Software Evaluation for Software-Hardware Migration: An Approach Focusing Energy Savings and Based on a Coloured Petri Net Model

ABSTRACT

In this paper a framework for software-hardware migration evaluation is proposed. This framework is conceived to be applied to embedded system design in which energy consumption is the main concern. The proposed framework allows designers to identify consumption distribution along the executable code so that partitioning such centers in hardware, in order to promote energy savings. The processor behavior is modeled in Coloured Petri Net (CPN) so that to use CPN properties to capture behavioral parameters. This approach allows to analyze software-power metrics using widespread analysis approaches presents in the CPN universe. Such approaches are very consolidated on well-known CPN tools, as CPNTools and Design/CPN. Due to proposed architecture‘s modeling, general-purpose CPN tools can be used as a retargetable evaluation environment. Additionally, this framework was integrated into an environment in order to implement some analysis functions and for optimizing the designer‘s interface. The framework also allows architecture flexibility based on an instruction-CPN models library. An important contribution included in this framework is the proposition of a method for software-power cost centers identification, based on a formal token based computing model.

1. INTRODUCTION

An embedded system may be defined as a digital system embedded in a bigger system, being such embedded system responsible for very specifics, and normally very crucial, tasks. In more general case embedded systems is composed by a dedicated processor surrounded by specific hardware devices. In order to improve designs, methods for design space exploration play an important role, searching for optimizations of parameters relatives to performance and energy consumption. In power critical embedded systems, energy consumption may be regarded to the processor and to specific hardware device. Despite of processor hardware optimization, processor consumption is affected by the software dynamic behavior [14], meaning that the software-power analysis is crucial. Many embedded computing applications are power critical, such as: portable medical instruments, notebook computers, personal digital assistant and cellular phones. Hence, power constraints are an important part of design specifications. Additionally, the embedded software is basically not modified during the system life. Even though embedded systems normally allows *firmware* updates, such modification does not change radically the software structure. Based on these facts, works as [13] and [12] have shown that power-aware software-hardware migration can promote up to 76% of energy savings. This paper proposes a framework for binary-code evaluation in order to software-hardware migration. The proposed framework allows designers to identify consumption distribution along the executable code so that partitioning such centers in hardware, for promoting energy savings [12]. Additionally, an environment to deal with such framework is presented. In previous work [1, 2], we proposed a software cost analysis framework based on Coloured Petri Net (CPN) and a specific taxonomy. Moreover, analysis entities were presented in order to help the identification of code structures and its energy cost. This paper extend previous work in order to apply such concept for providing practical resources for the designer. This paper is organized as follows: Section 2 introduces Coloured Petri Nets concepts, Section 3 presents the power analysis model, Section 4 shows the proposed framework, Section 5 presents a case study and Section 6 concludes the paper.

2. COLOURED PETRI NET: AN OVERVIEW

Petri nets are families of formal net-based modeling techniques, that model actions and states of systems using four basics entities: places, transitions, arcs and tokens. In the majority of models, places are associated with local states and transitions with actions. Arcs represent the dependency among actions and states. An action occurs when a transition is “fired”, moving tokens from incoming places to outgoing places. A parallel process is described in Figure 1. Arcs describe which action (or actions) is possible from a given local state. In this figure, arc A1 links place P1 (state 1) to transition T1 (action 1), representing that action T1 requires local state P1. Arcs A2 and A3 connect transition T1 to places P2 and P3. It is easy to see two parallel paths (P3-T3-P5 and P2-T2-P4) as a representation of parallel processes. In order to accomplish the action T4 is necessary to reach state in which places P4 and P5 are marked. In order to represent possible local states, it is used a mark, called token. This simple net is known as place-transition net [11]. Place-transition nets are adequate to analyze some characteristics of system such as: repetitiveness, liveness and reachability, that means determinating whether a specific state is reachable. There are various extended Petri net models, each one dealing with a specific modeling problem and distinct abstraction level. CPN is a high-level model that consider abstract data-types and hierarchy. CPN tools provide an environment for design, specification, validation and verification of systems [8]. Informally, Coloured Petri Net is a Petri net with some modeling improvements: (i) tokens express values and data structures with Types (colors); (ii)places have associated Type (color set) determining the kind of data (token) those places may contain; (iii) transitions may express complex behavior by changing token value; (iv) Hierarchy can be handled at different abstraction levels . Transitions in a hierarchical net may represent more complex structures, where each transition (substitution transition) expresses another more complex net, and so on. A hierarchical net may describe complex systems by representing their behavior using a compact and expressive net. (v) Behaviors can be also described using high level program language.

In this way, it is possible to model a complex system using tokens for carrying sets of internal data values. Transitions represent actions that modify internal set of data values. The entire net represents the flow of changes into the system during its states evolution. The model can be analyzed by simulation, states and invariant analysis. State analysis means to study all possible system states, or an important subset, in order to capture system patterns. There are some widespread academic tools such as Design/CPN and CPN-Tools, for handling with Colored Petri Net and its analysis techniques.

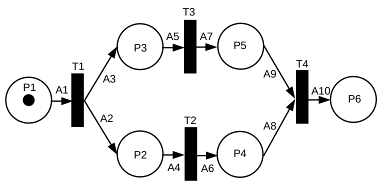


Figure 1: Petri net example

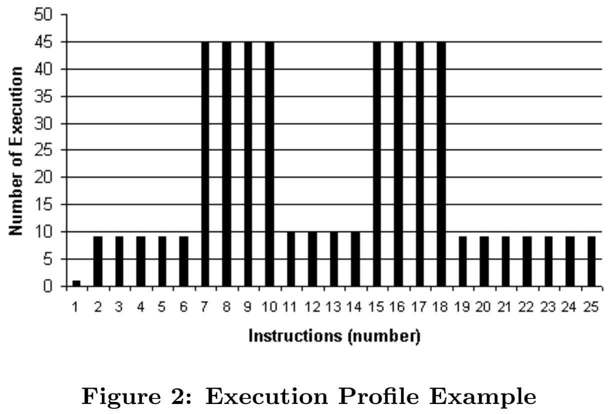


Figure 2: Execution Profile Example

3. SOFTWARE POWER COST MODEL

In previous work [2] we proposed a new approach based on Coloured Petri Net for performing software power cost analysis. This framework explore Coloured Petri Nets as formal description language. The simulation mechanism is used to capture behavioral patterns that characterize energy consumption. The atomic behavior associated to consumption is modeled as processor instruction and, based on it, a power model is yielded. During the program execution flow, each instruction changes the internal processor context so that processor reaches new states. In this approach, the program flow is modeled as a Coloured Petri Net where each possible processor state is modeled as a place, the internal context as a data structure within a token[1], and instructions as transitions that processes this token. Figure 3 shows a JNC instruction model for 8051 architecture (MCS51), the instruction behavior is defined by an associated CPN-ML code (a Standard ML Language subset) [8] that invokes CPN-ML functions defined in a shared area. Such function can be accessed by every instructions of the code-model and represents processor-hardware resources. For example, the *write_m()* and *read_m()* implements memory accesses. In contrast to previous CPN processor architecture models [5][4], where the processor is modeled based on architectural/RTL hardware model, i.e. based on its internal hardware resources, (pipeline stages, AU, Icache, and Dcache, for example), this approach models processor mainly by its instruction set. Models based on architectural/RTL hardware have to be constructed based on very detailed hardware consumption data related with implementation technologies, that is not often available to the system designer [10]. On other hand, instruction-level power model can be implemented only based on physical measures [9]. The proposed CPN modeling explicitly represents control and data dependencies, hence allowing mapping consumption features to software structures. Information about the internal behavior pattern is capture by *probes variables* strategically embedded on function and instructions description.

3.1 Nomenclatures and Definitions

In [1] was presented a taxonomy and some definitions that are necessary for better understanding the proposed method. These definition are informally showed in the followings.

DEFINITION 3.1 (EXECUTION VECTOR). *: Execution Vector is a vector,enumerated by the instructions memory ordering, where each component represents the number of instruction execution.*

DEFINITION 3.2 (CONSUMPTION VECTOR). *: Consumption Vector is a vector, enumerated by the instruction memory ordering , where each component represents the respective instruction energy cost (base cost + inter-instruction cost)[2].*

Figure 2 depicts the Execution Profile for a nested-loop example. Analyzing the Execution Profile, it is possible to identify execution patterns such as:

[1]In fact the model has one token in pipeline-less architectures. For SimpleScalar architectures, for example, the model will deal with two tokens. SimpleScalar architecture models will be presented in futures papers.

[2]Base cost is the instruction specific energy cost. Inter-instructions cost means the *circuit overhead* that appears when two instruction are executed consecutively[9]

Definition 3.3 (Patch). *:*
Patch is a set of instructions that are located in consecutive addresses and are executed the same number of times.

In Figure 2, five patches are identified : from instruction 2 to instruction 6 (patch 1), from instruction 7 to instruction 9 (patch 2), from instruction 10 to instruction 14 (patch 3), from instruction 15 to instruction 18 (patch 4) and from instruction 19 to instruction 25(patch 5).

Definition 3.4 (Loop-Patch). *:*
Loop-Patch represents a single loop. It consists of a patch in which the incoming (first) and outgoing (last) instruction are executed only once.

There is no Loop-Patch in Figure 2.

Definition 3.5 (Cluster). *:*
Cluster is a set of Patches joined together (aggregated) in consecutive addresses, in which the incoming (first) and outgoing (last) instruction are executed only once.

In Figure 2 there is only one cluster: {patch1, patch2, patch3, patch4, patch5}.

Definition 3.6 (Bound-Patch Set). *:*
Bound-Patch Set is a set of Patches executed the same number of times and belong to the same Cluster.

In Figure 2, there are two Bound-Patch Set: {patch1, patch5}, {patch2, patch4}.

Definition 3.7 (Free-Patch). *:*
Free-Patch is a Patch present in a Cluster but not within the Cluster Bound-Patch Set.

In Figure 2, there is only one Free-Patch: patch3.

The Execution Profile can also define metrics such as:

1. Instruction Consumption
$$I_i = (N_i \times B_i)$$
where I_i is the total consumption due to instruction i, N_i is its number of executions and B_i is its energy cost (base cost + inter-instruction cost).

2. Patch Consumption
$$Pc_j = \sum_i (N_j \times B_i) = N_j \times \sum_i (B_i)$$
where Pc_j is the consumption of Patch j, N_j is the Patch j execution number, and B_i the instruction i energy cost.

3. Cluster Consumption
$$Cc_k = \sum_j \sum_i (N_j \times B_i) = \sum_j (Pc_j)$$

4. Consumption Profile Vector
$$Cp_m = (Ev_m \bullet Cv_m)$$
where Ev_m is program m Execution Vector and Cv_m is its Consumption Vector.

From this moment on, *Execution Profile* and *Consumption Profile* denote graphics representation of Execution and Consumption Profile Vectors, respectively.

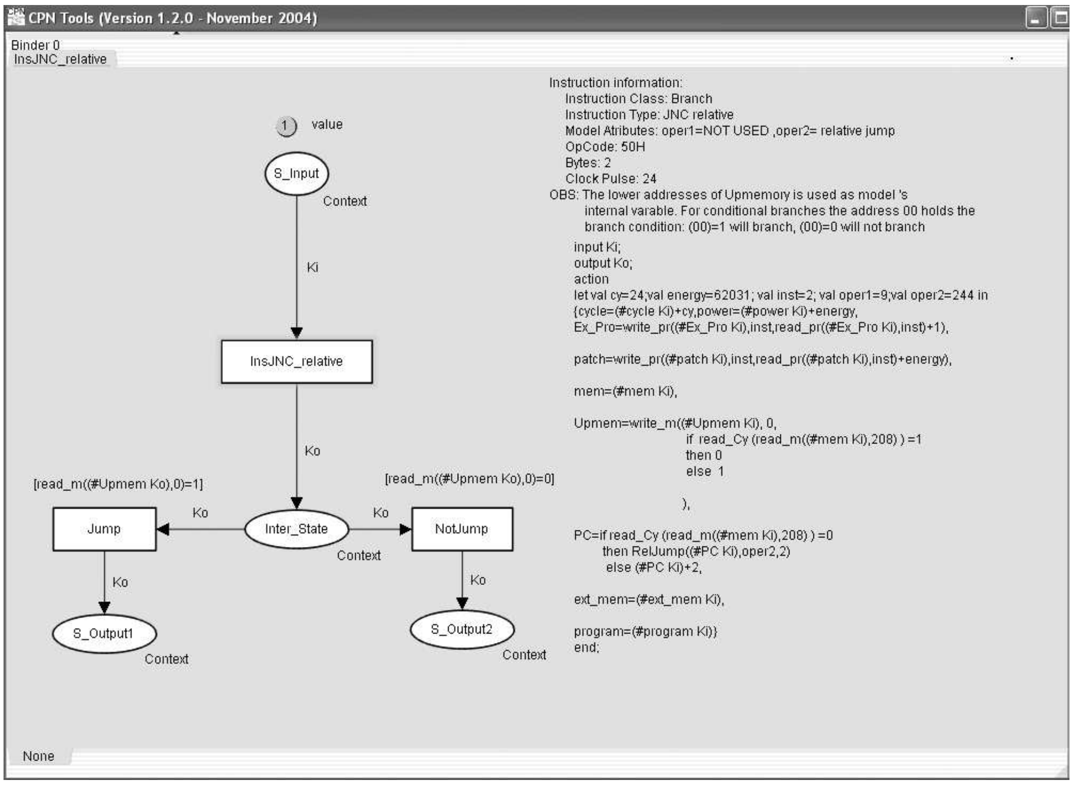
Such definitions and metrics help the designer to figure out code structures and their energy consumption. For example, a Loop-Patch represents an isolated loop within the code. A Cluster represents consumption and time cost regions. A *Cluster* with *Bound-Patch* Sets such that its *Patches* have symmetric positions in the *Execution Profile* may represent a nested-loop (see Figure 2). Inspecting *Execution Profiles* and the *Consumption Profile* graphics, the designer is able to map consumption to code structures. In fact, entities as *Patches*, *Clusters* and *Bound-Patch Sets* are clues to figure out the code flow characteristic. In the scope of embedded systems is very important to analyze optimization with respect to total consumption and consumption profile. Under the consumption profile point of view, the designer may opt for some consumption distribution allowing best software-hardware migration from code-segments (*Patches/Clusters*), improving total consumption, as postulated in [12].

4. PROPOSED FRAMEWORK

Analyzing software in order to identify its better sector for partitioning on hardware implies capture the Highest Consumption Patch (HCP) and Cluster (HCC) on binary-level code. In order to guarantee the best possibilities exploration, such binary-codes[3] should comes from different implementation of the target algorithm described in high-level language, typically C. Different implementation here means binary-code generated by different compilers or/and different compiler-optimization option. Different implementation does not mean a different high-level description of the same algorithm or a different algorithms for the same functionality, even though such options enclose interesting possibilities. Thus, the framework proposed performs a binary-code space exploration in order to capture the best one, taking into account the system energy constraints. The basic framework, illustrated in Figure 4, consists of a CPN engine, a Binary-CPN compiler that translates machine-code to CPN model, and a set of specific functions for power and performance evaluation. These functions return metrics, among then, *Consumption Profile Vector*, *Patch Consumption*, *Cluster Consumption* and *Patch Consumption.* The Binary-CPN compiler and analysis functions are encapsulated on "hidden-tools"[4] Environment [3]. Figure 4 illustrates the framework to tuning the compilation process, termed as "hidden-tools"-PCAF (*"hidden-tools"- Power Cost Analysis Framework*). The source-code is compiled under each implementation option. "hidden-tools"-PCAF performs Binary-CPN compilation, opens the CPN-Model in CPN-Engine, engages communication with CPN-Engine and evaluates analysis functions. Such actions can be visualized on Figure 5. For each code version a power analysis can be performed under the "hidden-tools"-PCAF environment. Afterward, the results can be compared in order to identify the more adequate code for partitioning process. The CPNtools [7] has been used as the CPN-Engine. The front-end is performed by the "hidden-tools" environment, providing a specific interface for power analysis. Moreover, the same framework can be used to either explore C code versions of the same algorithm or different algorithms for same functionality.

4.1 Binary-CPN Compiler

[3]In this context Binary means executable machine-code.
[4]The Tools's name is hidden due to blind review



```
Instruction information:
   Instruction Class: Branch
   Instruction Type: JNC relative
   Model Atributes: oper1=NOT USED ,oper2= relative jump
   OpCode: 50H
   Bytes: 2
   Clock Pulse: 24
OBS: The lower addresses of Upmemory is used as model 's
      internal varable. For conditional branches the address 00 holds the
      branch condition: (00)=1 will branch, (00)=0 will not branch
    input Ki;
    output Ko;
    action
    let val cy=24;val energy=62031; val inst=2; val oper1=9;val oper2=244 in
    {cycle=(#cycle Ki)+cy,power=(#power Ki)+energy,
    Ex_Pro=write_pr((#Ex_Pro Ki),inst,read_pr((#Ex_Pro Ki),inst)+1),

    patch=write_pr((#patch Ki),inst,read_pr((#patch Ki),inst)+energy),

    mem=(#mem Ki),

    Upmem=write_m((#Upmem Ki), 0,
                        if  read_Cy (read_m((#mem Ki),208) ) =1
                        then 0
                        else  1

                       ),

    PC=if read_Cy (read_m((#mem Ki),208) ) =0
          then RelJump((#PC Ki),oper2,2)
           else (#PC Ki)+2,

    ext_mem=(#ext_mem Ki),

    program=(#program Ki)}
    end;
```

Figure 3: Colored Petri net model for a branch instruction

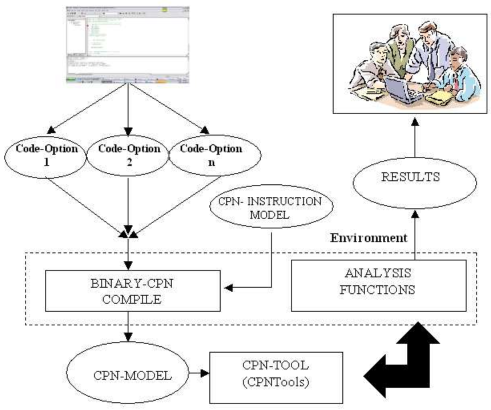


Figure 4: The Proposed Framework

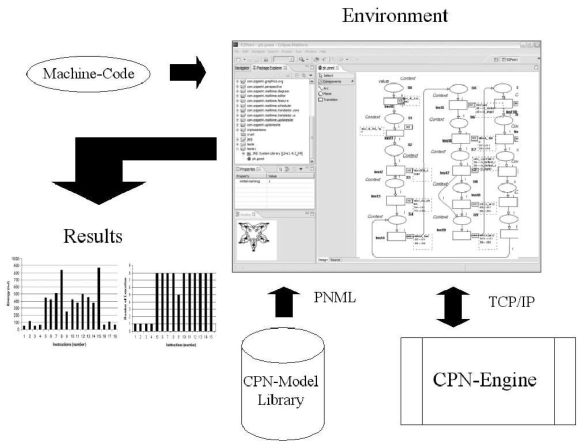


Figure 5: "hidden-tools" Power Cost Analysis Framework

The Binary-CPN compiler considers two entries files: the machine-code and the CPN-Instruction model. The CPN-Instruction model consists of sets of CPN nets representing target architecture (processor/microcontroller). Figure 3 shows a JNC (Jump if Not Carry) instruction model described using CPNTools tool. Based on these information, the Binary-CPN compiler generates a model, a Coloured Petri Net model, according to machine-code input. The model is represented in the XML file format defined by the CPNTools. In this work, the target architecture is the 8051 architecture, due to its widespread application in embedded systems. Note that, due to the standardization present on the formal model (CPN-Instruction model), the Binary-CPN compiler can deal with different architectures with minimal modifications. The net entities, as places and transitions, keep the same meaning whatever be the architecture. The changes will be concentrated on the CPN-instruction model, with the minimal impact on the net formation rules.

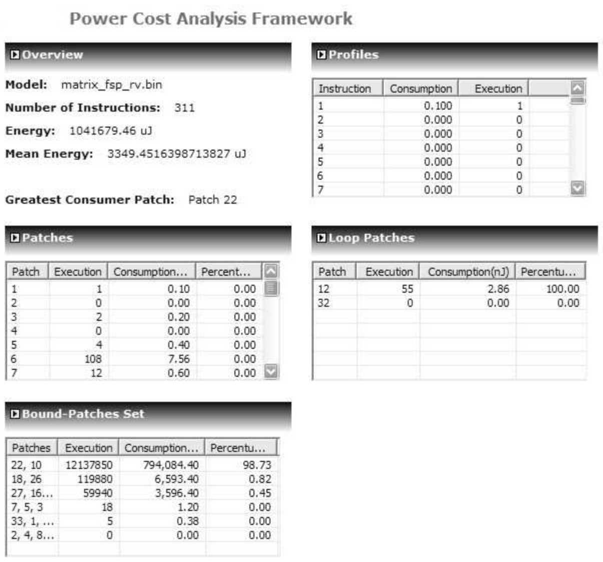
During the code(net) simulation dynamic jumps can deviate the program flow to, possibly, any point of the model. This would cause an arc to be created for every place existing in the net. When an interruption occurs the more critical situation appear, because interruption can be called from every point of the code driving the flow in the direction of an interruption routine. Such routines can return possibly to any of those points. This would result in an extremely complicated net. In order to solve this problem, a special structure was created. It acts as a dynamic branch processor in the net. Its function is for analyzing the state of the program, at the simulation time, and to lead the token by the correct arc. This structure is unique for each net. Every place in the net has an arc that goes to its entry point and an arc that comes from it. This structure reduces drastically the complexity of the net as it maintain the adequate net behavior in occurrence of interruptions and dynamic jumps.

4.2 “hidden-tools” Environment

The “hidden-tools” environment presents a new integration perspective combining existing Petri net tools and new applications into a single environment.This environment is based on the IBM-Eclipse platform, allowing the coupling of new features easily.

In order to integrate this work to “hidden-tools”, the “hidden-tools”-PCAF (*Power Cost Analysis Framework*) has been developed. It implements functions such as codes entities identification (*Patches and Clusters*) and consumption analysis. The results generated by the analysis are shown in a GUI as charts and tables. Figure 6 depicts the PCAF results. Additionally, these results may be presented as a portable document format (pdf) file.

To reach the results, several operations are executed by this environment (see Figure 5) . The “hidden-tools” begins loading the machine-code file, which one is compiled to a CPN model (see Section 4.1). After that, the model is sent to a CPN Tools proxy through a TCP/IP channel. Then, the proxy loads the model in the CPN Tools. Using the Comms/CPN[6], the model establishes a TCP/IP connection in order to interchange information with the proxy. The model is analyzed on CPNTools so that to capture information about energy consumption pattern. Finally, the proxy sends such information back to “hidden-tools”-PCAF, which performs analysis and presents the results as already discussed.


Power Cost Analysis Framework

Overview

Model: matrix_fsp_rv.bin

Number of Instructions: 311

Energy: 1041679.46 uJ

Mean Energy: 3349.4516398713827 uJ

Greatest Consumer Patch: Patch 22

Profiles

Instruction	Consumption	Execution
1	0.100	1
2	0.000	0
3	0.000	0
4	0.000	0
5	0.000	0
6	0.000	0
7	0.000	0

Patches

Patch	Execution	Consumption...	Percent...
1	1	0.10	0.00
2	0	0.00	0.00
3	2	0.20	0.00
4	0	0.00	0.00
5	4	0.40	0.00
6	108	7.56	0.00
7	12	0.60	0.00

Loop Patches

Patch	Execution	Consumption(nJ)	Percentu...
12	55	2.86	100.00
32	0	0.00	0.00

Bound-Patches Set

Patches	Execution	Consumption...	Percentu...
22, 10	12137850	794,084.40	98.73
18, 26	119880	6,593.40	0.82
27, 16...	59940	3,596.40	0.45
7, 5, 3	18	1.20	0.00
33, 1, ...	5	0.38	0.00
2, 4, 8...	0	0.00	0.00

Figure 6: Tables presented in the user interface

Table 1: Consumption Pattern

Code Options	Energy Cost (J)	HCP Cost (mJ)	Average Power (mW)	Average HCP Power (mW)
matrix_fs_rv	1.05	490.36	43.72	42.59
matrix_fs_cf	1.14	465.02	41.69	30.57
matrix_fs_cb	0.91	474.7	43.73	40.17
matrix_fsp_rv	1.04	490.36	43.72	42.59
matrix_fsp_cf	1.14	465.02	41.69	30.57
matrix_fsp_cb	0.91	474.7	43.73	40.17

5. EXPERIMENTS AND RESULTS

This section presents the proposed framework evaluating a matrix product example that was taken from The Dalton Project‘s Benchmark, University of California Riverside [5]. As C compiler was used the C51 Keil compiler, a widespread compiler for MSC51 architecture. The Keil compiler allows nine optimization levels under two emphasis: *favor size* (fs) and *favor speed* (fsp). The optimization selection works in cumulative way: level 0 implements only *constant folding* technique, level 1 implement *constant folding* and *dead codes elimination*, level 2 implements *constant folding*, *dead codes elimination* and *data overlaying*, and so on. Due to this, three optimization levels was chosen for analysis: *constant folding* (cf), *register variables* (rv) and *common block subroutines* (cb). Six binary-codes was analyzed, that was generated by composing emphasis and level optimization. The estimation was performed based on the AT89S8252 consumption-model constructed according to methodology suggested in [9]. Table 1, Figure 7 and Figure 8 show the consumption pattern captured by the “hidden-tools”-PCAF environment. Figure 9 depict the more expressive segment of *Execution Profile*, where a *Loop-Patch* and a *Cluster* can be identified. Rolling the *Execution Profile* and the *Consumption Profile*, from the user interface, the designer is able to find out cost centers along the code. Additionally, based on the proposed taxonomy and the code graphical representation (the CPN model), the designer can figure out *Clusters* control structure, being a support for codification on Hardware Description Language (HDL). The Table 1 shows the best compiler optimization level: the *common block subroutines.* Such optimization level provide the lowest total consumptions (20.1 %) the highest time and consumption percentage for its Highest Consumption Patch (HCP)-40% and 52,17% respectively (see Figure 7 and Figure 8). As result software-hardware migration should be applied on matrix_fs_cb[6]. The HCP is at *Cluster* showed in Figure 9, from instruction number 166 to number 250. Such migration reduces the code size in 29,9%. Additionally, observing column 4 and 5 at Table 1, the designer can compare the HCP power cost with code power cost.

6. CONCLUSION

This paper presented a framework for evaluating software of embedded systems in order to software-hardware migration. This framework is focused on energy savings by migration of high consumption code segments to hardware.

[5]www.cs.ucr.edu/ dalton/i8051/i8051benchmarks/index.html

[6]For this example the compiler generates same binary-code for both option: matrix_fsp_cb and matrix_fs_cb

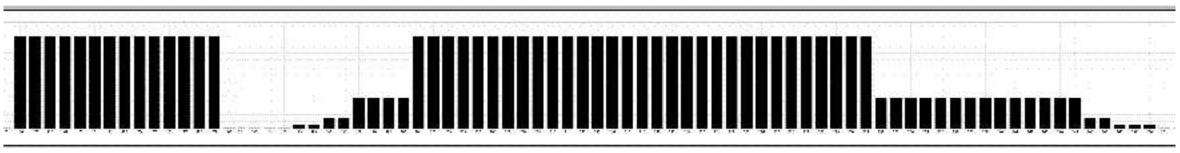


Figure 9: The Highest Loop-Patch and Cluster present on the case study

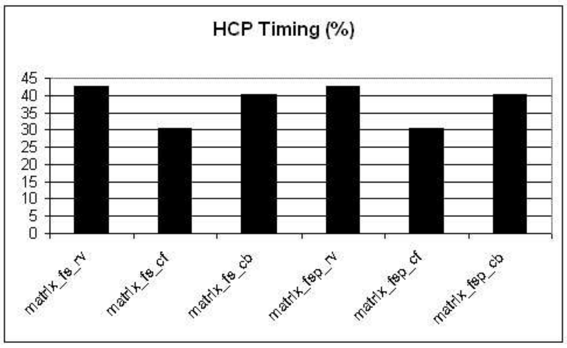


Figure 7: Time percentage for Highest Consumption Patch

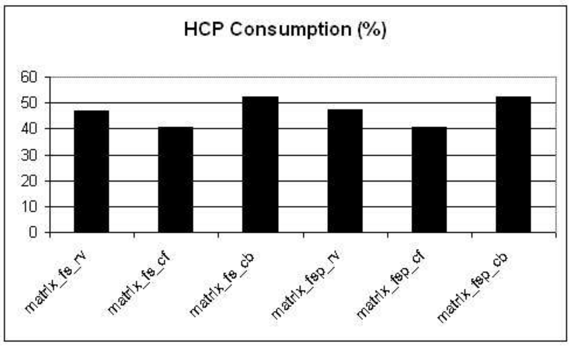


Figure 8: HCP consumption percentage at the total consumption

Such framework is integrated in the "hidden-tools" Environment in order to perform power-aware analysis to be applied on embedded systems binary-codes. Based on such analysis the designer can identify:(i)the best binary-code for migration, from a set of possibles high-level compiler outputs,(ii) the best code sector, within such binary-code, for mapping in hardware. The target software and processor architecture are modeled together by a formal computational model based on tokens, as a Coloured Petri Net. Due to this formal model, processor capabilities and behaviors are strictly translated into a set of rules and functions, defined by Coloured Petri Net semantics. Such modeling allow the usage of a set of widespread analysis technique available on the CPN universe, such as simulation- and state space exploration-driven techniques. The environment yielded is termed "hidden-tools"-PCAF, providing a powerful resource to explore binary-codes from the power-aware point of view.

7. REFERENCES

[1] Reference hidden due to blind review.

[2] Reference hidden due to blind review.

[3] Reference hidden due to blind review.

[4] F. Burns, A. Koelmans, and A. Yakovlev. Wcet analysis of superscalar processor using simulation with coloured petri nets. *Inter. Journal of Time-Critical Computing Systems,*.

[5] F. Burns, A. Koelmans, and A. Yakovlev. Modelling of superscala processor architectures with design/CPN. In Jensen, K., editor, *Workshop on Practical Use of Coloured Petri Nets and Design/CPN, 10-12 June 1998*, pages 15–30. Aarhus University, 1998.

[6] G. Gallasch and L. M. Kristensen. Comms/CPN: A communication infrastructure for external communication with design/CPN. In *3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pages 75–90. Aarhus University, Aug. 2001.

[7] K. Jensen. A brief introduction to coloured petri nets. *Tools and Algorithms for the Construction and Analysis of Systems. Proceeding of the TACAS'97 Workshop*, 1217(2):203–208, 1997.

[8] L. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *Inter. Jour. on Software Tools for Technology Transfer: Special section on Coloured Petri nets*, 2(2):98–132, 1998.

[9] T. Laopoulos, P. Neofotistos, and C. Kosmatopoulos. Current variations measurements for the estimation of software-related power consumption. *IEEE Inst. and Measur. Techn. Conference*, May 2002.

[10] S. D. Marcello Lajolo, Anand Raghunathan and L. Lavagno. Cosimulation-based power estimation for syste-on-chip design. *IEEE Trans. on Very Large Scale Integration (VLSI) System*, 10(3):253–266, June 2002.

[11] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, pages 541–580, April 1989.

[12] G. Stitt and F. Vahid. Energy advantages of microprocessor platforms with on-chip configurable logic. *IEEE Design and Test of Computers*, 19(6):36–43, Nov/Dec 2002.

[13] G. Stitt and F. Vahid. Hardware/software partitioning of software binaries. In *Proc. of the IEEE/ACM inter. conf. on Computer-aided design*, pages 164–170, 2002.

[14] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration Systems*, 2(4):437–445, December 1994.