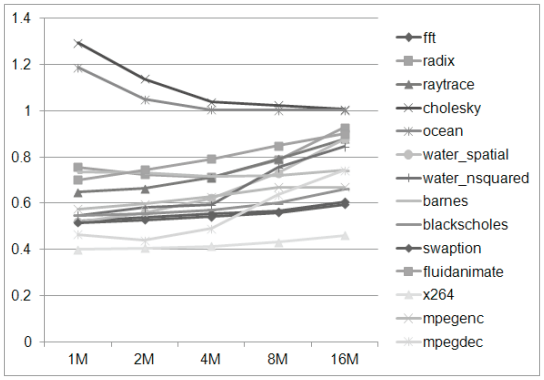
TWSS counts a CLA more than once in a monitoring interval, if it is accessed and replaced multiple times in the same monitoring interval. This results in overestimation of WSS. Hence, we examine distribution of the number of cache lines replaced per set for all applications. We profiled replacements per set of all sets in all L2 slices. Fig. 13 shows average replacements per set and its standard deviation for some of the applications. We have taken these readings after every 4M clock cycles. Fft shows less than 2 replacements per set in the initial execution intervals. The average replacements per set increases significantly in the last few execution intervals. On the contrary, cholesky and ocean show upto 10 replacements per set during the entire execution. Applications with large WSS show larger replacements per set. For larger average replacements per set, standard deviation is also larger. However, in applications with larger WSS, cache is not switched off. Hence, inaccuracies introduced due to uneven replacements per set, if any, does not introduce significant error in cache resizing. We observed similar behavior for SPEC workloads as well.

5.8 Sensitivity to Monitoring Interval

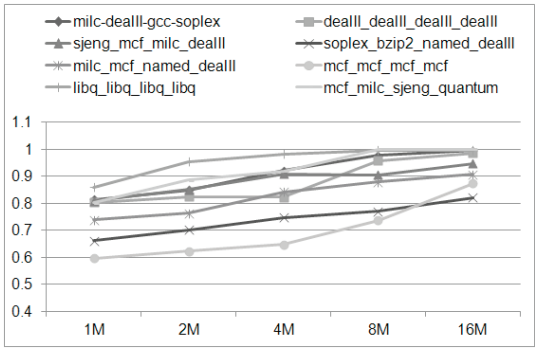
To study variation in the EDP savings with the monitoring interval, we performed experiments with 1M, 2M, 4M, 8M and 16M as the monitoring interval. The fraction of total addresses with reuse-time less than the monitoring interval, plays an important role in achieving higher energy savings without causing significant performance degradation. If the majority of accesses have reuse-time less than the monitoring interval, then lesser performance degradation is observed e.g. blackscholes, swaption, x.264 have 98-99% of CLAs with reuse- time less than 1M clock cycle (Fig. 3(a)). In such applications, monitoring interval of 1M clock cycles achieves highest EDP savings. With the monitoring interval of 4M and more clock cycles, data remains cached even if it has no future use. Hence, EDP gains start reducing as the monitoring interval increases. On the contrary, cholesky and ocean have larger WSS. Hence, with monitoring interval lesser than 4M clock cycles, execution time and EDP degrades. With 4M and larger monitoring interval, cache is not powered off and thus no EDP degradation is seen. In general, for the majority of applications, reuse-time versus monitoring period curves start tapering after 4M clock cycles. Hence, 4M clock cycles is a preferable monitoring interval.

App	WSS Ratio			Correlation		
	TWSS	SRAND	GALOIS	TWSS	AMAL	CMR
dealII-dealII-dealII-dealII	1	1.04	0.13	0.99	0.61	0.65
milc-dealII-gcc-soplex	0.99	1.26	0.1	0.99	0.34	0.33
soplex-bzip2-named-dealII	1	0.99	0.27	0.87	0.39	0.56
milc-mcf-namd-dealII	1	1.23	0.18	0.99	0.69	0.69
sjeng-mcf-milc-dealII	1	1.18	0.12	0.99	0.68	0.67
mcf-mcf-mcf-mcf	0.99	1.42	0.27	0.99	0.98	0.97
libq-libq-libq-libq	0.99	0.49	0.05	0.94	0.41	0.87
mcf-milc-sjeng-libq	1	1.29	0.4	0.99	0.29	0.21
geometric mean	1	1.07	0.15	0.96	0.55	0.62

TABLE 5
shows ratio of the WSS estimated using TWSS and DHP methods with srand and Galois-LFSR as hash functions, in terms of geometric mean. Table also shows correlation of various heuristics to the actual WSS

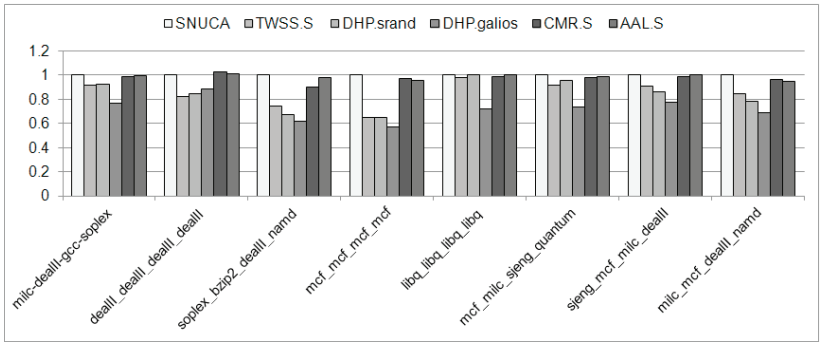


(a) Multi-threaded Workloads

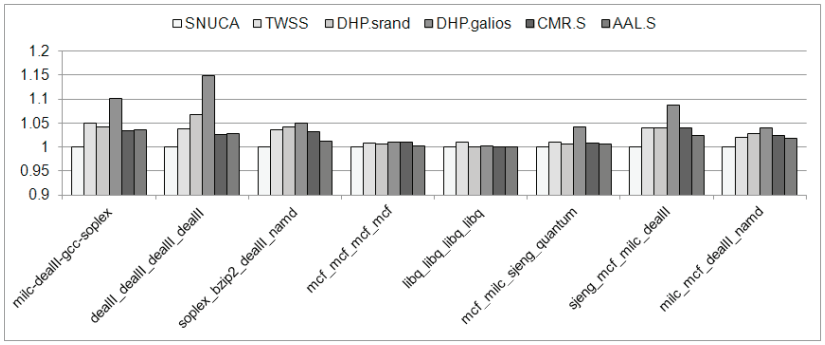


(b) Multiprogramming Workloads

Fig. 11. Sensitivity to the monitoring interval. Fig. shows variation in the normalized EDP on the Y axis with the monitoring interval on the X axis

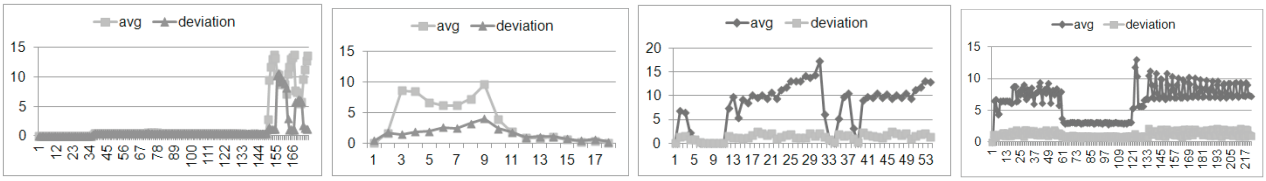


(a) EDP



(b) Execution Time

Fig. 12. Evaluation of SPEC benchmarks with SNUCA cache access policy



(a) fft (b) cholesky (c) ocean (d) mcf-milc-sjeng-quantum

Fig. 13. shows variation in average replacements per set during execution. The Y axis plots the standard deviation and average replacements per set against the monitoring interval on the X axis.

6 Related Work

WSS Estimation Methods: In embedded platforms, cache requirement of an application is determined using an offline mechanism and cache configuration is adjusted accordingly [28]. This is possible on embedded-platforms as designers have a prior knowledge of applications running on these platforms. TWSS is also ap-

plicable to these platforms. However, cache requirement varies drastically in the case of applications running on general-purpose platforms. Hence, cache configuration should be adjusted at run-time using some heuristic. CMR is one of such commonly used heuristic [29], [12]. CMR and AMAL increase if lesser than the required cache is allocated. In [9], authors use AMAL to decide the number of cores sharing a cluster of an L2 cache on a CMP. Dhodapkar et al. [4] estimate *WSS signature* of an application executing on a uniprocessor. As explained earlier in Section 2.1, this method might under-estimate WSS if multiple distinct addresses hash into the same bit. TWSS method overcomes this drawback by making use of tag bits maintained with cache lines. In [4], authors resize L1 instruction cache on a uniprocessor using the estimated WSS, whereas we switch-off unified NUCA cache on a CMP, which hosts concurrently executing threads.

In decay cache, Kaxiras et al. [7] use time interval between two consecutive accesses to an L1 cache line to decide whether it is dead and can be put into the sleep state. [30] conclude through experiments that time between two hits to a cache line is very small, whereas time before a miss is very large. Hence, they track various parameters such as, time between hits, the number of hits and time elapsed after last access for each cache line to determine when to switch it off. Both [7] and [30] incur a very large hardware overhead. On the contrary, TWSS sets an active bit for every cache line to determine whether it is in the working set. H. Zhou et al. [31] enhance decay cache by using *global control register* which predicts when a cache line can be put into sleep state. If a cache line is in in-active state for longer than value contained in the global control register, then it is switched into sleep state. The value of global control register is adapted by monitoring the hypothetical and actual CMR. Hypothetical CMR is a miss rate if all cache lines are in the active state. To determine the hypothetical CMR, only data arrays are changed into the switched-off state, whereas, tag arrays are maintained in the active state and retain their values. So after each monitoring interval, the additional cache misses incurred due to data arrays being in the switched-off state can be estimated. These additional cache misses are used to adjust value of the global control register. Even though the decay interval is adjusted at run-time, just like decay cache, this method also incurs higher hardware overhead. Enormous hardware overhead will be incurred, if a counter is maintained for every cache line in the case of large LLC in a CMP. Moreover, the tags array size is also considerable. Hence, significant amount of leakage power will be dissipated by keeping it into the active state. Koller et al. [32] proposed a "cacheGrabber" which increases cache utilization using non-intrusive cache prefetch instructions till a performance drop in a concurrently executing application and cacheGrabber is observed. This method is used to estimate CMR curves. The CMR works well to partition cache among applications executing on a cache constrained platform. However, our experiments show that the CMR is poorly correlated to WSS on platforms, where cache is over-provisioned. D. Albonesi [5] uses information profiled with an offline method to adjust cache associativity. However, offline method cannot adapt to WSS changes at run-time. Dropsho et al. [33] maintain hit counters along with each cache way and use the number of hits in each counter to adjust the cache associativity.

Power optimized caches: Bardine et al.[34] use a single bit per cache line in DNUCA cache to avoid unnecessary cache line promotions/demotions. With this, they reduce the dynamic power consumed by the DNUCA cache in a single or dual core scenario. In [5], D. Albonesi selectively disable the over-allocated cache associativity to reduce its dynamic power consumption.

Reducing supply voltage of less frequently used cache lines [35] and completely cutting off their supply voltage [36] are two major streams of solutions to reduce the leakage power in caches. Powell et. al [36] proposed gated-Vdd cache, where memory cells can be disconnected from the power supply using a gated-Vdd transistor. As a memory cell loses its data on disconnecting it from the power supply, this might cause additional cache misses if the required amount of cache is not allocated. On the contrary, K. Flautner et. al [35] proposed to reduce the supply voltage of a cache line so that it retains its data, while dissipating lesser leakage power. They call this mode a "drowsy mode". To access data from the cache lines in the drowsy mode, the supply voltage of a cache line is increased back to its normal supply voltage. With 70nm technology, the ratio of leakage power dissipated in the normal mode to that in the drowsy mode is 12.5 [35]. The same ratio reduces to 4 for 45nm technology [37]. Hence, the leakage power savings obtained for 45nm technology are much lesser than that obtained with the 70nm technology using the drowsy or dynamic voltage scaling (DVS) technique. This ratio will be still lower for 32nm and 22nm technology. This is because data-preserving techniques rely on a fine-grained adjustment of SRAM cell electrical parameters. These parameters are less controllable in smaller transistor sizes because of larger process variation effects [38], [39]. Hold noise margin of an SRAM cell degrades as supply voltage is reduced [38]. This makes caches operating at lower supply voltage more susceptible to transient errors caused by alpha particles [39], [40], [41]. Hence, with transistors in nanometer size, completely switching off the cache is the only feasible solution.

Past studies optimized leakage power for smaller L1 caches [4], [33]. Implementing it for the shared, distributed LLC NUCA is challenging due to the presence of concurrently executing threads. Commonly used heuristics such as AMAL [9] and CMR [12] cannot be applied in the case of over-provisioned caches.

7 Conclusions

We propose a highly accurate "tagged working set size estimation (TWSS)" method to estimate WSS of threads/processes executing concurrently on a chip multiprocessor. TWSS has a negligible hardware overhead (0.1% of cache size). TWSS makes use of the in-built hash function and tag bits of a cache line to overcome the drawback of aliasing seen in the past approach. The estimated WSS can be used to partition cache across virtual machines or to switch-off the over-allocated cache.

We use the estimated WSS to switch-off the over-allocated associativity of the last level SNUCA and DNUCA cache 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 the NoC, this method gives finer and independent control over associativity of each L2 slice. This makes it scale well with the increasing number of cores present on a CMP. Our adaptable associative SNUCA and DNUCA caches achieve 37% and 40% EDP savings over their reference SNUCA and DNUCA with negligible degradation in execution time (approximately geometric mean of 1% in both). Unlike average memory access latency (AMAL) and cache miss ratio (CMR) heuristics, TWSS responds to WSS changes quickly. TWSS achieves average 19% and 26% higher EDP savings than CMR and AMAL on SNUCA. Unlike previous implementation of adaptive way DNUCA [11], our adaptable way DNUCA is scalable with the number of cores present on CMP and achieves 6.3% higher EDP savings than Bardine's heuristic [11]. TWSS can be applied to both non-tiled or tiled architectures.

References

[1] "http://www.amd.com/us/products/server/processors/6000-series-platform/6200/Pages/6200-series-features.aspx."
[2] "Intel Nehalem." [Online]. Available: http://www.3dnow.net/phpBB2/viewtopic.php?f=1&t=1474&p=6720
[3] W. Don, W. J., and C. Victor, "The on-chip 3-MB subarray-based third-level cache on an Itanium microprocessor," in *ISSCC*, 2002.
[4] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *ISCA*, 2002.
[5] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *MICRO*, 1999.
[6] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting cache line size to application behavior," in *ICS*, 1999.
[7] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *ISCA*, 2001.
[8] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *MICRO*, 2000.
[9] M. Hammoud, S. Cho, and R. Melhem, "Dynamic cache clustering for chip multiprocessors," in *ICS*, 2009.
[10] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002.
[11] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete, "Way adaptable DNUCA caches," *Int. J. High Perform. Syst. Archit.*, 2010.
[12] 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," ser. HPCA, 2002.
[13] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *MICRO*, 2003.
[14] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," in *ISPASS*, 2004.
[15] 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.
[16] 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, "Multifacet general execution-driven multiprocessor simulator (gems) toolset," 2005.
[17] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: a memory system simulator," 2005.
[18] "Micron DRAM power data sheet." [Online]. Available: http://www.micron.com/products/ partdetail?part=MT47H128M8HQ-3E
[19] 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.
[20] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html
[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.
[22] 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.
[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.
[24] R. Manikantan and R. Govindrajan, "Performance oriented prefetching enhancements using commit stalls," in *Journal of Instruction Level Parallelism*, 2011.
[25] [Online]. Available: http://www.spec.org/
[26] "http://ark.intel.com/products/codename/22796/Woodcrest."
[27] E. Dubrova and S. S. Mansouri, "A BDD-based approach to constructing LFSRs for parallel CRC encoding," in *Proc. IEEE 42nd Int. Symp. on Multiple-Valued Logic*, 2012.
[28] A. Janapsatya, A. Ignjatović, and S. Parameswaran, "Finding optimal L1 cache configuration for embedded systems," in *ASPDAC*, 2006.
[29] M. Powell, S. hyun Yang, B. Falsafi, K. Roy, S. Member, and T. N. Vijaykumar, "Reducing leakage in a high-performance deep-submicron instruction cache," *IEEE Transactions on Very Large Scale Integration Systems 9*, 2001.
[30] A. Jaume, G. Antonio, V. X., and B. Micheal, "IATAC: A smart predictor to turn-off L2 cache lines." in *ACM TACO*, 2005
[31] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte, "Adaptive mode control: A static-power-efficient cache design," in *PACT*, 2001.
[32] R. Koller, A. Verma, and R. Rangaswami, "Estimating application cache requirement for provisioning caches in virtualized systems," in *MASCOTS*, 2011.
[33] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott, "Integrating adaptive on-chip storage structures for reduced dynamic power," in *PACT*, 2002.
[34] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete, "A power-efficient migration mechanism for D-NUCA caches." in *DATE*, 2009.
[35] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *ISCA*, 2002.
[36] M. Powell, S. hyun Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-vdd: A circuit technique to reduce leakage in cache memories," *ISLPED*, 2000.
[37] G. Gammie, A. Wang, H. Mair, R. Lagerquist, R. Philippe, G. Sumanth, and K. Uming, "Smartreflex power and performance management technologies for 90nm, 65nm, and 45nm mobile application processors," in *Proc. of IEEE Journal*, 2010.
[38] B. Calhoun and A. Chandrakasan, "Static noise margin variation for sub-threshold SRAM 65-nm CMOS," in *IEEE J. on Solid State Circuits*, 2006.
[39] V. Chandra and R. Aitken, "Impact of voltage scaling on nanoscale SRAM reliability," in *DATE*, 2009.
[40] ——, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS," in *Proc. of IEEE Int. Symp. on Defect and Fault Tolerance of VLSI Systems*, 2008.
[41] S. H. Shin, S. W. Chung, E.-Y. Chung, and C. S. Jhon, "Adopting the drowsy technique for instruction caches: A soft error perspective," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 2008.

Authors' Biography

Aparna Mandke Dani is a PhD student at the Indian Institute of Science (IISc) in Bangalore. She received her Masters degree in Computer Science from the IISc. Her research interests include multicore architecture, cache organization, performance modeling and workload characterization.

Bharadwaj Amrutur is an Associate Professor in the Electrical Communication Engineering Department at the IISc. His research interests are in embedded sensing, communication and processing. More information can be found at http://chips.ece.iisc.ernet.in

Y. N. Srikant received his B.E in Electronics from Bangalore University, and M.E and Ph.D in Computer Science from the IISc. His area of interest is compiler design. He is the editor of a handbook on advanced compiler design published by CRC Press in 2002 and 2008 (2nd ed.) His most recent research includes compiler optimizations for power reduction in embedded systems, performance estimation of programs through program analysis, and parallel programming paradigms and languages. Srikant is currently a Professor in the Department of Computer Science and Automation at the IISc.