Techniques for Increasing Effective Data Bandwidth

Christopher Nitta, Matthew Farrens
University of California, Davis
[cjnitta, mkfarrens]@.ucdavis.edu

***Abstract*—In this paper we examine techniques for increasing the effective bandwidth of the microprocessor off-chip interconnect. We focus on mechanisms that are orthogonal to other techniques currently being studied (3-D fabrication, optical interconnect, etc.) Using a range of full-system simulations we study the distribution of values being transferred to and from memory, and find that (as expected) high entropy data such as floating point numbers have limited compressibility, but that other data types offer more potential for compression. By using a simple heuristic to classify the contents of a cache line and providing different compression schemes for each classification, we show it is possible to provide overall compression at a cache line granularity comparable to that obtained by using a much more complex Lempel-Ziv-Welch algorithm.**

I. Introduction

Microprocessors have long had to deal with the problem of memory bandwidth constraints. Burger, Goodman, and Kagi [1] and [2] discussed the impact of bandwidth limits on future microprocessor designs, and emphasized the importance of increasing the *effective* pin bandwidth. As we now move into an era of multicore processors, the pressure on the I/O pin bandwidth is going to further intensify, making this an even bigger concern.

The bandwidth problem can be addressed in two orthogonal ways: by increasing the bandwidth available, and by decreasing the amount of bandwidth that is required. Bandwidth reduction techniques consist primarily of the extensive use of on-chip memories (mainly caches) to avoid having to go off-chip, while the bandwidth available can be increased by boosting the number of transactions per unit time (i.e. increasing the bus frequency), and/or by expanding the amount of data transferred per transaction (creating a wider interconnect).

Although microprocessor pin counts have steadily climbed over the past 15 years, this increase in pins has not translated into wider off-chip data interconnects. Microprocessors have increased memory bandwidth primarily via a dramatic increase in bus frequencies. For example, in 1993 Intel released the Pentium 75, which featured a 64-bit data bus running at 50MHz in a 296 pin PPGA package. Recently Intel released the Core 2 Extreme QX9650, which has a 64-bit data bus running at 1333MHz, packaged in the LGA775. The QX9650 has over 2.6 times the number of pins and a burst speed over 26 times that of the Pentium 75 [3][4], but the data bus is the same width.

Bus speed will not continue to increase indefinitely, however. Microprocessor core clock frequencies grew dramatically during the decade of the 1990's, but since 2000 (when 1GHz microprocessors first became commercially available) clock rates have climbed much less quickly. There is reason to believe that a similar slowdown of bus frequency acceleration will occur as bus speeds climb above 1GHz, although researchers are currently studying approaches (such as 1 TB/s differential signal paired I/O [5] and optical interconnects [6]) that may allow the increase in off-chip frequencies to climb a while longer.

The most common technique for reducing the amount of bandwidth needed is to place part of the memory hierarchy on-chip, reducing the number of off-chip references necessary. However, it is also possible to use data compression techniques to reduce the pressure on the off-chip interconnect by increasing the *effective* bandwidth – that is, by increasing the amount of useful information (the *information content*) sent over each I/O pin on each transaction, thus requiring fewer transactions.

It is this potential to increase the effective bandwidth that we will focus on in this paper. Section II describes related work that has been done in this area, while Section III discusses our simulation configuration and initial data analysis (which lead to a cache line classification heuristic). In Section IV we detail the compression algorithms analyzed and propose a compression technique for cache lines, and our results appear in Section V. Our conclusions and possible future work are provided in Section VI.

II. Related Work

Over the years, researchers have looked at a variety of ways to more effectively utilize I/O pins. Compressing the number of address lines was examined in [7], and in [8], [12], and [13] the compressibility of data in various benchmarks was analyzed. However, the previous studies primarily focused on either the compression *ratio* or lookup-table hit rates; this can be misleading, since a higher compression ratio does not directly translate into a higher effective bandwidth if more transactions or larger uncompressed transactions are required to achieve the improved compression ratio. This previous work also did not analyze full system traces, but focused on a series of application level benchmarks.

Storing data in compressed form in main memory has been investigated by [9][10], while a unified compressed memory hierarchy has been analyzed in [11][12]. The

978-1-4244-2658-4/08/$25.00 ©2008 IEEE

authors of these studies argue that a compressed memory hierarchy can improve program performance, but it is unclear if the added complexity of maintaining a compressed memory system and the handling of corner cases will be justified by the overall performance gains. Storing compressed data in the cache increases the effective size of the cache, but as we will show an increased cache size does not translate into as large a reduction in required bandwidth as interconnect data compression alone.

A wide range of compression algorithms have been evaluated in the previous work, ranging from the relatively simple and straightforward (Frequent Value Caching) to the more complex (Lempel-Ziv). There are also several versions of the Lempel-Ziv (LZ) compression algorithms to choose from - LZ77, Lempel-Ziv-Welch (LZW), and LZSS (a parallel version of LZ77.) LZ77 tends to perform poorly on small block sizes, such as single cache lines, while LZW is faster but typically does not achieve the quality of compression of LZ77 when larger block sizes are used. LZSS was analyzed in [11] for their unified compressed memory hierarchy.

Certain values appear frequently in program data. Both [14] and [15] attempt to exploit this data value commonality by the use of a Value Cache (VC), a structure which holds frequently occurring patterns. VC algorithms typically use a single bit to encode a hit or a miss in the value cache, and then transmit either the corresponding index into the cache (in the case of a hit) or the entire value (in the case of a miss). Both static and dynamically loaded value caches have been studied, and these structures can store part or all of the desired transmitted value. A VC is relatively easy to implement when compared to more complex compression schemes, and appears to work well for small data blocks. One downside of a VC is that the data structures on both ends of the interconnect must remain synchronized, but this should not be a problem for small table sizes.

Run Length Encoding (RLE) encodes runs of identical values into a (value, run length) pair. This compression technique is efficient on data with long runs of identical data values, which many uncompressed images contain. A Move To Front (MTF) transformation is often used prior to Zero Length Encoding ZLE (a special case of RLE) in the hopes of capturing more runs of zeros. However, RLE and ZLE algorithms tend to do poorly on data that has high entropy (such as floating point values).

Many techniques have been proposed for compressing floating point numbers. Floating point value prediction was investigated in [17] and achieved compression ratios between 1.2 and 4.2. This algorithm predicts what the next floating point value will be and then sends the distance between the predicted and actual values, with leading zeros compressed. Prediction is also used in [18] and [19], but the distance is transformed into an integer which is then encoded into entropy codes and raw bits.

With the exception of [10] (which evaluated an existing system but focused on compressed memory and not interconnect data) we are unaware of any previous work that has performed its analysis on full-system simulations as we have done.

III. Simulation Configuration And Data Analysis

In order to properly evaluate a data compression scheme for a real system, ***all*** the information going across the data lines (including data generated by the operating system) must be included. Therefore, we used a full-system simulator to gather our data. The data generated by the simulator included the base address of each reference, whether it was a read or a write, the data in the cache line associated with that address, and the method by which the microprocessor accessed the cache line. The four methods of access tracked by the simulator were instruction read, page table access, general load/store, and floating point load/store.

A. Full-System Configuration and Simulations

The simulator we chose to use was a modified version of bochs 2.3.5, which simulates an x86-based system. The simulated system was configured to have 512MB of RAM, a 2GB HDD (ext3), 512MB HDD (swap), and a CD-ROM. We modified the simulator to include a 16 way set associative write back L2 cache with 64 byte cache lines. Cache sizes of 1, 2, 4, and 8MB were run for all simulations. The simulated system was booted using Knoppix 5.1.1, a GNU/Linux bootable live CD image.

To generate our data we started the simulator, booted into Linux, and then executed a variety of programs meant to represent typical user behavior. During an entire session (from bootup until simulation termination) the memory activity at the L2/memory interface was gathered and written to a file for later analysis.

There were four different sessions:

1) To simulate a casual user working on a document while listening to music, the OpenOffice session consisted of opening and converting to postscript a 100 page OpenOffice document while an mp3 was being decoded in the background. Upon successful conversion of the document, the postscript file was opened using the Konqueror browser.
2) The SPEC2000 session consisted of running all the integer and floating-point benchmarks, with the exception of eon, perlbmk, galgel, facerec, lucas, and fma3d. (The four floating-point benchmarks that were not run were due to problems with Fortan 90, while eon and perlbmk were not run due to problems with compilation on the simulated system.) The “test” inputs were used during the runs.
3) During the Linux kernel build session the Debian kernel was built for the i386 target. The build configuration was done prior to running the session and remained consistent for all runs.

4) Since the system bootup occurred during all sessions, one session consisted solely of booting the machine, so that any bias towards an uninitialized system could be removed in later analysis.

Each session was run using one of four different cache configurations, so there were a total of 16 data files created (each approximately 30GB after being compressed).

B. Initial Data Analysis

Because the simulator output included information regarding the way the data was accessed (instruction read, floating point load, etc.) we knew the type of data within each line, and we made use of this information when doing our data analysis. We discovered that many cache lines contained values that represented small positive or negative 32-bit integers. We also found that the values 0x3E, 0x3F, and 0x40 occurred often in the 62:56 bits of the 64-bit aligned floating point data. This was not surprising, considering that these values represent small positive and negative exponents. We were unable to find any pattern or correlation in the mantissa - all possible byte values appeared with an almost perfectly uniform distribution.

Knowing how a data value is referenced, and the fact that each type exhibits different characteristics, one can create type-specific compression mechanisms. While we had access to this information because the simulator provided it, in a real system the data itself does not contain any indicator of its type. Thus, it was not clear how knowing the type information was going to be useful for anything other than calculating the limits of compressibility. However, after extensive analysis we identified a relatively simple technique for classifying cache lines that works quite well.

C. Heuristic Classification

To maximize compression, the heuristic developed classifies an entire cache line as being one of *Integer, Floating point, Pointer*, or *Other*, even though there may be a mix of types within a line. A line is classified as a particular type if 50% or more of the values in the cache line are identified as being of that type, and the value identification is done as follows: A value is considered to be an *Integer* if the most significant byte (MSB) of a 32-bit quantity is either 0x00 or 0xFF. *Float* values are identified as those where the MSB of a 64-bit quad word is 0x3E, 0x3F, 0x40, 0xBE, 0xBF, or 0xC0 (these patterns indicate a floating point value with a small positive or negative exponent). A *Pointer* is assumed if the most significant 16-bits of a 32-bit word match any other most significant 16-bits within the cache line. (Values of 0x0000 and 0xFFFF were ignored in order to avoid conflicts with the integer classification). Any cache line that is not identified as containing at least half of one of these types is classified as being *Other*. (As we were writing this paper we discovered that [13] had previously proposed a different approach to classification that bears many similarities to ours.)

Since a cache line is 64 bytes wide, it can hold 16 32-bit quantities, 8 64-bit quantities, or some combination of both. We assume that all 16 possible integer MSBs, all 8 possible floating point MSBs, and all 8 possible pointer MSB pairs will be evaluated in parallel, although this would not be necessary if it was not in a time-critical path.

IV. Evaluation of Compression Techniques

In order to evaluate the compressibility of the cache lines we used a variety of compression techniques for each full system simulation data set, including ones that are unlikely to be implementable with a reasonable amount of hardware. The compression techniques were analyzed using data sets obtained from the simulations. The evaluated techniques included LZW, LZ77, Delta Encoding (DE), a Sorted Delta Encoding (SDE), Value Caching (VC), and our proposed algorithm (described later in this section). The LZ77 algorithm was chosen because of previous related work, while the LZW was chosen for ease of implementation and improved performance on small block sizes.

The LZW implementation uses a 9-bit fixed length code for dictionary entries. The 9-bit code length was chosen since it could accommodate all single byte values as well as the 64 byte cache line. Due to the relatively small size of an individual cache line, our LZ77 sliding window implementation works on a nibble level instead of a byte level. In the DE algorithm each value is considered to be either 32 or 64 bits wide, and this size is kept constant for an entire analysis run. The arithmetic difference between consecutive values is calculated and is encoded into a {size, value} pair, where the size specifies the number of equal size segments required to represent the delta value. A full range of segment size granularity (1 bit up to width/4) was evaluated. The smaller the segment granularity the tighter the delta can be fit (there are fewer possible leading zeros in the delta), but this tighter fit comes with a tradeoff of higher static overhead for the size. The SDE algorithm is similar to the DE except the delta values are sorted prior to encoding. This algorithm has higher static overhead due to the fact that the ordering of the values must also be transmitted, but benefits from the fact that smaller average delta values require fewer bits and typically provide better overall results. The VC algorithm simulated treats each value as a 32-bit quantity. The entire 32-bit value is stored in the VC and cache sizes of 16, 64, 256, and 1024 entries were analyzed. (The 1024 entry VC would be unreasonably large, but it was included for completeness sake.)

The algorithm we are proposing, the Type-Specific Compression (TSC) algorithm, classifies cache lines using the heuristic discussed above and then uses a tailored compression algorithm for each of the four classifications. We will now explain this algorithm in more detail.

A. Type-Specific Compression

Value caches are much easier to implement than the more complex compression schemes, which makes them very

attractive. In TSC, we use several value caches, which employ a least frequently used replacement scheme. Each time an entry is found in the value cache, its frequency counter is incremented. This will bias the cache towards retaining commonly occurring patterns. However, to prevent a value from becoming permanently lodged in the cache, the frequency counters for each cache entry are also decremented at fixed regular intervals. This ensures that a given line must be referenced with some minimum frequency or it will become a candidate for eviction.

TSC uses value caches of different sizes and configurations, depending on the type of data that is being compressed. The *Pointer* classification uses a 16 entry 32-bit wide cache, which can provide a best case compression of 6.4:1 with a worse case growth of 3.13%. The *Other* classification similarly uses a 16 entry cache, but the entries are 16-bits wide, providing a best case compression of 3.2:1 per value with a worse case growth of 6.25%. (The 16-bit entry size was implemented for the *Other* category based on the observation that x86 instructions are variable length.)

Floating point data is highly entropic; as discussed earlier, this is especially true of the mantissa. The floating point exponent, however, appears to have somewhat lower entropy and therefore is slightly more compressible. This led us to use a value cache for the cache lines tagged as *Float* that has 16 entries that are each 14-bits wide. The 11-bit exponent and the three most significant bits of the mantissa are cached, giving a best case compression of 1.16:1 and a worse case growth of 1.5%. (A more conservative approach to dealing with this type was chosen due to the uncertainty that a value in a cache line marked as *Float* is actually a floating point value. The large required tables and the possibility of poor prediction due to interference from concurrent applications led us to abandon methods previously described in Section II.)

Integer cache lines use a hybrid approach - zeros and negative ones are removed prior to the value caching step. (This substitution is described in more detail in the next subsection.) A 16 entry 32-bit value cache was implemented for *Integer* cache lines, which in the best case will result in a 6.4:1 compression ratio. It should be noted that our relatively small value tables (188 bytes total) will require little additional overhead and should be easy to synchronize.

B. Zero and Negative One Substitution

Our initial data analysis revealed that two 32-bit values (0x00000000 and 0xFFFFFFFF) occurred with a high frequency in lines classified as *Integer*. We exploit this information by replacing the all zero pattern with the 2-bit pattern 0b00 and the all one pattern with 0b01. We call this technique Zero and Negative One Substitution (ZNOS). All values other than zero and negative one are prepended with a 1 before being placed in the value cache, which stores full 32-bit values. The worse case for *Integer* cache lines using TSC is an increase of 2-bits per 32-bit value (a 6.25% growth), with a best case compression of 16:1.

V. RESULTS

The bandwidth required by the Office, SPEC, and Kernel Build sessions with the system bootup information removed is shown in Figure 1. The **Single LZW** column is the amount of compression achieved by the LZW algorithm using no data classification information, while the **Heuristic** and **Access** columns show the best possible compression for a given L2 cache size assuming the data type is known (**Access**) or predicted (**Heuristic**). Within each column, different combinations of algorithms may have been used – the best performing algorithm for each data type is selected, including those algorithms we believe may not be reasonably implementable.

The best performing compression algorithm differs by each classification when using the **Heuristic** classification. The best performing *Integer* algorithm is the ZNOS with VC, while the *Other* data is best compressed by using LZW. The 64-bit and 32-bit SDE algorithms perform the best for *Float* and *Pointer* cache lines, respectively.

The **Single LZW** slightly outperforms the **Access** classification for all cache sizes (although it is very slight and difficult to see) because the LZW algorithm was the best one for all of the **Access** classifications at all cache sizes – thus, the slightly better performance overall of the **Single LZW** is due to the fact that it does not require the extra two bits per cache line that have to be added in order to indicate the classification of the line.

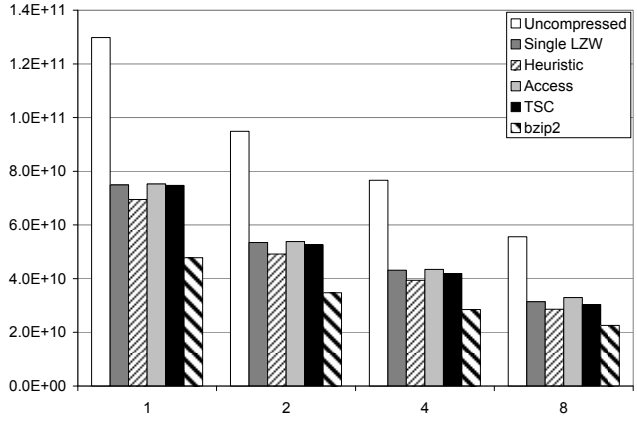

Figure 1. Total Required Bandwidth (Bytes) vs. L2 Cache Size (MB), All Sessions Combined

The **TSC** column represents the Type-Specific compression technique described in the previous section. It is worth noting that the **TSC** method slightly outperforms the **Single LZW** compression method in all cases. It is also worth noting that all the compression methods require much less bandwidth than an uncompressed cache that is double the size, and they all have bandwidth requirements that are comparable to caches four times the size. (However, it is still unclear if this reduction in bandwidth will translate into an overall comparative performance gain.)

To create the **bzip2** entry we ran bzip2 on the original data file. This provides an idea of the amount of compression available overall, and was added in order to gain perspective on the effectiveness of the compression algorithms analyzed. The **bzip2** value should be near the theoretical limit of compression for the data. (It is the large block size, 900k default, which makes the **bzip2** value unobtainable in practice).

Note that **Heuristic** (using our heuristic to classify the data) outperforms **Access** (knowing how the data was accessed) for all L2 cache sizes - this somewhat counter-intuitive result is due to the fact that the heuristic classifies the cache lines based on the actual values themselves, rather than on how the microprocessor accessed the data. A detailed breakdown of the traffic reduction (higher is better) for both **Access** and **Heuristic** is shown in Figure 2.

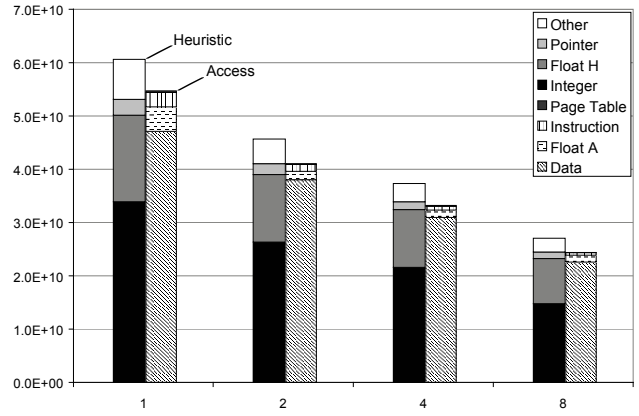

Figure 2. Heuristic and Access Traffic Savings (Bytes) vs. Cache Size (MB)

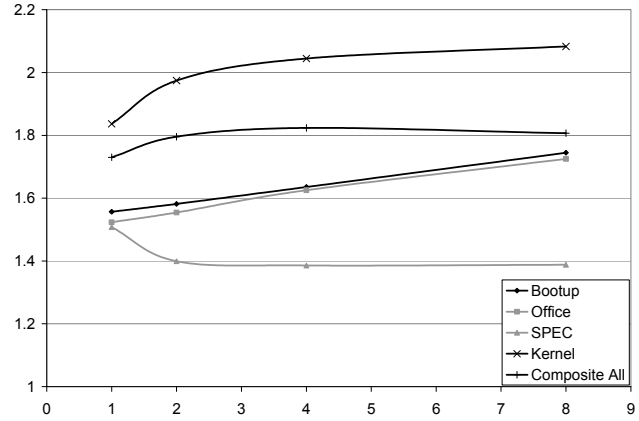

Figure 3. Simulation Compression Ratio Using TSC Algorithm vs. Cache Size (MB)

The compression ratios for each simulation and the composite of the Office, SPEC, and Kernel Build using the TSC algorithm are illustrated in Figure 3. These ratios were calculated by dividing the uncompressed data size by the compressed data size, and they range from 1.388 to 2.083. Note that for all simulations (with the exception of the SPEC simulation) the compression ratio increased as the L2 cache size grew. This is because more lines are being classified *Integer* and fewer marked *Other*. Unfortunately, as the cache size increases for the SPEC simulations there is a shift from lines marked *Integer* to lines marked *Float* - this is shown in Figure 4. It appears that increasing the L2 cache size does not affect how long the SPEC data resides in the cache, since the SPEC simulations were designed to isolate the CPU capability and the SPEC data may already be capable of residing in the cache for the duration. However, increasing the cache size may be allowing lines marked *Integer* that were brought in by the Operating System to remain in cache longer, thereby reducing the number of times they have to go across the bus.

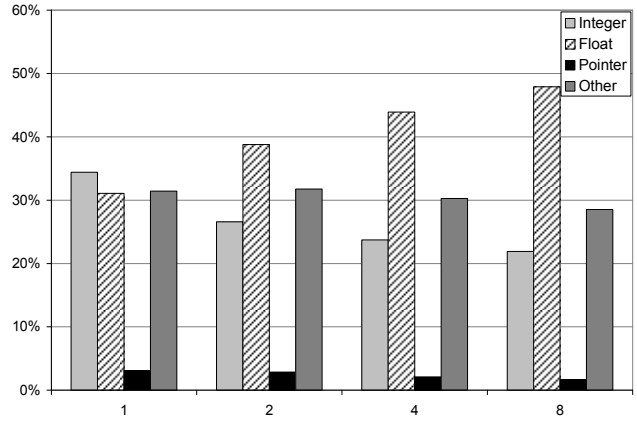

Figure 4. SPEC Heuristic Classification Occurrence by Cache Size

Figure 5 shows the compression ratio of individual heuristic classifications by cache size, using the TSC algorithm on the composite simulations. As expected, the *Integer* classification is the most compressible, with a compression ratio averaging 3.062. The average compression ratio of the *Float* is 1.094, which is within 5.5% of the theoretical maximum for the algorithm chosen. The *Pointer* classification average compression ratio is 1.854, with a 57% hit rate in the 16-entry table.

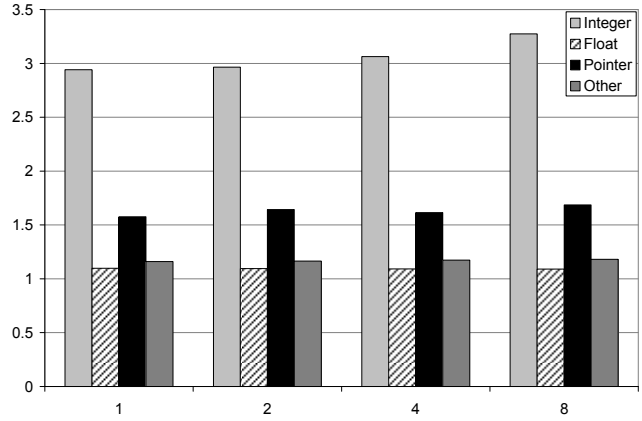

Figure 5. Classification Compression Ratio of TSC Algorithm by Cache Size

The compression ratios of the evaluated algorithms can be seen in Figure 6. The compression ratios are based upon the composite of the Office, SPEC, and Kernel Build data and

are categorized using the heuristic classification. Only the best granularity for the DE and SDE algorithms are displayed for sake of brevity. (This is also true of the VC algorithms - only the best table size is shown). What is immediately noticeable is the impact on the compression ratio of the ZNOS algorithm on *Integer* cache lines. The SDE algorithms outperform the other algorithms on the *Float* cache lines, achieving a 1.143 and 1.155 compression ratio for the 32-bit and 64-bit versions, respectively. It should be noted that these ratios approach the theoretical maximum of the TSC algorithm, but do not exceed it.

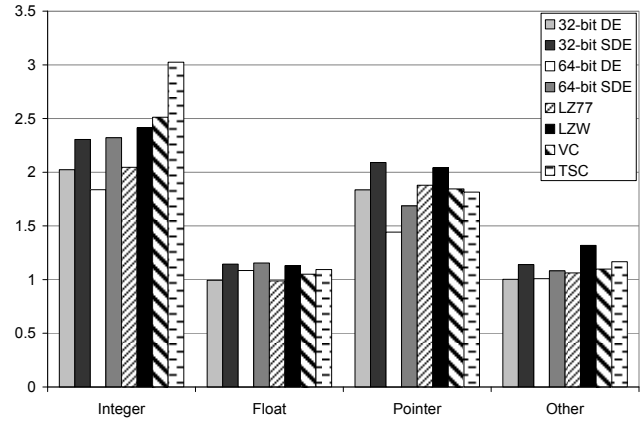

Figure 6. Compression Ratios of Individual Algorithms by Heuristic Classification

VI. Conclusions and Future Work

In this paper a simple heuristic for classifying cache lines and a Type-Specific Compression (TSC) algorithm are presented. This approach, requiring fewer than 200 extra bytes of storage, was evaluated using a series of extensive full-system simulations. Our results indicate TSC is able to significantly increase the effective pin bandwidth by reducing off-chip traffic by 44% on average, to a level lower than that required by a cache four times the size that does not use compression. We also show that TSC outperforms a more complex LZW algorithm by 2% on average, and our **Heuristic** method outperforms the **Access** method on average by 10%.

It is important to note that our compression technique is orthogonal to techniques aimed at increasing the actual bandwidth; furthermore our technique may reach its fullest potential when combined with an optical or differential signaling interconnect since serialized data transfers do not waste potential bandwidth (due to the fact that any transfer size is always a multiple of the 1-bit bus width.)

The small percentage of cache lines being classified as *Pointer* encourages us to further investigate other possible classification heuristics that may have the potential for providing higher effective bandwidth. A full timing and power analysis of the evaluated compression algorithms also needs to be done in order to determine the overall impact on performance and power consumption.

References

[1] D. Burger, J. Goodman, A. Kägi, "Limited Bandwidth to Affect Processor Design," *Micro*, IEEE vol. 17, pp. 55–62, 1997.

[2] D. Burger, J. Goodman, A. Kägi, "Memory Bandwidth Limitations of Future Microprocessors," Proceedings of the 23rd annual International Symposium on Computer Architecture, 1996.

[3] *Pentium Processor*, Intel Corporation, Mt. Prospect, IL, 1997, Available: ftp://download.intel.com/design/pentium/datashts/24199710.PDF

[4] *Intel® Core™2 Extreme Processor QX9000Δ Series and Intel® Core™2 Quad Processor Q9000Δ Series Datasheet*, Intel Corporation, 2008, Available: http://download.intel.com/design/processor/datashts/318726.pdf

[5] H. P. Hofstee, "Future Microprocessors and Off-Chip SOP Interconnect," Advanced Packaging, IEEE Transactions on, vol 27, pp. 301-303, 2004.

[6] B. E. Lemoff, M. E. Ali, G. Panotopoulos, G. M. Flower, B. Madhavan, A. F. J. Levi, D. W. Dolfi, "MAUI: Enabling Fiber-to-the-Processor With Parallel Multiwavelength Optical Interconnects," Lightwave Technology, Journal of, vol 22, pp. 2043-2054, 2004.

[7] M. Farrens, A. Park, "Dynamic Base Register Caching: A Technique for Reducing Address Bus Width," Computer Architecture, The 18th Annual International Symposium on, pp. 128-137, 1991.

[8] D. Citron, L. Rudolph, "Creating a Wider Bus Using Caching Techniques," High Performance Computer Architecture, 1995, Proceedings. First IEEE Symposium on, pp. 90-99.

[9] M. Kjelsø, M. Gooch, S. Jones, "Design and Performance of a Main Memory Hardware Data Compressor," Proceedings of the 22nd EUROMICRO Conference, 1996.

[10] B. Abali, H. Franke, X. Shen, D. Poff, T. B. Smith, "Performance of Hardware Compressed Main Memory," High Performance Computer Architecture, 2001, Proceedings. Seventh International Symposium on, pp. 73-81.

[11] E. Hallnor, S. Reinhardt, "A Unified Compressed Memory Hierarchy," High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, pp. 201-212.

[12] A. Alameldeen, D. Wood, "Interactions Between Compression and Prefetching in Chip Multiprocessors," High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, pp. 228-239.

[13] K. Kant, R. Iyer, "Compressibility Characteristics of Address/Data Transfers in Commercial Workloads," Proc. of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads, pp.59-67, 2002.

[14] Y. Zhang, J. Yang, R. Grupta, "Frequent Value Locality and Value-Centric Data Cache Design," Architectural Support for Programming Languages and Operating Systems, vol 34, pp. 150-159, 2000.

[15] M. Lipasti, C. Wilkerson, J. Shen, "Value Locality and Load Value Prediction," Architectural Support for Programming Languages and Operating Systems, vol 30, pp. 138-147, 1996.

[16] E. Hallnor, S. Reinhardt, "A Compressed Memory Hierarchy using and Indirect Index," Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st International Symposium on Computer Architecture, 2004.

[17] P. Ratanaworabhan, J. Ke, M. Burtscher, "Fast Lossless Compression of Scientific Floating-Point Data," Data Compression Conference, 2006. DCC 2006. Proceedings, pp. 133-142.

[18] P. Lindstrom, M. Isenburg, "Fast and Efficient Compression of Floating-Point Data," Visualization and Computer Graphics, IEEE Transactions on, pp.1245-1250, 2006.

[19] M. Isenburg, P. Lindstrom, J. Snoeyink, "Lossless Compression of Predicted Floating-Point Geometry," Computer-Aided Design and Applications. Vol. 1, no. 1-4, pp. 495-501. 2004.