ISSN (Online) : 2319 - 8753
ISSN (Print) : 2347 - 6710

International Journal of Innovative Research in Science, Engineering and Technology

Volume 3, Special Issue 3, March 2014

2014 International Conference on Innovations in Engineering and Technology (ICIET'14)
On 21st & 22nd March Organized by

K.L.N. College of Engineering, Madurai, Tamil Nadu, India

Low Power L2 Cache Architecture Using Partial Tag Bloom Filter

C.Gemma Benilda[1], R.Vijaya Bhasker[2]

[1]M.E. VLSI Design, Regional Centre , Anna University, Coimbatore, India

[2]M.E. VLSI Design, Regional Centre , Anna University, Coimbatore, India

ABSTRACT— In this paper, a new cache architecture referred to as Way-Tagged Cache is discussed to improve the energy efficiency of write-through caches. By maintaining the way tags of L2 cache in the L1 cache during read operations, it enables L2 cache to work in an equivalent direct-mapping manner during write hits, which account for the majority of L2 cache accesses. To reduce the energy consumed in tag comparison within highly associative L2 caches a Multistep Tag Comparison method is proposed along with the Partial Tag-Enhanced Bloom Filter to improve the accuracy of cache miss prediction. Similar results are also obtained under different L1 and L2 cache configurations. Simulation results on Altera demonstrate that the proposed technique achieves 50% reduced power consumption along with the reduced area overhead and no performance degradation. Furthermore, the idea of way tagging can be applied to existing low-power cache design techniques to further improve energy efficiency.

KEYWORDS— Cache; low power; write-through policy; Multistep tag comparison; Bloom Filter (BF);

I. INTRODUCTION

MULTI-LEVEL on-chip cache systems have been widely adopted in high-performance microprocessors. To keep data consistence throughout the memory hierarchy, write-through and write-back policies are commonly employed. Under the write-back policy, a modified cache block is copied back to its corresponding lower level cache only when the block is about to be replaced. While under the write-through policy, all copies of a cache block are updated immediately after the cache block is modified at the current cache, even though the block might not be evicted. As a result, the write-through policy maintains identical data copies at all levels of the cache hierarchy throughout most of their life time of execution. This feature is important as CMOS technology is scaled into the nanometer range, where soft errors have emerged as a major reliability issue in on-chip cache systems. It has been reported that single-event multi-bit upsets are getting worse in on-chip memories. Currently, this problem has been addressed at different levels of the design abstraction. At the architecture level, an effective solution is to keep data consistent among different levels of the memory hierarchy to prevent the system from collapse due to soft errors. Benefited from immediate update, cache write-through policy is inherently tolerant to soft errors because the data at all related levels of the cache hierarchy are always kept consistent. Due to this feature, many high-performance microprocessor designs have adopted the write-through policy.

While enabling better tolerance to soft errors, the write-through policy also incurs large energy overhead. This is because under the write-through policy, caches at the lower level experience more accesses during write operations. Consider a two-level (i.e., Level-1 and Level-2) cache system for example. If the L1 data cache implements the write-back policy, a write hit in the L1 cache does not need to access the L2 cache. In contrast, if the L1 cache is write-through, then both L1 and L2 caches need to be accessed for every write operation. Obviously, the write-through policy incurs more write accesses in the L2 cache, which in turn increases the energy consumption of the cache system. Power dissipation is now considered as one of the critical issues in cache design. Studies have shown that on-chip caches can consume about 50% of the total power in high-performance microprocessors [4].

In this paper, we propose new cache architecture, referred to as *way-tagged cache*, to improve the energy efficiency of write-through cache systems with minimal area overhead and no performance degradation. Consider a two-level cache hierarchy, where the L1 data cache is write-through and the L2 cache is inclusive for high performance. It is observed that all the data residing in the L1 cache will have copies in the L2 cache. In addition, the locations of these copies in the L2 cache will not change until they are evicted from the L2 cache. Thus, we can attach a tag to each way in the L2 cache and send this tag information to the L1 cache when the data is loaded to the L1 cache. By doing so, for all the data in

the L1 cache, we will know exactly the locations (i.e., ways) of their copies in the L2 cache. During the subsequent accesses when there is a write hit in the L1 cache (which also initiates a write access to the L2 cache under the write-through policy), we can access the L2 cache in an equivalent direct-mapping manner because the way tag of the data copy in the L2 cache is available. As this operation accounts for the majority of L2 cache accesses in most applications, the energy consumption of L2 cache can be reduced significantly. L2 caches is becoming increasingly popular in chips and are characterized by high switching power due to large amount of power consumed during tag comparison. High switching power in L2 cache is due to two factors. First is that the L2 cache is characterized by high associativity than the L1 cache. High associativity is adopted in L2 cache in order to reduce conflict misses. Second factor is that power consumed during tag comparison is expected to increase further due to cache coherence [1], [8].

II. Related Works

Cache architectures for low power are classified into three: tag comparison, data access and leakage. K. Inoue *et al.* [9] presented a way-predicting cache that predicts the most recently used (MRU) way which chooses one way before starting the normal cache-access process, and then accesses the predicted way. If the prediction is correct, the cache access has been completed successfully. Otherwise, the cache then searches for the remaining ways. They uses way-prediction and selective direct-mapping, to reduce L1 cache dynamic energy while maintaining high performance.

Z. Zhu *et al.* [3] presented a multiple MRU (MMRU) way that predicts an MRU way per partial tag. Based on this, cache hit and miss predictions are used for cache design with minimal energy consumption. Dai and Wang [6] proposed a way-tagged cache in order to reduce L2 cache tag accesses of a write-through L1 cache. Caches write-through policy gives performance improvement and at the same time achieving good tolerance to soft errors in on-chip caches. The cache hit prediction methods suffers from high penalty when the cache miss rates are high. The cache miss prediction methods will overcome this limitation.

Zhang *et al.* [2] proposed a new cache architecture, called a way-halting cache that reduces the energy while imposing no performance overhead. This way-halting cache is a four-way set-associative cache that stores the four lowest-order bits of all way tags into a fully associative memory, which we call the halt tag array. Further accesses to ways with known mismatching tags are then halted, thus saving power. M Ghosh *et al.* [12], proposed a new hit/miss predictor that uses a Bloom Filter to identify cache misses early in the pipeline.

Task scheduling of embedded application on multiple processors is meant to reduce the execution time. Benini *et al* [10] did the task scheduling using constraint programming and memory partitioning using integer linear programming.

The basic idea of way-tagged cache was initially proposed in our past work with some preliminary results. In this paper, we extend this work by making the following contributions. First, a detailed VLSI architecture of the proposed way-tagged cache is developed, where various design issues regarding timing, control logic, operating mechanisms, and area overhead have been studied. Second, we demonstrate that the idea of way tagging can be extended to many existing low-power cache design techniques so that better tradeoffs of performance and energy efficiency can be achieved. Third, a detailed energy model is developed to quantify the effectiveness of the proposed technique. Finally, a comprehensive suite of simulations is performed with new results covering the effectiveness of the proposed technique under different cache configurations. It is also shown that the proposed technique can be integrated with existing low-power cache design techniques to further improve energy efficiency.

The rest of this paper is organized as follows. In Section II, we provide a review of related low-power cache design techniques. In Section III, we present the proposed way-tagged cache. In Section IV, we discuss the detailed VLSI architecture of the way-tagged cache. Section V extends the idea of way tagging to existing cache design techniques using partial tag bloom filter to further improve energy efficiency. Simulation results are given in Section VI.

III. Way-Tagged Cache

In this section, we propose a way-tagged cache that exploits the way information in L2 cache to improve energy efficiency. We consider a conventional set-associative cache system when the L1 data cache loads/writes data from/into the L2 cache; all ways in the L2 cache are activated simultaneously for performance consideration at the cost of energy overhead. In Section V, we will extend this technique to L2 caches with phased tag-data accesses.

Only the L1 data cache and L2 unified cache are shown as the L1 instruction cache only reads from the L2 cache. Under the write-through policy, the L2 cache always maintains the most recent copy of the data. Thus, whenever a data is updated in the L1 cache, the L2 cache is updated with the same data as well. This results in an increase in the write accesses to the L2 cache and consequently more energy consumption.

Here we examine some important properties of write-through caches through statistical characterization of cache accesses. Unlike the L1 cache where read operations account for a large portion of total memory accesses, write operations are dominant in the L2 cache for all but three benchmarks (galgel, ammp, and art). This is because read accesses in the L2 cache are initiated by the read misses in the L1 cache, which typically occur much less frequently (the miss rate is less than 5% on average. For galgel, ammp, and art, L1 read miss rates are high resulting in more read accesses than write accesses. Nevertheless, write accesses still account for about 20%–40% of the total accesses in the L2 cache. From the results in Section VI, each L2 read or write access consumes roughly the same amount of energy on average. Thus, reducing the energy consumption of L2 write accesses is an effective way for memory power management.

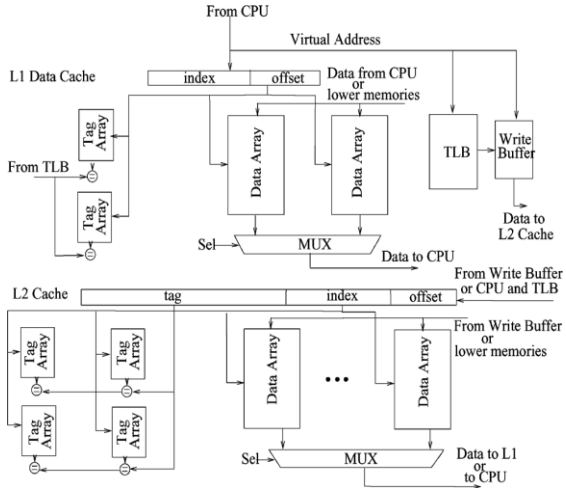

Fig. 1. Conventional two level cache

As explained in the introduction, the locations (i.e., way tags) of L1 data copies in the L2 cache will not change until the data are evicted from the L2 cache. The proposed way-tagged cache exploits this fact to reduce the number of ways accessed during L2 cache accesses. When the L1 data cache loads a data from the L2 cache, the way tag of the data in the L2 cache is also sent to the L1 cache and stored in a new set of way-tag arrays (see details of the implementation in Section IV). These way tags provide the key information for the subsequent write accesses to the L2 cache.

In general, both write and read accesses in the L1 cache may need to access the L2 cache. These accesses lead to different operations in the proposed way-tagged cache, as summarized in Table I. Under the write-through policy, all write operations of the L1 cache need to access the L2 cache. In the case of a write hit in the L1 cache, only one way in the L2 cache will be activated because the way tag information of the L2 cache is available, i.e., from the way-tag arrays we can obtain the L2 way of the accessed data. While for a write miss in the L1 cache, the requested data is not stored in the L1 cache. As a result, its corresponding L2 way information is not available in the way-tag arrays. Therefore, all ways in the L2 cache need to be activated simultaneously. Since write hit/miss is not known *a priori*, the way-tag arrays need to be accessed simultaneously with all L1 write operations in order to avoid performance degradation. Note that the way-tag arrays are very small and the involved energy overhead can be easily compensated for (see Section VII). For L1 read operations, neither read hits nor misses need to access the way-tag arrays. This is because read hits do not need to access the L2 cache; while for read misses, the corresponding way tag information is not available in the way-tag arrays. As a result, all ways in the L2 cache are activated simultaneously under read misses.

Write accesses account for the majority of L2 cache accesses in most applications. In addition, write hits are dominant among all write operations. Therefore, by activating fewer ways in most of the L2 write accesses, the proposed way-tagged cache is very effective in reducing memory energy consumption.

Fig. 3 shows the system diagram of proposed way-tagged cache. We introduce several new components: way-tag arrays, way-tag buffer, way decoder, and way register, all shown in the dotted line. The way tags of each cache line in the L2 cache are maintained in the way-tag arrays, located with the L1 data cache. Note that write buffers are commonly employed in write-through caches (and even in many write-back caches) to improve the performance. With a write buffer, the data to be written into the L1 cache is also sent to the write buffer. The operations stored in the write buffer are then sent to the L2 cache in sequence. This avoids write stalls when the processor waits for write operations to be completed in the L2 cache. In the proposed technique, we also need to send the way tags stored in the way-tag arrays to the L2 cache along with the operations in the write buffer. Thus, a small way-tag buffer is introduced to buffer the way tags read from the way-tag arrays. A way decoder is employed to decode way tags and generate the enable signals for the L2 cache, which activate only the desired ways in the L2 cache. Each way in the L2 cache is encoded into a way tag. A way register stores way tags and provides this information to the way-tag arrays.

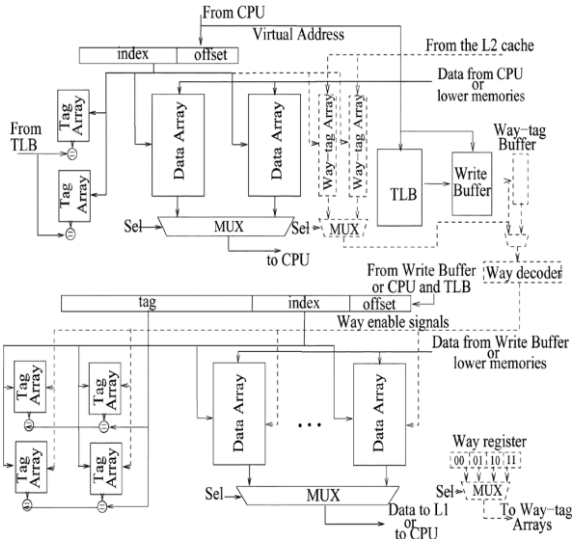

Fig. 2. Way-tagged Cache

TABLE I
EQUIVALENT ACCESS MODES UNDER DIFFERENT OPERATIONS IN THE L2 CACHE

	Operations in the L1 Cache			
	Read hit	Read miss	Write hit	Write miss
L2	No access	Set-associative	Direct-mapping	Set-associative

IV. IMPLEMENTATION OF WAY-TAGGED CACHE

In this section, we discuss the implementation of the way-tagged cache.

A. Way-Tag Arrays

In the way-tagged cache, each cache line in the L1 cache keeps its L2 way tag information in the corresponding entry of the way-tag arrays, as shown in Fig. 4, where only one L1 data array and the associated way-tag array are shown for simplicity. When a data is loaded from the L2 cache to the L1 cache, the way tag of the data is written into the way-tag array. At a later time

when updating this data in the L1 data cache, the corresponding copy in the L2 cache needs to be updated as well under the write-through policy. The way tag stored in the way-tag array is read out and forwarded to the way-tag buffer (see Section IV-B) together with the data from the L1 data cache. Note that the data arrays in the L1 data cache and the way-tag arrays share the same address as the mapping between the two is exclusive. The write/read signal of way-tag arrays, WRITEH_W, is generated from the write/read signal of the data arrays in the L1 data cache as shown in Fig. 4. A control signal referred to as UPDATE is obtained from the cache controller. When the write access to the L1 data cache is caused by a L1 cache miss, UPDATE will be asserted and allow WRITEH_W to enable the write operation to the way-tag arrays (WRITEH = 1, UPDATE = 1, see Table II). If a STORE instruction accesses the L1 data cache, UPDATE keeps invalid and WRITE_W indicates a read operation to the way-tag arrays (WRITEH = 1, UPDATE = 0). During the read operations of the L1 cache, the way-tag arrays do not need to be accessed and thus are deactivated to reduce energy overhead. To achieve this, the wordline selection signals generated by the decoder are disabled by WRITEH (WRITEH = 0, UPDATE = 0/1) through AND gates. The above operations are summarized in Table II.

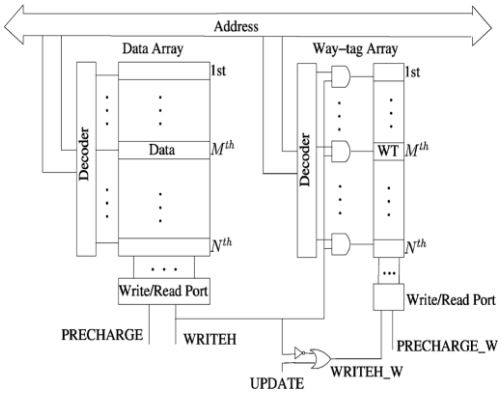

Fig. 3. Way-tag arrays

Note that the technique does not change the cache replacement policy. When a cache line is evicted from the L2 cache, the status of the cache line changes to "invalid" to avoid future fetching and thus prevent cache coherence issues.

TABLE II
OPERATIONS OF WAY TAG ARRAYS

WRITEH	UPDATE	OPERATION
1	1	write way-tag arrays
1	0	Read way-tag arrays
0	0	No access
0	1	No access

A read or write operation to this cache line will lead to a miss, which can be handled by the way-tagged cache (see Section III). Since way-tag arrays will be accessed only when a data is written into the L1 data cache (either when CPU updates a data in the L1 data cache or when a data is loaded from the L2 cache), they are not affected by cache misses. It is important to minimize the overhead of way-tag arrays. The size of a way-tag array can be expressed as

$$WTsize = \frac{SL1 \times Bway, L2}{Sline,L1 \times Nway,L1} \quad (1)$$

where SL1,Sline,L1 , and Nway,L1 are the size of the L1 data cache, cache line size, and the number of ways in the L1 data cache, respectively. Each way in the L2 cache is represented by Bway, L2 = log2Nway,L2 bits assuming the binary code is applied. As shown in (1), the overhead increases linearly with the size of L1 data cache SL1 and sublinearly with the number of ways in L2 cache Nway,L2. In addition, since Bway, L2 is very small compared with Sline, L1 (i.e., Bway, L2 / Sline, L1 <<1), the overhead accounts for a very small portion of the L1 data cache. Clearly, the technique shows good scalability trends with the increasing sizings of L1 and L2 caches. As an example, consider a two-level cache hierarchy where the L1 data cache and instruction cache are both 16 kB 2-way set-associative with cache line size of 32 B. The L2 cache is 4-way set-associative with 32 kB and each cache line has 64 B. Thus, SL1 = 16 kB, Sline, L1 = 32 B, Nway,L1 =2 , and Bway, L2 = 2. The size of each way-tag array is 16 K/(32×2)×2 = 512 bits, and two way-tag arrays are needed for the L1 data cache. This introduces an overhead of only (512×2)/(16 K × 8) = 0.78% of the L1 data cache, or (512×2)/((16 K +16 K+ 32 K) × 8) = 0.03% of the entire L1 and L2 caches.

To avoid performance degradation, the way-tag arrays are operated in parallel with the L1 data cache. Due to their small size, the access delay is much smaller than that of the L1 cache. On the other hand, the way-tag arrays share the address lines with the L1 data cache. Therefore, the fan-out of address lines will increase slightly. This effect can be well-managed via careful floor-plan and layout during the physical design. Thus, the way-tag arrays will not create new critical paths in the L1 cache. Note that accessing way-tag arrays will also introduce a small amount of energy overhead. However, the energy savings achieved by the technique can offset this overhead, as shown in Section VII.

B. Way-Tag Buffer

Way-tag buffer temporarily stores the way tags read from the way-tag arrays. The implementation of the way-tag buffer is shown in Fig. 4. It has the same number of entries as the write buffer of the L2 cache and shares the control signals with it. Each entry of the way-tag buffer has n+1 bits, where is the line size of way-tag arrays. An additional status bit indicates whether the operation in the current entry is a write miss on the L1 data cache. When a write miss occurs, all the ways in the L2 cache need to be activated as the way information is not available. Otherwise, only the desired way is activated. The status bit is updated with the read operations of way-tag arrays at the same clock cycle.

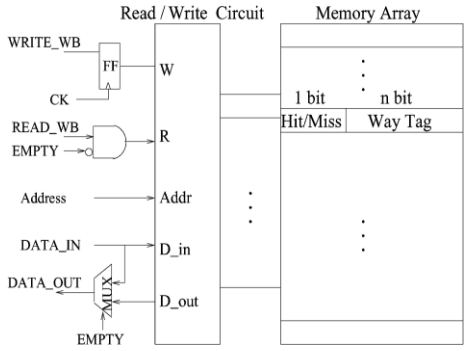

Fig. 4. Way-tag Buffer

Similar to the write buffer of the L2 cache, the way-tag buffer has separate write and read logic in order to support parallel write and read operations. The write operations in the way-tag buffer always occur one clock cycle later than the corresponding write operations in the write buffer. This is because the write buffer, L1 cache, and way-tag arrays are all updated at the same clock cycle when a STORE instruction accesses the L1 data cache (see Fig. 3). Since the way tag to be sent to the way-tag buffer comes from the way-tag arrays, this tag will be written into the way-tag buffer one clock cycle later. Thus, the write signal of the way-tag buffer can be generated by delaying the write signal of the write buffer by one clock cycle, as shown in Fig. 4.

The way-tagged cache needs to send the operation stored in the write buffer along with its way tag to the L2 cache. This requires sending the data in the write buffer and its way tag in the way-tag buffer at the same time. However, simply using the same read signal for both the write buffer and the way-tag buffer might cause write/read conflicts in the way-tag buffer. Assume that at the Nth clock cycle an operation is stored into the write buffer while the way-tag buffer is empty. At the (N+1)th clock cycle, a read signal is sent to the write buffer to get the operation while its way tag just starts to be written into the way-tag buffer. If the same read signal is used by the way-tag buffer, then read and write will target the same location of the way-tag buffer at the same time, causing a data hazard.

One way to fix this problem is to insert one cycle delay to the write buffer. This, however, will introduce a performance penalty. In this paper, we propose to use a bypass multiplexer (MUX in Fig. 3) between the way-tag arrays and the L2 cache. If an operation in the write buffer is ready to be processed while the way-tag buffer is still empty, we bypass the way-tag buffer and send the way tag directly to the L2 cache. The EMPTY signal of the way-tag buffer is employed as the enable signal for read operations; i.e., when the way-tag buffer is empty, a read operation is not allowed. During normal operations, the write operation and the way tag will be written into the write buffer and way-tag buffer, respectively. Thus, when this write operation is ready to be sent to the L2 cache, the corresponding way tag is also available in the way-tag buffer. With this bypass multiplexer, no performance overhead is incurred.

C. Way Decoder

The function of the way decoder is to decode way tags and activate only the desired ways in the L2 cache. As the binary code is employed, the line size of way-tag arrays is n = log2 N bits, where N is the number of ways in the L2 cache. This minimizes the energy overhead from the additional wires and the impact on chip area is negligible. For a L2 write access caused by a write hit in the L1 cache, the way decoder works as a n-to-N decoder that selects just one way-enable signal. The technique in [19] can be employed to utilize the way-enable signal to activate the corresponding way in the L2 cache.

The way decoder operates simultaneously with the decoders of the tag and data arrays in the L2 cache. For a write miss or a read miss in the L1 cache, we need to assert all way-enable signals so that all ways in the L2 cache are activated. To achieve this, the way decoder can be implemented by the circuit shown in Fig. 7. Two signals: read and write miss, determine the operation mode of the way decoder. Signal read will be “1” when a read access is sent to the L2 cache. Signal write miss will be “1” if the write operation accessing the L2 cache is caused by a write miss in the L1 cache.

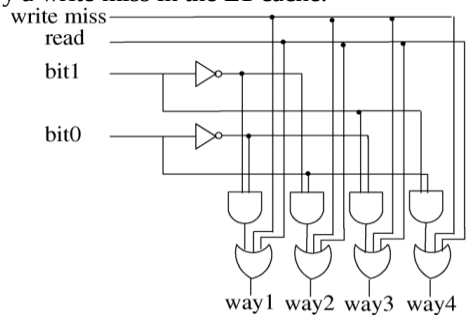

Fig. 5. Way decoder

D. Way Register

The way register provides way tags for the way-tag arrays. For a 4-way L2 cache, labels “00”, “01”, “10”, and “11” are stored in the way register, each tagging one way in the L2 cache. When the L1 cache loads a data from the L2 cache, the corresponding way tag in the way register is sent to the way-tag arrays. With these new components, the way-tagged cache operates under different modes during read and writes operations (see Table I). Only the way containing the desired data is activated in the L2 cache for a write hit in the L1 cache, making the L2 cache equivalently a direct-mapping cache to reduce energy consumption without introducing performance overhead.

V. PROPOSED L2 CACHE MULTIPROCESSOR USING PARTIAL TAG BLOOM FILTER

A.Implementation of L2 Cache

The implementation of the proposed method is shown in Fig.6. It includes Tag comparison control (TC), Bloom filter (BF), Tag, Data array, Miss State Holding Register (MSHR) and write buffer [5]. The proposed multistep tag comparison method combines both cache hit and miss prediction. The control logic for tag comparison called

tag comparison control takes the input address from L1 cache and determines how many steps the tag comparison requires. It also performs the timeout countdown i.e., the tag counter update, depending on the tag comparison results. The tag counter is updated on sampled accesses (once for 128 accesses in our experiments) in order to reduce the energy consumption of accessing local TO counters.

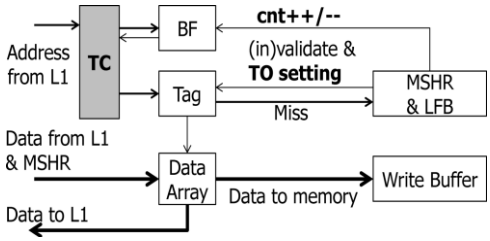

Fig. 6. Implementation of proposed methods

In Fig.6 miss state holding register (MSHR) is seen, which plays the major role in the implementation. From the figure it is clear that MSHR performs the existing functions, i.e. cache replacement, issuing a cache request to the main memory, and it sets the local TO counter of the cache line. In addition to the existing functions in the proposed method, the MSHR updates the Bloom filter in both cases of line-fill (to increment the corresponding counter) and eviction (to decrement the counter). Upon a L2 cache miss, when the requested line arrives at the L2 cache, it is first given to the L1 cache and then written to the L2 cache. The MSHR updates the bloom filter with subsequent accesses to the bloom filter. In such cases, the MSHR request has priority over the subsequent one and delays the latter by one clock.

B.Multistep Tag Comparison

The proposed multistep tag comparison method combines both cache hit and miss prediction. It is dynamic in nature i.e. dynamically adjusts the order of tag comparison steps to maximize the efficiency of cache hit and miss predictions. We utilize the hot hit ratio to determine which method to apply first. Fig.7 shows the two possible configurations used in the proposed dynamic multistep tag comparison [5]. At low or medium hot hit rates, we apply as Fig. 7(a), where a partially tagged Bloom filter (pBF) is first applied because the number of cache accesses filtered by the Bloom filter (= #total accesses × cache miss rate × cache miss prediction accuracy) increases as the hot hit rate.

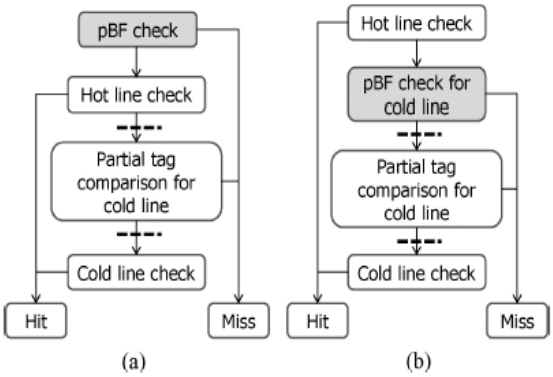

Fig. 7. Hot Hit Ratio Based Multistep Tag Comparisons. (a)Medium/low Hot Hit Ratio. (b) High Hot Hit Ratio.

This reduces the tag comparisons otherwise required only to give cache misses as the results while wasting energy. At high hot hit rates, as in Fig. 7(b) shows, the hot line check is performed before the partially tagged Bloom filter check. This order is adopted because the hot line check is more likely to give cache hits, which allows both subsequent Bloom filter checks and tag comparisons for cold lines to be skipped thereby reducing energy consumption.

A threshold of hot hit ratio is used in the proposed method (obtained during the design stage) and compared with the current hot hit ratio (calculated during runtime) and the threshold is used to make a runtime decision regarding which of the two configurations in Fig. 7 is applied first. The best performing threshold varies with programs as the cache access behavior varies between programs.

C. Partially Tag-Enhanced Bloom Filter

In the case of a singleton entry, the original bloom filter will not specify whether the incoming address is present in the cache way. A partial tag for each BF entry is proposed to check the presence of incoming address in the case of singleton entry. A BF entry has a tuple <C, Z, S, P>, where C is the counter, Z is the zero flag, S is the singleton flag, and P is the partial tag. The size of the partial tag is small, 3 bits then the singleton check and the partial tag match are performed. If there is a mismatch between the partial tag in the incoming address and the singleton entry, the result is a miss in the cache way.

Thus, compared to the original Bloom filter, the partial tag-enhanced Bloom filter has an overhead of 4 bits (including the S flag) per entry. On each entry/exit (program/de-program) of address to/from the Bloom filter, the corresponding partial tag is calculated in bitwise XOR operations as follows:

$$PTagnew = PTagold \text{ XOR } PTagin \quad (1)$$

$$PTagnew = PTagold \text{ XOR } PTagout \quad (2)$$

where PTagold and PTagnew represents the old and newly calculated partial tags, respectively. PTagin and PTagout represent the partial tags of the incoming (i.e., newly fetched) and outgoing (i.e., evicted) cache lines, respectively. Such a partial tag manipulation gives the partial tag of the currently existing address in the case of a singleton entry.

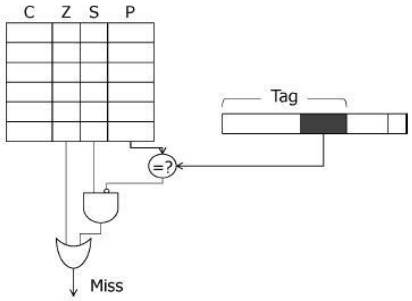

Fig. 8. Bloom filter with a partial tag

```
1  BF_query() {            // for each BF query
2    If Z=1, return 'miss'
3    Else
4      If S=1 & partial tag mismatch, return 'miss'
5      Else, return 'likely hit'
6  }
7  BF_entry/exit() {       // for each BF entry/exit
8    Counter++ for entry (Counter-- for exit)
9    If Counter = 0, Z=1
10   If Counter = 1, S=1
11   Partial tag calculation
12 }
```

Fig. 9. Partial tag enhanced bloom filter operation

A pseudo code of the partially tagged bloom filter operation when k=1 is shown in the Fig.9. Compared to the original functionality of the bloom filter for cache miss predictions, the new functionality is shown in bold [5]. If the counter of the corresponding BF entry is not zero, then the singleton check and partial tag match are performed. If there is a mismatch between the partial tags of singleton entry and the incoming address, the access is a miss for the corresponding way (line 4).

In the singleton cases where the partial tag comparison gives a match and in non-singleton cases, subsequent tag comparison needs to be performed ("likely hit" case in line 5). Note that when multiple ($k > 1$) hash functions are used, if all the Bloom filter entries hashed by the *k* hash functions are singletons, all the singleton entries give the same partial tag. Thus, in the case of multiple hash functions, the singleton test in line 4 checks to see if all the *k* S flags are "1."

VI. EXPERIMENTAL RESULTS

A. *Power Analysis And Comparison*

The total estimated power consumption of the level 2 cache architecture is shown in fig. 10. It is approximately about 350mW. This high power consumption is due to the write-through policy which incurs more write accesses in the L2 cache. Thus, whenever a data is updated in the L1 cache, the L2 cache is updated with the same data as well. This results in an increase in the write accesses to the L2 cache and consequently more energy consumption.

Fig. 11. shows the reduced total estimated power consumption of way-tagged cache of about 170mW. This is approximately 50% reduction in power analysis. It is done by activating only activating fewer ways in most of the L2 write accesses which accounts for the majority of L2 cache accesses in most applications.

Power summary:	I(mA)	P(mW)
Total estimated power consumption:		352
Vccint 2.50V:	138	346
Vcco33 3.30V:	2	7
Clocks:	0	0
Inputs:	11	27
Logic:	27	68
Outputs:		
Vcco33	0	0
Signals:	92	230
Quiescent Vccint 2.50V:	8	20

Fig. 10. Power Analysis of Conventional 2-Level Cache

Power summary:	I(mA)	P(mW)
Total estimated power consumption:		174
Vccint 1.80V:	93	168
Vcco33 3.30V:	2	7
Clocks:	0	0
Inputs:	7	13
Logic:	4	7
Outputs:		
Vcco33	0	0
Signals:	2	3
Quiescent Vccint 1.80V:	80	144

Fig 11. Power Analysis of Way-Tagged Cache

VII. CONCLUSION

This paper presents a new energy-efficient cache technique for high-performance microprocessors employing the write through policy. The proposed technique attaches a tag to each way in the L2 cache. This way tag is sent to the way-tag arrays in the L1 cache when the data is loaded from the L2 cache to the L1 cache. Utilizing the way tags stored in the way tag arrays, the L2 cache can be accessed as a direct-mapping cache during the subsequent write hits, thereby reducing cache energy consumption. We proposed a multistep tag comparison method to reduce the energy consumed in tag comparison within highly associative L2 caches. We presented a partial tag-enhanced Bloom filter to improve the accuracy of cache miss prediction. To further reduce tag comparisons, we presented a partial tag comparison that takes place during cold checks. Simulation results demonstrate significantly reduction in cache energy consumption with minimal area overhead and no performance degradation. The proposed method reduces the total cache energy consumption by 8.86% as compared to existing methods. In future, we will investigate the effectiveness of the proposed method in multi-core environments.

ACKNOWLEDGEMENT

The author would like to thank All India Council for Technical Education (AICTE), India. The authors would also like to thank the Management and Principal of Regional Center- Anna University,Coimbatore for providing excellent computing facilities and encouragement.

AUTHOR'S PROFILE

Dr. R.Vijayabhasker[2] completed U.G in EEE, P.G in Power Electronics & Drives, Ph.D in Electrical Engg.. Working as Assistant Professor, ECE Dept, Anna University Regional Center. Field of interests are VLSI Design, VLSI signal processing, DSP. C.Gemma Benilda[1] completed U.G in ECE and pursuing P.G in VLSI Design.

REFERENCES

[1] ARM Ltd. (2011). *CoreLink CCI-400 Cache Coherent Interconnect (CCI)* [Online]. Available: http://www.arm.com.

[2] C. Zhang, F. Vahid, J. Yang, and W. Najjar, "A way-halting cache for low-energy high-performance systems," in *Proc. ISLPED*, 2004, pp. 126–131.

[3] G. Keramidas, P. Xekalakis, and S. Kaxiras, "Applying Decay to reduce dynamic power in set-associative caches," in *Proc. Int. Conf.*

High- Performance Embedded Architectures Compilers, 2007, pp. 38–53.
[4] Hassan Salamy and Ramanujam, 'An Effective Solution to Task Scheduling and Memory Partitioning for Multiprocessor System-on-Chip,' *IEEE transaction on computer aided design of integrated circuits and systems*, vol.31, no.5, May 2012, pp. 717-726.
[5] Hyunsun, Park. Sungjoo, Yoo and Sunggu, Lee (2012) 'A multistep tag comparison method for a low-power L2 cache' *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 31, no.4,559-573.
[6] J. Dai and L. Wang, "Way-tagged cache: An energy-efficient L2 cache architecture under write-through policy," in *Proc. ISLPED*, 2009, pp. 159–164.
[7] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *Proc. Supercomputing*, 2002, pp. 189–198.
[8] K. Aisopos, C. Chou, and L. Peh, "Extending open core protocol to support system-level cache coherence," in *Proc. CODES+ISSS*, 2008, pp. 167–172.
[9] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *Proc. ISLPED*, 1999,
pp. 273–275.
[10] L. Benini, D. Bertozzi, A. Guerri, and M. Milano, "Allocation and scheduling for MPSOC via decomposition and no-good generation," in *Proc. IJCAI*, 2005, pp. 107–121.
[11] M. D. Powell, A. Agarwal, T. N. Vijaykumar, M. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proc. Int. Symp. Microarchitecture*, 2001,pp. 54–65.
[12] M. Ghosh and X. Zhang, "Access-mode predictions for low-power cache design," *IEEE Micro*, vol. 22, no. 2, pp. 58–71, Apr. 2002.