

# Compressibility Characteristics of Address/Data Transfers in Commercial Workloads

Krishna Kant and Ravi Iyer  
Enterprise Architecture Laboratory  
Intel Corporation  
{krishna.kant|ravishankar.iyer}@intel.com

## Abstract

In this paper, we evaluate the compressibility of address and data transfers in commercial servers. Our proposed compression scheme is geared towards improving the efficiency of the transfer medium (busses, links etc) and increasing the performance of the system. We evaluate the potential of the basic compression techniques for two commercial workloads – SPECweb99 [21] and TPC-C[22] – based on trace-driven simulations. Based on the obtained results, we show that simple compression schemes show significant promise for reducing address bus width and moderate benefits for data bus width reduction. We also show the sensitivity of these performance benefits to the number of bits compressed and the size of the encoding/decoding table used. Additionally, we propose enhancements to the compression schemes based on (1) recognizing and utilizing data-type specific knowledge and (2) improving the replacement policy of the encoding/decoding table. The performance benefits of bus compression schemes with these enhancements are also presented and analyzed.

## 1 Introduction and Motivation

With the increasing demand for high performance systems, commercial servers are now designed with large caches and larger memory resources. In order to reduce the amount of resources needed, researchers have proposed compression as a solution. This includes compressed storage techniques [19, 26], compressed main memory [1, 23] to reduce memory resources needed and cache compression techniques [14, 27] to reduce the amount of cache space needed.

Our focus in this paper is to design simple compression schemes that helps reduce the amount of information transferred between the processor caches and the memory subsystem. This compression is primarily geared towards improving the performance and efficiency of the transfer medium (busses, links etc). Current generation front-end

and back-end servers are typically bus-based, with each system bus supporting several processors. As we look at the potential for bus-based systems in the future, we find three key design pressures: (1) to scale in frequency comparable to processor frequency improvements, (2) to support larger bus widths for transferring larger cache lines within the same amount of time and (3) to continue to provide scalability with multiple processors. In this paper, we evaluate the benefits of simple compression techniques in reducing the amount of address and data transferred over the bus, thereby allowing for narrower busses, less crosstalk and potentially higher frequencies. It is important to note here that the benefit of compressing the information transferred between processor and memory not only applies to busses but also to point-to-point links that might replace busses in future servers. In the case of point-to-point links, the benefit of compression materializes as a reduction in the transfer latency (since much less data is transferred over the link).

Our main contribution in this paper is as follows. We present the basic premise of the compression techniques used for reducing address and data transfer lengths. We discuss the locality properties in address and data streams while running commercial workloads on servers. We evaluate the potential of the basic compression techniques for two commercial workloads – SPECweb99 [21] and TPC-C [22]. The evaluation is based on analyzing various traces collected on real systems. Based on these results, we show that simple compression schemes show significant promise for reducing address bus width and moderate benefits for data bus reduction. We show the sensitivity of these performance benefits to the number of bits compressed and the size of the encoding/decoding table used. Additionally, we propose enhancements to the compression schemes based on (1) recognizing and utilizing data-type specific knowledge and (2) improving the replacement policy of the encoding/decoding table. The performance benefits of bus compression schemes with these enhancements are also presented and analyzed.

The rest of this paper is organized as follows. Section 2

provides an overview of related work on compression schemes for servers. Section 3 presents the basic premise behind the address and data compression schemes and proposes potential enhancements. Section 4 provides an overview of our evaluation methodology covering details of workloads and traces. Section 5 presents the salient results and provides a detailed analysis of the benefits. Finally, section 6 summarizes the paper and presents a direction for future work in this area.

## 2 Background on Compression Techniques

In this section, we provide a brief overview of the past work in the compression area, particularly as it relates to increasing the system performance.

### 2.1 Disk/Memory Compression Techniques

Several papers including [19, 26, 13] have investigated the use of compressed storage to reduce paging. These *swap-space compression* techniques use a LRU main-memory cache to hold evicted pages in compressed form and intercept page faults to check if the requested page is available in the cache before a disk access is initiated. Such an approach could significantly improve the performance for applications that require a large amount of memory but do not manage their paging behavior. For example, scientific applications working with huge matrices or other data structures could benefit from this technique. Most of the server applications such as DBMS or web-servers carefully manage the paging activity and are unlikely to see any significant benefits from swap-space compression. In fact, the loss of memory to compressed cache and the overhead of compression/decompression could well deteriorate performance.

A slight extension of swap space compression is *compressed disk cache* which introduces a compressed cache for all disk I/O (including paging and file I/O). In addition to the evicted pages, this cache also stores the files evicted from the O/S or application managed file cache. Due to a much large space needed for the compressed disk cache, dynamic cache size adjustment similar to the one in [26] is necessary to ensure that the compressed cache does not starve the normal disk cache. Workloads that require a significant amount of I/O per transaction could benefit significantly from the compressed cache. Alternately, a compressed cache could allow the use of a lower performance disk subsystem and thereby significantly lower the total system cost. High density front-end servers could benefit from this because of their physical space and power limitations which do not allow large I/O subsystems.

An even more extensive use of compression involves storage of all main memory data in compressed form. Unlike the last two schemes, compressed storage of all data is much more complex since it introduces a new address space (the compressed physical address space) along with the issues of efficient storage and address translations. Furthermore, if memory accesses from bus masters are to be kept transparent, it is necessary to introduce a decompression cache where cacheline level accesses can be satisfied. IBM's MXT technology [1], takes this approach along with the added flexibility that certain regions of the memory can be set as compressed while others are uncompressed. Reference [23] describes the details of IBM's Pinnacle chipset that supports this technology. It compresses memory in 1 KB blocks and stores compressed blocks using up to four 256 byte segments. These segments could be located anywhere in the physical memory and are accessed using 4 pointers in the header part of each block. Although this generality avoids storage fragmentation, simpler schemes may be preferable.

### 2.2 Compression Algorithms

With the considerable interest in compression in the main memory, several studies have examined compressibility of main memory data and specialized compression algorithms. Reference [12] studies compressibility of many popular Unix desktop applications using both the traditional algorithms (e.g., LZW, Arithmetic coding [20]) and the X-RL algorithm invented by the authors [13]. The latter algorithm encodes 4 bytes at a time using partial matching of bytes and dynamic coding based on a small dictionary. It is claimed to be especially suited for small block sizes and hardware implementation. The authors show that this can be easily implemented in hardware to provide 4 bytes/cycle input rate to the compressor/decompressor. The adaptive LZ77 (de)compressor in IBM-MXT also achieves a similar rate, but appears more expensive to implement [4]. Since reads are typically lot more prevalent than writes, it generally helps to use *asymmetric algorithms* where the decompression speed is significantly higher than the compression speed. This property is particularly important to the implementation suggested in this document. Both LZ77 [28] and X-match [13] have this property, but LZ78 (or its variant LZW [25]) do not.

High order bits in basic data types often show a low entropy. This observation has been exploited in [26] which introduces a new compression scheme that examines 32 bits at a time and looks for redundancies in the 22 MSBs only. For certain types of data (e.g., integers, floating point), this could make the compression more effective than a traditional byte based compression. This leads to the following general observation: *when dealing with*

structured data, it is useful to do compression in the units in which the data items are accessed. In particular, in a relational database environment, it makes sense to base compression on data fields or sets of fields (e.g., keys), etc. Reference [7] addresses this primarily from the point of reducing the cost of I/O subsystem.

Compression of machine code presents a rather unique case. Viewed based on the bit/byte patterns alone, code does not provide too much opportunity for compression. However, it is possible to exploit the knowledge of about the instruction format in order to reduce the number of bits needed to identify the instructions. Reference [5] shows that it is possible to achieve compression ratios of 3-5 for code in this manner. Several other issues have also been examined in code compression including special algorithms that allow decompression starting with any cacheline [16] and procedure based compression where individual procedures are compressed as a unit and decompressed at run-time as needed [11]. Reference [15] reviews several code compression techniques and examines the performance of hardware managed code compression available in IBM PowerPC 405.

### 2.3 Cache Compression Techniques

Although much of the compression-related work has concentrated on compressed storage on disk and in main-memory, there are a few notable attempts to use compression within the processing core as well. In most cases, processor registers and L1 cache store uncompressed data and compressed storage is confined to L2 cache. Reference [14] discusses a selective compression scheme wherein the memory and L2 cache addressing is in terms of blocks of the size of one cacheline (as usual) except that a block may contain either one uncompressed cacheline or two adjacent compressed cachelines. This is driven by compressibility considerations — if two adjacent cachelines compress to no more than one block the storage is in compressed form, else it is in uncompressed form. The cache mapping scheme remains unchanged so that the cache storage of uncompressed lines is resolved in the usual way. For a compressed line however, the sets corresponding to both lines (i.e., with least significant address bit being both 0 and 1) are examined. The scheme uses a small decompression buffer between L2 and L1 which acts just like an intermediate cache. Eviction of a modified line from L1 that is a part of a compressed block in L2 results in decompression of the whole block, modification of the relevant part, recompression and writeback to L2. The main memory storage scheme used in [14] is rather simplistic and remains in the units of pages of normal size and half-size. A half-page is used if all blocks within a page are compressed. Additional bits are used in page tables to handle two different page sizes and to

identify the compressed/uncompressed nature of individual blocks within a page. The paper shows detailed simulation results for SPEC95 benchmark to confirm the reduced miss rate and read/write traffic in the core.

Compression at the L1 level has also been considered. Reference [27] considers a scheme similar to the one above, i.e., two adjacent cachelines are mapped to either of the two adjacent sets if it is to be stored in compressed form. The main difference is that the compression is not at the level of cachelines, but instead for individual “items” (e.g., 32-bit words) in a cacheline. This compression is based on the premise that a majority of accesses are to a small set of values (e.g., 0, 1, starting address of a large array, substring of spaces, etc.) and thus can be easily replaced by an index into a table containing “frequent values”. For this reason, the technique is also known as *frequent value compression*. The major problem with this scheme is the determination of most frequent values and the latency of additional table lookup.

## 3 Our Focus: Compression of Address/Data Transfers

The nature of the information can often be exploited for reducing the number of bits needed in the representation. For example, reference [6] makes the observation that the addresses appearing on the address bus show considerable locality which can be exploited for reducing the number of address lines. This is done by only transmitting high order bits of the address and obtaining the low order bits from an encoding/decoding table. Reference [3] proposes a similar scheme for the data lines based on the locality in data values. In particular, based on technical workloads, the paper claims that 16 LSB data lines carry 90% of the information contained in 32 bit data items. These studies were done for uniprocessor systems with very small caches and also for technical workloads only. In this paper, we start by verifying the effectiveness of such a scheme for commercial workloads as well as for multiprocessor servers. Additionally, we propose various enhancements to the compression technique and evaluate their effectiveness over the base scheme.

### 3.1 Compressing Address Transfers

We start by describing the basic scheme for compressing address bits for processor-memory transfers. If successive memory accesses are mostly concentrated within a small region, the high order address bits will change infrequently and need not be transmitted every time. Instead, a dynamic encoding scheme could put the high-order bits in an encoding table (or cache) and transmit only the table index for later “hits” in the table. The first time around,

the high order bits are transmitted so that an identical decoding table can be built on the other side as well without any special information transfer. Figure 1 illustrates this scheme with a 64 entry encoding/decoding table incorporated into a conventional (base) system as an example. The base system with support for 36-bit addressability is shown in Figure 1(a). In Figure 1(b), the system with address bus compression is based on 18 low-order bits and 18 high-order bits. The low order bits are transmitted directly. The high order bits are looked up in a 64-entry encoding table. If the entry is found (table hit), only a 6-bit index to the decoding table need be transmitted for the high-order part. Thus a 24-bit address bus suffices. A miss will, of course, require data transfer over 2 cycles. (Compressed and uncompressed transfers can be distinguished by a special treatment of table entry 0.) Thus, if a high table hit ratio can be achieved, the compressed bus provides 33% reduction in width over the original bus with only a small performance penalty.

### 3.2 Compressing Data Transfers

A scheme similar to the address transfer compression scheme can be envisioned for compressing data transfers over the data bus as well. One major difference, however, in the data transfer case is that each cache block needs be divided into further smaller chunks before being encoded or compressed. However, choosing the right chunk size is not an easy task since the data accessed from the memory subsystem can be of various types (integer, pointer, floating point, code etc.) and lengths. The compressibility characteristics are very much dependent on the data type. For example, large values are rare for integers. This implies that the high-order bits for integers are usually all 0's or 1's, thus making integers a good candidate for compression. Pointer data typically shows considerable locality — long jumps or successive references to data items that are far apart are rare. However, this locality is in the virtual address space; it will be preserved only if the physical memory allocation is reasonably contiguous. Floating point data (single precision *float* or double precision *double* types) are usually poor candidates for compression except for generally very small exponents and frequent values like 0 or 1. Code data type also offers poor compressibility unless the knowledge of instruction set is exploited (which may be expensive for on-line use). Finally, the character string data offers moderate compressibility if the string length can be determined.

The main virtue of data type identification is that it allows certain optimizations that would be unavailable if all data were to be treated identically. For example, it makes sense to use the same number of low-order bits for both addresses and pointer data, and one may even consider a single encoding table for these. Similarly, if the

double and/or code datatypes offer very poor compressibility, it might be preferable not to encode them at all since they would only pollute the encoding table. Even if all data is handled identically, it is interesting to find out what kind of hit ratios various data types get. In current architectures, data type information is not available for bus transfers (except for code fetch vs. data read/write); therefore, the identification must be based entirely on bit-patterns and could not be accurate. Fortunately, for our purposes, an accuracy of 80-90% is good enough. A good bit-pattern based identification depends on several factors including frequently used data types, machine addressability (32-bit or 64-bits), and compiler characteristics. Our method applies to the code generated by most C-compilers for most 32-bit machines. The method assumes implicitly that 8/16 bit integers and float datatype occur only sparingly. For simplicity, we consider data in 32-bit chunks only, which means that character and “short” integer data aligned on smaller boundaries will be missed. The details of our method are as follows.

- **Code Data Type** – Code fetches are easy to identify correctly since code fetches appearing on the bus usually possess a different request id than regular data reads.
- **Integer Data Type** – Here we check if the 8 MSBs are 00 or FF (in hex) in a 32-bit word. Note that if the 32-bit word contains 2 “short” (or 16 bit) integers, they will be regarded as single 32-bit integer. The scheme will work for 64-bit integer representations as well provided that the actual values of these variables rarely exceed  $2^{31}$ .
- **Text Data Type** – This is identified by checking whether in each 32-bit word, *every* 8th MSB is 0 and *at least one* 7th MSB is 1. This is basically looking for 7-bit ASCII characters, at least one of which is a printable character.
- **Double/Float Data Type** – If two contiguous 32-bit words are neither Integer nor Text, we look at first 5-bits to recognize small positive or negative exponents in the standard IEEE floating point format. If recognized as such, the data is declared as double. Note that this scheme will recognize two contiguous floats as a double — beyond this, no effort is made to recognize floats.
- **Pointer Data Type** – If a 32-bit word does not fall in any of the above categories, then the data type is assumed to be a pointer.

The above method needs some changes for 64-bit machines since the pointer data now occupies 64-bits. Most machines use (and are likely to use for quite some time)

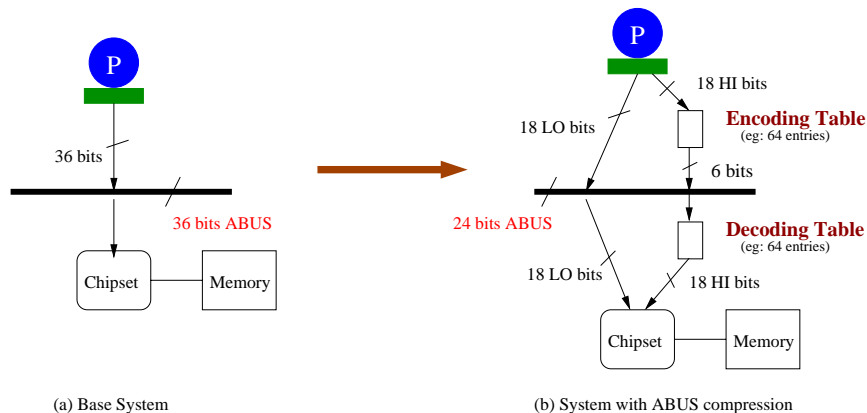


Figure 1: Illustration of Basic Compression Scheme (eg: for Address Transfers)

much fewer than 64-bits for addressing. For example, Intel’s IA-64 architecture uses only 44-bits, which can address 16 TB of memory. This information along with the fact that the top 8 or more bits are unlikely to change much can be exploited for identifying pointer data more directly and compressing it more efficiently. If the compiler allocates 64-bits by default for integers, one could exploit this too, but since the top 32 bits will almost always be all 0’s or all 1’s, dealing with 32-bit chunks will work just as well.

## 4 Workload Traces, Tools and Methodology

Our evaluation methodology relies on bus traces collected on real systems running commercial workloads such as SPECweb99 and TPC-C. Here we present an overview of the two benchmarks and the trace configuration. We also describe the trace-driven tools developed to simulate the encoding/decoding table and its performance benefits. Finally, we present the scope of the studies undertaken.

### 4.1 Workloads and Traces

For this study, we used bus traces that were collected on a web server running SPECweb99 and on database servers running TPC-C.

SPECweb99 [21] is a benchmark that attempts to mimic a web server environment. The benchmark setup uses multiple client systems to generate aggregate load on the system under test (a web server). Each client (mimicking browsers) makes static GET, dynamic GET or a POST request to the server. The get requests always retrieve a file from the server, but for dynamic gets, the file may be appended with some dynamically generated data. The retrieved file comes from a file-set with access character-

istics defined via a 2-level Zipf distribution. Some studies on cache/memory access characteristics of web workloads can be found in [17, 8, 9]. For our study, we used SPECweb99 traces from a 2P web server since a dual-processor configuration is more characteristic of systems used in this market segment.

TPC-C [22] is an online-transaction processing benchmark that mimics a parts ordering database system. The transactions include entering and delivering part orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Some studies on the cache/memory access characteristics of OLTP workloads can be found in [2, 18, 10]. For our study, we used TPC-C traces from a 4P system. In addition to the above mentioned 32-bit systems, we also used a TPC-C trace from a 4P 64-bit system for understanding the impact of compression schemes on 64-bit architectures.

### 4.2 Simulation Tools and Studies

To perform the compressibility analysis, we developed a tool called Simulator for COMPRESSED Transfers (SCOT). SCOT basically provides mechanisms to simulate encoding/decoding tables for both address & data transfers. The management of data in the encoding/decoding table is an important aspect that needs to be addressed. Our aim in this paper is only to show the potential of compression schemes for address/data transfers. We plan to investigate detailed design issues for the encoding/decoding table in the future after determining that there is some performance potential to be taken advantage of. So, in this early evaluation, we consider the encoding/decoding table to be a fully associative array. We determine the performance potential (in hit ratio) of this scheme as a function of the encoding size (number of high-order bits that are compressed) and the size of the encoding/decoding table. We then study the impact of different replacement strate-

gies (FIFO, LRU and our proposed MLRU policy). While the first two are well understood, our proposed MLRU scheme is based on giving lower priority to blocks that are first brought in. The plain LRU scheme can be thought of in terms of a stack with the blocks ordered in terms of their access time. So, in the LRU scheme, when a block is first placed on the stack, it is given the highest priority. The MLRU scheme alters the priority of incoming requests by using a parameter that controls where they are placed among the list it is placed. For the results presented here, an incoming block is placed only 25% above the lowest priority block. As a result, if the block is not accessed soon after it is entered into the stack, it is replaced. This tends to filter out one-touch references [24] and thereby leads to better performance than plain LRU.

## 5 Preliminary Results and Analysis

### 5.1 Compression Characteristics of Address Transfers

The performance benefits from the basic compression scheme for address transfer in 32-bit systems is shown in Figures 2 and 3. The x-axis in these figures denotes the table size (on a log scale) and the y-axis represents the hit ratio in the encoding/decoding table. For these results, a FIFO replacement policy is assumed. It is found that 85-90% hit ratio is easily achieved by using a table size of 64 entries and 20 high order bits (out of a total of 36 bits). The total number of address lines needed in this case is 22 (16 low order bits + 6 bit table index), which corresponds to a 39% reduction. This verifies that the compression scheme continues to work for commercial server workloads and for much larger caches than those considered in the original reference [6].

We experimented with a 64-bit system configuration as well. Here we chose the top 24 bits (out of a total of 44 bits) for compression. Only one trace (TPC-C with 16 GB of memory) was available in this case, so the results shown in Figure 4 should be considered as somewhat preliminary. We found that a 85-90% hit ratio needed a larger table size of 256 entries, which gives a savings of  $16/44 = 36\%$ . One reason for not getting better results is the larger database size and throughput for this configuration (consistent with 16 GB installed memory) which may reduce locality. The other possibility is the O/S (Windows 2000 for this configuration vs. NT4.0 for the 32-bit configuration). The O/S could affect the locality in two ways: (a) Size and locality of the O/S code itself, (b) Virtual to physical address mapping, since the mapping could perturb the inherent program locality significantly. In fact, an important point to keep in mind is that if address compression is to be exploited fully, the mapping algorithms

should also be redesigned so that they don't significantly increase the entropy of the high order address bits. In this sense, our results are conservative and would improve with better address mapping schemes.

In order to attempt to increase the address table hit ratio further, we explored the use of better replacement policies. We tried three schemes: FIFO being the base, the well-known LRU scheme and our proposed MLRU scheme as described in Section 3. Figure 4 presents the benefits of the MLRU scheme for 64-bit systems. As shown, the MLRU scheme improves the address table hit ratio by roughly 5%. We will show the effectiveness of the MLRU scheme for data transfers also in the following subsection.

### 5.2 Compression Characteristics of Data Transfers

For data transfer compression in 32-bit machines, we used 16-high order bits of each 32-bit word for compression (thus avoiding any special treatment of double datatypes). Note that this choice makes the pointer data and address compression mutually compatible since 16 lower order bits are used in each case. Figure 5 illustrates the results for TPC-C. For these results, all data types were encoded through the same table. This coupled with the fact that 16-high order bits are used for all data types means that the data type identification is not required and does not affect the results. (The identification was still done in order to obtain data-type specific hit ratios). It is seen from Figure 5 that assuming a maximum reasonable table size of 256 entries, the achievable hit ratio on 32-bit systems is only 73% with the FIFO policy and 78% with MLRU policy, both of which are perhaps not very attractive.

Windows O/S allocates integers as 32-bit quantities even on a 64-bit machine. Therefore, for data transfer compression in 64-bit machines, we continued to compress 16 high-order bits of successive 32-bit words for all data-types except pointers. In order to make the treatment of pointers compatible with addresses, we use top 44 bits of 64-bits for compression. Of course, first 20 bits out of these are guaranteed to be zero since the addresses are only 44-bits long. (This obviously requires identification of pointer data based on bit patterns.) Figure 6 illustrates the results for TPC-C. It is seen that the performance in this case is significantly better than it was on 32-bit systems. In particular, for a 256-entry encoding/decoding table, we get a hit ratio of roughly 87% with FIFO policy and 90% with MLRU policy.

In order to understand the above results, we next look at the frequency of access to the different data types and the hit ratio for each data type as a result. The data type access frequency is shown in Table 1. From the 64-bit data shown, we find that the access to integers dominates the

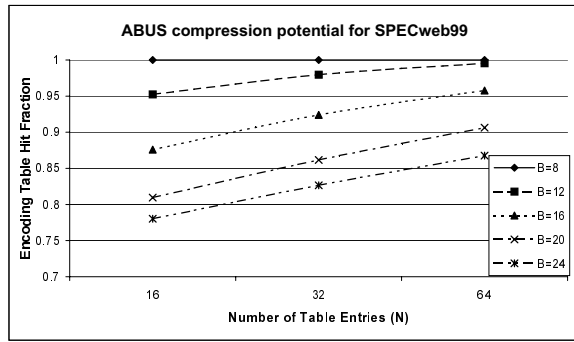


Figure 2: SPECweb99 Addr Compression: Varying Block Size, Table Size

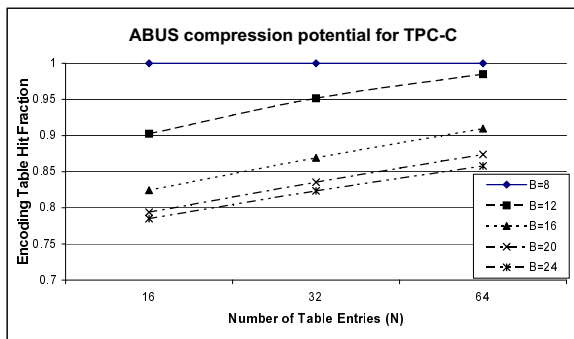


Figure 3: TPC-C (32-bit) Addr Compression: Varying Block Size, Varying Table Size

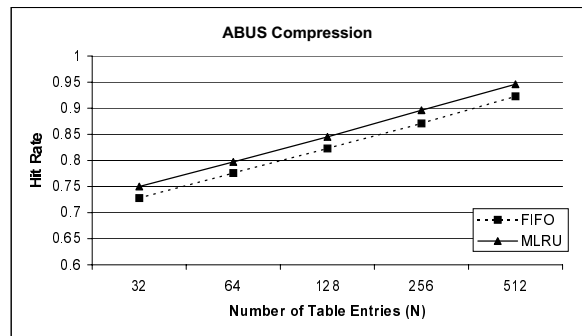


Figure 4: TPC-C (64-bit) Addr Compression: Varying Table Size, Replacement Policy

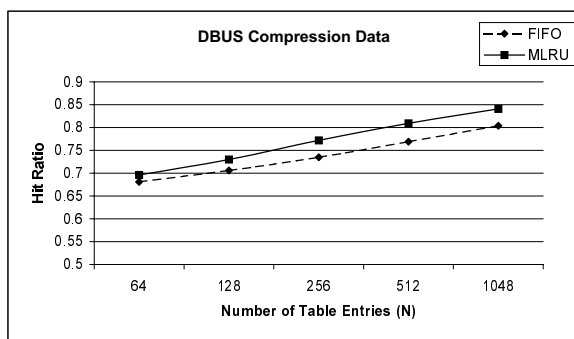


Figure 5: TPC-C (32-bit) Data Compression: Varying Table Size, Replacement Policy

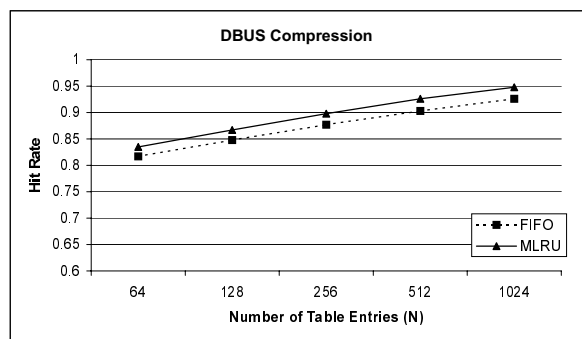


Figure 6: TPC-C (64-bit) Data Compression: Varying Table Size, Replacement Policy

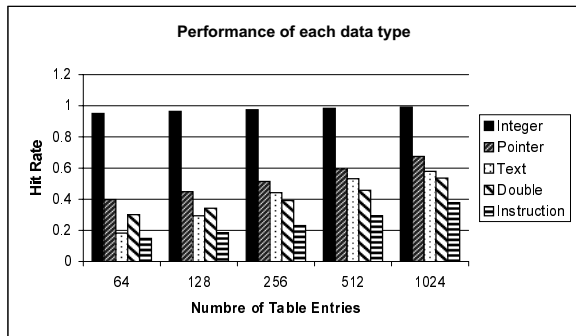


Figure 7: TPC-C (32-bit) Data Type Specific Hit Rates

Data Type	32-bit systems	64-bit systems
<i>Integers</i>	61.43%	91.35%
<i>Pointers</i>	19.80%	1.98%
<i>Text</i>	13.33%	6.50%
<i>Double</i>	0.34%	0.18%
<i>Instruction</i>	5.10%	NA

Table 1: Data Type Access Frequencies in TPC-C

hit ratio obtained. From the 32-bit data shown, we observe that the most frequently accessed data type in TPC-C is integers (with 62% access probability) and the least frequently accessed data type is double (with 0.34% access probability). Figures 7& 8 show the hit ratio for each data type. It is seen that integers have very substantial locality (and hence hit ratios), pointers have a reasonable locality, and code/double have the least amount of locality.

From Figure 7 it is clear that while pointer data type is the second most frequently accessed, its hit ratio is significantly lower than the integer hit ratio. We hypothesize that the main reason for the low hit ratio in 32-bit systems is that the top 16 bits of pointers still have quite a bit of entropy. In the 64-bit case, Figure 8 shows that pointer data type also provides a good hit ratio; although in the considered case, the percentage of pointer references is rather small, which means that the overall performance is dominated by the integer hit ratios. As other 64-bit traces become available, we plan to do further analysis and see if the good data hit ratio obtained for this trace persists.

Although the address/data bus compression schemes appear suited only for data transmission over a bus/link, they are really no different from the compression schemes used for storage. The encoded data retains all the information needed for reconstructing the encoding table and decoding the data (except for the static parameters such as the table size, which can be remembered elsewhere). It is crucial, however, that the information be decoded in the same order as it was encoded. This is not an issue with bus transfer; in other contexts, this property can be

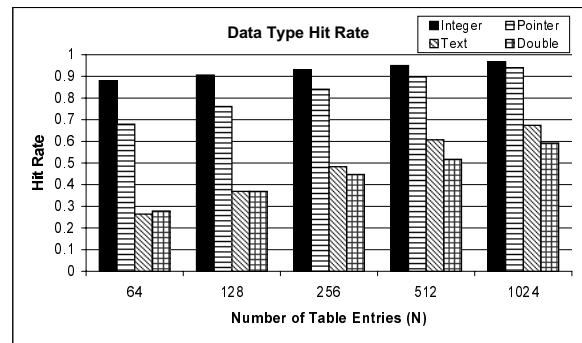


Figure 8: TPC-C (64-bit) Data Type Specific Hit Rates

enforced by dividing data into blocks so that each block is handled separately and involves a separate encoding table. Obviously, small block sizes will result in very little compression in general.

## 6 Summary and Future Work

In this paper, we evaluated the compressibility of address and data transfers in commercial servers. We started by presenting the basic premise behind simple compression schemes that use encoding/decoding tables. We showed that address transfers in web servers as well as OLTP servers show significant potential for compressibility (from 85 to 90% hit ratio) for a reasonable table size (64 entries). We also showed that data transfers in 32-bit systems show only moderate potential (roughly 75%) even for a reasonable table size (256 entries). We also showed that we can increase this compressibility potential by improving the replacement policy (MLRU) and by using data type specific optimizations.

In the future, we would like to explore this technique by delving into the implementation issues (set-associative tables, bus protocol issues) and associated performance evaluation. We would also like to incorporate the hit ratio data into a system-level performance model in order to evaluate the overall performance impact for commercial workloads. We would also like to expand our current suite of commercial workloads by including integer/floating point workloads (SPEC2000), JAVA workloads (SPECjbb) and e-commerce workloads (TPC-W).

## References

- [1] B. Abali, H. Franke, S. Xiaowei, et al., "Performance of hardware compressed main memory", The Seventh International Symposium on High-Performance Computer Architecture, 2001, (HPCA2001), pp. 73 -81
- [2] L. Barroso, et al. "Memory System Characterization of Commercial Workloads", *Proceedings of the 25th Annual*

- International Symposium on Computer Architecture*, pp3-14, June 1998.
- [3] D. Citron and L. Rudolph, "Creating a wider bus using caching techniques", Proc of first Intl symposium on high performance computer architecture, Jan 1995, pp 90-99.
- [4] D.J. Craft, "A fast hardware data compression algorithm and some algorithmic extensions", IBM Journal of R&D, Vol 42, No 6.
- [5] J. Ernst, W. Evans, et. al., "Code compression", Proc of 1997 SIGPLAN Conf. on Programming Language Design and Implementation, June 1997.
- [6] M. Farrens and A. Park, "Dynamic base register caching: a technique for reducing address bus width", Proc. of 18th annual Intl. symposium on computer architecture, May 1991, pp 128-137.
- [7] B.R. Iyer and D. Wilhite, "Data compression support in databases", Proc of 20th VLDB conference, Santiago, Chile, 1994, pp. 695-703.
- [8] R. Iyer, "Exploring the Cache Design Space for Web Servers," Invited Paper, International Parallel and Distributed Processing Symposium (IPDPS'01), May 2001.
- [9] R. Iyer, "Performance Implications of Chipset Caches in Web Servers," submitted to an international conference, Oct 2001.
- [10] R. Iyer, et al., "A Trace-driven Analysis of Sharing Behavior in TPC-C", *2nd Workshop on Computer Architecture Evaluation using Commercial Workloads*, 1999.
- [11] D. Kirovski, J. Kin, W.H. Mangione-Smith, "Procedure based program compression", Proceedings of 30th annual IEEE/ACM International Symposium on Microarchitecture, 1997, pp. 204 -213
- [12] M. Kjelso, M. Gooch, S. Jones, "Empirical study of memory-data: characteristics and compressibility", IEE Proceedings on Computers and Digital Techniques, Vol 145, No 1, Jan. 1998, pp. 63 -67
- [13] M. Kjelso, M. Gooch, S. Jones, " Design and performance of a main memory hardware data compressor", Proceedings of the 22nd EUROMICRO Conference, Beyond 2000: Hardware and Software Design Strategies, 1995, pp. 423 -430
- [14] Jang-Soo Lee, Won-Kee Hong, Shin-Dug Kim, "An on-chip cache compression technique to reduce decompression overhead and design complexity", Journal of systems Architecture, Vol 46, 2000, pp 1365-1382.
- [15] C. Lefurgy, E. Piccininni, T. Mudge, "Evaluation of a high performance code compression method", Proceedings. 32nd Annual International Symposium on Microarchitecture, 1999, MICRO-32, pp. 93 -102
- [16] H. Lekatsas and W. Wolf, "Code compression for embedded systems", Proc. 35th design automation conf., 1998.
- [17] P. Mohapatra, H. Thanthy and K. Kant, "Characterization of Bus Transactions for SPECweb96 Benchmark," 2nd Workshop on Workload Characterization (WWC), Oct 1999.
- [18] P. Ranganathan, K. Gharachorloo, et al., "Performance of Database Workloads on Shared Memory Systems with Out-of-Order Processors," Proceedings of the Eighth International Conference on Architecture Support for Programming Languages and Operating Systems, Oct. 1998.
- [19] S. Roy, R. Kumar, M. Prvulovic, "Improving system performance with compressed memory", Proceedings 15th International Parallel and Distributed Processing Symposium, Apr 2001, pp. 630 -636
- [20] K. Sayood, *Introduction to Data Compression*, 2nd edition, Morgan Kaufmann, 2000, chapter 5.
- [21] "SPECweb99 Design Document," available online on the SPEC website at <http://www.specbench.org/osg/web99/docs/whitepaper.html>
- [22] Transaction Processing Performance Council, TPC BENCHMARKTM C Standard Specification, <http://www.tpc.org/>, Jan. 2000.
- [23] R.B. Tremaine, T.B. Smith, et. al., "Pinnacle: IBM MXT in a memory controller chip", IEEE Micro, March-April 2001, pp 56-68.
- [24] U. Vallamsetty, P. Mohapatra, R. Iyer and K. Kant, "Improving the cache performance of network intensive workloads", Proceedings of the International Conference on Parallel Processing, 2001.
- [25] T.A. Welch, "A technique for high-performance data compression", IEEE Computer, pp 8-19, June 1984.
- [26] P.R. Wilson, S.F. Kaplan and Y. Smaragdakis, "The case for compressed cache in virtual memory systems", Proc. USENIX 1999.
- [27] Jun Yang, Youtao Zhang, R. Gupta, "Frequent value compression in data caches", Proc. of 33rd annual IEEE/ACM Intl Symposium on Microarchitecture, 2000. MICRO-33, 2000, pp. 258 -265
- [28] J. Ziv and A. Lempel, "A universal algorithm for data compression", IEEE trans. on information theory, Vol IT-23, No 3, pp 337-343, May 1977.