

Low-Power L2 Cache Architecture for Multiprocessor System on Chip Design

Sharon Mathew

ME VLSI Design
Sri Ramakrishna Engineering College
Coimbatore, India

Dr. M Jagadeeswari

ME VLSI Design
Sri Ramakrishna Engineering College
Coimbatore, India

Abstract—Significant portion of cache energy in a highly associative cache is consumed during tag comparison. In this paper tag comparison is carried out by predicting both cache hit and cache miss using multistep tag comparison method. A partially tagged bloom filter is used for cache miss predictions by checking the non-membership of the addresses and hotline check for cache hit prediction by reducing the tag comparisons. Current complex embedded application employs a multiprocessor system-on-chip (MPSoC). A MPSoC consists of multiple processors, shared memory hierarchy and a global off-chip memory. This architecture meets the performance requirements of multimedia application respecting the constraints on memory, cost, size, time and power. Scheduling the tasks of an embedded application on the processors and partitioning the available Level2 (L2) cache budget among these processors are two critical issues in such systems. In this paper, an integrated approach is used for task scheduling and L2 cache partitioning to further reduce the execution time of embedded applications.

Keywords—Multistep tag comparison; cache memory; bloom filter (BF); dynamic reordering; timeout tracking; memory partitioning; task scheduling; multiprocessor system-on-chip

I. INTRODUCTION

A CPU cache is a cache used by the central processing unit of a computer to reduce the average time while accessing the main memory. The cache memory is a smaller and faster memory which stores copies of the data from the locations which are most frequently used i.e. the main memory. As long as the most memory accesses of the processor are cached, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

Multi-level caches generally operate by checking the smallest Level 1 (L1) cache first, if it hits, the processor proceeds by taking data from L1 cache at high speed. If a miss is given by the smaller cache, then the next larger cache Level 2 (L2) cache is checked, and so on, before the main memory is checked. 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].

Architectures with multiple processors on a single chip is an attractive solution in embedded application. Multiprocessor system on chip (MPSoC) consisting of multiple processing elements, shared L2 cache hierarchy and I/O components interconnected. In embedded systems which are very complex, increase in memory access speed has failed to maintain with the increase in processor speed. This makes the latency of memory access a major issue in scheduling applications on embedded systems.

Multiprocessor system-on-chip model uses a memory hierarchy with fast on-chip L2 caches and a slow off-memory. Such a memory hierarchy enables proper allocation of variables to the on-chip multi-step tag comparison L2 cache by reducing the off-chip accesses. The execution time of a program by a processor depends on how much L2 cache is allocated to that processor.

II. RELATED WORKS

Cache architectures for low power are classified into three: tag comparison, data access and leakage. Koji *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. Powell *et al.* [11] uses way-prediction and selective direct-mapping, to reduce L1 cache dynamic energy while maintaining high performance.

Z. Zhu *et al.* [3] and [15] 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.* [10] and [11], 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. Panda *et al* [13], [14] presented a comprehensive allocation technique for scratchpad memory for uniprocessor.

III. PRILIMINARIES AND MOTIVATIONS

A. Implementation of L2 Cache

The implementation of the proposed method is shown in Fig.1. 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.

In Fig.1 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

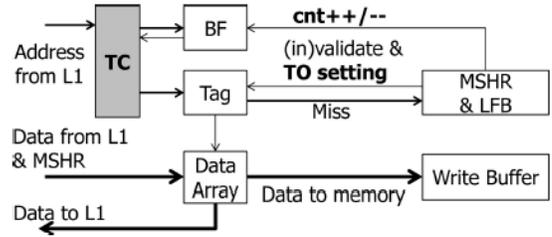


Figure. 1 Implementation of the proposed method

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 cycle.

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.2 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. 2(a), where a partially tagged Bloom filter (pBF) is first applied because the number of cache accesses filtered by the Bloom filter ($= \text{\#total accesses} \times \text{cache miss rate} \times \text{cache miss prediction accuracy}$) increases as the hot hit rate decreases (i.e., cache miss rate increases). 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. 2(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. 2 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. Fig. 3 shows a partially tagged counting bloom filter [5].

A BF entry has a tuple $\langle C, Z, S, P \rangle$, 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.

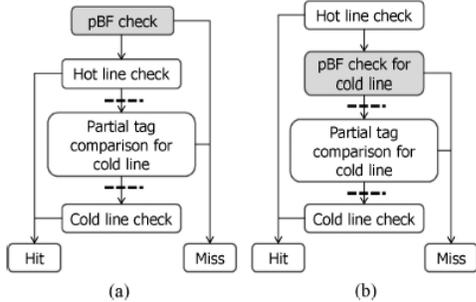


Figure.2 Hot Hit Ratio Based Multistep Tag Comparisons. (a)Medium/low Hot Hit Ratio. (b) High Hot Hit Ratio.

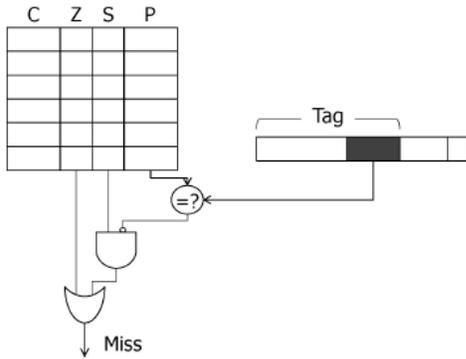


Figure. 3 Bloom filter with a partial tag

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:

$$P_{Tagnew} = P_{Tagold} \text{ XOR } P_{Tagin} \quad (1)$$

$$P_{Tagnew} = P_{Tagold} \text{ XOR } P_{Tagout} \quad (2)$$

where P_{Tagold} and P_{Tagnew} represents the old and newly calculated partial tags, respectively. P_{Tagin} and P_{Tagout} 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.

A pseudo code of the partially tagged bloom filter operation when $k=1$ is shown in the Fig.4. Compared to the original functionality of the bloom filter for cache miss predictions, the new functionality is shown in bold [5]. If the zero flag is zero of the corresponding bloom filter entry,

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 corresponding cache way.

```

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 }

```

Figure. 4 Partial tag enhanced bloom filter operation

IV. PROPOSED L2 CACHE MULTIPROCESSOR SYSTEM-ON-CHIP (MPSoC)

A. Architecture Overview

Fig. 5 shows a MPSoC architecture which consist of multiple processors, a shared L2 cache divided among the multiple processors and a global off-chip memory that can be accessed by these processors [4]. This technique can also be used for architecture where each processor has a private L2 cache such that each processor can access the L2 cache of other processors.

An embedded application is divided into a set of tasks where the available processors execute one or more independent tasks in parallel. This is extremely useful in MPSoC and leads the potential to speed up the execution time.

An embedded application usually consists of computational blocks, where each block is considered as a task. Each task is dependent on another task and should be respected in the schedule.

B. Task dependence graph (TDG)

A graph is a nodes and edges that connect the nodes. A TDG is a directed acyclic graph where each vertex is a task in the embedded application and with weighted edges. T_i and T_j are two tasks in the TDG. Fig. 6 shows a task dependence graph [4]. An edge from T_i to T_j represents a scheduling order where T_j is executed only when necessary data is obtained from T_i after the execution of it. Communication cost is defined as the weight of the edge.

The execution of a task T_j cannot be started until necessary data communication is carried out. Communication cost is the weight of the edge. Fig. 6 shows a TDG and from this figure it is clear that task T_i is

computed only after the execution of T_i , T_j and T_k . And also T_n is computed only after the execution of T_l and T_m .

Each task can be mapped into any of the available processors. The time taken to complete each task depends on the processor into which the task is mapped and also the L2 cache memory allocated to that processor. A larger L2

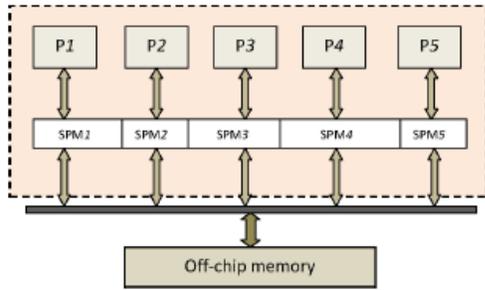


Figure. 5 Architectural model of five processors.

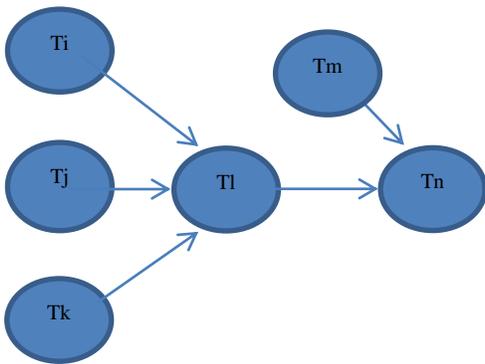


Figure. 6 Task dependence graph

cache results in less computational time because accessing the off-chip memory is expensive compared to the fast on-chip L2 cache.

C. An example

Consider an example in Fig. 7(a) of a task dependence graph with six tasks T_1 , T_2 , T_3 , T_4 , T_5 and T_6 . Task T_4 execution depends on task T_1 , T_2 and T_3 and task T_6 depends on the execution of tasks T_4 and T_5 . An edge between the tasks T_i and T_j has a weight and called as the communication cost should be accounted such that these two tasks are allocated to two different processors [4].

Define Min_{ik} , Avg_{ik} , and Max_{ik} as the computation time for task T_i on processor P_k assuming all of the available L2 cache budget is assigned to P_k , $1/n$ of the available L2 cache budget is assigned to P_k where n is the number of processors, and no L2 cache is assigned to P_k , respectively. These values are used later in the heuristic. Here we assume two processors.

Fig. 7(b) shows scheduling of tasks without L2 cache. Tasks T_1 , T_2 , T_3 and T_5 are ready to execute [4]. So we map first tasks T_1 and T_2 to processors P_1 and P_2 respectively. The scheduling algorithm schedules the task T_3 to P_2 because the processor P_2 is free before P_1 since the execution time of T_2 is less than T_1 . Same way the tasks T_4 and T_6 are assigned to the processor P_1 and task T_5 to processor P_2 . The overall cost of the schedule is 29.

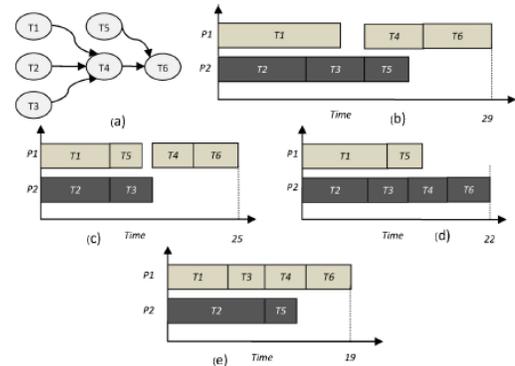


Figure. 7(a) TDG. Schedule based on (b) no L2 cache, (c) equal partitioned L2 cache, (d) non-equal partitioned L2 cache, (e) integrated approach

The scheduling of tasks with equal partitioned L2 cache between the two available processors is shown in Fig. 7(c). With this scheduling the L2 cache is divided equally between the two processors P_1 and P_2 regardless of what tasks are mapped to the processors. Here task T_1 is mapped to P_1 and task T_2 to P_2 . After the execution of T_2 , T_3 is given to P_2 . Since T_4 is executed only after the completion of the tasks T_1 , T_2 and T_3 , T_5 is mapped to P_1 after the execution of T_1 . Finally T_4 and T_6 are given to P_1 [4].

Fig. 7(d) shows the scheduling of tasks which further reduce the applications computational time by dividing the available L2 cache in any ratio between the processors [4]. In the TDG tasks are scheduled such that T_4 starts its execution only after completing the execution of task T_3 by P_2 . So there is a dead time between tasks T_1 and T_4 .

The issue is to reduce this dead time by imposing the computation time of tasks T_2 and T_3 . The solution used here to reduce the dead time is to allocate more L2 cache to processor P_2 in order to reduce the computation time of tasks T_2 and T_3 . If all the L2 cache is allocated to P_2 then the computation time of T_1 will increase and thereby the minimum start time of T_4 will increase. To avoid this some L2 cache is allocated to P_1 to balance the execution time.

Here the tasks are scheduled in such a way that T_1 is given to P_1 and T_2 to P_2 . Since the execution time of T_2 is less than that of T_1 , T_3 is mapped to P_2 . After the execution

of T2, tasks T3, T4 and T6 are mapped to P2 whereas T5 is mapped to P1.

The integrated approach integrates task scheduling and memory partitioning into one step. Here the L2 cache is divided both equally and unequally. The problem with the previous schedule is that it allocates T3 to the same processor P2 that is scheduled to execute T2. This scheduling is the reason for dead time in the schedule as T2 cannot benefit much from more L2 cache memory.

Fig. 7(d) shows the integrated approach. In an integrated approach task T1 is mapped to P1 and task T2 to P2 [4]. Since the execution time of T1 is less than T2, T3 is mapped to P1. Finally T4 and T6 is mapped to P1 and T5 to P2.

D. Integrated heuristic

In the integrated heuristic in Fig.8 it starts with profiling the application to extract important information [4]. From the information of profiling data, the embedded application can be divided into tasks with a necessary data communication between two tasks imposing a certain kind of dependence. Based on the extracted tasks and the communication between them, the task dependence graph is created. In this graph, each task is represented by a vertex [2] and each communication cost by a weighted directed edge.

For each available task T_i and processor P_j , we calculate the number of variables, the size of the variables, and Min_{ij} , Avg_{ij} , and Max_{ij} values. Profiling is used to compute these values. Then, the ASAP values for all tasks are calculated based on the Avg values that are assuming the L2 cache budget which is equally divided among the available processors. Tasks will be sorted in an increasing order of the ASAP values in a list L_1 . For each task, following the ASAP sort, we evaluate the best processor to assign this task so that the overall computation time is minimally increased.

The minimum start time of a task T_i on processor P_j , $Start-time(T_i, P_j)$, is equal to the maximum of the end time of processor P_j , $End-time(P_j)$, and the maximum end time of all its parent tasks, $MaxT_j \in Parent(T_i)(T_j)$, plus the corresponding communication time. Two dependent tasks mapped to the same processor will have zero communication cost. In general, task T_i will be scheduled on the processor P_j corresponding to the minimum additional overhead time in the schedule.

However, T_i may be scheduled on a processor P_k of higher overhead time provided that the predicted end computation time (PEC(P_k)) of this processor is at least $\delta\%$ less than that of P_j (Line 17 of integrated heuristic in Fig. 8). P_j on Line 17 of Fig. 8 represents the processor corresponding to the current min value. The min value in the heuristic is the minimum additional overhead time that will be added to the schedule based on a certain task scheduling

decision. We choose δ to be 10 in these experimental evaluations. This PEC (P_k) value is a guide to the scheduler of how much this overhead time may decrease with additional L2 cache memory transfers in future steps if T_i is mapped to P_k . PEC is an estimate of how much the end time of processor P_k will be if more L2 cache is assigned to it.

V. EXPERIMENTAL RESULTS

The simulation result of L2 cache architecture for Multiprocessor System on Chip (MPSoC) is obtained using XILINX ISE 12.3i with SPATAN 3E XCS500 as the target

```

Task Scheduling and Memory Partitioning
1. Divide the application into tasks  $T_i$ .
2. Perform dependence analysis between tasks.
3. Construct the TDG based on dependence analysis and communication costs.
4. Divide the SPM memory equally between the processors.
5. For each task  $T_i$  and processor  $P_j$ , extract the following:
6. (i) Minimum computation time on  $P_j$ ,  $Min_{ij}$ .
7. (ii) Maximum computation time on  $P_j$ ,  $Max_{ij}$ .
8. (iii) Average computation time on  $P_j$ ,  $Avg_{ij}$ .
9. Find ASAP for all the tasks based on Avg values.
10.  $L_1 =$  List of tasks in increasing order of ASAP.
11. While ( $L_1$  not empty) do:
12.   Get the first task  $T_j$  from  $L_1$ .
13.   For each processor  $P_k$ :
14.     Calculate the elasticity and PEC of  $P_k$  if  $T_j$  is mapped to  $P_k$ .
15.     Find the minimum start time of  $T_j$  on  $P_k$ .
16.     Find  $END\_time(P_k)$  if  $T_j$  is mapped to  $P_k$ .
17.     if  $((END\_time(P_k) < min \ \&\& \ PEC(P_j) \geq (1 - \delta\%)PEC(P_k))) \ \&\&$ 
18.        $(END\_time(P_k) > min \ \&\& \ PEC(P_k) \leq (1 - \delta\%)PEC(P_j))$ 
19.        $min = END\_time(P_k)$ 
20.     else if  $(END\_time(P_k) = min)$ 
21.        $min = END\_time$  of processor with the higher elasticity.
22.   End For
23.   Assign  $T_j$  to  $P_j$  corresponding to  $min$ .
24.   Delete  $T_j$  from  $L_1$ .
25.   Call Balance().
26. End While
27. For  $i = 1$  to  $t$  do:
28.   Call Balance().

```

Figure. 8 Integrated scheduling heuristic

device. The simulation result shows that the integrated heuristic with L2 cache is having less power and area compared to no L2 cache, equal partitioned L2 cache and non-equal partitioned L2 cache approaches. The power analysis shows that the MPSoC with shared L2 cache using integrated heuristic has a power consumption of 52mW which is less than the other three scheduling approaches. Table 1 shows the comparison of power analysis of the four different scheduling methods. The total equivalent gate count of scheduling based on integrated heuristic is 17562 which is 15210 gate count less than the other three approaches. Table 2 shows the gate count of all the scheduling. Thus it is clear that MPSoC with shared L2 cache using integrated heuristic is having 17% less power compared to the other three scheduling approaches.

VI. CONCLUSION

The partially tagged bloom filter and hotline check ensures use of both cache hit and cache miss prediction. A partial tag- enhanced Bloom filter is used to improve the accuracy of cache miss prediction and hot/cold checks (with dynamic time out tracking) as a cache hit prediction method. This multistep tag comparison L2 cache is used in multiprocessor system on chip for embedded application.

This embedded application is divided into several dependent tasks and using a task dependence graph embedded application is executed.

The power and area of the L2 cache architecture for Multiprocessor System on Chip (MPSoC) is analyzed using XILINX ISE 12.3i with STARTAN 3E XCS 250 as the target device. From the power and area analysis result it is clear that there is a reduction in power from 63mW to 52mW and reduction in gate count from 32772 to 17562.

TABLE. I COMPARISON FOR POWER ANALYSIS FOR INTEGRATED APPROACH

Power	Schedule based on no L2 cache (Mw)	Schedule based on equal partitioned L2 cache (mW)	Schedule based on non-equal partitioned L2 cache (mW)	Schedule based on integrated heuristic (mW)
Total Power	63mW	59mW	54mW	52mW

TABLE. II COMPARISON FOR AREA ANALYSIS FOR INTEGRATED APPROACH

Area	Schedule based on no L2 cache	Schedule based on equal partitioned L2 cache	Schedule based on non-equal partitioned L2 cache	Schedule based on integrated heuristic
Total equivalent gate count	32772	32716	176478	17562

ACKNOWLEDGEMENT

The author would like to thank All India Council for Technical Education (AICTE), India to financially support this work under the grant 8023/BOR/RID/RPS-48/2009-10. The authors would also like to thank the Management and Principal of Sri Ramakrishna Engineering College, Coimbatore for providing excellent computing facilities and encouragement.

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, pp 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, [4] Z. Zhu and X. Zhang, "Access-mode predictions for low-power cache design," *IEEE Micro*, vol. 22, no. 2, pp. 58–71, Mar.–Apr. 2002.

[13] P. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systemson- Chip: Optimization and Exploration*. Dordrecht, The Netherlands: Kluwer, 1999.

[14] P. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 5, no. 3, pp. 682–704, Jul. 2000.

[15] Z. Zhu and X. Zhang, "Access-mode predictions for low-power cache design," *IEEE Micro*, vol. 22, no. 2, pp. 58–71, Mar.–Apr. 2002.

AUTHORS PROFILE

Sharon Mathew pursuing M.E (Master of Engineering) in VLSI Design from Sri Ramakrishna Engineering College. She has received her B.Tech in Electronics and Communication Engineering from M G University.

