Embedded software energy characterization: Using non-intrusive measures for application source code annotation

Nicolas Fournel, Antoine Fraboulet, Paul Feautrier

HAL Id: hal-00391186

https://hal.archives-ouvertes.fr/hal-00391186

Submitted on 3 Jun 2009

Embedded Software Energy Characterization: using non-intrusive measures for application source code annotation

Nicolas Fournel[†] and Antoine Fraboulet[‡] and Paul Feautrier[†]
[†]INRIA/Compsys ENS de Lyon/LIP, Lyon F-69364 France
[‡]INRIA/Compsys INSA-Lyon/CITI, Villeurbanne F-69621 France

Abstract

In this paper we propose a complete system energy model based on non-intrusive measurements. This model aims at being integrated in fast cycle accurate simulation tools to give energy consumption feedback for embedded systems software programming. Estimations take into account the whole system consumption including peripherals. Experiments on a complex ARM9 platform show that our model estimates are in error by less than 10% from real system consumption, which is precise enough for source code application design, while simulation speed remains fast.

1 Introduction

With present day technologies, it is possible to build very small platforms with enormous processing power. However, physical laws dictate that high processing power is linked to high energy consumption. Embedded platforms are mostly used in hand-held appliances, and since battery capacity does not increase at the same pace as clock frequency, designers are faced with the problem of minimizing power requirements under performance constraints.

The first approach is the devising of low-energy hardware technologies, but this is outside the scope of this paper. The second approach is to make the best possible use of the available energy *e.g.* by adjusting the processing power to the instantaneous needs of the application, and by shutting down unused parts of the system. These tasks can be delegated to the hardware. However it is well known that the hardware only source of knowledge for scheduling is the past of the application; only software can anticipate future needs. Energy can also be minimized as a side effect of performance optimization. For instance, replacing a conventional Fourier transform by an FFT greatly improves the energy budget; the same can be said of data locality optimization, which aims at replacing costly main memory accesses by low-power cache accesses.

The ultimate judge in the matter of energy consumption is measurement of the finished product. However, software designers, compilers and operating

systems need handier methods for assessing the quality of their designs and directing possible improvements. Hence designers need simple models which must be expressed in term of software visible events like instructions, cache hits and misses, peripheral activity and the like. There are several ways of constructing such models. One possibility is electrical simulation of the design; this method is too time-consuming for use on systems of realistic size. Another method is to interpolate/extrapolate from measurements on a prototype. This is the method we have applied in this work.

The paper is organized as follows. After reviewing state of the art techniques in section 2 we present in section 3 a methodology to build complete platform energy consumption models for software development. Section 4 presents the resulting model for an ARM9 development platform. This section also validates our model on more significant pieces of code, multimedia applications, thanks to its implementation in a fast and cycle accurate simulation tool. We then conclude and discuss future work.

2   Related Works

Many works focus on energy characterization of VLSI circuits. They can be organized using two main criteria: their level of hardware abstraction and the calibration method. For the first criterion, we can group the models in three main categories which are, by increasing level of abstraction, transistor/gate level models, architectural level models and finally instruction level models. Among these models there are usually three methods for building consumption models. The first method is analytical construction, the second one is simulation based, and the third is based on physical measurements.

In transistor (gate) level models, all transistor (gate) state changes are computed to give an energy consumption approximation for a VLSI component. This method is highly accurate, but a complete description of the component is needed. Models built at this level of abstraction are generally reserved to hardware designers and are very long to simulate.

Upper level of abstraction, architectural or RTL level, models the system using functional units. Each unit can be represented by a specific model adapted to its internal structure.(*e.g.* bit-dependent or bit-independent models for Chen *et al.* [2]). To be more accurate, some works like Kim *et al.* [8], subdivide the block into sub-blocks to apply different models on each sub-block. This set of models allows to extend models to a complete platform.

The highest level is instruction/system level of abstraction. At this level, models are based on events such as instructions execution ([19, 10, 13]). Tiwari *et al.* in [19] propose to characterize hardware activity down to the inter-instructions energy consumption, which represents the logic switching activity between two different instructions. Others works also take into account the logic switching due to data parameters [17]. Systems considered in these models are generally composed of CPU, bus and memory. Only few works focus on modeling a complete platform with peripherals. Among them, EMSIM [18] is

a simulator based on Simunic *et al.* [15] which includes the StrongARM SA110 energy characterization. This simulator poorly characterizes the peripherals and focuses on the microprocessor. The SoftWatt simulator proposed in [6] uses a complete system simulator based on SimOS to monitor a complex operating system and interactions among CPU, memory hierarchy and hard disk operations. Their simulator is modified to include analytical energy models and the output reports kernel and user mode energy consumption up to the operating system services. Data are sampled during simulation and dumped to log files at a coarser granularity than cycle level, leading to *average* power consumption figures. The closest work to ours, AEON's model [9], is a complete platform energy consumption model based on measurement and uses simulator internal energy counters. The targeted platform is an 8 bit AVR micro-controller based sensor network node that does not include CPU pipelines, complex memory hierarchy and peripherals. Our model allows the simulation of much more complex hardware architectures while being independant of simulator internals. Some other works, like SoftExplorer [14], even propose to model energy consumption by relying on higher parameters like algorithmic parameters, or application implementation parameters like variable size in Mutterja *et al.* [11].

As far as calibration methods are concerned, analytical models are usually based on manufacturers data, *e.g.* in Simunic *et al.* [15] the model is built thanks to datasheet informations. Simulation based calibration needs a full knowledge of the underlying level architecture, which means that it needs a description of low level hardware (VHDL, or Verilog descriptions). Measurement based methods only need few informations on the hardware and works like [19, 3] have shown that it is possible to extract internal unit consumption from system measurements.

In this paper we propose a methodology for complete platform energy consumption model construction based on simple and non-intrusive measurements. The model is built at a level of abstraction close to the system level presented before, but is extended to the complete platform by coupling it with architectural level principles presented by Kim *et al.* in [8]. We also take peripherals energy models and dynamic frequency and voltage scaling into account. Our model is geared toward user feedback at the application source level.

3   Model construction basics

We present in this section our methodology to build complete platform models. We first give more details on the structure and the parameters of the energy model. Section 4 will present the target dependent model parameters through a case study on an ARM9 based platform.

3.1   Model structure and parameters

Our choice among all the modeling methods which have been presented in Sect. 2 is to build an architectural level model, in which the system is divided into its

main functional blocks at the platform based level such as CPU, interconnection bus, memory hierarchy, peripherals... The energy consumption of an application E_{app} is obtained be adding all blocks consumptions E_{bl}.

Each block can have its own energy consumption model. To have a platform model better suited for software development, we use instruction level abstraction for CPU. CPU energy consumption E_{CPU} is described in equation (1).

$$E_{\mathrm{CPU}} = E_{\mathrm{insn}} + E_{\mathrm{cache}} + E_{\mathrm{MMU}} \tag{1}$$

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

$$E_{\mathrm{app}} = E_{\mathrm{CPU}} + \sum E_{\mathrm{bl}} \tag{2}$$

This model aims at being integrated in a full platform cycle accurate simulation tool. The most interesting way of writing the model for this kind of purpose is to define a time slot energy consumption. The chosen time slot is the CPU instruction execution. There are two reasons 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 platforms. Secondly, interrupt requests, the only mean for the hardware peripherals to interact with the software, are managed at the end of instructions execution. From a software point of view, there is no need to use a finer time reference to report hardware events more precisely. From a precision point of view we do not model inter-instruction switch activity nor data dependent. These parameters can be ignored at this level of abstraction as we will see in section 4.4.

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 the consumption at instruction level of granularity will be noted as $\mathcal{E}_*$. This new model formula is expressed in the following equation:

$$\mathcal{E}_{\mathrm{slot}} = \mathcal{E}_{\mathrm{CPU}} + \sum_{blocks} \mathcal{E}_{\mathrm{bl}} \tag{3}$$

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 each block 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 instruction on the CPU. It can be expressed as in equation (5), as it is dependent on the instruction length l_{insn} in terms of

clock cycles. Equation (3) becomes equation (4).

$$\mathcal{E}_{slot} = \mathcal{E}_{\text{base}} + \mathcal{E}_{\text{CPU}} + \sum \mathcal{E}_{\text{bl}} \qquad (4)$$

$$\mathcal{E}_{\text{base}} = l_{insn} \times \mathcal{E}_{\text{c\_base}} \qquad (5)$$

The CPU and other blocks consumption are then expressed as overhead against the idle state. As described in equation (1), CPU energy consumption is given by the executed instruction energy cost. This model can be simplified by regrouping instructions in classes as proposed in [10]. 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 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, developers do 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 modes.

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   Model Construction Case Study

In this section we propose an extended example of our methodology application. This methodology was applied on a ARM based development board. This platform uses an ARM922T and usual embedded systems peripherals (*e.g.* UART, Timers, network interface) on the same chip. Our hardware architecture exploration shows that the platform has three distinct levels of memory, a cache, a scratchpad and main memory. All peripherals are accessible through two levels of AMBA bus. We will give details about the energy consumption model construction for this platform, then we will check the accuracy of the resulting model.

4.1   Methodology application

The complete platform modeling method presented in section 3.1 is applied on our ARM9 platform in this section. The measurement setup used for these experiments is close to the one depicted in [13]. We used a digitalizing oscilloscope, the shunt resistor is replaced by a current probe and we also used a voltage probe.

Calibration benchmarks. We built benchmarks to calibrate our model, more precisely our block models. The hardware exploration gives us the main blocks

Table 1: Benchmarks results for simple operation energy calibration

bench name	length	energy (nJ)	error (pJ)
loop-calibration	4	69.084	5.1777
insn-nop	1	16.747	1.2884
AHB1-access	6	101.33	7.7132
AHB2-access	18	300	22.998
Dcache-access	1	17.146	1.3007
mem-access	40	775.44	54.551
spm-access	8	131.72	10.168
timer-test_on(nop)	1	16.754	1.2857

to be modeled, namely the CPU, the different bus levels, the memory levels, and the other peripherals such as UART, interrupt controller or timers.

For example, the selected parameters for our CPU model are the CPU instructions, or possibly class of instructions, plus the caches and MMU activities. We thus built benchmarks to evaluate the cost of possible parameters, in order to select only relevant ones. Here are examples of benchmarks that were used, and their target event:

• `loop-calibration:` Measurement loop overhead benchmark. By running an empty loop, we can estimate the loop overhead.

• `insn-XXX:` Comparison of CPU instructions execution costs (add, mov, . . .). The target instruction is executed many times inside a loop.

• `XXX-access:` Calibration of costs of each bus level (AHB1/2) and memory level (cache, scratchpad or main memory), depending on the address accessed.

• `timer-test:` Example of peripherals energy characterization, this benchmark allows us to measure the timer power 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-XXX` benchmark with a nop instruction.

Calibration results. Benchmark energy results examples are listed in table 1. Full results are available in [4]. These results represent for each benchmark the length of the calibrated event in CPU clock cycles (second column), the per-event raw energy cost measured on the complete platform (third column) and the measurement error (fourth column). Energy costs reported here give the consumption of the complete platform for a full event execution. These raw costs have to be refined to get the final parameters.

As an example, the scratchpad memory access benchmark result (`spm-access`) gives the energy consumption of the CPU executing a load instruction, the bus conveying the load request and response and finally the scratchpad memory. The bus access cost includes the register accesses in the targeted peripherals since it is impossible to dissociate their costs. By removing the consumption of the CPU (one `load` and seven `nop`) and bus consumption, we finally obtain the scratchpad memory access cost. Experiments reported in [4] shows that the

scratchpad memory does not consume more energy than a register accessed via the bus.

Other model simplifications are possible. 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 basic model presented in section 3.1 can be rewritten, by using models simplifications obtained by calibration. We found that most instructions have the same energy consumption as long as they stay inside the CPU and that control and data logic switching between instruction execution can be neglected. Currently only ARM32 instruction set is modeled. Thumbs (16bit) instruction set can be modeled using the same benchmark methodology. In our setup, it is not possible to isolate the instruction cache consumption, which is lumped with the instruction consumption. ICache misses can be modeled as memory accesses.

We finally have a model for which CPU instructions are grouped in two classes, arithmetic and logic intra-CPU instructions, and load / store instructions. A memory load access is modeled as a load instruction, plus a bus overhead, plus a memory overhead. Peripherals energy consumption are taken into account thanks to state machines that give their consumption during instructions execution. The final model is written on equation (6).

$$\mathcal{E}_{\text{slot}} = \mathcal{E}_{\text{base}} + \mathcal{E}_{\text{insn}} + \mathcal{E}_{\text{bus\_access}} + \mathcal{E}_{\text{mem}} + \sum \mathcal{E}_{\text{periph\_state}} \qquad (6)$$

$\mathcal{E}_{\text{slot}}$ 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.

Frequency Scaling. The model presented before is valid for full speed software execution. However, the Integrator CM922T has frequency scaling capabilities but no dynamic voltage scaling (DVS) capabilities, hence when frequency is reduced, energy consumption does not decrease. When repeating five benchmarks at different frequencies, we obtain the curves in Fig. 1. This figure represents the per event energy values for the five benchmarks as a function of the clock divisor, $r = \frac{f^{\text{ref}}}{f}$ where f_{ref} is the nominal frequency (198 MHz here).

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 and external memories is not affected. Hence, the addition of a parasitic term which is roughly proportional
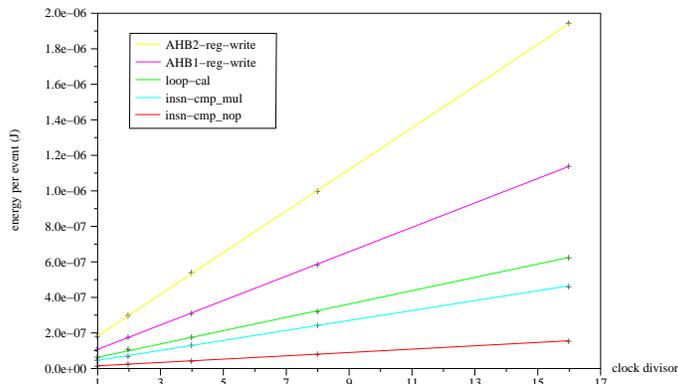
7

Figure 1: Multiple frequencies experiments: This figure shows that the **energy** per event increases linearly with the clock period (fixed voltage).

Table 2: Linear regression from curves of Fig. 1 based on the formula 7

Benchmark name	$\mathcal{E}_{\text{rp\_base}}$ (nJ)	$\mathcal{E}_{\text{mc}}$ (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

to the duration of the event or inversely proportional to frequency. This is clearly the case for the curves of Fig. 1.

We must underline that all five benchmarks generate activity in the modified clock domain (CPU), but not on the remaining part of the platform. On top of that, we kept all peripherals in the modified clock domain in an idle state. Hence, the event energy cost namely $\mathcal{E}_{\text{evt}}$ can be an instruction execution or a bus access for example. In this consumption we can identify two types of sources. The first is the energy due to modified clock domain $\mathcal{E}_{\text{mc}}$, which is constant. The second is the one due to the remaining part of the platform $\mathcal{E}_{\text{rp\_base}}$. Their relation in the total consumption of event is given by relation:

$$\mathcal{E}_{\text{evt}} = \mathcal{E}_{\text{rp\_base}} \times l_{\text{insn}} \times r + \mathcal{E}_{\text{mc}} \tag{7}$$

The first term is dependent on the frequency ratio r and the instruction length l_{insn}, whereas the second is not. Linear regressions on the results presented in Fig. 1 are shown on table 2. As shown in this table, equation (7) give 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 changing against software execution. The last two columns are the events real consumption and the regression error. The value for the base energy can be approximated by the mean value 10.82 nJ per cycle (with a standard deviation

8

of $\pm 2.6 \, 10^{-2}$).

4.2 Large targeted architecture range

The model building method was successfully applied on a second platform. This second platform is a wireless sensor network node, a WSN430 [1]. This system is based on a 16bits microcontroller, Ti MSP430, which has a completely different architecture compared to the ARM one. This second model construction validates the fact that the model is generic enough to be adapted on a large range of target.

On top of that, model adaptation to more complex peripherals is also verified since WSN430 embeds a wireless network interface. This interface is a good example of complex peripheral with multiple running states and dynamically modifiable parameters such as transmitting power.

4.3 Model usage

To ease their use, models obtained thanks to previous methodology are designed to be implemented in a simulation tool. The goal of such an implementation is to keep the main advantages of the model in its usage, namely simplicity and speed, and to offer higher level information on the application execution consumption.

Simulation architecture. To ensure a simple and fast usage of the model in software evaluation of energy consumption, we implemented it in a two step simulation tool as depicted on figure 2.
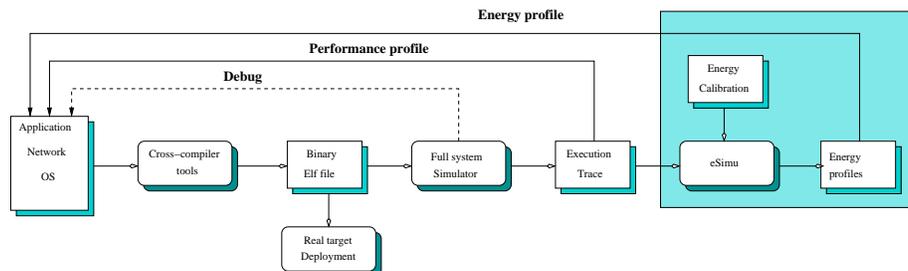
Figure 2: Simulation tool architecture

First, this implementation extends a standard development flow for embedded systems. As can be seen on figure 2, once software is written the first step consists in application compilation to produce application binary which will be executed on final system. However the process is iterative and debug phase is a cycle where user compiles, runs application either on real platform or on a simulator to identify bugs and correct them.

The simulation tool proposed replaces the previous one to offer energy and performance information on application execution on top of helping the developper to debug it. The two steps of this tool are first a full platform simulation and secondly an analysis of the trace produced by the first step simulation.

The first step of this tool is responsible of simulating the software execution on the full hardware platform in a reliable manner. It has to be functionally correct to accurately describe platform activities for energy consumption evaluation. It is also time accurate since energy consumption is dependent of time. Activities of the hardware platform are reported in an execution trace which fed the second tool step. The trace describes on an instruction execution time basis (the slot) all activities of the hardware platform, the instruction itself and peripheral events. These slots contain information on time and cycles spent in order to allow a DVFS support on platform natively embedding this technology.

The second step, `eSimu`, implements the model to produce energy consumption estimations based on information in the trace. This step processes the trace to get parameters, weights them with costs given by calibration informations to obtain energy consumption of the platform during an instruction execution. On top of energy consumption evaluation, this tools processes the instruction costs to report them in a higher level metric, at the source code level.

By relying on a given format for the execution trace, we can use the same implementation of the model for different platform adaptation. All platform dependent information are contained in the execution trace and the calibration data, given as input of `eSimu`. On top of that, the first step of this toolchain can be made thanks to a standard simulator, provided its sources are available, by extending it with a library distributed with `eSimu`. This extension consists in fetching some behavior information in the simulation loop and filling corresponding structures.

However, the main drawbacks of this architecture is the execution trace size and the I/O accesses necessary to write it. This is the reason why the library proposed for trace generation can produce diverse formats ranging from an ASCII human readable format to a binary Z compressed one. For example, the simulation of 60s of the sensor node sending packets produces a 0.57 MB compressed trace whereas raw ASCII trace size would have been 114 MB large.

Result outputs.

To ease the developper result interpretation, `eSimu` provides three kind of results output: global information, linear trace and source code annotation.

The first one is a summary of the temporal and energetic metrics for the application execution. This type of output can be used for quick software implementation comparition. Linear traces depicts the consumption of peripherals over time, which can be used to detect energy hungry peripheral behavior or misbehavior of a peripheral. Last but not least, source code annotation is the main source of information for developpers for application optimization. The format of this input is an emerging one used in current performance analysis tool such as the open-source project `Callgrind`, plugin of `Valgrind`. Using this standard format allows to reuse existing application for visualization like
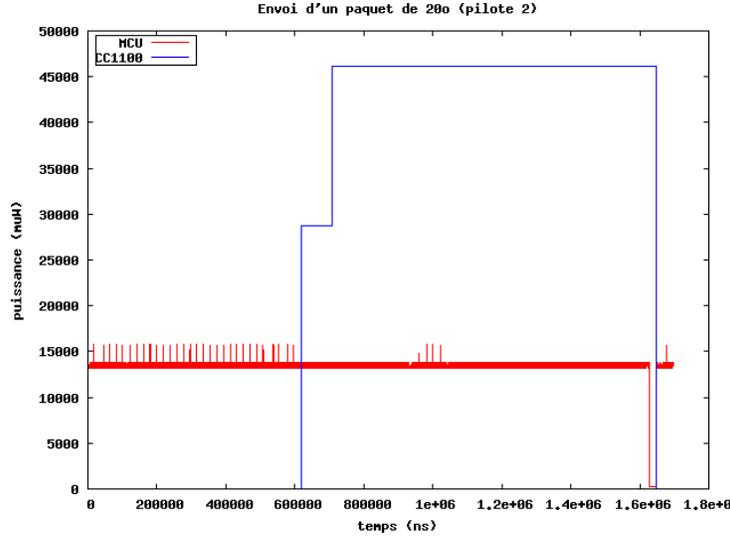
Figure 3: Linear Trace Example

the open-source project `KCachegrind` [7].

Figure 3 shows an exemple of linear trace output whereas figures 4 and 5 are respectively examples of annotated call graph and source code annotation visualization with `KCachegrind`.

Simulator Integration. Our models, the ARM Integrator and WSN430 ones, are implemented in this simulation tool suite. The first step, complete platform functional simulation, is accomplished thanks to two different simulators. For the ARM9 platform, the open-source project Skyeye [16] is extended to produce the cycle-accurate execution trace of the software. To fulfill this first task for the WSN430 platform, we augmented another simulator named WSim [5].

For the second step of the simulation tool suite, energy evaluation, `eSimu` was used for both platforms. As we said before, using an execution trace allows us to implement the model in the tool in a generic manner. The implementation of the model in `eSimu` enforces model properties, simplicity and genericity.

4.4 Model validation

We describe here our accuracy tests experiments. Our model is implemented in a simulator, and its results were compared to physical measurements.

Validation Methodology. To check the accuracy of the resulting model, we propose here a comparition between consumption estimations of the model, thus implemented in our tool and physical measurement on the real platform. This comparition is here given for the most complex tested platform, the ARM9
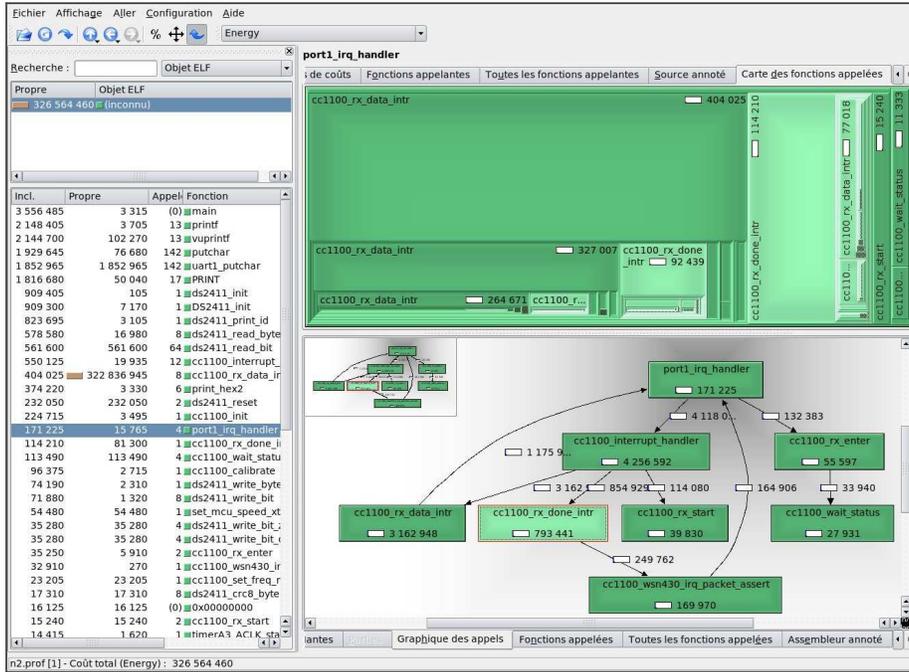
11

Figure 4: Call graph annotated with energy consumption information visualized thanks to KCachegrind

based one.

The test application chosen for this model validation are widely spread multimedia applications : MP3, JPEG, JPEG2000 and MPEG2. The implementations of these four 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 a lightweight one, Mutek [12]. Linux hardware layer abstraction makes interrupt request managment too long to allow a reasonable sized image or sound to be decoded in our measure time window. Four applications are executed in the simulation tools to get estimations of their execution.

Accuracy.

Results of model estimations and physical measurements are presented in table 3. The third column reports the physical measurement results, in terms of execution duration in CPU clock cycles and in terms of energy consumption in Joules. Fourth column gives the same kind of informations concerning the simulation results. Finally, the last column gives the percentile error of simulation errors of the simulation results against the physical measurement on the
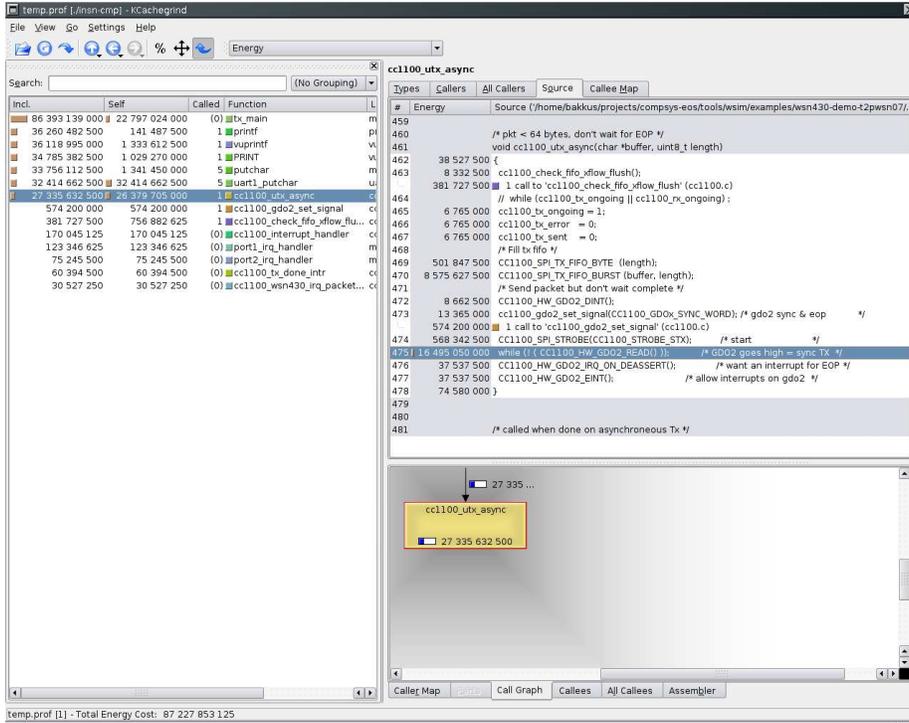
Figure 5: Source code annotation visualization obtained with the `KCachegrind` tool

benchmark	metric	measure	estimation	error (%)
mp3	cycles	13928480	14168406	+1.7
	energy (mJ)	233	243	+4.4
jpeg	cycles	6916836	7131529	+3.1
	energy (mJ)	114	120	+5.3
jpeg2k	cycles	7492173	7679416	+2.4
	energy (mJ)	126	137	+8.5
mpeg2	cycles	13990961	14542406	+3.9
	energy (mJ)	233	251	+7.7

Table 3: Simulators results: the results obtained for execution time and energy consumption by real hardware measurement are shown in third column, the simulation ones in fourth column. The last column gives the error percentile of the simulation

13

target hardware platform. These results show that a 10% error rate can be achieved by our complete platform energy model. These results validate that inter-instruction costs simplification is a reasonible choice to get fast yet precise estimations of the energy consumption. These 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.

5 Conclusion

In this paper we have explained how an accurate energy consumption model for a full embedded system can be built from external measurements and microbenchmarks. Our methodology requires a prototype platform of comparable technology. Quantitative energy data are gathered at the battery output and are translated into per instruction energy figures by data analysis. The resulting model is thus driven by the embedded software activity and can be used with a simulation execution trace as input. It is thus possible to very easily add an energy estimator to a software functional simulator so as to get feedback at the source level. As simulation tools modifications are kept at a minimum the simulation speed is not impacted. Consumption data clearly identify power hungry operations, thus offering hints for software design tradeoffs. The model built on an ARM9 based development board using this methodology achieved an error rate of less than 10 % at the source level, which is acceptable compared to its simplicity of implementation and its fast running time.

References

[1] O. Brevet, G. Chelius, E. Fleury, and A. Fraboulet. Wsn430: Ultra low power wireless module for sensor network applications. http://worldsens.citi.insa-lyon.fr/.

[2] R. Y. Chen, M. J. Irwin, and R. S. Bajwa. Architecture-level power estimation and design experiments. In *ACM TODAES*, volume 6, pages 50–66, January 2001.

[3] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G.-Y. Lueh. XTREM: a power simulator for the Intel XScale core. In *LCTES '04*, pages 115–125, 2004.

[4] N. Fournel, A. Fraboulet, and P. Feautrier. Embedded Systems Energy Characterization using non-Intrusive Instrumentation. Research Report RR2006-37, LIP - ENS Lyon, Nov 2006.

[5] A. Fraboulet, G. Chelius, and E. Fleury. Worldsens: Development and Prototyping tools for Application Specific Wireless Sensors Networks. In *IPSN'07/SPOTS*, April 2007.

[6] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John. Using complete machine simulation for software power estimation: The softwatt approach. In *International Symposium on High Performance Computer Architecture*, 2002.

[7] Available online, http://kcachegrind.sourceforge.net/, nov 2006.

[8] N. S. Kim, T. Austin, T.r Mudge, and D. Grunwald. *Power Aware Computing*, chapter Challenges for Architectural Level Power Modeling. Kluwer Academic, 2001.

[9] O. Landsiedel, K. Wehrle, and S. Götz. AEON: Accurate Prediction of Power Consumption in Sensor Nodes. In *SECON*, Santa Clara, October 2004.

[10] M. T.-C. Lee, M. Fujita, V. Tiwari, and S. Malik. Power analysis and minimization techniques for embedded dsp software. *IEEE Transactions on VLSI Systems*, 1997.

[11] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha. Automated energy/performance macromodeling of embedded software, 2004.

[12] F. Pétrot and P. Gomez. Lightweight Implementation of the POSIX Threads API for an On-Chip MIPS Multiprocessor with VCI Interconnect. In *DATE 03 Embedded Software Forum*, pages 51–56, 2003.

[13] 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.

[14] E. Senn, J. Laurent, N. Julien, and E. Martin. Softexplorer: estimation, characterization and optimization of the power and energy consumption at the algorithmic level. In *IEEE PATMOS 2004*, 2004.

[15] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *36th Design Automation Conference*, pages 867–872, May 1999.

[16] Available online, http://www.skyeye.org/, nov 2006.

[17] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *PATMOS*, 2001.

[18] T. K. Tan, A. Raghunathan, and N. K. Jha. EMSIM: An Energy Simulation Framework for an Embedded Operating System. In *ISCAS 2002*, May 2002.

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