THINKING COMPLEXITY AND ENERGY COMPLEXITY OF SOFTWARE IN EMBEDDED SYSTEMS

Kostas Zotos, Andreas Litke, Alexander Chatzigeorgiou, George Stephanides, Maria Nikita
Dept. of Applied Informatics, University of Macedonia, 54006 Thessaloniki, Greece

ABSTRACT

In this paper we introduce the notion of energy complexity of Software in Embedded Systems and the thinking complexity of human brain. Which type of algorithm we use to solve a problem depends not only on time complexity, there are some other factors like readability, energy efficiency etc. which are very important in order to understand, use and teach an algorithm.

KEYWORDS

computational complexity; thinking complexity; performance evaluation; software design and implementation; analysis of algorithms; energy complexity

1. INTRODUCTION

Fifty years ago when computer first was built, people made the best effort on empowering the machine to do some intellectual work (i.e. computation, memory) on behalf of us. We like the computer “know” as much as it could. Through such process we gradually realized how we think and begin to think what we want and who we are. It is a philosophical problem.

The vast majority of microprocessors being produced today are incorporated in embedded systems, which are mainly included in portable devices. The later ones require the lowest power operation achievable, since they rely on batteries for power supply. Furthermore, high power consumption raises other important issues, such as the cost associated with cooling the system, due to the generated heat, as well as reliability concerns.

A lot of optimization efforts have been devoted to restructuring the hardware used, to decrease power consumption. However, recent research has proved that software is the dominant factor in the power consumption of a computing system [9], since it determines the number of energy consuming “0” to “1” transitions in the underlying digital circuitry. Here, we attempt, in analogy to the well established computational complexity (time complexity), to estimate the energy complexity for a given algorithm, as a means to characterize the expected energy consumption.

2. THINKING COMPLEXITY

When a graduate solves a mathematical problem he/she generally uses a “high level” pattern to obtain the solution immediately. This should not happen during his/her training. In a Mathematics course, the educators de-compose the solution process into elementary steps and reinforce each step by a “low level” use of a pattern. This approach is absolutely essential in order to understand what is thinking complexity.

Thinking complexity is the thought that is required for the solution of a problem. At the first stage the person detects a low level solution, next finds a solution more structured and continues until the final version. Thus, we have certain stages which follow each other. The initial stages require less thought.
We still know so little about the process of thinking and the structure of the thinking brain that any theory claiming to explain this phenomenon as a whole is hypothetical. Thus, our conception of thinking complexity must also be treated as a hypothesis. However, the core of human conception is not some hypothesis regarding the concrete structure and working mechanism of the brain, but rather a selection of those functional concepts through which a consistent and sufficiently convincing explanation of the facts we know about thinking becomes possible.

3. ENERGY CONSUMPTION

To clarify the reasons why energy consumption of a program varies, it is necessary to name the main sources of power consumption in a simplified model of an embedded system. System power falls into mainly two categories, each of which is described next.

3.1 Processor Power

When instructions are fetched, decoded or executed in the processor, the nodes in the underlying CMOS digital circuits switch states. Without getting into details, the dynamic power dissipation for a digital system (the power consumed during the switching of gates) is given by [1]:

\[ P_{avg} = a \cdot C_L \cdot V_{DD}^2 \cdot f_{clk} \]

where \( C_L \) is the total capacitance being switched, \( V_{DD} \) is the supply voltage, \( f_{clk} \) is the clock frequency, and \( a \) is the node transition activity factor. This factor essentially represents the average number of times a given node in the circuit makes a power consuming transition (from logic “low” to logic “high”). For a computational system, this factor is primarily determined by the executed software, which dictates the operation (signal transitions) in the circuit.

For any computing system, the switching activity associated with the execution of instructions in the processing unit, constitutes the so-called base energy cost [9]. The change in circuit state between consecutive instructions is captured by the overhead or inter-instruction cost. To calculate the total energy, which is dissipated, all that is needed is to sum up all base and overhead costs for a given program.

3.2 Memory Power

We assume that the system architecture consists of two memories, namely the instruction memory and data memory (Harvard architecture) [5] and that there is no cache memory. The energy consumption of the instruction memory depends on the code size and on the number of executed instructions that correspond to instruction fetches, whereas that of the data memory depends on the volume of data being processed by the application and on how often the later accesses data.

4. ENERGY COMPLEXITY

Any computational complexity measure is related to the running time [3]. The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed. We shall assume a generic one-processor, random-access machine (RAM) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations [3]. It is usually assumed that a constant amount of time is required to execute each line code. This viewpoint is keeping with the RAM model, and also reflects how the pseudocode would be implemented on most actual computers. The RAM model contains instructions commonly found in real computers: arithmetic (add, subtract, multiply, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.
In analogy to the computational complexity as described in [3] the energy complexity of an algorithm could be used to characterize the energy dissipation. The aim here is to extract a polynomial expression of the number of data memory accesses in addition to the expressions of the number of executed primitive operations or instructions. As a result, the resources of interest are not time or space but rather energy dissipation. In the following examples we do not employ the Big O notation so as to reveal even small differences between the algorithms under study.

In this way it is possible to come up with a polynomial that represents the number of accesses to the data memory. Since each access to a memory has a known (measurable) energy cost [8] the data memory energy consumption can be easily calculated.

On the other hand, polynomial expressions of the number of primitive operations are suitable for extracting the energy consumed within the processor and in the instruction memory. Since each primitive operation maps approximately to an independent assembly instruction whose energy consumption can be measured [6] and since the number of accesses to the instruction memory are equal to the executed assembly instructions, these two energy components can be easily obtained.

Thus, the total energy complexity would be calculated as:

$$E_{total} = c_1 \cdot E_{proc} + c_2 \cdot E_{instr\_mem} + c_3 \cdot E_{data\_mem},$$

where:

- $c_1 \cdot E_{proc}$ corresponds to the processor energy and is a polynomial expression of the number of primitive operations times a coefficient $c_1$. Coefficient $c_1$ corresponds to the average energy consumed during the execution of an assembly instruction and can be accurately estimated from physical power measurements [6] and profiling of representative applications.

- $c_2 \cdot E_{instr\_mem}$ corresponds to the instruction memory energy and is a polynomial expression of the number of primitive operations times a coefficient $c_2$. Coefficient $c_2$ corresponds to the energy cost of an access to the instruction memory.

- $c_3 \cdot E_{data\_mem}$ corresponds to the data memory energy and is a polynomial expression of the number of memory accesses times a coefficient $c_3$. Coefficient $c_3$ corresponds to the energy cost of an access to the data memory.

The coefficients $c_1$, $c_2$ and $c_3$ are dependent on the computer system used and $E_{total}$ is the total calculated energy complexity of the algorithm under study. However, a calibration for the corresponding coefficient values should be performed for each target platform.

5. FRAMEWORK SETUP

To evaluate the proposed energy complexity, a generalized target architecture was considered (Fig. 1). It is based on the ARM7 integer processor core [10], which is widely used in embedded applications due to its promising MIPS/mW performance [4]. The process that has been followed during the conduction of the aforementioned experiments begins with the compilation of each C++ code with the use of the compiler of the ARM Developer Suite [2]. At this stage, we were able to obtain the code size. Next and after the debugging, a trace file was produced which logged instructions and memory accesses. The debugger provided the total number of cycles. A profiler was specially developed for parsing the trace file serially, in order to measure the memory accesses to the instruction memory (OPCODE accesses) and the memory accesses to the data memory (DATA accesses). The profiler calculated also the dissipated energy (base + interinstruction energy) within the processor core. Finally, with the use of an appropriate memory simulator (provided by an industrial vendor), the energy consumed in the data and instruction memories was measured.

6. SOLVING A PROBLEM

The algorithm that we finally choose in a problem depends on what we want it to offer us: speed of implementation, facility in the comprehension, economy in the consumption of energy etc. We take for example the algorithms of sorting: Bubblesort, Quicksort and Shellsort (Table 1). In a book about information technology for children, where it is important that students learn certain basic knowledge, sorting
in the programs should be done with the algorithm Bubble sort, which is the simplest. On the other hand if we have a portable computer and we want to decrease the consumption of energy we select Shellsort which has the lowest energy complexity. Finally, if the time of implementation is the only criterion we choose the algorithm Quicksort that has least cycles of clock (time complexity).

In the following example we examine how complexities are altered using different programming techniques. As we realise from the results (Table 2, Fig. 1) the change of the programming technique increases the complexity of thought as well as time complexity. We cannot pass in the Object Oriented planning if we have not fundamentally resolved our problem in a lower form, consequently the Object Oriented planning has bigger complexity of thought from unstructured or structured programming because it includes them. The difference in thinking complexity of Procedural and Object-Oriented Programming exist because learning a program in an Object-Oriented style seems to be very difficult after being used to a procedural style. Anecdotal evidence indicates that it takes the average programmer 6 to 18 months to switch his/her mind-set from a procedural to an Object-Oriented view of the world[7]. The following algorithm was posed by L. Collatz in 1937. The code of each programming method is easily found.

The algorithm (for 3n+1 sequence)
A1. [Input n]
A2. [Termination] If n = 1 then exit
A3. [Check number] If n is even then n := n / 2 
   else n := n * 3 + 1
A4. [Next number] Go to A2.
A5. [Finish]

No Structured Programming
This is perhaps the first technique of programming the computer, without any procedure or anything else. The most problematic thing about this style is its management and reusability because there is almost no abstraction layer in this style of programming. Therefore, it is quite difficult to handle a large-scale program using this technique.

Structured Programming
Break down the program into small modules (functions), so it will become more manageable and more reusable than the non-structured version. Logically, one function of the program should do the unit work of the algorithm, but it is quite subjective to describe the unit work. This programming style will increase the abstraction level by dividing the problem into small pieces and solving those pieces one by one.

Recursive Programming
This is a simple example of recursion. In other words, a more typical definition may be something like this: "Recursion is a way to specify a process by repeating a means of itself."
In programming language context, recursion means a function that calls itself. Typical examples of recursion are factorial, Fibonacci number, Tower of Hanoi, and so forth.

Object-Oriented Programming
Object-Oriented Programming is a programming style in which you make classes and create instances of those classes, called objects. Technically, a class is a prototype that defines the variables and methods of one kind. This problem is very small, so that you can not get the full advantage of object-oriented programming. This program is technically object based but not object oriented.

Generic Programming
Generic means general or common. Technically, generic programming refers to a program written in such a way that it should work independent of data type. A generic program should work on all data types that satisfy the required concepts use in the program. In C++, a template is used for generic programming.

7. CONCLUSIONS

In this paper, we have introduced the notion of thinking complexity as a measure to evaluate the difficulty of a problem. Also we have shown how to evaluate the energy consumption which is nowadays a major
concern for the development of portable computing devices. Future research on these fields should include experimentation on larger and more complicated programs to establish the accuracy of the energy and thinking complexity.

Table 1. The results of sorting a table with 500 elements

<table>
<thead>
<tr>
<th>Type of Sort:</th>
<th>Bubble</th>
<th>Shellsort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Cycles</td>
<td>358,770</td>
<td>102,939</td>
<td>100,865</td>
</tr>
<tr>
<td>Opcode Mem. Accesses</td>
<td>269,984</td>
<td>80,179</td>
<td>60,033</td>
</tr>
<tr>
<td>Data Mem. Accesses</td>
<td>508,780</td>
<td>144,686</td>
<td>228,522</td>
</tr>
<tr>
<td>Processor Energy</td>
<td>3,639,386,46</td>
<td>0,111,327,362</td>
<td>0,093,958,360</td>
</tr>
<tr>
<td>Instr mem. Energy</td>
<td>10,273,047</td>
<td>0,305,226</td>
<td>0,251,491</td>
</tr>
<tr>
<td>Data mem. Energy</td>
<td>8,017,399</td>
<td>0,228,118</td>
<td>0,360,513</td>
</tr>
<tr>
<td>Total Energy</td>
<td>21,927,384,646</td>
<td>0,644,671,362</td>
<td>0,709,962,360</td>
</tr>
</tbody>
</table>

Table 2. Measurement results for different programming styles

<table>
<thead>
<tr>
<th>Programming:</th>
<th>No Structured</th>
<th>Structured</th>
<th>Recursive</th>
<th>OO</th>
<th>Generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Cycles</td>
<td>1,234</td>
<td>1,290</td>
<td>1,056</td>
<td>2,088</td>
<td>2,094</td>
</tr>
<tr>
<td>Opcode Mem. Accesses</td>
<td>1,063</td>
<td>1,260</td>
<td>1,044</td>
<td>1,692</td>
<td>1,692</td>
</tr>
<tr>
<td>Data Mem. Accesses</td>
<td>112</td>
<td>18</td>
<td>6</td>
<td>272</td>
<td>284</td>
</tr>
<tr>
<td>Processor Energy</td>
<td>0.001283301</td>
<td>0.001228216</td>
<td>0.0001079247</td>
<td>0.00216191</td>
<td>0.002194499</td>
</tr>
<tr>
<td>Instr mem. Energy</td>
<td>0.003496</td>
<td>0.004146</td>
<td>0.003434</td>
<td>0.005142</td>
<td>0.005139</td>
</tr>
<tr>
<td>Data mem. Energy</td>
<td>0.000282</td>
<td>0.000045</td>
<td>0.000015</td>
<td>0.000686</td>
<td>0.000716</td>
</tr>
<tr>
<td>Total Energy</td>
<td>0.005061301</td>
<td>0.005419216</td>
<td>0.004528247</td>
<td>0.00798991</td>
<td>0.008049499</td>
</tr>
</tbody>
</table>

Figure 1. Programming styles versus total energy and cycles

REFERENCES