The Duplication of Content in Instruction Caches and its Performance Implications

Marios Kleanthous and Yiannakis Sazeides
Department of Computer Science
University of Cyprus

CS-TR-01-2005

January 2005

Abstract

This paper shows that when there is a miss for a block in a cache the required block of data may reside already in the cache but under a different tag. We refer to this phenomenon as Cache-Content-Duplication. This report characterizes cache-content-duplication for instruction caches and investigates its potential for improving their performance. The experimental results show that up to 10% of the misses of a regular instruction cache and up to 20% of the misses of a basic-block cache are for duplicated blocks. The paper proposes two memory hierarchy enhancements that exploit cache-content-duplication. One of these enhancements is evaluated and shown to have the potential to reduce instruction cache and basic-block cache misses by up to 49% and 33% respectively.

Keywords: caching, redundancy, duplication, compression

1. Introduction

The importance of caches and memory hierarchy has increased over the last two decades due to the growing gap between processor and memory performance [13]. 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 [11], predictability [2, 3, 8] and redundancy [1, 5, 6, 7, 9].

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 data reside already in the cache but under a different tag.

CCD is a manifestation of redundancy in cache content. Related notions to cache content redundancy have been investigated before. Lefurgy et al. [5] examined static compression of instruction sequences to increase the code density for embedded processors. The compression of code in main memory using an instruction dictionary was studied in [7]. The selective compression of level 2 cache-content was the subject of [1]. A scheme that compresses the contents of both main memory and caches was examined in [6]. The redundancy problem has been investigated also at the highest level of memory hierarchy, the registers [9]. What distinguishes our work is that we investigate cache content redundancy at the granularity of cache blocks instead of considering the compression of patterns in the cache content. And this may enable new mechanisms for memory hierarchy optimizations.

Examples of CCD based optimizations are: (a) a mechanism that identifies blocks with duplicated content and on a miss, for such blocks, we fetch the content of the duplicate block already in the cache instead of fetching from lower levels in the memory hierarchy, and (b) the unique-content-cache which contains only blocks with unique content and maintains multiple tags for one block.

We believe that cache-content-duplication exists in various types of workloads, for both instructions and data, and for different types of caches and cache configurations. As a first step towards understanding and exploiting CCD this paper is focused on the content duplication in instruction caches.

The frequency of CCD in instruction caches may be significant, because (1) high level language programs often contain identical instruction sequences in different segments of a program due to: copy-paste programming practices and reuse of standard library and loops in different parts of code, and (2) compiler transformations, such as compiler inlining and macro expansion, lead to duplicated code sequences.

This work introduces the phenomenon of cache-content-duplication and characterizes its frequency for various instruction cache configurations. The experimental results for SPEC benchmarks show that up to 10% of the misses of an instruction cache and up to 20% of the misses of a basic-block cache are for duplicated blocks. The paper also describes the required functionality to detect and use content-duplication, proposes two techniques to exploit the phenomenon and evaluates the potential of one of them, the unique-content-cache. The unique-content-cache is shown to have the potential to reduce misses for an instruction cache and a basic-block cache up to 49% and 33% respectively.

In Section 2, we are discussing previous work on cache redundancy. Section 3 presents the simulation environment and the different configurations considered for experimentation. In Section 4, we are presenting the simulation results that characterize CCD. Section 5 discusses possible applications of CCD and evaluates one of these applications. Finally, in Section 6 we conclude and give some directions for future work.

2. Related Work

The redundancy of the memory and cache content has been the subject of several previous papers. The main objectives of these proposals were to increase the effective memory/cache capacity and to achieve higher bandwidth during transfers of information between different levels of the memory hierarchy. Some of the most relevant of these papers are discussed below.

Lefurgy et al. [5] explores the idea of keeping compressed code in instruction memories of embedded processors. Based on static analysis common sequences of instructions are assigned unique codes. These codes are stored in instruction memory and are expanded to their original form after being read. Lefurgy et al. [7] studied the concept of keeping compressed code in main memory and "software decompressing" on a cache miss. More specifically, frequently used instructions in the original code are replaced by pointers to an entry of an instruction dictionary. Due to the high replication of instructions and the small size of the dictionary, the amount of memory required to store the static code in main memory is significantly reduced.

Alameldeen and Wood [1] keeps compressed information, both instructions and data, only in level 2 cache and can dynamically choose to keep data in uncompressed form when the overhead of compression may cause a performance degradation. Hallnor and Reinhardt [6] proposed a scheme that maintains compressed data both in main memory and on-chip cache. This enables the data to travel through the bus in compressed form. This scheme, therefore, offers both extra space on main memory and cache and higher transfer rates from main memory to cache.

Postiff and Mudge [9] proposed smart-register-files aiming to solve the aliasing problem of more than one registers referring to the same datum, either address or data.

The distinct feature of our work is that we consider redundancy at the granularity of cache blocks. Previous work considered the redundancy and compression of arbitrary length sequences of data or instructions or considered the compression at the granularity of individual instructions. We demonstrate later in the paper that approaching redundancy in terms of cache blocks enables new mechanisms for memory hierarchy optimization.

3. Experimental Framework

We used simplescalar 3.0 to implement two types of instruction caches. The first is an instruction cache where blocks are always block size aligned and identified by their tag. The second is a basic-block cache [4, 12] where blocks are divided on the boundaries of control flow instructions (CTI). A basic-block cache is used by the block-based trace cache proposed in [4]. This is a trace cache [10] that represents traces as a sequence of pointers to entries in a basic-block cache [4].

The experimentation was performed using benchmarks from the SPEC95 and SPEC2000 suites with train or reference inputs. The specific benchmark set used is shown in Table 1. These benchmarks were selected because of the different characteristics they exhibit. The table also shows the number of dynamic instructions skipped and simulated for each benchmark.

Benchmark	Instructions Skipped (Millions)	Instructions Simulated (Millions)
gcc (SPEC95)	0	178
ijpeg (SPEC95)	0	130
vortex (SPEC00)	100	100
gzip (SPEC00)	300	100
ammp (SPEC00)	50	100
equake (SPEC00)	1300	100

Table 1. Benchmarks Simulated

The performance metrics used in this study are miss rates and the duplication rates of each benchmark. The duplication rate refers to the fraction of misses that are for duplicate-blocks. Performance was measured for the following instruction and basic-block cache configurations:

Block Size	Associativity	Total Cache Sizes
32 byte	direct mapped	4KB, 8KB, 16KB, 32KB and 64KB
32 byte	4-way set associative	4KB, 8KB, 16KB, 32KB and 64KB
64 byte	direct mapped	4KB, 8KB, 16KB, 32KB and 64KB
64 byte	4-way set associative	4KB, 8KB, 16KB, 32KB and 64KB

Table 2. Cache Configurations simulated

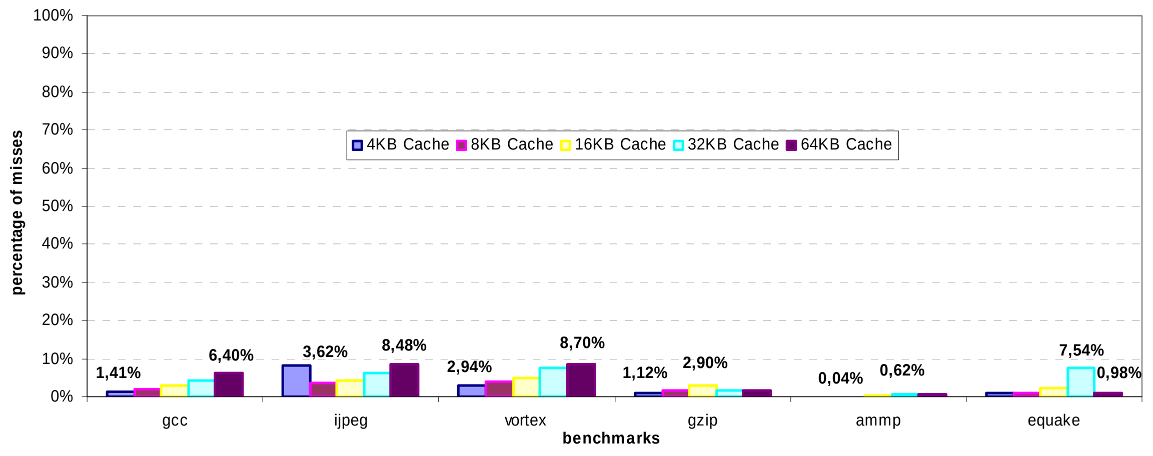


Figure 1. Cache-Content-Duplication for a 4-way set-associative, instruction cache with 32B cache block (% misses for duplicates).

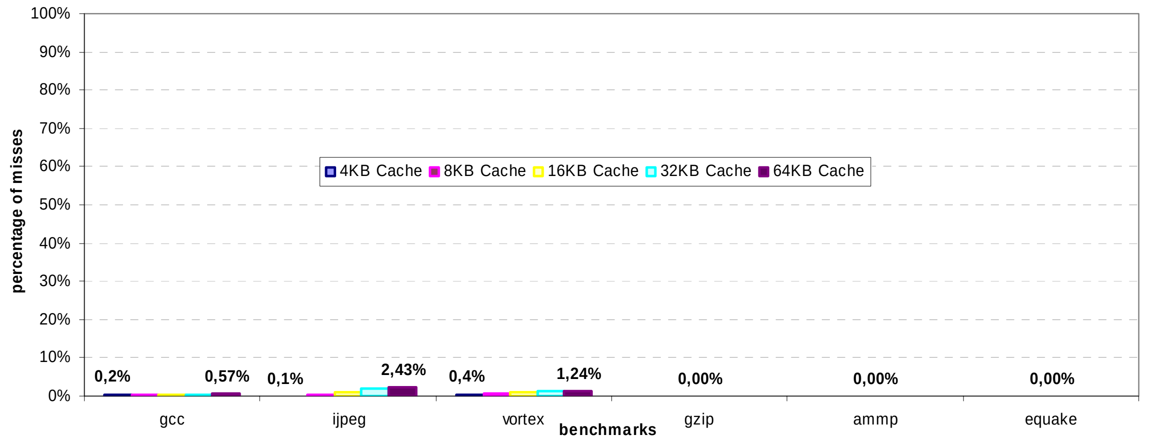


Figure 2. Cache-Content-Duplication for a 4-way set-associative, instruction cache with 64B cache block (% misses for duplicates).

4. Results

In this Section we present measurements that characterize the frequency of the CCD in an instruction cache and a basic-block cache. Two blocks are considered duplicates if each instruction in one block is identical in the exact order with its corresponding instruction in the other block.

4.1 CCD for Instruction Caches

Fig. 1 shows the frequency of CCD with a 4-way, 32B per block instruction cache for various cache sizes. The data show that the phenomenon exists in several benchmarks (*gcc*, *ijpeg* and *vortex*). The general trend with increasing cache size is that the frequency of content duplication, relative to the misses, also increases. This occurs because with a larger cache is more likely for a missed block to already have a duplicate in the cache.

Fig. 2 reports the same results as in Fig. 1 but for a 64B per block. It can be observed that CCD occurs more rarely with larger blocks. This is expected because with increasing block size it is less likely for two static sequences of instructions to be identical.

Benchmarks *ijpeg* and *equake* exhibit a rather distinct behavior in Fig. 1. Specifically, for few cache configurations their CCD frequency is decreasing while the cache size increases. This can happen when duplicated misses observed with smaller cache size are not misses with a larger cache. This is illustrated with the aid of an example shown in Fig. 3.

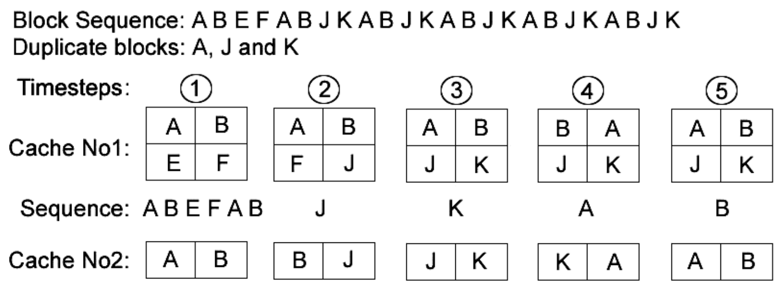


Figure 3. Example where there is Lower Frequency of CCD with a larger Cache.

Fig. 3 uses the following sequence of block references: *A B E F A B J K A B J K A B J K A B J K A B J K* and assumes that blocks *A*, *J* and *K* are duplicates. The example considers two caches, both 2-way set associative with LRU replacement but the one has two sets whereas the other has only one set. For the first cache, blocks *A* and *B* are mapped to the first set and blocks *E*, *F*, *J* and *K* are mapped to the second set. The figure shows the content of the two caches at different times for the same order of block reference. Time 1 shows the content after referencing the sequence *A B E F A B*. After that, blocks *J* (time 2) and *K* (time 3) follow, and they are both counted as duplicate misses in both caches. Then blocks *A* (time 4) and *B* (time 5) arrive. In the large cache they are hits and not counted as duplicates misses whereas in the small cache are misses for which there is already a duplicated block in the cache. If the sequence continuation is a repeated reference to blocks *J K A B*, the large cache will not incur any misses whereas the small cache will incur each time three duplicated misses.

We have also examined the effects of varying associativity on CCD. As shown in Fig. 4 and 5, the frequency and the trends of this phenomenon for a direct mapped cache appear almost the same as with a 4-way set-associative cache (Fig. 1 and 2).

4.2 CCD for Basic-Block Cache

We have already evidence, Section 4.1, that the smaller the block size the higher the CCD frequency. This is due to the smaller number of instructions that are compared and need to match in a block. Furthermore, an instruction cache block may contain instructions that never get executed, i.e. instructions after an always taken control flow instruction (CTI), and this may lead to otherwise identical blocks to appear dissimilar.

Motivated by the above, we measured CCD for a basic-block cache [12]. A basic-block is a sequence of instructions where only the first instruction is an entry and only the last instruction is an exit. Consequently, all instructions in a basic-block get executed as long as we enter the block. The cache block in a basic-block cache contains either an entire basic-block or a partial basic-block when it is larger than a cache block. The expectation is that CCD will be more dominant for basic-block caches as compared to instruction caches. An example that illuminates why CCD may be more frequent in basic-block cache is presented in Table 3.

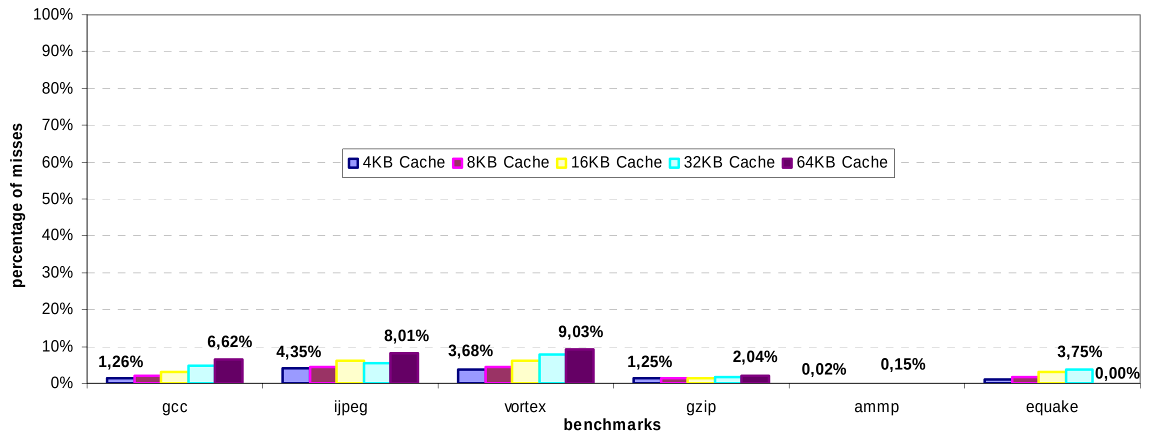


Figure 4. Cache-Content-Duplication for a direct-mapped instruction cache with 32B cache block (% misses for duplicates).

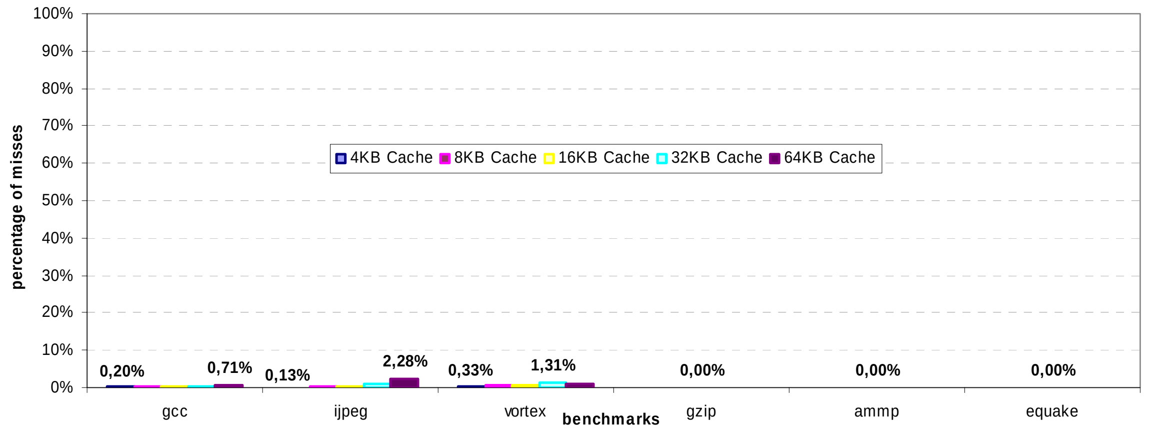


Figure 5. Cache-Content-Duplication for a direct-mapped instruction cache with 64B cache block (% misses for duplicates).

The basic-blocks in Table 3 were found to be duplicates in a simulation of benchmark *vortex* for a basic-block cache with 32B cache block (four instructions per block). Specifically, when BLOCK A produced a miss, a duplicate, BLOCK B, was already resident in the basic-block cache. The first two lines of each block are the contents of a cache block in a basic-block cache. The other two lines correspond to the next static instructions in the code. It can easily be verified that these two blocks correspond to cache blocks, in an instruction cache, since the address of the first instruction is aligned at a block boundary (multiple of 32). Therefore, these two blocks could not have been detected as duplicates in an instruction cache even when both branches are taken.

		BLOCK A	BLOCK B
cache block	basic block	[004a9f20] lw $a2[6],0($s1[17]) [004a9f28] beq $a2[6],$zero[0],20	[004c04c0] lw $a2[6],0($s1[17]) [004c04c8] beq $a2[6],$zero[0],20
		[004a9f30] addu $a0[4],$zero[0],$s2[18] [004a9f38] addu $a1[5],$zero[0],$s3[19]	[004c04d0] lw $a0[4],36($sp[29]) [004c04d8] lw $a1[5],96($sp[29])

Table 3. Identical blocks in a direct mapped basic-block cache with 32B cache block

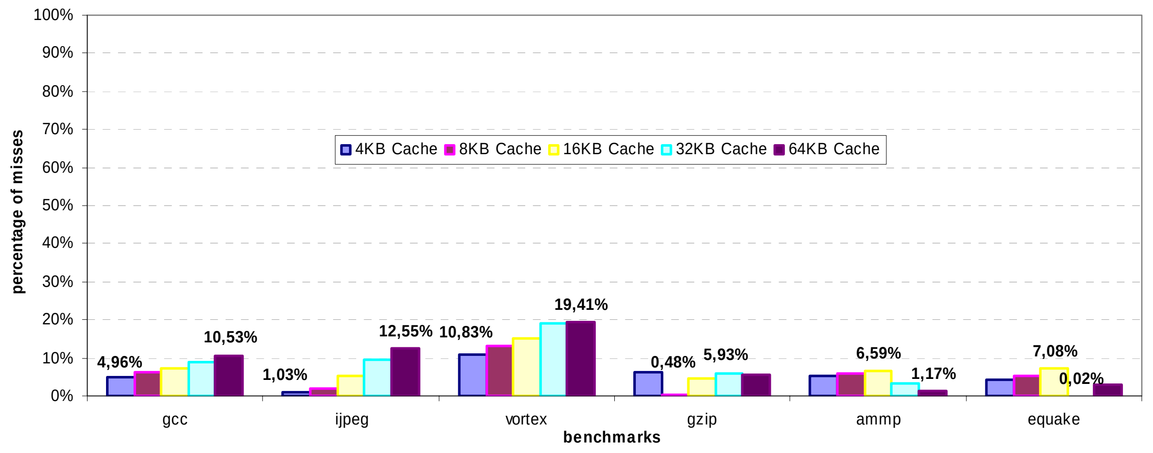


Figure 6. Cache-Content-Duplication for a 4-way set-associative, basic-block cache with 32B cache block (% misses for duplicates).

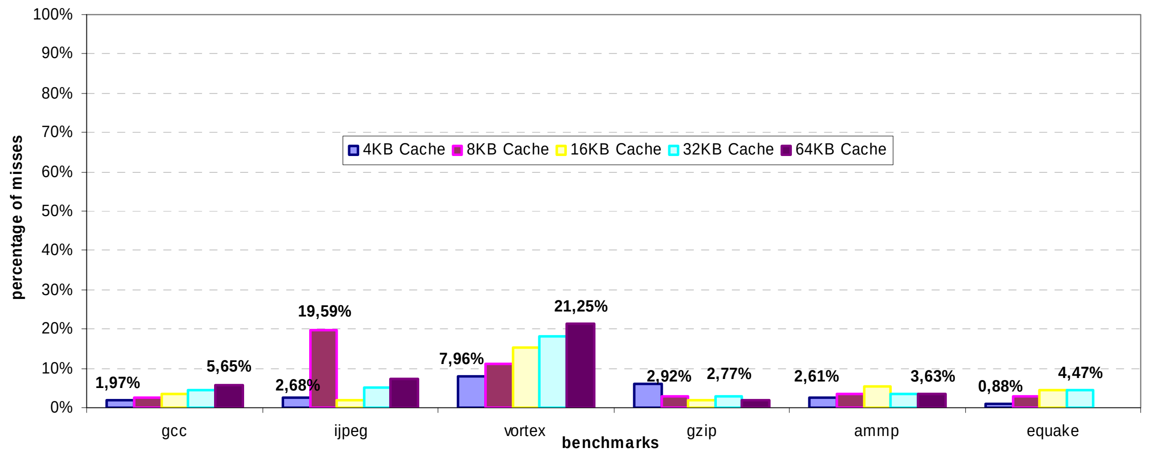


Figure 7. Cache-Content-Duplication for a 4-way set-associative, basic-block cache with 64B cache block (% misses for duplicates).

Fig. 6 and 7 present the CCD frequency in a 4-way set-associative cache with 32B and 64B block sizes. The data show clearly that across all benchmarks CCD is more prevalent with a basic-block cache as compared to an instruction cache (see Fig. 1). For almost all benchmarks the CCD is at least twice as high. The trend with increasing cache size is higher CCD frequency.

One other observation is that with bigger block size the frequency of CCD remains at the same levels. This is contrary to the behavior of an instruction cache where larger blocks meant lower CCD. This mainly occurs because the typical basic-block size is 4-5 instructions. This suggests that for a basic-block cache larger block size may result in block fragmentation and higher miss rates. This was confirmed by the experimental data that show for equal size basic-block caches larger block meant usually higher miss rate.

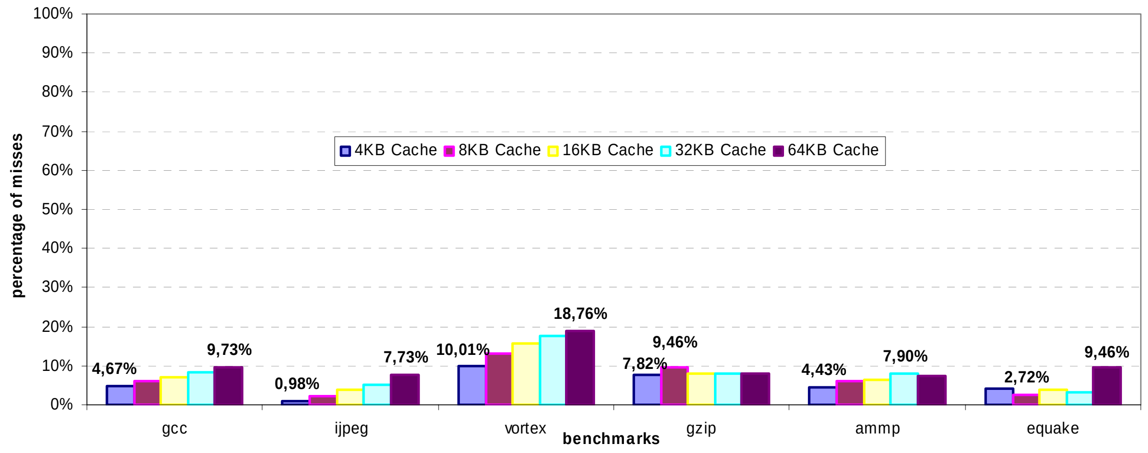


Figure 8. Cache-Content-Duplication for a direct-mapped basic-block cache with 32B cache block (% misses for duplicates).

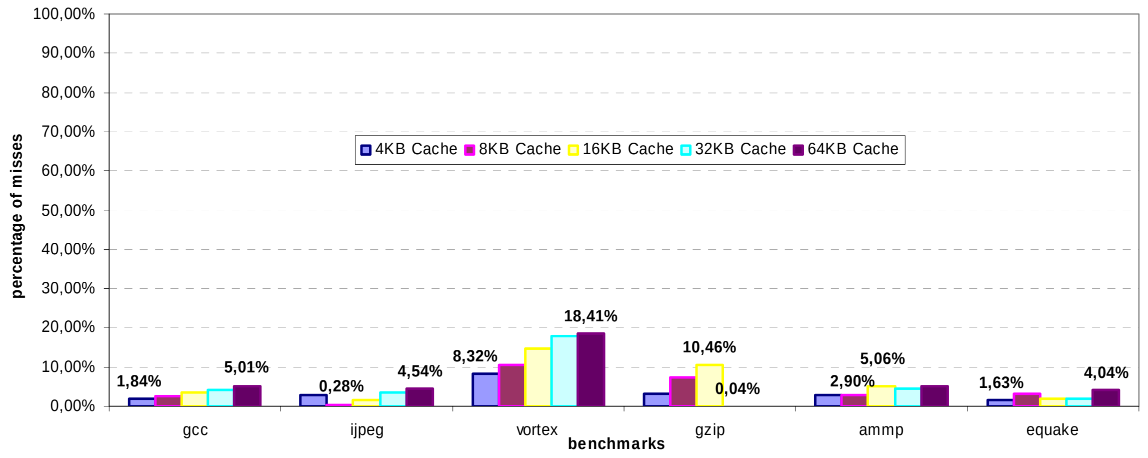


Figure 9. Cache-Content-Duplication for a direct-mapped basic-block cache with 64B cache block (% misses for duplicates).

Fig. 8 and 9 present the same results, as in Fig. 6 and 7, for a direct mapped basic-block cache. The results show that the CCD frequency is similar to the frequency for a 4-way set-associative basic-block cache.

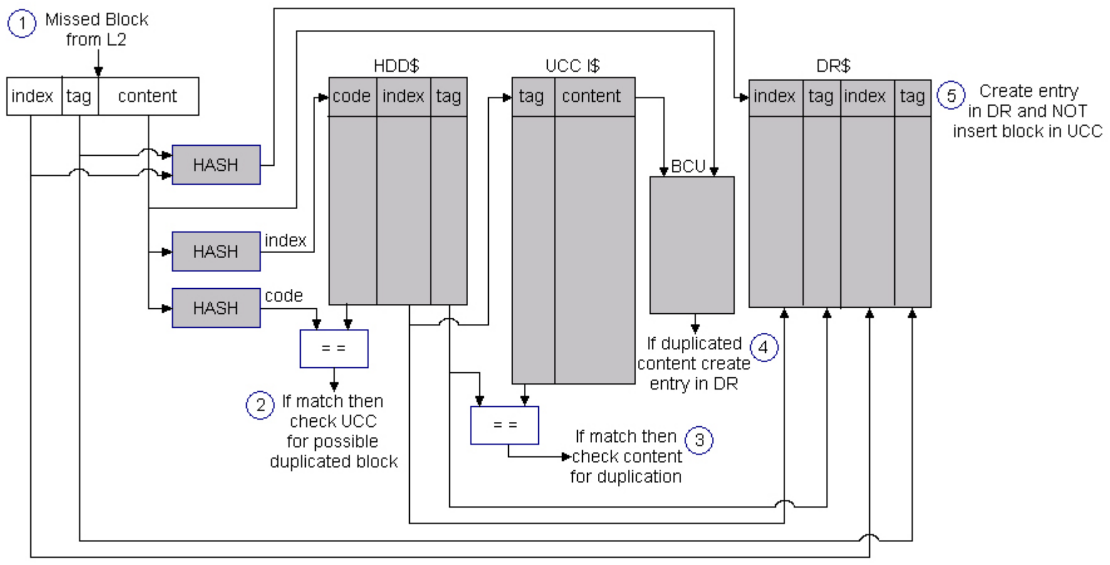
Overall, the experimental results suggest that CCD exists across benchmarks, for different cache types and configurations. The data indicate that with increasing cache size CCD gets more frequent. Finally, CCD is more dominant for basic-block caches, as compared to instruction caches, mainly due to the smaller number of instructions that need to be identical for duplication to occur. We believe that the degree of CCD observed provides a basis to explore mechanisms that can exploit CCD.

5. Exploiting Cache-Content-Duplication

This section describes two possible memory hierarchy enhancements based on CCD that can be useful to improve the performance. The performance potential of one of these mechanisms is investigated experimentally. This section also describes the essential functionality required to detect and use content duplication.

5.1 Memory Hierarchy Enhancements based on CCD

CCD can be exploited to reduce either or both cache latencies and miss rates. Cache latency can be reduced by detecting a miss to a block that has a duplicate already in the



1 Missed Block from L2
index
tag
content
HASH
HASH
index
HASH
code
HDD$
code
index
tag
= =
2 If match then check UCC for possible duplicated block
UCC I$
tag
content
= =
If match then 3 check content for duplication
BCU
If duplicated 4 content create entry in DR
DR$
index
tag
index
tag
5 Create entry in DR and NOT insert block in UCC

Figure 10. Cache enhancement exploiting CCD

cache. We refer to such cache as the *Duplicate-Aware-Cache (DAC)*. Latency can be reduced by fetching the block from the cache instead of reading it from a lower level in the memory hierarchy. CCD can also be used to reduce misses by maintaining in the cache only blocks with unique content. We refer to the latter as the *Unique-Content-Cache (UCC)*. In a UCC one block can have more than one tags.

To facilitate the above CCD based enhancements a mechanism is required that given the tag and index of a block that caused a cache miss it returns whether there is a duplicate in the cache and the tag-index of the duplicated block. This mechanism can be implemented using the *duplicate-relation cache (DR)*. A DR is accessed with a hashed tag-index of a missed block and in each entry contains a tag (the tag-index of a missed block) and another tag-index for its duplicated block. Each valid DR entry indicates a duplicate relation between two blocks. For instruction caches once a duplicated relation is established is assumed to be always correct (in the case of self-modifying code the DR may need to be flashed to ensure correctness).

To create an entry in the DR a mechanism is required to detect duplicated blocks. This mechanism can be implemented using the *Hashed Duplicate Detection cache (HDD)* and the *Block Compare Unit (BCU)*. The HDD is indexed using a hash of the content of a missed block after is fetched from the lower-level of memory hierarchy. Each HDD entry contains a code and a tag-index. The code represents a hash, different from the one used to index the HDD, of a block's content and the block's full tag-index. When the missed block's code and the code in the HDD entry match, this indicates the possibility for content duplication with the block in the HDD entry. In that case, the cache is accessed using the tag-index found in the HDD. If the required block is in the cache, its contents and the contents of the new block are compared using the BCU to detect whether there is duplication. If the BCU finds duplication, an entry is created in the DR. In the case of a cache block miss that also misses in the DR and the HDD, an entry can be allocated in the HDD, initialized with the block's hash content and tag-index, so that future blocks can check if their duplicates with this block.

A DAC and a UCC can use the above mechanism to detect duplicated block misses and read the missed block directly from the cache as long as the duplicated block is in the cache. To better understand the functionalities of the different abovementioned structures we show in Fig. 10 the sequence of steps in the case of a cache miss that has a duplicated in a unique-content cache.

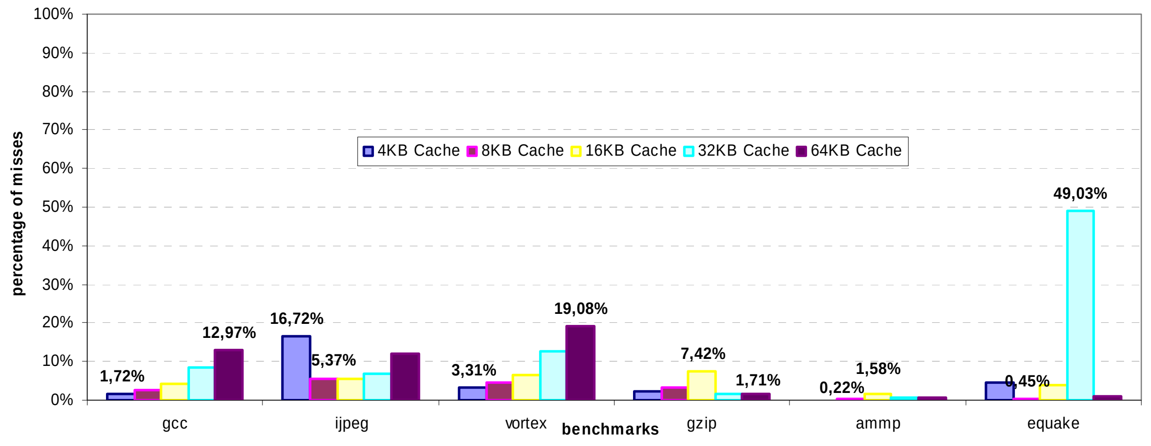


Figure 11. Miss Reduction due to UCC for a 4-way 32B per block instruction cache.

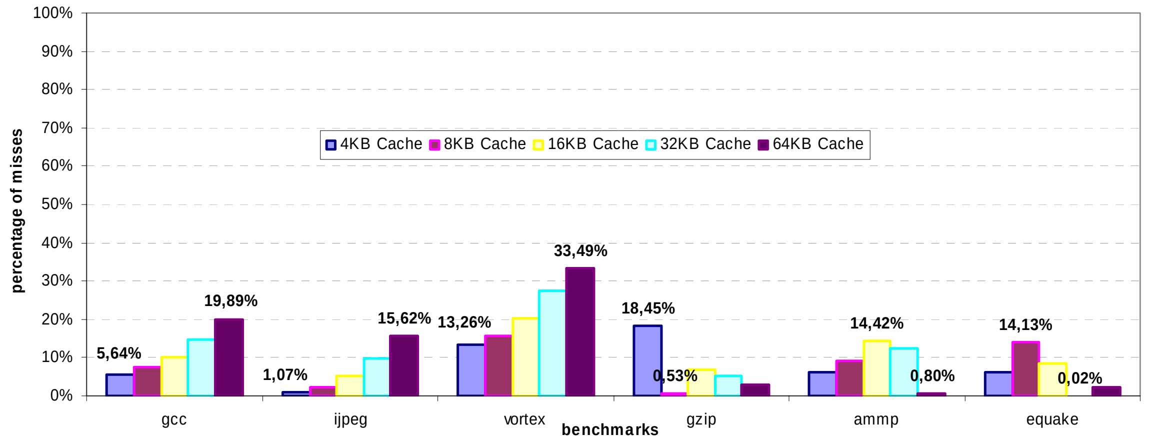


Figure 12. Miss Reduction due to UCC for a 4-way 32B per block basic-block cache.

The above description of the DR and HDD is incomplete since several issues regarding the timing and how the two caches are updated for different scenarios have not been addressed. The intention in this paper is to provide an indicative description about the required functionality and explore the potential assuming an ideal implementation of these mechanisms. Provided, the initial results for exploiting CCD appear promising, future work will examine a more detailed implementation of these mechanisms.

5.2 Exploiting cache-content-duplication using the Unique Content-Cache (UCC)

This Section reports on the performance potential of the UCC. The various mechanisms required to exploit CCD (described in Section 5.1) are assumed to be implemented ideally. Specifically, on a miss if there is a duplicated block in the cache, the duplicate content is always identified and used instead of fetching the missed block from the lower levels of memory hierarchy and inserting it in the cache. Also the LRU of the duplicate block is updated as if we had "a request and a hit" for that block.

Fig. 11 shows the miss rate reduction of a UCC instruction cache over a conventional instruction cache and Fig. 12 shows the miss rate reduction of a UCC basic-block cache over a conventional basic-block cache. More specifically, the data show the percentage of misses that were removed using a UCC as compared to conventional cache implementations that allow block duplication.

The data indicate that the UCC has reduced the number of misses up to 49% and 33% for an instruction cache and basic-block cache respectively. In Fig. 11 the large

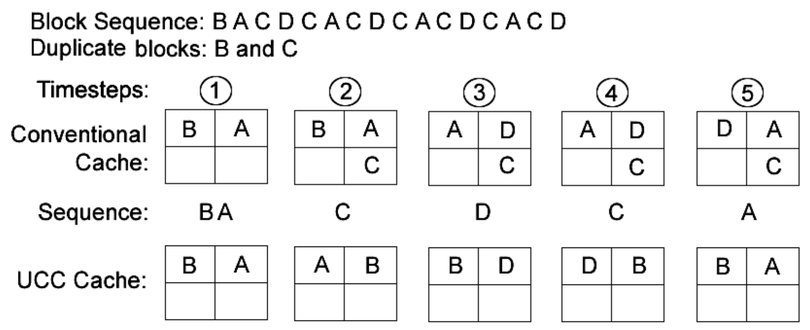


Figure 13. Analysis of Unique-Content Cache Behavior.

fraction of misses removed in *equake* for a 32KB cache is due to the very small number of misses for that cache configuration.

Comparing Fig. 1 with Fig. 11 and Fig. 6 with Fig. 12, it can be observed that, for most of the benchmarks and cache configurations, the fraction of misses removed in the UCC cache is more than the fraction of duplicate misses found in conventional caches. The reason for this is that without getting any duplicate content in the cache a lot of conflict misses are also avoided, consequently, reducing the total number of misses by more than the duplicate misses observed. However, the data also indicate that a UCC may not always removes as many misses as the duplicates found. For example, Fig. 6 shows that for *gzip* with 64KB cache 5,2% of the total misses were for blocks with duplicates but Fig. 12 shows the total misses removed by the UCC were only 2,7%. In Fig. 13 we provide an example that can lead to this behavior.

The example compares the performance of a UCC cache with a conventional instruction cache when both are 2-way set associative cache and contain two sets. Suppose we have the sequence of blocks *B A C D C A C D C A C D C A C D*. Blocks *A*, *B* and *D* are mapped to the first set of the 2 way set-associative cache and block *C* to the second set. Also assume that blocks *C* and *B* are duplicates.

At time 1, Fig. 13 shows the content of the cache after referencing blocks *B* and *A*. Next, the block *C* is requested. The conventional cache will load the block *C* in the second set but the UCC cache will detect that a duplicate of block *C* exists, the block *B*, and will use the content of this block and update the LRU stack as shown at time 2. After that, block *D* is requested. In the conventional cache block *D* will be mapped to the first set replacing block *B*, the least recently used, but in the second cache will replace block *A* (time 3). For the next two requests for block *C* and *A* (time 4 and 5), the conventional cache will have hits and update accordingly the LRU stack. But in the UCC block *C* will cause its duplicate block *B* to become the most recently used and then block *A* will replace *D*, resulting in one more miss.

By repeating the pattern *C D C A*, all references in the conventional cache will hit but in the UCC blocks *D* and *A* will always be replacing each other in the first set. In this example scenario, and other similar situations, the UCC can have more conflict misses than a conventional cache and therefore may have worse performance. Nevertheless, the experimental data do not show this to be a serious problem.

6. Conclusion and Future work

This paper presents a new cache property: the cache-content-duplication (CCD). CCD occurs when there is a miss for a block in a cache for which the data reside already in the cache but under a different tag. This work provides empirical evidence that CCD occurs frequently for instruction caches. Overall, the experimental results suggest that CCD exists across benchmarks, for different cache types and configurations. The data indicate that with increasing cache size CCD gets more frequent. Finally, CCD is more dominant for basic-block caches, as compared to instruction caches, mainly due to the smaller number of instructions that need to be identical for duplication to occur.

CCD considers redundancy at the granularity of blocks and this enables novel optimizations in the memory hierarchy. The paper introduces two such optimizations the duplicate-aware-cache (DAC) and the unique-content-cache (UCC). Experimentally, is shown that a UCC has the potential to reduce the misses that need to be serviced by a lower-level memory by up to 49% and 33% for an instruction cache and a basic-block cache respectively.

This work points to several direction of future research. There is a need to explore the design space for both the duplicate-aware-cache and the unique-content-cache. Furthermore, it is important to investigate the various units used to detect and exploit the content duplication (Section 5.1). Some of the functionality required by these units was discussed but a rigorous investigation was not performed. For example, the function employed by the BCU, to compare instructions of two potentially duplicate blocks, used very strict criteria. For two instructions to be identical the opcode, the destination operand and the source operands must have been be identical and the source operands must be in the same order. For example add r1, r2, r3 will be identified as different from add r1, r3, r2. Furthermore, instructions must appear in the same order in the two blocks. A more advanced compare function could rearrange source operands of commutative operations and reorder data independent instructions in a block to facilitate content duplication. Other transformations to be discovered may help uncover even more duplication. Another important direction of research is to consider CCD for data caches and for different levels in the memory hierarchy. Finally, the proposed mechanisms must be evaluated in the context of a processor timing simulator where the effects of wrong path instructions are accounted for and the actual performance benefits can be measured.

References

[1] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors", Proceedings of the 31st Annual International Symposium on Computer Architecture, Munich, Germany, June 2004, pg. 212-223.
[2] J. Baer and Tien-Fu Chen, "An effective on-chip preloading scheme to reduce data access penalty", Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Albuquerque, New Mexico, United States, 1991, pg. 176-186.
[3] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz and U. Weiser, "Correlated Load-Address Predictors", Proceedings of the 26th Annual International Symposium on Computer Architecture, Atlanta, Georgia, United States, May 1999, pg. 54-63.
[4] B. Black, B. Rychlik and J. P. Shen, "The Block-based Trace Cache", Proceedings of the 26th Annual International Symposium on Computer Architecture, Atlanta, Georgia, United States, May 1999, pg. 196-207.
[5] C. Lefurgy, P. Bird, I-Cheng Chen and T. Mudge, "Improving Code Density Using Compression Techniques", Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997, pg. 194-203.
[6] E. G. Hallnor and S. K. Reinhardt, "A Compressed Memory Hierarchy using an Indirect Index Cache", Technical Report CSE-TR-488-04, University of Michigan, 2004.
[7] C. Lefurgy, E. Piccininni and T. Mudge, "Reducing Code Size with Run-time Decompression", Proceedings of the 6th International Symposium on High-Performance Computer Architecture, Toulouse, France, 2000, pg. 218-227.

[8] Chi-Keung Luk and T. C. Mowry, "Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors", Proceedings of the 31st Annual International Symposium on Microarchitecture, Dallas, Texas, United States, 1998, pg. 182-194.
[9] M. A. Postiff and T. Mudge, "Smart Register File for High-Performance Microprocessors", University of Michigan CSE Technical Report CSE-TR-403-99, June 1999.
[10] E. Rotenberg, S. Bennett and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation", IEEE Transactions on Computers, February 1999, 48(2):111-120.
[11] A. Jay Smith, "Cache Memories", Computing Surveys, 1982, 14(3):473-530.
[12] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen, "Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA", Proceedings of the 2001 International Symposium on Low Power Electronics and Design, Huntington Beach, California, United States, 2001, pg. 4-9.
[13] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious", ACM SIGARCH Computer Architecture News, 1995, 23(1):20-24.

Appendix: Misses per 1000 Instructions

Benchmark	Conventional Cache with 32B cache line									
	Direct Mapped					4 way set-associative				
	4KB	8KB	16KB	32KB	64KB	4KB	8KB	16KB	32KB	64KB
gcc (SPEC95)	136,85	107,71	80,63	51,04	30,97	128,04	97,82	66,57	32,43	10,02
ijpeg (SPEC95)	9,50	6,82	4,55	1,98	0,81	10,90	3,18	1,36	1,01	0,82
vortex (SPEC00)	202,60	135,49	98,44	64,09	38,22	168,73	125,04	82,34	48,95	16,46
gzip (SPEC00)	31,12	31,10	31,09	0,01	0,01	0,06	0,026	0,01	0,00	0,00
ammp (SPEC00)	63,15	31,78	11,82	0,29	0,10	61,01	22,25	2,51	0,01	0,01
equake (SPEC00)	162,09	93,77	32,89	21,78	3,50	166,47	52,76	7,09	0,50	0,01

	Basic-block Cache with 32B cache line									
	Direct Mapped					4 way set-associative				
	4KB	8KB	16KB	32KB	64KB	4KB	8KB	16KB	32KB	64KB
gcc (SPEC95)	161,81	129,30	98,56	66,33	41,37	149,80	118,88	84,79	45,46	17,82
ijpeg (SPEC95)	52,76	13,24	5,15	3,56	2,25	78,38	12,93	2,84	1,37	1,10
vortex (SPEC00)	206,25	166,37	117,54	80,61	48,13	188,83	137,36	94,65	58,21	24,75
gzip (SPEC00)	35,89	29,64	2,97	2,95	2,94	6,68	0,52	0,02	0,01	0,01
ammp (SPEC00)	70,79	45,28	23,38	9,86	3,99	72,54	40,71	8,93	0,80	0,03
equake (SPEC00)	176,64	119,50	65,83	27,67	7,26	189,08	86,27	14,24	2,54	0,02

	Conventional Cache with 64B cache line									
	Direct Mapped					4 way set-associative				
	4KB	8KB	16KB	32KB	64KB	4KB	8KB	16KB	32KB	64KB
gcc (SPEC95)	87,54	69,75	53,22	34,68	20,87	81,81	62,95	45,06	23,94	7,46
ijpeg (SPEC95)	5,57	3,98	2,62	1,19	0,50	6,25	1,93	0,85	0,59	0,49
vortex (SPEC00)	123,85	84,72	61,99	41,09	24,77	105,92	77,61	51,47	31,60	11,66
gzip (SPEC00)	19,27	19,26	19,25	0,01	0,01	0,04	0,01	0,01	0,00	0,00
ammp (SPEC00)	37,49	21,03	7,90	0,26	0,07	37,57	16,50	2,27	0,01	0,00
equake (SPEC00)	99,95	61,10	23,17	15,71	2,10	106,60	37,94	7,51	0,43	0,00

	Basic-block Cache with 64B cache line									
	Direct Mapped					4 way set-associative				
	4KB	8KB	16KB	32KB	64KB	4KB	8KB	16KB	32KB	64KB
gcc (SPEC95)	132,13	109,82	85,72	62,10	41,06	123,83	100,19	77,24	50,97	24,61
ijpeg (SPEC95)	56,03	26,52	4,41	2,25	1,66	52,86	40,76	3,18	1,04	0,80
vortex (SPEC00)	162,05	130,31	104,15	70,79	44,56	150,52	113,57	82,01	53,41	28,26
gzip (SPEC00)	46,25	20,22	14,47	1,30	1,29	24,32	2,04	0,02	0,01	0,00
ammp (SPEC00)	53,02	40,86	25,64	12,97	6,44	54,50	40,55	17,86	2,15	0,20
equake (SPEC00)	158,39	120,70	74,69	41,07	18,61	160,35	112,75	43,75	3,96	0,26