Split Ways: An Efficient Replacement Policy for Larger Sized Cache Memory

Purnendu Das, Bishwa Ranjan Roy

***Abstract**: As the data requirement of today's application is increasing the size of the cache memory in a multicore processor is also increasing. A multicore processor has many levels of cache memory. The Last Level Cache (LLC) is normally shared by all the cores and its size must be large enough to handle today's data intensive applications. Such larger sized set-associative LLC facing major challenges to efficiently implement the replacement policy of its sets. As the number of ways are increasing in the LLC, the replacement policy becomes a bottleneck of the system. To improve the performance of the system and to reduce the hardware overhead, in this paper we propose a simple but hardware efficient replacement policy for the larger sized LLCs. We call the techniques as SplitWays as it divides (splits) the ways of a set into multiple groups called wayGroups. Each wayGroup maintains its own replacement policy. Experimental analysis using full-system simulator found that the proposed technique reduces the hardware overhead by up to 66% without any performance degradation..*

Keywords: **Computer Architecture, Cache Memories, Replacement Policies, Hardware Overheads.**

I. INTRODUCTION

Cache memory is one of the most important part of today's computing systems. Most of the modern multicore architectures have private L1 and L2 caches for each core and all the cores share a common large size L3 cache. The L3 cache is also called as the Last Level Cache (LLC) of the system. In the era of big data and machine learning the size of data is increasing rapidly. As a result, the multicore processors need larger sized LLC. The performance of LLC is very significant for the performance of the entire multicore system. Also it has been found that in case of LLC, the capacity is more important than the latency. Hence though a larger LLC may increase the latency of the cache but reducing the number misses is more important in LLC. The latency issue is handled by the upper level of the caches. Cache memories are traditionally designed with SRAM technologies. SRAM technology needs six transistors to store a single bit of information, hence consume significant chip area. It has been found that using SRAM for designing larger sized cache may consume 90% of the chip area which limits the number of cores in the system [1]. Researchers have proposed an alternative technology to use DRAM for designing the LLC. DRAM has 8 times better density over SRAM hence possible to design larger LLC in a small chip area [1].

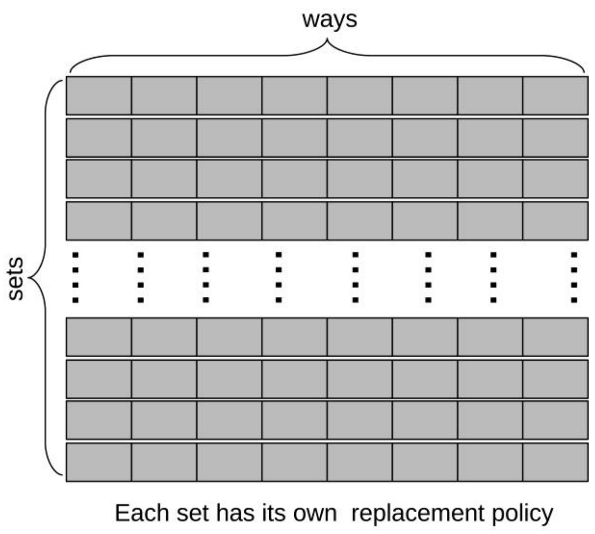

Fig. 1.An example of 8-way set associative cache.

Designing larger sized and highly associative LLC is the demand of today's data driven applications but managing such highly associative caches has some issues. One of the major issue on which this paper is based on is efficiently managing the replacement policy of the LLC. A set-associative cache maintains separate replacement policy for each set. Figure1 shows an example of set-associative cache. A core when requested for a block is first searched in the upper level of the cache memories. If the block is not found in the upper caches then the request is being forwarded to the LLC. As per the block-mapping policy of set-associative cache, a particular block can be placed in only once set. When the set is fully occupied but a newly incoming block has to be placed within the same sets then an existing block has to be removed from the cache to make room for the new block. The replaced block is called the victim block. A replacement policy is used to select the victim block [2]. For example in case of Least Recently Used (LRU) replacement policy, the victim block is the oldest accessed block in the set. Though the hardware overhead of implementing LRU based policy is less compared to the other replacement policies, the overhead of using it for highly associative caches cannot be ignored. In this work by hardware overhead we means the number of bits required to store the status of the replacement policy.

Revised Manuscript Received on October 30, 2019.
* Correspondence Author
Purnendu Das*, Department of Computer Science, Assam University Silchar, India. Email: purnen1982@gmail.com
Bishwa Ranjan Roy*, Department of Computer Science, Assam University Silchar, India. Email: brroy88@gmail.com

Section II discussed this in more detail. In this work we proposed an alternative replacement policy for highly associative caches. Our proposed technique divides the ways of each set into multiple groups and a separate replacement policy is being applied to each group. To replace a block from the cache, each group selects a possible victim and the final victim is selected from all the possible victim blocks.

II. BACKGROUND

Set-associative cache maintains replacement policy separately for each set. The replacement policy has three major parts [3]: (a) Insertion, (b) Eviction and (c) Promotion. Insertion means where to place a newly incoming block. e.g., in case of LRU, the insertion policy place a newly incoming block always in MRU position. Eviction means when a block from the set is need to be evicted then how to choose the victim block. Promotion means where to promote a block if after being re-accessed in the cache. LRU policy promoted such blocks into MRU position. Some of the well known replacement policies are: Least Recently Used (LRU), Moist Recently Used (MRU), First In First Out (FIFO), Least Frequently Used (LFU) and Random. These policies are well known and their operation details are not required to discuss here. However, the hardware overhead of implementing such policies is needed to be discussed. Each *n*-ways associative cache has *n* ways in each set. Hence to uniquely represent any way, log_2n bits are required. Hence for a single set, $n \times log_2n$ bits are required. For a 4-ways set-associative cache, each set has 4-ways and they can be uniquely represented as 00, 01, 10 and 11. Here 00 can be assumed as LRU and 11 can be assumed as MRU. A hardware circuit is required to maintain the insertion, promotion and eviction of any replacement policy. Most of the above mentioned replacement policies have same hardware overhead, except LFU and Random. LFU needs counter to store the frequency of each block to be used i.e. the number of accesses of each block. These policies are also called Access Based Replacement Policies. In case of such policies the size of the counter is varies design to design. Random has almost negligible hardware overhead but not very efficient in terms of performance. Some important extension of LRU based replacement policy are Dynamic Insertion Policy [4], PIPP [5] and [6], [7], [8], [9], [10] etc. DIP has dual insertion and promotion policy. It inserts blocks either in MRU or middle position based on some condition. The hardware overhead of DIP is same as LRU.

A. Dead Block

In a particular set of a cache, if a block is present but it never going to be reused by any core than the block is called a dead block in the cache. LRU policy takes longer time to remove a dead block from the cache as its take time for a block to become least recently used block. Such dead blocks unnecessarily waste the cache space. Detecting dead block in a cache is not possible as knowing the future is impossible. But fortunately there are many prediction mechanism proposed that can predict dead blocks with high accuracy [11]. Though the proposed dead block mechanisms are efficient but managing them needs to use some prediction tables, which leads significant hardware overhead [12]. Hence proposing a less expensive dead block detection remains an interesting research topic.

III. MOTIVATION

Though advanced replacement policies are already proposed [6], [13], [14], [15] which can give better performance, most of the cache memories still use LRU for its simplicity. The dead block prediction mechanism [11] discussed above is an LRU based policy with additional predictor. The major issue with LRU is to efficiently implement the dead block handling mechanism and the overhead while implementing in larger LLCs where the associativity may be up to 32 or 64. Higher the associativity higher the chances of a block being dead. Also implementing efficient dead block predictor in such highly associative cache has dual disadvantage: (a) highly associable cache already needs more bits to manage replacement policy and (b) the prediction table takes signification storage. Hence in this work we have been motivated to propose a simple but efficient replacement policy for larger sized LLCs.

IV. SPLITWAYS

The main motive of this work is to reduce the hardware overhead and dead blocks of the replacement policy used in highly associative LLCs. The disadvantage of LRU based replacement policies are already discussed in II. The hardware overhead of LRU is discussed in Section II. SplitWays is proposed to reduce the hardware overhead as well as the dead blocks from the highly associative LLCs. From now onwards, LLC means highly associative cache memory, designed with DRAM/eDRAM technology [1], [16].

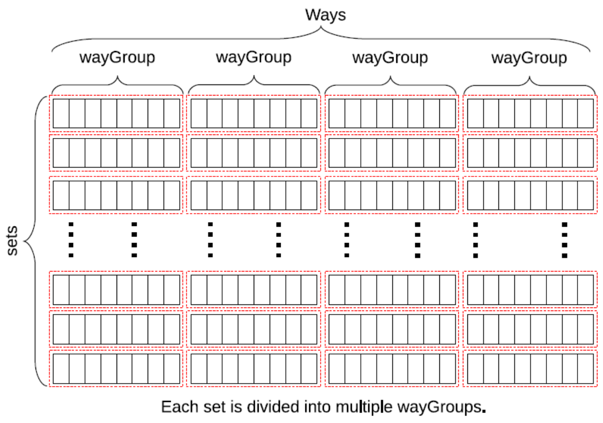

Fig. 2.An example of SplitWays, implemented in a 32-way set-associative cache. Each set is divided into 4 wayGroups.

A. Proposed Technique

Figure 2 shows a set-associative cache implementing Split-Ways. In this technique the ways of each set is split into multiple (say *m*) groups called wayGroups. For example, in the Figure 2, each set is split into 4 wayGroups having eight ways in each group. Each wayGroups has separate LRU based replacement policy. To evict a block from the set, the replacement policy of all the *m* wayGroups selects the victim block independently. Such independent operation gives *M* possible victim blocks as mentioned bellows.

$$S = \{b_1, b_2, b_3, \ldots, b_m\}$$

Out of all these possible victim blocks the final victim block will be chosen randomly. In figure 2, during the victim selection of a particular set *s*, all the 4 waysGroups of *s* give a possible victim block. The final victim block is decided randomly from the 4 possible victim blocks.

Figure 3 shows the complete flow diagram of the process used in SplitWays to select a victim block from a particular set.

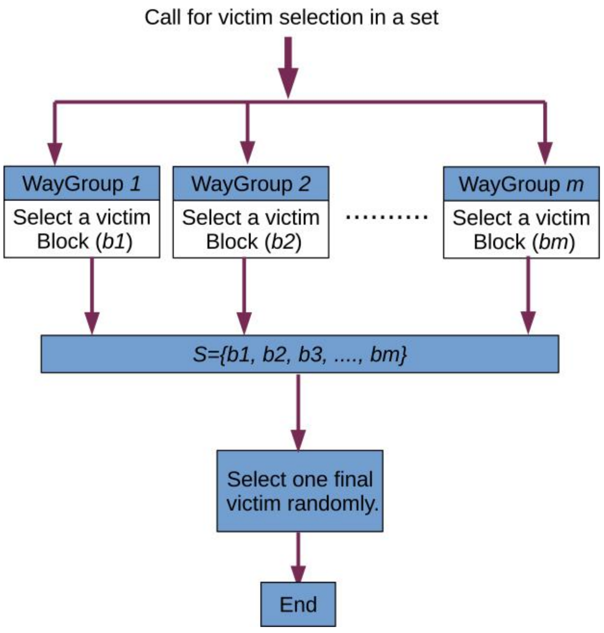
Call for victim selection in a set
WayGroup 1
Select a victim Block (b1)
WayGroup 2
Select a victim Block (b2)
WayGroup m
Select a victim Block (bm)
S={b1, b2, b3, ..., bm}
Select one final victim randomly.
End

Fig. 3.The flow diagram for SplitWays, explaining the process of selecting a victim block from a particular set.

B. Advantages of SplitWays

SplitWays has two major advantages: (a) It reduces the hardware overhead of replacement policy and (b) It also reduces the dead blocks from the highly associative sets.

1) *Hardware Overhead*: As discussed in Section II, an *n*-way set associative cache has to maintain $n \times log_2 n$ bits for each set for managing the LRU policy. Since SplitWays divides each set into m wayGroups, each wayGroup needs $m \times log_2 m$ bits and each sets needs $(n/m) \times (n \times log_2 n)$ bits, which is significantly less than the original design. The random selection for the final victim block from the *m* possible victim blocks has almost negligible hardware overhead. Table I shows the total bits saved by SplitWays as compared to the baseline design. The results are shown for different replacement policies. From the table it can be observed that SplitWays reduces hardware overhead of LRU by 17% to 66%, depending on the associativity and the number of wayGroups. The reduction is also same for the policies like DIP [4], BIP [4] and other similar techniques. The access based replacement policy needs more bits to store the information. SplitWays hardware overhead is significantly less than such replacement policies.

2) *Dead Block Removal*: The disadvantage of dead blocks is already discussed in Section II-A. A major advantage of SplitWays is that it can remove a dead block early from the cache. In a traditional LRU it takes time for a block to become dead. Hence waste unnecessary space of the cache. In SplitWays since each wayGroup maintains separate LRU mechanism, the chances of removing a dead block from the cache is high.

V. EXPERIMENTAL ANALYSIS

To implement this proposal we have used full-system simulator called gem5 [17]. It is a simulator that can simulate the whole computer system. We have used gem5 on System Emulation mode (SE mode). The simulated processor has 4 cores and all the core has its private L1 and L2 caches. A common large sized L3 cache as LLC is shared by all the cores. Installing gem5 on a system gives an environment to simulate any machine on it. All the replacement polices as well as the other cache related operations are maintained by a special module of gem5 called Ruby. The cache memory and replacement policy files need to be modified to implement our proposed technique. Seven Parsec benchmarks [18] are used for this experiment. the benchmarks are the application design by experts to measure the performance of a simulated system. Most of these benchmarks are multi threaded and need lots of data sharing among the threads. We installed gem5 for the alpha architecture with MESI CMP protocol. The Parsec executable for Alpha architecture is freely available to download from [18]. The benchmarks used are: *body, vips, dedup, swaption, x264, ferret* and *fluid*. All the benchmarks are initially executed for 2 million cycles for warm-up and then executed for 200 million cycles for the analysis. After executing each benchmark the statistics about the execution is stored in a file. Table II shows the parameters used for SplitWays for implementation. To compare the performance of SplitWays we have also implemented the LRU replacement policy. We call it here as Baseline-1. Note that in case of Baseline-1, all the parameters mentioned in Table II are same except the parameters related to SplitWays. In Baseline-1 each set has only one wayGroup.

A. Performance Comparison over Baseline-1

Define Figure 4 shows the comparison of SplitWays and Baseline-1 for Miss Per Kilo Instructions(MPKI). It can be observed that on average the MPKI of SplitWays reduces by 8.7% as compared to the Baseline-1. Such reduction is possible because of the reduction of dead blocks. Since SplitWays divides the sets into multiple wayGroups the chances of a dead block being removed is higher. The applications where SplitWays shows less improvements are the benchmarks where the block have longer reuse

Table- I: Comparison Of The Bits Required To Implement Original Lru Policy And The Proposed Splitways.

Associativity	wayGroups	Bits Required in Original LRU	Bits required in SplitWays	Reduction in SplitWays
16	2	64	48	25%
16	4	64	32	50%
32	2	160	128	20%
32	4	160	96	40%
32	8	160	64	60%
64	2	384	320	17%
64	4	384	256	33%
64	8	384	192	50%
64	16	384	128	66%

Table- II: specifications used for designing splitways.

Specification	Values
Number of cores	4
Level of cache memory	3
Private cache	L1 and L2
Shared cache	L3
L3 cache	8MB, 32-way set associative
L2 cache	256KB, 4-way set associative
L1 cache	64KB, 2-way set associative
Cache block size	64B
Number of wayGroups	4

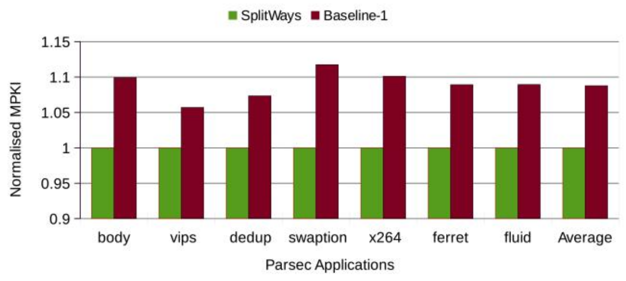

Fig. 4.Normalized comparison of SplitWays with Baseline-1 over MPKI.

interval. i.e., a block is re-accessed after longer time. SplitWays may remove such blocks early resulting a cache miss. But as observed from the figure such applications are very less. Reduction in MPKI also reduces the memory access time and hence reduces the Cycle Per Instruction (CPI) of the system. Figure 5 shows the comparison of SplitWays and Baseline-1 in terms of CPI. On average SplitWays reduces CPI by 4% more than Baseline-1. Though the performance improvement of SplitWays is not significant the main aim of this policy is to reduce the hardware overhead of replacement policy without degrading performance. Figure 6 shows that SplitWays has less hardware overhead compared to most of the existing efficient replacement policies. abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms

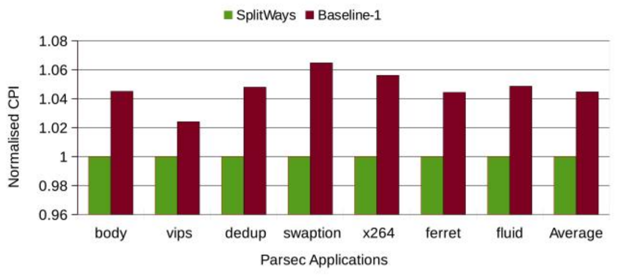

Fig. 5.Normalized comparison of SplitWays with Baseline-1 over CPI.

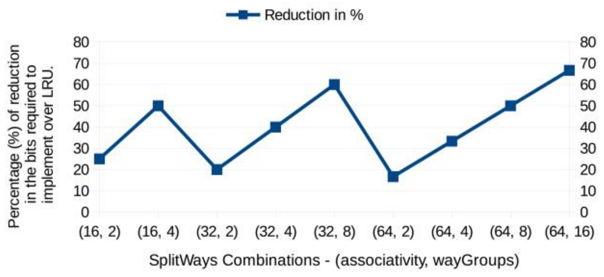

Fig. 6.The percentage of reduction in the bits required to implement SplitWays over LRU policy.

do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

B. Performance Comparison over Baseline-2

We have also compared SplitWays with an existing dead block prediction based replacement policy as defined in [11]. The comparison of MPKI between SplitWays and Baseline-2 is shown in Figure 7. It can be observed that Baseline-2 reduces MPKI 5.5% more as compared to SplitWays. The reason behind this is that the Baseline-2 has a highly efficient dead block prediction mechanism while SplitWays only divides the highly associative cache into multiple wayGroups. Figure 8 compares the two techniques in terms of CPI. The performance of SplitWays as compared to Baseline-2 is 3% less.

Through the performance of SplitWays is not as efficient as Baseline-2, but it improves the performance over Baseline-1. Note that Baseline-2 is a significantly better approach as compared to Baseline-1. The primary focus of SplitWays is not to reduce the MPKI but to reduce the hardware overhead of replacement policy. SplitWays has almost 400% less hardware overhead as compared to the Baseline-2. Note that the comparison is in terms of the bits required to implement the replacement policies.

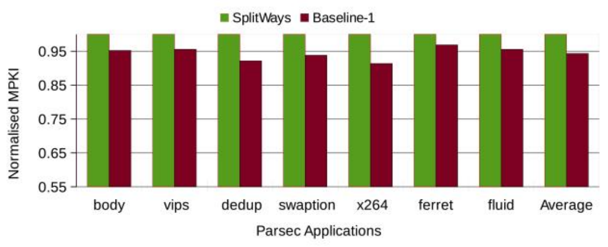

Fig. 7.Normalized comparison of SplitWays with Baseline-2 over MPKI.

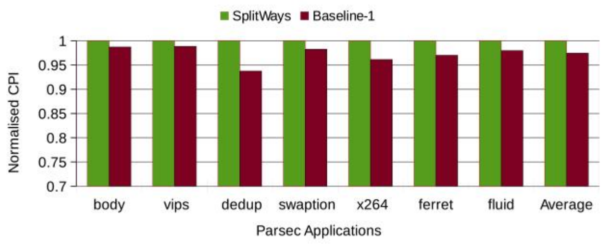

Fig. 8.Normalized comparison of SplitWays with Baseline-2 over CPI.

VI. CONCLUSION

Replacement policy are one of the most important part of todays highly associative Last Level Cache memories. The traditional LRU based policies are simple and efficient for smaller sized caches but for larger sized and higher associative caches these polices show significant hardware overhead. Also the replacement policy in such highly associative caches fails to remove the dead block early, hence unnecessarily use the cache space. In this paper we have proposed an hardware efficient replacement policy for such larger size LLCs. The proposed policy is reducing the hardware overhead by 66% without degradation of the performance. The proposed policy has divided the ways of a set into multiple groups to maintain separate replacement policy for each group. Hence the chances of the early removal of a dead block is higher than the existing LRU design.

REFERENCES

1. S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 6, pp. 1524–1537, June 2015.
2. L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, vol. 5, no. 2, pp. 78–101, 1966.
3. M. Kharbutli, M. Jarrah, and Y. Jararweh, "Scip: Selective cache insertion and bypassing to improve the performance of last-level caches," in 2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT), Dec 2013, pp. 1–6.
4. M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. S. Emer, "Adaptive insertion policies for high performance caching," in ISCA, 2007.
5. Y. Xie and G. H. Loh, "Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches," in ISCA, 2009.
6. F. Juan and L. Chengyan, "An improved multi-core shared cache replacement algorithm," in 2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science, Oct 2012, pp. 13–17.
7. K. Morales and B. K. Lee, "Fixed segmented lru cache replacement scheme with selective caching," in 2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC), Dec 2012, pp. 199–200.
8. A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak, "Cache replacement policies revisited: the case of p2p traffic," in IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004., April 2004, pp. 182–189.
9. W. A. Wong and J. . Baer, "Modified lru policies for improving secondlevel cache behavior," in Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550), Jan 2000, pp. 49–60.
10. Smith and Goodman, "Instruction cache replacement policies and organizations," IEEE Transactions on Computers, vol. C-34, no. 3, pp. 234–241, March 1985.
11. M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," IEEE Transactions on Computers, vol. 57, no. 4, pp. 433–447, April 2008.
12. K. J. Deris and A. Baniasadi, "Analysis of non-optimal lru decisions in high-performance processors," in 2008 International Conference on Microelectronics, Dec 2008, pp. 458–461.
13. L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Improving inclusive cache performance with two-level eviction priority," in 2012 IEEE 30th International Conference on Computer Design (ICCD), Sep. 2012, pp. 387–392.
14. C. Zhang and B. Xue, "A tag-based cache replacement," in 2010 IEEE International Conference on Computer Design, Oct 2010, pp. 92–97.
15. K. Chikhale and U. Shrawankar, "Hybrid multi-level cache management policy," in 2014 Fourth International Conference on Communication Systems and Network Technologies, April 2014, pp. 1119–1123.
16. W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading," in Proceedings of the 38th Annual International Symposium on Computer Architecture, ser. ISCA '11, 2011, pp. 247–258.
17. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti et al., "The gem5 simulator," ACM SIGARCH Computer Architecture News, vol. 39, no. 2, pp. 1–7, 2011.
18. C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011. [Online]. Available: http://parsec.cs.princeton.edu/

AUTHORS PROFILE

Dr. Purnendu Das is an assistant professor of the Department of Computer Science, Assam University Silchar. He has pursued Ph.D. Degree from Tripura University. He has published research papers in many reputed journals.

Bishwa Ranjan Roy is an assistant professor of the Department of Computer Science, Assam University Silchar. He has done M.Tech degree at NIT Silchar. He has many publication in many reputed journals.

.