Towards an Energy Aware DBMS – Energy Consumptions of Sorting and Join Algorithms

Hagen Höpfner
International University in Germany
Campus 3; 76646 Bruchsal; Germany
hoepfner@acm.org

Christian Bunse
International University in Germany
Campus 3; 76646 Bruchsal; Germany
chistian.bunse@i-u.de

Abstract

Database management systems comprise various algorithms for efficiently retrieving and managing data. Typically, algorithm efficiency or performance is correlated with execution speed. However, the uptime of battery-powered mobile- and embedded systems strongly depends on the energy consumption of the involved components. This paper reports our results concerning the energy consumption of different implementations of sorting and join algorithms. We demonstrate that high performance algorithms often require more energy than slower ones. Furthermore, we show that dynamically exchanging algorithms at runtime results in a better throughput.

1 Introduction and Motivation

Database management systems (DBMS) are software systems that are in widespread use for managing data independent of applications and underlying hardware. Hence, application developers can utilize DBMS in order to efficiently store and retrieve data without knowing their exact implementation (aka. black-box view). DBMS provide various access strategies (indexes) and retrieval algorithms for handling and manipulating data. These are typically optimized regarding their execution performance. Most existing DBMS are designed to run on desktop computers and powerful servers. However, due to the recent developments in the area of embedded and mobile devices, the need for lightweight and mobile DBMS appeared. In contrast to other (more powerful) platforms such devices are often battery powered. Therefore, their uptime strongly depends on the efficient usage of the resource energy. Whereas energy optimization regarding hardware (sleep mode, speed reduction, etc.) is state of the art, only little research has been performed regarding the energy consumption of software. In this paper we present first ideas on how to minimize the energy consumption of DBMS. Since, DBMS are complex software systems, we focus on the energy consumption of basic algorithms such as sort and join operations.

The remainder of the paper is structured as follows: Section 2 introduces the examined algorithms. Section 3 briefly presents the evaluation environment and discusses the underlying measurement theory. Section 4 discusses the results obtained in various empirical studies. Section 5 contains first ideas on how to adapt algorithm usage to reduce energy consumptions. Section 6 concludes the paper and gives an outlook on future research. Due to the strict space limitations we decided not to include a related work section into this paper but refer to [2].

2 Sorting and Join Algorithms

A first step towards the development of lightweight and energy-efficient DBMS is the analysis of the basic or ”foundation level” algorithms regarding their energy consumption. This paper presents first, preliminary results for sort and join algorithms. In the following we briefly describe these algorithms based on [5] (sort) and [3] (join):

Bubblesort belongs to the family of comparison sorting. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. Bubblesort has a worst-case complexity of $O(n^2)$ and best case of $O(n)$. Its memory complexity is $O(1)$.

Heapsort is comparison-based and part of the Selectionsort family. In practice it is slower on most machines than an efficient implementation of Quicksort, but it has the advantage of a worst-case complexity of $O(n \log n)$.

Insertionsort is a “naive” sorting algorithm belonging to the family of comparison sorting, too. It has a worst case complexity of $O(n^2)$, but it is known to be efficient on substantially sorted data sets. Its average complexity is $O(n^2/4)$ and $O(n)$ in the best case. As an in-place algorithm it requires a constant amount of memory space.

Mergesort belongs to the family of comparison-based sorting. It has an average and worst case complexity of $O(n \log n)$. Mergesort requires three times the memory of in-place algorithms such as Insertionsort.

Quicksort [4] belongs to the family of exchange sorting. On average, Quicksort makes $O(n \log n)$ comparisons to sort n items, but in its worst case it requires $O(n^2)$ comparisons. Typically, Quicksort is regarded as one of the most efficient algorithms. Its memory usage depends on factors such as choosing the right Pivot-Element. On average, having a recursion depth of $O(\log n)$, the memory complexity of Quicksort is $O(\log n)$ as well.

Selectionsort belongs to the family of in-place comparison sorting. It typically searches for the minimum value, exchanges it with the value in the first position and repeats the first two steps for the remaining list. On average Selectionsort has a $O(n^2)$ complexity that makes it inefficient on large lists. Selectionsort typically outperforms Bubblesort but is generally outperformed by Insertionsort.

Shakersort [1] is a variant of Shellsort that compares each adjacent pair of items in a list in turn, swapping them if necessary, and alternately passes through the list from the beginning to the end then from the end to the beginning. It stops when a pass does not executes any swaps. Its complexity is $O(n^2)$ for arbitrary data, but approaches $O(n)$ if the list is substantially sorted at the beginning.

Shellsort is a generalization of Insertionsort. The algorithm belongs to the family of in-place sorting but is regarded to be unstable. The algorithm performs $O(n^2)$ comparisons and exchanges in the worst case, but can be improved to $O(n \log_2 n)$. This is worse than the optimal comparison sorts, which are $O(n \log n)$. Shellsort improves Insertionsort by comparing elements separated by a gap of several positions. This lets an element take "bigger steps" toward its expected position. Multiple passes over the data are taken with smaller and smaller gap sizes. The last step of Shell sort is a plain Insertionsort, but by then, the array of data almost sorted.

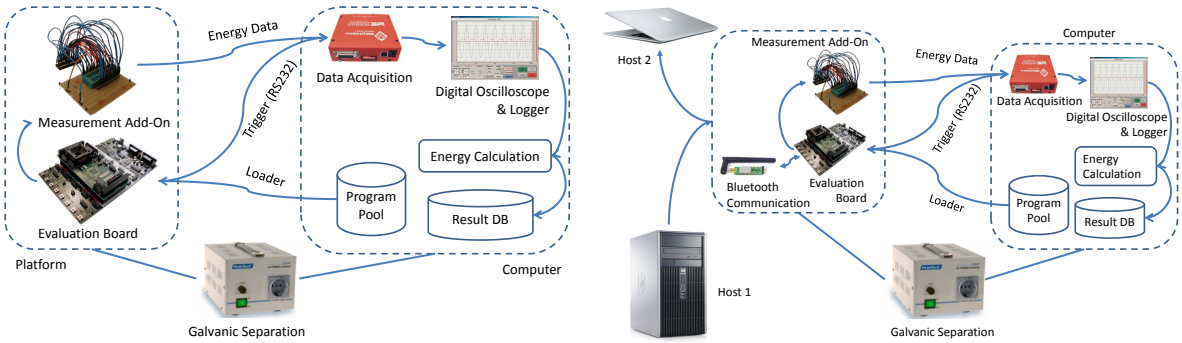
Nested-Loop-Join (NLJ) compares all tuples of one relation R to all tuples of the other relation S by using a nested loop. The complexity is $O(|R| \cdot |S|)$, where $|R|$ and $|S|$ represent the number of tuples per relation.

Sort-Merge-Join (SMJ) requires that both relations must be sorted on the joint attributes. If they are not physically or implicitly (index) sorted, an external sorting is performed first. The merge part then scans the input relations alternately using the monotony of sorted relations. In the worst case (all tuple have the same join attribute value), the complexity is $O(|R| \cdot |S|)$. In the best case only the smaller relation must be scanned ($O(\min(|R|, |S|))$.

Hash-Join (HJ) uses a hash function to find matching tuples. In a partitioning phase, all records of the smaller relation are hashed using the join attributes. In the subsequent probing phase the hash values for the tuples of the other relation are calculated. Hence, join candidates are hashed to the same bucket. In the worst case where only one hash bucket exists, the efficiency is equal to the Nested-Loop-Join. In the best case, assuming constant access time for values of the hash table, the complexity is $O(|R| + |S|)$.

3 Measurement Environment

The researched algorithms vary in their memory and CPU (e.g., no. of cycles) usage. In order to explicitly measure the energy consumption of single algorithm executions, we used a specific evaluation platform (Figure 1). In detail, the algorithms where executed on an AVR micro controller (i.e., ATMega 128 based on a STK501 board). Every program (algorithm) sent a TTL level signal [6] at the start and end of its run in order to trigger measurement and logging by a digital oscilloscope. The collected data was externally processed in order to calculate the consumed energy values and stored.



(a) without communication interface (ES_1)

(b) including communication via Bluetooth (ES_2)

Figure 1: Experimental Setup

Unfortunately, energy consumption cannot be directly measured since it is a function over time. However, using a sense resistor with a known value $R_{SENSE} = 100\Omega$ and a stabilized voltage $U = 4.93V$, we could calculate the needed energy or Joule value as follows: We measured the voltage drop U_{SENSE} at the sense resistor embedded

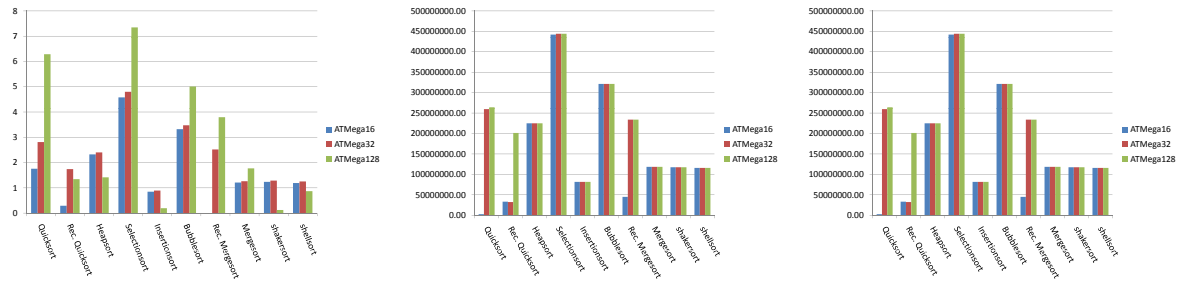
into an add-on board[1] and then calculated the current power consumed by the processor core. Using Kirchhoff's, Ohm's and some mathematical laws the resulting formula is $E = \frac{1}{100\Omega}\Delta t \cdot \sum_{n=0}^{\frac{t}{\Delta t}} 4.93V \cdot U_{SENSE}(n \cdot \Delta t) - U_{SENSE}^2(n \cdot \Delta t)$, whereby n represents the number of collected samples per second and $\Delta t = \frac{1s}{n}$ the interval for one sample. For details on this calculation we refer to [2].

Figure 1(a) illustrates the measurement setup without any network communications. Especially mobile devices very often use wireless networks to communicate with an information service provider. Therefore, we decided to extend the basic environment by a Bluetooth interface (see Figure 1(b)). The embedded node now wirelessly receives data-sets, processes them and sends the sorted set to another recipient.

4 Experimental Results

This section discusses the results of different experiments to investigate the energy consumption of sorting and join algorithms. All figures in this section use accumulated, non-normalized values. The following results are sums of measurement results of random, sorted and reverse-sorted data and are accumulated for 1,000 cycles.

We first examined whether software energy consumption, as widely believed, is strongly correlated to software performance. Therefore, we executed the algorithms on different processors of the AVR processor family, whereby the consumed energy (Figure 2(a)), the execution time (Figure 2(b)) as well as the number of cycles (Figure 2(c)) were measured. The latter two were obtained by using the AVR simulator of AVRStudio. "Empty" bars represent missing results. It was, e.g., not possible to use the recursive Mergesort for sorting 1,000 integer values on the ATMega16 since the system ran out of memory.



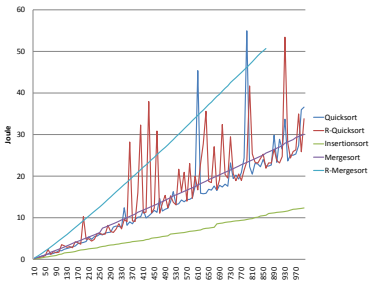
(a) Energy Consumption in Joule (b) Execution Time in ms (c) Used CPU Cycles

Figure 2: Experimental Results (sorting)

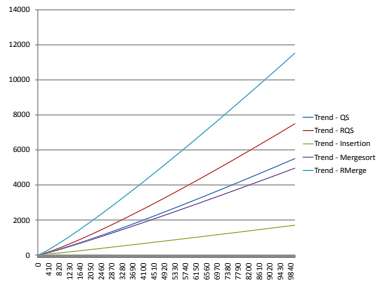
Initially, the results regarding the energy consumption of different sorting algorithms reveal that, in contrast to the initial assumption, sorting algorithms such as Insertionsort ($O(n^2)$) require significant less energy than high-performance algorithms such as Quicksort ($O(n \log n)$). The results show that this observation is valid and visible across different processors of the same family. Furthermore, energy consumption grows with the inbuilt flash memory size (15 KByte, 32 KByte, and 128 KByte). Interestingly, recursive Quicksort outperforms Insertionsort when running on the ATMega16 but is clearly behind at all other platforms. The reason is that the figure represents accumulated values (random, sorted, reverse sorted), and recursive Quicksort failed (out of memory) to sort the sorted and reverse sorted data. Therefore, the presented value is much lower.

The 2nd experiment series (500 execution cycles) had two goals: First, to measure the energy consumption of sorting algorithms for growing data sizes. Second, to evaluate the impact of using external memory on energy consumption. Therefore, we concentrated on Quicksort (standard and refined), Mergesort (standard and refined), and Insertionsort with respect to random, sorted and reverse sorted data and for increasing lengths of data (0 to 1,000 elements). The obtained measurement results (Figure 3(a)) in general confirm the results of the previous experiment series by showing that Insertionsort consumes significantly less energy than other algorithms, although it is slower. In fact, sorting 1,000 randomized elements with Insertionsort took 71.3 ms, whereas Quicksort needed 8.8 ms. Regarding the second goal, we extended the microprocessor's inbuilt SRAM memory from 4 to 132 KByte using an external memory chip. While comparing Figure 2 and Figure 3 it becomes obvious that using external memory requires significantly more energy. For Insertionsort, e.g., the energy consumption for sorting 1,000 random elements raised from 0.03 to 4.11 Joule. The difference cannot be explained by the standard energy the additional chip requires since the differences between both curves strongly diverge with a growing data size. We assume that this is caused by moving data to/from external memory and addressing/managing these additional memory cells.

[1] used to ease the change of processors, provide measurement points, etc.



(a) Comparison of Sums (rand, sort, rev)



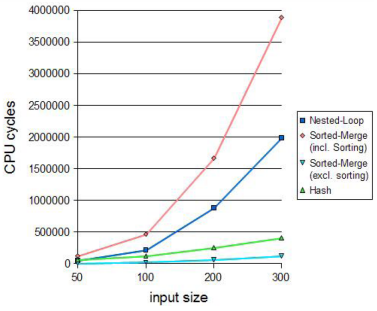
(b) Trends

Algorithm	trend function
Quicksort	$F(n) = n^{1.1388} \cdot 0.1533$
RQuicksort	$F(n) = n^{1.1771} \cdot 0.1467$
Insertionsort	$F(n) = n^{1.0679} \cdot 0.09121467$
Mergesort	$F(n) = n^{1.1035} \cdot 0.1912$
RMergesort	$F(n) = n^{1.1384} \cdot 0.3221$
Heapsort	$F(n) = n \cdot 0.2324 + 0.0286$
Shellsort	$F(n) = n^2 \cdot 0.0071 + n \cdot 0.0047 + 0.0939$
Selectionsort	$F(n) = n^2 \cdot 0.013 - n \cdot 0.0236 + 0.1908$

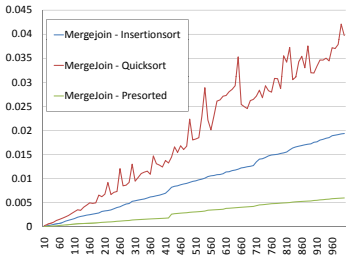
(c) Trend functions

Figure 3: Experimental Results – 2nd Measurement Series

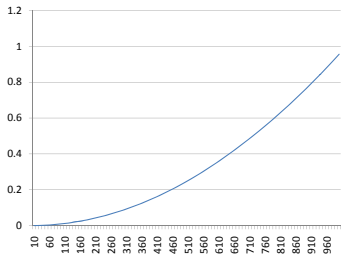
The differences between algorithms become clearer by watching the interpolated trend functions in Figures 3(b) and 3(c). Here, n is the number of processed data items and the R^2 value that represents the goodness of fit was 1.



(a) Various Join Implementations



(b) Merge-Join



(c) Nested-Loop-Join

Figure 4: Energy consumption comparison – normalized to one execution

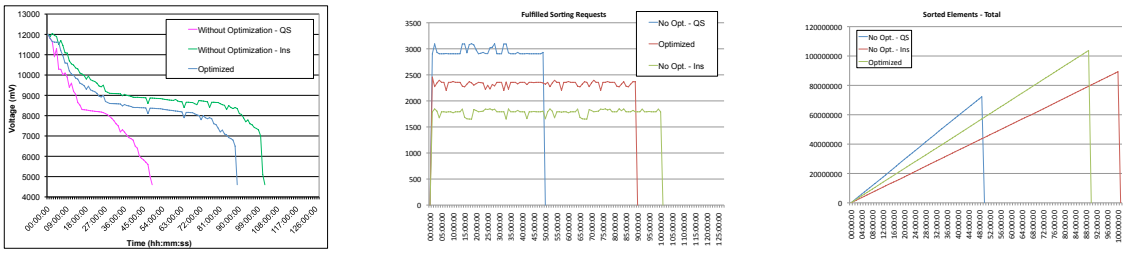
The final experimental run investigated the energy consumption of standard join operations. In detail, the joins were executed on the Atmel ATMega128 MCU. Each time two integer arrays were joined, whereas the join condition was the equality of both attributes. Both arrays contained random numbers between 0 and 100, generated with C's `rand()` function. As the generated number varied and such the execution of the different algorithms, the performance measurement was perform several times. The number of execution cycles each algorithm required to finish its task was measured. Note that SMJ used Insertionsort. Figure 4(a) shows results for array sizes of 50, 100, 200 and 300. For a small input size NLJ is most efficient, but the complexity rapidly increases as input gets bigger. By contrast, the number of HJ cycles takes grow only almost linearly, making it five times faster for input size of 300. The SMJ algorithms has two facets: for presorted data it is by far the fastest algorithm of the three, regardless of the input size. If the data has to be sorted first, the performance of the overall operation is highly slowed down by the sorting algorithm. Hence, HJ seems to be the most efficient join algorithm for unsorted input data.

From our sorting algorithm experiments we learned, that memory usage is important for energy consumptions. Whereas NLJ and SMJ (depending on the sorting algorithm used) need almost no additional memory space, HJ uses a considerable amount of memory to store the hash table. Thus, it is justifiable to assume that if considering the energy consumption instead of the performance, the outcome will be appreciably different. In order to determine the amount of energy in Joule, we measured the SMJ algorithm using Insertionsort and Quicksort. The results in Figure 4(b) conform to our previous finding that Insertionsort consumes much less energy in comparison to Quicksort. The energy consumption of a NLJ see Figure 4(c) grows quadratic. Thus, although NLJ require less execution cycles than MJ, its energy consumption is significantly higher. This supports our assumption that the energy consumption related to the execution of a specific algorithm mostly depends on its memory requirements and that the algorithms complexity class (i.e. number of execution cycles) plays only a minor role.

5 Optimized Algorithm Usage

The optimization of algorithmic energy-consumption requires a model (i.e., cost model) that can be used to predict the cost (i.e., *J* or energy) for executing a specific algorithm on a specific input set. Based on the measurement results concerning the energy consumption of sorting algorithms we therefore extrapolated trend functions that

calculate an estimation of the required energy for 1,000 executions [2] of an algorithm, based on the input size n. Since our goal was to find the optimal balance between sorting performance and energy consumption it is not sufficient to solely use the trend functions as cost function. Due to the nearly linear nature of the trend function the result would always indicate Insertionsort as the most energy-efficient algorithm but neglect algorithmic complexity and performance. Therefore we applied the following strategy: 1) By using the size n of the set as an input the energy-related costs for all algorithms are calculated and stored. 2) The minimum result and thus the most energy-efficient algorithm is identified. 3) Based on the algorithmic complexity, the minimum value is compared to those values that are related to algorithms of "lower" complexity classes. 4) If the difference between the energy requests is below a predefined threshold or delta the "faster" algorithm is chosen.



(a) Battery Lifetime (b) Request Performance (c) Total Number of Sorted Elements

Figure 5: Adaptive Algorithm Selection

The measurements are based on the networked experimental setup (cf. Section 3). A look at the battery level V over time supports the initial assumption that the uptime of a systems is directly correlates with the energy consumption related to the executed software system. However, a closer look at Figure 5(a) shows that a non-adaptive approach (i.e., using a fixed algorithm) either results in an excellent or a poor energy efficiency. Interestingly, the results for the adaptive version are close to those of the non-optimized Insertionsort variant. Figure 5(b) supports the initial assumption concerning the trade-off between energy efficiency and performance. Fast variants like Quicksort handle more sorting requests in a shorter period of time but result in a very limited V. Energy-efficient variants like Insertionsort result in an optimal V but handle significantly less sorting requests. Only adaptive systems provide a good balance of energy-efficiency and performance. This is also supported by Figure 5(c) that shows the total number of elements that were sorted over time.

6 Summary, Conclusions and Outlook

We presented first results towards realizing energy aware database management systems. We concentrated on sorting and join algorithms as those are essential for query processing. We introduced our measurement setup and discussed first, preliminary results. We highlighted that memory intensive implementations consume more energy then CPU intensive ones. Interestingly, we found out that energy consumption is not solely correlated with the complexity class (performance) of an algorithm. Furthermore, we discussed first ideas on how to optimize the algorithm usage based on trend functions reflecting the energy consumption of certain algorithm implementations.

The next step on our agenda is to replicate the experiments with other platforms (i.e., PIC, ARM, and PSoC) in order to get a more generalized "cost model". Furthermore, we plan to examine other DBMS algorithms for query processing (set operations, projection, selection, etc.) and for indexing data (B-Tree, tries, etc.). Afterwards, we plan to define an overall optimization strategy based on the user requirements (i.e., fast results vs. long up-time).

References

[1] B. Brejová. Analyzing variants of Shellsort. *Information Processing Letters*, 79(5):223–227, 2001.
[2] C. Bunse, H. Höpfner, E. Mansour, and S. Roychoudhury. Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments. In *ROSOC-M 2009 Proceedings*, 2009. forthcoming.
[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 5th edition, 2006.
[4] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.
[5] R. Lafore. *Data Structures and Algorithms in Java*. SAMS Publishing, Indianapolis, Indiana, USA, 2nd edition, 2002.
[6] D. E. Lancaster. *TTL Cookbook*. Sams, May 1974.

[2] trend functions are build upon measurements for 1,000 execution cycles