OzSort: Sorting over 246GB for a Penny

Nikolas Askitis
Department of Computer Science
and Software Engineering
The University of Melbourne.
ozsort@csse.unimelb.edu.au
askitisn@gmail.com

Ranjan Sinha
Department of Computer Science
and Software Engineering
The University of Melbourne.
ozsort@csse.unimelb.edu.au
sinhar@unimelb.edu.au

Abstract

OzSort is a fast and stable external sorting software that is specifically optimized for the requirements of the PennySort (Indy) benchmark. The software sorts a file containing records of 100 bytes each, with keys of length 10 bytes, where a key starts from the first byte of each record and whose byte-length characters are generated uniformly at random. OzSort can sort over 246GB in less than 2150.9 seconds; that is, for less than a penny and extends the previous record set by psort in 2008 by an additional 56.7GB. We have used a computer that costs US$439.85 and provides a time budget $\geq \frac{3\times365\times24\times3600}{43985}$ seconds. The number of records that could be sorted within the time budget was $2,463,105,024$. OzSort can also be extended to cater to the Daytona class and be able to sort records and keys of varying sizes.

1 Introduction

We have implemented a sorting application, OzSort, based upon the classic external merge sort (1). There are two phases in this algorithm: `Run` and `Merge`. During the `Run` phase, the unsorted records in the input file are divided into smaller sorted portions called *runs*. After the completion of the `Run` phase, the input file is partially sorted and is comprised of the sorted portions that are each the size of a run. Depending on the number of records in the input file, there could be one final run that may contain fewer records than a run. During the `Merge` phase, the sorted runs are each fetched to memory to be merged and written to a final sorted file in a single-pass.

In our implementation, only two files are necessary: input file containing the unsorted records and the final output file with records in sort order. The output of the `Run` phase are stored in the input file itself and no temporary files are created. Consequently, the maximum space usage is $2NL$, where N is the number of records to be sorted and L is the length of each record. OzSort employs multiple threads and asynchronous I/O to efficiently read and write to the disks while the processor is kept busy with in-memory processing.

OzSort is designed for a 64-bit Linux operating system and a 64-bit x86 processor. It can also be run on older 32-bit processors that have 64-bit memory address support, such as some Pentium IV processors. However, we recommend using at least an AMD Dual Core with native 64-bit support, or an Intel Core 2 Series Dual-core (64-bit) processor.

The yearly trends of the PennySort Benchmark winners are shown in Table 1 and compared to the OzSort submission in 2009. In 1998, the datasize sorted per penny was only 1.27GB, while OzSort can sort over 246GB in 2009, an increase by a factor-of 194 within the past decade. Interestingly, the

improvement of OzSort over the 2008 winner was by more than that accomplished by GPU-terasort (55GB) in 2006.

Table 1: *Approximate yearly trends (2000-2008) of the PennySort Indy/Daytona Benchmark.*

Year	GB/penny	US$	Seconds	Group
1998	1.27	1142	828	NTSort (Gray et al.)
1999	1.98	1031	917	Stenograph LLC
2000	4.50	1010	936	Stenograph LLC
2002	9.80	857	1104	Tsingua (Liu et al.)
2005	15.00	950	996	Robert Ramey
2006	40.00	563	1680	Tsingua (Huang et al.)
2007	39.00	330	2863	Kuszmaul
2007	55.00	1543	648	GPU-terasort
2008	189.60	392	2408	psort
2009	**246.31**	439	2150	OzSort

In Section 2, we describe our implementation in greater detail and discuss the `Run` phase in Subsection 2.1 and the `Merge` phase in Subsection 2.2. The software configurations are discussed in Section 3 and a description of the hardware configuration is provided in Section 4. The experimental methodology and results are described in Section 6 followed by our conclusion in Section 7.

2 Algorithm

The algorithm is divided into two phases:

- `Run` phase: Partition the input file into run-sized portions. Fetch each run and sort them in main memory and write them back to input file.
- `Merge` phase: Divide each sorted run into smaller portions (called microruns) and fetch a microrun from each run and perform a k-way merge using the help of a heap and a queue to write the records in sort order to the final sorted file.

These two phases are described in further detail below.

2.1 Run Phase

The `Run` phase comprises of four core activities: reading runs from the input file, processing the key, sorting a run, and writing sorted runs to disk. A maximum of three threads are used at any moment for these activities.

The main thread reads in a run from the input file. Upon completion, it spawns another thread to partially prefetch the next run from disk while the main thread sorts the run. A run is divided into two halves and another thread is spawned to sort one half while the main thread sorts the other half. Upon completion of sorting of the two halves of a run, the two sorted halves are merged by the main thread and copied to output buffers prior to being written back to the input file on disk. Two output buffers of size 50MiB each are used, such that while one is being written to disk, the other can continue to be used by the main thread for merging the two halves of a run. When one output buffer becomes full, a thread is spawned to write the contents of that buffer to disk asynchronously, that is, without impacting

on the merge processing that now writes to the second output buffer. Once the second buffer fills, the merge process waits until the first buffer is completely written to disk before spawning a thread to write the second buffer asynchronously to disk; at which time, the first buffer (which is now empty) is reused by the merge process.

Memory usage Depending on the Operating System (OS) and the amount of memory reserved by the on-board graphics card (if any), about 100-200MiB of memory is reserved for the OS. We observed that if the sorting application enroaches this space, the virtual memory usage rises rapidly and the process slows down significantly. Hence, the parameter choices needs to reflect this fact and space reserved for the OS in main memory.

Four buffers are required during the `Run` phase and the parameters for these are tuned such that the maximum available memory is used. First, a *run buffer* is allocated to store one complete run and is at least 2GiB in size. Second, a *key-pointer buffer* that represents an array of structures, one (structure) for each record in the run. Each structure contains two 8-byte fields: a key and a record pointer. Third, a *prefetch buffer* is used to partially store the next-in-line run while the current run is being sorted. The prefetch buffer is set to slightly greater than 1GiB. Fourth, two *output buffers* are allocated to store the records in sort order and to write them to disk asynchronously. These buffers are each of size 50MiB.

Key generation The aim of this phase is to store the keys such that these are amenable to being sorted by efficient integer sorting algorithms and to maintain stability. The 10-byte key is appended with a 4-byte offset to the record in the run buffer and stored in little-endian format. During sorting, the entire structure (the key and the record pointer) is input to the comparison function to enable a stable sorting algorithm.

Sorting The runs are sorted using a general-purpose cache-efficient iterative quicksort (2). More efficient algorithms based on merge sort and radix sort can also be used, but it is unclear if algorithms based on radix sort can scale efficiently enough to provide significant gains on such large keys due to the relatively larger number of passes on 14-byte keys.

Overlapped I/O For efficient external memory sorting on large datasets, overlapping the I/O with in-memory processing is of crucial importance. Thus, the disk reads to the prefetch buffer are overlapped with in-memory sorting while the disk writes are overlapped with the in-memory merging of the two sorted halves of a run.

2.2 Merge Phase

The role of the merge phase is to retrieve each sorted run created during the `Run` phase and merge them using a k-way merge algorithm, where k is the number of runs. Due to memory limitations, only a fraction of one run (called microrun) is fetched to memory at any one time. The output of this phase is written to the final sorted file.

Broadly, the steps in the merge phase are:

1. Fetch the first microrun from each run, to form the initial working set in main memory.
2. Store the first key of each microrun in the working set into the *merge* heap, used to find the smallest record amongst all microruns.

3. Access the last record from each microrun in the working set and store the key and microrun file address in the *run-predict* heap, used by the queue to prefetch the next set of microruns from disk.

4. Extract the smallest record from the *merge* heap and write it to the output buffer; then add to the heap the next record (if any) from the microrun that contained the smallest record.

5. Write the contents of the output buffers to the final sorted output file.

6. When a microrun in the working set becomes exhausted, fetch the next micron from the queue (if any) to replace it, updating the *merge* heap and the *run-predict* heap accordingly.

A maximum of three threads are used during this phase. The main thread processes the working set composed of the k microruns. Two additional threads are spawned for managing the queue and the output buffers. The output is written to disk asynchronously and only one thread is writing to the disk at any point of time. As in the `Run` phase, if both the output buffers fill up, one buffer has to become free, that is, its contents have been written to the sorted output file on disk before the merge processing can proceed.

Memory usage Four buffers are used during the merge phase. First, a large *microrun buffer* is allocated to store the k-microruns (also called the working set). This uses the majority of the available memory and is approximately of size 3300MiB. In addition, an small array of pointers is used to point to the start of each microrun. Second, a small *queue buffer* is used to prefetch microruns from the disk and is managed as a FIFO. As a microrun in the working set exhausts, the queue copies the subsequent microrun from that run to the working set. A microrun is first transferred from disk into the queue buffer prior to being explicitly copied to the working set, so as to avoid potential problems with virtual memory. The queue buffer maintains less than 10 microruns by default at any point of time and typically uses about 300MiB, however in our final run, we have used a maximum of 8 microruns in the queue. Third, two small *output buffers* are allocated to store the records in sort order prior to being written to the sorted output file on disk. Each of the output buffer is of size 125MiB. Fourth, two small *heap buffers* are used, one for the run prediction and the other for the merging algorithm. The size of these heap buffers are small — in the hundreds of kilobytes.

Run prediction heap A key component of the `Merge` phase is an algorithm that can prefetch the microruns (in the background) and store them in the queue. This is accomplished by the run prediction algorithm that reads the last record of each microrun after it has been fetched to memory and stores it in a priority heap (along with its associated microrun file address). A thread (i.e., the queue thread) is spawned to maintain the queue in the background. When a microrun is read from the queue, the queue thread extracts the smallest record from the heap. This will, in turn, give us the file address of the next microrun to fetch from disk, which is stored in the queue buffer. The heap is then updated with the last record from the latest prefetched microrun. Once the queue fills, the queue thread waits until the main thread reads a microrun from the queue. The queue thread continues in this manner until all microruns are exhausted.

Merge heap At any point of time, this heap is used to store up to k-keys in sort order, the smallest current key from each microruns in the working set. As the smallest key is removed from the heap, the next key from the subsequent record of that microrun (if any) is copied to the heap to replace it.

This ensures that a key from each microrun in the working set is present in the heap for much of the processing, except for when a run (and thus its microruns) has been completely consumed.

3 Software Configuration

Out-of-the-box, OzSort is configured to sort a 246GB Indy dataset file consisting of $2,463,105,024$ records[1]. An Indy dataset is composed of 100-byte ASCII records, each starting with a 10-byte key consisting of random printable characters.

3.1 Operating system

We tested several 64-bit Linux operating systems during the development and testing of OzSort. We initially developed our software on an Intel platform (Q9550 processor on the Asus P5K Premium motherboard), and then later purchased and assembled the relatively inexpensive AMD systems. On the Intel machine, we installed the Gentoo operating system using the stage 3 installation procedure (which does not require a bootstrap). In addition, we also installed Kubuntu 8.10 (KDE/AMD64) and Fedora Core 10. We observed Fedora Core 10 to be noticeably slower to boot and more importantly, slower on initial disk accesses. Sorting a 10GB file, for example, took slightly longer on Fedora than either Kubuntu or Gentoo (despite using the same disk/partitions/formatting/code/datasets). In addition, Fedora was not as compatible with our AMD hardware, and thus we did not consider it further.

Gentoo offered the most flexibility during installation, yet, after extensive profiling, we did not observe any significant improvements in the performance of OzSort between Gentoo and Kubuntu. Kubuntu was just as fast and much easier to install. As a result, we decided to use Kubuntu which also makes it substantially easier to reproduce our results. However, prior to running any experiments, we did update the operating system (after the initial installation) to install the latest patches and Kernel (2.6.27) — this service was provided automatically by the operating system. We also installed several additional software packages not provided in the base Kubuntu installation, including the latest mdadm (the software raid driver), the latest g++ compiler, and XFS file system support. The softwares and their version numbers are provided below:

1. Kernel 2.6.27-11-generic SMP x86_64.
2. g++ version 4.3.2
3. mdadm version 2.6.7 (6th June 2008)
4. md5sum version 6.10
5. mkfs.xfs version 2.9.8
6. fdisk (util-linux-ng 2.14)
7. GNU Make 3.81
8. uniq version 6.10

[1] The source code is available upon request

3.2 File system

The XFS file system is well known to offer good I/O performance for large files (in some cases, XFS can approach the raw bandwidth offered by the hard drives). Although there are several other file systems available, XFS is generally the best option for this application. We strongly recommend that you format the RAID drive using XFS, if you intend to reproduce our results.

4 Hardware configuration

The choice of hardware is a key aspect of this benchmark. For our experiments, we purchased and tested a variety of hardware, all of which were listed (at the time of writing) on http://www.newegg.com. Screen-shots are provided in the appendix for your reference. The components we used for our final assembled PC are listed below:

1. AsRock A780GM (packaged with 1 SATA-II data cable, 1 SATA power adaptor (1-to-1) and manuals).
2. AMD Kuma X2 7750+ 2.7Ghz (2MiB L3 cache) processor.
3. GeIL 4GB 5-5-5-12 DDR2-800Mhz RAM (2x2GB dual channel).
4. 5x Seagate Barracuda 7200.11 160GB SATA-II hard drives.
5. A computer case + (integrated) standard power pack.
6. 4x SATA-II data cables, 2x SATA power splitters (1-to-2)

Table 2: *Hardware used for the OzSort entry for PennySort (2009). All prices are in US$ and were extracted from NewEgg website (http://www.newegg.com/) in March 2009. Discounts were excluded from the budget calculation.*

Component	Unit Price	Quantity	Sub-Total
ASRock A780GM motherboard	59.99	1	59.99
GeIL 4GB CAS5 (DDR2-800)	37.99	1	37.99
Seagate Barracuda 7200.11 160GB SATA-II	39.99	5	199.95
Case+power	33.99	1	33.99
AMD Kuma X2 7750+ 2.7GHz	59.99	1	59.99
Sata Power Splitter Cable (1 to 2)	2.49	2	4.98
SATA-II data cable (straigt-to-right)	1.99	4	7.96
Software: Kubuntu, Linux 2.6.27-11	0.00	1	0.00
PC Assembly	35.00	1	35.00
Total			439.85
Time Budget (94608000s/43985cents)			**2150.92**

Below, we discuss in greater detail the reasons behind choosing the hardware and the various issues faced.

4.1 Hard disk drive

As the size of the dataset has increased substantially to over 189GB per penny in 2008, ensuring a high data transfer rate from non-volatile memory was critical. To obtain high data transfer rates, we used a 5-disk RAID 0 system.

The hard disk drives used were SATA-II and each drive provided a minimum peak throughput of 120MB/s. For instance, our hard drives offered a peak throughput (outer rim) of 120MB/s, 126MB/s, 120MB/s, 131MB/s and 120MB/s respectively. The disk was partitioned into two portions: inner and outer. The outer partition of the disk was used to store the data files while the inner partition stored the operating system. We observed that the transfer rate varied substantially from the outer to the inner tracks. Thus, disks with sufficient spare capacity was chosen such that much of the data could reside on the outer tracks.

We purchased six 80GB Western Digital SATA-II hard drives (WD800AAJS). However, after extensive testing, these drives were found to offer poor performance despite their claim of being SATA-II. Each drive offered a maximum throughput of about 60MB/s (on the outer-rim of the disk). Our 160GB Seagate Barracuda drives, on the other hand, offered a bandwidth of at least 120MB/s. We contacted Western Digital technical support in Singapore and were told that these were most likely a manufacturing and/or printing/label error — as such the drives were promptly refunded. Interestingly, we also purchased and tested a single SATA-II 80GB Seagate Barracuda (7200.10) drive, which reported a peak bandwidth of around 70MB/s (not much better than the Western Digital drives). Overall, this experiment was quite enlightening, as we learnt to avoid Western Digital and Seagate 80GB SATA-II drives altogether.

As the size of the input files and the temporary files were nearly 500GB, the 80GB capacity disks were ruled out as at least seven of them would be necessary to provide adequate capacity and sufficient space for the data to reside in the outer tracks. Larger disks of 320GB were more expensive and while two disks would be enough to provide the capacity, it was insufficient to provide a high enough data transfer rate to be competitive. Thus, we chose a disk capacity of 160GB and used 5 disks for this project. The choice between using four disks and six disks were based on experiments that showed that a 5-disk RAID offers the best tradeoff.

4.2 Memory

We chose a memory size that offered the best performance; a relatively large memory was used to reduce the number of runs. A system with a total of 2GB of main memory was observed to slow down the sorting and consequentially, the merge phase, and as the cost of memory was low, we initially decided to purchase 4GB (2x2GB) of low latency (4-4-4-12) DDR2-800 RAM. We initially purchased CAS4 RAM in order to help reduce the cost of cache-misses during the run phase (and to some extent, the merging phase). We initially ordered a G. Skill Pi Black DDR2-800 4GB (2x2GB) dual-channel memory with 4-4-4-12 timings. Later, we purchased and tested another G. Skill DDR2-800 4GB (2x2GB) dual-channel memory, also with 4-4-4-12 timings (the F2-6400CL4D PK model). This model was slightly cheaper and after extensive testing, we observed virtually no difference in performance (with respect to OzSort), which made it favorable for our cause. We also tested a 4GB (2x2GB) G. Skill Pi DDR2-1066 memory with 5-5-5-15 timings on our Intel platform (using the Intel Q9550 processor) and observed virtually no improvement in OzSort's performance over our AMD KUMA CAS4 system, for large datasets.

As a result of this observation, we later tested three different brands of standard CAS5 memory on our AMD KUMA system: 4GB (2x2GB) CAS-5 Kingston (KVR) DDR2-800 RAM, 4GB (2x2GB)

5-5-5-15 GeIL DDR2-800 RAM, and 4GB of G. Skill (NT model) with 5-5-5-15 timings. As anticipated, we measured only a small loss in performance between our G.Skill 4-4-4-12 memory chips and the GeIL and G.Skill 5-5-5-15 memory chips, during the run phase. There was no noticeable impact on performance during the merge phase, however. Furthermore, the loss in time observed during the run phase was offset by the cheaper price associated with the 5-5-5-15 memory chips, making their use worthwhile for our project. As such, for our final system, we decided to use our GeIL 5-5-5-15 memory chips; however, the G. Skill 5-5-5-15 memory chips were just as good, with respect to the overall performance of OzSort.

The Kingston KVR brand, on the other hand, were found to be slightly slower particularly during the run phase. Unlike the GeIL or G. Skill memory chips, as far as we could tell, the Kingston KVR memory did not explicitly state 5-5-5-15 timings in the packaging (just CAS-5). We noticed (through research) however, that the Kingston HyperX models do explicitly support 5-5-5-15 timings.

4.3 Motherboard

We initially bought the AMD-compatible Asus M3N78-PRO, but our experience suggests that it is best to avoid using it for these experiments. Although we found this to be a high performance motherboard, it was not particularly "Linux-friendly". In addition, the Asus motherboard offered a relatively poor BIOS. For example, the BIOS had no option to set memory timings to 4-4-4-12. We tested our main memory performance using `memtest86` (provided by most Linux distributions), which reported the memory at 5-5-5-15. Another annoying feature of the Asus motherboard is that it forced us to waste a minimum of 64MB of main memory for the on-board graphics card. Adding a graphics card to the PCI-Express port would eliminate this shared memory issue, allowing full access to the 4GB of RAM. However, given the time/money constraint imposed by PennySort, adding a graphics card — even a basic one — reduced our overall time budget with virtually no improvement in overall system performance.

We therefore acquired three AsRock motherboards for testing: the A780FullHD, the A780GM-LE and the A780GM. All three AsRock motherboards were cheaper than our Asus motherboard, and all three allowed us to reduce shared memory to 32MB and provided a wealth of options in the BIOS. On all three AsRock motherboards, memtest86 correctly reported memory speed at 4-4-4-12 (when using our CAS4 memory chips). From our experience, the AsRock A780FullHD and the AsRock A780GM-LE were found to be equivalent, offering the same performance and almost the same specifications. Indeed, the motherboards physically looked similar, except that the FullHD version supported 16GB of RAM and used a different on-board graphics card (which was of no concern for our project). Side-by-side, the A780GM and the A780GM-LE motherboards look the same. Indeed, their BIOS offer the same options; the only difference being slightly different chip-sets involving the onboard graphics card, etc. We compared the performance of the A780GM-LE and the A780GM with respect to OzSort with large datasets. The overall performance of OzSort was found to the same on both AsRock motherboard; not surprising considering their physical similarities.

The AsRock A780GM-LE was, however, slightly cheaper than its sibling, the A780FullHD. Unfortunately, however, we could not use the A780GM-LE motherboard in our project (which would have added even more time to our budget) because at the time of writing, it was not listed at newegg.com. Nonetheless, the AsRock A780GM was listed and was a little cheaper than the FullHD, and we therefore used it in our final system.

4.4 Processor

We tested several variants of the processors ranging from quad-core to single-core. The AMD single-core used was the 2.6GHz AMD Athlon 64 Processor (LE 1640) with 1MB L2 cache. This was observed to be substantially slower than the dual-core and the much cheaper price did not offset the slowdown in performance. We then tested two versions of the AMD dual-core. These two versions were: 2.7GHz Dual-Core AMD Athlon X2 Processor with 1MB total L2 cache (5200), and the 2.7GHz Dual-Core AMD Athlon X2 Processor with 3MB Total L2+L3 Cache (7750).

Of the two AMD dual-cores, the 5200 was $3 cheaper than the 7750 version and gave us an additional 15 seconds. However, the performance of the 5200 was relatively poor probably reflecting the importance of the larger cache capacity. We decided to use the 7750 for the final runs as the cheaper costs could not offset the drop in performance.

Given our time constraint, the components listed in Table 2 were found to offer the best tradeoff for the PennySort benchmark. In particular, the multi-core processors we tested yielded superior results during the `Run` phase compared to the single core LE-1640 processor — which incurred substantially higher costs for main-memory processing. The performance of the `Merge` phase, however, was observed to remain reasonably consistent between all processors tested. We favored the AMD KUMA processor primarily because of its high performance during the `Run` phase. We do note, however, that its performance was only slightly better than the single core during the `Merge` phase.

We also tested these processors (including our AMD KUMA processor) on an alternative AMD motherboard (ASUS M3N78 pro). In addition, we tested the software on an Intel Q9550 processor mounted on an Asus P5K premium motherboard with 4GB of DDR2-1066 CAS5 RAM — which interestingly, yielded no significant performance improvement over our AMD KUMA processor on an AsRock motherboard using 4GB of DDR2-800 CAS4 RAM. Although the Q9550 is a more powerful processor with a substantially larger cache (12MB), most of its time is spent storing and retrieving data to disk. The processor, however, still plays a role in accelerating the sorting phase.

As an aside, during assembly, we also looked at the extent of the loss of *heat paste* due to the swapping of processors during the testing phase. Fortunately, the amount of heat paste lost was minimal and caused no significant increase in the overall CPU temperature, even with the computer case closed. However, we would like to note that in the process of removing the processor (5200), the heat paste was effectively glued to the heat sink. Our first attempt at removing the processor resulted in several pins being bent. This prevented the processor from re-fitting into the AMD slot in the motherboards. Fortunately, we were able to straighten the pins without damaging the processor. Interestingly, we noticed that the heat paste in the Intel quad-cores did not stick as much to the heat sink as those of AMD.

4.5 RAID chunk size

We decided to set the raid chunk size to 256KB, which is also the maximum block size supported by the current XFS file system. We also tested 64KB and 128KB, but did not notice a significant change in overall performance. We therefore recommend a chunk size of 256KB, which was observed to work well with OzSort.

4.6 Recommendations

To reproduce our results, we recommend that you run our software using the hardware listed in Table 2, and to tune the BIOS as described in Section 8. However, after extensive testing and profiling

on different computer architectures, we are confident that you should be able to approach the performance stated below, on computer systems with similar or better specs. To run our software on a similar machine, you should at least ensure that each hard drive is SATA-II and that each drive provides a minimum throughput of 120MB/s. Also ensure that you use the outer-rims of the disks. Our hard drives offered a (outer-rim) throughput of 120MB/s, 126MB/s, 120MB/s, 131MB/s and 120MB/s respectively.

Also make sure that you use at least 4GB of RAM (preferably at least a DDR2 800 with CAS5). If possible, also try to avoid using a single-core processor such as an Intel Pentium IV or an AMD Athlon LE or the AMD Sempron processors. OzSort should work fine on these (64-bit compatible) processors, but we observed a substantial impact on the performance of `Run` phase relative to the multi-core processors. Finally, ensure that your Linux operating system is kept under light load with no or minimal swap usage and no GUI.

5 Limitations

OzSort is currently restricted to the Indy dataset, however, the source code can readily be modified to support Daytona. It is primarily designed to sort large datasets, and as such, it needs to be slightly modified to sort a small dataset (which can fit within a single run); the modifications are described in the accompanying README file. OzSort may need to be tuned for high performance to support much larger files (say in excess of 300GB), by configuring the constants in the header file.

6 Results

Our final timed experiment of OzSort was run on the hardware architecture listed in Table 2. OzSort was compiled with g++ using *only* the optimization flag -O3. We also tested several other flags including -m64 -msse3 -mtune=athlon64-sse3 -march=athlon64-sse3 -funroll-loops -mfpmath=sse -funsafe-loop-optimizations. However, the overall performance of OzSort with these compiler flags enabled did not differ significantly from using just the -O3 optimization flag.
Prior to running any experiments, we ensured that the operating system was kept under light load by stopping all unnecessary services, and by running the experiments in console mode. The stopped services include kdm, powernowd, cron, cups, sysklogd, bluetooth, klogd, rsync, and citadel.

We observed that the 4-disk RAID times are slower than that of the 5-disk RAID which in turn was slower than the 6-disk RAID. These are along expected lines, and the choice came down to the time budget for these options. The 4-disk RAID had a time budget which was significantly lower than the reported time while for the 6-disk RAID the time budget was slightly lower than the reported time. Consequently, we decided to choose the 5-disk RAID system for this project. The results for the 5-disk RAID is shown in Table 3.

Table 3: *Time breakdown for OzSort of each phase for the PennySort Indy Benchmark. All timings are shown in seconds. The time budget for our application was* 2150.92 *seconds.*

Phase	Time (in seconds)
Run	**1016.34**
Merge	**1133.77**
Wall time (unix *time* command)	**2150.69**

7 Conclusion

OzSort is a fast and stable external sorting application that extends the PennySort Benchmark by a further 56.7GB over the 2008 Indy winner. It was able to sort $2,463,105,024$ records (which is over 246GB) within the time budget of 2150.9 seconds; that is, for less than a penny. OzSort can also be extended to the Daytona class benchmark.

8 Acknowledgements

This project is primarily supported by the Australian Research Council under the auspices of the ARC Discovery Project grant DP0771504 for Dr. Ranjan Sinha. In addition, it is also supported by a University of Melbourne Early Career Research Grant of Dr. Ranjan Sinha. We thank the Department of Computer Science and Software Engineering of the University of Melbourne for providing the facilities to undertake this project.

References

[1] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Reading, Massachusetts, 1973.

[2] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *J. Algorithms*, 31(1):66–104, 1999.

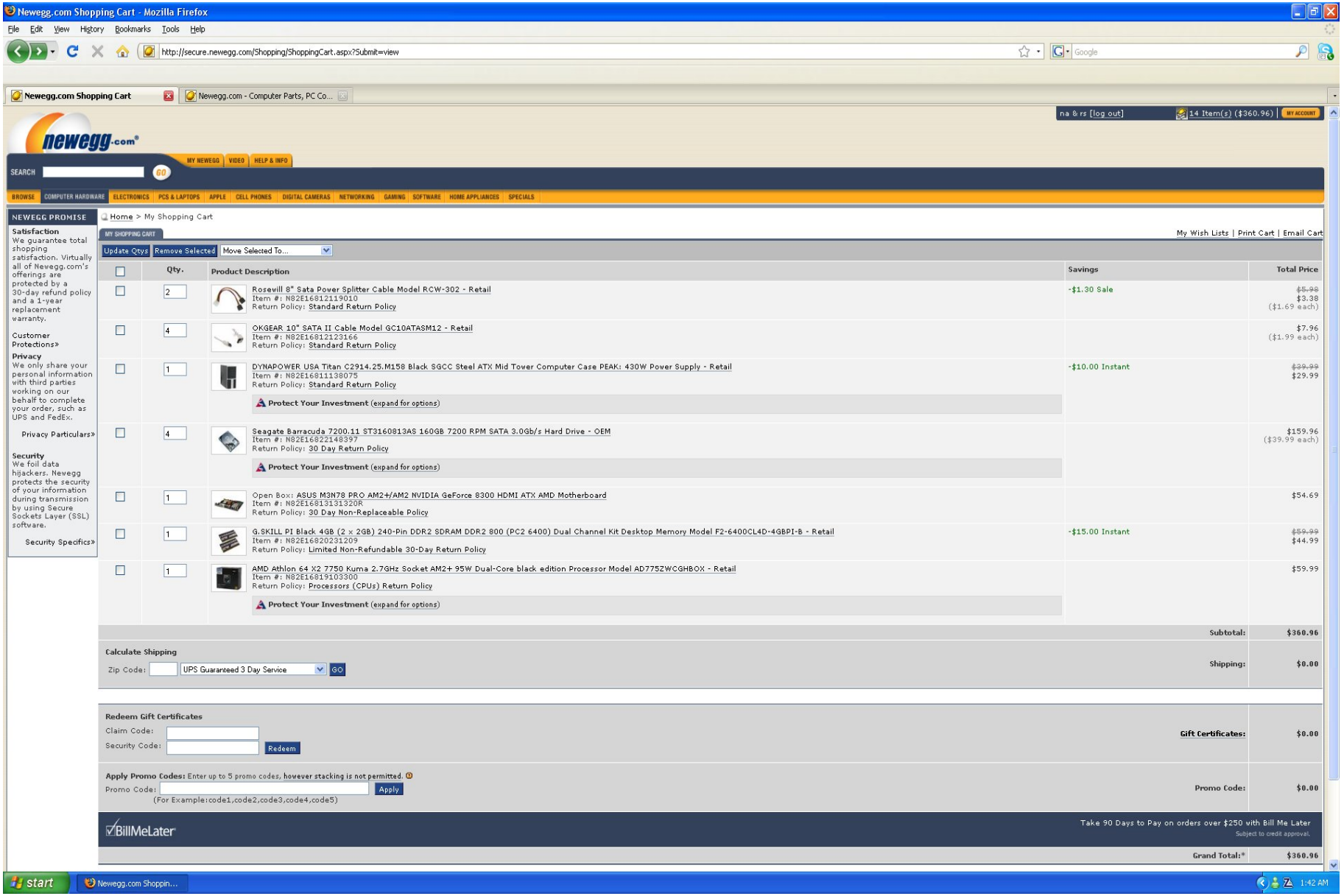

Figure 1: *Appendix A1: Component prices of original machine @ Newegg on 6th March 2009.*

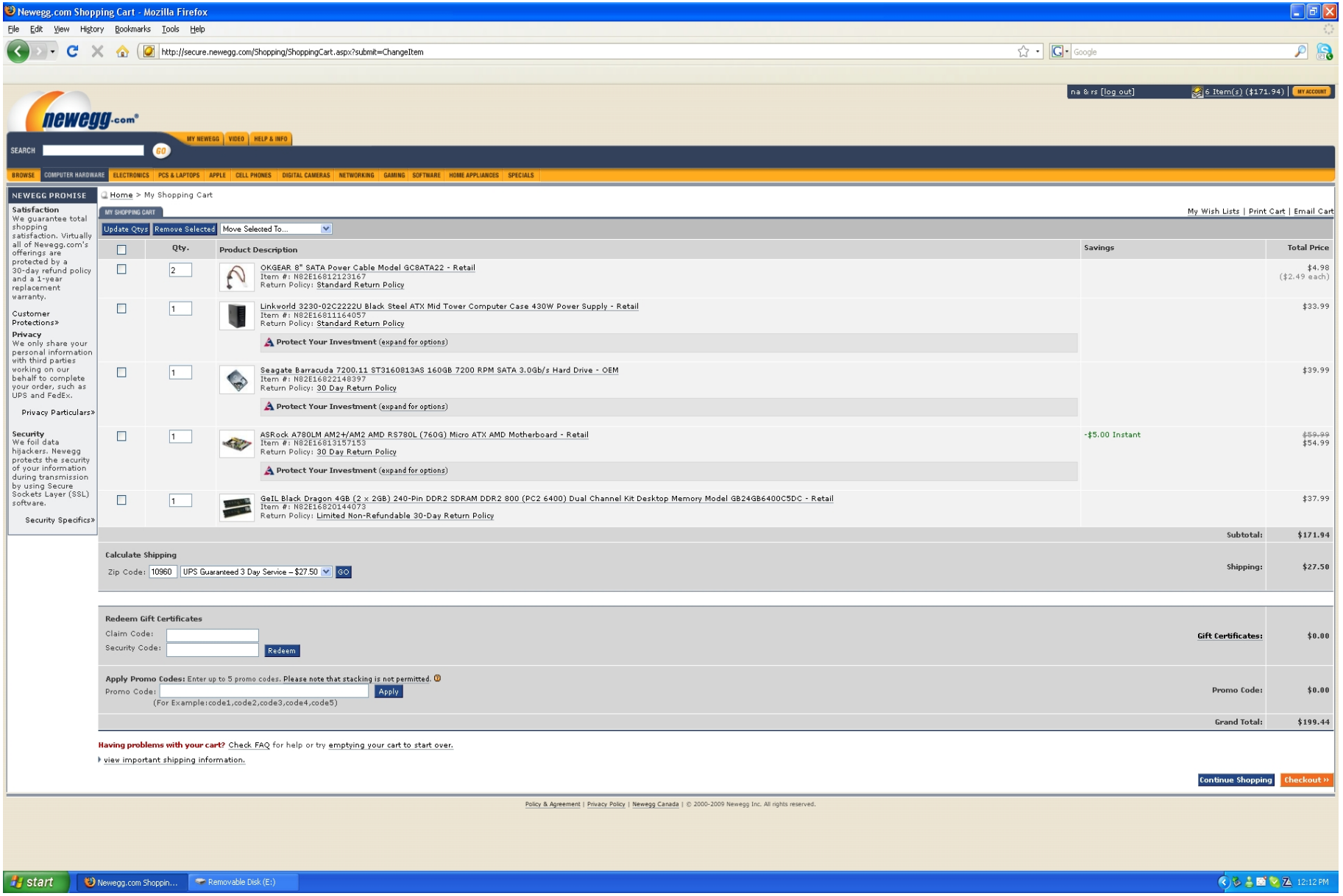

Figure 2: *Appendix A2: Component prices of added/replacement parts @ Newegg on 28th March 2009.*

Appendix B: BIOS configurations

8.1 BIOS configurations

We spent some time to explore the wealth of options provided by the AsRock BIOS. We first loaded the default BIOS settings, then updated the following:

```
CPU configuration:
   Overclock Mode: Auto      // default option, which means no over-clocking.
   Cool 'n' Quiet: Disabled
   L3 Cache Allocation: All Cores
   Memory Clock: 400MHz (DDR2 800)
   Memory Controller Mode: Unganged
   CAS Latency (CL): 5CLK
   TRCD: 5CLK
   TRP:  5CLK
   TRAS: 15CLK:

Chipset configuration:
   OnBoard HD Audio: Disabled
   Primary Graphics Adapter: Onboard
   Share Memory: 32MB
   CPU - NB Link Speed: 1800Mhz
   CPU - NB Link Width: 16-bit

IDE configuration:
   SATA Operation Mode: AHCI

Floppy configuration:
   Floppy A: disabled
```

Appendix C: Processor specifications

```
ozsort@ubuntu:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 2
model name      : AMD Athlon(tm) 7750 Dual-Core Processor
stepping        : 3
cpu MHz         : 2694.133
cache size      : 512 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb
rdtscp lm 3dnowext 3dnow constant_tsc rep_good nopl pni monitor cx16 popcnt
lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch
osvw ibs
bogomips        : 5388.25
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate

processor       : 1
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 2
model name      : AMD Athlon(tm) 7750 Dual-Core Processor
stepping        : 3
cpu MHz         : 2694.133
cache size      : 512 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
apicid          : 1
```

```
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb
rdtscp lm 3dnowext 3dnow constant_tsc rep_good nopl pni monitor cx16 popcnt
lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch
osvw ibs
bogomips        : 5388.44
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate
```

Appendix D: Available memory

The total available memory, as reported by our Kubuntu operating system.

```
cat /proc/meminfo | grep MemTotal;
MemTotal: 4021824 kB
```

Appendix E: Setting up the software RAID

To aid in reproducing our results, we have provided a screen-shot of our partitions below using the Linux command "fdisk -l". Your partition sizes can vary and the operating system can be kept on a separate disk if you prefer. In fact, we found that keeping the O/S on a separate disk proved to be a simple and convenient option during the initial development and testing phase, as it allowed for greater simplicity and flexibility in raid construction and testing. It also made it easier to move the raid drives between different machines, and eliminated to threat of losing the operating system and data due to partitioning/formatting errors, or damage caused by transit).

General rule-of-thumb: make sure the partitions used for the raid are of the same size and start at the same cylinder. Although software raid offers a lot of flexibility with regards to partition sizes and positions, maintaining uniform partition sizes/locations and the same brand/size disks can help improve overall performance. Also, you should always start your Linux raid partition from the first cylinder in each disk (which is usually the outer-rim, as was in our case).

```
Disk /dev/sda: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0xace7b1b4

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1               1       18630   149645443+  fd  Linux raid autodetect

Disk /dev/sdb: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x3cb500c0

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1       18630   149645443+  fd  Linux raid autodetect

Disk /dev/sdc: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x707b6276

   Device Boot      Start         End      Blocks   Id  System
/dev/sdc1               1       18630   149645443+  fd  Linux raid autodetect

Disk /dev/sdd: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x000265b7

   Device Boot      Start         End      Blocks   Id  System
/dev/sdd1               1       18630   149645443+  fd  Linux raid autodetect
```

```
/dev/sdd2   *    18631    18646    128520   83  Linux
/dev/sdd3        18647    18778   1060290   82  Linux swap / Solaris
/dev/sdd4        18779    19457   5454067+  83  Linux

Disk /dev/sde: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x5492c744

  Device Boot      Start         End      Blocks   Id  System
/dev/sde1               1       18630   149645443+  fd  Linux raid autodetect

ozsort@ubuntu:~# df -h
Filesystem     Size  Used Avail Use% Mounted on
/dev/sdd4      5.3G  2.7G  2.6G  52% /
tmpfs          2.0G     0  2.0G   0% /lib/init/rw
varrun         2.0G  116K  2.0G   1% /var/run
varlock        2.0G     0  2.0G   0% /var/lock
udev           2.0G  2.8M  2.0G   1% /dev
tmpfs          2.0G     0  2.0G   0% /dev/shm
lrm            2.0G  2.4M  2.0G   1% /lib/modules/2.6.27-11-generic/volatile
/dev/sdd2      118M   26M   86M  23% /boot
/dev/md0       714G  699G   15G  98% /mnt/raid
ozsort@ubuntu:~#
```