INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Zero-Content Augmented Caches

Julien Dusser — Thomas Piquet — André Seznec

N° 6705

Octobre 2008

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6705--FR+ENG

Zero-Content Augmented Caches

Julien Dusser , Thomas Piquet , André Seznec

Thème COM — Systèmes communicants
Équipe-Projet CAPS

Rapport de recherche n° 6705 — Octobre 2008 — 19 pages

Abstract: It has been observed that some applications manipulate large amounts of null data. Moreover these zero data often exhibit high spatial locality. On some applications more than 20% of the data accesses concern null data blocks. Representing a null block in a cache on a standard cache line is clearly a waste of resources.

In this paper, we propose the Zero-Content Augmented cache, the ZCA cache. A ZCA cache consists of a conventional cache augmented with a specialized cache for memorizing null blocks, the Zero-Content cache or ZC cache. In the ZC cache, the data block is represented by its address tag and a validity bit. Moreover, as null blocks generally exhibit high spatial locality, several null blocks can be associated with a single address tag in the ZC cache.

For instance, a ZC cache mapping 32MB of zero 64-byte lines uses less than 80KB of storage. Decompression of a null block is very simple, therefore read access time on the ZCA cache is in the same range as on a conventional cache. On applications manipulating large amount of null data blocks, such a ZC cache allows to reduce up to 81% the miss rate and memory traffic, and therefore to increase performance for a small hardware overhead. In particular, the write-back traffic on null blocks is limited. For applications with a low null block rate no performance loss is observed.

Key-words: super-scalar processor, memory hierarchy, cache compression, null data, zero

Caches de zéros

Résumé : Plusieurs études ont observé que les applications manipulent de grandes quantités de données nulles. Ces blocs de zéros ont une forte localité spaciale. Pour certaines applications plus de 20% des accès à des données concernent des blocs nuls. Stocker ces blocs nuls dans des lignes de cache traditionnelles est donc une perte de ressources.

Cette étude propose de gérer plus efficacement la hiérarchie mémoire en ajoutant au cache traditionnel un cache dédié au blocs nuls. Ce cache exploite la forte localité spaciale de ces blocs de zero.

Mots-clés : processeur superscalaire, hiérarchie mémoire, compression de cache, données nulles, zéro

1 Introduction

It has been observed that some applications manipulate large amounts of null data. Ekman and Stenstrom [7] showed that on many applications many data are null in memory and that in many cases, complete 64-bytes blocks are null. This study was performed through dumping the memory content. For SPEC2000 benchmarks, they report that 30% of 64-bytes memory blocks are only zeros with some benchmarks such as *gcc* exhibiting up to 75% of null blocks in memory. While these results stand for static blocks in memory, our experiments further show that on some applications more than 20 % of dynamic accesses to data are accesses to null 64-bytes data blocks. Moreover these zero data often exhibit high spatial locality. Resources are wasted in representing null block data on a standard cache line.

Null blocks could be represented in an adjunct cache accessed in parallel with the cache as was suggested for frequently used values for the Frequent Value Cache in [24]. A null block would be represented by its address tag and a single validity bit. In such an adjunct cache, the address tag would constitute the major storage cost. However, the spatial locality of null blocks can be leveraged. The Zero Content Augmented cache, ZCA cache (Fig. 4) presented in this paper associates a conventional cache with a zero-content cache, ZC cache. The ZC cache only stores null blocks. A ZC cache entry consists of an address tag and N validity bits. Therefore a single ZC cache entry can map up to N null blocks. The ZC cache is accessed in parallel with the cache. The ZC cache can represent a large number of null blocks at a very limited storage cost. For instance, if block size is 64 bytes, the null blocks in an 8 KBytes page can be represented with a single address tag and 128 validity bits: a 4096-entry ZC cache can map up to 32 MBytes of null blocks and uses only 78 KBytes of storage. While using more general compressed caches has been considered in several previous studies [24], the ZCA cache features a very simple compression/decompression hardware. Compression just requires a tree of OR gates for detecting a null block. Decompression does not induce extra access latency.

On applications manipulating large amounts of null data blocks, the ZC cache allows to reduce the miss rate on the main cache and on the memory traffic. Moreover as a side-effect, some write-back traffic is suppressed: null blocks are often overwritten with null data, the ZC cache captures this situation and avoids writing back these blocks.

The remainder of the paper is organized as follows. Section 2 analyzes the occurrences of accesses to null data blocks through the whole memory hierarchy. Section 3 presents the architecture of the ZCA cache. In Section 4, we present our experimental framework. Section 5 presents the performance evaluation of the ZCA cache. Section 6 discusses related work. Section 7 shows that ZCA functionalities can be added to a decoupled sectored cache at a very limited hardware cost. Section 8 concludes this study.

2 Accesses to Null Data Blocks in Applications

Storing a null memory block in the memory hierarchy can be seen as a waste of cache space. Our study focuses on storing these null blocks in a compressed form. Such a mechanism can be justified only if the accesses to null blocks

Application	DL1		L2		L3		Memory	
	NAPKI	APKI	NAPKI	APKI	NAPKI	APKI	NAPKI	APKI
gzip	2.79	235	0.44	12.27	0.34	1.20	0.32	0.85
wupwise	7.75	274	0.17	4.52	0.16	4.22	0.15	4.11
swim	0.06	429	0.06	50.72	0.06	44.25	0.06	34.04
mgrid	9.82	430	1.05	10.86	1.05	10.21	1.02	7.35
applu	0.39	345	0.07	14.16	0.07	13.68	0.07	13.67
vpr	2.74	268	1.55	21.00	0.54	11.42	0.01	1.13
gcc	107.11	417	9.54	28.63	6.38	20.26	0.52	1.16
mesa	10.19	333	0.58	2.58	0.58	1.23	0.58	1.15
art	0.63	274	0.53	145.13	0.53	145.10	0.53	87.15
mcf	0.06	509	0.03	153.96	0.03	129.19	0.03	102.37
equake	9.55	390	1.93	23.37	1.92	22.22	1.92	22.09
crafty	1.36	286	0.06	11.98	0.00	0.48	0.00	0.05
ammp	0.52	369	0.07	23.18	0.07	16.85	0.07	6.27
parser	4.79	297	0.59	13.09	0.44	5.73	0.14	2.32
sixtrack	25.15	228	0.21	1.34	0.17	0.81	0.07	0.25
bzip2	2.19	294	0.28	10.90	0.25	3.38	0.09	0.32
twolf	0.01	341	0.00	33.92	0.00	22.11	0.00	5.46
apsi	17.02	301	1.11	14.42	0.84	8.60	0.45	3.39

Table 1: Null block Access Per Kilo-Instruction (NAPKI) and Access Per Kilo-Instruction (APKI) on a 32KB L1 data cache, a 256KB L2 and a 1MB L3 cache with 64B blocks.

represent a significant part of cache accesses. In this section, we first analyze quantitatively the occurrences of accesses to null data blocks in applications showing that some applications exhibit a quite significant amount of access to null data blocks. Then, for two applications, we analyze how null data blocks are used all along the execution.

2.1 Quantifying accesses to null blocks

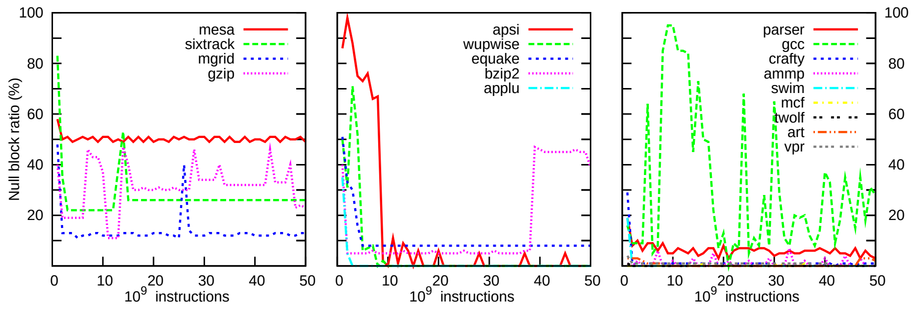

Figure 1: Null block ratio during execution from beginning to 50.10^9 instructions.

Table 1 represents the dynamic occurrences of accesses to null blocks in the different levels of a memory hierarchy for the first 50 billions instructions on SPEC2000 CPU benchmarks, thus eliminating initialization phase effects. We represent the frequency of accesses to null blocks, both misses and write-backs on a three-level memory hierarchy. Access per kilo-instructions (APKI) features previous level misses and writebacks. Our experimental framework is further described in Section 4.

We observe that most of the SPEC CPU 2000 applications manipulate some null data blocks, but in very different proportions. In particular, for some applications e.g. *mesa, gcc* and *mgrid*, more than 20% of the accesses flowing out up to the main memory concerned null data blocks.

From Table 1, we can infer that avoiding traffic on null data blocks, particularly on the main memory, may help to improve performance on many applications. On our benchmark set, one can expect some performance gain on *gzip, wupwise, mgrid, gcc, equake, parser, bzip2* and *apsi.*

Figure 1 illustrates the proportion of accesses to null data blocks on the main memory over 50 billions of instructions, each point representing an interval of a billion instructions. This figure shows that while some applications (*wupwise* and *apsi*) essentially manipulate null blocks during their initialization phases, other applications manipulate significant numbers of null blocks over the whole execution, e.g. *mesa, gcc, sixtrack* and *mgrid.*

2.2 Null Block Usage Analysis

We analyze the use of a large ratio of null data blocks on two application examples, *mesa* and *gcc*.

```
static void Render( int frames, [...] )
{
   [...]
   for (i=0; i<frames; i++) {
      [...]
      glClear([...]);

      glPushMatrix();
      glRotatef(−Xrot, 1, 0, 0);
      glRotatef(Yrot, 0, 1, 0);
      glRotatef(−90, 1, 0, 0);
      SPECWriteIntermediateImage(fip, fop, width, height, buffer, i);
      DrawMesh();
      glPopMatrix();
      Yrot += 5.0F;
   }
}
```

Figure 2: A code section manipulating large number of null blocks in *mesa.*

Figure 2 illustrates a code section in *mesa.* Null blocks are manipulated all along the execution of the function *Render*. Function *glClear* sets a whole buffer of 5MB ($1280 * 1024 * 4$) to the default color which is .0. This generates lot of writes of null blocks on the main memory. Then these null blocks are read back and modified. This sequence is repeated for each frame.

gcc also manipulates a large amount of null blocks. The use of null data blocks varies during the execution. During flow analysis and instruction scheduling, some data structures are initialized to zero. As an example, during instruction scheduling, function *schedule_block()* illustrated in Figure 3 is called for each basic block. In this function, large structures are associated with each pseudo-

```
static void schedule_block (b, file)
     int b;
     FILE *file;
{
  [...]
  i = max_reg_num ();
  reg_last_uses = (rtx *) alloca (i * sizeof (rtx));
  bzero ((char *) reg_last_uses, i * sizeof (rtx));
  reg_last_sets = (rtx *) alloca (i * sizeof (rtx));
  bzero ((char *) reg_last_sets, i * sizeof (rtx));
  reg_pending_sets = (regset) alloca ((size_t)regset_bytes);
  bzero ((char *) reg_pending_sets, regset_bytes);
  reg_pending_sets_all = 0;
  clear_units ();
  [...]
}
```

Figure 3: A code section manipulating large number of null blocks in *gcc*.

register. These structures are initialized to zero using *bzero* at the beginning of the instruction scheduling process.

3 The Zero-Content Augmented Cache

In Section 2, we have pointed out that for some applications a quite significant proportion of the memory accesses are performed on null data blocks. This phenomenon already exists for accesses on the L1 data cache and is even more pronounced for access flowing down through the memory hierarchy L2, L3 and main memory. Moreover, as it will be illustrated in the experimental Section 5.4, zero data often exhibit a quite high spatial locality.

In this section, we present the Zero-Content Augmented Cache which leverages these two properties.

3.1 Overview of the Zero-Content Augmented Cache

A Zero Content Augmented cache is represented on Figure 4. It consists of a conventional cache, the main cache, augmented with the Zero Content cache or ZC cache, a specialized cache for storing null blocks. Each ZC cache entry consists of an address tag which allows rebuilding the address of an N-block sector and N validity bits associated to the N blocks in the sector. A block is present in the ZC cache if its englobing sector is represented in the cache and if its associated validity bit is set.

The nullity of a block is checked on the two update paths of the cache coming down from the processor or the upper level cache on a write and up from the lower memory hierarchy level on a miss. The zero detectors (Figure 4) perform a global OR on the value to be stored in the cache.

Read scenario: On a read on the ZCA cache, both the main cache and the ZC cache are checked in parallel. On a hit on the main cache, a conventional cache read is executed. On a hit in the ZC cache, zero data are propagated.

On a miss on both the main and ZC caches, the missing block is retrieved from a lower level in the memory hierarchy. As ZC cache miss information is needed to propagate the miss in the memory hierarchy, the ZC cache latency should be less or equal to the main cache latency. When the missing block comes back on the miss path, a zero-detector is used to determine whether the missed block should be allocated on the main cache or on the ZC cache.

Notice that while the overall block is needed before deciding whether the block should be allocated on the ZC cache or on the main cache, zero-detection does not induce extra latency for processor use since the data words can be directly bypassed to the processor as soon as they arrived from the main memory or intermediate memory hierarchy levels.

Write and coherency scenarios: On a write on the ZCA cache, a zero-detector is also used. Non-null writes hitting on the main cache as well as null writes hitting on the ZC cache do not require any special attention.

When a non-null write hits on the ZC cache then the block must be invalidated in the ZC cache. Different scenarios can be considered. For instance in the experiments illustrating this paper, the block is allocated in the main cache and then modified. An alternative scenario would be to forward the block directly to the next memory hierarchy level.

When a null write hits on the main cache, the block is maintained in the main cache since the block will have to be written back to the next memory hierarchy level. This allows implementing the write-back protocol without adding dirty state to each block on the ZC cache: null blocks hitting on the ZC cache have never to be written back.

ZCA caches support the multiprocessor coherency protocols without any extra coherency state bits: the ZC cache is never the owner of a modified copy of the block. When the local processor is writing a non-null data in a shared memory space, one has to acquire ownership of the block: if the block was already present in the ZC cache, then one has to invalidate it. When a remote

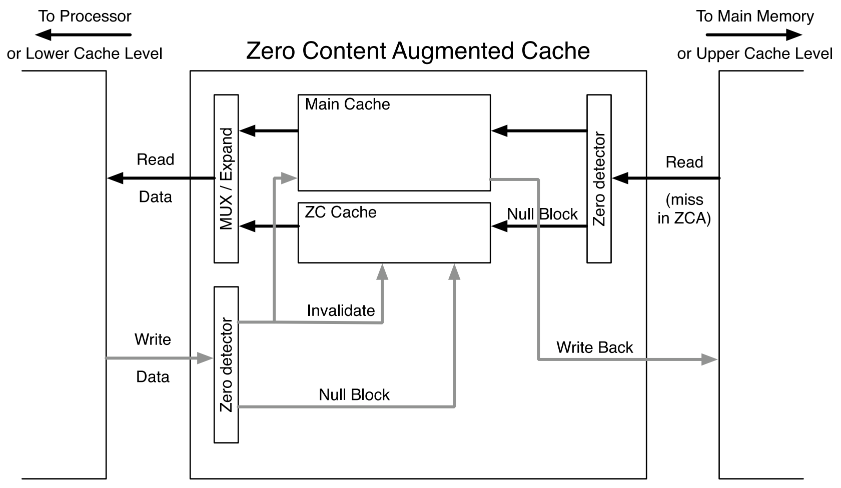

Figure 4: The ZCA Cache

processor is writing a non-null data in a shared memory space, the block has to be invalidated in the ZCA cache.

Optionally, when a dirty block is evicted from the main cache and is found as a null block, in parallel with updating the memory, one can create a copy of the block in the ZC cache, thus potentially saving an extra miss on the next access to the block. This option requires implementing an extra zero-detector on the write-back path to the memory.

3.2 Storage Complexity Evaluation

The size of the sector on the ZCA cache should not exceed the size of the physical page of the system. Therefore we will only consider sector sizes lower or equal than 8KB.

The storage cost of an entry in the ZC cache consists of the address tag and $N = 2^n$ validity bits [1]. Let A, $S = 2^s$ respectively be the associativity and the number of sets of the ZC cache, let $B = 2^b$ be the size of a block in the cache, and P the number of bits in the physical address. Each entry in the ZC cache feature $N + (P - s - n - b)$ bits, and therefore the storage volume of the ZC cache is $A * S * (N + (P - s - n - b))$ bits while it may represent up to A*S*N*B bytes of memory space.

Sector size (KB)	size of the mapped memory in MBytes							
	1	2	4	8	16	32	64	128
2	32	63	124	244	484	964	1924	3944
4	24	47.5	94	186	368	728	1440	2848
8	20	39.75	79	157	312	620	1232	2448

Table 2: Storage budget (in Kbits) of the ZC cache for various sectors sizes and sizes of mapped memory

Table 2 illustrates the total storage volume of 4-way set-associative ZC cache considering sectors from 2KB to 8KB and mapping from 1 to 128 MBytes of memory space [2]. A 50 bits physical address space and a 64 bytes block size are considered. This table clearly shows that when 4KB or 8KB sectors one can represent all the null blocks in a very large memory space with a limited storage requirement.

For instance, considering 8KB sectors, one can represent 4 MBytes of with less than 10KBytes of memorization or 32 MBytes with less than 78 KBytes.

3.3 Energy Consumption Issues

As it will be illustrated in Section 5, the ZCA cache is efficient on some applications, but is of poor help on other applications. On applications featuring a very limited number of accesses to null blocks, the parallel access to the ZC cache is a waste of power, since the ZC cache is checked on each cache access, but provides no performance improvement. This waste of energy can be avoided

[1]For simplicity, we will ignore the LRU tags used for managing the replacement policy

[2]Using a sector size larger than the physical page size does not make sense since the virtual to physical address translation would break the spatial locality of null data blocks

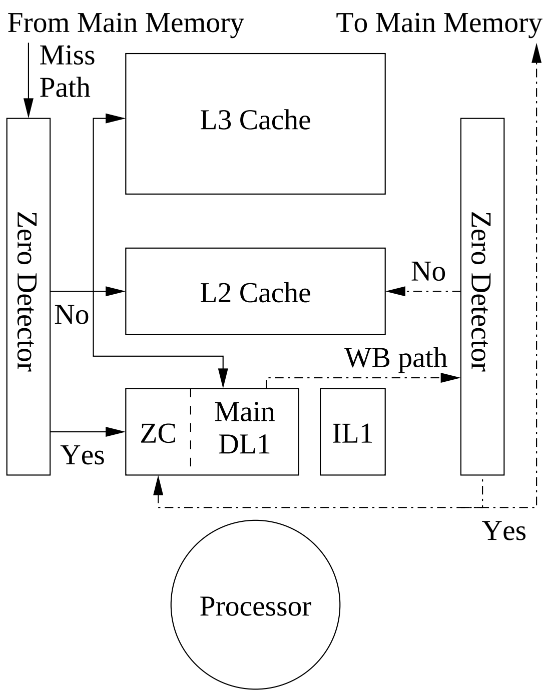

Figure 5: A L1 ZCA cache

through a simple monitoring of the ratio of null blocks traffic from the memory; when the ratio of null blocks is lower than a threshold, the ZC cache can be disabled without any write-back.

3.4 ZCA Cache and the Memory Hierarchy

Many design options can be considered. In particular, one may use a complete hierarchy of ZCA caches using increasing sizes. Another option is to use only a ZCA cache at a particular memory hierarchy level, e.g. the L1 data cache or the L3 cache.

ZCA cache as the L1 data cache For instance, in Section 5, we will consider the hierarchy illustrated on Figure 5. On a L3 miss, when tested null a data blocks flowing out from the memory are not stored in the L3 neither in the L2 cache, but only on the ZC cache in the L1 cache. On a write back, the block is tested when it is null the block is directly written back on memory and allocated on the ZC cache unless the write back is forced by an invalidation.

As pointed above in Section 3.2, the silicon area occupied by a ZC cache is limited, a possible design might then be to implement a ZCA cache as the L1 data cache. The global read access time of the ZCA cache will be slightly longer than the maximum of the access times to the main cache and to ZC cache, since it features an extra multiplexor. In a realistic design of a L1 ZCA cache, the access time of the ZC cache has to be in the same order as the access time of the main cache. In L1 caches, the access time is dominated by the tag path.

Therefore the ZC cache should feature the same number or fewer tags than the normal cache. The configuration evaluated in Section 5 features a 32KB 4-way associative main cache and a 128-entry 8KB sector 4-way ZC cache. The ZC cache occupies only 2.5KB of storage area, and can map up to 1MB of null blocks. The overall cache –ZC cache + main cache– is 8-way associative. A very rough estimation of the access time of the ZCA cache would be the access time of a 64KB 8-way set-associative cache since the tag path delay is longer than the data path delay. We used CACTI 4.2 [20] to get such an evaluation of the access time for 45nm technology. Access time for a 64KB 8-way associative cache (i.e., very close to our ZCA cache) and access time to a 32KB 4-way associative cache (i.e., our main cache) are in the same range: 483ps against 451ps.

ZCA cache as the L2 cache Using a ZCA cache as a L2 cache allows using a ZC cache able to map a larger memory area. This is a compromise between a fast and a large ZCA cache. For instance, the configuration evaluated in section 5 features a 256KB 4-way set-associative main cache and a 1024-entry 8KB sector 4-way ZC cache. The ZC cache occupies less than 20KB of storage area, and can map up to 8MB of null blocks.

ZCA cache as the L3 cache Using a ZCA cache as a L3 cache allows mapping a very large memory area. The configuration evaluated in Section 5 features a 1MB 8-way set-associative main cache and a 4096-entry 8KB sector, 4-way ZC cache. The ZC cache occupies only 78KB of storage area, and can map up to 32MB of null blocks.

On the ZCA cache, the main cache and the ZC cache are accessed in parallel. The complexity of this access is in the same range as the complexity of the tag path in the main L3 cache. However, many L3 cache designs implement sequential access to tags and data in order to reduce power consumption. The hit time on a null block in the L3 cache can therefore be shorter than the hit time on a non null block.

A hierarchy of ZCA caches On a complex memory hierarchy, implementing ZCA caches at every level is possible. Such a hierarchy allows a design optimization that may significantly reduce the miss ratio on the faster cache. For instance, let us consider that ZCA caches are implemented at L1 and L2 cache levels. Let us consider a miss on a null block on the L1 cache. In case of a hit on the L2 ZC cache, an entry is allocated on the L1 ZC cache and the validity bit of the missing block is set, however a simple optimization consists in copying the entire hitting entry of the L2 ZC cache in the L1 ZC cache. This corresponds to prefetching all the null blocks of the sector in the L1 ZC cache. This potentially limits the number of misses on the L1 cache. Such a prefetching of the null blocks could be handled through using the usual data bus from L2 to L1 cache: for instance considering an 8KB sector and a 64B block, the ZC entry is only 16 bytes wide.

However, our evaluation will show in Section 5 that for most applications implementing ZCA caches in the whole memory hierarchy is not warranted.

3.5 Replacement Policy

A sector is allocated in the ZC cache when a null block is accessed. However, the block can be overwritten with non null data; its validity bit in the ZC cache is then reset. Such situations may lead to sectors present in the ZC cache without any valid null blocks. In our replacement policy, we consider these sectors as invalid.

4 Experimental Framework

4.1 Simulation Setup

Parameter		**Configuration**
Decode, Issues, width		4
Retire width		5
ROB size		26 Issue + 48 entries
LSQ size		10 Issue + 40 entries
Branch predictor		O-GEHL [18], 64Kbits, 6-cycles mispred. penalty
L1 inst.		64KB, direct-map, 64B/block, 1-cycle
L1 ZCA	data	32KB, 4-way, 64B/block, LRU, 1-cycle, WB
	ZC (Optional)	128-entry, 8KB/sector 4-way, LRU, 1MB mapped in 2.5KB
L2 ZCA unified	data	256KB, 4-way, 64B/block, LRU, 11-cycles, WB
	ZC (Optional)	1024-entry, 8KB/sector 4-way, LRU, 8MB mapped in 20KB
L3 ZCA unified	data	1MB, 8-way, 64B/block, LRU, 30-cycles, 16B/cycle, WB
	ZC (Optional)	4096-entry, 8KB/sector 4-way, LRU, 32MB mapped in 78KB
Main Memory		500-cycles, 16B/cycle

Table 3: Simulated machine parameters.

Our experiments were performed on SESC, an execution-driven simulator [15]. Our baseline processor is a 4-way out-of-order superscalar architecture. Table 3 summarizes the configuration we used as a reference.

4.2 Benchmarks

We evaluate our proposal on the subset of SPEC 2000 benchmarks that run on SESC: *gzip, wupwise, swim, mgrid, applu, vpr, gcc, mesa, art, mcf, equake, crafty, ammp, parser, sixtrack, bzip2, twolf, apsi*. All applications were compiled for the MIPS ISA with the `-O3` optimization flag enabled. We used the reference data as input. The applications are simulated for 50 billions instructions. Note that for *gzip, gcc* and *bzip2*, 50 billion instructions correspond to the use of several of the input files.

5 Performance Evaluation of the ZCA Cache

In this section, we first evaluate the performance of our reference ZCA L3 cache in terms of hit/miss ratios, overall performance improvement and memory traffic reduction. Then we analyze the position of ZCA in the memory hierarchy, and the usage of ZC cache. Finally we measure the performance on a dual-core CMP architecture featuring a shared ZCA L3 cache.

5.1 Using a L3 ZCA Cache

Appli.	Null miss %	base MPKI	base IPC	MPKI red. %	IPC imp. %	mem WB red.	mem traffic red.
gzip	38	0.54	0.72	4	1	7	5
wupwise	4	3.25	0.81	2	1	3	2
swim	0	24.9	0.28	0	0	0	0
mgrid	14	5.13	0.48	17	22	1	12
applu	1	9.43	0.51	0	0	0	0
vpr	1	0.80	0.79	16	4	12	15
gcc	45	0.83	0.84	27	7	63	37
mesa	50	0.59	1.22	51	15	40	46
art	1	80.7	0.28	1	2	4	1
mcf	0	79.7	0.04	0	0	0	0
equake	9	19.6	0.24	8	2	1	7
crafty	2	0.03	1.33	4	0	4	4
ammp	1	5.35	0.39	1	0	3	1
parser	6	1.45	0.60	6	2	4	5
sixtrack	30	0.24	1.44	81	7	77	81
bzip2	28	0.19	1.09	4	0	7	5
twolf	0	3.36	0.43	0	0	0	0
apsi	13	2.39	1.13	2	0	14	6

Table 4: Impact of a ZCA L3 cache and null block rate on MPKI, IPC, memory write traffic and global memory traffic

Implementing a ZCA cache on the last cache before exiting the chip to access the main memory should allow reducing the number of the most costly misses. Table 4 illustrates the potential increase in performance and decrease memory traffic associated with using a L3 ZCA cache.

As we were expecting from Table 1, several applications featuring a large proportion of misses on null blocks on the L3 cache on the base architecture benefit significantly from the use of the ZCA cache, e.g. *mgrid, gcc, mesa* and *sixtrack*. For these applications, the ZCA cache captures temporal locality on null data blocks: null data blocks are used and remain null till their reuse. Therefore a significant fraction of the misses on the L3 cache are removed and many write backs are also avoided. This memory traffic gain translates in a quite significant overall performance gain.

However in some cases, a null block is fetched from the memory and then immediately overwritten with non-null data. In this case, the ZC cache only slightly delays the allocation of the block in the main cache. This situation occurs quite often on *gzip*. On *gzip*, despite a high proportion of misses on null blocks, the ZC cache does not reduce significantly the miss ratio on the main cache.

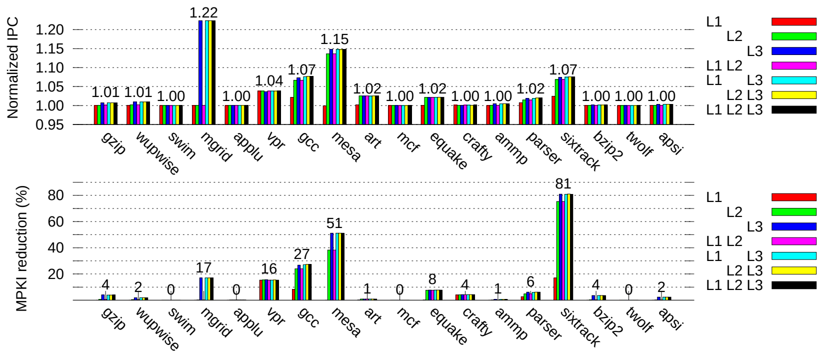

Figure 6: Normalized IPC and MPKI reduction for various ZCA cache configurations

5.2 ZCA Caches in the Memory Hierarchy

As pointed out in Section 3, different options can be considered for using ZCA caches. We explored the whole spectrum of possibilities ranging from using a single ZCA cache at one level to using ZCA caches at all cache levels.

Figure 6 illustrates the performance gain and L3 miss rate reduction achieved when using ZCA caches at various places in the memory hierarchy, ZCA configurations are those listed in Table 3. For instance, bar "L1 L3" indicates that a ZCA cache is considered for L1 and L3 caches, but that the L2 cache is a conventional cache.

Our experimental results essentially show that using a single ZCA cache at the L3 level is sufficient to capture most of the potential benefits. In practice, a superscalar execution allows to tolerate a miss on the L1 cache hitting on the L2 cache. The most significant benefit is when the ZCA cache prevents a long latency miss. To avoid lengthening the access time to the L1 cache, one has to limit the size of the ZC cache. In our experiments, the ZC cache at L1 level only maps a 1MB of null blocks: the overall region mapped by cache hierarchy is not widely enlarged.

Threshold effects appear for several benchmarks; e.g. *gcc* is perfectly accommodated with a ZC cache mapping 4MB, but *mgrid* needs a ZC cache mapping 32MB. We also noted that *apsi* would need a ZC cache mapping 128MB during its initialization phase.

Since experiments have shown that implementing a ZCA cache at the last cache level in the memory hierarchy, from now on, our experiments will assume a single ZCA cache at the L3 level.

5.3 Temporal Behavior of the ZCA Cache

In order to better understand the dynamic behavior of applications with the ZCA cache, we measured their behavior on different consecutive slices of executions. Figure 7 illustrates the relative speed-up enabled by the use of the ZCA cache over 50 billion instructions, each point representing an interval of one billion instructions. It can be noted that on some applications there are different phases corresponding to different behaviors.

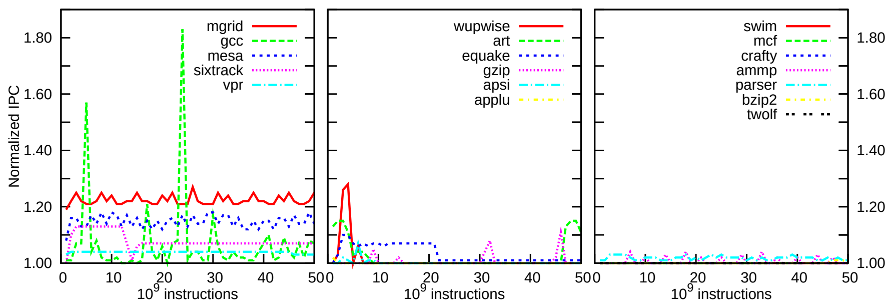

Figure 7: Normalized IPC during execution from beginning to 50.10^9 instructions.

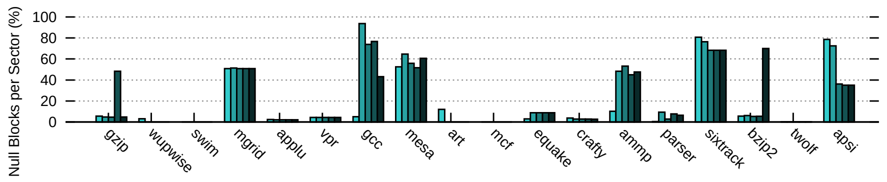

Figure 8: Ratio (%) of valid null blocks per allocated sector in the ZC cache at different execution points from 10.10^9 to 50.10^9 instructions

5.4 Spatial locality of null blocks

In Section 3, we mentioned that the ZCA cache is designed to leverage the spatial locality of null data blocks, since one single tag can represent a large sector of N blocks.

In order to illustrate the spatial locality of the null blocks in applications, we have measured the average number of null blocks per valid sector in the ZC cache for a sector size of 128 64B-blocks at different execution points. These results are presented in Figure 8.

In practice, the number of null blocks per valid sector is in general relatively high for the applications featuring a high proportion of accesses to null blocks, i.e. *mgrid, gcc, mesa* and *sixtrack*. This is exploited by the ZC cache.

5.5 Evaluating ZCA caches on a Multicore

ZCA caches can be used for uniprocessors as well as multicores. On multicores, ZCA caches could be implemented on private cache or on shared caches. The impact of using a ZCA cache in a private level is very similar to the uniprocessor case: miss rates are lowered and traffic with the remainder of the memory hierarchy is reduced.

The same benefits can be anticipated for using the ZCA cache for shared cache levels. In order to illustrate this phenomenon, we evaluate here the performance impact of using a ZCA cache on a dual-core sharing the L3 cache. Our performance estimation is for multiprogrammed workloads.

We use weighted IPC [19, 10] as performance metric where $Wipc = \frac{ipc_{parallel}}{ipc_{alone}}$ with ipc_{alone} be the IPC of the application running alone on the processor.

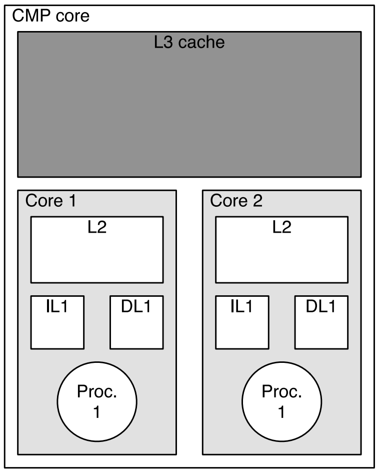

Figure 9: Multicore architecture simulated

Application	Wipc no ZCA	Wipc ZCA	Wipc imp. (%)
apsi	1.00	1.00	0
gzip	0.98	0.99	1
bzip2	0.88	0.89	2
apsi	0.99	1.00	1
mesa	1.00	1.13	13
gzip	0.98	1.00	2
mesa	1.00	1.12	13
vpr	0.79	0.86	9
vpr	0.88	0.95	8
gzip	0.98	0.99	1
wupwise	0.99	1.00	1
apsi	0.99	0.99	0
wupwise	1.00	1.01	1
bzip2	0.86	0.87	1
wupwise	1.00	1.01	1
gzip	0.98	1.00	2

Table 5: Relative speed-up, shared L3 2MB

In order to keep comparable values, we choose to evaluate the performance on a 50 billion-instruction slice. The simulation is run until each application commits at least 50 billion instructions.

For our experiments, we assume a dual-core. Each core is identical to the processor described in the previous section, the L3 cache is shared among the two cores as illustrated in Figure 9. We first simulate each application as standalone with a conventional L3 cache, and then we simulate the two applications running together in parallel first sharing a conventional 1MB L3 cache, then sharing a 1MB L3 ZCA cache.

Table 5 and Table 6 represent performance measured as weighted IPC and miss rates assuming a 1MB traditional L3 cache and a 1MB ZCA cache. The ZCA cache is able to map up to 32MB of null data blocks.

Application	MPKI alone	MPKI no ZC	MPKI with ZCA	MPKI red. (%)
apsi	2.34	2.34	2.31	1
gzip	0.36	0.47	0.41	12
bzip2	2.34	3.31	3.15	5
apsi	2.34	2.42	2.35	3
mesa	0.57	0.57	0.29	50
gzip	0.36	0.43	0.40	8
mesa	0.57	0.58	0.30	49
vpr	0.81	1.92	1.47	24
vpr	0.81	1.40	1.01	28
gzip	0.36	0.48	0.43	10
wupwise	3.13	3.30	3.26	1
apsi	2.34	2.43	2.39	2
wupwise	3.13	3.23	3.18	2
bzip2	2.34	3.47	3.38	3
wupwise	3.13	3.14	3.07	2
gzip	0.36	0.48	0.42	12

Table 6: L3 miss rate on a dual-core

For some application pairs, e.g. *gzip* and *vpr* sharing the L3 cache induces a large increase of the miss rates comparing with stand alone execution. As expected, the ZCA cache allows a reduction of the L3 miss rates, in the range $1-50\%$ for our experiments. These miss rate reductions induce some significant performance improvements. For instance *mesa* experiences a near 13% speedup when combined with *vpr*. The same phenomenon as for uniprocessor is encountered: misses on null blocks are avoided and space in the main cache occupied by the null blocks is freed.

6 Related Work

Several studies have shown that memory content [6, 21, 1, 16, 5, 7, 9] as well as cache content [11, 2, 3, 9] are often highly compressible.

The MXT technology [1] from IBM proposes a compressed memory. 1 KByte uncompressed physical blocks are stored in one to four 256 bytes compressed physical blocks. Compression/decompression latency (64 cycles) of such a large block is a major issue. However the main difficulty with this approach is the granularity of the main memory access. On MXT, this is addressed through the use of a L3 cache featuring a 1 KByte block size. Ekman and Stenstrom [7] also propose a compressed memory but use smaller blocks approximately corresponding to a cache line (64 bytes). Four different sizes of compressed blocks are considered and a simple compression scheme is considered to limit the decompression latency.

Compressing data in the L1 data cache was considered by Zhang, Yang and Gupta with the Frequent Value Cache, FVC [24]. Zhang et al. show that, for many applications, a significant portion of the data accessed by the applications exhibit a very high degree of value locality, i.e. the 10 most frequently used data represent a significant portion of the memory accesses. They propose to augment the L1 data cache with a direct mapped FVC, where the most frequently used data are stored in a compressed form. The same authors then propose a compressed cache design based on the same principle [22]. In this design each cache line location can store either one uncompressed block or two compressed blocks. This study show that cache miss rates can be reduced by this technique, but does not consider the performance loss associated with the longer access time on a compressed cache.

Alameldeen and Wood [2] build upon [22] to study cache compression on L2 caches. They show that compression in the L2 cache is often quite efficient at increasing the effective size of the working set resident in the cache. They also show that compression/decompression latency may sometimes hurt performance and proposes an adaptive policy to control the use of compression.

Hallnor and Reinhart [9] leverage their previously presented Indirect Index Cache, IIC [8] to propose a compressed cache design, IIC-C. Each block is compressed in several sub-blocks. Pointers to the sub-blocks are associated to the tag array. IIC uses indirect access to the data array. This allows the IIC-C cache to take part of a fragmented cache. Therefore IIC-C allows higher compression factor than the previous solutions. This method is also compatible with a MXT-like compressed memory design.

The Selective Compressed Memory System (SCMS)[12] proposes a scheme where memory lines are compressed by adjacent pairs. If the compression factor of the adjacent pair is larger than 50 % then they are stored in the cache in single set. This allows using a single address tag to map the two compressed cache blocks.

Our ZCA cache is also a compressed cache. The main differences with all the previous compressed cache proposals are 1) its specialization since only null blocks are compressed, 2) the simplicity of the decompression. As the Frequent Value Cache [24], the ZCA cache features checking in parallel a compressed cache and a conventional cache, but the ZC cache may map orders of magnitude larger memory area than the FVC. As in SMCS [12], a single address tag of the ZC cache may map several compressed cache blocks, but instead of two blocks in SMCS, a ZC cache address tag may map N blocks.

The ZCA cache reduces write back traffic through avoiding write-back when an already null data block is rewritten. The same phenomenon of exploiting the equality between new and old data was also exploited by silent stores in another context [13].

7 Augmenting a Decoupled Sectored Cache with ZC Cache Functionalities

Decoupled sectored caches [17], DSC, were introduced to reduce the address tag area on a L2 cache, while maintaining a miss ratio in the very same range as a traditional one address tag per block cache. On DSC, the address tag array and the data array are decoupled, a selection tag, also called back-pointer in [23] is associated with the cache block in the data array and allows retrieving the correct address tag at access time. In DSC, the blocks valid in the cache are mapped by an address present in the tag array. The selection tag associated with the cache line allows retrieving the tag the associated tag in the tag array. The overall area mapped by the DSC tag array can be much larger than the size of its data array (e.g. 8 or 16 times). DSC was recently shown to be a cost-effective solution to implement coarse grain tracking of the cache behavior [23], e.g. for limiting coherency transactions in a multicore [14, 4] or for generating efficient prefetch.

The DSC address tag array can be augmented to implement a ZC cache content as follows. N validity bits are associated with each address tag. When the corresponding validity bit is set, the block is present in the cache and null. Therefore DSC can be augmented at a minimal cost with the ZCA cache functionalities.

8 Conclusion

Some applications access and manipulate a large number of null data blocks. These null data blocks occupy cache space in the memory hierarchy. In this paper, we have presented Zero Content Augmented Cache, a feasible design for storing null data blocks in an adjunct ZC cache associated with a main conventional cache. The ZC cache can map the null blocks from a memory zone orders of magnitude larger than the main cache, while its hardware cost remains a fraction of the one of the main cache. For instance, a ZC cache mapping up to 32MB of null data blocks can be implemented with less than 78KB of storage: the ZC cache retains null blocks from a very large area, it also frees space for non-null blocks in the main cache.

We have shown that using a ZCA cache as the last level of cache before accessing to the long latency main memory is a good trade-off. Our experiments showed that, for SPEC CPU 2000, the ZCA cache can reduce the overall cache miss rate by a very significant factor for some applications (e.g. 81%. on *sixtrack* in our experiments). Moreover, the ZCA cache also reduces the write-back traffic. This memory traffic reduction translates into up to a global performance increase (up to 22 % on *mgrid* in our experiments). The use of the ZCA cache never degrades performance even on applications featuring no null block access.

ZCA caches can be used in uniprocessor, as well as on multiprocessors. On multicores, memory bandwidth is a scarce resource. On applications manipulating null data blocks, using ZCA caches will allow to reduce memory traffic, and therefore to globally enhance the overall performance.

References

[1] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith. Performance of hardware compressed main memory. In *HPCA*, pages 73–, 2001.

[2] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *ISCA*, pages 212–223. IEEE Computer Society, 2004.

[3] A. R. Alameldeen and D. A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. *Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison*, Apr 2004.

[4] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *ISCA*, pages 246–257, 2005.

[5] R. S. de Castro, A. P. do Lago, and D. D. Silva. Adaptive compressed caching: Design and implementation. In *SBAC-PAD*, pages 10–18. IEEE Computer Society, 2003.

[6] F. Douglis. The compression cache: Using on-line compression to extend physical memory. In *USENIX Winter*, pages 519–529, 1993.

[7] M. Ekman and P. Stenström. A robust main-memory compression scheme. In *ISCA*, pages 74–85. IEEE Computer Society, 2005.

[8] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *ISCA*, pages 107–116, 2000.

[9] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *HPCA*, pages 201–212. IEEE Computer Society, 2005.

[10] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 2006.

[11] J.-S. Lee, W.-K. Hong, and S.-D. Kim. A selective compressed memory system by on-line data decompressing. In *EUROMICRO*, pages 1224–1227, 1999.

[12] J.-S. Lee, W.-K. Hong, and S.-D. Kim. An on-chip cache compression technique to reduce decompression overhead and design complexity. *J. Syst. Archit.*, 46(15):1365–1382, 2000.

[13] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *International Symposium on Microarchitecture*, pages 22–31, 2000.

[14] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *ISCA*, pages 234–245, 2005.

[15] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, 2005. http://sesc.sourceforge.net.

[16] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed memory. In *IPDPS*, page 66. IEEE Computer Society, 2001.

[17] A. Seznec. Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio. In *ISCA*, pages 384–393, 1994.

[18] A. Seznec. Analysis of the o-geometric history length branch predictor. In *ISCA*, pages 394–405, 2005.

[19] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 international conference on Measurement and modeling of computer systems*, 2002.

[20] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.2. http://quid.hpl.hp.com:9081/cacti/.

[21] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference, General Track*, pages 101–116. USENIX, 1999.

[22] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 258–265, New York, NY, USA, 2000. ACM Press.

[23] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *MICRO*, 2007.

[24] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. *SIGPLAN Not.*, 35(11):150–159, 2000.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399