International Journal of Computer Science and Network (IJCSN)
Volume 1, Issue 3, June 2012 www.ijcsn.org ISSN 2277-5420

Dynamic Cache Compression Technique in Chip Multiprocessors.

[1]Prof. H. R. Deshmukh,[2]Dr. G. R. Bamnote

[1]Associate professor, B.N.C.O.E.
Pusad (M.S.),India

[2]Professor & Head, PRMIT&R
Badnera (M.S.), India

Abstract

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. 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. However, decompressing cache lines also increases cache access latency, potentially degrading performance.to minimized performance degradation occurs due to decompression in this paper, we develop an dynamic compression policy which dynamically adapts the cost and benefit of cache compression, and block will be compressed only when compression is beneficial, otherwise block will be uncompressed.

We use two-level cache hierarchywhere the Level1 cache holds uncompressed data and the Level2cache dynamically selects between compressed anduncompressed storage.The effective size of an Level2 cache can be increased by using a LZW dictionary-based compression scheme. Variable-sized compressed blocks tend to increase the design complexity of the compressed cache architecture. This paper suggests technique to reduce the decompression overhead and to manage the variable sized compressed blocks efficiently.We evaluate dynamic cache compression usingsimulation and different workloads. Weshow that compression can improve performance formemory-intensive commercial workloads. However, always using compression hurts performancefor low-miss-rate workloads due to unnecessarydecompression overhead. By dynamically monitoring workload behavior,the dynamic compression policy achieves comparable benefitsfrom compression, while never degrading performance for memory intensive workloads. Here also we compare performance of dynamic cache compression under Least Recently Used (LRU) replacement policy as well as Efficient Replacement Policy (ERP). Performance of dynamic compression scheme is observed under both the replacement policies and it is find out that for most of the cases efficient replacement policy work well in dynamic cache compression technique.

KEYWORD: *Level1, Level2, LZW, LRU, ERP.*

1. Introduction

As the processor-memory performance gap increases every year, long memory access latencies are becoming the primary obstacle to improve the performance of computer systems. Modern processor designers have attempted to reduce the processor-memory performance gap by using a large space of on-chip cache memory. However, simply increasing the amount of on-chip cache space results in a large die area and high fabrication cost, and using the latency-tolerance techniques has introduced the limited pin bandwidth as another bottleneck for improvingsystem performance. These two issues, i.e., growingmemory access latencies and limited off-chip bandwidth should address more seriously. One interesting technique to address the latency and thebandwidth issues is to compress the cache blocks fetchedfrom memory and the blocks that must be written back.Compression can reduce the latency by simply requiringfewer cache/memory interconnect cycles to transfer a blockand this naturally also translates into less bandwidth consumed, which can reduce not only cache miss ratio butalso miss penalty. However, compression increases the cache hit time, since the decompression overhead lies on the critical access path. Decompression time causes a criticaleffect on the memory access time and variable-sizedcompressed blocks tend to increase the design complexityof the compressed cache architecture.Depending upon the balance between hits and misses, cache compression has the potential to either greatly help or greatly hurt performance.

This paper suggeststechnique to reduce the decompression overhead and to manage the compressed blocks efficiently.In this paper, we develop a dynamic cache compression scheme to dynamically optimize on-chip cache performance. Our design has two major parts.First, we use a two-level cache hierarchy where the Level cache holds uncompressed data and the Level2 cache dynamically selects between compressed and uncompressed storage. We use dictionary based LZW compression algorithm, to compress Level2 lines. The Level2 cache is fully associative cache with LRU replacement as well as ERP replacement, where Level2 cache canstore up to N compressed lines but has space for only N/2 uncompressed lines. Thus compressions can potentiallydouble the effective capacity of the cache.

Second, our dynamic compression policy keeps a track whether compression would help, hurt, or make no difference to a given reference. The key insight is that thecompressed size determineswhether a given reference hits because of compression,would have missed without compression, or would havehit or missed regardless. The controller updates a single,Compression saturating counter on each reference, incrementing by the Level2 miss penalty when compression couldhave or did eliminate a miss and decrementing it by thedecompression latency when a reference would have hitregardless. The controller uses the predictor when theLevel2 allocates a line, storing the line uncompressed if thecounter is negative and compressed otherwise. This paper shows that always compressing Level2 cache lines increases the effective cache capacity for few workloads, which in turn reducesLevel2 miss ratios and overall run-time as. However, theincreased Level2 access latency (due to decompressionoverhead), degrades performance for workloadswith low Level2 miss rates. Wepropose a dynamic cache compression policy that dynamicallybalances the benefit of compression (i.e., miss ratioreduction) with the cost (i.e., increased Level2 accesslatency). We present simulation results, showingthat dynamic cache compression can improve performancefor memory-intensive workloadswhile never degrading the performance.

2. Related Works

Jang-Soo Lee et al.,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 *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 partthrough the addition of a simple cache management instruction that writes decompressed code directly into aninstruction cache line. This study focuses on designing a fast decompressor (rather than generatingthe smallest code size) in the interest of performance. Paper shown that a simple highly optimized dictionary compressionperform even better than CodePack, but at a cost of 5 to 25% in the compression ratio

Prateek Pujara et al 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. 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, work 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) 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 dictionarymatch probability. The proposed algorithm yields an effective system-wide compression ratio of 61%, and permits a hardware implementation with a maximum decompression latencyof 6.67 ns.

Martin et al. [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. [31] propose a scheme that dynamically partitions the cache into sections of different compressibility, 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.

3. Management of Dynamic Compressed Cache

We propose two level cache hierarchy using cache compressions to increase effective cache size, reduce off-chip misses and pin bandwidth demand, and ultimately improve system performance. Our proposed compressed cache design for a multiprocessor system uniprocessor system is shown in Figure 1.

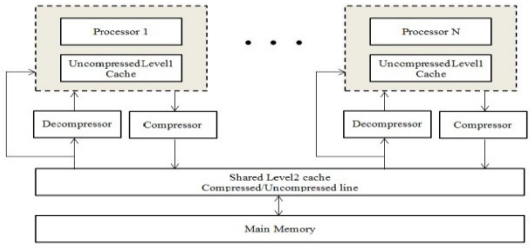

Figure 1: Cache Architecture

We propose storing cache lines in a compressed format in the level2 cache while leaving the Level1 cache uncompressed. The benefits of this technique for many workloads are, storing compressed cache lines can increase the effective cache size, potentially decreasing Level2 miss rates and achieving better overall performance. In addition, transferring compressed data between the Level2 cache and memory decreases the demand on pin bandwidth. Unfortunately, cache compression also has a negative side effect, since L2 cache lines have to be decompressed before moving to the Level1 cache or being used by the processor. This means that storing compressed lines in the Level2 cache increases the Level2 hit latency. While achieving a high compression ratio is important to increase the effective cache size, any cache compression algorithm should also have a small impact on Level2 hit latency so as not to hamper performance in the common case.

To remove negative side effect of compression, we propose an dynamic cache compression scheme to determine when compression helps or hurts individual cache references. We use this cost and benefit information to implement a predictor that measures whether compression is helping performance (and therefore should be used for future cache lines) or

hurting performance (and therefore should be avoided for future cache lines). This dynamic scheme achieves most of the benefits of always compressing while avoiding significant performance slowdowns when compression hurts performance. The goals of this design are to use compression to increase effective Level2 cache capacity in order to reduce Level2 misses, limit the impact of cache decompression overhead by providing a bypass path for uncompressed lines, enable an policy to dynamically control compression based on workload demands.

3.1 Management of Variable Sized Compress Cache Block

in the memory. This may entail replacing one or more Level2 lines or compacting invalid/not present lines to make space. More than one line may have to be replaced if the newly allocated line is larger than the LRU line plus the unused segments. In this case, we replace at most two lines by replacing the LRU line and searching the LRU list to find the least-recently-used line that ensures we have enough space. Compacting a set requires moving tags and data segments to maintain the contiguous storage invariant. This operation can be quite expensive, because it may require reading and writing all the set's data segments. For this reason, compaction is deferred as long as possible and is never needed on a read access.

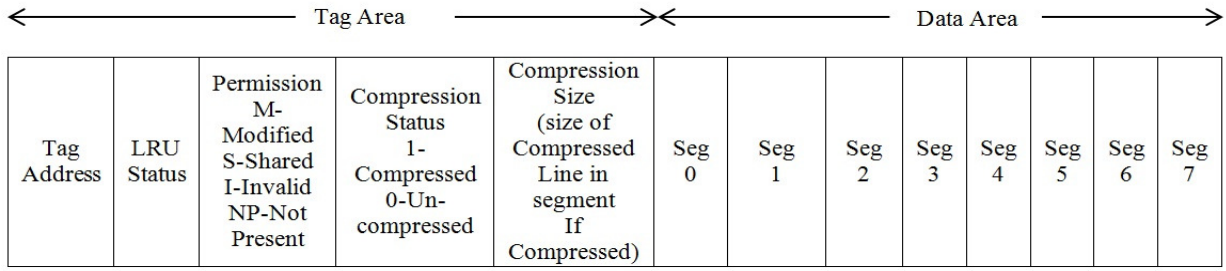

Figure 2: Variable segment Cache

To exploit compression, the Level2 cache must be able to pack more compressed cache lines than uncompressed lines into the same space. One approach is to decouple the cache access, adding a level of indirection between the address tag and the data storage.Decoupled variable-segment cache contains Tag area and data area. Data area is broken in to eight byte segment with size of line is 32 byte. Each line is compressed into and between one and four segment, with four segments being uncompressed form. Tag area contains the fields like 1) tag address to represent address of line 2) LRU status used while replacement of data area of tag3) Compression status represent whether line is in compress or un compress form represent by 1 and 0 respectively 4) Compression size represents size of compress line in number of compress segments.

Because the Level1 and Level2 caches maintain exclusion, an Level1 replacement writes back both clean and dirty lines to the Level2 cache. In many cases, the writeback finds a matching address tag, with space allocated, in state NP. If the compressed size is the same as before, this writeback is trivial. However, if the address tag is not found, or the compressed size has changed, the cache controller must allocate space

4.Dynamic CacheCompressionsTechnique

While compression helps in eliminate long-latency Level2misses, it increases the latency of the (usually more frequent)Level2 hits. Thus, some workloads will benefit from compression, but others willsuffer. Level2cache compression will help if avoided level2 misses and level2 miss penalty is greater than penalized level2 hits and decompression penalty. Where avoided miss is reference to any address hits in level2 cache if and only if any other lines in the level2 cache are compressed called avoided miss. Penalized hits is compression of line done without any need of compression and if such a line is hits called as penalized hits.For a 5 cycle decompression penalty and 400 cycle Level2miss penalty, compression wins if it eliminates at leastone Level2 miss for every 400/5=80 penalized Level2 hits.While this may be easily achieved for memory-intensivel workloads.Smaller workloadsthat fit in alarge Level2 cachemay suffer from degradation of performance if we compress the blocks. Ideally, a compression scheme should compress datawhen the benefit (i.e., avoided misses) outweighs thecost (i.e., penalized Level2 hits). This section describes thecentral innovation in this paper.Adynamic predictor,that monitors the actual

effectiveness of compressionand uses this feedback to dynamically determinewhether to store a line in a compressed or uncompressedform.

Dynamic compression predictor (DCP)is used to decide whether compression is beneficial or not if it is beneficial thencompression is applied otherwise block will be placed in uncompressed form in level2 cache, i.e. DCP uses the past behavior to predict the future. DCP estimates the recent cost or benefitof compression. On a penalized hit, the controller biasesagainst compression by subtracting the decompressionpenalty. On an avoided or avoidable miss, the controllerincrements the counter by the (unloaded) Level2 miss penalty.To reduce the counter size, we normalize these valuesby dividing level2 miss penalty to decompression penalty (e.g., 400 cycles / 5 cycles = 80). We use the DCP when allocating a line inthe Level2 cache. Positive values mean compression hasbeen helping in eliminating misses, so we store the line incompressed form. Negative values mean compressionhas been penalizing hits, so we store the line in uncompressed form. Following flowchart shows the complete centralized idea of dynamic cache compression technique.

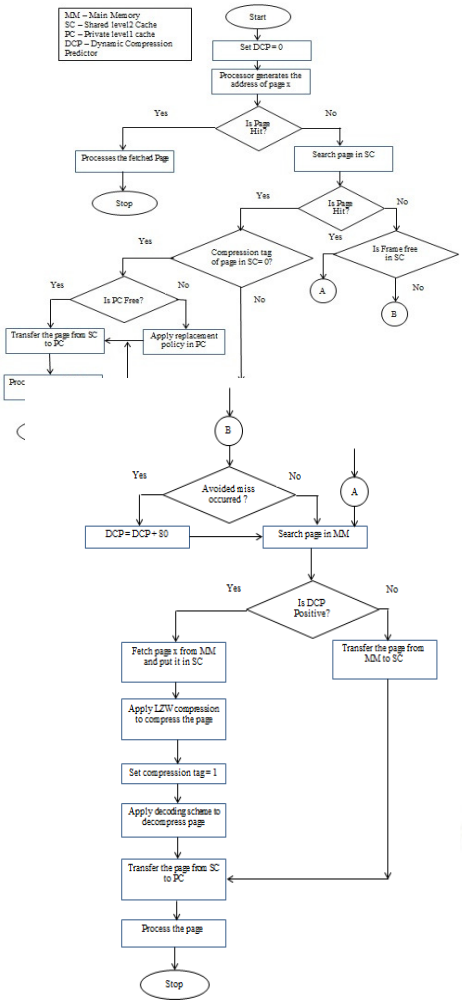

Figure 3: Flowchart of Dynamic compress cache

4. Compression Techniques

In this paper we used LZW compression technique for compression and decompression.LZW compression is a dictionary lookup-based algorithm. Two important features are that the dictionary is built dynamically and the dictionary is included within the compressed message.

LZW is lossless compression technique. It is the foremost technique for general purpose data compression due to its simplicity and versatility. LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, it just adds every new string of characters it sees to a table of strings. Compression occurs when a single code is output instead of a string of characters. The code that the LZW algorithm outputs can be of any arbitrary length, but it must have more bits in it than a single character. The first 256 codes (when using eight bit characters) are by default assigned to the standard character set. The remaining codes are assigned to strings as the algorithm proceeds. The sample program runs as shown with 12 bit codes. This means codes 0-255 refer to individual bytes, while codes 256-4095 refers to substrings. The LZW compression algorithm in its simplest form is shown below. LZW is always trying to output codes for strings that are already known, & each time a new code is output, a new string is added to the string table.

STRING = get input character
WHILE there is still input character DO
CHARACTER = get input character
IF STRING + CHARACTER is in string table
STRING = STRING + CHARACTER
ELSE
Output the code for STRING
Add STRING + CHARACTER to the string table
STRING = CHARACTER
END OF IF
END OF WHILE
Output the code for STRING

The companion algorithm for compression is the decompression algorithm. Following algorithm shows decompression using LZW. It needs to be able

to take the stream of codes output from the compression algorithm, and use them to exactly recreate the input stream. One reason for the efficiency of the LZW algorithm is that it does not need to pass the string table to the decompression code. The table can be built exactly as it was during compression, using the input stream as data. This is possible because the compression algorithm always outputs the STRINGand CHARACTER components of a code before it uses it in the output stream. This means that the compressed data is not burdened with carrying a large string translation table.

Read OLD_CODE
Output OLD_CODE
WHILE there are still input characters Do
Read New_Code
STRING = get translation of New_Code
OUTPUT STRING
CHARACTER = first character in STRING
add OLD_CODE + CHARACTER to the translation table
OLD_CODE = NEW_CODE
END of WHILE

5. Replacement Policies in cache compression

This compression technique use two replacement policy one is least recently used (LRU) and other is efficient replacement policy (ERP). Performance of compression scheme is observed under both the replacement policies and it is find out that for most of the cases efficient replacement policy work well in dynamic cache compression technique. On cache misses, data loaded from main memory is always allocated in the level2 shared cacheto level1 private cache. This normally requires that one block is evicted from the private partition of the cache. The evicted block is allocated in the shared cache partition. Eviction of blocks from private as well as shared done through following two policies.

6.1 LRU replacement policy

The LRU replacement policy replaces the least recently used pages in the cache. It only maintains the recency of pages. LRU policy uses the recent past as an approximation of the near future, and therefore replaces the line that has not been used for the longest period of time. The LRU performs well in some applications, but it isn't able to cope with access patterns such as file scanning, regular accessed over more pages than the cache size, and accessed on pages with distinct frequencies. LRU has a high overhead cost ofmoving cache blocks into the most recently used position each time acache block is accessed. LRU does not use 'frequency' informationof memory accesses and LRU is prone to cache pollutionwhen a sequence of single-use memory accesses that are larger thanthe cache size is fetched from memory. This policy fails for sequential access.

6.2 Efficient Replacement policy

ERP tries to replace the page which is not referenced more often. To implement the ERP cache replacement policy, we created a pointer called *Replace_Ptr*and an array of *Hit_Bit called* reference bits for each cache block in a cache set. The ERP policy uses a circular buffer with the*Replace_Ptr* pointing to the cache block that is to be replaced when a cache miss occurs. The use of the circular queue avoids the movement of cache blocks from the head of the queue to the tail of the queue; instead it replaces the block by advancing the *Replace_Ptr*to point to the next cache block in the circular queue. Replace_Ptr will only be advance by resetting hit bit when hit bit of page that Replace_Ptr is pointing is 1. During a cache hit, the ERP policy will set the *Hit_Bit*of the accessed cache block to 1 to indicate that the cache block has been hit. When a cache miss occurs, *Replace_Ptr will* notto advance to the next cache block whenever the *Hit_Bit*of the cache block pointed by the *Replace_Ptr*is equal to '0' and new cache block will be placed atReplace_Ptr position. Initially reference bit is 0, policy sets it to 1 as soon as the corresponding cache block is referenced. Reference bit = 0 means that the cache block has not been referenced and hence, it can be replaced. Reference bit = 1 means the corresponding cache block has been referenced and hence, is likely to be used soon therefore, it is not replace. The main purpose in the development of the ERP is to create a cache replacement policy that has lower maintenance cost compared to LRU replacement policies.

6. Result of simulation

To understand the utility of dynamic compression we compare the dynamic compression with two extreme

policy never and always. In never data is never stored in compressed form and in always data is always stored in compressed form. Thus never tries to reduce hit latency while always tries to reduce miss rate. Dynamiccompression use compression only when, it predicts that the benefit are more than the overhead. Here workload1 is smaller workload, workload2 whose size is slightly greater than Level2 cache, workload3 memory intensive workload, Workload4 is too large workload.

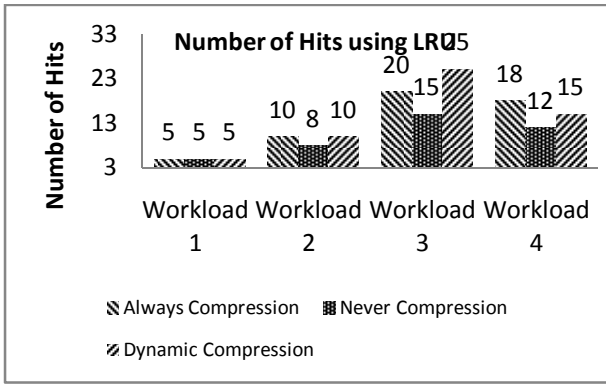

Figure 4: Number of hits for LRU replacement policy under three compression techniques for different workload.

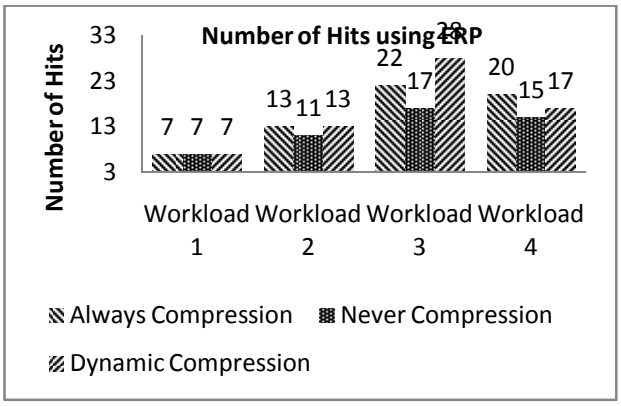

Figure5: Number of hits for ERP replacement policy under three compression techniques for different workload.

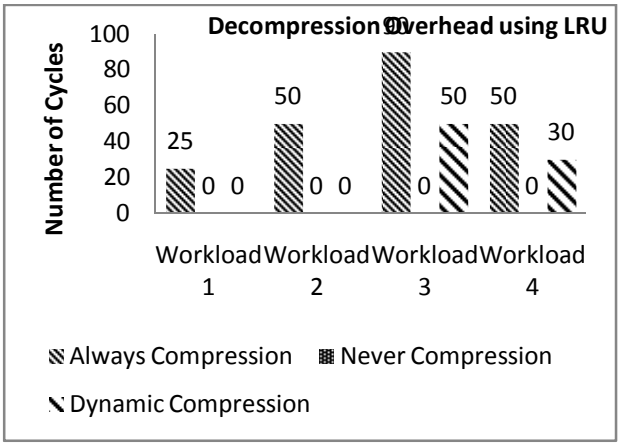

Figure6: Decompression overhead for LRU replacement policy under three compression techniques for different workload.

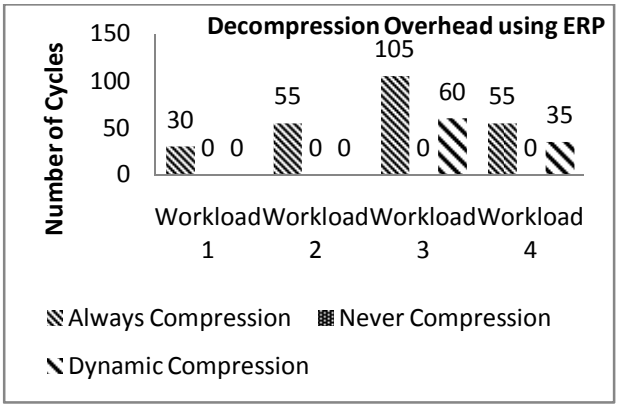

Figure7: Decompression overhead for REP replacement policy under three compression techniques for different workload.

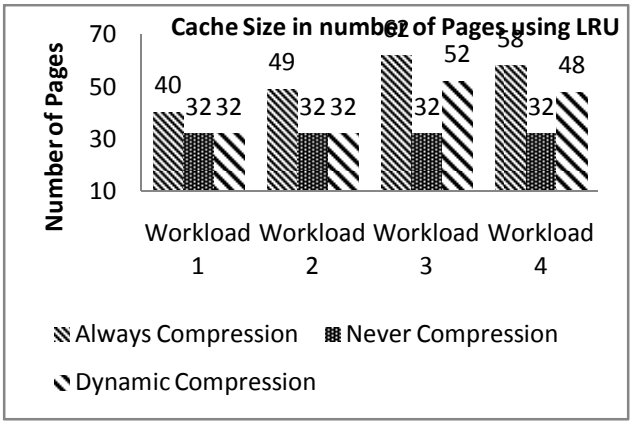

Figure8:Cache size in number of pages for LRU replacement policy under three compression techniques for different workload

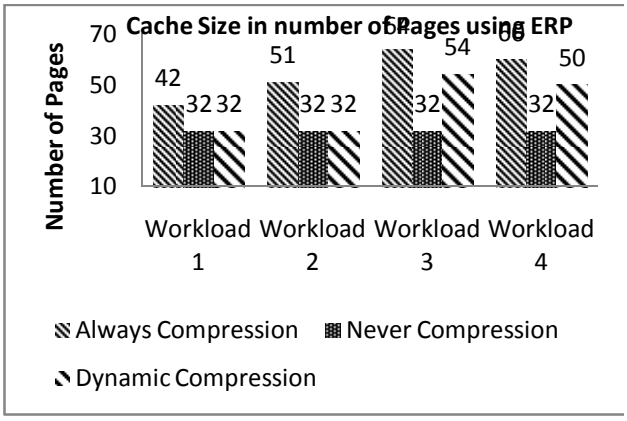

Figure9: Cache size in number of pages for ERP replacement policy under three compression techniques for different workload.

From the above results it is observed that for small workload (workload1) never compression and dynamic compression work better than always compression because it does not required any decompression overhead. For workload whose size is slightly greater than size of level2 cache(workload2) and for memory intensive workload(Workload 3) dynamic gives better performance than never and always because always compression required more decompression overhead and never generate less hits. For too large workload always compression gives more hit but it also required more decompression overhead. Always compression increase size of cache in comparison with never and dynamic compression but always required decompression overhead.

If we compare LRU and ERP replacement policy for replacement of page in level2 cache performance under ERP is better than conventional LRU replacement policy for three compression techniques.

7. Conclusions

In this paper, we propose adynamic cache compression policyto improve the performance of chip multiprocessor running under different workload. Weuse a two-level cache hierarchy where the Level1 holdsuncompressed data while the Level2 can store data either in compress or uncompressed form. Our dynamic policy dynamicallyadjusts to the costs and benefits of compression. A dynamic compression predictor counter predicts whether the Level2cache should store a line in compressed or uncompressedform. A dynamic compression predictor updates the counterbased on whether compression could eliminatea miss or incurs an unnecessary decompressionoverhead.We have shown that dynamic compression improves performance for maximum type of workload i.e. memory intensive workload, small workload and workload whose size is slightly more than size of level2 cache. If we compare performance of LRU and ERP replacement policy under three compression technique performance of ERP replacement policy is slightly better than conventional LRU replacement policy.

References

[1].Jang-SooLee,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.

[2]. 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.

[3]. PrateekPujara, AneeshAggarwal, "Restrictive Compression Techniques to Increase Level Cache Capacity", IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD 2005, PP: 327-333.

[4]. 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.

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

[6]. Martin, Burtscher and ParujRatanaworabhan, "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data" IEEE Transaction on Computers, vol. 58(1), January 2009, PP: 18-31.

[7]. David Chen, Enoch Pegerico and Larry Rudolpha, "A Dynamically Partitionable Compressed Cache", Proceeding of Singapore-MIT Alliance Symposium, 2003.

Guido Araujo, Paulo Centoducatte, Mario Cartes, and Ricardo Pannain. Code Compression Based on Operand Factorization. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 194–201, November 1998.

[8]. Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley Jones, and BodoParady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer

Performance. In *Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[9]. Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[10]. Luiz A. Barroso, KouroshGharachorloo, and EdouardBugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[11]. Luiz Andre Barroso, KouroshGharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

Authors

1) Prof. H. R. Deshmukh is Associate professor in Dptt. Of CSE B.N.C.O.E., Pusad (India), He did his B.E.(Computer Science & Engineering) in 1995 & M.E. (Computer Science & Engineering) in 2008 from from PRMIT&R, Badnera-Amravati, affiliated to SGB Amravati University, Amravati. He is research scholar from 2010 & National Executive Council member of Indian Society for Technical Education(ISTE) New Delhi.

2) Dr. G. R. Bamnote is Professor & Head 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).