Energy-Efficient Sorting using Solid State Disks

Andreas Beckmann* and Ulrich Meyer
Goethe-Universität Frankfurt am Main
Robert-Mayer-Straße 11-15
60325 Frankfurt am Main, Germany
{beckmann,umeyer}@cs.uni-frankfurt.de

Peter Sanders and Johannes Singler†
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
{sanders,singler}@kit.edu

ABSTRACT

We take sorting of large data sets as case study for making data-intensive applications more energy-efficient. Using a low power processor, solid state disks, and efficient algorithms, we beat the current records in the JouleSort benchmark for 10 GB to 1 TB of data by factors of up to 5.1. Since we also use parallel processing, this usually comes without a performance penalty.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*; E.5 [**Files**]: Sorting and searching

General Terms

Algorithms, Design, Experimentation, Measurement, Performance

Keywords

Algorithm Engineering, SortBenchmark, Benchmark, Energy Efficiency, Power, Sort, Solid State Disk, Intel Atom

1. INTRODUCTION

Computers (and their cooling devices) have become a major factor in the consumption of electrical energy. Also, battery lifetime is the main limiting factor for many applications of mobile devices. Hence, reducing energy consumption of computers is now an important economical and environmental goal.

While this has fundamentally changed the way chips are designed, there is much less work on adapting the entire system, consisting of the hardware *and* the software running on it. A notable exception is the *JouleSort* benchmark introduced by Rivoire et al. [17] in 2007. JouleSort is a new category to the well-established Sort Benchmark[1], introduced by Turing award winner Jim Gray in 1985. As in the other categories of Sort Benchmark, the task is to sort 100-byte records containing a 10-byte key. There are two classes to participate in, the highly tuned Indy class (only needs to sort the Sort Benchmark record type and may use prior knowledge about the key distribution) and the more general Daytona class (needs to be able to sort general inputs without significant loss of performance). Orthogonal to the class, there are many categories, featuring different input sizes and metrics.

In the JouleSort category, the goal is to sort the input in as little energy as possible. The result is stated as sorted records per Joule. There are three size scales — 10^8, 10^9, and 10^{10} records, corresponding to 10 GB, 100 GB, and 1 TB of data.[2] We believe that JouleSort is an interesting framework for research in designing energy-efficient systems, since the sorting problem is at the same time non-trivial, simple to state, and of fundamental importance in many applications.

In this paper, we reconsider the design choices made in [17] and come to considerably different solutions, leading to a factor of up to 5.1 better performance per Joule. Instead of a large number of laptop disks, now, a small number of solid state disks (SSDs) wins. Instead of an ordinary (mobile) PC processor, we go down one scale on the ladder of processor categories and use the Intel Atom processor, which is designed for mobile devices and home servers. It turns out that using parallelism, this does *not* imply significantly longer running times.[3]

Section 2 explains our hardware choices, Section 3 the algorithms used in our programs, and Section 4 the implementation details. Then, Section 5 evaluates the performance of our approach with focus on the JouleSort benchmark. We conclude in Section 6.

Related Work

Rivoire et al. introduce JouleSort in [17], giving excellent arguments on why energy-efficient sorting is interesting. With

*Supported in part by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, and by DFG grant ME 3250/1-2.

†Partially supported by DFG grant SA 933/3-2.

[1] `http://sortbenchmark.org`

[2] In this paper as in the Sort Benchmark regulations, 1 GB = 10^9 bytes, 1 TB = 10^{12} bytes, and in addition 1 MiB = 2^{20} bytes, 1 GiB = 2^{30} bytes.

[3] We do have longer running time in the 1 TB scale, but we will explain that this is a very special case.

CoolSort, they present a highly efficient machine based on laptop technology, including a large number of 2.5" disks. This system is an order of magnitude more energy-efficient than systems currently used for database servers [3]. It uses the commercial *Nsort* [14] program in the Daytona class. In their conclusion, they already predict that flash memory will play a role in future JouleSort records. However, in [18], they report on several experiments including systems with SSDs and low power processors, none of which beats CoolSort. Only in the Jan 1 2010 Sort Benchmark results, they improve the 10^8 record Daytona result significantly, by factor of 2.1 with *FlashSort* [7]. Before, there was only a minuscule improvement in the Indy 10^9 scale by *OzSort* in 2009 [20].

Andersen et al. [3] propose to build a Fast Array of Wimpy Nodes (FAWN) for fast server-style computing that uses ARM processors and flash memory storage. It would be interesting to run the 10 GB JouleSort on one node of such a machine. Using FAWN for the larger inputs would be difficult since JouleSort requires input and output to be in a single file, and running a high performance distributed file system on FAWN sounds challenging. Moreover, FAWN uses Ethernet for communication between the nodes. It is not clear to us whether this is very energy-efficient for large networks, once one scales to a large number of nodes, because of the high bisection bandwidth required for the 1 TB JouleSort.

MapReduce has become a popular approach to large scale data processing. Its energy efficiency is considered in [6] where it is pointed out that optimizing the sorting phase inside the system is crucial for energy efficiency.

Lang and Patel [13] show how reducing clock frequencies helps to reduce the energy consumption of database queries (TPC-H style). We go one step further and propose to use low-clock rate processors optimized for energy efficiency.

2. HARDWARE CHOICE

Hardware components are usually most energy-efficient when they are fully loaded with necessary work. Usually, it is not useful to slow down operation artificially on a given device, because the energy savings do not outweigh the longer operation time, due to the idle power consumption, which may include underutilized peripheral components. Fully loading the system means fully loading all used components at the same time, resulting in what is called a "balanced" system in [17]. In our case, the crucial basic operations are I/O from/to non-volatile storage, performed by the disk controller and the disks, and internal computation, executed by the CPU.

In the field of non-volatile storage, a new technology has matured to become a competitor to hard disk drives (HDDs). Solid state drives (SSDs) combine many flash chips for HDD-like capacity. The absence of mechanical components allows for low access times and energy consumption, at least for reading. But still, for acceptable performance, accesses should be made in blocks of at least a few kilobytes, so SSDs cannot be treated as just slow RAM. *Writing* to SSDs should happen in even larger blocks, as we will explain later.

This newly available technology motivated us in tackling the JouleSort records. Together with the availability of lower-power CPUs, we saw a potential for significant improvements.

For the non-volatile storage, the criterion for comparing potential hardware is pretty clear. For a wide range of input sizes, the algorithm has to read and write every record twice (i. e. two passes). Therefore, we compare the devices by the average number of bytes they can read and write per Joule. This is justified since idle times will be avoided as much as possible, and also, the idle power consumption is very low, in particular when compared to the overall system.

Table 1 shows the average amount of data for reading and writing per Joule, for several HDDs and SSDs. The values are taken from [10, 9, 11].

The SSDs clearly outperform the HDDs, and moreover, they consume almost no power in idle mode, while the HDDs have a significant idle power consumption. The only drawback of the SSDs is their quite limited size and their high price (currently about 3 US$ per gigabyte, compared to 0.15 US$ per gigabyte for HDDs). But still, four 256 GB SSDs are sufficient even for the largest input considered here when using an in-place algorithm, and monetary cost is not included in the JouleSort metric.

The Intel X-25M looks the best theoretically, but was unavailable to us. Instead, we chose Super Talent UltraDrive GX MLC 256GB (FTM56GX25H, firmware 1916) drives, which have the same controller as the OCZ model listed, and is thus expected to show similar performance. We also tried Samsung PB22-J SSDs, but they let us down with very erratic performance behavior.

Independently of the choice of disk, the absolute I/O performance can still be varied by changing the number of SSDs attached, allowing to balance the I/O with computation.

The second important choice for JouleSort is the CPU. However, this decision also limits the options for the close peripherals, i. e. the mainboard including the I/O controllers (or slots to inserts them) and memory slots. We figured that I/O is quite "cheap" in terms of energy consumption, so we wanted to attach as many SSDs as possible to the system. On the other hand, the number of attachable disks usually does not scale well with the power consumption. Systems that can handle very many disks (i. e. having many SATA ports or powerful slots to insert additional SATA controllers) are designed for power-hungry high-end processors, and usually have a quite high idle power, caused by power-wasting chipsets and peripherals. To compensate for this, the number of disks would have to be increased hugely, sooner or later hitting other system bottlenecks like the internal bus bandwidth, or suffering overheads from high-degree RAIDs, like the necessity of a very large block size.

Thus, we were looking for very power-saving systems that could handle a reasonable number of SSDs out-of-the-box. We decided on the Intel Atom processor because it was explicitly designed for building systems with little power overhead. Still, we had to choose between a single-core or a

Disk Model Unit	Capacity GB	Read MiB/s	Write MiB/s	Read W	Write W	Efficiency MiB/J
SSD						
Intel X-25M	80	225	79	1.0 W	2.5 W	128
Samsung PB22-J	256	201	180	1.1 W	2.8 W	124
OCZ OCZSSD2-1VTX120G Vertex Series	120	214	123	1.3 W	2.2 W	110
Intel X-25E	32	226	198	1.7 W	2.7 W	103
HDD						
Western Digital WD7500KEVT/00A28T0	750	82	82	2.0 W	2.0 W	41
Samsung HM500JI SpinPoint M7	500	87	87	2.3 W	2.3 W	28
Samsung HD502HI SpinPoint F2 EcoGreen	500	106	108	6.6 W	6.6 W	16

Table 1: Energy efficiency of SSDs and modern HDDs.

dual-core variant. For us, this decision turned out to be tied to the decision for the chipset and the mainboard (Atom processors are usually soldered directly on the board).

The system is based on a Zotac IONITX-A mainboard, equipped with an Atom 330. This processor consumes more than three times the power (8 W TDP) but features two cores and four hardware threads. The main advantage of this system is that its nVidia Ion chipset provides four SATA ports that handle the SSD transfers at full speed. Moreover, it allows two DIMMs for a total of 4 GiB of RAM. The 64 bit logical address space is less prone to fragmentation, which we experienced on the 32-bit Atom N270.

Table 2 summarizes the key features of the system.

In [17], a quite exotic combination of hardware components was used, namely a mobile processor plus 13 laptop hard drives, attached to server-class RAID controllers. In contrast to this, our machine has only few components and would fit into a small ITX case. The only exotic aspect is the use of very expensive SSDs. But note that smaller, much cheaper disks could be used for the 10^8 and 10^9 scales. Figure 1 shows a picture of our system.

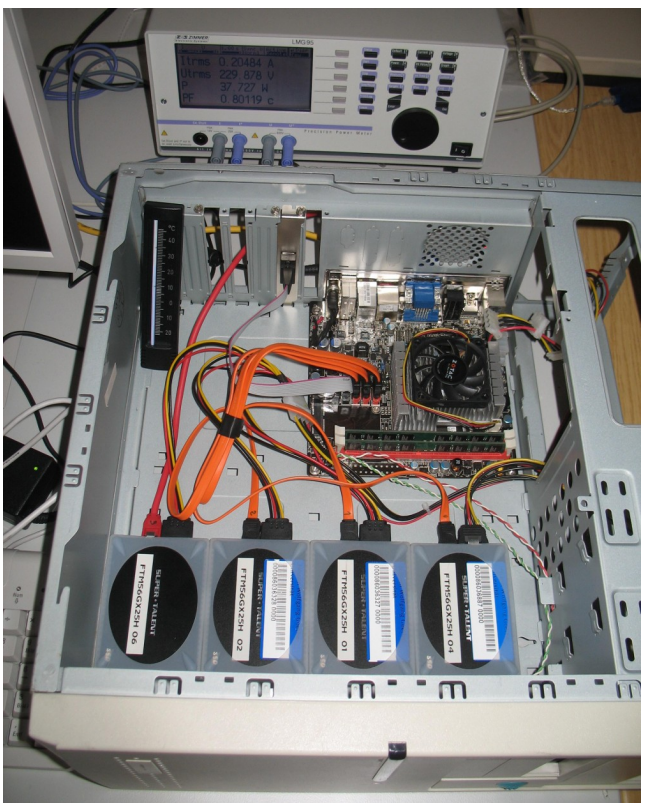

Figure 1: Our Zotac IONITX-A machine together with the power meter (displaying 37.7 W of current overall power consumption).

2.1 SSD Issues and Configuration

While USB flash drives and flash-based memory cards are ubiquitous nowadays, flash devices that are intended to replace HDDs have appeared only recently. SSDs provide a capacity of several hundred gigabytes, whereas current HDDs store as much as 2 TB.

By accessing many flash chips in parallel, SSDs achieve bandwidths that are significantly higher than those of HDDs. While reading, they also feature very short access times, allowing random accesses on blocks in the kilobyte range with almost peak bandwidth. Writing is a completely different story, though. A data item can be written to only after being erased (flashed), which happens in comparatively large blocks. Thus, random writes of kilobyte-scale blocks can be very slow. Therefore, algorithms have to be adapted to this asymmetry between reading and writing [2].

Even worse, the maximum number of erase cycles is limited. To prevent early failure of the disk, the built-in *wear-leveling* methods strive to distribute the writes evenly [12]. This management can cause erratic performance behavior. Certain write patterns may lead to internal fragmentation,

Component	Type	TDP	Estimated Price (08/2009)
Mainboard	Zotac IONITX-A		250 $
Processor	Intel Atom 330 2 cores, 4 threads, 1.6 GHz, x86_64	8 W	
Chipset	nVidia Ion	12 W	
Memory	Kingston 2x2 GiB	4 W	75 $
Disks	4x Super Talent FTM56GX25H	4 W	4x 740 $
Fan		1 W	
OS drive	High Speed USB Pen Drive	1 W	65 $
Case, Cables, ...			125 $
Assembly			35 $
Estimated Total (net)		30 W	
Estimated Total (overall)		37.5 W	
Typical Idle (overall)		26.3 W	
Typical Loaded (overall)		37 W	
Total Cost			3500 $

Table 2: The tested hardware. Higher overall energy values are caused by losses within the external DC power supply (typically running at about 80% efficiency). Prices (including all taxes and no discounts) are based on the German market (August 2009) and converted at a rate of 1.43 US$/EUR.

causing an even permanent performance degradation. Fortunately, this fragmentation can be reset by applying a low-level formatting operation (ATA SECURITY ERASE) to the disk. But in that case, of course, all data is lost. File systems can also help avoid fragmentation of the disk by reporting blocks that are not used anymore to the disk, by issuing operations called *trim* or *block discard*. However, hardware and operating system support is limited so far, e. g. the functionality is unavaible on RAID volumes.

The question arises how to treat storage devices that are unreliable in this sense in a benchmark. You get best performance and reproducible results by formatting before the benchmark, but this is not a realistic model for production use. We study this question by running the test many times after formatting, evaluating whether degradation takes place.

3. ALGORITHMICS

In this section, we describe the algorithms applied by our programs, which we use for the Indy results. For the Daytona results, we will use Nsort.

The amounts of data to handle do not fit into internal memory, i. e. the RAM. The rules also require both the input and the output to reside on disk.

For efficiency, we apply the *external memory* model here, which allows to transfer data from and to disk only in blocks of a certain size[4] B. We base our algorithms on multiway mergesort, which has been proven optimal for this setting [1]. It allows to sort up to $\mathcal{O}(M^2/B)$ records with two passes, where M is the internal memory size. Similar algorithms were already used several times by successful Sort Benchmark entries, e. g. *psort* [5].

The algorithm has two main phases. In the first phase, it splits the input into many *runs* of size $\mathcal{O}(M)$. In practice, the constant factor varies between 1 and 1/4, depending on the implementation. These runs are read from disk, sorted internally, and written back to disk. In the second phase, all runs are merged using a multiway merger. A *prefetch* sequence ensures that the right blocks are prefetched asynchronously, while the CPU performs the merging. Writing out the finished blocks is also overlapped with computation.

When looking closer at the algorithm, one notes that there are random block accesses only in one stage, namely when reading the blocks for merging. In contrast, run formation can transfer $\mathcal{O}(M)$ records at once. The output chunk size in writing the result in the merge phase is also only limited by the available buffer memory. So fortunately, the only operation that really needs random block access only *reads* data, a discipline that SSDs are particularly good in.

4. IMPLEMENTATION

In the Indy class of JouleSort, we are allowed to take full advantage of the fixed record size and the short keys that are chosen uniformly at random.

Our programs, namely EcoSort and DEMSort, support full overlapping of I/O and computation, in both phases using the techniques available from STXXL [8]. Due to the limited speed of the Atom processors, it was important to further tune the internal sorts and merges.

- In memory, the algorithm sorts only the keys associated with an index, and permutes the sequence of full records later in parallel.
- Sorting the keys is done by applying parallelized least significant digit radix sort to the three most significant bytes of the key, and then sorting the partially sorted sequence by insertion sort. Giving prefetching hints to the distribution stage improves performance here.
- We use the parallelized function `multiway_merge` from the libstdc++ parallel mode, based on the Multi-Core

[4]All sizes in this discussion are in number of records.

Standard Template Library (MCSTL) [19].[5] The parallel multiway merger reaches best performance if a single invocation can output at least B records. Having 3 blocks per run buffered in internal memory (2 for the merger to guarantee at least B elements available per run, the third block for prefetching) can only be done by further reducing the block size at the cost of I/O performance. Therefore, we use only 2 blocks per run and combine blocks that will run empty soon.

As a consequence, the disk bandwidth is fully utilized at about 90% of the time, i. e. we are close to I/O-bound. Further tuning of internal computation just for the sake of consuming a little less power by the processor (by idling or reducing its clock rate) seemed not worthwhile, given the overall power consumption of the system. However, we have conducted tests on the number of threads to use (see Section 5.3).

4.1 Sorting 10^8 and 10^9 Records

For the 10^8 and the 10^9 record Indy category, we have developed the program *EcoSort* using the parallelized version of the Standard Template Library for XXL Data Sets (STXXL) [4].

The STXXL provides two sort implementations, `stxxl::sort()`, which writes the output back to the location of the input, and `stxxl::stream::sort()`. The latter does not need to know the number of elements to be sorted in advance, and produces output in a way that can be pipelined to further algorithmic steps. Combining the best features of these two sorters, we engineered EcoSort using direct (non-pipelined) access to the input blocks (like the standard sorter) and writing output to a new file (as usually done with the pipelined sorter). The following additional tuning measures were applied.

- Avoid repeated allocation/deallocation of memory for the temporary buffers by using a free list.
- Overlap loading the first run with initialization and writing the last run with cleanup operations.
- Reducing size of the first two runs to $\frac{1}{3}$ and $\frac{2}{3}$ of the regular size, thus loading the CPU earlier from the start.

4.2 Sorting a Terabyte

For the largest scale category, 10^{10} records, things get more complicated.

First of all, we have only 2.4 % more storage capacity than absolutely needed, so we have to sort *in-place*. Nsort is not capable of doing that, but overwriting the input disqualifies the result for the Daytona class anyway. There is just not enough disk space in the machine to qualify for 1TB Daytona[6], so we stick with Indy for this input size.

[5]The corresponding sort routine could also be used to provide fully comparison-based sorting. The overhead is significant, but not fatal.

[6]Using 512 GB SSDs would have solved this problem, but given the additional monetary effort, this did not seem worthwhile.

We used a variant of *DEMSort* [16, 15] because of its capability to sort *in-place*. DEMSort currently leads the Indy GraySort and Indy MinuteSort categories, and is originally designed to run on distributed-memory clusters, but the overhead is only small when running on a single node. It was augmented with the features mentioned at the beginning of this section.

We get close to the $\mathcal{O}(M^2/B)$ limit for two passes, which we like to adhere to by all means. From the 4 GiB of RAM, the BIOS allows to use only 3.5 GiB. Subtracting the system memory usage and the program binary, we are left with only about 3.25 GiB. Due to overlapping, we need space M once for writing the last run and reading the next one, and once to store the current run, $0.16M$ for the keys (10 bytes key, 2 bytes filler, 4 bytes index), and another M for doing the permutation out-of-place. Thus, for run formation, M is limited to about 1 GiB or 10^7 records, resulting in $R = 973$ runs.

For the multiway merging, we need at least two buffer blocks per run, but have the full memory available, so this yields for the block size: $2BR \leq M \Leftrightarrow B \leq M/(2R) = 3.25 \cdot 10^7/(2 \cdot 973) = 16\,700$ records, or 1.67 MB. Finally, we chose a block size of 1 433 600 bytes, to allow a larger write buffer.

This configuration is very tight. Enlarging the input size, e. g. by a factor of 2, would require either more internal memory (hardware incapable), an additional pass (+50% running time), or even smaller blocks (further decreasing the effective bandwidth for merge reading).

To achieve external in-place operation, the blocks read in the merging phase must immediately be returned to the file system. In the version used for GraySort, this was achieved by creating a file *for each block* of the formed runs, and deleting it when obsolete. This incurred quite some file system overhead. Thus, we improved this by using a file *per run*, reducing the number of files by three orders of magnitude. The data is written in (block-wise) reversed order, so that in the merging phase, blocks are immediately returned to the file system after reading, by truncating the run file appropriately.

However, even if the runs are placed sequentially on disk (with respect to logical disk addressing), the output file will necessarily be fragmented. This is because the file system has to fill the gaps between the runs due to the lack of additional space. The SSDs can counter this by doing so-called *write combining*, but nevertheless, transferring a *logically* non-contiguous range will incur overheads.

To improve this situation that suffers from small blocks, we replaced the block I/O by *range I/O*, except for reading in the merge step. The program issues a single system call for reading/writing an entire run, and for writing a large chunk of the output. This is equivalent to a huge block size that amortizes all overheads. Also, this approach does not contradict the external memory model.

5. EXPERIMENTS

5.1 System Details

We removed the included wireless module from the IONITX-A mainboard and disabled other unused hardware in the BIOS setup if possible. The machine had no keyboard or monitor attached, but was connected to the LAN and remotely controlled via secure shell (SSH). To preserve all SSD space for sorting, the OS was installed on a USB flash drive.

The Zotac IONITX-A board requires 19 V DC. We used the included 90 W power supply, a smaller one (Dehner SYS1319-3019, 30 W) actually consumed about 5% more energy, despite the fact that it was better utilized.

As operating system, we employed the Debian *sid* distribution with a 2.6.33 Linux kernel. There was only a minimal set of services running, and no graphical user interface. Our programs were compiled with GCC 4.4.3, optimization switched to `-O3`. The RAID was managed in software by the *device mapper* of the Linux kernel. All partitions were formatted using XFS.

5.2 Energy Measurement

For measuring the energy consumption, we used a ZES ZIMMER LMG95 precision power meter. This device can be controlled via the serial port, and has an accuracy of less than ±0.1% for the applicable measurement range. We started the measurement just before the sorting program, and stopped it right after termination, while sampling power in 1-second intervals. We calculated the energy by multiplying the average power by the time reported by `/usr/bin/time`. The temperature was about 23 ° C throughout the tests, the CPU fan was running.

The results in time and energy consumption are averaged over five runs for each category, we also give the respective standard deviation.

All measured power values for our machine denote overall power consumption, i. e. they include losses within the DC power supply.

5.3 Setup

For all tests, we used four SSDs configured as a RAID-0. Sequential transfer tests showed a peak performance of about 1 000 MiB/s read and 600 MiB/s write for the RAID.

The input for all runs was generated by the *gensort* program provided by the Sort Benchmark committee, running in ASCII mode.

Before each program run, the (old) output file was removed. After each run, the output was checked for correctness, either by running *valsort* or by comparing its MD5 sum with a reference MD5 sum precomputed from a checked output.

The programs were allowed to consume up to 3.25 GiB of RAM for their operation.

For the 10^8 category we removed one of the two 2 GiB RAM modules and allowed the programs to consume up to 1.75 GiB of RAM. This comes with a small performance penalty (running time increases up to 5 %) which is outweighed by reducing power consumption by about 10 % (about 2.8 W idle, 3.8 W during the experiments). The run

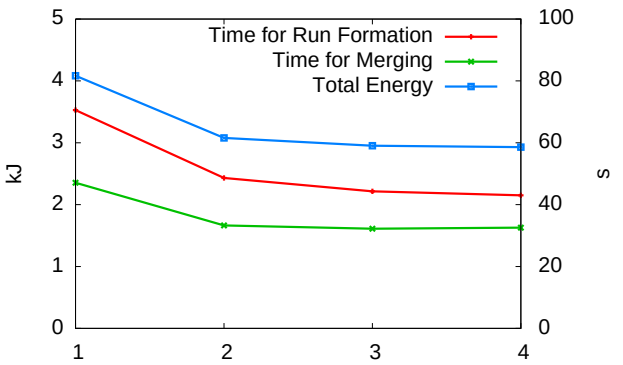

Figure 2: Evaluating the optimal number of threads.

formation phase of the 10^8 category was further restricted to 0.85 GiB of RAM because the increased number of runs (49 instead of 24) resulted in better overlapping and lower running time without affecting the merging phase.

For categories 10^8 and 10^9, we used a block size of 13 107 200 bytes.

For the TB scale using DEMSort, we lowered the block size to 1 433 600 bytes, as explained before. We newly generated the input before each run due to the lack of space, after the RAID was cleaned of all files, which implies that we newly created the temporary files and the output file each time.

For choosing the optimal number of threads, we conducted pre-tests for 10 GB. As shown in Figure 2, 4 threads for run formation have the best running time, and there is a tie between 3 and 4 threads for merging. The energy consumption goes along these lines, so we simply took 4 threads in all cases.

For Nsort (version 3.4.28, trial), we used the following command line:

```
nsort -format=size:100
-field=name:key10ascii,size:10,pos:1,char
-key=key10ascii -statistics -processes=4
-method=radix -memory=3250M
-file_system=/mnt/ssdraid0,
                transfer_size:16M,count:C
-temp:/mnt/ssdraid0/nsort/,
                transfer_size:16M,count:C
```

with `C=4` for 10^8 records, and `C=2` for 10^9 records.

5.4 Categorization

The results of our EcoSort and DEMSort programs count for the Indy class because our programs can handle only fixed-size records. Also, some optimizations rely on a certain key universe.

The Nsort program is generally acknowledged to adhere to the Daytona specifications. Hence, we submit its results,

	Indy		
JouleSort Category (records)	**10^8**	**10^9**	**10^{10}**
Program	EcoSort	EcoSort	DEMSort
Memory Configuration	1 × 2 GiB	2 × 2 GiB	2 × 2 GiB
Data Volume	10 GB	100 GB	1 TB
Number of records	100 000 000	1 000 000 000	10 000 000 000
Checksum	2faf0ab746e89a8	1dcd615efb9dfe11	12a06cd06eeb64b16
Time	72.4±0.24 s	691±3.2 s	17 026±29 s
Energy	**2 345±9 J**	**25.09±0.11 kJ**	**571.8±1.1 kJ**
Average Power	32.4±0.07 W	36.3±0.01 W	33.6±0.03 W
Records per Joule	**42 635±168**	**39 853±183**	**17 489±33**
Typical Bandwidth Run Formation	540 MiB/s avg	600 MiB/s avg	711 MiB/s avg 867 MiB/s read 603 MiB/s write
Typical Bandwidth Merge	610 MiB/s avg	600 MiB/s avg	151 MiB/s avg 187 MiB/s read 127 MiB/s write

	Daytona	
JouleSort Category (records)	**10^8**	**10^9**
Program	Nsort	Nsort
Memory Configuration	1 × 2 GiB	2 × 2 GiB
Data Volume	10 GB	100 GB
Number of records	100 000 000	1 000 000 000
Checksum	2faf0ab746e89a8	1dcd615efb9dfe11
Time	75.7±0.10 s	756±0.9 s
Energy	**2 485±4 J**	**27.94±0.03 kJ**
Average Power	32.8±0.02 W	37.0±0.02 W
Records per Joule	**40 249±59**	**35 789±44**
Typical Bandwidth Input Reads	230 MiB/s	240 MiB/s
Typical Bandwidth Temp Writes	230 MiB/s	240 MiB/s
Typical Bandwidth Temp Reads	330 MiB/s	300 MiB/s
Typical Bandwidth Output Writes	330 MiB/s	300 MiB/s

Table 3: Summary of the Sort Benchmark results.

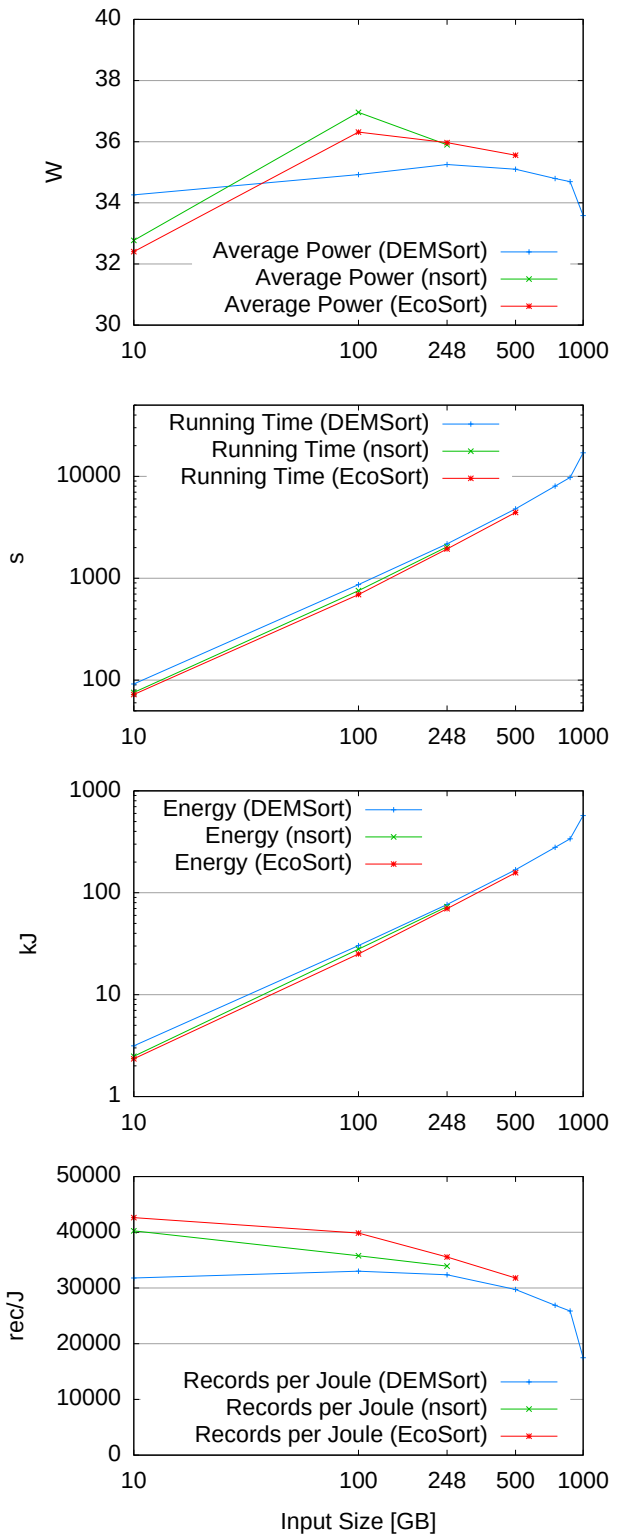

Figure 3: Performance for inputs of increasing size.

which are a bit worse than the Indy results, to the Daytona class.

5.5 Indy Results

All Sort Benchmark results are subsumed in Table 3, accompanied by the respective checksums. No duplicates were found for all inputs.

The Indy results for 10^8 and 10^9 records improve over the 2009 Daytona and Indy records [17, 20] (11 600 and 11 300/ 11 600 records per Joule) by factors 3.7 and 3.5, respectively. The absolute running times are also better than for the competitors. The achieved bandwidth is about 600 MiB/s averaged over reading and writing, equivalent to about 75% peak performance.

For the 1 TB scale, however, the energy efficiency is far worse compared to the 100 GB scale. To find out the reason, we ran tests for other input sizes in the gap between 100 GB and 1 TB using the same parameter values, namely 248 GB, 500 GB, 750 GB, and 875 GB. The results stated in Figure 3 show that the degradation is correlated to the filling degree of the SSDs. The curve bends sharply from 875 GB to 1 TB, indicating that filling the SSDs almost completely is inadvisable. The performance degradation must be caused by increasing internal fragmentation, as explained in Section 2.1. For 1 TB, the I/O rate drops dramatically after run formation, when the merger has to fill the gaps on disk, namely from 603 MiB/s to 151 MiB/s (averaged over read and write). At least, there is no further degradation for multiple runs in a row, even when a multiple of the disk capacity is written in total.

We still beat the previous record [17] (3 425 records per Joule for Indy) by a factor of 5.1. However, the cited result was not achieved using the mentioned low-power system, but with solid server hardware accessing 12 disks and consuming 400 W. Thus, we cannot compete in terms of running time in this case, our program takes about 2.4 times as long as the competitor.

We had even worse results for the I/O bandwidth degradation when generating the input using regular OS-buffered I/O. Instead, we wrote 1 GB blocks to the RAID by piping the `gensort` output through `dd oflag=direct,nonblock obs=1G`. We consider this justified because you cannot blame the sorter for a non-optimally generated input.

Ultimately, the terabyte result is an artifact of the space scarceness in our system. Using twice as large disks would most likely have provided the performance of the 500 GB run, which is almost 30 000 rec/J, and would beat the previous record by a factor of more than 8, needing only 33% more absolute running time. We hope to achieve a similar result by using the block discard feature of the file system, as soon as this is supported for RAID volumes.

The improvement of our Indy results compared to our Jan 1 2010 submission are mostly due to optimizing the internal sorter, using prefetching, insertion sort instead of bubble sort, and only three rounds of radix sort instead of four. Also, the further degradation of the SSDs after the first run for 1 TB disappeared, maybe some firmware had not been

correctly installed before.

5.6 Daytona Results

The result of Nsort for 10^8 records and 10^9 records are about 6% and 11% worse than their respective Indy counterparts, which is not too bad. The running time is 5% and 10% higher respectively, and the average power consumption is 1–2% higher.

The Jan 1 2010 Daytona competitor for 10^8 records, FlashSort, only achieves 24 800 records per Joule, which is about a third less efficient. The system used there is not really low-power, featuring a quad-core AMD processor. It mainly benefits from the 16 GiB of RAM, which can hold the full 10^8 record input, making intermediate storing unnecessary. From the 80 GB net capacity of the Fusion-IO high-performance SSD used, only 37 GB were available for use, so the price per gigabyte approached about 100 US$. Due to these restrictions, there is no result for the 10^9 record category, showing that this approach does not scale well.

6. CONCLUSIONS

We have demonstrated, that home server hardware together with highly tuned, parallelized sorting algorithms can sort large amounts of data considerably more energy-efficiently than previous systems.

7. REFERENCES

[1] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In Jan Vahrenhold, editor, *Experimental Algorithms, 8th International Symposium*, volume 5526 of *Lecture Notes in Computer Science*, pages 16–27, 2009.

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a fast array of wimpy nodes. In *22nd ACM Symposium on Operating Systems Principles*, pages 1–14, 2009.

[4] Andreas Beckmann, Roman Dementiev, and Johannes Singler. Building a parallel pipelined external memory algorithm library. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[5] Paolo Bertasi, Marco Bressan, and Enoch Peserico. psort, yet another fast stable sorting software. In Jan Vahrenhold, editor, *Experimental Algorithms, 8th International Symposium*, volume 5526 of *Lecture Notes in Computer Science*, pages 76–88, 2009.

[6] Yanpei Chen and Tracy Xiaoxiao Wang. Energy efficiency of map reduce. UC Berkeley, 2008.

[7] John D. Davis and Suzanne Rivoire. Flashsort: Joulesort benchmark entry, 2010 daytona 10 gb class, 2010.

[8] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.

[9] Boi Feddern. Entdeckungsreise. *c't: Magazin für Computer Technik*, 11/2009:100–104.

[10] Boi Feddern. Platten-Karussel. *c't: Magazin für Computer Technik*, 10/2009:104–107.

[11] Boi Feddern. Platten-Karussel. *c't: Magazin für Computer Technik*, 21/2009:142–145.

[12] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, June 2005.

[13] Willis Lang and Jignesh M. Patel. Towards eco-friendly database management systems. In *4th Biennial Conference on Innovative Data Systems Research*, 2009.

[14] Chris Nyberg, Charles Koester, Ordinal Technology Corp, Jim Gray, and Microsoft Corporation. Nsort: a parallel sorting program for NUMA and SMP machines, March 19 1997.

[15] Mirko Rahn, Peter Sanders, and Johannes Singler. Scalable distributed-memory external sorting. In *26th IEEE International Conference on Data Engineering (ICDE)*, pages 685–688, 2010.

[16] Mirko Rahn, Peter Sanders, Johannes Singler, and Tim Kieritz. DEMSort — distributed external memory sort, 2009.

[17] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 365–376, 2007.

[18] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, Christos Kozyrakis, and Justin Meza. Models and metrics to enable energy-efficiency optimizations. *IEEE Computer*, 40(12):39–48, 2007.

[19] Johannes Singler, Peter Sanders, and Felix Putze. The Multi-Core Standard Template Library. In *Euro-Par 2007: Parallel Processing*, volume 4641 of *LNCS*, pages 682–694. Springer-Verlag.

[20] Ranja Sinha and Nikolas Askitis. Ozsort: Sorting 100 GB for less than 87 kJoules, 2009.