Software Energy-Efficiency with Sweet Spot Frequencies

Sebastian Götz*, Thomas Ilsche*, Jorge Cardoso*†, Josef Spillner*
* Technische Universität Dresden
Faculty of Computer Science
01062 Dresden, Germany
Email: {sebastian.goetz1, thomas.ilsche, josef.spillner}@tu-dresden.de
† University of Coimbra
Department of Informatics Engineering
3030-320 Coimbra, Portugal
Email: jcardoso@dei.uc.pt

***Abstract*—A common misconception is to equate software energy-efficiency to CPU performance. The rationale of this fallacy is that increasing CPU clock frequency involves a reduction of CPU usage in time and, hence, energy consumption. In this paper, we give empirical evidence for scenarios where a server is more energy-efficient when its CPU(s) operate(s) at a lower frequency than the maximum allowed frequency. Our approach uses a novel high-precision, fine-grained energy measurement infrastructure to investigate the energy (joules) consumed by three different sorting algorithms. Our experiments show the existence of algorithm sweet spots: CPU clock frequencies at which algorithms achieve the lowest energy consumption to complete the same computational task. To leverage these findings, we describe how a new kind of self-adaptive software applications can be engineered to increase their energy-efficiency.**

I. Introduction

In 2010, electricity used in global data centers likely accounted for between 1.1% and 1.5% of total electricity use. For the US that number was between 1.7 and 2.2%. Data center traffic is expected to quadruple by 2016. This calls for the development of new energy and power efficient approaches to reduce their consumption [1].

To address this increasing concern, the concept of energy-proportional computing [2] was introduced by Google to describe an ideal system which consumes no energy when idle and whose power consumption grows linearly with utilization. Such a computing system would enable a greater efficiency at any level of utilization compared to today's systems. To calculate power efficiency, utilization is divided by its corresponding power value. Thus, the formula to evaluate power efficiency is given in [2] by:

$$\eta_P = \frac{Util}{P} \tag{1}$$

η_P is the power efficiency, $Util$ the utilization, and P the power consumed by a computing system (server). Such an ideal system would always yield the same optimal power efficiency since P grows proportionally with $Util$. As an example, a utilization of 25% would require only 25% of computing power.Utilization can be considered the application performance normalized to the maximum possible performance of the system.

This suggests that when $Util$ increases, P should be increased to maintain a high efficiency or, for a running system, P can be decreased to force $Util$ to increase. This would yield a better η_P. In this paper, we explore how the interplay of P and $Util$ affects energy-efficiency, and shed light on the fallacy that increasing the P (CPU clock frequency) always involves a reduction of energy consumption. We will show that some computational tasks are more energy-efficient when executed at lower CPU clock frequencies.

To better understand this claim, it is adequate to study energy-efficiency instead of power efficiency, as done in [2]. Since energy E is a function of power P and time, the energy-efficiency η_E can be written as:

$$\eta_E \updownarrow = \frac{Util \downarrow}{P \downarrow \times t \uparrow} \tag{2}$$

Since Eq. 2 accounts for the time a system will be under a certain utilization, it provides a more realistic model compared to Eq. 1. Specifically, the equation suggests the hypothesis that by decreasing the CPU clock frequency, and, thus, its P ($\downarrow$) and $Util$ ($\downarrow$), the time t ($\uparrow$) to complete a computational task increases. The goal of this paper is to study for which configurations of P, $Util$, and t, computational tasks are executed in a more energy-efficient way, thus in which direction they affect η_E ($\updownarrow$).

Fig. 1 shows the reason why the model presented in [2] needs to be relaxed. It is already known from previous academic and industrial research that in practice, processors do not follow a proportional path. Single processors have power states and associated frequencies for which the power efficiency, i.e., the ratio between utilization and power consumption is maximized in so-called *sweet spots* [3] and often minimized in high-performance *turbo mode* [4], [5]. In the figure, the sweet spot is represented by the minimum of the function divided by x, denoted as maximum efficiency. There is always an offset, even when unused through *idle power*. Furthermore, in multi-processor systems, additional overlap

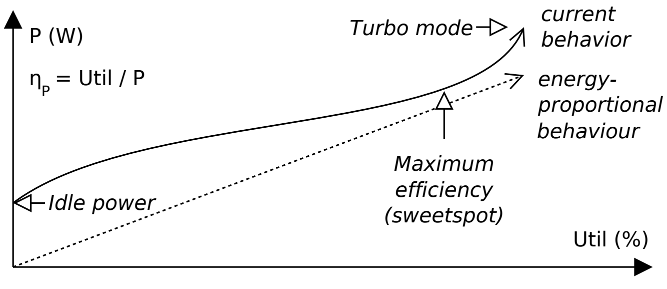


Fig. 1. Energy-proportional and current, non-linear power consumption.

effects result from using the turbo mode as a gap-filler before switching on the next core when the utilization increases [6]. This effect, in particular the sweet spot, translate into time-dependent energy-efficiency due to Dynamic Voltage Frequency Scaling (DVFS; a commonly used power-management technique).

The importance and implications of this effect has triggered preliminary research on the design of energy-efficient software. Proposals suggested to control the power states from the application to select the most efficient frequency. In [7], the authors control the CPU frequency of a laptop while running specific applications (e.g., video encoders, web browsers, and word processors) to reduce energy consumption. In [8], similar experiments are done with message passing interface programs running in high-performance computing systems. In both studies, the notion of sweet spots was not known since the experiments where done at a macro level, measuring only consumption at the "plug". While Livingston et al. [3] mention sweet spots in their work, they did not fully explore their sources, characteristics, and implications.

Therefore, a conclusive analysis on the design of self-adaptive software applications which select the algorithms to execute depending on the existence and characteristics of sweet spots is missing. This sets our work apart from previous approaches by providing an important contribution to foster research on the development of *energy-efficient software* as a complement to current hardware level energy optimizations.

This paper is structured as follows. Sect. II briefly describes our motivation, and the importance and timeliness of this work. Sect. III enumerates three central research questions that are addressed throughout this paper. Sect. IV describes our approach and the methodology we have followed to experimentally analyze the energy-efficiency of computing systems. We discuss the results of our experiments in Sect. V and show in Sect. VI, how these findings can be used, to build self-adaptive software, able to leverage this knowledge to save energy. Finally, Sect. VII and VIII present related work and our conclusions, respectively.

II. Motivation

Google estimates[1] that it requires 1 kJ of energy per search. This corresponds to 0.2 grams of CO_2. One thousand keyword searches have approximately the same ecological footprint as driving a car for one kilometer. Furthermore, as stated in the introduction, existing data centers consume approximately between 1.1% and 1.5% of all energy used in the world.

[1]http://googleblog.blogspot.de/2009/01/powering-google-search.html

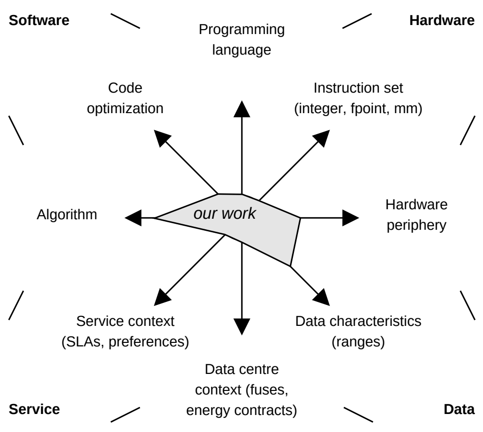

Software
Programming language
Hardware
Code optimization
Instruction set (integer, fpoint, mm)
Algorithm
our work
Hardware periphery
Service context (SLAs, preferences)
Data characteristics (ranges)
Data centre context (fuses, energy contracts)
Service
Data

Fig. 2. Dimensions and directions of energy-efficient software research.

The magnitude of these numbers have driven researchers and industries to look into new ways to make information and communication technologies more energy-efficient. The solutions found include high-efficiency power supplies, water cooled servers, efficient multi-core CPUs, virtualization, dynamic power management, and live virtual machine migration. For example, the consolidation of virtual machines through live migration enables to aggregate work onto fewer server nodes and shutdown idle nodes to reduce power consumption [9].

While these solutions are important and complementary (focusing on hardware and computing environments), another form of energy reduction is to use more energy-efficient software. The concept of energy-efficient software is to "use less energy to achieve the same computational task". Compared to, e.g., live virtual machine migration, which is an energy conservation technique (migrating and turning off idle servers), the energy-efficiency of software looks into which software to use to execute a specific computational task.

Our long-term research goal is to study how energy-efficient mechanisms can be implementation as part of self-adaptive software and service systems that change their behavior and implementation, and affect the computing environment to reduce energy consumption. Fig. 2 shows relevant dimensions of research in this field. The short-term goal for this paper is to answer three research questions while looking at a computational task implemented in three different ways. Later we extend the considerations to complex data centre setups with trade-offs between energy, performance, and dependability (as, e.g., in [10]).

III. Research Questions

The energy-efficiency of software looks into how software, the underlying computing system and the environment affect energy consumption. Our research questions (RQ) are the following:

- RQ1 (Measurement Setup). How to instrument a computing system (server) with measurement devices to obtain fine-grained measurements for its individual parts (e.g., fan, disk, power supply, and CPU sockets)? (Sect. IV-A).
- RQ2 (Sweet spots). How can sweet spot frequencies be identified? Which mathematical functions characterize them? Do sweet spot frequencies still exist on newer computer architectures? (Sect. V).
- RQ3 (Dynamic Software Adaptation). How to capitalize on the existence of sweet spots to dynamically adapt software to achieve a higher energy-efficiency? (Sect. VI).

In this paper, we focus on studying how different implementations of the same software application affect differently the energy-efficiency of a computing system. We take the computational task of sorting n numbers and explore how different implementations of sorting algorithms consume different amounts of energy.

While much research has looked into how to make information and communication technologies more energy-efficient, it is rather hard to find a precise definition for *software energy-efficiency*. Thus, to remove any possible ambiguity on the results of our research, we define the concept as follows:

Definition 1 (Software energy-efficiency)**.** *Energy is defined as the amount of joules, required by a full or partial computing environment, to execute a software application. A software application S_1 is said to be more energy-efficient than an application S_2, if it requires less energy to accomplish the same computational task.*

Definition 2 (Computing environment)**.** *A full computing environment includes all the devices that, directly or indirectly, consume energy to enable a software application to be executed. For example, it typically includes CPUs, fans, and disks. A partial computing environment only includes a subset of those devices.*

To simplify our study, we do not directly set a certain utility, but focus on different CPU frequencies, which indirectly affects the utility. For a single fixed task that executes without interruptions, the utility is the inverse of the runtime so that Eq. 2 can be reduced to to Eq. 3.

$$\eta_E = \frac{1/t}{(P \times t)} = \frac{1}{(P \times t^2)} \tag{3}$$

IV. Approach and Methodology

Computational complexity (i.e., big O notation) is often a first step in assessing the performance of an algorithm. However, in practice, the best big O algorithm may perform worse due to large constant factors or practical memory constraints. Sorting is such an example, where quicksort is often used as default even though it does not have the best big O (worst case) performance. It is common practice to optimize implementations for run-time and, in most cases, optimizations will also reduce energy consumption. However, in recent hardware with increasing energy-efficiency features, such as DVFS, the fastest algorithm and system setting is often no longer the most energy-efficient one.

Our approach to gain insights is pragmatic and experimental. We use energy as a main optimization goal and vary the algorithm and hardware configuration for comparison. To limit the search space, we do not investigate specific hardware micro-optimizations, but use generic compiler optimization flags instead. The methodology has the following activities:

- Measurement environment (Sect. IV-A).
  - Instrument server with energy sensors.
  - Determine static power consumption of the server.
  - Setup software infrastructure to conduct the experiments.
- Software under test (Sect. IV-B).
  - Select the computational task to be tested experimentally.
  - Select different software implementations for the task.
- Experimental results analysis (Sect. V).
  - Determine resources affected by task.
  - Interpret measurement results.
- Generalization and application of the results (Sect. VI).

A. Measurement Setup: Energy Monitoring

1) Hardware: The system under test is a dual socket system with Intel Xeon E5-2690 processors. Several layers of power measurement instrumentation are required. The complete AC input is measured with a calibrated ZES Zimmer LMG450 power analyzer. Several custom-built, shunt-based sensors are added to the system. All sensors are pluggable via Molex connectors used in many standardized systems. For this paper, we monitor the 12 V input of the two individual sockets separately. They supply power for the CPUs and their attached memory. The voltage drop over the measurement shunt is amplified with calibrated amplifiers and digitally captured by a National Instruments PCI-6255 data acquisition board with 7541 samples per second. The power consumption is computed digitally from individual readings for current and voltage. During the experiment, all data processing happens on a separate system to avoid perturbation of the system under test. This comprehensive measurement infrastructure serves as an answer to RQ1, but requires significant effort, as described by Hackenberg et al. [11].

The processors provide 15 different frequencies from 1.2 to 2.9 GHz and the turbo mode with frequencies up to 3.8 GHz, depending on thermal and power budget. Both, frequency and voltage are set uniformly for all cores of a socket by the hardware. As demonstrated in [12], the available memory bandwidth depends on the core frequency for the Sandy Bridge-EP architecture. Earlier intel architectures, such as the one used in [3], provided a constant memory bandwidth independent of the selected frequency. The variable memory bandwidth did not allow for a straight-forward selection of the optimal frequency for applications that become memory-bound at a certain frequency. However, our approach does not

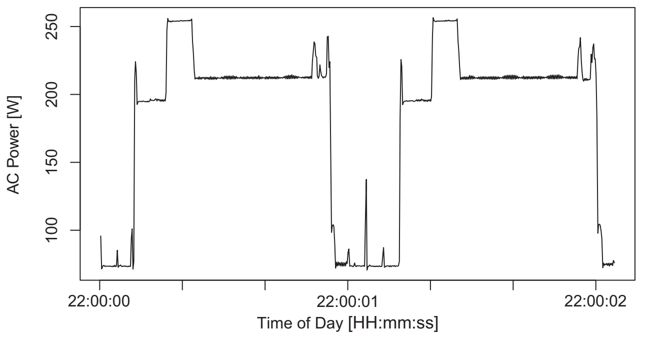


Fig. 3. Energy trace of two succeeding sorting invocations with idle phase.

Power supply & Fans	Board	SSD	Sockets	Total
≈26W	≈7W	≈1W	≈20W × 2	≈74W

TABLE I
STATIC IDLE POWER CONSUMPTION OF THE SERVER.

require specific bottleneck analysis, because it uses the energy measurement results to select among different settings instead.

2) Measurements: To investigate the energy consumption of algorithms, we run different sort algorithms for different input sizes multiple times. Prior to each invocation, we randomly generate integer lists to be sorted. Across all invocations, we used lists of sizes between 10 and 50 million elements, always containing integers with a value range of 6 million (i.e., $0 \leq x < 6 \times 10^6$). Each invocation is preceded by a pause of 1 second to "cool down" the CPU and other resources (i.e., let them switch to idle mode).

Measurements of sort invocations utilizing a single core of a multi-core machine are not representative, due to the static power consumption of other devices besides the CPU, which does not change, regardless of how many cores are used. Hence, we fully utilized all cores of the machine with a separate sort invocation operating on a copy of the same list. By using MPI barriers [13], we ensure that all start invocations are started at the same time. As execution time (or response time), we measured the longest duration of the parallel sort invocations and ensured that variation of durations among parallel processes was less than 5%.

Fig. 3 visualizes the invocation scheduling by AC power consumption (i.e., at the wall) over time for two consecutive sort invocations. The spikes at the beginning and end of each invocation are due to (MPI) synchronization. The short period of 250 W in each run denotes the list generation.

As shown in Table I, the static power consumption of the server originates from the power supply, the fan, the motherboard, the disk, and the sockets.

For the investigation of the effect of different CPU frequencies on the timing and energy-efficiency of sorting, we used the userspace CPU governor of Linux to explicitly set the frequency of the CPU. We executed list generation and sorting for three algorithms with different list sizes for all possible frequencies of the CPU and the turbo mode. We collected the total energy consumption of the server per execution, the energy consumption of the sockets, and the response time. This enables to investigate whether a sweet spot frequency exists and if static power consumption leads to a shift of the sweet spot frequency for sockets, only compared to the whole server.

B. Software Setup: Sorting Algorithm

As mentioned before, data centres and network infrastructures serving millions of users in massive online applications are a key target for energy-efficiency with huge absolute savings, even for small percentages in relative savings. However, these systems are very complex. Due to the novelty of the topic, we suggest to understand a well-known generic algorithm, sorting, which is used in many applications. According to sources cited by Knuth, more than 25 percent of the running time of computers has at some point been spent on sorting [14], which may still be the case with today's databases, graphs, and other large data structures. Other software tasks which are of interest but were not explored in this research include search, indexing, and executing arithmetic operations over large volumes of data.

Our choice of sorting algorithms encompasses the ones generally considered the fastest. In the following, we elaborate in two stable sort implementations – radix sort and counting sort – by closely following textbook descriptions. In addition, we look at the already implemented non-stable sorting of the C++ Standard Library (std::sort). These algorithms can be used to sort lists of integer values (or any data mappable to integers). Depending on the size of the lists, the range of the integer values to sort and the hardware in use, we have produced interesting trade-offs such as that for small lists radix sort is faster, whereas for bigger lists counting sort is faster. However, these findings cannot yet be generalized across hardware architectures. Pure performance comparisons with intersections are therefore omitted from our experiments.

Counting sort takes a list A of size n as input and produces a sorted list C of size n as output using the elements of A. To sort the elements, the $range$ of the elements in A has to be known, because in a first step a list B of size $range$ is created. Every $B[i]$ is set to the number of occurrences of the element i in A (i.e., the frequency of elements in A is counted). B is created based on an address/index computation over B. Counting sort is a stable sorting algorithm with a linear time complexity of $O(n)$.

Radix sort is stable and has linear time complexity, too. It also requires the $range$ of the elements in the input list to be know. But, in contrast to counting sort, radix sort does not create an intermediate list B of size $range$. Instead it works on the individual elements of the $range$. Thus, if the lists only contain numbers, 10 lists are created–one for each digit. Radix sort works in two phases: in the first phase, the elements of A are moved to the intermediate lists (partitioning). In the second phase, the elements are "stacked" into A. The number of iterations is the number of alphanumerical characters of $range$. For example, for a range of $[0..999]$, 3 iterations

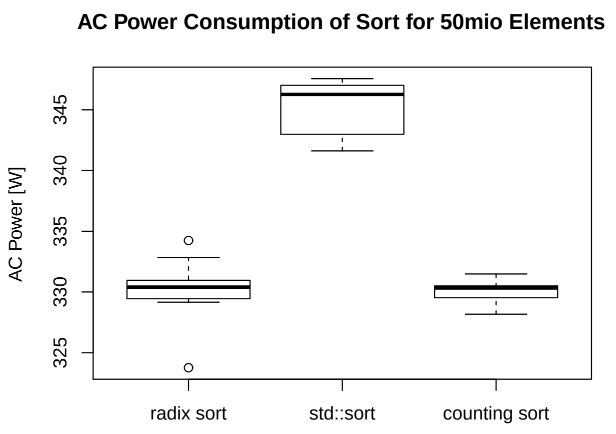


Fig. 4. AC power consumption for sorting 50 million elements.

result. Because of this, the range has a stronger effect on the response time of counting sort compared to radix sort ($time \sim range$ vs $time \sim log_{10}(range)$). Per definition std::sort has a linearithmic complexity over the number of elements in the list. It performs $O(n \times log_2(n))$ comparisons[15]. Our implementation (GNU libstdc++) uses a combination of intro sort and insertion sort.

Based on this common knowledge about the time complexity of sorting algorithms, we investigated the energy consumption of the three algorithms. Since sorting is compute bound, one could assume the CPU to be the predominant consumer of energy amongst all other resources in a server. We collected empirical support for this hypothesis and, in addition, determined further resources consuming energy due to sorting. It is important to know that the frequency of a CPU affects timing, power and, consequently, the energy consumption of algorithms. In general, we will give empirical support for the following claims:

1) The highest frequency of a CPU does not necessarily lead to the lowest energy consumption (power integrated over time).
2) Each algorithm has a detectable frequency at which the resulting energy consumption is lowest (sweet spot frequency).
3) Different algorithms can have different sweet spot frequencies.

V. Results of the Experiments

Before investigating the impact of different frequencies on energy-efficiency in Sect. V-B, we analyze in Sect. V-A whether algorithms differ in their power consumption and show that algorithms with low power consumption are not necessarily the best in terms of energy consumption.

A. On the Power Level of Software

All raw measurements retrieved from the experiments were processed with the statistics software R. The weighted moments, among them the mean value, quartiles, minimum and maximum excluding extreme outliers, are shown in Fig. 4 for

TABLE II
Sorting algorithms characteristic comparison for 50mio elements (in turbo mode).

Algo.	t_{min} (s)	t_{max} (s)	t_{range} (s)	P (W)	E (J)
radix	1.880	1.901	0.021	330.4	626.9
std::sort	4.226	4.230	0.004	346.3	1464.1
count.	8.444	8.516	0.072	330.3	2801.5

a sorting task on 50 million elements where both CPUs of the server operate in turbo mode.

From the figure, one can infer that using radix or counting sort leads to a lower power consumption than std::sort. The former two have an almost equal level. The savings compared to std::sort amount to 4.6%. Yet, radix sort is clearly the fastest algorithm and, hence, the best when combining both metrics without prioritization of one over the other. Counting sort is, despite its low power consumption, the worst in terms of energy consumption. From this so-called *sweet spot perspective*, which will be further elaborated on in the following section, the savings for radix sort amount to 77.6% compared to counting sort. Table II summarizes the key numbers from the experiments: Duration including range, power consumption per time unit and overall energy consumption for the mean duration. All power values include 74 W idle consumption.

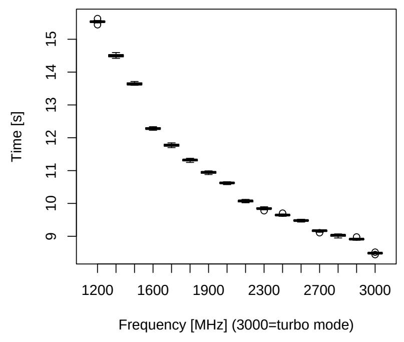
B. On the Sweet spot of Software

In this section, we investigate and prove wrong the common misconception that software energy-efficiency equates directly to CPU performance. For this purpose, we have collected evidence that executing software applications at high CPU frequencies may lead to lower software energy-efficiency. Therefore, we refine research question RQ2 into:

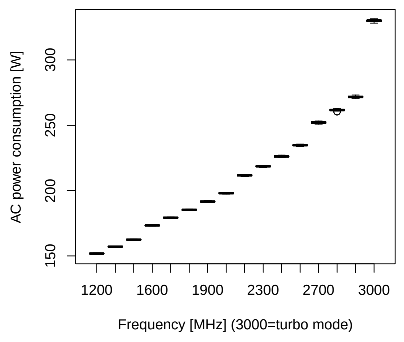
- RQ2a (Effectiveness). Can we increase software energy-efficiency by changing the clock frequency of the CPU executing a software task?
- RQ2b (Determinability). Can the clock frequency of the CPU executing a software task, which makes the software more energy-efficient, be determined?

As outlined in the previous section, we use three different algorithmic implementation of sorting: radix sort, std::sort and counting sort. For each algorithm we measured the energy consumed to sort 10, 20, ..., 50 million integers. Fig. 5 shows the results obtained for sorting 50 million elements using counting sort. Measurement results for all other list sizes and algorithms are shown in Fig. 6. The figure shows three charts: time per frequency, power per frequency, and energy per frequency. Note that turbo mode frequency varies over time, depends on different factors and can be between 2.9 and 3.8 GHz. Very high frequencies are unlikely as we fully use all cores.

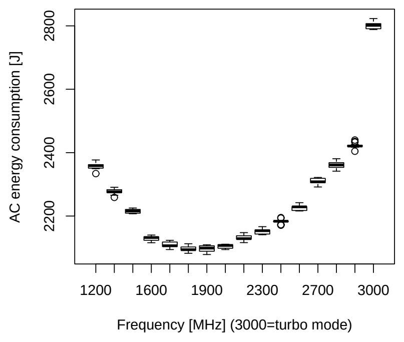
a) Time×Freq: Fig. 6(a) shows that as the clock frequency of the CPU increases from 1.2 GHz to 2.9 GHz and turbo mode, the mean time required to execute the software task of sorting decreases in a non-linear form. What should be noticed in this finding is that at 1.4 GHz the mean time to complete the task drops significantly.



(a) Time by Frequency



(b) Power by Frequency



(c) Energy by Frequency

Fig. 5. The sweet spot of counting sort to sort 50mio elements.

b) Power×Freq: Fig. 6(b) shows the power consumed based on the CPU clock frequency selected. The results could be considered foreseeable, since the power consumption of the CPU increases by its frequency. Nonetheless, up to 2 GHz the power consumption is more modest and has visible increments at 1.5 GHz, 2.2 GHz, 2.7 GHz and for the turbo mode.

c) Energy×Freq: Fig. 6(c) provides the results of our findings that are most striking: the number of joules required to execute the task of sorting has a sweet spot at 1.8 GHz. The energy consumption declines by approximately 25% (708 J) when the CPU frequency is changed from turbo mode to 1.8 GHz.

For the other algorithms, the results are also interesting. std::sort is more energy-efficient at 2.4 GHz. This corresponds to energy savings of approximately 12% (or 176 J) compared to the turbo mode, which leads to the shortest response time. Running the CPU at a low frequency, i.e., 1.2 GHz, increases the energy consumed by 25% (427 J) compared to the sweet spot frequency.

Radix sort is more energy-efficient at 2.2 GHz, which corresponds to energy savings of approximately 15% (or 96 J) compared to the turbo mode and 18% (119 J) compared to the lowest frequency.

Fig. 6 clearly shows the different sweet spot frequencies (vertical dotted line), in ascending order, for counting, radix and std::sort. The lines in the diagram correspond to list sizes of 10..50 million elements.

Table III provides an overview of the results and identifies the sweet spots for each algorithm (sweet spots are marked with a star '*'), and the energy savings that can be achieved when the most energy-efficient CPU clock frequency is selected compared to using the maximum frequency (i.e., turbo mode).

In order to gain more insights, we calculate the energy savings from running the algorithms at the sweet spot frequency compared to the maximum frequency (AC-Save) and the associated loss of performance (AC-Penalty). The penalty is always higher than the savings. Fig. 7 compares AC-Save and AC-Penalty for all three algorithms and all list sizes.

While researchers have already found that energy-efficiency can be achieved by redesigning software code, by making better use of memory and, by using more efficient hardware components (see [16]), it is not well known that the energy-efficiency of software is also affected by the frequency of the CPU at very precise frequencies other than the maximum frequency. Namely, our findings provide an answer (A) to our two research questions:

- A2a (Effectiveness). Software energy-efficiency can be improved by choosing the most adequate CPU clock frequency. CPU clock frequency leads to a considerable variability of the energy needed to complete a software task.
- A2b (Determinability). The CPU clock frequency which makes software more energy-efficient can be determined. In the case of counting sort for 50mio elements, reducing the clock frequency by ≈60% of its maximal speed can lead to an energy reduction of 25%.

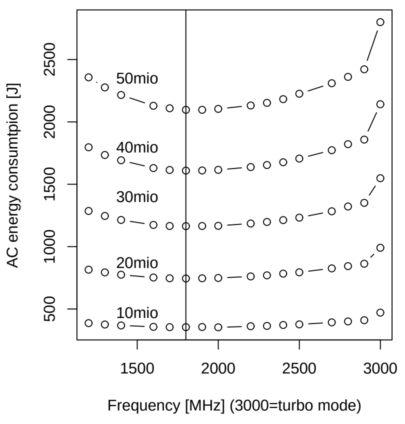
These results entail not only that for a given software application the sweet spot of CPU frequency can, and should be determined, but it also shows that software with the same algorithmic complexity can have a different energy-efficiency. Therefore, it seems natural to consider developing energy-efficiency benchmarks for software applications. While ISO software quality parameters include over 50 metrics [17], SPEC CPU2006 provides comparative studies on hardware performance and SPECpower for hardware energy-efficiency[2], the same does not happen to software energy-efficiency.

Since we have only studied CPU/memory intensive appli-

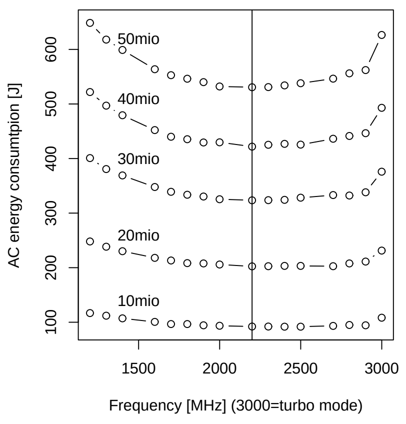
[2]https://www.spec.org

TABLE III
THE ENERGY-EFFICIENT SWEET SPOT OF SORTING ALGORITHMS.

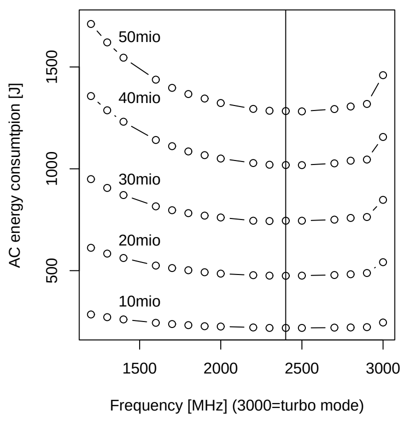
Algorithm	E (J)	Freq. (MHz)	t (s)	P (W)
radix	530.8	*2200	2.6	204.2
radix	626.9	turbo	1.9	330.4
std::sort	1282.3	*2400	5.9	216.9
std::sort	1464.1	turbo	4.2	346.3
count.	2093.8	*1800	11.3	184.9
count.	2801.5	turbo	8.5	330.3



(a) counting sort (sweet spot at 1.8 GHz)

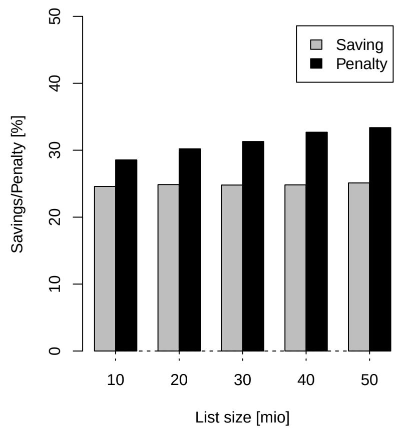


(b) radix sort (sweet spot at 2.2 GHz)

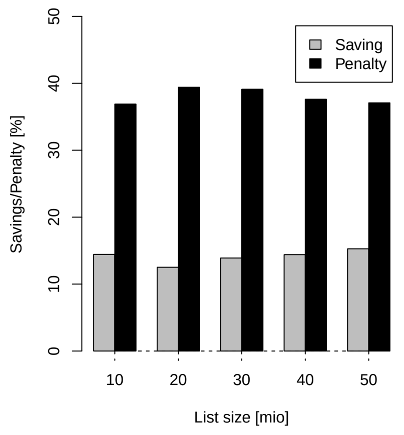


(c) std::sort (sweet spot at 2.4 GHz)

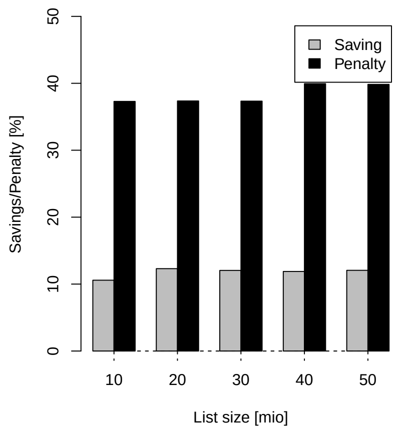
Fig. 6. AC energy consumption and the corresponding sweet spot frequencies.



(a) counting sort



(b) radix sort



(c) std::sort

Fig. 7. AC energy saving compared with time penalty.

cations which do not access hard disks and other components, further insights can be obtained by experimenting with other types of software applications which require input/output access to data storage systems. However, due to the interest in overall system energy-efficiency, we relate the CPU efficiency with the alternating current power supply (AC).

We conducted this paper as an "executable paper". Hence, all raw measurement results, executable source code used for the experiments, logs and traces can be found online in a dataset hosted at the experimental results platform Areca[3].

VI. Dynamic Software Adaptation

The results presented in the last section show empirical support for the existence of sweet spot frequencies for sort algorithms. Our idea to implement a new kind of self-adaptive software system is to enable applications to select the frequency of the CPU, based on the type of requests made for execution. We will demonstrate how such a system could run using the same computing task evaluated through this paper: sorting. A client can make a request for sorting, having as constraints the performance or the energy-efficiency of the execution, or a combination of both. When a self-adaptive software system receives a request, it uses optimization techniques to determine at which frequency the CPU should be clocked to fulfill the constraints of the request. Optimization uses the approximated functions from Table IV and the number of elements requested to be sorted. The information is stored in so-called QoS contracts. The result of the optimization is the CPU frequency at which the sorting algorithm should be executed. Thus, we propose a three-phase approach:

1) Approximate functions of sweet spot frequencies based on micro-benchmarks to determine a server's individual sweet spot frequency,
2) QoS contracts to capture the assessed non-functional behavior,
3) Optimization to compute the optimal frequency and algorithm for a given user request at runtime.

The general approach has been published in [18]. We extend previous work by incorporating hardware reconfiguration by

[3]http://areca.co/26/The-Cost-of-Sorting

means of explicit frequency scaling. In the following, each step will be examined in more detail.

A. Approximate functions

Fig. 6 depicts the energy consumption across all possible frequencies for sort invocations of the three investigated sort algorithms. As can be seen, for each algorithm a sweet spot frequency can be determined independently from the list size.

It is possible to predict the sweet spot frequency by approximating a function of the energy consumption depending on the frequency using multiple linear regression and searching the minimum value of this function for a given list size. This task can be automated using R statistics tool.

For the measurements of radix sort, fourth grade polynomial functions approximate the measured values very precisely as shown in Table IV. The first five rows show functions for 10..50 millions elements, whereas the last row represents the generic function $E(freq, size) = a \times freq + b \times freq^2 + c \times freq^3 + d \times freq^4 + e + f \times listsize$ with an adjusted R^2 of more then 99%. The minimum of this general function is at 2.4 GHz for all list sizes, which is not the measured mean sweet spot at 2.2 GHz, but is less than 1% distant from it. The cause of this difference is the (small) deviation of the approximated function and the closeness of the frequencies around the sweet spot frequency.

Thus, to automatically determine a sweet spot frequency on a target platform (unknown at design time), a developer has to provide a (micro) benchmark for different algorithmic implementations. Using the approach described above, the system can compute the sweet spot frequency automatically.

B. QoS Contracts

The Quality Contract Language – QCL [18] – allows to capture the non-functional behavior of an implementation. A contract in QCL specifies for an implementation of a task (e.g., radix sort for sort) pairs of non-functional provisions and requirements. If the requirements are fulfilled, the provisions are guaranteed to hold. Listing 1 depicts an example of a QCL contract for the radix sort implementation used in this paper. It specifies 2 modes, which are alternative pairs of requirements and provisions. The two modes represent the most energy-efficient and the fastest way to execute the algorithm. Thus, for the first mode, the sweet spot frequency determined in the previous phase is specified as a requirement on the CPU. The second mode specifies the highest possible frequency as a requirement. In addition, the runtime and energy consumption on the CPU are specified as functions depending on the list size. They are determined analogously to the sweet spot functions in the first phase. All modes specify which guarantees are given if a set of requirements is fulfilled. In the example contract, a specific maximum response time, which is equal to the time required on the CPU and a small overhead (`x1` and `x2`, respectively), is guaranteed.

Contracts, like the discussed example, can then be used to generate problem formulations for off-the-shelf constraint satisfaction and optimization problem solvers.

```
contract Radixsort implements Sort.sort {
  mode efficient {
    requires resource CPU {
      frequency = 2.400 [MHz]
      max time = f1<list_size> [ms]
      max energy = f2<list_size> [J] }
    provides max response_time = time + x1
  }
  mode fastest {
    requires resource CPU {
      frequency = 3.000 [MHz]
      max time = f3<list_size> [ms]
      max energy = f4<list_size> [J] }
    provides max response_time = time + x2
  }
}
```

Listing 1. Example of a QCL contract for radix sort.

C. Optimization

One approach to compute the decision of which algorithm to use on which server is the application of an integer linear program (ILP) as shown in Listing 2. For clarity, we only show an ILP example for the decision whether radix sort shall be executed on one of two servers in either the most energy-efficient or the fastest way. In the example, all values referring to server $N1$ correspond to the measurement values shown in the last section for a sort request of 30 million elements. The values referring to server $N2$ are not based on measurements, but introduced to show the general applicability of the approach to multiple servers.

The ILP example comprises 4 decision variables and 3 usage variables per server.The decision variables of the optimiza-

```
min: energy#N1 + energy#N2;
//decide for one server and variant
b#rdx#eff#N1 + b#rdx#fast#N1 + b#rdx#eff#N2
+ b#rdx#fast#N2 = 1;
//approximated runtime per decision
time#N1 =              1620b#rdx#eff#N1
                     + 1164b#rdx#fast#N1;
//base load + decision-induced consumption
energy#N1 =       97 +  324b#rdx#eff#N1
                     +  376b#rdx#fast#N1;
//min frequency + sweet spot-min frequency
frequency#N1 = 1200 + 1000b#rdx#eff#N1
                    + 2000b#rdx#fast#N1;

time#N2 =              1120b#rdx#eff#N2
                     +  940b#rdx#fast#N2;
energy#N2 =      100 +  420b#rdx#eff#N2
                     +  530b#rdx#fast#N2;
frequency#N2 = 1200 + 1000b#rdx#eff#N2
                    + 2000b#rdx#fast#N2;

bin b#rdx#eff#N1, b#rdx#fast#N1,
    b#rdx#eff#N2, b#rdx#fast#N2;
```

Listing 2. Example of an Integer Linear Program for self-adaptive software.

e (intercept)	$a \times freq$	$b \times freq^2$	$c \times freq^3$	$d \times freq^4$	$f \times listsize$	adj.r.squared
98.558887	-17.000578	24.473429	1.497380	5.577048	1.00E+07 (fixed)	0.939935
214.645925	-30.237388	45.255654	2.977515	10.231982	2.00E+07 (fixed)	0.9699779
344.115647	-43.960454	78.485585	1.588671	15.814538	3.00E+07 (fixed)	0.9519011
450.068206	-49.751193	103.690842	0.926805	21.180903	4.00E+07 (fixed)	0.9654956
564.112603	-51.629321	131.365326	6.015857	26.856475	5.00E+07 (fixed)	0.9554555
-15.65866	-86.12392	171.4039	5.816562	35.62546	1.16653E-05	0.994221

TABLE IV
FOURTH GRADE POLYNOMIAL FUNCTIONS FOR RADIX SORT ON 10 TO 50 MILLION ELEMENTS.

tion problem have the form $b\#algorithm\#variant\#server$, whereby the prefixing b denotes the Boolean type of the variable, which is explicitly stated as constraint on lines 22/23, and $algorithm$, $variant$ and $server$, delimited by pounds (#), denote the respective algorithm, variant (energy-efficient or fast) and server. The meaning of $b\#rdx\#eff\#N1 = 1$ is the decision to use radix sort in its energy-efficient variant on server $N1$. Thus, each decision is represented by a variable.

Each server is characterized by the variables $time$, $energy$ and $frequency$, which include the name of the server separated by a pound. They denote the time required on, the energy spent by and the CPU frequency used for the respective server.

The objective function of the ILP is a linear combination of the problem's variables. In the example, the objective function specifies the goal to minimize the energy consumption of both servers for user requests to sort n elements.

Then, two types of constraints are generated for a specific request (e.g., the invocation of sort for a list of 30 million elements as in the example above). First, a structural constraint to ensure the selection of at least one variant is generated (cf. line 3 and 4). Second, for each server variable (time, energy and frequency) a constraint, reflecting the impact of a decision on them is generated. For example, the constraint in line 6/7 specifies that deciding for radix sort in efficient mode on server N1 will require 1620 ms, whereas using fast mode will require only 1164 ms. For energy consumption, the idle consumption has to be considered in addition. The constraint on line 9/10 reflects an idle consumption of 97 J and the respective consumption of radix sort in the most efficient and fastest mode. Considering the idle consumption is important if the servers are always powered. The frequency constraint on line 12/13 reflects the minimum possible frequency (1200 MHz), the sweet spot frequency (1000+1200 MHz), and the maximum possible frequency (1000+2000 MHz).

To solve this optimization problem, standard solvers like LP Solve [19] can be applied. The time required to solve problems as shown in the example, depends on the number of modes (i.e., execution variants) for which we used $m = 2$ representing the most efficient and the fastest variant, the number of algorithms for the same task A (e.g., different sort algorithms) and the number of servers N. Fig. 8 shows the time required to solve ILPs with $A = [1..10]$ algorithms with 2 modes each on $N = [2..10]$ servers. As can be seen, the time to derive the decision is negligible. It took ≈3.1 ms to identify which algorithm out of 10 to run on which of 10 servers.

VII. RELATED WORK

The closest existing research to our work was conducted by Livingston et al. [3]. Their work classifies software applications as memory- and compute-bound. For memory-bound applications, they demonstrate that a higher energy-efficiency is achieved at lower CPU frequencies since memory behaves as a bottleneck. For compute-bound applications, a higher energy-efficiency is achieved at higher CPU frequencies since finishing work quickly is the best approach for efficiency. For algorithms which cannot be purely classified as memory- or compute-bound, they propose to use sweet spot frequencies, a benchmarked optimal frequency between the lowest and highest frequency. Our work confirms the findings by Livingston et al. also in newer computer architectures and makes a detailed analysis of sweet spot frequencies.

Other related work can be classified taking into account the level at which energy-efficiency analysis was conducted. We use the terms macro-, meso-, and micro-level to express studies conducted with large software applications, algorithms, and instructions. At the macro-level, researchers (cf. [16]) have looked into the energy-efficiency of large management information systems such as ERP, CRM, and databases. While it is important to look into the efficiency of such systems to identify fields of improvements, the approach taken does not allow to gather insights on how software could be re-engineered differently to obtain energy reductions. At the meso-level, Bunse et al. [20] evaluate various sorting algorithms in battery powered mobile communication using smartphones. The results indicate that insertion sort is most efficient. Rivoire

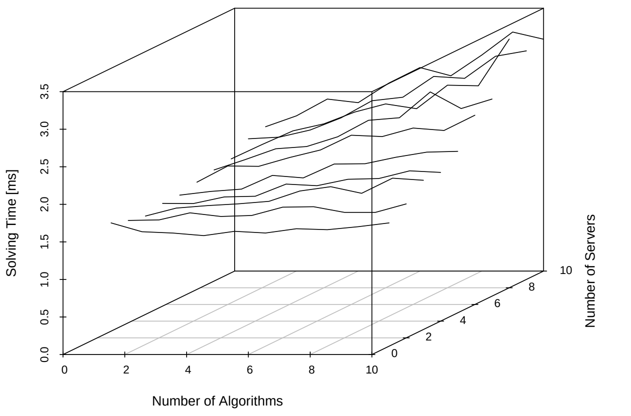


Fig. 8. Time for adaptation decision making using LP Solve.

et al. [21] investigate system-level benchmarks for sorting. Nonetheless, the work does not explore the effect of CPU frequency on software energy-efficiency. At the micro-level, Ong and Yan [22] use an abstract machine to study the energy consumption of search and sorting algorithms. The energy requirement of each instruction was estimated and, e.g., an ALU access consumes 8×10^{-12} joule per 32 bits. Their findings indicate that the energy consumption can differ in orders of magnitude between algorithms, and, also, that faster algorithms can sometimes consume more energy than slower ones. In [23], the authors propose a first-order, linear power estimation model that uses performance counters to estimate CPU and memory consumption. The accuracy of the model estimates consumption within 4% of the measured CPU consumption.

Related to future work, Beloglazov et al. [24] observe that modern large servers currently use 32 or 64 DIMMs that lead to power consumption by memory higher than by CPUs. This suggests that the study and design of energy-efficient software should account for CPU and memory efficiency.

VIII. Conclusion

Over the years, hardware energy-efficiency has significantly improved. Nevertheless, research on software efficiency has not received the same attention. Thus, in this paper we study mechanisms to make software more energy-efficient. Our findings indicate that the existence of sweet spots can be explored to realize software energy-efficiency in at least three fields: 1) by adapting the CPU frequency to sweet spots, the maximum power of the upper limit used by a computing system can be established; 2) the consideration of sweet spots leads to effective energy gains which reached up to 25% for the investigated sorting algorithms; and 3) the existence of sweet spots enables the design of new and more efficient self-adaptive software architectures. Our results are important and relevant since experiments were conducted using the most advanced hardware measuring devices.

Acknowledgements

This work has been partially funded by the German Research Foundation (DFG) under project agreements SFB 912/1 2011 and SCHI 402/11-1.

References

[1] J. Koomey., "Growth in Data center electricity use 2005 to 2010," Analytics Press, 2011. [Online]. Available: http://www.analyticspress.com/datacenters.html

[2] L. Barroso and U. Holzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec 2007.

[3] K. Livingston, N. Triquenaux, T. Fighiera, J. Beyler, and W. Jalby, "Computer using too much power? give it a rest (runtime energy saving technology)," *Computer Science - Research and Development*, pp. 1–8, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00450-012-0226-0

[4] K. Choi, R. Soma, and M. Pedram, "Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-Off Based on the Ratio of Off-Chip Access to On-Chip Computation Times," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, February 2004, Paris, France.

[5] L. Brochard, R. Panda, and F. Thomas, "Power consumption of clusters: Control and Optimization," Industry Talk at Fourth International Conference on Energy-Aware High Performance Computing (EnA-HPC), September 2013.

[6] D. Versick, I. Waßmann, and D. Tavangarian, "Power consumption estimation of CPU and peripheral components in virtual machines," *ACM SIGAPP Applied Computing Review*, vol. 13, no. 3, pp. 17–25, September 2013.

[7] X. Liu, P. Shenoy, and M. Corner, "Chameleon: application level power management with performance isolation," in *Proceedings of the 13th Annual ACM International Conference on Multimedia (MULTIMEDIA)*, November 2005, Singapore.

[8] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs," in *Proceedings of the 19th ACM/IEEE Conference on Supercomputing (SC)*, November 2006, p. Article 107, Tampa, Florida, USA.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251203.1251223

[10] M. Cinque, D. Cotroneo, F. Frattini, and S. Russo, "Cost-Benefit Analysis of Virtualizing Batch Systems: Performance-Energy-Dependability Trade-offs," in *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, December 2013, pp. 264–268, Dresden, Germany.

[11] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 0, pp. 194–204, 2013.

[12] R. Schöne, D. Hackenberg, and D. Molka, "Memory performance at reduced cpu clock speeds: an analysis of current x86_64 processors," in *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, ser. HotPower'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387869.2387878

[13] M. P. I. Forum. (2012, Sep.) Mpi: A message-passing interface standard - version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[14] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1998, vol. 3 - Sorting and Searching, 2nd Edition.

[15] ISO/IEC, "ISO/IEC 14882:2011: Programming languages – C++," Tech. Rep., 1998.

[16] E. Capra, C. Francalanci, and S. Slaughter, "Measuring application software energy efficiency," *IT Professional*, vol. 14, no. 2, pp. 54–61, March 2012.

[17] ISO/IEC, "ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[18] S. Götz, C. Wilke, S. Richly, and U. Aßmann, "Approximating quality contracts for energy auto-tuning software," in *Proceedings of First International Workshop on Green and Sustainable Software (GREENS 2012)*, 2012.

[19] K. Eikland and P. Notebaert, "LP Solve 5.5 reference guide," http://lpsolve.sourceforge.net/5.5/ (access on 26.11.2012).

[20] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the best sorting algorithm for optimal energy consumption," in *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, 2009, pp. 199–206.

[21] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "Joulesort: A balanced energy-efficiency benchmark," in *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data (SIGMOD)*, 2007.

[22] P.-W. Ong and R.-H. Yan, "Power-conscious software design-a framework for modeling software on hardware," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, Oct 1994, pp. 36–37.

[23] G. Contreras and M. Martonosi, "Power prediction for intel xscale reg; processors using performance monitoring unit events," in *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, Aug 2005, pp. 221–226.

[24] A. Beloglazov, R. Buyya, Y. C. Lee, and A. Y. Zomaya, "A taxonomy and survey of energy-efficient data centers and cloud computing systems." *Advances in Computers*, vol. 82, pp. 47–111, 2011.