# CHOOSING THE ”BEST” SORTING ALGORITHM FOR OPTIMAL ENERGY CONSUMPTION

Christian Bunse, Hagen Höpfner, Suman Roychoudhury and Essam Mansour
*International University in Germany*
*Campus 3, 76646 Bruchsal, Germany*

Keywords: Energy awareness, Software engineering, Adaptivity, Mobile information systems.

Abstract: Reducing the energy consumption of mobile systems in order to prolong their operating time has been an active research topic for quite some time. Such systems are typically battery powered and thus, their uptime depends on the energy consumption of the used hardware and software components. Novel strategies that allow software systems to dynamically adapt themselves at runtime can be effectively used to reduce energy consumption. The focus of this paper is based on a case study that uses an energy management component that can dynamically choose the “best” sorting algorithm during a multi-party mobile communication. The results indicate that Insertionsort is the most optimal sorting algorithm when in comes to saving energy.

## 1 INTRODUCTION

Mobile and embedded devices become more and more important in all areas of the daily life. Nowadays, they also form the basis of the ubiquitous or pervasive computing paradigm. Information is accessible everywhere at any time via mobile phones, small sensors collectively measure environmental parameters, and micro controller based computers are embedded in many devices starting from toys and ending with cars or air planes. In fact, the landscape of embedded computing changes dramatically. Unfortunately, all these small components share a need for a (mobile) power supply. As smaller the device, as more the uptime depends of the efficient usage of the limited energy resource. Hence, all parts of a mobile or embedded system have to be energy aware.

Recently signifcant research effort has been spent on optimizing hardware related energy consumption. Only littlesearch has been done regarding the energy efficient usage of hardware components by optimizing the underlying software. The aspect of software optimization is addressed in this paper. Devices contain several hardware components (e.g. CPU, external memory, communication devices), which have different levels of energy consumption. Therefore, the software must be able to adapt itself to meet the underlying user requirements while conserving maximum energy. In previous works (Höpfner and Bunse, 2007) we therefore introduced the concept of resource substitution. Preliminary results (Bunse et al., 2009) have shown that different implementations of algorithms result in varying energy consumptions. In particular, we implemented various sorting algorithms because sorting efficiency is relevant to almost all applications. Furthermore, many database management algorithms that implement join (e.g. Sort-Merge-Join) or set operations implicitly use sorting algorithms. Our experiments revealed that memory intensive implementations consumed much more energy than in-place implementations. However, if processing speed is given priority over energy, Insertionsort performs slower than Quicksort, thus resulting in a longer execution time. Therefore the question is, how much data can be sorted by an algorithm implementation by keeping an optimal balance between energy consumption and processing speed.

In this paper we present our approach for energy saving software by choosing the appropriate algorithm. Based on experimental results we introduce trend functions for each implementation of the examined sorting algorithms. These trend functions are then used to decide on which algorithm to use under certain conditions or based on the users needs (faster speed vs. saving energy). Furthermore, we describe a dynamic optimization approach that changes the used implementation at runtime.

The remainder of this paper is structured as follows: Section 2 describes the related work. Section 3 briefly introduces the researched sorting algorithms. Section 4 introduces the optimizer component and the trend functions. Section 5 explains the experimental



Bunse C., Höpfner H., Roychoudhury S. and Mansour E. (2009).
CHOOSING THE "BEST" SORTING ALGORITHM FOR OPTIMAL ENERGY CONSUMPTION.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 199-206
DOI: 10.5220/0002245401990206

setup and the experiments performed as proof of concept. Section 6 includes the interpretation of the experimental results. Section 7 discusses the validity of these results. Section 8 summarizes and concludes the paper and gives an outlook to future research.

## 2 RELATED WORK

Due to the orientation towards mobile- and embedded systems, several research projects have been conducted regarding the topic of energy consumption. Optimizing energy consumption is one of the most fundamental factors for an efficient battery-powered system. Research on energy consumption falls into one of the following categories: (1) Hardware, or (2) Software (Jain et al., 2005). Research that belongs to the hardware category, attempts to optimize the energy consumption by investigating hardware usage, such as (Chen and Thiele, 2008; Liveris et al., 2008), and innovating new hardware devices and techniques, such as (Tuan et al., 2006; Wang et al., 2006). Research in second category attempts to understand how the different methods and techniques of software affect energy consumption. Research in this category can be further classified according to the main factors affecting energy consumption: *networking*, *communication*, *application nature*, *memory management*, and *algorithms*. Concerning *networking* work such as (Feeney, 2001; Senouci and Naimi, 2005), provide new routing techniques that are aware of energy consumption. Other efforts of this category focus on providing energy-aware protocols for transmitting data in wireless networks (Seddik-Ghaleb et al., 2006; Singh and Singh, 2002) and ad-hoc networks (Gurun et al., 2006). Memory consumption is an important factor concerning a system's energy consumption. In this regard work such as (Koc et al., 2006; Ozturk and Kandemir, 2005) have provided energy-aware memory management techniques. In battery-powered systems, it is not sufficient to analyze algorithms based only on time and space complexity. Energy-aware algorithms such as (Jain et al., 2005) supporting randomness, (Potlapally et al., 2006) focusing on cryptographic, and (Sun et al., 2008) investigating into wireless sensor networks were published.

## 3 SORTING ALGORITHMS

In the first days of computing, sorting data (numbers, names, etc.) was in focus of research. One reason might be that although sorting appears to be "easy", its efficient execution by machines is inherently complex. Even today, sorting algorithms are still being optimized or even newly invented. When it comes to mobile systems and information retrieval, efficient sorting is a major concern regarding performance and energy consumption. In the following we describe the set of sorting algorithms that were used in the context of this study. This set was defined to include major algorithms that are either used in form of library functions (e.g., Quicksort), are easily programmable (e.g., Bubblesort) or that are regularly taught to IT students. In other words, our goal was to cover those algorithms that are in widespread use. More details on them can be found in standard text books on algorithms and data structures such as (Lafore, 2002).

- **Bubblesort** belongs to the family of comparison-based sorting. It works by repeatedly iterating through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The worst-case complexity is $O(n^2)$[1] and the best case is $O(n)$. Its memory complexity is $O(n)$.
- **Heapsort** is a comparison-based sorting algorithm and part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quicksort, it has the advantage of a worst-case time complexity of $O(n \log n)$.
- **Insertionsort** is a naive algorithm that belongs to the family of comparison-based sorting. In general insertion sort has a time complexity of $O(n^2)$ but is known to be efficient on data sets which are already substantially sorted. Its average complexity is $O(n^2/4)$ and linear ($O(n)$) in the best case. Furthermore insertion sort is an in-place algorithm that requires a linear amount $O(n)$ of memory space.
- **Mergesort** was invented by John von Neumann and belongs to the family of comparison-based sorting. Mergesort has an average and worst-case performance of $O(n \log n)$. Unfortunately, Mergesort requires three times the memory of in-place algorithms such as Insertionsort.
- **Quicksort** was developed by Sir Charles Antony Richard Hoare (Hoare, 1962). It 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 and is therefore typ-

[1] $n$ represents the size of input; the number of elements to be sorted

ically used for all sorting tasks. Quicksort's memory usage depends on factors such as choosing the right Pivot-element, etc. 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-based 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 bubble sort but is generally outperformed by Insertionsort.
- **Shakersort** (Brejová, 2001) 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 no swaps. Its complexity is $O(n^2)$ for arbitrary data, but approaches $O(n)$ if the list is nearly in order at the beginning.
- **Shellsort** is a generalization of Insertionsort, named after its inventor, Donald Shell. It belongs to the family of in-place sorting but is regarded to be unstable. It 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-based 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 Shellsort is a plain Insertionsort, but by then, the list of data is guaranteed to be almost sorted.

# 4 OPTIMIZING ENERGY

To dynamically adapt/optimize the energy consumption of a mobile and/or embedded system, we developed an architecture (see Figure 1) that closely follows the idea of the energy management component presented in (Bunse and Höpfner, 2008). EMC aimed at optimizing the communication effort of a system. Here the focus is on using the "optimal" algorithm concerning energy and processing speed.

At its core the experiment system defines a family of sorting strategies (algorithms), encapsulates each of these, and makes them interchangeable. This is nicely supported by the strategy pattern defined by

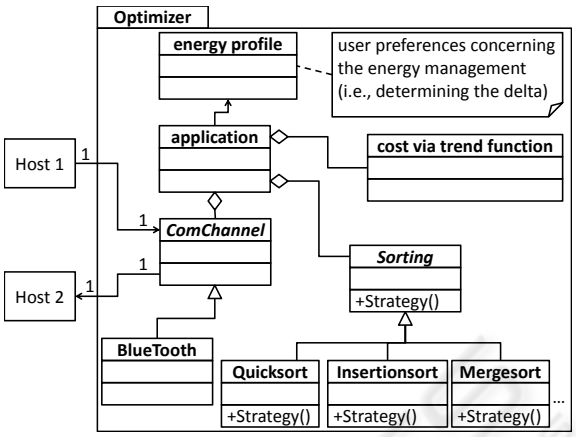


Figure 1: Algorithm-Energy Optimizer.

(Gamma et al., 1995). The strategy pattern supports the development of software systems as a loosely coupled collection of inter-changeable parts. The pattern decouples a strategy from its host, and encapsulates it into a separate class. It thus, supports the separation of an object and its behaviour by organizing them into two different classes. This allows switching to the strategy that is needed at a specific time.

There are several advantages of applying the strategy pattern for an adaptable software system. First, since the system has to choose the most appropriate strategy concerning performance and energy it is simpler to keep track of them by implementing each strategy by a separate class instead of embedding it in the body of a method. Having separate classes allows simply adding, removing, or changing any of the strategies. Second, the use of the strategy pattern also provides an alternative to sub-classing an object. This also avoids the static behavior of sub-classing. Changes therefore require the creation/instantiation of a different subclass and replacing that object with it. The strategy pattern allows switching the object's strategy, and it will immediately change how it behaves. Third, using the strategy pattern also eliminates the need for various conditional statements. When different strategies are implemented within one class, it is difficult to choose among them without resorting according to the conditional statements. The strategy pattern improves this situation since strategies are encapsulated as an object that is interchangeable at runtime.

To select the most appropriate strategy/algorithm, the optimizer has to be aware of the cost of its execution with respect to energy. Therefore, a cost model is needed that provides a "rough" estimate of an algorithms energy consumption based on the input size. However, it has to be noted that the used cost model instance is only valid for the actually used platform.

Basically we followed the empirical data gathered in the context of (Bunse et al., 2009) concerning the energy consumption of sorting algorithms running on AVR processors.

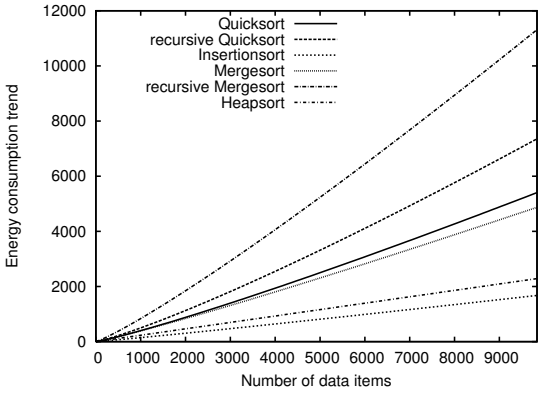


Figure 2: Trend functions - excerpt.

We used the extrapolated trend functions of the different sorting algorithms (see Figure 2) as a basis for the cost functions. The trend function calculates an estimation of the required energy for 1,000 executions[2] of an algorithm, based on the input size $n$. In detail, the trend functions listed in Table 1 were extrapolated, whereby the $R^2$ value that represents the goodness of fit was 1.

Table 1: Trend functions.

| 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 | $F(n) = n^2 \cdot 0.013 - n \cdot 0.0236 + 0.1908$ |

Since our goal was to find the optimal balance between sorting performance and energy consumption it was not sufficient to simply use the trend functions as cost function. Due to the linear nature of the trend function the result would always indicate Insertionsort as the most energy-efficient algorithm. Keeping these formulas and assumptions in mind the following selection algorithm can be defined:

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 in energy-consumption is below a predefined threshold or delta the "faster" algorithm is chosen.

The goal of an adaptive application (e.g., our experimental system) is to optimize the Quality-of-Service (QoS) perceived by the user. Unfortunately, often optimization approaches either enforce predetermined (fixed) policies or offer only limited mechanisms for controlling optimization. According to (Sousa et al., 2008) those limitations prevent adaptive systems from addressing these important issues. User goals often entail trade-offs among different aspects of quality (e.g., enhancing battery life or faster execution times).

The Optimizer architecture (see Figure 1) allows users to actually determine the trade-off between performance and energy consumption, simply by changing the cost function delta. As larger the delta becomes as higher the sorting performance and as lower the battery-lifetime of the system. Thus, the delta determines the size of data-sets to be sorted by a specific algorithm. In the context of our experiments we experimented with different delta values and observed that defining the delta as $1,200$ provides the "best" optimization results.

# 5 EXPERIMENTAL DESIGN

Our previous experiments provided some insight into the area of software-related energy consumption. In these experiments we collected data concerning the energy consumption of sorting algorithms as well as algorithms that apply them. The results show that energy consumption is driven by factors such as memory consumption and performance leading to the fact that the fastest algorithm (e.g., Quicksort) is not the most efficient algorithm concerning energy-consumption.

In this sense we developed a simple system (see Figure 3). An embedded node wirelessly receives data-sets, sorts them and sends the sorted set to another recipient. The goal of the node is to primarily sort data. However, since the node is battery powered and the end-user expects short response times, sorting is optimized according to response time and average energy consumption (i.e., maximize up-time).

[2] to level out measurement errors

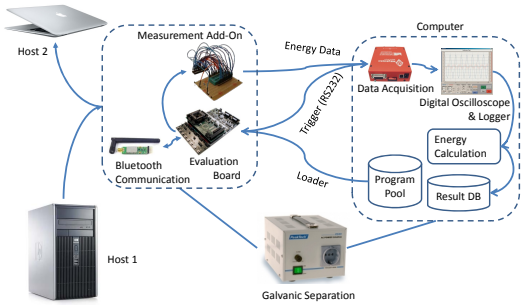


Figure 3: Experiment System Overview.

On application scenario for such a node is a wireless sensor network that collects position or life-data of cattle. Notes are communicating wirelessly (e.g., ZigBee or Bluetooth), whereby the controller-node (i.e., the node that pre-processes the data) finally provides the pre-processed data to a PC. In the context of our experiments we use a simplified version for brevity of illustration.

The system comprises a micro controller (i.e., ATMega128, external SRAM, running on a STK500/501 board) and two Bluetooth interfaces (i.e., BlueSmirf modules). The system establishes two data-connections to different hosts via its Bluetooth modules. It reads values (variable size sets of unsorted integers) via on data-connections, selects the most appropriate sorting algorithm according to its optimization goal, and transmits the sorted set to another host via the second data-connection.

### 5.1 Experimental Runs

Within this experiment we compared three different approaches to optimization concerning their performance (time and size) and their energy consumption. In the first run the data was sorted by only using the Quicksort algorithm, Thus, this run was optimized for speed. In the second run,the data was sorted by only using the Insertionsort algorithm. This represents an optimization for energy-efficiency (max. battery lifetime) based on our previously reported findings(Bunse et al., 2009). In the third run we then made use of the algorithm optimizer/selector (see Figure 1). The Optimizer aims at balancing speed (performance) and energy efficiency to increase the amount of sorted data and processing speed while at the same time increasing battery lifetime through energy savings.

### 5.2 Data Collection

As described previously the experiment systems is connected to two different hosts via Bluetooth (i.e., Figure 3 provides an overview). It receives sets (i.e., sequences of random integer values) of varying sizes from the first host. The actual size of a transmitted set is limited to 1,000 elements but is randomly chosen.

According to the actual experimental run, the system either applies a specific sorting algorithm (i.e., Quicksort or Insertionsort) or selects the “best” sorting algorithm from a set of algorithms and applies the selected algorithm to the received set. The sorted set is then sent to Host 2 and the next data set is received. This cycle is executed until the battery is empty.

Send and receive (Host 1 and Host 2) are synchronized by the clock. This leaves sufficient time for sorting and retransmission. In addition the Bluetooth modules (externally powered) provide send/receive buffers to level out overlaps.

During the experimental runs the following data was collected:

- The size of every set to be sorted (i.e., $n$). The size was randomly chosen but limited to a maximum of 1,000 elements.
- The number of sets sorted (i.e., $N$). This is the overall number of successfully executed sorting requests throughout the experimental run.
- The battery level (i.e., $V$). $V$ was measured at fixed time intervals, whereby we assume that the actual voltage level of a battery indicates its status and charge level.
- Run and Execution Time (i.e., $P$). Run and execution time was measured by hosts one and two (i.e., time when a set was sent and when the sorted set was received). Times were not measured at the target platform in order to not falsify the measurements.

## 6 EVALUATION OF RESULTS

Initial measurement within the experimental runs shows that optimization results are as expected. When looking at the battery level $V$ over time (i.e., up-time of our system) it supports the initial assumption that the uptime of a systems is directly related towards the energy consumption related to the executed software system. However, a closer look at Figure 4 shows that a non-adaptive approach either results in an excellent or a poor energy efficiency. Interestingly, the results for the adaptive are close to those of the non-optimized Insertionsort variant.

When recalling the results of Figure 4 the question arises why should we adapt the system or is optimization really necessary. It seems that by choosing a specific algorithm better results can be achieved than by

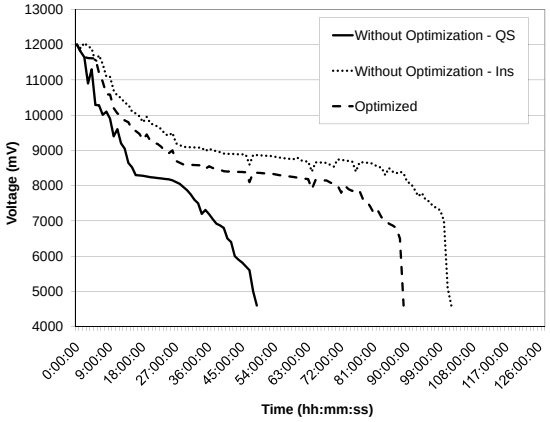


Figure 4: Battery Lifetime.

dynamic adaption. Therefore, we have to examine the performance of the different variants concerning the amount of sorting requests and the amount of data.

Figure 5 supports the initial assumption concerning the trade-off between energy efficiency and performance. High-performing variants (i.e., Quicksort) handle more sorting requests (i.e., $N$) in a shorter period of time but result in a very limited $V$. Energy-efficient variants (i.e., Insertionsort) result in an optimal $V$ but handle significantly less sorting requests. Only adaptive (i.e., optimized) systems provide a good balance of energy-efficiency and performance.

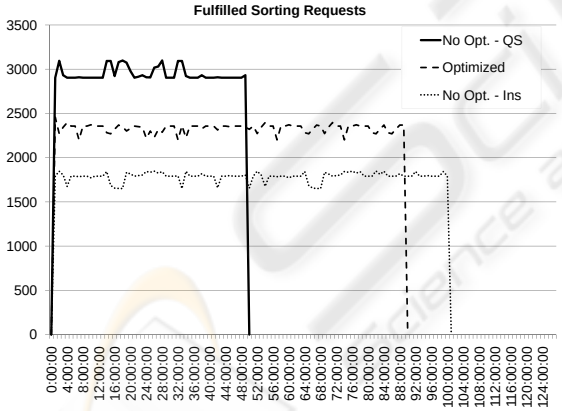


Figure 5: Request Performance.

This is also supported by Figure 6, which shows the total number of elements that were sorted over time. Measurement results expose a linear growth that roughly represent the sums of sorting requests sizes. The results confirm our initial assumption that an optimization for speed (Quicksort variant) results in a fast growing curve that covers a short time period. Optimization for energy (Insertionsort variant) results in a curve that spans a broad time range but that does not grow as fast as the Quicksort related curve. Finally, the optimized system seems to combines the advantages of the other approaches. It covers a broad time range and sorts a high number of elements. In others words, the optimized system variant provides a well-balanced behavior regarding performance and energy consumption.

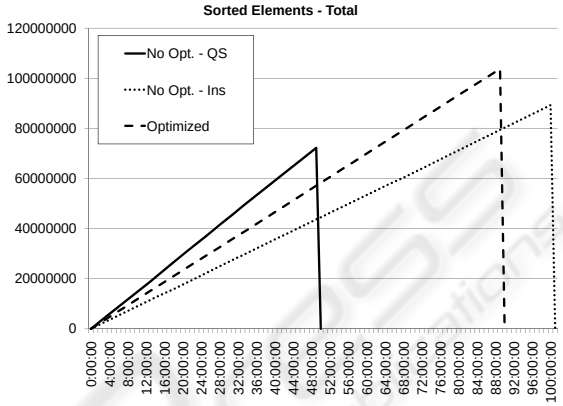


Figure 6: Total Number of Sorted Elements.

The findings concerning the effects of dynamically choosing an algorithm at runtime are also supported by the fact that this approach sorts more data in total than the other two approaches. Thus, optimization does not provide results somewhere between those of the non-adaptive systems but uses their synergy effects to provide even better results.

# 7 THREATS TO VALIDITY

The authors view this study as exploratory. Thus, threats limit generalization of this research, but do not prevent the results from being used in further studies.

**Construct Validity** is the degree to which the independent and dependent variables accurately measure the concepts they purport to measure. In specific, energy consumption is a difficult concept to measure. In the context of this paper it is argued that the chosen approach (assessing the battery voltage level $V$) is an intuitively reasonable measure. Of course, there are several other dimensions of the energy measurement problem but this is future work.

**Internal Validity** is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variable. In specific, a history effect may result from measuring systems at different times (varying context temperature). Additional experiments and runs have shown that the temperature effect in a heated lab room can be neglected.

**External Validity** is the degree to which the results of the research can be generalized to the population under study and other research settings. In specific, the materials (platforms, software, etc.) may not be representative in terms of size and complexity. However, experiments in a university context do not allow the use of realistic systems for reasons such as cost, availability, etc. However, the authors view this study as exploratory and view its results as indicators that allow further research.

In order to improve the empirical study and address some of the threats to validity identified above, the following actions can be taken:

- Improve data collection. Data collection can be improved in several ways. First, by measuring not only battery voltage as an indirect energy consumption indicator but also the exact energy consumption, in joule, of the platform regarding each algorithmic run. Second, by increasing the sampling/measurement frequency (e.g., have sampling rates of 1 microsecond or below). A third option would be to automate the whole experimental procedure, thereby making time collection trivial.
- Improve the distinction of algorithmic complexities. The actual experiment used random data that was only comparable between runs. However, we did not made any distinction regarding best-, worst- and average-case data. By separating and explicitly distinguishing between these cases would allow for fine-grained analysis.
- Improve the generalizability of results by running the experiment on different platforms. Currently, results are limited to the AVR processor family and can thus, only serve as an indicator of the general situation. Therefore the experiment needs to be replicated on different platforms to get more and more reliable data.

# 8 SUMMARY AND CONCLUSIONS

Given the rising importance of mobile and small embedded devices, energy consumption becomes increasingly important. Currents estimates by EUROSTATS predict that in 2020 10-35 percent (depending on which devices are taken into account) of the global energy consumption is consumed by computers and that this value will likely rise (Bunse and Höpfner, 2008). Therefore, means have to be found to reduce the energy consumption of such devices.

The focus of this paper is on dynamically adapting a simple system at runtime by algorithm substitution as a means for energy saving. Following the ideas of resource substitution strategies as presented in (Höpfner and Bunse, 2007) we presented an Optimizer Component that follows the ideas of the dynamic energy management component EMC (Bunse and Höpfner, 2008) and that can be plugged into other component based systems.

At its core the optimizer uses the energy-related trend functions of different sorting algorithms. In detail, the optimizer uses the different trend functions for determining the energy-related cost of a specific algorithm with respect to the algorithm's input-size. It then compares the results and uses a user-defined delta for selecting the best algorithm.

Our initial results are based on a micro-controller system (AVR processor family) that communicate wirelessly by BlueTooth. The main system functionality is to receive data of varying size, sort it and to send it to another host. The data collected for different system variants was then used to examine if energy-consumption and sorting performance can be significantly improved. The collected data reveals that by optimization the amount of sorted data, battery lifetime. Moreover, the overall performance can be significantly increased. The experiments show the impact of software onto a systems energy consumption and a way to easily optimize a system in this regard.

To systematically evaluate the observed effects and to rule out the aforementioned threats to validity we currently prepare a case study for mobile information systems running on a PDA or Smartphone.

# REFERENCES

Brejová, B. (2001). 'Analyzing variants of Shellsort', *Information Processing Letters* 79(5), 223–227.

Bunse, C. and Höpfner, H. (2008). Resource substitution with components — Optimizing Energy Consumption, *in* J. Cordeiro, B. Shishkov, A. K. Ranchordas and M. Helfert, eds, 'Proceedings of the 3rd International Conference on Software and Data Technologie', Vol. SE/GSDCA/MUSE, INSTICC, INSTICC press, Setúbal, Portugal, pp. 28–35.

Bunse, C., Höpfner, H., Mansour, E. and Roychoudhury, S. (2009). Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments, *in* 'Proceedings of the MDM Workshop ROSOC-M 2009'. ( accepted for publication, forthcoming.

Chen, J.-J. and Thiele, L. (2008). Expected system energy consumption minimization in leakage-aware DVS systems, *in* 'ISLPED '08: Proceeding of the thirteenth international symposium on Low power elec-

tronics and design', ACM, New York, NY, USA, pp. 315–320.

Feeney, L. M. (2001). 'An Energy Consumption Model for Performance Analysis of Routing Protocols for Mobile Ad Hoc Networks', *Mobile Networks and Applications* 6(3), 239–249.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

Gurun, S., Nagpurkar, P. and Zhao, B. Y. (2006) Energy consumption and conservation in mobile peer-to-peer systems, *in* 'MobiShare '06: Proceedings of the 1st international workshop on Decentralized resource sharing in mobile computing and networking', ACM, New York, NY, USA, pp. 18–23.

Hoare, C. A. R. (1962). 'Quicksort', *Computer Journal* 5(1), 10–15.

Höpfner, H. and Bunse, C. (2007). Resource Substitution for the Realization of Mobile Information Systems, *in* J. Filipe, M. Helfert and B. Shishkov, eds, 'Proceedings of the 2nd International Conference on Software and Data Technologie', Vol. Software Engineering, INSTICC, INSTICC press, Setúbal, Portugal, pp. 283–289.

Jain, R., Molnar, D. and Ramzan, Z. (2005). Towards understanding algorithmic factors affecting energy consumption: switching complexity, randomness, and preliminary experiments, *in* 'Workshop on Discrete Algothrithms and Methods for MOBILE Computing and Communications — Proceedings of the 2005 joint workshop on Foundations of mobile computing', ACM, New York, NY, USA, pp. 70–79.

Koc, H., Ozturk, O., Kandemir, M., Narayanan, S. H. K. and Ercanli, E. (2006). Minimizing energy consumption of banked memories using data recomputation, *in* 'ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design', ACM, New York, NY, USA, pp. 358–362.

Lafore, R. (2002). *Data Structures and Algorithms in Java*, 2nd edn, SAMS Publishing, Indianapolis, Indiana, USA.

Liveris, N., Zhou, H. and Banerjee, P. (2008). A dynamic-programming algorithm for reducing the energy consumption of pipelined system-level streaming applications, *in* 'ASP-DAC '08: Proceedings of the 2008 conference on Asia and South Pacific design automation', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 42–48.

Ozturk, O. and Kandemir, M. (2005). Nonuniform Banking for Reducing Memory Energy Consumption, *in* 'DATE '05: Proceedings of the conference on Design, Automation and Test in Europe', IEEE Computer Society, Washington, DC, USA, pp. 814–819.

Potlapally, N. R., Ravi, S., Raghunathan, A. and Jha, N. K. (2006). 'A Study of the Energy Consumption Characteristics of Cryptographic Algorithms and Security Protocols', *IEEE Transactions on Mobile Computing* 5(2), 128–143.

Seddik-Ghaleb, A., Ghamri-Doudane, Y. and Senouci, S.-M. (2006). A performance study of TCP variants in terms of energy consumption and average goodput within a static ad hoc environment, *in* 'IWCMC '06: Proceedings of the 2006 international conference on Wireless communications and mobile computing', ACM, New York, NY, USA, pp. 503–508.

Senouci, S.-M. and Naimi, M. (2005) New routing for balanced energy consumption in mobile ad hoc networks, *in* 'PE-WASUN '05: Proceedings of the 2nd ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks', ACM, New York, NY, USA, pp. 238–241.

Singh, H. and Singh, S. (2002) 'Energy consumption of tcp reno, newreno, and sack in multi-hop wireless networks', *ACM SIGMETRICS Performance Evaluation Review* 30(1), 206–216.

Sousa, J. P., Balan, R. K., Poladian, V., Garalan, D. and Satyanarayanan, M. (2008). User guidance of resource-adaptive systems, *in* 'Proceedings of the 3rd International Conference on Software and Data Technologie', Vol. Software Engineering, INSTICC, INSTICC press, Setúbal, Portugal, pp. 36–45.

Sun, B., Gao, S.-X., Chi, R. and Huang, F. (2008). Algorithms for balancing energy consumption in wireless sensor networks, *in* 'FOWANC '08: Proceeding of the 1st ACM international workshop on Foundations of wireless ad hoc and sensor networking and computing', ACM, New York, NY, USA, pp. 53–60.

Tuan, T., Kao, S., Rahman, A., Das, S. and Trimberger, S. (2006) A 90nm low-power FPGA for battery-powered applications, *in* 'FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays', ACM, New York, NY, USA, pp. 3–11.

Wang, L., French, M., Davoodi, A. and Agarwal, D. (2006) 'FPGA dynamic power minimization through placement and routing constraints', *EURASIP Journal on Embedded Systems* 2006(1).