eSimu : a Fast and Accurate Energy Simulator for Software Based Estimations

Nicolas Fournel
INRIA/Compsys
LIP/ENS de Lyon, France
nicolas.fournel@ens-lyon.fr

Antoine Fraboulet
INRIA/Compsys
CITI/INSA de Lyon, France
antoine.fraboulet@insa-lyon.fr

Paul Feautrier
INRIA/Compsys
LIP/ENS de Lyon, France
paul.feautrier@ens-lyon.fr

ABSTRACT

This paper presents eSimu, a performance and energy consumption simulator for deeply embedded hardware platforms such as sensor network nodes. These tools are based on cycle accurate simulation of complete hardware platforms executing the real application code. This allows designers to get fast performance and consumption estimations without deploying software on sensors, while being independent of any compilation tools or software components such as applications, network protocols or operating systems. The eSimu tools use a two steps approach. The first step consists in a fast cycle accurate simulation generating an annotated execution trace. The second step is an offline analysis that generates energy consumption profiles from this trace. The energy estimation is based on a platform-level energy model based on micro-benchmarks calibration using non-intrusive measurements on the real target hardware. Results presented in this paper show that an average error of 10% can be achieved using this method on a complete hardware platform based on a complex chip including an ARM9 processor.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Monitors*; D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments*

Keywords

Sensor networks, development framework, simulation, power consumption.

1. INTRODUCTION

Sensor network nodes development trend is currently following very closely the micro-electronic technology advances. This is particularly true for embedded micro-controllers and processors that are used in network nodes. What was, only a few years ago, a state of the art 32 bits processor with mid-range power consumption and packaging volume can be now re-engineered and repackaged using newer technologies. These changes have promoted this class of processors to ultra low-power consumption and very small physical footprint suitable for small devices such as sensors. Current state of the art nodes are using high-end 16 bits micro-controllers and chances are that current 32 bits embedded processors will be the future targets of sensor networks architecture.

This shift in processing hardware coupled with the ever increasing memory size available in embedded systems allows loading more and more software code on systems to drive physical sensors and peripherals. These shifts in complexity and capabilities have an important impact on the software stack that can be developed for wireless sensor networks.

Despite being ultra-low power, the hardware nodes configurations are, and still will be for a long time, highly constrained by energy consumption and battery capacity. It is now well known that energy consumption can be derived from the activity of the platforms and that the software part has a very important role in energy consumption of a full system. Indeed, in current processors and in smarter devices, power management policies are delegated to software drivers and application. This power management includes processors and peripherals standby modes that can range from the classic on/off model to more elaborate dynamic voltage scaling (DVS) techniques.

In such a context, much care must be invested not only in the design of applications, communication protocols and operating systems, but also in the code implementation and the hardware platform that will be deployed.

We believe that the development process for this kind of application must include early in the design stages not only a performance evaluation close to the cycle accurate model but also an energy and power consumption feedback. Using this feedback the developer can make important choices and tests while setting up the sensor application. This is particularly true in our current context where most of the sensor networks are application specific and are deeply embedded in the sense that no intervention is possible once the nodes have been deployed.

In this paper we present a methodology to elaborate an energy consumption model and use it in the traditional software development flow. Our energy model is built for a full platform that includes the processing units, memories and peripherals. The constructed model is geared toward software development on the deployment platform once the

hardware design choices have been made. Energy models are derived from real platform measurements using micro-benchmarks. The combination of cycle accurate simulation and energy consumption allows a fast and accurate feedback that can help the designer during the process of performance evaluation and software optimization of the target application.

In the remainder of this paper, Section 2 presents the usual embedded system development environment and the detailed introduction of our power estimation tools in this environment. Then we present in Section 3 an in-depth experimentation and evaluation of our approach using a complex platform including an ARM9 processor running a full operating system and multimedia benchmarks. Section 4 presents related work on power estimation for embedded platforms and we finally conclude and present our future works in Section 5.

2. FAST ENERGY AND PERFORMANCE ESTIMATION TOOLS

In this section we present the embedded system software development flow. We then present our proposal for precise timing and power consumption estimations in the software development process and finally, we present the integration of this proposal within the standard tool suite.

2.1 Standard embedded systems development tool suite

The usual software development for embedded system follows the process depicted in Figure 1.

Figure 1: Software development tool suite

The first step in this process is software implementation to produce the source code. All software component choices are made beforehand. For example, at this stage the choice of using or not an operating system (OS) to increase code re-usability must be made. The source code can be made of application, operating system and/or ad-hoc drivers.

The following development phase is source code compilation for the desired hardware target. Generally speaking, it is a cross compilation, since host computer and platform architectures are different. The development computer is frequently based on the x86 architecture, whereas sensors are often micro-controller or RISC architecture based platforms (ARM or MIPS). The output of this phase is a binary file ready to be deployed on the hardware.

The last stage in the development process is the debugging and deployment of the software. This phase is one of the most delicate phase in the software development process.

Two solutions are available to developers in this case. The first one is the deployment and debugging directly on the hardware. This solution makes the debugging task very difficult for two main reasons. Firstly, debugging an embedded system is usually made difficult due to the poor input/output possibilities of the platform. Secondly, hardware dedicated debugging tools like JTAGs are not necessarily of a good assistance, for example when debugging a device driver. Indeed, this kind of tool is intrusive: by activating a debug mode, timings can be modified making peripherals behave differently. The second solution consists in using a simulator during all the debugging phase and deploying on the real hardware only at the last moment. The simulation in such a process must reproduce the full platform behavior (including peripherals), helping the developer for the first part of the debugging.

Only few simulators targeting complete platform emulation can be found, we present some of them in section 4. These rare simulators are mainly functional simulation oriented, timing of hardware components are thus seldom respected. In such conditions, deployment is the last debugging and performance evaluation phase since the simulator debugged software cannot be certified to work due to timings, for example in device drivers development. In case developers need to optimize their software for energy and/or time performance, the preceding simulators are not of a good help. Indeed, they give no real output on the time performance of the software. As far as energy consumption is concerned, the only source of information the developer can have is the hardware itself. Optimization of time and energy are thus made at the source level but can only be quantified and validated by the deployment phases which remain highly time consuming.

2.2 Timing and Power Aware Simulators

To overcome the problem of picking out energy and performance informations altogether, we propose in this paper our `eSimu` energy instrumentation tool that augments the simulation tools available during the software development process. The simulation has to be more precise in terms of timing and performance analysis than the functional simulators can be, and it must simulate the platform with a full knowledge of power states (or events influencing the power state).

Before giving more details about the integration of this proposition in the standard tools-suite described previously we describe here the core of the energy estimation tool: the energy consumption model.

Our model is build toward end-user usability. The energy model must be simple enough to be used at the software level but still provides accurate information. Our energy model construction methodology has the following goals:

- The model must keep simulation runtime overhead as low as possible by using only a few architectural parameters.
- The model must not require a full knowledge of the hardware design that could lead to heavy modifications in the simulator.
- Energy calibration must be made on full target platform to represent the complete energy consumption.
- Measurements must be done using off-the-shelf measurement instruments.

It also has to be accurate enough to allow developers to take implementation and architectural decisions. Another desired characteristic for this model is that it must be easy to adapt to a new complete platform. This implies that the model is designed by following rules, and then calibrated on the target hardware.

Energy consumption model design

First, we propose a model abstraction level selection, which means select the size of black-boxes in this model to rep-

resent hardware components. Indeed, this choice influences the main trade-off in the model design between accuracy, model complexity and computation time.

Energy consumption models are usually built using abstraction levels ranging from transistor-level to system-level. At the lowest level, transistor level (or gate-level), every transistor (or gate) consumptions are computed and merged to give a component energy model. This requires the low-level description of the circuit (VHDL or Verilog description), and has a long running time.

At the highest level of abstraction, system-level, the consumption is computed using only very high level parameters. For example, for a CPU characterization, only instructions types are used to predict the consumption. Energy estimations are less accurate.

Between these two levels of abstraction, we find a family of models which can be regrouped in an architectural level of abstraction. At this level, all gates are not simulated, but grouped by architectural block. The granularity of the blocks can vary and architectural block can even be considered recursive. For example, a CPU can be a block, but it can be subdivided in core CPU, MMU, caches, ...

Our studies showed that the best trade-off we can achieve to obtain a fast but accurate estimation of energy consumption for a complete platform is with an architectural level model. In the resulting model, processor, memories and peripherals are considered as black-boxes connected by the interconnect (a bus or a Network-on-Chip). The data and control informations are kept for bus transactions, since they give the access level in the bus hierarchy. The main components are CPUs or micro-controllers, memory hierarchy blocks (including caches if they exist) and peripherals. Figure 2 give examples of platform architectural blocks. Each of these blocks has to be modeled using a set of power states that represent the energy consumption of the block for a given activity or configuration. In this paper we focus on the simulation tool flow and the use of our energy model but the interested reader can find all details about our energy architectural models in [**?**].

Figure 2: Platform architectural blocks examples.

The second design decision concerns the collection of data needed to adjust the model to the hardware platform. We want it to be simple enough to allow developer to adapt the model easily to a new platform. For this reason, the model is based on a measurement based data gathering instead of complex models or manufacturer informations. The first one needs far too complex informations on the platform, for example low-level description that would probably not be available to the developer at this stage. The second is not precise enough, since manufacturers of each chip of the complete platform gives consumption figures that do not take into account the energy cost of the integration components. On top of that, there is little chance that figures given by all chip manufacturers are usable together, for example because they are not given in the same usage condition.

The best situation is when the measurement are simple enough to be made by the developers themselves. The setup should need only rudimentary skills in electronics. To meet this requirement, the model is based on measurements collected at the power supply input of the platform. This also allows measures to take into account the energy cost of integration components of chips and technological process variability, resulting in a better accuracy.

The last choice in the model design concerns the parameters selection. The abstraction level dictates the granularity of the possible parameters, but it is still necessary to select the relevant parameters for each type of platform. The parameters selected in our model are behavioral and architectural informations on the hardware. In all cases these parameters are informations which can be gathered from the hardware description manuals. The most important thing is that it only needs informations available during the development phase, and not low level description of the hardware.

The main parameters categories of this model are:

- CPU instruction classes: CPU instructions are the CPU model parameters, they are regrouped into categories whose consumption per cycle is very close. For example we can have arithmetic and logic instructions in a category and load/store instructions in another.
- Memories and busses accesses: The energy consumption of interconnects and memories may depend on the different level of busses, or memory accessed, this mainly depends on addresses.
- Other peripheral events and/or states: The cost of other peripherals are mainly due to their running state and the events in these peripherals, *e.g.*: an interruption request for the timer.

Each of the selected parameters has an energy cost and can also induce a state change in one or more peripherals. The cost of each event depends on factors such as its number of cycle, the supply voltage. The global energy of the platform is then computed by summing the event costs and the peripherals consumption in the the current state. Formula 1 summarizes the model.

$$E_{app} = \sum E_i^{eff} \times t_i + t \times E_{base}^{cycle} + \sum E_{peri}^{over} \qquad (1)$$

E_{app} is the application energy consumption, E_i^{eff} is energy cost by cycle of class i instructions. t_i is the cycle count accumulation of class i instructions execution. The E_{base}^{cycle} term represents the per-cycle platform base consumption, this cost is always consumed when the platform is powered whatever happens in the software running on it, that is the reason why it is only multiplied by the application duration in cycle. Last term, E_{peri}^{over} corresponds to the peripheral consumption overhead, composed of the peripheral state cost and state change cost. For example the UART idle state has a specific cost, but changing to sending state induces additional consumption. All energy terms $E*$ are dependent on extra factors such as their power supply voltage. These factors allow the dynamic voltage scaling (DVS) capabilities to be modeled. Details about this support are described in [**?**].

Calibration measures

Once the model is built, it must be calibrated to the real hardware platform. Our model is based on simple non intrusive measures. This kind of measure only allow to get global informations on the complete platform. The difficulty here is to extract a precise evaluation of a model parameter cost among these global figures. The solution proposed is

to structure the model calibration phase with micro benchmarks. These benchmarks are run on the platform to keep it in a known state and trigger only a given amount of the calibrated event. This procedure allows us to recompute the platform consumption and get the cost of each event. Benchmarks are built for each parameter or event: instructions, bus accesses, ... The complete benchmark structure description is given in [**?**].

2.3 Simulation structure

In this section we present the details on the integration of our `eSimu` tool in the standard tool suite.

We decided to replace the simulation phase described in Figure 1 by a two step simulation in which `eSimu`, the second step, implements the preceding model.

Figure 3: eSimu tool flow including energy consumption estimation

The extended tool suite is transformed as depicted in Figure 3. The first step simulator takes the same place as the functional simulation in Figure 1. Its aim is to help developers to debug their application but also to produce an instrumented cycle accurate execution trace. The second step is then in charge of generating the performance and energy profiling data of the software.

Execution trace generation

The first simulation step aims at replacing the functional simulation to allow developer to make standard debugging in terms of software and hardware behavior. The hardware simulator is modified to be cycle accurate by using instruction timing as reference. This implies that all hardware behaviors are described more accurately in terms of timings. The main aim of this first simulation in our simulation flow, is to produce the cycle accurate trace of the software execution on the complete platform. The platform simulator must take as input the target binary code, for instance the ELF file, in order to not imply any extra work for the developer like code rewriting for simulation.

The output of the simulation is the cycle accurate trace. It is human readable file which allows first timing checking. This file describes the behavior of the platform during the software execution by reporting all instructions executed on the CPU, memory accesses, but also peripherals activities like UART sending a byte. All these informations will be used in the energy simulation, for time and energy performance profiling.

The trace is a linear description of the platform activity in discrete time. The time reference of this trace is the instruction executed on the CPU. Each instruction length is reported in CPU cycles. The reasons of this choice are that on one hand the CPU clock frequency of the platform is generally the fastest, and thus the more accurate time reference. On the other hand, the only interactions between hardware peripherals and the CPU are the interrupt requests. These exceptions, among which we can find interruption requests, are taken into account at the end (or beginning) of an instruction execution.

This simulation phase gives developers the mean for functional debugging of its software in a cycle accurate manner before beginning the profiling phase. To perform this second phase, they have an instrumented execution trace. A standard functional simulator can be used in this step, but it has to be upgraded for cycle accurate simulation of CPU and peripherals. It must also be able to produce the execution trace for the second step simulator.

Energy estimation

The second simulation phase, handled by `eSimu`, is in charge of the energy profiling of the software.

The cycle accurate execution trace generated by the previous step is one of the input of the energy simulator. Calibration data gathered as described in section 2.2 are grouped in a file, the second input of `eSimu`. The implementation of the model in `eSimu` can then remain platform independent.

`eSimu` implements the model presented in section 2.2, whose calibration is given by the second input file. The last step in the model usage is the estimation of parameters to be fed in the model to generate the energy estimations. To achieve this task, it computes the parameters value from the execution trace.We decided to base `eSimu` on a realistic execution of the software on the hardware platform since it gives more accurate results than offline stochastic estimations of occurrence number for events like cache misses, TLB (Translation Look-aside Buffer) misses.

This simulation gives two kinds of results. The first is the global energy consumption of a software 'run'. The second one is composed of the profiling informations of the software.

These profiling informations are formatted as a call tree. `eSimu` is then responsible of aggregating the linear time based execution trace into this software call tree. This output formatting allows better optimization decisions, since developers can concentrate their efforts on the most power hungry functions in the software. `eSimu` gives its energy estimations, in this format, but it also keeps and reformats the performance informations too. The developer has the possibility to perform energy and/or performance optimizations. We must underline the fact that in case an operating system is used, its energy and performance profiles are also generated, at least for its used part.

The file format chosen for these outputs is an emerging file format used in open-sourced projects like `CallGrind` a skin (plugin) of `Valgrind` [**?**]. Some profiles visualization tools are also available like `KCacheGrind` [**?**].

Two step simulation advantages and drawbacks

This simulation architecture has two main advantages.

The first advantage is that we benefit from a fast debugger in the first step simulation since the energy estimation, which needs much processing, is not necessary at this time. The developer will begin energy and or performance profiling and optimization only when the software is functionally correct.

The second advantage of this architecture is that `eSimu` is more generic. All platform dependant informations are given to `eSimu` by its input. All platform dependent code must be developed in the first step simulation at the execution trace generation by reporting the event classes. As far as the first step simulation is concerned, such a structure encourages the reuse of existing simulation projects. The execution trace generation extension of such simulator can be made quickly as we demonstrate in the next section.

The main drawback of this structure is the size of the execution trace. If developers target a long software run, the

trace size will rapidly increase. This can be easily overcome by compressing the trace or even using operating system pipes between simulation process.

3. EXPERIMENTS

We present in this section an example of the tool suite usage and show the energy and performance accuracy of the simulation tool flow. These two experiments are driven on a next generation sensor platform based on an ARM9 processor. We will first give a short description of its architecture and then more details about the experiments and tools.

3.1 Experimental Setup

The sensor platform used during our experiments is an ARM9 based platform. Its processing power and the current evolution in processor size and energy consumption make it a rather good representative for next generation sensor network nodes. The decrease of processor size implies a reduction of energy consumption and an increase of their target applications. For example, the ARM7 processors used to be considered as processors, and are now considered as 32 bits micro-controllers. Nowadays we can find ARM7 CPUs in nearly all bluetooth devices, and in some full featured IEEE 802.11abg wireless devices. On top of that, current sensor network data, like temperature, require only few processing on the nodes, but we can state that next generation sensors will capture sound or even image which will need more processing on the nodes. This will induce the use of processors with the computing power of the same class of architecture as an ARM9 core based CPU.

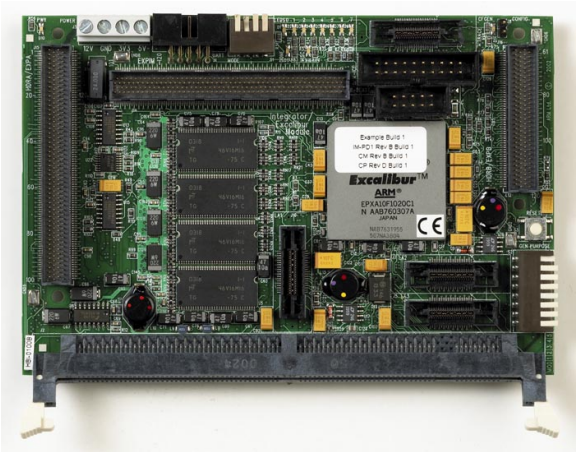

Figure 4: ARM Integrator CM922T-XA10 front side

The global architecture of our platform is depicted in Figure 5. This platform is based on an ARM922T processor, more precisely on an Altera Excalibur EPXA10 which is a FPGA including an ARM CPU and some peripherals in the same chip.

Figure 5: ARM Integrator CM922T-XA10 architecture

This chip contains a two level AMBA bus to allow the processor to access to all peripherals. Memories are organized in a three level hierarchy, from nearest to farthest of the processor: caches, SRAM memories and main memory (SDRAM). Finally, all ordinary peripherals are integrated in the Excalibur chip, like UART or Timers.

We choose this platform as it represents a reasonable node of the next generation thanks to its huge computing power and because most of the peripherals are integrated within the same physical chip as the processor. Only the lower level of the memory hierarchy (SDRAM) and peripheral physical interfaces are external. This makes this kind of architecture a good representative of system on chip devices with reduced number of components on the printed circuit board.

The architecture simulator we use is derived from the open source `skyeye` [**?**] simulator. Skyeye is a functionnal simulator targeted to ARM based embedded systems. Several full platforms are available for simulation like full featured PDA. This simulator is augmented in our case for our CM922T-XA10 platform support and we also added instruction cycle accuracy timing and peripheral activity reporting. This simulator is responsible for generating the linear execution trace that is later used by `eSimu`.

The cross compiler we use is the standard GCC C compiler targeted to ARM processors.

Energy consumption calibration of the platform has been done by running several micro-benchmarks on the real hardware platform. These calibration data are collected as described in section 2.2. Our measurement setup is very simple since it consists in a digitalizing oscilloscope (Tektronix TDS 7054A), and current and voltage probes. These probes are placed at the power supply input to sample the power consumption of the system at a sampling rate chosen so that we meet Shannon's Law. This setup is completed with a trigger signal controlled by the platform processor, allowing us to collect only the desired data. Finally, the benchmarks were developed on top of the lightweight operating systems Mutek, because it allows us to control what is running on the platform more tightly than an Unix like OS would. An example of calibration measurement, is given by Figure 6. It represents the power of the platform while the UART is sending bytes.

Figure 6: UART calibration: Measurement allowing to calculate the byte sending cost.

3.2 Tools Suite Usage Example

We present in the section the process used to produce the JPEG test applications used in the following part to estimate the simulation results accuracy. This process is the usage of tools described in the section 2. We give details about the steps of this process.

The implementation of the JPEG application is the standard Linux `libjpeg` library. This implementation effectively uses operating system services and standard libc functions. We decided here to replace Linux and the standard libc by the Mutek [**?**] lightweight operating system which also includes a libc implementation.

All sources, JPEG application and Mutek operating system, were compiled by using the standard GCC cross compilation toolchain for ARM.

The execution trace presented in Figure 7 is taken from the JPEG execution trace. The trace contains for each instruction executed on the processor ('@'ed lines), the total number of CPU cycles spend since the beginning of the sim-

```
[...]
@ 107651 {2} 0x003098a8 3 0x0a000016 J
@ 107784 {1} 0x00309908 1 0xe24bd01c IC [0x00309908]
+ 5 4 W [0x005029bc<1>]
+ 28 4 W [0x005029c4<1>]
+ 28 4 W [0x005029c8<1>]
+ 71 4 R [0x00309900<8>]
@ 107795 {3} 0x0030990c 11 0xe89da8f0
@ 107796 {1} 0x00301bb4 1 0xe1a00006
@ 107857 {2} 0x00301bb8 3 0xeb002033 IC [0x00301bc0] C
+ 58 4 R [0x00301bc0<8>]
@ 107906 {1} 0x00309c8c 1 0xe1a0c00d IC [0x00309c8c]
+ 48 4 R [0x00309c80<8>]
@ 107914 {3} 0x00309c90 8 0xe92dd8f0
@ 107915 {1} 0x00309c94 1 0xe3a02054
@ 107964 {3} 0x00309c98 1 0xe5906004 IC [0x00309ca0]
+ 48 4 R [0x00309ca0<8>]
@ 107965 {1} 0x00309c9c 1 0xe24cb004
@ 107966 {1} 0x00309ca0 1 0xe3a01001
@ 107967 {1} 0x00309ca4 1 0xe1a04000
@ 107968 {1} 0x00309ca8 1 0xe1a0e00f
@ 107973 {3} 0x00309cac 5 0xe596f000
@ 107974 {1} 0x0030fc4c 1 0xe1a0c00d
@ 107985 {3} 0x0030fc50 11 0xe92ddff0
@ 107986 {1} 0x0030fc54 1 0xe24cb004
@ 107987 {1} 0x0030fc58 1 0xe24dd004
@ 107988 {3} 0x0030fc5c 1 0xe59f31a4
@ 107989 {1} 0x0030fc60 1 0xe1a04001
[...]
```

Figure 7: Execution trace example, output from the cycle accurate simulator with instruction timing annotation (first field after @), class of instructions between brackets, address and memory hierarchy accesses (lines starting with +).

ulation, the instruction class, the instruction address, length in CPU cycle and the instruction itself (used only for debuging purpose). The trace also contains instruction or data cache line fills in case of misses. The bus accesses are reported on lines starting with '+', with their duration in CPU cycles, level of access, address and size of burst.

In this example, we can underline that a call occurred at the tenth and eleventh lines. The called function is named `jinit_inverse_dct` and is placed at address `0x00309c8c`.

The second step simulation is performed by the `eSimu` with the calibration data collected beforehand. A profile example is presented in Figure 8. This example gives the profile informations of the function `jinit_inverse_dct`. This profile is given per C line for the following metrics: instruction fetches, CPU cycles, energy consumption and instruction and data cache misses. We can see from this example that this function calls the `memset` libc function and that it costed about 32474760 nJ (for all 34 calls).

In terms of simulation performance, the execution time of the JPEG test application is about 30 ms on the real platform. The first step simulation takes 25 s, with approximately a 10 s overhead due to execution trace generation, and the second step 20 s.

The profiling informations generated by `eSimu` are finally visualized with `KCacheGrind` [**?**] as shown on Figure 9. This tool allows a fast overview of the repartitions on the software of each of the metrics placed in the final profiling result, for example, energy consumption or execution time in cycles or even data or instruction cache misses as proposed by performance evaluation tools such as `Valgrind`. `eSimu` allows to add an energy annotation to the usual performance counters used in the visualization tool. The energy per function and per C line is made available along with the peripherals energy consumption (not taken into account in the

```
fl=./jpeg/jpeg-6b/jddctmgr.c
[...]
fn=jinit_inverse_dct
248 2 57 872 1 0
+5 2 50 753 1 0
-5 1 1 15 0 0
+5 1 1 15 0 0
-5 1 1 15 0 0
+5 2 6 100 0 0
+6 1 1 17 0 0
-2 1 49 738 0 1
+2 1 49 736 1 0
-2 1 1 17 0 0
-1 1 1 17 0 0
+3 4 4 62 0 0
+3 6 86 1304 1 0
+3 1 1 15 0 0
-3 1 1 17 0 0
+3 2 80 1202 1 0
cfi=../../../src/libc/string/memset.c
cfn=memset
calls=1030 34
* 216688 2149452 32474760 1 0
+2 2 24 363 0 0
-8 7 195 2954 1 0
```

Figure 8: Profile file example: functions (fn fields) and C line performance counters for several metrics includind energy consumption.

`KCacheGrind` tool).

3.3 Simulation Accuracy

We show the results of the accuracy verification of the simulation results, *i.e.* energy and execution time estimations.

Test applications

For the software part of our experiments, our platform and simulator can run several operating systems such as Linux, uClinux and Mutek [**?**, **?**]. We choose to run on top of these systems a range of 3 multimedia applications : a JPEG image decompressor, a JPEG2000 image decompressor and a MPEG 2 video application.

The three test applications were developed by following the process described before. We used the Mutek to fulfil the operating system and libc tasks, but we could also have used Linux. The choice of Mutek is only a question of measurement duration. Indeed our oscilloscope allows us to capture a 100 ms measure. With Linux (or even uClinux) used as operating system, this time window would have been to short to complete test applications with reasonable image sizes. This is mainly due to the hardware abstraction layers of Linux that makes interruption requests processing much longer.

An example of a test application consumption measure on real hardware is given on Figure 10. The figure shows the power consumed during the test application execution. The power is recomputed from voltage and current samples and is represented by the upper curve. The second curve represents the trigger signal, a LED voltage. The execution of the MPEG2 test is done when the trigger signal is low. We can observe different phases in the MPEG2 consumption profile. These phases correspond to the image slices decompression.

The test applications were run on the hardware platform and measured with the same measurement setup as the one used for energy model calibration phase. These measurements give us global consumption of the test application,

		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 1: 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.

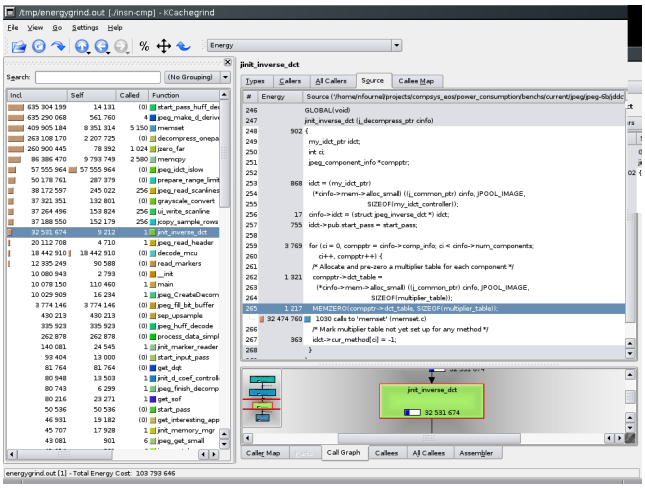

Figure 9: Performance/energy repartition visualization using KCachegrind tool

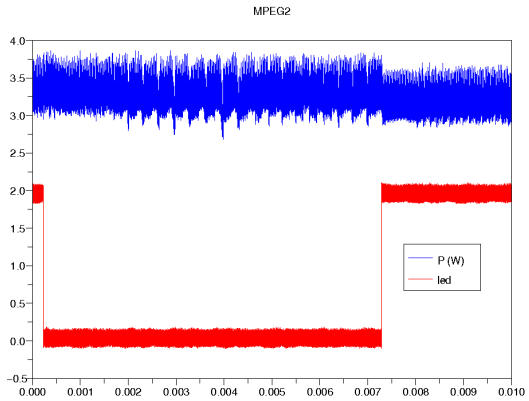

Figure 10: MPEG2 application power consumption measurement on real hardware (upper figure) and measurement trigger (lower figure).

which can be then compared to the global informations given by the simulation tools, `skyeye` + `eSimu`.

Results of these two different runs of each test application are presented in table 1. The second columns gives the code length in lines, these figures report only the application and library code length. The Mutek operation system source code length is not included. Third and fourth columns presents the results of the measurements series in terms of cycles for execution time and in Joules for energy consumption. Indeed, due to variability in the measurement process the value presented are mean values of twenty repetitions of the test application measurements. Standard deviation of measures is about 0.1% of the measured values (for time and energy). Fifth and sixth columns give the same figures for the simulators results. Finally, the last two columns give the relative errors made by the simulators against real (measured) values.

These results show that the first step simulation gives approximations of the execution time accurate within a 5% error. The platform modeled here is a very complex system. To keep a fast execution trace generation and functional simulation using our architectural level of abstraction, some simplification of the timing model were made, which has a cost in term of accuracy. For example, the memory controller is one of the more complex part (CPU excepted) of the plaform and is not included in our platform description. Its model would have been too complex and time consuming to be fully implemented with regard to the estimation error we can acheive by bypassing it.

As far as the second step simulation is concerned, the estimation accuracy is below the 10% error rate. One part of this error can be explained by the fact that energy estimations inherit of the time prediction errors. Indeed, the error in the first step simulation is mainly due to pipeline stall length miss-predictions. If the first step simulation declares less pipeline stall cycles than the real hardware effectively made, the second step simulation will underestimate the instruction cost. The second part of the error is due to intrinsic model errors.

Despite these limitations our energy model and simulation environment allow us to achieve an overall error on power consumption between 5 and 10% for our applications which we consider to be a very good result at this level of abstraction.

4. RELATED WORKS

The main contribution of our proposition is the energy consumption estimation calibrated with simple non-intrusive measures. Many works in the literature focus on energy modeling, their model use abstraction layers which range from transistor or gate level to system level. At the lowest level it is impossible to model a complete platform as the method does not scale very well. Intermediate level, also called architectural and RTL level, for example Brooks *et al.* [**?**], are mainly oriented for hardware design. It is possible to model a complete platform, but not to run software on it, more precisely not for fast energy optimization. As far as the highest level is concerned, only a few works are interested in modeling a full platform. Indeed, some works call system the combination of a CPU and the main mem-

ory, for example [?]. In sensor networks, some higher energy consumption are proposed, like number of sent packets on wireless medium. These approximations do not take into account the consumption of remaining part of the sensor or even sending device behavior. Our model is based on the principles proposed for these system level models, as Tiwari's one [?], coupled to principles proposed for architectural models by Kim *et al.* [?], to extend the model to the complete platform. As far as measures are concerned, we used a methodology close to the one presented by Russell and Jacome in [?].

In the past years, simulators of ARM based embedded systems were proposed, JouleTrack [?] and EMSIM [?]. The first one only models the energy consumption of the ARM processor. By measuring the energy consumption of a StrongARM SA1100, which implements the ARMv4T Instruction set, they observed the same phenomenon than us, there are two main class of instruction consumption. Our ARM922T also implements the ARMv4T instruction set. EMSIM is a full platform simulation relying on Simunic *et al.*'s model [?] for the consumption estimation of the SA1100, which allows to model dynamic and voltage scaling capabilities of the StrongARM. Unfortunately, the peripherals consumption models are poor, which would have not be precise enough in our case.

The closest project in the literature, AEON, was proposed by Landsiedel *et al.* [?]. This work proposes to model the energy consumption in sensor networks based on the consumption profile of the executed code. AEON was implemented in the Avrora [?] Mica2 node simulator as an internal energy counter. Avrora models a Mica2 sensor platform based on an AVR micro-controller from ATMEL and includes some peripherals such as the wireless interface. Their energy consumption model is also instruction based and the overall reported power estimation precision is 5%. Our simulator is built on a hardware abstraction that allows to have much more complex hardware architecture while being independent of the hardware simulator internals. The obtained results from `eSimu` for our ARM9 based platform are of the same order of magnitude than AEON in accuracy.

5. CONCLUSION

We present in this paper `eSimu`, a fast and accurate energy and time estimation simulation tool suite. These simulators are proposed to overcome the traditional embedded software development suite lack of information for precise time and energy consumption performance. These two performances data are important in deeply embedded systems such as sensor network nodes, since developers needs to optimize their code to meet time constraints, in network protocol for example, and to improve the sensor node life time.

This proposition is based on a complete platform energy consumption model calibrated thanks to micro-benchmarks based measurement on the target hardware. This makes the model easy to adapt to sensor platforms. The `eSimu` tool suite is structured in a two step simulation flow. The first is a functional simulation adaptation to cycle accurate complete platform simulation and execution trace generation. The second phase is a highly parameterizable and generic implementation of the consumption model, which produces time and energy profile informations of target software.

This tool suite was tested on a complex architecture platform and achieved an accuracy of 10% in terms of energy estimation, which is almost as better as previous propositions in sensor nodes energy estimation, generally based on micro-controllers (less complex architecture). The tool allows to have a feedback from the simulation at the function and C level using standard code profiling tools. Our future work is to include a data visualization tool that can take into account peripherals energy consumption reported by our execution trace.