Object-Oriented Analysis of Fibonacci Series Regarding Energy Consumption

KOSTAS ZOTOS, ANDREAS LITKE, GEORGE KAIAFAS, ALEXANDER CHATZIGEORGIOU, GEORGE STEPHANIDES
Department of Applied Informatics
University of Macedonia
156 Egnatia Street
54006 Thessaloniki, GREECE

Abstract: The importance of low power consumption is widely acknowledged due to the increasing use of portable devices, which require minimizing the consumption of energy. Energy dissipation is heavily dependent on the software used in the system. In this paper we analyze the energy consumption of Fibonacci series.

Key-Words: Software Engineering, Energy Consumption, Fibonacci Series

1 Introduction

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 heat generated. A lot of optimization efforts have been made, regarding the hardware used, to decrease power consumption [1]. However, recent research has proved that software is the dominant factor in the power consumption of a computing system [5].
Even though most of Embedded and Realtime programming is now carried out in high level languages, a good understanding of the generated assembly code really helps in debugging, performance analysis and performance tuning. Here we present a description of C++ to assembly translation. We will be analyzing the code generated by a compiler targeting the ARM processor family [7]. The concepts learnt here can easily be applied to understand the generated code for any other processor-compiler combination. We draw some useful conclusions regarding whether the energy consumption is increased with the use of different algorithms.

The rest of the paper is organized as follows: Section 2 provides a general overview of power consumption, while Section 3 explains important issues about Fibonacci Series. In Section 4 we describe the experimental framework setup. Next (Section 5), the results are presented and discussed whereas in the final section (Section 6) some conclusions are drawn.

2 Energy Consumption

In this section, we describe basic elements that characterize the energy consumption in a system. To clarify the reasons why energy consumption of a program varies, it is necessary to name the main sources of power consumption in an embedded system. The system power falls into mainly two categories, each of which is described in the following paragraphs.

2.1 Processor Power

When instructions are fetched, decoded or executed in the processor, the nodes in the underlying CMOS digital circuits switch states. For any computing system, the switching activity associated with the execution of instructions in the processing unit, constitutes the so-called base energy cost. The change in cir the overhead or inter-instruction cost. To calculate total energy, which is dissipated, all that is needed is to sum up all base and overhead costs for a given program.

2.2 Memory Power

We assume that the system architecture consists of two memories, namely the instruction memory and data memory (Harvard architecture). Having presumed that, the energy consumption has to be calculated on a twofold basis, one for each 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.

3 Fibonacci Series

The Fibonacci Series is a sequence of numbers first created by Leonardo Fibonacci in 1202. It is a deceptively simple series, but its ramifications and applications are nearly limitless. It has fascinated and perplexed mathematicians for over 700 years, and nearly everyone who has worked with it has added a new piece to the Fibonacci puzzle, a new tidbit of information about the series and how it works. Fibonacci mathematics is a constantly expanding branch of number theory, with more and more people being drawn into the complex subtleties of Fibonacci's legacy. The first two numbers in the series are one and one. To obtain each number of the series, you simply add the two numbers that came before it. In other words, each number of the series is the sum of the two numbers preceding it [6].

4 Framework Setup

To evaluate the energy cost of software design decisions a generalized target architecture was considered (Fig.1). It was based on the ARM7 integer processor core [7], 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 (Fig. 2) 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. The results we will present in the following section regard the number of cycles, the OPCODE Memory Accesses, the DATA Memory Accesses, the energy consumed in the processor, the data memory energy and the instruction memory energy.

5 Results

In this section we examine two algorithms: *non-recursive* and *binet's formula*. The results of the experiments are on Table 1, Figure 3. The C++ code [3] of these algorithms is the following:

Non-recursion

```
/*Returns the n'th Fibonacci number */
  unsigned int f(int n) {
  int i;
  unsigned int n,n1=1,n2=1;
    for(i=1; i<=n; i++) {
        n1+=n2;
        n=n1;
        n1=n2;
        n2=n;
    }
    return n1;
}
```

Binet's formula

```
/* Function f(n) returns the n'th Fibonacci number */
unsigned int f(int n) {
    if (n<2) return 1;
     double phi  = (1+sqrt(5))/2;
    return (pow(phi, n+1) - pow(1-phi, n+1)) / sqrt(5);
}
```

Surprisingly, although binet's formula is faster than non recursion algorithm(475-341=134 cycles), total energy is increased 4,56% ! The basic reason for this difference between Cycles and Total Energy is that some instructions are more energy effective than others. Also the code generation from assembly code sometimes doesn’t mirror the C++ code. For example code generation for the binet's formula use the inner function *pow* which has a high energy consumption. We can see what happen in a function calling to the ARM processor in the following example. Before do this we need to demonstrate some basics principles about ARM architecture.

5.1 ARM Basics

- Processors contain 8 data registers (D0-D7) and 8 address registers (A0-A7).

- The MOVE instruction has the source on the left side and destination on the right side.
- Register D0 is used to return values to the calling function.
- The stack grows from higher address to lower address. Thus a push results in a decrement to the stack pointer. A pop results in an increment to the stack pointer.
- Address register A7 is the stack pointer. The pre-decrement (i.e. decrement the register before use) addressing mode is used to implement a push. The post-increment addressing mode (i.e. use register and then increment) is used to implement a pop.
- A6 is used as the frame pointer. The frame pointer serves as an anchor between the called and the calling function.
- When a function is called, the function first saves the current value of A6 on the stack. It then saves the value of the stack pointer in A6 and then decrements the stack pointer to allocate space for local variables.
- The frame pointer (A6) is used to access local variables and parameters. Local variables are located at a negative offset to the frame pointer. Parameters passed to the function are located at a positive offset to the frame pointer.
- When the function returns, the frame pointer (A6) is copied into the stack pointer. This frees up the stack used for local variables. The value of A6 saved on the stack is restored.

5.2 Function Calling

The following block shows the C++ code and the corresponding generated assembly code.

```
int CallingFunction(int x)
{
  int y;
  CalledFunction(1,2);
  return (5);
}
void CalledFunction(int param1, int param2)
{
  int local1, local2;
  local1 = param2;
}
```

The generated assembly code is shown along with the corresponding code.

```
int CallingFunction(int x)
{   int y;
     * Reserving space for local variable y (4 bytes)
     LINK A6, #-4
  CalledFunction(1,2);
     * Pushing the second parameter on the stack
     MOVE.L #2, -(A7)
     * Pushing the first parameter on the stack
     MOVE.L #1, -(A7)
     * Calling the CalledFunction()
     JSR _CalledFunction
     * Pop out the parameters after return
     ADDQ.L #8, A7
  return (5);
     * Copy the returned value 5 into D0 (As a convention, D0 is used to pass return values)
     MOVEQ.L #5, D0
}
     * Freeing up the stack space taken by local variables
     UNLK A6
     * Return back to the calling function
     RTS

void CalledFunction(int param1, int param2)
{
  int local1, local2;
     * Reserving space for local1, local2 (4 bytes each)
    LINK A6, #-8
   local1 = param2;
   MOVE.L 12(A6), -4(A6)
}      * Freeing up the stack space taken by local variables
     UNLK A6
     * Return back to the calling function
     RTS
```

5.3 Function Calling Sequence

The generated assembly code is best understood by tracing through the invocation of *CalledFunction()* from *CallingFunction()*.

CallingFunction() pushes values 2 followed by 1 on the stack. These values correspond to param2 and param1 respectively. (Note that pushing order is reverse of the declaration order.). This is implemented by the MOVE.L instructions that implement a push operation by using the predecrement mode on A7.

CallingFunction() invokes the *CalledFunction()* by the JSR instruction. JSR pushes the return address on the stack and transfers control to *CalledFunction()*.

Called Function executes the LINK instruction. The link instruction performs the following operations:

- Saves the *CallingFunction()*'s frame pointer on the stack (i.e. content of A6 is saved on the stack)
- Moves the contents on the stack pointer (A7) into A6. This will serve as the frame pointer for *CalledFunction()*.
- Decrements the stack pointer by 8 to create space for the local variables local1 and local2.

After the LINK instruction, code in the *CalledFunction()* accesses passed parameters by taking positive offsets from the frame pointer. Local variables are accessed by taking negative offsets from the frame pointer.

Before the function ends UNLK is executed to undo the actions taken by the LINK instruction. This is done in the following steps:

- Copy the contents of A6 to A7. This will free the stack entries allocated for local variables local1 and local2.
- Pop the saved frame pointer from the stack. (This will make sure that the *CallingFunction()* gets its original frame pointer value on return).

The processor now executes the RTS instruction. This instruction pops the return address from the stack and transfers control to the *CallingFunction()* at this address.

The *CallingFunction()* now pops the parameters that were passed to the *CalledFunction()*. This is done by adding 8 to the stack pointer.

As we can see from the above analysis the assembly code, which is generated from function calling, has higher energy consumption. Approximately 22% more instructions are executed which produce a high energy cost.

6 Conclusions

The power consumption of an embedded system depends heavily on the executing code. The necessity to consider low energy consumption arises from the wide use of portable devices, which obviously require low power operation. In this paper, we have explored the energy consumed using an example (Fibonacci algorithm). We have observed that function calling causes high energy consumption. However, further research is required in order to investigate accurately the degree of this effect.

References:

[1] Chandrakasan, A., and R. Brodersen, “Low Power Digital CMOS Design”, Kluwer Academic Publishers, Boston, 1995

[2] Chatzigeorgiou, A., D. Andreou, and S. Nikolaidis, Description of the software power estimation framework, IST-2000-30093/EASY Project, Deliverable 24, February 2003

[3] Deitel, H.M., and P.J. Deitel, “C++: How to Program”, Prentice Hall, Upper Saddle River, 2001

[4] Furber, S., “ARM System-on-Chip Architecture”, Addison-Wesley, Harlow, UK, 2000

[5] Tiwari, V., S. Malik, and A. Wolfe, *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization*, IEEE Transactions on VLSI Systems, vol. 2 (1994), 437-445

[6] http://library.thinkquest.org/27890

[7] http://www.arm.com

Fig. 1: Target Architecture

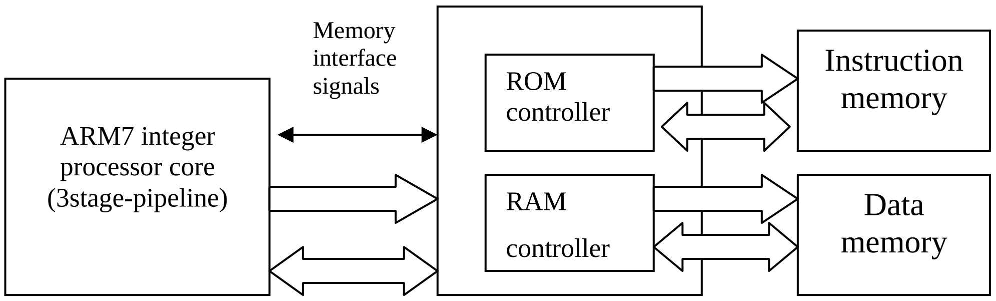

Fig. 2: Framework Steps

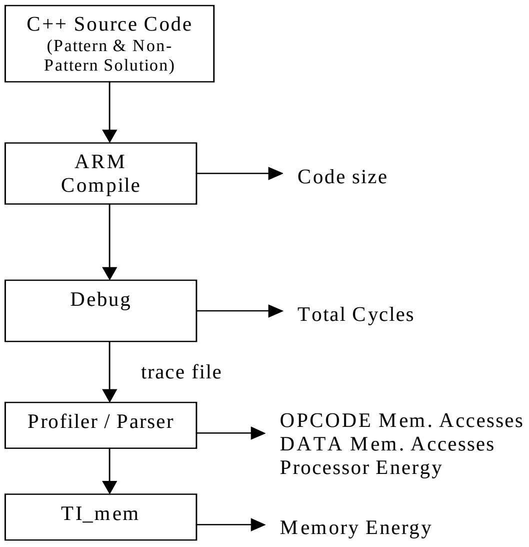

Fig. 3: Cycles and Total Energy

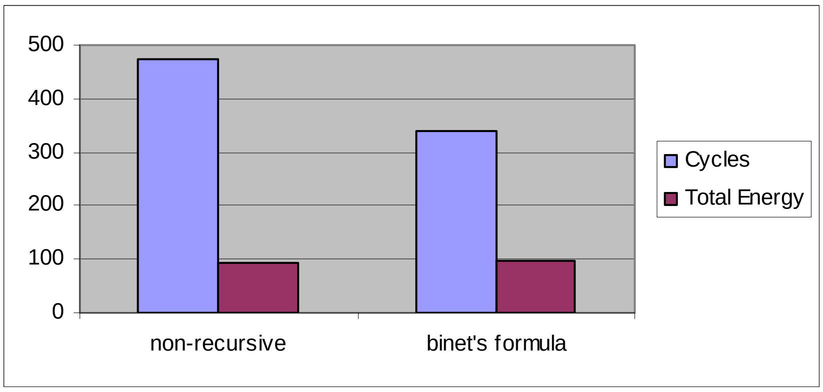

Table 1: Performance and Power

	non-recursive	binet's formula
#Cycles	475	341
Processor Energy	0,000497662mJ	0,000383579mJ
Instr mem. Energy	0,00042mJ	0,000377mJ
Data mem. Energy	0	0,000201mJ
Total Energy	0,000917662mJ	0,000961579mJ