(Published in ACM Mobile Networks and Applications 6, 291-305, 2001)

Software Implementation Strategies for Power-Conscious Systems

Kshirasagar Naik
School of Computer Science
Carleton University
Ottawa K1V 0X2, CANADA
knaik@scs.carleton.ca

David S. L. Wei
Dept. of Comp. and Info. Science
Fordham University
Bronx, New York 10458, USA
wei@trill.cis.fordham.edu

Abstract

A variety of systems with possibly embedded computing power, such as small portable robots, hand-held computers, and automated vehicles, have power supply constraints. Their batteries generally last only for a few hours before being replaced or recharged. It is important that all design efforts are made to conserve power in those systems. Energy consumption in a system can be reduced using a number of techniques, such as low-power electronics, architecture-level power reduction, compiler techniques, to name just a few. However, energy conservation at the application software-level has not yet been explored. In this paper, we show the impact of various software implementation techniques on energy saving. Based on the observation that different instructions of a processor cost different amount of energy, we propose three energy saving strategies, namely (i) assigning live variables to registers, (ii) avoiding repetitive address computations, and (iii) minimizing memory accesses. We also study how a variety of algorithm design and implementation techniques affect energy consumption. In particular, we focus on the following aspects: (i) recursive vs. iterative (with stacks and without stacks), (ii) different representations of the same algorithm, (iii) different algorithms—with identical asymptotic complexity—for the same problem, and (iv) different input representations. We demonstrate the energy saving capabilities of these approaches by studying a variety of applications related to power-conscious systems, such as sorting, pattern matching, matrix operations, depth-first search, and dynamic programming. From our experimental results, we conclude that by suitably choosing an algorithm for a problem and applying the energy saving techniques, energy savings in excess of 60% can be achieved.

Key Words: Low power system, energy saving, software design and implementation

1 Introduction

Miniaturization of computing systems is finding applications in special areas such as hand-held computation, tiny robots, guidance systems in automated vehicles, to name just a few. Also, these systems or their users move from place to place. Because of their small size and the mobility requirement, they are powered by a few batteries of low rating. In order to avoid frequent recharging and/or replacement of the batteries, there is significant interest in low-power system design. Power consumption is increasingly becoming an area of growing concern in system design. It affects a variety of system concerns such as battery life, thermal limits, packaging constraints, and cooling options [5].

Let a program P run for T seconds to achieve its goal, V_{CC} be the supply voltage of the system, and I be the average current in ampere drawn from the power source for T seconds. We can rewrite T as $T = N \times \tau$, where N is the number of clock cycles and τ is the clock period. Then, the amount of energy consumed by P to achieve its goal is given by: $E = V_{CC} \times I \times N \times \tau$ *Joules*. Since for a given hardware, both V_{CC} and τ are fixed, $E \propto I \times N$. However, at the application level, it is more meaningful to talk about T than N, and therefore, we express energy as $E \propto I \times T$. Given the fact that power is the rate of energy consumption, in this paper, we refer to power and energy interchangeably.

Low power design is a complex endeavor requiring a broad range of strategies from floor planning on silicon substrate to design of application softwares. In Figure 1, we enumerate several strategies for achieving energy efficiency in a power-conscious system. In Section 2, we review some of these strategies. Though energy is actually consumed by the hardware, energy consumption can be reduced—apart from using low-power electronics—by suitably manipulating the software systems. This is because the hardware activities are controlled through the softwares. As the expression for energy suggests, the main idea in the design of energy-efficient softwares is to reduce both T and I.

From the running time (average case) of an algorithm we get a measure of T. However, to compute I, one must consider the current drawn during each clock cycle. Actual measurement of the current for different instructions shows that *different instructions lead to different amount of currents being drawn* [27], [28]. For example, in case of the Intel 486DX2 processor, moving data between two registers takes *one* clock cycle and draws 291.2 mA of current, whereas moving data from a register to a memory location can take up to *two* clock cycles drawing 451.7 mA during those cycles [28]. This exposes the potential for reducing the average current I by employing a number of implementation strategies which tend to use the instructions drawing less current and less number of instructions. This also leads to a reduced T.

In this paper, we study the impacts of the following three software implementation techniques on energy consumption:

EC1: coding a software by employing energy saving techniques,

EC2: choice of algorithms, and

EC3: general implementation strategies.

The first strategy **EC1** is based on the experimental result that different instructions draw different amount of current and take different number of clock cycles. We show a few programming techniques leading to a C compiler generating low-energy instructions and less number of total instructions for a program. The main ideas are to eliminate some redundant computations, reduce the number of memory accesses, assign temporary variables to registers, and avoid repetitive address computations.

Second, we study how a variety of algorithm design techniques affect energy consumption. Here, the main idea is to reduce the constant factor in the complexity of an algorithm. Finally, we study how different general implementation strategies affect the energy cost of a program. In particular, we study the following aspects of an implementation:

- recursive vs. iterative (with stacks and without stacks) techniques,
- different coding techniques, and
- different input data representations such as array, link-list, adjacency matrix, and adjacency list.

Remark 1 Energy conservations due to **EC1**, **EC2**, and **EC3** are orthogonal and can independently be applied to a software design.

To study the impacts of **EC1**, **EC2**, and **EC3** on energy conservation, we studied algorithms for a number of problems, such as sorting, depth-first search, dynamic programming, matrix multiplication, Gauss elimination, and pattern matching. Sorting is a very general problem encountered in almost all applications. Users of the laptop computers generally do a lot of text editing, where pattern matching is an important function. Also, in mobile robots equipped with computer vision, pattern matching is essential to identifying objects. Other problems such as depth-first search, dynamic programming, matrix multiplication, and Gauss elimination are central to controlling the motion of robots. Though we have not identified all kinds of softwares used in power-conscious systems, we believe that our selection represents a cross-section of the kinds of problems encountered in those systems.

We study these impacts by implementing the algorithms with the energy saving programming techniques and computing their energy costs. The energy cost is computed by considering the patterns of code generated by a compiler for various control structures of the C language and the current drawn by each instruction of the Intel 486DX2 processor [28]. Similar analysis can also be done for other compilers and processors. Our finding is that by suitably implementing a solution to a problem, one can save in excess of 60% of energy.

Remark 2 The growth in battery technology in terms of *energy density* and *power density*[1] [17]

[1]The energy content of a battery expressed in watt-hours per liter is referred to as energy density, and the power delivered by a battery expressed in watts per liter is referred to as power density.

is expected to increase only 20% over the next decade [20]. Thus, energy saving in access of even 20% achieved through careful design of high-level softwares is of immense practical importance.

The paper is organized as follows. In Section 2, we review a number of techniques—both hardware and software—used in the design of low-power systems. Section 3 contains our main ideas and the experimental results. First, we identify three energy saving techniques that can be applied to any program in general. Second, we make a link between the constructs of the C language and the instructions of a processor from the viewpoint of computing the energy cost of a program. Third, we apply the energy saving techniques to a number of algorithms. Fourth, we study the impact of algorithm design techniques on energy cost, and then the impacts of implementation strategies on energy costs are explored. Finally, we give some concluding remarks in Section 4.

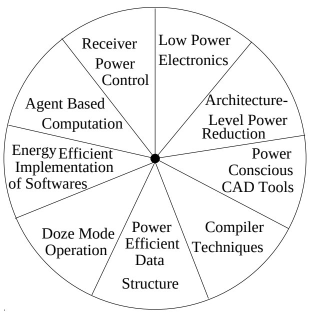

Receiver Power Control
Low Power Electronics
Agent Based Computation
Architecture-Level Power Reduction
Energy Efficient Implementation of Softwares
Power Conscious CAD Tools
Doze Mode Operation
Power Efficient Data Structure
Compiler Techniques

Figure 1: Various ways of reducing energy consumption in a system (not to scale).

2 Review of Energy Saving Strategies

We review a wide spectrum of strategies, shown in Fig. 1, ranging from the hardware fabrication process to receiver power control. Energy saving due to different approaches are, in the best case, multiplicative. For example, a 10% energy savings from low-power electronics together with a 15% savings from compiler techniques will yield a total energy saving of: (1 - (((1 - 0.1)* 0.85)) * 100 = 23.5%. However, generally the total energy savings is less, say in this example 20%, because the various energy saving strategies may adversely affect each other.

Low Power Electronics

The power equation in a CMOS digital circuit is expressed as [18]:

$$P = (C_L V_{DD}^2 f_p) + (I_{SC} V_{DD}) + (I_{leakage} V_{DD}), \tag{1}$$

where V_{DD} is the supply voltage, f_p is the output switching repetition frequency, C_L is the output capacitance load, I_{SC} is the short circuit current pulse generated when both n- and p-transistors are briefly turned on during output switching, and $I_{leakage}$ is the leakage current. The first term on the right-hand side of the power equation is the dominant factor. It is expected that power saving with two orders of magnitude can be achieved using low-power electronics. About half of the power reduction will come from architecture changes and management of switching activity. The other half of power reduction will come from using advanced materials technology to allow reduction of V_{DD} to 1 Volt or below from 5 or 3.5 Volts while also reducing C_L [12], [1].

Architecture-Level Power Reduction

The microprocessor can account for up to 33% of a notebook's power budget, which is around 15W. The processors designed for high-end desktops are not suitable for small mobile systems, because these processors dissipate up to 16W with a 5V power supply. Therefore, processor designers include a number of features to reduce power consumption. For the PowerPC 603^{TM} [18], some of the power reduction features are dynamic idle-time shutdown of separate execution units, low-power cache design, and power considerations for standard cells, data-path elements, and clocking. The processor also supports three static power management modes *doze, nap*, and *sleep*. These modes reduce power at a global level when the processor is idle for an extended period of time. Since CMOS circuits consume power during the charging and discharging of capacitances, reducing switching activity saves power. At the architecture-level, two strategies to reduce switching activities are Gray code addressing and cold scheduling of instructions [3]. Experimental results show that cold scheduling reduces switching by 20 to 30%. The Gray code's advantage over the binary code is that each memory access changes the address by only one bit. Thus, a significant number of bit switches can be eliminated using Gray code addressing. Also, by decomposing a finite-state machine into several submachines, it is possible to selectively turn off portions of a circuit, thereby reducing the switching activities [19].

Power-Conscious CAD Tools

The design of low-power systems cannot be achieved without good power-conscious CAD tools. CAD tools are used at all levels of hardware design: behavioral, architectural, logic and physical. For a detailed exposition of power-conscious CAD tools, the reader is referred to two tutorials by Singh, et. al [5] and Pedram [16].

Compiler Techniques

Compiler design techniques contribute to energy saving in several ways. Lee and Tiwari [11] address the problem of allocating memory to variables in embedded DSP (Digital Signal Processing) softwares. The goal is to maximize simultaneous data transfers from different memory banks to registers. In several DSP applications, two registers are loaded with the required data and an arithmetic operation is performed. Loading two registers with a single *double transfer* instruction draws a little more current than a move instruction. Both the instructions take one clock cycle each. However, energy is saved by using the double transfer, because the double transfer instruction loads the two registers in one clock cycle, whereas we need two clock cycles to sequentially load the registers. Experimental results for a few applications on a Fujitsu DSP processor show that up to 47% of energy can be saved by this approach.

Instructions with memory operands have much higher energy costs than instructions with register operands [27]. This suggests that energy can be saved by suitably assigning the live variables of a program to registers. But, a processor has only a small number of registers. When the number of simultaneously live variables is larger than the number of available registers, some of the variables must spill to memory. Register assignment for loop variables is important because loops are typically executed many times. Kolson, et. al. [4] present an algorithm for optimal register assignment to loop variables for embedded systems. This algorithm can be included in the code generation part of a compiler.

Power Efficient Data Structures

Wuytack, Francky and De Man [21] propose a method of implementing set data types with minimum power consumption. A set data type is an abstract data type widely used in communication systems [10] and database systems. In a programming language, one can implement the set data type using a variety of concrete data structures such as linked list (LL), binary tree (BT), array (AR), and pointer array (PA). Thus, to implement the set operations, such as locate, insert, and remove a record from a set, one has to manipulate the memory elements in a concrete data structure. It is the memory accesses in the process of set operations that actually consume power. Thus, the power consumption in set operations is a function of the number of memory elements used in implementing a set data type, the number of read and write operations done in the implementation, and some logic details such as capacitance of memory elements, voltage level, and frequency of operation. The concrete data structures are compared on the basis of a *filling factor*, which is the fraction of the locations that would be filled if implementation is in arrays. It has been shown that for different levels of filling factor, different concrete data structures lead to low values of the power cost function. For example, for filling factor greater than 60%, arrays are better than the LL, BT, and PA structures in implementing power efficient set data type.

Doze Mode of Operation

The *doze* mode is an innovative approach to conserving energy [18]. It is very attractive in a communication environment where a mobile system may occasionally send or receive messages. In the doze mode, the clock speed is reduced and no user process is executed. Rather, a mobile host simply waits for any incoming message. Upon receiving a message, the host resumes its normal mode of operation. The energy saving due to this mode depends on the local computations on a mobile and the pattern of communication between a mobile and a support station. Simulation studies show that energy saving due to this mode spreads over a wide range of 2–98% [15].

Agent Based Computation for Energy Saving

Agent based computation is a relatively new idea in distributed computing [9], [14], [22], [25], [26]. General agent-based distributed computing systems have been designed using the concept of Linda's *tuple space* [7], [8], [24]. Sato et. al. [22] have built a distributed autonomous system called

Noah (Network oriented application harmony) in their Mitsubishi laboratory. Though the purpose of *Noah* is not to save energy, it demonstrates how agent based systems can be built using a tuple space as the medium for process communication. Naik and Wei [15] discuss how energy-efficient distributed algorithms in a mobile computing environment can be designed using a tuple space managed on the fixed network of a mobile system.

Receiver Power Control

The receiver subsystem of a mobile station need not be active all the time. Most digital cellular and cordless systems provide power cycling at the mobile units. Mobile stations can periodically relax (power cycle) their receivers as a means of conserving energy. Since the receiver of a mobile unit is not continuously ready to receive messages from the local support station (base station), some kind of coordination between a base station and a mobile unit is necessary. Salkintzis, et. al. [2] propose a *page-and-answer* protocol. Intuitively, the protocol works as follows. When a base station has a message for a mobile unit, the base station sends a small paging packet to the mobile unit. If the mobile unit receives the paging packet, that is if the mobile's receiver is up, the mobile sends an answer packet to the base station. Obviously, if the paging message is sent at a time when the receiver is powered off, no answer packet is generated by the mobile and the base station will once again page the mobile after some time. Upon receiving an answer packet, the base station sends the desired message to the mobile unit.

3 Energy Saving Potential of Implementation Techniques

To study the energy saving potential of algorithm design strategies and implementation techniques, we consider the C language. Though energy consumption of a software can precisely be computed by summing up the energy requirements of all the machine instructions executed, we are interested in computing the energy requirements of a software described in a high-level language. We selected C because we can easily make a correspondence between a statement in C and a block of instructions by studying some example assembly codes produced by a C compiler. The energy consumption in such a block of instruction, in turn, is used in computing the energy consumption in a C program.

3.1 Energy Efficient Implementation Techniques

Measurement of energy costs of the instruction set of a few processors [27], [13] reveal that different instructions in the *same class* of instructions incur different energy costs. In Table 1, we show the energy cost of just a few instructions. By a class of instructions we mean a group of instructions with similar functionalities. For example, a conditional jump and an unconditional jump fall in the same class, whereas a jump and an add fall in different classes. Thus, we need to investigate the possibility of generating less number of instructions and low-cost instructions from a high level software. Toward this goal, we propose the following three strategies to reduce the energy cost of a program.

P1: Assign live variables to registers.

P2: Avoid repetitive computation of addresses.

P3: Minimize memory accesses.

To observe the impacts of the above strategies on energy saving, we apply those to the Bubblesort algorithm. We present the straightforward implementation of the algorithm in Table 2(a), and we apply the energy saving strategies to the algorithm in Table 2(b). The input to both the implementations is represented by an array called `list`. In the straightforward version, values of the live variables `n, i, j` and `temp` are stored in the memory, whereas in the second version they are stored in registers. Also, in the second version, we define a few extra variables to save some values to avoid repetitive computations.

Table 1: Energy cost of a subset of instructions of the Intel 486DX2 processor.

Instruction	**Current**(mA)	**Cycles**	**Instruction**	**Current**(mA)	**Cycles**
`MOV reg,imm`	299.2	1	`JCC imm - taken`	372.2	3
`MOV reg,reg`	291.2	1	`JCC imm - not taken`	356.8	1
`MOV reg,disp[base]`	434.7	1	`JMP imm`	370.1	3
`MOV reg,[base][index]`	409.0	2	`NOP`	275.7	1
`MOV disp[base],reg`	560.1	1	`ADD reg,imm`	315.6	1
`MOV disp[base],imm`	404.8	2	`ADD reg,reg`	309.0	1
`SAL reg,CL`	302.7	3	`ADD reg,dis[base]`	400.2	2
`CMP reg,imm`	296.0	1	`ADD disp[base],imm`	382.4	4
`CMP reg,reg`	288.0	1	`IMUL reg`	287.7	13
			`IMUL [base]`	305.0	13
			`IDIV [base]`	278.9	20
			`IDIV [base][index]`	281.8	21

In step 4 of Table 2(a), the `if` statement checks if the two array entries `list[j-1]` and `list[j]` array are to be swapped, and in steps 5 through 7, the swap is actually carried out. It may be noted that in case of a swap, the two array elements are read *twice*.

In Table 2(b), we implement the condition checking and swapping steps in an efficient manner. In steps 4 and 5, we first save the addresses of the two array elements in two register variables `p1` and `p2`, so that these addresses need not be recomputed while swapping the two array elements. However, it may be the case that no swapping is done in a certain iteration, in which case the two assignments become unnecessary. In the following section, experimental results show that, on the average, the two assignments contribute to energy saving. In step 6, we move the two array elements `list[j-1]` and `list[j]` into two registers denoted by `k1` and `k2`, respectively, and perform the condition checking. If the swap condition is satisfied, we carry out the swap by simply moving the contents of `k1` and `k2` into the appropriate locations, `p2` and `p1`, respectively, without reading `list[j-1]` and `list[j]` once again.

In Fig. 2, we illustrate the difference between the two versions of swap implemented in Table 2. A processor cannot directly move data from one memory location to another—the data must be read into a register and then written to the destination memory location. An assignment statement involving two memory locations needs a `load` and `store` instruction. Thus, in the straightforward implementation of the swap operation, shown in Fig. 2(a), *three* `load` and *three* `store` instructions are used. In the energy-efficient implementation of swap, illustrated in Fig. 2(b), we explicitly load the contents of the desired memory locations into two registers, and swap them through the registers needing only *two* `load` and *two* `store` instructions. This is verified by comparing their assembly codes given in Table 3.

Due to the high-level programming techniques **P1**, **P2**, and **P3**, less executable code is generated by a compiler from the source code. As an example, we apply these techniques to the bubblesort algorithm. The straightforward C representation of the bubblesort algorithm and the revised algorithm with the three techniques incorporated are shown in Table 2(a) and Table 2(b), respectively. For these two implementations of the bubblesort algorithm, the executable codes generated by a compiler are shown in Table 3. The left column of Table 3 corresponds to the straightforward implementation of Table 2(a), and the right column corresponds to Table 2(b). Comparing the two columns of code, it is clear that the high-level programming techniques **P1**, **P2**, and **P3** lead to less executable code.

Remark 3 Though in general less code does not mean less energy cost, experimental results in Section 3.3 show that less code obtained using the strategies discussed in this section leads to much less energy cost.

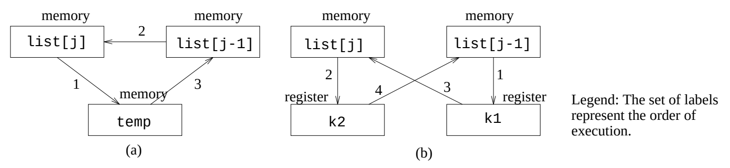


Figure 2: Swap Operation: (a) Straightforward, (b) Energy Efficient.

```
0. int list[10];                | 0.  int  list[10];
1. int i, j, n, temp;           | 1.  register int i, j, n, k1, k2, *p1, *p2;
2. for (i = n-1; i > 0; i--)    | 2.  for (i = n-1; i > 0; i--)
3.   for (j = 1; j <= i; j++){  | 3.    for (j = 1; j <= i; j++){
4.     if (list[j-1] > list[j]){ | 4.      p1 = list+j-1;
5.       temp = list[j];        | 5.      p2 = list+j;
6.       list[j] = list[j-1];   | 6.      if ((k1 = *p1) > (k2 = *p2)){
7.       list[j-1] = temp;      | 7.         *p2 = k1; *p1 = k2;
       }                        |         }
     }                          |       }
               (a)                                 (b)
```

Table 2: Bubble Sort: (a) Straightforward implementation, (b) Application of P1, P2, and P3.

```
----- Straightforward Implementation -----
.LL12:
        ld [%fp-60],%o0
        add %o0,-1,%o1
        st %o1,[%fp-424]
.LL20:
        ld [%fp-424],%o0
        cmp %o0,0
        bg .LL23
        nop
        b .LL21
        nop
.LL23:
        mov 1,%o0
        st %o0,[%fp-428]
.LL24:
        ld [%fp-428],%o0
        ld [%fp-424],%o1
        cmp %o0,%o1
        ble .LL27
        nop
        b .LL25
        nop
.LL27:
        ld [%fp-428],%o0
        mov %o0,%o1
        sll %o1,2,%o0
        add %fp,-16,%o1
        add %o1,%o0,%o0
        ld [%fp-428],%o1
        mov %o1,%o2
        sll %o2,2,%o1
        add %fp,-16,%o2
        add %o1,%o2,%o1
        ld [%o0-404],%o0
        ld [%o1-400],%o1
        cmp %o0,%o1
        ble .LL28
        nop
        ld [%fp-428],%o0
        mov %o0,%o1
        sll %o1,2,%o0
        add %fp,-16,%o1
        add %o0,%o1,%o0
        ld [%o0-400],%o1
        st %o1,[%fp-432]
        ld [%fp-428],%o0
        mov %o0,%o1
        sll %o1,2,%o0
        add %fp,-16,%o1
        add %o0,%o1,%o0
        ld [%fp-428],%o1
        mov %o1,%o2
        sll %o2,2,%o1
        add %fp,-16,%o2
        add %o2,%o1,%o1
        ld [%o1-404],%o2
        st %o2,[%o0-400]
        ld [%fp-428],%o0
        mov %o0,%o1
        sll %o1,2,%o0
        add %fp,-16,%o1
        add %o1,%o0,%o0
        ld [%fp-432],%o1
        st %o1,[%o0-404]
.LL28:
.LL26:
        ld [%fp-428],%o1
        add %o1,1,%o0
        mov %o0,%o1
        st %o1,[%fp-428]
        b .LL24
        nop
.LL25:
.LL22:
        ld [%fp-424],%o1
        add %o1,-1,%o0
        mov %o0,%o1
        st %o1,[%fp-424]
        b .LL20
.L21:   :

-----Energy Efficient Implementation-----
        add %l2,-1,%l0
.LL2:
        cmp %l0,0
        bg .LL5
        nop
        b .LL3
        nop
.LL5:
        mov 1,%l1
.LL6
        cmp %l1,%l0
        ble .LL9
        nop
        b .LL7
        nop
.LL9:
        add %fp,-416,%l7
        mov %l1,%o0
        sll %o0,2,%o1
        add %l7,%o1,%l7
        add %l7,-4,%l7
        add %fp,-416,%i0
        mov %l1,%o0
        sll %o0,2,%o1
        add %i0,%o1,%i0
        ld [%l7],%l4
        mov %l4,%o0
        ld [%i0],%l5
        mov %l5,%o1
        cmp %o0,%o1
        ble .LL10
        nop
        st %l4,[%i0]
        st %l5,[%l7]
.LL10:
.LL8:
        add %l1,1,%l1
        b .LL6
        nop
.LL7:
.LL4:
        add %l0,-1,%l0
        b .LL2
.LL3:    :
```

Table 3: Assembly codes from the two versions of Bubble sort.

3.2 Computation of Energy Cost

We explain how to compute the energy cost of a program in C. The basic idea is to make a link between a sequential block of C statements and the corresponding block of instructions produced by a C compiler. Intuitively, by summing up the energy costs of the instructions in a sequential block, we obtain the energy cost of the sequential block of C statements. The instructions generated from the bubblesort algorithm of Table 2(b) are presented in a block format in Table 4(a). There are nine blocks of instructions `BLOCK0` through `BLOCK8`. Precise computation of energy costs becomes more involved for a control statement, because the instructions corresponding to such a statement are not found as a contiguous block. For example, corresponding to the `for` statement in line 2 of Table 2(b), there are three blocks of instructions `BLOCK0`, `BLOCK1`, and `BLOCK7`. Also, these three blocks of instructions do not execute for an equal number of times—`BLOCK0` executes only once, whereas `BLOCK1` executes for $n-1$ times.

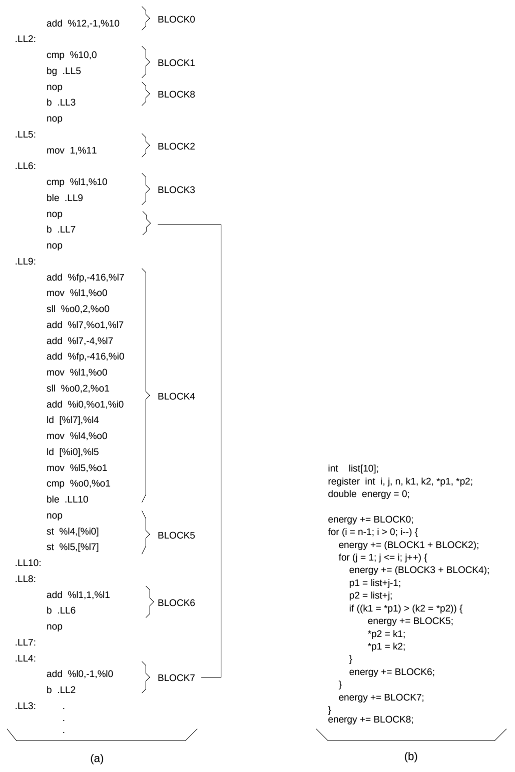
From the instructions generated by a C compiler, we can identify a block of instructions corresponding to a sequence of C statements, and several blocks for one C statement. For example, `BLOCK4` corresponds to the C statements in lines 4, 5, and 6 of Table 2(b). Generation of multiple blocks of instructions for one C statement has already been explained in the context of the `for` statement above. From the knowledge of the energy cost of an instruction, we can compute the energy cost of a block of instructions. Thus, the total energy cost of an algorithm can be computed by suitably placing counting statements using the energy costs of blocks of instructions in the algorithm. We compute the energy cost of the bubblesort algorithm of Table 2(b) in Table 4(b). In Table 4(b), the blocks `BLOCK0` through `BLOCK8` represent the energy costs of the corresponding blocks of instructions in Table 4(a).

Thus, we compute the average energy cost of an algorithm in three steps as outlined below.

- Given an algorithm in C, identify a correspondence between the C statements and the blocks of instructions produced by a compiler.
- Using the block structures identified above, insert energy counting statements into the algorithm.
- Run the algorithm in a loop for many times using a variety of inputs, and compute the average energy cost.

In this approach, identification of instruction blocks and insertion of statements to accumulate the energy cost of an algorithm is done by hand. This approach is useful to study the energy costs of small programs. However, in our opinion, this needs to be done by an energy cost-conscious compiler which is outside the scope of this paper.

In the computation of energy cost of a program, the basic cost of an instruction is represented by the product of the *number of clock cycles* taken by the instruction and the *average current* drawn during that period. Thus, the energy cost of an instruction is represented in *ampere-cycles* unit. By dividing the *ampere-cycle* cost of a program for a given input by the clock frequency in *cycles/second* of the processor, we obtain the energy cost of the program in *ampere-seconds.*



```
        add  %12,-1,%10
.LL2:
        cmp  %10,0
        bg  .LL5
        nop
        b  .LL3
        nop
.LL5:
        mov  1,%11
.LL6:
        cmp  %l1,%10
        ble  .LL9
        nop
        b  .LL7
        nop
.LL9:
        add  %fp,-416,%l7
        mov  %l1,%o0
        sll  %o0,2,%o0
        add  %l7,%o1,%l7
        add  %l7,-4,%l7
        add  %fp,-416,%i0
        mov  %l1,%o0
        sll  %o0,2,%o1
        add  %i0,%o1,%i0
        ld  [%l7],%l4
        mov  %l4,%o0
        ld  [%i0],%l5
        mov  %l5,%o1
        cmp  %o0,%o1
        ble  .LL10
        nop
        st  %l4,[%i0]
        st  %l5,[%l7]
.LL10:
.LL8:
        add  %l1,1,%l1
        b  .LL6
        nop
.LL7:
.LL4:
        add  %l0,-1,%l0
        b  .LL2
.LL3:       .
            .
            .
```

```
int    list[10];
register  int  i, j, n, k1, k2, *p1, *p2;
double  energy = 0;

energy += BLOCK0;
for (i = n-1; i > 0; i--) {
    energy += (BLOCK1 + BLOCK2);
    for (j = 1; j <= i; j++) {
        energy += (BLOCK3 + BLOCK4);
        p1 = list+j-1;
        p2 = list+j;
        if ((k1 = *p1) > (k2 = *p2)) {
               energy += BLOCK5;
               *p2 = k1;
               *p1 = k2;
        }
        energy += BLOCK6;
    }
    energy += BLOCK7;
}
energy += BLOCK8;
```

Table 4: (a) Blocks of Instructions generated from Bubblesort, (b) Computing Energy Cost.

3.3 Impact of Energy Saving Techniques

In this section, we apply the energy saving techniques **P1, P2**, and **P3** to a number of algorithms and study their impacts on energy saving. As examples, we apply these techniques to the bubblesort algorithm of Table 2(a), Table 2(b), Table 5(a), and Table 5(b). A program's energy cost is computed using the strategy explained in Section 3.2. We experiment algorithms for three different classes of problems, namely sorting, dynamic programming, and matrix operations. In particular, we choose bubblesort, matrix chain product, matrix multiplication, and Gauss elimination algorithms.

```
int list[10];                                |   int list[10];
register int i, j, n, k1, k2;                |   register int i, j, n, temp;
                                             |
for (i = n-1; i > 0; i--){                   | for (i = n-1; i > 0; i--){
  for (j = 1; j <= i; j++){                  |   for (j = 1; j <= i; j++){
    if ((k1 = list[j-1]) > (k2 = list[j])){  |     if (list[j-1] > list[j]){
      list[j] = k1;                          |        temp = list[j];
      list[j-1] = k2;                        |        list[j] = list[j-1]; list[j-1] = temp;
    }                                        |     }
  }                                          |   }
}                                            | }
```

Table 5: Bubble Sort: (a) Using techs. P1 and P3, and (b) Using P1.

For the bubblesort algorithm, we show its energy cost in Fig. 3. Fig. 3 shows that technique **P1** alone saves 46% of energy. Both **P1** and **P3** together save 54% of energy, and all three techniques applied together save up to 59% of energy. In case of the matrix chain product, matrix multiplication, and Gauss elimination algorithms, Figures 4, 5, and 6 show energy savings up to 27%, 46%, and 48%, respectively. The suitability of a particular technique depends on the algorithms. For example, we cannot apply **P2** to the matrix multiplication algorithm.

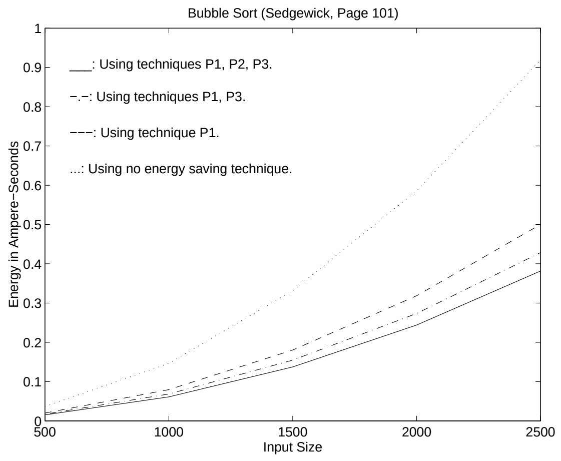


Figure 3: Impact of energy saving techniques demonstrated using Bubblesort.

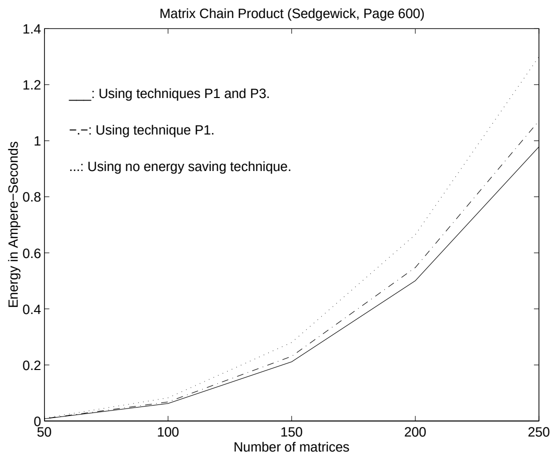


Figure 4: Impact of energy saving techniques demonstrated using Matrix Chain Product.

3.4 Impact of Algorithm Design Techniques on Energy Cost

In general, any strategy that can reduce the running time of an algorithm is also useful in saving energy. Algorithms, for the same problem, with identical asymptotic complexity differ in their

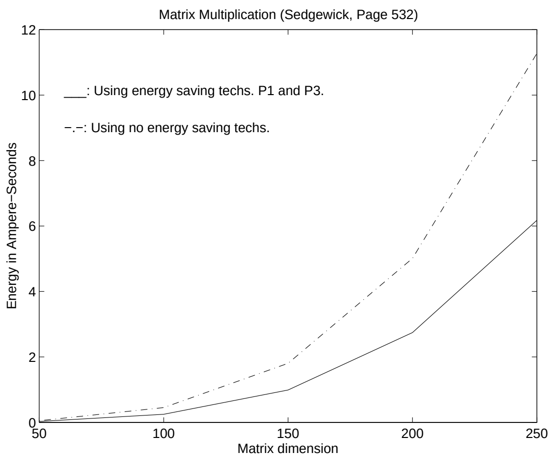


Figure 5: Impact of energy saving techniques demonstrated using Matrix Multiplication.

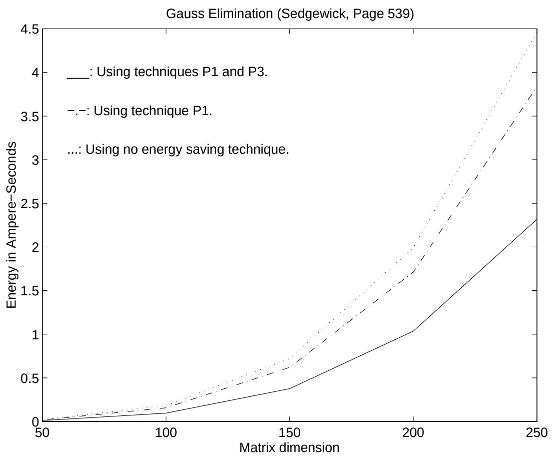


Figure 6: Impact of energy saving techniques demonstrated using Gauss Elimination.

energy costs depending on the constant factor behind the big-O. Thus, to design an energy efficient algorithm, we need to reduce not only its asymptotic complexity, but also the constant factor. So far the idea of reducing the asymptotic complexity has received a great amount of attention and this area is quite mature. However, reducing the constant factor side has been largely ignored, which is vital to saving energy. Reducing the constant factor is important in the sense that bringing a constant factor from say 4 down to 2 leads to a 50% energy saving.

Intuitively, loop control contributes to the asymptotic complexity, whereas the number of statements in a loop affects the constant factor. Therefore, reducing the number of statements in each loop is likely to reduce the constant factor. Let us consider the sorting problem. Assuming that the input is an array, reducing the number of array accesses, that is the read and write operations, can potentially reduce the constant factor. In Fig. 7, we show the energy costs of three sorting algorithms, namely quicksort[2], mergesort, and heapsort, with identical asymptotic complexity, say $O(n \lg n)$. It is apparent that the constant factor of quicksort is the lowest among the three. Precisely, it is about half of that of the heapsort. Also, the constant factor of the mergesort is about 0.85 times that of the heapsort. According to our observation, this is due to quicksort's relatively less access, that is read and write operations, of the input array. To confirm our observation, we count the number of array accesses in the three algorithms and show the result in Fig. 8. This follows from the fact that the merge and heapsort algorithms have more read/write operations per iteration than those in the quicksort.

As another example, we study the impact of algorithm design techniques on energy saving using the pattern matching problem. We consider four algorithms, namely the brute-force approach, Rabin Karp (RK) algorithm, Boyre Moore (BM) algorithm, and the Knuth, Morris and Pratt (KMP) algorithm, whose energy costs are shown in Fig. 9. It is clear that the RK algorithm is the most expensive one and the BM algorithm is least expensive. This is because of two reasons: (i) the RK algorithm uses the modulo operation, which costs more than 20 times than a simple operation such as an addition, in the loop; and (ii) the BM algorithm uses an elegant way of skipping while comparing the pattern against the input string. Due to the efficient skipping, the algorithm accesses the input fewer number of times.

We also show the impact of the energy saving techniques on the pattern matching algorithms in Fig. 10. Since the RK algorithm is anyway a lot more expensive than the others, we do not show it in Fig. 10. The figure shows that energy saving in the range of $18 - 29\%$ can be achieved. In case of the BM algorithm, the energy saving techniques do not save significant amount of energy. In spite of this, the BM algorithm is the best choice because overall it is the least expensive one.

3.5 Impact of Implementation Strategies on Energy Cost

Usually, after an algorithm is designed, not much attention is paid to the style of implementations. In terms of energy saving, the constant factor behind the big-O of an algorithm is important, and different implementation techniques may lead to different constant factors. We investigate several

[2]In our experiment, we use randomly generated input data leading to an average behavior of the algorithm.

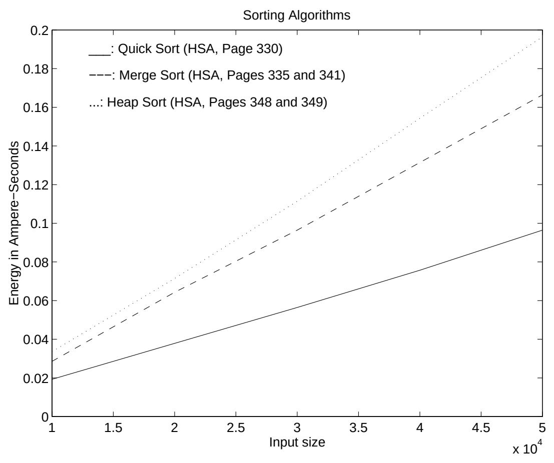


Figure 7: Energy cost of different algorithms for the same problem (sorting).

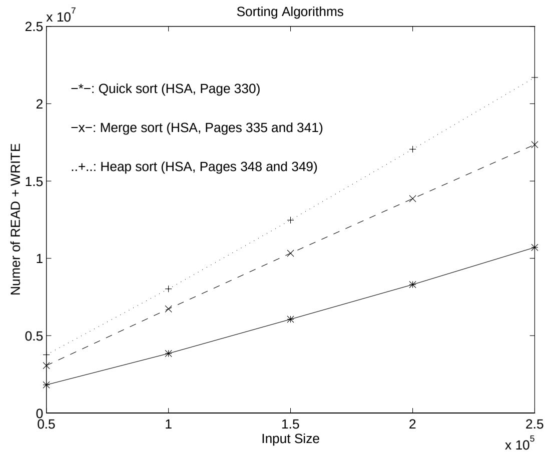


Figure 8: Read and Write Cost of a few sorting algorithms.

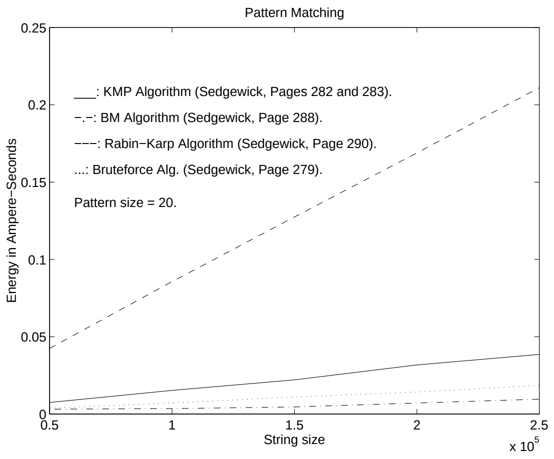


Figure 9: Energy cost of different algorithms for the same problem (pattern matching).

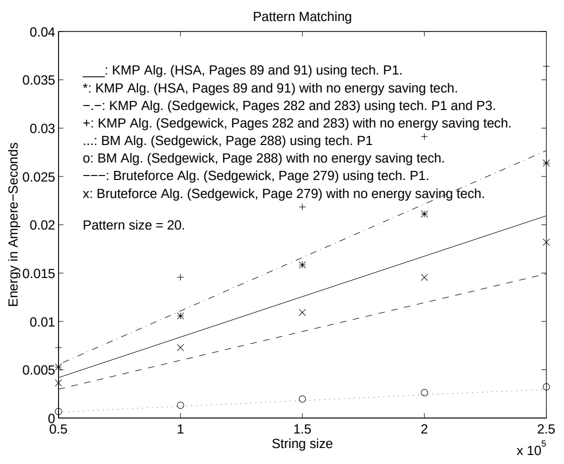


Figure 10: Energy cost of different algorithms for the same problem (pattern matching).

factors influencing the constant factors in what follows.

3.5.1 Recursion vs. Iteration

For some problems, algorithms are designed using both recursive and iterative styles. An advantage of recursive representation is that it is compact. However, due to the necessity of a stack managed by the system, it is often the case that a recursive algorithm takes longer time than its iterative counterpart. In the following, we compare the energy costs of recursive algorithms with those of iterative algorithms—with and without stacks.

The energy costs of two versions of quicksort—a recursive one and an iterative one using a stack at the user level—are compared in Fig. 11. The iterative one is beaten by the recursive one because of using a stack at the user level. One the one hand, managing a stack at the user level requires implementing the *push* and *pop* functions. On the other hand, the idea of stack involved in a recursive algorithm is efficiently managed by the compiler using static information about a procedure, such that the stack is basically implemented as a simple procedure call at the instruction level.

The energy costs of two versions of mergesort—a recursive one and an iterative one *without* using a stack—are compared in Fig. 12. Because the iterative one does not need a stack, it is less expensive.

This suggests that an iterative algorithm without using a stack tends to be better than its recursive counterpart. Comparing Figures 11 and 12, we see that the recursive version of the quicksort costs less energy than the iterative version of the mergesort. Thus, one might ask if one can design an iterative version of the quicksort without using a stack. The answer is that one cannot avoid using a stack, because the loop control parameters (the boundaries of each sub-list) must be remembered, rather than be computed as in the case of the mergesort.

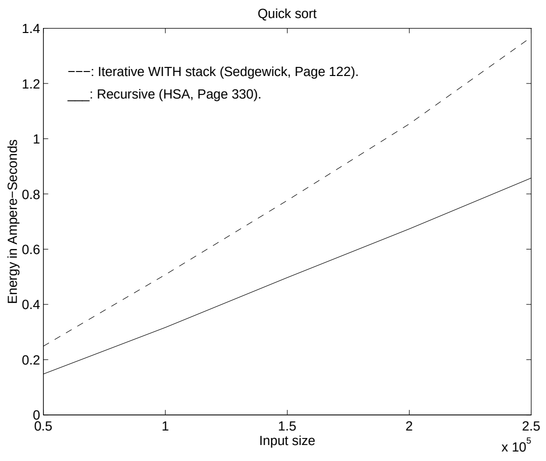


Figure 11: Energy cost of different implementations of the same algorithm (quicksort).

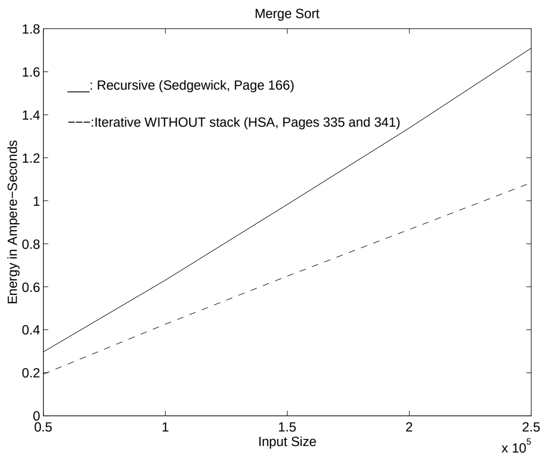


Figure 12: Energy cost of different implementations of the same algorithm (mergesort).

3.5.2 Various Coding Techniques

We show how coding the same algorithm even in slightly different ways can lead to widely varying energy costs. Those slightly varying codes can easily be overlooked, but their energy costs may significantly be different. We use the KMP pattern matching and quicksort algorithms to explore this aspect. In Table 6, we show two different codings of the KMP algorithm whose energy costs are shown in Fig. 13. It is apparent that the code in Horowitz, Sahni and Anderson (HSA) [6] consumes 31% less energy than the code in Sedgewick [23]. The reason behind this is that in the former case each iteration executes about one or two statements less than the latter case out of about five or six statements depending on the loop condition.

```
int pmatch(char *string, char *pat) {   | int kmpsearch (char *p, char *a){
/* KMP algorithm           */            |   int i, j;
  int i = 0, j = 0;                      |   int M = strlen(p), N = strlen(a);
  int lens = strlen(string);             |   initnext(p);
  int lenp = strlen(pat);                |   for (i = 0, j = 0; j < M && i < N; i++, j++)
  while (i < lens && j < lenp) {         |     while ((j >=0) && (a[i] != p[j])) j = next[j];
    if (string[i] == pat[j]) {           |     if (j == M) return i-M;else return i;
          i++; j++;                      | }
    }                                    |
    else if (j == 0) i++;                |
        else j = failure[j-1]+1;         |
  }                                      |
  return ((j == lenp) ? (i-lenp) : -1);  |
}                                        |
                (a)                      |                  (b)
```

Table 6: KMP Algorithm: (a) Horowitz, Sahni and Anderson, P. 89, (b) Sedgewick, P. 282.

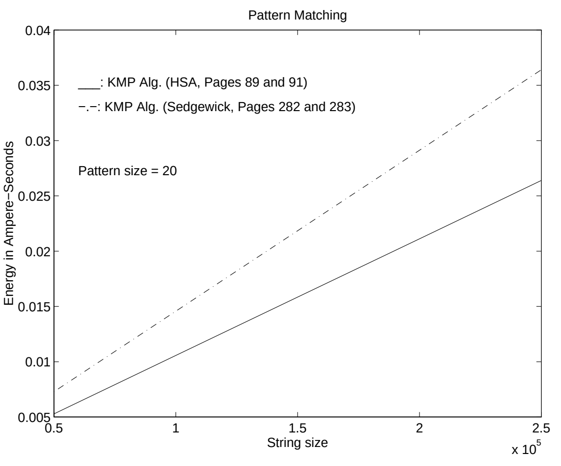


Figure 13: Energy cost of different coding techniques applied to the KMP algorithm.

Fig. 14 shows the energy costs of two different codings of quicksort. The code in Sedgewick [23] consumes about 14% less energy than that in HSA [6]. This is because of two reasons. First, the `for` loop in the Sedgewick code allows one less conditional check to be done than the HSA code using the `while` construct. Second, the `while` statements in the Sedgewick code generate a segment of more efficient instructions than the HSA's `do-while` statements. More precisely, the former generates one less `load` instruction. It is our belief that by carefully designing the control flow in a program, less number of statements are executed.

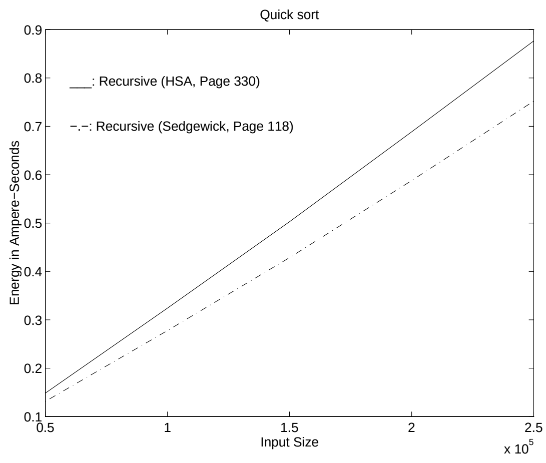


Figure 14: Energy cost of different coding techniques applied to the same algorithm (quicksort).

3.5.3 Different Input Representations

In this section, we show that even input formats can also significantly affect the energy cost of a program. We choose the depth-first search and mergesort algorithms for this study. In case of the depth-first search problem, an input graph can be represented by its adjacency matrix or adjacency list. Because of the different input formats, the algorithms have to be designed in different ways, and it turns out that the one with adjacency list input performs better in cases of sparse graphs. Figure 15 demonstrates this aspect in terms of energy cost. The suitability of an algorithm depends on the *sparsity factor*, where sparsity factor is intuitively defined as the ratio of the number of edges in an N-node graph to $N(N-1)/2$.

It is apparent from Fig. 16 that in the mergesort, algorithms with array inputs outperform those with linked list input. In our experiment we used only integer data. However, for large record data type, we believe that the algorithms with linked list input will perform better. This is because swapping the records, which is a lot more expensive than manipulating the pointers of the list data, will be avoided.

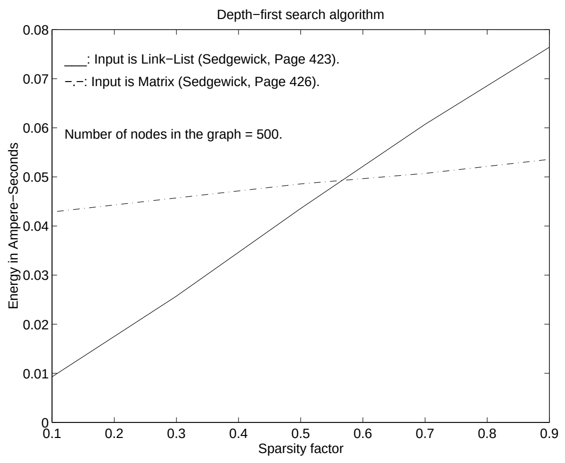


Figure 15: Energy cost of algorithms with different input formats.

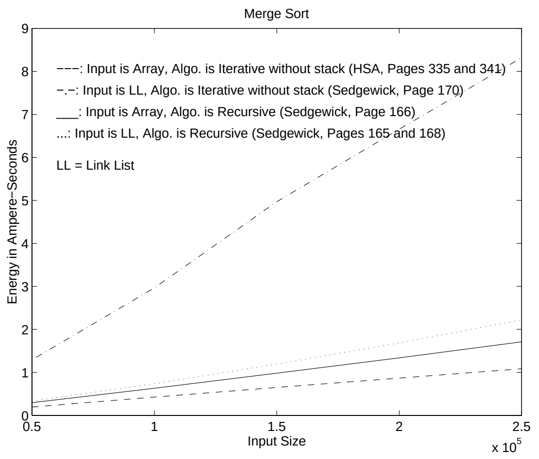


Figure 16: Energy cost of algorithms with different input formats.

4 Concluding Remarks

In this paper we addressed how by employing various techniques at the software implementation level, we can save energy in power-conscious systems. In particular, we presented three energy saving techniques, namely assigning live variables to registers, avoiding repetitive address computations, and minimizing memory accesses. Using the energy cost of individual instructions, we proposed a way to compute the energy cost of a high level program in C. We applied these techniques to algorithms for sorting, matrix chain product, matrix multiplication, Gauss elimination, and pattern matching. These techniques lead to energy savings from 18% to 60%.

We also studied the impact of algorithm design and implementation techniques with various formats of inputs on energy saving. From the viewpoint of energy saving, we need to reduce the constant factor in the complexity of an algorithm. Using a few sorting and pattern matching algorithms with identical asymptotic behavior, we showed how different constant factors lead to different energy costs. In case of sorting, one can save up to 50% of energy by choosing the quicksort over the heapsort, and in case of pattern matching, one can save up to 95% of energy by using the Boyre-Moore algorithm instead of the Rabin-Karp algorithm. Among the many possible ways of implementing an algorithm, we selected three general aspects, namely recursion versus iteration, different ways of coding an algorithm, and input formats. Experimental results show that an iterative implementation without using a stack performs better than a recursive one which in turn performs better than an iterative implementation using a stack. Using this knowledge, one can save energy in excess of 35%. By carefully designing the control flow in a program, it is possible to save energy. In the two versions of the quicksort and pattern matching algorithms, energy savings up to 14% and 31%, respectively, can be achieved. We also showed how different input representations, such as array, link-list, adjacency matrix, and adjacency list, affect the energy costs.

Minimizing the energy cost due to a high-level program is a new research idea. In this paper, we studied how different approaches to software design can be exploited in reducing the energy costs.

References

[1] Special issue on low power electronics. *Proceedings of the IEEE, Vol. 83, No. 4*, April, 1995.

[2] C. Chamzas A. K. Salkintzis and C. Koukourlis. An energy saving protocol for mobile data networks. *International Conference on Advances in Communications and Control (COMCON 5)*, pages 26–30, June, 1995.

[3] C-Y. Tsui C-L. Su and A. M. Despain. Saving power in the control path of embedded processors. *IEEE Design and Test of Computers*, pages 24–30, Winter, 1994.

[4] A. Nicolau D. J. Kolson and N. Dutt. Optimal register assignment to loops for embedded code generation. *ACM Trans. on Design Automation of Electronic Systems, Vol. 1, No. 2*, pages 251–279, April, 1996.

[5] et. al. D. Singh. Power conscious cad tools and methodologies: A perspective. *Proc. of the IEEE, Vol. 83, No. 4*, pages 570–594, April, 1995.

[6] S. Sahni E. Horowitz and S. Anderson-Freed. *Fundamentals of Data Structures in C*. Computer Science Press, 1993.

[7] D. Gelernter. Generative communication in linda. *ACM TOPLAS, Vol. 7, No. 1*, pages 80–112, Jan. 1985.

[8] J. G. Steiner H. E. Bal and A. S. Tanenbaum. Programming languages for distributed computing. *ACM Computing Surveys, Vol. 21, No. 3*, pages 261–322, Sept. 1989.

[9] H. Ihara and K. Mori. Autonomous decentralized computer control systems. *Computer*, pages 57–64, August, 1984.

[10] J.-Y. LeBoudec. The asynchronous transfer mode: A tutorial. *Computer Networks and ISDN Systems, Vol. 24*, pages 279–309, 1992.

[11] M. T. Lee and V. Tiwari. A memory allocation technique for low-energy embedded dsp software. In *Symposium on Low Power Electronics*, pages 24–25, 1995.

[12] Z. J. Lemnios and K. J. Gabriel. Low-power electronics. *IEEE Design and Test of Computers*, pages 8–13, Winter, 1994.

[13] S. Malik M. T. Lee, V. Tiwari and M. Fujita. Power analysis and minimization techniques for embedded dsp software. *IEEE Trans. on VLSI Systems, Vol. 5, No. 1*, pages 123–135, March, 1997.

[14] K. Mori. Autonomous decentralized systems: Concept, data field architecture and future trends. In *1st Intl. Symposium on Autonomous Decentralized Systems*, pages 28–34, 1993.

[15] K. Naik and D. S. L. Wei. Energy-conserving software design for mobile computers. In *DIMAC Workshop on Mobile Networks and Computing*, March 25-27, 1999.

[16] M. Pedram. Power minimization in ic design: Principles and applications. *ACM Trans. on Design Automation of Electronic Systems, Vol. 1, No. 1*, pages 3–56, Jan. 1996.

[17] R. A. Powers. Batteries for low power electronics. *Proc. of the IEEE, Vol. 83, No. 4*, pages 687–693, April, 1995.

[18] et. al. S. Gary. Powerpc 603, a microprocessor for portable computers. *IEEE Design and Test of Computers*, pages 14–23, Winter, 1994.

[19] Y-C. Ho S-H. Chow and T. Hwang. Low power realization of finite state machines–a decompostion approach. *ACM Trans. on Design Automation of Electronic Systems, Vol. 1, No. 3*, pages 315–340, July, 1996.

[20] A. Chandrasekaran S. Sheng and R. B. Broderson. A portable multimedia terminal for personal communications. *IEEE Comm. Magazine*, pages 64–75, Dec., 1992.

[21] F. V. M. Catthoor S. Wuytack and H. J. De Man. Transforming set data types to power optimal data structures. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 6*, pages 619–629, June, 1996.

[22] F. Sato, H. Kozuka, K. Miyazaki, and H. Fukuoka. Noah: An environment for distributed autonomous systems. In *2nd Intl. Symposium on Autonomous Decentralized Systems*, pages 412–418, 1995.

[23] R. Sedgewick. *Algorithms in C.* Addison-Wesley Publishing Co., 1990.

[24] A. H. Sherman. C-linda user's guide and reference manual. *Scientific Computing Associates Inc., New Haven, USA*, 1991.

[25] K. Sugawara. Agent-oriented architecture for flexible networks. In *2nd Intl. Symposium on Autonomous Decentralized Systems*, pages 135–141, 1995.

[26] M. Inoue T. Kondo and K. Nakai. Application of autonomous decentralized system to the steel production computer control. In *3rd International Workshop on Future Trends of Distributed Computing Systems*, pages 419–423, 1992.

[27] S. Malik V. Tiwari and A. Wolfe. Power analysis of embedded software: A first step towards software power estimation. *IEEE Trans. on VLSI Systems, Vol. 2, No. 4*, pages 437–445, Dec., 1994.

[28] S. Malik V. Tiwari and A. Wolfe. Power analysis of the intel 486dx2. *Tech. Report No. CE-M94-5*, Jan., 1994.