ISSUES IN CACHING TECHNIQUES TO IMPROVE SYSTEM PERFORMANCE IN CHIP MULTIPROCESSORS

H. R. Deshmukh[1], G. R. Bamnote[2]
[1]Associate professor, B.N.C.O.E., Pusad, M.S., India
[2]Associate professor & Head, PRMIT&R, Badnera, M.S., India

ABSTRACT

As cache management in chip multiprocessors has become more critical because of the diverse workloads, increasing working sets of many emerging applications, increasing memory latency and decreasing size of cache devoted to each core due to increased number of cores on a single chip in Chip multiprocessors (CMPs). This paper identifies caching techniques and important issues in caching techniques in chip multiprocessor for managing last level cache to reduce off chip access to improve the system performance under critical conditions and suggests some future directions to address the identified issues.

KEYWORDS: Multiprocessors, Partitioning, Compression, Fairness, QoS.

I. INTRODUCTION

Over the past two decades, speed of processors has increased at much faster rate than DRAM speeds. As a result, the number of processor cycles it takes to access main memory has also increased. Current high performance processors have memory access latency of well over more than hundreds of cycle, and trends indicate that this number will only increase in the future. The growing disparity between processor speed and memory speed is popularly referred in the architecture community as the *Memory Wall* [1]. Main memory accesses affect processor performance adversely. Therefore, current processors use caches to reduce the number of memory accesses. A cache hit provides fast access to recently accessed data. However, if there is a cache miss at the last level cache, a memory access is initiated and the processor is stalled for hundreds of cycles [1]. So as, to sustain high performance, it is important to reduce cache misses.

The importance of cache management has become even more critical because of, diverse workloads, increasing working sets of many emerging applications, increasing memory latency and decreasing size of cache devoted to each core due to increased number of cores on a single chip.

Improvements in silicon process technology have facilitated the integration of multiple cores into modern processors and it is anticipated that the number of cores on a single chip will continue to increase in chip multiprocessors in future. Multiple application workloads are attractive for utilising multi-core processors, put significant pressure on the memory system [2]. This motivates the need for more efficient use of the cache in order to minimize the expensive, in terms of both latency and, requests to off-chip memory. This paper discusses the exiting approaches and limitations of exiting approaches in caching techniques in chip multiprocessors available in literature and investigates the important issues in this area.

II. REPLACEMENT TECHNIQUE

Different workloads and program phases have diverse access patterns like, Sequential access pattern in which all block are accesses one after another and never re-accessed, such as file scanning. Looping-like access patterns in which all blocks are accessed repeatedly with a regular interval, Temporally-clustered access patterns in which blocks accessed more recently are the ones more likely to be accessed in the near future and Probabilistic access patterns in which, each block has a

stationary reference probability, and all blocks are accessed independently with the associated probabilities.

Previous researchers [3]-[11] have shown that one replacement policy usually performs efficiently under the workload with one kind of access pattern; it may perform badly once the access pattern of the workload changes. For example, MRU replacement policy (Most Recently Used) performs well to sequential and looping patterns, LRU replacement policy performs well to temporally clustered patterns, while LFU replacement policy performs well to probabilistic patterns. From the study of existing replacement policies, it is found that none of the single cache replacement policy performs efficiently for mix type of access pattern like Sequential references, Looping references, Temporally-clustered references and Probabilistic references, which may be occurs simultaneously in one workload during execution. Some of the policies require additional data structures to hold the information of non-residential pages. Some policies require data update in every memory access, which necessarily increases memory and time overhead, in result degrade the performance.

Kaveh Samiee et al. (2009, 2008) [3][4] suggested weighted replacement policy. The basic idea of this policy is to rank pages based on their recency, frequency and reference rate. So, pages that are more recent and have used frequently are ranked higher. It means that the probability of using pages with small reference rate is more than the one with bigger reference rate. This policy behaves like both LRU and LFU by replacing pages, that were not recently used and pages that are used only once. WRP needs three elements to work and will add space overhead to system. Algorithm needs a space for recency counter L_i, frequency counter F_i, and for weight value W_i, which is as weighting value for each object in the buffer. Calculating weighting function value for each object after every access to cache will cause a time overhead to system. This policy fails for sequential access and loop access patterns.

Dr Mansard Jargh et al. (2004) [5] describes improved replacement policy (IRP) which perform some key modifications to the LRU algorithm and combine it with a significantly enhanced version of the LFU algorithm and take spatial locality into account in the replacement decision. IRP also uses the concept of spatial locality and therefore efficiently expels only blocks, which are not likely to be, accessed again. This algorithm-required memory overhead to store recency count 'rc', frequency count 'fc' and block address 'ba' for each block. Algorithm required time and processor overhead to search smallest 'fc' value and largest 'rc' value, as well as time and processor overhead to changing value of 'fc' and 'rc' to every access to block. Algorithm does not perform well for loop access pattern and sequential access pattern.

Jiang et al. (2002) [6] presented low inter-reference recency set policy (LIRS). Its objective is to minimize the deficiencies presented by LRU using an additional criterion named IRR (Inter-Reference Recency) that represents the number of different pages accessed between the last two consecutive accesses to the same page. The algorithm assumes the existence of some behaviour inertia and, according to the collected IRR's, replaces the page that will take more time to be referenced again. This means that LIRS does not replace the page that has not been referenced for the longest time, but it uses the access recency information to predict which pages have more probability to be accessed in a near future. The LIRS divides the cache into two sets, high inter-reference recency (HIR) block set and low inter-reference recency (LIR) block set. Each block with history information has a status either LIR or HIR. Cache is divided into a major part and a minor part in terms of size. Major part is used to store LIR blocks, and the minor part is used to store HIR blocks. A HIR block is replaced when the cache is full for replacement, and the LIRS stack may grow arbitrarily large, and hence, it needs to be required large memory overhead. This policy does not perform well for sequential access pattern.

Zhan-Sheng et al. (2008) [7] proposed CRFP Policy. It is and novel adaptive replacement policy, which combines LRU and LFU policies. CRFP propose a novel adaptive replacement policy that combined the LRU and LFU Policies (CRFP), CRFP is self-tuning and can switch between different cache replacement policies adaptively and dynamically in response to the access pattern changes. Memory overhead is required to store the cache directory, recency value, and frequency value, hit value, miss value, switches time and switch ration. Policy also required time overhead to search cache directory, and computational time to switch from LRU to LFU. However, this policy fails in the case, where accesses inside loops with working set size slightly larger than the available memory.

E. J. O'Neil et al. (1993) [8] presented LRU-K policy which makes its replacement decision based on the time of the K^{th} to last reference to the block i.e. reference density observed during the past K-reference. When K is larger, it can discriminate well between frequently and infrequently referenced blocks. When K is small, it can remove cold block quickly since such block would have wide span between the current time and K^{th} to last reference time. Time complexity of algorithm is O (log (n));

however this policy does not perform well for loop access pattern, and sequential access pattern.
Zhuxu Dong (2009) [9] proposed spatial locality based, block correlations directed cache replacement policy (BCD), which uses both of history and runtime access information to predict spatial locality, prediction results are use to improve the utilization of the cache and reduces the penalty incurred by incorrect predications. For most of real system workloads, BCD can reduce the cache miss ratio by 11% to 38% compared with LRU.
Y. Smaragadaki et al. (1999) [10] described early eviction LRU policy (EELRU) which was proposed as an attempt to mix LRU and MRU, based only on the positions on the LRU queue that concentrate most of the memory references. This queue is only a representation of the main memory using the LRU model, ordered by the recency of each page. EELRU detects potential sequential access patterns analyzing the reuse of pages. One important feature of this policy is the detection of non-numerically adjacent sequential memory access patterns. This policy does not perform well for loop access pattern.
Andhi Janapsatya et al. (2010) [11] proposed a new adaptive cache replacement policy, called Dueling CLOCK (DC). The DC policy developed to have low overhead cost, to capture recency information in memory accesses, to exploit the frequency pattern of memory accesses and to be scan resistant. Paper proposed a hardware implementation of the CLOCK algorithm for use within an on-chip cache controller to ensure low overhead cost. DC policy, which is an adaptive replacement policy, that alternates between the CLOCK algorithm and the scan resistant version of the CLOCK algorithm. This policy reduced maintenance cost of LRU policy. Research issue here is to explore how replacement policy will perform efficiently under diverse workload (mix access pattern) and how processor and memory overhead will be, reduce for novel replacement policy.

III. PARTITIONING TECHNIQUE

Chip multiprocessors (CMPs) have been widely adopted and commercially available as the building blocks for future computer systems. It contains multiple cores, which enables to concurrently execute multiple applications (or threads) on a single chip. As the number of cores on a chip increases, the pressure on the memory system to sustain the memory requirements of all the concurrently executing applications (or threads) increases. An important question in CMP design is how to use the limited area resources on chip to achieve the best possible system throughput for a wide range of applications. Keys to obtaining high performance from multicore architectures is to provide fast data accesses (reduce latency) for on-chip computation resources and manage the largest level on-chip cache efficiently so that off-chip accesses are reduced. While limited off-chip bandwidth, increasing latency, destructive inter-thread interference, uncontrolled contention and sharing, increasing pollution, decreasing harmonic mean and diverse workload characteristics pose key design challenges. To address these challenges many researchers [12]-[24] have proposes different cache partitioning scheme to share on-chip cache resources among different threads, but all challenges are not address properly.
Cho and Jin et al. (2006) [12], proposed software-based mechanism for L2 cache partitioning based on physical page allocation. However, the major focus of their work is on how to distribute data in a Non-Uniform Cache Architecture (NUCA) to minimize overall data access latencies. However, they do not concentrate on the problem of uncontrolled contention on a shared L2 cache.
David Tam et al. (2007) [13], demonstrated a software-based cache partitioning mechanism and shown some of the potential gains in a multiprogrammed computing environment, which allows for flexible management of the shared L2 cache resource. This work neither supports the dynamic determination of optimal partitions nor dynamically adjusts the number of partitions.
Stone et al. (1992) [14] investigated optimal (static) partitioning of cache resources between multiple applications, when the information about change in misses for varying cache size is available for each

of the competing applications. However, such information is non-trivial to obtain dynamically for all applications, as it is dependent on the input set of the application.

Suh et al. (2004) [15] described dynamic partitioning of shared cache to measure utility for each application by counting the hits to the recency position in the cache and used way partitioning to enforce partitioning decisions. The problem with way partitioning is that it requires core-identifying bits with each cache entry, which requires changing the structure of the tag-store entry. Way partitioning also requires that the associativity of the cache be increased to partition the cache among a large number of applications.

Qureshi et al. (2006) [16] proposed the cache monitoring circuits outside the cache so that the information computed by one application is not polluted by other concurrently executing applications. They provide a set sampling based utility monitoring circuit that requires storage overhead of 2KB per core and used way partitioning to enforce partitioning decisions. TADIP-F is better able to respond to workloads that have working sets greater than the cache size while UCP does not.

Chang et al. (2007) [17] used time slicing as a means of doing cache partitioning so that each application is guaranteed cache resources for a certain time quantum. Their scheme is still susceptible to thrashing when the working set of the application is greater than the cache size.

Suh et al. (2002) [18] described a way of partitioning a cache for multithreaded systems by estimating the best partition sizes. They counted the hits in the LRU position of the cache to predict the number of extra misses that would occur if the cache size were decreased. A heuristic used this number combined with the number of hits in the second LRU position to estimate the number of cache misses that are avoided if the cache size is increased.

Dybdahl et al.,(2006) [19] presented the method which adjust the size of the cache partitions within a shared cache, work did not consider a shared partition with variable size, nor did they look at combining private and shared caches.

Kim et al. (2004) [20] presented cache partitioning in shared cache for a two-core CMP where a trial and fail algorithm was applied. Trial and fail as a partitioning method does not scale well with increasing number of cores since the solution space grows fast.

Z. Chishti et al. (2005) [21] described spilling evicted cache blocks to a neighbouring cache. They did not consider putting constraints on the sharing or methods for protection from pollution. No mechanism was described for optimizing partition sizes.

Chiou et al.(2000) [22] suggested a mechanism for protecting cache blocks within a set. Their proposal was to control which blocks that can be replaced in a set by software, in order to reduce conflicts and pollution. The scheme was intended for a multi-threaded core with a single cache.

Dybdahl et al.(2007) [23] presented a approach in which the amount of cache space that can be shared among the cores is controlled dynamically, as well as uncontrolled sharing of resources is also control effectively . The adaptive scheme estimates, continuously, the effect of increasing/ decreasing the shared partition size on the overall performance. Paper describes NUCA organization in which blocks in a local partition can spill over to neighbour core partitions. Approach suffers from pollution and harmonic mean problem.

Dimitris Kaseridis et al. (2009) [24] proposed a dynamic partitioning strategy based on realistic last level cache designs of CMP processors. Proposed scheme provides on average a 70% reduction in misses compared to non-partitioned shared caches, and a 25% misses reduction compared to static equally partitioned (private) caches. This work highlights the problem of sharing the last level of cache in CMP systems and motivates the need for low overhead, workload feedback-based hardware/software mechanisms that can scale with the number of cores, for monitoring and controlling the L2 cache capacity partitioning.

Research issue here is to explore cost effective solution for future improvements in caching requirement, including thrashing avoidance, throughput improvement, fairness improvement and QoS guarantee under above key design challenges.

IV. COMPRESSION TECHNIQUE

Chip multiprocessors (CMPs) combine multiple processors on a single die, however, the increasing number of processor cores on a single chip increases the demand of two critical resources, the shared L2 cache capacity and the off-chip pin bandwidth. Demand of critical resources are satisfied by the

technique of cache compression. From the existing research work [25][26][27][28][29][30][31] it is well known that Compression technique, which can both reduce cache miss ratio by increasing the effective shared cache capacity, and improve the off-chip bandwidth by transferring data in compressed form. Jang-Soo Lee et al., (1999) [25] proposed the selective compressed memory system based on the selective compression technique, fixed space allocation method, and several techniques for reducing the decompression overhead. The proposed system provide on the average 35% decrease in the on-chip cache miss ratio as well as on the average 53% decrease in the data traffic. However, authors could not control the problem of long DRAM latency and limited bus bandwidth.

Charles Lefurgy et al (2002)[26] presented a method of decompressing programs using software. It relies on using a software managed instruction cache under control of the decompressor. This is achieved by employing a simple cache management instruction that allows explicit writing into a cache line. It also considers selective compression (determining which procedures in a program should be compressed) and show that selection based on cache miss profiles can substantially outperform the usual execution time based profiles for some benchmarks. This technique achieves high performance in part through the addition of a simple cache management instruction that writes decompressed code directly into an instruction cache line. This study focuses on designing a fast decompressor (rather than generating the smallest code size) in the interest of performance. Paper shown that a simple highly optimized dictionary compression perform even better than CodePack, but at a cost of 5 to 25% in the compression ratio

Prateek Pujara et al. (2005) [27] investigated *restrictive compression* techniques for level one data cache, to avoid an increase in the cache access latency. The basic technique all words narrow (*AWN*) compresses a cache block only if all the words in the cache block are of narrow size. *AWN* technique here stores a few upper halfwords (*AHS*) in a cache block to accommodate a small number of normal-sized words in the cache block. Further, author not only make the *AHS* technique adaptive, where the additional half-words space is adaptively allocated to the various cache blocks but also propose techniques to reduce the increase in the tag space that is inevitable with compression techniques. Overall, the techniques in this paper increase the average L1 data cache capacity (in terms of the average number of valid cache blocks per cycle) by about 50%, compared to the conventional cache, with no or minimal impact on the cache access time. In addition, the techniques have the potential of reducing the average L1 data cache miss rate by about 23%.

Martin et al. (2008) [28] shown that it is possible to use larger block sizes without increasing the off-chip memory bandwidth by applying compression techniques to cache/memory block transfers. Since bandwidth is reduced up to a factor of three, work proposes to use larger blocks. While compression/decompression ends up on the critical memory access path, works find its negative impact on the memory access latency time. Proposed scheme dynamically chosen a larger cache block when advantageous given the spatial locality in combination with compression. This combined scheme consistently improves performance on average by 19%.

Xi Chen et al. (2009) [29] presented a lossless compression algorithm that has been designed for fast on-line data compression, and cache compression in particular. The algorithm has a number of novel features tailored for this application, including combining pairs of compressed lines into one cache line and allowing parallel compression of multiple words while using a single dictionary and without degradation in compression ratio. The algorithm is based on pattern matching and partial dictionary coding. Its hardware implementation permits parallel compression of multiple words without degradation of dictionary match probability. The proposed algorithm yields an effective system-wide compression ratio of 61%, and permits a hardware implementation with a maximum decompression latency of 6.67 ns.

Martin et al. (2009) [30] presents and evaluates FPC, a lossless, single pass, linear-time compression algorithm. FPC targets streams of double-precision floating-point values. It uses two context-based predictors to sequentially predict each value in the stream. FPC delivers a good average compression ratio on hard-to-compress numeric data. Moreover, it employs a simple algorithm that is very fast and easy to implement with integer operations. Author claimed that FPC to compress and decompress 2 to 300 times faster than the special-purpose floating-point compressors. FPC delivers the highest geometric-mean compression ratio and the highest throughput on hard-to compress scientific data sets. It achieves individual compression ratios between 1.02 and 15.05.

David Chen et al. (2003)[31] propose a scheme that dynamically partitions the cache into sections of different compressibilities, in this work it is applied repeatedly on smaller cache-line sized blocks so as to preserve the random access requirement of a cache. When a cache-line brought into the L2 cache or the cache-line is to be modified, the line is compressed using a dynamic, LZW dictionary. Depending on the compression, it is placed into the relevant partition. The partitioning is dynamic in that the ratio of space allocated to compressed and uncompressed varies depending on the actual performance, a compressed L2 cache show an 80% reduction in L2 miss-rate when compared to using an uncompressed L2 cache of the same area.

Research issues here is, when the processor requests a word within a compressed data block stored in the compressed cache, the compressed block has to be all decompressed on the fly and then the requested word is transferred to the processor. Compression ratio, compression time and decompression overhead, causes a critical effect on the memory access time and offsets the compression benefits, these issues are interesting and challenging for future research. Another issue associated with the compressed memory system is that, compressed blocks can be generated with different sizes depending on the compression efficiency. Therefore, in worst case, the length of any compressed block can be rather longer than that of its source block, this will adversely affect the performance of system.

V. CONCLUSION

From the above discussion following conclusion can be arrived to address the above research issues in caching techniques in chip multiprocessors to improve system performance

- To develop low overhead novel replacement policy, which will performs efficiently under under diverse workload, different cache size and varying working set.
- To develop efficient caching partitioning scheme in Chip Multiprocessors with different optimization objectives, including throughput, fairness, and guaranteed quality of service (QoS)
- To develop low overhead caching compression/decompression scheme in Chip Multiprocessors to increase shared cache capacity and off chip Bandwidth.

REFERENCES

[1] John L. Henneaay and David A. Patterson, “Computer Architecture a Quantitative Approach”, 3rd Edition ,Elsevier publication, 2003.

[2] Konstantinos Nikas, Matthew Horsnell, Jim Garside, “An Adaptive Bloom Filter Cache Partitioning Scheme for Multicore Architectures”, International Conference on, Embedded Computer Systems: Architectures, Modelling, and Simulation, July 21-24 2008, SAMOS 2008, pp. 21-24.

[3] Kaveh Samiee, GholamAli Rezai Rad, “WRP: Weighting Replacement Policy to Improve Cache Performance”, Proceeding of the International Symposium on Computer Science and its Applications, 2008, pp. 38-41.

[4] Kaveh Samiee “A Replacement Algorithm Based on Weighting and Ranking Cache”, International Journal of Hybrid Information Technology Volume Number 2 , April, 2009

[5] Dr Mansard Jargh, Ahmed Hasswa, “Implementation Analysis and Performance Evolution of the IRP-Cache Replacement Policy”, IEEE, 4th International Conference on Computer and Information Technology Workshops, 2004.

[6] S. Jiang and X. Zhang, “LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance”, Proceedings of the ACM SIGMETRICS Conference on Measurement and Modelling of computer Systems, pp. 31–42, 2002.

[7] Zhan-sheng, Da-wei, Hui-juan1, “CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies”, IEEE 8th International Conference on Computer and Information Technology Workshops 2008.

[8] E. J. Neil, P. E. Neil, and Gerhard Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering”, Proceedings of the 1993 ACM SIGMOD Conference, pp. 297–306, 1993.

[9] Zhu Xu-Dong, Ke Jian, Xu Lu, “BCD: To Achieve the Theoretical Optimum of Spatial Locality Based Cache Replacement Algorithm”, IEEE International Conference on Networking, Architecture, and

Storage, 2009

[10] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement", Proceedings of ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems, 1999.

[11] Andhi Janapsatya, Aleksandar Ignjatovic, Jorgen Peddersen and Parameswaran, "Dueling CLOCK: Adaptive Cache Replacement Policy Based on The CLOCK Algorithm"

[12] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches through OS-level Page Allocation", Proceedings of the Workshop on Memory System Performance and Correctness, 2006.

[13] David Tam, Reza Azimi, Livio Soares, and Michael Stumm, "Managing Shared L2 Caches on Multicore Systems in Software", Workshop on the Interaction between Operating Systems and Computer Architecture, 2007.

[14] H. S. Stone, J. Turek, and J. L. Wolf., "Optimal Partitioning of Cache Memory" IEEE Transactions on Computers, 41(9):1054–1068, 1992.

[15] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory" Journal of Supercomputing, 28(1):7–26, 2004.

[16] M. K. Qureshi and Y. Patt, "Utility Based Cache Partitioning: A Low Overhead High-Performance Runtime Mechanism to Partition Shared Caches", The 39^{th}Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'06

[17] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors", Proceeding of 22^{nd}Annual International Conference on Supercomputing, ICS-21, 2007.

[18] G. Suh, S. Devadas, and L. Rudolph, "Dynamic Cache Partitioning for Simultaneous Multithreading Systems", International Conference On Parallel and Distributed Computing Systems, 2002.

[19] H. Dybdahl, P. Stenstrom, and L. Natvig "A Cache Partitioning Aware Replacement Policy for Chip Multiprocessors", In 13^{th} International Conference High Performance Computing, HiPC, 2006.

[20] C. Kim, D. Burger, and S. W. Keckler, "Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches", IEEE Micro 2004, 23(6): 99-107,

[21] Z.Chishti, M.D.Powell, and T. N. Vijaykumar, "Optimizing Replication Communication and Capacity Allocation in CMPs", 32^{nd} Annual International Symposium on Computer Architecture, ISCA, 2005, pp: 357-368.

[22] D.Chiou, P.Jain, S. Devadas, and L. Rudolph, "Dynamic Cache Partitioning via Columnisation", Proceedings of the 37^{th} Conference on Design Automation, Los Angeles, June 5-9, 2000, ACM, 2000.

[23] Haakon Dybda, Perstenstrom, "An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors", IEEE 13^{th} International Symposium on High Performance Computer Architecture, 2007, pp: 2 – 12.

[24] Dimitris Kaseridis, Jeffrey Stuechelix and Lizy K. John, "Bank-aware Dynamic Cache Partitioning for Multicore Architectures 38^{th} International Conference on Parallel Processing 2009

[25] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim, "Design and Evaluation of a Selective Compressed Memory System", International Conference On Computer Design (ICCD), 1999, pp: 184-191.

[26] CHARLES LEFURGY, EVA PICCININNI, AND TREVOR MUDGE, "REDUCING CODE SIZE WITH RUN-TIME DECOMPRESSION", PROCEEDINGS ON 6^{th} INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE HPCA, 2002, PP. 218-228.

[27] Prateek Pujara, Aneesh Aggarwal, "Restrictive Compression Techniques to Increase Level Cache Capacity", IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD 2005, PP: 327-333.

[28] Martin Thuresson and Per Stenstrom, "Accommodation of the Bandwidth of Large Cache Blocks using Cache/Memory Link Compression", 37^{th} International Conference on Parallel Processing, ICCP 2008, PP: 478-486.

[29] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas, "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm", IEEE Transaction on Very large Scale Integration System 2009, 44(99), PP: 1-11.

[30] Martin, Burtscher and Paruj Ratanaworabhan, "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data" IEEE Transaction on Computers, vol. 58(1), January 2009, PP: 18-31.
[31] David Chen, Enoch Pegerico and Larry Rudolpha, "A Dynamically Partitionable Compressed Cache", Proceeding of Singapore-MIT Alliance Symposium, 2003.

Authors

H. R. Deshmukh received his M.E. CSE degree from SGB Amravati University, Amravati in 2008, and research scholar from 2009. Working as associate professor in deptt. Of CSE B.N.C.O.E., Pusad (India), & life member of Indian Society for Technical Education New Delhi.

G. R. Bamnote is Professor & Head of Department. Of Computer Science & Engineering at Prof. Ram Meghe Institute of Technology & Research, Badnera – Amravati. He did his BE (Computer Engg) in 1990 from Walchand College of Engineering, Sangli, M.E. (Computer Science & Engg) from PRMIT&R, Badnera-Amravati in 1998 and Ph.D. in Computer Science & Engineering from SGB Amravati University, Amravati in 2009. He is life member of Indian Society of Technical Education, Computer Society of India, and Fellow of The Institution of Electronics and Telecommunication Engineers, The Institution of Engineers (India).