CASHIER: A Cache Energy Saving Technique for QoS Systems

Sparsh Mittal, Zhao Zhang, Yanan Cao

CASHIER: A Cache Energy Saving Technique for QoS Systems

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

***Abstract*—With each CMOS technology generation, leakage energy has been increasing at an exponential rate and hence, managing the energy consumption of large, last-level caches is becoming a critical research issue in modern chip design. Saving cache energy in QoS systems is especially challenging, since, to avoid missing deadlines, a suitable balance needs to be made between energy saving and performance loss. We present CASHIER, a Cache Energy Saving Technique for Quality of Service Systems. Cashier uses dynamic profiling to estimate the memory subsystem energy and execution time of the program under multiple last level cache (LLC) configurations. It then reconfigures LLC to an energy efficient configuration with a view to meet the deadline. In QoS systems, allowed slack may be specified either as percentage of baseline execution time or as absolute slack and Cashier can work for both these cases. The experiments show the effectiveness of Cashier in saving cache energy. For example, for an L2 cache size of 2MB and 5% allowed-slack over baseline, the average saving in memory subsystem energy by using Cashier is 23.6%.**

***Keywords*-QoS systems, cache leakage energy saving, low power, online profiling, last level cache**

I. Introduction

As we are entering into an era of green computing, the primary objective in chip design is shifting from achieving highest peak performance to achieving highest performance-energy efficiency. In battery-powered mobile systems, such as cell phones and laptops, achieving energy efficiency is especially important, since these systems work on batteries which store limited energy. Moreover, since these systems also need to fulfill application quality-of-service (QoS) requirements [1, 2], a fine balance is required to meet the dual goals of energy saving and minimum performance loss.

For several reasons reducing energy consumption of LLCs in QoS systems remains a significant challenge. Firstly, as the applications are becoming computation-intensive [3, 4], the pressure on memory system is increasing and to mitigate this pressure, modern processors are using large size LLCs. Secondly, with shrinking CMOS feature size, leakage power has been increasing at an exponential rate [5]. Since leakage accounts for over 90% of the energy spent in LLCs [6], the energy consumption of LLCs is becoming a major fraction of chip energy consumption. Many existing techniques are designed to save the dynamic energy of cache, however, a large fraction of energy spent in LLCs is in the form of leakage energy, and thus, these techniques having limited utility in saving energy in LLCs. Further, the cache energy saving techniques which require offline profiling are difficult to scale and hence cannot be easily used in real-world QoS applications. Thus, saving cache energy in QoS systems is a challenging, yet important research issue and new techniques are required to effectively address it.

In this paper, we present Cashier, a Cache energy saving technique for quality-of-service (QoS) systems. Cashier uses a small microarchitecture component called "reconfigurable cache emulator" (RCE), which uses set sampling idea to estimate program miss rate for various cache configurations in an on-line manner. Additionally, Cashier uses CPI stacks to estimate program execution time under different LLC configurations. Using these estimates, the energy saving algorithm (ESA) estimates memory subsystem energy under different cache configurations. Then, a suitable cache configuration is chosen to strike a right balance between opportunity of energy saving and performance loss, thus making best possible efforts to not miss the deadline. For hardware implementation of cache line switching, Cashier employs the gated-V_{dd} scheme [7].

Cashier has several features which address the limitations of existing techniques. It uses low cost, non-intrusive, dynamic profiling technique which does not affect the operation of LLC. Also, Cashier optimizes memory subsystem (which includes LLC and main memory) energy, instead of merely LLC energy. Simulations have been performed using Sniper [8, 9] and workloads from SPEC2006 suite. The results show that Cashier is very effective in saving energy while still meeting most of the deadlines. For example, for 2MB L2 cache with 5% allowed performance slack, the average saving in memory subsystem energy is 23.6%.

II. Related Work

Recently, several researchers have proposed techniques for saving cache energy [10–12]. Mittal and Zhang [12] use selective sets and selective ways to reconfigure the cache for saving energy. However, their technique cannot be used for applications with deadlines. Further, the cache coloring technique used in our work provides much finer granularity of cache reconfiguration than the previous cache reconfiguration techniques (e.g. [11, 12]). Some researchers have presented techniques for saving cache energy while meeting deadlines [13, 14]. Wang and Mishra [14] use offline analysis to profile a large number of configurations of

two-level cache hierarchy and explore these configurations during run-time for finding the best configuration. However, since product systems execute trillions of instructions of arbitrary applications, offline profiling becomes infeasible for use in such systems.

Apart from cache reconfiguration, dynamic voltage/frequency scaling (DVFS) has also been used for saving energy while still meeting the deadlines (e.g. [15–18]). DVFS aims to save the dynamic energy of the processor, while Cashier aims to save the leakage energy of the processor. Thus, Cashier can be synergistically used with DVFS to save additional amount of energy.

III. System Design

Several real-world applications present soft real-time resource demands. In such applications, the task deadlines are usually more relaxed than the task completion time and as long as a task is completed by its deadline, the actual completion time does not matter from user's perspective. In such systems, Cashier can save leakage energy by using cache reconfiguration, while making best possible effort to meet the task deadline. For enabling cache reconfiguration, Cashier uses cache coloring technique (Section III-A). For estimating miss rates under different L2 configurations, it uses RCE and using CPI stack method, it estimates program execution time with those configurations (Section III-B and III-C). The energy saving algorithm (ESA) uses these values to estimate memory subsystem energy and finds the configuration with minimum energy and bounded performance loss (Section IV). We assume that LLC is L2 cache and based on this description, Cashier can be easily extended to case when LLC is L3 cache.

A. Cache coloring

To selectively and dynamically allocate cache to an application, Cashier uses cache coloring technique [19, 20] which works as follows. Firstly, the cache is logically divided into multiple non-overlapping bins, called cache colors. The maximum number of colors, N, is given by

$$N = \frac{CacheSize}{PageSize \times Associativity} \quad (1)$$

Further, the physical pages are divided into N *memory regions* based on the least significant bits (LSBs) of their physical page number. In Fig. 1, these bits are referred to as Region ID. Cache coloring maps a memory region to a unique color in the cache. For this purpose, Cashier uses a small *mapping table* (MT) which stores the cache color assigned to each memory region. By manipulating the mapping between physical pages and cache colors, Cashier allocates a particular cache color to a memory region and thus, all physical pages in that memory region are mapped to the same cache color.

Cashier works on the key idea that for restricting the amount of active cache, all memory regions can be allocated to merely few cache colors. Thus, the rest of the colors are effectively not utilized and can be turned off to save cache energy. This is implemented using the mapping table (MT). At any point of execution, if M ($\leq N$) colors are allocated to the application, the mapping table stores the mapping of N regions to M colors. Thus, Cashier reconfigures the cache at the granularity of a single cache color. A salient feature of this cache coloring technique is that, unlike previous approaches (e.g. [20]), it does not require a change in underlying virtual address to physical address mapping, and thus can be implemented with little overhead. We refer to "active" or "turned on" color, as one that stores data and consumes power normally. Also, an "inactive" color is one that has been "turned off" to save leakage energy and hence does not store data.

Figure 1 shows the flow diagram of Cashier with values from the following example. We assume a 2MB, 8-way L2 cache of 64B block size and a $PageSize$ value of 4KB. Then from Equation 1, we get N= 64 colors. Hence, in this case, MT has 64 entries, each 6-bits wide (Figure 1). Also note that the size of mapping table is small and hence, its access latency and energy consumption are negligible.

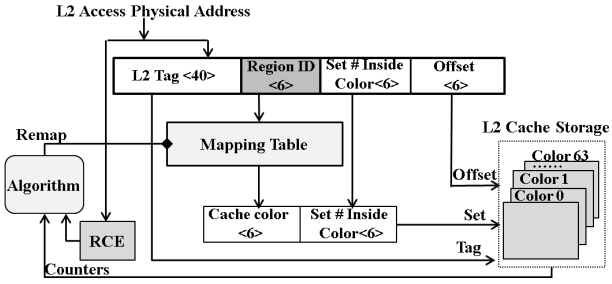

Figure 1. Cashier Flow Diagram (Using example of N= 64)

B. Reconfigurable Cache Emulator (RCE)

RCE builds on the idea of set sampling, which states that the miss rate characteristics of the cache can be estimated by sampling only a few sets [12, 21]. RCE uses profiling units, which are data-less (tag only) components, having the same replacement policy and associativity as that of the L2 cache it emulates. To estimate cache miss rate for each possible cache size[1], a separate profiling unit may be required. However, even with set sampling technique, this may lead to large overhead. To reduce this overhead, while still obtaining rich profiling information, Cashier profiles only selected cache sizes (called 'levels') and uses piecewise linear fit to estimate miss rates for other cache sizes. In this paper, we use a six-level RCE, each level (unit) profiling 1X/16, 2X/16, 4X/16, 8X/16, 12X/16 and 16X/16 sizes respectively, where X shows the size of L2 cache. The

[1]Note that since both associativity and block size are fixed, change in cache size simply means change in cache set-count.

reasoning behind the use of these profiling levels is that most of these levels profile a cache of power-of-two set-count and the level profiling 12X/16 size is chosen to get more uniformly spaced profiling levels between 8X/16 and 16X/16. This helps in obtaining more accurate miss rate estimates. Unlike previous schemes (e.g. [11, 12]), which only profile caches of power-of-two set-counts, the RCE design of Cashier has the ability to emulate caches of *non*-power-of-two set-counts also, using cache coloring scheme.

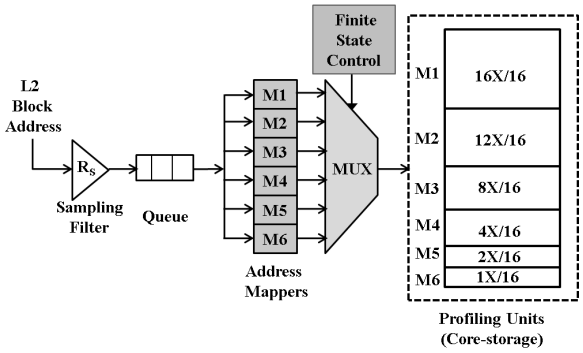

Figure 2. Reconfigurable Cache Emulator (RCE) Design

Figure 2 shows the design of RCE. First, the L2 accesses are sampled using a sampling filter, which uses a sampling ratio (R_S) of 64. Then, these addresses are fed to a queue (to avoid congestion) and then, using the mapping table corresponding to each profiling level, the set (index) value for each level is computed. Then, using a small multiplexer (MUX), the addresses are sequentially fed to the respective storage regions of the RCE for emulating cache access. RCE operates in parallel to L2 cache and does not lie in critical access path. We now calculate the size of RCE. Let the number of sets in L2 be Z, then the number of sets in the 16X/16 profiling unit is Z/R_S. Further, let T and B denote the tag-size and block-size in bits, respectively and S show the total number of sets in RCE. Let D denote the ratio of RCE size and L2 cache size. We have

$$S = \frac{(1+2+4+8+12+16)Z}{16R_S} \quad (2)$$

$$D = \frac{\text{RCESize}}{\text{L2CacheSize}} = \frac{43T}{16R_S(B+T)} \quad (3)$$

For T=40, B=64×8 and R_S=64, we get D=0.003 or 0.3%. Thus, the overhead of RCE is extremely small. We take this overhead into account in our energy model.

C. CPI Stack for Execution Time Estimation

For estimating program execution time under different L2 configurations, Cashier uses the CPI stack technique [8, 22]. A 'CPI stack' is a stacked bar that shows the different components contributing to overall performance. It presents base CPI and 'lost' cycle opportunities due to instruction interdependencies, cache misses etc., taking into account the possible overlaps between execution and miss events. Out of various components of CPI-stack, Cashier uses the memory stall cycle component, since the change in L2 configurations shows its effect on execution time in terms of change in memory stall cycles. We assume that, in an interval, memory stall cycles vary linearly with the number of load misses, and thus, their ratio, called SPM (Stall cycles Per load Miss), remains independent of the number of load misses themselves. Then, the stall cycles under any cache configuration can be computed by multiplying SPM with the number of estimated load misses with that configuration. Using stall cycle estimates and base CPI value from the CPI stack, the total number of cycles (and hence total execution time) under that configuration can be computed. These estimates are used for computing memory subsystem energy values (Section VI-B). Also, the execution time and energy estimates are used by ESA (Section IV).

If the number of load misses vary significantly between different cache configurations, the above mentioned linearity assumption does not hold well. However, as shown in Section IV, in an interval, Cashier only searches for configurations which differ from current configuration in a small number of active colors. Thus, the above assumption holds reasonably well and energy estimation accuracy is minimally affected.

IV. Cashier Energy Saving Algorithms

We now explain the energy saving algorithms (ESAs) of Cashier, which can run as kernel modules. We refer to 'baseline cache' as the full size cache which does not use cache reconfiguration or energy saving technique. We assume that the available slack can be specified in one of the two ways. First, the slack can be specified as extra time itself (T_{slack}), e.g. a T_{slack} value of 100μs denotes that an application can be slowed down by 100μs, without missing the deadline. This is called *Magnitude Slack Method* (MSM). Second, the slack can be specified as a percentage of extra time over baseline, denoted as Υ, e.g. Υ=3% denotes that an application can be slowed down by 3% and still it meets its deadline. This is called *Percentage Slack Method* (PSM). Both these methods have been used in previous studies [15, 20, 23, 24]. We now discuss the algorithms for each of these methods. A salient feature of Cashier is that *neither* of these two algorithms require *a priori* knowledge of the baseline execution time for their operation.

We first discuss the steps which are common to both the algorithms. In any interval i with $C_\star$ active colors; both the algorithms select those configurations as candidates which satisfy following two conditions. Firstly, to avoid thrashing, a configuration should have at least $N/16$ active colors. Secondly, to keep the reconfiguration overheads small, in any interval, only up to L ($L = 8$ in this paper) colors can be turned ON or OFF. If E denotes the set of configurations,

fulfilling these conditions, we have $E = \{C \mid (C_\star - L) \leq C \leq (C_\star + L)$ and $C \geq N/16\}$.

For explaining the algorithms, we define a quantify t_i, as follows. Using program execution time estimates, in every interval, the algorithms estimate the extra time, which the current configuration is taking over and above the baseline configuration[2]. Over all the intervals, the Algorithm accumulates these values. At the end of any interval i, this gives the estimate of increased execution time (t_i) due to ESAs (viz. PSM or MSM), *till that interval* i. Thus, t_i shows the amount of slack already exploited.

We now explain the algorithm-specific steps.

MSM Algorithm:

1) To be conservative, MSM Algorithm keeps a reserved slack of $T_{reserve}$ ($T_{slack}/10$ in this paper) and assumes an effective slack of T_{eff} =$T_{slack} - T_{reserve}$.
2) At the end of interval i, $(T_{eff} - t_i)$ shows the amount of slack remaining. Based on this, MSM Algorithm decides allowed *maximum absolute slack* (MAS_{i+1}) for interval $i+1$, e.g. if the remaining slack is 60μs, the Algorithm may choose to use MAS_{i+1} as 2μs.
3) The configurations having a slack greater than MAS_{i+1} are rejected from E. In effect, the configurations with number of active colors below a certain threshold color are rejected. We call this step as thresholding.
4) If $E \neq \phi$, then the configuration from E with minimum estimated energy is selected for interval $i+1$.
5) If $E = \phi$ then the configuration closest to the threshold, viz. $(C_\star + L)$ is chosen for next interval. This is to avoid possible oscillations due to sudden change in working set size of the application. Since the algorithm aims to meet a global deadline, and not per-interval deadline; by feedback adjustment, it compensates for positive or negative deviations from the allowed slack.

PSM Algorithm:

1) If the total execution time at the end of interval i is T_i, then $(T_i - t_i)$ gives the estimate of baseline time till interval i. Using this, Δ_i is calculated as follows:

$$\Delta_i = \frac{t_i \times 100}{(T_i - t_i)} \tag{4}$$

   Clearly, Δ_i gives the estimate of percentage of extra time taken by the PSM Algorithm over the baseline.
2) PSM Algorithm always tries to conservatively keep Δ_i below the actual allowed percentage slack (Υ), by a small margin δ (0.3% in this paper). Thus, $\Delta_i \leq \Upsilon - \delta$.
3) Based on Δ_i and Υ, Algorithm computes *maximum percentage slack* over the baseline for $i+1$. This is termed as MPS_{i+1} and represents the maximum percentage slack allowed in next interval. Then, to make performance aware choices, the configurations with estimated percentage slack greater than MPS_{i+1} are removed from E. Thus, in effect, the configurations with number of active colors below a certain threshold color are rejected. We call this step as thresholding.
4) If $E \neq \phi$, then the configuration from E with minimum estimated energy is selected for interval $i+1$.
5) If $E = \phi$ then the configuration closest to threshold, viz. $(C_\star + L)$ is chosen for next interval. The reason for this is same as explained above.

[2]Note that the execution time estimates for baseline cache configuration are also obtained in run-time using RCE and not in offline manner.

We now explain the MSM algorithm with a simple example and PSM can be similarly understood. Assume N=64 and L=8 and in any interval, $C_\star$=28. Then, initially, $E = \{20, 25...35, 36\}$. If MAS_{i+1} is such that the configurations with $C < 20$ give an absolute slack value greater than MAS_{i+1}, then all configurations in E pass thresholding step and the one with minimum energy is selected for next interval. However, if MAS_{i+1} were such that configurations with $C < 40$ were to be removed, then after thresholding step, $E = \phi$. In such case, the algorithm selects the configuration with 36 (i.e. $C_\star + L$) active colors, which is the closest to threshold. In the next interval, $C_\star$ becomes 36 and then depending on MAS_{i+2} and threshold-color, a suitable color value can be chosen.

V. Hardware Implementation

For cache block switching, we use the gated-V_{dd} scheme [7]. We assume a specific implementation of gated-V_{dd} transistor (NMOS gated V_{dd}, dual V_t, wide, with charge pump) which results in minimal impact on access latency, but 5% increase in the cell area [7]. We account for this overhead in our energy model (Section VI).

Cashier ESA runs after a large interval length (e.g. 5M instructions). Cache reconfiguration changes the mapping of memory regions to cache colors. During such time, when a color is to be turned off, its dirty data is written back to memory and the clean data is discarded; and then the cache color is turned off. When a color is turned on, some regions are mapped to this color and thus, this color starts storing data. The cache reconfiguration scheme of Cashier may introduce a one time overhead but is simple and has less overhead than the previous techniques (e.g. [11, 12]).

VI. Experimental Methodology

A. Simulation Environment and Workload

We conduct out-of-order simulations using interval core model from Sniper simulator [8, 9]. The processor frequency is 2 GHz and ROB size is 128. Dispatch width is 4 micro-operations. Each of L1I and L1D is 4-way 32KB, LRU cache with 1ns latency. The L2 is 8-way 2MB, LRU cache with 6ns latency. All caches use a block size of 64B. Main memory latency is 70ns with peak bandwidth of 8GB/s and memory queue contention is also modeled. The interval length is 5M instructions. To simulate the representative behavior of

SPEC2006, while still limiting the simulation time, we use 12 benchmarks from this suite, as suggested by Phansalkar et al. [25] based on their multivariate statistical data analysis. These 12 benchmarks, 6 each from integer point (gcc, hmmer, libquantum, mcf, sjeng, xalancbmk) and floating point (cactusADM, lbm, milc, povray, soplex, wrf) benchmarks, represent the behavior of entire SPEC2006 suite [25]. We use *ref* inputs. We fast-forwarded each benchmark for 10B instructions and then simulated for 1B instructions.

B. Energy Modeling

We take into account the energy spent in L2 cache, main memory and the cost of executing the algorithm (E_{Algo}), since other components are minimally affected by our approach. Note that for baseline experiments, $E_{Algo} = 0$.

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

Here energy spent in L2 and memory is composed of both leakage and dynamic energy. Further, 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, memory, RCE). To calculate L2 energy, we assume that an L2 miss consumes twice the energy as that of an L2 hit [10, 12]. The leakage energy is proportional to active area of the cache [11, 12]. Thus,

$$E_{L2} = E^{dyn}_{L2} \times (2M_{L2} + H_{L2}) + (P^{leak}_{L2} \times Time \times C)/N \quad (6)$$

Here N shows the total number of colors and for any interval with C active colors, M_{L2} and H_{L2} show the corresponding number of L2 misses and L2 hits respectively and $Time$ shows time consumed in the interval. The L2 energy values are obtained using CACTI [26] for 4-bank caches at 45nm technology. For 2MB L2 cache, we get E^{dyn}_{L2}=0.985 nJ/access and P^{leak}_{L2}=1.568 Watt. To account for the increased area due to use of gated-V_{dd} technique, we assume 5% higher value of P^{leak}_{L2} for Cashier, but not for baseline cache (Section V).

To calculate memory energy, we note that E^{dyn}_{mem}=70 nJ and P^{leak}_{mem}=0.18 Watt [12, 27]. Using A_{mem} to denote the number of memory accesses, we get

$$E_{mem} = E^{dyn}_{mem} \times A_{mem} + P^{leak}_{mem} \times Time \quad (7)$$

Using A_{RCE} to denote the number of RCE accesses and E_{Tran} to denote block-transition energy, we get

$$E_{Algo} = E^{dyn}_{RCE} \times A_{RCE} + P^{leak}_{RCE} \times Time + E_{Tran} \quad (8)$$

To calculate RCE energy consumption, we use Equation 2 and take power-of-two upper bound of S as $S = 64Z/16R_S$. We estimate energy using CACTI for a single bank structure, with 8B block size and count energy consumption of tag arrays only, since RCE is a tag-only structure. For an RCE corresponding to 2MB L2, we get E^{dyn}_{RCE}=0.004 nJ/access and P^{leak}_{RCE}=0.007 Watt. Since for every 64 L2 accesses, RCE is accessed only 6 times, RCE energy consumption is a very small fraction of L2 cache energy consumption. Each block transition is assumed to take 0.002 nJ. Using $Tran$ to denote the total number of blocks transitions, we get

$$E_{Tran} = 0.002 \times Tran \text{ nJ} \quad (9)$$

VII. Results and Analysis

We now present the results of evaluation of Cashier. Note that we evaluate Cashier under much more strict deadlines than that used by previous works (e.g. [15]). For brevity, we use the names cactus, libquant and xalan to denote cactusADM, libquantum and xalancbmk, respectively.

A. Magnitude Slack Method

For evaluating MSM, we need to assign a randomly chosen slack to each application, which is neither too high, nor too low. Hence, we use two tests which assign slacks randomly, while still ensuring reasonably strict deadlines and evaluation. In first test, we generated a list P of 12 random numbers in the range of $[0, 1]$, using on-line random number

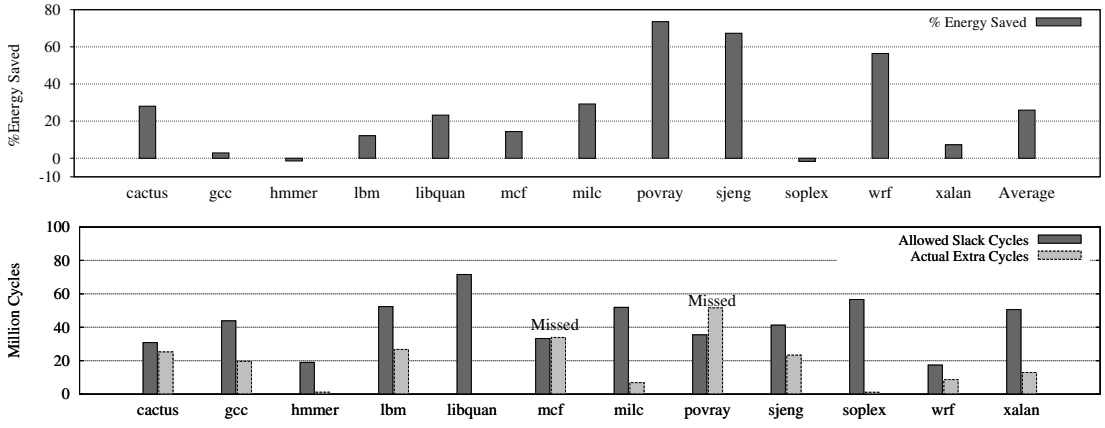

Figure 3. Results with Magnitude Slack Method: Percentage Energy Saving and Simulation Cycle Increase (mcf and povray miss their deadlines)

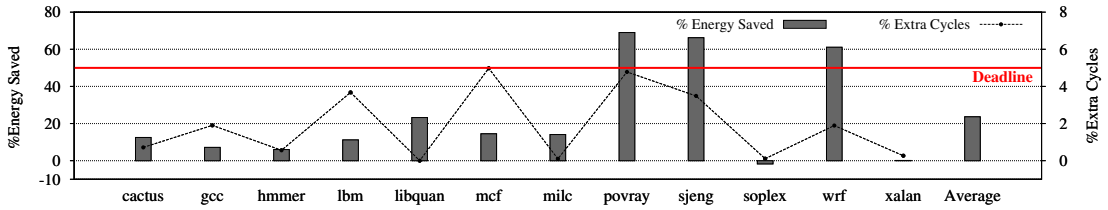

Figure 4. Results with Percentage Slack Method: Percentage Energy Saving and Percentage Simulation Cycle Increase for $\Upsilon = 5\%$ (No benchmark misses the deadline)

generation utility [28] and then calculated $(4 + p_i)\%$ of baseline simulation cycles, where $p_i \in P$, $i = \{1, 2..12\}$. We then set it as a T_{slack} for MSM algorithm for each of the 12 benchmarks. Figure 3 shows the results. The average saving in energy over baseline cache is 25.9%, and for two benchmarks (mcf and povray), the deadline is missed.

To test MSM under arbitrary slack value, we use a second test. We take baseline simulation cycles of all benchmarks and sort these values in ascending order. We then take the mean of two medians, and set 5% of this value as T_{slack} for *all* the benchmarks. In our experiments, T_{slack} value was 46.081M cycles. Using this, we observe 26.8% saving in energy, and two benchmarks (cactusADM and povray) miss the deadline (figure omitted due to space limitation).

We further test MSM, as outlined in first test, but this time with $(4+q_i)\%$ of baseline simulation cycles, where $q_i \in Q$, $i = \{1, 2..12\}$ and Q is another randomly generated list. We get average energy saving of 25.8% and two benchmarks (mcf and povray) miss the deadline (figure omitted).

B. Percentage Slack Method

Figure 4 shows the percentage energy saved over a baseline cache, for percentage slack, Υ=5%. The average saving in energy is 23.6% and none of the benchmarks misses the deadline. For Υ=3%, the average saving in energy is 22.4% and two benchmarks (lbm and mcf) miss their deadlines and for Υ=7%, the average saving in energy is 25.0% and no benchmark misses its deadline (figures omitted).

VIII. Conclusion

In this paper, we presented Cashier, a dynamic reconfiguration based cache energy saving approach for QoS systems. Cashier achieves a right balance between the opportunity of energy saving and performance loss and fully adapts itself according to the available slack to maximize energy saving. Thus, Cashier saves energy with a small and bounded performance loss and may allow using a larger cache for the same energy budget to obtain even higher performance.

Acknowledgements

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

References

[1] W. Yuan *et al.*, "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," *ACM SIGOPS OSR*, 2003.
[2] J. Li *et al.*, "Real-time constrained task scheduling in 3d chip multiprocessor to reduce peak temperature," in *EUC*, 2010.
[3] S. Khaitan *et al.*, "A class of new preconditioners for linear solvers used in power system time-domain simulation," *IEEE TPS*, 2010.
[4] A. Pande *et al.*, "BayWave: Bayesian Wavelet-based Image Estimation," *IJSISE*, vol. 2, no. 4, 2009.
[5] http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf.
[6] H. Homayoun *et al.*, "Adaptive techniques for leakage power management in L2 cache peripheral circuits," in *ICCD*, 2008, pp. 563–569.
[7] M. Powell *et al.*, "Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories," in *ISLPED*, 2000, pp. 90 – 95.
[8] T. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," *Supercomputing*, 2011.
[9] http://snipersim.org.
[10] H. Hanson *et al.*, "Static energy reduction techniques for microprocessor caches," *IEEE Trans. on VLSI*, 2003.
[11] S. Yang *et al.*, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *HPCA*, 2002.
[12] S. Mittal *et al.*, "EnCache: Improving cache energy efficiency using a software-controlled profiling cache," in *IEEE EIT*, May 2012.
[13] J. Chi *et al.*, "Cache leakage control mechanism for hard real-time systems," in *CASES*, 2007, pp. 248–256.
[14] W. Wang *et al.*, "Dynamic reconfiguration of two-level caches in soft real-time embedded systems," in *ISVLSI*. IEEE, 2009, pp. 145–150.
[15] A. Weissel and F. Bellosa, "Process cruise control: event-driven clock scaling for dynamic power management," in *CASES*, 2002.
[16] R. Jejurikar *et al.*, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *DAC*, 2005.
[17] P. Pillai *et al.*, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *ACM SOSP*, 2001, pp. 89–102.
[18] K. Choi *et al.*, "Off-chip latency-driven dynamic voltage and frequency scaling for an mpeg decoding," in *DAC*, 2004, pp. 544–549.
[19] R. Kessler *et al.*, "Page placement algorithms for large real-indexed caches," *ACM TOCS*, vol. 10, no. 4, pp. 338–359, 1992.
[20] J. Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," 2008.
[21] M. K. Qureshi *et al.*, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006, pp. 423–432.
[22] S. Eyerman *et al.*, "A performance counter architecture for computing accurate CPI components," in *ASPLOS*. ACM, 2006, pp. 175–184.
[23] N. Roy *et al.*, "Toward effective multi-capacity resource allocation in distributed real-time and embedded systems," in *ISORC*, 2008.
[24] P. White, "Rsvp and integrated services in the internet: A tutorial," *IEEE Communications Magazine*, vol. 35, no. 5, pp. 100–106, 1997.
[25] A. Phansalkar *et al.*, "Subsetting the SPEC CPU2006 benchmark suite," *ACM SIGARCH CAN*, vol. 35, no. 1, pp. 69–76, 2007.
[26] CACTI 5.3, http://quid.hpl.hp.com:9081/cacti/.
[27] H. Zheng *et al.*, "Decoupled DIMM: building high-bandwidth memory system using low-speed dram devices," in *ISCA*, 2009.
[28] http://www.random.org/decimal-fractions/.