DMSort: A PennySort and Performance/Price Sort

Aaron Darling* Alex Mohr *

University of Wisconsin, Madison

Abstract

This work describes our approach to creating a fast and low-cost sorting system. The goal of this work is to win the 2002 PennySort and Performance/Price sort. We have designed a sorting program called DMSort that is capable of more than double the performance of previously published results when run on our system configuration. This paper discusses the DMSort system alongside a discussion of topics relevant to PennySort and Performance/Price sort. In particular, the DMSort sorting algorithm, hardware system, system implementation, and performance characteristics are discussed in detail.

1 Introduction

Sorting data is one of the fundamental problems of Computer Science. Much research was done on the theoretical aspects of sorting algorithms during the 1950s and 60s [11, 8], and the general sorting problem is considered solved. Even so, external disk-to-disk sorting competitions (where the data set is typically much larger than physical memory) have become popular in more recent years. These competitions do not often present theoretical results to the field of sorting. Rather, they act as benchmarks and provide quantitative measures of hardware and software system performance. External sorts are considered well-rounded system benchmarks because they can tax the disk I/O subsystem, the CPU, the memory and bus performance, and the software subsystem implementations concurrently. This way, external sort benchmarks can be used to expose performance deficiencies in these system components (or their interactions), and to track the relative performance improvements of different subsystems over time.

There are many different external sort benchmarks of this type, characterized by different sets of rules. The earlier external sort competitions simply measured how fast a fixed dataset could be sorted, or how much data could be sorted in a fixed timeframe. These competitions led to large, expensive, and elaborate systems that were often only practical for sorting large amounts of data and similar applications.

Because of this impracticality, more recent external sort benchmarks have added system cost into the performance metric to obtain high relative performance/price ratios. Since the largest market and much of the competition for computers are found in the consumer segment, prices tend to be minimized. Thus many of the competitors and all of the current winners of these competitions use consumer-grade machines.

The PennySort competition that DMSort competes in was introduced in 1998. Unlike many other sort benchmark competitions, this competition takes the sorting system's cost into account. The PennySort measures how much data a system can sort for "one penny's worth" of compute time. To determine a penny's worth of computing time, the system's value is assumed to depreciate at a constant rate over three years. Therefore we take the number of seconds in three years and divide by the system cost in pennies to determine a running time for one penny.

To do PennySort well, the whole computer system including the hardware, the system software, and the sorting application software must function well together. Since we can choose any possible combination of sys-

* {darling,amohr}@cs.wisc.edu

tem components, operating systems, file systems, and sorting algorithms, the number of parameters is very large and they are often interdependent. This makes selecting a set of components that performs well difficult.

This paper presents our sorting system, DMSort. We begin with a review of related work, followed by a discussion of previous relevant PennySort competitors. Next we examine DMSort's sorting algorithm from a high level, and compare it to some other PennySort competitors' algorithms to provide context for a discussion of hardware system design issues. We then present our selected system components, followed by a detailed description of the sorting software on our custom hardware configuration. Finally, we analyze the performance of our system and compare it to previous PennySort entrants.

2 Related Work

The original sorting benchmark came from the famous paper, "A Measure of Transaction Processing Power" [3]. This benchmark became known as Datamation. The task for Datamation is to sort 100,000,000 bytes (one million 100-byte records) as fast as possible, and results are scored by the amount of time required to perform this task. This tests the operating system's file system performance, disk performance, and processor speed. This has inevitably led to larger, more expesive, and more specialized systems. In fact the winner for the year 2000 used a dedicated hardware-based sorting device [2].

The first Datamation sort result we found was one hour [5] while the record at the time of this writing is under a half second [10]. This is certainly an excellent improvement; however, as the sort time becomes smaller, the machines become more complex, and 100,000,000 bytes becomes a small dataset. Therefore, Datamation has become less useful as a measure of general system performance.

Because Datamation sort times became so small, MinuteSort—a measure of how much data can be sorted in a minute, was introduced in 1994 [9]. MinuteSort fixes the amount of time allowed for sorting, rather than fixing the amount of data as in Datamation. While MinuteSort allows datasets to scale arbitrarily, there is no penalty for high system cost, encouraging extravagant machines [5] as in Datamation.

These benchmarks are useful for measuring very large and expensive systems but are not particularly useful for those people who must consider system cost. We believe that this is the large majority.

The first proposed sort benchmark accounting for price, and a precursor to PennySort was the DollarSort, which measures the number of 100-byte records that can be sorted for a dollar [4]. "One dollar's worth" of computing time is computed using linear depreciation in the same manner as PennySort. The DollarSort benchmark was quickly revised to the PennySort benchmark when it was noticed that a $2,000 system would run for nearly 50,000 seconds, or just over 13 hours [4].

2.1 Penny Sort

As mentioned earlier, the PennySort simply measures the number of 100-byte records that can be sorted for one penny assuming linear depreciation. A $1,000 system, for example, would be allowed to sort for 946 seconds, or almost 16 minutes. Benchmarks like PennySort that take price into account create an interesting tradeoff: one can use a very cheap system and spend a long time sorting, or one can use a very expensive system and spend a short time sorting. All the past winners in this category have used a reasonably priced consumer-grade system [5, 7].

Most recently, there has been impetus to migrate from PennySort to a more general Performance/Price sort [4]. Performance/Price sort is similar to PennySort, but it fixes the sort time at one minute, and scores a system by computing its sorted GB per dollar. This revision was proposed to allay fears of PennySort becoming a "shopping competition," and also to reduce the run cycle time. However, recent trends in system cost have had an interesting effect in conjunction with this competition's fixed time limit. This is addressed in Section 4.

PennySort has two divisions: Indy and Daytona. The Indy division is open to any sorting hardware and software. The Daytona division is only open to commercially available sorting software. DMSort is a research sorting system that is not supported so we compete in the Indy division.

The winners of the first PennySort competition in 1998

were PostmanSort in the Daytona category and NTSort in the Indy category. PostmanSort sorted 1.27GB using a MSB radix sort algorithm on a Windows NT machine with two disks. NTSort is the command line sort program provided with Windows 2000, and it sorted 1.45GB using a quicksort and merge technique [4].

Since 1999, PennySort has been dominated by Stenograph LLC's product, HMsort. HMsort sorted 2.58GB in 1999 and 4.2GB in 2000 and 2001 on a $1010 Windows/Intel system. HMsort is a commercially available sort for Windows. Its approach to sorting is similar to NTSort's in that it creates several small sorted files on the first pass and merges them into a single file on the second. Unlike NTsort, HMsort effectively overlaps computation with I/O by employing a multithreaded design. In their 1999 results they cite disk seek overhead as the primary bottleneck in their system [6].

The goal of DMSort is to win the 2002 Indy-class PennySort. The trend in PennySort scores during the past several years suggests that we need to at least double the previous winner's score to be successful. Since the previous winners sorted 4.2GB, our aim is to sort 8.4GB for one penny.

3 The DMSort Algorithm

For any sorting problem, a good choice of sorting algorithm is critical for high performance. In particular, since we have many hundreds of seconds to run and our goal is to sort over 80 million records, our algorithm choice must scale well with large data sizes. We chose to base our sorting algorithm on most-significant-byte radix sort. We chose radix sort instead of a pairwise-comparison sort like quicksort because radix sort has asymptotic complexity $O(n)$ while pairwise-comparison sorts are $O(n \log n)$.

Sorting a large amount of data which doesn't fit in memory cannot be implemented as a linear scan over the data—there must be at least two passes. Therefore DMSort divides the sorting into two phases. The first phase does a "rough" sort where the sort data is divided by key value into a fixed set of "bins". These bins are recorded on disk for the second phase. The second phase processes the bin files generated in phase one, reading, sorting, and writing each one sequentially to the output file. The process is illustrated in Figure 1 and described below.

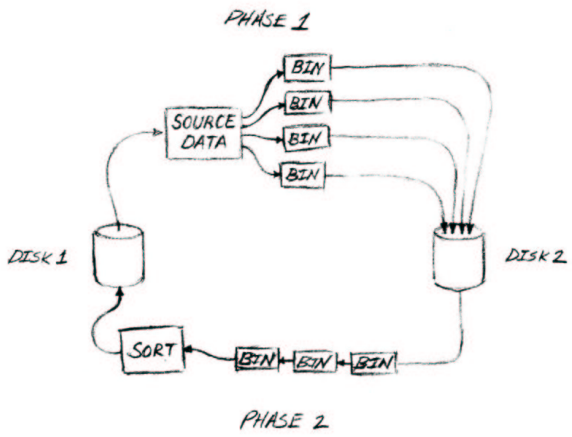

Figure 1: An overview of the DMSort algorithm. Sorting is divided into two phases. The first phase reads data from disk 1 on the left, collects it into bins based on its key values, and writes these bin files to disk. The second phase reads the bin files produced in phase one, sorts them, and writes them in order to the output file back on disk 1.

In the first phase, the sort data is read one segment at a time. When a segment is read into memory, each data record inside is first examined and then placed based on its key value in a "bin" in memory. There are a fixed number of these bins (set by the user), and each corresponds to a distinct range of key values. When a bin's allocated memory fills, its data is written to a corresponding temporary "bin file" on disk. There is one bin file for each bin in memory. Each bin in memory can be considered a cache for its corresponding bin file on disk. Thus, as the first phase progresses, these bin files accumulate data records that are guaranteed to all have keys in a specific non-overlapping range.

The second phase takes as input the bin files generated by phase one. Each bin file is read in order according to the range of key values it contains. When one is fully read into memory, it is sorted using a hybrid radix and quicksort technique and written to the output file in order.

In this algorithm, each phase must read data, process data, and write data to disk. To take full advantage of a system's hardware, none of these tasks should wait for another one to finish unnecessarily. That is, reading, processing, and writing should all happen concurrently when possible. Concurrency can be obtained in the sorting algorithm by using multiple threads or by writing code that switches internally between tasks with fine granularity.

There is a subtle issue regarding the concurrency of reading and writing. In a system with two disks, two blocks can be accessed concurrently. That is, one disk can write a block while the other is reading a block. In a system with only one disk, however, only one block can be accessed at a time. Thus if R blocks must be read and W blocks must be written, and r and w are constant factors representing the relative difference between read and write performance, the single disk machine takes $rR + wW$ time to perform the operation, whereas the two disk machine takes $\min(rR, wW)$. So if r and w are close in value, then the two disk machine will double the performance of the single disk machine. This argument generalizes to multiple disks in a straightforward manner. Therefore it is desirable to use at least two disks for our algorithm, and to ensure that the read/write performance is similar for both.

3.1 Theoretical Performance

DMSort overlaps reading, writing, and computation. All the sorted data passes through each of these sections during each phase. Therefore, assuming perfect overlap, the expected total time taken for a phase is $\max(R, W, P)$ where R is reading time, W is the writing time and P is processing time in seconds. In practice, our algorithm is limited by I/O and we get nearly perfect overlap of P with R and W. Therefore, we shall model performance as $\max(R, W)$. Given this model, DMSort's total performance measured as bytes per second is given by

$$\frac{B}{\max(R_1, W_1) + \max(R_2, W_2)}$$

where B is the number of bytes in the sort data size, and R_n and W_n refer to the reading and writing times for each phase of the algorithm.

HMSort's algorithm operates in a similar manner to DMSort's, so with good overlapping of computation and I/O, the theoretical performance is similar. However, NTSort does not overlap tasks, so it has a different theoretical performance. It's expected running time is $R + W + P$. This is clearly a poor design for high-performance sorting.

4 Hardware and System Software for Sorting

DMSort uses a hardware configuration designed to maximize the algorithm's performance while minimizing cost. Given our algorithm's theoretical performance, it is important to select a system with at least two disks and each disk should have similar read/write performance.

The two predominant disk interfaces for consumer machines are SCSI and IDE. While SCSI disks offer higher rotational speeds and better average seek times, their performance/price ratios are much poorer than IDE disks. Therefore we chose what we thought to be the best of the consumer-grade IDE disks at the time, the 60GB IBM Deskstar 60GXP. We clocked the raw device access of these disks at about 38 MB/s; about 1.25 times the bandwidth of HMSort 2000's disks.

In order to support fast I/O we selected a motherboard with two onboard IDE disk controllers. We chose this style of motherboard because it has four IDE channels, allowing us to use up to four disks simultaneously, and because it was only slightly more expensive than the motherboard without the extra controller.

We selected a 1GHz AMD Athlon processor for our CPU. These processors are very inexpensive, and provide performance comparable to alternative processors which are more than twice the price. For memory, we chose 768 MB of PC133 memory because it is also very inexpensive and provides good performance.

All of the previous entrants to PennySort have used Windows NT as their operating system. In the 2000 PennySort winner's system, Windows cost about $120 or 12% of the total system cost (Figure 2). Today we find Windows still retails for at least $110, while the price of other system components has dropped significantly. For a system between $500 and $1000, Windows consumes between 10% and 20% of the system cost! However, in recent years Linux has become a mature and well supported operating system and it is free. Assuming these operating systems have similar performance, it is difficult to justify the increasing percentage of total system cost consumed by Windows in light of a free alternative. DMSort runs on both Windows 2000 and Linux, so we are able to compare the performance of both. (Section 6.)

DMsort system						HMsort system				
Quantity	**Component Type**	**Description**	**Cost**	**% Cost**		**Quantity**	**Component Type**	**Description**	**Cost**	**% Cost**
1	Barebones System	ASUS A7V133 RAID Motherboard	$306.00	45.51%		1	Barebones System	500 MHz Pentium III	$372.00	36.83%
		AMD Athlon 1GHz						Promise Ultra66 ATA		
		Floppy Drive						Floppy Drive		
1	Operating System	Linux	$0	0.00%		1	Operating System	Windows 2000	$119	11.78%
3	Memory Chips	256MB PC133 RAM	$94	13.98%		1	Memory Chips	128MB	$146	14.46%
2	IDE Hard Drives	60GB IBM Deskstar 60GXP	$246	36.59%		2	IDE Hard Drives	Maxtor 7200rpm 10GB	$278	27.52%
1	Network Card	Realtek 8139c 10/100	$9.90	1.47%		1	Network Card	Acer 10/100	$25.00	2.48%
	Shipping	Ground from CA to WI	$41.48	6.17%			Shipping		$48.08	4.76%
		Total System Price	**$672.38**					**Total System Price**	**$1,010.00**	
		PennySort Time (sec)	**1406**					**PennySort Time (sec)**	**937**	

Figure 2: Breakdown of hardware component costs in our system as compared to the HMsort 2000 system. It is interesting to note that the relative cost of hard drives has increased in our system, despite a decrease in their price. Also notable is the sharp increase in memory size without a corresponding increase in price.

Based on the specifications of our disks we realized that a linear scaling of the bandwidth of four disks would saturate the 32-bit, 33MHz PCI bus in our system. Because IDE commands must go over the PCI bus as well, it would be impossible get a four-fold performance increase with four disks. However, there is a four-fold price increase with four disks, so our performance/price ratio drops. Using a faster bus could alleviate this problem but we were unable to find a motherboard with a faster PCI bus at a reasonable price.

Restricting ourselves to three or fewer disks, we benchmarked sequential disk performance in Linux and Windows. The results were suprising. Raw device read performance for two disks on separate controllers gives almost the expected doubling, but reading from three devices concurrently gives only an extra 5MB/sec of bandwidth over two disks (Figure 3). Because this is observed even when reading directly from the disk (skipping the operating system's file system layer) we conclude that a driver level limitation exists. We did not discover the root of this problem but instead decided to operate with two disks.

Doing further analysis with two disks revealed another problem. When reading from or writing through the file system to a single disk at a time, we obtained very fast performance. Reading from one disk and writing to another (when they are on different IDE channels) can proceed in parallel, so we expected to see nearly the same performance as the in non-concurrent case. However, as Figure 4 shows, we did not obtain this scaling. We contacted the Linux IDE subsystem maintainer, Andre Hedrick, about this problem. He explained that it is a known problem with the IDE subsystem on Linux, and he is working on a fixed version. Hoping to improve performance, we tried our system on Windows but got a similar performance measure for concurrent operation.

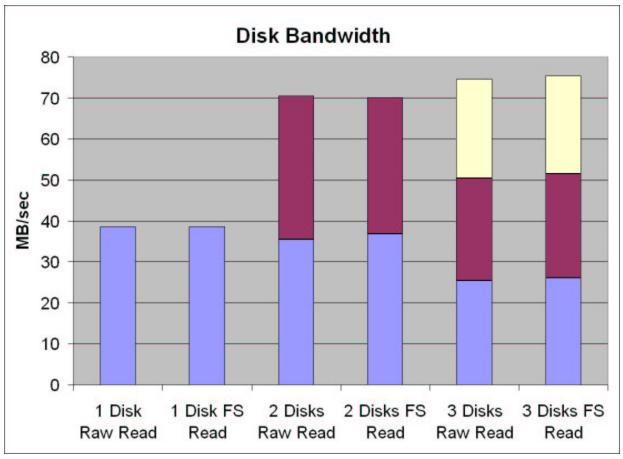

Figure 3: Performance measurements of disk bandwidth on our test system. From the left, the first two bars show about 38MB/sec reading rate for both raw device access and reading through the file system. The second two bars show concurrent read performance. The aggregate bandwidth here is nearly double the single disk case, as expected. However, in the three disk case, performance drops severely and we obtain hardly any performance increase at all. Tests performed under Linux.

Based on these and other measurements we concluded that DMSort would do best using two disks and that our maximum performance under Linux for each phase would be at most 21 MB/s, assuming perfect sequential access. Our final choice of system hardware with prices is presented in Figure 2 along with the components and prices of the most recent PennySort winner.

It is interesting to note the trends in component cost relative to the previous PennySort system. While memory cost has remained about 14% of total system cost, our systems memory is 13% of the target data set size, com-

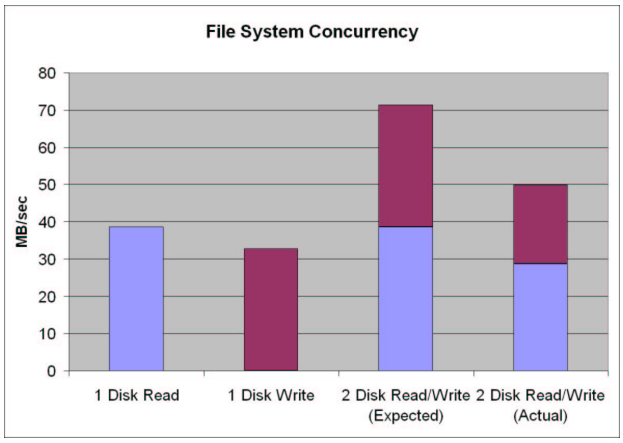

Figure 4: Expected concurrent read/write performance. From left to right, the single-disk read followed by the single-disk write bandwidth are shown. Next we see the performance we might expect if we got a linear scaling of the single-disk cases. However, we instead see only $\frac{2}{3}$ of this performance. This is attributed to an interaction between the Linux IDE subsystem and the filesystem.

pared to only 3% for the previous winner. The rapid increase in memory density and decrease in cost has also significantly affected Performance/Price sort. Because disks remain very slow compared to memory size and CPU speed, and memory prices have fallen significantly, the 60 second Performance/Price sort can now be implemented as an internal sort using 1GB of memory. We conjecture that these trends will continue for the near term. These trends imply that a fixed running time for Performance/Price sort does not ensure that a two-pass sort is practical. Thus, fixing the running time may not make for good I/O subsystem benchmarking.

5 DMSort Implementation

DMSort was implemented for the system configuration shown in Figure 2. The software works on both Linux and Windows. All three fundamental operations, reading, writing, and processing are overlapped for both sort phases. Overlapped execution is accomplished by switching internally between tasks with fine granularity.

Before running DMSort, the sort data resides on disk 1. The first phase of the sorting process reads the sort data and bins it as described in Section 3. Bins in memory are written to disk 2 as they fill. Because bins fill relatively evenly in memory, blocks of binned data are written to disk in an unpredictable order. The second phase reads the bin files from disk 2 in order. Because the bin file blocks are interleaved on disk during phase one, phase two's reading rate suffers due to seeking. When each bin file is finished reading, its records are sorted using MSB radix sort. Once they have been placed into a user-definable number of radix bins, the sort is finished off with quicksort on each radix bin. Finally, the sorted bins are written in order to the output file on disk 1.

Asynchronous I/O can be implemented in several ways, and a common method employed in PennySort in the past is multithreading [7]. However, Windows provides a specific asynchronous I/O implementation (via ReadFileEx() and WriteFileEx()), and Linux has two asynchronous I/O implementations available (KAIO from SGI [1], and glibc's POSIX asynchronous I/O implementation). When designing DMSort, we conjectured that the more general asynchronous I/O implementations would give us good performance, because asynchronous I/O is a common problem. From our testing and experiments, we found no significant performance limitations of asynchronous I/O for the kinds of access patterns we require, so we continue to employ it.

DMSort has a number of configurable parameters. For example, the number of bin files, the size of internal buffers, the amount of memory allocated to the sorting process, and the operational granularity may be set. Finding the best set of parameters for DMSort is difficult, but permits experimentation and fine grain benchmarking of operating system and hardware behavior.

6 Performance and Results

As mentioned in Section 3.1, we model DMSort's expected best running time for each phase as $\max(R, W)$ where R and W are the read and write running times. Read and write performance are measured in MB/sec. As shown in Figure 3, the concurrent reading and writing bandwidths of our disks are 28.6 and 21.1 MB/sec respectively. Assuming that $R_1 = R_2$ and $W_1 = W_2$, we can set the upper bound on our throughput to 21.1 MB/sec for phase one and phase two, for an overall throughput of 10.5 MB/sec. We emphasize that this figure is a conservative upper bound. In practice we will not achieve it because the disk I/O is not fully sequential.

DMSort on Linux is able to sort 125 million 100 byte

records on our system in 1380 seconds. The total sorting rate is 8.64 MB/sec. The binning rate (phase 1) is 18.45 MB/sec and the sorting rate (phase 2) is 16.31 MB/sec. Compared to the 2000 and 2001 results for HMSort, DMSort sorts 2.7 times as much data and achieves a sorting rate 1.78 times greater.

To draw a different comparison to year 2000 HMsort, we measured DMSort in Windows 2000. Since our system is cheaper than the HMSort system, comparing sheer volume of data sorted is of dubious value. Instead the total sorting throughput makes a better point of comparison since the sort is I/O bound. Furthermore, we must take into account that our disks are 25% faster than theirs. Our measured sorting throughput under Windows when sorting 80 million records is 7.38 MB/s, while HMsort 2000 achieved 4.84 MB/s sorting 45 million records. DMSort shows an improvement of 52%, which is beyond what might be expected given the 25% increase in disk bandwidth. We have concluded that our system's larger amount of RAM relative to the data set size allows us to achieve better locality on the temporary storage disk, hence reducing seek time. This implies that HMsort could do at least as well as DMSort in Windows if it were run on a system with a large amount of memory.

Because Linux is the primary platform for DMSort and it provides a longer PennySort running time, all of the following measurements were taken under Linux.

The phase one bandwidth is very close to the upper bound performance mentioned previously. This is not surprising because the I/O proceeds sequentially. However, the second phase does significantly worse. We first surmised that this was due entirely due to the disk seek overhead when performing a non-sequential read of the bin files. We modified the second phase of DMSort to measure the read performance alone. The modified version simply read the bin files and discarded the results, without sorting or writing. We obtained 31MB/sec of reading performance. This is 20% slower than the measured sequential read performance on our drives. Looking at our previous measurements of concurrent reading and writing, the sequential read performance is 28MB/sec. If we assume that we lose 20% of the concurrent read rate due to seeking, we would expect about 22MB/sec read performance. This is significantly higher than the observed 14-16MB/sec sort rate. However, if we assume that the writing disk must wait while the reading disk is seeking (for example, the block I/O layer might lock while one disk is operating) and scale back the expected sequential concurrent write performance of 21MB/sec accordingly, we get 16.8MB/sec. Given the numbers, this conjecture seems plausible, but we have yet to find a way to test it.

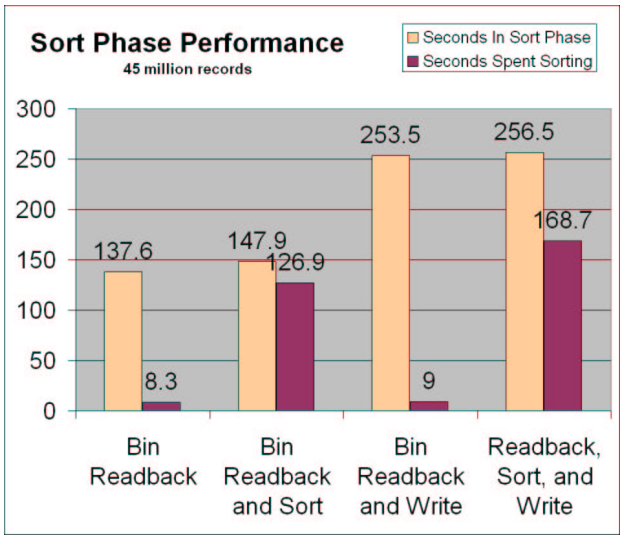

Figure 5: A measure of the degree we are overlapping sorting with I/O time. The pair of bars at the far left were obtained by running the second phase of DMSort modified to simply read back the bin files and discard them. Since sorting and writing were disabled, the reading time itself was measured. The next pair of bars to the right show what happens to execution time when sorting is enabled. Even though the sorting time has grown by more than a factor of 15, the total sort rate has grown by less than a factor of 1.1. The sets of bars on the right demonstrate DMSort runs with writing enabled and obtaining nearly perfectly overlapped computation and I/O.

We have performed "null-sort" experiments so that DMSort simply reads the bin files and writes them to disk without sorting. These experiments give almost exactly the same performance results as when we are sorting (Figure 5). This suggests that DMSort successfully overlaps sorting computation with I/O, and further suggests that the slowdown in the second phase is a limitation in the OS.

7 Conclusion and Future Work

We have presented a sorting system, DMSort, that beats all previously published PennySort records. DMSort obtains high PennySort scores using a combination of inexpensive, high performance hardware, sorting software that successfully overlaps operations, and a free operating system.

This work shows that sorting benchmarks can expose deficiencies in commodity hardware and software subsystems, like the Linux IDE concurrency problem. It also highlights the relative rates of technological advances in hardware and software systems.

DMSort performs well but could be further improved. At present it runs best on data which has keys evenly distributed throughout the keyspace. An adaptive approach to dividing the keyspace would be necessary to make DMSort robust. Furthermore, DMSort does not cache the first several bins in memory between phase one and two. Because memory sizes are so large (our memory size is approaching 10% of our sort data size) it could potentially save a significant amount of time by retaining all or part of the first few bins in memory between the sort phases.

8 Acknowledgments

We would like to thank Andrea Arpaci-Dusseau for her insightful comments and suggestions on DMSort, PennySort, and sorting benchmarks in general. We would like to thank Remzi Arpaci- Dusseau for reassuring us that using two or more disks is a good idea for PennySort. Finally, we would like to thank Nicole T. Perna's laboratory for sacrificing a workstation for the cause.

References

[1] Rajagopal Ananthanarayanan and SGI. Posix asynchronous i/o. http://oss.sgi.com/projects/kaio/.

[2] Shinsuke Azuma, Takao Sakuma, Tetsuya Takeo, Takaaki Ando, and Kenji Shirai. Diaprism hardware sorter – sort a million records within a second, 2000.

[3] Anon. et al. A measure of transaction processing power, 1985.

[4] J. Gray, J. Coates, and C. Nyberg. Performance/price sort and pennysort, 1998.

[5] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[6] Brad Helmkamp and Keith McCready. 1999 performance/price sort and pennysort. http://research.microsoft.com/barc/SortBenchmark/HMsort_1999PennySort.doc, 1999.

[7] Brad Helmkamp and Keith McCready. 2000 performance/price sort and pennysort. http://research.microsoft.com/barc/SortBenchmark/Y2000_PennySort.pdf, 2000.

[8] C.A.R. Hoare. Quicksort, 1962.

[9] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alphasort: A cache-sensitive parallel external sort. http://research.microsoft.com/barc/SortBenchmark/AlphaSort.html, 1994.

[10] Florentina I. Popovici, John Bent, Brian Forney, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Datamation 2001: A sorting odyssey. *not sure what journal...*, 2001.

[11] D. L. Shell. A high-speed sorting procedure, 1959.