Regular Paper

# A Behavior-based Adaptive Access-mode for Low-power Set-associative Caches in Embedded Systems

Jiongyao Ye[1,a) ] Hongfeng Ding[1] Yingtao Hu[1] Takahiro Watanabe[1]

Received: February 22, 2011, Accepted: September 12, 2011

**Abstract:** Modern embedded processors commonly use a set-associative scheme to reduce cache misses. However, a conventional set-associative cache has its drawbacks in terms of power consumption because it has to probe all ways to reduce the access time, although only the matched way is used. The energy spent in accessing the other ways is wasted, and the percentage of such energy will increase as cache associativity increases. Previous research, such as phased caches, way prediction caches and partial tag comparison, have been proposed to reduce the power consumption of set-associative caches by optimizing the cache access mode. However, these methods are not adaptable according to the program behavior because of using a single access mode throughout the program execution. In this paper, we propose a behavior-based adaptive access-mode for set-associative caches in embedded systems, which can dynamically adjust the access modes during the program execution. First, a program is divided into several phases based on the principle of program behavior repetition. Then, an off-system pre-analysis is used to exploit the optimal access mode for each phase so that each phase employs the different optimal access mode to meet the application's demand during the program execution. Our proposed approach requires little hardware overhead and commits most workload to the software, so it is very effective for embedded processors. Simulation by using Spec 2000 shows that our proposed approach can reduce roughly 76.95% and 64.67% of power for an instruction cache and a data cache, respectively. At the same time, the performance degradation is less than 1%.

**Keywords:** configurable cache, low power, embedded systems

## 1. Introduction

With the wide application of embedded devices (e.g., notebook computers, mobile phones, consumer electronics, etc.), a highly efficient microprocessor design has become a hot issue. Modern microprocessors employ caches to bridge the great speed variance between a main memory and a central processing unit, but these caches consume a larger and larger proportion of the total power consumption. For example, the cache of DEC Alpha21164 [1] or StrongARM SA-110 [2] dissipate 25% or 43% of its total power, respectively.

Especially, to reduce a miss rate of a cache, modern microprocessors employ set-associative caches that provide a better tradeoff between miss rate and complexities of implementations. However, traditional set-associative caches have their drawbacks in terms of power consumption. A typical traditional access mode of the n-way set-associative cache probes the tag and data arrays in parallel on all the ways, and then select the data from the matched way. This operation is inefficient from the perspective of power consumption because $(n - 1)$ out of n data reads and tag reads are useless. To address this problem, various techniques [3], [4], [6], [7], [8], [11], [12], [13] have been proposed to reduce the power consumption of set-associative caches by using a more power-efficient access mode instead of the traditional mode. However, the efficiency of these techniques strongly depends on the program behavior [*1] (e.g., way-prediction accuracy). Unfortunately, since the difference in program behavior widely exists across applications, there is no special access mode that is the most efficient in all applications. Further, even within an application, the program behavior also varies in different phases of this application [15]. Thus, a more efficient approach is to dynamically adjust cache access mode to adapt to different program behaviors of a certain application. This approach has been applied to researches [9], [10] that are proposed to combine two different access modes to achieve low power target. These two schemes predict an optimal access mode for the application under execution based on the cache access history (e.g., cache miss/hit rate or way-prediction accuracy), which is essentially a temporal-based adaptive approach. The temporal approach evaluates and applies the access modes tied to successive intervals in time. Thus, both of two approaches present high efficiency only if the program can keep its execution phase for a number of time-intervals. In other words, if a variation of the cache performance such as a cache miss rate is very large within several consecutive time-intervals, these techniques provide low efficiency due to the low prediction accuracy. Therefore, the most crucial issue of the adaptive access-mode scheme is how to decide the most efficient access mode for different program behavior of an application and when to use the

[1] Graduate School of Information, Production and Systems, Waseda University, Kitakyushu, Fukuoka 808–0135, Japan

[a)] yejy_asgard@suou.waseda.jp

[*1] In this paper, the program behavior is characterized by a range of statistics, such as cache hit/miss rate, prediction accuracy, power consumption or runtime.

optimal access modes during the application execution.

In this paper, we propose a behavior-based adaptive access-mode, named BAAM for short, for set-associative caches in embedded systems, which can reduce power consumption without significant performance degradation. In our scheme, during the application execution, the cache access-mode is dynamically adjusted at the boundary of phases (a phase is a period of execution with predictable behavior) in order to adapt to the changing program behavior. In order to achieve this goal, we first implement an off-system analysis to partition the program into several phases and test every access mode for every phase to select the optimal mode. Different from the previous works [9], [10] that exploit the optimal access mode based on the cache access history, our proposed design follows the prior research [14] that exploits program behavior within a single application to identify the program phases based on the position in the code. In the case of the behavior-based scheme, an application is divided into modules (e.g., subroutines or loop) that are natural candidates of phases. Prior execution of a module can be used to accurately predict behavior of future instances. In other words, the program behavior of the different instances of the same module can sustain great similarities from an architecture perspective. Thus, once the optimal access mode is selected for a certain module, there is a reason to believe that this access mode would exhibit similar efficiency in future instances of the same module.

The behavior-based adaptive approach uses a static off-system exploitation to select the optimal access mode for different phases, resulting in requiring a previous study of the code and losing some performance because of the lack of execution time information. Fortunately, we employ the static approach based on a basic consensus that an embedded system is designed to run a fixed application for the specific target. Thus, based on simulations on the platform, we would pre-determine the optimal access mode for different phases of a certain application, but only once. In addition, our approach exploits program behavior repetition based on code sections, which can significantly reduce the period of the access-mode exploitation without affecting the simulation accuracy. Reference [14] has indicated that, in general, the behavior repetition of modules under different input and architecture configuration can maintain great stability so that the optimal access-mode exploitation is a one-time effort for a particular application regardless of the input or the architecture configuration. But, in terms of advantages, the behavior-based approach is more accurate than the temporal-based adaptive approach in predicting the future access mode, and this approach requires less additional hardware and has a much wider view of the program. Taking those into account, our approach makes sense in an embedded application where the drawbacks are slight. According to our experimental results, BAAM reduces the power consumption of an instruction cache and a data cache by up to 76.95% and 64.67% compared to the conventional access-mode, and 9.16% and 12.88% compared to the access-mode prediction scheme [10]. Moreover, the performance degradation is less than 1% compared with the conventional access-mode.

This paper is organized as follows: Related works are introduced in Section 2. Then, we discuss the motivation of our work in Section 3. In Section 4, we explain our hardware design supports and Section 5 describes the implementation of the behavior-based configurable cache. Section 6 summarizes an experimental framework and shows the evaluation results. Finally, we conclude the paper and discuss the future works in Section 7.

## 2. Related Works

Reference [3] proposed a Phased cache that separates the cache access into two phases. Tag array is accessed and compared in the first phase. Then, only the matched data way is accessed at the second phase. This technique reduces the useless data way access energy at the cost of doubling cache access latency. Filter cache [11], block buffer [12] and multiple line buffer [13] attempt to employ a small storage unit between the processor and the L1 cache to avoid unnecessary L1 cache lookups. The efficiency of these schemes strongly depends on the temporal and spatial locality of program.

References [4] and [5] employ the Way-prediction technique that attempts to predict a single way where the required data may be located before accessing tags. A correct prediction results in a fast hit and yields power benefits roughly proportional to the associativity (a single way out of n is accessed). On the other hand, a way misprediction results in no power savings and a slow hit because the rest of the ways need to be searched with a subsequent cache access. In contrast, way-selection technique [6], [7] predetermines the access way by detecting certain extra state bits or comparing partial tag so it has a fixed hit time compared to way-predicting techniques. However this technique cannot maximize power efficiency because more than one way may be selected, simultaneously. Partial tag comparison scheme [8] is another approach to reduce the power consumption of the set-associative cache, which selectively disables the sense amplifiers in data array by the results of partial tag comparison.

Powell et al. proposed a scheme that exploits direct mapping accesses for the predicted nonconflicting accesses and way prediction for those predicted conflicting accesses [9]. This approach reduces the power consumption for cache hits but does not optimize the consumption for cache misses. An aggressive scheme proposed by Zhang [10] uses two access modes: way-prediction and phased-access. First, it decides whether it uses way-prediction or not; if not then all the ways in a set associative cache are accessed with phased-access mode. Whereas, if the decision is to use way prediction, then the predicted way is accessed first, and the remaining ways are accessed only when the prediction misses. In essence, the scheme is a virtual selective-direct-mapping extension of the Most Recently Used (MRU) way-prediction, which requires line swapping and complex replacement policies so it is difficult to be applied to embedded design because of the excessive design overhead.

## 3. Motivation

Before explaining the proposed design in detail, we first illustrate the benefits that we could achieve with the adaptive access-mode scheme. To show the efficiency of the different access mode in different application and different phases of an application, we select three access modes: way-prediction (WP), Partial Tag

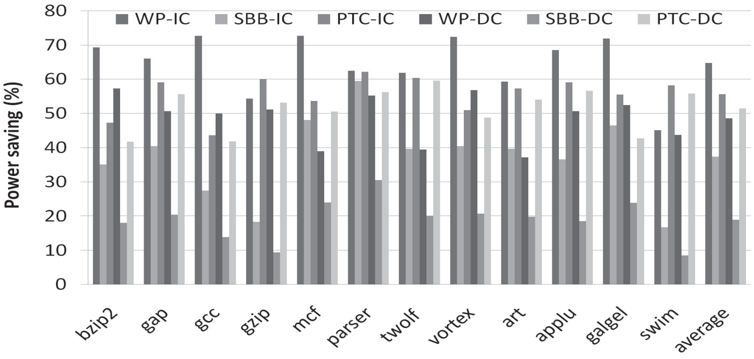


**Fig. 1** Power saving for different access modes.

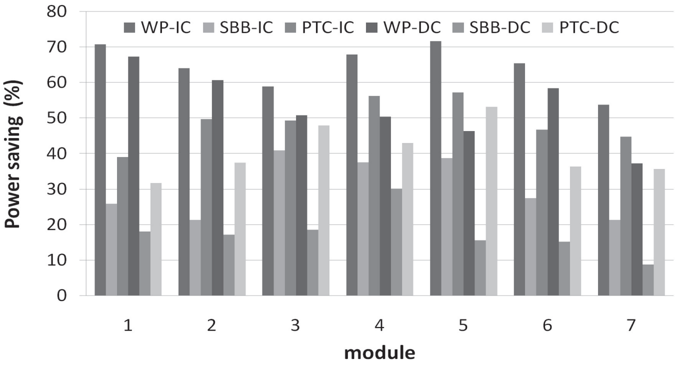


**Fig. 2** Power saving for the different modules of *bzip*.

Comparison (PTC) and Single Block Buffer (SBB) because they are typical power-efficient schemes and more easily achieved for embedded systems. Throughout the next experiments, we employ 32 KB, 4-way set-associative cache with 32B block size for L1 Instruction Cache (IC) and Data Cache (DC) — the experimental environment is described in detail in Section 6.

**Figure 1** shows that the power saving of IC and DC are achieved by three access modes for SPEC 2000 benchmark data, compared to the traditional access mode. The key observation is that the efficiency of those schemes presents difference across applications. For example, in case of IC power saving, WP clearly outperforms other schemes in most applications except *Gzip*. Similarly, in the cases of DC power saving, PTC is a better choice for some applications like *mcf*, *art* and *swim*, but other applications such as *bzip*, *gcc* and *galgel*, WP works better than PTC.

A single access mode also exhibits the efficiency variation in the different phases of the given application. **Figure 2** shows the power saving of IC and DC for the different modules of the given application (*bzip*). Following the prior work [14], the application is divided into 7 modules. A module is the basic program phase, which is explained in the next section. The simulation results show that, although WP works well in most of the modules both for IC and DC, PTC is a better choice in some cases, such as power saving of DC for the module 5.

Overall, no single access-mode can be regarded as a panacea for all the applications, and works well in all the program phases. Thus, an adaptive access-mode scheme is required to dynamically adjust the access mode for different phases during the application execution.

## 4. Hardware Support

Our proposed design tends to adjust the access mode to match the requirements of each application phase. Considering performance and design complexity, three access modes are employed by our paper: way-prediction (WP), Partial Tag Comparison (PTC) and Single Block Buffer (SBB). However, to avoid performance degradation due to the twofold miss (i.e., the referenced data is neither in SBB nor in the predicted way that is decided by WP or PTC), SBB is accessed as soon as cache is accessed by another access-mode (WP or PTC). Thus, there are four access-modes in BAAM: WP with and without SBB and PTC with and without SBB. The off-system pre-analysis evaluates the performance and runtime of every access-mode for every program phase, and selects the most optimal one. In general, if the accuracy of way prediction is above a certain limitation in the current code segment, WP would be adopted because of its high power-efficient and low design complexity. Otherwise, PTC is used to reduce the major power consumer (i.e., sense amplifier) of cache. Different from research [10] that employs the phased cache as alternative mode to reduce the power consumption at the cost of increasing the access cycle, PTC reduces the major power consumption of set-associative caches without performance degradation. Note that only one of the above two schemes can be selected for each program phase. On the other hand, since the power efficiency of SBB is not noteworthy (on average, reducing total cache power by up to 37% and 18% for IC and DC, respectively), it should not be used alone. However, SBB is considered as a complement to other access modes because it can not increase the access delay, easy to use together

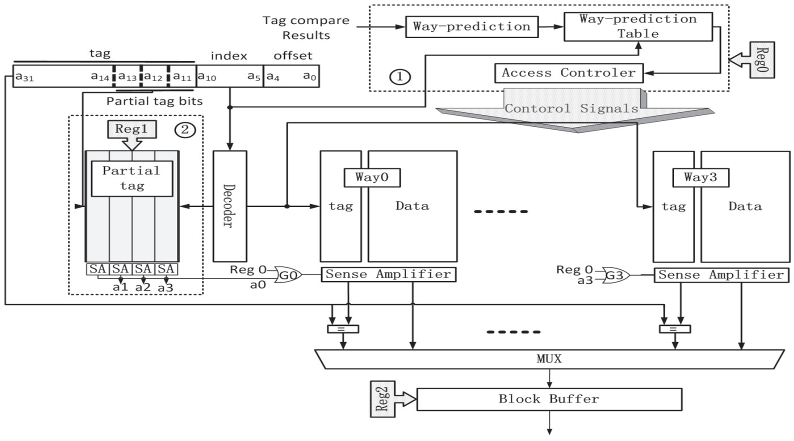


**Fig. 3** Architecture of behavior-based adaptive access modes cache.

with others modes, and the power consumption and area overhead of SBB itself is very small. Therefore, if there is good locality in the current code segment, SBB is utilized to associate with one of other two modes so that the power consumption of set-associative caches is further reduced without extra access delay.

**Figure 3** illustrates our proposed design. Components in the dotted frame numbered 1 are way-prediction, including a way predictor, a way-prediction table and an access controller. Way predictor determines the value of each way-prediction flag according the the Most-Recently Used (MRU) algorithm. Way prediction table contains a two bits way-prediction flag for each set to choose one way from the corresponding set. The dotted frame numbered 2 is an additional very small tag array for partial tag comparison. This small tag array is organized and accessed as a regular tag, which keeps a copy of the least 3 significant bits of the original tag. Our proposed design also adds a single block buffer for output latch. The block buffer saves the instructions which come from the last accessed block. Thus, the next required data are likely to be directly fetched from this block buffer so that the normal level-one cache access is avoided. Three extra configuration registers (Reg0, Reg1 and Reg2) are used to achieve different access modes. Each of them is 1 bit. All the registers are set/reset at the boundary of the corresponding phase in order to change the access modes.

### 4.1 Operations

During the application execution, BAAM dynamically adjusts the access modes based on the results of the off-system preanalysis for this application. When Reg0 is set and Reg1 is reset, the WP modes is activated and the partial tag array is disabled by Reg1. In this case, once an effective address is generated, the way-prediction flag is read from the corresponding set of the way-prediction table to determine the predicted way. Then, only the predicted way is activated, and implements the cache access in the same manner as conventional set-associative caches. The remaining ways are accessed only when the way-prediction miss. Note that Reg0 masks the output of partial tag array by an OR gate, which ensures the sense amplifier of predicted way is valid. When Reg0 is reset and Reg1 is set, PTC is used and WP is prevented from predicting and updating. In this case, for each way, partial tag comparison is implemented. Only when there is a match, the sense amplifiers attached to the data array bit-lines can be enabled. Reg2 specifies whether the block buffer needs to be accessed. If there has good spatial locality in the current code segment, the block buffer is allowed to access to reduce the number of unnecessary level one cache access. Otherwise, it is unused.

### 4.2 Overhead

Compared to the conventional access mode, BAAM would induce delay penalty only when way prediction miss in case of WP mode. For PTC, Ref. [8] pointed out that the partial tag comparison cannot increase the access latency due to its small size. The block buffer access is implemented concurrently with set decoding. In other words, delay penalty of SBB can be completely hidden by the set decoding.

The baseline cache uses 6T SRAM cell technique. For an 32 KB four-way cache with a block size of 32B, the total storage area size is about 1,689,600 transistors including tag array (set number 256 * set-associativity 4 * tag size 19 bit * 6T) and data array (set number 256 * set-associativity 4 * data size 256 bits * 6T). The major area overhead due to BAAM design comes from three sources: 1) an extra block buffer uses a 9T RAM cell to implement the tag and index part (the width is 27 bit), and the data part can be implemented with 8T latch, the size of which is same as the base line size (i.e., 256 bits). Thus, its total area overhead is roughly (9*27) + (8*256) = 2,291 transistors, which is about 0.1% of the total storage area. 2) Partial tag comparison copies the least 3 significant bits of the original tag. The size of partial tag array is about 18,432 transistors. The area overhead of PTC is less 1.1% of the total storage area. 3) For WP, an added hardware includes n bits way information registers, and a way prediction state machine which is the most essential part in WP, and several bits of access controller. Because we choose 4-way set-associative caches in this paper, 2 bits register are used to store predicted way information. Compared with tag and data array, the area overhead of WP is fairly limited and it can be ignored.

## 5. Behavior-based Policy

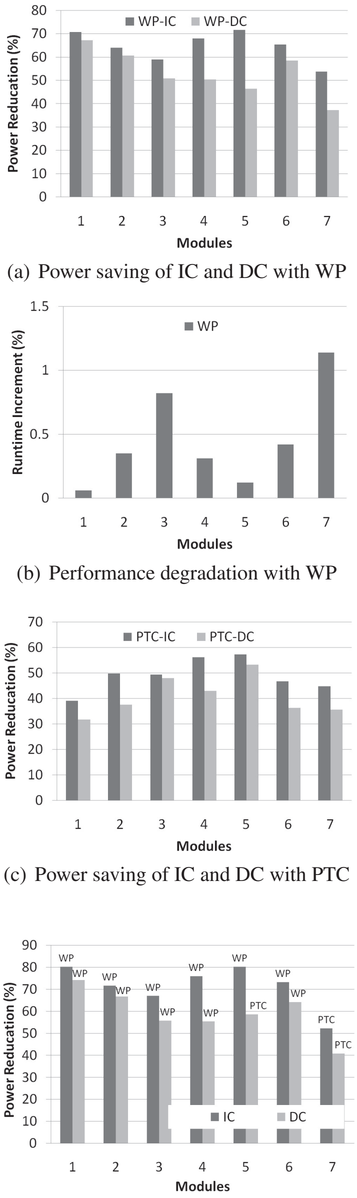
In order to apply the optimal cache access modes during the course of the program, we perform an off-system pre-analysis for each application, which has four steps: 1) Module selection, 2) Configuration exploration, 3) Choosing the configurations and 4) Instrumentation. We consider the issues in these steps in the following sections.

### 5.1 Module Selection

The first step in the analysis is to statically partition the program into smaller units, called modules. A specified optimal access mode is performed at the beginning of each module. There are two reasons that lead us to using the behavior-based approach instead of the previous works [9], [10] that are based on the temporal approach. First, intuitively, the code strongly affects cache demand. In fact, the processor only uses an instruction address to access cache, so the cache access mode is more related with code's section being executed than with the fixed time interval. Second, the behavior of the dynamic instances of a static module often remains very stable. These two observations agree with [14], namely in that is, the tying adaptive configuration to the code's position is generally more effective than the temporal-based adaptive scheme because the different instances of the same code exhibit highly similar behavior during program runtime.

The granularity of a module has a huge effect on the performance of our scheme. The ideal granularity of a module depends on the particular goal. A module with a large granularity may contain smaller modules with different access mode. On the other hand, the module cannot be too small because adaption overhead at runtime will be large, which results in more power consumption and performance degradation. Meanwhile, in this case, simulation statistics tend to be determined by the microarchitectural state rather than by the code itself. In our paper, we select two basic structures as natural modules: subroutines and loops. Reference [14] pointed out that different iterations of a loop or different invocations of a subroutine can exhibit a very similar behavior.

But, not all subroutine have the ideal grain size or the same importance, so we use two thresholds (TH$_{low}$ and TH$_{large}$) to select the subroutine with the appropriate granularity. We empirically set these thresholds to 50K instructions per subroutine and 1M instructions per subroutine, respectively. If the size of a subroutine is smaller than TH$_{low}$, it is merged with its caller. Instead, if the size is larger than TH$_{large}$, it may contain sections with different behaviors. Within this large subroutine, we further exploit behavior repetition of loop iterations. To reduce the configuration overhead, we only exploit long loops that have an average instance size higher than TH$_{large}$. Although these two thresholds are selected based on empirical values, there is no significant variation across various thresholds for a particular application under the same simulation environment. This is because these thresholds largely depend on the target architecture rather than on the application. Furthermore, in order to reduce the complexity of the access-mode exploration, certain modules can be completely ignored for subsequent analysis if they are executed too infrequently. This approach is called *subroutines filtering*. In our paper, if the number of instructions of a certain module is below 0.5% of the total number of dynamic instructions, then that module is ignored. According to the filtering, the average number of modules changes from 40 to 21. Therefore, the filtering approach helps most applications reducing more than half the static code, which simplifies the subsequent analysis.



(a) Power saving of IC and DC with WP

(b) Performance degradation with WP

(c) Power saving of IC and DC with PTC

(d) Power saving of IC and DC with SBB

**Fig. 4** Configuration exploration for *bzip*.

### 5.2 Access-modes Exploration and Choice

After dividing the application into modules with proper granularity, we explore the different configurations on each module in order to determine the optimal choice. We use a straightforward approach where the application is executed many times to directly measure the execution time and power consumption. An exhaustive mode exploration using an exhaustive algorithm is used to test each access modes on every module. In this paper, there are only 3 access modes that need to be evaluated.

**Figure 4** illustrates the mode exploration and choice in the case of *bzip* benchmark of SPEC 2000 suite, consisting of 7 modules. First, we implement the fastest implementation (called basic mode) for each modul, which probes the tag and data arrays in parallel on all four ways, and then select the data from the matching way, and record the execution time and power consumption of each module. Compared with the basic mode, Fig. 4 (a) shows that WP reduces the power consumption (PS) of IC and DC for each module, and Fig. 4 (b) shows the performance degradation (PD) of each module due to using WP. Similarly, PTC is tested in Fig. 4 (c), but only the power consumption needs to be recorded and compared with basic mode because of the scheme without performance degradation. After evaluating these two access modes, we first select the better one for each module. To select the optimal access mode, the maximum performance degradation ($\mathrm{PD}_{\mathrm{max}}$) is given according to user requirements (in our paper, we empirically set it to 1%). Then, we select the access mode that shows the highest power saving and whose PD does not exceed the given $\mathrm{PD}_{\mathrm{max}}$. For *bzip*, the simulation results show that WP is the optimal access mode for IC and DC, except DC of module 5. Note that, for module 7, although power efficiency of WP is better than that of PTC, we select PTC for module 7 because WP results in the performance degradation over the $\mathrm{PD}_{\mathrm{max}}$. At last, the SBB is tested based on the above-selected access mode for each module. If the power saving of the module is improved, then the SBB is used. Otherwise, it is closed. As shown in Fig. 4 (d), all the modules can reduce the power consumption by SBB, so SBB is always used during the *bzip* execution.

### 5.3 Instrumentation

After each module of a particular application selects the most optimal access mode, the binary-code application needs to be modified in order to instrument entry and exit points in modules. Two extra instructions are introduced into the instruction set architecture, which change the values of the configuration registers (i.e., Reg0, Reg1 and Reg2) to adjust access modes when alternating between the two modules. The first special instruction is writing configuration registers (ConReg_we) that writes the corresponding values into the configuration registers at the head of a certain module. At the tail of the module, the other instruction, exit configuration (Exit_Con), is inserted to instruct the cache to return the previous access mode. **Figure 5** illustrates an example of instrumentation and the operation of the stack. Entries of a subroutine are inserted by ConReg_we, and Exit_Con is added before the return instruction. For a loop, we identify iteration boundaries and loop termination by marking the backward branch and all the side exit branches of all chosen loops.

In the case of nesting, the current configuration must be saved when another module is called, and be restored after the return. We use a stack to store the previous configuration. The stack, which has 6 bits (IC and DC each 3 bits) and 16 entries, is enough to cover most situations in our experiment. If the stack overflows, the cache remains in the current configuration until an entry of the stack is free.

## 6. Simulation

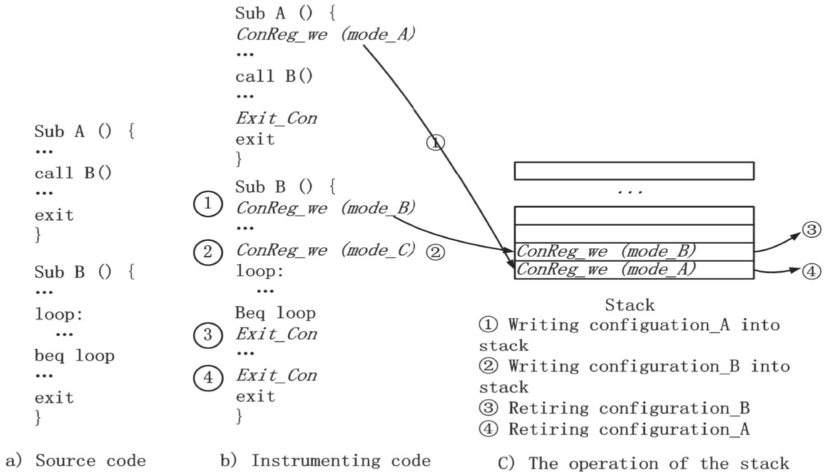
### 6.1 Evaluation Environment

To evaluate the power and performance of our proposed adaptive access-mode, we use Wattch 1.0 [16], which is an architecture-level power analysis tool built on SimpleScalar [17] and integrated a modified version of CACTI [18] to model level-one cache. Wattch reports both the execution time and the power/energy consumption of simulated processors. **Table 1** shows our basic system configuration parameters.

Twelve benchmark applications are taken from the SPEC 2000 suites that are typical representatives as embedded applications.

**Table 1** Basic system configuration.

| | |
|---|---|
| CMOS Technology | 70 nm, power supply = 1.2 V |
| Issue/decode width | 4 intrs. per cycle |
| ROB/LSQ | 64 /32 entries |
| Branch predictor | 16K entries Gshare |
| Writeback buffer | 8 entries |
| Base L1 I-cache | 32 KB, 32 line-size, 4-way set-associative,<br>3cycle access latency,<br>power consumption per cache access = 925.9 mw,<br>leakage power = 141.6 mW |
| Base L1 D-cache | 32 KB, 32 line-size, 4-way set-associative,<br>3cycle access latency,<br>power consumption per cache access = 925.9 mw,<br>leakage power = 141.6 mW |
| L2 unified cache | 2 MB, 64 line-size, 8-way set-associative,<br>12 cycles access latency,<br>power consumption per cache access = 23.89 watt,<br>leakage power = 8793 mW |



```
Sub A () {
…
call B()
…
exit
}
Sub B () {
…
loop:
  …
beq loop
…
exit
}
```

a) Source code

```
Sub A () {
ConReg_we (mode_A)
…
call B()
…
Exit_Con
exit
}
Sub B () {
ConReg_we (mode_B)
…
ConReg_we (mode_C)
loop:
  …
Beq loop
Exit_Con
…
Exit_Con
exit
}
```

b) Instrumenting code

① Writing configuation_A into stack
② Writing configuration_B into stack
③ Retiring configuration_B
④ Retiring configuration_A

C) The operation of the stack

**Fig. 5** Instrumented module and operation of the stack.

**Table 2** The average power and access delay of the 32 KB 4-way cache.

| | Data path | | Tag path | |
|---|---|---|---|---|
| | Delay (ns) | Power (mw) | Delay (ns) | Power (mw) |
| Row decoder | 0.35 | 23.4 | 016 | 8.75 |
| Wordline and bitline | 0.11 | 48.2 | 0.06 | 33.3 |
| Sense amplifier | 0.07 | 628.6 | 0.05 | 120 |
| Tag compare | 0 | 0 | 0.16 | 1.25 |
| Mux driver | 0 | 0 | 0.13 | 02.3 |
| Output driver/valid output | 0.06 | 56.7 | 0.06 | 3.4 |
| Power per access | 0.59 | 756.9 | 0.62 | 169 |
| Average delay per access | 0.62 ns | | | |
| Average power per access | 925.9 mw | | | |

To cover the wider variety of behavior, we simulated more than two billion instructions for each application. For off-system analysis, we used the test input set, which greatly help off-system analysis reduce the simulation time. It is based on the observation that the small inputs has a similar behavior to a big input sets by exploring the behavior repetition [14].

Our cache models are based on the CACTI cache model. We assume all cache models have the same cache configuration (including cache size, associativity, block size and technology that are shown in Table 1). According to the power model of different cache models (described in Section 6.2), the average power per access of different cache models is estimated. For example, for WP cache model, only one tag and data subarray, a related decoder and an output driver consume the power. Furthermore, we modified CACTI to incorporate the power model of the block buffering for BAAM. We use the modified Wattch built on the modified version of Simplescalar 3.0 to model a high-performance out-of-order processor for overall power and performance analysis.

### 6.2 Power Consumption Evaluations

In this section, we describe the power model of different cache models. The power model is the computation of Average Power per Access (APA) for different cache models. For the conventional parallel-access set-associative cache, there are three main power consumers: power consumed by the data array, power consumed by tag array and power to drive the output. The breakdown of the power consumption and access delay of the given cache is shown in **Table 2**. As a result, the power per access of the conventional 4-ways set-associative parallel-access cache (APA$_{cache}$) is about 925.9 mw.

In way prediction cache, the predicted way first needs to be accessed, and the remaining ways are accessed only when the way prediction is missed. The APA of the way-prediction cache (APA$_{wp}$) can be calculated by Eq. (1):

$$APA_{wp} = \left(\frac{1}{N} \times APA_{cache}\right) + M_{wp} \times \left(\frac{3}{N} \times APA_{cache}\right) \qquad (1)$$

where N is the associativity. M$_{wp}$ is the way prediction miss rate.

In partial tag comparison cache, sense amplifiers of the unmatched way are disabled, so this part of power should be moved from the total power consumption. The APA of partial tag comparison cache (APA$_{ptc}$) can be calculated by Eq. (2):

$$APA_{ptc} = APA_{cache} - P_{Sen} \times R_{unmatch} + P_{pc} \qquad (2)$$

where P$_{Sen}$ is the power consumed by the data sense amplifier. R$_{unmatch}$ is the unmatched rates of the tag partial comparison. Here, P$_{pc}$ is the power consumed by the extra 3-bit partial tag comparison per cache access, but this part of power is very small. By using CACTI, we estimate the power of 3-bits partial tag comparison. P$_{pc}$ is approximately 33.52 mw which is less than 4% of part of APA$_{cache}$.

In block buffering cache, if there is a block buffering hit, the data is directly read from the block buffer and the cache does not operate. Thus, APA of a block buffering cache (P$_{bb}$) can be expressed as Eq. (3):

$$APA_{bb} = P_{buff} + APA_{cache} \times M_{buff} \qquad (3)$$

where P$_{buff}$ is the power consumption per block buffer access. P$_{buff}$ obtained from the HSPICE simulation is 9.35 mw. M$_{buff}$ is the block buffering miss rate.

For WP and PTC cache models with SBB, SBB is accessed at the same time that the cache is accessed by another access-mode (WP or PTC), with the result that cache access can not be entirely avoided even if SBB is hit. We perform the HSPICE timing simulation to measure the access time of SBB. The latency to determine SBB hit is approximately 0.4 ns. By comparing this latency to the access delay shown in Table 2, we found that the latency to access SBB can be overlapped with the decoder and wordline/bitline delays of the data path. Thus, the power consumption of a sense amplifier of data array of predicted way can be reduced when SBB is hit. Equations (4) and (5) are the power model of APA$_{wp\_sbb}$ and APA$_{ptc\_sbb}$, respectively:

$$APA_{wp\_sbb} = P_{buff} + APA_{wp} - \frac{1}{N} P_{Sen} \times H_{buff} \qquad (4)$$

$$APA_{ptc\_sbb} = P_{buff} + APA_{ptc} - R_{match} \times P_{Sen} \times H_{buff} \qquad (5)$$

where P$_{Sen}$ is the power of the sense amplifier of data array of predicted ways. H$_{buff}$ is the block buffering hit rate. Compared to P$_{wp}$ and P$_{ptc}$, SBB adds the block buffering power per access, but can reduce the sense amplifier of data array of the predicted ways when the block buffering is hit. Note that the predicted way in WP mode is always one. However, the number of predicted ways in PTC mode is decided by the matched rates of the tag partial comparison. According to the power model of WP and PTC modes associating with SBB, we found that associating PTC or WP with SBB is effective even if its hit rate is not high because the power consumed by the sense amplifier of data array is much larger than that by SBB.

AMPS cache uses two access-modes: way prediction and phased-access. Based on cache hit and miss way prediction, AMPS selects an optimal access-mode for the current program execution. The power model of way prediction is explained in Eq. (1). We estimate APA of the phased-access cache by Eq. (6):

$$APA_{phased} = P_{tag} + \frac{1}{N} \times P_{data} \times H_{cache} \qquad (6)$$

$$APA_{AMPS} = APA_{wp} \times t + APA_{phased} \times (1 - t) \qquad (7)$$

where P$_{tag}$ is the power consumed by tag array. N is the associativity, and $1/N\times$ P$_{data}$ is the power consumption of data array for the hit way. H$_{cache}$ is the cache hit rate. Therefore, the APA$_{AMPS}$

**Table 3** Module selection.

| | Modules | Sub-routines | Loops | Average static intr. per module | T ($\mu$s) |
|---|---|---|---|---|---|
| bzip2 | 7 | 5 | 2 | 2,458 | 771.2 |
| gap | 14 | 8 | 6 | 906 | 272.1 |
| gcc | 9 | 4 | 5 | 8,471 | 1,960.2 |
| gzip | 9 | 7 | 2 | 643 | 875.2 |
| mcf | 11 | 6 | 5 | 689 | 55.6 |
| parser | 35 | 23 | 12 | 1,411 | 13.5 |
| twolf | 31 | 16 | 15 | 3,478 | 506.2 |
| vortex | 7 | 4 | 3 | 1,354 | 175.3 |
| art | 43 | 14 | 29 | 2,720 | 168.7 |
| applu | 62 | 14 | 48 | 27,484 | 1,593.4 |
| galgel | 15 | 10 | 5 | 5,896 | 955.4 |
| swim | 9 | 4 | 5 | 8,719 | 5,846.3 |
| average | 21 | 9.6 | 11.4 | 5,352 | 932.8 |

is expressed as Eq. (7), where t is the percentage of the total execution time to use WP mode, which is decided by the cache accesses history. For AMPS associating with a block buffering, the block buffering is accessed for every cache access since the block buffering is orthogonal to other two access-modes. Equation (8) presents the power model of APA$_{AMPS\_sbb}$ and Eq. (9) is that of APA$_{phased\_sbb}$

$$APA_{AMPS\_sbb} = APA_{wp\_sbb} \times t + APA_{phased\_sbb} \times (1 - t) \quad (8)$$

$$APA_{phased\_sbb} = P_{buff} + APA_{phased} - \frac{1}{N} P_{data} \times H_{buff} \quad (9)$$

BAAM cache employs three access-modes: WP, PTC and SBB. Since SBB mode and other two access-modes are orthogonal, there are four access-modes in BAAM cache (i.e., WP with and without SBB and PTC with and without SBB). The power model of WP and PTC with and without SBB are presented in Eqs. (1), (2), (4) and (5) respectively. Thus, APA$_{BAAM}$ can be estimated by Eq. (10):

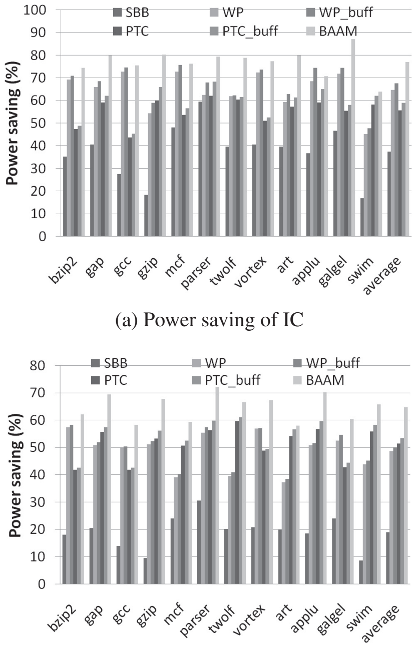
$$\begin{aligned} APA_{BAAM} = {} & APA_{wp} \times f_{wp} + APA_{wp\_sbb} \times f_{wp\_sbb} \\ & + APA_{ptc} \times f_{ptc} + APA_{ptc\_sbb} \times f_{ptc\_sbb} \end{aligned} \quad (10)$$

where f$_{wp}$, f$_{ptc}$, f$_{wp\_sbb}$ and f$_{ptc\_sbb}$ are the percentage of the total invocations of subroutine, according to the different access-modes used in BAAM, respectively. Because the access-mode of each subroutine has been pre-decided by the off-system analysis, APA$_{BAAM}$ depends on the number of calls to subroutines that use the same access-mode.

According to the power model of different cache models, we estimate the average power per access of different cache models for all benchmarks by using the Simplescalar tool to collect the program execution statistics, such as cache hit rate, way prediction rate, and so on. The simulation results are presented in the next section.

### 6.3 Simulation Results

**Table 3** shows the number of modules (including subroutines and loops), the static instructions per module of these benchmarks and the average execution time per module. The actual average number of static code sections is over 40 for all experimental applications, but only 21 sections remain after executing module filtering. For this reason, the subsequent off-system analysis of applications is significantly simplified. The average time of modules are also shown in the table, which is dynamically obtained at run time. The time ranges from more than ten $\mu$s to thousands of $\mu$s. Overall, the total period of off-system analysis for an application is quite acceptable.



(a) Power saving of IC

(b) Power saving of DC

**Fig. 6** Power comparison between the single access-modes and BAAM.

**Figure 6** (a) and (b) show the power saving comparison for IC and DC between the single access-modes with and without block buffering and BAAM. SBB cache is the conventional cache with a block buffering. The block buffering is accessed at the first cycle. If there is a hit, then the cache does not function. Otherwise, the cache is accessed at the second cycle. Except for SBB, other single access-modes with the block buffering access a cache at the same time that the block buffering is accessed. The purpose is to avoid the performance degradation due to the high block buffering miss rate. In terms of power saving, WP_buff and PTC_buff are the best approaches among all the single access-modes for IC and DC, respectively. However, BAAM obtains about 9.31% and 11.39% power saving compared with WP_buff in IC and PTC_buff in DC, respectively. In terms of performance, PTC does not increase the cache-access time because the partial tag comparison and cache are operated, simultaneously. SBB does not increase the cache-access time if the block buffering is hit, but it needs an additional one cycle to access cache in the case of the block buffering miss. Similarly, WP needs a extra cycle to access the remaining ways in case of the way prediction miss. For BAAM, the performance degradation is limited less than the value that is set by the requirement of customers. By our simulation results, the increase in runtime is about 18.7%, 5.6% and 0.57% for SBB, WP and BAAM cache models, on average. Overall, BAAM achieves much more power-efficiency than the conventional single access-modes and the performance degradation of it is also smaller than others.

We also model an Access-Mode Prediction Scheme (AMPS) [10] to compare with our proposed design based on the 32 KB 4-way set-associative cache. The AMPS is the most sophisticated adaptive access-mode selection scheme that

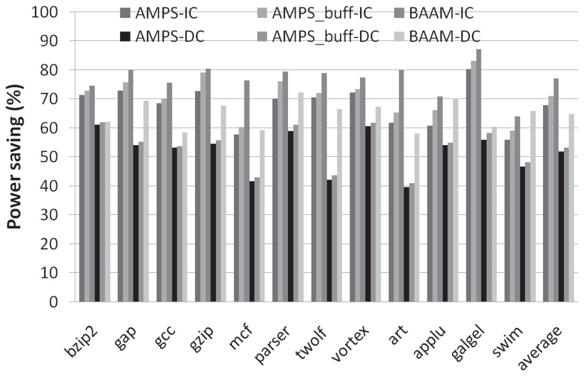


**Fig. 7** The power savings achieved by two schemes.

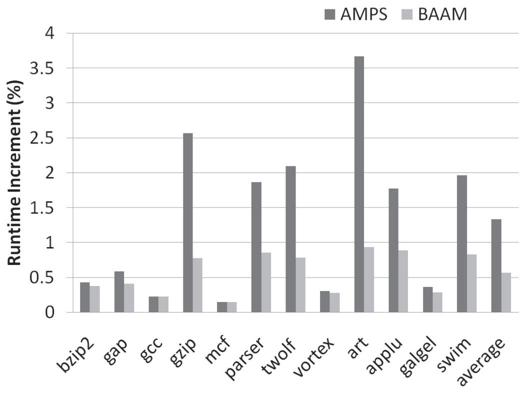


**Fig. 8** The execution time increment.

can outperform most of other current access modes. **Figure 7** shows the power saving of IC and DC by using AMPS with and without SBB and BAAM, in contrast to the conditional access mode with the same simulation configuration. For brevity, we only show the total power consumption per application. Compared to the basic mode, the total cache power savings are achieved by our proposed scheme in all of applications because the cache access modes during run-time are dependent on the applications. The power reduction in BAAM outperforms AMPS benefits for different dynamic instances of the same module being more stable for access-mode compared with several successive instruction intervals, so that the behavior-based adaptive access-mode is able to predict future access modes more accurately than AMPS. For IC power saving, BAAM can reduce the average power consumption by up to 76.95%, 9.16% and 5.95%, compared to the basic mode, AMPS and AMPS_buff, respectively. For DC power saving, BAAM can reduce the average power consumption by up to 64.67%, 12.88% and 11.68%, compared to the basic cache, AMPS and AMPS_buff, respectively. Thus, BAAM outperforms AMPS with and without the block buffer in both IC and DC.

**Figure 8** shows the performance degradation of all applications for the BAAM and AMPS. Although BAAM saves about 76.95% and 64.67% of the average power consumption for IC and DC, respectively, it does not slow down the program execution much because of the limited target slack. But, AMPS reduces the performance by over 1%. This is because it regards the cache access history as straightforward exploration metric, which could easily cause more misprediction due to the variation of the prediction accuracy among several successive intervals. Furthermore,

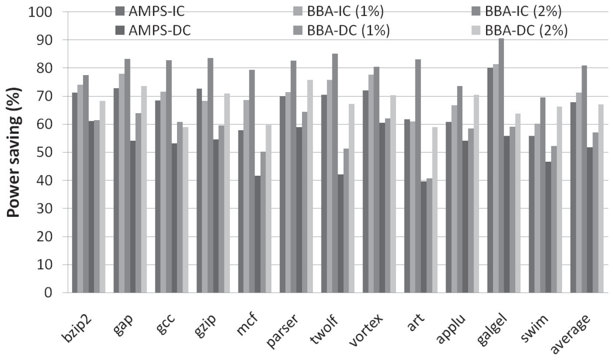


**Fig. 9** Power saving with different adaptive scheme.

AMPS uses the phased cache as an alternative modes for WP to reduce the power consumption, which also leads to performance degradation.

### 6.4 Comparison of Power Efficiency

Finally, we compare the power efficiency between the behavior-based access-mode (BAAM) and the conventional adaptive scheme (AMPS). In order to make a fair comparison, BAAM also uses the same two access-modes used in AMPS: way prediction and phase-based access. **Figure 9** shows the power saving of the two schemes. Here, we assume two performance constraints: $PD_{\max 1}$ and $PD_{\max 2}$ that are 1% and 2%, respectively. ($PD_{\max}$ is explained in Section 5.2). We select the more power-efficient access-mode until the performance degradation exceeds $PD_{\max}$. Compared to AMPS, BAAM within 1% performance degradation obtains a little power saving in most applications. The largest power saving is about 10.88% in mcf, average power saving is about 3.37% and 5.17% for IC and DC, respectively. The reason for the limited power saving is that because that the phased-based access-mode used in BAAM reduces the power consumption by sacrificing the access time. As a result, BAAM is limited to select the optimal access-modes because of the exorbitant performance constraint. When the limitation of the performance degradation is up to 2%, all applications show significant power saving. Compared to AMPS, BAAM within 2% performance degradation can reduce the power consumption by an average of 13.1% and 12.18% for IC and DC, respectively. On the whole, BAAM is better than the conventional adaptive scheme in terms of selecting an optimal access-mode for cache accesses. We also expect that the behavior-based adaptive scheme to prove more efficient in other adaptive mechanisms.

## 7. Conclusions and Future work

In this paper, we have proposed a new approach for power-efficient set-associative caches on an embedded system, which can dynamically adjust the access mode for different phases of an application during runtime. Our proposed scheme utilizes program behavior repetition to implement off-system access-mode exploration for each module of a certain application. Once the optimal access-mode is determined, it can be applied to future execution of the same module. It requires less additional hardware and has a much wider view of the program. Although it needs a previous analysis of the application, the analysis is an one-time effort for each application and the development period of off-system analysis is greatly reduced by behavior-based exploration. So it is very suitable to embedded applications because such a system always executes a fixed or routine application. Meanwhile, the behavior-based scheme reduces the exploration complexity, regardless of difference in architecture or a set of input. As a result, our proposed behavior-based adaptive access-mode scheme can reduce the average power consumption by up to 76.95% and 64.67% for IC and DC, respectively. The performance degradation for all application is less than 1% , compared to the basic cache (i.e., the fastest access mode),

In this paper, we evaluated only for level-one caches. Future work is to demonstrate the effectiveness of the behavior-based adaptive scheme on the level-two cache. More power-efficient access modes can be expected from our scheme so that the power consumption, performance degradation and design complexity can be further reduced. Furthermore, dynamic modules selection and modes decision should to be introduced in order to meet the requirements of a general purpose system.

**Acknowledgments** This research was supported by JSPS KAKENHI 11515400, Core Research for Evolutional Science and Technology by JST and Program for Fostering Regional Innovation by MEXT, Japan.

## Reference

[1] Edmondson, J.H., Rubinfeld, P.I., et al.: Internal organization of the Alpha 21164, a 300MHz 64-bit quad-issue COMS RISC microprocessor, *Digital Technical Journal*, pp.119–135 (July 1995).

[2] Montenaro, J. et al.: A 160MHz 32bit 0.5W CMOS RISC Microprocessor, *The Int'l Solid-State Circuits*, Vol.31, pp.1703–1714 (Nov. 1996).

[3] Calder, B., Grunwald, D. and Emer, J.: Predictive Sequential Associative Cache, *the 2nd IEEE International Symposium on High Performance Computer Architecture, Sab Hise*, pp.244–254 (Feb. 1996).

[4] Inoue, K., Ishihara, T. and Murakami, K.: Way-predicting set-associative cache for high performance and low energy consumption, *Proc. Int. Low Power Electronics and Design Symp.*, pp.273–275 (1999).

[5] Inoue, K., Ishihara, T. and Murakami, K.: A High-Performance and Low-Power Cache Architecture with Speculative Way-Selection, *IEICE Trans. on Electron*, Vol.E83-C, No.2, pp.24–36 (Feb. 2000).

[6] Zhang, C. and Asanovic, K.: A way-halting cache for low-energy high-performance systems, *ACM Trans. Archit. Code Optim.*, Vol.2, No.1, pp.126–131 (2005).

[7] Keramidas, G., Xekalakis, P. and Kaxiras, S.: Applying decay to reduce dynamic power in set-associative caches, *HiPEAC 2007*, Vol.4367, pp.38–53 (2007).

[8] Min, R., Xu, Z., Hu, Y., et al.: Partial Tag Comparison: A New Technology for Power-Efficient Set-Associative Cache Designs, *Proc. 17th International Conference on VLSI Design*, pp.183–188 (2004).

[9] Powell, M. et al.: Reducing set-associative cache energy via way-prediction and selective direct-mapping, *Proc. Int. Symp. on Microarchitecture*, pp.54–65 (2001).

[10] Zhu, Z. and Zhang, X.: Access-mode predictions for low-power cache design, *IEEE Micro*, Vol.22, No.2, pp.58–71 (Mar.-Apr. 2002).

[11] Kin, J., Gupta, M. and Mangione-Smith, W.H.: The filter cache: An energy efficient memory structure, *Proc. 30th Int. Microarchitecture Symp.*, pp.184–193 (Dec. 1997).

[12] Su, C.L. and Despain, A.M.: Cache design for energy efficiency, *Proc. 28th Int. System Sciences Conf.*, pp.306–315 (1995).

[13] Ghose, K. and Kamble, M.B.: Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation, *Proc. Int. Low Power Electronics and Design Symp.*, pp.70–75 (1999).

[14] Liu, W. and Huang, M.C.: EXPERT: Expedited simulation exploiting program behavior repetition, *ICS'04: Proc. 18th Annual International Conference on Supercomputing*, pp.126–135 (June 2004).

[15] Sherwood, T., Perelman, E., Hamerly, G., Sair, S. and Calder, B.: Discovering and exploiting program phases, *IEEE Micro: Micros Top Picks from Computer Architecture Conferences*, pp.217–227 (Dec. 2003).

[16] Brooks, D., Tiwari, V. and Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations, *Proc. 27th Annual International Symposium on Computer Architecture*, pp.83–94 (June 2000).

[17] Burger, D.C. and Austin, T.M.: The SimpleScalar tool set, version 2.0, *Comput. Architecture News*, Vol.25, No.3, pp.13–25 (June 1997).

[18] Shivakumar, P. and Jouppi, N.: CACTI 3.0: An integrated cache timing, power, and area model, *Compaq, Palo Alto, CA, WRL Res. Rep.* (Feb. 2001).

**Jiongyao Ye** was born in Shanghai, China on May, 1978. He received his B.E degree in electronic engineering in 2000, from Shanghai Marine University, where he was an assistant during 2000–2002. He received M.E. degree and his Ph.D. in engineering. from Waseda University in 2005 and 2011 respectively. He joined the Sony LSI Design Inc., where he worked in the field of LSI design from 2005 to 2008. He is currently working toward a Dr.Eng. degree in Graduate School of Information, Productions and Systems, in Waseda University. His research interests include micro-architecture, low-power FPGA and their applications. He is a member of IEICE.

**Hongfeng Ding** was born in Liaoyan, China on April, 1986. He received his B.E. degree in mathematics and applied mathematics from Beijing University of Posts and Telecommunications, China in 2009. His research interest includes high performance processor design and low-power design.

**Yingtao Hu** was born in Wuxi, China on May, 1986. He received his B.E. degree in software engineering from Dalian University of technology, China in 2008, and M.E. degree from Waseda University, Japan in 2011. His research interests includes low-power LSI design and information security. He is a student member of the IEICE.

**Takahiro Watanabe** was born in Ube, Japan on October, 1950. He received his B.E. and M.E. in electrical engineering from Yamaguchi University, and Dr.Eng. from Tohoku University. In 1979, he joined the Research and Development Center of Toshiba Corporation, where he worked in the field of LSI design automation. In August 1990, he joined Yamaguchi University, the Department of Computer Science and Systems Engineering, and in April 2003, he moved to Graduate School of Information, Production and Systems, Waseda University. His current research interests are EDA algorithm, Microprocessor and MPSoC, NoC, FPGA and their applications. He is a member of IEICE, IPSJ and IEEE.