

# Design and Performance of Compressed Interconnects for High Performance Servers

*Abstract*—As microprocessors scale rapidly in frequency, the design of fast and efficient interconnects becomes extremely important for low latency data access and high performance. Furthermore, in a multiprocessor configuration, the width of the shared interconnect can pose a significant hurdle in terms of design complexity, cost, and achievable interconnect frequency. In this paper, we evaluate a technique for reducing the interconnect width by exploiting the spatial and temporal locality in communication transfers (addresses & data). The width reduction implies a number of other advantages including higher operating frequency, reduced pin-count, lower chip & board cost, etc. We evaluate the effectiveness of the proposed scheme by performing trace-driven simulations for two well-known commercial server workloads (SPECweb99 and TPC-C). We also study the sensitivity of the compression hit ratio with respect to the number of bits compressed, size of the encoding/decoding table used and the replacement policy. The results indicate that the proposed technique has a potential to reduce address bus width in most cases and data bus widths in some cases while maintaining equal or better performance than in the uncompressed case.

## I. INTRODUCTION AND MOTIVATION

With the rapidly increasing processor speed comes the need for a matching improvement in system bandwidth and latency. To support high bandwidth in the platform, designers constantly look for improvements in the latency and bandwidth of the platform interconnect. In a multi-processor platform, the speed and design time of the interconnect is limited severely by the width of the interconnect. At the same time, it is necessary to increase either the speed or the width of the interconnect to provide higher bandwidth. Compression has been proposed as a solution for more effective utilization of available storage including compressed disk cache [11], [16], compressed main memory [1], [14], and compressed caches [10], [17]. In this paper, we attempt to use compression as a scheme to reduce the communication data transferred in a system, thereby reducing either the width of the required interconnect or the amount of data transported.

Our focus in this paper is to design simple compression schemes that are applicable in the communication context since small amounts of data tend to be transferred between agents that are interconnected within a platform. The compression scheme is primarily geared towards improving the performance and efficiency of the transfer medium (busses, links etc) and associated protocols. While the compression scheme we propose and evaluate in this paper can be applied to any general interconnect, its efficacy is limited by the types of data that the interconnect is used to communicate. High order bits in basic data types often show a low entropy. This observation has been exploited in [16] 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 that when dealing with structured data, it is useful to do compression in the units in which the data items are

accessed. In this paper, we look for similar simple schemes and evaluate its effectiveness for a front-side bus (FSB) in a dual-processor and quad-processor server as discussed below.

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 higher performance bus-based systems in the future, we find three key design pressures: (1) to scale in transfer speed comparable to CPU 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 increasing number of processors sharing the bus. 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 cross-talk and potentially higher frequencies. Achieving higher (or even similar) performance with narrower busses also helps the design time and cost significantly since fewer lines now have to be routed through the same limited area on the board. Finally, 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 web server and OLTP workloads on servers. We evaluate the potential of the basic compression techniques for two commercial benchmarks – SPECweb99 [12] and TPC-C [13]. The evaluation is based on analyzing several memory reference traces collected on real systems and then feeding them into a compressed interconnect simulator that we call SCOT (Simulator for Compressed Transfers). Based on our 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, the size of the encoding/decoding table used, the associativity of the table and the potential of sharing tables among multiple agents. Finally, we discuss the overall implications of these compression schemes on system/benchmark performance, design and cost.

Related work in the area of compression are as follows. Several papers including [11], [16], [9] have investigated techniques that use a LRU main-memory cache to hold pages and files evicted out to disk in compressed form. The purpose is to reduce disk I/O (due to paging or file I/O) thereby significantly improving the performance for applications that require a large amount of memory. An even more extensive use of compression

involves storage of all main memory data in compressed form. IBM's MXT technology [1], [14], takes this approach in order to reduce the need for large memory capacity on servers. With the considerable interest in compression in the main memory, several studies have examined compressibility of main memory data and started to look for specialized compression algorithms [3], [9]. Compression at the L1 and L2 cache level has also been considered. Reference [17] considers a frequent value compression scheme to replace frequently occurring values in the cache into indices to a table containing the frequent values. 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.). The major problem with this scheme is the determination of most frequent values and the latency of  $h$  additional table lookup.

The rest of this paper is organized as follows. Section II presents the basic premise behind the address and data compression schemes and its design implications in terms of table organization and bus protocol changes. Section III provides an overview of our evaluation methodology covering details of workloads and traces. Section IV presents the salient results and provides a detailed analysis of the benefits. Finally, section V summarizes the paper and presents a direction for future work in this area.

## II. OUR FOCUS: COMPRESSION OF ADDRESS/DATA TRANSFERS IN SERVERS

The nature of the information can often be exploited for reducing the number of bits needed in the representation. For example, reference [4] 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 [2] 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.

### A. 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.

### B. 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

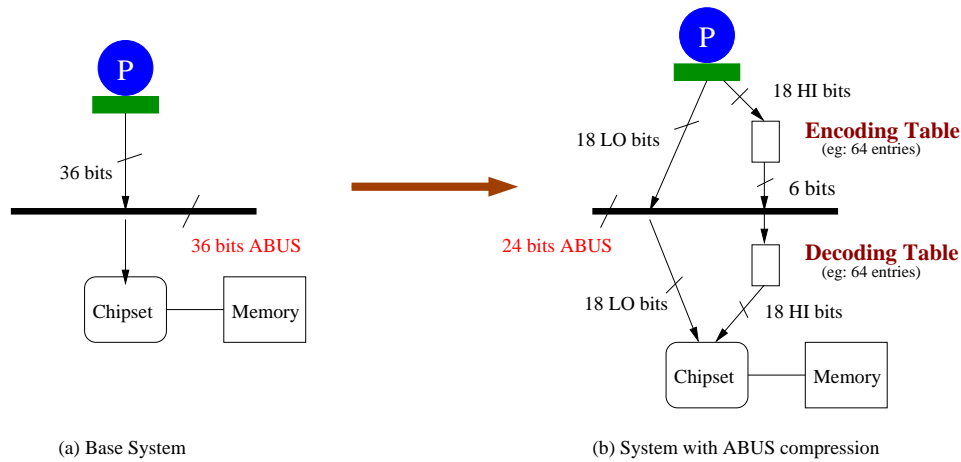


Fig. 1. Illustration of Basic Compression Scheme (eg: for Address Transfers)

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) 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.

### C. Design Issues and Tradeoffs for Compressed Interconnects

The compression scheme described above for address and data transfers needs to be evaluated and characterized for its potential effectiveness. We will address this in the next section. In this section, we discuss the design implications of this scheme in terms of table organization and any necessary protocol changes.

The main storage required for the above scheme to work is the encoding/decoding table on every agent (numbered 0 to N-1) that transmits or receives information. This can be done by having N tables on each agent, with each table storing the entry based on the transmitter. In cases where the interconnect is shared among multiple agents (e.g. a bus shared between multiple CPUs and the chipset) the tables can also be shared among the agents as long as there is a point of serialization. The serialization point is required so that a change/replacement to the table at the receiving and sending end are not done asynchronously, causing a race condition. Additionally, the replacement policy for the two matching tables should be exactly the same on both ends. The following example explains the need for serialization. In a two-agent system with only one shared table at each end, it could happen that an entry X replaces an entry Y on one node’s (A’s) table, whereas at the same time, a request for entry Y arrives at the other node (say B) to be sent to node A. If there is no serialization point, node B would assume that there exists an entry Y in its table and the receiver’s (A’s) table. However, node A has actually replaced that entry and will use a wrong high address value when this request is looked up on its end. A serialization point is available in systems with a bus-based interconnect shared among all the CPUs. In systems that use point-to-point interconnects like serial links, this is not feasible and therefore each node needs to maintain an individual table for every agent in the system that it is directly connected to. In the next section, we will evaluate both shared as well as split tables to understand the performance requirements. It is also possible to construct a hierarchy of these tables, but it may be too complex to maintain efficiently.

Another important issue in table organization is the associativity since full associativity is usually impractical. We will also evaluate multiple replacement policies (FIFO, LRU and a customized variant of LRU) to understand the replacement policy

that provides the most benefit.

The final issue that needs to be kept in mind is the impact of compression on the communication protocol. For example, in the case of a link-based interconnect, the communication protocol should allow for messages of arbitrary length packets or at least packets of several pre-determined lengths to allow for compressed and uncompressed message transfer. In a bus-based system, uncompressed transfers will require two cycles whereas the compressed ones take only one cycle. Functions such as address snooping and delivery of critical chunk need to accommodate this.

### III. 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.

#### A. 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 [12] 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 characteristics defined via a 2-level Zipf distribution. 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 [13] 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. For our study, we used TPC-C traces from a 2P and 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.

#### B. 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. We start by evaluating the characteristics for a full-associative table for each pair of agents. 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.

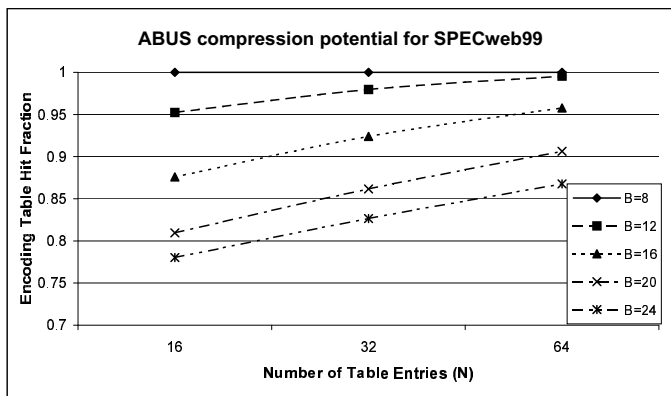


Fig. 2. SPECweb99 Addr Compression: Varying Block Size, Table Size

We then study the impact of different replacement strategies (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 [15] and thereby leads to better performance than plain LRU.

After evaluating the replacement policy, we also evaluate the sensitivity of this scheme to the degree of associativity in the encoding/decoding table. Finally, we evaluate the benefits of sharing tables among multiple agents (which is feasible only for systems with shared interconnects). In particular we also evaluate whether any given agent in the system achieves significantly hit ratios as compared to the other agents.

Finally, we will attempt to provide an initial analysis of the overall system-level performance of the compression scheme as compared to the base system without compressed interconnects. We achieve this based on our analytic performance model that allows us to evaluate the benefits of latency and bandwidth improvements in the platform on overall commercial workload performance.

## IV. SIMULATION RESULTS AND ANALYSIS

### A. 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 under the FIFO replacement policy. 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. The parameter  $B$  in the graphs represents the number of high-order bits that are encoded. Obviously, a small value of  $B$  (e.g., 8 or 12) yields excellent hit ratio, but it doesn't reduce the bus width very much. On the other hand, a large value of  $B$  not only gives a poor hit

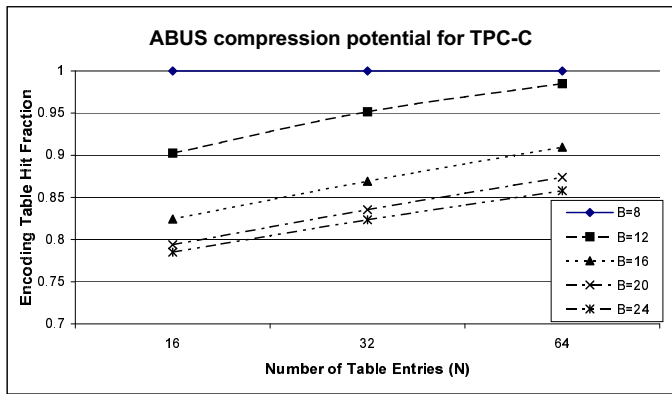


Fig. 3. TPC-C (32-bit) Addr Compression: Varying Block Size, Varying Table Size

ratio but also might provide “too much” compression. To see the latter point, consider the case of 32-bit Intel systems with 36-bit address bus. If we want to make sure that the uncompressed transfers require no more than 2 bus cycles, the compressed bus width must be at least 18 bits wide. Thus, nothing is gained by squeezing the bus width to  $< 18$  bits for compressed transfers. In view of this, a good value for  $B$  in Figs 2 and 3 is 20. This yields a hit ratio of 85-90%, and the total number of address lines needed is 22 (16 low order bits + 6 bit table index). This corresponds to a 39% width reduction which is quite substantial. These results verify that the address bus compression scheme continues to work for commercial server workloads and for much larger caches than those considered in the original reference [4].

We experimented with a 64-bit system configuration as well. Here we chose the  $B = 24$  bits (out of a total of 44 bits) for compression. We find that a 85-90% hit ratio needs a larger table size of 256 entries here, 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 II. 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.

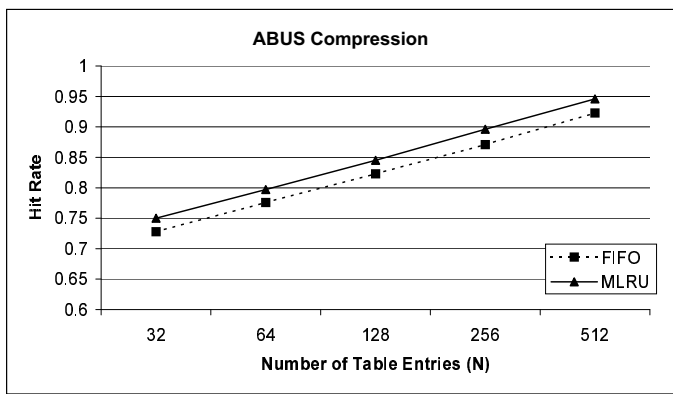


Fig. 4. TPC-C (64-bit) Addr Compression: Varying Table Size, Replacement Policy

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

TABLE I  
DATA TYPE ACCESS FREQUENCIES IN TPC-C

### B. 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

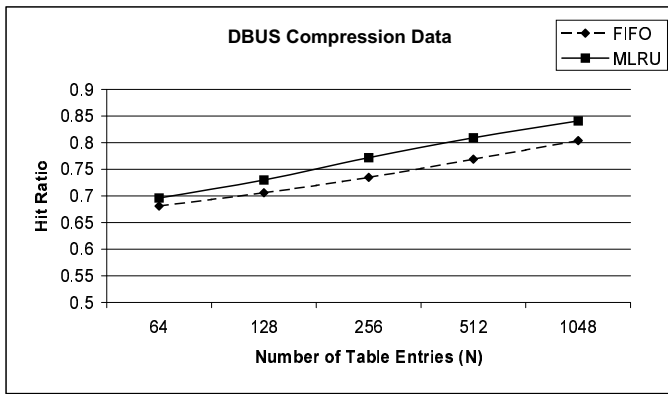


Fig. 5. TPC-C (32-bit) Data Compression: Varying Table Size, Replacement Policy

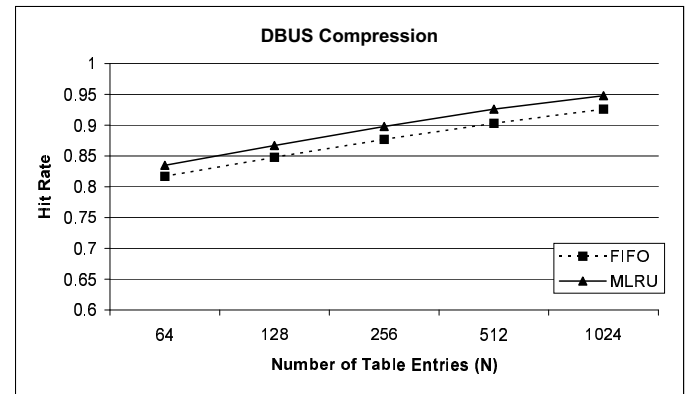


Fig. 6. TPC-C (64-bit) Data Compression: Varying Table Size, Replacement Policy

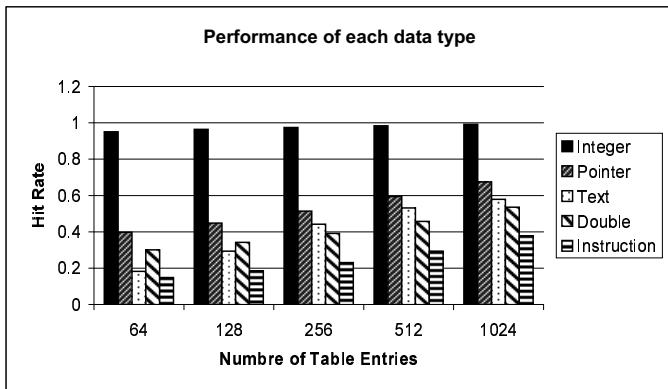


Fig. 7. TPC-C (32-bit) Data Type Specific Hit Rates

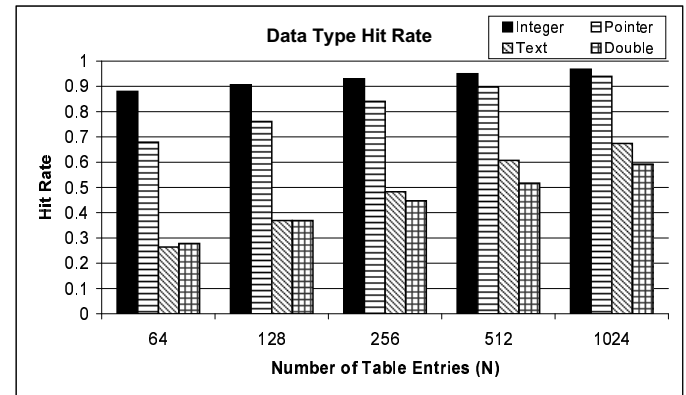


Fig. 8. TPC-C (64-bit) Data Type Specific Hit Rates

for each data type as a result. The data type access frequency is shown in Table I. From the 64-bit data shown, we find that the access to integers dominates the 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 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.

### C. Impact of Associativity

Figure 9 shows the sensitivity of ABUS and DBUS table hit-ratio to degree of associativity. The graphs are for TPC-C on a 4P and 2P system respectively. It is seen that there is a significant jump in hit-ratio from direct-map (SA=1) to 2-way (SA=2) table structure, but subsequent increase in associativity only results in minor improvements. From these figures, it appears that an 8-way set associative structure shows performance very close to that of fully associative structure. Also, the results do not show much change between 2-way and 4-way SMP systems. Almost identical results were seen for SPECweb99 as well (not shown). Since 8-way caches are relatively inexpensive to implement, it appears that associativity is not an hurdle to implementing interconnect compression.

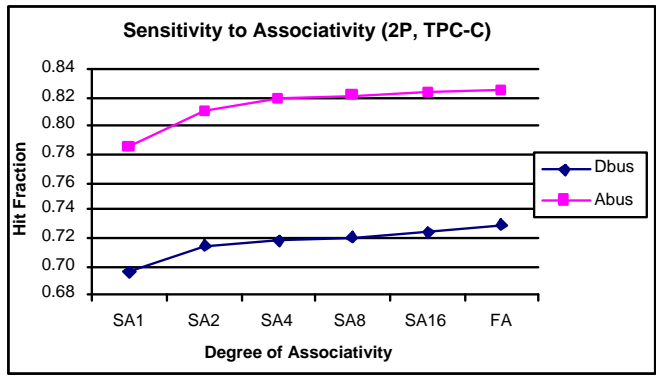
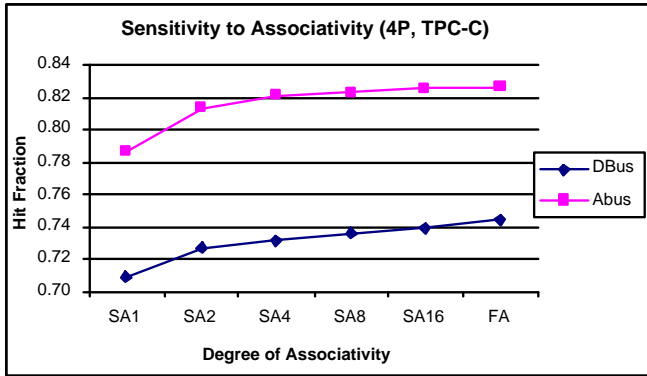


Fig. 9. Sensitivity of Abus & Dbus tables to associativity (TPC-C)

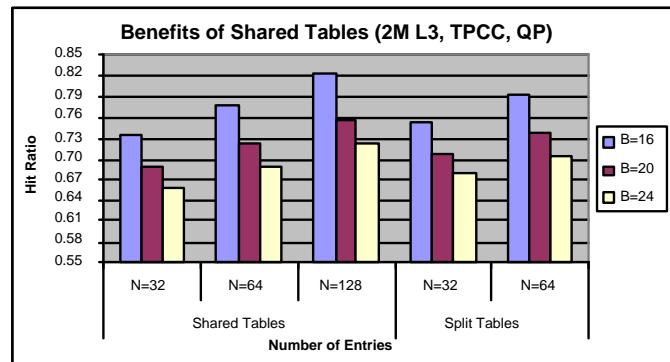
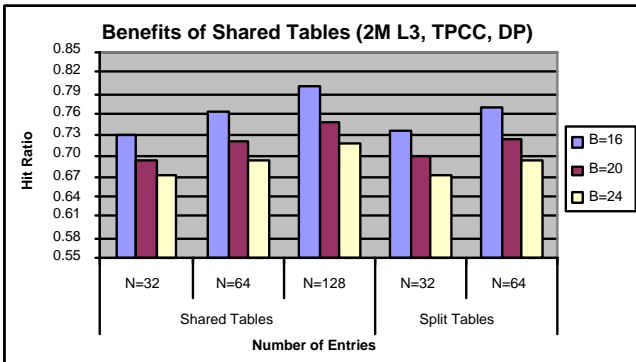


Fig. 10. Hit ratios for Shared vs. Split Tables (TPC-C, CPUs only)

#### D. Impact of Shared Tables

As stated in section II-C, in the case of shared communication medium such as a bus, each agent needs to maintain a table for communication with every other agent. That is, with a total of  $n$  agents, each agent needs to maintain  $n - 1$  tables. In the simplest case, all these tables can be kept separate – we call this the “split table” case. The alternative is to combine all tables into one “shared table” in order to exploit the commonality between the transfers this agent and all others. In the normal case where all CPUs share the entire workload (including interrupt handling), one could expect a significant commonality between the address and data streams of all processors. Consequently, for the same total amount of table storage, the shared table should be able to achieve higher hit ratio than individual (or split) tables. However, the shared table of the same size requires more bits for indexing (2 more bits in a 4-CPU system) which tempers its advantage.

Figure 10 compares the hit ratios for shared vs. split tables for 2P and 4P TPCC-C systems (each with 2MB cache). The results shown correspond to the address bus. The parameter  $B$  here specifies the number of high order bits used for compression. That is, the 3 sets of bars shown are for 16, 20 and 24 high-order bits (out of a total of 36 bits). We assume that the table sharing, if any, is only among the CPUs – the chipset bus transactions are handled via a separate table. The reasons for making this assumption were as follows:

1. Chipset address stream is quite different from processor ad-

dress stream, but at the same time has much higher locality. Thus, mixing it with CPU stream may be undesirable.

2. The bus traces used in our experiments only contained the addresses from the chipset side (Note that all such addresses must be snooped on the processor bus, but data follows a separate path to memory).

These results (and others, not included here due to lack of space) show that shared tables with the same overall size indeed have higher hit ratio than split tables (e.g., size 32 case for split table, and size 64 for shared table in the DP cases). (The behavior for different values of  $B$  is almost identical.) Furthermore, this improved hit ratio is about the same as one would expect from double-size split tables (e.g., compare size 64 tables in split and shared cases for DP). This shows that different processors do show a significant sharing among them. This result is observed in all the cases shown; i.e., two different workloads, 2 different cache sizes, and 2 different SMP sizes and thus could be considered to be typical. However, the crucial characteristic here is that in all cases the CPUs are not bound specifically for any particular tasks. If such a binding were present (e.g., binding all interrupts to a specific CPU, which happens frequently), the shared tables may not show the improvements noted here.

#### V. SUMMARY AND DISCUSSION

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.

A reduction in bus widths accrues several advantages including the cost reduction at chip and board level due to fewer parallel lines and the potential for higher speeds. These advantages become more important as the gap between the processor and interconnect speeds widens and running wide busses becomes increasingly expensive.

In this paper we concerned ourselves with processor interconnects; however, the same ideas can be applied to others, including I/O hub to memory controller interconnect, interconnection network for multiple processing cores on a die, etc. In particular, the nature of I/O hub to memory controller transfers depends on the application and could be made highly compressible by using the appropriate technique. For example, in case of a web-server, most of such data transfer is text which can be compressed easily by using Huffman encoding and the like. Examination of such extended techniques and handling of mixed data type environments (where compression technique is based on "guesses" about the data type) are interesting areas for future work.

#### 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] 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.
- [3] D.J. Craft, "A fast hardware data compression algorithm and some algorithmic extensions", IBM Journal of R&D, Vol 42, No 6.
- [4] 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.
- [5] B.R. Iyer and D. Willite, "Data compression support in databases", Proc of 20th VLDB conference, Santiago, Chile, 1994, pp. 695-703.
- [6] ———, "An Evaluation of Memory Compression Alternatives", Proc. of CAECW (Computer Architecture Evaluation using Commercial Workloads), Feb 2003, Anaheim, CA.
- [7] D. Kirovski, J. Kin, W.H. Mangione-Smith, "Procedure based program compression", Proceedings of 30th annual IEEE/ACM International Symposium on Micro-Architecture, 1997, pp. 204 -213
- [8] 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
- [9] 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
- [10] 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.
- [11] 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
- [12] "SPECweb99 Design Document," available online on the SPEC website at <http://www.specbench.org/osg/web99/docs/whitepaper.html>
- [13] Transaction Processing Performance Council, TPC BENCHMARK C Standard Specification, <http://www.tpc.org/>, Jan. 2000.
- [14] R.B. Tremaine, T.B. Smith, et. al., "Pinnacle: IBM MXT in a memory controller chip", IEEE Micro, March-April 2001, pp 56-68.
- [15] ———, "Improving the cache performance of network intensive workloads", Proceedings of the International Conference on Parallel Processing, 2001.
- [16] P.R. Wilson, S.F. Kaplan and Y. Smaragdakis, "The case for compressed cache in virtual memory systems", Proc. USENIX 1999.
- [17] Jun Yang, Youtao Zhang, R. Gupta, "Frequent value compression in data caches", Proc. of 33rd annual IEEE/ACM Intl Symposium on Micro-Architecture, 2000. MICRO-33, 2000, pp. 258 -265