An Evaluation of Memory Compression Alternatives

KRISHNA KANT, Intel Corporation
(krishna.kant@intel.com)

Abstract: **This paper presents a comparative study of three different memory compression schemes, full memory compression and two forms of compressed disk cache. The full memory compression maintains all of the primary storage in compressed form whereas the compressed disk cache compresses only the memory pages that would otherwise be "freed" (and thus the corresponding content retrieved from disk on a later access). The two forms of the latter scheme considered are a purely software solution and a hardware assisted solution. The paper discusses the advantages and disadvantages of each scheme and evaluates their relative performance via a detailed platform model for SPECweb99 benchmark. The results indicate that the compressed disk cache scheme is not only much cheaper than full memory compression, but can also be quite effective in spite of its limited scope.**

1. Introduction

Memory compression has been considered as a technique for delivering following potential benefits to the commercial server segment:

- Significant savings in memory costs for large and medium servers.
- Savings in physical space, required power, and thermal dissipation by the memory subsystem for high-density servers.
- Reliance on lower performance disk subsystems through the use of the compressed disk cache technique, significantly lowering total system cost.
- Improved efficiencies through reduced memory and I/O subsystem BW requirements and costs from the application of compression end-to-end.

Whether and to what extent these potential benefits are actually achieved depends on the implementation details, which are explored in this paper. Two major variants of memory compression have been explored in the past: (a) full memory compression (FMC) and compressed disk cache (CDC). A brief overview of the requirements of these is included here.

1.1 Full Memory Compression (FMC)

Full memory compression keeps the entire memory compressed (with the possible exception of some specialized regions such as DMA). In order to localize the changes needed to support compressed memory, the O/S initializes the system with certain amount of uncompressed memory (e.g., twice the physical memory) and all accesses are to this ***real address space***. The accessed addresses are eventually converted to the compressed space addresses (***physical address space***) by the memory controller before the actual access is initiated. The access would retrieve the compressed memory block, decompress it and provide it to the processor. Since decompression is a slow process, acceptable performance requires a chipset cache that maintains the recently used uncompressed data. FMC is best illustrated by IBM's MXT (memory extension technology) that includes the following components [ibm-mxt, pinnacle]:

1. 32 MB of fast (SRAM) chipset cache.
2. Memory compressed in blocks of 1 KB size. Compressed blocks are stored using 1-4 segments, each of size 256 bytes.
3. A compressed block is accessed via a header entry that contains pointers to the 4 segments and other relevant information. For blocks that compress to 64:1 or better, it also allows for an "immediate data" type of representation (in which case all 4 segment pointers would be null).
4. The chipset provides a hardware compression-decompression unit (henceforth called **Codec**) based on a variant of the LZ77 compression algorithm [ibm-lz].
5. The chipset also provides the TLB (similar to paging TLB) for address translation between real and physical address spaces.

Since FMC involves "in-line" decompression, low decompression latency is a lot more important than the compressibility. A detailed modeling of the FMC scheme shows that a small compression block size is preferable from the latency perspective (except that a small size would lead to larger address translation tables and hence a poorer TLB hit ratio for the same TLB size). A block size of 256 bytes appears to be near optimal from the performance perspective. A chipset cache with no-load latencies that are 50-75% of memory access latencies and a size in the range of 16-32 MB appears essential to mitigate the latencies of decompression and achieve about the same level of performance as the case with increased physical memory (and no compression).

The MXT solution uses a highly parallelized implementation of the traditional ***LZ77*** file compression algorithm [ibm-lz, lz-77]. Although a codec based on this

algorithm is expensive, it achieves better compressibility than other algorithms that are especially designed with hardware implementation in mind. In particular, the ***X-match pro*** algorithm works with 4 bytes at a time and also encodes partial matches [x-match]. Furthermore, run-length encoding is used for repetitive strings. A Codec based on this algorithm is claimed to provide a similar decompression rate at a much lower cost, however, the compression ratio achieved by this algorithm is usually significantly lower than that achieved by LZ77. In our performance comparisons in section 2, we assume the use of X-match-pro algorithm for FMC since the cost of a highly parallelized LZ77-based Codec would make FMC unattractive.

As stated above, MXT 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, up to 4 separate accesses may be needed to retrieve the entire block. An alternate scheme is to attempt to store all segments of a block contiguously. This allows for a shorter header and more efficient access. However, the implementation must deal with storage fragmentation and the cost of ***fat-writes*** (i.e., an increase in compressed block size when it is modified and written back, thereby requiring reallocation in a different place).

1.2 Compressed Disk Cache (CDC)

Compressed disk cache is intended to act as a buffer between the main memory and the disk. A portion of the main memory is designated as a CDC, and memory pages evicted from the regular memory are compressed and stored in the CDC so that future accesses to these pages can avoid disk I/O (so long as these pages have not been evicted out of CDC). Thus, only the misses out of CDC require disk I/O. Generally, evicted dirty pages are not written to CDC directly; instead such pages are queued up for writing to disk and thereby made "clean". This avoids a direct IO to/from the CDC space.

Several flavors of CDC have been examined in the past [page-cache1, page-cache2]. The most limited use is in form of a ***compressed swap cache***, where hard page faults are intercepted to look for the required page in the cache, and freed clean pages are posted into the cache. Such an approach may be useful for workstation applications under severe memory limitations, but is generally not useful for server applications that typically manage their paging activity carefully. A more general use of CDC involves storing not only the pages freed by the paging activity but also the files evicted out of the OS file cache. In Microsoft Windows, the swap file is cached like an ordinary file in the file cache, so a ***compressed file cache*** can easily hold pages evicted by both the memory manager and the file cache manager (FCM). In general, a ***compressed disk cache*** should be able to hold pages evicted from any of the IO buffering agents including the buffer cache manager (BCM) of a DBMS, web cache manager (WCM) of a web server, OS file cache manager (FCM), etc. *In any case, due to a large space needed for the compressed disk cache, a dynamic cache size adjustment is necessary to ensure that the CDC does not starve the caches maintained by OS or application.*

It is clear from the above that in order to be universally useful, the CDC must be usable by not only the OS but also any application that does large scale IO caching. This implies the need for a uniform API for invoking the CDC functionality. For simplicity, from now on we only speak of CDC used to maintained pages evicted from the OS file cache; however, the need for a general interface must be kept in mind.

It should be clear that CDC would be a worthwhile approach only if (a) a substantial portion of the main memory is occupied by some kind of IO cache, and (b) the workload can benefit from the availability of extra memory. Both of these conditions are generally true for servers. If the memory manager led page evictions are also covered by the CDC (as would be the case in MS Windows), these conditions also apply whenever the workload requires huge amounts of memory that causes substantial paging. *In other cases, CDC wouldn't be very useful. Note that FMC could be beneficial for a wider spectrum of applications since it requires condition (b) only.*

CDC helps improve performance in two ways: (a) Significantly lower latency associated with page retrieval from CDC, and (b) Reduced path-length associated with this retrieval. With respect to (b), it is possible to consider two variants of CDC:

1. **CDC-FD**: In this case, CDC is implemented purely as a "filter driver", i.e., it intercepts the IO requests and satisfies them out of the CDC w/o explicit knowledge of the OS. The compression/decompression is assumed to be implemented in hardware and is treated like a DMA capable IO device. All management of the compressed address space is done in software.
2. **CDC-OPT**: This is an optimized version of CDC-FD where (a) Lookup & address translation are accelerated via extra hardware, and (b) File cache manager (FCM) is aware of the CDC and thus can work with it directly.

For convenience, all hardware associated with compression/decompression (Codec, address translator, buffers, etc.) is referred to as **compression decompression engine (CDE)**. The basic motivation for considering CDC-OPT is to minimize the path-length and MPI impact associated with accessing the compressed cache. The latency reduction over CDC-FD is not expected to yield any measurable benefit (since the access latency of a DMA capable CDC-FD implementation is already in several microsecond range, which is far smaller than the IO

latency). A possible implementation of CDC-OPT has been worked out but not included here for brevity.

1.3 Relative merits and architectural options

It is instructive to do a gross comparison of the 3 techniques introduced above. The major points regarding full memory compression are as follows:

+ Can double or triple available memory => Significant memory savings.
- Very expensive to implement
  - A large and fast chipset cache.
  - In-line operation => Needs very fast Codec
  - Fast address translation paraphernalia (TLB, translation tables, etc.)
- Latency considerations favor small block sizes
  - Small block size => poorer compressibility (e.g., 3X @ 4 KB vs. 2X @ 128B).
  - Large space requirements for address translation tables.

The major points regarding either form of disk compression are as follows:

+ No chipset cache is required (large savings in gates) => Relatively inexpensive hardware.
+ Software is straightforward for CDC-FD (disk IO request interceptor, no O/S impact).
- Significant O/S impact of CDC-OPT (change in file-caching component of the OS, etc.).
+ Accessed on a miss in regular memory =>
  - Use of a large compression block size (e.g., memory page size) is okay.
  - Slower but more effective compression algorithms (LZ77 vs. X-match-pro) okay.
+ Large compression block size => Good compressibility & easier space management.
-? Less savings in memory & power and correspondingly less performance than FMC.

The reason for a question mark on the last item is that our study indicates that it not always the case that CDC performs poorer than FMC. In fact, as shown in the following, CDC-OPT can actually do better than FMC in certain situations.

With both memory compression techniques, there are several implementation variants that one could consider. For example, with full memory compression, there are multiple choices relative to the location of the chipset cache (embedded in memory controller vs. external), location of CDE (embedded in memory controller vs. on DIMM), storage schemes for compressed segments (contiguous vs. non-contiguous allocation), and chipset cache management granularity (in terms of simply the compression block size, or the finer processor cacheline size). One could also consider speculative decompression schemes in order to reduce access latency at the cost of greater complexity. CDC has similar choices including location of CDE (integrated with memory controller, integrated with DIMM, integrated with an IO bridge, or as a separate device), and different schemes for data transfer between compressor input/output buffer & normal memory. Two possibilities in the latter case are: (a) Copying under processor control (programmed I/O model), and (b) Specialized DMA engine for data transfer.

2. SPECweb99 performance modeling results

The performance of MXT has been reported in several research publications [ibm-mxt, pinnacle]. They report results from experiments conducted on dual-processor systems w/ and w/o MXT running database workloads and using estimation tools on live production servers running web server workloads. They report a compressibility of 2.68 for their database workload (running an insurance company schema on a DB2 database) and an estimated compressibility of 2.1 (for a live web server workload). In overall performance, they show that their database workload runs 25% (1GB memory size) and 66% (512MB memory size) faster w/ MXT enabled. However, they also mention that in these cases the system was memory starved. They also report observing similar memory-dependent performance benefits with the SPECweb99 benchmark (45% performance improvement when increasing memory from 256MB to 512MB). In this paper, we concentrate only on SPECweb99, although we did obtain some rather simplistic results on TPC-C like database which show about 6.5% performance improvement for 4GB system, and 3.5% for 16 GB system.

2.1 SPECweb99 Performance Model

The results in this section are based on a detailed model of SPECweb99 benchmark that includes support for both FMC and both versions of CDC. The model is based on a number of measurements on both Intel Pentium\TM III and IV systems using Microsoft IIS5 as the web server. The model includes the impact of disk I/O both in terms of path length and the IO latency based on a set of measurements with different amounts of installed main memory. The basic model, similar to the one in [kant-sweb96], includes CPUs, processor bus, memory channels, IO busses (chip-to-chip interconnects and PCI), network and disk adapters. The internals of the CPU and the processors caches are not modeled; instead, a high level model coupled with the explicit calculation of MPIs and bus coherence traffic is used. The compression support includes compressor & decompressor units, chipset (or "L3" cache) and memory and bus traffic & latency impact of these resources. It is assumed that the L3 cache is dual-ported thereby allowing lookup and retrieval to progress in parallel. The L3 cache is also assumed to have mixed granularity, i.e., on a miss, a complete block is brought and installed into the cache,

however, lookups and retrieval can occur in the units of processor cacheline size.

In the following we briefly discuss some major points regarding the model. A comprehensive discussion of the model is beyond the scope of the paper.

Basically, the model is a transaction level queuing model of a typical SPECweb99 setup. It assumes a number of client machines, each represented by a single queue. Each client runs a number of processes, each of which cycles through sleeping, generating a request, and waiting for the response (or file) from the server. The client is connected to the server via an Ethernet network, which is modeled rather simplistically in terms of pure delay & queuing delays. On the server side, each process obtains a thread, performs request processing, and releases the thread. Processing a client request involves the following 6 service phases:

- Reception of the client request (including PCI transfers and memory to memory copies).
- Computation: All processing including host-side memory reads/writes.
- Disk reads and writes (including PCI transfers and memory to memory copies).
- Sending of response to client (including file-cache lookup and reading, PCI transfers, memory to memory copies for dynamic content).

Each phase generates 3 auxiliary transactions in the bus-memory subsystem:

- **Bus invalidation**: Required for claiming exclusive access to a shared cacheline. This transaction goes through address bus only and dies.
- **Implicit Writeback**: Generated as a result of a hit-modified (HITM) situation. In this case, the original transaction goes through the FSB and the implicit writeback goes through memory pipeline and dies.
- **Explicit Writeback**: Generated as a result of cache eviction of modified data.

For efficiency, the emulation of bus-memory transactions is done in "chunks", i.e., batches of 10's of cachelines. A side effect of this is some additional traffic burstiness that isn't there in reality. The disk I/O is modeled in units of 4 KB blocks (memory page sizes) and network sends in units of packets (1.5 KB). The memory-to-memory copies related with IO, memory controller-IO hub transfers, and PCI bus transfers are also modeled explicitly. In particular, PCI transfers are emulated on a burst-by-burst basis.

The model represents address and data busses (ABUS & DBUS) as separate queuing stations and memory as a delay station in series with a queuing station. The memory stations are calibrated based on a 4-stage memory model (e.g., RAS, CAS, data access & bus data transfer in case of reads). The dead cycles occurring on memory channels and DBUS are also modeled. Every read/write memory transaction stays in IOQ (in-order queue) until DBUS transfer is completed (or until the end of the snoop phase for deferred transactions). Following DBUS transfer, a memory write enters MRQ (memory request queue) and stays there until completion. Bus invalidations are initiated probabilistically and go through the IOQ as well. HITMs are also initiated probabilistically and create implicit writebacks. The writeback of the modified data to memory and its delivery to the processor over proceed in parallel. The model also supports deferred bus transactions, but their need and treatment is clearly dependent on the platform being modeled.

The chipset cache (L3) is accessed via the bus and intercepts all processor side accesses to the memory; however, the PCI side memory accesses do not go through the L3 cache. Note that the L3 access size is same as the compression block size, which can be much larger than the processor cacheline size (512 bytes vs. 64 byte respectively). It is assumed that the installation of a line in L3 cache is in its natural units, but the retrieval from the processor side can be in processor cacheline units. Such a policy complicates L3 design but minimizes access latencies. It is further assumed that the L3 cache is dual ported such that all lookups queue up on one port whereas data retrievals queue up on another. The lookup is assumed to be completed in one bus cycle. The L3 peak data rate is assumed the same that of the memory, but the overall latency will generally be smaller than that of memory access. Note that the introduction of L3 cache introduces a synchronization issue since all accesses to it must return data in the correct order to the processor. Thus, any hits into L3 must wait behind a miss of an earlier transaction.

Next, we briefly describe modeling of compression and decompression. We assume that reading compressed data from memory and its decompression are pipelined, with compressed data put into the input buffer of the decompressor one cacheline at a time. The data is made available to the processor as soon as the desired cacheline has been decompressed. The decompression service time is modeled as a fixed overhead plus a per byte decompression latency. The assumptions for compression are similar. In particular, reading data from L3, compressing it, and writing it to memory are all pipelined. As expected, the compressed data is put into the memory input buffer one cacheline at a time.

The model includes several other details in a manner similar to the ones in [sweb96] in order to handle performance projections for a variety of configurations. In particular, the model can automatically scale the misses per instruction (MPI) for L2 cache based on the L2 size and the throughput level. The scaling allows the MPI's for the baseline configuration to be converted to the appropriate MPIs for the configuration of interest. Since this aspect is not central to this paper, it is not discussed here. The L3 cache MPI's were obtained based on separate simulations of chipset cache fed with SPECweb99 traces. No further scaling of these MPI's is done in the model based on L2 size, throughput level or other model parameters. This is clearly a simplification, but perhaps a reasonable one. The

model can also do a variety of other scaling to account for factors such as prefetching, cacheline size differences, etc., but these again are not central to the discussion here. The model computes the bus coherence traffic based on a Markovian model of the MESI protocol that we have developed.

The model also attempts to do a rather detailed accounting of memory requirements and computation of the disk IO rates. Part of this is based on the file caching study of SPECweb99 in [sweb99]. (Actually, for computational efficiency, we use a 2-segment spline fit to the results quoted in [sweb99]). A somewhat surprising observation here is that file-caching requirements do not dominate either the overall memory requirements or the disk IO rates; instead, the dominant factor is the DLL buffers used by the ISAPI implementation (and the IO resulting from lack of buffers). ISAPI is required in SPECweb99 to handle the dynamic content. Scaling of DLL buffer sizes and the corresponding change in IO rates are computed based on simple power-law equations derived from a series of measurements. The size of the system cache is yet another variable component of the overall memory requirements, but its size is usually much smaller than the file-cache and the DLL buffers.

One complexity in estimating disk IO requirements is "equitable" distribution of available memory between various buffers (or caches). This essentially amounts to a proper memory allocation for file-cache and DLL buffers. An imbalanced memory allocation would overestimate the disk IO requirements and hence will result in suboptimal throughput estimate. Based on the measurements, a good strategy appears to be to cache about 20% of the file-set and about 40% of the total DLL space required. The model starts with these as the "claims" and divides the available memory weighted by these claims. The disk-write rate estimation also needs to include HTTP log writes and POST log writes; these parts are usually fixed and easily estimated.

The throughput impact of disk IO is difficult to estimate accurately. The three basic quantities needed in estimating workload throughput are path-length, MPI and access latencies. Disk I/O may alter all these quantities. In particular, a higher disk IO per transaction implies:

- More I/O management overhead and more threads required => Increased path-length.
- More context switches and cache flushes => Higher MPI.
- Greater chance of not being able to hide the I/O latency => Increased CPU stalls.

The path-length impact is relatively straightforward in that the path-length can be adjusted linearly based on the number of IO's/transaction and path-length per IO. We model the other two impacts indirectly via an increase in CPI (cycles/instruction). In particular, we use the following equation:

$$CPI_{tot} = CPI_{mem} + BF_{disk} * MPI_{mem} * LAT_{disk}$$

where

BF_{disk}: Fraction of disk read latency that is visible to CPU.

MPI_{mem}: Memory access MPI (misses out of memory resulting in a disk block read).

LAT_{disk}: Latency for one disk read operation.

CPI_{mem} : Total CPI assuming infinite memory.

The MPI_{mem} is easily estimated from the disk read rate and path length and LAT_{disk} is estimated directly from the model. Finally, BF_{mem} estimated by matching measured & projected throughputs for a set of memory sizes. Because of heavy measurement dependencies and inadequate validation across more than one IO subsystem, it is not clear how generally applicable this CPI estimation approach is. However, the projections done for the system under measurements estimated the IO impact quite accurately.

Given the overall path-length and CPI, the workload throughput can be computed easily. (Note that for SPECweb99, the real metric is simultaneous connections, which is closely related to the throughput. In particular, we consistently found about 2.8 trans/sec per simultaneous connection.)

Although the model is built around a simulation package, it is primarily solved analytically by assuming product form and treating each queue as a GI/D/1 or GI/G/1 queue, as appropriate. The blocking delay in IOQ is difficult to estimate accurately – a simple Erlang-C type of formula is used to estimate the blocking probability and hence the additional latency due to IOQ being full. The entire analysis requires iteration since most of the parameters (MPIs, memory requirements, disk I/O, blocking delays, etc.) depend on the throughput (the final outcome of the model). We have devised an iterative procedure that has been found to converge quickly in all cases except when the bottleneck device utilizations reach unsustainable levels. The entire model has been validated using a number of measurements available on both Intel Pentium III and 4 platforms.

2.2 Sample Results from the Model

For generating the results quoted in this section, an important parameter is the CDC access path-length for retrieving a disk block as opposed to doing an actual IO. We assume that this path-length is 30% that of disk IO path-length based on some prototype implementations of ram-disk type of capability; however, a better estimate is necessary to be sure. With the CDC-OPT implementation, access to the data in the CDC involves much fewer instructions (basically 2 DMA setups/completions plus lookup in translation tables in case of a TLB miss). Data writes into the CDC also involves very few instructions. The precise instruction count has not been determined for these results, instead, the minimum possible instructions were assumed. *Thus, the improvement shown here by CDC-OPT over CDC-FD may be somewhat overstated*. Latency

implications of CDC access are also accounted for in the model, but the latency impact on CPI turns out to be very small in almost all the cases.

Tables 1-3 show some sample results obtained from this model w/ and w/o compression. Instead of considering a current platform for this evaluation, we thought that might be more useful to consider a more futuristic platform. Accordingly we considered a hypothetical Intel Pentium\TM IV based platform with a 6.0 GHz processor, 1 MB second-level (L2) cache, 266 MHz bus and a DDR 533 memory. These numbers are essentially double of current values -- they are not based on any actual future platform and no effort was made to account for architectural differences from current platforms. The disk subsystem is also assumed to be twice as fast as the system on which model calibration is based both in seek/rotation delays and data transfer rate.

As stated earlier, we assume the use of X-match-pro algorithm for FMC for reasons of much lower cost/complexity. For CDC, LZ77 is a better choice, however, in order to show a side-by-side comparison with the same algorithm, we show the performance with X-match-pro as well. The used compression algorithm is shown in the Tables within parenthesis as "X" or "L" in the **compression ratio** column.

The columns and values listed in the following tables are as follows:

- **Compression technique**: Following possibilities are examined
  1. *Base*: Baseline case (no compression, 1 MB L2 cache).
  2. *Large L2*: Baseline case with 2 MB L2 cache. This situation quantifies L2 related performance delta as compared to compression related delta.
  3. *CDC-OPT*: CDC-OPT implementation with the given main memory cache size.
  4. *CDC-FD*: CDC-FD implementation with the given main memory cache size.
  5. *FMC*: FMC implementation with the given chipset cache size and latency.
  6. *None*: No compression but with the given chipset cache size and latency. This situation quantifies pure chipset cache related performance delta as opposed to the compression related delta.
- **Memory size**: Total physical memory in MB (held constant for all variants).
- **Block size**: Compression block size (same as chipset cache line size for FMC).
- **Compression ratio**: Achieved average compression ratio.
- **Cache size**: *This refers to the size of disk cache for CDC and chipset cache size for FMC*. Each CDC scenario considers two cache sizes: (a) Max cache size that yields the peak performance, and (b) Max size such that the performance drops only about 0.5% below the peak. The motivation for (b) is to maximize the compressed memory w/o hurting the performance.
- **Cache latency**: *This refers to the disk cache (i.e., main memory) latency for CDC and to chipset cache latency for FMC*. Also, this represents only the pure-delay component of the latency in memory clocks. The queuing part (which determines the bandwidth) is 2 memory clocks per processor cacheline. This part, along with memory dead clocks, address & data bus latencies, and queuing latencies make up the total memory access latency, but that is not reported here.
- **SWEB99 opcount**: Estimated ops/sec from the model. (The real SPECweb99 performance metric is simultaneous connections; generally one gets very close to 2.8 ops/sec per connection.)
- **Real memory**: Total memory that the system sees (including the effect of compression).
- **Memory savings**: Computed as real_memory/ physical_memory - 1
- **Perf delta wrt base**: Percentage performance delta over the base case (no compression, no chipset cache).

In the following, 3 cases are shown to exhibit the impact of memory size and memory channel bandwidth.

Case 1: Small memory size (4 GB) and limited memory bandwidth (1 channel[1]).

Case 2: "Reasonable" memory size (8 GB) but limited memory bandwidth (1 channel).

Case 3: "Reasonable" memory size (8 GB) and adequate memory bandwidth (2 channels).

Major observations for Table 1:

1. FMC wins hands down in this scenario (but it requires a CDE 2-4 times as fast as the other techniques). With a slower CDE, the performance advantage will go down.
2. Since CDC works with much larger compression block size (4KB vs 256B) it enjoys better compressibility and hence can match the memory saving achieved by FMC.
3. Since CDC can easily use the LZ77 compression algorithm (which is quite expensive to implement in FMC context), CDC can actually beat FMC in terms of memory savings!
4. The performance boost due to just a large L2 is muted by the fact that the system is disk IO bandwidth limited and large L2 only serves to increase the IO traffic. Consequently,

[1] Depending on the chipset architecture, a "channel" may retrieve only a portion of a cacheline. Here by channel, we mean a "parallel server", i.e., all physical channels that collectively deliver one cacheline are considered to form one logical channel.

compression can yield better performance than a large L2.

5. FMC yields better performance than just the chipset cache because the compression decreases the disk BW requirements.

Table 1 Relative performance of various techniques w/ limited memory size & BW

Case1: 4 GB physical memory, 1 channel									
Comp tech.	***Memory size (MB)***	***block size (B)***	***comp ratio***	***cache size (MB)***	***cache latency***	***SWEB99 opcount***	***Real mem (MB)***	***Memory savings***	***Perf delta wrt base***
Base	4096	64	1(none)	0	10clks	10986	4096	0%	0.0%
Large L2	4096	64	1(none)	0	10clks	12581	4096	0%	14.5%
CDC-OPT	4096	4096	3.33(X)	2400	10clks	12416	9688	137%	13.0%
CDC-OPT	4096	4096	3.33(X)	2600	10clks	12350	10154	148%	12.4%
CDC-OPT	4096	4096	4.76(L)	2100	10clks	12725	11992	193%	15.8%
CDC-OPT	4096	4096	4.76(L)	2500	10clks	12569	13496	229%	14.4%
CDC-FD	4096	4096	3.33(X)	2300	10clks	11831	9455	131%	7.7%
CDC-FD	4096	4096	3.33(X)	2600	10clks	11770	10154	148%	7.1%
CDC-FD	4096	4096	4.76(L)	1600	10clks	12107	10112	147%	10.2%
CDC-FD	4096	4096	4.76(L)	2100	10clks	12035	11992	193%	9.5%
FMC	4096	256	2.33(X)	16	4clks	14044	9544	133%	27.8%
FMC	4096	256	2.33(X)	32	4clks	14629	9544	133%	33.2%
FMC	4096	256	2.33(X)	32	8clks	14052	9544	133%	27.9%
None	4096	256	2.33(X)	16	4clks	12299	4096	0%	12.0%
None	4096	256	2.33(X)	32	4clks	12345	4096	0%	12.4%
None	4096	256	2.33(X)	32	8clks	12072	4096	0%	9.9%

Table 2 Relative performance of various techniques w/ limited memory BW

Case2: 8 GB physical memory, 1 channel									
Comp tech.	***Memory size (MB)***	***block size (B)***	***comp ratio***	***cache size (MB)***	***cache latency***	***SWEB99 opcount***	***Real mem (MB)***	***Memory savings***	***Perf delta wrt base***
Base	8192	64	1(none)	0	10clks	13528	8192	0%	0.0%
Large L2	8192	64	1(none)	0	10clks	16653	8192	0%	23.1%
CDC-OPT	8192	4096	3.33(X)	5000	10clks	14363	19842	142%	6.2%
CDC-OPT	8192	4096	3.33(X)	6200	10clks	14280	22638	176%	5.6%
CDC-OPT	8192	4096	4.76(L)	4500	10clks	14456	25112	207%	6.9%
CDC-OPT	8192	4096	4.76(L)	5600	10clks	14382	29248	257%	6.3%
CDC-FD	8192	4096	3.33(X)	2600	10clks	14037	14250	74%	3.8%
CDC-FD	8192	4096	3.33(X)	4000	10clks	13966	17512	114%	3.2%
CDC-FD	8192	4096	4.76(L)	2500	10clks	14113	17592	115%	4.3%
CDC-FD	8192	4096	4.76(L)	4000	10clks	14036	23232	184%	3.8%
FMC	8192	256	2.33(X)	16	4clks	14800	19087	133%	9.4%
FMC	8192	256	2.33(X)	32	4clks	15527	19087	133%	14.8%
FMC	8192	256	2.33(X)	32	8clks	14834	19087	133%	9.7%
None	8192	256	2.33(X)	16	4clks	15240	8192	0%	12.7%
None	8192	256	2.33(X)	32	4clks	15384	8192	0%	13.7%
None	8192	256	2.33(X)	32	8clks	14699	8192	0%	8.7%

Major observations for Table 2:

1. FMC again shows better performance than CDC, however, FMC performance is mostly a result of the chipset cache. Note that unlike case1, the decrease in disk BW requirements because of FMC no longer yields any significant benefit. Consequently, a system with just the chipset cache does almost as well or better as FMC.
2. A large L2 gives a huge performance boost since IO is no longer a limitation and the memory BW bottleneck is eased by the large L2. *The performance with large L2 is better than that obtained by any kind of compression.*
3. CDC again yields excellent memory savings and beats out FMC in this regard if LZ77 compression algorithm is considered.

Table 3 Relative performance of various techniques w/ adequate memory size & BW

Case3: 8 GB physical memory, 2 channels									
Comp tech.	***Memory size (MB)***	***block size (B)***	***comp ratio***	***cache size (MB)***	***cache latency***	***SWEB99 opcount***	***Real mem (MB)***	***Memory savings***	***Perf delta wrt base***
Base	8192	64	1(none)	0	10clks	14559	8192	0%	0.0%
Large L2	8192	64	1(none)	0	10clks	17075	8192	0%	17.3%
CDC-OPT	8192	4096	3.33(X)	4800	10clks	15699	19376	137%	7.8%
CDC-OPT	8192	4096	3.33(X)	6000	10clks	15615	22172	171%	7.3%
CDC-OPT	8192	4096	4.76(L)	4300	10clks	15823	24360	197%	8.7%
CDC-OPT	8192	4096	4.76(L)	5600	10clks	15739	29248	257%	8.1%
CDC-FD	8192	4096	3.33(X)	2400	10clks	15381	13784	68%	5.6%
CDC-FD	8192	4096	3.33(X)	4000	10clks	15296	17512	114%	5.1%
CDC-FD	8192	4096	4.76(L)	2800	10clks	15472	18720	129%	6.3%
CDC-FD	8192	4096	4.76(L)	4000	10clks	15389	23232	184%	5.7%
FMC	8192	256	2.33(X)	16	4clks	13742	19087	133%	-5.6%
FMC	8192	256	2.33(X)	32	4clks	14283	19087	133%	-1.9%
FMC	8192	256	2.33(X)	32	8clks	13729	19087	133%	-5.7%
None	8192	256	2.33(X)	16	4clks	15281	8192	0%	5.0%
None	8192	256	2.33(X)	32	4clks	15406	8192	0%	5.8%
None	8192	256	2.33(X)	32	8clks	14719	8192	0%	1.1%

Major observations for Table 3:

1. Since there is no memory size or BW shortage in this case, FMC has little to exploit and actually gives performance worse than the base case!
2. CDC still provides a moderate gain because of its ability to further reduce IO without too much overhead.
3. CDC once again saves more memory than FMC. This indicates that compressing the entire memory is not essential for achieving a good memory savings.
4. A large L2 gives much better performance than any of the compression schemes.

The last observation implies that *if there are no significant memory or IO BW limitations to exploit, a large processor cache works provides a better way of boosting performance than memory compression*. However, the memory cost and power saving potential of memory compression still remains.

3. Conclusions and Discussion

With FMC, the CDE (Codec + MMU) must necessarily be located close to the memory since the latencies associated with other locations would be highly detrimental to performance. Two possible locations in these cases are the memory controller or right on the DIMM. However, with CDC, the decompression is needed only on a miss in regular memory, which makes it far less latency sensitive. In this case, it is not critical to locate CDE close to memory controller; instead, it could either be implemented inside an IO bridge. Assuming a 2 GB/sec IO interface between the memory controller and the IO bridge and fully pipelined DMA capability, the round-trip latency for a processor cacheline would be in a few hundred nanosecond range (this includes queuing delays on the IO bus and the chipset). Such a latency addition is much smaller than the base latency (in several µs range) and appears to have a

negligible performance impact.[2] However, if the implementation requires a complete disk block to be transferred across the wire at any point before processing can start, the latency addition may be too much.

Implementing the CDE as a discrete device hanging off an IO bridge will experience another few hundred ns latency, but it opens up CDE for easy enhancements and innovations including running multiple algorithms in parallel, streaming buffers, speculative decompression, larger buffers for translation tables, more sophisticated storage management, etc.

Disk compression opens up the possibility of supporting **end-to-end compression** in a very flexible way. For example data on the disk and data arriving into the NIC may or may not be in the compressed form; the CDE can compress/decompress it as needed. The same holds for data being written to disk or being sent out on the network. If much of the disk and network I/O is in compressed form, the scheme enables increased I/O BW (or alternately, cost and power savings by allowing the use of lower BW I/O subsystems).

Acknowledgements: Ravi Iyer performed detailed chipset-cache simulations with SPECweb99 traces in order to generate miss-ratio and coherence data necessary for the modeling presented in this paper. Raed Kanjo The author would like to express his appreciation for feedback on this work from Warren Morrow, Yamada Koichi, Ravi Iyer and Nitin Singhvi.

4. References

[ibm-mxt] 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

[ibm-lz] D.J. Craft, “A fast hardware data compression algorithm and some algorithmic extensions”', IBM Journal of R\&D, Vol 42, No 6.

[sweb99] K. Kant and Y. Won, ``Performance Impact of Uncached File Accesses in SPECweb99", Proceedings of 2nd IEEE Workshop on workload characterization, Austin TX, Oct 1999. (Published by Kluver).

[sweb96] K. Kant and C.R.M. Sundaram, ``A Server Performance Model for Static Web Workloads", Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS 2000), April 2000.

[2] The main performance advantage of CDC-OPT comes from the reduced path length and far lower latency than disk accesses; a small increase in latency (a few hundred ns on top of a base latency of several microseconds) has little impact on performance.

[charac] 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

[x-match] 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

[selective] 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.

[page-cache1] 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

[pinnacle] R.B. Tremaine, T.B. Smith, et. al., ``Pinnacle: IBM MXT in a memory controller chip", IEEE Micro, March-April 2001, pp 56-68.

[page-cache2] P.R. Wilson, S.F. Kaplan and Y. Smaragdakis, ``The case for compressed cache in virtual memory systems", Proc. USENIX 1999.

[lz-77] 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.