HAL
open science

Decoupled Zero-Compressed Memory

Julien Dusser, André Seznec

To cite this version:

Julien Dusser, André Seznec. Decoupled Zero-Compressed Memory. Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI), Mani Azimi (Intel), Rajeev Balasubramonian (Univ. Of Utah), Partha Kundu (Intel), Jan 2010, Bangalore, India. 11 p. inria-00468354

HAL Id: inria-00468354

https://inria.hal.science/inria-00468354v1

Submitted on 30 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Decoupled Zero-Compressed Memory

Julien Dusser
julien.dusser@inria.fr

André Seznec
andre.seznec@inria.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract

For each computer system generation, there are always applications or workloads for which the main memory size is the major limitation. On the other hand, in many cases, one could free a very significant portion of the memory space by storing data in a compressed form. Therefore, a hardware compressed memory is an attractive way to artificially increase the amount of data accessible in a reasonable delay.

Among the data that are highly compressible are null data blocks. Previous work has shown that, on many applications null blocks represent a significant fraction of the working set resident in main memory. We propose to leverage this property through the use of a hardware compressed memory that only targets null data blocks, the decoupled zero-compressed memory. Borrowing ideas from the decoupled sectored cache [12] and the zero-content augmented cache [7], the decoupled zero-compressed memory, or DZC memory, manages the main memory as a decoupled sectored set-associative cache where null blocks are only represented by a validity bit.

Our experiments show that for many applications, the DZC memory allows to artificially enlarge the main memory, i.e. it reduces the effective physical memory size needed to accommodate the working set of an application without excessive page swapping.

Moreover, the DZC memory can be associated with a zero-content augmented cache to manage null blocks across the whole memory hierarchy. On some applications, such a management significantly decreases the memory traffic and therefore can significantly improve performance.

1 Introduction

Over the past 50 years, the progress of integration technology has allowed to build larger and larger memories as well as to build faster and faster computers. This trend has continuously triggered the development of new applications and/or new application domains that demand for always more processing power and for more memory space. For each new generation of computers, some end-users are faced with the limited size of the physical memory while swapping to disks kills performance.

In order to artificially enlarge the physical memory space, it has been proposed to compress data in memory. Both software [6, 11, 13, 5] and hardware compressions [1, 3, 8] have been considered.

In this paper, we propose a new organization of a hardware compressed memory, the decoupled zero-compressed memory or DZC memory. Only null data blocks are compressed since they represent a significant fraction of the data resident in memory (up to 30% [8]). Only non-zero blocks are really stored in the compressed physical memory (*CP memory*). Our compressed memory design is inspired by the decoupled sectored cache [12] and the Zero Content Augmented cache [7]. In the (N,P)-decoupled sectored cache, a sector (i.e. P consecutive cache lines) can map cache blocks belonging to P different cache sectors. Typically, the size of the region that can be mapped by the tag array is larger than the size of the cache. In the DZC memory, we borrow the same principle. P being the size of a page in memory, a region of the CP memory consisting of $M * P$ bytes will map blocks of memory belonging to Q pages of the uncompressed physical memory (*UP memory*) with $Q \geq M$. The memory controller is in charge of translating the UP memory address (the only address seen by the processor) towards the CP memory address (or to return a null block).

Our simulations show that the DZC memory is quite effective at enlarging the size of the working set that can reside in the main memory and thus avoids a large number of page faults.

The remainder of this paper is organized as follows. In Section 2, we review the previous works on memory compression and point out the important issues that must be addressed in the design of a hardware compressed mem-

ory. Section 3 presents experiments confirming the observation that a significant amount of the memory blocks are made only of null values. In Section 4, we present the DZC memory principles, then we present a possible implementation. Section 5 presents simulation results showing the benefits of using a DZC memory in place of a conventional memory. Section 6 presents a solution managing null blocks across the cache hierarchy through combining a DZC memory with a zero-content augmented cache. Section 7 concludes this study.

Notation Throughout the paper, we will use the terms memory blocks and memory lines. A *memory block* will be the chunk of data that can be manipulated by a program. A *memory line* is the location in memory where a *memory block* is stored.

2 Related Work and Issues

2.1 Software Compression

In previous approaches to compress memory [6, 13, 11, 5, 4], it was proposed to dedicate part of the physical memory to store memory pages in a compressed form. The remainder of the physical memory is left uncompressed. Only this uncompressed memory is directly addressed by the processor. On a page fault on the uncompressed memory, the page is searched in the compressed memory. When found the page is uncompressed. That is the compressed memory acts as a cache to hide the latency of swapping on disks. This approach enlarges the memory space that applications can use without swapping, but a high compression/decompression penalty is paid on each page fault on the uncompressed memory. Dimensioning the respective sizes of the compressed and uncompressed memory is rather challenging.

2.2 Hardware Compressed Memories

Only a few studies have addressed using hardware compressed memories and they face some issues.

The MXT technology [1] from IBM uses compression on 1-Kbyte uncompressed physical blocks. Blocks are stored in one to four 256-byte compressed physical blocks. Blocks are accessed through an indirect access: one must first retrieve the address in the compressed memory in a table, then access the compressed memory.

Compression/decompression latency (64 cycles) of such a large block is a major issue. However the main issue with this approach is the granularity of the main memory access: the indirect access table must store an address per memory block. In order to limit the volume of this table, the memory block size is 1-Kbyte. However using large L3 blocks may drastically increase the memory traffic on read misses as well as on write-backs. On a write-back on memory, the size of the compressed block can change. The solution adopted on MXT is to free the previous allocation of the block and allocate a new position in the memory.

Ekman and Stenström [8] tried to address two issues associated with the MXT technology: indirect access and the large block granularity. In order to address large block granularity, (i.e. the size of the indirect access table), Ekman and Stenström proposed to store compressed blocks from a physical page in the uncompressed memory in consecutive memory locations. They consider 64-byte memory blocks. A simple compression scheme, FPC [5], resulting in only four different sizes of compressed blocks is considered in order to limit the decompression latency. A descriptor of the compressed form is maintained for each physical page. Moreover, since a descriptor is attached with the physical page, [8] suggests to modify the processor TLB to retrieve the compressed address, thus saving the indirection to access the compressed memory.

The basic compression scheme suffers from an important difficulty on write operations. On a write, the size of a memory block in the compressed memory may increase. When the compressed physical blocks of a page are stored sequentially, any size increase of block triggers the move of the whole page. In order to prevent moving compressed blocks and compressed pages in the memory, [8] proposed to insert gaps between blocks at the ends of sub-pages and pages. This avoids moving all the pages on each block expansion, but wastes free space and does not prevent all data movements.

Moreover, the proposed implementation suffers from other drawbacks. First the uncompressed to compressed address translation is performed through TLB for access misses, but write-backs need some other uncompressed to compressed address translation mechanism (not described in [8]). A second difficulty is to maintain cache coherency in a multiprocessor: the access to the main memory is done using the address in the compressed memory while the cache hierarchy is accessed with the address in the uncompressed memory.

2.3 Formalizing Hardware Compressed Memory

The difficulties mentioned above for the compressed memory scheme proposed by Ekman and Stenström have lead us to better formalize the concept of a compressed

memory system.

We distinguish two address domains, the Uncompressed Physical memory space, or *UP memory* space and the Compressed Physical memory space, or *CP memory* space. All the transactions seen by the processors or IOs are performed in the uncompressed memory space. The memory compression controller performs a translation of the UP address in the CP address.

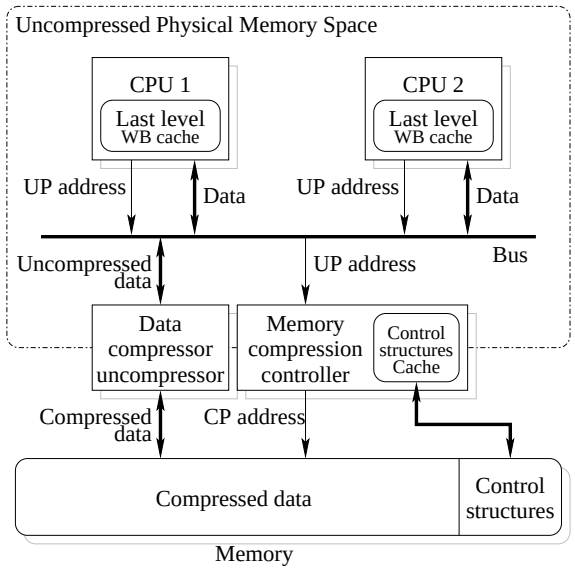

Figure 1: Hardware compressed memory architecture: the CP address is seen only by the compressed memory.

Figure 1 illustrates this model for a bus-shared memory multiprocessor. The model would also fit a distributed memory system, with pairs of associated CP memory space and UP memory space.

3 Null Blocks in Memory

Several studies have shown that memory content [6, 13, 1, 11, 5, 8, 9] and cache content [10, 2, 3, 9] are often highly compressible. Ekman and Stenström [8] showed that on many applications many data are null in memory and that in many cases, complete 64-byte blocks are null. For SPEC2000 benchmarks, they reported that 30% of 64-byte memory blocks are null blocks with some benchmarks such as *gcc* exhibiting up to 80% of null blocks.

Our own simulation confirms this study. Using the Simics simulation infrastructure [1] thus taking into account the Linux operating system simulation, we took a snapshot of the memory content after simulating 50 billion instructions. Figure 2 represents the proportion of null blocks in memory assuming 64-byte blocks. One could note that, on many applications the ratio of null blocks is quite high exceeding 40% in several cases.

A compression scheme targeting only null memory blocks might be considered as targeting only the low hanging fruits in memory compression. However these null blocks represent a sizable portion of the compressible blocks on a hardware compressed memory. As an argument for our thesis, Figure 2 also reports the ratio of 64-byte blocks in memory that could be compressed in 32 bytes using FPC [5], a word level compression algorithm. FPC is here applied with four patterns: uncompressed, null, MSB-null and all ones.

4 Decoupled Zero-Compressed Memory

As pointed out above, the ratio of null memory blocks is quite high for many applications. In this section, we present a hardware compressed memory that exploits this property, the decoupled zero-compressed memory or *DZC memory*. As Ekman and Stenström proposal [8], our proposition targets medium grain memory block sizes in the same range of the cache block sizes, i.e. around 32-128 bytes.

The principle of the DZC memory is derived from the decoupled sectored cache [12] and the zero-content augmented cache [7].

We divide the DZC memory in equal size regions, we will call C-spaces, for compressed memory spaces. Typically the size S of a C-space is about 64 to 512 times larger than the size P of an uncompressed page, i.e. in the range of 512 Kbytes to 4 Mbytes if one consider 8-Kbyte physical pages.

An uncompressed physical page, UP page, is mapped onto a given C-space, i.e., the non-null blocks of the UP page are represented in the C-space. The null blocks are not represented in the DZC memory but are only represented by a null bit. To store the non-null blocks, we manage the main memory as a set-associative decoupled sectored cache. Each page is treated as a sector. It is allocated within a C-space, i.e., each non-null block of the page has S/P possible line positions in the C-space (see Figure 3).

Read access to an uncompressed memory block at UP address $PA + P_{offset}$ on the main memory is performed as follows. A P-page descriptor is associated with the uncompressed memory page at address PA. This P-page descriptor contains a pointer CA on the C-space where the page is mapped. For each blocks in the page, the P-page

[1] Simics is a Virtutech software.

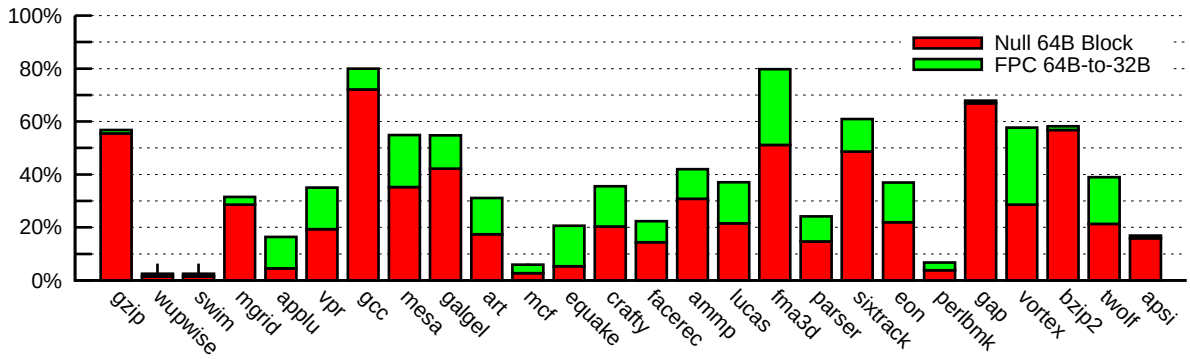

Figure 2: Ratio of 64B blocks null and 64B blocks compressible to 32B using FPC after 50 billions instructions

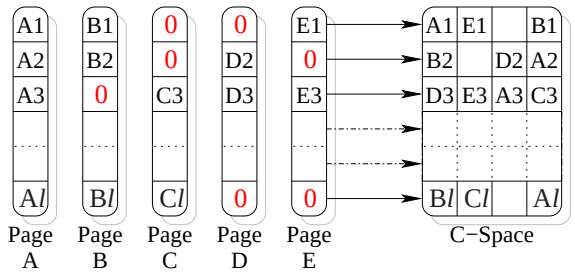

Figure 3: Five uncompressed physical pages are stored in a four-page C-space. Each non-null memory block is stored in one of 4 possible memory lines

descriptor also features a null bit n to represent whether the block is null or not and a way pointer WP to retrieve the block when it is non-null. When n is set the memory block is null, otherwise the block resides in the compressed memory at address $CA*S+WP*P+P_{offset}$ (see Figure 4).

On the write of an uncompressed memory block, four situations must be distinguished depending on both the previous and new status.

- *write a null block in place of a null block:* no action. The P-page descriptor was already featuring a null way pointer for the block

- *write a non-null block in place of a non-null block:* retrieve the address of the non-null block in the compressed memory and update the block in compressed memory

- *write a null block in place of a non-null block:* update the way pointer to null and, as explained later, free the memory line previously occupied in the C-space

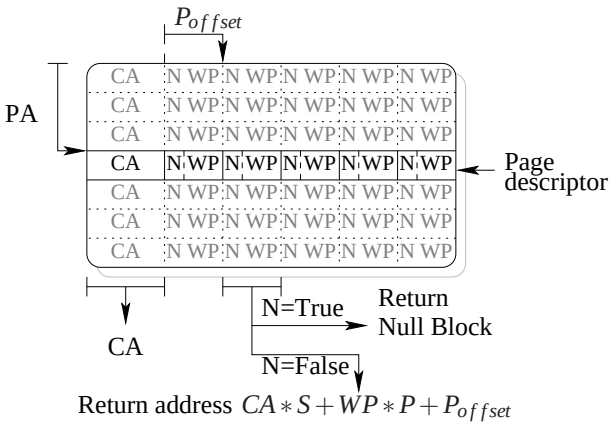

Figure 4: A page descriptor inside page descriptor array. CA, the C-space address and a way-pointer WP and a null bit N per block in the page

- *write a non-null block in place of a null block:* allocate a memory line in the C-space and update the way pointer accordingly.

To manage these two last situations, a C-space descriptor (Figure 5) is associated with each C-space. The C-space descriptor consists of a set of *validity* bits v, one bit per memory line in the C-space. The validity bit indicates whether or not a line is occupied.

Freeing a memory line in the DZC memory is then just simply setting its validity bit to zero.

Allocating a line in the DZC memory for a physical memory line requires finding a free block in the set of possible locations in the C-space. That is scanning the validity bits of all the possible lines. When there is no free line for the block, an exception must be raised and an action must be taken: the overall physical page is moved from

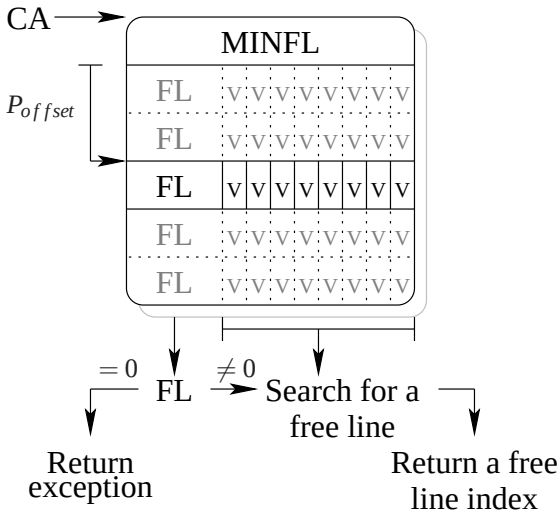

Figure 5: C-space descriptor and free line allocation.

the C-space to another C-space with enough free lines. If no such space exists one or more page must be ejected.

While the validity bits are sufficient to manage the allocation of pages in C-space, storing some redundant information allows an easier management. Therefore, in order to guide such a physical page move, the C-space descriptor maintains a counter FL (free lines) per set of lines indicating the number of free lines in the set. Moreover $MINFL$, the minimum of the FL counters on all the sets of memory lines is also maintained.

On a write-back involving a state change of the memory block, in the general case the FL counter and $MINFL$ are also updated. The exceptional case where a new block has to be allocated and no free line is valid corresponds to $FL = 0$. The physical page must then be moved to another C-space: in order to guarantee that the chosen C-space will be able to store the whole physical page, one has only to chose a target C-space for which $MINFL$ is non-null.

4.1 Randomizing Null Blocks Distribution

The DZC memory acts as a set associative cache in which null blocks would not cost a memory line. Unfortunately, experimental results showed that null blocks are not distributed evenly in the pages. There are often more null blocks at the end of the page than at the beginning of the page. Therefore the conventional set mapping would lead to more saturation on sets corresponding to the beginning of pages.

In order to randomize the distribution of null blocks across all sets, we use a exclusive-or indexing on the sets of the DZC memory: the page offset in the UP address is exclusive-ored with the lowest page number bits.

4.2 Benefits

Unlike IBM MXT, the DZC memory can handle small blocks and has very short decompression latency. This translates in a finer memory traffic granularity and a shorter access time to the missing data in memory.

As already pointed, Ekman and Stenström approach [8] faces a difficulty on write-backs when the size of a block increases. This often necessitates to move the data in the page or to reallocate and move the overall memory page. Such data movements are rare events on the DZC memory as illustrated in the experimental section.

4.3 Control Structure Overhead

The control structures needed in a DZC memory are respectively the P-page descriptors associated with the physical pages of the uncompressed memory and the C-space descriptors.

The P-page descriptor (Figure 4) features a C-space pointer, a way-pointer and a null bit for each memory block. For instance, if the respective sizes of the C-space and of the physical page are 4 Mbytes and 8 Kbytes, the way-pointer will be 9-bit wide. Assuming 64-byte memory blocks, 128 way-pointers are needed per physical page. A 4-byte C-space pointer would allow to map pages anywhere in a 2^{52} bytes compressed memory. That is a P-page descriptor represents 164 bytes.

The C-space descriptor features a validity bit per memory line in the C-space, i.e. assuming 4-Mbyte C-space and 64-byte memory block, 64 KBits per C-space. The FL counters could be represented on 10 bits as well as $MINFL$. That is a C-space descriptor represents 8353 bytes.

The storage requirement for the control structures of the DZC memory depends on the size of the physical memory mapped onto the DZC memory. Assuming that this size is 1.5 times the size of the DZC memory and the parameters used in the example above, this storage requirement would be $1.5 * 512 * 164 + 8353 = 134,305$ bytes in average per C-space, i.e around 3.2 % of the memory. For a large physical memory, these control structures should be mapped onto the physical memory.

4.4 Memory Compression Controller

In any system, the memory controller is in charge of managing the access to the memory. The memory compression controller can be integrated in this memory controller.

The memory compression controller is in charge of determining whether a block is null or not. On a read, when the block is non-null, the memory compression controller also determines its CP address. On a write-back changing a null block in a non-null block, the memory compression controller is in charge of allocating a free block in the DZC memory. When no free block is available in targeted set, the memory compression controller is in charge of moving the physical page to another C-space in the DZC memory.

All these tasks require the access to the control structures of the DZC memory. In order to allow fast UP address to CP address translation, parts of the control structures must be cached within the memory compression controller.

5 Performance Evaluation

5.1 Metrics for Evaluating a Compressed Memory

The performance evaluation of a compressed memory is a difficult issue.

When the working set of a computer workload fits in the main memory, using a compressed memory does not result in any performance benefit. It even results in a performance loss due to the extra memory latency associated with the UP address to CP address translation and with the decompression latency algorithm.

In our particular case, there is no hardware decompression algorithm. Therefore the extra memory access latency is only associated with the UP to CP address translation latency. This UP to CP address translation is relatively simple (read of a table). When the P-page descriptor is cached in the memory controller, the UP to CP address translation latency only adds a few cycles to the overall latency of the memory controller. As pointed out by Ekman and Stenström [8], such extra 2-3 cycles on a main memory latency has very limited impact on the overall performance of the system.

The main interest of using a compressed memory is to enlarge the memory footprint that a given system may accommodate without swapping on the external disks. Access time to hard-drive is in the 10ms range thus representing about 30,000,000 cycles for current processor technology. Therefore the most important performance metric for a compressed memory is the page fault rate of the applications: we will present this metric for a large spectrum of memory sizes in order to characterize how the use of DZC memory can artificially "enlarge" the memory space available for the application.

However, our DZC memory system may also suffer from C-space saturation on a write-back. Such a C-space saturation forces to move the physical page to another C-space, thus leading to some performance penalty. Ekman and Stenström [8] pointed out that moving pages in the compressed memory are local to the memory and can be performed in the background. They are unlikely to really impact the performance of the system if they remain sufficiently rare. However, these physical page moves may become quite frequent when the memory becomes saturated. We will present statistics on these page moves as our second performance metric.

5.2 Experimental Methodology

Our simulation environment was derived from Simics. Simics allows to monitoring processor to memory transfers, and to plug a memory hierarchy simulator. A standard memory hierarchy configuration illustrated Table 1 was simulated. Both DZC memory and uncompressed memory were simulated for sizes ranging from 8 Mbytes to 192 Mbytes on the first 50 billions of instructions of the SPEC 2000 benchmarks.

CPU	x86 processor
L1 Cache	32KB, 64B line-size, 4-ways, LRU rempl., write allocate
L2 Cache	256KB, 64B line-size, 4-ways, LRU rempl., write allocate
L3 Cache	1MB, 64B line-size, 8-ways, LRU rempl., write allocate
O.S.	Linux Red Hat - Linux 2.6
Benchmarks	SPEC CPU 2000 - ref data set
Compressed Memory	4MB spaces, 8KB pages, 64B blocks

Table 1: Baseline simulator configuration.

Simulated Page Replacement Policy On a page fault, the page must be brought in main memory. If no free memory space is available then some pages must be swapped on the disk in order to make room for the new page.

We simulated a simple LRU replacement for the conventional memory. For the DZC memory, the page must be allocated in some C-space. We simulated a policy derived from the LRU policy. At a first step, we look for a C-space able to store the whole page, i.e., featuring at least a free line in each set (in other words $MINFL \geq 1$). If no C-space is available then we eject the LRU page. However, this ejection does not guarantee that there is room for the whole missing page in the corresponding C-space (if there was some null block in the ejected page, some sets can still be full), in this case we eject extra pages from this C-space until we guarantee that the whole missing page can fit in the memory (i.e., $MINFL \geq 1$).

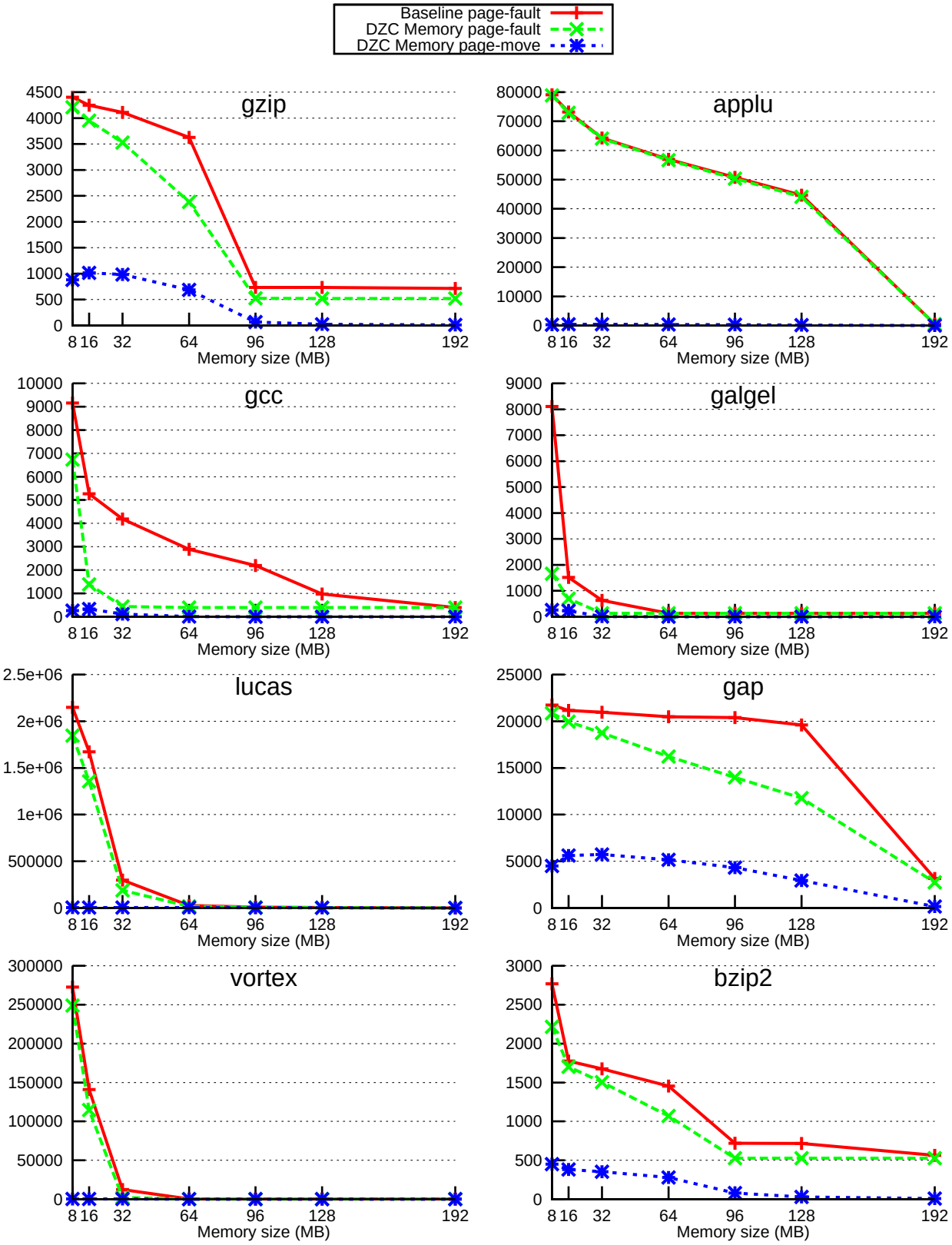

Figure 6: Number of page-faults and page-moves per billion instructions.

Moving Pages on Write-backs As already mentioned, on the DZC memory, the write-back of a non-null block in place of a null block may create an overflow in the corresponding set of the C-space. In this case, the page is moved to another C-space with available free storage (i.e., with $MINFL \geq 1$). When needed, room in memory is made for the page using the freeing policy described above.

5.3 Simulation Results

Figure 6 illustrates the page fault rates and the page move rates for several benchmarks. Results are illustrated in occurrences per billion instructions. For space reasons, only 8 of the 26 benchmarks are illustrated.

Among the omitted benchmarks, *vpr, art, crafty, sixtrack, eon* and *twolf* presented very small page fault numbers (i.e. essentially cold start page faults) for all tested memory sizes. *facerec, fma3d, ammp* and *mesa* presented negligible page fault numbers apart for 8-Mbyte memory sizes. Among the other omitted benchmarks *wupwise, swim, mcf, equake* and *perlbmk* exhibit very similar behavior as *applu*, i.e. quite similar behavior for compressed memory and conventional memory as could be expected from their low rate of null blocks in memory (see Figure 2). *mgrid, parser* and *apsi* exhibited behaviors quite similar to the one illustrated for *vortex*, i.e. a slight decrease of page faults for some memory sizes.

5.3.1 Page Faults Measures

The working sets of the SPEC 2000 benchmarks typically fit in a main memory of 192 Mbytes and in most cases in a much smaller memory. As soon as the working set of the application fits in the main memory, the behaviors of the compressed memory and the uncompressed memory are equivalent.

However, when the memory size is not sufficient to accommodate the working set, using the DZC memory may allow to accommodate the working set of an application while the same conventional memory suffers a significant page number faults. In particular, in our simulations for the *gcc* benchmark, a 32-Mbyte DZC memory is sufficient while 192 Mbytes are needed with the conventional memory. To a smaller extent, the phenomenon also arises with *galgel* (32 Mbytes vs 64 Mbytes) and *ammp* and *mesa* (8 Mbytes vs 16 Mbytes) and *bzip2* (96 Mbytes vs 192 Mbytes). The granularity of our simulations did not allow to capture the effective sizes where the effective working set becomes memory resident for every benchmark. However, the simulations showed that the use of compressed memory allows to significantly reduce the number of page faults for many benchmarks when the working set does not fit in the main memory. This is true for the above mentioned benchmarks as well as on the illustrated *gzip* and *gap*.

On the other hand, as already mentioned on applications exhibiting poor zero compressibility, the behavior of the compressed memory is similar as to conventional memory, e.g. *wupwise, swim, applu, equake,* and *perlbmk*.

5.3.2 Page Moves on the DZC Memory

As illustrated in Figure 6 the page move rate is generally much lower than the page fault rate for the DZC memory. As the penalty for a page move is several orders of magnitude lower than the one on a page fault, page moves induced by writes will not be a major performance issue on a DZC memory.

6 Managing Null Blocks Across the Memory Hierarchy

The Zero-Content Augmented cache, or ZCA cache was recently proposed [7] in order to exploit null blocks in the cache. They showed that adding an adjunct zero-content cache to the last level cache could artificially increase the capacity of this last level cache and therefore allows to increase performance. In the zero-content cache, the spatial locality of the null blocks is leveraged through associating the information on nullity of the blocks in a whole page with a single address tag.

Combining the DZC memory with the ZCA creates an opportunity to manage null blocks throughout the whole memory hierarchy as illustrated on Figure 7. The vector of information representing the nullity of the blocks in a 8-Kbyte page represents only 128 bits. When reading a null block on the memory, one can send back this overall vector instead of the null block. This can improve performance through two phenomena. First, all the null blocks in a page are prefetched as soon as a null block is touched in the page. Second, the traffic on memory is reduced since a single data transfer of 16 bytes reads all the null blocks in the page.

In order to evaluate the performance of this combined DZC-ZCA memory hierarchy, we integrated a L3 ZCA cache in the simulator. We simulated two configurations of the Zero-Content cache with 4K entries (similar to [7]) and 128 entries, each entry being able to map the null blocks of a 8-Kbyte page.

Simulation results are illustrated Figure 8. Benchmarks *crafty, eon* and *perlbmk* are omitted due to their very small

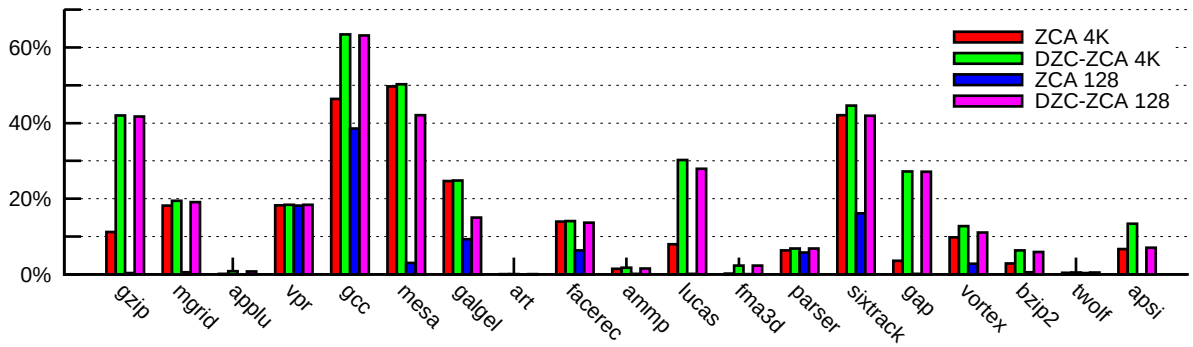

Figure 8: Miss rate reductions using ZCA cache and DZC-ZCA memory

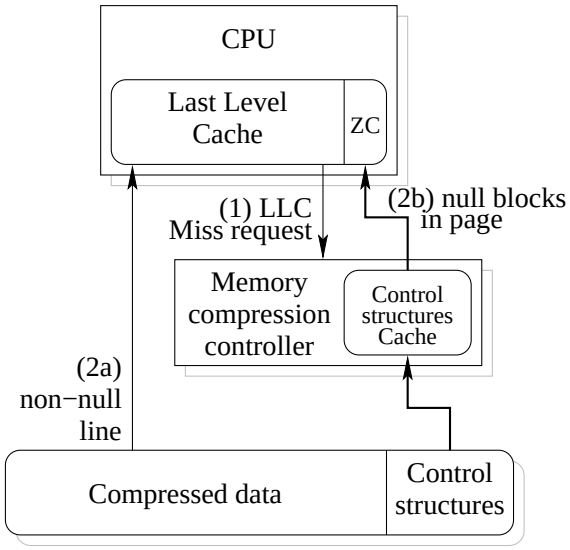

Figure 7: Managing null blocks across the memory hierarchy.

miss rates (Table 2). *Wupwise, swim, mcf* and *equake* are also omitted due to their low null block rates, see Figure 2. For these applications DZC-ZCA has no significant impact.

For the large 4K-entry Zero-Content cache, the ZCA per itself can significantly reduce the L3 miss rate, thus saving memory traffic and increasing performance. This phenomenon is particularly significant (miss rate reduction higher than 10%) for *gzip, mgrid, vpr, gcc, mesa, galgel, facerec, sixtrack* and *vortex*. For most of these benchmarks, the prefetching impact of using DZC-ZCA is negligible: in practice, the large Zero-Content cache was able to retain the null blocks once they were initially loaded in the L3 cache. However *gzip, gcc, lucas, gap* and *apsi* are benefiting significantly from the DZC-ZCA prefetching impact.

The simulation results for the small 128-entry Zero-Content cache emphasize this prefetching impact for a larger set of applications. While the small ZCA cache is not sufficient to retain the null blocks through the whole execution, the spatial locality of the null blocks is leveraged by the DZC-ZCA memory hierarchy. This prefetching impact significantly reduces the miss rate sometimes matching or exceeding the impact of a large Zero-Content cache without prefetching. This phenomenon is particularly pronounced on *gzip, mgrid, gcc, mesa, facerec, lucas, gap, vortex* and *bzip2*. *Apsi* presents a paradoxical behavior: using the small Zero-Content cache without prefetching degrades performance because it is not able to retain the pages with null blocks for a sufficient time to allow significant reuse of the same null block, but DZC-ZCA prefetching leverages the spatial locality of null blocks and is therefore able to improve the overall performance.

7 Conclusion

Main memory size will always remain a performance bottleneck for some applications. The use of a hardware compressed memory can artificially enlarge the working set that can reside in main memory.

For many applications, null data blocks represent a significant fraction of the blocks resident in memory. The DZC memory leverages this property to compress the main memory. As in zero-content augmented caches [7], null blocks are only represented by a single bit. For representing non-null blocks, the main memory is treated as

Application	MPKI	Application	MPKI
gzip	0.45	facerec	4.77
wupwise	2.20	ammp	2.22
swim	16.65	lucas	9.94
mgrid	4.45	fma3d	1.53
applu	9.05	parser	1.69
vpr	1.09	sixtrack	0.42
gcc	1.98	eon	0.00
mesa	0.66	perlbmk	0.12
galgel	10.36	gap	2.61
art	60.63	vortex	0.84
mcf	79.87	bzip2	2.83
equake	19.75	twolf	1.12
crafty	0.11	apsi	3.87

Table 2: Baseline L3 Miss rates (Misses per Kilo-Instructions)

a decoupled sectored cache [12].

Unlike the IBM MXT technology [1], the DZC memory is compatible with the use of conventional cache block size since it can manage 64 bytes memory blocks. The compression/decompression algorithm is trivial and is very efficient. Compared with the scheme proposed by Ekman and Stenström [8], the DZC memory allows a smooth management of compressed data size changes on writes.

Our experimental simulations have shown that the decoupled zero-compressed memory allows to enlarge the size of the working set that can reside in the main memory for many applications, thus avoiding costly page faults when the size of the working set is close to the size of the main memory.

We have also shown that the trivial null block compression can be exploited throughout the whole memory hierarchy through combining a zero-content augmented caches [7] with a DZC memory. Asides enlarging the size of the possible data set resident in memory through DZC memory and decreasing cache miss rates through ZCA cache, the DZC-ZCA combination allows to increase the performance of the overall system through decreasing the memory traffic and prefetching at the same time: a single read request on a null block fetches all the null blocks in a page.

References

[1] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith. Performance of hardware compressed main memory. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 73–81, Washington, DC, USA, 2001. IEEE Computer Society.

[2] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, pages 212–223, Washington, DC, USA, 2004. IEEE Computer Society.

[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] V. Beltran, J. Torres, and E. Ayguadé. Improving web server performance through main memory compression. In *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 303–310, Washington, DC, USA, 2008. IEEE Computer Society.

[5] R. S. de Castro, A. P. do Lago, and D. Da Silva. Adaptive compressed caching: Design and implementation. In *SBAC-PAD '03: Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, pages 10–18, Washington, DC, USA, 2003. IEEE Computer Society.

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

[7] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 46–55, New York, NY, USA, 2009. ACM.

[8] M. Ekman and P. Stenström. A robust main-memory compression scheme. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

[9] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 201–212, Washington, DC, USA, 2005. IEEE Computer Society.

[10] J.-S. Lee, W.-K. Hong, and S.-D. Kim. A selective compressed memory system by on-line data decompressing. *EUROMICRO Conference*, 1:1224–1227, 1999.

[11] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed memory. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 66, Washington, DC, USA, 2001. IEEE Computer Society.

[12] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *ISCA '94: Proceedings of the 21st annual international symposium on Computer architecture*, pages 384–393, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[13] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 101–116, Berkeley, CA, USA, 1999. USENIX Association.