International Journal of Engineering Research and General Science Volume 2, Issue 3, April-May 2014
ISSN 2091-2730

Improvement in performance of Chip-multiprocessor using Effective Dynamic Cache Compression Scheme

Poonam Aswani[1], Prof B. Padmavathi[1]

[1]Department of Computer Engineering, GH Raisoni College, Pune, India

E-mail- aswanipoonam41@gmail.com

Abstract— Chip Multiprocessors (CMPs) combine multiple cores on a single die, typically with private level-one caches and a shared level-two cache. The gap between processor and memory speed is alleviated primarily by using caches. However, the increasing number of cores on a single chip increases the demand on a critical resource: the shared L2 cache capacity. In this dissertation work , a lossless compression algorithm is introduced for fast data compression and ultimately CMP performance. Cache compression stores compressed lines in the cache, potentially increasing the effective cache size, reducing off-chip misses and improving performance. On the downside, decompression overhead can slow down cache hit latencies, possibly degrading performance. While compression can have a positive impact on CMP performance, practical implementations of compression raise a few concerns: Compression algorithms have high overhead to implement at the cache level. Decompression overhead can degrade performance . Generally compression algorithm are not effective in compressing small blocks. Hardware modification is required. In this dissertation work , we make contributions that address the above concerns. We propose a compressed L2 cache design based on an effective compression algorithm with a low decompression overhead. We developed dynamic cache compression scheme that dynamically adapts to the costs and benefits of cache compression, and employs compression only when it will enhance the performance. We show that cache compression improve CMP performance for different workloads.

Keywords: Cache Compression, Compression Ratio, LRU, ERP, Off-chip Memory, Memory latency,L1 Cache, L2 Cache

1. Introduction

The widening gap between processor and memory speeds, results because of tight constraints on the amount of on-chip cache memory and the high latency of off-chip memory, such as dynamic random access memory. More time is essential to access off-chip memory time required to access generally takes an accessing on-chip cache. Hence to improve memory system efficiency cache hierarchies is been incorporated on chip, but it is constrained by die area and cost. Cache compression is one such technique; data in last-level on-chip caches, e.g., L2 resulting in larger usable caches. In the past, researchers have reported that cache compression can improve the performance of uniprocessors . However past work requires complex hardware for cache compression. Past work does not considered the performance, area and power consumption requirement In this dissertation , we explore using compression to effectively increase these resource and ultimately overall system throughput. To achieve this goal, we identify a distinct and complementary design where compression can help improve CMP performance: Cache Compression. Cache compression stores compressed lines in the L2 cache, potentially increasing the effective cache size, reducing off-chip misses, and improving performance. Moreover, cache compression can also allow CMP designers to spend more transistors on processor cores. On the downside, decompression overhead can slow down cache hit latencies, which degrades performance for applications that would fit in an uncompressed cache. Such negative side-effectsmotivate a compression scheme that avoids compressing cache lines when compression is not beneficial.

The ideal cache compression technique would be fast, simple, and effective in saving storage space. Clearly, the resulting compression ratio should be large enough to provide a significant upside, and the hardware complexity of implementing the scheme should be low enough that its area and power overheads do not offset its benefits. Perhaps the biggest problem to the adoption of cache compression in commercial microprocessors is decompression latency. Unlike cache compression, which takes place in the background upon a cache fill (after the workload is supplied), cache decompression is on the critical path of a cache hit, where minimizing latency is extremely important for performance., we consider compression of the L2 caches. The three desired goals of having fast, simple, and effective cache compression are at odds with each other (e.g., a very simple scheme may yield too small a compression ratio, or a scheme with a very high compression ratio may be too slow, etc.), the challenge is to find the right balance between these goals. To achieve significant compression ratios while minimizing hardware complexity and decompression latency, we propose a new cache compression technique called Dynamic cache compression. It dynamically adapts to the costs and benefits of cache compression, and

implements compression only when it helps performance. We propose and evaluate a CMP design that implements cache compression.

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 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 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 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. [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.

3. Cache Compression in Chip Multiprocessors

The increasing number of processor cores on a chip increases demand on shared caches. Cache compression addresses the increased demand on both of these critical resources in a CMP. In this project, we propose a CMP design that supports cache compression. CMP cache compression can increase the effective shared cache size, potentially decreasing miss rate and improving system throughput. In addition, cache compression can decrease demand on pin bandwidth due to the decreased miss rate.

Due to the significant impact of the memory wall on performance, many existing uniprocessor and CMP systems implement hardware prefetching to tolerate memory latency. Prefetching is successful for many workloads on a uniprocessor system. For a CMP, however, prefetching further increases demand on both shared caches and pin bandwidth, potentially degrading performance for many workloads. This negative impact of prefetching increases as the number of processor cores on a chip increases.

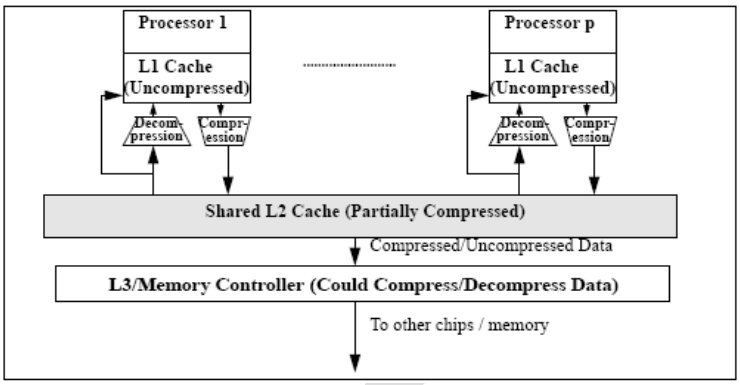

Figure: A Single-Chip p-core CMP with Compression Support

4. Cache Compression Technique

Compression is achieved by two means:

(1) It uses statically decided, compact encodings for frequently appearing data words

(2) It encodes using a dynamically updated dictionary allowing adaptation to other frequently appearing words.

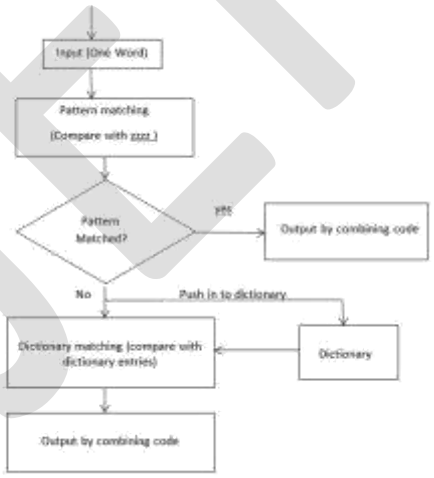

Figure :Cache Compression

The dictionary supports partial word matching as well as full word matching. The ‘Pattern’ column describes frequently appearing patterns, where ‘z’ represents a zero byte, ‘m’ represents a byte matched against a dictionary entry, and ‘x’ represents an unmatched byte. In the ‘Output’ column, ‘B’ represents a byte and ‘b’ represents a bit. During one iteration, each word is first compared with patterns “zzzz” and “zzzx”. If there is a match, the compression output is produced by combining the corresponding code and unmatched bytes . otherwise the compressor compares the word with all dictionary entries and finds the one with the most matched bytes. The compression result is then obtained by combining code, dictionary entry index, and unmatched bytes, if any. Words that fail pattern matching are pushed into the In each output, the code and the dictionary index, if any, are enclosed in parentheses.

Although a 4-word dictionary can be used, the dictionary size is set to 64B in our implementation. The dictionary is updated after each word insertion. During decompression, the decompressor first reads compressed words and extracts the codes for analyzing the patterns of each word, which are then compared against the codes.

5. Block diagram of Dynamic compression policy

Here address is generated by the CPU in the form of process id (pid) and page no of the corresponding process (pno). Thus the address consists of combination of pid and pno.

Once the address is generated, CPU first searches the address in page mapped table of private cache. Here, there are two chances:

a) Page hit occurs
b) page miss occurs.

If page is present in the page mapped table of private cache, we say that page hit occurs. If so, first search the respective frame no. using search in private method and get the required data using get Page Private method and then stop. If page is not available in private cache and shared cache also we need to fetch it from main memory. But, in this project we are using adaptive policy to decide whether to compress the data before storing it in the shared cache or not. So first we need to observe whether the miss is of avoidable or unavoidable type. If any of the tag in shared cache is 0 it means that we might avoid the present miss if that page was compressed and thus we need to increment the value of global compression predictor. Once this is done search the page in main memory and before transferring it to shared cache decide whether to compress the data.

6. Cache 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 not to 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 at Replace_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.

7. Implementation Details

Two policies that is never compress and always compress is developed. From two policies we can predict that in compression the number of hits is more as compared to uncompressed cache format.

A) Mathematical Model

- U = Set of all modules in the system = {P, CM,MM} Where, P-Processor, C-Cache, MM-Main Memory
- CM = It contains shared and private cache IS = {SC,PC}

  Where, SC-Shared Cache and PC-Private Cache
- P generates Page Numbers and Process Id.

- CM={LRU, ERP} where LRU is Least Recently Used and ERP is Efficient Replacement Policy.
- PNo € {SC,PC} where PNo is page numbers generated by processor.

Page Numbers generated by P is first searched in Cache Memory If it is found in cache memory, then we can say hit is occurred . The page status is checked (Compressed/Uncompressed). If it is Compressed then page is decompressed and provided to processor. If Page is not present in Cache memory then Main Memory is searched for page . Now the page is transferred from main Memory to Cache memory and then it is fetched to processor for processing.

8. Result

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. Dynamic compression use compression only when, it predicts that the benefit are more than the overhead. In results, we have compared no of hits occurred under different workloads. In never data is in uncompressed form and in always data is always stored in compressed form. Thus never compression technique reduces hit time , while always tries to reduce miss rate. different workloads are considered. Dynamic compression use compression only when, it predicts that the benefit are more than the overhead. Here different workloads are considered. 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. For small workloads (Workload1), the number of hits is same under all three policies are same. For Workload2, the number of hits for dynamic is more as compared to other two policies. For Workload3 (Memory intensive), the number of hits for dynamic policy are more as compared to other two policies. Decompression overhead is also another factor for comparing result between three policies. Decompression overhead is zero for never compression technique. In Always, decompression overhead is more as compared to Dynamic compression.

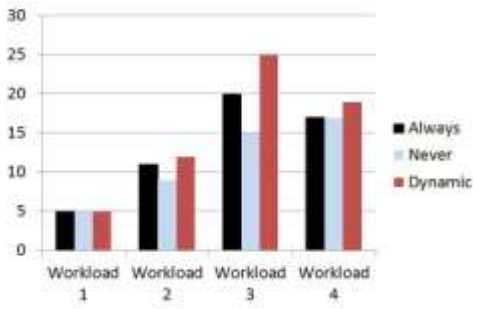

Figure : Hits In Shared Cache

9. Conclusion and Future Work

A) Conclusion

Chip multiprocessors (CMPs) combine multiple processors on a single die. The increasing number of processor cores on a single chip increases the demand on the shared cache capacity. Demand on this critical resource can be further exacerbated by various compression techniques. In this project, we explored using compression to effectively increase cache size and ultimately CMP

performance. Cache compression stores compressed lines in the cache, potentially increasing the effective cache size, reducing off-chip misses, and improving performance. On the downside, decompression overhead can slow down cache hit latencies, possibly degrading performance. While compression can have a positive impact on CMP performance, practical implementations of compression raise a few concerns (e.g., compression's overhead).We proposed a compressed shared cache design based on a simple compression scheme with a low decompression overhead. Such design can double the effective cache size .We developed an adaptive compression algorithm that dynamically adapts to the costs and benefits of cache compression, and uses compression only when it helps performance. We presented a simple analytical model that helps provide results of applying compression in multiprocessor system.

B) Future Scope

Different compression algorithms can be used to compress the data. For ex for compressing audio, video, image data and so on. Multiprocessing can be used for running several process and generating address simultaneously. It can be extended for server farm where memory becomes bottleneck. We can analyze the performance of the system using industry benchmarks. We can implement this system at the kernel level.

REFERENCES:

[1] Alaa R. Alameldeen and David A. Wood. Adaptive Cache Compression for High-Performance Processors. In *31st AnnualInternational Symposium on Computer Architecture (ISCA-31) Munich, Germany, June 19-23, 2004*

[2] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium onHigh- Performance Computer Architecture*, pages 7–18, February2003.

[3] Alaa R. Alameldeen and David A. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. Technical Report 1500, Computer sciences Department, University of Wisconsin–Madison, April 2004.

[4] A. R. Alameldeen and D. A.Wood, "Interactions between compression and refetching in chip multiprocessors," in Proc. Int. Symp. High- Performance Computer Architecture, pp. 228– 239, Feb. 2007

[5] A. Moffat, "Implementing the PPM data compression scheme," IEEE Trans. Commun. , vol. 38, no. 11, pp. 1917–1921, Nov. 1990

[6] E. G. Hallnor and S. K. Reinhardt, "A compressed memory hierarchy using an indirect index cache," in Proc. Workshop Memory Performance Issues, pp. 9–15, 2004.

[7] Sharada Guptha M N, H. S. Pradeep & M Z Kurian. A VLSI Approach for Cache Compression in Microprocessor.In International Journal of Instrumentation, Control and Automation (IJICA) ISSN : 2231-1890 Volume-1, Issue-2, 2011
third Post Graduate Symposium for Computer Engineering cPGCON 2014

[8]Bulent Abali, Hubertus Franke, Dan E. Poff, Jr. Robert A. Saccone, Charles O. Schulz, Lorraine M. Herger, and T. Basil

Smith. "Memory Expansion Technology (MXT): Software Support and Performance." *IBM Journal of Research andDevelopment*, 45(2):287–301, March 2001.

[9]Jacob Ziv and Abraham Lempel. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on InformationTheory*, 23(3):337–343, May 1977.

[10]Jacob Ziv and Abraham Lempel. "Compression of Individual Sequences Via Variable-Rate Coding." *IEEE Transactions onInformation Theory*, 24(5):530–536, September 1978.

[11]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.

[12]Charles Lefurgy, Eva Piccininni, and Trevor Mudge,"Reducing Code Size with Run-time Decompression",Proceedings on 6th International Symposium on High Performance computer Architecture HPCA, 2002, PP. 218- 228