Energy Consumption Analysis of Design Patterns

Andreas Litke, Kostas Zotos, Alexander Chatzigeorgiou, and George Stephanides

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. Applying design patterns in object-oriented designs is a common practice nowadays. In this paper we analyze six design patterns and explore the effect of them on energy consumption and performance.

Keywords—Design Patterns, Embedded Systems, Energy Consumption, Performance Evaluation, Software Design and Development, Software Engineering.

I. 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 [12].

Design patterns name, abstract and identify the key aspects of a common design structure that make it useful for creating a reusable object-oriented design [5]. Modern software design practice points towards the direction of design patterns, thanks to the advantages they ensue. Indeed, the software generated is reusable and flexible thus being much of a help to designers. For the aforementioned reasons, it is strongly recommended in the designers' community that they use design patterns whenever possible.

In this paper, we take a rather unconventional approach in evaluating the application of design patterns: We compare the energy being consumed in six C++ [11] code examples,

Andreas Litke is with the Applied Infromatics Department, University of Macedonia, 54006 Thessaloniki, Greece (e-mail: litke@uom.gr).

Kostas Zotos is with the Applied Infromatics Department, University of Macedonia, 54006 Thessaloniki, Greece (e-mail: zotos@uom.gr).

Alexander Chatzigeorgiou is with the Applied Infromatics Department, University of Macedonia, 54006 Thessaloniki, Greece (phone: +30 2310 891886; fax: +30 2310 891875; e-mail: achat@uom.gr).

George Stephanides is with the Applied Infromatics Department, University of Macedonia, 54006 Thessaloniki, Greece (e-mail: steph@uom.gr).

before the application of the appropriately chosen design pattern and afterwards. The aim is to quantify one aspect of software, namely the energy consumption of the underlying hardware, in cases where quality is substantially improved by the use of design patterns. We draw some useful conclusions regarding whether the energy consumption is increased with the use of design patterns (and if so, to what extent) or not. Of course, this is a first investigation and we by no means infer that the use of patterns generally increases or not the power consumption in a system.

II. 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.

A. 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 circuit state between consecutive instructions is captured by 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.

B. 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.

III. DESIGN PATTERNS

The need to design reusable and flexible object-oriented software, so that future possible problems can be overcome,

has led to the use of design patterns. Patterns are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience [6]. Each design pattern focuses on a particular problem or issue. Designers with design patterns can reuse solutions that have worked well in previous similar situations and so they do not need rediscover new ones. Design patterns even help improve the documentation of object-oriented software by explicitly specifying the class and object interactions and their intent [5]. There are a lot of different design patterns for common or not situations that need to be handled by the software designer. They have to pick up the appropriate one and adjust it to the given situation to gain reusability and flexibility.

IV. FRAMEWORK SETUP

To evaluate the energy cost of software design decisions generalized target architecture was considered. 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 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. Our initial assumption was that the power consumption of the pattern solution would be greater than that of the non-pattern solution. We also show the class diagram of each design pattern, using UML (Unified Modeling Language) [8], [9].

The design patterns selected for the experiments where of all three categories, namely Creational (Factory Method), Structural (Adapter), as well as Behavioral (Observer). Creational design patterns abstract the instantiation process. The system created is independent of how its objects are created, composed and represented. The intention of the Factory Method is to define an interface for creating an object, but to let the subclasses decide which class to instantiate. So, it lets a class defer instantiation to the subclasses. The Adapter pattern makes one interface conform to another, so that they are not incompatible. Behavioral patterns are mainly concerned with algorithms and the assignment of responsibilities between objects. The Observer pattern defines

a one-to-many dependency between objects so that one object alters its state, all its dependents are notified and updated automatically. We have also investigated the Bridge and the Composite patterns which belong to the Structural patterns. However, they both showed very little to none difference in power consumption and performance.

V. RESULTS

In this section we will present the results of the experiments. Starting with the Creational Pattern, we will first demonstrate the performance and energy consumption (Table I) before and after the application of the Factory Method design pattern (Fig. 2). It should be mentioned that in the non-pattern solution we have four classes in total (including three concrete products each of which having one method) while in the pattern solution there is an additional `Creator` class with one public static `factoryMethod`. Essentially what happens in the above example, is that code that is being executed within the client (function `main()`) in order to create instances of the three products, is moved within the `Factory` class method. Consequently, the code size is expected to increase to a small extent but the number of executed instructions remains almost unchanged. This is evident in Table I, where the accesses to the instruction memory do not increase significnantly, while the instruction memory energy consumption increases more due to the small increase in the code size. However, the increase in the total energy consumption is insignificant. Thus we can infer that the application of the Factory Method pattern does not worsen the energy consumption neither the performance of the program using it.

The next pattern we will examine is a Structural one, the Adapter design pattern (Fig. 1) and the corresponding results are shown in Table II.

In the non-pattern solution, the `Adapter` class invokes the methods of the parent `Adapted` class, while in the pattern solution `Adapter` class invokes the methods of "contained" `Adapted` class. The code in both solutions is similar and therefore there is no significant difference in the energy consumed and the performance between the two cases. "Heavy lifting" in both cases is delegated to the `Adapter` class. However, according to the well-defined Design Heuristic "*favor composition over inheritance*" the pattern solution should be the design of choice.

The last pattern we will examine is the Observer (Fig. 3), which is classified as a Behavioral design pattern. Again, we summarize the results measured in the Table III.

An object-oriented design that employs the Observer pattern, introduces one additional class to the system, namely the abstract `Observer` class. Consequently the code size increases therefore increasing in turn the instruction memory energy consumption, as it can be observed from Table III. The number of methods also increases (by the addition of the `attach` and `notify` methods) and consequently the number of executed assembly instruction also increases. Another

observation is that the number of load/store instructions in the pattern solution is larger. Load and store instructions require more than one cycle per instruction and in addition have a larger energy cost than arithmetic or branch instructions [10]. As a result, the number of executed cycles was greater when using the design pattern and we expected the energy consumed to be much different. This has been proved by measurements, since the difference between the two solutions is significant, with the pattern solution consuming a lot more energy than the non-pattern.

However, there is clear advantage by the use of polymorphism: the ability for objects of different classes related by inheritance to respond differently to the same message. As known, in C++ polymorphism is implemented via virtual functions: when a request is made through a base-class pointer to invoke a virtual function, C++ chooses the correct overridden function in the appropriate derived class. In our example, at pattern solution we used a virtual class. Every virtual function call not only requires additional execution time, but the Vtable constructs and Vtable pointers added to each object containing a virtual function, increase significantly the required memory and moreover lead to a tremendous increase of memory accesses. Judging from the increase in energy consumption, the decision lies to the designers whether such a solution is acceptable or not. They must balance the benefits of applying a design pattern and opting for a low power solution.

VI. CONCLUSIONS

The power consumption of an embedded system depends heavily on the executing software. The necessity to consider low energy consumption arises from the wide use of portable devices, which obviously require low power operation. The application of design patterns is a common practice due to the benefits they ensue. In this paper, we have explored the energy consumed and the performance before and after the application of design patterns on sample systems. We have observed that except for one example where the energy consumption of the pattern solution increased significantly (and thus making the application of the pattern questionable from a power point of view), the use of design patterns does not necessarily impose a significant penalty on power consumption. However, further research is required in order to investigate the effect of other design patterns on performance and power.

REFERENCES

[1] A. Chandrakasan, and R. Brodersen, "Low Power Digital CMOS Design", Kluwer Academic Publishers, Boston, 1995.
[2] A. Chatzigeorgiou, D. Andreou, and S. Nikolaidis, Description of the software power estimation framework, IST-2000-30093/EASY Project, Deliverable 24, February 2003, Available: http://electronics.physics.auth.gr/easy.
[3] H.M. Deitel, and P.J. Deitel, "C++: How to Program", Prentice Hall, Upper Saddle River, 2001.
[4] S. Furber, "ARM System-on-Chip Architecture", Addison-Wesley, Harlow, UK, 2000.
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.
[6] http://hillside.net/patterns/.
[7] http://www.arm.com/armtech/ARM7TDMI?OpenDocument.
[8] OMG Unified Modeling Language Specification, version 1.3, June 1999, Available: http://www.rational.com.
[9] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1999.
[10] G. Sinevriotis, and Th. Stouraitis, Power Analysis of the ARM 7 Embedded Microprocessor, *9th Int. Workshop on Power and Timing Modeling, Optimization and Simulation* (PATMOS'99), Kos, Greece, (1999), 261-270.
[11] B. Stroustrup, "The C++ Programming Language", 3rd Edition, Addison-Wesley, 1997.
[12] V. Tiwari, 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.
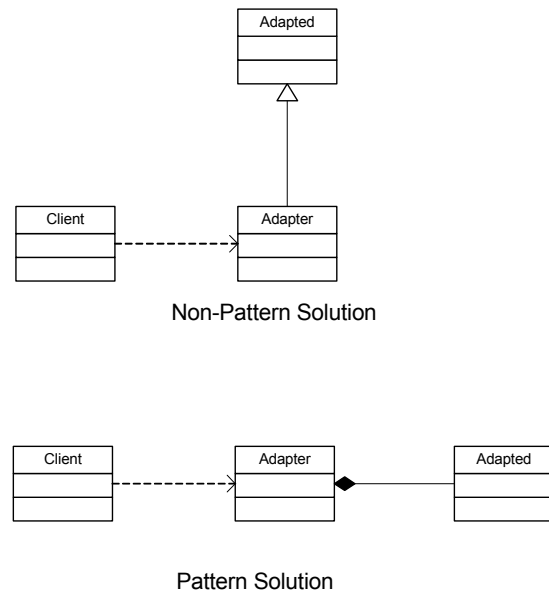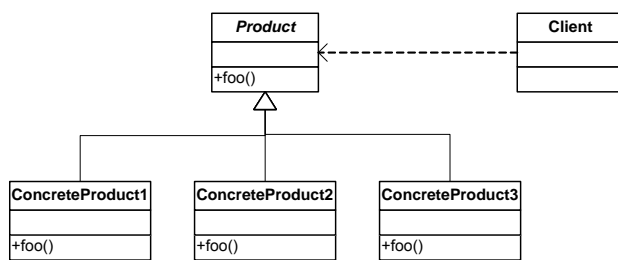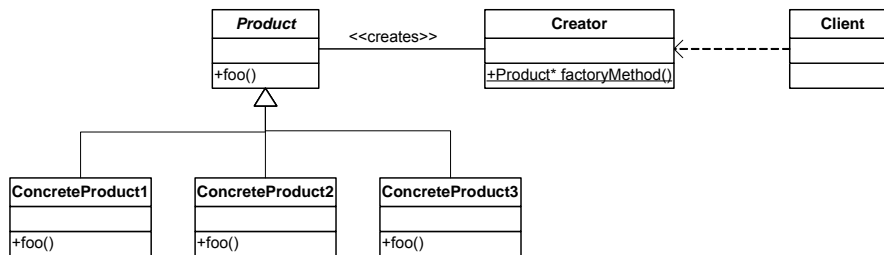
Fig. 1 Adapter Design Pattern diagrams

TABLE I
PERFOMANCE AND POWER FOR FACTORY METHOD PATTERN

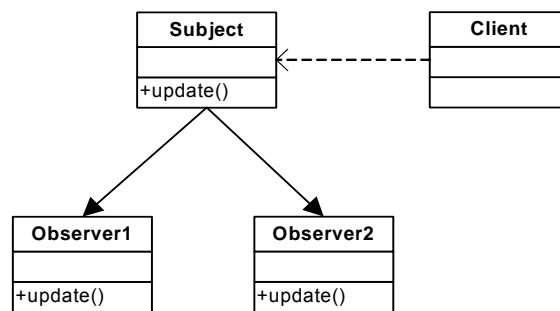|	Non-pattern solution	Pattern solution	Difference
# Cycles	17336	17358	0,127%
OPCODE Memory Accesses	10999	11005	0,055%
DATA Memory Accesses	4314	4322	0,185%
Processor Energy	0,019224856 mJ	0,019235994 mJ	0,058%
Instr_mem Energy	0,033464 mJ	0,03352 mJ	0,167%
Data_mem Energy	0,053743 mJ	0,053922 mJ	0,333%

Non-Pattern Solution

Pattern Solution

Fig. 2 Factory Method Design Pattern class diagrams

TABLE II
PERFOMANCE AND POWER FOR ADAPTER

	Non-pattern solution	Pattern solution	Difference
# Cycles	14312	14346	0,237%
OPCODE Memory Accesses	9608	9639	0,322%
DATA Memory Accesses	3186	3188	0,063%
Processor Energy	0,016087601 mJ	0,016123306 mJ	0,221%
Instr_mem Energy	0,032076 mJ	0,032179 mJ	0,320%
Data_mem Energy	0,043716 mJ	0,043744 mJ	0,064%

TABLE III
PERFORMANCE AND POWER FOR OBSERVER PATTERN

	Non-pattern solution	Pattern solution	Difference
# Cycles	37218	53615	44,06%
OPCODE Memory Accesses	23592	34027	44,23%
DATA Memory Accesses	9486	13618	43,56%
Processor Energy	0,04120657 mJ	0,059416259 mJ	44,19%
Instr_mem Energy	0,078097 mJ	0,112819 mJ	44,46%
Data_mem Energy	0,129948 mJ	0,187006 mJ	43,91%

Non-Pattern Solution
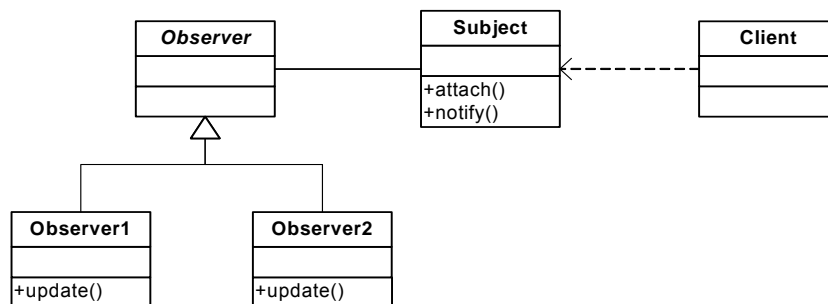
Pattern Solution

Fig. 3 Observer Design Pattern class diagrams