Embedded Systems Energy Characterization using non-Intrusive Instrumentation

Nicolas Fournel, Antoine Fraboulet, Paul Feautrier

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Embedded systems energy characterization methodology using non-intrusive instrumentation

Nicolas Fournel
Antoine Fraboulet
Paul Feautrier

November 2006

Research Report N° RR2006-37

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

Embedded systems energy characterization methodology using non-intrusive instrumentation

Nicolas Fournel
Antoine Fraboulet
Paul Feautrier

November 2006

Abstract

This research report presents a non intrusive methodology for building embedded systems energy consumption models. The method is based on measurement on real hardware in order to get a quantitative approach that takes into account the full architecture. Based on these measurements, data are grouped into class of instructions and events. These classes can then be reused in software simulators and in high-level source code transformation cost functions for optimizing compilers. The computed power model is much more simpler than previous power models while being accurate at the platform level.
The methodology is illustrated using experimental results made on an ARM Integrator platform for which an accurate and full system energy model is build.

Keywords: Embedded Systems, Energy Consumption Model, Simulation Instrumentation, Optimizing Compilers Cost Functions.

Résumé

Ce rapport de recherche présente une méthodologie non intrusive de construction de modèles de consommation pour des architectures embarquées. La méthode utilise des mesures effectuées sur des plateformes réelles afin d'avoir une approche quantitative prenant en compte la plateforme complète. Les mesures sont ensuite groupées en classes d'instructions et d'événements pour simplifier le modèle. Ces données peuvent ensuite être facilement réutilisées dans des simulateurs instrumentés ou comme indiquation dans des modèles de coût utilisés dans les transformations de haut niveaux des compilateurs optimiseurs.
Une application de la méthode est présentée en utilisant une plateforme ARM Integrator pour laquelle un modèle de consommation est construit au nivveau système.

Mots-clés: Systèmes embarqués, Modèle de consommation énergétique, Instrumentation pour la simulation, Fonctions de coût pour compilateurs optimiseurs.

Contents

List of Figures

List of Tables

1 Introduction

Embedded computing systems go through a dramatic increase of computational power. While this computational power is growing, electrical consumption follows the same kind of trend. Unfortunately battery capacity used to power these systems does not keep the same pace. The consequences of these differences in evolution drive designers to take the electrical consumption as a major constraint. We reached a point where hardware solutions are not sufficient anymore. One of the multiple solutions to reduce significantly energy consumption is to organize software and drive the hardware in a power efficient way.

One can use software optimization techniques during software design or source code optimization at compile time. To know where the software is the more power-hungry and what are the best optimization choices, quantitative consumption data are of great value. Manufacturers reference manuals usually provide some consumption figures for different system parts. Unfortunately, these figures are generally not precise enough to build a complete system power consumption model. The usability of these figures (CPU instructions, caches, bus access, scratch-pad and external memories, peripherals, ...) is made difficult for software developers and tools by their varying levels of description that are most of the time different from each other among components of the same embedded system. The result is that it is very difficult to aggregate the figures of different components for a full system electrical consumption. In the same manner it is often difficult to handle them during hardware simulation or within optimizing compilers.

In this report, we propose a methodology which aims at building electrical consumption models that overcome these problems. As it is targeting embedded software development for fixed architectures, the power consumption model is based on measurement on real hardware. The measurement procedure should not be intrusive because these setups are generally difficult to implement and represent a lot of work that might not be feasible for software developers. The proposition of this report use a model building methodology based on external system measurements only. The model generated by this methodology must be simple enough to be used in fast simulation for software organization choices or automated optimizations at compile time. Experimentations have been made on an ARM Integrator platform and an accurate full system energy consumption model is built from a series of measurements using micro benchmark codes.

The report is organized as follows. In Section 2 we review the different existing techniques used for building hardware power consumption model and their use. Section 3 presents our methodology used to build full platform energy consumption models. Section 4 presents our experimental results on an ARM target platform that highlights the points mentioned above and shows that an accurate energy consumption model can be build and used in software platform simulators using only simple and non intrusive measurements. Finally, the model will be validated in section 5.1.

2 State of the Art and Related Work

Before giving some examples of existing models, we explain in this section how an energy consumption model is used and why it is interesting.

2.1 Energy consumption models

We give here an overview of the end user model usage. This overview will be preceded by an architecture presentation, which will help to understand this usage.

The aim of using a model in energy consumption estimation is to reduce the design complexity and to test some solutions before production. We will give more details about these specific aims after model building description. The architecture of targeted systems can be seen as a hierarchical stack. At the lowest level, we find the transistors, elementary components of VLSI (Very Large Scale Integration) circuits, and at the upper level, there is the overall system. This hierarchy is depicted on Figure 1.

Every model is built at a specific level of granularity. The major influence of this level of granularity is on the model parameters. The parameters of a system-level model are not as fine grained as the one taken for a gate-level model. For example in the first case we will have microprocessor instructions whereas in the second we will use binary test vectors.

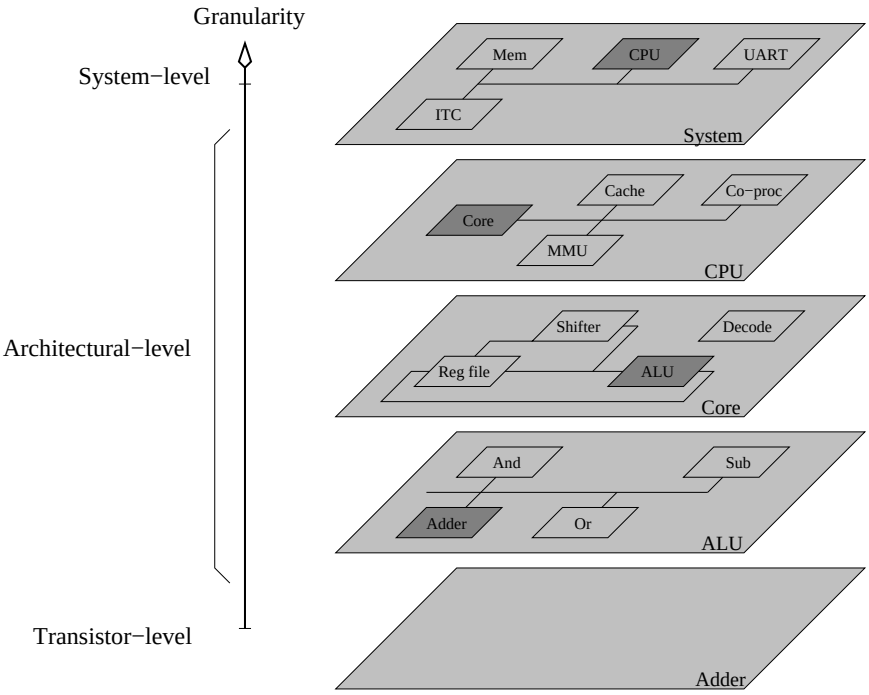

Figure 1: Architectural hierarchy from transistor to embedded system

When using the model, we must use the correct parameters values to get the most precise energy consumption estimation. If the model has architectural level parameters and if our goal is to know how many Joules were spent by a given software application, we should proceed as follows. We should simulate the behavior of the platform between system level (usage level) and architectural level (model level), to estimate the model parameters values. At system level, activity informations are given by the application itself. Once the parameters values are fed in the model, a computation phase is needed to aggregate the data. This example is illustrated by figure 2.

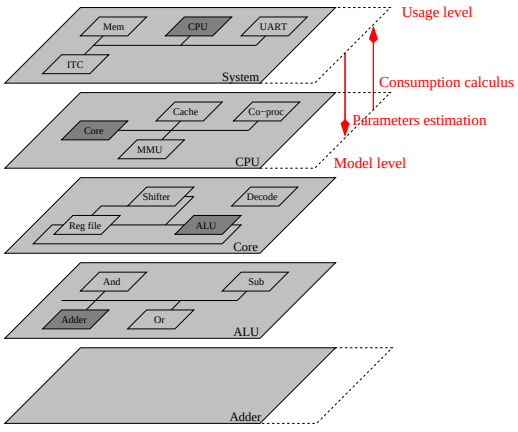

Figure 2: Energy consumption model usage.

2.2 Model classification and related work

We will now see that many work has been already done on the task of energy characterization or power estimation. Before giving examples of these works, we will expose here how we decided to classify them. These models can be characterized by two main orthogonal criteria. The first is the level of granularity of the resulting model, and the second is the data acquisition method at model building time.

2.2.1 Level of granularity

The first criterion is level of granularity of the model. Up to now there exists several power consumption models for processing systems (general purpose and specialized processing unit). These models can be ordered by level of abstraction, from circuit-level (the lowest) to system-level (the highest). Figure 1 depicts the increasing level of granularity of the model as far as we consider a higher level in the architectural hierarchy. We can classify them into three main groups : circuit/gate-level models, architectural models and finally system/instruction-level models.

Thus, in the highest class of abstraction level, which is instruction/system level, efforts are made to characterize the energy consumption of instructions. Some only builds an instruction cost table, where others also take into account the inter-instruction cost (control logic switching cost) or the data values (data logic switching cost).

At the architecture level, the system is first divided into functional blocks. Each of these blocks has its own energy consumption model. An important advantage of this kind of model is the possibility to choose completely different types of model for each functional block, and thus completely different parameters for them. Most of the time, models used for functional unit are analytical models, but statistical or empirical models built thanks to simulated or measured data can be used too.

Finally, as far as circuit/gate level model are concerned, the work is to describe very fine grained behavior of the target system or processor.

2.2.2 Data gathering method

In the model building process, quantitative value of energy consumption are needed to calibrate the final model to suit more closely the underlying architecture energy consumption. Two of the methods can be distinguished by their data acquisition method and the third does not even need data. For this criterion, we can find the three following classes : analytical models, simulation based models and physical measurement based models.

The first method of model building is analytical construction. In fact, in this method no energy consumption data are needed. This method use physical laws and architecture description to predict accurately the electrical consumption of the targeted element. This method is often used on low-level and regularly structured units such as caches memories. It appears to be really difficult on irregular structures or not really accurate due to extreme simplifications.

The second method gathers quantitative data from simulation. In this method, we build the model by a bottom-up process, which consists in using models of lower level of granularity, as described in section 2.1. This process can be repeated level by level. At each step, parameters of the lower level must be calculated by simulation and quantitave data gathered. All informations are then deduced from the hardware architecture. Figure 3 depicts this method. Since this method only requires the detailed architecture of the system, no hardware platform is needed. It can be used during design of VLSI circuits to estimate the resulting consumption or to test the viability of new solutions before production.

An alternate solution to simulation and lower level model based methodology is to use measurements. Indeed, by measuring the energy consumption on real hardware, it is possible to build an energy consumption model. In fact, the production process of this kind of models is the exact opposite of the one described before for simulation based ones. The process is a top-down construction, since the data gathered by measurement concern the overall system, and detailed informations must be extracted from these global ones, and not aggregated. Figure 4 gives an overview of this method. The models built by this kind of method are generally less complex since quantitative data are coarse-grained.

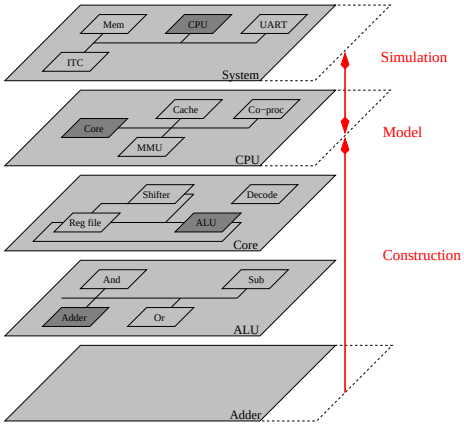

Figure 3: Simulation based energy consumption modelling.

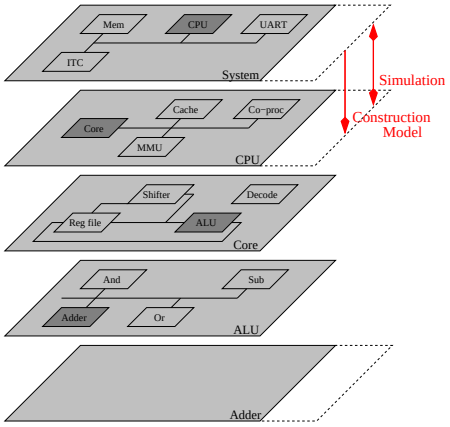

Figure 4: Measurement based energy consumption modelling.

We will give examples of each methods at different level of granularity in the following section. As our interest is in a measurement based method, we give here a few examples of measurement setups proposed in the literature. The solutions proposed for measurement have a wide range of complexity.

- The simplest solution is proposed in [20], by Tiwari et al. It consists in using a ammeter connected to the power supply pins of the processor. This method has for advantage to be very simple to implement, but it has also a huge disadvantage, which is the loss of information. Due to the poor time resolution of the ammeter, we only get mean values.

- A way to overcome this problem is proposed by Russell and Jacome in [18]. Their data acquisition solution use a digitalizing oscilloscope coupled with a resistor placed in series with the power supply connection. For this kind of measurement, they use a high performance oscilloscope (LeCroy LC534) which has a high sample rate. The setup even allows the authors to have a trigger signal, which gives the beginning and the end of the measurement period. The sample rate of the data is limited by the performance of the oscilloscope.

- Another solution for instantaneous current measures is proposed by Nikolaidis and Laopoulos in [16]. The setup is based on an oscilloscope and a current mirror unit. The reason of a current mirror usage is the reduction of the influence of the measurement setup on measured hardware. The authors can measure power consumption of the system at high sampling speed without interfering with the supply voltage.

- Finally, Chang et al., in [3], propose a setup that gives the consumption cycle by cycle. This setup is based on the charge and discharge of capacitors. Indeed, the setup is placed in series with the power supply. The principle is that at each cycle one capacitance charges and the other discharges. Thus by sampling only twice a clock cycle they can deduce the power consumed by the target system during this cycle. The setup is completed by an AD converter (Analog-Digital Converter) which is in charge of sampling, and a Fast SRAM that is fed by the ADC with the voltage values. The data are then computed from the SRAM.

2.2.3 Related work

The two criteria of classification are not fully orthogonal, since all combination are not possible or pertinent. Indeed, analytically built models targets low level units due to the complexity of the building process. This building method will not be detailed any further in the remaining of this report, because our interest is in a higher level model. As far as simulation based model are concerned, they are rarely used in the highest level of granularity since they are highly time consuming at building time. Finally we do not find lower level model based on measurements since it is complicated to extract low level informations from measures which are representing the whole system consumption.

In the following paragraphs, we will give an overview of the basic models of electrical consumption of circuits, and then some examples of abstraction level/building method combinations investigated in related works. The presentation of these works will be organized by level of abstraction.

The basic model Before giving examples of models by architectural layer class, we present here the basic power model of VLSI (Very Large Scale Integration) circuits. This model is one of the lowest level power model, since it models the power consumption of the most elementary gate, the inverter (2 transistors).

The consumption of a gate can by divided into two main parts : static and dynamic power. The first part, static power, does not depend on the gate input changes and hence gate activity. However, the second part, dynamic power, is correlated to these input signal changes.

The static power is composed of the current leakage of a closed transistor. It is usually modeled by equation (1). V_{dd} is the power supply voltage and I_{leak} is the leakage current.

$$P_{static} = V_{dd} \times I_{leak} \tag{1}$$

The dynamic power can be subdivided into two parts : short-cut power and output capacitance load power.

The short-cut power is due to the fact that the transistors are all (N-MOS group and P-MOS group) opened at the same time in the transition period. Some are switching from 1 to 0 and the others from 0 to 1. When the input signal is around the threshold voltage V_{th}, the two group of transistors are open and a current can pass between ground V_{cc} and V_{dd}. This power can be modeled by equation (2). Once again V_{th} is the threshold voltage and V_{dd} is the power supply voltage. Finally, τ is the signal raise time.

$$P_{short\_cut} = K.(V_{dd} - 2V_{th})^3.\tau \tag{2}$$

The second part of the dynamic power is the output capacitance load power. This formula express the fact that the energy stored in the output capacitance is shorted to ground when the input switches from 1 to 0.

$$P = \frac{1}{2}CV_{dd}^2 \tag{3}$$

where C is the output capacitance of the gate.

In present day technologies, the first two sources of power dissipation are negligible against the amount of power that the third represents. The power consumption of the inverter is then simplified to the output capacitance load described by equation (3).

This model is generalized to all gates by the approximation that all other consumption can be neglected against the output capacitance charge.

A second generalization is used in architectural and system level models. A naive extension of the model described before (equation (3)) to a full block or chip gives formula (4) :

$$P = \frac{1}{2}C_m V_{dd}^2 f \alpha \tag{4}$$

where C_m is the total output capacitance of the system, f is the operating clock frequency and α is the proportion of gates switching from 0 to 1 in a clock cycle. The parameters α and C_m are difficult to estimate but can be obtained by detailed simulation.

This latter model is widely used, even in other models, where it gives approximation of the power consumption of a system or block, but it is not adapted to the specificity of the logic contained in the block.

Transistor/Gate-level Models One of the most accurate methods to estimate power consumption before a circuit is realized is doubtlessly transistor/gate level simulation. In fact most of synthesis tools provide power consumption prediction. For example PowerMill [8] from Synopsys and QuickPower [7] from Mentor. These tools are low level (circuit or HDL) simulators. Other simulators operate at circuit level, such as Spice-based simulators (Star-Sim [11] for example). This kind of simulation gives accurate fine-grained results, but are very time-expensive. In fact the time of simulation limits the number of events simulated.

A first improvement for this drawback is the gate level simulation. The elementary unit is not the transistor anymore but the gate (an assembly of transistor). Mynoch [17] for example, runs 450 times faster than Spice based simulation.

The models (and simulators based on these models) presented here requires detailed information on the hardware modeled, HDL source or equivalent informations. At software development phase, it is almost impossible to get these information from the manufacturers. Then this kind of models do not meet our needs of simplicity. Moreover measurement based model does not exist and are probably impossible to build at such a low level of granularity.

Architectural-level models As we saw before, the peculiarity of architectural models is that they are centered on the functional units. Energy consumption is estimated by estimating the consumption of all the blocks.

The first architectural-level model we will talk about is proposed by Chen et al. in [6, 5, 4]. The system modeled in this work is a full system comprising a CPU and a DSP, plus a bus and memories. This system is divided into functional blocks, such as registers, ALU, MAC, ... On top of that, the decomposition proposed by the authors is hierarchical, and apply to all features of the target. As far as efficiency is concerned, the authors want their model to be accurate, hence they decided to take into account logic

switching generated by instructions and data values. The blocks are grouped into two families : *bit-dependent* and *bit-independent* blocks, regarding if their consumption varies while input vector changes. The models used in the functional blocks are based on tabular form called Look-Up Tables (LUT) filled thanks to the model presented in equation (4). The final results gives an accuracy at about 9% of the real values.

Li and Henkel [14] build a model at a higher architectural level. Indeed, the system class targeted by their model is also a full system integrating a CPU, memories and even custom ASICs (representing peripherals), but in their model the CPU is a unique functional block. Other blocks are then main memory, caches and specific hardware. The models used for each of the units are purely analytical ones, based on architectural data found in the literature (number of row and columns for memory, ...), or behavioral data of the application (number of miss, ...).

Kim et al. [10] augment a cycle accurate simulator with an architectural power consumption model. The targeted system of their model is a CPU. The model proposed has the peculiarity to be a "recursive" architectural level model. The system is divided into functional units called micro-architectural blocks. What is interesting in their model is that the micro-architectural models can be subdivided in turn. The micro-architectural block consumption is defined by three components : load of the input capacitance, switching of the logic due to the switching of inputs and finally the leakage power. These data are stored in LUTs calculated off-line (the LUTs can be replaced by analytical models for example).

Wattch [1] is a power model integrated in SimpleScalar [2], a cycle accurate instruction set simulator. The modified SimpleScalar tracks the access to functional units to predict energy consumption of a CPU. The authors regroup the functional unit power model into four classes : array structures, CAM (Content Addressable Memory) structures, complex logic blocks and clocking. Array structures represents caches for example. CAM structures are part of TLBs (Translation Look-aside Buffer) or write queues. Complex logic blocks are ALU, FPU. Clocking represents the clock distribution tree. In each class, the consumption is estimated by a parameterized analytical model based on architectural parameters. As far as Wattch performances are concerned, it performs power estimation 1000 times faster than circuit level simulator with an error of only 10%.

SimplePower [22] works in the same way as SimpleScalar, by simulating the execution of each instruction in each pipeline stage of the CPU. From these information only activated functional blocks of the architectural model of SimplePower are called to estimate power. The simulator use a cache simulator, analytical models and LUT (Look-Up Tables) models to predict power consumption. These models are fed with input values and behavioral informations (ex : cache misses, ...). The resulting accuracy of this model is about 15% against transistor level simulation.

To conclude, architectural level power models are build to be flexible. In fact, the main aim of most of them is the reuse of part of the model between different target system. Indeed, there is no need to recreate an entirely new model for a new architecture, but to add, remove or modify existing functional blocks. Their goal is to be less complex than circuit-level models, and more accurate than instruction-level models. In this kind of model, every gate or transistor are not simulated, hence the model is less time-consuming, but the model take into account the specificity of certain part of the architecture by using different model for each functional unit of the system. Some of the models presented in this family are easily adaptable to our objectives of modelling the full platform, since they can be augmented to take into account the full system (CPU plus peripherals, memories, ...). This is the case of the first two examples of architectural model. In the examples cited here no measurement based models are present.

Instruction-level models Tiwari et al. [20, 21] model of power consumption estimation for a CPU (x86) is based on measurement. The power measurements are obtained by measuring the current drawn by the CPU with a digital ammeter. As this tool averages the values, simple measurement would have meant nothing. The method proposed by the authors gives a mean consumption value for each instruction by executing each one in a well sized loop and measuring the consumption of the overall application. The consumption is then divided by the number of instruction executed. The sum of instructions mean consumption does not reflect the effect of control logic switching between instructions. In order to take this into account, the authors proposed to characterize what they call inter-instruction consumption. To measure this, they proposed to renew the same experience with each combination of two instructions in the loops.

Once the consumption obtained, they subtract the amount of energy due to the instructions themselves, and then divide the rest by the number of instruction switch. In their approach, inter-instructions are symmetric. The main difficulty of this method is that the table of inter-instruction has a size of N^2 (if N is the size of instruction set). The accuracy of their model is within 3% of the measured values.

Lee et al. [12] enhance the model described before on a DSP, by regrouping instructions into classes. Thus the complexity of the inter-instruction table falls under $O(N^2)$. The accuracy still stays under 10% of error.

Another work, from Steinke et al. [19], uses an ammeter to get the desired consumption on an ARM7TDMI. Their model is build thanks to linear regression with detailed instruction informations like the instruction, and its parameters (register number, register value, immediate value, ...). The error falls here under 2%.

The previous measurement procedure has an important drawback, which is its narrow frequency spectrum. The setup only allows them to get mean power consumption values. In this condition they loose peak power consumption. Russell and Jacome in [18] propose a solution that has a better temporal resolution. They use a digital oscilloscope (Lecroy LC534). Thanks to the temporal resolution enhancement, instruction power consumption are more precise. Indeed, Tiwari et al. are forced to measure consumption of the instruction repeated in a loop. But their measures include the consumption of the branch instruction too, neglected because it is executed only once in a huge loop. Russell and Jacome [18] use a trigger signal to only measure the consumption of the body of the loop (without the branch instruction). The resulting model proposed by the authors is based on a statistical method, a constant parameter model. The parameters of this model are the types of instructions. Experiments were made on an Intel i960 and the estimated values give about 8% of error.

Lee et al. [13] propose another solution to get rid of the inaccuracy of the measures based on statistics. Their technique is also based on a simple setup for measuring real power consumption on an existing target (CPU), and aimed at building an instruction-level model. What is different in their approach is the use of regression analysis. If applied correctly, their model allows to know even less architectural informations on the target system/processing unit and reach a good accuracy of 2.5%. In fact by applying the regression successively with each selected parameter, one does not have to know the underlying architecture because the parameter are not estimated from the hardware. The parameters are finer-grained than the previous proposition, since in this model takes into account the per pipeline stage cost of instructions.

All models here are centered on estimating CPU power consumption. Our interest is to estimate this consumption plus the consumption generated by the rest of the embedded system. System/Instruction level models would not allow this, our proposition will not be based at system level of granularity. But measurement are widely used at this level of abstraction, some of the building methods proposed here have the characteristics we are wanting of our methodology, simplicity, minimum architectural information.

To conclude, simulation and analytical model building methods are generally oriented for early stage VLSI design, before material production. Conversely measurement based method needs less information on the underlying material architecture. These last points drive us to propose a measurement based methodology, since at software development phase, all informations needed for simulation based method will not necessarily be available.

The second point is that the setup used for data acquisition in our methodology must not require electronic skills. Compared to all setup proposed in the literature we should use one of the simplest.

Finally, all system/instruction level models proposed before are centered on a CPU, and do not take into account peripherals. The whole system is then not modelled with these models. Some exceptions are present in the architecture level models. For example, Li et al. [14] propose a model in which CPU, memory and busses are different units. This model is closer to system-level modeling than the instruction-level ones.

3 A methodology for energy consumption model construction

As presented before, until now overall embedded system energy consumption model were proposed, but they are not based on simple data acquisition methods. Conversely simple acquisition method based models were proposed, but they are usually taking into account only a subset of the system, the CPU. We will

propose in this report a methodology which is both using simple non-intrusive measurements and a model that is representing the whole embedded system. The aim of this methodology is to be simple and generic enough to be reproducible by software developpers on their hardware platform.

In this section, we will give details about this proposition of methodology for building an energy consumption model based on simple non-intrusive measurement. We will describe the measurement setup, we will give an overview of the measurement protocol and the data extraction. Finally we will say a few words about the setup validation.

3.1 Measurement setup

First of all, it is necessary to remember what we have to measure exactly to know what our system is consuming. Basically, every electrical appliance power is given by the following formula :

$$P = IV \tag{5}$$

Where I is the current drawn at input and V is the power supply voltage.

In fact we will be more interested by energy, since it represents the real cost in terms of electrical consumption. The definition of the energy is the following :

$$E = Pt \tag{6}$$

where P is defined by formula (5) and t is the time.

These two definitions are correct if the current and the voltage are constants. This is exactly what Tiwari et al. [20] considered by using using human reading on an off-the shelf ammeter. It should not be the case on our target system, so current and voltage become functions of time i and v. On top of that, the process of data acquisition, that is to say physical measures induce that i and v will not be continuous, but in discrete time. We then have the new definitions :

$$p_j = i_j v_j \tag{7}$$

$$E = \int^t pdt \approx \sum_{j=1}^{n_{samp}} p_j \Delta t_{samp} \tag{8}$$

Where $\Delta t_{samp} = \frac{1}{f_{samp}}$ and f_{samp} is the sampling frequency.

In that condition, measurement will consists in sampling current and voltage. From these informations we can deduce the power and energy consumption of the system during the time of the experiment.

Now that we know what we have to measure, we must define where to take these measures.

The choice of measurement point is very important. In fact, this choice will have an influence on many other choices in the following steps. The most important thing is that it is tightly coupled with the informations we can/want to extract from the measures.

What we mean here is that if we want to measure the electrical consumption of the CPU, we would have better sampling the current and voltage at the power supply input of the CPU. This is a huge prerequisite, which is that it must be accessible and that you can instrument it for current sampling (voltage sampling is easier).

As we said before, we are not necessarily interested in very fine grained measures. On top of that we base this approach on non intrusive measures, because software developers often do not have time and skills to make this kind of measures. The better way to be non intrusive is to place our measurement point at the power supply input of the system.

More precisely, our approach is based on the whole system electrical consumption in order to closely fit to the real consumption of the studied system. This means that we should place at the point where the battery is fitted, since it is the only energy supplier in embedded systems.

This approach has an interesting advantage, since it allows the model generated to take into account the integration costs (in term of electrical consumption). We not only measure the consumption of the main chip containing the desired device, but of the components.

Finally, once we have chosen such a measurement point, we must select the measurement material, which will be used in the model building. There is only one contraint in this selection, which is that probes bandwidth and sampling frequencies must meet Shannon's law on target signal. If we still would like to have CPU electrical consumption and if our measurement point is placed at the power supply input of this CPU, we might probably have signals whose frequency is the CPU frequency. We would have then to select an acquisition material of which bandwidth is sufficient to acquire this signal.

In the experiments depicted in section 4 we decided to sample current and voltage at the power supply input of the development board ARM Integrator CM922T-XA10. At this boards power supply input, we have the signal depicted by figure 5.

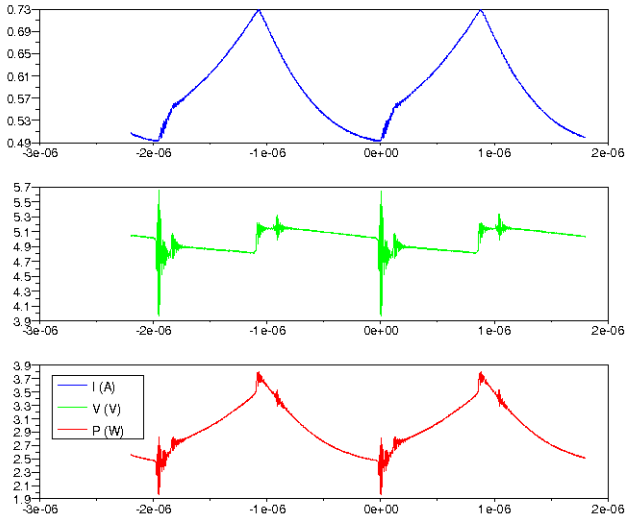

Figure 5: Sampled signals at 2.5G samples/s : We can observe that the sampled current and voltage present a frequency of 500kHz which is far away from the CPU frequency (198MHz)

The sampling rate for this figure is 2.5G samples per second. The frequency of the signal depicted is about 500kHz. The explanation for this is that the board current switching stabilizer has an operating frequencies of around 500kHz (as stated on the manufacturer's specifications). As a consequence, it is evident that we will not have cycle accurate energy consumption. In fact, the voltage stabilizer does not do a perfect job. Power variation are due both to events in the evaluation board and to the supply voltage variations. The latter effect must be averaged out of the measurements.

The second important fact is the duration of the measurement. In fact it can be limited by the hardware used for the acquisition. If it is the case, it will give an upper bound to the sampling rate. In our case there are two stabilizers. Their frequencies are slightly different *i.e.* about 2 or 3 kHz. The time of experience must then be sufficient to take this variation into account. We had to adjust the time window by modifying the sampling rate.

The resulting setup is the following. For current sampling, we selected an ampermetric clamp Tektronix P6021. As far as voltage sampling is concerned, we used a voltage probe Tektronix P6139A. Finally, the acquisition was made by a DPO (Digital Phosphor Oscilloscope) TDS 7054A series. This oscilloscope was only used as an acquisition device.

In terms of sampling frequencies, we finally used a 2.5M sample per second rate, which gives us a 10^{-2} s time window thanks to its 250 000 sample memory.

3.1.1 Protocol

Once the measurement setup is chosen, which means we have decided where and what will be measured and with what material, we have to organize our experiments in order to extract the information we are

interested in. Indeed, placing the measurement point at the power supply input, will certainly not allow us to get CPU cycle accurate measures.

As our main aim is to build a software electrical energy model, the events which will be the subjects of our measures will range from CPU instructions to operating system services. More details about those events will be given in section 3.3. What is important to underline here is that the duration of most of the target events will fall below the time accuracy of the setup.

To solve this problem, there are two solutions. The first will be to change the setup to collect measures closer to the CPU, for example. This solution is not acceptable, since this measurement point was chosen for reasons expressed above. The second solution is then to repeat each event a certain number of times, and average the energy consumption.

In this second solution, measurement experiments must meet a several requirements. We must know how many times the target event occurs during the measurement interval, we also need to know the exact duration of the experiment, and finally it is important that nothing else than the target event happens during the time window. All these requirements induce that we base the measurements on benchmarks raising each a different target event.

First of all, we have to know exactly how many times the event occurs during our measure in order to make a correct average. Indeed, supposing that we can calculate exactly the energy spent in the benchmark, we will have to divide this amount of energy by the number of occurrence to have the per event energy cost. The only way to get this information is to build the benchmark to execute the event a known-in-advance time. This number of repetition should be adapted to the acquisition window imposed by the measurement material. Two alternatives are available for implementation of the benchmark, the first would be a fully linear program containing the required number of events, or the implementation of a loop containing a submultiple of the number of event.

In our case, we select the second solution, since many embedded systems integrates caches, and then the loop implementation can take advantage of it, if the size of the loop is adapted to the cache geometry. In that situation, instruction cache misses will not pollute benchmarks results. The cost of the loop in time and power must be estimated and can be taken into account in the results extraction.

As we said above, the loop must be of the right size in order to eventualy take advantage of the instruction cache.

In fact, the loop body must be large enough to minimize the influence of the loop skeleton on the final results. It must also be smaller than the size of the instruction cache to avoid cache misses during the loop execution.

Other factors are important and must not be forgotten, which are cache loading of the loop. Indeed, this loading phase generates compulsory cache misses. The best way to get rid of these misses is to preload the loop body in the cache before starting the measurements. This is possible on some processors. The second solution is to minimize the influence of these misses on the measurement.

The influence of this second class of factors is opposed to the first class ones. The smaller the loop is, the less misses we have. The loop size choice is then a trade off problem.

To formalize this problem, we have to identify the two major components. Equations (9) and (10) give the number of occurences of each of these components n_{loop} and n_{misses}, against the total number of events in term of instruction n_{insn_evt}, the loop size l_{loop}, and the cache line width in words w_{cache_line}.

$$n_{loop} = \frac{n_{insn\_evt}}{l_{loop}} \tag{9}$$

$$n_{misses} = \frac{l_{loop}}{w_{cache\_line}} \tag{10}$$

Once these two amount are weigthed by their cost we obtain the problem expressed by equation (11).

$$E = \frac{n_{insn\_evt}}{l_{loop}} \times c_{loop} + \frac{l_{loop}}{w_{cache\_line}} \times c_{miss} \tag{11}$$

where c_{loop} and c_{miss} are the costs of loop skeleton and cache misses. These two costs can be either time costs or energy costs. The most relevent one would be energy cost, but a first approximation can be

estimated thanks to time (cycles) informations, since time and energy consumption are correlated in this kind of problem.

The optimal value is hence obtained thanks to the solution given by (12) :

$$l = \sqrt{\frac{n_{insn\_evt} \times c_{loop} \times w_{cache\_line}}{c_{miss}}} \quad (12)$$

$$l < \frac{s_{cache}}{w_{insn}} \quad (13)$$

$$(14)$$

where s_{cache} is the cache size in bytes and w_{insn} is the instruction width in bytes. For example, if we have a 8 word wide line cache of 8kB, with a c_{loop} of 4 cycles and a c_{miss} of 40 cycles, the optimal loop size is 2048, for 15000000 events.

To conclude on loop size choice, we must underline the fact that the estimation of number of misses is based on the fact that the number of misses is predictable. In fact it is a little simplistic in the previous problem, since in some associative caches the replacement policy is random. If we put two lines in the same set, the probability of the replacement of the previously loaded line is not zero, as it is for a round-robin replacement rule. The estimation of the number of misses should be adjusted to the level of associativity. The solution to get predictability in the benchmark with random replacement rule is to only use one line of each associative set. Such a restriction avoid a line to be replaced by a second load in this same set.

The final solution is then to use prefetching solution if they are available, or to minimize the misses influence by resolving the trade-off exposed before, with the predictability constraint.

As we mentioned above it is important to be able to calculate the energy consumption of the benchmark, and only it. We should thus know when the core of the benchmark starts and when it ends. These informations can be collected thanks to a trigger signal controlled by the benchmark itself. This trigger signal must change state just before the first iteration of the loop and after the last one. The interesting data can then be selected thanks to this third sampled signal.

The last benchmark requirement for accurate event cost measure is the control of what is running on the platform. Indeed, during the benchmark core loop, only the target event should happen. This means that the platform must be initialized in a way that deactivate all useless peripherals, and only the benchmark should run on the platform.

To conclude the measurement experiments are organized as follows : each event energy cost measure is the subject of a particular benchmark, whose task is to trigger this event a predefined number of time, using a loop.

For example, in our experiments on the Integrator CM922T-XA10, each cycle of the measured signal represents approximately the electrical consumption of 400 CPU cycles. CPU runs at a frequency of 198 MHz and the voltage stabilizers frequency is about 500 kHz.

In the experiments on the CM922T-XA10 we opt for a LED as triggering signal. The benchmark turns it off just before the beginning of the loop, and on just after the last loop, the data acquired between these two fronts are relative to the energy spend in the benchmark. The interesting data can then be selected thanks to a third sampled signal. Figure 6 gives an overview of the acquired data (two upper graphs) and the trigger signal (lower graph). The payload data are present between the two state change of the trigger.

In order to have a full control of the platform, we use a lightweight operating system called Mutek [15]. At present, we only use hardware initialization of Mutek. OS initialization is replaced by the benchmark body.

As we explained in section 3.1, at the end of a data collection phase we have two series of measures, on current and on voltage. We also presented how we can extract power information from the data (7).

From this sampled power we can extract two different informations. The first is a mean power over the benchmark.

$$P = \frac{\sum_{i=1}^{n_{samp}} p_i}{n_{samp}} \quad (15)$$

where n_{samp} is the number of samples. Mean power gives us information about gate activity. This is an interesting information, but it does not allow us to have the real cost of an addition or a memory access.

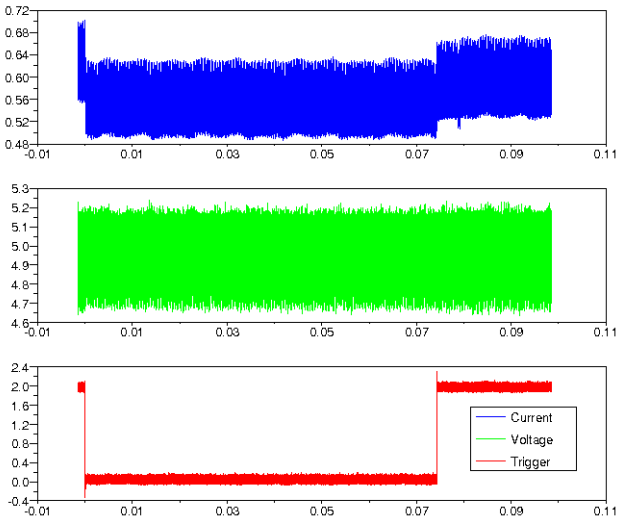

Figure 6: Acquisition results : the two upper graphs are interesting data i and v, and the lower one is the trigger signal LED

The main reason is the heterogeneity in event lengths. In our case we want to know the cost in term of battery consumption.

We will work in energy rather. As we mentioned before it is given by equation (8). This equation gives the total energy consumed during the benchmark, this information is important, but should be normalized to give the event consumption. To do so, all we have to do is to normalize the energy by the number of event executed during the benchmark :

$$E_{evt} = \frac{E_{bench}}{n_{evt}} \tag{16}$$

where E_{evt} is our event energy cost, E_{bench} is the benchmark total energy and n_{evt} is the number of times our event happened during the benchmark.

If the loop skeleton cost is negligible, the previous result can be considered as the final result, but if it is not the case, we should subtract the cost of loops to the total energy of the benchmark.

The loop skeleton may be evaluated by an empty benchmark, and its cost E_{loop} may be calculated by the previous equation. Once this cost estimated, we can use the following equation for the other benchmarks :

$$E_{evt} = \frac{E_{bench} - E_{loopcost}}{n_{evt}} \tag{17}$$

where $E_{loopcost} = E_{loop} \times n_{loop}$, and E_{loop} is the cost of one loop cycle and n_{loop} is the number of loop executed in the benchmark.

This gives more accuracy to the value obtained.

3.2 Measurement error verification

In order to validate the measurement setup, we have to estimate the error made on measures. The error will allow us to know the reachable precision in measures.

We make a measurement on both sampled signals, i and v. This error is modelled as follows :

$$i = i^{meas} + i^{err} \tag{18}$$

$$v = v^{meas} + v^{err} \tag{19}$$

where i and v are real values, i^{meas} and v^{meas} are measured values and i^{err} and v^{err} are errors due to measurement setup.

If we put these new definitions of i and v in equation (7), we get the following result :

$$p_j = (i_j^{meas} + i_j^{err})(v_j^{meas} + v_j^{err}) \tag{20}$$

By deduction, we can state that $p_j^{meas} = i_j^{meas} v_j^{meas}$. In that case we obtain the following error p_j^{err} :

$$p_j^{err} = i_j^{err} v_j^{meas} + i_j^{meas} v_j^{err} + i_j^{err} v_j^{err} \tag{21}$$

The error made on power measures (equation (21)) can be then reported in the energy calculus defined by equation (8).

$$E_{bench} = \sum_{j=1}^{n_{samp}} (p_j^{meas} + p_j^{err}) * \Delta t_{samp} \tag{22}$$

As we did previously for power, we can state that $E^{meas} = \sum_{j=1}^{n_{samp}} p_j^{meas} * \Delta t_{samp}$. The error on total benchmark energy is then given by equation (23).

$$E^{err} = \sum_{j=1}^{n_{samp}} p_j^{err} * \Delta t_{samp} \tag{23}$$

The final step in information extraction is the normalization of the results at the event. This is made thanks to equation (16). In the same way, the error can be normalized. Then we obtain :

$$E_{evt}^{err} = \frac{E_{bench}^{err}}{n_{evt}} \tag{24}$$

In our particular case, the measurement error is bounded by the error made at quantization by the oscilloscope. Error of the ampermetric clamp and the voltage probe are negligible in respect to the error made by the oscilloscope Analog-to-Digital Converter (ADC). According to the adjustments of the oscilloscope, we obtain the following errors on each channel :

As far as current is concerned, we convert every measure of 2A into a range of 16bits. The quantization error is then :

$$i^{err} = \pm\frac{2}{2^{16}} = \pm 3.052\ 10^{-5} A$$

Voltage error calculus is very close, since we have 10V converted on binary values on 16bits. He have then :

$$v^{err} = \pm\frac{10}{2^{16}} = \pm 1.526\ 10^{-4} V$$

To validate the measures taken during our benchmarks, we computed their error. The resulting conclusion is that we have a good accuracy. For example, on a benchmark which aim is to give us the consumption of a `load` instruction, the energy cost found is $1.242761\ 10^{-08}$ J and the measurement error is equal to $1.154064\ 10^{-12}$ J.

3.3 Model structure and parameters

Our choice among all the modeling method which have been presented in Sect. 2.2.3 is to build an architecture level model, in which the platform is divided in functional blocks, as in figure 7.

The energy consumption of an application E_{app} is obtained be adding the all blocks consumptions E_{bl}:

$$E_{\text{app}} = \sum_{blocks} E_{\text{bl}} \tag{25}$$

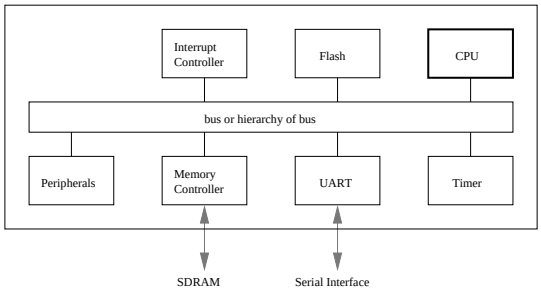

Figure 7: model block examples

Each block can have its own energy consumption model. To have a platform model better suited for software development, we apply the instruction level model solution for CPU modeling. The CPU energy consumption E_{CPU} is thus model as described in the next equation.

$$E_{\mathrm{CPU}} = E_{\mathrm{insn}} + E_{\mathrm{cache}} + E_{\mathrm{MMU}} \tag{26}$$

The energy consumption is the sum of the energy consumed by instruction execution, plus cache and MMU overheads consumptions, and consumption of the other blocks of the platform.

$$E_{\mathrm{app}} = E_{\mathrm{CPU}} + \sum_{blocks} E_{\mathrm{bl}} \tag{27}$$

This model aims at being integrated in a cycle accurate simulation tool of the complete platform. The most interesting way of writing the model for this kind of purpose is to define a per time slot energy consumption. The chosen time slot is the CPU instruction execution. There are two reason for choosing this time reference. The first is that it is the finest time reference since CPU have generally the highest clock frequency in embedded systems. Secondly, interrupt requests, the only mean for the hardware peripherals to interact with the software, are managed at the end (or beginning) of the instruction execution. From a software point of view, there is no need to use a finer time reference to report hardware events more precisely.

The model can be rewritten in a form where the consumption of CPU and other blocks are reported for the currently executed instruction. All E_* will be kept for overall application consumptions, for the sake of notation simplicity instruction reported consumptions will be noted as $\mathcal{E}_*$. This new model formula is expressed in the following equation:

$$\mathcal{E}_{\mathrm{slot}} = \mathcal{E}_{\mathrm{CPU}} + \sum_{blocks} \mathcal{E}_{\mathrm{bl}} \tag{28}$$

$$E_{\mathrm{app}} = \sum_{insn} \mathcal{E}_{slot} \tag{29}$$

$$\tag{30}$$

The last peculiarity in this model is the measurement based data collection. As we only get global measures for the platform consumption, we can foresee that the base consumptions of every blocks will not be easily distinguishable. We mean here that once the embedded system is put in its laziest state, idle state for example with all possible units powered off, the resulting consumption is considered as a base consumption regrouping the base consumption of every powered peripherals. Obviously, a part of this consumption is static power dissipation. We will call this term $\mathcal{E}_{\mathrm{base}}$, it is important to note that this consumption is reported to the current executed instrustion on the CPU. It can be expressed as in equation (32), as it is dependent on the instruction length l_{insn}. Equation (28) becomes equation (31).

$$\mathcal{E}_{slot} = \mathcal{E}_{\mathrm{base}} + \mathcal{E}_{\mathrm{CPU}} + \sum \mathcal{E}_{\mathrm{bl}} \tag{31}$$

$$\mathcal{E}_{\mathrm{base}} = l_{insn} \times \mathcal{E}_{\mathrm{c\_base}} \tag{32}$$

The CPU and other blocks consumption are then expressed as overhead against the idle state.

As described in equation (26), CPU energy consumption is given by the executed instruction energy cost. This model can be simplified by regrouping instructions in classes as proposed in [12].

As far as other blocks are concerned, we can expand them as bus, memories and other peripherals. This is interesting since bus and memories will be subject to events generated by the processor, such as memory writes. The peripherals will be then modeled by state machines giving the energy consumption of the peripheral during the time slot.

The last step in model construction consists in defining all possible parameters for these components. Due to the limited information available, the developers would not necessarily know the behavior of intra-blocks logic. The parameters for the CPU are already selected, since it is modeled thanks to instructions consumptions. The same can be done for cache, MMU and even co-processors consumptions. The parameters for other blocks are limited to behavioral parameters (UART sending a byte) and their states such as operating mode (running, stopped).

Each energy cost in this model is function of the running frequency and power supply voltage to allow dynamic and frequency scaling capabilities of the platform to be modeled. An example of this is presented in the next section.

4 Measurement results : experimentations on the ARM Integrator/CM 922T-XA10

In this section we will give more details about this methodology, by giving an example of application. As we said before, we applied this methodology on an ARM based platform, ARM Integrator CM922T-XA10. We will explain the architecture exploration procedure, give a few details about the benchmarks implementation and finally give the results of these measures.

4.1 Architecture exploration

As the first step of the model construction consists in defining the model parameters, we have to explore the architecture of our platform to find clues about which events could be the most representatives of the energy consumption. We will give here a rapid overview of the hardware architecture, the interested reader could find for more details about it in [9]

As all Core Module (CM) sold by ARM, this CM is based on an ARM CPU, but this one is a little special since the ARM processor is in fact integrated in a FPGA (Field Programmable Gate Array), an Altera Excalibur EPXA10. With the CPU, few peripherals were integrated in the stripe, which means that they are placed in the chip along the FPGA. On top of that, some peripherals can be implemented in the FPGA, since there are bridges that allow communications between the stripe and the PLD (Programmable Logic Device), *i.e.* the peripherals placed in the FPGA.

The global architecture is depicted on figure 8. As we can see from this figure, the stripe has a two level bus architecture, where the second level could be connected to the PLD. As far as memory is concerned, there are different level of memories, with different access time. The first level is SP (single port) and DP (dual port) SRAM, which are static RAMs connected to the AHB (Advanced High-speed Bus). These RAMs are accessed with only one bus access cycle. Further, we have first level of SDRAM, accessible through the memory controller. This memory is accessed in a longer time since we access to the memory controller through the first level of bus, and then we access to the RAM chips by going out of the EPXA10 chip. A second level of SDRAM or SSRAM could be used by implementing a second memory controller in the FPGA, since SDRAM can be connected thanks to a DIMM connector or a SSRAM chip is integrated on the CM board. This second level will not be mentioned any further.

As we said before, our model is placed at the architectural level. At this level, we found the following blocks :

- CPU
- SP/DP SRAM

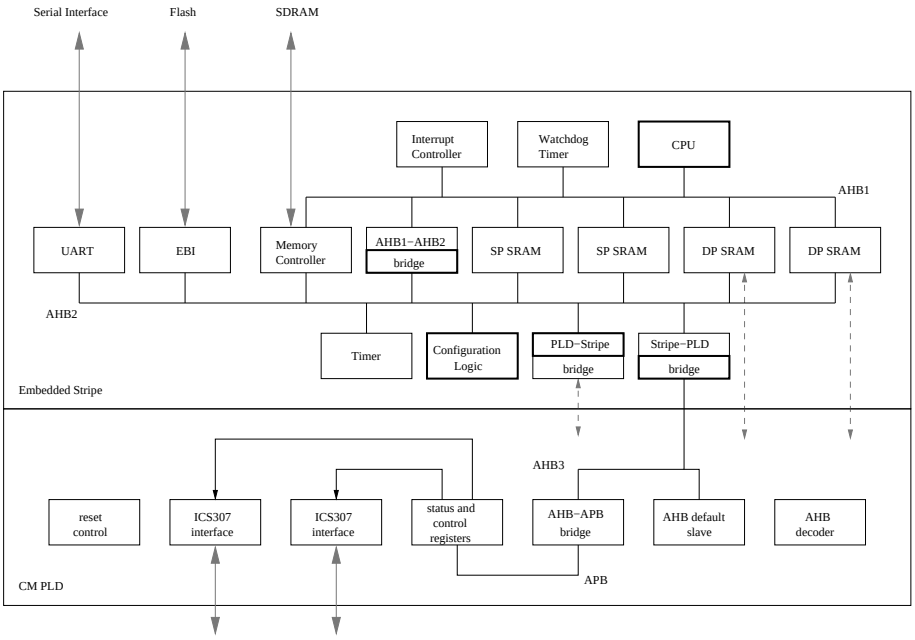

Figure 8: Basic image architecture. Bold blocks are bus masters

- busses (AHB1 and AHB2)
- ITC (Interrupt Controller)
- Timer
- Memory controller and SDRAM

This blocks enumeration gives us clues on the potential parameters of our model. In fact from the software point of view, some of the parameters becomes evidents. In fact, since ARM CPU's have a RISC architecture, all operations are from registers to registers except the load and store operations. In this situation, we can define two classes of instructions : load and store, and the second group all other instructions. To complete the coverage, we should add a third group, the co-processors operations.

Once we have made this first classification, we can easily imagine that the address of the load and stores will define the memory level of the access, and then define if we access to memories or memory mapped registers, through bus level 1 and level 2, and so on.

These different remarks are focused on the CPU and memories. On top of these two classes, we find peripherals. This class of blocks can have an autonomous operating mode. In that case their energy consumption should be modeled by finite state machines, since only access to their control registers can change their state.

The parameters selected after this architecture exploration are the following :

- CPU instructions of class 2.
- load/stores in cache
- bus accesses
- main memory accesses
- Scratch Pad memory access
- peripherals states consumptions

On top of these informations, the architecture exploration reveals that this platform possesses an interesting feature. The PLL (Phase-locked loop) of the clock signal generation is programmable by the mean of a control register. Unfortunately, there is no possibility of adjusting the supply voltage.

4.2 Benchmarks construction

Once every possible parameters are listed, we have to measure their cost. As described before, we build benchmarks for each of the parameters.

The procedure is the same as the one depicted in section 3.1.1, which means that we generate the targeted events a predefined number of occurrence thanks to a loop.

ARM922T, the CPU embedded in the EPXA10, has caches. Theses caches are separated 8kB data and instruction caches. They are 4 way set associative with 8-words wide lines, which means that they have 64 sets of 4 lines. The instruction cache of the ARM 922T allows prefetching. We used this feature in the benchmark implementation. The second point concerns the replacement rule in the associative sets. Two rules are available, random and round robin. As we mentioned in section 3.1.1, the most predictable rule is round-robin, but we kept the possibility of testing the random one by using only one line per set. Then as far as the loop sizing is concerned, we selected a 512 instructions loop body, since it is the size of the direct mapped cache (we are using one line of each set).

Most of the events could not be measured directly, which means that we cannot build a benchmark that makes this event and only this one to occur. By measuring two series of benchmark's consumption, one causing the event and the other not, we can determine the consumption increase due to the event. For example, to estimate the memory access overhead, we should run benchmarks that load or store in memory cached and not cached. Then the CPU executes the same instruction in the two tests, but one generates a memory access and not the other. The difference in energy consumption represents the overhead of a memory access.

Here is a short list of benchmarks that were built, and their target event :

- `insn-XXX` : This benchmark allows us to compare different instructions executed in the CPU. (add, mul, mov, ...). It is only executing the instruction in the loop.
- `Dcache-access` : With this benchmark we can get the ldr/str instruction cost. It is accessing a cached memory address. Then all the access read or writes in a memory cell in the D-cache.
- `AHB1-reg-write` : This one gives the bus access (level 1) overhead. We write in a peripheral register a value that has no effect on the preipheral. The register must be accessible through the first level of bus. For example, we use ITC register (see figure 8).
- `AHB2-access` : Like the previous, this one is giving a bus access (level 2) overhead. In that case, the benchmark is working like the previous one, but with a register accessible through the second level of bus. For example, we use timer register (see figure 8).
- `mem-access` : To get data memory access overhead with this benchmark, we desactivate D-cache. Indeed, all `load` or `store` instructions access the main memory.
- `spm-access` : The aim of this benchmake is to get data scratch pad memory overhead. To have this result, we access scratch pad addresses instead of main memory addresses. The D-cache is desactivated. Every memory access is then made on the SP-SRAM (see figure 8).
- `timer-test` : As an example of peripherals energy characterization, this benchmark allows us to get running/stopped timer consumption. It is subdivided into two benchmarks, one in which the timer is stopped and the second in which the timer is running. The structure of the loop is the same as the `insn-cmp` benchmark with a nop instruction, since it is the instruction generating the less activity.
- `loop-calibration` : Finally, this benchmark is the one which gives loop skeleton overhead. By running an empty loop benchmark, we can estimate the loop overhead.

On top of these various benchmarks, which correspond to most of the parameters events, they all have parameters that allow us to estimate the remaining parameters of the model. Then all benchmarks can be run at different frequencies. As we mentioned before, frequency and voltage scaling are potential parameters of the model, and our tested platform only allows us to modify frequency. However, we can extrapolate our results to a platform with Dynamic Voltage Scaling (DVS) by assuming that the supply voltage can be scaled down in proportion to the clock frequency, which is a reasonable approximation if far away from the threshold voltage.

4.3 Measures and model adaptation

bench name	length	energy (nJ)	error_meas (pJ)	error_stdev (pJ)	p_mean (W)
loop-calibration	4	69.084	5.1777	20.338	3.421
insn-nop	1	16.747	1.2884	6.1914	3.317
insn-add	1	16.762	1.2889	3.0943	3.319
insn-and	1	16.769	1.2892	6.2788	3.32
insn-mov	1	16.76	1.2889	5.1092	3.319
insn-lsr	1	16.764	1.289	6.4375	3.32
insn-mul	3	50.378	3.8494	20.599	3.327
AHB1-access	6	101.33	7.7132	154	3.347
AHB2-access	18	300	22.998	542.9	3.304
timer-test_on(nop)	1	16.754	1.285	27.298	3.298
timer-test_off(nop)	1	16.732	1.2857	6.0922	3.3
Dcache-access_ldr	1	17.146	1.3007	5.942	3.397
Dcache-access_str	2	34.341	2.603	13.381	3.4
mem-access_ldr	40	775.44	54.551	171.04	3.818
mem-access_str	28	543	38.296	231.29	3.801
spm-access_ldr	8	131.72	10.168	38.48	3.259
spm-access_str	7	115.37	8.9021	21.916	3.263

Table 1: Results of benchmarks

bench name	energy w/ prefetch (nJ)	energy wo/ prefetch (nJ)
loop-calibration	69.084	67.925
insn-nop	16.747	17.309
insn-add	16.762	17.307
insn-and	16.769	17.315
insn-mov	16.76	17.307
insn-lsr	16.764	17.314
insn-mul	50.378	51.967
AHB1-access	101.33	101.49
AHB2-access	300	300.45
Dcache-access_ldr	17.146	17.66
Dcache-access_str	34.341	35.364
mem-access_ldr	775.44	775.29
mem-access_str	543	543.08
spm-access_ldr	131.72	135.82
spm-access_str	115.37	118.79

Table 2: Results of benchmarks : influence of prefetch

The results summarized in table 1 allows us to draw the following conclusions. First of all, we can conclude that the type of instruction does not matter as far as it is not a data access or it does not misses in instruction cache. This conclusion is underlined by the results of `insn-XXX` benchmark. We should note

bench name	length	energy (nJ)	error_meas (pJ)	error_stdev (pJ)	p_mean (W)
loop-calibration	4	69.084	5.1777	20.338	3.421
loop-calibration-H	4	114.08	9.6173	162.21	2.822
loop-calibration-Q	4	201.4	18.418	211.53	2.49
loop-calibration-E	4	376.22	36.029	930.49	2.329
loop-calibration-S	4	724.58	71.213	450.41	2.244
insn-nop	1	16.747	1.2884	6.1914	3.317
insn-nop-H	1	27.422	2.3924	24.747	2.717
insn-nop-Q	1	48.638	4.5942	44.488	2.407
insn-nop-E	1	90.792	8.9897	87.111	2.247
insn-nop-S	1	174.93	17.776	115.9	2.164
insn-mul	3	50.378	3.8494	20.599	3.327
insn-mul-H	3	82.482	7.1462	97.676	2.723
insn-mul-Q	3	145.81	13.707	145.18	2.405
insn-mul-E	3	272.07	26.818	245.46	2.245
insn-mul-S	3	524.32	53.034	120.3	2.162
AHB1-access	6	101.33	7.7132	154	3.347
AHB1-access-H	6	169.97	14.426	161.89	2.803
AHB1-access-Q	6	303.26	27.733	389.56	2.5
AHB1-access-E	6	568.79	54.319	390.06	2.347
AHB1-access-S	6	1098.2	107.44	774.36	2.263
AHB2-access	18	300	22.998	542.9	3.304
AHB2-access-H	18	505.74	43.068	522.74	2.781
AHB2-access-Q	18	905.39	82.903	823.1	2.489
AHB2-access-E	18	1702.8	162.52	3224.9	2.337
AHB2-access-S	18	3289.2	321.5	1203.4	2.26

Table 3: Results of benchmarks at different frequencies : H means half speed, Q quarter, E eighth and S sixteenth. These ratio are given according to reference clock speed of 198MHz

that there is a special case, the multiplication. In fact it is not a one cycle operation, its consumption is then a strict multiple of the instruction cost. The ratio is the multiplication length, between 3 and 6 (depending on the multiplier).

As we mentioned before, we used prefetch to overcome the instruction cache loading bias on measures. As table 2 shows, cache loading bias is about 1% of the energy figures. The difference between the benchmark using prefetch and the ones not using it is due to the cache misses induced by this cache loading. More precisely the loop body is 512 instructions long and the cache line is 8 words wide. We have 64 cache misses generating an amount of energy divided by the number of event. This quantity explains the difference underlined before.

The second remarkable fact is that reducing platform frequency increases energy consumption of event. We believe the explanation as follows. Since we cannot reduce the power supply voltage of the circuit and since the dynamic activity is the same, the event energy consumption is at least equal. It can be higher due to static consumption which increases since the event takes more time.

By finally repeating the exact same experiment a large number of time, we can observe that the variation in the results is higher than the measurement error. This variation is due to various possible factors such has previous activity on the platform, temperature of the chip, ...

4.4 Resulting model

4.4.1 Basic model

The basic model presented in section 3.3 can be rewritten, by using models simplifications obtained by calibration.

We found that most instructions have the same energy consumptions as long as they stay inside the CPU. Currently only ARM32 instruction set is modeled. Thumbs (16bit) instruction set could be modeled using them same benchmark methodology in no time. In our setup, it is not possible to isolate the instruction cache consumption, which is lumped with the instruction consumption. Cache misses will then be modeled as simple memory accesses.

We finally have a model where CPU instructions are regrouped in two classes, the logical and integer intra-CPU instructions, and the load and store instructions. A memory load access is modeled as a load instruction, plus a bus overhead, plus a memory overhead. Last but not least, the peripherals energy consumption are taken into account thanks to state machines that give their consumption during the instructions execution.

Other model simplifications are possible in the case of this platform. For example, the CPU cache models are simplified by taking into account only memory access bursts in case of misses since the overhead can be neglected. The MMU has the same kind of simplification, since the TLB (Translation Look aside Buffer) misses generate memory accesses, and the table walk represents only a negligible amount of energy.

This model is finally resumed by Equation (33).

$$\begin{aligned} \mathcal{E}_{\text{slot}} &= \mathcal{E}_{\text{base}} \\ &+ \mathcal{E}_{\text{insn}} + \mathcal{E}_{\text{bus\_access}} + \mathcal{E}_{\text{mem}} \\ &+ \sum_{periph} \mathcal{E}_{\text{periph\_state}} \end{aligned} \tag{33}$$

where $\mathcal{E}_{\text{slot}}$ is the energy consumption of the instruction execution time slot, $\mathcal{E}_{\text{insn}}$ is the cost of instruction given by its class cost, $\mathcal{E}_{\text{bus\_access}}$ is the bus overhead cost for load or store instructions, $\mathcal{E}_{\text{mem}}$ is the overhead for memory accesses. The last term represents the sum of the energy overhead of peripherals state. These cost are all overhead costs, since the full consumption of a peripheral cost, for example, is given by its base energy cost comprised in $\mathcal{E}_{\text{base}}$ and the overhead.

4.5 Advanced information extraction

As we stated before, the Integrator/CM has no dynamic voltage scaling (DVS) capabilities, hence when we reduce the frequency we cannot decrease energy consumption.

4.5.1 Frequency scaling

Let us have a look at what is happening. First, when we reduce the frequency, it should be underlined that the modified frequency domain contains only the stripe's peripherals and CPU. Our clock frequency reduction does not affects the SDRAM modules or the FPGA.

Figure 9 depicts the measured values for three different benchmarks reproduced at 5 different frequencies. This graph represents energy consumption per event, in that case instruction, against clock divisor, $r_f = \frac{f_{ref}}{f}$ where f_{ref} is the nominal frequency in our case 198 MHz.

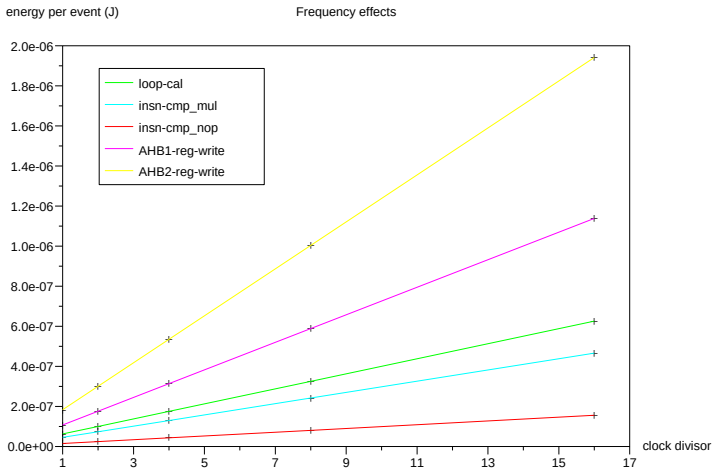

Figure 9: Multiple frequencies experiments : This figure shows that the energy per event increases linearly against frequency ratio.

These curves show that energy per event increases when frequency is decreased, and this may seem counter-intuitive. To understand these results observe first that a given event, *e.g.* the execution of some specific instruction, entails an almost constant number of bit flips, and that each flip uses a fixed amount of energy. Hence, to a first approximation, and in the absence of voltage scaling, the energy for a given event should be a constant. However, in our platform, frequency scaling acts only on the processor and Excalibur embedded peripherals; the consumption of other peripherals, external memories and FPGA is not affected. Hence, the addition of a parasitic term which is roughly proportional to the duration of the event or inversely proportional to frequency. This is clearly the case for the curves of Fig. 9.

$$\mathcal{E}_{evt} = \mathcal{E}_{\mathrm{rp}} + \mathcal{E}_{\mathrm{mc}} \tag{34}$$

Where $\mathcal{E}_{\mathrm{evt}}$ is the per event energy cost as measured before. $\mathcal{E}_{\mathrm{mc}}$ is the energy consumed by the modified clock domain of the platform. The $\mathcal{E}_{\mathrm{rp}}$ term is the base energy of the remaining part of the platform.

The event triggered by these five benchmarks stays in the frequency domain of the stripe. `loop-calibration`, `insn-mul` and `insn-nop` instructions are CPU only instructions and AHB 1 and 2 register read access only to the two busses, whose clocks are the CPU clock for AHB1 and half the CPU clock for the AHB 2. This information is important since in these benchmarks the same activity is generated in the stripe but at different frequencies. If we apply the power model of VLSI presented in section 2.2.3 (equation (4)), in our case the power supply voltage is constant and the activity denoted by α too.

$$P = \frac{1}{2} C V^2 \alpha f \tag{35}$$

In terms of energy :

$$\begin{aligned}\mathcal{E}_{\mathrm{mc}} &= P_{evt}t_{evt} \\ &= \frac{1}{2}CV^2\alpha f_{ref}t_{evt}^{ref} \\ &= \frac{1}{2}CV^2\alpha l_{\mathrm{insn}}\end{aligned}$$

In terms of time, if we divide the frequency by two, we multiply the time of the event by two. Then the product ft_{evt} is constant. Thus all members of the previous formula are constant. We can state that the real energy drawn by the event is constant against frequency. This energy is the lined square on figure 10.

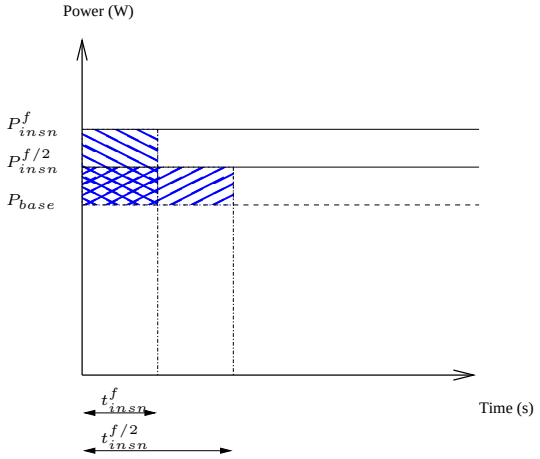

Figure 10: energy consumption

This figure drives us to modify the last equation to take into account the variation of $\mathcal{E}_{rp}$ against frequency divisor, or time. If the "base power" is constant against clock frequency variation on a same benchmark, we can consider the following equation :

$$\mathcal{E}_{\mathrm{evt}} = \mathcal{E}_{rp\_base}r_f + E_{mc} \tag{36}$$

Where $\mathcal{E}_{rp\_base}$ is the base energy, which means that it is the base energy consumed during one event at full speed.

This final formula fits perfectly to the linear character of the experimental results. The last fact which is not explained by this last equation is the difference in the slopes of the three lines. To explain these slopes, we should have a closer look at the benchmarks characteristics and more precisely on event lengths. In the first benchmark, the nop one, the length of the event is 1 cycle. In the second one, AHB1 register access, the length is 7 cycles and finally the AHB2 register access length is 12 cycles.

If the $\mathcal{E}_{base}$ is really invariant against frequency and represents energy consumption of the remaining part of the platform, the elementary base energy cost presented before should be 7 times higher in the second benchmark than in the first, and 12 times higher in the third benchmark. This remark drives us to rewrite the equation like this :

$$\mathcal{E}_{evt} = \mathcal{E}_{\mathrm{rp\_base}} \times l_{evt} \times r_f + \mathcal{E}_{mc} \tag{37}$$

where E_{base}^{cycle} is the base energy consumed by the remaining part of the platform in one full speed CPU cycle. The CPU cycle is the unit of time since it is the shortest event which can be measured and also which can happen on the platform. l_{evt} is the event length in terms of full speed CPU cycles.

Linear regressions on the previous results gives the following results :

These results show that equation (37) gives a good explanation for the experiments on clock frequency variation. These results gives us an estimation of what we can consider as base energy, which is not

Benchmark name	$\mathcal{E}_{rp\_base}$ (nJ)	$\mathcal{E}_{mc}$ (nJ)	error (pJ)
insn-mul	10.91	26.37	572.36
loop-calibration	10.52	19.22	258.90
insn-nop	10.54	6.35	105.61
access-AHB1	11.06	36.72	1085.37
access-AHB2	11.06	106.32	3431.46

Table 4: Linear regression results

changing against software execution. The value obtained is about 10.82 nJ (with a standard deviation of $\pm 2.6\,10^{-2}$) per CPU clock cycle. This previous value can be used to re-interpret benchmarks results. We only presented $\mathcal{E}_{\text{evt}}$ values in table 1. But these values give an idea of the cost in energy of the events, but they are biased by the large influence of $\mathcal{E}_{rp\_base} \times l_{evt}$. The new results obtained by subtracting this base energy are shown in table 5.

bench name	energy (nJ)	eff_energy (nJ)	error_meas (pJ)	error_stdev (pJ)
loop-calibration	69.084	26.442	5.1777	20.338
insn-nop	16.747	6.086	1.2884	6.1914
insn-nop-H	27.422	6.1012	2.3924	24.747
insn-nop-Q	48.638	5.9956	4.5942	44.488
insn-nop-E	90.792	5.5071	8.9897	87.111
insn-nop-S	174.93	4.3556	17.776	115.9
insn-add	16.762	6.1009	1.2889	3.0943
insn-and	16.769	6.1088	1.2892	6.2788
insn-mov	16.76	6.0992	1.2889	5.1092
insn-lsr	16.764	6.1032	1.289	6.4375
insn-mul	50.378	18.396	3.8494	20.599
insn-mul-H	82.482	18.518	7.1462	97.676
insn-mul-Q	145.81	17.884	13.707	145.18
insn-mul-E	272.07	16.211	26.818	245.46
insn-mul-S	524.32	12.614	53.034	120.3
AHB1-access	101.33	37.362	7.7132	154
AHB2-access	300	108.11	22.998	542.9
timer-test_on(nop)	16.754	5.9714	1.285	27.298
timer-test_off(nop)	16.732	5.9937	1.2857	6.0922
Dcache-access_ldr	17.146	6.4852	1.3007	5.942
Dcache-access_str	34.341	13.02	2.603	13.381
mem-access_ldr	775.44	349.02	54.551	171.04
mem-access_str	543	244.5	38.296	231.29
spm-access_ldr	131.72	46.436	10.168	38.48
spm-access_str	115.37	40.746	8.9021	21.916

Table 5: Effective results

4.5.2 DVS extension

In this section we present an hypothetical extension of the previous model for a DVS enabled platform. The effective energy of different classes events presented before as $\mathcal{E}_{\text{evt}}$ are approximated be the basis dynamic power model (equation (36)).

As we saw before the frequency inffluence on these part is null. This is the reason why frequency scaling has no effect on energy consumption in our experiments. But if we introduce the fact that V_{dd} can be adjusted this not true any more. We take the assumption that if we divide the frequency by r we can divide V_{dd} by an $r_{\text{v}} = \frac{V_{dd}^{ref}}{V_{dd}}$ amount depending on r_f. For example, Siminuc *et al.* prove experimentally

that the relation between the voltage and frequency of their StrongARM SA1100 can be approximated by: $\frac{1}{r_{\mathrm{v}}} = 0.66\frac{1}{r} + 0.33$. In that situation the $\mathcal{E}_{\mathrm{evt}}$ is modified and expressed like this for instruction execution for example:

$$\mathcal{E}_{\mathrm{insn}} = \frac{1}{2} C \frac{V_{\mathrm{dd}}^2}{r_{\mathrm{v}}^2} \alpha l_{\mathrm{insn}} \tag{38}$$

The benefit is then of $\frac{1}{r_{\mathrm{v}}^2}$. In case we approximate the relation between voltage and frequency in our platform, by the one given earlier, we would have a benefit of $(0.66\frac{1}{r} + 0.33)^2$. For a clock ratio of 2, half the speed, the energy benefit would be of 56 % less consumption. This relation can be applied on the modified clock domain, and thus a NOP instruction would have cost 2.76 nJ instead of 6.35. The base energy would not be affected.

Finally, concerning the base energy term. The $\mathcal{E}_{rp\_base}$ could also be modified by DVS, but in our case we have no mean to reduce the frequency, then it has no interest. The E_{peri}^{over} can be however modified by the same amount than the previous one, since the peripheral are also in the modified clock domain. Voltage scaling would have been of great interest, but our platform was not designed for it.

5 Model validation

5.1 Model validation

To check the accuracy of the model thus built for the ARM Integrator CM922T-XA10, we describe here our accuracy tests experiments. The model were implemented in a simulator, and its results were compared to physical measurements.

Simulator integration

Our model is implemented in a simulation tool suite. This simulation tools are composed of two simulator.

The first is a complete platform functional simulator in charge of generating a cycle-accurate execution trace of the software. This trace reports all executed instructions, and all peripherals activities (state changes). This first step allow software developers to functionally debug their applications and supply them the material to make the second step simulation. To fulfill this step task, we implemented the behaviour of the Integrator platform in the open source simulator `skyeye`. We also upgraded it to the cycle accurate trace generation.

The second step is energy simulation tool proper. This simulator implements the model presented in the previous section. Its main task is to compute model parameters from the cycle-accurate execution trace. It accumulates all computed energies, and reports them in an energy profile file. The format of this file is an emerging file format, which can be visualized thanks to the open source project `KCacheGrind`. This simulation step allows to get the overall consumption of the software 'run', figures we will use in the next step of this validation.

Validation methodology

To check the accuracy of the resulting model, we propose to compare the consumption estimation of the model, thus implemented in our tool to physical measurement on the real platform.

The test application chosen for this model validation are widely spread multimedia applications : JPEG, JPEG2000 and MPEG2. The implementations of these three applications are Linux standard libraries. Hence they use operating system services and standard libc functions. All experiments could have been made with Linux (or even uClinux), since the simulation tools are complete enough to run these operating systems. For limited measurement duration reasons, we decided to replace these heavy OS by the lightweight one, Mutek [15]. Linux hardware layer abstraction makes interrupt request managment too long to allow a reasonable sized image to be decoded in our measure time window.

The three applications are executed in the simulation tools to get model estimations of their executions. As far as the measurement setup is concerned, we kept the same setup as the one used for model calibration, presented in section **??**.

Accuracy

		Measured values		Simulated values		Error	
Bench-name	code lines	cycles	energy (J)	cycles	energy (J)	cycles (%)	energy (%)
jpeg	25819	6916836	1.142440e-01	6607531	1.037940e-01	- 4.4	- 9.1
jpeg2k	4686	7492173	1.268535e-01	7663016	1.200488e-01	+ 2.2	- 5.3
mpeg2	24657	13990961	2.335522e-01	14387358	2.208065e-01	+ 2.8	- 5.4

Table 6: Simulators results: the results obtained for execution time and energy consumption by real hardware measurement are shown in second and third columns, the simulation ones in fourth and fifth columns. The last two columns give the error percentile of the simulation.

Results of model estimations and physical measurements are presented in table 6. The second column gives an idea of the application code complexity, by giving the total number of source code lines. These figures do not integrate the operating system source code.

The third and fourth columns reports the physical measurement results, in terms of execution duration in CPU clock cycles and in terms of energy consumption in Joules. Fifth and sixth columns gives the same kind of informations concerning the simulation results. Finally, the last two columns gives the percentile error of simulation errors of the simulation results against the physical measurement on the target hardware platform.

These results show that a 10% error rate can be achieve by our simple complete platform energy model. This estimations are obtained in roughly less than a minute (25s for the first simulation plus 20s for the second). We think that the error rate of 10% is largely acceptable in regard of the simulation time.

6 Conclusion and Future Works

In this report we have explained how an accurate energy consumption model for a full embedded system can be build from external measurements and micro-benchmarks. Our methodology is made for systems using fixed architectures (as opposed to codesign based developments) for which a real hardware platform is available. Quantitative energy data are gathered at the battery power input such that total system consumption can be estimated. Measurements are made in a non-intrusive manner (without hardware modification) so as to reduce electronic equipment and skills needed to perform the acquisition. Most of the time used to setting the measurement up is spent in writing the micro benchmark code used to activate different part of the hardware.

The resulting model is thus driven by the activity generated by the embedded software that is run on the platform. Quantitative values obtained during the benchmark tests can be used in a number of ways: raw consumption can be used in software platform simulators to estimate software energy cost, differences between access at different levels of the memory hierarchy can be used to control and calibrate tradeoffs used in compilers during high level transformations such as loop optimizations for data locality.

Based on our experiments, we were able to build an accurate energy model for our test platform. The resulting model is simple enough to be used efficiently in fast software platform simulators. Consumption data clearly identify computation operations, memory accesses at different levels of the memory hierarchy, bus and peripheral activity.

Our aim is to derive from these figures the tradeoffs involved in using techniques such as software caches or scratch pad memories for software development. These tradeoffs will be used to build a dynamic model of local memory usage to couple compilation techniques and operating systems services. This will be integrated to efficiently use scratch pad memories and shared low energy resources for multi-tasking software development in embedded systems.

Future works will provide energy data input tables that can be used along with simulators such as SimpleScalar [2]. Other work will provide extensions to this methodology in order to support advanced energy consumption optimization techniques such as digital voltage scaling available on most recent platforms.

References

[1] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture ISCA*, 2000.

[2] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[3] Naehyuck Chang and Kwanho Kim. Real-time per-cycle energy consumption measurement of digital systems. *IEE Electronics Letters*, 36(13):1169–1171, 2000.

[4] Rita Yu Chen, Mary Jane Irwin, and Raminder S. Bajwa. Architecture-level power estimation and design experiments. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, volume 6, pages 50–66, January 2001.

[5] Rita Yu Chen, Robert M. Owens, Mary Jane Irwin, and Raminder S. Bajwa. Validation of an architectural level power analysis technique. In *Proceedings of the 35th annual conference on Design automation*, 1998.

[6] R.Y. Chen, M.J. Irwin, and R.S. Bajwa. An architectural level power estimator. In *Proceedings of the Power-Driven Microarchitecture Workshop*, 1998.

[7] Mentor Graphics Corporation, 1999.

[8] Synopsys Corporation. Powermill data sheet, 1999.

[9] N. Fournel, A. Fraboulet, and P. Feautrier. Porting the mutek operating system to arm platforms. Technical report, LIP - ENS Lyon, 2006.

[10] Nam Sung Kim, Todd Austin, Trevor Mudge, and Dirk Grunwald. *Power Aware Computing*, chapter Challenges for Architectural Level Power Modeling. Kluwer Academic, 2001.

[11] Ram K. Krishnamurthy. *Mixed Swing Techniques for Low Energy/Operation Datapath Circuits*. PhD thesis, Carnegie Mellon University, 1997.

[12] Mike Tien-Chien Lee, Masahiro Fujita, Vivek Tiwari, and Sharad Malik. Power analysis and minimization techniques for embedded dsp software. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1997.

[13] Sheayun Lee, Andreas Ermedahl, and Sang Lyul Min. An accurate instruction-level energy consumption model for embedded risc processors. *ACM SIGPLAN Notices*, 2001.

[14] Yanbing Li and Jög Henkel. A framework for estimating and minimizing energy dissipation of embedded HW/SW systems. In *35th Conference on Design Automation Conference (DAC'98)*, pages 188–193, 1998.

[15] Mutek Operating System. Disydent web site. Available online, http://www-asim.lip6.fr/recherche/disydent/, October 2005.

[16] S. Nikolaidis and T. Laopoulos. Instruction-level power consumption estimation embedded processors low-power applications. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, International Workshop on*, pages 139–142, 2001.

[17] D. J. Pursley. *A gate level simulator for power consumption analysis*. PhD thesis, Carnegie Mellon University, May 1996.

[18] J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *International Conference on Computer Design*, October 1998.

[19] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, 2001.

[20] 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 (VLSI) Systems*, 1994.

[21] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 1996.

[22] W. Ye, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. The design and use of simplepower: A cycle-accurate energy estimation tool. In *Proceedings of the Annual ACM IEEE Design Automation Conference*, 2000.

A Complete results list

bench name	energy (nJ)	eff_energy (nJ)	error_meas (pJ)	error_stdev (pJ)
loop-calibration	69.084	26.442	5.1777	20.338
loop-calibration-H	114.08	28.79	9.6173	162.21
loop-calibration-Q	201.4	30.825	18.418	211.53
loop-calibration-E	376.22	35.079	36.029	930.49
loop-calibration-S	724.58	42.298	71.213	450.41
loop-calibration	67.925	25.283	5.1421	128.93
insn-nop	16.747	6.086	1.2884	6.1914
insn-add	16.762	6.1009	1.2889	3.0943
insn-and	16.769	6.1088	1.2892	6.2788
insn-mov	16.76	6.0992	1.2889	5.1092
insn-lsr	16.764	6.1032	1.289	6.4375
insn-mul	50.378	18.396	3.8494	20.599
insn-nop	17.309	6.6485	1.3058	6.584
insn-add	17.307	6.6462	1.3058	8.0992
insn-and	17.315	6.6548	1.306	6.9669
insn-mov	17.307	6.6462	1.3057	3.7529
insn-lsr	17.314	6.6532	1.306	5.0697
insn-mul	51.967	19.985	3.8987	8.9715
insn-mul-H	82.482	18.518	7.1462	97.676
insn-mul-Q	145.81	17.884	13.707	145.18
insn-mul-E	272.07	16.211	26.818	245.46
insn-mul-S	524.32	12.614	53.034	120.3
insn-nop-H	27.422	6.1012	2.3924	24.747
insn-nop-Q	48.638	5.9956	4.5942	44.488
insn-nop-E	90.792	5.5071	8.9897	87.111
insn-nop-S	174.93	4.3556	17.776	115.9

Table 7: Effective results (part 1)

bench name	energy (nJ)	eff_ energy (nJ)	error_meas (pJ)	error_stdev (pJ)
AHB1-access	101.33	37.362	7.7132	154
AHB1-access	101.49	37.528	7.7194	24.95
AHB1-access-H	169.97	42.039	14.426	161.89
AHB1-access-Q	303.26	47.402	27.733	389.56
AHB1-access-E	568.79	57.077	54.319	390.06
AHB1-access-S	1098.2	74.733	107.44	774.36
AHB2-access	300	108.11	22.998	542.9
AHB2-access	300.45	108.56	22.99	235.33
AHB2-access-H	505.74	121.96	43.068	522.74
AHB2-access-Q	905.39	137.83	82.903	823.1
AHB2-access-E	1702.8	167.69	162.52	3224.9
AHB2-access-S	3289.2	218.97	321.5	1203.4
timer-test_on(nop)	16.754	5.9714	1.285	27.298
timer-test_off(nop)	16.732	5.9937	1.2857	6.0922
Dcache-access_ldr	17.146	6.4852	1.3007	5.942
Dcache-access_str	34.341	13.02	2.603	13.381
Dcache-access_ldr	17.66	6.9996	1.3167	3.801
Dcache-access_str	35.364	14.043	2.6348	12.459
mem-access_ldr	775.44	349.02	54.551	171.04
mem-access_str	543	244.5	38.296	231.29
mem-access_ldr	775.29	348.87	54.553	87.694
mem-access_str	543.08	244.58	38.308	132.05
spm-access_ldr	131.72	46.436	10.168	38.48
spm-access_str	115.37	40.746	8.9021	21.916
spm-access_ldr	135.82	50.53	10.295	32.952
spm-access_str	118.79	44.162	9.0074	19.565

Table 8: Effective results (part 2)