Towards Understanding Algorithmic Factors Affecting Energy Consumption: Switching Complexity, Randomness, and Preliminary Experiments

Ravi Jain
jain@docomolabs-usa.com

David Molnar
dmolnar@gmail.com

Zulfikar Ramzan
ramzan@docomolabs-usa.com

ABSTRACT

Mobile devices consider energy to be a limiting resource. Over the past decade significant research has gone into how one can reduce energy consumption at the hardware level, network protocol level, operating system level, and compiler level. In almost all algorithm analysis, a single resource such as time or communication is often taken as a proxy for energy. We address this problem by defining an algorithmic model for energy, designing algorithm variants that reduce energy cost in this model, and then performing preliminary experiments to test the model.

Our starting point is an algorithmic energy model inspired by work from the compilers community [26]. Augmenting and simplifying this model motivates the need to consider an algorithm's "switching" complexity; this measure captures the extent to which one switches between different functional units during execution. We carry out preliminary experiments on the Itsy pocket computer, which contains a StrongARM SA-1100 processor running Linux, to compare "high-switch" versions of bubble sort and other algorithms to optimized "low-switch" versions. Our preliminary results show that switching does not appear to affect energy consumption at the algorithmic level.

We then look at a factor that does not appear to have been studied, namely the cost of generating (pseudo)random bits. Derandomization is a goal in cryptography and complexity theory. To our knowledge the energy cost of randomness itself has not been studied. Nonetheless, many mobile protocols and algorithms might utilize randomness, for example to handle contention resolution, generate cryptographic keys, or to accomplish efficient sorting. We consider three common mechanisms for generating randomness: the standard C library random number generator, and the Linux /dev/random, and /dev/urandom generators. We perform tests that compare the energy consumed by these generators compared to the cost of performing basic arithmetic instructions. We use Quicksort as an example of a classic basic application-level algorithm to understand the energy cost of randomness, and compare the energy consumed by randomized Quicksort to standard Quicksort. Our preliminary results show that generating randomness does in fact incur a significant energy cost, and /dev/random is the most expensive of the three mechanisms.

We conclude that understanding energy consumption at the algorithmic level is an important but overlooked area of investigation, and discuss the implications of our results. We end with directions for for further work.

Categories and Subject Descriptors: F.2.3 [Analysis of Algorithms and Problem Complexity]: Tradeoffs between Complexity Measures

General Terms: Algorithms, Experimentation, , Measurement, Theory.

Keywords: Switching Cost, Energy Measurement, Randomness Cost.

DIALM-POMC'05, September 2, 2005, Cologne, Germany.

1. INTRODUCTION

Energy is a critical resource for battery-powered mobile wireless devices. In fact, since the CPU speed, bandwidth, and storage available to such devices are increasing rapidly, but battery capacity is increasing at the rate of only a few percent per year, it is likely that energy will be the most critical resource in future devices, and an *energy bottleneck* appears to be looming. It is thus important to study energy consumption, and ways to reduce it, at all system levels.

We want a method that will yield results independent of particular applications. Therefore we focus on the energy complexity of algorithms. This is analogous to the manner in which the time consumption (e.g., in terms of CPU operations) of applications has been studied by looking at the time required for basic algorithms such as sorting, searching, and so on. Obviously, an application's energy consumption is not entirely reflected in the sum of the energy consumption of its basic algorithms, but studying an algorithm's energy consumption is a useful way to partition the problem.

To our knowledge, relatively little work focuses on understanding energy consumption at the algorithmic level. There has, however, been a great deal of work over the past decade on ways of reducing energy at various other system levels; e.g., hardware, operating system, network protocol, and compiler levels. We use this work to guide our study of algorithmic-level energy consumption. Clearly, there is no

such thing as an algorithmic "level" in the computer organization sense; algorithms are used at all levels. When we say "algorithmic level" we mean that we focus on the methods used for analysis and creation of new algorithms. In this paper, we will also focus on staple algorithms: sorting, generating random numbers, and the like. The main previous work of which we are aware is Naik and Wei [7], who compare, as we do, several different algorithms for the same task by their energy consumption. Naik and Wei explain their results, however, by appealing to smaller constant factors in time cost hidden by the traditional big-Oh notation; we go further to investigate factors beyond time cost.

To start our study of energy consumption at the algorithmic level, we look at the way time consumption has been studied as an analogy. Like time, an algorithm's energy consumption is not a fundamental quantity, it is the quantity that we want to estimate. It is determined by fundamental operations encoded in the algorithm itself, such as CPU operations. The CPU operations themselves can be represented at various levels, ranging from bit-level boolean operations to complex assembly-level instructions such as "Fetch and Add." What level of abstraction should we use? Thus we require a model that is abstract enough to be tractable, yet reasonably accurate in the sense that it allows one to compare the energy consumption of different algorithms, particularly in an asymptotic sense. For time, a model that has become widely accepted is the RAM model of computation [22]. Indeed, the analysis of time and space complexity at the algorithmic level, particularly asymptotic analysis and the big-Oh notation, is an essential part of the toolkit of theoreticians and practitioners alike.

Our principal goal in this paper is to try to identify some of the factors that would be important to consider for an algorithmic energy model, and to quantify their importance. We stress that the current study is preliminary; we do not claim to have the final model for the job. Instead, we show how such a model could be defined and measured against experimental evidence.

Clearly the number of CPU operations is an important factor in an algorithm's energy cost. While I/O operations are typically ignored when estimating the time required for an algorithm, they cannot be ignored for energy. The ratio between the cost to communicate a single bit and the cost of executing a single instruction can be as high as 1000 to 1. Therefore, many researchers focus on algorithms that minimize communication in order to minimize energy, and neglect computation cost. Recently, however, Barr and Asanovìc analyzed the issue of compressing data before sending it. They found that the amount of computation may outweigh the savings in communication [4]. Through extensive experimentation, they showed that both communication and CPU operations must be considered to reduce total energy cost.

More generally, energy consumption involves a complicated tradeoff between several factors, including CPU operations, memory accesses, I/O (and communication) operations, and others. Using any one factor alone, such as time, as a proxy for energy, is likely to result in a poor estimator. Even using two factors, such as time and communication, may lead us astray if another factor contributing to energy consumption is dominant.

To account for and understand the impact of different factors we need a model of algorithmic energy complexity that reflects all significant energy-consuming components of the device architecture. At the same time, the kinds of tradeoffs we make in algorithms are not necessarily the kind made in other layers, such as the MAC, compiler optimization, or OS layers. Therefore, some factors may not, in fact, be useful for algorithm design.

We start with previous work on energy-optimizing compilers that uses an instruction-level model based on the Harvard architecture; the Harvard architecture is similar to the so-called von Neumann architecture except that it assumes separate pathways connecting the CPU to instruction memory and to data memory. Steinke et al. [26] verify this instruction-level model empirically on the ARM7TDMI platform. In section 3 we augment the model to include the effect of I/O and simplify it to ignore the effect of differences between consecutive values that are sent on the pathways; we call the result the Augmented Simplified Harvard Architecture (ASHA) model.

It is possible to show that ASHA is theoretically well-behaved, and we can derive "energy" analogues for various fundamental time-complexity theorems (e.g., gap and speed-up theorems) using classical proof techniques. We do not dwell on this aspect of ASHA in this paper. Instead, we focus on our preliminary experiments to determine which factors, other than CPU and I/O operations, significantly contribute to energy consumption.

One of the energy factors that has been studied previously is the notion of switching cost, namely the energy consumed when switching between one functional unit to another. We study this cost in an experimental setup on the Itsy pocket computer which contains a StrongARM SA-1100 processor running at 59 MHz [5]. Our initial experiments indicate that while this cost may be important at lower system levels, it does not seem significant at the algorithmic level.

In contrast, the energy cost of generating random bits has not been systematically considered in depth. Randomness has proven to be a powerful tool for developing efficient algorithms as well as analyzing them, particularly distributed algorithms. One important use of randomness in mobile wireless devices is in network protocols which use backoff algorithms to resolve contention. Another is for cryptographic functions, such as key generation. In fact, there has been substantial prior work focused on developing energy-efficient network protocols and algorithms that use randomness as a fundamental building block. We carry out some preliminary experiments in our testbed to quantify the energy cost of randomness for a variety of random number generators, and find it to be a significant factor.

In summary, our contributions in this paper are as follows. We argue that understanding energy consumption at the algorithmic level is an important but overlooked area in the search to find ways to mitigate the energy bottleneck in mobile wireless devices. Based on previous experimental work in the compilers community, we introduce an initial algorithmic model of energy complexity, ASHA. We then focus on studying the important factors that contribute to the energy consumed at the algorithmic level, and carry out experiments to quantify them. We find one "negative" result, in the sense that we show that switching between functional units, which has been shown to be significant at the compiler level, is in fact not significant for algorithm design, at least in our experiments on the Itsy pocket computer. We also find one "positive" result, which is that the energy cost

of generating random bits is quite significant and should not be ignored when designing randomized algorithms. We believe that these results, as well as our approach, can form the foundation for a deeper and broader study of the factors affecting energy consumption in mobile wireless devices.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 motivates the need for studying switching complexity via a a model that simplifies and augments existing work in the compilers community. Section 4 expounds on two algorithmic factors in energy consumption: switching complexity and randomness. Section 5 describes our experimental testbed and methodology, as well as the specific experiments we ran. Section 6 discusses the results from our experiments. Finally, section 7 makes concluding remarks.

2. RELATED WORK

Work on improving the energy consumption on embedded devices has been done at four levels: the logic design level, the processor level, the operating system level and the compiler level. Due to space limitations, and since there are numerous papers discussing these topics (e.g., see [1, 3, 9, 10, 11, 13, 18, 25, 29, 30, 31] and the references therein) we omit a discussion of the work on reducing energy complexity at these levels in the paper. In addition, a considerable amount of work has been done in reducing energy at various levels of the networking protocol stack. We will also not discuss this work here due to space limitations, but refer the reader to some recent papers and references therein [16, 6, 4, 28]. This section discusses some other work that is more germane to our study of algorithmic energy complexity.

Time-Energy Cost Models. Martin [20] sought to develop a complexity theory of energy. He argues that an overall energy-time cost measurement for an algorithm should be $E \times t^2$ (where E is the energy used and t is the time used) as opposed to $E \times t$ (which others have used). The reasoning is as follows. Time is directly related to voltage in that if one reduced the voltage in a processor by a certain factor, than the amount of time needed for execution would essentially go up by that same factor. Energy, on the other hand, is directly proportional to the square of the voltage. So, suppose you had two algorithms A_1 and A_2 where the time required by A_1 is twice that required by A_2, but the energy consumed by A_1 is half that consumed by A_2. According to the measure $E \times t$, these algorithms have equivalent cost. But, if you halve the voltage on the machine running A_2, then both A_1 and A_2 would run in the same amount of time. However, the energy consumed by A_2 would go down by a factor of 4 (because of the quadratic relationship between energy and voltage). Martin left open the question of how one could treat energy as a complexity-theoretic resource at the algorithmic level. Further, he did not consider factors beyond time, as we have here. Our study follows on our preliminary theoretical investigation, which considered an "augmented Turing Machine" model for energy cost and proved an energy hierarchy theorem for augmented Turing Machines [15]. The augmented Turing Machine model, however, is far removed from practical implementation.

3. THE AUGMENTED SIMPLIFIED HARVARD ARCHITECTURE (ASHA) MODEL

This section motivates the consideration of switching complexity as an algorithmic factor that might affect energy consumption. We begin by describing a Harvard-architecture based energy model due to Steinke et al. [26], hereafter referred to as the SKWM model. This model, which extends the model of Tiwari et al. [27], was motivated by work on energy-optimizing compilers. We consider *augmenting* the model to take I/O considerations into account since these play an important factor in energy consumption. Since the SKWM model is compiler oriented, it is difficult to reason about it at the algorithmic level. Therefore, we make a number of seemingly reasonable simplifying assumptions, which suggests three important algorithmic factors for understanding power consumption: time complexity, I/O complexity, and switching complexity. The last is "non-traditional," and captures the extent to which an algorithm might switch between using different functional units in a microprocessor.

3.1 Preliminaries

The SKWM model assumes a Harvard architecture for the underlying processor (though, it trivially adapts to von Neumann architectures), and considers the following factors:

1) **Actual instructions that could be executed:** executing a given instruction incurs a base cost. 2) **Different instruction schedules:** a pair of consecutive instructions using different functional units is likely to consume more energy than a pair using the same functional unit. This is due to the fact that functional units may be deactivated when not in use, thereby reducing the energy consumption. 3) **Memory hierarchy:** If a system includes an off-chip memory in addition to an on-chip memory, then placement in memory can impact energy (since loading from off-chip memory is likely to consume more energy than loading from on-chip memory). 4) **Bit toggling on busses:** Energy consumption increases with the number of bits toggled on the bus.

The impact of each of these factors are termed the *parameters* of the energy model. These parameters have to be derived experimentally on a given processor, perhaps by running sufficiently many code samples.

To specify the model in more detail, we introduce some notation. For a bit string x, let $w(x)$ denote its Hamming weight; i.e., the number of 1's in x. We will weight $w(x)$ with parameter α_i. If x and y are two bit strings of the same length, then $h(x, y)$ denotes the Hamming distance; i.e., $h(x, y) = w(x \oplus y)$. We will weight $h(x, y)$ with parameter β_i. If x represents an instruction, we denote by $BaseCPU(x)$ and $BaseMem(x)$ the base costs associated with the execution of x within the CPU and memory respectively. Finally we denote by $FUChange(x, y)$ the cost associated with activating / deactivating a functional unit if x is executed just prior to y. Since these last three functions represent actual costs, we need not weight them. Now, there are four parts to the SKWM model: E_{cpu_instr} – the instruction-dependent costs inside the CPU; E_{cpu_data} – the data dependent costs inside the CPU; E_{mem_instr} – instruction dependent costs in the instruction memory; and E_{mem_data} – data-dependent costs in the data memory. Finally, SKWM sums these components to model total energy consumption:

$$E_{total} = E_{cpu\_instr} + E_{cpu\_data} + E_{mem\_instr} + E_{mem\_data}.$$

We can now compute energy consumption for a sequence of m instructions. We assume that the i^{th} instruction $Instr_i$ ($1 \leq i \leq m$), located at address $IAddr_i$, is broken up into four components: an opcode (denoted $Opcode_i$); t register locations (denoted $Reg_{i,1}, \ldots, Reg_{i,t}$); t Register values (denoted $RegVal_{i,1}, \ldots, RegVal_{i,t}$); and s immediate values (denoted $Imm_{i,1}, \ldots, Imm_{i,s}$). Now, we can write:

$$\begin{aligned} E_{cpu\_instr} = \sum_{i=1}^{m} & BaseCPU(Opcode_i) \\ & + FUChange(Instr_{i-1}, Instr_i) \\ & + \alpha_1 \times w(IAddr_i) \\ & + \beta_1 \times h(IAddr_{i-1}, IAddr_i) \\ + \sum_{j=1}^{s} & (\alpha_2 \times w(Imm_{i,j}) \\ & + \beta_2 \times h(Imm_{i-1,j}, Imm_{i,j})) \\ + \sum_{k=1}^{t} & (\alpha_3 \times w(Reg_{i,k}) \\ & + \beta_3 \times h(Reg_{i-1,k}, Reg_{i,k})) \\ + \sum_{k=1}^{t} & (\alpha_4 \times w(RegVal_{i,k}) \\ & + \beta_4 \times h(RegVal_{i-1,k}, RegVal_{i,k})) \end{aligned}$$

The SKWM model computes the data-dependent CPU costs when making n data accesses. These costs depend on the i^{th} data address and data item (denoted $DAddr_i$ and $Data_i$ respectively), as well as the direction dir depending on whether a read or write is occurring.

$$\begin{aligned} E_{cpu\_data} = \sum_{i=1}^{n} & \alpha_5 \times w(DAddr_i) \\ & + \beta_5 \times h(DAddr_{i-1}, DAddr_i) \\ & + \alpha_{6,dir} \times w(Data_i) \\ & + \beta_{6,dir} \times h(Data_{i-1}, Data_i) \end{aligned}$$

Next, the SKWM model computes the instruction dependent costs in the instruction memory. Here we denote by $Word\_width_i$ the bit width of the i^{th} instruction.

$$\begin{aligned} E_{mem\_instr} = \sum_{i=1}^{m} & BaseMem(InstrMem, Word\_width_i) \\ & + \alpha_7 \times w(IAddr_i) \\ & + \beta_7 \times h(IAddr_{i-1}, IAddr_i) \\ & + \alpha_8 \times w(IData_i) \\ & + \beta_8 \times h(IData_{i-1}, IData_i) \end{aligned}$$

Finally, SKWM computes the data-dependent memory costs:

$$\begin{aligned} E_{mem\_data} = \sum_{i=1}^{n} & BaseMem(DataMem, Word\_width_i) \\ & + \alpha_9 \times w(DAddr_i) \\ & + \beta_9 \times h(DAddr_{i-1}, DAddr_i) \\ & + \alpha_{10,dir} \times w(Data_i) \\ & + \beta_{10,dir} \times h(Data_{i-1}, Data_i) \end{aligned}$$

Augmenting the SKWM Model. We can augment the SKWM Model to incorporate additional I/O devices. To be general, we can imagine that the underlying processor has a Harvard architecture, but augmented to include I/O devices – as is depicted in fig. 1.

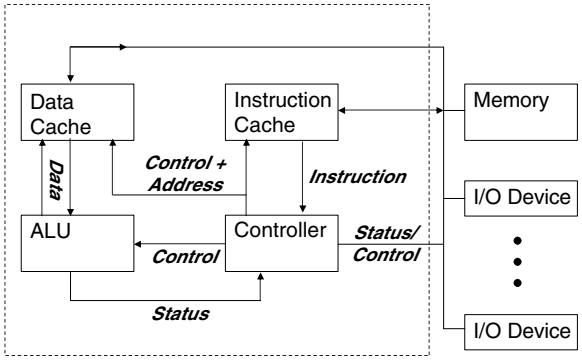

Figure 1: The I/O Augmented Harvard Architecture

Assume that we have ℓ I/O devices; $BaseIO(k, dir)$ denotes the base cost of the I/O device on a single operation for $1 \leq k \leq \ell$. Suppose that r I/O device calls are made, and let $D(i)$ denote the I/O device associated with the i^{th} call for $1 \leq i \leq r$. We further denote by $IOAddr_i$ the address used to specify the I/O device – this is natural for memory-mapped I/O; however, in the case of non-memory mapped I/O, there must be some mechanism (such an using dedicated I/O instructions in the processor) to specify which device is being used. In either case, some information in the form of a bit string is needed for this purpose. In general, $IOAddr_i$ will be used to denote this bit string. Finally, we denote by $IOData_i$ the data associated with the i^{th} I/O call. Now, we can model the energy consumed by I/O devices as follows:

$$\begin{aligned} E_{IO\_data} = \sum_{i=1}^{r} & BaseIO(D(i), dir(i)) \\ & + \alpha_{11} \times w(IOAddr_i) \\ & + \beta_{11} \times h(IOAddr_{i-1}, IOAddr_i) \\ & + \alpha_{12,dir} \times w(IOData_i) \\ & + \beta_{12,dir} \times h(IOData_{i-1}, IOData_i) \end{aligned}$$

Finally, in the augmented model, we have:

$$\begin{aligned} E_{total} = & E_{cpu\_instr} + E_{cpu\_data} \\ & + E_{mem\_instr} + E_{mem\_data} + E_{IO\_data} \end{aligned}$$

3.2 Simplifying the Augmented SKWM Model

The SKWM model is fairly complex from an algorithmic perspective, especially since it considers a variety of parameters. Because the model uses various measures related to Hamming weights and distances, it is not clear, for example, how to reason about this model when we do not know a-priori the data on which an algorithm operates or the memory addresses accessed. We propose an Augmented Simplified Harvard Model (ASHA) that neglects some of the factors that make analysis difficult. This is akin to how complexity theorists have ignored constant factors in time and space complexity – while the results are less accurate, the models have become widely used and are still "good enough"

to provide insight into how algorithms perform with respect to the relevant performance metrics.

THE ASHA MODEL. In the ASHA model, we only take into account three factors: 1) the base cost for the instruction – denoted $BaseCPU(\cdot)$); 2) the cost associated with switching between functional units – denoted $FUChange(\cdot,\cdot)$; and 3) the cost associated with I/O calls – denoted $BaseIO(\cdot,\cdot)$. We can now do simpler computations:

$$E'_{cpu\_instr} = \sum_{i=1}^{m} BaseCPU(Opcode_i) + FUChange(Instr_{i-1}, Instr_i)$$

$$E'_{IO\_data} = \sum_{i=1}^{r} BaseIO(D(i), dir(i))$$

And finally, the energy consumption evaluates to:

$$E'_{total} = E'_{cpu\_instr} + E'_{IO\_data}$$

The simplifications do not end here. In particular, suppose that we made $BaseCPU(\cdot)$ directly proportional to the time required for the instruction (multiplied by some constant factor, call it ρ_1). Suppose also that we assume all I/O calls to have the same cost – call it ρ_2. Finally, suppose that we assume all switches between different functional units to have the same cost, ρ_3 (and that there is no cost if the functional units are the same). Then, if an algorithm on an n-bit input executes in $T(n)$ time steps, makes $I(n)$ I/O calls, and $S(n)$ functional unit switches, we have a far less complicated expression for energy complexity[1]:

$$E''_{total} = \rho_1 \times T(n) + \rho_2 \times I(n) + \rho_3 \times S(n).$$

Naturally, this expression is considerably simplified. However, it leverages quantities that algorithm designers may have already analyzed; namely, the time complexity and the I/O complexity[2]. This expression suggests examining a third quantity: the "switching" complexity $S(n)$. Of course, factors like whether or not specific branches are taken affect the complexities. As in the traditional complexity-theoretic practice, one can measure the worst case, average case, and possibly the best case for $T(n), I(n), S(n)$.

4. SWITCHING AND RANDOMNESS

We now use sorting as an algorithmic test case to study the impact of switching complexity. We also consider the cost of generating randomness in efficient sorting. We note that Naik and Wei also considered the energy consumption of different sorting algorithms, including quicksort; we extend that work by considering switching and randomness energy cost as well as time cost [7].

[1]For simplicity, the formula only expresses one I/O device; we can trivially extend it to multiple devices, at a corresponding increase in the formula's descriptive complexity.

[2]With respect to the I/O complexity, one often counts the number of parallel I/O calls (c.f., [24]). However, we are further interested in the actual number of individual calls I/O devices. For example, if $c(n)$ I/O calls are made and each call does p parallel I/O operations, then the I/O complexity for our purposes is $p \times c(n)$ whereas the I/O complexity in the model studied in [24] is $c(n)$.

4.1 Switching Complexity

ANALYZING SWITCHING COMPLEXITY FOR SIMPLE ALGORITHMS. We now describe a sample analysis of algorithms for switching complexity. Refer to Algorithm 1 for a simple bubble sort example. We use the term "High Switch" for this version since it has higher switching complexity than the other version we analyze. We consider the following functional states: Move, Compare & Branch, ALU, Load, and Store. We treat Compare and Branch as the same unit (since for our purposes all compares will lead to a branch); however, for clarity of exposition in describing the analysis, we use both terms.

Algorithm 1 Bubble Sort (High Switch)

```
1: for i = 1 to n − 1 do
2:   for j = i + 1 to n do
3:     if A[i] > A[j] then
4:       temp = A[i]
5:       A[i] = A[j]
6:       A[j] = temp
7:     end if
8:   end for
9: end for
```

In the outer loop we perform the following:

Functional Unit	Algorithmic Step
Move	(set a value for i)
Compare	(is $i \leq n-1$?)
ALU	(compute $i+1$)
Perform inner Loop	
ALU	(increment loop counter)
Branch	(back to the start of the loop)

In the inner loop we perform the following:

Functional Unit	Algorithmic Step
Move	(set $j = i+1$)
Compare	(is $j \leq n$?)
Load	(obtain value of $A[i]$)
Load	(obtain value of $A[j]$)
Compare	(is $A[i] > A[j]$?)
Load	($A[i]$ is assigned temp)
Load	(($A[j]$) Assign to $A[i]$)
Store	(temp into $A[j]$)
ALU	(increment j)
Branch	(back to start of inner loop)

In the inner loop there are 8 switches (a consecutive pair of loads occurs twice and we don't switch between). This count also incorporate the switch between the branch at the end and going back to the start of the loop. The outer loop introduces 6 switches. Again, this count incorporates the switch between the final branch and the start of the outer loop. Now, observe that the inner loop is executed $n-(i+1)+1 = n-i$ times and the outer loop executed $n-1$ times. Therefore, the total switching complexity is:

$$\sum_{i=1}^{n-1}(8(n-i)+5) = 4n^2 - 7n - 5$$

Now, we determine the switching complexity for the other bubble sort as referred to in Algorithm 2.

Algorithm 2 Bubble Sort (Low Switch)

```
1: for i = 1 to n − 1 do
2:   for j = i + 1 to n do
3:     temp1 = A[i]
4:     temp2 = A[j]
5:     if temp1 > temp2 then
6:       A[i] = temp2
7:       A[j] = temp1
8:     end if
9:   end for
10: end for
```

Start Unit	End Unit	Switching Overhead
ALU	Load / Store	2.2 mA
Multiplier	Load / Store	2.5 mA
BarrelShifter	ALU	3.3 mA
Register File	ALU	3.8 mA
Register File	Multiplier	2.1 mA

Table 1: Switching overhead between ARM7TDMI functional units, given by Steinke et al.

In the outer loop, we perform the following:

Functional Unit	Algorithmic Step
Move	(set value for i)
Compare	(is $i \leq n-1$?)
ALU	(compute $i+1$)
Perform inner Loop	
ALU	(increment loop counter)
Branch	(back to start)

In the inner loop, the following steps happen.

Functional Unit	Algorithmic Step
Move	(set j to $i+1$)
Compare	(is $j \leq n-1$?)
Load	(obtain value of $A[i]$)
Load	(obtain value of $A[j]$)
Compare	(is $temp1 < temp2$?)
Store	($A[i] \leftarrow temp2$)
Store	($A[j] \leftarrow temp1$)
ALU	(increment loop counter)
Branch	(to inner loop start)

Let us analyze the cost. The outer loop introduces 5 switches (as before) and the inner loop introduces at most 7. Therefore, the worst-case switching complexity is: $\sum_{i=1}^{n-1}(7(n-i)+5) = (3.5)n^2 - (5.5)n - 5$. The savings, in terms of switches, between these two is: $(0.5)n^2 - (1.5)n$.

Impact of Switching Complexity. The next key question is how much switching overhead is involved in a typical processor for mobile devices, such as an ARM7. Such numbers already exist in the literature when considering the compiler level. In particular, table 1 provides numbers taken from the work of Steinke et al. [26].

By itself, however, such "micro"-cost numbers do not tell us much. We are interested in overall energy consumption; other factors may dominate the switching complexity. Switching complexity may also interact with other design choices at lower processor levels in ways that are not predicted by the instruction-to-instruction energy cost. At lower logic levels, a mechanism like clock gating can be exploited when there is low switching complexity. Likewise, energy optimizations associated with turning off parts of state machines can also be applied. Similarly, the Hamming distance between consecutive bus values might be smaller since the use of the same functional unit suggests that instruction op-codes will be similar or even identical. At the processor level, decreasing switching complexity by grouping calls to specific I/O devices will save energy since the devices can stay in a sleep state for a longer period of time.

One approach to quantifying the impact of switching complexity would be to estimate the "switching coefficient" for the ASHA model, using the methods outlined above. We decided, however, to try something even simpler as a first step: compare the total energy consumption of "standard" algorithms to the consumption of those optimized to have low switching cost. *Our preliminary experiments indicate that for our testing environment, switching complexity does not seem to play a role at the algorithmic level for the Itsy processor.* There are several possible reasons for this. First, switching complexity really matters when a processor is intelligent enough to turn off functional units that are not being used. Second, for switching complexity to be meaningful, the switches might have to be for long durations; e.g., spending a lot of time in one functional unit before switching is what saves time. If the switches happen rapidly, then it may not make sense to turn off one processor unit.

4.2 Incorporating Randomness

We used sorting as a basic example when describing how to go about measuring switching complexity. We now use sorting as an example of the issues in measuring the energy cost of randomness. Sorting is a fundamental process used in numerous applications. Bubble sort is fairly slow, and there are other algorithms that outperform it asymptotically; in particular, quicksort is among the fastest algorithms for sorting arbitrary data. Recall that in quicksort a pivot element is first chosen. The array is arranged so that all elements smaller than the pivot are placed to its left, and the remaining elements are placed to its right. For an n-element array, this step requires $O(n)$ work. Next, quicksort is recursively applied to both the portion of the array before the pivot and the portion after the pivot. See algorithm 3. If these two portions are always the same size, then there are only $O(\log n)$ levels of recursion, and the entire array is sorted using only $O(n \log n)$ comparisons. However, if the pivot results in lopsided portions, then more recursive levels are required. In the worst case, $O(n^2)$ comparisons are required. Thus, the pivot choice is central to achieving the best possible running time.

Algorithm 3 Quicksort(Array A, index i, index j)

```
1: Pick pivot index p ← {i, i + 1, ..., j};
2: Partition A around A[p];
3: Let k be the index of the pivot element after partitioning;
4: Quicksort(A, i, k − 1);
5: Quicksort(A, k + 1, j);
```

Therefore, one approach to quicksort involves choosing the pivot randomly. This is known to yield $(2\ln 2) \cdot n \log_2 n - \Theta(n)$ expected comparisons. Also, the random bits used

must be chosen from a reasonably good source. If, for example, a linear congruential generator is used, then we may experience $\Omega(n^2)$ worst-case behavior [17]. A more general study on randomized quicksort with a low-entropy random source may be found in [19]; in particular, as the entropy of the source decreases, the worst-case running time essentially approaches $\Omega(n^2)$. Therefore, as we detail in section 5, much of our experimental effort focuses on quicksort.

Random bits are used to improve the performance of numerous algorithms; see the text of Motwani and Raghavan for a wealth of examples [21]. Beyond their algorithmic uses, random bits are used in numerous other settings in computer science. For example, as we intimated above, cryptographic keys require random bits; further these bits must be chosen in a way that makes it intractable for an adversary to learn any information about them. In wireless networking, the IEEE 802.11 MAC protocol [14] uses CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) as part of its Distributed Coordination Function mechanism. A sender wishing to transmit data picks a *random* backoff value between 0 and its contention window. Clearly, one could mount a trivial denial of service attack if the backoff values can be determined. Therefore, they should be chosen using a strong (unpredictable) random source.

5. EXPERIMENTS

This section describes the experiments we performed to measure the energy consumption associated with switching complexity and randomness generation. We first describe our experimental set up. Next, we describe experiments to measure the relevance of switching complexity. Then, we turn our attention to randomness generation. We begin by describing several random number generators available on Linux. Finally, we describe specific experiments. A discussion of the actual results, and the methodology used to obtain them, is presented in Section 6

The Experimental Setup. Our experimental testbed consists of a Compaq Itsy handheld pocket computer running Linux 2.0.30. The Itsy v2 is based on the StrongARM SA-1100 processor, and has 32MB of DRAM, 32MB of flash memory, an LCD screen (320x200 pixels grayscale), a touch screen, audio input and output, a rechargeable lithium-ion battery, and several serial interfaces. While the Itsy's frequency is scalable, our experiments were conducted at 59MHz. An Agilent 6611C power supply is connected to the Itsy; the power supply is used instead of the battery. For all experiments, we set the power supply to 3.75 V. A National Instruments SCB-68 breakout box connects to the lines from the power supply both before and after a 100mΩ resistor in the Itsy; the resistor is in series with the Itsy battery. The breakout box is in-turn connected to a National Instruments DAQCard 6036E, which monitors the voltage drop across the resistor. The DAQCard is a PCMCIA card, and is connected to a 1.6GHz Pentium Mobile Sony Vaio laptop. The laptop ran Windows XP Professional. National Instruments Measurement & Automation Explorer version 3.0.2.3005 was used to record the power measurements. We connected to and commanded the Itsy via a serial port from a separate desktop (an HP desktop with a 3GHz Pentium 4 running Linux version 2.6.3).

All test programs were compiled on the HP desktop using gcc version 3.3.2; each program put the procesor to sleep for one second between iterations of the algorithm. See fig. 2 for a diagram of our experimental setup. For all of our experiments, we derived energy consumption from the power trace as follows. First, we eliminated samples below 60 mW; this screened out samples from the processor sleep times. Then we truncated traces for two programs in the same experiment to the same length; this corrected for the fact that some traces ran longer than others. Finally, we summed the power consumed at each trace sample to find the total energy consumption. In all our experiements, we report relative energy consumption; the most expensive trace is normalized to 1 and other traces reported as a fraction of this baseline cost.

Switching Complexity Experiments. We focused first on experiments related to switching complexity. The programs bubble_high and bubble_low are the two bubble sort implementations described above; the former has a higher number of switches, and the latter a lower number. We show graphs from an example trace of bubble_high and bubble_low in fig. 3. We then show the relative energy cost in fig. 5. As we see, switching appears to have little or no impact.

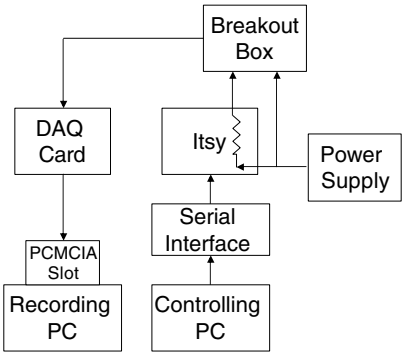

Figure 2: Our Experimental Testbed

In an effort to conduct a more extreme test of the impact of switching complexity, we wrote a program, switches-slower, that interleaves a sequence of additions and array assignment statement. This interleaving is designed to cause a large amount of "switching" between different processor units. In contrast, switches-1 computes an equivalent sequence of adds and array assignments, but performs all the add operations before the array assignments; in this second case, the switching complexity is low. Both programs

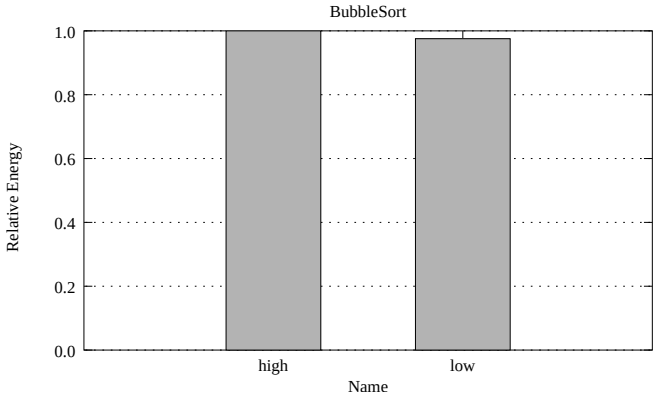

Figure 4: BubbleSort relative energy consumption.

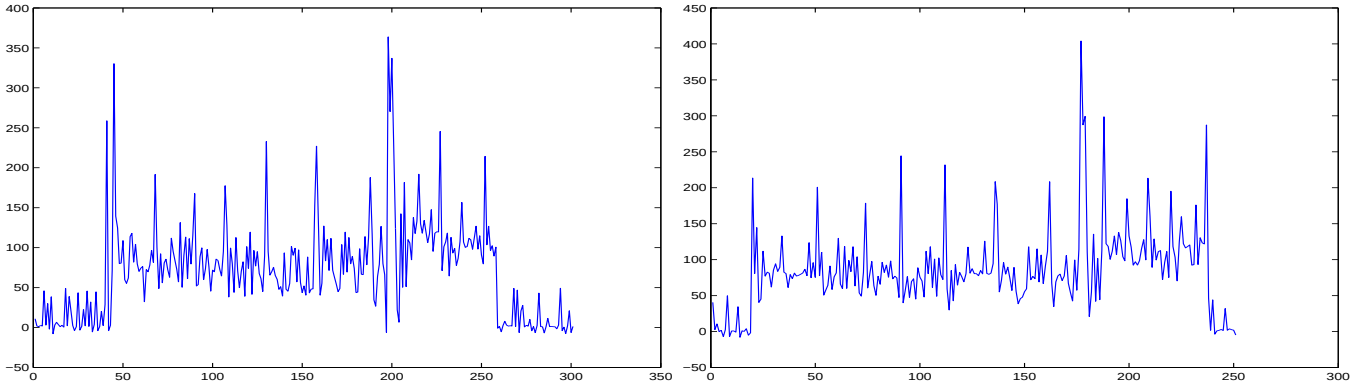

Figure 3: Traces from High (on left) and Low (on right) Switch Bubble Sort.

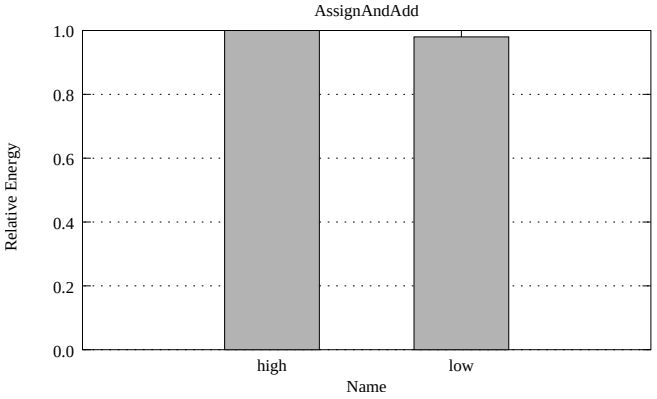

Figure 5: Switching relative energy consumption.

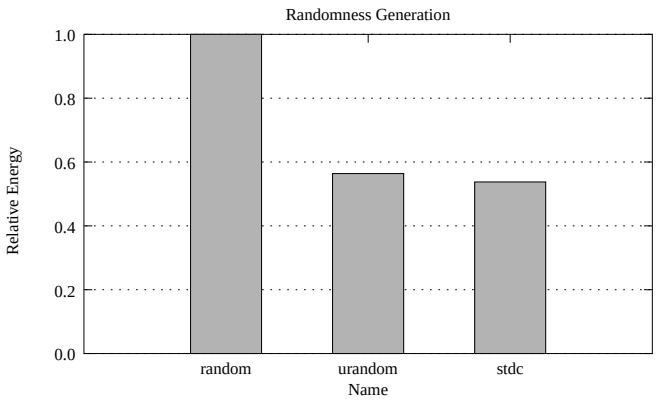

Figure 6: Relative cost of randomness sources.

seemed to require the same amount of energy; we show the results in fig. 5. We recognize that our experimental setup has a sampling rate too low to see the instantaneous effect of switching complexity, but we believe that the overall effect in total energy consumption is more important; our preliminary experiments do not show a significant savings for lower switching cost.

Linux Random Number Generators. We conducted our randomness generation experiments using three random number generators available on Linux: `random()` (the standard C library random number generator), /dev/random, and /dev/urandom. The standard C library generator is very simplistic, and not suitable for applications that require good random bits (e.g., generating cryptographic keys). In fact, it may not even be suitable for use in a randomized algorithm. It works by replacing a seed value with a simple arithmetic transformation of it, and then outputting another arithmetic transformation of the seed. There is a default value for the seed, but the seed can be changed by the user. Both /dev/random and /dev/urandom are character devices that serve as an interface to the Linux kernel's random number generator. They were designed to be used in applications where strong randomness is required; e.g., cryptography. They gather data from a variety of "unpredictable" system sources; e.g., inter-interrupt timings. This data is added to an entropy pool which is mixed using a primitive polynomial over $GF(2)$. When actual random bytes are desired, the MD5 hash function is applied to the entropy pool. As this happens, the amount of entropy in the pool is estimated. The key difference between /dev/random and /dev/urandom is that if the estimated entropy in the pool is less than what the user needs, then /dev/random will block and wait for the pool to be replenished; on the other hand, /dev/urandom applies MD5 to the entropy pool again, and returns an answer. In this sense, /dev/urandom does not provide the same level of randomness as /dev/random, but due to the properties of MD5, it is unlikely that an attacker can exploit this weakness in practice.

Measuring the Energy Consumed by Randomness Generation. Our first set of experiments were geared towards understanding the relative cost of generating random bits as compared to a basic addition operation. The programs std-crandom_512, std dev_random_512, and urandom_512, measure the cost of reading 512 bytes of randomness at a time using the standard C library generator, /dev/random, and /dev/urandom respectively. The results are summarized in the graph in fig. 6.

Our second set of experiments measured the cost of randomness generation within a common application: sorting. Recall that quicksort with a randomly chosen pivot can sort an n-element array using an expected $O(n \log n)$ comparison operations. We first set up twenty-five character arrays, each containing sixty-four elements, and randomly chose the elements to fill them. The experiments first assigned elements to the arrays, and then ran quicksort twenty-five times, once on each array. These operations were themselves contained in an infinite while loop, with a one-second sleep between iterations. The qsort program used the standard C library deterministic quicksort implementation. The exp_rqsort_itsy program is based on the same code as the standard C li-

Name	Mean (mW)	Stddev
sort-base-cost	155.2768	79.1525
qsort	242.8968	37.5129
rqsort	242.4834	34.2327
rqsort-II	242.2659	34.2307
devrqsort	145.2534	32.6342
devrqsort-buf4	146.7216	34.0659
devrqsort-buf8	146.6130	34.0959
devrqsort-buf16	146.5220	33.7784
devrqsort-buf32	146.1758	33.2338
devurqsort	233.3415	47.9378
devurqsort-II	241.9896	39.7807
devurqsort-buf32	246.2699	36.7392
devurqsort-buf32-II	246.1741	36.9452
devurqsort-buf4	247.8152	36.2923
devurqsort-buf4-II	248.2391	38.1438
devurqsort-buf4-III	248.6971	37.1571
devurqsort-buf8	247.4079	37.1755
devurqsort-buf8-II	246.4884	38.3804

Table 2: Quicksort Experiment Data

brary implementation, but replaces the pivot choosing step with a random pivot choice, using the C standard library generator to choose the pivot. The devrqsort and devurqsort programs are similar, except that the /dev/random and /dev/urandom generators are used respectively. Finally, to understand the relative cost of the actual sorting algorithms versus the cost of assigning elements to the arrays, etc., the sort_base_cost program performs the same operations as the other programs, without doing any sorting. As before, we took power traces of all programs at a sampling rate of 1000 samples per second, then filtered out samples less than 100 mW and took the mean and standard deviation of the left-over samples. The results are summarized in table 2.

6. RESULTS AND ANALYSIS

Switching Complexity. Our results do not support the claim that switching complexity is a significant contributor to algorithmic energy consumption. We first observed that the difference between our "high-switch" and "low-switch" variants of bubble sort was extremely small. The low-switch variant's energy consumption was roughly .98 of the high-switch variant. We then attempted to exaggerate the results of switching complexity by measuring switches-1 and switches-slower. As described above, switches-slower purposely interleaves assignment and ALU operations but produces the same end result as switches-1. Again, we found that a very small difference in mean energy consumption.

Randomness Generation. In contrast, we observed differences due to choice of random number generators. In our microbenchmarks, we observed that /dev/random took the longest, while /dev/urandom and the C standard random() call both took roughly half as long. We did, however, see little difference between /dev/urandom and the standard C random() call. See the bar chart in fig. 6. Some of this difference stems from the different behavior of the generators when reading "large" amounts of randomness: in particular, if not enough randomness is available in the kernel's entropy pool for a particular read, /dev/random will block and cause the processor to spin-wait until enough bytes are available.

In the case of quicksort, we measured average power consumption instead of total energy consumption. We found an initially surprising result: using /dev/random led to *lower* average consumption than using /dev/urandom or deterministic quicksort. For example, with no buffering, quicksort with /dev/random had a mean energy consumption of 145.2534 mW, while /dev/urandom had a mean energy consumption of 233.3415 mW. Deterministic quicksort, on the other hand, has a mean energy consumption similar to that of /dev/urandom. See Table 2 for full data.

We hypothesized that this phenomenon occurs since whenever the entropy pool is sufficiently low, /dev/random blocks. The machine consumes less energy when blocking than when doing quicksort-related operations. For /dev/urandom, on the other hand, there is no notion of blocking. Instead, it computes an MD5 hash of the current entropy pool and returns that. To validate the hypothesis, we ran a separate test on the Itsy to determine whether /dev/random blocked. This test involved outputting data to the standard output (in this case a terminal window on our command PC). We found that indeed /dev/random blocked before completing the 25 randomized quicksort calls in the first iteration of the infinite loop. Of course, /dev/urandom and the C random() call never blocked.

To see how this might play numerically, consider the following example. Suppose that during peak sorting the Itsy draws around 300 mW, but while blocking only about 200mW. The average /dev/random quicksort consumption will be closer to 200 mW because all it does is block. The /dev/urandom quicksort, on the other hand, will be closer to 300 mW because most of its time is spent sorting. Thus, /dev/random will consume less energy, but will not accomplish nearly as much. In the microbenchmark tests, however, we are measuring the energy consumed when actually generating randomness; in this case, /dev/random is more expensive. The upshot is that we believe /dev/urandom to be a better choice for practical purposes. Its energy consumption is comparable to that of the standard C library generator, but at the same time it does not suffer from blocking issues. While the bits it produces are not truly random, they are pseudorandom, so long as the output bits of MD5 are unpredictable and it hides information about its input bits (note that the recent hash function collision results [33] do not imply anything about the statistical properties of the hash function).

7. CONCLUSION

There are several possible directions for future work. The high-level direction is to find a better model for analyzing the energy cost of an algorithm. Our results suggest that the ASHA model as we have presented it is not the right model, because switching cost appears to make only a small difference in practical energy consumption. Along these lines, we can continue studying the energy cost of generating random bits. We can focus on specific application scenarios such as cryptography or network protocols, and determine the relative cost of random bit generation within these contexts. Then we can study the wealth of existing randomized algorithms and determine their "energy" complexity. Finally, there are many data structures that use randomness. A study of these with a view towards joint cache, timing, and randomness effects would also be a promising direction.

Overall, we believe that because energy might become the resource bottleneck in future applications, it should be stud-

ied at all levels. If algorithm designers are given the tools to reason about energy complexity, then they can make informed design choices that will have significant impact.

8. ACKNOWLEDGEMENT

We thank Lawrence Brakmo for a number of very helpful discussions on energy consumption and for his invaluable help in setting up our experimental testbed. We also thank the Mobisys and DIALM/POMC anonymous reviewers for valuable comments that significantly improved our work.

9. REFERENCES

[1] A. Abnous and J. Rabaey. "Ultra-low-power domain specific multimedia processors," *Proc. VLSI Signal Processing IX*, Nov 1996.

[2] B. Alpern, L. Carter, E. Feig, and T. Selker. "The Uniform Memory Hierarchy Model of Computation." *Algorithmica*, 12(2/3):72–109. Aug-Sep 1994.

[3] A. Vahdat, C. Ellis, and A. Lebeck. "Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency." In *Proc. 9th ACM SIGOPS European Workshop*, Sep 2000.

[4] K. Barr and K. Asanovic, "Energy Aware Lossless Data Compression", *Proc. MobiSys 2003.*

[5] W. R. Hamburgen, D. A. Wallach, M. A. Viredaz, L. S. Brakmo, J. F. Bartlett, C. A. Waldspurger, T. Mann, and K. I. Farkas. "Itsy: Stretching the Bounds of Mobile Computing." *Computer*, 34(4), Apr 2001.

[6] C. F. Chiasserini, P. Nuggehalli, V. Srinivasan and R. R. Rao, "Energy-Efficient Communication Protocols," (Invited), *Proc. Design Automation Conf*, Jun 2002.

[7] K. Naik and D.S.L. Wei, "Software implementation strategies for power-conscious systems." Mobile Networks and Applications, Volume 6, Issue 3 (June 2001). Pages 291-305. 2001.

[8] M. Dietzfelbinger. "Primality Testing in Polynomial Time From Randomized Algorithms to "PRIMES Is in P."" *LNCS Vol. 3000.* Springer Verlag, 2004.

[9] F. Douglis, F. Kaashoek, B. Marsh, R. Caceres, K. Li, and J. Tauber. "Storage Alternatives for Mobile Computers," *Proc. OSDI 1994.*

[10] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive Disk Spindown Policies for Mobile Computers," *Proc. USENIX Symposium on Mobile and Location Independent Computing*, 1995.

[11] M. P. Frank. "Reversibility for Efficient Computing," *Manuscript based on MIT Ph.D. Thesis.* Dec 1999.

[12] P. J. M. Havinga. "Mobile Multimedia Systems" Ph.D. thesis, University of Twente, Feb 2000.

[13] P.J.M. Havinga and G.J.M. Smit. "Design Techniques for Lower Power Systems," *Journal of Systems Architecture,* Volume 46, Number 1, 2000.

[14] IEEE Standard for Wireless LAN-Medium Access Control and Physical Layer Specification, P802.11, 1999.

[15] R. Jain, D. Molnar, and Z. Ramzan "Towards A Model of Energy Complexity for Algorithms." WCNC 2005.

[16] C. E. Jones, K. M. Sivalingam, P. Agrawal and J. C. Chen, "A Survey of Energy Efficent Network Protocols for Wireless Networks," *Wireless Networks*, vol. 7, no. 4, 343-358, July 2001.

[17] H. Karloff and P. Raghavan. "Randomized algorithms and pseudorandom numbers." *Proc. STOC 1998.*

[18] M. Koegst, G. Franke, S. Ruelke, and K. Feske. "Lower power design of FSMs by state assignment and disabling self-loops," *Proc. Euromicro 1997.* Pages 323–330, September 1997.

[19] B. List, M. Maucher, U. Schöning, R. Schuler. "Randomized Quicksort and the Entropy of the Random Number Generator." *Electronic Colloquium on Computational Complexity* Rept 59, 2004.

[20] A. J. Martin. "Towards an Energy Complexity of Computation," *Information Processing Letters,* 77 (2001) 181–187.

[21] R. Motwani and P. Raghavan. "Randomized Algorithms." *Cambridge University Press,* 1995.

[22] C. Papadimitriou. "Computational Complexity," *Addison-Wesley Publishing Company,* (Reprinted with corrections, 1995.)

[23] Rambus, Inc. `http://www.rambus.com.`

[24] E. Shriver and M. Nodine. "An introduction to parallel I/O models and algorithms," *in R. Jain, J. Werth and J. C. Browne, Input/Output in Parallel and Distributed Computer Systems,* Kluwer, 1996.

[25] M.R. Stan and W.P. Burleson. "Bus-Invert Coding for Lower-Power I/O." *IEEE Trans. on VLSI Systems,* Volume 3, Number 1, pp 49–58, 1995.

[26] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. "An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations," *Proc. PATMOS 2001.*

[27] V. Tiwari, S. Malik, and A. Wolfe. "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Volume 2, Number 4, December 1994.

[28] J. Subramanian, M. Bs and S. R. Murthy, "On Using Battery State for Medium Access Control in Ad hoc Wireless Networks," *Proc. Mobicom*, 2004.

[29] J. Flinn and M. Satyanarayanan. "Energy-aware adaptation for mobile applications." *Proc. SOSP 1999.*

[30] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. "ECOSystem: Managing Energy as a First-Class Operating System Resource," *Proc. Architectural Support for Porgramming Languages and Operating Systems (ASPLOS)*, 2002.

[31] J. Lorch and A. J. Smith. "Software strategies for portable computer energy management," *IEEE Personal Communications Magazine*, 5(3):60-73, June 1998.

[32] X. Wang, D. Feng, X. Lai, and H. Yu. "Collisions for Hash Functions MD4, MD5, HAVAL 128, and RIPEMD," *Cryptology E-print Archive*, report 199-2004. Available from http://eprint.iacr.org.

[33] X. Wang, H. Yu. "How To Break MD5 and Other Hash Functions," *EUROCRYPT 2005*