OzSort: Sorting 100GB for less than 87kJoules

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

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

Abstract

OzSort is a fast and stable external sorting software specifically optimized for the requirements of the 2009 PennySort (Indy) benchmark. OzSort can sort over 246GB of data for under a penny, using a standard desktop PC. The data sorted consisted of 100 byte records with (random) keys of length 10 bytes starting from the first byte of each record. In this paper, we apply our OzSort software and its associated hardware components to address the important issue of energy efficiency. By using our software and hardware components, we demonstrate that the power consumption of OzSort on a general-purpose desktop PC can rival that of CoolSort. OzSort could sort 100GB of data in about 827s using less than 87kJoules — 11,597 records sorted/joule. These results exceed that of CoolSort (Daytona class) in 2007, which used mobile processor technology and a RAID of laptop hard drives.

1 Introduction

To compete in the 2009 JouleSort competition, we reused our OzSort sorting software (and its associated hardware components), which was designed for PennySort 2009. Our OzSort implementation is 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, though this is not recommended.

The previous JouleSort Benchmark winners are shown in Table 1 and are compared to our OzSort submission in 2009. In 2007-2008, the energy consumption reported was 88.1kJoules using energy-

efficient hardware components designed for mobility. OzSort, in contrast, can sort 100GB using on average 86.2kJoules, which is slightly better than that of CoolSort (3) while using standard workstation components. Note that while OzSort was used for the Indy Benchmark, CoolSort was used for the Daytona Benchmark.

Table 1: *Yearly trends (2007-2009) of the JouleSort Benchmark.*

Year	kJoules/100GB	Application	Benchmark
2007	88	CoolSort	Daytona
2008	88	CoolSort	Daytona
2009	**86**	OzSort	Indy

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 2426MiB. In addition, a 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 can maintain a maximum of 12 microruns. 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 300MiB. 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.

2.3 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.

3 Software Configuration

Out-of-the-box, OzSort is configured to sort a 100GB Indy dataset file consisting of $1,000,000,000$ 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 installed the Gentoo operating system (on a Intel-based PC) 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 on our AMD system. 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 configurations, that is, 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

The Kubuntu operating system was kept on partitions in the inner-region of one of disks in the RAID-0 configuration. We show the partition layout of each drive in Appendix D.

3.2 Operating System CPU scaling

A key configuration used in this project was *CPU frequency scaling*, which we enabled in the Motherboard BIOS (this is usually enabled by default). This allowed us to change the CPU frequency on-the-fly, from a minimum of 1000MHz up to a maximum of 2600MHz (with 200MHz increments). CPU frequency was observed to be a key factor in overall power consumption and performance. A high CPU frequency (2600MHz) allowed OzSort to complete in less time, but consumed more power. Similarly, a low CPU frequency (1000MHz) consumed less power, but was significantly slower. What was needed was a balance between CPU frequency (power consumption) and performance. Fortunately, the command `cpufreq-set` found in most Linux distributions allowed us to set the CPU frequency to *on-demand*. Coupled with the BIOS configurations shown in Appendix A, and after disabling all unnecessarily services such as the Linux X (kdm) server (effectively Linux user level 2), the use of on-demand CPU-scaling was key to achieving low power usage.

As a result, when our system was *idle*, meaning no user processes besides those associated with the operating system were run, the total power consumed by our system (including CPU-fan, power-pack, 7 hard drives in RAID + PCI-E SATA-II controller, 2x2GB main memory units, on-board graphics, etc) remained at an average of 63 Watts.

3.3 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.

3.4 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 Hardware configuration

The choice of hardware is a key aspect of this benchmark. The components we used for our final assembled PC are listed below:

1. AsRock A780GM-LE (packaged with 1 SATA-II data cable, 1 SATA power adaptor (1-to-1) and manuals).
2. AMD Athlon LE-1640 2.6GHz (1MiB L2 cache).
3. Corsair XMS2 4GB 4-4-4-12 DDR2-800Mhz RAM (2x2GB dual channel).
4. 6x Seagate Barracuda 7200.11 160GB SATA-II hard drives.

5. 1x Seagate Barracuda 7200.11 320GB SATA-II hard drive.
6. A computer case, which included:
   (a) 500W Antec EarthWatts Power-pack (Model no. EA-500D, model rev. 00F).
   (b) No case fans.
   (c) The power pack (and perhaps to some extent the motherboard) consumed a total of 1 Watt when the computer is turned off but not unplugged.
7. 6x SATA-II data cables, 3x SATA power splitters (1-to-2).
8. 1x Linux-compatible PCI-Express SATA-II port card.
9. CCI Power-Mate Portable Watt Meter PM10A, with 1/10W precision and 1 second refresh intervals. According to the manufacturers, the typical accuracy error is below 0.5%. Further information can be obtained from the websites http://www.power-mate.com.au/ and http://www.power-mate.com.au/support.html.
10. Olympus μ1010 10.1 Mega-Pixel digital camera (with HQ 30FPS video capture capability) and a 2GB mem-card.

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

4.1 Hard disk drive

To obtain high data transfer rates, we used a 7-disk RAID 0 system. A 4-disk, 5-disk and 6-disk RAID-O setup was found to be inadequate. Although they were cheaper with respect to overall energy consumption, they could not offer enough bandwidth to be competitive for JouleSort. We observed that adding a disk to the RAID increased the system idle wattage by about 5 Watts. This is consistent with the manufacturer's claims of a 5.3 idle wattage for the Seagate Barracuda 7200.11 Drive. These drives have also been advertised to "Consume up to 43 percent less power during idle than previous products"[2].

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, 120MB/s, 123MB/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

[2] http://www.seagate.com/www/en-us/products/desktops/barracuda_hard_drives/barracuda_7200.11/

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. Thus, we chose a disk capacity of 160GB and used 7 disks for this project. (One of the disks was a 320GB seagate due to stock limitations; however, its performance was found to be equivalent to the 160GB disks).

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 2GiB of main memory was observed to slow down the sorting and consequentially, the Merge phase. We therefore decided to use 4GB (2x2GB) of low latency (4-4-4-12 timings) DDR2-800 RAM. We also tested 4GB (2x2GB) RAM (GeIL/G-Skill) with 5-5-5-15 timings and found their power consumption to be slightly less (by about a watt) than the low latency RAM. However, the reduced power consumption could not compensate for the cost to performance, particularly during the Run phase making the low latency RAM preferable for this project.

4.3 Motherboard

For our PennySort submission, we initially purchased the AMD-compatible AsusM3N78-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/power constraint imposed by JouleSort, adding a graphics card — even a basic one — increased overall energy consumption with virtually no improvement in overall system performance.

We therefore used our A780GM-LE AsRock motherboard. The AsRock motherboard allowed us to reduce the shared memory to 32MB and provided a wealth of options in the BIOS, which proved beneficial in reducing overall energy consumption. The BIOS modifications are shown in Appendix A. In addition, the AsRock motherboards supported and correctly reported memory clock timings at 4-4-4-12 (when using our CAS4 memory chips). For JouleSort, we decided to use the A780GM-LE motherboard.

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 1MiB of L2 cache. This was observed to be substantially more power-efficient than other alternative processors; the 2.7GHz Dual-Core (5200) AMD Athlon X2 Processor with 1MiB total L2 cache, the 2.7GHz Dual-Core (7750) AMD Athlon X2 Processor with 3MiB Total L2+L3 Cache, and an Intel Q9550 processor with 12MiB of shared L2 cache. Although these processors were substantially faster than the AMD single-core processor, their processing speed could not compensate for their high power consumption, even when

CPU scaling was enabled. Hence, to be competitive with JouleSort, the single-core was our best (though slowest) option.

5 Experimental Methodology

We assembled the PC and ran the experiments in a standard office in the ICT Building of the University of Melbourne during business hours, to take advantage of the (typical) climate control facilities, which kept the ambient temperature at a comfortable 20-25 degrees Celsius.

The steps we took to reproduce our results are as follows:

1. Start up the machine, drop into console mode and disable all unnecessary OS processes (details of such processes are provided in the README FILE provided with the software).
2. Set the CPU frequency to `on-demand` using the command `cpuset-freq -g ondemand`. The power consumed by our system when idle (that is, when no other processes are run besides those associated with the OS) was about 63 Watts.
3. Build the 100G dataset using gensort (gensort -a 1000000000). Make sure that no other files (apart from those associated with OzSort) are stored in the RAID to maximize the use of outer-sectors.
4. Position the video camera in a manner that captures the screen output along with the output from the power meter. We needed to use a video camera in these experiments because our power meter did not have data logging capabilities.
5. Start the video recorder (for logging purposes).
6. Run Ozsort and wait until it completes.
7. Stop the video recorder.
8. Transfer the video to a PC to be manually analyzed. Enter each second of power recorded by the power meter into a spreadsheet/database. Also keep track of the total time required to complete the task.
9. Repeat steps 3-8 five times.
10. Use the data logged in the previous steps to calculate the average time, average wattage and the standard deviations for all five runs. Use these calculations to determine the average energy consumption in joules and the number of records sorted per joule and their standard deviations for all five runs.

6 Results

Our final timed experiment of OzSort was run on the hardware architecture listed in Section 4. 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.

Table 2: *The power, run time, energy and records sorted per joule for each run is shown in columns 2-6. The average of these measures for all the five runs is shown in column 7. The total time (Run+Merge) represents the running time for the entire sort process (using Linux `time` command).*

	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	Average
Power (Watts)						
Run phase	104.5	103.8	103.6	103.5	103.6	103.8
Merge phase	105.0	104.7	104.4	104.8	105.2	104.8
Total (Run+Merge)	104.7	104.2	103.9	104.0	104.2	**104.2**
Run Time (seconds)						
Run phase	495.13	497.74	498.01	498.18	497.36	497.28
Merge phase	330.39	328.61	329.15	330.01	329.31	329.49
Total (Run+Merge)	826.21	827.08	827.88	828.97	827.38	**827.50**
Energy (Joules)						
Total (Run+Merge)	86504.2	86181.7	86016.7	86212.9	86213.0	**86225.7**
Records/Joule						
Total (Run+Merge)	11560.1	11603.4	11625.6	11599.2	11599.2	**11597.5**

Table 3: *The minimum, maximum, average power consumed during each phase and for each run.*

	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5
Run Phase					
Minimum	95.5	93.7	89.6	93.6	92.9
Maximum	113.2	112.6	112.3	112.5	112.6
Average	104.5	103.8	103.6	103.5	103.6
Merge Phase					
Minimum	94.0	88.4	85.8	93.3	94.6
Maximum	112.1	112.2	111.8	112.2	111.9
Average	105.0	104.7	104.4	104.8	105.2

Prior to running any experiments, we ensured that the OS 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, and rsync. The results for each run for the 7-disk RAID is shown in Table 2 and Table 3. The consolidated results are provided in Table 5.

Table 4: *The CPU utilization for the* `Run` *and the* `Merge` *phases for three runs. The Elapsed time is measured using* `gettimeofday()` *and the CPU time is measured using* `clock()`.

	RUN 1	RUN 2	RUN 3	Average
Run Phase				
Elapsed time	495.23	499.42	497.99	497.55
CPU time	479.49	480.66	481.53	480.56
Merge Phase				
Elapsed time	330.47	330.09	330.62	330.39
CPU time	272.14	272.86	272.74	272.58

Table 5: *The average (of five runs) of power, run time, energy, and records sorted per joule for the JouleSort Indy Benchmark. The standard deviation is also provided for each measure.*

Power (Watts)	Time (seconds)	Energy (kJoules)	Records/Joule
104.2 $\pm$ 0.3	827.50 $\pm$ 1.02	86.22 $\pm$ 0.17	11,597.5 $\pm$ 23.6

7 Conclusion

OzSort is a fast and stable external sorting application that is designed for the requirements of the Pennysort (Indy) benchmark, where it was shown to sort a further 56.7GB over the 2008 Pennysort (Indy) winner, that is, over 246GB for less than a penny. To our knowledge, OzSort is the first such submission to compete for the JouleSort (Indy) benchmark. It was able to sort 100GB for less than 87kJoules using standard workstation components, which was less than that of Daytona CoolSort (88.1kJoules) in 2007, which used mobile (laptop) components. The OzSort submission demonstrates that substantial savings in power usage can be obtained by using a well-engineered algorithm on standard workstation components.

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. Dr. Nikolas Askitis is also supported by the 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.

[3] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 365–376, New York, NY, USA, 2007. ACM.

Appendix A: 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:
    AM2 Boost: Disabled
    Overclock Mode: Auto // default option, which means no over-clocking.
    Cool n Quiet: Enabled
    Secure Virtual Machine: Disabled
    Memory Clock: 400MHz (DDR2 800)
    CAS Latency (CL): 4CLK
    TRCD: 4CLK
    TRP: 4CLK
    TRAS: 12CLK
Chipset Configuration:
    OnBoard HD Audio: Disabled
    OnBoard Lan: Disabled
    Primary Graphics Adapter: Onboard
    Share Memory: 32MB
    OnBoard GPU Clock Override: Enabled
    OnBoard GPU Clock: 450
    CPU - NB Link Speed: 1000Mhz
    CPU - NB Link Width: 16-bit
    CPU Thermal Throttle: Disabled
IDE Configuration:
    SATA Operation Mode: AHCI
    PCI/PnP Configuration:
    PCI IDE BusMaster: Disabled
    Floppy Configuration:
    Floppy A: Disabled
SuperIO Configuration:
    OnBoard Floppy Controller: Disabled
    Serial Port Address: Disabled
    Infrared Port Address: Disabled
    Parallel Port Address: Disabled
H/W Monitor:
    CPU Quiet Fan: Enabled
    Target CPU Temperature: [52C/122F]
    Target Fan Speed: [Slow]
```

Appendix B: Processor specifications

```
ozsort@ubuntu:~$ cat /proc/cpuinfo
processor         : 0
vendor_id         : AuthenticAMD
cpu family        : 15
model             : 95
model name        : AMD Athlon(tm) Processor LE-1640
stepping          : 3
cpu MHz           : 2600.000
cache size        : 1024 KB
fpu : yes
fpu_exception     : yes
cpuid level       : 1
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 syscall nx mmxext fxsr_opt
rdtscp lm 3dnowext 3dnow up rep_good nopl pni cx16 lahf_lm svm extapic
cr8_legacy
bogomips          : 5226.10
TLB size          : 1024 4K pages
clflush size      : 64
cache_alignment   : 64
address sizes     : 40 bits physical, 48 bits virtual
power management : ts fid vid ttp tm stc
```

Appendix C: Available memory

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

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

Appendix D: 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: 320.0 GB, 320072933376 bytes
255 heads, 63 sectors/track, 38913 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0xad00c8a8

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1               1       18630   312568641   83  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: 0x000265b7

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1       18630   149645443+  fd  Linux raid autodetect
/dev/sdb2   *       18631       18646      128520   83  Linux
/dev/sdb3           18647       18778     1060290   82  Linux swap / Solaris
/dev/sdb4           18779       19457     5454067+  83  Linux

Disk /dev/sdc: 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/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: 0x5492c744
```

```
   Device Boot      Start         End      Blocks   Id  System
/dev/sdd1               1       18630   149645443+  83  Linux raid autodetect

Disk /dev/sde: 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/sde1               1       18630   149645443+  fd  Linux raid autodetect

Disk /dev/sdf: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x000234d8

   Device Boot      Start         End      Blocks   Id  System
/dev/sdf1               1       18630   149645443+  fd  Linux raid autodetect

Disk /dev/sdg: 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/sdg1               1       18630   149645443+  83  Linux raid autodetect
```

To generate the software raid, open up a console (assuming your Linux operating system is up-and-running, that you have the same device/partition configuration, and that no other raid using these partitions has been defined) and type the following command:

```
./mdadm -create /dev/md0 -level=0 -chunk=256 -raid-devices=7
/dev/sda1 /dev/sdb1 /dev/sdc1 /dev/sdd1 /dev/sde1 /dev/sdf1
/dev/sdg1;
```

Then type:

```
./mkfs.xfs -f /dev/md0

mkdir /mnt/raid;
```

Once done, launch a text editor and open /etc/fstab and add the following line:

```
/dev/md0 /mnt/raid xfs users,async,exec,rw,dev,noatime,nodiratime,noauto 0 0
```

```
You should now be able to mount your raid:
mount /mnt/raid;

Our RAID-0 setup offered a peak bandwidth of 796MB/s.
(i.e., dd if=/dev/md0 of=/dev/null iflag=direct bs=1024000000 count=10;)
```

Appendix E: CPU utilization

The CPU utilization graphs for the Run and the Merge phases (based on a single run), using the `sar` command in Linux. This shows the CPU utilization at the user-level, the system-level, and the io-wait level. The definitions for these measures are

```
%user
Percentage of CPU utilization that occurred while executing at the user
level (application).

%system
Percentage of CPU utilization that occurred while executing at the
system level (kernel).

%iowait
Percentage of time that the CPU or CPUs were idle during which the
system had an outstanding disk I/O request.
```

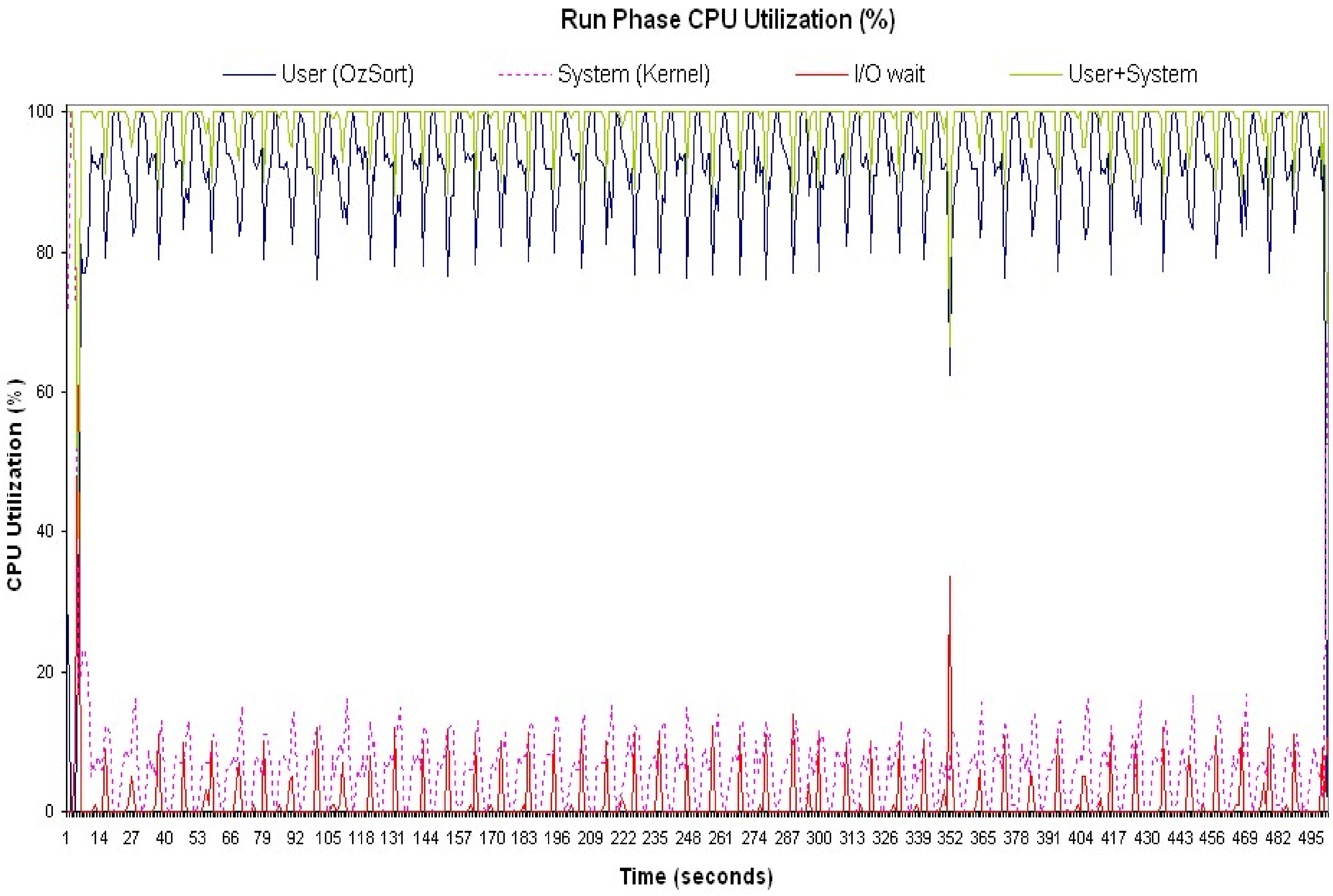

Figure 1: *Run Phase: The percentage of CPU utilization that occurred while executing at the user-level (application), the system-level (kernel), and the io-wait level using the Linux* `sar` *command. These results are based on a single run.*

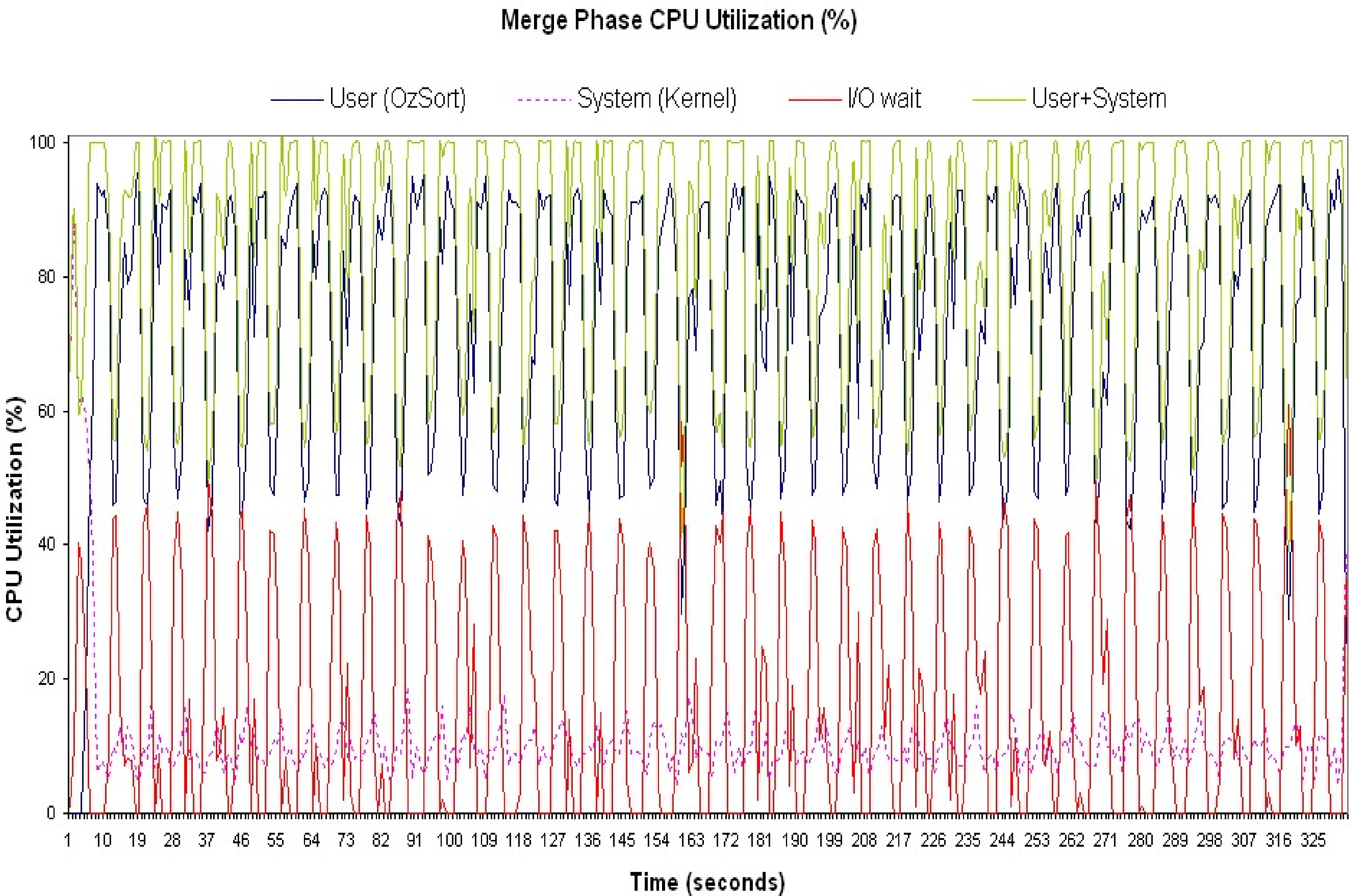

Figure 2: *Merge Phase: The percentage of CPU utilization that occurred while executing at the user-level (application), the system-level (kernel), and the io-wait level using the Linux* `sar` *command. These results are based on a single run.*