# 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 use the JouleSort Benchmark as a case study for making data-intensive applications more energy-efficient. Using home server hardware, solid state disks, and efficient algorithms, we beat the previous records by a factor of 3 and 4 respectively, and more traditional systems by an order of magnitude. Since we also use parallel processing, this comes basically 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. Moreover, 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. [16] in SIGMOD 2007. JouleSort is a new category to the well-established SortBenchmark[1], introduced by Turing award winner Jim Gray in 1985. As in the other categories of SortBenchmark, the task is to sort 100-byte records containing a 10-byte key. There are two classes where sorters may participate, the highly tuned Indy class (only needs to sort the SortBenchmark input and may use prior knowledge about the key distribution) and more general Daytona class (needs to be able to sort other inputs without significant loss of performance). In JouleSort, the winning system has to sort as many records per Joule as possible. 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 energy-efficient system design since the sorting problem is at the same time non-trivial, simple to state, and of fundamental importance in many applications, in particular in data bases. To date, the performance from [16] (*CoolSort*, using the *Nsort* [13] program) in the Daytona class (which serves as a lower bound for Indy) has not been beaten, except for a small improvement in the $10^9$ scale by *OzSort* in 2009 [19].

In this paper, we reconsider the design choices made in [16] and come to considerably different solutions, leading to a factor of 3 to 4 better performance per Joule. Instead of a large number of notebook 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, 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 [16] giving an excellent explanation why energy-efficient sorting is interesting. With CoolSort, they present a highly efficient machine based on notebook technology, including a large number of note-

[*] 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 SortBenchmark regulations, 1 GB = $10^9$ bytes, 1 TB = $10^{12}$ bytes, and in addition 1 KiB = $2^{10}$ bytes, 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.

book disks. This system is an order of magnitude more energy-efficient than systems currently used for data base servers [3]. In their conclusion, they already predict that flash memory will play a role in future JouleSort records. However, in [17], they report on several experiments including systems with SSDs and low power processors, none of which beats CoolSort. We profit from the increased performance of recent SSDs.

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. But 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 the 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 [12] show how reducing clock frequencies helps to reduce the energy consumption of data base queries (TPC-H style). We go one step further and propose to use lower clock frequency 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 system, because the energy savings do not outweigh the longer operation time, due to the base load. Thus, we did not experiment with the clock rate or the like. We just rely on the automatic frequency management.

Recently, a technology has matured to become a competitor to hard disk drives (HDDs) in the field of non-volatile storage. *Flash* chips are combined to SSDs with 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, the access should be done in blocks of at least a few kilobytes, so SSDs cannot be treated as just slow RAM. Writing to an SSDs should take place in even larger blocks, as we will explain later.

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

For the non-volatile storage, the desired classification is pretty clear. For a wide range of input sizes, the algorithm has to read and write every record twice (i. e. two passes). So we can compare the available devices by the average number of bytes they can read and write per Joule. Table 1 shows the average amount of data for reading and writing per Joule, for several HDDs and SSDs. The values are taken from [9, 8, 10].

The SSDs clearly outperform the HDDs, and moreover, they consume almost no power in idle mode, while the HDDs have a significant base energy consumption. Their only drawback 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 the 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. We also tried Samsung PB22-J SSDs, but they let us down with very erratic performance behavior. The I/O performance of the system can now be varied by changing the number of SSDs attached.

The next most important choice for JouleSort is the processor. We decided on the Intel Atom processor because it was explicitly designed for delivering good performance per Watt. A further choice is whether to use a single-core or a double-core processor. For us, this decision turned out to be tied to the decision for the chip-set and board to be used (Atom processors are usually soldered directly on the board).

The system is based on a Zotac IONITX-A board, equipped with an Atom 330. This processor consumes more than three times the power of an N270 (8 W TDP) but supports two cores and four hardware threads. The main advantage of this system is that its nVidia Ion chipset provides four SATA ports that can 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.

A quite exotic combination of hardware components was used in [16], namely a mobile processor plus 13 notebook 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.

### 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. The 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 hard disk drives. While reading, they also feature very short access times, allowing random accesses on blocks in the kilo-

| Disk Model | Capacity | Read | Write | Read | Write | Efficiency |
|---|---|---|---|---|---|---|
| Unit | GB | MiB/s | MiB/s | W | W | 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.**

| Component | Type | TDP | Estimated Price (08/2009) |
|---|---|---|---|
| Mainboard | Zotac IONITX-A | | 250 $ |
| Processor | Intel Atom 330<br>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) | | 25 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.**

byte 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 [11]. This management can cause erratic performance behavior. Certain write patterns can lead to internal fragmentation, causing a 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 could 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.

The question arises how to treat storage devices like this in a benchmark. You get best performance and reproducible results by formatting before the benchmark, but this is not very realistic. We study this question by running the test many times after formatting. The JouleSort rules require averaging over at least five runs.

## 3. ALGORITHMICS

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

To be efficient, 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 SortBenchmark 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

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

out the finished blocks is also overlapped with computation.

When looking closer at the algorithm, one notes that there are random blocks accesses only in one place, namely for 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. Fortunately, the only operation that really needs random block access only *reads* data, a discipline where SSDs have a particular advantage.

# 4. IMPLEMENTATION

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

Our programs support full overlapping of I/O and computation, in both phases using the techniques available in STXXL [7]. Due to the limited speed of the Atom processors, it was important to further tune the internal sorts and merges.

- In memory, we sort only the keys, associated with an index, and permute the sequence of full records later.
- Sorting the keys is done by applying parallelized least significant digit radix sort to the four most significant bytes of the key and then sorting the partially sorted sequence by insertion sort.
- The libstdc++ parallel mode, based on the Multi-Core Standard Template Library (MCSTL) [18], exploits the parallelism on the two cores and four threads provided by the processor. In particular, we use the parallelized function call `multiway_merge`.[5]
- The parallel multiway merger reaches best performance if a single invocation can output at least $B$ records. Having 3 blocks per run in internal memory (2 for the merger to guarantee $\geq B$ available elements 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 only use 2 blocks per run and combine blocks that will run empty soon.

As a consequence, the disks are fully loaded at about 80% 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 Respectively

For the $10^8$ and the $10^9$ record category, we have developed the program *EcoSort* using the parallel version of the Standard template library for XXL data sets (STXXL) [4]. The STXXL provides two sorter implementations, `stxxl::sort()` that writes the output back to the location of the input, and `stxxl::stream::sort()` for pipelined processing that 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 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.
- Allow to use a different degree of parallelization in the two phases.

## 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, as mentioned before, we have to sort *in-place*. Also, all disks have to be configured as a RAID-0. With other configurations, we could not have explored the full bandwidth, but would have had to use file striping and fill one disk after the other with the output.

We used a variant of *DEMSort* [15, 14], because of its capability to sort *in-place*. It was augmented with the features mentioned at the beginning of this section. 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.

Secondly, 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, so 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 leave more space for the write buffer.

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

This configuration is very tight. Enlarging the input size, e. g. by a factor of 2, would require either more internal memory (hardware not capable), 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 by DEMSort. 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 that have already been read, can immediately be returned to the file system 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 could counter this by doing write combining, but nevertheless, transferring a *logically* non-contiguous range will also incur overheads.

To improve this situation that suffers from small blocks, we replaced the block I/O by *range I/O*, i. e. 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, which should amortize all overheads. Also, this approach does not contradict the external memory model.

# 5. EXPERIMENTS

## 5.1 System Details

The Zotac IONITX-A mainboard was shipped with a wireless module installed. This module was removed and other unused hardware was disabled in the BIOS if possible. The machine has no keyboard or monitor attached, but is connected to the LAN instead 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 loaded.

As operating system, we used the Debian *sid* distribution with a 2.6.30 Linux kernel. There was only a minimal set of services running, 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 Linux Logical Volume Manager. 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.

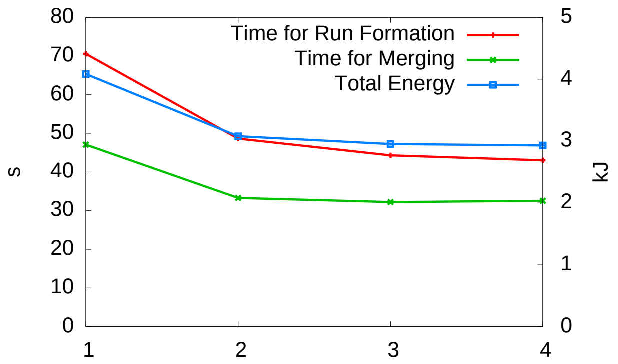

**Figure 1: Evaluating the optimal number of threads.**

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` for the program run. 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 show a peak performance of about 1 000 MiB/s read and 600 MiB/s write for the RAID.

In each category, the respective program was run five times. Before each run, the (old) output file and all temporary files were 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 could use up to 3.25 GiB of RAM for their operation. The run 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 12) resulted in better overlapping and lower running time without affecting the merging phase.

For categories $10^8$ and $10^9$, we used block sizes of 26 214 400 and 13 107 200 bytes, respectively.

For for the TB scale using DEMSort, we lowered the block size to 1 433 600 bytes, as explained before, and newly generated the input before each run due to the lack of space, after the RAID was cleaned of all files. This 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 1, 4 threads for run formation and 3 threads for merging have the best running times. The energy consumption goes along these lines, so we chose these values for the $10^8$ and $10^9$ record input. For $10^{10}$ records, we used 4 threads also for merging because the high merge degree is more compute-intensive.

The input for all runs was generated by the *gensort* program provided by the SortBenchmark jury, running in ASCII mode.

## 5.4 Categorization

We submit our results for the Indy category because our programs can only handle fixed-size records, and we had to sort the 1 TB input in-place due to space constraints. Also, some optimizations rely on a certain key universe.

## 5.5 Results

Here, we state our results for the Indy JouleSort category.

We sorted $10^8$ records in 76.7 s, consuming 2 821 J of energy, i. e. 36.8 W on average. This is equivalent to 35 453 records per Joule.

We sorted $10^9$ records in 756 s, consuming 27.49 kJ of energy, i. e. 36.4 W on average. This is equivalent to 36 381 records per Joule.

These two results improve the current records [16, 19] (11 600, 11 300 and 11 600 records per Joule) by a factor of more than 3. The absolute running times are also better than for the competitors. The achieved bandwidth is around 600 MiB/s when averaging over reading and writing, equivalent to about 75% peak performance.

For the 1 TB scale, the resulting energy efficiency is only about a third, compared to the smaller inputs. In addition, we noticed a degradation in performance after the first run on freshly reset disks (see Figure 2). Thus, we ran tests for other input sizes, in the gap between 100 GB and 1 TB, using the same parameter values. This showed that the degradation is correlated to the filling degree. The higher complexity of the merge step can be ruled out to have that much influence. For the 10 GB and 100 GB inputs, there was no degradation, even when a multiple of the disk capacity was written in total. The performance degradation is due to increasing fragmentation, as explained in Section 2.1. The I/O rate drops dramatically after the first run formation, when the merger has to fill the gaps, namely from 632 MiB/s to 144 MiB/s (averaged over read and write). However, after the first iteration (writing 3 TB in total), the behavior looks stable, the disks do not degrade more. Thus, we exclude the first run and state the average of five runs on already degraded disks as our result.

We sorted the TB in 21 906 s, consuming 724 kJ of energy, i. e. 33.0 W on average. This is equivalent to 13 818 records per Joule.

Still, we beat the previous record [16] (3 425 records per Joule) by a factor of 4. 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 4 times as long.

We had even worse results for the I/O bandwidth degradation when generating the input using regular OS-buffered I/O. Instead, we wrote 400 MB blocks to the RAID by piping the gensort output through `dd oflag=direct obs=409600000`.

We consider the 1 TB result an artifact of the space scarceness in our system. Using twice as large disks would probably provide much better performance.

The results for JouleSort are summarized in Table 3.

# 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, `http://www.eecs.berkeley.edu/~ychen2/cs262a/EnergyEfficientMapReduceReport-Final.pdf`, 2008.

[7] 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.

[8] Boi Feddern. Entdeckungsreise. *c't: magazin für computer technik*, 11/2009:100–104.

[9] Boi Feddern. Platten-Karussel. *c't: magazin für computer technik*, 10/2009:104–107.

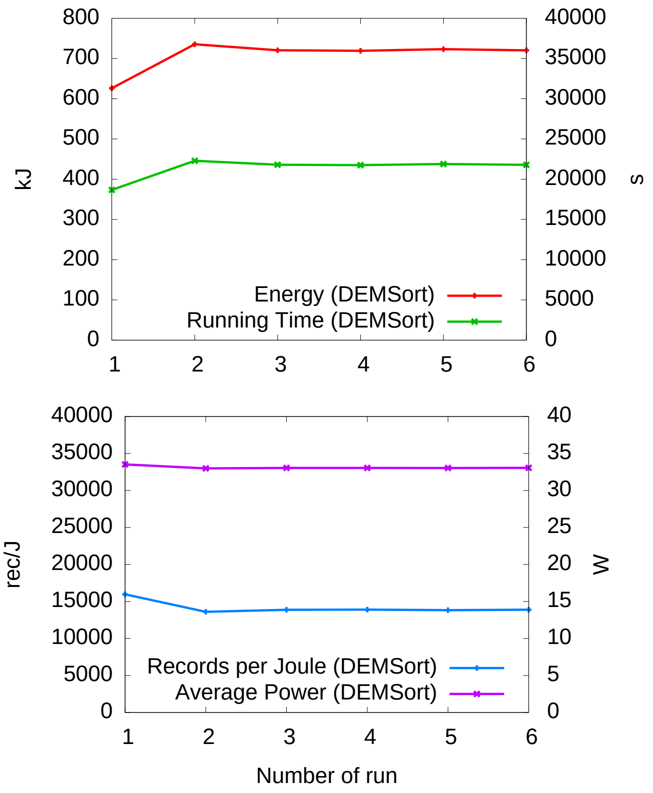

**Figure 2: Degradation of the 1 TB sorting results, for six consecutive runs after formatting.**

| **JouleSort Category (records)** | **$10^8$** | **$10^9$** | **$10^{10}$** |
|---|---:|---:|---:|
| Program | EcoSort | EcoSort | DEMSort |
| Data Volume | 10 GB | 100 GB | 1 TB |
| Number of records | 100 000 000 | 1 000 000 000 | 10 000 005 120 |
| Checksum | 2faf0ab746e89a8 | 1dcd615efb9dfe11 | 12a06d707e22a96eb |
| Time | 76.7±0.64 s | 756±7.8 s | 21 906±220 s |
| **Energy** | **2 821±25 J** | **27.49±0.3 kJ** | **723.7±7.3 kJ** |
| Average Power | 36.8±0.11 W | 36.4±0.13 W | 33.0±0.03 W |
| **Records per Joule** | **35 453±313** | **36 381±399** | **13 818±139** |
| Typical Bandwidth Run Formation | 530 MiB/s avg | 600 MiB/s avg | 280 MiB/s avg<br>460 MiB/s read<br>290 MiB/s write |
| Typical Bandwidth Merge | 640 MiB/s avg | 610 MiB/s avg | 130 MiB/s avg<br>160 MiB/s read<br>110 MiB/s write |

**Table 3: Summary of the SortBenchmark results.**

[10] Boi Feddern. Platten-Karussel. *c't: magazin für computer technik*, 21/2009:142–145.

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

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

[13] 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.

[14] Mirko Rahn, Peter Sanders, and Johannes Singler. Scalable distributed-memory external sorting. *CoRR*, abs/0910.2582, 2009.

[15] Mirko Rahn, Peter Sanders, Johannes Singler, and Tim Kieritz. DEMSort — distributed external memory sort. `http://sortbenchmark.org/demsort.pdf`, 2009.

[16] 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.

[17] 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.

[18] 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.

[19] Ranja Sinha and Nikolas Askitis. Ozsort: Sorting 100 GB for less than 87 kJoules, 2009. `http://sortbenchmark.org/OzJoule2009.pdf`.