CATCH: A Method for Dynamically Detecting Cache-Content-Duplication

Marios Kleanthous and Yiannakis Sazeides

Department of Computer Science, University of Cyprus, Kallipoleos 75, 1678 Nicosia, Cyprus

ABSTRACT

Cache-content-duplication occurs when there is a miss for a block in a cache and the required block of data resides already in the cache but under a different tag. Caches aware of content-duplication can have smaller miss penalty by fetching, on a miss to a duplicate block, directly from the cache instead from lower in the memory hierarchy, and can have lower miss rates by allowing only blocks with unique content to enter a cache. This paper proposes Catch, a mechanism for dynamically detecting cache-content-duplication

KEYWORDS: caching; redundancy; duplication

1 Introduction

The importance of caches and memory hierarchy has increased over time due to the growing gap between processor and memory performance [11]. Caches, consequently, have been central to numerous research studies. Several techniques have been proposed to improve various aspects of caches by reducing their miss rates, cache size, latency and energy. Most of these techniques attempt to exploit different types of properties of memory addresses and data, such as locality [3], predictability [2], and redundancy [1], [4], [5], [6], [7], [8], [9], [12]. This work identifies a new cache property that may influence cache performance: the cache-content-duplication (CCD). This phenomenon occurs when there is a miss for a block in a cache for which the block data reside already in the cache but under a different tag.

CCD is a manifestation of redundancy in cache content and notions related to cache content redundancy have been investigated before [1], [4], [5], [6], [7], [8], [9], [12]. What distinguishes CCD from previous work is that it exploits cache content redundancy at the granularity of cache blocks instead of considering the compression of patterns in the cache content. And this enables new mechanisms for memory hierarchy optimization.

We have experimental evidence that show cache-content-duplication to occur for various types of workloads, both for instructions and data, and for different types of caches and cache configurations. CCD may also exist in caches with compressed content since, as far as we know, previous compression techniques allow different compressed blocks to be identical.

This work introduces *Catch,* a hardware mechanism that can dynamically detect cache-content-duplication. Several optimizations are considered to increase the Catch's accuracy and/or cost-efficiency.

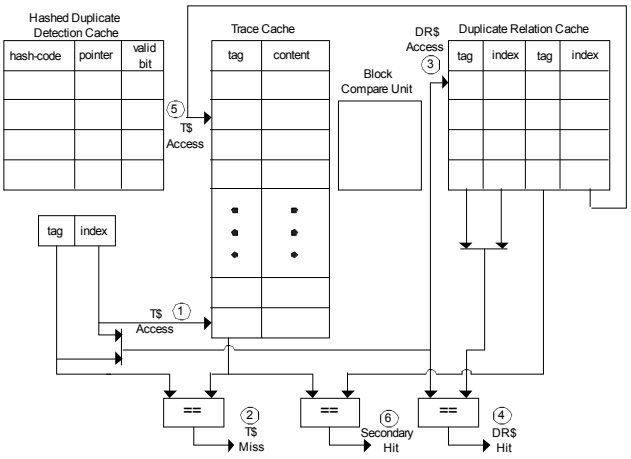

Fig. 1. The Catch and the flow for a Cache miss, DR hit, Cache hit

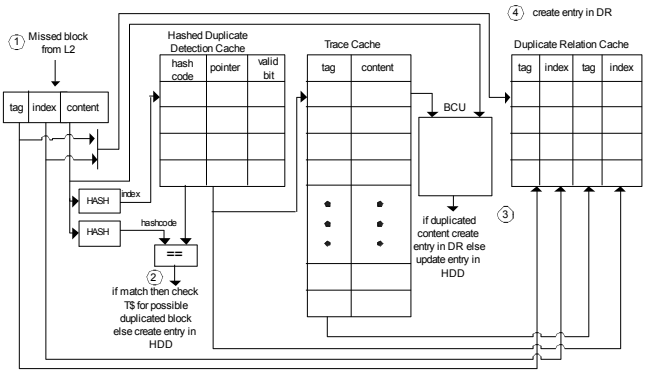

Fig. 2. The flow for a Cache miss, DR miss and HDD miss

2 CATCH: A method for dynamically detecting CCD

In order to exploit CCD a mechanism is required for detecting cache-content-duplication. Specifically, this mechanism given the tag and index (tag-index) of a block with a cache miss should return whether there is a duplicate in the cache and the tag-index of the duplicated block. This section presents a method for dynamically detecting CCD. We will refer to this mechanism as *Catch.* The *Catch* can be applied to any type of instruction cache but we focus on its application to a trace cache.

The architecture of Catch, shown in **Fig. 1**, includes the *Duplicate-Relation cache (DR)*, the *Hashed Duplicate Detection cache (HDD),* and the *Block Compare Unit (BCU)*. The functionality of these components and their updating policies is the subject of this section.

2.1 The Duplicate-Relation cache (DR)

The *duplicate-relation cache (DR)* contains block duplication-relations detected by the *Catch*. Each DR entry contains a tag-index of a missed block and the tag-index of its duplicate block. The use of a full tag-index is sufficient to prevent false duplicate relations. For instruction caches once a duplicated relation is established it is assumed to be always correct (in the case of self-modifying code or page remapping the DR may need to be flashed to ensure correctness). **Fig. 1** illustrates the sequence of steps in the case of a cache miss that has an entry in the DR and a duplicate in the cache.

2.2 The Hashed-Duplicate-Detection cache (HDD)

An entry in the DR is created when a block with a cache miss is fetched from a lower level cache and is found to be a duplicate with a block already in the cache. The detection of CCD requires a mechanism that given the content of a block it provides a tag-index for a candidate duplicate-block currently in the cache.

This functionality is provided by the *Hashed-Duplicate-Detection cache (HDD)*. Each entry in the HDD contains a valid bit, a hash-code, which encodes the content of a block, and the corresponding tag-index of the block. The use of a hash-code reduces the cost and complexity of detecting duplication but may lead to unnecessary tests for duplication.

Nevertheless, we found that a simple folding of the block content into few bits provides very accurate encodings (often 99.9% accurate).

An HDD entry is maintained valid as long as its corresponding block is in the cache. This prevents unnecessary cache access when the duplication candidate is not in the cache. Consequently, when a block is evicted it invalidates its corresponding HDD entry, unless the HDD entry was updated in the mean time by another block.

The above ensures that each valid entry in the HDD has a corresponding block in the cache. Because of this and inspired by the idea proposed in [10], the full tag-index in HDD is replaced with a small pointer in the cache. This reduces the size of each HDD entry considerably, for example for a 2-way set associative 16KB cache with 32B block size we can maintain a 9 bit pointer -eight bits for selecting a set and one bit for the way- instead of complete tag-index.

The HDD is indexed using a hash of the content of a missed block after is fetched from the lower-level of memory hierarchy. This hash can be different from the one used for producing the hash-code for a block. **Fig. 2** illustrates the sequence of steps in the case of a cache miss that has a duplicate in the cache but not an entry in DR.

2.3 The Block Compare Unit (BCU)

When two blocks are signaled by the HDD as possible duplicates, their contents are compared using the BCU to detect whether there is indeed duplication. The compare function used in the BCU can be a simple bitwise comparison of the instructions in the two blocks. BCU optimizations that advanced compare functions are discussed in Section 2.5.

2.4 Allocating and Updating an HDD and a DR entry

An HDD entry is allocated when a block is both a cache and an HDD miss. The different scenarios for allocating an HDD entry are the following:

Cache miss, DR miss, HDD miss: A block is a miss in the cache and no entry in the DR matches its tag-index. The block is fetched from a lower lever of memory hierarchy and its content hash-code is calculated. The HDD is accessed with this hash-code and on a miss a new HDD entry is created.

Cache miss, DR hit, Cache miss, HDD miss: same as above except that there is a DR hit that leads to a cache access that misses because the duplicate block was evicted.

An HDD entry is updated when a block is a miss in the instruction cache and hits in the HDD. In this case a DR entry is also created or updated. The different cases for updating an HDD entry and allocating or updating a DR entry are the following:

Cache miss, DR miss, HDD hit: A block is a miss in the cache and the DR. The block is fetched from a lower level in the memory hierarchy and its hash-code is calculated. The HDD is accessed with the hash-code, and

- If we hit in the HDD and the entry is valid then the cache is accessed with the pointer in the HDD. The two blocks contents are then compared in the BCU and if there is duplication a DR entry is created with the missed tag-index and the duplicate tag-index pointed by the HDD. The HDD entry remains the same.
- If we miss or the HDD entry is invalid then an HDD entry is allocated and points to the newly fetched block in the cache.
- When the content of the missed block and the one pointed by the HDD do not match in the BCU, we have a case of a *false hash-code match*. This we found it to occur very rarely for hash-codes with 24 to 32 bits. When this happens, the HDD entry will be updated with a pointer to the missed block.

Cache miss, DR hit, Cache miss, HDD hit: same as above except (a) there is a DR hit that leads to a cache access that does not hit, and (b) if the HDD points to a trully duplicate block then a new DR entry will be allocated with the missed block's tag-index and the duplicate tag-index pointed by the HDD.

2.5 Performance Optimizations

The ***keep-offset-in-dr*** optimization aims to increase content-duplication by masking out from the compare process in BCU the offsets and targets of conditional and unconditional direct branches and keeping in the DR for each duplicate block its offsets

and targets. This is aimed to convert blocks that contain exactly the same computation but at different program locations. Two possible caveats of this optimization is the extra cost per DR entry, and that secondary cache reads may need to combine information from the cache and the DR which may make fetching more complicated.

Other examples of BCU optimizations, not evaluated in this work, is to consider a compare function that could rearrange source operands of commutative operations and reorder data independent instructions in a block to facilitate content duplication.

2.6 Cost Reduction Optimizations

An HDD entry contains a hash-code, a pointer to the cache (8 bits for the set, and 1 bit for the way), and a valid bit. The 32-bit hash-code produced very rarely false-hash-matches (at most for 0.002% of the misses for one benchmark). This is an indication that we can reduce the HDD cost without sacrificing performance.

The DR entry cost can be reduced by introducing criteria for when to insert a duplicate relation in the DR. The following criteria were reached after some cursory analysis: do to not insert a relation in the DR unless the following are the same for the two tag-indices (i) most significant 9 bits of the starting address, and (ii) conditional branch directions (for trace caches). This reduces the cost of a DR entry by 12 bits.

When the ***keep-offset-in-dr*** optimization is employed, the DR should have space for four direct targets. For the ISA consider in this study this means 4x26=104 extra bits for each DR entry. To reduce the number bits required by the offsets and direct targets extra insertion criteria can be applied. These criteria are to insert duplicated relations only when all of the following are true: traces have at most two basic blocks, and the upper 10 bits of direct targets must the same.

References

1. R. Alameldeen and D. Wood, "Adaptive Cache Compression for High-Performance Processors", Proc. of the 31st International Symposium on Computer Architecture, June 2004, pg. 212-223.
2. J. Baer and Tien-Fu Chen, "An effective on-chip preloading scheme to reduce data access penalty", Proc. of the 1991 ACM/IEEE Conference on Supercomputing, Nov. 1991, pg. 176-186.
3. P. J. Denning, "Virtual Memory", ACM Computing Surveys (CSUR), Sept. 1970, 2(3):153-189
4. E. Hallnor and S. Reinhardt, "A Compressed Memory Hierarchy using an Indirect Index Cache", Tech. Rep. CSE-TR-488-04, 2004.
5. S. Hines, J. Green, G. Tyson and D. Whalley, “Improving Program Efficiency by Packing Instructions into Registers”, Proc. of the 32nd International Symposium on Computer Architecture, June 2005.
6. M. Kjelso, M. Gooch, S. Jones, "Design and Performance of a Main Memory Hardware Data Compressor", Proc. of the 22nd EUROMICRO Conference, Sept. 1996, pg 423-430
7. C. Lefurgy, E. Piccininni and T. Mudge, "Reducing Code Size with Run-time Decompression", Proc. 6th International Symposium on High-Performance Computer Architecture, Jan. 2000, pg. 218-227.
8. C. Lefurgy, P. Bird, I-Cheng Chen and T. Mudge, "Improving Code Density Using Compression Techniques", Proc. 30th International Symposium on Microarchitecture, Dec. 1997, pg. 194-203.
9. M. A. Postiff and T. Mudge, "Smart Register File for High-Performance Microprocessors", University of Michigan CSE Tech. Rep. CSE-TR-403-99, June 1999.
10. A. Seznec, "Don’t use the page number, but a pointer to it", Proc. of the 23rd International Symposium on Computer Architecture, May 1996, pg. 104-113.
11. W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious", Computer Architecture News, March 1995, 23(1):20-24.
12. Y. Zhang, J. Yang and R. Gupta, “Frequent Value Locality and Value-Centric Data Cache Design,” Proc. of the 9th International Symposium on Architectural Support for Programming Languages and Operating Systems, Nov. 2000, pg. 150-159.