EnCache: Improving Cache Energy Efficiency Using a Software-Controlled Profiling Cache

Sparsh Mittal and Zhao Zhang
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa 50011, USA
Email: sparsh@iastate.edu, zzhang@iastate.edu

***Abstract*—Leakage energy has been estimated to increase by five-times with each CMOS technology generation and this is expected to become a major source of energy dissipation, especially in large last-level caches (LLCs). We present EnCache, a novel cache leakage energy saving scheme that uses a new micro-architecture component called profiling cache. With this hardware support, system software can accurately predict the cache usage and energy efficiency of a running program for 32 possible cache configurations and vary the cache configuration for saving cache leakage power. Unlike existing schemes, EnCache can directly optimize for the overall memory subsystem power efficiency instead of only cache power efficiency. EnCache employs dynamic performance regulation and hence does not require offline profiling or per-application tuning. The experiments performed over our workload confirm the superiority of EnCache scheme over conventional energy-saving schemes. On a baseline configuration of 2MB cache size, the average improvement of memory subsystem energy and energy-delay-product (EDP) by using EnCache are 31.7% and 28.8%, respectively.**

***Keywords*: Last-level cache, leakage energy saving, profiling cache, dynamic performance regulation, set-sampling**

I. Introduction

Leakage current and leakage power have been predicted to increase by 7.5 times and 5.0 times respectively, with each CMOS technology generation [1]. Further, leakage power is likely to dominate the power consumption for sub-45nm technology. Caches occupy more than 50% of the total area of the processor [2] and most modern processors use a large shared LLC to bridge the widening gap between the speed of main memory and processor core. Therefore, power consumption of caches is increasingly becoming a concern in modern processor design.

Since programs show large intra- and inter- program variations in their cache requirements, processor designers have to use cache with average case in mind. This, however, leads to a large wastage of energy for the applications with small working set size (WSS), or cache thrashing for the applications with large WSS. Recently, several approaches have been proposed which dynamically reconfigure the caches to save leakage power. However, the hardware-based approaches (such as [3]) have a common limitation. Reducing cache size increases cache miss rate and program execution time. Therefore, it involves a delicate trade-off between performance and energy efficiency. In the absence of online profiling, downsizing cache may cause severe cache thrashing and therefore may dramatically increase program execution time and power consumption in the processor core and DRAM memories. This increase may even offset the power savings in cache. It is very difficult, if not impractical, for non-adaptive hardware-based schemes of cache energy saving to also take into account the components other than the cache. Furthermore, their control mechanism depends on arbitrary parameters (e.g miss-bound, decay interval) that must be tuned per application. The presence of large intra-program variations and the differences between the profiled runs and actual programs make the approach of per-application tuning highly ineffective and difficult-to-scale.

We present EnCache (Energy saving approach for Caches), a new software-based approach on top of lightweight hardware support. The key component of the hardware support is a simple **profiling cache**. It is tag-only cache and uses set-sampling to predict cache miss rates of multiple cache configurations of much larger sizes in an *online* manner. It works non-intrusively and due to its decoupled and parallel operation and small-size, its latency is easily hidden. Profiling cache is not a part of the cache hierarchy and it does not lie at the critical access path of the cache. Further details are explained in Section III-A.

The previous approaches such as [4] utilize sampling only to profile different associativities of the current size of the cache, while EnCache provisions a separate cache structure which can profile different associativities at different cache sizes. Thus, EnCache considerably expands upon the potential of sampling. This is a significant difference, which enables the prediction of energy efficiency of multiple cache sizes and thus, guide reconfiguration. Profiling cache has an energy overhead of less than 0.5% of L2 cache energy. Our simulation results show that a profiling cache is highly accurate, with an average error of 0.26MPKI (miss-per-Kilo-instruction) in predicting the cache miss rates for 100 benchmark/configuration combinations.

Our profiling cache is designed to also estimate the impact of cache miss-rates on performance, in terms of memory stall cycles. Using these estimates and other performance counters, an OS component periodically predicts the memory-subsystem (which includes LLC and main memory) energy for multiple cache configurations. Then, the cache configuration with the minimum estimated energy is chosen for the next interval and, if necessary, the cache is reconfigured to that configuration.

EnCache addresses the aforementioned shortcomings of the hardware-based approaches. It optimizes for memory-subsystem energy rather than merely cache energy. It optimizes directly for *energy*, unlike previous approaches which

978-1-4673-0818-2/12/$31.00 ©2012 IEEE

work by trying to control *miss-rate* and thus optimizing cache energy indirectly. Furthermore, EnCache uses dynamic performance monitoring and regulation and thus, does not require offline profiling or per-application tuning. A comparison with a popular technique named Hybrid Dynamic ResIzing (HDRI) cache [3, 5] shows the superiority of EnCache approach.

The rest of the paper is organized as follows. Section II discusses related work and Section III and IV explain the design and algorithms in more detail. Section V discusses the simulation environment, workload, and energy model. Section VI presents results on profiling cache accuracy and energy saving. Finally, we conclude in Section VII.

II. Background and Related Work

Profiling cache is based on the idea of set-sampling, which states that the behavior of the cache can be estimated by sampling only a small subset of cache sets [6, 7]. Qureshi et al. use set-sampling method to decide the best replacement policy among two choices [8]. Qureshi and Patt employ sampling idea for profiling L2 cache configurations having different number of active ways [4]. Profiling cache's ability to estimate performance of multiple cache configurations (Section III-A) is a significant improvement over these works, where set-sampling is used to predict the performance of only the different ways of the current set-count. This difference is critical for the purpose of improving energy efficiency.

In literature, several techniques have been proposed for saving cache-energy. For saving cache leakage energy Kaxiras et al. propose turning off the cache lines which have not been accessed for a certain number of cycles (called decay interval) [9]. However, the techniques based on a fixed decay-interval are shown to be less effective for L2 than for L1 [10]. Also, the optimal value of decay interval varies widely for different benchmarks [11]. Thus, for real-world applications, the utility of these approaches is limited. Flautner et al. [12] and Hanson et al. [13] propose placing idle cache lines in state-preserving mode and thus reduce static power consumption. However, these techniques are not suitable for high-performance processors, because of their requirement of multiple votage supplies. Leakage energy forms a large fraction of energy spent in last-level caches and hence some techniques (such as [14]) which focus on saving dynamic energy of cache are not so useful for saving energy in LLCs. A few techniques (e.g. [10, 11]) turn off only data-array of inactive regions of the cache and must always keep tag fields on; however, EnCache can have both tag and data arrays of the inactive regions turned off.

Kin et al. propose a small filter cache which is placed in front of L1 cache [15]. In essence, filter cache is a yet another level of cache in the hierarchy. Our profiling cache is different in that it is *not* a part of the cache hierarchy and hence, it does not lie at the critical access path of the cache.

Since miss-rates for some benchmarks can be arbitrarily small, even a small change in a small miss-rate shows up as a large percentage distorting its contribution to performance. Hence, to quantify increase in miss rates from the use of the energy saving techniques, we use *absolute difference* in misses (measured in MPKI) instead of *percentage absolute difference* in misses, following [16].

III. System Design and Implementation

In what follows, we assume that the LLC is an L2 cache, although the discussion can also be applied to an L3 cache.

A. Profiling cache

The energy consumption of a program with any configuration depends on dynamic activity and execution time of the program with that configuration. Hence, energy estimation requires estimation of program response for that configuration. For this purpose we use "*profiling cache*", which employs set-sampling to estimate cache miss rate.

Profiling cache is a data-less cache and gives accurate predictions even for sampling ratios (R_S) as high as 32 (Section VI-A); thus its storage size is very small compared with L2 cache. Furthermore, it is *decoupled from L2 cache*. These properties of profiling cache enable us to further extend it to a multi-level profiling cache: each level emulating a cache of 1X, 0.5X, 0.25X and 0.125X size of the L2 cache. Here all the four L2 caches are assumed to have same block-size and associativity and differ only in number of sets. As shown below, this extension increases the size of profiling cache to only double the size of a single-level profiling cache.

Let M be the number of sets in L2 and a be the number of sets in a single-level profiling cache for profiling L2 cache of M sets. Thus, we have $a = M/R_S$. Then the total number of sets in an n level profiling cache is

$$S = a + \frac{a}{2} + \frac{a}{2^2} + \frac{a}{2^3} + \frac{a}{2^4} + \ldots + \frac{a}{2^{n-1}} \tag{1}$$

A profiling cache must contain at least one set. Hence

$$\frac{a}{2^{n-1}} = 1 \Rightarrow a = 2^{n-1} \tag{2}$$

$$\Rightarrow S - \frac{S}{2} = a - \frac{a}{2^n} \;\; \Rightarrow S = 2a - \frac{2a}{2^n} \tag{3}$$

$$\Rightarrow S = 2a - 1 \approx 2a = \frac{2M}{R_S} \tag{4}$$

To see the typical values of n and S, we substitute $R_S = 32$ and $M = 4096$ (for 2MB baseline cache) . Then, we get from above, $n = 8$ and $S = 256$.

The L2 cache in our experiments uses LRU replacement policy and for such cases, the profiling cache uses extra counters to provide miss rates for configurations having different number of ways as well. This is based on the Mattson stack algorithm [17] for LRU caches. Thus, with merely $(2M)/R_s$ sets, a profiling cache can simultaneously emulate many caches of much large sizes. This feature is especially useful for miss-rate curve generation. Since profiling cache accounts for all accesses made to L2, calculation of actual miss rate does not require post-processing of data; this is a significant improvement over previous approaches (e.g. [16]). For the purpose of energy saving, we provision the configuration to only four levels, since this gives a large saving in energy, with a small performance loss. Thus, EnCache chooses a

suitable configuration from a large configuration space of 32 configurations of L2 (four states with eight ways each). These configurations are shown as an ordered 2-tuple (S, W), where S and W denote the L2 state and active ways respectively.

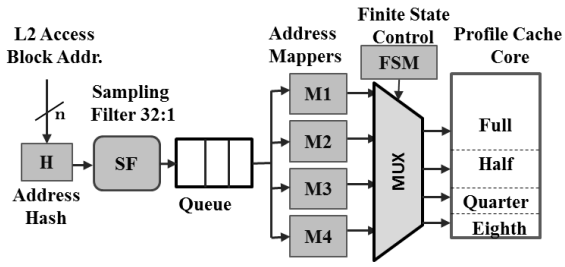

Fig. 1. The Design of Profiling Cache.

Figure 1 shows the frontend logic and storage component of the profiling cache design. Its core storage is a tag-only cache, which has the same set-associative structure and replacement policy as the L2 and thus, it emulates normal cache accesses. Each L2 cache access block address first passes a hashing logic (for randomization) and then goes through to a sampling filter. The sampling ratio is chosen at design time and is taken as 32 in our experiments, which means that only 1 out of 32 of memory block addresses in the physical address space will pass the filter. Sampling is implemented by merely a bit-shifting operation. Then, those addresses are sent through a small queue to the profiling cache core.

The core storage is split into four regions, called "$Full$", "$Half$", "$Quarter$" and "$Eighth$", respectively. Each region represents an emulated cache size (called L2 cache state), i.e. "$Full$" for full size, "$Half$" for half size, and so on. Each hashed address from the head of the queue is sent to four address mappers (M1, M2, M3 and M4). Each mapper is a simple logic that removes a subset of the address bits (decided by R_S) and then inserts a subset of bits that is the offset of each region in the profiling cache core. Thus it maps the addresses onto a unique cache set in one of those four regions (M1 for "$Full$" and so on). The four mapped addresses are sent to a multiplexer (MUX), from where they are sequentially sent to the profiling cache core by the control of a small finite state machine. A "miss" in profiling cache does not generate any further request for other caches; rather, the LRU block is evicted and the tag of the address missed is copied in its place.

The profiling cache core is accessed four times for each sampled address. Note that this does not cause congestion even in the case of bursty last-level cache accesses, because of a large value of R_S and the presence of queue. Profiling cache has small size and does not lie at the critical access path of the cache; thus its latency is small and easily hidden; moreover, it does not affect L2 cache access time. In what follows, we simply use the term profiling cache to denote multi-level profiling cache, unless otherwise mentioned.

B. Stall Cycle Prediction Based on Load Miss Count

For accurately estimating leakage energy for different configurations, the program execution time under those configurations needs to be estimated. However, the variation in number of cache-misses with cache configurations and the possibility of overlap between different miss events makes prediction of CPI of superscalar out-of-order processors a challenging task.

To address this issue, EnCache works as follows. Firstly, it uses a hardware counter to continuously measure the memory stall cycles due to L2 load miss events, taking into account the overlap between different L2 miss events. Now, for different L2 cache sizes, the miss-rate and hence the number of stall cycles would be different. However, it is observed that for a same interval i, if the miss-rate does not change considerably between two cache configurations, the ratio StallCycles$_i$/LoadMisses$_i$ does not change significantly between those cache configurations. This ratio is called Penalty Per Miss (PPM). Thus, we take PPM$_i(S, W)$ = PPM$_i(S^\star, W^\star)$=PPM$_i$, for all configurations, where $(S^\star, W^\star)$ is the actual cache configuration in that interval.

The profiling cache is so designed that apart from estimating total misses, it can also estimate total load misses for various configurations using additional counters. Thus, using LoadMisses$_i(S, W)$ and PPM$_i(S, W)$, StallCycles$_i(S, W)$ are estimated for all configurations. Then, using these estimates, along with the observed compute cycles for current cache size, Cycles$_i(S, W)$ values for all configurations are computed.

When the number of load-misses changes considerably with L2 cache size, the PPM value varies with L2 cache sizes and this affects the accuracy of energy estimation. However, a large saving in energy (as shown in results) indicates that for most cases, the variation is small enough and program performance and choice of best configuration are minimally affected.

C. Dynamic Performance Monitoring and Regulation (DPMR)

Energy minimization through reconfiguration involves performance trade-off. To make performance-efficient choices and control the aggressiveness of cache reconfiguration, EnCache makes use of DPMR, which works as follows. For any configuration (S, W), we define,

$$\Delta\text{Time}_i(S, W) = \frac{\text{Time}_i(S, W) - \text{Time}_i(Full, 8)}{\text{Time}_i(Full, 8)} \times 100$$

Note that Time$_i(Full, 8)$ is obtained in run-time with the help of profiling cache, even though the actual configuration in interval i may be different. ΔTime$_i(S, W)$ gives an estimate of percentage extra time that the configuration (S, W) would take compared to the baseline. If ΔTime$_i(S, W)$ exceeds a threshold (λ), EnCache rejects the configuration (S, W) for interval $i + 1$; thus dynamically adjusting the configuration space available for reconfiguration algorithm. Here λ is an application-independent constant and its value is set as 3%.

D. Energy Saving Algorithm

At any time, by allocating just appropriate LLC space to an application so that its working set can fit, the rest of the L2

cache can be turned off with little impact on performance. Using this observation, system software (which could be a kernel module) periodically chooses a configuration with minimum estimated energy using the algorithm given below. Here $Energy(S, W)$ is defined in Equation (5) in Section V-C. Initially L2 cache configuration is set to ($Full$, 8).

Algorithm 1 EnCache: Algorithm For Energy Saving

Input: $Misses$ and $Time$ estimates (for all configurations)
Output: Best $State$ and $Ways$ for interval $i+1$

```
Energy* = ∞, S*= -1, W*= -1
for S = {Full,Half,Quarter, Eighth} do
  for W = 1 to Assoc do
    Estimate PPM_i(S,W), ΔTime_i(S,W)
    if ΔTime_i(S,W) ≥ λ then
      Disregard (S,W) for interval i + 1; Continue to next
      configuration
    end if
    Estimate Energy_i(S,W)
    if Energy_i(S,W) < Energy* then
      Energy*=Energy_i(S,W), S*=S, W*=W
    end if
  end for
end for
RETURN (S*, W*) for interval i + 1
```

IV. Hardware Implementation

The L2 cache controller is shown in the Figure 2. The controller uses an eight-bit way-selection mask. For selecting or deselecting a particular way k, the corresponding bit W_k (for k={1, 2, ..., 8}) in this mask can be turned on or turned off respectively. The L2 cache has an eight-bank structure and switching to $Half$, $Quarter$ and $Eighth$ states is accomplished by keeping four, two and one bank of cache ON respectively and turning off the rest of the banks. This is achieved by a simple logic controlled by just eight bits. The approach of turning-off cache banks for leakage saving has been used by other researchers also, e.g. [18, 19].

To quantify the average percentage of active L2 cache lines, we define L2 ActiveRatio as:

$$\begin{aligned} \text{ActiveRatio} &= \frac{\sum_{i=1}^{N} Fraction(S_i^\star) \times W_i^\star}{N \times Assoc} \times 100 \\ S_i^\star &= \{\text{Full, Half, Quarter, Eighth}\} \\ Fraction(S_i^\star) &= \{1, 0.5, 0.25, 0.125\} \end{aligned}$$

Here $Assoc$ = L2 associativity, N = number of intervals and $(S_i^\star, W_i^\star)$ shows the actual configuration used in an interval i.

To handle the change in set and tag decoding resulting from change in L2 state, suitable tag and index(set) mask are used (Figure 2). We show the calculation of these masks for 2MB, 8-way cache with block size of 64. For index mask, a total of 12 bits are required since $Full$ state has 4,096 sets. Out of these, 9 LSBs are always set to 1, since $Eighth$ state has 512 sets. The three MSBs are: $a_2a_1a_0$ = Binary($8 \times Fraction(S_i) - 1$). Assuming a 45-bit address, maximum number of bits in tag-mask are $45 - 6 - 9 = 30$,

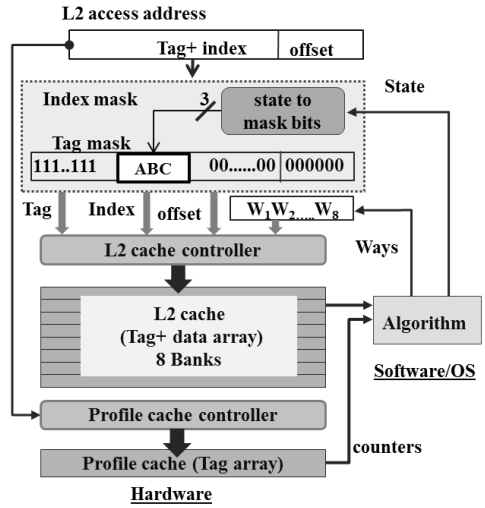

Fig. 2. L2 cache access logic (index mask omitted for clarity).

as required for $Eighth$ state (6 bits for block offset). Out of these, 27 MSBs are always set to 1 since a minimum of $45 - 6 - 12 = 27$ bits are required for $Full$ state. The three LSBs are simply $\overline{a_2a_1a_0}$. In Figure 2, these are shown as ABC.

Reconfigurations are handled in the following manner. When only the number of $Ways$ is decreased, the dirty data of disabled ways are written back to memory and other blocks are discarded. On a change in state, the new set and tag for cache blocks are computed and the blocks are re-located to the new addresses. Out of the blocks not fitting the available cache space, the dirty blocks are written back to memory and others are discarded. Such an approach can potentially incur a "one time" high overhead but is simple and requires small state storage and ensures consistency. The reconfigurations take place at a fixed interval boundary and hence, block transitions do not lie at the critical path of cache access. Because of a large interval-size and small number of reconfigurations (Section VI), EnCache keeps reconfiguration overhead small, which is easily amortized over the phase length. Further, on reconfigurations involving an increase in only active ways or active sets, writebacks are not required.

V. Experimental Methodology

A. Simulation Environment and Workload

We have used sim-outorder simulator [20], enhanced with a more realistic memory model. The important simulation parameters are shown in Table I. The interval length is set to be 1M instructions. Except *sixtrack*, which fails to simulate on sim-outorder, 25 out of the 26 SPEC2000 benchmarks are used with *ref* input data set. To emulate the representative behavior of benchmarks, while still limiting the simulation time, a representative simulation point of 100M instructions is selected for each benchmark using SimPoint 3.0 [21].

B. Comparison with Hybrid DRI Technique

We compare our technique to a well-known technique, which we refer to as Hybrid Dynamic ResIzing (HDRI)

TABLE I
SIMULATOR PARAMETERS

L1-I	64KB, 64B line-size, 4-way LRU, 1 cycle
L1-D	64KB, 64B line-size, 4-way LRU, 1 cycle
Decode/issue	4-wide, 64 RUU size, 4 INT ALUs,1 INT Mult/Div
	4FP ALUs, 1FP Mult/Div, 2 memory ports
Unified L2	2MB, 64B, 8-way, LRU, 10 cycles
Memory	150 cycle initial latency, 8-cycles for every 16 bytes, bandwith limit and 32-entry request queue are modeled.

technique [3],[5]. HDRI employs selective-ways and selective-sets to allow a fine resizing granularity. It requires choosing a benchmark-specific $MissBound$ through offline profiling and up-sizing or down-sizing the cache size based on whether the observed misses in an interval are smaller or greater, respectively, than the $MissBound$. For comparison purposes, we have implemented their technique for the L2 cache. Following [3], four states are used and throttling mechanism is also implemented . For 8-ways and 4 choices of number of sets, 12 among 32 cache configurations become redundant [3]. In such cases, a cache size with higher number of ways is chosen [3]. The reconfigurations are handled as explained in Section IV.

Yang et al. do not detail the systematic procedure for finding $MissBound$ [3, 5]. Based on inferences, the miss-bound is chosen as follows. For each benchmark, the number of L2 misses ($TotL2Miss$) for 100M instructions in baseline cache were recorded. Using this, average miss per interval (η) was calculated as follows: $\eta = TotL2Miss \times (1M/100M)$. Now, for each benchmark, simulations were performed using HDRI technique, with three values of $MissBound$ (η+200, η+300, $\eta + 400$), and the value of $MissBound$ leading to smallest (i.e. best) EDP was chosen for that benchmark. These offset values (between 200 and 400) were chosen considering the costs of cache access and off-chip access. At each of the three $MissBound$ values, at least four benchmarks shows best EDP value. This confirms the validity of the offset values chosen.

C. Energy Modeling

We calculate the energy spent in L2 cache, main memory and the run-time cost of executing the algorithm (E_{Algo}), since other components are minimally affected by our approach. Note that for baseline experiments, $E_{Algo} = 0$ and for HDRI technique $E_{prof} = 0$ (shown below). Using DE to show dynamic energy and LE to show leakage energy, we get,

$$Energy = DE_{L2} + LE_{L2} + DE_{mem} + LE_{mem} + E_{Algo} \quad (5)$$

We use the symbols E_{dyn}^{XYZ} and P_{leak}^{XYZ} to show the dynamic energy *per access* and leakage energy *per second* respectively, spent in any component XYZ (e.g. L2 cache, memory etc.).

The L2 energy is calculated as follows. An L2 miss is assumed to consume twice the energy as that of an L2 hit [13]. The leakage energy depends on active ratio (and hence, number of enabled sets and ways) of the cache [5]. The dynamic energy depends on the number of enabled ways [14] but is independent of the number of enabled sets, since it is the energy spent for accessing a read/write port. Hence, $P_{leak}^{L2}[S^\star] = Fraction(S^\star) \times P_{leak}^{L2}[Full]$. Thus,

$$DE_{L2} = E_{dyn}^{L2} \times (2 \times M_{L2} + H_{L2}) \times (W^\star/Assoc) \quad (6)$$

$$LE_{L2} = P_{leak}^{L2}[S^\star] \times (Cyc/Freq) \times (W^\star/Assoc) \quad (7)$$

Here, for any interval with configuration $(S^\star, W^\star)$, or entire execution, we have corresponding M_{L2} =L2 misses, H_{L2}=L2 hits, Cyc =cycles consumed and $Freq$ =processor frequency (2GHz). $Time$ and Cyc are related as $Time = Cyc/Freq$. The L2 energy parameters, as obtained from CACTI 5.3 (http://quid.hpl.hp.com:9081/cacti/) are shown in Table II.

TABLE II
ENERGY VALUES FOR L2 CACHE AND CORRESPONDING PROFILING CACHE. UNITS: NANOJOULE/ACCESS FOR E_{dyn}; WATT FOR P_{leak}

Baseline Cache Size	L2 Cache		Profiling Cache	
	E_{dyn}^{L2}	$P_{leak}^{L2}[Full]$	E_{dyn}^{prof}	P_{leak}^{prof}
2MB	1.086	2.016	0.005	0.007
4MB	1.24	3.264	0.008	0.013
8MB	1.44	5.81	0.01	0.025

The memory energy is calculated as follows. Each memory access consumes 70 nJ [22], thus $E_{dyn}^{mem} = 70nJ$. The main memory is assumed to use aggressive power saving mode as allowed in DDR3 DRAM, and thus $P_{leak}^{mem} = 0.18nJ/nSec$ when there is no memory access [22]. Thus,

$$DE_{mem} = E_{dyn}^{mem} \times A_{mem} \quad (8)$$

$$LE_{mem} = P_{leak}^{mem} \times Cyc/Freq \quad (9)$$

Here A_{mem} denotes the number of memory accesses (including additional write-backs generated due to reconfigurations).

The energy-cost of algorithm is calculated as follows.

$$E_{Algo} = (\#Tran \times 0.002 \text{ nJ}) + E_{prof} \quad (10)$$

Here $\#Tran$ denotes total block transitions (blocks turned off or on) and each transition is assumed to consume 0.002 nJ energy. Using A_{prof} to denote profiling cache accesses, E_{prof} is calculated as

$$E_{prof} = E_{dyn}^{prof} \times A_{prof} + P_{leak}^{prof} \times Cyc/Freq \quad (11)$$

We use CACTI 5.3 to estimate profiling cache energy, using Equation (4), single bank, 8-way structure for block-size of 8. The value of M for 2MB, 4MB and 8MB cache are 4096, 8192, 16384 respectively. CACTI assumes normal tag+data design and presents leakage power and dynamic energy values and also their percentage distribution in tag and data arrays. We take the energy of tag arrays only, since profiling cache is a tag-only structure. This value acts as an upper bound, since in absence of data arrays and less circuitry (e.g. no dirty bit, no communication with memory etc.) the profiling cache can be designed more efficiently. These values are shown in Table II. Since, on average, for every 32 accesses to L2, profiling cache is accessed only four times, profiling cache energy parameters

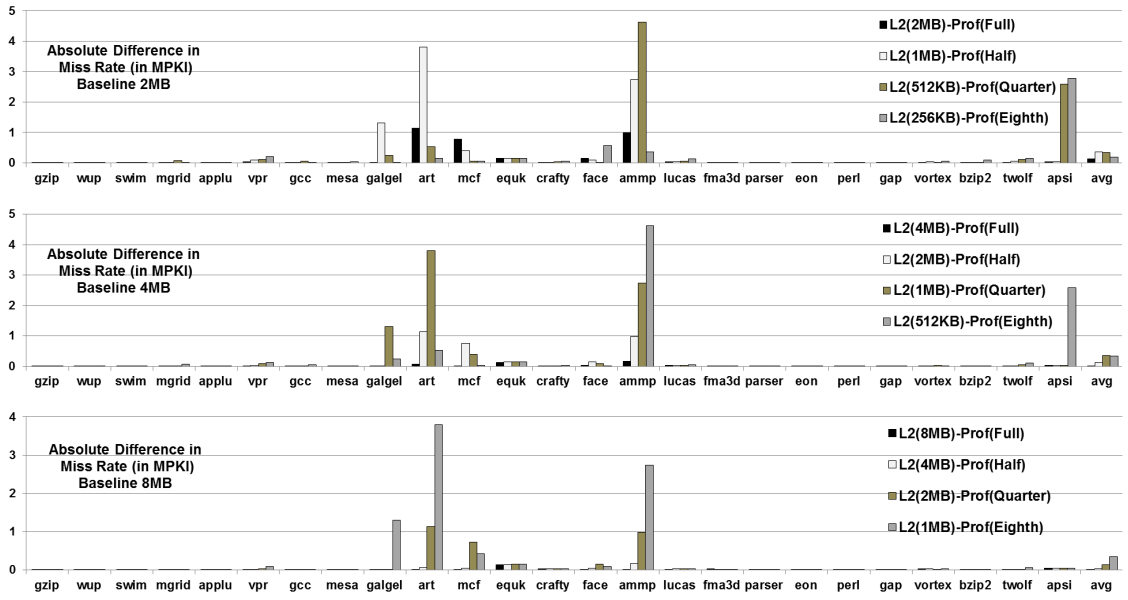

Fig. 3. Profiling Cache Prediction Verification. Absolute Difference in MPKI obtained from actual L2(size) and Multi-level Profiling cache.

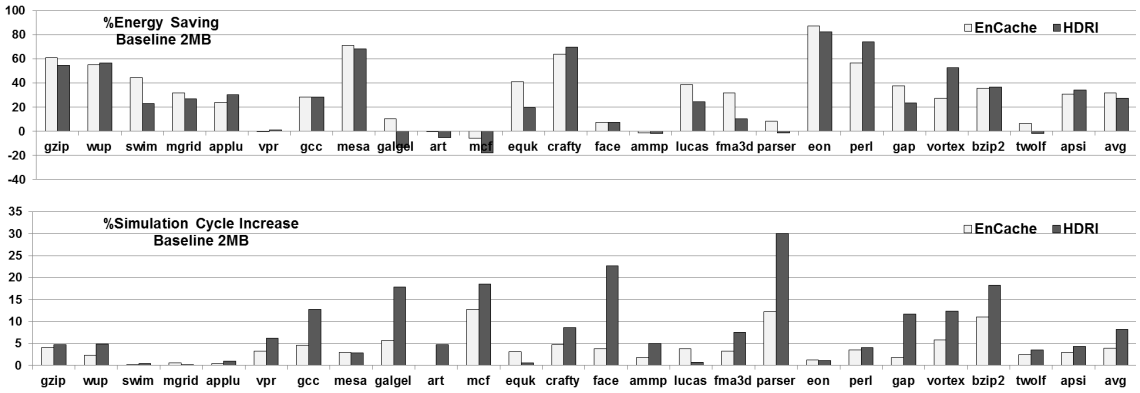

Fig. 4. Experimental Results with 2MB Baseline Cache (Here wup=wupwise, equk=equake, face=facerec, perl=perlbmk, avg=average).

are less than 0.5% compared to the corresponding parameters for baseline L2.

For each of the 32 configurations, EnCache maintains counters to measure L2 misses, load misses, predicted cycles, $\Delta Time$ etc. Many processors already contain several counters for measuring performance or for operating system [9]. These can be used as global counters, and hence the overhead of counters is ignored in energy calculations.

VI. Results and Analysis

A. Profiling Cache Prediction Accuracy

We explain profiling cache accuracy verification procedure with 2MB baseline cache; a similar procedure applies for 4MB and 8MB baseline caches. The profiling cache predicts miss-rates for four states and with 2MB baseline cache, these states correspond to L2 cache sizes of 2MB, 1MB, 512KB and 256KB. Hence, for each benchmark, experiments were carried out using baseline cache configuration of size 2MB, 1MB, 512KB and 256KB (each having 8-way and 64B block size) and miss per Kilo instructions (MPKI) were recorded. These values were compared with corresponding estimates obtained from a four-level profiling cache. For example, the miss-rate obtained from 2MB cache was compared with the miss-rate estimate obtained from profiling cache region that emulates "$Full$" size cache and so on. The results are shown in Fig. 3. Across 25 benchmarks, the average absolute error in profiling cache miss-rate estimates is merely 0.26 MPKI and average percentage absolute error is 5.91%. For 4MB baseline cache, these values are 0.22 MPKI and 5.34% and for 8MB baseline cache, these values are 0.13 MPKI and 4.15%. These results confirm the high accuracy of the multi-level profiling cache.

B. Energy Saving Results

Fig. 4 shows the saving in memory-subsystem energy when baseline cache of 2MB and its average value for EnCache and HDRI are 31.7% and 27.4% respectively. Average increase

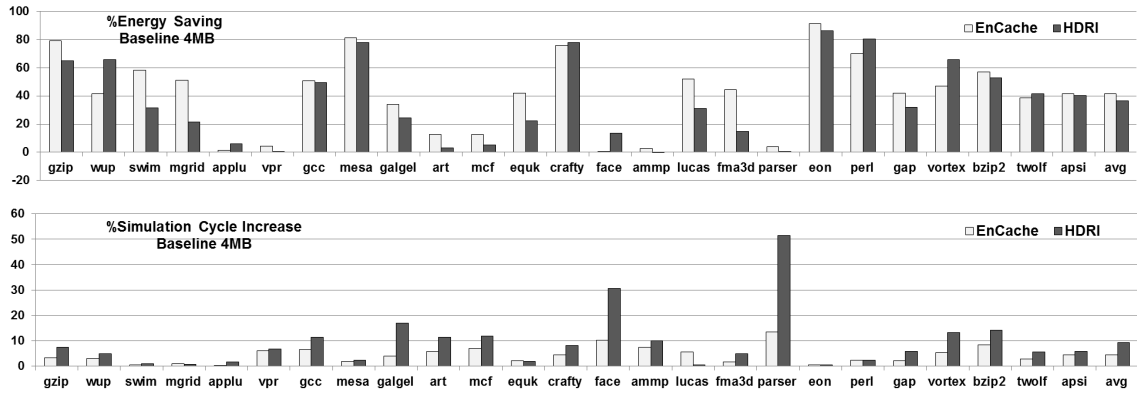

Fig. 5. Experimental Results with 4MB Baseline Cache (Here wup=wupwise, equk =equake, face=facerec, perl=perlbmk, avg=average).

in simulation cycles for EnCache and HDRI are 3.93% and 8.2% respectively and average saving in EDP are 28.8% and 20.6% respectively. The average ActiveRatio with EnCache and HDRI are 49.5% and 47.1% respectively and average increase in MPKI are 0.45 misses and 0.62 misses respectively. The figures for other quantities have been omitted for brevity.

The adaptive nature of both the algorithms especially benefits benchmarks such as *eon*, *gzip*, *mesa*, *crafty*, *wupwise*, *perlbmk* etc, where a large saving in energy is achieved. The worst-case performance of HDRI is very poor; as *mcf* shows loss in EDP of 39%. Similarly *galgel* shows loss of energy of 13% and *parser* shows simulation cycle increase of 30%. For EnCache, the worst-case performance happens on *mcf*, where loss in EDP is 19%. For *art* EnCache does not choose to reconfigure the cache at all, since the extra misses generated by reconfiguration would have offset energy saved in cache. On the other hand, HDRI performs poorly for *art* and shows loss in energy. A negligibly small (0.2%) loss in energy, observed with EnCache arises due to the use of profiling cache.

Fig. 5 shows saving in energy with 4MB baseline cache and its average value for EnCache and HDRI are 41.4% and 36.3% respectively. Average increase in simulation cycles for EnCache and HDRI are 4.39% and 9.2% respectively and average saving in EDP are 38.4% and 29.4% respectively. The average increase in MPKI by using EnCache and HDRI are 0.44 misses and 0.61 misses respectively and average ActiveRatio are 43.8% and 43.6% respectively.

Fig. 6 shows energy saving with 8MB baseline and its average value for EnCache and HDRI are 52.63% and 47.72%, respectively. The average increase in simulation cycles for EnCache and HDRI are 4.5% and 9.5% respectively and average saving in EDP are 50.02% and 41.67%, respectively. The average increase in MPKI by using EnCache and HDRI are 0.43 misses and 0.60 misses, respectively and average ActiveRatio are 36.7% and 35.9% respectively.

C. Further Analysis and Insights

Firstly, for both techniques, the saving in cache energy is large enough to offset the energy-cost of the algorithm (E_{algo}). At all three cache sizes, for both energy and EDP saving, EnCache performs superiorly to HDRI in terms of best-case, average-case and worst-case behavior.

With HDRI, benchmarks show best EDP for different values of $MissBound$ and some benchmarks show large variation in EDP saving with change in $MissBound$. For example, with 8MB baseline cache, the saving in EDP in *wupwise* increases from 4.5% to 45.5% when going from $\eta + 200$ to $\eta + 400$. Also, intra-program variations make the HDRI approach of using fixed value of $MissBound$ highly ineffective. This is evident from *parser* benchmark at 8MB baseline, where the loss in EDP is 59% even at $\eta + 200$ and even worse at other $MissBound$ values. Thus even a small offset of 200 misses leads to severe cache thrashing (Fig. 6).

EnCache allows a direct change in one state from any other state without having to go through intermediate state (e.g. $Full$ to $Eight$ without going through $Quarter$). Thus, whenever L2 WSS changes drastically, the EnCache algorithm directly reconfigures the cache to the most appropriate size. On the other hand, HDRI approach must go through all the intermediate configurations before reaching a desired configuration; and thus it incurs large reconfiguration overhead.

For different benchmarks, the impact of increased cache misses on energy is different. HDRI approach fails to capture this relationship since it works by trying to keep number of extra misses small and thus, it does not directly work to choose an energy-efficient configuration. On the other hand, EnCache optimizes directly for energy and captures the effect of increased misses on energy consumption. EnCache uses profiling cache to provide online profiling results for guiding reconfiguration, while the choice of suitable $MissBound$ (and hence η and offset) in HDRI scheme requires multiple simulation-runs in offline profiling. Given that the $TotL2Miss$ for different benchmarks varies over 3 orders of magnitude, choosing a benchmark-specific $MissBound$ is absolutely necessary with HDRI technique. Finally, EnCache can optimize based on the energy consumption in other components (such as main-memory) also, while the HDRI scheme is insensitive to the overall energy picture.

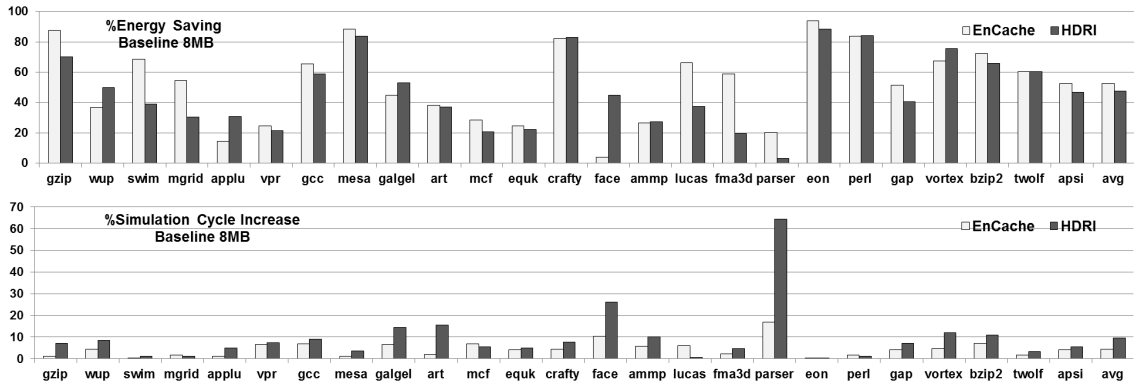

Fig. 6. Experimental Results with 8MB Baseline Cache (Here wup=wupwise, equk =equake, face=facerec, perl=perlbmk, avg=average).

VII. Conclusion

We have proposed EnCache, a novel scheme for saving leakage power consumption of last-level caches. It uses a system-level approach with lightweight hardware support. Using a novel, low-cost hardware component called profiling cache, system software can accurately predict memory-subsystem energy of a program for multiple cache-configurations. The dynamic performance monitoring allows controlling aggressiveness of reconfiguration and strike suitable balance between energy minimization and performance loss. The experiments performed show the superiority of EnCache over conventional energy-saving scheme. It is straightforward to extend EnCache to optimize overall system energy efficiency by merely including processor core energy in the energy model.

Acknowledgements

The authors appreciate the constructive comments from the anonymous reviewers. This work is supported in part by the National Science Foundation under grants CNS-0834476 and CNS-1117604.

References

[1] S. Borkar, "Design challenges of technology scaling," *Micro, IEEE*, vol. 19, no. 4, pp. 23 –29, jul. 1999.

[2] P. Shivakumar *et al.*, "Exploiting microarchitectural redundancy for defect tolerance," in *ICCD-21*, 2003.

[3] S. Yang *et al.*, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *HPCA*, 2002, pp. 151–161.

[4] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO-39*, 2006, pp. 423–432.

[5] S. Yang *et al.*, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches," in *HPCA-7*, 2001, pp. 147–157.

[6] R. Kessler *et al.*, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Trans. on Computers*, vol. 43, no. 6, pp. 664–675, 1994.

[7] S. Laha, J. Patel, and R. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems," *IEEE Transactions on Computers*, vol. 37, pp. 1325–1336, 1988.

[8] M. K. Qureshi *et al.*, "A case for MLP-aware cache replacement," in *ISCA-33*, 2006, pp. 167–178.

[9] S. Kaxiras *et al.*, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *ISCA-28*. ACM, 2001, pp. 240–251.

[10] J. Abella *et al.*, "IATAC: a smart predictor to turn-off L2 cache lines," *ACM TACO*, vol. 2, no. 1, pp. 55–77, 2005.

[11] H. Zhou *et al.*, "Adaptive mode control: A static-power-efficient cache design," *ACM TECS*, vol. 2, no. 3, pp. 347–372, 2003.

[12] K. Flautner *et al.*, "Drowsy caches: simple techniques for reducing leakage power," in *ISCA-29*, 2002, pp. 148–157.

[13] H. Hanson *et al.*, "Static energy reduction techniques for microprocessor caches," *IEEE Trans. on VLSI*, vol. 11, no. 3, pp. 303 –313, 2003.

[14] M. Powell *et al.*, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *MICRO*, 2001, pp. 54–65.

[15] J. Kin *et al.*, "The filter cache: an energy efficient memory structure," in *MICRO-30*, 1997, pp. 184–193.

[16] D. K. Tam *et al.*, "RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations," in *ASPLOS*, 2009, pp. 121–132.

[17] R. L. Mattson, "Evaluation techniques in storage hierarchies," *IBM Journal of research and development*, vol. 9, 1970.

[18] X. Wang *et al.*, "Achieving fair or differentiated cache sharing in power-constrained chip multiprocessors," in *ICPP-39*, 2010, pp. 1–10.

[19] S. Ramaswamy and S. Yalamanchili, "Improving cache efficiency via resizing+ remapping," in *ICCD-25*. IEEE, 2007, pp. 47–54.

[20] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[21] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *ASPLOS-10*, 2002, pp. 45–57.

[22] H. Zheng *et al.*, "Decoupled DIMM: building high-bandwidth memory system using low-speed dram devices," in *ISCA-36*, 2009, pp. 255–266.