Decoupled Zero-Compressed Memory

Julien Dusser, André Seznec

Decoupled Zero-Compressed Memory

Julien Dusser
Université de Rennes 1
IRISA - UMR6074
Campus de Beaulieu, 263 av. du Général Leclerc, 35042 Rennes Cedex, France
julien.dusser@irisa.fr

André Seznec
INRIA, Centre Inria Rennes – Bretagne Atlantique
Campus de Beaulieu, 263 av. du Général Leclerc, 35042 Rennes Cedex, France
andre.seznec@inria.fr

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, or DZC memory. Main memory is managed as a decoupled sectored cache with physical pages treated as sectors and 64-byte memory blocks treated as subblock. Null memory blocks are represented through a single bit, thus freeing physical memory space for the applications.

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, through caching null memory blocks in the memory controller, the DZC memory also decreases the average access time to the main memory for many applications.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures—*Design Styles*

General Terms

Design

Keywords

Memory compression, Null block, Zero block

HiPEAC 2011 Heraklion, Crete, Greece

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 ever-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 [7, 17, 22, 6] and hardware compressions [21, 3, 9] 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 (30% are reported in [9]). 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 [20] and the Zero Content Augmented cache [8]. 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). In order to be efficient, the memory controller must implement an UP to CP address translation cache, the UCT cache. Interestingly, the UCT cache also caches the null blocks and is able to service requests to null blocks without effectively accessing the memory.

Our simulations shows that the DZC memory is quite effective at enlarging the size of the working set that can reside in the main memory. Moreover, when the working set fits in the main memory, the UCT cache acts as an off-chip Zero-Content cache [8], therefore it decreases the average effective memory access time for many applications.

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 memory. Section 3 presents experiments confirming that a significant amount of the memory blocks stored in memory 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 illustrating the benefits of using a DZC memory in place of a conventional memory. Section 6 concludes this study.

Throughout the paper, we will use the terms memory blocks and memory lines:

memory block is a contiguous, aligned block of data in physical memory the same size as a last-level cache block.

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 compressed memory [7, 22, 17, 6, 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 memory is compressed 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.

Some other studies focus on compressing heap data inside JAVA Virtual Machines. *Sartor et al.* [19, 18] notices an important usage of zero blocks inside and at the end of arrays. Compressing zero blocks with the method describe in [5] allows to save up to 40% of heap size, with an average of 14%. This approach decreases memory usage but requires an extra indirection on access that may impact performance.

2.2 Hardware Compressed Memories

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

2.2.1 IBM Memory eXpansion Technology (MXT)

In the early 2000's IBM produced servers with MXT [11, 21, 10, 12]. The whole main memory is compressed with a parallel variant [13] of LZ77 [23] applied on 1 KB physical blocks. Blocks are stored up 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 KB. However using large L3 blocks may drastically increase the memory traffic on read misses as well as on write-backs.

Another issue is the frequent changes of the size of the compressed blocks. 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.

2.2.2 Ekman and Stenström

Ekman and Stenström [9] try 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* propose 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 [6], 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, they suggest 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, they propose 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 the paper [9]). 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 General Architecture of Hardware Compressed Memory System

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.

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 the translation of the UP address in the CP address.

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.

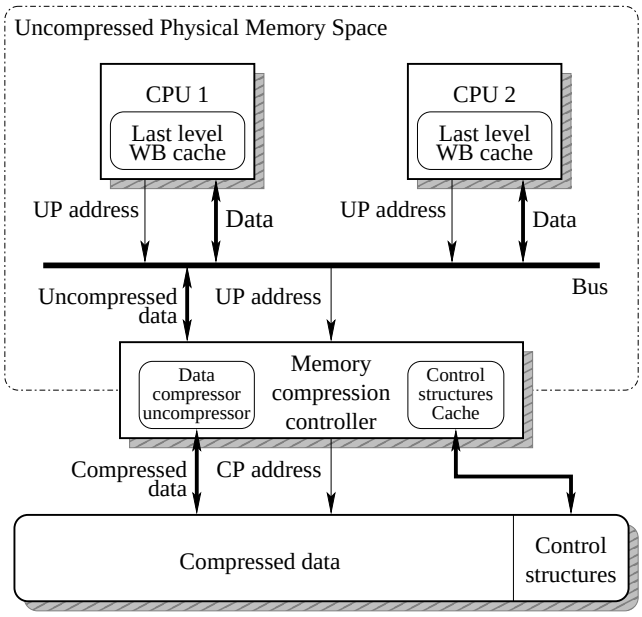

Figure 1: Hardware compressed main memory architecture: the CP address is seen only by the compressed memory.

3. NULL BLOCKS IN MEMORY

Several studies have shown that memory content [7, 22, 1, 17, 6, 16, 9, 14] and cache content [15, 2, 3, 14] are often highly compressible. *Ekman and Stenström* [9] showed that on many applications many data are null in memory and that in many cases, complete 64-byte blocks are null. For SPEC CPU 2000 benchmarks, they reported that 30% of 64-byte memory blocks are null blocks with some benchmarks such as *176.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 content of the physical memory pages touched by the application after simulating 50 billion instructions; Figure 2 represents the proportion of null blocks in memory assuming 64-byte blocks on SPEC CPU 2006 benchmarks. Experiments on SPEC CPU 2000 presented similar trend. One could note that, on many applications the ratio of null blocks is quite high, exceeding 80% in several cases.

Others experiments, with the framework described in [8] showed that the initialisation phase is less than 2 billions instructions for SPEC CPU 2006. After this initialization phase, the rate of null blocks in accesses tend to be relatively stable and close to the static rate measured in Figure 2.

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 [6], a word level compression algorithm. FPC is here applied with four patterns: uncompressed, null, MSB-null and all ones. In most cases, the null blocks represent the most significant part of the compressible blocks.

[1]Simics is a Virtutech software.

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 [9], 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 [20] and the zero-content augmented cache [8].

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 KB to 4 MB if one consider 8 KB physical pages.

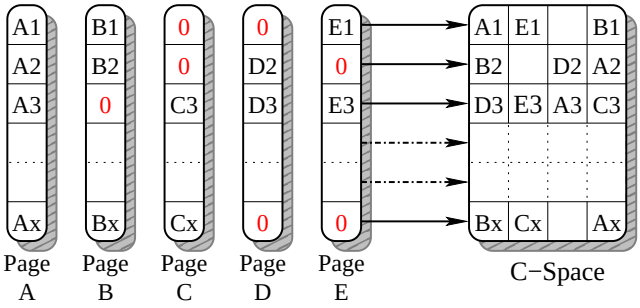

Figure 3: Five uncompressed physical pages are stored in a four-page C-space. Each non-null memory block is stored in one of 4 possible memory lines

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 and allocated within a C-space. Each non-null block of the page has S/P possible line positions in the C-space (see Figure 3).

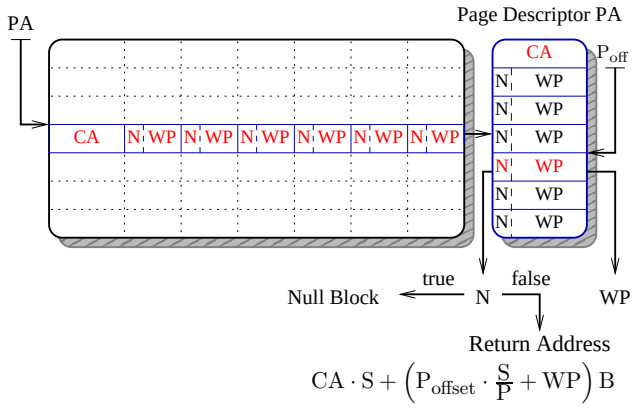

Figure 4: A page descriptor access in page descriptor array. CA, the C-space address and a way-pointer WP and a null bit N per block in the page. B is the line size.

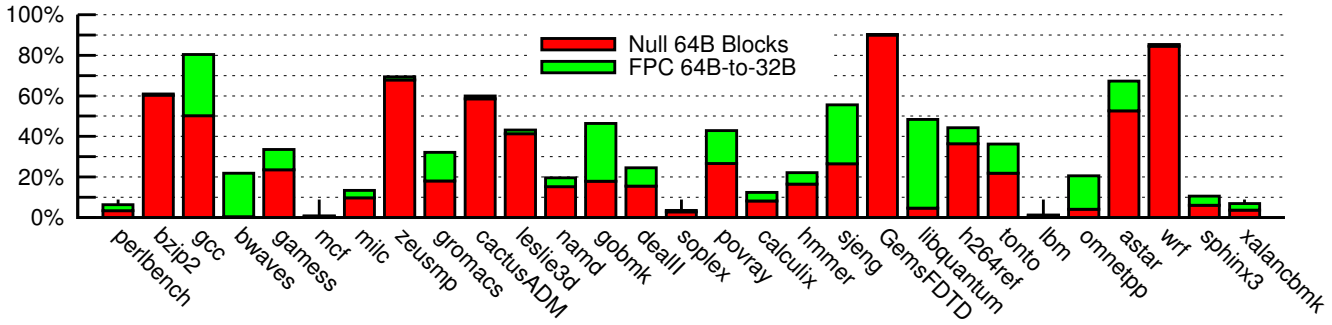

Figure 2: Ratio of 64B blocks null and 64B blocks compressible to 32B using FPC after 50 billions instructions

Read access to an uncompressed memory block at UP address $PA + P_{offset}$ on the main memory is performed as follows (Figure 4). A P-page descriptor, associated with the uncompressed memory page at address PA, contains a pointer CA on the C-space where the page is mapped. For each blocks in the page, the P-page descriptor also contains 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:
$\mathrm{CA} \cdot \mathrm{S} + \left(\mathrm{P_{offset}} \cdot \frac{\mathrm{S}}{\mathrm{P}} + \mathrm{WP}\right) \mathrm{B}$.

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 bit set 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.
- *write a null block in place of a non-null block:* set the null bit, and, as explained later, free the memory line previously occupied in the C-space.
- *write a non-null block in place of a null block:* allocate a memory line in the C-space and update the null bit and 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 the C-space to another C-space with enough free lines. If no such space exists one or more page must be ejected.

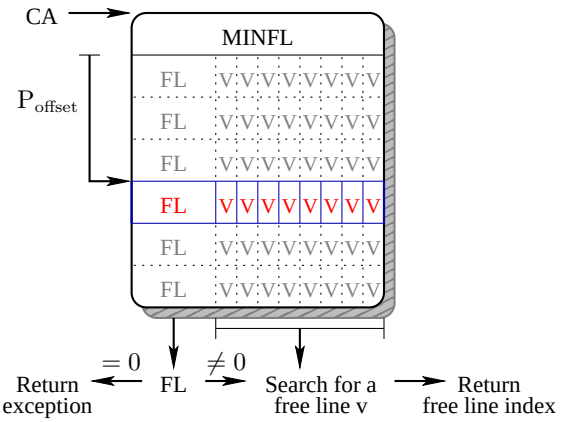

Figure 5: C-space descriptor and free line allocation.

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 $MINFL \geq 0$.

4.1 Improving Null Blocks Distribution

Unfortunately, experimental results showed that null blocks are not distributed evenly on the pages. There are often more null blocks at the end of pages than at the beginning. 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 index computation: the set index in UP address is exclusive-ored with the lowest bits of page number.

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 mentioned, *Ekman and Stenström* approach [9] 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) contains 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 MB and 8 KB, 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 MB C-space and 64-byte memory block, 8 KB per C-space. The FL and $MINFL$ counters could be represented on 10 bits. 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: Bypassing Memory Reads on Null Blocks

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. While page moves and free block allocations are quite infrequent, UP address to CP address translation must be performed on every access. In order to allow fast UP address to CP address translation, P-page descriptors must be cached within the memory compression controller as illustrated in Figure 1.

P-page descriptors are cached in the **U**P address to **C**P address **T**ranslation cache, the UCT cache. The read accesses to null blocks can be directly serviced by the UCT cache, thus the round-trip to the memory chips is saved and the access time to the main memory is shortened. That is when servicing a request on null block, the UCT cache acts as an off-chip Zero-Content cache [8].

5. PERFORMANCE EVALUATION

5.1 The Two Objectives

The performance evaluation of a compressed memory must address two very different objectives. First, we must evaluate to which extent the compressed memory "enlarges" the main memory. Second, we must evaluate the performance loss or increase associated with the use of the compressed memory when the application fits in the memory system since the computer will also be used on that type of applications.

5.1.1 Page faults and page moves

The main interest of using a compressed memory is to enlarge the memory footprint that the 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 even a very small page fault rate in the order of one hundred per billion instructions is unacceptable. Therefore as the main metric of the behavior of the memory system, we will present 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 write-back accesses (see Section 4). Such a C-space saturation forces to move the physical page to another C-space, thus leading to some performance penalty. *Ekman and Stenström* [9] 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.

As our second performance metric, we will present statistics on these page moves in order to ensure that such page moves are quite rare when the page fault rate is acceptable to envisage realistic execution.

5.1.2 Average memory access time

When the working set of a computer workload fits in the main memory, using a compressed memory does not directly result in any performance benefit. Normally, 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 performed through the UCT cache. When the P-page descriptor hits in the UCT cache, the UP to CP address translation latency only adds a few cycles to the overall latency of the memory controller. It might even be performed in parallel with other activities in the memory controller. As pointed

out by *Ekman and Stenström* [9], such extra 2-3 cycles on a main memory latency has very limited impact on the overall performance of the system.

However, when a P-page descriptor is missing, the P-page descriptor must be brought back in the UCT cache from the memory, thus resulting in a significant longer memory access time. On the other hand, on a read on a null block hitting in the UCT cache, the null block can be returned directly to the processor without reading the memory, thus resulting in a shorter memory access time.

To reflect these two opposite contributions to performance, we will illustrate the average memory access time on a DZC memory and an uncompressed main memory. To provide the impact of the contribution on the effective performance, we will use the average contribution of the main memory access time per instruction.

5.2 Experimental Methodology

As explained in section 3 simulation environment was derived from Simics. Simics is a full system simulator that allows to monitor the processor to memory transfers, and to plug a memory hierarchy simulator. We choose a standard memory hierarchy configuration as illustrated in Table 1. Both DZC memory and uncompressed memory were simulated for sizes ranging from 8 MB to 1 GB on the first 50 billions of instructions of each application. We choose the well known SPEC CPU 2006 for their great variability, presenting very different behaviors.

CPU	x86 processor
L1 Cache	32 KB, 64 B line-size, 4-ways, LRU, write allocate
L2 Cache	256 KB, 64 B line-size, 4-ways, LRU, write allocate
L3 Cache	1 MB, 64 B line-size, 8-ways, LRU, write allocate
O.S.	Linux Red Hat - Linux 2.6
Benchmarks	SPEC CPU 2006 - ref data set
Compressed Memory	4 MB C-spaces, 8 KB pages, 64 B blocks
Total Memory access time	250 cycles (25 + 200 + 25) see Section 5.4

Table 1: Baseline simulator configuration.

5.2.1 Simulated Page Replacement Policy

On a page fault, the page must be brought into main memory. If no free memory space is available then some pages must be swapped on the disk.

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., $MINFL \geq 1$).

5.2.2 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 featuring available free storage (i.e., with $MINFL \geq 1$). When needed, space in memory is freed for the page using the same deallocation policy described above.

5.3 Page Faults and Page Moves

Figure 6 illustrates the page fault rates and the page move rates for several representative benchmarks. Note that first touch (the allocation) of a page is not counted as a page fault since it does not trigger a disk access. Results are illustrated in page fault occurrences per billion instructions. Typically, the application would run at an unacceptable poor performance if the page fault rate is higher than a hundred per billion instructions. Then the interesting points on Figure 6 are the points where the number of page faults nearly vanish.

5.3.1 Page Faults on the DZC Memory

We would like to mention that there is a very strong correlation between the interest of using a DZC memory and the ratio of static null blocks in the pages manipulated by the application (see Figure 2). In practice, for 14 out of the 29 SPEC CPU 2006 benchmarks, our simulations show that the DZC memory is able to accommodate the application with a smaller size than an uncompressed memory.

On a first class of benchmarks, such as *410.bwaves*, *470-.lbm*, *483.xalancbmk*, *429.mcf*, there is no visible benefit of using the DZC memory instead of an uncompressed memory. These applications feature a very small rate of static null blocks in allocated pages (less than 15-20 %). Therefore to accommodate their working set requirements, the DZC memory requires approximately the same size that the uncompressed memory.

For the other benchmarks with more significant null block rates in their working set, our simulations essentially confirm that, the DZC memory of size S can accommodate an application with a working set larger than S. This "enlargement" of the memory size increases naturally with the static null block rate.

This can be observed for instance for applications with about 30-50 % of null blocks, such as *403.gcc*. A 128 MB DZC memory is needed to accommodate the application against 256 MB for an uncompressed memory. *437.leslie3d* or *464.h264ref* have a very similar behavior.

As expected, applications featuring a very high ratio of null blocks benefit the most from the DZC memory. For instance, *481.wrf* features about 85 % of null blocks. While it requires a 368 MB of uncompressed memory to accommodate it, a 64 MB DZC memory is sufficient. Similar behaviors are encountered for all benchmarks with very high null block ratios, for instance *459.GemsFDTD*, *473.astar*, *436.cactusADM* and *434.zeusmp*.

5.3.2 Page Moves on the DZC Memory

As illustrated in Figure 6, the page move rate associated with saturated sets of non-null blocks in a C-space are never a performance issue. In practice a large number of such page moves occur when the rate of page faults is very high and would prevent any reasonable execution of the application. When the workload just fit in the DZC memory, some page

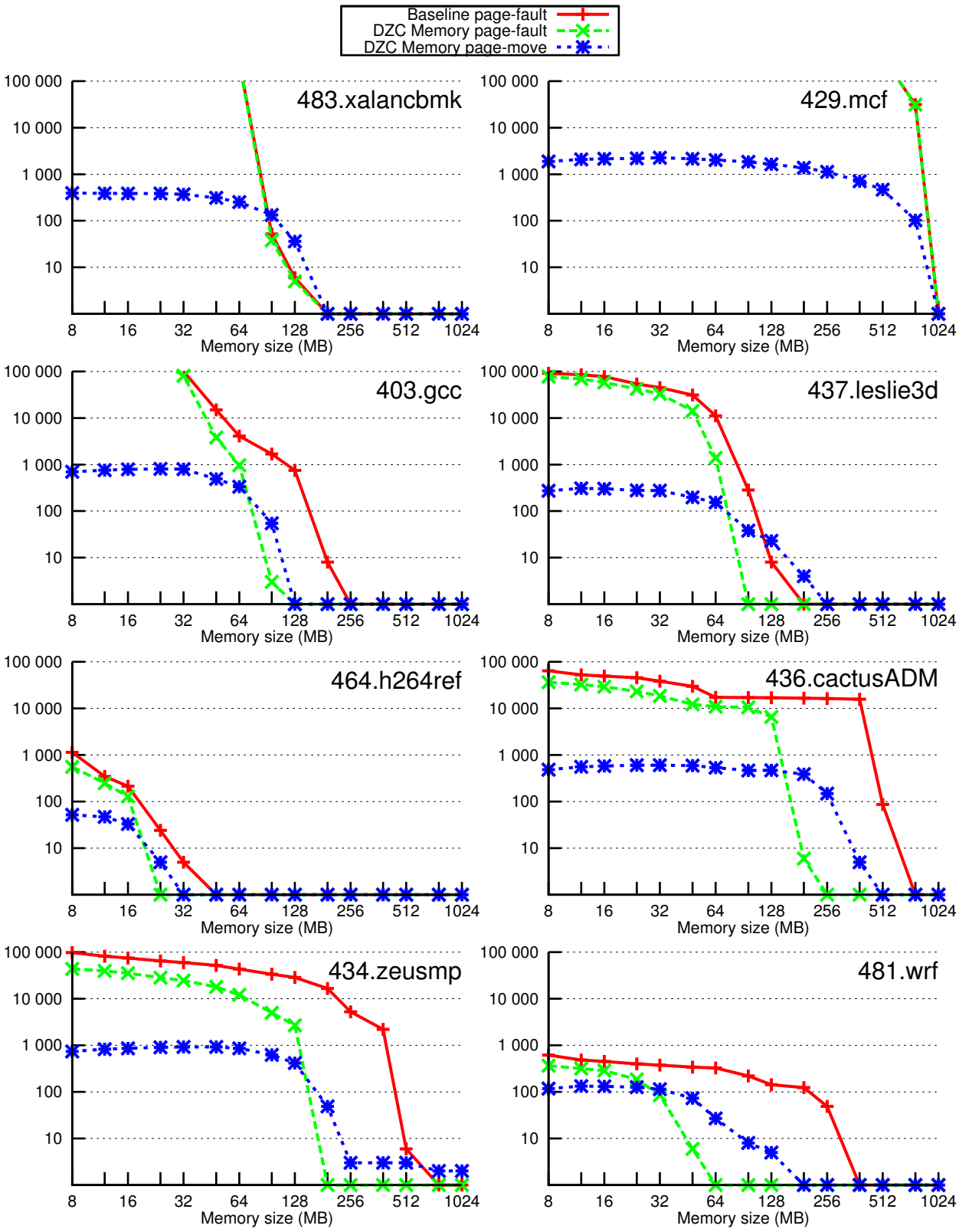

Figure 6: Page Faults and Page Moves per billion instructions.

moves might be encountered (a few tens per billion instructions), but 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.

Note that these simulation results point out that there is no need for optimizing page moves in the cache controller.

5.4 Main memory average access time

In this section, we assume that the main memory size is sufficient to accommodate the complete working set of the application. As pointed out in Section 5.1, two contradictory phenomena influence the average memory access time on the DZC memory, potential misses on the UCT cache and direct servicing of null blocks.

For estimating the main memory access time, we consider here that the main memory read access time can be roughly divided in three consecutive phases, processor-to-memory-controller, round-trip from memory controller to DRAM components and memory-controller-to-processor. When using DZC memory, on a UCT cache miss, an extra round-trip to the DRAM components is needed to load the P-page descriptor in the UCT cache. On the other hand, on a read access on a null block hitting on the UCT cache, the round-trip to the DRAM components is saved.

In the simulation results presented in this section, we consider that processor to memory controller path and memory-controller to processor path both lasts 25 cycles and that the round-trip to memory components lasts 200 cycles. That is a hit on a null block in the memory controller is serviced in 50 cycles, a hit on a non-null block in the memory controller is serviced in 250 cycles and a miss on the controller is serviced in 450 cycles. In practice, when the rate of accesses to null blocks is four times higher than the rate of misses on the UCT cache, the average memory access time is lower on the DZC memory than on the uncompressed memory.

Table 2 illustrates the relative memory access time for the DZC memory and the uncompressed memory for various UTC cache sizes. Figure 7 illustrates the average main memory access time contribution per instruction, i.e. the number of main memory cycles used per instruction for respectively the DZC memory and the uncompressed memory with a reasonable UTC Cache size of 328 KB (2K entries).

For most applications, the use of 328 KB DZC memory reduces the average memory access time. The reduction is spectacular for some benchmarks, in particular *459.GemsFDTD*, even a 20 KB UCT cache divide average access latency by 3. For applications exhibiting a high dynamic null-block ratio (Figure 8), many round-trips to the DRAM components are saved. It should be noted that the static null-block rates (Figure 2) may be quite different from the dynamic null-block rate e.g. *401.bzip2*.

Among our benchmarks, five applications *429.mcf*, *433.milc*, *458.sjeng*, *471.omnetpp* and *483.xalancbmk* exhibit a higher memory access time when using DZC than when using an uncompressed memory. *429.mcf*, *433.milc*, *471.omnetpp* and *483.xalancbmk* exhibit very low null block rates (Figure 8) and a significant miss rate on the UCT cache. *458.sjeng* exhibits a quite high dynamic null-block rate (38 %), but also a very high UCT cache miss rate (around 30 %).

In order to avoid such possible performance losses on applications exhibiting a very low null block rate or a very high UCT cache miss rate, a possible solution would be to disable compression at run-time when the ratio of number of null blocks serviced by the UCT cache on the number of UCT cache misses falls below some threshold.

Application	UCT Cache Size (in KB)						
	20	41	82	164	328	656	1312
400.perlbench	*1.12*	*1.08*	*1.03*	**0.99**	**0.96**	**0.95**	**0.95**
401.bzip2	**0.95**	**0.87**	**0.85**	**0.85**	**0.85**	**0.85**	**0.85**
403.gcc	**0.92**	**0.91**	**0.90**	**0.89**	**0.87**	**0.85**	**0.84**
410.bwaves	**0.90**	**0.90**	**0.90**	**0.90**	**0.90**	**0.90**	**0.90**
416.gamess	**0.78**	**0.76**	**0.73**	**0.72**	**0.72**	**0.72**	**0.72**
429.mcf	*1.20*	*1.16*	*1.11*	*1.05*	*1.03*	*1.02*	*1.01*
433.milc	*1.07*	*1.06*	*1.03*	*1.02*	*1.01*	*1.01*	*1.01*
434.zeusmp	**0.58**	**0.52**	**0.47**	**0.44**	**0.43**	**0.43**	**0.43**
435.gromacs	**0.95**	**0.94**	**0.94**	**0.93**	**0.93**	**0.93**	**0.93**
436.cactusADM	**0.58**	**0.57**	**0.56**	**0.55**	**0.55**	**0.55**	**0.55**
437.leslie3d	**0.86**	**0.85**	**0.85**	**0.85**	**0.84**	**0.84**	**0.84**
444.namd	**0.86**	**0.84**	**0.83**	**0.82**	**0.82**	**0.82**	**0.82**
445.gobmk	*1.01*	**0.96**	**0.88**	**0.83**	**0.82**	**0.82**	**0.82**
447.dealII	**0.90**	**0.90**	**0.89**	**0.88**	**0.88**	**0.88**	**0.88**
450.soplex	*1.32*	*1.27*	*1.17*	*1.04*	**0.96**	**0.93**	**0.93**
453.povray	**0.81**	**0.78**	**0.74**	**0.72**	**0.72**	**0.72**	**0.72**
454.calculix	**0.95**	**0.94**	**0.94**	**0.93**	**0.93**	**0.93**	**0.93**
456.hmmer	*1.01*	*1.00*	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
458.sjeng	*1.19*	*1.18*	*1.15*	*1.12*	*1.06*	**0.97**	**0.84**
459.GemsFDTD	**0.32**	**0.29**	**0.27**	**0.25**	**0.25**	**0.25**	**0.24**
462.libquantum	**1.00**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	**0.98**
464.h264ref	*1.04*	*1.01*	**0.99**	**0.98**	**0.98**	**0.98**	**0.98**
465.tonto	**0.97**	**0.94**	**0.90**	**0.88**	**0.88**	**0.88**	**0.88**
470.lbm	*1.00*	*1.00*	*1.00*	**1.00**	**1.00**	**1.00**	**1.00**
471.omnetpp	*1.24*	*1.23*	*1.22*	*1.20*	*1.17*	*1.13*	*1.00*
473.astar	*1.45*	*1.36*	*1.19*	*1.04*	**0.99**	**0.98**	**0.97**
481.wrf	**0.37**	**0.36**	**0.35**	**0.34**	**0.34**	**0.34**	**0.34**
482.sphinx3	**0.98**	**0.98**	**0.98**	**0.98**	**0.97**	**0.97**	**0.97**
483.xalancbmk	*1.13*	*1.12*	*1.11*	*1.10*	*1.08*	*1.05*	*1.01*

Table 2: Relative access time on DZC memory and uncompressed memory for different UCT cache sizes. Bold values are faster accesses.

6. 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 [8], null blocks are only represented by a single bit. For representing non-null blocks, the main memory is treated as a decoupled sectored cache [20].

Unlike the IBM MXT technology [21], the DZC memory is compatible with a 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* [9], the DZC memory allows a smooth management of compressed data size changes on writes.

Our experimental simulations have shown that the DZC 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.

Memory compression hardware generally results in some performance loss when the working set of the application

Figure 7: Average main memory access time contribution per instruction with a 328 KB UTC Cache

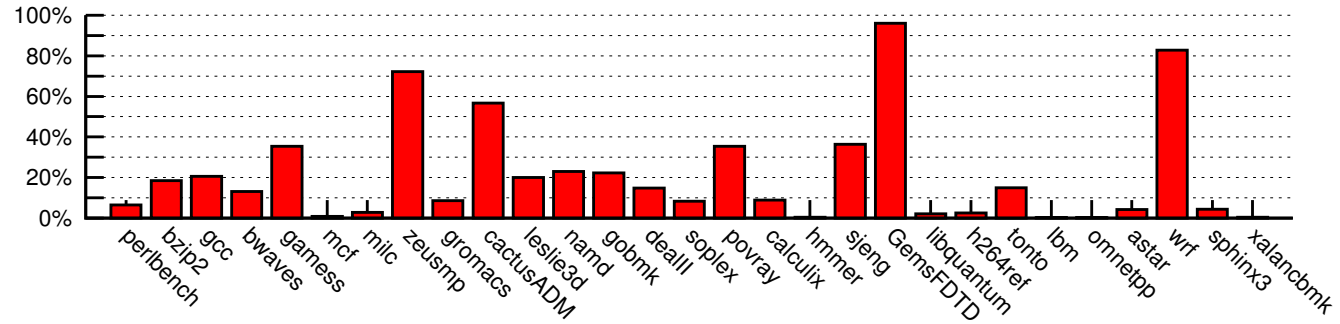

Figure 8: Dynamic null block access rate on main memory

already fits in the memory. An artifact of the DZC memory is the use of a translation cache, the UCT cache, in the memory controller. This UCT cache translates the address in uncompressed memory in the address in the compressed memory. The UCT cache is able to serve read accesses to null blocks without effectively reading the main memory. For most applications, this results in a reduced average main memory access time.

7. ACKNOWLEDGEMENTS

This work was partially supported by an Intel research grant and by the European Commission in the context of the SARC integrated project #27648 (FP6).

8. 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 annual international symposium on High-Performance Computer Architecture*, pages 73–81, Monterrey, NL, Mexico, Jan. 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, Munich, Germany, June 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 14th International Conference on Parallel and Distributed Systems*, pages 303–310, Melbourne, VIC, Australia, Dec. 2008. IEEE Computer Society.

[5] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained java environments. In *OOPSLA '03: Proceedings of the 18th annual conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 282–301, Anaheim, CA, United States, Oct. 2003. ACM.

[6] 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, São Paulo, SP, Brazil, Nov. 2003. IEEE Computer Society.

[7] 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, San Diego, CA, United States, Jan. 1993. USENIX Association.

[8] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *ICS '09: Proceedings of the 23rd annual International Conference on Supercomputing*, pages 46–55, Yorktown Heights, NY, United States, June 2009. ACM.

[9] 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, Madison, WI, United States, June 2005. IEEE Computer Society.

[10] P. A. Franaszek, P. Heidelberger, D. E. Poff, and J. T. Robinson. Algorithms and data structures for compressed-memory machines. *IBM Journal of Research and Development*, 45(2):245–258, Mar. 2001.

[11] P. A. Franaszek and J. T. Robinson. Design and analysis of internal organizations for compressed random access memories. *Technical Report RC 21146, IBM T. J. Watson Research Center*, Oct. 1998.

[12] P. A. Franaszek and J. T. Robinson. On internal organization in compressed random-access memories. *IBM Journal of Research and Development*, 45(2):259–270, Mar. 2001.

[13] P. A. Franaszek, J. T. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. In *DCC '96: Proceedings of the 6th annual Data Compression Conference*, pages 200–209, Snowbird, UT, United States, Mar. 1996. IEEE Computer Society.

[14] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *HPCA '05: Proceedings of the 11th annual international symposium on High-Performance Computer Architecture*, pages 201–212, San Francisco, CA, United States, Feb. 2005. IEEE Computer Society.

[15] J.-S. Lee, W.-K. Hong, and S.-D. Kim. A selective compressed memory system by on-line data decompressing. In *Euromicro '99: Proceedings of the 25th annual Euromicro Conference*, volume 1, pages 224–227, Milan, Italy, Sept. 1999. IEEE Computer Society.

[16] A. Moshovos and A. Kostopoulos. Memory state compressors for giga-scale checkpoint/restore. In *PACT '05: Proceedings of the 14th annual international conference on Parallel Architectures and Compilation Techniques*, pages 303–314, St. Louis, MO, United States, Sept. 2005.

[17] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed memory. In *IPDPS '01: Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 66–71, San Francisco, CA, United States, Apr. 2001. IEEE Computer Society.

[18] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. In *PLDI '10: Proceedings of the 2010 international conference on Programming Language Design and Implementation*, pages 471–482, Toronto, ON, Canada, June 2010. ACM.

[19] J. B. Sartor, M. Hirzel, and K. S. McKinley. No bit left behind: the limits of heap data compression. In *ISMM '08: Proceedings of the 7th annual International Symposium on Memory Management*, pages 111–120, Tucson, AZ, United States, June 2008. ACM.

[20] 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, Chicago, IL, United States, Apr. 1994. IEEE Computer Society.

[21] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland. IBM memory eXpansion technology (MXT). *IBM Journal of Research and Development*, 45(2):271–285, Mar. 2001.

[22] 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, Monterey, CA, United States, June 1999. USENIX Association.

[23] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.