

Buri: Scaling Big-memory Computing with Hardware-based Memory Expansion

JISHEN ZHAO*, University of California, Santa Cruz

SHENG LI*, Intel Labs

JICHUAN CHANG*, Google

JOHN L. BYRNE, HP Labs

LAURA L. RAMIREZ, HP Labs

KEVIN LIM*, Google

YUAN XIE, University of California, Santa Barbara

PAOLO FARABOSCHI, HP Labs

Motivated by the challenges of scaling up memory capacity and fully exploiting the benefits of memory compression, we propose Buri, a hardware-based memory compression scheme, which simultaneously achieves cost efficiency, high performance, and ease-of-adoption. Buri combines (1) a self-contained, ready-to-adopt hardware compression module, which manages metadata compression and memory allocation/relocation operations, (2) a set of hardware optimization mechanisms, which reduce the area and performance overheads in accommodating the address indirection required by memory compression, and (3) lightweight BIOS/OS extensions used to handle exceptions. Our evaluation with large memory workload traces shows that Buri can increase capacity by 70%, in addition to the compression ratio already provided by database software.

Categories and Subject Descriptors: B.3.1 [Memory Structures]: Semiconductor Memories

General Terms: Design, Performance

Additional Key Words and Phrases: Memory, Compression, Performance, Big Data, Scalability

ACM Reference Format:

Jishen Zhao, Sheng Li, Jichuan Chang, John L. Byrne, Laura L. Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi, 2015. Buri: Scaling Big-memory Computing with Hardware-based Memory Expansion. *ACM Trans. Architect. Code Optim.* 9, 4, Article 39 (March 2015), 22 pages.

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

1. INTRODUCTION

Low-latency web services and real-time analytics demand fast access to large-scale data stores. In-memory computing, which deploy application's primary data stores in main memory, prospers in this new scenario. In-memory databases, such as SAP HANA [SAP 2014] and VoltDB [VoltDB 2014], promise up to $1000\times$ speedup over disk-based implementations; web-scale applications, such as memcached [Memcached 2014], Spark [Foundation 2014b], and Giraph [Foundation 2014a], bring in-memory computing into both front-end and back-end tiers. While big-memory workloads continue to require terabytes of or even much larger memory capacity, today's system architecture reveals a drastic mismatch towards this requirement.

*This work was done while the authors were with HP Labs.

Author's addresses: J. Zhao, Computer Engineering Department, University of California at Santa Cruz, email: jzhao31@ucsc.edu; S. Li, Intel Labs, email: sheng.sli@gmail.com; J. Chang, Google, email: jichuan@gmail.com; J. L. Byrne, HP Labs, email: john.l.byrne@hp.com; L. L. Ramirez, HP Labs, email: laura.ramirez@hp.com; K. Lim, Google, email: kevin.t.lim@gmail.com ; Y. Xie, Department of Electrical and Computer Engineering University of California at Santa Barbara, email: yuanxie@ece.ucsb.edu; P. Faraboschi, HP Labs, paolo.faraboschi@hp.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ACM 1544-3566/2015/03-ART39 \$15.00

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

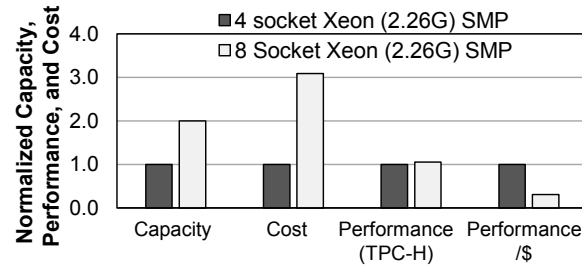


Fig. 1. System performance, cost, and efficiency of running a commercial large-scale in-memory database on large SMPs. With the 2X improvement on memory capacity, the system efficiency (measured as performance/\$) shows $3\times$ degradation, which indicates the inefficiency of scaling memory capacity through scaling up systems. Detailed system configuration can be found in Section 8.7.

Today, memory is attached to CPU memory channels with limited off-chip pins. One way to increase memory capacity is to design scale-up systems by *adding more CPU sockets* to obtain additional memory channels. Although DRAM prices continue to drop over decades (at less than \$1/Gb in 2013), purchasing extra CPU sockets significantly increase the total system cost. The extra hardware cost adds significant, often nonlinear overheads to the system budget, making the scale-up solution only suitable for accommodating software with high license fees. We realize that value-based pricing has a significant effect on the nonlinear cost increase of large-socket-count servers. While we approach the problem in an academic way, taking this cost effect into account largely reflects the reality and has significant impact to the end users. We believe that it is important to include real-world practical consideration when discussing new technology.

Furthermore, scale-up hardware introduces NUMA latency with remote-socket memory accesses, leading to substantial slowdown and performance idiosyncrasies [Hardavellas et al. 2011]. Software performance gains diminishing returns with increasing number of CPU sockets and cores due to Amdahl's law [Esmailzadeh et al. 2011] and non-compute bottlenecks [Lim et al. 2013], [Basu et al. 2013]. Our measurements on real systems shown in Figure 1 highlights the inefficiencies of scaling memory capacity with scale-up solutions. When running TPC-H [TPC 1988] on a commercial large-scale in-memory database, a 4-socket system with 2TB memory (its maximum memory capacity) cannot hold our desired data set. Therefore, we need to migrate to a larger SMP system with 8 sockets that support 4TB of memory. While this migration offers doubled memory capacity, the system cost increases by $3\times$ as shown in Figure 1. Unfortunately, the raw performance of running TPC-H queries only yields 5.4% speedup, when we move from the 4-socket to the 8-socket system. While the TPC-H queries running on top of commercial large-scale in-memory databases do benefit marginally from the increased computation resource*, they are negatively impacted by the extra NUMA latency of the 8-socket SMP.

Alternatively, scale-out systems [Dean and Ghemawat 2004] leverage commodity clusters to obviate the requirements for expensive SMP hardware. The scale-out paradigm, however, can only accommodate a limited subset of use cases with data-parallel, communication-light algorithms. With the rest of the big-memory applications, the scale-out solution either requires programmers to rewrite their software or system architects to adopt low-latency interconnect fabric, shifting the memory scaling burden to the software or communication infrastructures. Similarly, recent proposals on increasing memory capacity with disaggregated memory [Lim et al. 2009], [Magenheimer et al. 2009] and DRAM/NVM hybrid memory [Qureshi et al. 2009] either require complex software changes or rely on long-term technology development and integration efforts. These factors strongly motivate a practical memory scaling architecture which can be applied to future software and technologies.

*For TPC-H queries, there are more software threads than the supported hardware threads in both systems. Therefore, the 8-socket system does benefit from the increased computing resources.

In response, recent system implementations adopt memory compression as a new dimension in scaling in-memory computing. For example, modern databases use columnar table compression to reduce memory footprint and transfer bandwidth with multi-core hardware [Zukowski et al. 2006], [Kallman et al. 2008]. New client OS [Apple 2014], [Wilson et al. 1999] compresses idle pages to reduce swapping. Software compression, however, works only for limited workloads or use-cases, and incurs the costs of software rewrite, CPU processing overhead and memory access latency. Prior work on hardware memory compression [Tremaine et al. 2001], [Ekman and Stenstrom 2005] aims to boost capacity without such software overhead, but they all face challenges in achieving ease-of-adoption and/or high performance.

Motivated by the demand for processor-independent memory scaling and the gap in fully realizing memory compression benefits, we propose Buri[†], a hardware-based compression scheme which simultaneously achieves cost efficiency, high performance, and ease-of-adoption. To the best of our knowledge, Buri offers the first implementable and low-cost memory compression mechanism that achieves both transparency and high performance on big data workloads. A large variety of memory compression schemes have already been proposed, yet little has been adopted by modern big data workloads due to their complexity and/or performance degradation. We identify the promising aspects of prior proposals and combines them in a high-performance, practical solution. Furthermore, we propose a set of new technical solutions to optimize performance, including metadata prefetching, metadata cache-aware memory scheduling, page buffer, and overflow handling.

In particular, we make the following contributions:

- We propose a self-contained hardware compression architecture that manages common operations for compression metadata, memory indirection, and memory allocation and relocation. We also present various integration options, and evaluate their performance trade-offs. The proposed architecture is orthogonal to system- and technology-level techniques, including disaggregated memory [Lim et al. 2009], [Magenheimer et al. 2009] and DRAM/NVM hybrid memory [Qureshi et al. 2009]. Thus they can be easily combined with Buri architecture to achieve synergy among different techniques with maximized benefits.
- We introduce a set of hardware optimization mechanisms to reduce the area and latency overheads of Buri’s hardware-based memory indirection. To secure correctness, we also present a set of lightweight BIOS/OS extensions to address a number of worst-case exceptions.
- We evaluate our Buri design using big-memory workload traces collected from high-end SMP servers. Our results demonstrate that Buri can increase effective memory capacity by 70% in addition to the software compression algorithms already applied by in-memory database applications. Furthermore, Buri achieves $1.9\times$ improvement in performance per cost on average.

2. BACKGROUND AND RELATED WORK

In-memory workloads require big and fast data processing, the combination of which makes memory capacity a potential scaling bottleneck. Today’s hardware scaling paradigms, scale-up SMPs and scale-out clusters, increase memory capacity by increasing the number of processor sockets and the size of the interconnect network in the system. To different degrees, these architectures have coupled memory scaling with compute and communication scaling, which can be unnecessarily costly and technically challenging.

Compression, on the other hand, offers a new dimension in scaling memory capacity that does not require scaling the compute and network infrastructures in tandem. With compression ratios averaging between $2\text{--}4\times$ (up to $50\times$ for columnar databases), memory compression provides multiplicative benefits, on a comparable scale to SMP and clustering schemes. For power and cost-constrained systems, compression improves efficiency; for capacity-limited systems, compression enables larger problem sizes and higher performance that would have been impossible to achieve without developing and deploying new architectures.

[†]We name our design Buri after the Scandinavian mythology figure who can expand or shrink to any size he wishes.

2.1. Why Hardware-based Memory Compression is Hard

Considering the adoption of software-based compression schemes and their drawbacks of program rewriting and CPU/license costs, and the demonstrated efficiencies of hardware accelerators, it seems natural to offload memory compression burdens to the hardware. Surprisingly, hardware-based compression, including general-purpose memory compression proposals [Tremaine et al. 2001], [Ekman and Stenstrom 2005], [Sardashti and Wood 2013] and code compression designs for embedded systems [Wolfe and Chanin 1992], [Lekatsas and Wolf 1998], [Lin et al. 2004], have not been widely adopted in today's systems[‡]. To understand this effect, below we describe the major challenges of hardware memory compression and their implications on implementation and adoption.

Overheads of addressing compressed data: Processors access data at a cache block granularity following the virtual-physical page address translation. Locating data in uncompressed memory is simplified by regular-sized cache blocks and OS pages. In contrast, compressed pages and cache blocks can have variable lengths and therefore compacting the data breaks addressing regularity. As a result, keeping track of the addresses of large amounts of compressed data is non-trivial. There are two main approaches to provide this tracking: (1) Modifying existing designs to address compressed data, which entails changes to the OS, TLB, and memory controllers. Requiring such comprehensive, concerted changes from both software and hardware ecosystems creates significant adoption hurdles. (2) Encapsulating the addressing complexities in a self-contained hardware module (e.g., IBM MXT [Tremaine et al. 2001]). A naive implementation, however, could add significant space, time, and complexity overheads. An optimized while easy-to-adopt approach needs to reduce such overheads.

Additional memory management complexity: The memory management functionalities now need to understand the concept of variable-length pages and value-dependent page resizing to provide an optimized solution. Making the OS compression aware is a conceptually straightforward approach, but would require changes to memory allocation and deallocation, and page overflow and recomaction. Invoking the OS to handle dynamic page resizing and relocation incurs overhead. The non-uniform compression ratios across large memory regions further complicate optimizations such as direct segments [Basu et al. 2013], whose benefits of reducing TLB miss overhead is critical for big memory workloads. Alternatively, hiding such details from the OS would burden the hardware with memory management functions. Ideally, a balanced approach should let the hardware handle the common cases efficiently, preferably without OS intervention, while using software to handle rare events (e.g., preventing the OS from overcommitting compressed memory).

Implications on hardware optimizations: TLB and processor caches are two examples of hardware optimizations that assume fix-sized granularities. Exposing compressed main memory to software would require non-trivial changes to these critical hardware structures. For physically-tagged caches, this implies the need for fast calculation of the actual address of compressed data without lengthening cache access latency. Augmenting TLBs with compressed data addresses means inheriting the inefficiencies of page-based address translation for big memory [Basu et al. 2013], essentially shifting the cost of memory indirection to TLB miss handling. These factors motivate hardware-based management of address space and compressed memory.

To address these challenges, our proposal uses the combination of (1) hardware-based memory expansion for ease-of-adoption, (2) hardware optimizations for reducing indirection overheads, and (3) lightweight BIOS/OS changes to handle rare events.

2.2. Related Work

Prior memory compression designs fall into two categories: hardware-based design and software-based design. Most software-based schemes are implemented in application programs or at the OS/runtime level.

[‡]Current implementations are primarily for accelerating software compression, e.g., compression instructions in IBM Power and Fujitsu SPARC, or domain-specific ASIC IPs or I/O cards for packet compression.

Hardware-based Memory Compression: IBM Memory Extension Technology (MXT) [Tremaine et al. 2001] is a primary example of hardware-based memory compression. To avoid complex changes to the OS and cache tags, MXT adopted an additional level of address indirection, a real address space, between virtual and physical address spaces. Accessing compressed memory now constitutes at least two separate memory accesses: one for real-physical address translation and another for actual data access. For compression ratio, MXT chooses a Lempel-Ziv algorithm whose block-level decompression latency can be as high as 64 cycles. The main optimization to reduce access latency is a 32MB last-level cache, indexed by real addresses for storing hot, decompressed data. A cache hit can avoid both the address indirection and decompression latency. This optimization, as shown later in the results section, is insufficient for today's in-memory workloads, whose working sets are much bigger than large processor caches [Ferdman et al. 2012].

Hardware-software codesign for memory compression: Proposals in this category modify the OS and hardware to be compression-conscious, and trade compression ratio with latency and complexity improvements. For example, Ekman et al. proposed a robust main-memory compression scheme [Ekman and Stenstrom 2005], where the OS maps the virtual address space directly to a compressed physical address space. This design uses Frequent Pattern Compression (FPC) to reduce its decompression latency (5 cycles), much lower than MXT's Lempel-Ziv compression algorithm. Pekhimenko et al. proposed Linearly Compressed Pages (LCP) [Pekhimenko et al. 2013] to mitigate the performance penalty of physical addresses calculation. LCP compresses all cache lines within a page to the same size to reduce the address calculation latency as well as metadata storage overhead. While these designs leverage processor and OS supports to directly address compressed data, their efficiency heavily depends on TLB efficacy, which has become a major performance bottleneck for big memory workloads [Basu et al. 2013], [Lim et al. 2013]. More critically, these designs face practical adoption hurdles because they require concerted modifications of both the OS and processor hardware. For example, both proposals [Pekhimenko et al. 2013], [Ekman and Stenstrom 2005] have to modify OS page tables for storing compression metadata. Consequently, the OS virtual memory manager should be made compression aware. On the hardware side, these designs mandate processor changes to reduce cache access latency: e.g., LCP [Pekhimenko et al. 2013] modifies the tag structures of all cache levels (including performance-critical L1 caches), and Ekman et al. [Ekman and Stenstrom 2005] proposes an additional special TLB. The difficulties of realizing such prevalent modifications is a key issue in adopting such compression designs.

Software-based memory compression: We have briefly discussed software-based database compression schemes [Zukowski et al. 2006], [Kallman et al. 2008] and their hardware acceleration (e.g., [Franaszek et al. 2006]). Similar to multimedia compression, the user application controls the compression and access to data, with the flexibility in choosing compression algorithms and policies. The OS and runtime can also compress memory for a wider range of applications. Such designs exploit idle processor cores to compress rarely-accessed cold pages or swap space. Using the OS as the single control point, they are easy to adopt without disrupting application software. Unfortunately, software-based compression can increase access latency, especially when moving pages between compressed and uncompressed regions. Software compression also requires additional CPU cycles. For example, running IBM Active Memory Expansion (AME) [IBM 2012] requires at least 13% additional CPU cycles. When processors are highly utilized, this implies application-level performance degradation. For enterprise software that charges license fees based on system core count, utilizing CPU resources for compression can incur a significant total-cost-of-ownership (TCO) penalty.

Table I compares Buri with the two mostly related previous work, MXT [Tremaine et al. 2001] and LCP [Pekhimenko et al. 2013].

3. BURI DESIGN

To achieve hardware-based memory compression for big memory workloads without sacrificing ease-of-adoption or performance, we draw inspiration from prior proposals and build an effective, self-contained design solution. Similar to MXT, Buri expands memory with hardware-based mecha-

Table I. Comparison of Buri, MXT [Tremaine et al. 2001], LCP [Pekhimenko et al. 2013].

	Hardware Modification	OS Modification Needed	Location of Performance Optimization Components	Compression Algorithm	Performance of Big Data Workloads
MXT	Last-level Cache, Memory Controller	Light	Last-level Cache	Lempel-Ziv	Low
LCP	Tag Structures in Caches	Heavy	Memory Controller	Any	High
Buri	Memory Controller or Memory-side Controller	Light	Memory-side Controller	Zero-aware, BDI, FVC	High

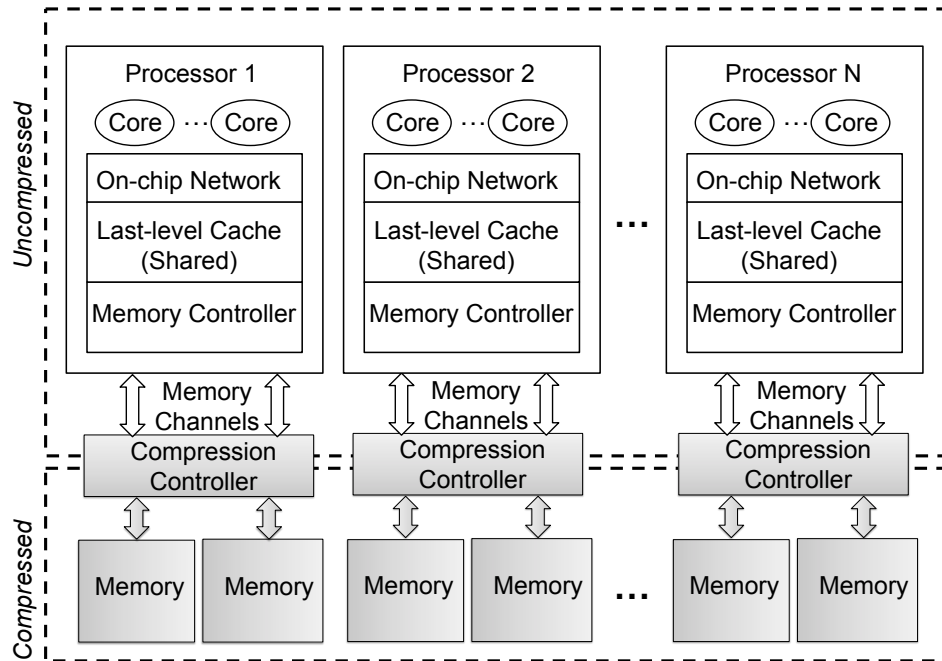


Fig. 2. An example of Buri's memory hierarchy. Buri's compression controller sits between uncompressed caches and compressed memory.

nisms. It encapsulates common-case functionalities of addressing and managing compressed memory in a self-contained compression controller located between the processor cache and the main memory. The particular functions and the location of the compression controller make Buri easy to adopt, without the need to modify the OS or processor cores. Similar to LCP, Buri trades-off higher compression ratios with simple, fast algorithms by employing uniformed page layout. To simplify its implementation and speed up common operations, Buri limits data relocation and memory management functions within large chunks. For rare events, Buri avoids hardware complexity by leveraging a few lightweight BIOS/OS modules.

3.1. Memory Hierarchy

Figure 2 shows an example multicore cluster system employing Buri. Buri's compression controllers divide the memory hierarchy into two regions: uncompressed caches and the compressed memory. A multi-processor system can incorporate multiple compression controllers (one per processor socket). The compression controllers perform compression and decompression operations to the data flow between the uncompressed caches and the compressed memory.

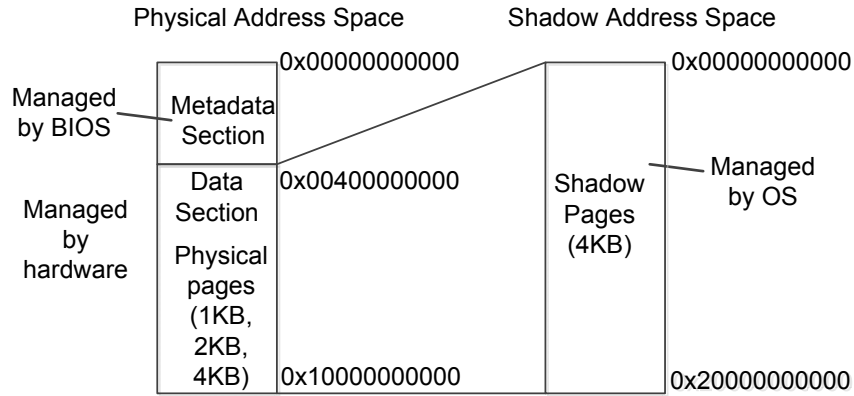


Fig. 3. Layout of physical and shadow address spaces.

We allow the compression controllers to be integrated in two locations: (1) in processor's memory controllers and (2) as memory-side modules, which are attached to the processor via coherence interface (e.g., e.g., Intel QPI, ARM AXI, and AMD HyperTransport). Multiple compression controllers can be used to compress and decompress the memory traffic going through different memory channels. Different from MXT [Tremaine et al. 2001], Buri compression controller adopts simple compression algorithms (see Section 4 for details of these algorithms) that only cost 5-7 cycles to perform a compression/decompression operation. Furthermore, we improve system performance by employing a set of optimization mechanisms in each compression controller (see Section 5 for details of the optimizations).

3.2. Address Space Management

Similar to MXT, Buri adopts one layer of address indirection. We define the following address spaces:

- **Virtual address:** It is visible to software applications the same way as that of the virtual address in systems without memory compression.
- **Shadow address:** It is visible to the OS the same way as that of the physical address in systems without memory compression.
- **Physical address:** It identifies the real physical location of a page in the memory.

Software applications access the memory using virtual addresses. Processor caches are managed by shadow addresses. The TLB stores address translations between virtual and shadow addresses. Compression controllers intercepts shadow addresses and calculates physical addresses. Buri enables the expansion of the shadow address space to be much larger than the physical address space. With a compression ratio of R , a Buri-enabled system can offer a shadow address space that is $R \times$ the size of the installed physical memory devices.

We define pages in virtual, shadow, and physical address spaces as virtual, shadow, or physical pages, respectively. A shadow page can be implemented as a 4KB OS page, a superpage with the size of multiple OS pages, or a direct segment [Basu et al. 2013].

Physical address space organization: Each compression controller manages the memory devices installed at a socket and the corresponding physical address space. The compression controller divides its physical address space into a metadata section and a data section (Figure 3). The metadata section stores address mapping information between the shadow and physical pages. It also stores auxiliary information of each physical page, including the compression algorithm used and the pointers referenced to the physical data blocks that form the physical page. (Details about the layout of the metadata section will be described in Section 4.) The metadata section is only visible

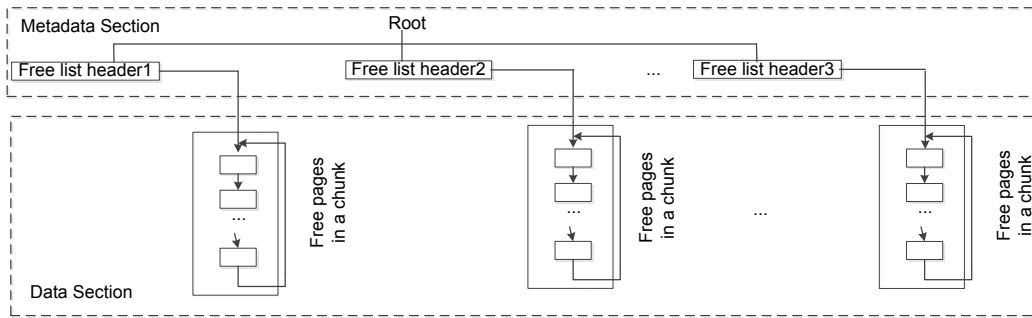


Fig. 4. A logical view of free pages managed in chunks. The data section is managed as a hierarchy.

and directly accessed by Buri hardware and BIOS code. The data section stores physical pages in compressed form. Depending on varying data values, compression ratio of a page can dynamically change during application execution. The size of physical pages can also vary over time. In order to be flexible and accommodate the varying physical page sizes, Buri dynamically allocates and deallocates 1KB blocks to each physical page, i.e., the physical pages consist of one or multiple 1KB data blocks.

Data section organization: Manipulating (searching, allocating, and deallocating) 1KB blocks in a huge physical memory at the scale of terabytes requires non-trivial effort and can incur significant performance overhead. To optimize the performance of physical page allocation and deallocation, we partition the data section into fix-sized large chunks (Figure 4). Each chunk consists of contiguous 1KB data blocks. Buri page allocation mechanism ensures that most of the physical pages will consist of 1KB blocks in the same chunk. We evaluated various chunk sizes, and find that 16MB chunk size offers the best system performance with various applications. Each chunk is organized as a logical linked list. It incorporates an embedded free list to keep track of unused 1KB data blocks (free blocks) within the chunk. Each free block stores a pointer to the next free block. As illustrated in Figure 4, the metadata section also stores the head pointers of each chunk. As a result, the physical address space is managed in a two-level hierarchy: a pool of chunks and a pool of 1KB data blocks in each chunk. With this two-level hierarchy, Buri can quickly identify a free block either in the same chunk or in a neighboring chunk without the space and traversal overheads of maintaining a huge amount of global free-space information. Different from MXT which adopts a global pool of free blocks, Our chunk-based design is critical to achieve scalable memory management in big memory systems.

Physical page layout: We adopt linearly compressed pages (LCP) [Pekhimenko et al. 2013] as the basic framework to design our physical page layout. A compressed physical page is divided into a compressed region and an optional uncompressed region. As shown in Figure 5, to eliminate the complexity of manipulating variable-sized cache lines, Buri stores all the cache lines of a page in slots with a fixed size. The slot size is determined by the compression algorithm and dependent on page value regularity. A cache line with higher compression ratio still occupies its entire slot, leaving the remaining space empty. If a cache line cannot be compressed to fit into its slot, we store the original values in the uncompressed region of the physical page. The corresponding slot in the compressed region will store a pointer to the uncompressed line instead of its compressed value. This organization allows simple hardware implementation to quickly locate data within a page. Yet it can potentially reduce compression ratio, especially when cache lines in a page have a non-uniform compressibility. Yet our evaluation shows that this scheme achieves high compression ratios with various real-world in-memory workloads.

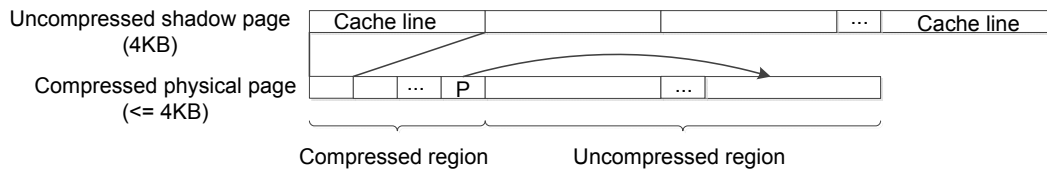


Fig. 5. Page layout. Within a page, all cache lines are stored in same-sized compressed slots. A cache line that cannot be compressed into its slot is stored in the uncompressed region, referenced by a pointer stored in the slot.

3.3. Hardware-based Memory Management

Buri hardware can handle page allocation, page deallocation, and compression overflow without OS intervention.

Page allocation: Physical pages are setup when they are accessed for the first time, in one of the following scenarios. A page can be initialized or zero-cleared by the OS, or directly accessed by application software. It can also be memory-mapped from the file or I/O system, and potentially receive DMA transfers. Triggered by the first access to a shadow page, the compression controller will recognize it as an unmapped page and allocate a 4KB entry in its small page buffer. The temporary 4KB page will accumulate cache line values while serving memory access requests. When Buri decides to retire this page from its page buffer, it will compress current cache lines to the best possible compression ratio. The compressed page size (i.e., 1KB, 2KB, 3KB or 4KB) is used for allocating free physical blocks. The compression controller then writes compressed data into physical blocks, updates the metadata section with address indirection and compression information, and installs it in the metadata cache.

Page deallocation: Deallocating a page takes two steps. First, the physical data block of the page is returned to the free list. The corresponding entry in the metadata section is cleared, and its entry invalidated from the metadata cache. Second, if the page has been dirtied and is backed by a file or I/O device, the updates need to be flushed. A compressed page can be flushed to storage in either compressed or uncompressed format. Keeping the page compressed can save storage I/O bandwidth and space. New chipsets (e.g., from Intel) now support data decompression over storage I/O, and can be leveraged to speed up compressed storage accesses. Flushing pages uncompressed, on the other hand, makes the data portable.

Handling compression overflow: During program execution, dynamic data updates can change the compression ratios of certain cache lines or an entire page. A previously highly-compressed page can consequently become less compressible, or even uncompressible. A *compression overflow* event refers to such an event where a compressed data item no longer fits in its original storage. Compression overflow can happen at page or chunk granularities. If a page overflows, we allocate an additional 1KB block from the chunk's free list to the page. A chunk overflow occurs when it runs out of free blocks. To handle a chunk overflow, we will allocate a 1KB data block from the neighboring chunk by traversing the hierarchy of chunk-level free lists. With extremely low probability, all chunks under a compression controller can run out of free space, causing a controller-level overflow. Buri monitors chunk free lists, and preempts controller overflow by triggering BIOS/OS handlers for memory overcommit.

3.4. Working with DMA

Memory requests need to go through the compression controllers, because the compression controllers manage metadata of each memory page. Compressing DMA writes will involve complex driver modifications. To address this issue, we label the memory region accessed by DMA as uncompressible. This is easy to implement with asynchronous DMA operations (e.g., pushing I/Os to memory) by simply mark the memory pages. Buri will also compress the data after the corresponding DMA transfer completes and update the related metadata information. With synchronous DMA operations (e.g., reading from memory), the OS has to pin the specific memory region to transmit.

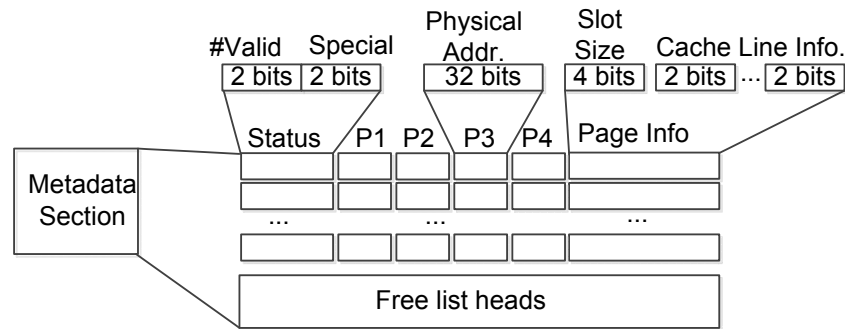


Fig. 6. Layout of the metadata section.

Therefore, we intercept the DMA setup functions in the OS and ensure that the pages being pinned are labeled as uncompressible. In this case, the last copy operation performed by the OS will trigger Buri to decompress the data being moved from a compressed to an uncompressible region.

4. IMPLEMENTATION

In this section, we describe the implementation details of Buri metadata management, compression algorithms, and OS support.

4.1. Metadata Management

As mentioned in Section 3, Buri includes a metadata section in the physical address space to store address indirection information. Figure 6 illustrates the layout of the metadata section. Besides an array of head pointers to chunk-level free lists (discussed in Section 3.2), it also stores an array of metadata entries of all shadow pages. The entries are directly indexed by the corresponding shadow page addresses. The number of entries is determined by the shadow address space size as specified by the BIOS. A metadata entry contains (1) a status field, (2) four pointers for the physical blocks storing compressed data, (3) compressed slot size, and (4) cache line information.

Buri maintains two levels of metadata in each entry, the metadata of a whole shadow page and the metadata of each cache lines in the page. The status field includes (a) special bits indicating page allocation/mapping status and (b) the number of valid pointers in the pointer region, whose value ranges from 1 to 4. The pointers contain the physical addresses of allocated 1KB blocks. To support 4TB of physical address space per compression controller with 1KB physical blocks, each pointer takes 32 bits. The cache line information records the compression type used by different cache lines in a page. Compression type 3 is reserved for uncompressed lines, and other types correspond to different compression algorithms. Combined with the compressed slot size, the compression controller can calculate the actual physical address of the cache line, and invoke the proper decompression engine. We store the metadata for both page and cache lines within a page together in one metadata entry. Because locating a cache line requires both information, this format allows the hardware to calculate data address with a single metadata access. Each entry is 264 bits, so the metadata storage overhead is approximately 0.8% for 4KB shadow pages.

4.2. Compression Algorithms

Sophisticated compression algorithms, such as Lempel-Ziv and Huffman encoding, can achieve high compression ratios. But they also cause performance degradation induced by long latencies to compress and (more importantly) decompress data. Our design deliberately chooses simple compression algorithms for their latency benefits, while improving compression ratio with the following two-level compression scheme.

Level-1: Zero cache lines. Zero cache lines are represented as a zero in the corresponding cache line information field in the shadow page's metadata entry. By having a special code for a zero cache

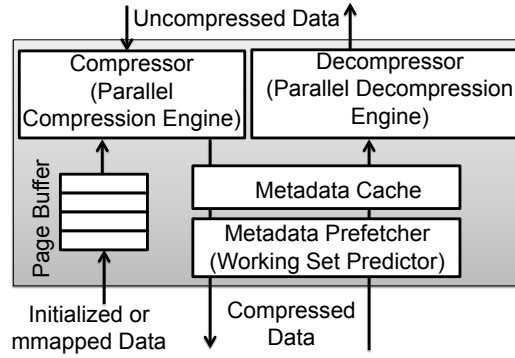


Fig. 7. Compression controller. The predictor and metadata cache can avoid extra overhead of metadata access, by prefetching and caching metadata before they are needed

line, the compression controller will know that it does not need to perform an extra read to the data region; zeroes can immediately be returned. Buri can also use the existence of only zero cache lines in a page to detect zero pages. Zero pages do not need any data blocks. Compression of a cache line stops when detecting an all-zero value.

A non-zero cache line will proceed to level-2 compression, where one of the following compression schemes will be used. With the 1KB physical block to 4KB shadow page ratio, Buri tries to compress a cache line to 1/4 of its original size. This target ratio can be configured by the BIOS based on workloads and use-case information. For simplicity, if this ratio cannot be achieved, the cache line is stored uncompressed. While our Buri design supports multiple slot sizes, our current implementation only uses the 16 B slot size; exploring the effectiveness of using other sizes is left to future work.

Level-2.1: Base-delta-immediate (BDI) [Pekhimenko et al. 2012]. We use the first segment of a cache line as base value and represent the remaining segments as the offset to the base value. Cache line compression completes if BDI can be applied to the entire line. However, if the compressed size of one of the offsets is larger than 1/4 of the original size, the cache line cannot be compressed with BDI. In that case, Buri will use a dictionary-free frequent value compression method.

Level-2.2: Dictionary-free frequent value compression (FVC): This algorithm encodes a data segment as a short special code if its value is the same as its previous segment. To this end, we developed our frequent value compression algorithm by exploiting frequent value locality [Zhang et al. 2000], [Yang et al. 2000], [Yang and Gupta 2002] found in neighboring data segments of cache lines. For example, a cache line with data segments of AABBBBC can be compressed as a sequence of: *A, code for A, B, code for B, code for B, and C*. Compared with prior work, our algorithm does not require extra storage to hold the dictionary information of frequent values.

This two-level compression scheme can achieve high compression ratios using simple algorithms. After compression, the compression type information for the cache lines is stored in the metadata section and used for decompression.

4.3. OS Support

We only use OS support to handle the rare cases of *overcommitment of compressed memory* (Section 3). To preempt overcommitting, the OS needs to be aware of the physical space consumption and preempt those events when potential overcommitment is detected. We implemented an OS patch to implement these functionalities. Compared with previous work [Ekman and Stenstrom 2005], [Pekhimenko et al. 2013], [IBM 2012], which heavily modifies the OS, this patch is easy to apply and strikes a balance between hardware complexity and ease-of-adoption.

We provide two options to monitor physical space usage: (1) the memory controller sends an interrupt to the OS, notifying that the physical memory address space is almost fully exhausted

(e.g., 80% is allocated) or (2) we set up a special memory mapped register that can be written by the compression controller and read by the OS periodically. The compression controller raises a flag in the register when physical memory space is almost fully exhausted.

To prevent overcommitment, the OS proactively starts swapping out pages to storage when predicting an overcommitment event could happen, based on the above information from the compression controllers. To achieve this, we modify the management of the low-memory watermark already incorporated in conventional Linux kernels. We modify the watermark such that its value is raised when physical memory utilization exceeds a preset threshold. The rise will trigger the OS to proactively page-swap and reclaim space, eventually increasing the number of entries in the free lists, avoiding overcommitment.

5. PERFORMANCE OPTIMIZATIONS

We design a compression controller to compress and decompress data with a two-level compression algorithm. With its modular nature, the compression controller can be integrated with either on-chip memory controller or implemented as a memory-side controller. A memory-side controller can communicate with the processor through coherent ports, e.g., Intel QPI, ARM AXI, or AMD HyperTransport, to support direct memory accesses.

Illustrated by Figure 7, the compression controller cuts metadata access overhead through a combination of hardware optimizations: simple and fast compression algorithms, a metadata cache assisted by working-set prediction and prefetching, and a small page buffer. The compression controller also employs parallel compression engines to improve compression performance. Below we provide more details on these optimizations.

Metadata cache and prefetching: With hardware-based memory compression, a single memory access now involves both metadata and data accesses. Because we incorporate metadata information and the pointers within each metadata entry, Buri can require up to two memory transactions to access a data in the worst case. This potentially increases the access latency by a factor of two or more. To address this performance issue, we develop three optimization techniques to reduce the latency overhead of hardware-based memory expansion: *two-level metadata cache*, metadata prefetching, and metadata aware memory scheduling.

- We introduce a metadata cache in the compression controller as shown in Figure 7. It caches two levels of metadata information: one level for a shadow page's metadata entry and another for pointers of uncompressed cache lines within a page. Data requests that fully hit in the metadata cache can directly calculate the physical address, and only need one memory access.
- We incorporate two simple prefetching policies that work with the two-level metadata cache. In the *first-level metadata cache*, i.e., the cache for metadata entries, we develop a history-based predictor for pages that are likely to be accessed next. Based on the prediction outcome, the compression controllers will issue prefetch commands if such metadata are not already in the cache. In the *second-level metadata cache*, we prefetch pointers of uncompressed cache lines in a page. Without prefetching, fetching an uncompressed cache line incurs two memory accesses: one to the pointer stored in a fixed slot, another to the uncompressed region. We design an *extended prefetching* mechanism that prefetches one more slot in the compressed region of a page. On access to a slot, a prefetch command will be issued when the compression controller detects that a pointer, instead of compressed data, is stored in the next slot. Instead of storing the prefetched pointer in the last-level cache or the first-level metadata cache, we store it in a separate second-level metadata cache to avoid cache misses for predicted prefetching.
- For compression controllers that are integrated with the on-chip memory controller, we further develop a metadata cache-aware memory scheduling policy to prioritize memory accesses who hit in the metadata cache. Such metadata aware scheduling policy is orthogonal to existing scheduling policy such as FR-FCFS [Rixner et al. 2000], [Rixner 2004] and PAR-BS [Mutlu and Moscibroda 2008], and can be integrated with existing designs.

Page buffer. The compression controller also integrates a page buffer as shown in Figure 7 that can store a small number of 4KB pages. Notice its purpose differs from MXT's large cache for decompressed data. To perform compression, the compression engine will buffer an entire uncompressed page and apply the two-level compression algorithms to achieve the best compression ratio. The page buffer also acts as a victim cache, and prefetching buffer for uncompressed pages.

Parallel compression and decompression schemes. We also develop parallel compression and decompression schemes as shown in Figure 7. We employ multiple compression/decompression engines in a compression controller. These parallel compression/decompression engines can apply different compression algorithms simultaneously and achieve much better throughput and latency than a serial implementation of our two-level compression algorithm.

6. OPERATION EXAMPLES

Putting it all together, we can walk through a typical operation under our design. When a program accesses a data item, the MMU translates its virtual address to a shadow address. If the page is touched for the first time, the Buri compression controller will traverse the free list of the default chunk, allocate a physical page with an appropriate size, map the shadow page, and initialize its metadata.

On read access to an already mapped page, the compression controller will receive a shadow memory address and translate it into a physical address. The compression controller uses the shadow page address to index the two-level metadata cache. Hitting in the metadata cache, Buri will calculate the physical address using the metadata, decompress the data (if needed) and send it back to the requesting processor. A metadata cache miss will trigger a read of the metadata entry from the memory. With zero cache lines, Buri will directly return a cache line of 0's to the processor. With compressed data, the compression controller will decompress the data based on its metadata information, and return the decompressed cache line. With uncompressed data, the data from the slot must be read to determine whether it is a pointer to the real data in the uncompressed region of the page. One final memory access is required to dereference that pointer and send the uncompressed data back to the processor. Write accesses follow a similar procedure to find the location of the cache line. If an updated cache line no longer fits into its original slot, Buri will store it as uncompressed and write a pointer to its location in the slot. If necessary, Buri will invoke the overflow handling described in Section 3. Various performance optimizations, including working set prediction, metadata prefetching, and metadata cache-aware scheduling policy, also occur during read and write accesses. Most of these operations manipulate data localized within a chunk, except for the case, when no available free physical blocks can be found in the chunk. When deallocating a physical page, Buri puts the memory block back to free list and writes the data value back to storage devices.

7. EXPERIMENTAL SETUP

The sheer scale of large memory systems makes architectural studies a challenging task. Full-system simulation would require months to boot a system with terabytes of memory, and often needs twice the amount of host memory than simulated systems. It is particularly difficult to simulate compression dynamic and memory management events, as they are much less frequent than other architectural events. We address this challenge through a combination of real-machine profiling with large memory workloads, trace-based simulation with reduced data sizes, and cost analysis for scaling trade-offs.

7.1. Experimental Framework

To understand scaling constraints, memory compression ratio and memory management activities, we profile large memory workloads running on 4-socket HP DL580 and 8-socket HP DL980 servers. We run these workloads with large datasets on varying number of sockets and cores to understand the runtime and NUMA effects. For brevity, we do not present the full results. But as shown in

Table II. Parameters of the simulated multi-core system.

Processor/Fab. Proc.	Intel Core i7 like / 22 nm
Cores	8(2.5GHz), 16 threads
L1 Cache (Private)	64KB, 4-way, 64B lines, 1.6ns latency
L2 Cache (Private)	256KB, 8-way, 64B lines, 4.4ns latency
L3 Cache (Shared)	12MB, 16-way, 64B lines, 10ns latency
Memory Controller	Two dual-channel memory controllers
Memory Technology	30 nm
DRAM DIMM	DDR3-1066, 40GB

Table III. Evaluated workloads.

Benchmarks	Description
Database “H”	Performing TPC-H [TPC 1988] row-store and column-store queries.
Memcached [Ferdman et al. 2012]	A memcached data caching server with client running twitter dataset.
Graph500 [Graph500 2014]	A data-intensive supercomputer application.
NAS [NASA 2009]	Computational fluid dynamics applications.

Figure 1, SMP scale-up only provides diminishing performance gains due to NUMA effects and Amdahl’s bottleneck.

We further develop a kernel module that can track page access activities and dump the content of selected OS pages. Through offline analysis, we can understand hot page statistics, and memory compression ratios.

To evaluate the performance of various memory compression designs, we use McSimA+ [Ahn et al. 2013], a Pin [Luk et al. 2005]-based cycle-accurate simulation infrastructure for multi- and many-cores. Because of in-memory nature of our workloads makes I/O and OS activities less important, we were able to collect traces for the entire execution time of various workloads using Pin for reduced datasets. McSimA+ can model various types of processor cores, caches, directories, on-chip networks, and memory channels. Table II lists the detailed parameters and architecture configurations of the processor and memory system in our experiments. The multi-core processor consists of eight out-of-order cores, each is similar to one of the Intel Core i7 cores [Intel 2014]. The processor employs a two-level hierarchical directory-based MESI protocol to maintain cache coherence.

7.2. Workloads

Table III lists the workloads we evaluated. Database “H” in a commercial large-scale in-memory database, whose name can not be disclosed due to legal restrictions. In order to make the simulation and profiling manageable, we use a dataset with 30GB memory footprint and perform TPC-H [TPC 1988] row-store and column-store queries. We select five representative queries (query1, query2, query6, query17, and query19) out of the 22 TPC-H suite. These queries incorporate different types of threads to perform various data access activities [Delimitrou et al. 2012]. Memcached is one of EPFL CloudSuite benchmarks [Ferdman et al. 2012]. It uses the memcached data caching server to simulate the behavior of a Twitter caching server using the twitter dataset. Graph500 [Graph500 2014] is a core part of many analytics workloads. The benchmark has two computation kernels. The first kernel generates the graph. The second kernel does a parallel breadth-first search of random vertices. Limited by the scale of our simulation, we use the dataset of 2^{26} vertices with 17 GB memory footprint. NAS [NASA 2009] benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels (IS, EP, CG, MG, and FT) that mimic the computation and data movement in CFD applications. Among the five kernels, IS performs random memory accesses to sort integers; EP is an embarrassingly parallel workload; CG solves conjugate gradient, performing irregular memory access and communication; MG is a memory-intensive workload

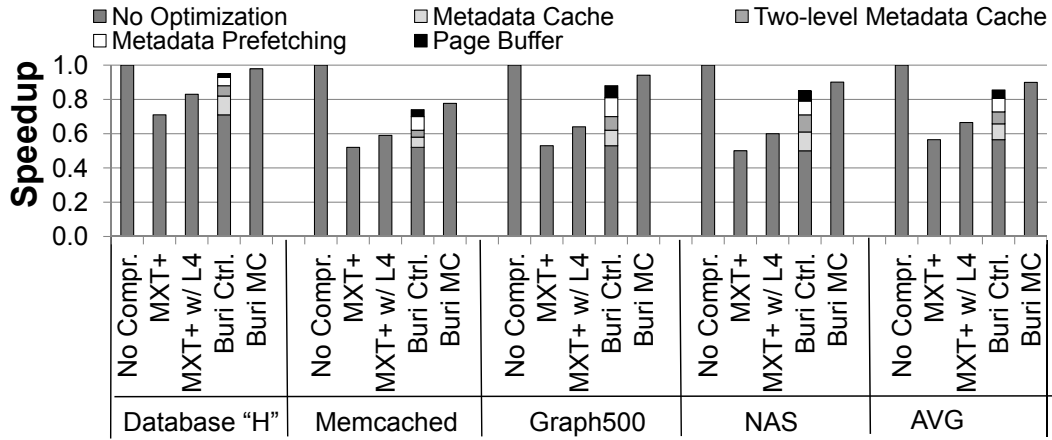


Fig. 8. Runtime performance of various designs: no compression, MXT+ (MXT [Tremaine et al. 2001] with fast compression and decompression), MXT+ with a 64MB on-chip L4 data cache, Buri with compression controller optimizations (Buri Ctrl.), and Buri also with metadata-cache-aware scheduling policy implemented in the memory controller (Buri MC).

that performs long- and short-distance communication in multi-grid on a sequence of meshes; FT performs discrete 3D fast Fourier Transform. We use dataset class A in our experiments.

8. EVALUATION RESULTS

8.1. Performance

We compare the performance of various memory compression schemes over the idealized “no compression” baseline. Performance is measured as the inverse of workload run time. Figure 8 shows that because of memory indirection overhead, MXT [Tremaine et al. 2001] incurs 56% of performance overhead compared with a system without memory compression. Adding a large on-chip L4 data cache to the original MXT memory hierarchy can boost system performance by 25% with Database “H”, which is optimized to fit the working set in the last-level cache. However, adding a very large cache only has modest effect other workloads. In particular, caching has little effect on Memcached who has a larger-than-cache working set and random access pattern. On average, MXT+ with a L4 on-chip data cache incurs 45% performance overhead compared with the system without memory compression. The last bar of each category of workloads shows Buri performance, a breakdown of the effect of various performance optimization mechanisms. Buri without our proposed performance optimizations actually performs worse than MXT+ with a L4 data cache. But by adopting performance optimizations, Buri yields 63% higher performance than MXT+ with L4 cache. Across all workloads, Buri incurs 11% performance overhead to the system without memory compression.

It is interesting to observe that Buri’s different optimization mechanisms appear to have different effect on various workloads. For example, with database “H”, metadata cache yields the highest performance improvement among all the optimization mechanisms, because database “H” generates a data access pattern that is optimized to fit the working set in the last-level cache, thus corresponding to better metadata optimizations. With Memcached and Graph500, employing a 16MB metadata cache still incurs 40% and 21% miss rate, respectively. With these two workloads, metadata prefetching yields the highest performance improvement. With different workloads, the data access patterns are different, and therefore we used different prefetching algorithms. Memcached and Graph500 access data in an irregular manner so Markov-based prefetching algorithm [Joseph and Grunwald 1997] provides the best prefetching accuracy of up to 73% and 68%, respectively. With database “H”, however, striding-based prefetching algorithm provides the best prefetching accuracy of up to 96%.

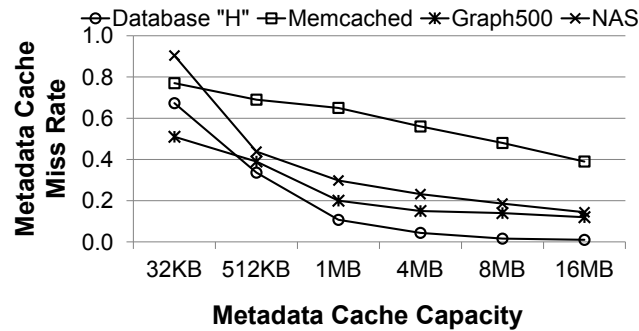


Fig. 9. Metadata cache miss rate with metadata cache of different capacities.

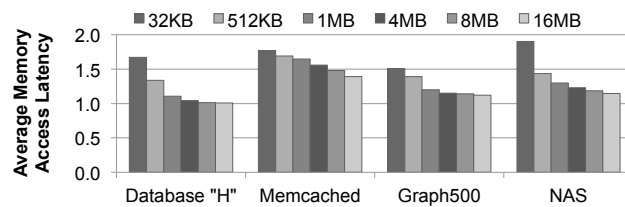


Fig. 10. Average memory access latency with metadata cache of different capacities.

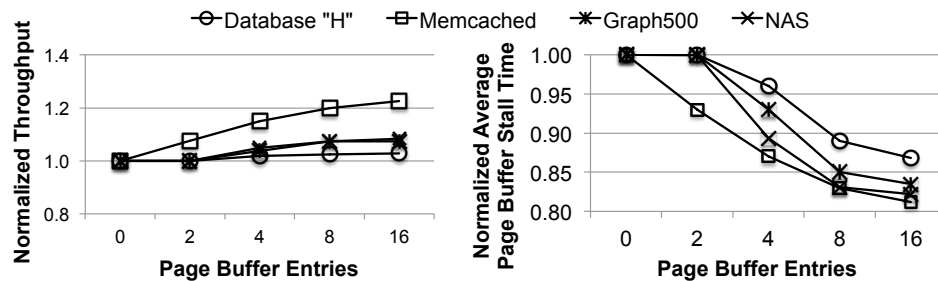


Fig. 11. Average system throughput and page buffer stall time of various numbers of entries in the buffer.

Sensitivity to metadata cache capacity: We studied metadata cache miss rate and memory access latency by varying the capacity of the metadata cache. Metadata cache miss rate indicates the rate of data accesses that require two memory transactions (to obtain both data and metadata in the memory). In Figure 9, a 4MB metadata cache can capture the working set of Database “H” pretty well. A metadata cache as large as 16MB still works poorly for Memcached, due to its random access behaviors. These observations are inline with the relationship between memory access latency and metadata cache capacity, as shown in Figure 10. While we could incorporate a megabyte-sized metadata cache in Buri design, the cost makes it less practical for early adoption. Consequently, we choose to incorporate a metadata cache of only 512KB and use other optimization mechanisms, such as prefetching, metadata cache-aware scheduling policy, and page buffer, to improve the system performance.

Sensitivity to page buffer size: We evaluated our page buffer design across various buffer sizes. Zero-entry page buffer represents that the system does not adopt a page buffer. As shown in Figure 11, increasing the number of page buffer entries from zero to eight can lead to substantial system performance improvement. However, when we further increase the page buffer size, system performance tends to be stable. Consequently, our design employs an eight-entry page buffer.

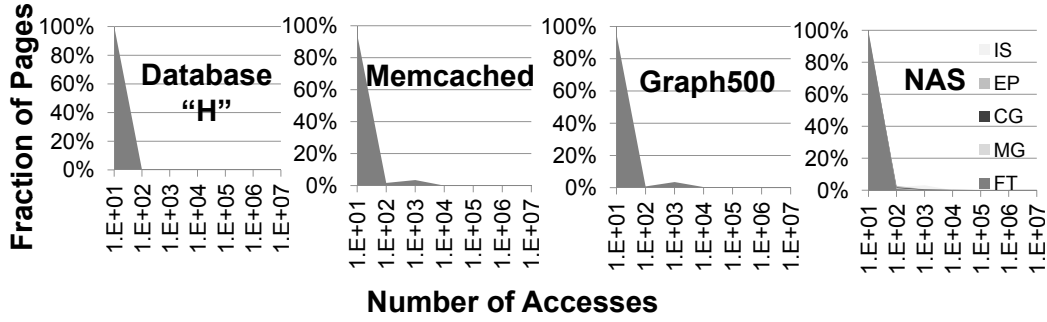


Fig. 12. Histograms that illustrate the hotness of pages in various workloads. The x-axis is the number of accesses.

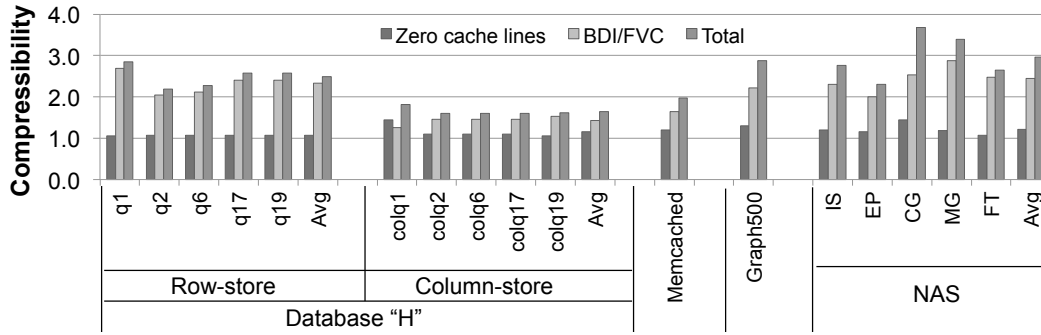


Fig. 13. Compressibility of in-memory database workloads.

8.2. Hotness of Pages

We further explored the hotness of physical pages in a compressed memory. Figure 12 illustrates the coverage of hot pages across the entire memory space of 2TB with various workloads based on large dataset profiling on real servers. As shown in Figure 12, all the workloads leave a large fraction of pages (more than 90%) cold. They only frequently access a small portion of pages. In fact, with Database "H", most of the pages are accessed only once at the initial time. During the rest of time, these pages simply stay in the memory as cold page. Even with Memcached, 95% of pages are accessed less than ten times during its execution. These observations support the effectiveness of our memory compression design without incurring significant performance loss on compression and decompression, which are infrequent operations.

8.3. Compressibility

Figure 13 shows the compression ratio Buri achieved with various workloads. It shows that Buri can achieve $2.5\times$ and $1.7\times$ of compressibility with Database "H", running row-store and column-store queries, respectively. With Memcached, Graph500, and NAS, the compression ratios are $2\times$, $2.9\times$, and $3\times$, respectively. Figure 14, Figure 15, and Figure 16 illustrate typical compressibility distribution in memory when running various workloads. We observe three typical types of distribution. Memcached appears to have a large number of interleaved memory pages with high and low compression ratios (Figure 14). Most pages of Graph500 have high compressibility (Figure 15). Database "H" and NAS workloads appear to have regions highly compressible pages surrounded by pages with low compressibility (Figure 16).

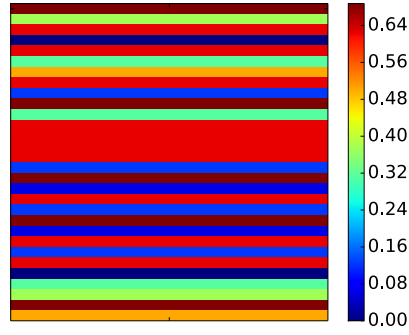


Fig. 14. Memcached's compression ratio distributions in the memory.

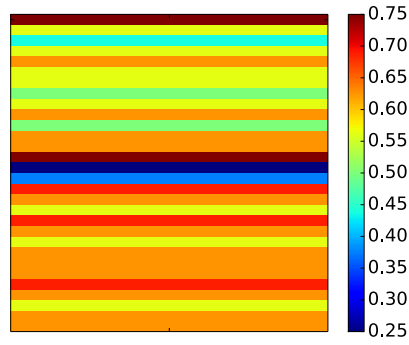


Fig. 15. Graph500's compression ratio distributions in the memory.

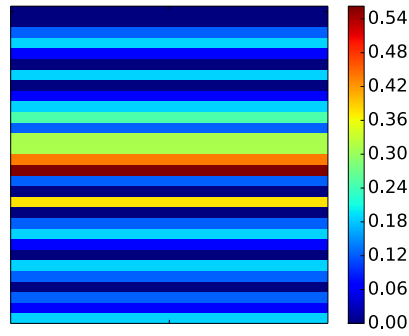


Fig. 16. EP (one of NAS workloads)'s compression ratio distributions.

8.4. Overflow Rate

In Figure 17, we evaluated the overflow rate of various workloads. With database “H”, column-store cache line overflow occurs less than 2% of all memory accesses, while page overflow rate is less

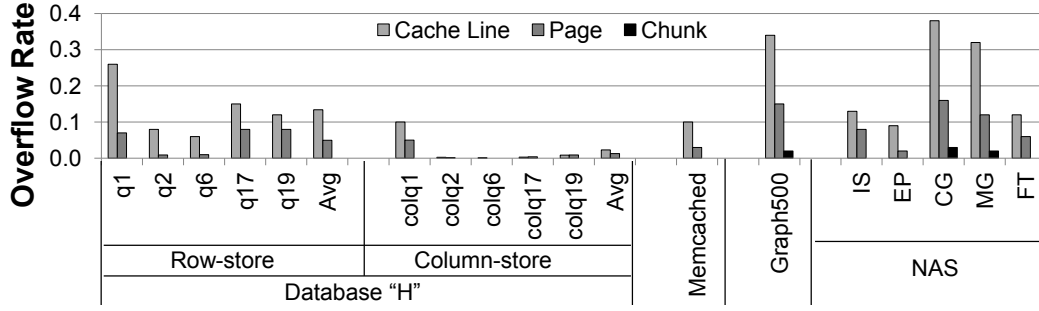


Fig. 17. Cache line, page, and chunk overflow rate.

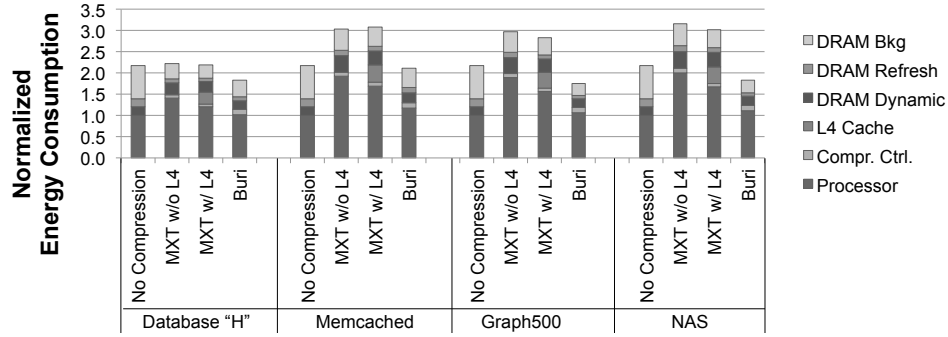


Fig. 18. Breakdown of energy consumption of various designs: no compression, MXT+ (MXT [Tremaine et al. 2001] with fast compression and decompression), MXT+ with a 64MB on-chip L4 data cache, and Buri. With each benchmark, the energy consumption results are normalized to the processor energy in the baseline (no compression).

than 1%; row-store has a higher cache line overflow rate of 13% on average, while page overflow rate is only 5%. Among all the workloads, Graph500 incurs the highest cache line overflow rate, as high as 34%. But the page overflow rate is reduced rapidly to 15%, and the chunk overflow rate is only 2%.

8.5. Area Overhead

We store 4KB pages in the page buffer. The page buffer size is 32KB (8x4KB), sufficient for our workloads. We only use a 0.5MB metadata cache, which outperforms larger metadata caches with the combination of prefetching, page buffer, and compression-aware scheduling. We calculate the area overhead of our hardware implementation using McPAT [Li et al. 2009]. Overall, Buri only incurs 0.4 mm^2 area overhead.

8.6. Energy Consumption

Memory compression introduces performance overhead, which implies increase of application execution time. This can lead to an increase of system energy consumption. However, systems with memory compression can adopt smaller DRAM capacities, leading to reduced memory background energy and refresh energy. We explored the energy tradeoffs by evaluating the system energy consumption of various designs. We calculate processor and memory power using McPAT [Li et al. 2009] and Micron's DRAM power model [Janzen 2010], respectively. We estimate system energy consumption by feeding our performance simulation results into the power models. Figure 18 illustrates system energy consumption of various designs, broken down to processor energy, compression controller energy (all components of Buri compression controller or compression/decompression logic of MXT+ designs), L4 cache (only applicable to the design of MXT+

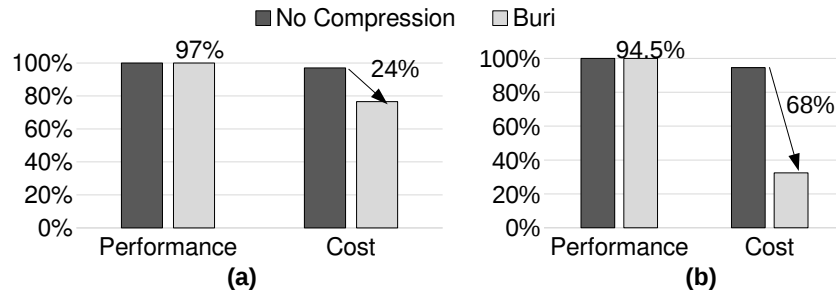


Fig. 19. Cost efficiency running TPC-H queries in database “H” on DL580 server with Buri on 2TB and 3TB data set respectively, in comparison to running the same queries on (a) 2TB data on DL580 server without Buri. (b) 3TB data on DL980 server without Buri.

with L4 cache), DRAM dynamic energy, DRAM refresh energy, and DRAM background energy (DRAM Bkg). With each benchmark, the energy consumption results are normalized to the processor energy in the baseline (no compression). As shown in Figure 18, Buri consumes less energy than the baseline with no memory compression, because smaller physical DRAM capacities lead to reduced DRAM background and refresh power. Buri also consumes much less energy than MXT+ designs due to reduced application execution time. Buri offers lower compression ratio to Memcached, compared to other benchmarks. Therefore, the energy consumption benefit of Buri with Memcached is less visible than with other benchmarks.

8.7. Cost Efficiency

We show that Buri design is cost effective by comparing it to two baselines – a 4-socket HP DL580 server with 2TB memory (2TB using 64 32GB DIMMs) and a 8-socket HP DL980 server with 4TB memory (128 DIMMs with 32GB per DIMM) – both without hardware-based memory compression.

Without hardware-based memory compression, database “H” with 2TB data set will utilize the entire physical memory. Buri can offer at least $1.6\times$ compressibility to database “H”, and therefore, we only need 1TB physical memory to support the whole data set. The cost of the system is reduced by 24% due to the reduced cost on physical memory (Figure 19). Buri only impose 3% performance overhead in this case.

We also evaluate database “H” with a 3TB dataset. Without Buri, the size of the dataset is beyond the maximum memory capacity of a 4-socket DL580 server. This represents a capacity-constrained use-case, where customers have to pay a high premium for larger capacity, e.g., a DL980 machine. Yet a 4-socket DL580 server augmented with Buri can provide at least 3.2TB effective physical memory capacity, accommodating the 3TB database “H” dataset. We observed that while the TPC-H queries on database “H” do benefit marginally from the increased computation resource when moving from 4-socket to 8-socket, they are negatively impacted by the extra NUMA latency of 8-socket SMP. We observe that Buri only incur 1% performance overhead on decompression latency and 3% performance overhead introduced by metadata cache misses. As a result, the 4-socket DL580 server with Buri is only 5.5% slower than the baseline 8-socket DL980 server. Because of the much lower hardware cost of the 4-socket DL80 server with Buri achieves 68% cost reduction compared to the baseline 8-socket DL980 server (Figure 19). Note that this calculation only considers hardware CapEx; adding software license and OpEx costs will further strengthen the case of adopting memory compression in high-end SMPs.

9. CONCLUSION

Today, memory is enslaved to the CPU memory channels, which are themselves limited by off-chip pins. The only way to increase memory capacity is to add more CPU sockets. This is a very inflexible

and expensive way of provisioning memory, and many customers are forced to populate the sockets just for the additional controllers to expand capacity. Memory compression can be seen as a first step to break this dependence. As shown by our Buri design, even a moderate amount of compression could enable reducing the CPU sockets and thus generate savings that far exceed the memory cost itself. Buri is a hardware-based memory expansion scheme which simultaneously achieves ease-of-integration, high performance, and cost efficiency. It combines the best elements of prior designs into a self-contained compression controller, sitting between processor caches and main memory. The compression controller manages common operations for compressed data access, metadata maintenance, and memory management functionalities. In a broader context, Buri fits with recent proposals on big data [Lim et al. 2013], [Basu et al. 2013], by scaling the memory capacity wall without expensive support from the compute and communication sides.

REFERENCES

- Jung Ho Ahn, Sheng Li, Seongil O, and Norman P. Jouppi. 2013. McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. 74–85.
- Apple. 2014. Apple OSX Mavericks. (2014). <http://www.apple.com/osx/>
- Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 237–248.
- Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. 10–10.
- Christina Delimitrou, Sriram Sankar, Badriddine Khessib, Kushagra Vaid, and Christos Kozyrakis. 2012. Time and Cost-efficient Modeling and Generation of Large-scale TPCC/TPCE/TPCH Workloads. In *TPC Technology Conference on Topics in Performance Evaluation, Measurement and Characterization*. 146–162.
- Magnus Ekman and Per Stenstrom. 2005. A Robust Main-Memory Compression Scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. 74–85.
- Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*. 365–376.
- Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafae, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 37–48.
- The Apache Software Foundation. 2014a. Apache Giraph. (2014). <http://spark.incubator.apache.org/>
- The Apache Software Foundation. 2014b. Apache Spark. (2014). <http://spark.incubator.apache.org/>
- Peter A. Franaszek, Luis A. Lastras-Montano, Song Peng, and John T. Robinson. 2006. Data Compression with Restricted Parsings. In *Proceedings of the Data Compression Conference*.
- Graph500. 2014. Graph500 benchmarks. (2014). <http://www.graph500.org/>
- Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward Dark Silicon in Servers. *IEEE Micro* 31, 4 (July 2011), 6–15.
- IBM. 2012. Active memory expansion: overview and usage guide. (2012). https://www-O3.ibm.com/systems/power/hardware/whitepapers/am_exp.html
- Intel. 2014. Intel Core i7. (2014). <http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html>
- Jeff Janzen. 2010. The Micron power calculator. (2010). <http://www.micron.com/products/dram/syscalc.html>
- Doug Joseph and Dirk Grunwald. 1997. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*. 252–263.
- Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499.
- H. Lekatsas and W. Wolf. 1998. Code compression for embedded systems. In *Proceedings of the Design Automation Conference*. 516–521.
- Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the International Symposium on Microarchitecture*. 469–480.

- Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. 267–278.
- Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 36–47.
- Chang Hong Lin, Yuan Xie, and W. Wolf. 2004. LZW-based code compression for VLIW embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 76–81.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*. 190–200.
- Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. 2009. Transcendent memory and linux. In *Proceedings of the Linux Symposium*. 191–200.
- Memcached. 2014. Memcached, a distributed memory object caching system. (2014). <https://memcached.org>
- Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *International Symposium on Computer Architecture*. 63–74.
- NASA. 2009. NAS Parallel Benchmarks. (2009). <http://www.nas.nasa.gov/publications/npb.html>
- Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 172–184.
- Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. 377–388.
- Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. 24–33.
- Scott Rixner. 2004. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. 355–366.
- Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 128–138.
- SAP. 2014. SAP HANA. (2014). <http://www.saphana.com/>
- Somayeh Sardashti and David A. Wood. 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-optimized Compressed Caching. In *Proceedings of the International Symposium on Microarchitecture*. 62–73.
- TPC. 1988. TPC-H, an ad-hoc, decision support benchmark. (1988). <http://www.tpc.org/tpch/>
- R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland. 2001. IBM memory expansion technology (MXT). *IBM Journal of Research and Development* 45, 2 (2001), 271–285.
- VoltDB. 2014. VoltDB: Smart data fast. (2014). <http://voltdb.com/>
- Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. 1999. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '99)*. 8–8.
- Andrew Wolfe and Alex Chanin. 1992. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO 25)*. 81–91.
- Jun Yang and Rajiv Gupta. 2002. Energy Efficient Frequent Value Data Cache Design. In *Proceedings of the International Symposium on Microarchitecture*. 197–207.
- Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent Value Compression in Data Caches. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*. 258–265.
- Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent Value Locality and Value-centric Data Cache Design. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 150–159.
- Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. 59–70.