Electronic Preprint for Journal of Information Processing Vol.27

Regular Paper

A Low-power Shared Cache Design with Modified PID Controller for Efficient Multicore Embedded Systems

Huatao Zhao[1,2] Jiongyao Ye[3] Takahiro Watanabe[1]

Received: May 14, 2018, Accepted: November 7, 2018

Abstract: Nowadays, on-chip cache scales are oversized in multicore embedded systems, and those caches even consume half of the total energy debit. However, we observe that a large portion of cache banks are wasted, meaning that those banks are rarely used but consume a great deal of energy during their entire lifetime. In this paper, we propose a controllable shared last level cache (SLLC) scheme to dynamically trace cache bank demands for each thread. Thus, energy on useless banks can be largely saved. Specifically, we (1) propose an effective cache bank allocating policy to explore bank demands corresponding to executed applications, (2) discuss the interrelations between application locality change and bank demands for further energy saving and (3) represent a modified PID controller to generate optimal banks for each thread. Experimental results show that our controllable SLLC design can save on average 39.7 percent shared cache access energy over conventional cache, while its performance is slightly improved and additional hardware overhead is less than 0.6 percent.

Keywords: Cache optimization, PID control, Low power, Embedded system, Chip multicore

1. Introduction

On-chip cache hierarchies which are used to balance the access speed of processor and memory, are rapidly increasing in scale as the number of on-chip cores is increasing. Typically, shared last level cache (SLLC) performs the key task of caching costly off-chip memory accesses, but its size in a single die tends to occupy a large portion of both on-chip area and energy consumption. Hence, optimizing energy saving of SLLC is a key issue for both high computing density and low energy consumption under given constraints.

Many previous researches illustrate that common SLLCs provide effective multicore sharing, but they are inefficient, because that great majority of their cache parts (i.e., cache lines) are rarely accessed before evicted [1], [8]. Those parts provide very few cache hits during their entire lifetime, but consume a great deal of energy for keeping hot. An effective method to reduce such energy wasting is to set the useless parts of SLLC into low-power mode, meaning that those parts can be shut down for energy saving [6]. However, the efficiency of such method is highly relying on how accurate it can locate useless parts, Otherwise, it will lead to extra off-chip memory accesses which have expensive access latencies. Inevitably, previous studies employ searching algorithms to find suitable parts corresponding to applications [9], [20]. And many efforts focusing on energy saving have been done with different angles. Researches aim at catching the locality with phase cache design [11], [18], or using complex algorithms for accurate searching results [12], or OS-level cache allocating [13], [19], and so on.

In this work, we introduce a new controllable cache approach to manage the SLLC efficiently. Based on experimental profiling of hit rate and energy values on SLLC, we discover that access energy is highly correlated with allocated cache bank numbers, and then a new bank allocating policy is constituted to find optimal bank numbers for each thread instead of clueless bank searching. Moreover, we study locality features of embedded applications, and propose locality-aware control intervals based on runtime appearances of hot subroutines. Finally, we efficiently integrate PID control method with our cache design to control runtime bank allocating, and the control results are converging to the optimal (low energy) SLLC banks corresponding to each thread. As a result, all threads are allocated with their desired SLLC banks in every interval synchronously. Thus, our controllable cache can work with low energy consumed.

The rest of this paper is organized as follows: Section 2 shows the motivation of this work, including analysis of pre-experiments. Section 3 describes our controllable cache design. Sections 4 and 5 show experimental results and conclusions.

2. Motivation

To study runtime cache bank demands of each thread, we employ *gcc* benchmark executed as an example in our experimental platform [16]. By analyzing the pre-experimental results, we observe two key appearances to motivate our proposed approach: (1) many allocated SLLC banks are useless but consume a great deal of energy (**Fig. 1**), and (2) demands of SLLC banks are shifting largely due to application locality change (**Fig. 2**).

[1] The Graduate School of IPS, Waseda University, Kitakyushu, Fukuoka 808–0135, Japan
[2] Jiangxi University of Science and Technology, Ganzhou, China
[3] East China University of Science and Technology, Shanghai, China

This work is partly supported by GJJ180486.

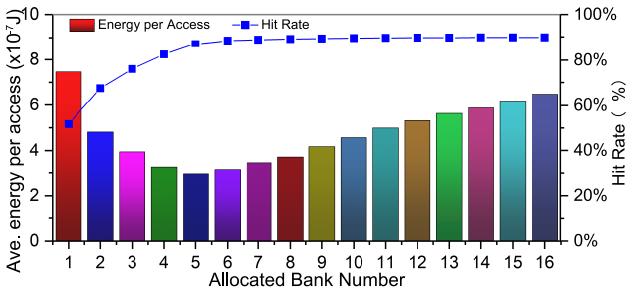

Fig. 1 Energy per access and hit rate varying with allocated SLLC bank number ranging from 1 to 16. All tests use *gcc* benchmark. 128 KB size per bank.

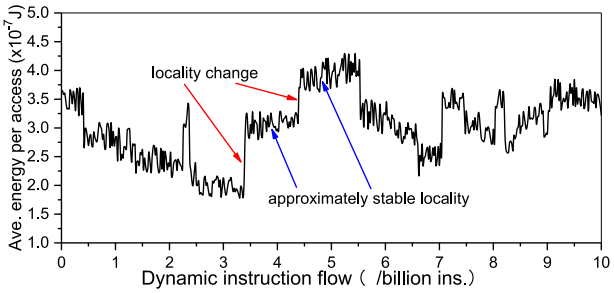

Fig. 2 Average energy per access sampling in ten billion instructions. *gcc* benchmark, fixed five banks allocated.

2.1 The Proportion of Active SLLC Banks is Small

Figure 1 shows average hit rate and energy per access values varying with allocated SLLC bank number which ranges from one to sixteen. For each test group, we run ten billion instructions and take samples at every ten million instructions. And then we calculate the runtime hit rate and energy values on SLLC. In this paper, we define that energy per access is calculated as total SLLC energy which is divided by the numbers of SLLC calls, and the total energy includes static energy consumed by SLLC during off-line memory accesses (SLLC miss repairing).

We highlight two key appearances in Fig. 1: (1) in the first several banks, hit rate increases very fast, while energy per access decreases sharply. (2) in the last several banks, hit rate remains unchanged, but energy per access is rising. To analyze those appearances, we deduce that thread starving happens in the first several banks, which results in many costly off-chip memory accesses. Once more cache banks are added, energy per access decreases sharply because of great hit rate improvement and also energy is largely saved. But in the last several banks, adding more banks only causes more static energy consumption while hit rate remains unchanged. As a consequence, those appearances motivate us to find suitable cache bank number (i.e., five banks) with low energy consumed.

2.2 Bank Demand Shifting with Application Locality

In this section, we try to represent that bank demand may be shifting with application locality. In other words, the minimum energy (optimal) allocated bank number corresponding to its locality may be different among sampling periods. Here, we employ an experimental example to represent such demand shifting. Firstly, we fix the allocated bank number as five to run *gcc* benchmark in our platform. And then we can simulate ten billion instructions in each thread, while we take samples at every fifty million instructions and calculate the average energy per access value for each sample (see Section 4.1 for details). As shown in Fig. 2, we can observe two key appearances: (1) Locality changing happens frequently, and results in sharp energy value variation, i.e., energy value changes in 3.4-th point of horizontal axis. (2) Locality appearing as energy variation in the figure may be approximately stable in a long sampling period, i.e., the sampling period from 4.5-th to 5.3-th. Meanwhile, we should note that such locality variation may be caused by many reasons, due to less or more reused data accesses in new period, cache read or write proportion changing, and so on, thereby hangs a tale [21], [22]. Hence, it is very difficult or even impossible to unify a single reason on locality changing for one application.

Actually, we can make use of those appearances in another angle: instead of pursuing those reasons of locality variation hardly, we represent that energy values change greatly with different fixed bank numbers, as shown in Fig. 1. And we consider that those energy values during same period (i.e., from 4.5-th to 5.3-th) but in different instruction flows (16 flows in total) are quite different, while there only has one variable quantity (fixed bank number) changed. So we can infer that the energy value with five banks allocated, for example, must be quite different with the rest fifteen energy values even they are in same sampling period (similar locality too). In other words, there should have an minimum energy bank number (may not be five banks here, decided by feedback control result) during the period between 4.5-th and 5.3-th. Hence, energy saving problem is shifted to find every energy-lowest bank number corresponding to each locality. Through spreading the above period to entire instruction flow, we are motivated to employ controllable allocation method to rapidly converge the allocated bank number into the minimum energy value after a locality variation happens. That is to say, we try to find the minimum energy bank numbers for every sampling period particularly with controllable allocation. Moreover, we need the long locality-stable period for applying our control method. Actually, based on the second appearance, there are many stable periods which can be gathered together to hold the majority of entire instruction flow.

3. Controllable SLLC Design

3.1 System Architecture Design

Starting with a conventional static non-uniform cache architecture (sNUCA) [4], we design our controllable cache through modifying cache scheduler with control loops. Note that sNUCA cache works well in small but fast bank level distributing [15], as a result, we reasonably map SLLC banks to threads at per-bank granularity. Hence, our goal of low energy is converted into dynamically adjusting mapped banks to threads with the help of control loop. To actualize this loop in SLLC, as shown in **Fig. 3**, we design three key components: (1) Metric evaluator, (2) Intensive cache controller, and (3) Cache co-scheduler. The metric evaluator is designed for counting runtime information, and then calculates desired metric values (see details in next subsection). In this paper, we define an interval as a number of clock cycles (i.e., one million cycles as an interval) or a piece of dynamic instruction flow (i.e., dynamic instructions in a call of one

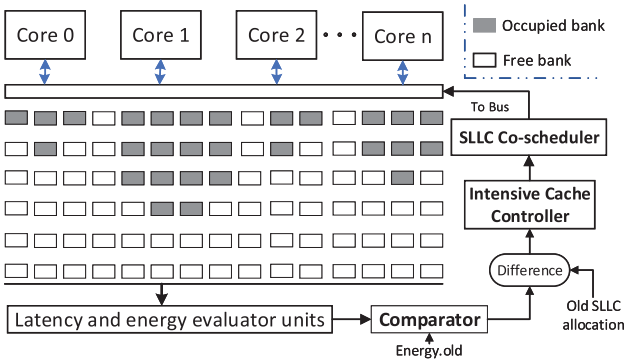

Fig. 3 Controllable SLLC architecture. Contains a feedback based control loop.

subroutine). If one interval is executed, we can do SLLC access energy difference between current interval and the former control interval. And the difference value is acting just as a feedback control signal to indicate demand of cache banks. Then, our intensive cache controller can generate suitable bank increment rapidly through applying modified discrete PID control, and the control results are converged to bank number which is corresponding to the lowest energy under current locality. Finally, we use cache co-scheduler to coordinate the allocation of cache banks in multi-thread parallel, and then one control loop ends here. Consequently, our design can save much energy with optimal banks allocated in most intervals.

3.2 Energy and Latency Evaluator

To provide feedback values for SLLC control, we first need to trace runtime information change once in one interval. Note that straightly achieving such information is difficult and costly in modern embedded systems, but actually, our goal is to quantize such information change corresponding to allocated bank number change in per-interval granularity. In other words, estimating such information is more suitable for control purpose rather than precise measuring. Hence, we count runtime events (i.e., hit numbers) in one interval, and calculate them with energy and latency models in established technical level. Based on sNUCA model in CACTI v6.5 [4], the average latency and energy of one shared cache access can be written as follows:

$$L_{PerAccess} = P_{hit} * (max(L_{tag}, L_{data}) + L_{outdriver} + L_{PreChaege}) + P_{mis} * (L_{tag} + L_{memory}) \quad (1)$$

$$E_{PerAccess} = P_{hit} * (E_{dyn-hit} + E_{leakage-hit}) + P_{mis} * (E_{dyn-mis} + E_{leakage-mis}) \quad (2)$$

Where L_{tag}, L_{data}, $L_{outdriver}$ and $L_{PreCharge}$ represent per access latency in tag side, data side, multiplexor driver and pre-charge operation respectively, and L_{memory} depicts the off-chip access latency. P_{hit} and P_{mis} describe the probability of cache hit and miss in one access respectively. $E_{dyn-hit}$ and $E_{leakage-hit}$ represent the per access dynamic energy and leakage energy in cache hit scenario. $E_{dyn-mis}$ and $E_{leakgae-mis}$ represent the per access dynamic energy and leakage energy in cache miss scenario including energy on waiting off-chip access. Note that all the above parameters are calculated in detailed gate level analytical model which covers all cache components. Fortunately, we can get those parameters through implanting CACTI tool in our test system. And then latency and energy consumption in one interval can be calculated with runtime information counted, while very little hardware overhead can be ignored.

3.3 Intensive Cache Controller Design

For a runtime sequence S, we split S into $\{1, 2, \ldots N\}$ intervals orderly where $n \in \{1, 2, \ldots N\}$ stands for the n-th interval. Based on runtime energy and latency evaluations, we can achieve energy and latency values of application x in n-th interval, denoted as $E_x[n]$ and $L_x[n]$. If energy per access is chosen as the controlled volume, we can calculate the difference values of application x in each interval, denoted as follows.

$$e_x[n] = E_x[n] - E_x[n-1], \quad n \in \{2, \ldots N\} \quad (3)$$

As shown in Fig. 1, energy per access values are highly related with allocated cache banks. Once a large $e_x[n]$ appears in n-th interval, which is caused by locality change, current allocated bank number is inefficient also. Thus, the goal of our control design is shifted to dynamically adapt bank numbers for new locality. Note that Fig. 1 shows a similar concave-curve of energy values along with cache bank increasing. As a consequence, we can treat those intervals as sampling periods in control theory, and then control the allocated bank number through evaluating energy change. Similar with the closed-loop feedback control model [3], the bank increment value $\triangle C_{bank}$ (integer value) can be represented with energy difference values in our negative feedback based discrete retroaction control model. And our increment Proportional−Integral−Derivative (PID) based intensive cache controller is represented as follows:

$$\triangle C_{bank} = K_P * (e_x[n] - e_x[n-1]) + K_I * e_x[n] + K_D * (e_x[n] - 2e_x[n-1] + e_x[n-2]) \quad (4)$$

Where K_P is the proportional ratio, K_I and K_D are the period-dependent integral ratio and derivative ratio, respectively.

To effectively tune those PID parameters, we should take a full consideration on several features of our control system: (1) We employ intervals to act as the sampling periods, and each energy per access value from its corresponding interval acts as the controlled volume value. Then, we can achieve a series of discrete controlled volume values and their corresponding sampling periods (intervals) to be controlled. (2) Our control outputs are not convergent to an exclusive bank value, but convergent to some bank values that each value is highly related on the concomitant locality. (3) Plenty of hardware and computing time are needed to achieve very accurate control outputs. Instead, we tend to sacrifice some accuracy for using minimal hardware and computing time in our control system. Note that all control outputs are even normalized to be integer bank numbers, so that there is a trade-off between control accuracy and hardware overhead, and energy saving is the primary goal in our control design. Based on above considerations, many methods such as self-tuning method are not suitable in our control system, because of their complex implementing structures [3], [23]. As it is difficult to achieve these parameters directly, we employ a static tuning method for deciding

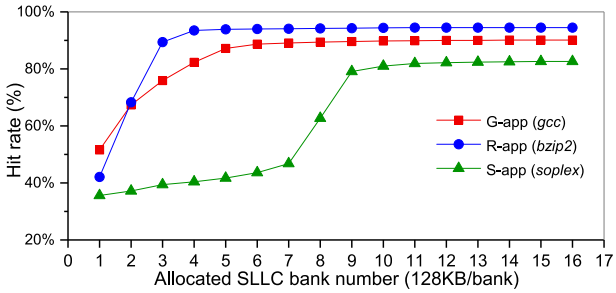

Fig. 4 Hit rate profiling along with allocated bank increasing. Bank number varies from one to sixteen.

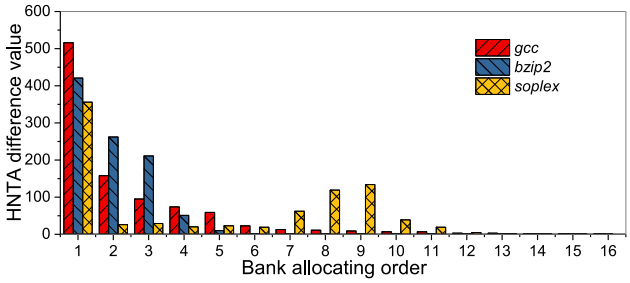

Fig. 5 HNTA difference profiling along with bank allocating order. The number of horizontal axis stands for the ordinal number.

the values of PID parameters through experimental searching.

Firstly, we set K_D and K_P to be zero, and then run a group of tests with the value of K_I, which changes from zero to k under the granularity of 0.05 (cut-and-try value), where k is an experiential constant and is assigned as five in our tests, so that there are one hundred test combinations in this group. Meanwhile, we compare the achieved average energy values and find an approximate K_I, which is corresponding to the minimum energy value among current group of tests. Secondly, we set K_I as the approximate K_I and K_D as zero, and then run a group of tests with K_P value varying from zero to k under the granularity of 0.01. After that, we can decide the first parameter K_P through energy comparing. Thirdly, we set K_P as the decided value and K_D as zero, and run a group of tests again with K_I varying to decide the parameter K_I. Finally, we set K_P and K_I as the decided values, and run a group of tests with K_D varying under the granularity of 0.05 to decide the parameter K_D. Note that each test combination only takes about dozens of seconds, and the total time of eight hundred test combinations added up is less than ten hours on tuning parameters for a single benchmark, so that the cost of this one time job is acceptable for embedded systems.

Note that our controller uses incremental PID method to greatly simplify calculation circuits, thus required area and energy overhead are ignorable. In other word, $\triangle C_{bank}$ is related to three latest energy difference values during four intervals, so that fine control effect can be easily achieved through control parameter tuning, and control results are tolerable with steady state error.

3.4 Dynamic Cache Co-scheduling

In this section, we draw fairness guarantee into our cache co-scheduler design. The original intention on fairness is to prevent thread starving [14], for example, hot threads will occupy most SLLC banks with LRU replacement policy, causing inefficient thread starving in the rest of threads. However, absolute fairness in SLLC allocating just brings about low efficiency on bank-desired threads. In fact, bank demand of one thread is highly relevant to application behaviors [11]. Thus, we employ three SPEC benchmarks as examples to show that hit rate curves change with allocated bank increasing. As shown in **Fig. 4**, hit rate in bzip2 shows great improvement in first three banks and then remains unchanged despite of bank increasing. But for other benchmarks, hit rate curves are diverse in both range ability and steady values although same bank number is allocated. Those appearances inspire us to classify applications into three types: (1) Rapid increasing application (R-app), i.e., *401.bzip2*, (2) Gradual increasing application (G-app), i.e., *403.gcc*, and (3) Sharp increasing application (S-app), i.e., *450.soplex*. And those application types are determined through experimental selections as follows.

As to an application x, we first run the application in our platform with allocated bank number ranging from one to N respectively, and then we can calculate the average Hit Numbers per Thousand Accesses value (HNTA), expressed as H_n ($n \in \{1, 2, \ldots N\}$). Consider that the $(n-1)$-th test has the same experimental conditions as the n-th test except the difference on allocated bank number, so that the difference of HNTA values between H_n and H_{n-1} seems to be only related to the additional cache bank and such difference, expressed as h_n, can be formulated as follows:

$$h_n = H_n - H_{n-1} \qquad H_0 = 0, \quad n \in \{1, 2, \ldots N\} \tag{5}$$

where h_1 represents the HNTA value in allocated one bank case, and the rest of h_n values can represent the improvement of HNTA when the n-th bank is allocated. In other word, h_n can be regarded as the potential hit improvement corresponding to n-th bank. As shown in **Fig. 5**, we plot the HNTA difference values of three benchmarks as examples to reveal such improvement. Clearly, we can observe that (1) Among the improvement values of *bzip2* benchmark, first four banks contribute the majority of hits while the rest of banks from fifth to sixteenth contribute very few hits. (2) The improvement values of *gcc* benchmark are always increasing with more banks, even the rate of such increment in last several banks is much lower than the one in the first several banks. (3) As the reuse distance of *soplex* benchmark is large, many hits can be achieved when allocated bank number exceeds such distance [21]. Hence, we employ an evaluation metric to estimate the application types based on above appearances in improvement values. And such metric named as M_x (x stands for application x) is formulated as follows:

$$M_x = \frac{\sum_{n=1}^{N} h_n - \sum_{n=1}^{N/2} h_n}{\sum_{n=1}^{N} h_n - h_{min}} \tag{6}$$

Where h_{min} here is set as h_1. And if M_x exceeds a threshold T_{S-app} ($M_x > T_{S-app}$), we can characterize the application x as S-type application. If M_x is smaller than T_{S-app} and is larger than a threshold T_{G-app} ($T_{S-app} > M_x > T_{G-app}$), we characterize the application x as G-type application. And if M_x is smaller than T_{G-app} ($M_x < T_{G-app}$), very few hits can be achieved with allocated banks after $N/2$-th bank, so that we characterize the application x as R-Type application. Note that the threshold values (T_{S-app} and T_{G-app}) are specially used for our platform and can

be explored by sensitive analyses on the SPEC benchmark set. Here, we set T_{G-app} = 0.06 and T_{S-app} = 0.27 which can be employed to classify all the benchmarks into three types clearly.

For R-app, we should ensure first several banks allocated and then is inclined to add few banks. For G-app, if we allocate or reduce several banks from stead values, energy change is in a relative small amplitude. And for S-app, bank numbers allocated should exceed its sharp point (nine banks for *soplex*) at least. Consequently, we set up the minimum bank numbers for each application through off-line training to efficiently prevent thread starving. After that, the priority order of bank allocation should be R-app first, S-app second and G-app last. Then we can adjust the control amplitude on bank allocating by reasonably setting proportional control ratios K_P, K_I and K_D (details see Section 3.6). In an extreme case, bank demands of all applications exceed available SLLC banks, we can reduce some banks of G-app with few performance overhead. And in most cases, our co-scheduling policy can well balance the trade-off between fairness and efficiency based on application behavior analyses and dynamic runtime control.

3.5 Locality-aware Interval Design

A simple method to generate control intervals is to divide entire instruction flow into equal parts, or to employ a certain amount of clock cycles as one interval [6], [9]. Those methods can be easily actualized in control system and has the advantage of synchronous control for all threads, meaning that all threads can share same control intervals if we endow each thread with independent control hardware. However, such division method loses sight of tracing application locality change (shown in Fig. 2) which may cause acute vibration within control intervals. To ensure control stability, we should ease the interference causing by locality change. Thus, we try to find more stable intervals which can closely trace locality change. Inspired by SPEC CPU2006 benchmark analysis [5], dynamic instructions from several subroutines called frequently can take the majority of total instructions while static instructions counted in those subroutines are very few, meaning that there may be thousands of calls in one hot subroutine and each call in same subroutine represent very similar locality as well as application behaviors. In other words, continuous calls on same subroutine show stable locality in a period, and calls switched on other subroutines may bring sharp locality change. And we choose hot subroutines for all benchmarks based on runtime selection, and those hot subroutines are repeatedly called during executing, as a result, dynamic instructions counted in those calls can take more than ninety percent of all instructions (as listed in **Table 1**). Thus, those subroutine calls cover majority of instruction sequence and are more suitable to act as control intervals for tracing locality change rather than equal-cycle based intervals. To implement our control intervals, we insert interval start and end marks into subroutine source codes to indicate control intervals.

3.6 Modified Discrete PID Control Algorithm

As shown in algorithm 1, we represent the pseudo-code of our modified PID control design. If a trigger mark of one interval is detected, runtime information between this mark and the former mark can be counted as values in n-th interval, then we can calculate $E_x[n]$ and $R_{hit}[n]$. Next, required bank increment $\triangle C_{bank}$ is calculated with incremental PID formula (from lines 1 to 6). In case of steady locality, energy values are very similar so that $\triangle C_{bank}$ is almost equal to zero. And in case of large locality change, if hit rate of n-th interval is increased, such appearance represents that cache demand in n-th interval is smaller than the one in last interval, in other words, previous allocated bank number are redundant for n-th interval, then control result is to reduce some banks (lines 7 and 8). If the hit rate is decreased, cache demand may be shifted to desire more banks allocated (lines 9 and 10). Moreover, in extreme rare case that free banks may be less than required banks, we should retire some banks of G-app to ensure the allocating priority of R-, G- and S-app, and remaining

Table 1 Control interval and parameter tuning.

Benchmark	Type	Hot Sub.	$C_{app(x)}[0]$	PID-K_P, K_I, K_D
403.gcc	G-app	19	5	0.89, 0.1, 0.2
444.namd	G-app	13	7	0.82, 0.1, 0.3
445.gobmk	G-app	15	7	0.74, 0.1, 0.25
401.bzip2	R-app	6	4	1.28, 0.2, 0.3
473.astar	R-app	7	4	1.15, 0.2, 0.2
434.zeusmp	R-app	9	5	1.07, 0.2, 0.3
450.soplex	S-app	6	11	0.63, 0.1, 0.2
429.mcf	S-app	8	13	0.71, 0.1, 0.25
482.sphinx3	S-app	12	9	0.68, 0.1, 0.2

Algorithm 1 Intensive SLLC increment Generating

Input:

$E_x[n]$: energy per access of app. x in n-th interval; $E_x[0] = E_x[-1] = 0$;

$C_{app(x)}[n]$: allocated banks of app. x in n-th interval; K_P, K_I, K_D and $C_{app(x)}[0]$ are listed in Table 1;

$R_hit[n]$: hit rate in n-th interval;

r, g and s: act as trigger marks of interval start for R-app, G-app and S-app;

Bank: free bank set;

Output:

$C_{app(x)}[n]+\triangle C_{bank}$: required bank in next interval;

1: $x \leftarrow$ application loading;
2: r, g, or $s \leftarrow$ application type;
3: **for** form n=1 to N−th interval appears **do**
4: $e_x[n]$ and $R_{hit}[n]$ calculation;
5: $\triangle C_{bank} = K_P*\{e_x[n] - e_x[n-1]\} + K_I*e_x[n] + K_D*\{e_x[n] - 2e_x[n-1] + e_x[n-2]\}$;
6: $\triangle C_{bank} = int[\triangle C_{bank}]$; //normalized in integer
7: **if** $R_{hit}[n] > R_{hit}[n-1]$ **then**
8: $\triangle C_{bank} = -|\triangle C_{bank}|$; //banks decreased
9: **else**
10: continue;
11: **end if**
12: **if** $\triangle C_{bank}$ > Bank & one of r and s appears **then**
13: retire banks of G-app for R- and S-app until G-app banks are reduced to $C_{app(x)}[0]$;
14: $\triangle C_{bank}$=retired banks + Bank;
15: **else**
16: **if** $\triangle C_{bank}$ > Bank & g appears **then**
17: $\triangle C_{bank}$= Bank;
18: **end if**
19: **end if**
20: **return** $C_{app(x)}[n]+\triangle C_{bank}$
21: **end for**

banks of G-app should be no less than $C_{app(x)}[0]$ (from lines 12 to 19). Note that we amplify $E_x[n]$ values by 10^7 in control acting purpose to generate integral $\triangle C_{bank}$.

Moreover, we decide the $C_{app(x)}[0]$ for all benchmarks through experimental exploring based on fixed bank number profiling strategy (details see Section 4.1). The experimental platform is set as four cores, 64 banks, and each core is partitioned with 16 banks, while all cores run the same application in 10 billion dynamic instructions. We set allocated bank number ranging from 1 to 16 for each core to run the application, and then we can calculate sixteen average energy per access values from each test. As shown in Fig. 1, we select the bank number corresponding to the energy-lowest value as the $C_{app(x)}[0]$, here, x stand for the application x. Thus, we can decide a $C_{app(x)}[0]$ especially for the application x through statical exploring. Moreover, the $C_{app(x)}[0]$ values can be employed as the start bank number of our control progress for preventing violent vibration in the beginning of feedback control loop. And also the fixed bank profiling results can be treated as excellent comparing candidates to reveal the energy efficiency of our controlled bank results.

In addition, we employ case studies for describing the algorithm clearly. If the energy-lowest allocated bank number in $(n-1)$-th interval is $C_{app(x)}$[n-1] (locality remains stable before) and we suppose that the locality is changed to be another one in the next several intervals, and the allocated bank number for n-th interval is still $C_{app(x)}$[n-1] because the bank increment of PID control result in $(n-1)$-th interval is zero. But the PID control output in n-th interval tends to be changed towards two possible directions as $C_{app(x)}$[n-1] is not acceptable anymore in n-th interval [21]:

- Case 1: reuse distance during n-th interval is larger than the one during $(n-1)$-th interval (desire more banks).
- Case 2: reuse distance during n-th interval is smaller than the one during $(n-1)$-th interval (desire less banks).

In the former case, allocating more banks can help to improve the hit rate toward the energy-lowest one (i.e., varying bank number from four to five in Fig. 1). But in the latter case, some banks of C_n is redundant for the n-th interval (i.e., varying bank number from six to five in Fig. 1). So that we can employ our control algorithm for generating the required bank number in the next interval corresponding to above two cases. Hence, we can firmly believe that allocating more banks in the former case will bring some improvement on hit rate, and some energy can be saved also. But in the latter case, allocating more banks only cause more energy consumption while hit rate remains unchanged.

4. Experiments and Analysis

4.1 Experimental Setup

We employ a full-system modular platform, gem5 simulator with detailed Ruby memory model, to constitute a four-core, four-thread embedded system [7]. The system configuration details are listed in **Table 2**. Each core is allocated with private L1 cache. And a 64-bank sNUCA cache is extended as shared L2 cache. We count the runtime information with Ruby and calculate energy consumption and latency with CACTI v6.5 [4]. We choose SPEC-cpu-2006v1.1 benchmarks as limited mounts of embedded applications in one system [16]. In this paper, we design three experimental strategies as follows:

Table 2 Test system configurations.

Processor	4-core, 4-thread, 2.0 GHz, 1.1 V Vdd
Private L1 I/D	32 KB, 4-way, 64B line size, 2 cycles latency
SLLC	8 MB, 64B line size, 64 banks (128 KB, 16-way, LRU per bank), 8 cycles latency
Main Memory	Double Data Tate (DDR4 2,133 MHz, 1.2 V), 8 KB page size, 95 cycles latency
Tech/Temp	32 nm/70°C

- Fixed bank number profiling strategy: Each core has sixteen banks partitioned and runs the benchmark separately, while allocated banks consume both dynamic and static energy, and the rest of banks will consume leakage energy. The energy per access values and hit rate values (all are mean values) are calculated through averaging the values of four cores.
- Fixed bank number profiling strategy with sleep mode [10]: The setups in fixed banks are same with the former strategy, and the difference is that the rest of banks will be set into sleep mode for saving some leakage energy.
- Controllable bank profiling strategy: Sixteen banks are partitioned to each core, while allocated banks consume both dynamic and static energy, and the difference contrasted with the first strategy is that the rest of banks are set up in sleep mode, in which those banks can be activated rapidly and some leakage energy can be saved.

In order to classify all benchmarks by cache bank sensibility and profiling control parameters, we employ three operations: (1) Run all benchmarks with allocated bank number ranged from one to sixteen to profile the hit rate and energy change, and then achieve the initial required bank numbers $C_{app(x)}[0]$ (i.e., five banks for *gcc* in Fig. 1) and classify allocating type of R-, G- or S-app for each application. (2) Use PIN dynamic instrumentation tool to select hot subroutines of each benchmarks [17]. Through inserting call marks into each subroutine, the tool allows us to count three values in details, call numbers in one subroutine, dynamic instructions in one subroutine and static instructions in one subroutine. Thus, we can select some hot subroutines according to their call frequencies (limited by space, partly listed in Table 1). (3) Do PID parameter tuning test for each application. As described in Section 3.3, we achieve K_P, K_I and K_D through orderly adjusting each control parameter with criterion that the lower energy, the better parameter. And we list the profiling results in Table 1. Note that one embedded system usually has several applications been executed, thus, those operations can be easily pre-treated with off-line training. For parallel execution, we also mix four benchmarks as one group to be executed in four threads and details are listed in **Table 3**.

4.2 SLLC Access Energy Control Results and Analyses

We first separately test our design in per-application condition, and benchmarks *gcc*, *bzip2* and *soplex* are employed as example for three application locality types of G-, R- and S-app. For fair comparison, we uniformly skip first one hundred intervals and count control details in next five hundred intervals. Then, we

Table 3 Mixed benchmark groups.

Group	Benchmark	$\sum C_{app(x)}[0]$
gabz	gcc, namd, bzip2, zeusmp	20 banks
ngsm	namd, gobmk, soplex, mcf	38 banks
azsm	astar, zeusmp, soplex, mcf	33 banks
msgz	mcf, sphinx3, gobmk, zeusmp	34 banks
azsg	astar, zeusmp, soplex, gobnk	27 banks
ngbm	namd, gobmk, bzip2, mcf	31 banks
smsg	soplex, mcf, sphinx3, gobmk	40 banks
bazg	bzip2, astar, zeusmp, gcc	18 banks

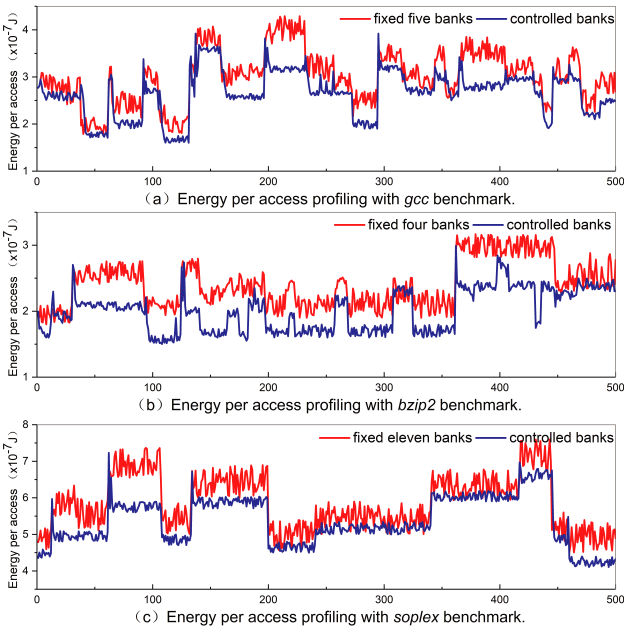

Fig. 6 Energy per access profiling in two scenarios: fixed $C_{app(x)}[0]$ and controlled bank allocated.

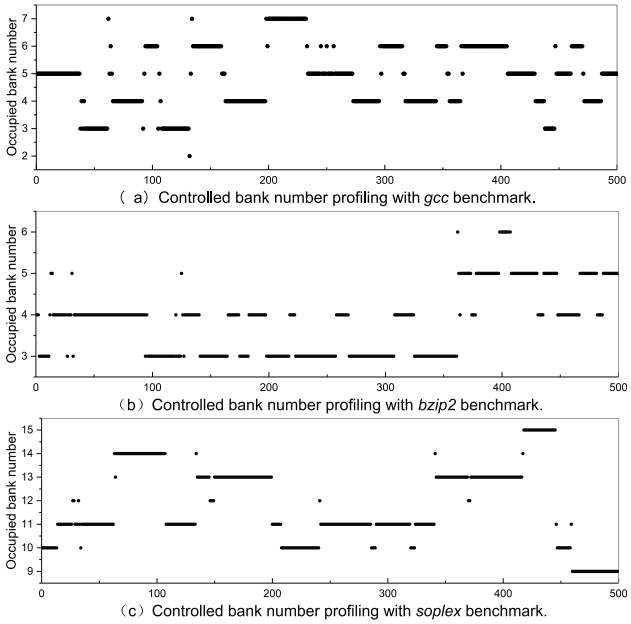

Fig. 7 Controlled bank number profiling in three benchmarks.

profile average energy per access values in each interval with two scenarios that fixed $C_{app(x)}[0]$ banks are allocated without control and flexible $C_{app(x)}[n]$ banks are allocated with control.

As shown in **Fig. 6** (a), the energy values during each interval vary from 1.64×10^{-7} J to 4.36×10^{-7} J around with fixed $C_{app(x)}[0]$ banks allocated while the energy per access value averaged in five hundred intervals is about 2.85×10^{-7} J. Along with locality varying, energy per access values vary in very large range too, for example, such value is even doubled at 204-th interval around. As to locality stable periods, energy values also remain stable in a number of intervals. For example, energy per access is about 2.98×10^{-7} J between 165-th and 203-th intervals. In the other scenario, we allocate flexible $C_{app(x)}[n]$ banks based on control results to closely follow after locality change. For each locality change, allocated banks at this time is unfitted with new locality, resulting in high energy values. Then, we can achieve suitable bank number for next interval through PID-based control computing, and this bank number can be quickly stabilized as optimal number according with the new locality. Thus, energy per access values with control are much lower than non-control ones. We also profile each $C_{app(x)}[n]$ values corresponding to interval sequence in **Fig. 7** (a). We can observe that controlled bank choices vary from three banks to seven banks, and some choices are associated to reveal similar locality in one interval period. For example, bank choice is always four between 165-th and 203-th interval while application locality keeps stable during this interval period, and more twenty percent energy can be saved than non-control scenario.

Figures 6 (b) and (c) show energy per access profiling results on R-app (*bzip2*) and S-app (*soplex*) benchmarks, respectively. Energy values of *bzip2* vary from 1.52×10^{-7} J to 3.06×10^{-7} J around and average energy value is about 2.47×10^{-7} J with fixed four banks allocated. For *soplex*, energy values range from 3.95×10^{-7} J to 7.62×10^{-7} J while average value is 5.96×10^{-7} J with eleven banks allocated. In the control scenario, both R-app and S-app show similar appearances as G-app that energy values are closely related to locality change. And average values are about 2.15×10^{-7} J and 4.89×10^{-7} J, respectively.

In order to analyze the appearances of energy varying, we should consider two key parameters, hit rate and bank number, where hit rate mostly affects dynamic energy consumption and bank number is related to static energy consumption. Along with allocated bank increasing, hit rate is raising fast in first several banks and next increases very slow. On the contrary, the number of costly off-chip memory accesses is firstly reducing very fast and next keep steady as allocated bank increasing. In other words, with allocated bank increasing, energy consumption first reduce fast due to hit rate rising, and if hit rate goes smooth, dynamic energy will remain unchanged as well as the off-chip access number. Then, we can get a low-energy allocation of banks that start from this number of allocated banks, and adding more banks only wastes more static energy and has few hit rate improved. Unfortunately, those low-energy allocation points are not changeless but change acutely with application locality. Thus, the goal of our SLLC control design is turned to trace locality varying and then generates those low-energy bank numbers ($C_{app(x)}[n]$).

To well actualize our control design, we need some new control intervals which can make a clear distinction between locality steady and locality change, instead of intervals partitioned blindly by execution time. Inspired by application analyses in Ref. [5], hot subroutine calls which take majority of dynamic instructions

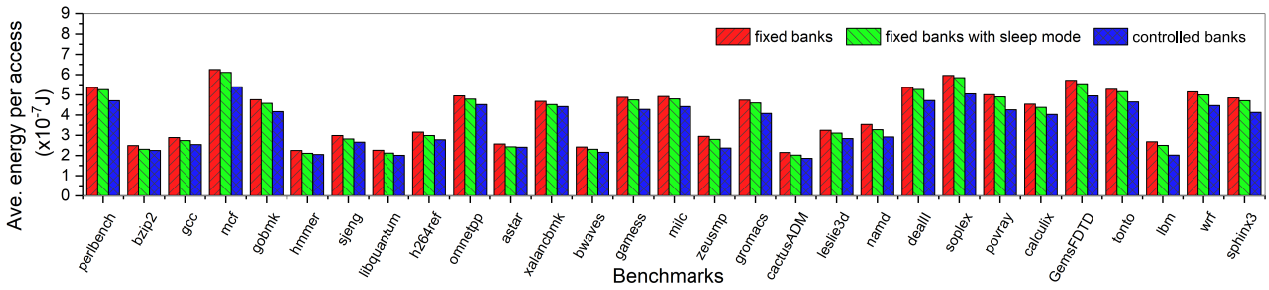

Fig. 8 Average energy per access comparisons of fixed banks and controlled banks allocated on all benchmarks.

act as natural intervals to trace locality change. Normally, calls on same subroutine show very similar locality and hit rate, that is to say, energy per access values are nearly equal in those calls. From experimental results in Fig. 6, we can affirm that same allocated banks correspond with similar energy values. During application executing, one subroutine is usually called by many times in an executing period, and those calls stick together to show their special locality. In other words, energy values may change acutely among those executing periods which belong to different subroutine calls. Hence, more energy can be saved with our locality-aware interval design.

4.3 Control Effects Analyses

In this section, we try to analyze the availability of our control design based on experimental results. We profile the control results of each intervals ($C_{app(x)}[n]$) in executing sequence. As shown in Fig. 7, allocated banks vary from two banks to seven banks for *gcc*, from three to six banks for *bzip2* and from nine to fifteen banks for *soplex*. When the locality is stable, the difference of energy values is not large enough to trigger allocated bank increment so that allocated bank number remains unchanged. For example, bank number is always four between 165-th and 203-th interval in the sequence of *gcc*. When the locality is changed or there has a call on another subroutine, previous banks may be unbefitting of the new cache demand of another subroutine. If energy value in n-th interval is much less than the value in $(n-1)$-th interval and hit rate with old allocated banks is increased, the control result of n-th interval is to reduce allocated banks for adapting new locality (i.e., 163-th interval around in *gcc*). If energy is enlarged and hit rate remains unchanged, some redundant banks should be reduced for saving energy. And if energy value is increased and hit rate is decreased, more cache banks are needed to appease the hit rate reduction for dynamic energy saving (i.e., 202-th interval of *gcc*). Moreover, we employ a PID controller which is composed of a proportional controller for representing locality change, an integral controller for eliminating static error and a derivative controller for fast control regulating [3]. Thus, new low-energy bank number which is suitable for new locality can be achieved within several control intervals. Note that alteration of allocated banks is in integer multiple of bank, and the altering frequency is low enough to ignore the latency cost of bank retirement as we use LRU replacement policy.

In addition, we only select the fixed bank number strategy for comparing with our method, because energy curves of the strategies with and without sleep mode are almost same. And the only difference is that setting the rest banks (i.e., eleven banks in *gcc*) into sleep mode can result in saving some leakage energy. In a particular case, the average controlled bank number may be same as the fixed bank number while their energy values are different (i.e., from 20-th to 95-th in Fig. 7 (b)). This is because that the occupied bank number is an average value while allocated bank number to each core may be different at same clock. For example, although controlled bank numbers may be 3, 3, 3 and 5 to four cores, the integral number is always 4 which seems to accomplish nothing over fixed four bank method, but actually some energy can be saved with our controllable method. Moreover, the leakage energy of our method on the rest banks is smaller than fixed banks method. As shown in **Fig. 8**, the average energy per access values of fixed banks with sleep mode case can save energy consumption by 4.28 percent over fixed banks case (5.65 percent in *gcc*).

To analyze the bank allocation at per-interval level, we insert start and end marks into selected subroutines and monitor their call frequencies during specified profiling period along with instruction flow. We employ two profiling periods as examples to illustrate two control effects, bank number increased and decreased. During the period from 193-th to 243-th in Fig. 7 (a), the stable bank number is seven and the former stable bank number is four. Note that during this period a subroutine is continually accessed to do the function of semantic parsing. And the energy-lowest value tested on this subroutine is corresponding to seven bank as such subroutine has a long retire distance [21], meaning that a lot of reuse data may be retired if allocated bank number is small. Hence, four banks allocated to this subroutine will bring about high energy value (190-th interval around in Fig. 6 (a)), and such energy difference will trigger the control progress to work exactly as the case 1 in Section 3.6, where the control output is adding a bank until the energy difference is small enough (up to seven banks). Considering another period from 239-th to 274-th in Fig. 7 (a), bank demand tends to be five as the retire distance is short here. So that the control progress will work as the case 2 in Section 3.6 to generate smaller bank number.

In order to show the energy saving of our control design, we calculate the average energy per access of each application with and without control design. As shown in Fig. 8, the average energy value of all applications is about 4.07×10^{-7} J with $C_{app(x)}[0]$ allocated, and such value is 3.48×10^{-7} J with controllable banks. Energy savings turn from 9.61 percent (*omnetpp*) up to 27.46 percent (*lbm*) with average 17.34 percent energy improved. Moreover, we also test the fixed banks with sleep mode case to com-

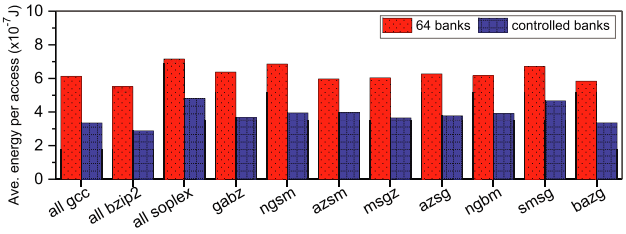

Fig. 9 Average energy per access during parallel executing.

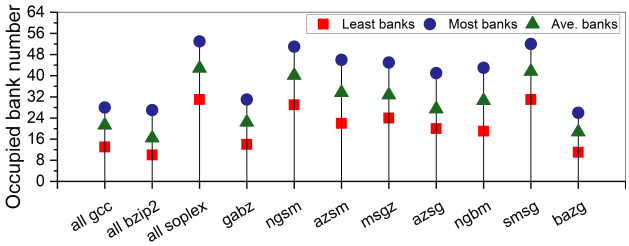

Fig. 10 Occupied bank number during parallel executing. Range from least banks to most banks occupied among intervals.

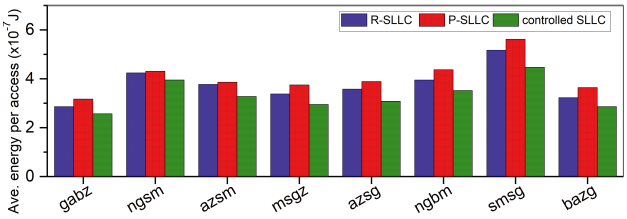

Fig. 11 Average energy per access comparisons over three SLLC optimization designs.

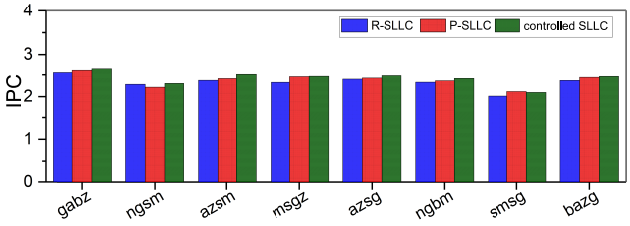

Fig. 12 IPC comparisons over three SLLC optimization designs.

pare with our method, and about 13.62 percent energy can be saved with our controllable method. Note that all control calculations are parallel with executing sequence which cost very few latency overhead, and executing times of fixed banks and controlled banks methods are differed within 0.3 percent.

4.4 Parallel Computing Control

To verify our control design in parallel computing case, we mix applications into eight groups as workloads listed in Table 3, and add three groups of all *gcc*, all *bzip2* and all *soplex*. We normalize the energy per access values of controllable SLLC to values of a conventional 64 bank SLLC. As shown in **Fig. 9**, energy value with *all bzip2* workload can save 47.92 percent over conventional case, and the least case is 31.86 percent with *smsg* group. The control cache design can save average 39.71 percent energy compared to conventional cache. And we also profile the occupied bank numbers during runtime, as shown in **Fig. 10**, allocated bank number in parallel varies from the least banks of 10 banks (all *bzip2* group) to the most banks of 53 (all *soplex* group) with average 29.7 banks. As can be seen, the optimal cache demands are very different among workloads, and even in same workload bank demands are varying acutely as locality change. As a result, many banks produce very few benefit but consume a large proportion of static energy. Our control design can save those static energy and is adapting to locality change in per-interval granularity for further energy saving.

4.5 Comparisons with Related Works

In this section, we compare our proposed SLLC design with two kinds of low-energy SLLC designs: (1) reconfigurable SLLC (R-SLLC) [9], [20]. Those designs employ search algorithms to explore low-energy cache parameter combinations in time or cycle based intervals. (2) phase based SLLC partitioning schemes (P-SLLC) [2], [11]. Those schemes save energy by dynamically partitioning SLLC spaces based on phase characteristics. For fair comparison, we skip first ten billion cycles and then profile energy per access values and IPCs in next ten billion cycles. As shown in **Fig. 11**, P-SLLC consumes more energy than other two schemes in each workload group. On average controlled SLLC scheme can save 11.6 percent energy compared to R-SLLC and 18.2 percent compared to P-SLLC, respectively. But for IPC of four cores shown in **Fig. 12**, three schemes show very few differences that IPCs of R-SLLC and P-SLLC decrease on average 3.9 percent and 1.7 percent compared to controlled SLLC scheme. To analyze at the insight, R-SLLC scheme can achieve very accurate cache allocation by exploring in dozens of intervals, but in case of locality change, such scheme needs many intervals explored to achieve the new optimal bank number while energy values during those intervals are not optimal ones. As to P-SLLC, this scheme works through dynamically partitioning SLLC based on thread parallel and phase tracing, and its allocating results are effective but not very efficient because of its small exploring set. To combine advantages of both, our controlled SLLC can fast trace locality change and control results are rapidly converged to efficient values, then more energy are saved.

4.6 Latency and Hardware Overhead Analyses

To compare with the conventional sNUCA cache [4], we employ CACTI v6.5 tool to compute the hardware overhead of our cache through estimating all components in transistor level. For a 8 MB, 64B block size, 256 K entries sNUCA cache, it total size can be calculated as 256 K entries × (34 bits tag side/entry + 512 bit data side/entry) = 140,800 Kbits. The additional hardware mainly comes from three parts: (1) Runtime information evaluator units and comparator. We employ eight counters to record those information for calculating energy and latency values. The total size is about $8 \times 32 \times 16$ T flip-flop. (2) Intensive memory controller. In our discrete PID controller, it only needs several add and multiply operations to calculate required bank increments. (3) SLLC co-scheduler. We implement two flag bits for four threads on each bank to mark co-scheduling operation and one bit for signing occupied or free status. With respect to four-thread, 64 banks condition, additional size is about 3×64 bits. Note that PID control operation works once in one interval, and such operation only occupy dozens of cycles to compute $\triangle C_{bank}$.

Therefore, comparing with the interval length in millions of cycles, control latency can be ignored and hardware overhead is less than 0.6 percent over the conventional 8 MB sNUCA.

5. Conclusion

In this paper, we propose a controllable SLLC scheme to largely save SLLC access energy. This scheme employs locality-aware intervals to trace phase change, and a modified discrete PID controller is designed to rapidly generate suitable bank increments. After tuning PID parameters effectively, control results are fast converged to low-energy bank allocating in case of both locality stable and locality change. Experimental results show that access energy with controlled SLLC scheme can be largely saved compared to conventional SLLC, and our scheme also reveals more energy saving compare to related works, while its tiny control latency and hardware overhead are negligible.

References

[1] Hameed, F., Khan, A.A. and Castrillon, J.: Performance and Energy-Efficient Design of STT-RAM Last-Level Cache, *IEEE Trans. Very Large Scale Integration* (*VLSI*) *Systems*, Vol.27, No.99, pp.1–14 (Jan. 2018).

[2] Wei, W. et al.: HAP: Hybrid-Memory-Aware Partition in Shared Last-Level Cache, *ACM Trans. Architecture and Code Optimization*, Vol.14, No.3, pp.1–25 (Sep. 2017).

[3] Bubnicki, Z.: Modern Control Theory, Springer, 2005925392 (2005).

[4] Muralimanohar, N., Balasubramonian, R. and Jouppi, N.P.: CAC-TI 6.0: A Tool to Model Large Caches, Bragantia HPL-2009-85 (Apr. 2009).

[5] Phansalkar, A., Joshi, A. and John, L.K.: Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite, *International Symposium on Computer Architecture* (*ISCA*), Vol.35, No.2, pp.412–423 (June 2007).

[6] Rawlins, M. and Gordon-Ross, A.: A Cache Tuning Heuristic for Multicore Architectures, *IEEE Trans. Computers*, Vol.62, No.8, pp.1570–1583 (Mar. 2013).

[7] Binkert, N. et al.: The gem5 simulator, *ACM SIGARCH Computer Architecture News*, Vol.39, No.2, pp.1–7 (2011).

[8] Khan, S.M., Tian, Y.Y. and Jimenez, D.A.: Sampling dead block prediction for last-level caches, *Proc. 43rd Annual IEEE/ACM Int Microarchitecture* (*MICRO*) *Symp.*, pp.175–186 (Dec. 2010).

[9] Wang, W. and Mishra, P.: Dynamic Reconfiguration of Two-Level Cache Hierarchy in Real-Time Embedded Systems, *Journal of Low Power Electronics*, Vol.7, No.1, pp.17–28 (2011).

[10] Homayoun, H. et al.: MZZ-HVS: Multiple Sleep Modes Zig-Zag Horizontal and Vertical Sleep Transistor Sharing to Reduce Leakage Power in On-Chip SRAM Peripheral Circuits, *IEEE Trans. Very Large Scale Integration* (*VLSI*) *Systems*, Vol.19, No.12, pp.1–14 (Dec. 2011).

[11] Adegbija, T. and Gordon-Ross, A.: PhLock: A Cache Energy Saving Technique Using Phase-Based Cache Locking, *IEEE Trans. Very Large Scale Integration* (*VLSI*) *Systems*, Vol.26, No.1, pp.1–10 (Jan. 2018).

[12] Wang, W., Mishra, P. and Ranka, S.: Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems, Design Automation Conference, pp.948–953 (June 2011).

[13] Kim, H., Kandhalu, A. and Rajkumar, R.: A Coordinated Ap-proach for Practical OS-Level Cache Management in Multi-core Real-Time Systems, *Real-time Systems*, No.8114, pp.80–89 (2013).

[14] Zhou, X., Chen, W. and Zheng, W.: Cache Sharing Management for Performance Fairness in Chip Multiprocessors, *International Conference on PACT*, pp.384–393 (Nov. 2014).

[15] Kim, C., Burger, D. and Keckler, S.W.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches, *ACM Sigops Operating Systems Review*, Vol.36, No.5, pp.211–222 (2002).

[16] Henning, J.H.: SPEC CPU2006 benchmark descriptions, *ACM Sigarch Computer Architecture News*, Vol.34, No.4, pp.1–17 (2006).

[17] Luk, C.K. et al.: PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation, *Proc. 2005 ACM SIPLAN Notices*, Vol.40, No.6, pp.190–200 (2005).

[18] Liao, X. et al.: A Phase Behavior Aware Dynamic Cache Partitioning Scheme for CMPs, *International Journal of Parallel Programming*, Vol.44, No.1, pp.1–19 (2014).

[19] Cheng, J. et al.: An I/O Scheduling Strategy for Embedded Flash Storage Devices With Mapping Cache, *IEEE Trans. Computer-aided Design of Integrated Circuits and Systems*, Vol.37, No.4, pp.1–14 (Jan. 2018).

[20] Chen, G. et al.: Reconfigurable cache for real-time MPSoCs: Scheduling and implementation, *Microprocessors & Microsys-tems*, Vol.42, pp.200–214 (May 2016).

[21] Zhong, Y. et al.: Program Locality Analysis Using Reuse Distance, *ACM Trans. Programming Languages and Systems*, Vol.31, No.9, pp.1–39 (Aug. 2011).

[22] Schuff, D.L. et al.: Multicore-aware reuse distance analysis, *IEEE International Symposium on Parallel & Distributed Processing*, pp.1–8 (May 2010).

[23] Aleksandrov, A.G. and Palenov, M.V.: Self-tuning PID/I controller, *Automation & Remote Control*, Vol.72, No.10, pp.2010–2022 (Sep. 2011).

Huatao Zhao received his B.E degree of both Electronics Engineering and Economics in 2011, from Beijing Institute of Technology. In 2013, he received his M.S. degree form Graduate School of IPS, Waseda University. And after, he is working toward Ph.D. degree in Graduate School of IPS, Waseda University. In 2016, he joined in Jiangxi University of Science and Technology as a cadet teacher. His current research interests are mainly including low-power high performance cache and processor design.

Jiongyao Ye received his B.E degree in Electronic Engineering in 2000, form Shanghai Marine University. In 2005, he received his M.S. degree from Graduate School of IPS, Waseda University. He then joined the Sony LSI Design Inc., where he worked in the field of LSI design from 2005 to 2008. He received his Dr. Eng. degree from Graduate School of IPS, Waseda University in 2011, then work in waseda university as a researcher. In Dec. 2012, he joined Fujitsu, and in April 2014, he moved to East China University of Science and Technologh. His current research interests include MPSoc, Microprocessor, low-power Design and their applications.

Takahiro Watanabe was born in Ube, Japan, 1950. He received his B.E. and the M.E. in Electrical Engineering from Yamaguchi University, and his Dr. Eng. from Tohoku University. In 1979, he joined R&D Center of TOSHIBA CORPORATION. In August 1990, he joined Yamaguchi University, the Department of CSSE, and in April 2003, he moved to Waseda University, Graduate School of IPS. His current research interests are EDA algorithm, Processor architecture, NoC, FPGA and their applications. He is a member of IEICE, IPSJ, EIJ and IEEE.