Design and implementation of an improved C source-code level program energy model

Xiang Zhou  
School of Computer Science & Engineering, SiChuan University, ChengDu, SiChuan 86-28-610064 China  
xiangchow@163.com

Yan Shen  
University of Electronic Science & Technology of China, ChengDu, SiChuan 86-28-610054 China  
Shenyan02@163.com

Bing Guo  
School of Computer Science & Engineering, SiChuan University, ChengDu, SiChuan 86-28-610064 China  
guobing@scu.edu.cn

Qi Li  
School of Computer Science & Engineering, SiChuan University, ChengDu, SiChuan 86-28-610064 China  
liqi_scu@126.com

Abstract—In embedded systems, the frequent execution of software drives the activities of hardware, which causes the energy consumption of systems. In this paper, we first propose an improved C source-level code program energy model by considering the energy consumption of instruction fetching phase, in which a accurate energy algorithm is used. Then, we implement a method to dynamically calculate the run-time energy consumption between program breakpoints through GDB. The simulation experiment results show that, at the same condition, we can obtain the more accurate energy consumption values between program breakpoints flexibly through GDB, and this provides some conveniences for us to optimize the program energy consumption in the future.

Keywords- EMSIM; i-EMSIM; GDB

I. INTRODUCTION

With the rapid development of embedded computing technology, embedded systems are increasingly small and powerful, e.g., a miniature SoC (System-on-a-chip) can implement diverse functions, including general-purpose computation, image processing and wireless communication, etc. Due to the power consumption limitation, the duration, cost and the use of high performance components of embedded systems are greatly affected. The power consumption is becoming one of the main constraints in embedded system design. The frequent execution of software drives the activities of hardware, which causes the energy consumption of systems. Currently, many advanced hardware power control technologies and system power management policies have been used. The question of how to reduce the software energy consumption starts to become an important one for controlling and optimizing the overall power consumption of systems.

At present, the research of software energy consumption often depends on software simulator, which can rapidly acquire some needed energy consumption data. Documents [1] and [2] point out that the EMSIM (Embedded StrongARM Energy Simulator) not only can simulate the common embedded systems, e.g., embedded Linux platform and its embedded Linux applications, but also can dynamically calculate the software energy consumption.

In this paper, we first discuss the improvement of energy consumption calculation model in the simulator, which we term as i-EMSIM. Then, we add some new functions in the GDB on Linux platform to dynamically calculate the source-code level program energy consumption between breakpoints. This will give us much convenience to study the energy consumption of applications in the future.

II. STRUCTURE OF I-EMSIM AND PROGRAM ENERGY MODEL

A. Structure of i-EMSIM

i-EMSIM, an improved EMSIM, is implemented based on the GDB/ARMulator, on which we have made some modifications and extensions to meet the requirement of energy consumption data acquisition. It can communicate with GNU GDB, and simulate a variety of complete embedded systems. In general, i-EMSIM can be divided into 4 modules: user interface module, symbols processing module, target control module and target simulation module.

User interface module contains the command-line user interface and the graphical user interface, which implements the user command input, and outputs the relevant debugging data to user. We almost use the user interface module of GDB directly except for making a little expansion.

Symbols processing module mainly deals with the head information of executable file, explains and executes the debugger information embedded in the file, manages the
symbol table, analyzes the code expressions, locates the position of the source code and binary code.

Target control module mainly controls the execution of program, e.g., sets the conditions of program interruption, analyzes the structure of program stack and uses several ways to debug applications, e.g., local debugging, remote debugging and simulation debugging.

Target simulation module is the core module of i-EMSIM, which simulates the running of the main hardware of embedded systems, including CPU, memory and peripherals, interpreters the binary files into machine instructions, executes each machine instruction and causes the corresponding hardware module to respond.

Depending on i-EMSIM’s function, target simulation module can be further divided into several modules that including configuration options analysis and initialization module, processor processing macro module, CPU instruction simulation module, MMU/Cache modules, coprocessor simulation module, I/O simulation macro module, the system I/O simulation module, and memory simulation module. The simulated hardware structure of i-EMSIM target simulation module is shown in Fig. 1.

![The simulated hardware structure of i-EMSIM](image)

Figure 1. The simulated hardware structure of i-EMSIM target simulation module

### B. An improved program energy model in i-EMSIM

In Fig. 2, an improved energy model is described, which uses the various components of i-EMSIM to collect the energy consumption. Since the main purpose of building i-EMSIM is to analyze the energy consumption from the software perspective, we do not resort to analytical energy models. In fact, Simunic et al. [3] have demonstrated that reasonably accurate estimation (within 5% error) can be obtained even if the power models of the constituent components are inferred directly from the data-sheet information for the system components. Since the energy consumption of instruction fetching phase has not been considered before, we will focus on it to improve the program energy model.

In our work, we follow this philosophy for most components of i-EMSIM except for the processor, for which we have obtained the StrongArm SA-1100 instruction-level energy model. As we know, an instruction needs at least three stages to execute, which are instruction fetching, decoding and execution. So, the total energy consumption of embedded systems is equal to the sum energy consumption of all implemented instructions. Generally, the energy consumption of StrongARM’s idle mode contributes to the energy consumption of system execution, since many components are still active. The total energy consumption of embedded systems can be accumulated as follows:

$$E_{\text{total\_system}} = E_{\text{total\_fetch}} + E_{\text{total\_decode}} + E_{\text{total\_execute}}$$  \hspace{1cm} (1)

Where $E_{\text{total\_system}}$ denotes the total accumulated system energy consumption, $E_{\text{total\_fetch}}$ denotes the accumulated instructions fetching energy consumption, $E_{\text{total\_decode}}$ denotes the accumulated instructions decoding energy consumption, and $E_{\text{total\_execute}}$ denotes the accumulated instructions execution energy consumption.

When fetched instructions are executed, it needs to access memory through the external bus. So, the energy consumption of instruction fetching phase almost is equal to the energy consumption of accessing memory. From the data provided by [4], we estimate that the memory energy consumption for each bus clock cycle is $4.70\, \text{nJ}$ (denoted as $E_{\text{mem\_cyc}}$). The accumulated instruction fetching energy consumption is calculated as:

$$E_{\text{total\_fetch}} = E_{\text{mem\_cyc}} \times N_{\text{mem\_cyc}}$$  \hspace{1cm} (2)

Where $N_{\text{mem\_cyc}}$ is the number of memory cycles when the memory part is active.

After one instruction is fetched, embedded processor decodes the instruction. So, the decoded instruction energy...
consumption is incurred. We calculate $E_{\text{total\_decode}}$ as:

$$E_{\text{total\_decode}} = E_{\text{decode}} \times N_{\text{decode\_cy}}$$  \hspace{1cm} (3)$$

Where $E_{\text{decode}}$ is the energy consumption of decoded instructions per cycle, and $N_{\text{decode\_cy}}$ is the total number of processor core cycles when the decoder is active.

According to [5], during one instruction is executed, both the CPU and the various peripheral components are running, e.g., memory, UART, etc. We use the following formula to calculate this stage’s energy consumption:

$$E_{\text{total\_execute}} = E_{\text{total\_proc}} + E_{\text{total\_idle}} + E_{\text{total\_mem}} + E_{\text{total\_uart}} + E_{\text{total\_peri}}$$  \hspace{1cm} (4)$$

Where $E_{\text{total\_proc}}$ is the accumulated processor energy consumption in the active state, $E_{\text{total\_idle}}$ is the accumulated processor energy consumption in the idle state, $E_{\text{total\_mem}}$ is the accumulated memory energy consumption, $E_{\text{total\_uart}}$ is the accumulated UART energy consumption, and $E_{\text{total\_peri}}$ is the accumulated energy consumption of other peripherals.

The accumulated processor energy consumption is calculated based on the instruction-level energy model as follows:

$$E_{\text{total\_proc}} = \sum_{i=1}^{N_{\text{total\_instr}}} E_{\text{proc\_type}(i)} \times N_{\text{cy}(i)}$$  \hspace{1cm} \hspace{1cm} (5)$$

Where $N_{\text{total\_instr}}$ is the total number of executed instructions, $E_{\text{proc\_type}(i)}$ is the processor energy consumption of instructions type $\text{instr\_type}(i)$ per cycle, and $N_{\text{cy}(i)}$ is the number of cycles required to execute one instruction $i$.

To calculate the corresponding energy consumption per processor core cycle when an instruction is executed, we use the following conversion formula:

$$E_{\text{proc}} = \frac{V_{dd} \times I_{\text{instr}}}{f_{clk}}$$  \hspace{1cm} (6)$$

In our experiments, the core supply voltage $V_{dd}$ used was 1.46V, and the clock frequency $f_{clk}$ was 206MHz. The instruction current $I_{\text{instr}}$ ranges from 0.165A to 0.237A. Using Equation (6), we determine that the energy consumption of processor core in each cycle range from 1.17nJ to 1.68nJ, depending on the instruction being executed. The detailed $E_{\text{proc}}$ is categorized by instruction groups are listed in Table I.

<table>
<thead>
<tr>
<th>Instruction group</th>
<th>$E_{\text{proc_type}}$ (nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>1.23</td>
</tr>
<tr>
<td>ADD</td>
<td>1.26</td>
</tr>
<tr>
<td>AND</td>
<td>1.23</td>
</tr>
<tr>
<td>BIC</td>
<td>1.23</td>
</tr>
<tr>
<td>CMN</td>
<td>1.28</td>
</tr>
<tr>
<td>CMP</td>
<td>1.28</td>
</tr>
<tr>
<td>EOR</td>
<td>1.27</td>
</tr>
<tr>
<td>MOV</td>
<td>1.25</td>
</tr>
<tr>
<td>MVN</td>
<td>1.25</td>
</tr>
<tr>
<td>ORR</td>
<td>1.27</td>
</tr>
<tr>
<td>RSB</td>
<td>1.28</td>
</tr>
<tr>
<td>RSC</td>
<td>1.28</td>
</tr>
<tr>
<td>SBC</td>
<td>1.28</td>
</tr>
<tr>
<td>SUB</td>
<td>1.27</td>
</tr>
<tr>
<td>TEQ</td>
<td>1.27</td>
</tr>
<tr>
<td>TST</td>
<td>1.46</td>
</tr>
<tr>
<td>MLA</td>
<td>1.46</td>
</tr>
<tr>
<td>MUL</td>
<td>1.31</td>
</tr>
<tr>
<td>LDR</td>
<td>1.47</td>
</tr>
<tr>
<td>LDRB</td>
<td>1.47</td>
</tr>
<tr>
<td>LDRBT</td>
<td>1.31</td>
</tr>
<tr>
<td>LDRT</td>
<td>1.34</td>
</tr>
<tr>
<td>STR</td>
<td>1.63</td>
</tr>
<tr>
<td>STRB</td>
<td>1.62</td>
</tr>
<tr>
<td>STRT</td>
<td>1.63</td>
</tr>
<tr>
<td>LDM</td>
<td>1.68</td>
</tr>
<tr>
<td>STM</td>
<td>1.62</td>
</tr>
<tr>
<td>MRS</td>
<td>1.23</td>
</tr>
<tr>
<td>MSR</td>
<td>1.20</td>
</tr>
<tr>
<td>SWP</td>
<td>1.17</td>
</tr>
<tr>
<td>B</td>
<td>1.20</td>
</tr>
<tr>
<td>NOP</td>
<td>1.22</td>
</tr>
</tbody>
</table>

i-EMSIM also supports StrongARM’s idle mode. In this mode, the clock of processor core is stopped, but all other peripheral resources, e.g., clock generator, timer, interrupt controller, UARTs, power manager, etc., are still active. When an interrupt occurs, the processor core is re-activated. When the Linux kernel without any task loaded is running on the processor, the idle task in the kernel kicks the processor into idle mode. According to [4], the power consumption of the processor in idle mode is 65mW at a clock frequency of 206MHz. Since the clock generator is still running in idle mode, we can calculate the total idle energy consumption based on the clock tick (same as clock cycle, the smallest unit of time recognized by processor):

$$E_{\text{total\_idle}} = P_{\text{idle}} \times T_{\text{cy}} \times N_{\text{idle\_cy}}$$  \hspace{1cm} (7)$$

Where the idle power consumption $P_{\text{idle}}$ often is 65mW, $T_{\text{cy}}$ is the processor core clock period, and $N_{\text{idle\_cy}}$ is the total number of core clock during the processor stays idle.

When cache misses occur, additional energy is consumed as a result of external bus and memory access. So, the accumulated memory energy consumption that similar as the stage of instruction fetching is calculated as:

$$E_{\text{total\_mem}} = E_{\text{mem}} \times N_{\text{mem\_cy}}$$  \hspace{1cm} (8)$$
According to [4], the additional energy consumption incurred by UARTs is fairly constant at approximately 44mW. At a core clock frequency of 206MHz, we estimate that the energy consumption of UART module in each core cycle is 0.21nJ (denoted as \( E_{\text{uart}} \)). We calculate \( E_{\text{total, uart}} \) as:

\[
E_{\text{total, uart}} = E_{\text{uart}} \times N_{\text{uart, cyc}}
\]  

(9)

Where \( N_{\text{uart, cyc}} \) is the total number of core cycles when the UART is active. Noting that the UART can be active whether the processor core is in idle mode or not.

Energy models for other peripherals can be added in the same manner like the UART, i.e., the accumulated peripheral energy consumption \( E_{\text{total, peri}} \) can be calculated as:

\[
E_{\text{total, peri}} = E_{\text{peri}} \times N_{\text{peri, cyc}}
\]  

(10)

Where \( E_{\text{peri}} \) is the per-cycle energy consumption of one peripheral module, and \( N_{\text{peri, cyc}} \) is the total number of core cycles when the peripheral module is active.

III. ENERGY CONSUMPTION BETWEEN PROGRAM BREAKPOINTS

A. Communication mechanism between GDB and i-EMSIM

GDB is a powerful debugger, in which set breakpoints, step in/step over executing, show/modify variable values, show/modify the registers’ values, watch the programs’ stacks, debug remote programs, and debug threads, etc, can be implemented. Generally, there are two means of communication between i-EMSIM and GDB. One is directly to call the i-EMSIM’s function through GDB, another is to use socket or pipe transport RDP protocol to connect GDB and i-EMSIM. Since the first means is simpler and more convenient, we use the first mean as the communication between i-EMSIM and GDB (shown in Fig 3).

![Figure 3. Communication mechanism between GDB and i-EMSIM](image)

After GDB is running, we need execute the command `target sim` to make the i-EMSIM active. Then, `GDB MAIN` module calls the function `sim_open` to establish the connection with i-EMSIM. The function pointers of structure `Target_ops` (defined in target.h) are implemented to point to the i-EMSIM’s correlated functions `Sim_XXX` by one-to-one way.

B. How to obtain program energy consumption

We obtain the energy consumption of one application program based on a kind of statistics method. This implies that all programs share the same energy table in i-EMSIM.

The structure of energy table is as follow:

```c
struct sym_func {
    ...
    /* Define variable for energy statistics */
    long long total_energy;
};
```

When one program is running on i-EMSIM, i-EMSIM modifies the energy tables by calculating the energy consumption of this program. The table synchronizes to record the program’s energy consumption value when the program is running.

After i-EMSIM is loaded by the GDB, i-EMSIM suspends if the running program encounters a breakpoint that we set to show the energy consumption. GDB calls function `sim_show_power` to get and displays the current accumulated energy consumption value of program at this breakpoint after program is executed. So, we can get the energy consumption of any code block between two breakpoints by subtracting the two breakpoints’ values.

Since GDB don’t have some commands to show the energy consumption of breakpoint, the command `showenergy` is added into GDB to implement this purpose. `Showpower` calls function pointers in structure `target_ops`. The complementation of command `showenergy` is as follows:

```c
void _initialize_remote_sim_showenergy ( )
{
    /*We add showenergy command*/
    c=add_com ("showenergy", class_breakpoints,
               "Get total instructions total   energy and total cycle.");
}
```

In GDB, we modify the pointers of structure `target_ops`’s function, which points to `sim_show_energy`. The modification is as follow:

```c
struct target_ops
{
    ...
    /*We add function pointer that point to sim_show_power*/
    void (*to_sim_show_power)PARAMS ((int));
}
```

In i-EMSIM, we add function `sim_show_energy` to show the energy consumption of breakpoint. Some code is shown as follow:

```c
void sim_show_energy ( sd, verbose)
{
    /*Show total energy consumption*/
```
fprintf ( stdout, " total energy = %fn", I2ENERGY ( state->t_energy ) );
}

IV. SIMULATION EXPERIMENTS

A. The experimental environment and methods

In order to verify that i-EMSIM is more accurate and effective than EMSIM in obtaining C source-level program energy consumption, we adopted the same test programs running on both EMSIM and i-EMSIM. These test programs include sum from 1 to 100 (code NO. 1), simple math calculation (code No. 2), find the big number of two num (code No. 3), ASCII shifter encryption (code No. 4), transpose of 3 × 3 matrix (code No. 5). At the same time, we used the GDB to dynamically calculate energy consumption of program block by setting breakpoints.

The experimental environment use Intel Pentium 4 1.8GHz CPU, 512MB RAM, 60GB Hard disk, Redhat Linux 9.0, arm-linux-gcc, arm-linux-GDB, EMSIM and i-EMSIM.

B. The results and analysis

<table>
<thead>
<tr>
<th>Code No.</th>
<th>EMSIM (nJ)</th>
<th>i-EMSIM (nJ)</th>
<th>Increased Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>696554.053398</td>
<td>1348900.558252</td>
<td>93.65%</td>
</tr>
<tr>
<td>2</td>
<td>692858.592233</td>
<td>1343244.320388</td>
<td>93.87%</td>
</tr>
<tr>
<td>3</td>
<td>693739.514563</td>
<td>1355218.834951</td>
<td>95.35%</td>
</tr>
<tr>
<td>4</td>
<td>708419.563107</td>
<td>1392052.281553</td>
<td>96.50%</td>
</tr>
<tr>
<td>5</td>
<td>767048.883495</td>
<td>1565948.300971</td>
<td>104.15%</td>
</tr>
</tbody>
</table>

Table II. Energy consumption obtained by EMSIM and i-EMSIM

Fig. 4 shows that the energy consumption of test programs running on i-EMSIM is bigger than that on EMSIM about 90 percent. The reason why the difference is so big is that i-EMSIM have accumulated energy consumption of instruction fetching and decoding phases, while EMSIM ignored them. To fetch one instruction, processor needs to access memory. This consumes much energy that should not be ignored. This demonstrates that somehow the energy consumption of program obtained by i-EMSIM is more accurate than by EMSIM. Meanwhile, we use the GDB to dynamically calculate C source-level energy consumption between breakpoints. Compared with EMSIM which can only obtain the energy consumption of function block after the program ends to run, this method has two advantages, one is that it can acquire some energy consumption data while the program is running, another is that it allows us to deeply investigate the energy consumption of a single program statement or program block. So, it is much more convenient for us to research the energy consumption of program by using i-EMSIM in the future.

V. CONCLUSIONS

In this paper, we first propose an improved C source-level code program energy model by considering the energy consumption of instruction fetching phase, in which the accurate energy algorithm is used. Then, we implement a method to dynamically calculate the run-time energy consumption between breakpoints through GDB. The experiment result shows the energy consumption of programs obtained by the improved simulator i-EMSIM is much more accurate than by EMSIM. Such work paves the way for our future works in optimize energy consumption on embedded systems software, and how to improve the structure and code of program to consume less energy on the condition that function not affected.

REFERENCES
