Energy Issues in Software Design of Embedded Systems

A. CHATZIGEORGIOU, G. STEPHANIDES
Department of Applied Informatics
University of Macedonia
156 Egnatia Str., 54006 Thessaloniki
GREECE
alec@ieee.org, steph@uom.gr

Abstract: - The increasing use of programmable processor cores in embedded systems which are mainly used in portable devices, creates an intense need for low power operation. Although power has been primarily addressed at the circuit/technology level it has become obvious that power consumption is heavily dependent on the executing software. The goal of this paper is to illustrate the impact of several software decisions on the energy consumption of the underlying hardware and the need to consider power in software design.

Key-Words: - Software Engineering, Low Power Design, Embedded Systems, Energy Consumption

1 Introduction

Around 90 percent of the microprocessors sold, end up in embedded systems, often not even recognizable as computers [1]. Such systems are often embedded in portable devices, which rely on batteries for power supply. Therefore, low power operation is required to prolong battery lifetime and to reduce weight. Moreover, power consumption causes heat dissipation leading to cooling/reliability issues, which are usually very expensive to solve.

Although power consumption is usually the goal of hardware optimizations [2] (supply voltage reduction, transistor sizing, operating frequency control) it has become clear that software has also a significant impact on the energy consumption of the underlying hardware. Software decisions are usually addressed at higher levels of the design hierarchy and therefore the resulting energy savings are larger. Moreover, software energy optimization aims at implementing a given task using fewer instructions and in this way there is no trade-off between performance and power as in the case of hardware.

Previous literature on energy aware software design includes research efforts that estimate power consumption using instruction level power models [3], [4], [5]. Based on the fact that inter-instruction energy costs differ for different pairs of instructions [3] several energy-optimizing techniques have been proposed that apply scheduling algorithms to minimize power consumption [4], [6]. Other methodologies target memory related power consumption and apply code transformations to reduce the number of accesses to memory layers [7].

The purpose of this paper is to highlight the effect of several software design decisions on the energy consumption of an embedded system. Power exploration is not restricted to the processor but also considers the energy consumption of the instruction and data memories. Representative programming paradigms have been selected and integrated into test programs, which have been evaluated in terms of performance and energy consumption.

The rest of the paper is organized as follows: Section 2 describes the target architecture and the experimental setup while Section 3 provides an overview of the sources of power consumption. In Section 4 several programming cases are explored and the impact of design decisions on power is discussed. Finally, we conclude in Section 5.

2 Target Architecture

To evaluate the energy cost of software design decisions a generalized target architecture will be considered (Fig. 1). It is based on the ARM7 integer processor core, which is widely used in embedded applications due to its promising MIPS/mW performance [8]. The instruction memory is an on-chip single-port ROM. The size of this memory is determined by the code size and to emphasize the energy variations as a result of software design, a custom memory fitting exactly the code is assumed. The data memory also resides on-chip. Both the

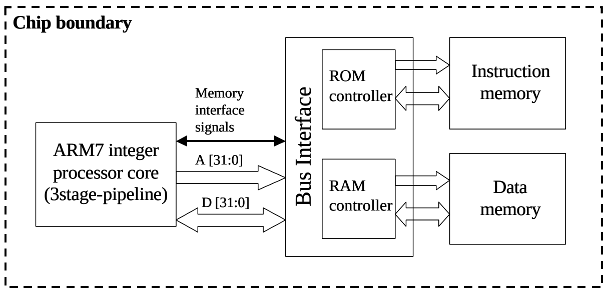

Fig. 1: Target architecture

instruction and data memories are connected to the processor with separate 32-bit busses. No data or instruction caches are considered which will lead to an overestimation of the energy consumption.

The experimental process that has been followed begins with the compilation of each code using the C or the C++ compiler of the ARM Software Development Toolkit, which provided the minimum ROM requirements for each program. Next, the execution of the code using the ARM Debugger provided the number of executed assembly instructions as well as the total number of cycles. The debugger was set to produce a trace file logging instructions and memory accesses. A separate profiler has been developed which parses the trace file serially, in order to obtain the energy that is consumed within the processor and the energy that is consumed in the data and instruction memories.

3 Sources of Power Consumption

In order to analyze the reasons that cause the variations in energy consumption of a program, the main sources of power consumption in an embedded system will be described next. There are three major components for the system power, namely the power that is consumed within the processor, the power dissipated on the instruction and data memories and the power related to the interconnect busses.

3.1 Processor Power

When instructions are fetched, decoded or executed within the processor, energy is consumed due to the switching activity of the corresponding circuits [2]. In other words, the charging/discharging of the node capacitances causes dynamic power dissipation. To quantify this power component, instruction level power models have been developed which calculate the energy consumption of a given program based on the trace of executed instructions. Each instruction is assumed to dissipate a specific amount of energy (base cost) and the change in circuit state between consecutive instructions is captured by the so-called overhead cost. Both the base and overhead energy cost can be measured, using the current drawn by the processor from the power supply. The total energy dissipation is obtained by summing all base and overhead energy costs and by multiplying them with the supply voltage and clock period.

3.2 Memory Power

Since the target architecture consists of separate instruction and data memories, energy consumption has to be extracted for each memory separately. This power component is related to the application: The instruction memory energy consumption depends on the code size, which determines the size of the memory and on the number of executed instructions that correspond to instruction fetches. The energy consumption of the data memory depends on the volume of data that are being processed by the application and on whether the application is data-intensive, that is whether data are often being accessed. For a typical memory power model, power is directly proportional to the number of accesses, and depends on the memory size, the number of ports, the power supply and the technology.

3.3 Interconnect Power

The interconnect lines that transfer data or instructions between the processor and the memory present large parasitic capacitances which are orders of magnitude larger than the node capacitances of the processor. Similarly to the switching activity in a digital circuit, energy consumption is proportional to the number of 0-1 and 1-0 transitions on the interconnect busses. This source of power will not be explored in this study; however, since it depends on the number of data being transferred, it can be considered that a larger number of accesses to the instruction and data memory will result in higher interconnect energy dissipation.

4 Impact of Software Decisions

In this section the impact of several software decisions on the energy consumption of the underlying hardware (ARM7 processor core, instruction and data memories) will be explored.

4.1 Less instructions less power

The most common approach in reducing the power consumption by means of software is to perform a given task by selecting the program with fewer instructions. To this end, the outcome of the research towards more efficient software can be exploited, since faster programs mean fewer instructions executed in the processor, fewer fetches from the instruction memory and therefore reduced energy consumption.

To illustrate this point, let us consider an example from matrix algebra, namely matrix-matrix multiplication. This computation can be performed using (a) a dot product computation, b) a generalized SAXPY operation and c) an outer product update. The three algorithms have been implemented in *C* with the programs mmdot.c, mmsax.c and mmout.c, respectively. The dot product multiplication is known to be the fastest one and this has a clear impact on the power consumption of the system

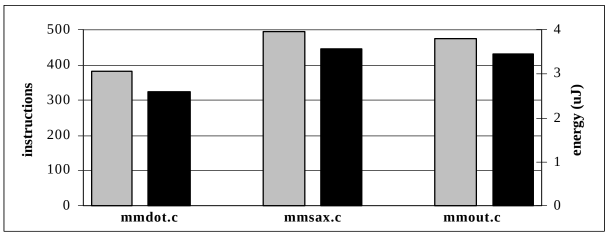

Fig. 2: Performance and energy consumption for matrix multiplication algorithms ▨ : instructions, ■ : energy

executing the algorithm. In Fig. 2 the number of executed instructions and the total energy consumption of the system is shown. In this example reducing the instruction count leads to a reduction of the energy consumption. However, this is not always the case as will be shown in the next paragraph.

4.2 Energy vs. Performance

This example is drawn from the multimedia domain and refers to the full search motion estimation algorithm. Motion estimation algorithms compare two consecutive frames of a video stream in order to obtain for each of the predefined blocks of an image, a motion vector which indicates for each block in the current frame the displaced block in the previous frame which matches best, according to a matching criterion [9]. The full search kernel consists of three double nested loops (Fig. 3). The outer loop selects all blocks in the current frame, the intermediate loop implements the displacement in both directions and the most inner loop is used for selecting all pixels in the block under study. Within the most inner loop, a check is performed whether the displaced pixel in the previous frame lies outside the frame and in that case its value is being read as zero. The evaluation of the matching criterion is omitted for clarity.

```
for(x=0;x<N/B;x++)      //for all blocks
 for(y=0;y<M/B;y++)     //in curr. frame

  for(i=-p;i<p+1;i++)   //for all candid.
   for(j=-p;j<p+1;j++)  //blocks

    for(k=0;k<B;k++)    //for all pixels
     for(l=0;l<B;l++)   //in the block
     {
      cur_pixel=current[B*x+k][B*y+l];
      if ((B*x+i+k) < 0       ||
           (B*x+i+k) > (N-1) ||
           (B*y+j+l) < 0     ||
           (B*y+j+l) > (M-1))
        prev_pixel = 0;
      else
        prev_pixel=
          previous[B*x+i+k][B*y+j+l];
     }
```

Fig. 3: Full search motion estimation algorithm

In such a data-intensive algorithm it is possible to rearrange (transform) the code in order to reduce the number of executed instructions. This kind of data-reuse transformations, are normally part of a Data Transfer and Storage Exploration (DTSE) methodology [7]. The primary target of such transformations is to move data accesses to smaller, less power costly memory layers.

One possible transformation introduces a line of reference windows as an additional buffer (array). The philosophy is to move data that are often being accessed in a short period of time to an intermediate array, in order to relax the complexity of the addressing equations and the conditional statements in the most inner loop, and in this way to reduce the overall number of executed instructions.

Considering for the sake of simplicity a frame consisting of 2×3 blocks, a block size of 8×8 pixels and a search space of [-7,7], the results in Table I concerning performance and power are obtained.

As it can be observed, the applied transformation works well concerning the number of executed instructions, which are reduced by 13.04 %. Due to the simplification of the addressing equations and the conditional statements in the most inner loop a reduction in the processor power consumption is also achieved. Using intermediate arrays, data can be addressed in a simpler manner and check of array boundaries can be performed using fewer inequalities. For example, in order to check in the original code whether the accessed pixel of the previous frame lies outside the frame, four inequalities have to be considered, implemented by 8 assembly instructions. For the transformed code, only two inequalities have to be checked leading to an implementation of only 4 instructions. Since these instructions are nested within 6 loops with a total number of 86.400 iterations, the reduction in the instruction count is obvious.

However, the reduced number of executed instructions in this case comes at the cost of an increased number of accesses to the data memory (+13.21%), which is reasonable since additional accesses are required to transfer data from the previous frame to the introduced buffer. Since the

Table I: Performance and Power for full search algorithm

	original code	**transformed code**	**Diff.**
#instructions	1321145	1148838	- 13.04 %
#cycles	2002043	1833656	- 8.41 %
#mem accesses	142338	161140	+ 13.21 %
processor energy	6.710 mJ	6.159 mJ	- 8.21 %
data_mem energy	2.262 mJ	2.561 mJ	+ 13.21%
instr_mem energy	2.162 mJ	1.880 mJ	- 13.04 %
total energy	11.136 mJ	10.602 mJ	- 4.79 %

energy consumption of a memory is proportional to the number of accesses, an equal increase is observed in the data memory energy consumption.

This is a typical example of the trade-off between performance and power consumption and proves that writing code bearing in mind only performance constraints can lead to inefficient use of energy resources. In the above example the final energy savings at the system level are far less impressive than the savings, which result when only the power due to executed instructions is considered and if memory size were larger, total power consumption could have been increased for the transformed code.

4.3 Call by value vs. Call by reference

When an argument is passed call-by-value to a function, a copy of the argument's value is made and the called function cannot affect the original variable in the caller. To overcome this, it is possible to pass the address of a variable (e.g. a pointer) to the function, simulating call-by-reference, in order to let the function modify the variable in the caller. In case of a single argument, an alternative would be to pass an argument call-by-value, let the called function modify its value and then return its value that has to be read and assigned to the variable of the caller. Another possibility is to employ global variables, which can be accessed and modified both by the called function and the caller (although this solution is not recommended since it increases coupling). These three alternatives have all a different impact on energy consumption and are listed below together with the corresponding assembly statements:

In case variable x is declared outside any function, it has file scope and is globally accessible. In this case, function setData( ) stores the value of register r1 (which holds the value to be assigned to x) to the memory address indicated by r0 plus a zero offset (base plus offset addressing mode). Exactly two memory accesses are performed at each call of setData( ), one for reading the address of x and one for storing the value of x to the specified memory location. At the end of the function, the value of the link register r14 that carries the return address [8], is transferred to the program counter (pc) in order to return at the address following the function call.

On the other hand, when passing the address of variable y to function setData, the value of y is stored at the memory address, which is indicated by the value of register r13 plus a zero offset. Then, it is the value of register r13 that is passed as argument to function setData through register r0, which on entry of function SetData is transferred to

Call by value (use of global variables)

```
C code                 Assembly (interleaved)
int x=0;
void setData( ){
  x = 5;               mov r1,#5
                       ldr r0, 0x000080dc
                       str r1, [r0, #0]
}
                       mov pc,r14

void main( ){
  setData( );          bl setData
}
```

Call by Reference

```
C code                 Assembly (interleaved)
void setData(int *x){
                       mov r1,r0
  *x = 5;
                       mov r0,#5
                       str r0,[r1,#0]
}
                       mov pc,r14
void main( ){
  int y = 0;
                       mov r0,#0
                       str r0,[r13,#0]
  setData(&y);
                       mov r0,r13
                       bl setData
}
```

Call by value with return

```
C code                 Assembly (interleaved)
int setData(int x){ mov r1,r0

  x = 5;               mov r1,#5

  return x;            mov r0,r1
                       mov pc,r14
}
void main( ){
  int y = 0;           mov r2,#0

  y = setData(y);      mov r0,r2
                       bl setData
                       mov r2,r0
}
```

register r1. The result of every operation affecting the value of this argument, is written back to the corresponding memory location using register r1 which holds the memory address of y.

When a single value is returned by function setData, the value of y is passed as argument through register r0 (which is one of the four registers used for argument passing in ARM [8]). Subsequently its value is transferred (copied) to register r1, which is the local copy of the argument which will be processed by the function code. The value of variable x in function setData is returned before exit to register r0, which will be accessed by the calling function to read the return value of the

Table II: Performance and power for passing arguments

	Use of global	Call by ref.	Call by value with return
#instructions	10004	11007	12005
#cycles	23006	22010	22007
#mem accesses	2000	1001	0
processor energy	0.0525	0.0488	0.0473
data_mem energy	0.0230	0.0115	0
instr_mem energy	0.0089	0.0098	0.0106
total energy	0.0844	0.0701	0.0580

called function. It is obvious that in the above example the value of y need not be passed as argument, however it is included for generality.

The program that has been used to compare ways of passing values between functions in terms of power makes thousand calls to function setData within a loop. The results, derived using the C compiler of the ARMulator, are shown in Table II.

As it can be observed, when a return value is used for function setData the number of executed assembly instructions it the largest between all alternatives, since 3 instructions are involved in calling function setData from main (one for branching and two for passing and reading a value from register r0) instead of 2 instructions for the case of calling setData passing a pointer. However, in the case of setData with a return value, only register operations are involved, **leading to zero memory accesses.** This, as it will be shown next, reduces the system power consumption significantly and indicates that any decision, which is made on a first thought based on the number of executed instructions, can be misleading.

It is obviously much more efficient in terms of power to pass an argument employing call by value and then read its modified value using the return value of the function than passing a pointer as argument or using a global variable. Call by value offers also the lowest instruction-level (processor) power consumption, in spite of the fact that for this case the number of executed instructions is larger. That is because no load/store instructions are used which require more than once cycle per instruction and in addition have a larger base energy cost than arithmetic or branch instructions [5]. The previous observation is also valid for the case of passing multiple arguments: It is always preferable in terms of power to pass and read one of the parameters by call by value.

4.4 Dynamic vs. Static Binding

One of the main advantages of object-oriented design is polymorphism - the ability for objects of different classes related by inheritance to respond differently to the same message. 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 [10]. Since the function, which will be invoked is not known at compile time, these function calls are resolved at run-time with dynamic binding. On the other hand, when the compiler is able to determine the function which will be called, for example when non-virtual functions are called through a pointer of the corresponding class, the function calls are resolved with static binding. The simple example below, employing a base and a derived class, illustrates the use of static and dynamic binding (constructors and destructors are omitted):

Static Binding

```
class parent {
   private:
       int x;
   public:
       parent(int a) {x=a;}
       void set();            };

class child:public parent {
   private:
       int x;
   public:
       child(int a) {x=a;}
       void set();            };

void parent::set() { x=10; }

void child::set() { x=20; }

int main(){
  parent par1(5);
  child chi1(7);
             //base-class pointer
  parent *parentPtr = &par1;
             //derived-class pointer
  child *childPtr = &chi1;

  parentPtr -> set();    //static bind.
  childPtr -> set();     //static bind.
  return 0; }
```

Dynamic Binding

In this case, function set( ) in the base class is declared **virtual**, and function main takes the form:

```
int main(){
  parent par1(5);
  child chi1(7);
                  //base-class pointer
  parent *parentPtr = &par1;

  parentPtr -> set(); //dynamic bind.

                 //base-class pointer points
                 //to derived class object
  parentPtr = &chi1;
  parentPtr -> set(); //dynamic bind.
  return 0; }
```

In the static binding example, calls to function set() through pointers of the corresponding class are compiled to a jump to the address of the corresponding function, e.g. :

```
C++ code                  Assembly
parentPtr -> set();       bl set_6parentFv
```

The only accesses to memory are being made in order to store the values for variable x within function set( ). In the dynamic binding example, when statement `parentPtr->set()` is compiled, the compiler determines that the call is being made off a base-class pointer and that set( ) is a virtual function. Taking into account that a pointer to the Virtual Function Table (VTable) of each class, exists in front of each object, the address of VTable for the corresponding object (par1 or chi1) is loaded into a register (r1 in the following example) and then execution continues from the address in the VTable that points to function set( ).

Consider for example the following code (function set( ) is a virtual function), where the program counter is loaded with the address in the VTable that contains function set( ) for class par1:

```
C++ code                  Assembly
parent *parentPtr = &par1;

parentPtr -> set();       ldr r1,[r13, #0xc]
                          mov r14, pc
                          ldr pc, [r1,#0]
```

As a result, 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 an tremendous increase of memory accesses. Table III compares dynamic versus static binding for the previous programs in terms of both performance and power, while the last column presents the introduced penalty when virtual functions are employed.

It becomes obvious, that despite the merits of dynamic method binding concerning flexibility and reuse, a significant performance and power penalty should be expected when opting for virtual functions and polymorphic behavior.

5 Conclusion

The power consumption of an embedded system is heavily dependent on the executing software. Taking into account the intense requirement for low power operation of portable computing devices, it becomes a necessity to consider energy consumption during software design. Through a number of examples it has been illustrated that careful design decisions can significantly affect the power consumption of the underlying hardware. Particularly, the fact that a major component of the total power is due to memory accesses, alters the traditional programming approaches, which aim only at improving efficiency.

Table III: Performance and power results for dynamic vs. static binding

	Static Binding	Dynamic Binding	Penalty (%)
#instructions	15	22	46.67
#cycles	27	48	77.78
#mem accesses	4	12	200
processor energy	0.000058	0.000110	89.66
data_mem energy	0.000046	0.000138	200
instr_mem energy	0.000013	0.000020	53.85
total energy	0.000117	0.000268	129.06

References:

[1] B.Santo, Embedded Battle Royale, *IEEE Spectrum*, vol. 38, no. 12, 2001, pp. 36-41.

[2] A. Chandrakasan, R. Brodersen, *Low Power Digital CMOS Design*, Kluwer Academic Publishers, Boston, 1995.

[3] V. Tiwari, S. Malik, A. Wolfe, Power Analysis of Embedded Software: A First Step Towards Software Power Minimization, *IEEE Transactions on VLSI Systems*, vol. 2, no. 4, 1994, pp. 437-445.

[4] V. Tiwari, S. Malik, A. Wolfe, T.C. Lee, Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing*, vol. 13, no. 2, 1996, pp. 1-18.

[5] 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, pp. 261-270.

[6] T.C. Lee, V. Tiwari, S. Malik, M. Fujita, Power Analysis and Minimization Techniques for Embedded DSP Software, *IEEE Trans. on VLSI Systems*, vol. 5, no. 1, 1997, pp. 123-135.

[7] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*, Kluwer Academic Publishers, Boston, 1998.

[8] S. Furber, *ARM System-on-Chip Architecture* Addison-Wesley, Harlow, UK, 2000.

[9] V. Bhaskaran, K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic Publishers, Boston, 1999.

[10] H.M. Deitel, P.J. Deitel, *C++: How to Program*, Prentice Hall , Upper Saddle River, 2001.