A Coloured Petri Net Based Approach for Estimating Execution Time and Energy Consumption in Embedded Systems

Gustavo Callou, Paulo Maciel, Ermeson Carneiro, Bruno Nogueira, Eduardo Tavares
Informatics Center(CIn), Federal University of Pernambuco(UFPE) – Recife, PE, Brazil
grac, prmm, ecda, bcsn, eagt@cin.ufpe.br

ABSTRACT

This paper presents a Coloured Petri net based approach for estimating execution time and energy consumption in embedded systems. The aim of this work is to provide, in the design phase, a mechanism that helps the designer informing the energy consumption and the performance of the code in analysis. Experimental results have demonstrated an accuracy of 96% using the proposed formal method in comparison with the values obtained with the hardware platform.

Categories and Subject Descriptors
D.1.2 [Programming Techniques]: Automatic Programming; D.2.2 [Design Tools and Techniques]: Petri Nets; D.2.4 [Software Engineering]: Software/Program Verification - Formal Methods; D.4.8 [Performance]: Measurements, Simulation, Stochastic analysis, Modeling and prediction

General Terms
Design, Measurement, Performance and Verification

Keywords

1. INTRODUCTION

Nowadays embedded systems are present in so many areas of our lives. Sometimes, these systems are used without being noticed, for instance, ATMs, watches, refrigerators, microwaves, scopes, router, camcorders, mobile phones, palms, mp3 players and so on. In fact, embedded systems are characterized by the adoption of a dedicated processor which executes a specific software application. Most embedded devices are battery-operated, such that the overall system energy consumption is a great concern during system design. Since most of the functionalities are implemented as software components, a previous estimation of the software energy consumption can provide important insights to the designer about the battery lifetime as well as parts of the software application that need optimization [2]. In addition, predictability for embedded software execution is an important concern when considering time-critical systems with energy constraints.

Without loss of generality, there are two basic approaches for estimating embedded software energy consumption: (i) simulation based on instruction and (ii) simulation based on hardware [10]. In hardware simulation, despite the very high computation effort, better results are obtained in comparison with instruction simulation. Thus, instruction simulation has been adopted by the majority of works in order to provide energy consumption estimation in a satisfactory period of time. Although there are some works about these methods, to the best of our knowledge, only a small number adopts formal models as basis for simulation.

This work presents a formal approach, based on Coloured Petri Nets (CPN), for estimating execution time as well as energy consumption of embedded software using the instruction simulation approach. The formal mechanism, such as CPN, has being adopted in order to get a good accuracy of these estimates in comparison with the values obtained with the hardware platform. Besides, with this formal approach, it is also possible to have different level of abstraction, where a code may be modeled instruction-by-instruction or by blocks of instructions. In addition, CPN also provides a set of well-established methods for structural and behavioral property analysis and verification.

This paper is organized as follows: Section 2 presents related works. Section 3 introduces Coloured Petri Net. Section 4 presents the target platform models. Section 5 shows the proposed framework. Section 6 describes the simulation process. Section 7 presents some experiments and results to validate the model. Section 8 concludes the paper.

2. RELATED WORKS

Over the last years, many approaches have been developed to deal with the estimation of software execution time and energy consumption in embedded systems. In [14], Tiware et al. have developed an instruction level simulation mechanism that quantifies the energy cost of individual instruction. The main limitation of this approach is that it will not work for programs with larger execution times since the ammeter may not show a stable reading.

Nogueira et al [6] have developed an approach for power-aware code exploration, through an analysis mechanism based...
on Coloured Petri Net (CPN). In that approach, a methodology for stochastic modeling of 8051 based microcontroller instruction set is demonstrated. The main drawback of the method is the models complexity, and as direct consequence simulation time for evaluation. This restriction does not allow evaluation of real life complex applications or even reasonable size programs.

Another approach related to energy consumption estimation is based on functional decomposition [7]. This work has been extended [12], so that a tool to estimate the power and energy consumption related to C programs and assembly code was proposed. This work does provide means for structural and behavioral property analysis and verification.

Nikolaidis et al [11] presented a method for measuring power consumption of embedded applications on an ARM7-based microcontroller. This method, takes into account inter-instruction costs and pipeline stalls in the energy and execution time estimation process.

3. COLOURED PETRI NETS

Petri Nets (PNs) [9] are a graphic and mathematical modeling tool that can be applied in several types of systems and allow the modeling of parallel, concurrent, asynchronous and non-deterministic systems. Since its seminal work, many representations and extensions have been proposed for allowing more concise descriptions and for representing systems feature not observed on the early models.

Among this models propositions, it is important to stress Jensen’s Petri nets high-level model, the so called Coloured Petri net (CPN) [5]. In this model a token may have complex data type as in programming languages; each place has the correspondent data type, hence restricting the kind of tokens that it may receive; the transitions process the tokens values and create new ones with different values; hierarchy structure can be modeled with different abstraction levels, where each transition may describe another net (called subnet), and so on. In a nutshell, CPN is a high-level model that considers abstract data-types and hierarchy. The formal definition is presented below.

The non-hierarchical definition of Coloured Petri Net is a nine-tuple $CPN = (\sum, P, T, A, N, C, G, E, I)$ where $\sum$ is a finite set of non-empty types, called colour sets; $P$ is a finite set of elements, named Places, that represents local states; $T$ is a finite set of elements, called transitions, that depicts events and actions; $A$ is a finite set of arcs such that $P \cap T = P \cap A = T \cap A = \emptyset : N : A \rightarrow P \times T \rightarrow T \times P$ is a node function; $C : P \rightarrow \sum$ is a colour function; $G$ is a guard function, which is defined from $T$ into expressions such that $\forall t \in T : Tp(G(t)) = Bool \land Tp(Var(G(t))) \subseteq \sum$; $E$ is an arc function that is defined from $A$ into expressions, such that $\forall a \in A : Tp(E(a)) = C(p(a))_{MS} \land Tp(Var(E(a))) \subseteq \sum$, where $p(a)$ is the place of $N(a)$ and $C_{MS}$ denotes the set of all multi-sets over $C$; $I$ is an initialization function defined from $P$ into expressions such that $\forall p \in P : Tp(I(p)) = C(p)_{MS} \land Var(I(p)) = \emptyset$, where $Tp(expr)$ denotes the type of an expression, $Var(expr)$ denotes the set of variables in an expression, and $C(p)_{MS}$ denotes a multi-set over $C(p)$. Places are graphically represented by ellipses, transition by rectangles, and arcs by direct arrows.

In addition, CPNs allow to model hierarchical structures where transitions, called substitution transitions, can represent complex models [4]. The model that is represented by the substitution transitions is named subpage. The higher model, which have substitutions transitions, is the page. These pages are connected to each other by places called ports, which can be input or output types.

CPN reduction process [9] has been adopted in order to transform a CPN model into an equivalent simplified model, in which all important characteristic for estimating the energy consumption and execution time is preserved.

4. EMBEDDED SOFTWARE MODELING

The embedded software has been represented by composition of basic CPN models that depict processor’s instructions. Each instruction model computes the respective energy consumption and execution time related to instructions or set of instructions during the model evaluation.

This works focuses on the Philips LPC2106, which is an ARM7-based microcontroller. Five basic CPN instruction models have been conceived to characterize the such processor instruction set. The proposed basic models are the ordinary instruction model, conditional model, branch and link model, conditional branch model, and a return of branch instruction. These individual nets are represented in a XML format compatible with CPN Tools [1]. These models are presented as follows.

Figure 1 depicts the ordinary instruction model. This model represents the energy consumption ($val\ energy$) and execution time ($val\ cy$) by the function $addData(energy, cy)$. This model has just two ports, one is input type and the other is output type.

![Figure 1: Ordinary model](image)

ARM7 instructions have a specific characteristic, in which all instructions may be conditional or not. In the conditional context, has been adopted random numbers that can be generated directly on computers and have a uniform distribution over the open interval (0,1). Figure 2 presents the conditional model related to ordinary instructions. The principal difference, from the last model, are variables, such as $prob$ and $aval$. The $prob$ represents the probability of conditional values to be accepted. $Aval$ is responsible for evaluating if the respective condition is satisfied or not. The respective evaluation is performed by checking if a random number between zero and one is smaller then $prob$ value; if so, $aval$ receives “1” (true) meaning that the condition is satisfied. Besides, the values of energy consumption and execution time are computed as in the ordinary instruction model. Otherwise, $aval$ receives “0” (false) meaning that the condition is not satisfied. The energy consumption and execution time are computed with values for the respective non-conditional instruction.

Figure 3 shows the model for branch and link instruction type. This model, besides using the function $addData$
Figure 2: Conditional model

(already depicted), considers the function push. The function push stores (pushes) the current address position into a stack, which is adopted by return of branch and link model to provide the desire branch.

Figure 3: Branch and link model

Figure 4 shows the model for conditional branch instructions, where the branch just happens if the condition is satisfied. It is important to know that the conditional statement is represented in a probabilistic manner, and there are some differences to the other models already explained. The principals differences are transitions, such as Jump and NotJump, with guard expressions that defines which transition should be fired. In this model, the Jump transition could fire if the value of jump variable is equals to “1” (one). The other transition is only “fireable” if the jump value of the token is “0” (zero). These transitions are linked to two different output ports that represent distinct program’s execution flows.

Figure 4: Conditional branch model

It is important also to remark that prob represents the probability of making a branch, and aval is considered for evaluating the possibility of branching. The branching evaluation is performed by generating a random number and checking if it is smaller then prob. If it is the case, aval receive “1” (one) as well as the value of the variable on token. In this case, thus, the Jump transition is fired for representing the branching. Otherwise, the transition to be fired is the NotJump, since the execution flow does not branch.

In the return of branch and link instruction model, the number of transitions, places, arcs and output ports are not always the same, in other words, it depends on the application code that is analyzed. For instance, if a procedure is called more than once, the return instruction model is linked to different places.

Figure 5 presents a net with three ports, in which one is an input type port and all the others are output type ports. This structure is adopted, since this model is a subnet of a hierarchical net. More specifically, the transition in the main net, which this subnet represents, is linked to two different places. It is important to note that only one of these transition (Jump or NotJump) is able to “fire” at a time, that is, considering that the input port stores at most one token, the firing of one these transitions disable the other. Guard expressions are assigned to transitions Jump and NotJump, so that Jump transition performs a branch to the address “276” and, NotJump to “280”. The desire return address, that is presented in the guard expressions, is provided by the startPos stack.

Figure 5: Return of branch and branch model

This model has also being adopted for the last instruction of the code in analyze. In the net simulation, the last instruction in the code is represented by a model in which that stack is empty, so that pos receive 0 and the stop criteria evaluation is performed (see Section 6).

5. FRAMEWORK

This section describes the proposed framework for estimating the execution time and energy consumption of embedded systems. The framework is illustrated on Figure 6. This framework takes into account assembly code labeled with probability data assigned to conditional instructions, and also parameters for the stop criteria. The assembly code is provided as input to the assembler that generates two outputs: the Binary Code (machine code) and the LST-File (file where the probabilities and the stop criteria parameter are depicted). After that, the Binary-CPN Compiler reads these two files and the CPN Instruction Models as input, and generates two CPNs Models to be analyzed. The first model is namely as CPN Model, in which each transition represents an instruction of the code. The other model, named, CPN-Optimized, represents instructions and
blocks of instructions by one transition. Transitions representing blocks of instructions describe energy consumption and execution time by summing up all related values of each instruction that belongs to the respective instruction block. Finally, these two nets might be read by CPN Tools and/or by CPN Simulator, so as to generate the evaluation Files.

**Figure 6: The proposed framework**

The CPN Simulator is a tool that simulates the proposed CPN models in order to compute the respective energy consumption and execution time. This tool has been conceived as an alternative to CPN Tools, since its respective simulation mechanism is quite consuming when analyzing large models. In a nutshell, the CPN Tools is adopted to validate the generated nets by the Binary-CPN Compiler and the CPN Simulator is adopted for performance and energy consumption simulation.

In addition, an automatic CPN Generator is purpose to receive the processor characterization tables as input in order to create the Instruction-CPN Models for the ARM7 processors. These characterization tables consist on energy consumption as well as execution time values for each ARM7 instruction. The execution times and the energy consumption values might be obtained from datasheets, low-level simulation mechanisms, measuring etc.

6. SIMULATION PROCESS

Figure 7 depicts the simulation process adopted by CPN Simulator [8]. The simulation starts on the main program which invokes the initialization routine. The initialization routine sets the simulation clock to “0” (variable giving the current value of simulated time), initialize counters (variables used for storing statistical information about system performance and energy consumption), and initialize event list (list that contains the transitions time for each transition able to fire). After that, the main program invokes the timing routine. The timing routine determines the next event type (the transition that is fired) and advance the simulation clock. Next, the main program invokes the event routine. The event routine updates system state, updates statistical counters, generates future events and add it to event list. After that, it is checked if the simulation should be finished or not, according to stop criteria evaluation. After finishing the simulation, a simulation report is generated.

The stop criteria evaluation is adopted in order to provide simulation results, taking into account specified confidence degree. In order to obtain estimates considering degree of confidence, the simulation process have to execute several simulation runs. The number of simulation runs depends on (among other factors) the specified confidence degree. The initial number of replication runs adopted is specified by the analyzer (the default value adopt in the proposed simulation engine is ten) [3].

Stop Criteria considers: Absolute Precision for energy consumption and execution time (the designer informs the desired precision), the respective means and standard deviations. The Absolute Precision, calculated by Equation 1, in which the t critical value is calculated for 1 − α/2 confidence degree and n − 1 degrees of freedom; s is the standard deviation of replication and n is the number of replications in the example.

\[
\text{AbsolutePrecision} = t_{1-\alpha/2,n-1} \times \frac{s}{\sqrt{n}}
\]

(1)

Afterwards, the desired precision related to both energy consumption and execution time are compared with the current results. The simulation is finished if these calculated values are smaller than the desired precision specified. Otherwise, the simulation continues by calculating required number of new simulation runs (replications) by Equation 2. There are two numbers of replications (one for energy consumption and other for execution time), the largest value is adopted.

\[
i = \left[ \frac{t_{1-\alpha/2,n-1} \times s}{\text{AbsolutePrecision}} \right]^2
\]

(2)

7. EXPERIMENTS AND RESULTS

Some experiments have been adopted in order to evaluate the proposed framework, and also, to demonstrate the importance of the CPN reduction process as well as to justify the creation of a specific simulation tool, namely, CPN Simulator. All experiments were performed on a Pentium IV HT 3GHz, 1.5Gb RAM, and OS WinXP.

7.1 Example One

This example has been conducted in order to exemplify the proposed method. Figure 8 depicts small application code, where the values for the stop criteria evaluation are depicted on the code line. Note that the confidence degree is set to 95%, the specified precision for energy consumption
Table 1: Simulation Results of the code on figure 8.

<table>
<thead>
<tr>
<th>N Inst.</th>
<th>CPN Model</th>
<th>CPN Tools</th>
<th>CPN Simulator [CPN Tools/CPN Simulator]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0,69 s</td>
<td>187 ms</td>
<td>1,1</td>
</tr>
<tr>
<td>20</td>
<td>0,333 s</td>
<td>219 ms</td>
<td>1,04</td>
</tr>
<tr>
<td>30</td>
<td>0,46 s</td>
<td>234 ms</td>
<td>0,78</td>
</tr>
<tr>
<td>40</td>
<td>0,78 s</td>
<td>281 ms</td>
<td>0,69</td>
</tr>
<tr>
<td>50</td>
<td>0,9 s</td>
<td>297 ms</td>
<td>0,53</td>
</tr>
<tr>
<td>100</td>
<td>1,1 min</td>
<td>515 ms</td>
<td>0,46</td>
</tr>
<tr>
<td>200</td>
<td>5 min 6 s</td>
<td>219 ms</td>
<td>0,4</td>
</tr>
<tr>
<td>400</td>
<td>17 min 25 s</td>
<td>438 ms</td>
<td>0,333</td>
</tr>
</tbody>
</table>

Table 2: Comparative of the Simulation Results.

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Energy Consumption</th>
<th>Simulation Time</th>
<th>Time Error(Keil)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,2320 μs</td>
<td>167,4008 ηJ</td>
<td>22 s</td>
<td>2,3860 %</td>
</tr>
<tr>
<td>2,2279 μs</td>
<td>167,2632 ηJ</td>
<td>172 ms</td>
<td>2,1972 %</td>
</tr>
</tbody>
</table>

7.2 Example Two

This experiment has been adopted aiming at showing the importance of the CPN reduction process and the CPN Simulator. It consists on codes with instructions that only uses the ordinary model (see Figure 1). The codes have been performed with 10, 20, 30, 40, 50, 100, 200, 400 instructions.

Table 3 shows a comparison of the simulation time of models related these codes in which different numbers of instructions have been taken into account. It is worth stressing that the runtime simulation on CPN tools is quite time consuming when analyzing large models.

Table 3 also presents the runtime related to the CPN Simulator, which is at least 90 times shorter than the respective of CPN Tools. The column, named, CPN Tools / CPN Simulator shows the ratio between the runtime in CPN Simulator and the respective runtime in CPN Tools. It is possible to observe that the CPN Simulator has performed much more faster than CPN tools. However, it is also important to mention that the CPN Simulator performs even better for larger models. For example, when simulating a 10 instruction application, the CPN Simulator spent 1,10% of the CPN Tools time and; for 200 a instruction application, the CPN Simulator spent 1,10% of the CPN Tools time. The time spent by the CPN Tools (6 s) and CPN Simulator (187 ms) is constant when the optimized CPN Model is considered. The reason for this is that the reduction process clusters all instruction into two blocks.

Figure 8: Annotated Assembly Code

The registers values are not stored in the model. Instead of comparing values, a probabilistic approach is adopted. In this example, there is a loop (lines 16-20) that is executed 10 times, so, the probability is performed using the equation $p = 1 - (1/N)$. In this case, $p = 1 - 1/10 = 0.9$ (probability of the conditional instruction `bl loop`). The probability is adopted to set the variable prob (see Figure 2 as an example) in the CPN model built.

7.1.1 Simulation Results

The code depicted in Figure 8 has been simulated in three different ways: (i) using CPN Tools with the CPN model, (ii) using CPN Tools with the optimized CPN Model, (iii) and adopting the proposed CPN Simulator with the optimized CPN Model. The simulations using CPN Tools considering both CPN model and optimized model are identical. The proposed CPN Simulator considered only the optimized.

Table 1 presents the simulation results. The time standard deviation obtained when considered the CPN Tools was 0,19μs. The confidence degree adopted was 95% (see header annotation on Figure 8), so that the execution time value (2,23μs) should be within [2,042μs; 2,422μs]. These results have been compared to Keil UVision IDE, in which 2,18μs for the execution time has been measured (see those provided through in Table 2).

Table 3: Comparative of the simulation time.

<table>
<thead>
<tr>
<th>N Inst.</th>
<th>CPN Tools</th>
<th>CPN Simulator</th>
<th>Time Error(Keil)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>187 ms</td>
<td>1,1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>219 ms</td>
<td>1,04</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>234 ms</td>
<td>0,78</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>281 ms</td>
<td>0,69</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>297 ms</td>
<td>0,53</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>515 ms</td>
<td>0,46</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>219 ms</td>
<td>0,4</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>438 ms</td>
<td>0,333</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Comparative of the Simulation Results.

<table>
<thead>
<tr>
<th>CPN Tools</th>
<th>CPN Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Time: 2,2320 μs</td>
<td>Mean Time: 2,2279 μs</td>
</tr>
<tr>
<td>Time SD: 0,1905 μs</td>
<td>Time SD: 0,1444 μs</td>
</tr>
<tr>
<td>Time Error: 0,1363 μs</td>
<td>Time Error: 0,1033 μs</td>
</tr>
<tr>
<td>Mean Energy: 167,4008 ηJ</td>
<td>Mean Energy: 167,2632 ηJ</td>
</tr>
<tr>
<td>Energy SD: 14,1533 ηJ</td>
<td>Energy SD: 10,7243 ηJ</td>
</tr>
</tbody>
</table>

non-optimized models), and the proposed CPN simulator. In this case, the simulation results provided by the CPN Simulator are similar to those obtained by the CPN Tools, since the differences are smaller than 2%. The reader should observe the error (about 2%) with relation to execution time when comparing with Keil UVision (2,18μs).
many other functionalities than the CPN Simulator does. CPN Tools is a general purpose environment, and provides sake of fairness, the reader should also bear in mind that energy errors are lesser than 1% in both environments. For Figure 9: Comparative of simulation time between CPN Tools and CPN Simulator.

7.3 BCNT Algorithm

The BCNT Algorithm was proposed by Motorola as an integrated part of Power Stone Benchmark. The BCNT adopts a series of operations between two arrays, explores the memory space manipulation, and also adopts bitwise operations.

<table>
<thead>
<tr>
<th>Table 4: BCNT Simulation Results.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>CPN Tools</td>
</tr>
<tr>
<td>Time(μs)</td>
</tr>
<tr>
<td>Energy(J)</td>
</tr>
<tr>
<td>Runtime</td>
</tr>
<tr>
<td>T. Error(Keil)</td>
</tr>
<tr>
<td>T. Error(Hard)</td>
</tr>
<tr>
<td>E. Error(Hard)</td>
</tr>
</tbody>
</table>

Table 4 depicts a comparative study between the estimated values and measurements conducted on Keil and on hardware platform according to the methodology described in [13]. The execution time measured on keil was 34.73μs, the execution time measured on hardware was 35.86 μs and the energy consumption was 226953 J. The time and energy errors are lesser than 1% in both environments. For sake of fairness, the reader should also bear in mind that CPN Tools is a general purpose environment, and provides many other functionalities than the CPN Simulator does.

8. CONCLUSIONS

This paper presented an approach based on Coloured Petri nets for estimating embedded software execution time and energy consumption. The proposed method provides a set of integrated tools that allow the automatic translation of a binary code into a CPN Model, such that non-specialized users do not need to interact directly with the Petri net formalism. The aim of this work is to provide, in the design phase, a mechanism that helps the designer informing the energy consumption and the performance of the code in analysis. The presented case studies clearly show that the proposed methodology and the framework have provided meaningful results with small errors. It is also important to highlight that the simulation provides good results about these estimation with smaller computational effort than measurements on the hardware platform. Besides, the CPN Simulator performed much faster simulation results than the CPN Tools environment. CPN Simulator was, in some cases, 304 times faster than CPN Tools.

As a future work and on going works, we may stress that we are extending the proposed approach in order to consider not only assembly code, but also C code applications. Another improvement is related to add a functionality that performs blocks analyzes of codes, instead of only analyzing the entire program.

9. REFERENCES