AlphaSort: A Cache-Sensitive Parallel External Sort

Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, Dave Lomet

Abstract *A new sort algorithm, called AlphaSort, demonstrates that commodity processors and disks can handle commercial batch workloads. Using commodity processors, memory, and arrays of SCSI disks, AlphaSort runs the industry-standard sort benchmark in seven seconds. This beats the best published record on a 32-CPU 32-disk Hypercube by 8:1. On another benchmark, AlphaSort sorted more than a gigabyte in a minute.*

AlphaSort is a cache-sensitive memory-intensive sort algorithm. We argue that modern architectures require algorithm designers to re-examine their use of the memory hierarchy. AlphaSort uses clustered data structures to get good cache locality. It uses file striping to get high disk bandwidth. It uses QuickSort to generate runs and uses replacement-selection to merge the runs. It uses shared memory multiprocessors to break the sort into subsort chores.

Because startup times are becoming a significant part of the total time, we propose two new benchmarks:

(1) MinuteSort: how much can you sort in a minute, and

(2) PennySort: how much can you sort for a penny.

An abridged version of this paper appeared in the ACM SIGMOD'94 Proceedings.
This paper appeared in VLDB Journal 4(4): 603-627 (1995)

This work was sponsored by Digital Equipment Corporation.
Authors' Addresses:
Chris Nyberg, Ordinal Technology Corp., 20 Crestview Dr., Orinda, CA 94563 chris@ordinal.com
Tom Barclay, Microsoft Corp., One Microsoft Way, Redmond, WA 98052 tbarclay@microsoft.com
Zarka Cvetanovic, Digital, 60 Codman Hill Rd, Boxborough, MA 01717 zarka@danger.enet.dec.com
Jim Gray, 310 Filbert St., San Francisco, CA 94133 @crl.com
David Lomet, Microsoft Corp., One Microsoft Way, Redmond, WA 98052 lomet@microsoft.com

1. Introduction

In 1985, an informal group of 25 database experts from a dozen companies and universities defined three basic benchmarks to measure the transaction processing performance of computer systems.

DebitCredit: a market basket of database reads and writes, terminal IO, and transaction commits to measure on-line transaction processing performance (OLTP). This benchmark evolved to become the TPC-A transactions-per-second and dollars-per-transaction-per-second metrics [12].

Scan: copy a thousand 100-byte records from disk-to-disk with transaction protection. This simple mini-batch transaction measures the ability of a file system or database system to pump data through a user application.

Sort: a disk-to-disk sort of one million, 100-byte records. This has become the standard test of batch and utility performance in the database community [3, 4, 6, 7, 9, 11, 13 18, 21, 22]. Sort tests the processor's, IO subsystem's, and operating system's ability to move data.

DebitCredit is a simple interactive transaction. Scan is a mini-batch transaction. Sort is an IO-intensive batch transaction. Together they cover a broad spectrum of basic commercial operations.

2. The sort benchmark and prior work on sort

The Datamation article [1] defined the sort benchmark as:

- Input is a disk-resident file of a million 100-byte records.
- Records have 10-byte key fields and can't be compressed.
- The input record keys are in random order.
- The output file must be a permutation of the input file sorted in key ascending order.

The performance metric is the elapsed time of the following seven steps:

(1) launch the sort program.
(2) open the input file and create the output file.
(3) read the input file.
(4) sort the records in key-ascending order.
(5) write the output file.
(6) close the files.
(7) terminate the program.

The implementation may use all the "mean tricks" typical of operating systems utilities. It can access the files via low-level interfaces, it can use undocumented interfaces, and it can use as many disks, processors and as much memory as it likes. Sort's price-performance metric normalizes variations in software and hardware configuration. The basic idea is to compute the 5-year cost of the hardware and software, and then prorate that cost for the elapsed time of the sort [1, 12]. A one minute sort on a machine with a 5-year cost of a million dollars would cost 38 cents (0.38$).

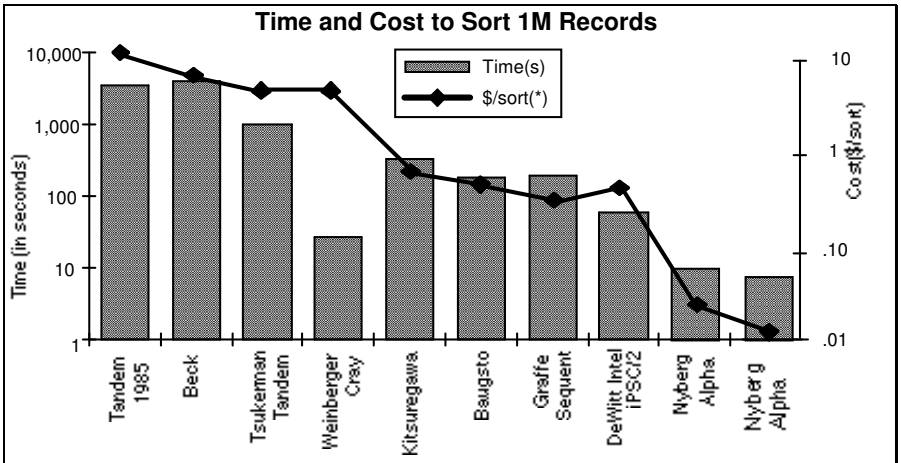
In 1985, as reported by Tsukerman, typical systems needed 15 minutes to perform this sort benchmark [1, 6, 21]. As a super-computer response to Tsukerman's efforts, Peter Weinberger of ATT wrote a program to read a disk file into memory, sort it using replacement-selection as records arrived, and then write the sorted data to a file [22]. This code postulated 8-byte keys, a natural size for the Cray, and made some other simplifications. The disks transferred at 8 MB/s, so you might guess that it took 12.5 seconds to read and 12.5 seconds to write for a grand total of 25 seconds. However there was about 1 second worth of overhead in setup, file creation, and file access. The result, 26 seconds, stood as the unofficial sort speed record for seven years. It is much faster than the subsequently reported Hypercube and hardware sorters.

Table 1: Published sort performance on the Datamation 100 MB benchmark in chronological order. Extrapolations marked by (*). Prices are estimated.

System	Seconds	$/sort(*)	Cost M$*	CPUs	Disks	Reference
Tandem	3600	4.61	2	2	2	[1, 21]
Beck	6000	1.92	.1	4	4	[7]
Tsukerman + Tandem	980	1.25	.2	3	6	[20]
Weinberger + Cray	26	1.25	7.5	1	1	[22]
Kitsuregawa	320*	0.41	.2	1+	1	[15]
Baugsto	180	0.23	.2	16	16	[4]
Graefe + Sequent	83	0.27	.5	8	4	[11]
Baugsto	40	0.26	1	100	100	[4]
DeWitt + Intel iPSC/2	58	0.37	1.0	32	32	[9]
DEC Alpha AXP 7000	9.1	0.022	.4	1	16	1993
DEC Alpha AXP 4000	8.2	0.011	.2	2	14	1993

Since 1986, most sorting effort has focused on multiprocessor sorting, either using shared memory or using partitioned-data designs. DeWitt, Naughton, and Schneider's efforts on an Intel Hypercube was the fastest reported time: 58.3 seconds using 32 processors, 32 disks and 224 MB of memory [9]. Baugsto, Greispland and Kamberbeek mentioned a 40-second sort on a 100-processor 100-disk system [4]. These parallel systems stripe the input and output data across all the disks (30 in the Hypercube case). They read the disks in parallel, performing a preliminary sort of the data at each source, and partition it into equal-sized parts. Each reader-sorter sends the partitions to their respective target partitions. Each target partition processor merges the many input streams into a sorted run that is stored on the local disk. The resulting output file is striped across the 30 disks. The Hypercube sort was two times slower than Weinberger's Cray sort, but it had better price-performance, since the machine is about seven times cheaper.

Table 1 and Graph 2 show that prior to AlphaSort, sophisticated hardware-software combinations were slower than a brute-force one-pass memory intensive sort. Until now, a Cray Y-MP super-computer with a gigabyte of memory, a fast disk, and fast processors was the clear winner. But, the Cray approach was expensive.



Graph 2: The performance and price-performance trends of sorting displayed in chronological order. Until now, the Cray sort was fastest but the parallel sorts had the best price-performance.

Weinberger's Cray-based sort used a fast processor, a fast-parallel-transfer disk, and lots of fast memory. AlphaSort's approach is similar, but it uses commodity products to achieve better price/performance. It uses fast microprocessors, commodity memory, and commodity disks. It uses file striping to exploit parallel disks, and it breaks the sorting task into subtasks to utilize multiprocessors. Using these techniques, AlphaSort beats the Cray Y-MP in two dimensions: it is about 4x faster and about 100x less expensive.

3. Optimizing for the memory hierarchy

Good external sort programs have always tried to minimize the wait for data transfers between disk and main memory. While this optimization is well known, minimizing the wait between processor caches and main memory is not as widely recognized. AlphaSort has the traditional optimizations, but in addition it gets a 4:1 processor-speedup by minimizing cache misses. If all cache misses were eliminated, it could get another 3:1 speedup.

AlphaSort is an instance of the new programming style dictated by modern microprocessor architectures. These processors run the SPEC benchmark very well, because most SPEC benchmarks fit in the cache of newer processors [14]. Unfortunately, commercial workloads, like sort and TPC-A, *do not* conveniently fit in cache [5]. These commercial benchmarks stall the processor waiting for memory most of the time. Reducing cache misses has replaced reducing instructions as the most important processor optimization.

The need for algorithms to consider cache behavior is not a transient phenomenon. Processor speeds are projected to increase about 70% per year for many years to come. This trend will widen the speed gap between memory and processor caches. The caches will get larger, but memory speed will not keep pace with processor speeds.

The Alpha AXP memory hierarchy is:

- Registers,
- On-chip instruction and data caches (I-cache & D-cache),
- Unified (program and data) CPU-board cache (B-cache),
- Main memory,
- Disks,
- Tape and other near-line and off-line storage (not used by AlphaSort).

To appreciate the issue, consider the whimsical analogy in Figure 3. The scale on the left shows the number of clock ticks to get to various levels of the memory hierarchy (measured in 5 ns. processor clock ticks). The scale on the right is a more human scale showing time based in human units (minutes). If your body clock ticks in seconds, then divide the times by 60.

AlphaSort is designed to operate within the processor cache ("This Campus" in Figure 3). It minimizes references to memory ("Sacramento" in Figure 3). It performs disk IO asynchronously and in parallel – AlphaSort rarely waits for disks ("Pluto" in Figure 3).

Suppose AlphaSort paid no attention to the cache and that it randomly accessed main memory at every instruction. Then the processor would run at memory speed – about 2 million instructions per second – rather than the 200 million instructions per second it is capable of, a 100:1 execution penalty. By paying careful attention to cache behavior, AlphaSort is able to minimize cache misses and run at 72 million instructions per second.

This careful attention to cache memory accesses does not mean that we can ignore traditional disk IO and sorting issues. Rather, once the traditional problems are solved, one is faced with achieving speedups by optimizing the use of the memory hierarchy.

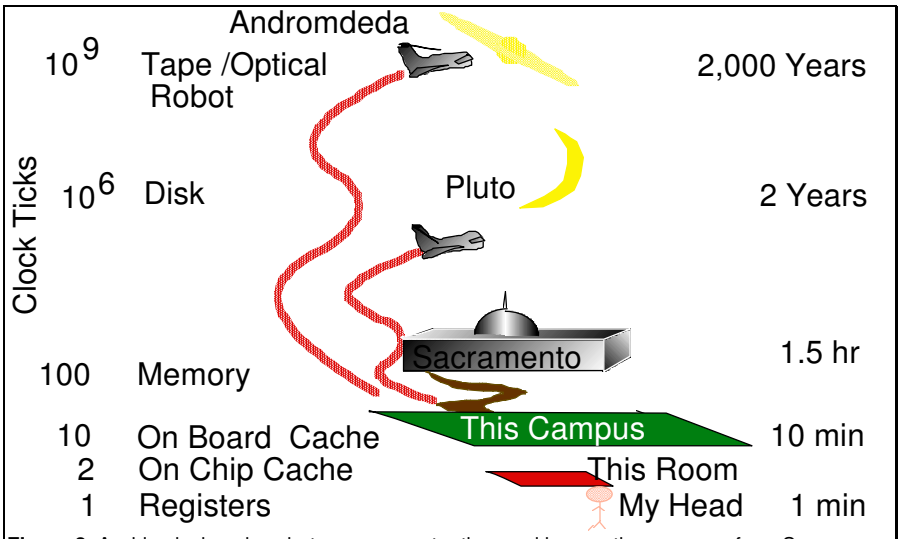


Figure 3: A whimsical analogy between computer time and human time as seen from San Francisco. The scale on the left shows the number of processor cycles to get to various levels of the memory hierarchy (measured in 5 ns. processor clock ticks). The scale on the right is a more human scale showing time based in human units (minutes).

4. Minimizing cache-miss waits

AlphaSort uses the following techniques to optimize its use of the processor cache:

1. QuickSort input record groups as they arrive from disk. QuickSort has good cache locality.
2. Rather than sort records, sort (key-prefix, pointer) pairs. This optimization reduces data movement.
3. The runs generated by QuickSort are merged using a replacement-selection tree. Because the merge tree is small, it has excellent cache behavior. The record pointers emerging from the tree are used to copy records from input buffers to output buffers. Records are only copied this one time. The copy operation is memory intensive.

Comment [JG1]: reference

By comparison, OpenVMS sort uses a pure replacement-selection sort to generate runs [17]. Replacement-selection is best for a memory constrained environment. On average, replacement-selection generates runs twice as large as memory, while the QuickSort runs are typically smaller than half of memory. However, in a memory-rich environment, QuickSort is faster because it is simpler, makes fewer exchanges on average, and has superior address locality to exploit processor caching.

Dividing the records into groups allows QuickSorting to be overlapped with file input. Increasing the number of groups allows for more overlap of input and sorting - QuickSorting cannot commence until the first group is completely read in, and output cannot commence until the last group is QuickSorted. But increasing the number of groups also increases the burden of merging them during the output phase. We observed that using 10-30 groups provided a good balance between these two concerns.

The worst-case behavior of replacement-selection is very close to its average behavior, while the worst-case behavior of QuickSort is terrible (N^2) – a strong argument in favor of replacement-selection. Despite this risk, QuickSort is widely used because, in practice, it has superior performance. Baugsto, Bitton, Beck, Graefe, and DeWitt used QuickSort [4, 6, 7, 9, 11]. On the other hand, Tsukerman and Weinberger used replacement-selection [21, 22]. IBM's DFsort and (apparently) Syncsort™ use replacement selection

in conjunction with a technique called offset-value coding (OVC). We are evaluating OVC[1].

We were reluctant to abandon replacement-selection sort – it has stability and it generates long runs. Our first approach was to improve replacement-selection sort's cache locality. Standard replacement-selection sort has terrible cache behavior unless the tournament fits in cache. The cache thrashes on the bottom levels of the tournament. If you think of the tournament as a tree, each replacement-selection step traverses a path from a pseudo-random leaf of the tree to the root. The upper parts of the tree may be cache resident, but the bulk of the tree is not (see Figure 4).

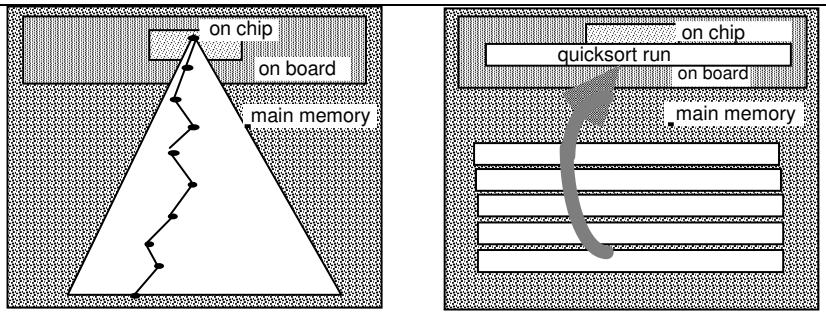


Figure 4. The tournament tree of replacement-selection sort at left has bad cache behavior unless the entire tournament fits in cache. The diagram at left shows the memory references as a winner is removed and a new element is added to the tournament. Each traversal of the tree has many cache misses at the leaves of the tree. By contrast, the QuickSort diagrammed on the right fits entirely in the on-board cache, and partially in the on-chip cache.

We investigated a replacement-selection sort that clusters tournament nodes so that most parent-child node pairs are contained in the same cache line (see Figure 5). This technique reduces cache misses by a factor of two or three. Nevertheless, replacement-selection sort is still less attractive than QuickSort because:

1. The cache behavior demonstrates less locality than QuickSorts. Even when QuickSort runs did not fit entirely in cache, the average compare-exchange time did not increase significantly.

[1] Offset-value coding of sort keys is a generalization of key-prefix-pointer sorting. It lends itself to a tournament sort [2, 8]. For binary data, like the keys of the Datamation benchmark, offset value coding will not beat AlphaSort's simpler key-prefix sort. A distributive sort that partitions the key-pairs into 256 buckets based on the first byte of the key would eliminate 8 of the 20 compares needed for a 100 MB sort. Such a partition sort might beat AlphaSort's simple QuickSort.

2. Tournament sort is more CPU-intensive than QuickSort. Knuth, [17, page 149] calculated a 2:1 ratio for the programs he wrote. We observed a 2.5:1 speed advantage for QuickSort over the best tournament sort we wrote.

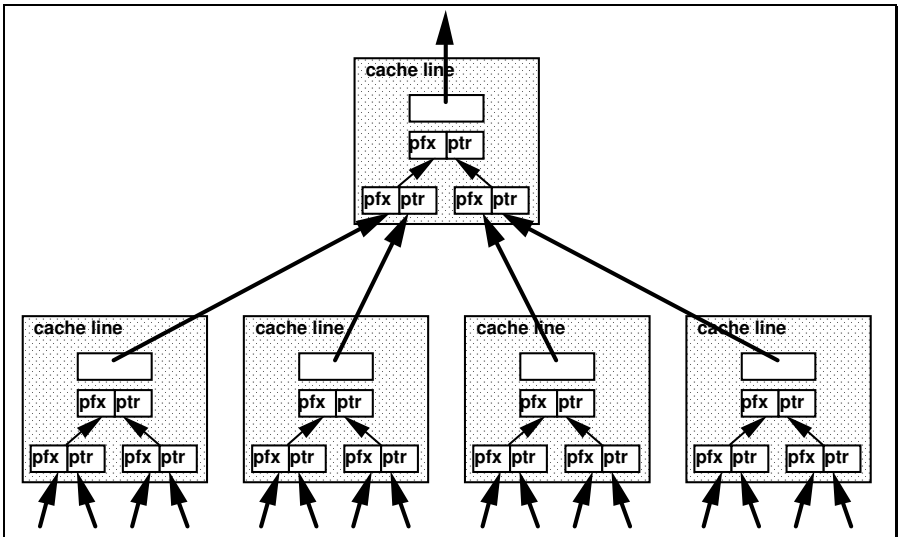
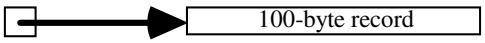


Figure 5: Example of a line-clustered tournament tree. Within each 32-byte cache line are 3 key prefix, record pointer pairs (8 bytes each). The pointers internal to each cache line, while shown in the figure, are implicit. Each line also includes a pointer to the parent prefix/pointer pair.

The key to achieving high execution speeds on fast processors is to minimize the number of references that cannot be serviced by the on-board cache (4MB in the case of the DEC 7000 AXP). As mentioned before, QuickSort's memory access patterns are sequential and so have good cache behavior. But, even within the QuickSort algorithm, there are opportunities to improve cache behavior.

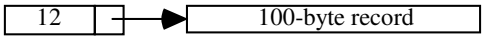
We compared four types of QuickSorts sorting a million 100-byte records in main memory (no disk IO). Each record contained a random key consisting of three 4-byte integers (a slight simplification of the Datamation benchmark). Each of the different QuickSort experiments ended with an output phase where the records are sent, one at a time, in sorted order to an output routine that verifies the correct order and computes a checksum. The four types of QuickSorts were as follows:

Pointer



A million-entry array of 4-byte record pointers is generated and QuickSorted. The records must be referenced during the QuickSort to resolve each key comparison (hence the wide pointer arrow), but only the pointers are moved. The records are sent to the output routine by following the pointers in sorted order.

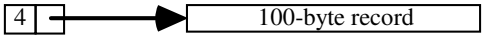
Key/Pointer



A million-entry array of 12-byte keys and record pointers is generated and QuickSorted. This is known as a detached key sort [19]. The pointers are not dereferenced during the QuickSort phase because the keys are included with the pointers - hence the narrow pointer arrow.

Traditionally, detached key sorts have been used for complex keys where the cost of key extraction and conditioning is a significant part of the key comparison cost [21]. Key conditioning extracts the sort key from each record, transforms the result to allow efficient byte or word compares, and stores it with the record as an added field. This is often done for keys involving floating point numbers, signed integers, or character strings with non-standard collating sequences. Comparison operators then do byte-level or word compares on the conditioned strings. Conditioned keys can be stored in the Key/Pointer array.

Key-Prefix/Pointer



A million-entry array of 4-byte key prefixes and record pointers is generated and QuickSorted. The QuickSort loop checks for the case where the key prefixes are equal and, if so, compares the keys in the records. If the key-prefixes are not equal, the keys in the records do not need to be referenced - hence the medium width pointer arrow.

Record

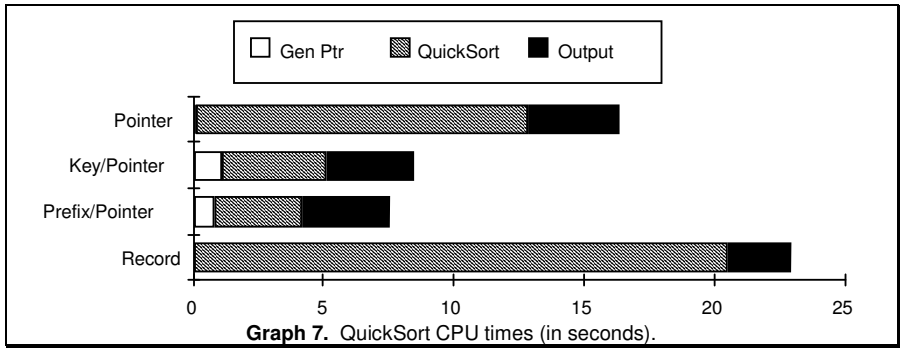
100-byte record

A million-record array of records is QuickSorted in place via exchanges of 100-byte records.

The CPU times for these four QuickSorts on a 150 Mhz Alpha AXP machine (in seconds) are given in Table 6 and Graph 7.

Table 6: CPU times (seconds) for four type of QuickSorts sorting a million records.

	Generate Pointer Array	QuickSort	Output
Pointer	0.08	12.74	3.52
Key/Pointer	1.07	4.02	3.41
Key-Prefix/Pointer	0.84	3.32	3.41
Record	-	20.47	2.49



Graph 7. QuickSort CPU times (in seconds).

The time to generate the Pointer array is so small it barely appears on the graph. The Key/Pointer and Key-Prefix/Pointer arrays take significantly longer to generate because the key values in the record array must be read. This amounts to one cache miss per record to generate the Key-Prefix/Pointer array, and slightly more for the Key/Pointer array since the 100-byte records are not aligned on cache line boundaries.

The QuickSort times for Key/Pointer and Key-Prefix/Pointer are relatively small because the record array is not accessed during this step, whereas the Pointer QuickSort must access the record array to resolve key comparisons. The Record QuickSort must not only reference the 100 MB record array, but modify it as well.

During the output phase, Pointer, Key/Pointer and Key-Prefix/Pointer all take the same approximate time to reference the 100 MB record array in sorted order. Since the records are referenced in a pseudo-random fashion and are on not aligned on cache line boundaries, some of the data brought into the cache is not referenced by the processor. This results in additional cache misses. The Record QuickSort output phase takes less time since the record array is accessed linearly, resulting in the minimum number of cache misses. The linear access also reduces the number of TLB misses.

The optimal QuickSort depends on the record and key lengths, and the key distributions. It also depends on the size of the data relative to the cache size, but typically the data is much larger than the cache.

For long records, Key/Pointer or Key-Prefix/Pointer will be fastest. The risk of using a key-prefix is that, depending on the distribution of key values, it may not be a good discriminator of the key. In that case the comparison must go to the records and Key-Prefix/Pointer sort degenerates to Pointer sort. Baer and Lin made similar observations [2]. They recommended keys be prefix compressed into *codewords* so that the codeword/pointer QuickSort would fit in cache. We did not use codewords because they cannot be used to merge the record pointers.

If the record is short (e.g. less than 16 bytes), record sort has the best cache behavior. It has no setup time, low storage overhead, and leaves the records in sorted order. The last advantage is especially important for small records because the pointer-based sorts must randomly access records to produce the sorted output stream.

To summarize, use record sort for small records. Otherwise, use a key-prefix sort where the prefix is a good discriminator of the keys, and where the pointer and prefix are cache line aligned. Key-prefix sort gives good cache behavior, and for the Datamation benchmark gives more than a 3:1 CPU speedup over record sort.

For AlphaSort, we used a Key-Prefix QuickSort. It had lower memory requirements. It also exploited 64-bit loads and stores to manipulate Key-Prefix/Pointer pairs. This sped up the inner loop of the QuickSort. AlphaSort's time to copy records to output buffer was dominated by cache miss times and could not be reduced.

Once the key-prefix/pointer runs have been QuickSorted, AlphaSort uses a tournament sort to merge the runs. In a one-pass sort there are typically between ten and one hundred runs – the optimal run size balances the time lost waiting for the first run plus time lost QuickSorting the last run, against the time to merge another run during the second phase.

The merge results in a stream of in-order record pointers. The pointers are used to gather (copy) the records into the output buffers. Since the records do not fit in the board cache and are referenced in a pseudo-random fashion, the gathering has terrible cache and TLB behavior. More time is spent gathering the records than is consumed in creating, sorting and merging the key-prefix/pointer pairs. When a full buffer of output data is available, it is written to the output file.

5. Shared-memory multiprocessor optimizations

DEC AXP systems may have up to six processors on a shared memory. When running on a multiprocessor, AlphaSort creates a process to use each processor. The first process is called the *root*, the other processes are called *workers*. The root requests affinity to CPU zero, the *i'th* worker process requests affinity to the *i'th* processor. Affinity minimizes the cache faults and invalidations that occur when a single process migrates among multiple processors.

The root process creates a shared address space, opens the input files, creates the output files and performs all IO operations. The root initiates the worker processes, and coordinates their activities. In its spare time, the root performs sorting chores.

The workers start by requesting processor affinity and attaching to the address space created by the root. With this done, the workers sweep through the address space touching pages. This causes the VMS operating system to allocate physical pages for the shared virtual address space. VMS zeroes the allocated pages for security reasons. Zeroing a 1 GB address space takes 12 CPU seconds – this chore has terrible cache behavior. The workers perform it in parallel while the root opens and reads the input files.

The root process breaks up the sorting work into independent chores that can be handled by the workers. Chores during the QuickSort phase consist of QuickSorting a data run. Workers generate the arrays of key-prefix pointer pairs and QuickSort them. During the merge phase, the root merges all the (key-prefix, pointer) pairs to produce a sorted string of record pointers. Workers perform the memory-intensive chores of gathering records into output buffers using the record pointer string as a guide. The root writes the sorted record streams to disk.

Comment [JG2]: reference

6. Solving the disk bottleneck problem

IO activity for a one-pass sort is purely sequential: sort reads the sequential input file and sequentially creates and writes the output file. The first step in making a fast sort is to use a parallel file system to improve disk read-write bandwidth.

No matter how fast the processor, a 100MB external sort using a single 1993-vintage SCSI disk takes about one minute elapsed time. This one-minute barrier is created by the 3 MB/s sequential transfer rate (bandwidth) of a single commodity disk. We measured both the OpenVMS Sort utility and AlphaSort to take a little under one minute when using one SCSI disk. Both sorts are disk-limited. A faster processor or faster algorithm would not sort much faster because the disk reads at about 4.5 MB/s and writes at about 3.5 MB/s. Thus, it takes about 25 seconds to read the 100 MB, and about 30 seconds to write the 100 MB answer file[2]. Even on mainframes, sort algorithms like Syncsort and DFsort are limited by this one-minute barrier unless disk or file striping is used.

Comment [JG3]: Get the detailed breakdown on AlphaSort and on VMS Sort on one disc.

Disk striping spreads the input and output file across many disks [16]. This allows parallel disk reads and writes to give the sum of the individual disk bandwidths. We investigated both hardware and software approaches to striping.

The Genroco disk array controller allows up to eight disks to be configured as a stripe set. The controller and two fast IPI drives offer a sequential read rate of 15 MB/s (measured). We used three such Genroco controllers each with two fast IPI disk drives in some experiments reported below.

Software file striping spreads the data across commodity SCSI disks that cost about 2000$ each, hold about 2 GB, read at about 5 MB/s, and write at about 3 MB/s. Eight such disks and their controllers are less expensive than a super-computer disk, and are faster. We implemented a file striping system layered above the OpenVMS file system. It allows an application to spread (stripe) a file across an array of disks. A striped file is defined by a stripe definition file, a normal file whose name has the suffix, ".str". For every file in the stripe, the definition file includes a line with the file name and number of

[2] SCSI-II discs support write cache enabled (WCE).that allows the controller to acknowledge a write before the data is on disc. We did not enable WCE because commercial systems demand disk integrity. If WCE were used, 20% fewer discs would be needed.

file blocks per stride for the file. Stripe opens or creates are performed with a call to `StripeOpen()`, which works like a normal open/create except that if the specified file is a stripe definition file then all files in the stripe are opened or created.

The file striping code bandwidth is near-linear as the array grows to nine controllers and thirty-six disks. Bottlenecks appear when a controller saturates; but with enough controllers, the bus, memory, and OS handle the IO load.

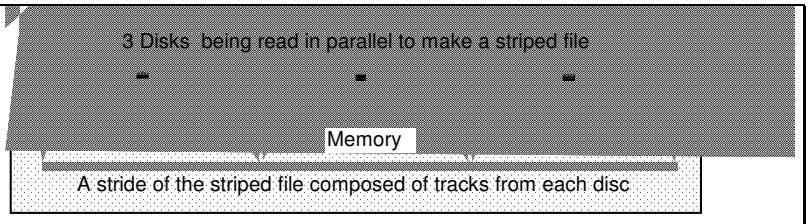


Figure 8. A stride of a striped file being read from three disks. Each disk contributes a track of information to the stride. The reads proceed in parallel so that one can read at the sum of the speeds of the individual disks.

Soft SCSI arrays are less expensive than a special disk array, and they have more bandwidth than a single controller or port. File striping is more flexible than disk striping since the stripe width (number of disks) can be chosen on a file-by-file basis rather than dedicating a set of disks to a fixed stripe set at system generation time. Even with hardware disk arrays, one must stripe across arrays to get bandwidths beyond the limit of a single array. So, software striping must be part of any solution.

Table 9. Two different disk arrays used in the benchmarks.

	many - slow RAID	**few - fast RAID**
drives	36 RZ26	12 RZ28 + 6 Velocitor
controllers	9 SCSI (kzmsa)	4 SCSI + 3 IPI-Genroco
capacity	36 GB	36 GB
disk speed (measured)	1.8 MB/s	SCSI: 4MB/s IPI: 7 MB/s
stripe read rate	64 MB/s	52 MB/s
stripe write rate	49 MB/s	39 MB/s
list price (1993) includes cabinets	85 k$	122 k$

Table 9 compares two arrays: (1) a large array of inexpensive disks and controllers, and (2) a smaller array of high-performance disks and controllers. The many-slow array has slightly better performance and price performance for the same storage capacity.

It might appear that striping has considerable overhead since opening, creating, or closing a single logical file translates into opening, creating or closing many stripe files. A *N*-wide striping does introduce overhead and delays. `StripeOpen()` needs to call the operating system once to open the descriptor, and then *N* times to open the *N* file stripes. Fortunately, asynchronous operations allow the *N* steps to proceed in parallel, so there is little increase in elapsed time. With 8-wide striping the fixed overhead for AlphaSort on an 200 Mhz processor is:

Comment [JG4]: check this against Zarka's results

Load Sort and process parameters	.11
Open stripe descriptor and eight input stripes	.02
Create and open descriptor and eight output stripes	.01
Close 18 input and output files and descriptors	.01
Return to shell	.05
Total Overhead	**.19 seconds**

This is a relatively small overhead.

To summarize, AlphaSort overcomes the IO bottleneck problem by striping data across many disks to get sufficient IO bandwidth. Asynchronous (NoWait) operations open the input files and create the output files in parallel. Triple buffering the reads and writes keeps the disks transferring at their spiral read and write rates. Striping eight ways provided a read bandwidth of 27 MB/s and a write bandwidth of about 22 MB/s. This put an 8-second limit on our sort speed. Later experiments extended this to 36-way striping and 64 MB/s bandwidth.

A key IO question is when to use a one-pass or two-pass sort. When should the QuickSorted intermediate runs be stored on disk? A two-pass sort uses less memory, but uses twice the disk bandwidth since intermediate runs must be written out to scratch disks during the input phase, and read back during the output phase.

Even for surprisingly large sorts it is economic to perform the sort in one pass. The question becomes: What is the relative price of those scratch disks and their controllers versus the price of the memory needed to allow a one-pass sort? Using 1993 prices for

Alpha AXP, a disk and its controller costs about 2400$ (see Table 9). Striping requires 16 such scratch disks dedicated for the entire sort, for a total price of 36k$. A one-pass main memory sort uses a hundred megabytes of RAM. At 100$/MB this is 10k$. It is 360% more expensive to buy the disks for a two-pass sort than to buy 100MB of memory for the one-pass sort. The computation for a 1 GB sort suggests that it would be 15% less expensive to buy 36 extra disks, than to buy the 1 GB of memory needed to do the 1 GB sort (see Table 9).

Multi-gigabyte sorts should be done as two-pass sorts, but for things much smaller than that, one-pass sorts are more economical. In particular, the Datamation sort benchmark should be done in one pass.

7. AlphaSort measurements on several platforms

With these ideas in place, let's walk through the 9.11 second AlphaSort of a million hundred-byte records on a uniprocessor. The input and output files are striped across sixteen disk drives.

AlphaSort first opens and reads the descriptor file for the input stripe set. Each of the 16 input stripe files is opened asynchronously with a 64 KB stride size. The open call returns a descriptor indicating a 100MB input file. Asynchronously, AlphaSort requests OpenVMS to create the 100MB striped output file, and to extend the process address space by 110 MB. AlphaSort immediately begins reading the 100MB input file into memory using triple buffering.

It is now 140 milliseconds into the sort. As each stride-read completes, AlphaSort issues the next read. AlphaSort is completely IO limited in this phase.

When the first 1 MB stripe of records arrives in memory, AlphaSort extracts the 8-byte (record address, key-prefix) pairs from each record. These pairs are streamed into an array. When the array grows to 100,000 records, AlphaSort QuickSorts it. This QuickSort is entirely cache resident. When it completes the processor waits for the next array to be built so that it too can be QuickSorted. The pipeline of steps (read-disk array-build then QuickSort) is disk bound at a data rate of about 27 MB/s.

The read of the input file completes at the end of 3.87 seconds. AlphaSort must then sort the last 100,000 record partition (about .12 seconds). During this brief interval, there is no IO activity.

Now AlphaSort has ten sorted runs produced by the ten QuickSort steps. It is now 4 seconds into the sort and can start writing the output to the striped output file. Meanwhile, it issues `Close()` on all stripes of the input file.

AlphaSort runs a tournament scanning the ten QuickSorted runs of the (key-prefix,pointer) pairs in sequential order, picking the minimum key-prefix among the runs. If there is a tie, it examines the full keys in the records. The winning record is copied to the output buffer. When a full stripe of output buffer is produced, `StripeWrite()` is called to write the sorted records to the target stripe file in disk. This merge-gather runs more slowly than the QuickSort step because many cache misses are incurred in gathering the records into the output buffers. It takes almost four seconds of processor and memory time (the use of multiprocessors speeds this merge step). This phase is also disk limited, taking 4.9 seconds.

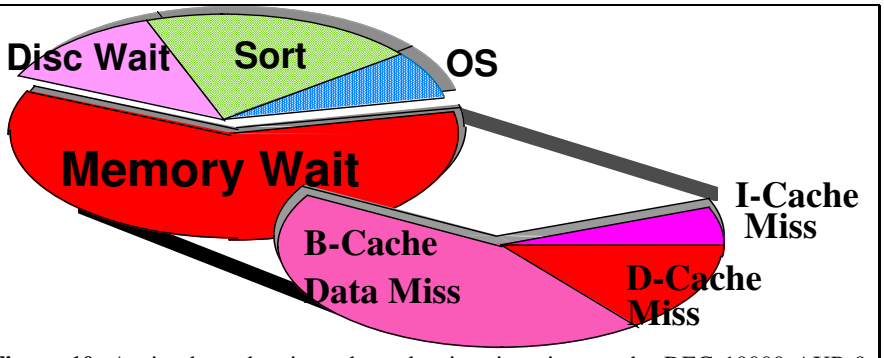


Figure 10. A pie chart showing where the time is going on the DEC 10000 AXP 9-second sort. Even though AlphaSort spends GREAT effort on efficient use of cache, the processor spends most of its time waiting for memory. The vast majority of such waits are for data, and the majority of the time is spent waiting for main memory. The low cost of VMS to launch the sort program, open the files, and move 200 MB through the IO subsystem is impressive. Not shown is the 4% of stalls due to branch mispredictions.

When the tournament completes 8.8 seconds have elapsed. AlphaSort is ready to close the output files and to return to the shell. Closing takes about 50 milliseconds. AlphaSort

then terminates for a total time of 9.1 seconds. Of this 0.3 seconds were consumed loading the program and returning to the command interpreter. The sort time was 8.8 seconds, but the benchmark definition requires that the startup and shutdown time be included

Some interesting statistics about this sort are (see Figure 10):

- The CPU time is 7.9 seconds, 1.1 seconds is pure disk wait. Most of the disk wait is in startup and shutdown.
- 6.0 seconds of the CPU time is in the memory-to-memory sort.
- 1.9 seconds are used by OpenVMS AXP to:
  load the sort program,
  allocate and initialize a 100MB address space,
  open 17 files,
  create and open 17 output files and allocate 100MB of disk on 16 drives,
  close all these files, and
  return to command interpreter and print a time stamp.
- Of the 7.9 seconds of CPU time, the processor is issuing instructions 29% of the time. Most of the rest of the time it is waiting for a cache miss to be serviced from main memory (56%). SPEC benchmarks have much better cache hit ratios because the program and data fit in cache. Database systems executing the TPC-A benchmark have worse cache behavior because they have larger programs and so have more I-cache misses.
- The instruction mix is: Integer (51%), Load (15%), Branch (15%), Store (12%), Float (0%), and PAL (9%) mostly handling address translation buffer (DTB) misses. 8.4% of the processor time is spent dual issuing.
- The processor chip hardware monitor indicates that 29% of the clocks execute instructions, 4% of the stall time is due to branch mis-predictions, 11% is I-stream misses (4% I-to-B and 7% B-to-main), and 56% are D-stream misses (12% D-to-B and 44% B-to-main).
- The time spent dual-issuing is 8%, compared to 21% spent on single-issues. Over 40% of instructions are dual issued.

AlphaSort benchmarks on several AXP processors are summarized in Table 11. All of these benchmarks set new performance and price/performance records. The AXP-3000 is the price-performance leader. The DEC AXP 7000 is the performance leader. As

spectacular as they are, these numbers are improving. Software is making major performance strides as it adapts to the Alpha AXP architecture. Hardware prices are dropping rapidly.

Table 11. Performance and price/performance of 100MB Datamation sort benchmarks on Alpha AXP systems (October 1993). The disk price column includes disk and controller prices.

System	CPU&clock	cntrllrs	disks	(MB)	time(s)	total$	disk$	$/sort
7000	3x5ns	7 fast-SCSI	28 RZ26	256	7.0	312k$	123k$	0.014$
4000	2x6.25ns	4 SCSI, 3 IPI	12scsi+6ipi	256	8.2	312k$	95k$	0.016$
7000	1x5ns	6 fast-SCSI	16 RZ74	256	9.1	247k$	65k$	0.014$
4000	1x6.25ns	4 fast-SCSI	12 RZ26	384	11.3	166k$	48k$	0.014$
3000	1x6.6ns	5 SCSI	10 RZ26	256	13.7	97k$	48k$	0.009$

To summarize, AlphaSort optimizes IO by using host-based file striping to exploit fast but inexpensive disks and disk controllers – no expensive RAID controllers are needed. It uses lots of RAM memory to achieve a one-pass sort. It improves the cache hit ratio by QuickSorting (key-prefix, pointer) pairs if the records are large. If multiprocessors are available, AlphaSort breaks the QuickSort and Merge jobs into smaller chores that are executed by worker processors while the root process performs all IO.

8. Ideas for reducing cache misses in general programs

In spite of our success in reducing cache misses in an external sort, we do not advocate attempting similar optimizations for all programs. In the general case the program's instructions and data may effectively fit in the cache, and there is no significant improvement to be made. For instance, most of the SPEC benchmark programs fit in the board cache.

A program that doesn't fit in the cache will suffer from either instruction or data cache misses. The most promising technique for reducing instruction cache misses is to have the compiler cluster the frequently used basic blocks together based on run-time statistics. With AlphaSort, instruction cache misses were not a problem due to the small main loop.

Reducing data cache misses can be an intractable problem if the data references are made by a very large number of instructions. For instance, code to execute the TPC-A benchmarks is usually characterized by a very large number of basic blocks that do not loop. In this environment it is very difficult to understand the data access patterns, let alone modify them to reduce cache misses.

In contrast, sorting belongs to a class of programs that make a very large number of data accesses from a small amount of looping code. In this environment it is feasible to control data accesses via algorithm or data structure modifications.

We have already discussed the clustering of tournament tree nodes together in a cache line. Another data structure for reducing cache misses we call a *line-list*. This is a variation of a link-list where each structure has the size of a cache line (or multiple cache lines) and is aligned on a cache line boundary. It can make sense to use this data structure in environments where items are always added to the ends of lists, and removed from lists in a sequential fashion. An example of line-list is given in Figure 12.

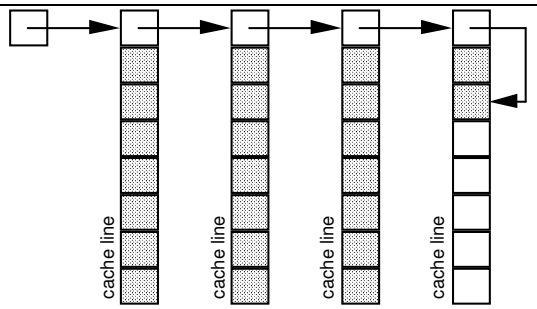


Figure 12: Example of a line-list. Each line structure contains a pointer and up to 7 elements. The last line structure contains only 2 elements. This is indicated by having its pointer point to the last element in the line, rather than a subsequent line structure.

If the elements being stored are 4 bytes in size and the cache line size is 32 bytes, the size of each line-list structure would be 32 bytes and could hold 7 4-byte elements. The last line-list structure in a list might hold fewer than the maximum number of elements. This could be indicated by having its pointer point to the last element in the line (this also indicates the end of the line-list). There are potential fragmentation problems with line-lists. Memory space could be wasted if most line structures contain only a few elements. However this situation would not occur if the number of line-lists is small compared to the number of line structures. Line-lists could, in fact, improve memory utilization by reducing the number of pointers per element. It only makes sense to use a line-list if the data being accessed is much bigger than the cache. Line-lists require slightly more instructions than the traditional, single-item link-lists. If they save cache misses, this is a good tradeoff.

Line-lists in memory are somewhat analogous to files on disks (although files also have a hierarchical structure). Both are designed to reduce accesses to a particular level of the memory hierarchy by reading blocks at time. Similarly, the idea of clustering tournament nodes in cache lines is similar to B-trees (although the former is designed to be static in structure while the latter is dynamic). There may be other main memory data structures that can be adapted from disk data structures to reduce cache misses. However not every technique pertinent to disk IO or virtual memory is necessarily pertinent to reducing data cache misses. Cache misses are a much smaller penalty than a disk IO. Hence one needs to be much more cognizant of the number of additional instructions required to reduce data cache misses. Disk IO penalties are so high that this tradeoff is rarely considered.

9. New sort metrics: MinuteSort and PennySort

The original Datamation benchmark has outlived its usefulness. When it was defined, 100MB sorts were taking ten minutes to one hour. More recently, workers have been reporting times near one minute. Now the mark is seven seconds. The next significant step is 1 second. This will give undue weight to startup times. Already, startup and shutdown is over 25% of the cost of the 7-second sort. So, the Datamation Sort benchmark is a startup/shutdown benchmark rather than an IO benchmark.

To maintain its role as an IO benchmark, the sort benchmark needs redefinition. We propose the following:

MinuteSort:

- Sort as much as you can in one minute.
- The input file is resident on external storage (disk).
- The input consists of 100-byte records (incompressible).
- The first ten bytes of each record is a random key.
- The output file is a sorted permutation of the input.
- The input and output files must be readable by a program using conventional tools (a database or a record manager.)

The elapsed time includes the time from calling the sort program to the time that the program returns to the caller – this total time must be less than a minute. If Sort is an operating system utility, then it can be launched from the command shell. If Sort is part of a database system, then it can be launched from the interactive interface of the DBMS.
MinuteSort has two metrics:

Size (bytes): the number of gigabytes you can sort in a minute of elapsed time

Price-performance ($/sorted GB): To get a price-performance metric, the price is divided by the sort size (in gigabytes). The price of a minute is the list price of the benchmark hardware and operating system divided by one million.

This metric includes an *Nlog(N)* term (the number of comparisons) but in the range of interest range ($N > 2^{30}$), *log(N)* grows slowly compared to *N*. As *N* increases by a factor of 1,000, *log(N)* increases by a factor of 1.33.

A three-processor DEC 7000 AXP sorted 1.08 GB in a minute. The 1993 price of this system (36 disks, 1.25 GB of memory, 3 processors, and cabinets) is 512k$. So the 1.1 GB MinuteSort would cost 51 cents (=512k$/1M). The MinuteSort price-performance metric is the cost over the size (.51/1.1) = 0.47$/GB. So, today AlphaSort on a DEC 7000 AXP has a 1.1 GB size and a 0.47$/GB price/performance.

MinuteSort uses a rough 3-year price and omits the price of high-level software because: (1) this is a test of the machine's IO subsystem, and (2) most of the winners will be "special" programs that are written just to win this benchmark – most university software is not for sale (see Table 1). There are 1.58 million minutes in 3 years, so dividing the price by 1M gives a slight (30%) inflator for software and maintenance. Depreciating over 3 years, rather than the 5-year span adopted by the TPC, reflects the new pace of the computer business.

Minute sort is aimed at super-computers. It emphasizes speed rather than price performance – it reports price as an afterthought. This suggests a dual benchmark that is fixed-price rather than fixed-time: PennySort. PennySort is just like MinuteSort except that it is limited to using one penny's worth of computing. Recall that each minute of computer time costs about one millionth of the system list price. So PennySort would allow a million dollar system to sort for 1/100 minute, while a 10,000$ system could sort for one minute. PCs could win the PennySort benchmark.

Penny Sort:

- Sort as much as you can for less than a penny.
- Otherwise, it has the same rules as MinuteSort

PennySort reports two metrics:

Size (bytes): the number of gigabytes you can sort for a penny.

Elapsed Time: The elapsed time of the sort (reported in to the nearest millisecond).

Given the cost formula, PennySort imposes the following time limit in minutes :

$$\frac{10^4}{\textit{system list price (\$)}}$$

MinuteSort and PennySort are an interesting contrast to the Datamation sort benchmark. Datamation sort was fixed size (100MB) and so did not scale with technology.

MinuteSort and PennySort scale with technology because they hold end-user variables constant (time or price) and allow the problem size to vary.

Industrial-strength sorts will always be slower than programs designed to win the benchmarks. There is a big difference between a program like AlphaSort, designed to sort exactly the Datamation test data, and an industrial-strength sort that can deal with many data types, with complex sort keys, and with many sorting options. AlphaSort slowed down as it was productized in Rdb and in OSF/1 HyperSort.

This suggests that there be an additional distinction, a *street-legal* sort that restricts entrants to sorts sold and supported by someone. Much as there is an Indianapolis Formula-1 car race run by specially built cars, and a Daytona stock-car race run by production cars, we propose that there be an *Indy* category and a *Daytona* category for both MinuteSort and PennySort. This gives four benchmarks in all:
Indy-MinuteSort: a Formula-1 sort where price is no object.
Daytona-MinuteSort: a stock sort where price is no object.
Indy-PennySort: a Formula-1 biggest-bang-for-the buck sort.
Daytona-PennySort: a stock sort giving the biggest-bang-for-the buck .
Super-computers will probably win the MinuteSort and workstations will win the PennySort trophies.

The past winners of the Datamation sort benchmark (Barclay, Baugsto, Cvetanovic, DeWitt, Gray, Naughton, Nyberg, Schneider, Tsukerman, and Weinberger) have formed a committee to oversee the recognition of new sort benchmark results. At each annual SIGMOD conference starting in 1994, the committee will grant trophies to the best MinuteSorts and PennySorts in the Daytona and Indy categories (4 trophies in all). You can enter the contest or poll its status by contacting one of the committee members.

10. Summary and conclusions

AlphaSort is a new algorithm that exploits the cache and IO architectures of commodity processors and disks. It runs the standard sort benchmark in seven seconds. That is four times better than the unpublished record on a Cray Y-MP, and eight times faster than the 32-CPU 32-disk Hypercube record [9, 23]. It can sort 1.1 GB in a minute using multiprocessors. This demonstrates that commodity microprocessors can perform batch transaction processing tasks. It also demonstrates speedup using multiple processors on a shared memory.

The Alpha AXP processor can sort VERY fast. But, the sort benchmark requires reading 100MB from disk and writing 100MB to disk – it is an IO benchmark. The reason for including the Sort benchmark in the Datamation test suite was to measure "how fast the real IO architecture is" [1].

By combining many fast-inexpensive SCSI disks, the Alpha AXP system can read and write disk data at 64 MB/s. AlphaSort implements simple host-based file striping to achieve this bandwidth. With it, one can balance the processor, cache, and IO speed. The result is a breakthrough in both performance and price-performance.

In part, AlphaSort's speed comes from efficient compares, but most of the CPU speedup comes from efficient use of CPU cache. The elapsed-time speedup comes from parallel IO performed by an application-level striped file system.

Our laboratory's focus is on parallel database systems. AlphaSort is part of our work on loading, indexing, and searching terabyte databases. At a gigabyte-per-minute, it takes more than 16 hours to sort a terabyte. We intend to use many processors and many-many disks handle such operations in minutes rather than hours. A terabyte-per-minute parallel sort is our long-term goal (not a misprint!). That will need hundreds of fast processors, gigabytes of memory, thousands of disks, and a 20 GB/s interconnect. Thus, this goal is five or ten years off.

11. Acknowledgments

Al Avery encouraged us and helped us get access to equipment. Doug Hoeger gave us advice on OpenVMS sort. Ken Bates provided the source code of a file striping prototype he did five years ago. Dave Eiche, Ben Thomas, Rod Widdowson, and Drew Mason gave us good advice on the OpenVMS AXP IO system. Bill Noyce and Dick Sites gave us advice on AXP code sequences. Bruce Fillgate, Richie Lary, and Fred Vasconcellos gave us advice and help on disks and loaned us some RZ74 disks to do the tests. Steve Holmes and Paline Nist gave us access to systems in their labs and helped us borrow hardware. Gary Lidington and Scott Tincher helped get the excellent DEC 4000 AXP results. Joe Nordman of Genroco provided us with fast IPI disks and controllers for the DEC 4000 AXP tests. The referees for both versions of this paper gave us many valuable suggestions and comments.

12. References

[1] Anon-Et-Al. (1985). "A Measure of Transaction Processing Power." *Datamation*. V.**31**(7): PP. 112-118. also in *Readings in Database Systems,* M.J. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.

[2] Baer, J.L., Lin, Y.B., "Improving Quicksort Performance with Codeword Data Structure", IEEE Trans. on Software Engineering, **15**(5). May 1989. pp. 622-631.

[3] Baugsto, B.A.W., Greipsland, J.F., "Parallel Sorting Methods for Large Data Volumes on a Hypercube Database Computer", Proc. 6th Int. Workshop on Database Machines, Deauville France, Springer Verlag Lecture Notes No. 368, June 1989, pp.: 126-141.

[4] Baugsto, B.A.W., Greipsland, J.F., Kamerbeek, J. "Sorting Large Data Files on POMA," Proc. CONPAR-90VAPP IV, Springer Verlag Lecture Notes No. 357, Sept. 1990, pp.: 536-547.

[5] Cvetanovic, Z. , D. Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads", Proc. Int.Symposium on Computer Architecture, April 1994.

[6] Bitton, D., *Design, Analysis and Implementation of Parallel External Sorting Algorithms*, Ph.D. Thesis, U. Wisconsin, Madison, WI, 1981

[7] Beck, M., Bitton, D., Wilkenson, W.K., "Sorting Large Files on a Backend Multiprocessor", IEEE Transactions on Computers, V. **37**(7), pp. 769-778, July 1988.

[8] Conner, W.M., Offset Value Coding, IBM Technical Disclosure Bulletin, V **20**(7), Dec. 1977, pp. 2832-2837

[9] DeWitt, D.J., Naughton, J.F., Schneider, D.A. "Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting", Proc. First Int Conf. on Parallel and Distributed Info Systems, IEEE Press, Jan 1992, pp. 280-291

[10] Filgate, Bruce, "SCSI 3.5" 1.05 GB Disk Comparative Performance", Digital Storage Labs, Nov. 10 1992

[11] Graefe, G., "Parallel external sorting in Volcano," U. Colorado Comp. Sci. Tech. Report 459, June 1990.

[12] Graefe, G, S.S. Thakkar, "Tuning a Parallel Sort Algorithm on a Shared-Memory Multiprocessor", Software Practice and Experience, **22**(7), July 1992, pp. 495.

[13] Gray, J. (ed.), *The Benchmark Handbook for Database and Transaction Processing Systems,* Morgan Kaufmann, San Mateo, 1991.

[14] Kaivalya, D., The SPEC Benchmark Suite, Chapter 6 of *The Benchmark Handbook for Database and Transaction Processing Systems, Second Edition, Chapter 6*, Morgan Kaufmann, San Mateo, 1993.

[15] Kitsuregawa, M., Yang, W., Fushimi, S. "Evaluation of an 18-stage Pipeline Hardware Sorter", Proc. 6th Int. Workshop on Database Machines, Deauville France, Springer Verlag Lecture Notes No. 368, June 1989, pp. 142-155.

[16] Kim. M.Y., "Synchronized Disk Interleaving," IEEE TOCS, V. **35**(11), Nov. 1986, pp. 978-988.

[17] Knuth, D.E., *Sorting and Searching, The Art of Computer Programming,* Addison Wesley, Reading, Ma., 1973.

[18] Lorie, R.A., and Young, H. C., "A Low Communications Sort Algorithm for a Parallel Database Machine," Proc. Fifteenth VLDB, Amsterdam, 1989, pp. 125-134.

[19] Lorin, H. Sorting, Addison Wesley, Englewood Cliffs, NJ, 1974.

[20] Salzberg, B., et al., "FastSort– An External Sort Using Parallel Processing", Proc. SIGMOD 1990, pp. 88-101.

[21] Tsukerman, A., "FastSort– An External Sort Using Parallel Processing" Tandem Systems Review, V **3**(4), Dec. 1986, pp. 57-72.

[22] Weinberger, P.J., Private communication 1986.

[23] Yamane, Y., Take, R. "Parallel Partition Sort for Database Machines", *Database Machines and Knowledge Based Machines*, Kitsuregawa and Tanaka eds., pp.: 1117-130. Klwar Academic Publishers, 1988.