Analyzing and Optimizing Energy Efficiency of Algorithms on DVS Systems

A First Step towards Algorithmic Energy Minimization

Tetsuo Yokoyama Gang Zeng Hiroyuki Tomiyama Hiroaki Takada

Graduate School of Information Science
Nagoya University
Nagoya, Aichi 426-0863
Tel: +81-52-789-2795
Fax: +81-52-789-5889
e-mail: {yokoyama,sogo,tomiyamahiro}@ertl.jp

Abstract— The energy efficiency at the algorithmic level on DVS systems and its analysis and optimization methods are presented. Given a problem the most energy efficient algorithm is *not* uniquely determined but dependent on multiple factors, including intratask dynamic voltage scaling (IntraDVS) policies, the size of intermediate data structure, and the size of inputs. We show that at the algorithmic level principles behind energy optimization and performance optimization are *not* identical. We propose a metric for evaluating optimal energy efficiency of static voltage scaling (SVS) and a few new effective IntraDVS policies employing data flow information. Experimental results on sorting algorithms show the existence of several tradeoffs in terms of energy consumption. Transforming algorithms by employing problem specific knowledge and data flow information successfully improves their energy efficiency.

I. Introduction

Energy efficiency has been one of the crucial concerns in modern software development as well as in a system design. The prediction of energy consumption in the upper process reduces the number of iterative development cycles. When designing and selecting algorithms at the upper process, techniques for analyzing and optimizing energy efficiency repeatedly at the algorithmic level assure us of enhancing software productivity.

However, the energy complexity theory is a rudimentary art [5, 3, 4], compared to the time and space complexity theories. This is partly because there is no synthetic system-level energy model available for either analyzing the energy complexity of algorithms or guidelines to construct energy efficient algorithms. To be concrete, we are unable to determine which algorithm, either quicksort or heapsort, is more energy efficient, before executing their implementations or without selecting specific hardware and its configurations. It is also uncertain how to construct new energy efficient sorting algorithms.

In contrast, the hardware technology and techniques for trading off energy for performance have been established. For example, dynamic voltage scaling (DVS) has been implemented in the off-the-shelf processors. To use the benefit of DVS processors to the greatest extent, techniques at various levels, including circuit logics, processors, compilers, and operating systems, have been exploited. We take one more step towards *algorithmic level energy minimization*.

To reduce energy consumption, in contrast to almost all of the existing DVS techniques, our techniques exploit the logic and structure of programs. Algorithmic reconfiguration, which transforms algorithms by employing problem specific knowledge and data flow information, results in energy optimization at a more abstract level than at any of the above mentioned lower levels. This abstraction facilitates general discussion, independently of particular hardware and software implementation.

The main contribution of this paper includes:

- findings that at the algorithmic level energy and performance optimization are *not* always identical on DVS systems. The fastest algorithm on average is *not* always the most energy efficient on average. Energy and memory space are sometimes a tradeoff.
- a measure of optimal energy consumption of static voltage scaling (SVS). Under an ideal assumption for comparing energy efficiency of algorithms, this measure is independent of the deadline constraint.
- formulation of intratask DVS (IntraDVS)[1] policies employing data flow information. Based on this formulation, we propose a few new effective policies.
- new energy efficient sorting algorithms. The best result achieves 62.6 % energy savings on the system of 10/3 voltage and frequency scalability from heapsort.

The rest of this paper is organized as follows. Sect. II explains the reason why we rely on CMOS based model even at the algorithmic level and describes the assumptions used in next sections. Sect. III reviews a few sorting algorithms and discusses energy optimization methods at the algorithmic level.

[1]While ordinary *inter*-task DVS adjusts the processor speed and voltage task-wise, *intra*-task DVS does within tasks. IntraDVS is detailed in [7] and references therein.

978-1-4244-2749-9/09/$25.00 ©2009 IEEE

Sect. IV introduces a metric of optimal SVS and compares energy efficiency when SVS is used. Sect. V represents several IntraDVS policies and evaluates a few algorithms with those IntraDVS policies on DVS systems. Finally, we end with a few remarks in Sect. VI.

II. Our Energy Model of DVS Systems and Assumptions

Since our objective is to investigate algorithmic factors affecting energy efficiency at a system level, the model should be abstract enough to be independent of specific architecture and environment. Tiwari, Malik, and Wolfe [8] proposed an instruction set simulator (ISS) for estimating power consumption. It is modeled and parameterized with the aid of current measurement of real chips in the case that the same instructions are consecutively executed. For given instructions, their estimation is accurate. However, for given an algorithm, the power estimation is too much affected by characteristics of specific instances, such as chips, compilers, programming, and programming languages. This ISS level abstraction itself is not sufficient for studying general tendencies and characteristics of energy and performance tradeoffs at the algorithmic level. In the upper processes of software development many hardware configurations are unspecified, and ISS simulators require many indeterminate HW/SW configurations. Their simulations are also not lightweight.

The model, on the other hand, should be precise to reflect realistic characteristics and behavior of the latest chips. For example, in augmented Turing machine (ATM) model [4], which is of highest abstraction, the amount of dissipated energy is augmented in each transition between states. This model could be one of the bases for constructing the energy complexity theory. However, none of real architecture has reflected Turing machine and low-power technologies for CMOS based architectures have not been simply modeled on it. Therefore, we focus on CMOS based architectures.

For the sake of simplicity, in this paper, we use the following assumptions. Dynamic power consumption is proportional to the cube of V:

$$P_{\mathrm{dyn}} \propto V^3 . \tag{1}$$

The scale of normalized voltage ranges over $[0.3, 1]$. The number of operations, instead of the number of instructions, is used for the measure of work, although in Sect. V IntraDVS policies are formulated using cycles for the sake of compatibility to the existing work. The amount of work does not change before and after voltage scaling, and the speed of processing work is proportional to frequency. Power consumption is the same for any algorithms. The overhead of voltage transition in time and energy is negligible. We focus on dynamic power, dominant in the current technology, and ignore leakage power. Peripherals are ignored. We also ignore the effect of programming languages, specific program structures, such as loop vs recursion, data structures, memory access, architecture, logic, OS, and compilers. The DC–DC converter has constant efficiency, and therefore it can be ignored when comparing power and energy efficiency of multiple configurations. Our targets are hard real time systems with a single processor given a single task.

III. Sorting Algorithms

Over sorting algorithms with $\Theta(n \lg n)$ average-case execution time, *quicksort* is in practice dominant for large input arrays. Although its worst-case execution time is $\Theta(n^2)$, it generally outperforms heapsort two to five times on average in C implementation [6]. The slack time generated from the workload variation between the worst- and average-cases affords a chance to lower voltages. However, for a large input array, the limited frequency scalability of processors usually hinders us from using up all the slack time.

The recursive case of quicksort is:[2]

```
qsort (x:xs) = qsort (us) ++ [x] ++ qsort (vs)
          where (us,vs) = splitby (x,xs)
```
(2)

The head `x` of a given list `x:xs` is selected as a pivot, and function `splitby` splits list `xs` into list `us`, in which all elements are greater than or equal to `x`, and `vs`, in which all elements are less than `x`. The problem of sorting `xs` is divided into two smaller subproblems, sorting `us` and `vs`. The results of those subproblems `qsort (us)` and `qsort (vs)` are concatenated by `(++)` at the front and back of a singleton list `[x]`, respectively.

The step-counting is one of the quantitative measures, representative of the execution time [1]. We denote by $\delta^f(n)$ a predicted time complexity of function f that takes the size n of the input list as an argument. The time complexity of quicksort can be described as

$$\begin{aligned}\delta^{\mathtt{qsort}}(n) = {} & \delta^{\mathtt{splitby}}(n-1) + \\ & \delta^{\mathtt{qsort}}(d) + \delta^{\mathtt{qsort}}(n-d-1)\end{aligned} \tag{3}$$

with d a randomly chosen integer from range $[0, n-1]$. The steps of `splitby` are assumed to be proportional to the size of the input lists, *i.e.*, $\delta^{\mathtt{splitby}}(n) = cn$. For the sake of simplicity, the cost of `qsort` with a list of size zero or one and concatenation `(++)` of two lists is assumed to be zero. Because of those assumptions and the equation of $\delta^{\mathtt{qsort}}$ (3), the worst- and best-case complexities of $\delta^{\mathtt{qsort}}(n)$ are cn^2 and $cn \lg n$, respectively. For example, if the problem of size n is divided into halves, the remaining worst-case execution time becomes exactly half.

The keys to optimizing IntraDVS are the precise prediction of the remaining execution time and its reduction as early an execution stage as possible [7]. If we ignore the order of evaluation of quicksort, during the execution of `qsort xs` the intermediate expression exhibits the form:

$$\begin{aligned}&\mathtt{qsort}\,(\mathtt{xs}_1)\ \mathtt{++}\ [\mathtt{x}_1]\ \mathtt{++}\ \mathtt{qsort}\,(\mathtt{xs}_2)\ \mathtt{++}\ [\mathtt{x}_2] \\ &\qquad \mathtt{++}\ \cdots\ \mathtt{++}\ [\mathtt{x}_{m-1}]\ \mathtt{++}\ \mathtt{qsort}\,(\mathtt{xs}_m)\end{aligned} \tag{4}$$

in which any of $\mathtt{qsort}\,(\mathtt{xs}_i)$, the size of which is greater than zero, can be updated using the above recursive case definition of `qsort` (3). As implemented in many libraries, firstly computing the smaller list of `us` and `vs`, for each recursion, results in reducing the maximum size of a stack frame. This execution order guarantees the maximum stack size up to the logarithm

[2] We adopt Haskell-like notation [1].

of the input list size. On the other hand, the remaining execution time, computed with $\sum_i \delta^{\mathtt{qsort}}(size(\mathtt{xs}_i))$, does not decrease rapidly, in which *size* returns the size of a given list.

However, if we always compute the longer lists in $\mathtt{xs}_i$ $(1 \leq i \leq m)$ for each iteration in the different program structure, the remaining execution time sharply decreases in most cases. The generated surplus slack time is available for lowering voltages. The size k of a stack for controlling the evaluation order and the decrease quantity of the remaining execution time are a tradeoff. We will conduct an experiment for evaluating this tradeoff in Sect. V-B. We call this division technique *an early division with stack size* k. The number of steps does not change by increasing the stack size k of an early division while the consumed memory space increases. Thus, the order of execution is important for lowering power.

Heapsort is often used in embedded systems with real-time constraints, since its asymptotic execution time bound $\Theta(n \lg n)$ is the optimum in general sorting algorithms and the auxiliary storage outside the input array has constant upper bound. Hence, heapsort does not generate much workload variation and proper SVS minimizes energy consumption in the case that only a task of heapsort itself is taken into account. In addition to inefficient performance compared to quicksort in most cases and on average, a drawback of heapsort on DVS systems is incapability of exploiting the biased input list distribution even known in advance in real applications.

Introsort [6, 9] is a hybrid algorithm of quicksort and heapsort. Introsort begins with quicksort and stops with excellent performance for most inputs. If the recursion depth is detected exceeding a predefined threshold by means of *introspection*, it switches to heapsort to stay in $\Theta(n \lg n)$ asymptotic performance.

Introsort has two advantages on DVS systems. First, it preserves the benefit of quicksort; The early division technique, applicable in quicksort phase, generates the slack time due to the workload variation. Second, the ratio between average- and worst-case execution time is bounded by a small constant; unlike quicksort, it does not generate much useless slack time for practical DVS systems with a limited voltage scalability.

Fig. 1 provides a performance comparison among heapsort, Musser's introsort [6], and quicksort. The number of executed operators is normalized with the result of heapsort given the worst-case input. For fair evaluation, throughout this paper, experimental data and programs are excerpted from [9].[3] In Musser's introsort, a threshold to switch to heapsort and insertion sort are $2 \lg n$, in which n is the size of an input array, and 16, respectively. The code of quicksort is derived from the Hewlett-Packard implementation of the C++ Standard Template Library [9]; Median-of-3 pivot selection is used and a threshold to switch to insertion sort is 16. Overall, heapsort with the worst-case and random input show negligible execution time difference for generating the slack time. Given random input, the performance of introsort and quicksort is almost identical. In the case that the input size is either 52 or 100, quicksort has the best performance. In the case that the input size is more than 100, introsort is superior to quicksort both in the worst- and average-cases, and introsort and heapsort are a tradeoff.

[3] According to [9], the data was "obtained on a MIPS R5000 processor with 512 kilobytes of secondary cache and 64 megabytes of main memory, using version 7.2.1 of the Silicon Graphics MIPSpro C++ compiler." "A minimum of five trials were performed for each pair of algorithm and sequence, ..."

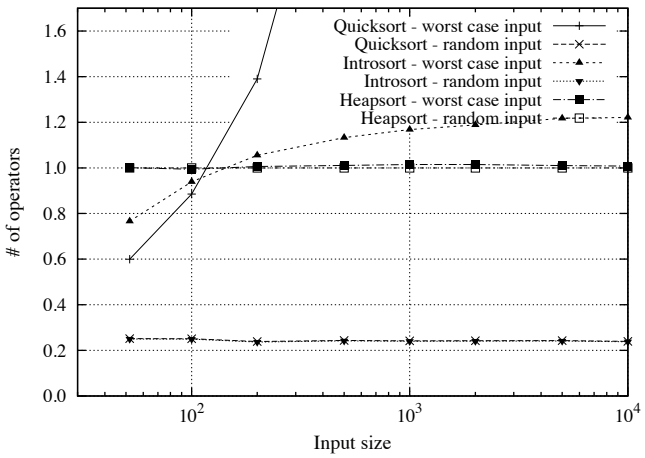

Fig. 1. Normalized operator numbers of sorting programs.

IV. Static Voltage Scaling

Static voltage scaling (SVS) is a technique for lowering energy consumption of systems by assigning a single supply voltage to each task before execution. The optimum voltage for each task is determined as the minimum voltage to finish the task execution exactly at the deadline. In SVS, the slack time due to workload variance cannot be used for the further energy reduction at runtime.

We propose a metric for evaluating energy consumption of algorithms when a hard deadline is assumed and SVS is used. Under the ideal assumption that power consumption is cubically proportional to voltage (1), relative energy efficiency of algorithms is *insensitive* to deadlines. We denote by δ_{w} and δ_{a} the worst- and average-case execution cycles, respectively.

Lemma 1 (Optimal SVS)**.** *For SVS without scaling bound, an algorithm of a given task is energy optimal on average iff*

$$\delta_{\mathrm{w}}^2 \delta_{\mathrm{a}} \tag{5}$$

is minimum.

Proof. Let D be deadline. The execution at frequency δ_{w}/D finishes exactly on deadline. The corresponding power consumption $P(f)$ is the smallest in the case that $f = \delta_{\mathrm{w}}/D$ because P monotonically increases. The execution time is the number of cycles X divided by frequency, *i.e.*, $X/(\delta_{\mathrm{w}}/D)$. Thus, energy consumption is $P(\delta_{\mathrm{w}}/D)DX/\delta_{\mathrm{w}}$. Because we are comparing the different algorithms under the same deadline, D is constant. The average energy consumption is proportional to the average of $\delta_{\mathrm{w}}^2 X$, which is equivalent to (5). □

It should be noted that the scalability of real processors is bounded by the minimum frequency $f_{\min}$. This realistic assumption provides us with a measure $\delta_{\mathrm{w}}^2 (D f_{\min} \uparrow \delta_{\mathrm{a}})$, in which binary operator $\uparrow$ returns the maximum of two arguments. Hence, energy efficiency depends on deadlines.

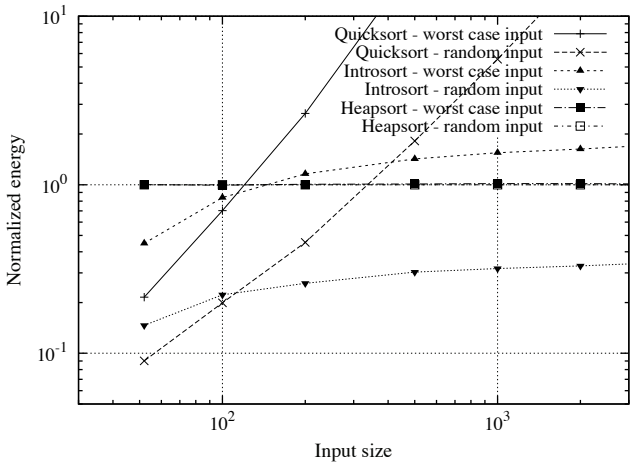

Fig. 2. Normalized energy of sorting programs using SVS.

The SVS energy measure (5) designates two elements affecting energy consumption. The worst-case execution cycles δ_{w} have the more impact; The major effort should be dedicated to reduce this. The average-case execution cycles linearly affects the SVS energy measure (5), which is not negligible as we will see in the following.

The double logarithmic chart Fig. 2 shows energy consumption normalized by using heapsort with random input. The plotted data is obtained by multiplying the values in Fig. 1 using the SVS energy measure (5). The characteristics of Fig. 1 and Fig. 2 differ. The energy consumption of quicksort with random input, as well as the worst-case input, increases sharply due to quadratic effect of the worst-case execution cycles δ_{w}, which quadratically increases the SVS energy measure (5). For heapsort, the less execution time variance results in the less energy consumption variance. The input data size, as well as its distribution, affects energy consumption.

The most energy efficient algorithm both on average and at the worst-case depends on the input size. First, in the case that the input size is either 52 or 100, quicksort with random inputs shows the best result. Quicksort with the worst-case input is superior to both introsort with the worst-case and heapsort with any input. Those results can be anticipated from Fig. 1, since, given random inputs, quicksort and introsort outperform heapsort and the difference of the worst-case execution time between quicksort and heapsort reflects the difference of energy consumption between them. Second, in the case that the input size is 200, heapsort and either of quicksort or introsort is a tradeoff. The most energy efficient algorithm depends on both the worst- and average-case execution time. Third, more than 200, quicksort is no longer useful since it is outperformed by the worst-cases of heapsort and introsort. Heapsort and introsort remain a tradeoff.

The fact that quicksort with random input is not always the best shows that the algorithm of the best performance on average is *not* always the most energy efficient when SVS is used. It should be noted that the frequency lower bound, not assumed here, exacerbates energy efficiency, since the less slack time is generated from workload variation in quicksort. In the last two cases, algorithms should be selected by comparing the significance of the worst- and average-case energy consumption and the size and characteristics of input arrays.

In embedded systems, the severe timing constraints are imposed and therefore heapsort is usually adopted as sorting algorithm. However, with the little additional slack time, introsort outperforms heapsort in terms of average and typical energy consumption. The performance superiority of introsort has been reported in [6, 9].

V. Dynamic Voltage Scaling

Dynamic voltage scaling is a technique for scaling the processor's supply voltages and working frequencies at runtime. Whereas SVS must always use the worst-case execution time to meet deadlines, DVS takes into account actual available execution-time information at runtime. To the best contrast of energy efficiency of algorithms, we focus on energy optimization of a single task and voltage scaling during its execution, *i.e.*, IntraDVS [7].

The development using IntraDVS consists of four parts. First, we make a model of execution time of a given program and estimate parameters. As seen in Sect. III, this process can be automated, at least, for sorting algorithms by means of step-counting functions. We exploit data flow information, but not only control flow information. Second, programs are transformed into ones with the steeper reduction of the worst-case execution time or other time metrics likewise. For example, in Sect. III, we suggested that the order of processing expressions be changed to obtain the programs with the better remaining execution time prediction in the early stages. Third, voltage scaling points (VSPs) annotated with parameterized scaling factors are inserted. For the sake of simplicity, in this section, we insert VSPs right after division of quicksort and introsort, at which the remaining execution time can decrease by more than single time unit. The scaling factors are contingent on IntraDVS scheduling strategies, which are detailed in the next subsection. Fourth, at runtime the remaining execution time is predicted and voltages are scaled at each VSP.

A. IntraDVS policies

In IntraDVS, voltage changes are determined by IntraDVS policies at runtime. Those policies are classified with respect to the remaining predicted execution cycles (RPEC). For the divide-and-conquer approach, *e.g.*, used for quicksort and introsort, we define the equation

$$\delta(d) = c(d) + \bigoplus_j \left(\sum_{i=0}^{m_j} \delta(d_i^j) \right), \quad \{d_0^j, \ldots, d_{m_j}^j\} \in \mathrm{div}(d) \tag{6}$$

as a general scheme for predicting the remaining execution time δ of a given program with input d. Here, $c(d)$ is the number of cycles in order to divide data d into a set $\{d_0^j, d_1^j, \ldots\}$. Variable j is determined by the way of separation. The second term on the right hand side disappears if $\mathrm{div}(d)$ is an empty set. In this section, we use the number of cycles to estimate the remaining execution time. However, without loss of generality, this measure can be replaced with, *e.g.*, the actual time and the

number of operations and steps, subject to the necessary precision and available prediction time and data. At each VSP, the frequency is set to

$$\left(\frac{\delta}{D-\sigma} \downarrow f_{\max}\right) \uparrow f_{\min} \tag{7}$$

to meet deadline and minimize the energy consumption, in which σ represents the elapsed time, a range $[f_{\min}, f_{\max}]$ represents available core frequencies, and binary operators $\uparrow$ and $\downarrow$ return the maximum and minimum of two arguments, respectively. We call such frequency the *minimum sufficient frequency* with respect to the predicted cycles. At each VSP, the frequency is set to the minimum sufficient frequency with respect to RPEC. This idea is based on the fact that the most power efficient frequency is the smallest constant, at which the task exactly meets its deadline under the idealistic assumptions such that power is a convex increasing function of frequency (1) (cf. [2]).

In general, data flow diverges into many branches. We abbreviate a task of size d to task d, if it is clear which task is designated in the context. For designing IntraDVS policies, we follow the concept presented in [7]. To the best of the authors' knowledge, the formulation of IntraDVS policies suitable for divide and conquer algorithm is new.

We show several representative RPECs. The remaining *worst*-case execution cycles (RWEC)

$$\delta_{\mathrm{w}}(d) = c(d) + \max_j \left(\sum_i^{m_j} \delta_{\mathrm{w}}(d_i^j)\right), \quad \{d_0^j, \ldots, d_{m_j}^j\} \in \mathrm{div}(d) \tag{8}$$

is a simple but ineffective RPEC, which just returns the cycles of split $c(d)$ plus the maximum of the sums of the direct subsequences. The policy based on RWEC is the most conservative and in most cases not utterly effective. The steps of quicksort (3) becomes RWEC if $d = 0$.

The remaining near *average*-case execution cycles (RAEC)

$$\delta_{\mathrm{a}}(d) = c(d) + \sum_j \left(\sum_i^{m_j} \delta_{\mathrm{a}}(d_i^j)\right) p_j, \quad \{d_0^j, \ldots, d_{m_j}^j\} \in \mathrm{div}(d) \tag{9}$$

are much smaller than RWEC in many applications due to workload variation at runtime. Here, we assume that the sum of all probabilities is one, $\sum p_j = 1$. The minimum sufficient frequency with respect to RAEC also becomes much smaller. For example, RAEC of quicksort is:

$$\delta_{\mathrm{a}}^{\mathtt{qsort}}(n) = \delta_{\mathrm{a}}^{\mathtt{splitby}}(n-1) + \sum_{d=0}^{n-1} \left(\delta_{\mathrm{a}}^{\mathtt{qsort}}(d) + \delta_{\mathrm{a}}^{\mathtt{qsort}}(n-d-1)\right) \frac{1}{n}. \tag{10}$$

To compute $\delta_{\mathrm{a}}(d)$, the RPECs of all successors are needed and it is in general impossible to analytically obtain it. However, RAEC is near real average-case execution cycles and can be statistically estimated by using a number of real measurements of the execution time. The effect of *hot paths* is appropriately reflected by the weight of probabilities p_j. The maximum frequency bound makes RAEC policy not safe, violating deadline.

The remaining *safe* execution cycles (RSEC) (cf. [7])

$$\delta_{\mathrm{s}}(d) = \frac{D-\sigma(d)}{z(d)-\sigma(d)} c(d) \tag{11}$$

are the number of cycles in the case that the maximum frequency is used for all descendant nodes and the smallest possible frequency while meeting the deadline is used for the current node with $z(d)$ a deadline of division of d:

$$z(d) = D - \frac{\delta_{\mathrm{w}}(d) - c(d)}{f_{\max}}, \tag{12}$$

$\sigma(d)$ a given starting time of processing d, and $f_{\max}$ the maximum frequency. The runtime computation of the policy derived from RSEC is lightweight; The right hand side on the RSEC equation (11) does not include the RSECs of the successors, and both σ and δ_{s} are statically determined.

The RSEC policy is the most aggressive, in the sense that it decreases frequencies as much as possible at any moment while satisfying deadline constraints. However, if RWEC does not substantially decrease at runtime, the RSEC policy exhausts the slack time in the early stage of task execution, resulting in the worse energy consumption.

To remedy this, we combine two policies of different characteristics. A hybrid policy with RSEC guarantees meeting the deadline constraints while using possibly aggressive prediction. The hybrid remaining execution cycles between RSEC and some RPEC:

$$\delta_{\max(x,y)}(d) = \max(\delta_x(d), \delta_y(d)) \tag{13}$$

avoid overestimation by using the RSEC policy in the early stage of task execution. We refer to the hybrid cycles $\delta_{\max(\mathrm{s},\mathrm{a})}$ between RSEC and RAEC as the remaining safe average execution cycles (RSAEC). The construction of this RSAEC is new.

B. Experimental Results

Fig. 3 shows a comparison of energy consumption normalized with the result of heapsort with the worst-case input. Each deadline is set to the worst-case execution time of introsort with the same size input. Since, as expected, quicksort returns much poorer results than introsort and heapsort, we here ignore it. At runtime, required voltage changes, specified by each VSP, are designated by using a lookup table, the remaining execution time, the current time, and deadline. Lookup tables are constructed by the experimental data presented in Fig. 1, and the intermediate values among them are predicted by using linear interpolation. Therefore, the cost of introsort for the smaller input size is underestimated, since fast insertion sort is used for the lower size data. For heapsort, voltage can be statically assigned. Introsort with the RWEC policy and the worst-case input runs at the highest frequency. Since heapsort with random input shows almost the same result as heapsort with the worst-case input, we omit it. The tradeoff between heapsort

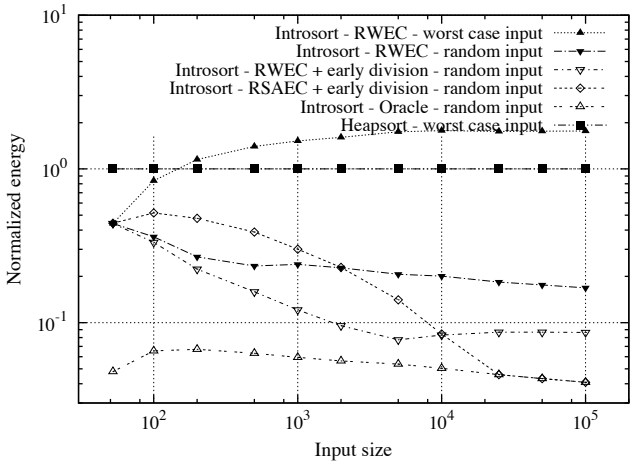

Fig. 3. Normalized energy of sorting programs using DVS.

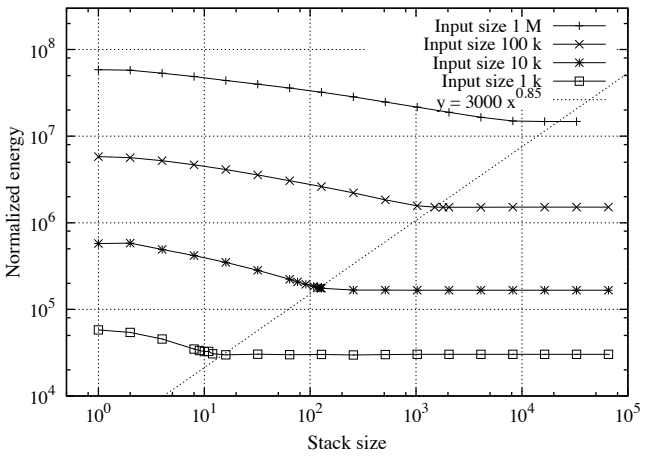

Fig. 4. Effects of the stack size on energy consumption.

and introsort with the RWEC policy and the worst-case input is preserved from Fig. 1. Since the slack time, if any, is limited, at input size 52, introsort with many configurations has almost the same result. The stack size of an early division is $4 \lg n$, in which n is the input array size. The result of the RSAEC policy is not superior to the RWEC policies in the case that the input size is the smaller. This is because the early consumption of the slack time makes frequencies scaled up near deadline. Oracle introsort runs at the minimum frequency without taking deadline constraints into account. The ratio between this oracle and introsort using the RWEC policy with the worst-case input should be and was approximately squared frequency scalability times the ratio of their execution time, which is approximately 3 %. Early division strictly improves energy efficiency of introsort with the RWEC policy. At size 5 000, for random inputs early division introduced to introsort with the RWEC policy results in 62.6 % energy savings. It is only 44 % greater than the result of oracle. The RSAEC policy approaches the result of oracle, given the larger input. The experimental result shows that the difference of the RSAEC policy and oracle is less than 1 % from 25 000 to 1 000 000.

Fig. 4 shows a tradeoff between space and energy in introsort using an early division. The greater stack size it has, the more energy is saved. The energy savings are saturated to approximately 25 % of energy consumption of a single unit temporary stack. This saturation occurs approximately at line $y = 3000 \times x^{0.85}$. It should be noted that the energy overhead due to the larger memory was ignored and taking into consideration those effects may exacerbate the energy consumption.

VI. Concluding Remarks

Through a case study on sorting algorithms, we have shown a lightweight, automatic analysis and optimization method on DVS systems under the time constraint for improving algorithmic energy efficiency, which is needed in the upper process in software development. The sorting algorithms showed tradeoffs between energy and other metrics in multiple factors. The most energy efficient algorithm also depends on the size of input data. The algorithmic reconfiguration successfully reduces energy consumption. Therefore, at the algorithmic level energy optimization and performance optimization differ.

Acknowledgments

We would like to thank Tatematsu Tomohiro, Krzysztof Jozwik, and Yuko Hara for their insightful and helpful comments. This work is partly supported by Core Research for Evolutional Science and Technology project from Japan Science and Technology Agency.

References

[1] R. Bird, *Introduction to functional programming using Haskell (second edition)*. Prentice Hall, 1998.

[2] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," In *International Symposium on Low Power Electronics and Design*, pp. 197–202, 1998.

[3] R. Jain, D. Molnar, and Z. Ramzan, "Towards a model of energy complexity for algorithms," In *Wireless Communications and Networking Conference*, vol. 3, pp. 1884–1890, 2005.

[4] R. Jain, D. Molnar, and Z. Ramzan, "Towards understanding algorithmic factors affecting energy consumption: switching complexity, randomness, and preliminary experiments," In *Foundations of Mobile Computing*, pp. 70–79, 2005.

[5] A. J. Martin, "Towards an energy complexity of computation," *Inf. Process. Lett.*, 77(2-4):181–187, 2001.

[6] D. R. Musser, "Introspective sorting and selection algorithms," *Softw. Pract. Exper.*, 27(8):983–993, 1997.

[7] D. Shin and J. Kim, "Optimizing intratask voltage scheduling using profile and data-flow information," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):369–385, 2007.

[8] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," In *International Conference on Computer-aided Design*, pp. 384–390, 1994.

[9] J. D. Valois, "Introspective sorting and selection revisited," *Softw. Pract. Exper.*, 30(6):617–638, 2000.