VLSI DESIGN
2001, Vol. 12, No. 2, pp. 245–273
Reprints available directly from the publisher
Photocopying permitted by license only

An Instruction-Level Power Analysis Model with Data Dependency

GIUSEPPE ASCIA[a,*], VINCENZO CATANIA[a,†], MAURIZIO PALESI[a,‡] and DAVIDE SARTA[b,¶]

[a]*Dipartimento di Ingegneria Informatica e delle Telecomunicazioni, Università di Catania, V.le. A. Doria, 6–95125 Catania–Italy;* [b]*STMicroelectronics, Stradale Primosole 95100 Catania–Italy*

(Received 20 June 2000; In final form 3 August 2000)

Power constraints are becoming a critical design issue in the field of portable microprocessor systems. The impact of software on overall system power is becoming increasingly important as more and more digital applications are implemented as embedded systems, part of which are hardware (ASICs) and part software in which a specific application is executed on a processor. In this paper, a data-dependent instruction-level power analysis model is presented. It is compared with the average cost model proposed by Tiwari *et al.* [1] in both estimation accuracy and characterisation time. The data-dependent model can be generalised to be applied to generic RISC processor. Application of the data-dependent model we propose sensibly reduces errors in estimating software power consumption per clock cycle which is lower than 10%, in the case of the ST20-C2P core.

Keywords: Embedded systems; Low-power; Power estimation; Instruction-level power model; Data dependency; Software consumption

1. INTRODUCTION

The spread of portable microprocessor systems (mobile phones, PDAs, MP3 players) that use a battery as a source of energy is making power consumption a key issue. Power consumption constraints are now equally important system design specifications as memory area and performance. The reasons for this sudden interest are not only the evolution of technological processes (which makes it possible to integrate millions of transistors on a $1\,cm^2$ die) but also the development of the *System-On-Chip* concept whereby a processing system with all its peripherals can be implemented on a single integrated circuit, thus allowing us to hold in the palm of our hand what

*Corresponding author. Tel.: +39-95-7382353, Fax: +39-95-338280, e-mail: gascia@iit.unict.it
†e-mail: vcatania@iit.unict.it
‡e-mail: mpalesi@iit.unict.it
¶Tel.: +39-95-7407742, Fax: +39-95-7407720, e-mail: davide.sarta@st.com

up to a few years ago took up a whole desk. Competition on the portable electronics market concerns three different aspects: autonomy, functionality and volume. Only an optimal combination of these three indexes is the winning solution. It is not a simple task as these three variables are closely correlated and enhancing one is often to the detriment of another. If, for example, the aim is to increase autonomy, one possibility is to use a larger battery, but this increases the volume; to increase functionality more complex software is needed (the functions of an embedded system are usually implemented by the software): this generates a greater processing workload, which means more power consumption and consequently less autonomy.

Current research in the field of low-power devices focuses on two issues: methods to design low-consumption architectures and the definition of techniques to analyse power consumption right from the earliest design stages. The former include low-power synthesis techniques [5, 6] which map a technology-independent circuit onto a dependent one, using a power metric to choose gates from a library; techniques for power management applied to the single blocks of a circuit which allow blocks whose result will not be used to be turned off (clock gating, operand isolation *etc.*) [7–10]; encode/decode techniques to minimise switching on high-capacity buses (*e.g.*, those interfacing the memory) [13–19]. Among the latter we have methods operating at a higher level of abstraction which conduct a search through all possible system configurations and choose the one that optimises a function usually depending on the variables area, performance and power. In [21, 22] the possible configurations of a reference system comprising a CPU, a hierarchy of memories and a certain number of ASICs and peripherals are explored for varying memory hierarchy configurations (size, associative capacity, cache block size) and bus features (number of lines, encoding techniques). In [23, 24] the architecture of more complex, highly parametrised systems is explored, defining a wide range of possible configurations.

Optimisation in terms of power is only possible if estimation methods and power analysis tools are available to measure the impact of any modifications made. Power consumption analysis often focuses more on the impact of the hardware components of a system, not taking into account the impact of software on overall system power. Software is, however, becoming increasingly important as more and more digital applications are implemented as embedded systems, part of which are hardware (ASICs) and part software in which a specific application is executed on a processor. Although it is clearly of fundamental importance to assess the impact of software on power consumption, most research efforts have focused on defining analysis methods at a low level of abstraction, *i.e.*, at the circuit and gate level, and little has been done at a higher level, in particular the software level. Analysis at lower levels gives the best results in terms of accuracy, but it takes a very long time to estimate the consumption of a system on which a program is executed. Estimating the power consumption for a program with millions of instructions is not realistically possible with low-level analysis tools. A higher-level, *i.e.*, software-level, methodology is less accurate but has the advantage of taking much less time.

An approach specifically defined to estimate the amount of power absorbed by a processor when it executes a software program was presented by Tiwari *et al.*, in [1]; it consists of characterising the instruction set of a processor in terms of power, by assigning an average cost to the various instructions. Once the instruction set has been characterised in terms of power it is possible to calculate the amount of power absorbed by the processor when it executes any program with a known trace, by summing the power contributions of each instruction executed. This model, however, does not take into account the contribution of data to power consumption. While it is low for highly complex processors, in the case of very simple processors (frequently used as the core of embedded systems) the contribution made by data

may be significant. In this case considerable errors may be made in estimating power consumption.

In this paper we present a software power consumption analysis model that can capture its dependence not only on instructions but also on data. The cost model is characterised by means of a circuit-level analysis tool. Characterisation using very low-level analysis tools is the only type possible when the processor silicon is not available but only an HDL model or the netlist of the core, as happens in the design of embedded systems. Although the instruction set is characterised at a low level, this is not a limiting factor because it only needs to be done once. Once the characterisation stage has been completed, software power consumption can be estimated by executing the program on a simulator of the processor and summing the contributions made by the instructions and the data involved at each clock cycle. This means that it is not necessary to know details of the architecture of the processor to estimate the software power consumption, but it is necessary to have the power cost model provided by the manufacturer.

Application of the data-dependent model we propose sensibly reduces errors in estimating software power consumption per clock cycle. In the case of the ST20-C2P core, which we took as a case study, the error per cycle remained lower than 10%, whereas the mean error for the whole of the test was below 5%.

In 2 we will describe the average cost model proposed by Tiwari *et al.* [1] and apply it to the ST20-C2P. As will be seen, the great data-dependency prevents an instruction from being labelled with an average cost. To overcome this problem, in 3 we propose a data-dependent model which measures certain activity indexes to obtain more precise estimates for all architectures in which data makes a significant contribution to power consumption. In 4 we will discuss the strategy used to design a high-level tool to estimate software power consumption on the basis of a trace generated from a functional model of the processor. Finally, 5 will give our conclusions and discuss future developments.

2. AN AVERAGE MODEL

In [1] Tiwari *et al.*, proposed a method for estimation of the power absorbed by a processor executing any program, starting from a trace of the instructions it executes. The authors state that by measuring the power absorbed by a processor X repeating certain instructions or short sequences of instructions it is possible to obtain much of the information needed to calculate the power dissipated when the processor X executes the instructions of any program Y. Note that we will use terms *power* and *energy* interchangeably since for a processor power is just the energy per cycle times the clock frequency.

Given a processor, and indicating the code of a generic program as P, an estimate of the amount of power absorbed by the processor during the execution of P is given by the following equation:

$$E_P = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \quad (1)$$

where the basic cost, B_i, of the instruction i (*i.e.*, the average amount of power absorbed when all the stages of the pipeline are processing the instruction i) is weighted by the number of times, N_i, the instruction is executed. To this it is necessary to add the overhead, $O_{i,j}$, due to the change in system configuration (since the instruction i is preceded by the instruction j) weighted by the number of times, $N_{i,j}$, this pair occurs during execution of the program. Finally, the power consumed by any stalls or cache misses that occur during execution of the program is added. The terms B_i, $O_{i,j}$, E_k refer to the processor being investigated and are determined in a preliminary characterisation stage in which the *basic cost* of each instruction in the instruction set and the cost of changing instructions for each pair of instructions (or each pair of classes of instructions) are defined. In Section 2.1.2 we will discuss the techniques used to determine these values.

Model 1 was applied to three different processors [3]: an Intel 486DX2 (a CISC processor based

on the x86 architecture) [1], a Fujitsu SPARCliteMB86934 (a 32-bit RISC processor based on the SPARC architecture) [4] and a proprietary Fujitsu DSP [2]. For the fist two the model was simplified, representing the contribution $O_{i,j}$ for any pair of instructions with a single constant term. This was because the overhead (with respect to the basic cost) of the power absorbed during execution of the instruction i when it is preceded by the instruction j is negligible if compared with the average amount of power absorbed during the execution of a program. These contributions range from 5 to 30 mA as compared with 300–400 mA during execution of a program on the Intel 486DX2, and are below 20 mA on the Fujitsu SPARClite as compared with a range of 250–400 mA. With the DSP, on the other hand, the overhead is not negligible and has to be taken into account.

2.1. Application to the ST20-C2P

Equation (1) was adapted for application to the ST20-C2P, which is a *core processor*, *i.e.*, a processor that does not exist as a single component but is the core of a microprocessor-based *embedded system*. It is used in a range of applications from the cheapest portable embedded systems to more exacting applications with the performance requirements typical of DSPs [25]. It is a 32-bit processor, the control unit is micro-programmed, it has a two-stage pipeline (the first stage involving instruction fetch and decoding, the second execution and write back), and is manufactured using HCMOS7 technology, 0.25 μm, 2.5 V. The analysis was made considering the two main blocks making up the CPU the *fetch unit* and the *execution unit* separately. The *fetch unit* features average power absorption per cycle of about 6.6 mA with an average deviation of ±2 mA. The *execution unit* absorbs on average 30.3 mA ± 15 mA. It was therefore decided to label the *fetch unit* with a fixed cost of 6.6 mA and to treat the execution unit separately, as it is the main power consumer.

In general, the average amount of power absorbed by a processor in a given clock cycle depends on the instructions processed by the various pipeline stages in that clock cycle. If we take the ST20-C2P as a reference and define the clock cycle in which the instruction I_i is executed as the cycle in which the instruction I_i is processed by the last stage in the pipeline, we can write:

$$Cost(I_i) = base(I_i) + (I_{i-1} \rightarrow I_i)_{back} + (I_i \rightarrow I_{i+1})_{forw} \quad (2)$$

$Cost(I_i)$ indicates an estimate of the average amount of power absorbed during execution of the instruction I_i. The term $base(I_i)$ represents the basic cost of the instruction I_i, *i.e.*, the average amount of power absorbed by the processor during execution of instruction I_i when the latter is preceded by the same instruction I_i, or more generally, when all the stages in the pipeline are processing the instruction I_i. The terms $(I_{i-1} \rightarrow I_i)_{back}$ and $(I_i \rightarrow I_{i+1})_{forw}$, which we will call the *backward* and *forward costs* respectively, represent the inter-instruction contribution, *i.e.*, an excess added to the basic cost due to the fact that the instruction currently being executed is preceded and/or followed by a different instruction. The former is the inter-instruction cost in the execution stage while the latter is the inter-instruction cost in the fetch and decoding stage (see Section 3.4). We will indicate this model as ACM (*average cost model*).

2.1.1. Explanation of Inter-instruction Costs

When an instruction I_i is preceded by an instruction I_{i-1} ($I_i \neq I_{i-1}$) or followed by an instruction I_{i+1} ($I_i \neq I_{i+1}$), its cost exceeds that of $base(I_i)$. This excess is called the inter-instruction cost.

It can be seen as the sum of two contributions: a backward cost and a forward cost. To see how much the inter-instruction cost affects the basic cost of an instruction, consider Table I. The first column shows the instruction executed, the second column the average amount of power absorbed by the execution unit, the third the average amount of power absorbed by the micro-controller block

TABLE I How much the inter-instruction cost affects the basic cost of an instruction

Instruction	I_{eu} (mA)	$I_{\mu C}$ (mA)	overhead %
ldnlp	15.209	10.885	
stl	12.994	3.5719	85%
stl	7.0202	2.6162	0%
stl	31.801	27.223	353%
add	12.212	3.3052	

alone, and the fourth the percent increase in power absorbed when an instruction is followed or preceded by a different instruction as compared to when it is preceded or followed by the same instruction. As can be seen, an `stl` [25] followed and preceded by the same `stl` causes an absorption of about 7 mA in the *execution unit*, 2.6 mA of which are absorbed by the micro-controller alone. The first `stl` preceded by an `ldnlp`, on the other hand, absorbs about 13 mA, causing an overhead of 85% as compared with the previous case, due to the change in configuration. The last `stl` followed by an `add` causes an absorption of about 32 mA with an overhead of 353% as compared with the first case discussed. In this last case, the excess is all concentrated in the micro-controller block (86% of the power absorbed by the *execution unit* in this case is concentrated in the micro-controller). Later on it will be seen in the case study that the average amount of power absorbed by the execution unit during execution of an instruction greatly depends on the data it is processing. The values given in Table I are there not absolute values but can vary considerably along with variations in the data involved. The values given were measured in particular conditions of null activity (the concept of null activity will be dealt with in further detail below).

A great inter-instruction effect was also observed in [3] for a proprietary Fujitsu DSP in which the overhead was in some cases over 25 mA when variation of the power absorbed during execution of a program was distributed in a range between 20 and 60 mA. For more complex architectures like the Intel 486DX2 and the Fujitsu SPARClite, these effects were considered to be negligible as compared with the basic cost: in the worst case the excess was no higher than 10% of the basic cost ([3, 4]).

As for many microprocessors, in the ST20-C2P functioning at the circuit level is controlled by a micro-programmed sequence of instructions stored in a ROM in the CPU. Retrieval of the instructions in machine language and decoding causes the execution of a micro-program. The latter controls all the activity of the signal lines and all the data transfer or operations in the CPU during execution of the instruction. By way of example, let us refer to the execution of three instructions, I_{i-1}, I_i, I_{i+1}, and consider the clock cycle in which the last stage is processing instruction I_i. In this case, $(I_{i-1} \rightarrow I_i)_{back}$ is due to the change in the configuration of the execution stage which was processing instruction I_{i-1} in the previous clock cycle. $(I_i \rightarrow I_{i+1})_{forw}$, on the other hand, is due to the change in the configuration of the fetch/decode stage that was processing instruction I_i during the last cycle, while it is currently processing instruction I_{i+1}.

2.1.2. *Determination of Costs*

Application of the model described by Eq. (2) requires the basic and inter-instruction costs to be determined. The basic cost of instruction I_i is determined by constructing a test entailing repetition of instruction I_i and making it operate on data uniformly distributed within the admissible range of variation. To clarify this concept, let us assume that the instruction being investigated is `Inst n`, operating on an explicit 32-bit operand. The test to calculate the cost *base*(*Inst*) will be:

```
Inst  N1
Inst  N2
Inst  N3
...
Inst  Nn
```

where $N_i \in$ [0x00000000, 0xFFFFFFFF].

Having obtained the average current absorbed by each instance of the instruction `Inst n` (we say

$c_1, c_2, \ldots, c_n$) we have:

$$base(Inst) = \frac{1}{n}\sum_{k=1}^{n} c_k \tag{3}$$

To determine the inter-instruction costs $(I_{i-1} \rightarrow I_i)_{back}$ and $(I_i \rightarrow I_{i+1})_{forw}$ it is sufficient to construct a test in which a sequence of I_i is embedded between a pair of instructions I_{i-1} and I_{i+1}. The first occurrence of I_i will contribute towards determining the cost $(I_{i-1} \rightarrow I_i)_{back}$, while the last will determine the cost $(I_i \rightarrow I_{i+1})_{forw}$.

To make this even clearer, let us consider the following code fragment:

```
...
1. I(k) i-1
2. I(k) i
3. I(k) i
4. I(k) i+1
...
```

Let us assume that the sequence is repeated N times ($k \in \{1, 2, \ldots, N\}$), keeping the operands of I_i uniformly distributed. Let $c_i^{(k)}$ be the power absorbed by the i-th instance of the sequence at the k-th repetition. We therefore define:

$$(I_{i-1} \rightarrow I_i)_{back} = \sum_{k=1}^{N} \frac{c_2^{(k)} - base(I_i)}{N} \tag{4}$$

$$(I_{i-1} \rightarrow I_i)_{forw} = \sum_{k=1}^{N} \frac{c_3^{(k)} - base(I_i)}{N} \tag{5}$$

A basic hypothesis for characterisation of the instruction set using the ACM is the possibility of creating sequences formed by the same instruction (the one to be characterised) operating on uniformly distributed data. In the architectures taken as references in [1–4, 11] most of the instructions have an explicit argument, in which case the structure of the test to determine the basic cost of a generic instruction `Inst` can be as follows:

```
Inst n1
Inst n2
...
Inst nn
```

where n_i are uniformly distributed. The ST20-C2P has a load/store stack-based architecture with a stack comprising three registers. Most of the instructions work on the data stored in the registers, so it is necessary to alternate register loading with the repetition of the instructions. As the execution of an instruction causes stack rotation, making the last register indefinite, the maximum length of the sequence is three instances: during execution of the second instance the measurement is made, while the first and third instances give the measure without the inter-instruction effects. The test to determine the basic costs for instruction `Inst` will be structured as follows:

```
load registers
Inst
Inst
Inst
...
```

In this case the amount of power absorbed has to be measured as corresponding to the framed instruction, so we need measurement tools that make it possible to conduct an analysis at the instruction level. From an operational point of view, the use of a simulation tool provides a cycle-accurate analysis but the simulation takes a long time. In this case, in fact, to determine the basic cost of an instruction it is necessary to take N (in [11], for example, $N = 100$) measures, which require the execution of N sequences of this kind.

Characterisation of the model (*i.e.*, determination of the basic and inter-instruction costs) requires a tool to measure the amount of power absorbed by the processor during execution of a software test. The approach adopted in [1–4] was to use an ammeter in series between the electric power supply pin of the processor and the power source. Although this method has the advantage of allowing measurements to be taken rapidly, it presents a number of problems: it supplies average information concerning a time window in which

various instructions are executed, and it requires availability of the processor as an integrated component. In many cases, as in the hardware/software codesign of an embedded system for example, the silicon of the core alone does not exist and the only way to measure the power absorbed by the core is to use a simulation tool for power analysis. As the ST20-C2P is a core processor, *i.e.*, a processor that does not exist as a single component, a transistor-level simulator, Powermill, was used as a measurement tool.

Figure 1 shows the flow of operations performed to construct the database of basic and inter-instruction costs, to estimate the code and make a comparison with the results of the electrical simulation. The program being analysed was used

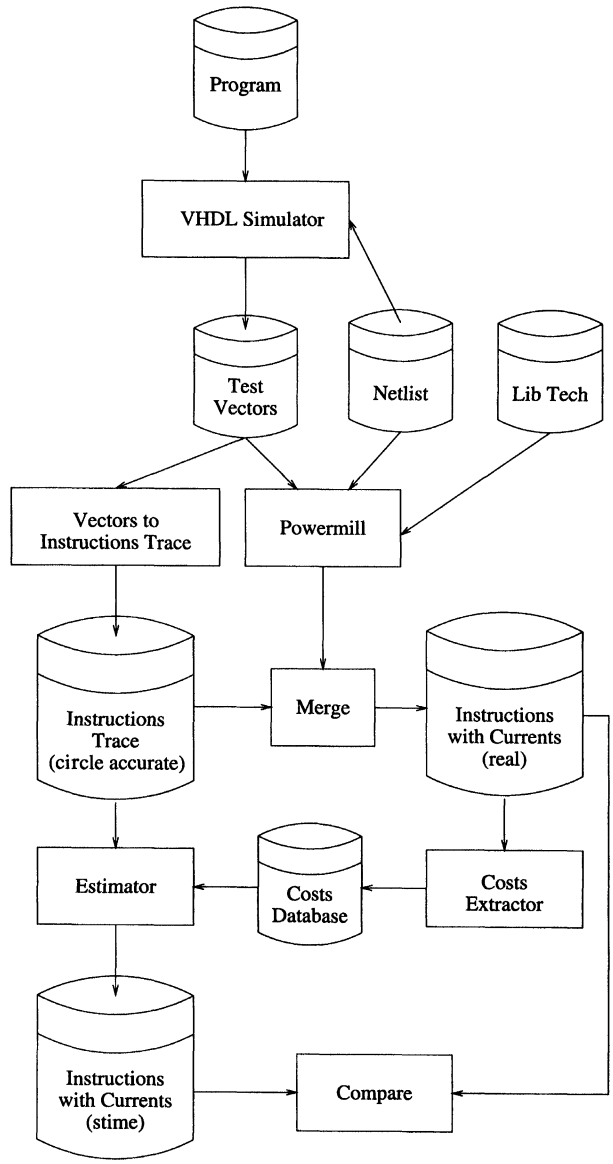

FIGURE 1 Flow of operations performed to construct the database of basic and inter-instruction costs.

as input for VHDL simulation, to generate stimuli for simulation with Powermill. VHDL simulation also provides the signals needed to generate a cycle-accurate trace of the flow of instructions executed. The results of the Powermill simulation are combined with the trace of the program to generate an instruction trace file including power information. This file is processed to extract any basic and inter-instruction costs, which will be stored in a database. Together with the trace of the instructions executed, this will represent the input for the estimation program, the results of which will be compared with those obtained in the Powermill simulation.

2.1.3. Results of Estimation

In [1] the ACM gave good results when used to estimate the power consumption of two commercially available processors, the Intel 486DX2 and the Fujitsu SPARClite 934, in which a low data-dependency was observed and the hypothesis of labelling instructions with an average value was confirmed. In [2] the same authors then applied the ACM to a DSP, noticing that in this case data-dependency is greater and the inter-instruction effect more marked. These effects are always present, but their contribution in terms of the fraction of power absorbed grows in an inversely proportional fashion to the architectural complexity of the processor. In DSPs with simple architectures, these effects are much more evident. In [1] the range of variation on the basic cost as the data varied always remained below 5% for the Intel 486DX2. In [3], on the other hand, a greater variation was observed for a proprietary Fujitsu DSP, but it remained lower than 10%.

With the core ST20-C2P we observed that consumption strongly depended on data. To highlight the contribution of data to power consumption, Figure 2 shows the average amount of power absorbed by the *execution unit* cycle by cycle during execution of a sequence of `add` instructions operating with null arguments

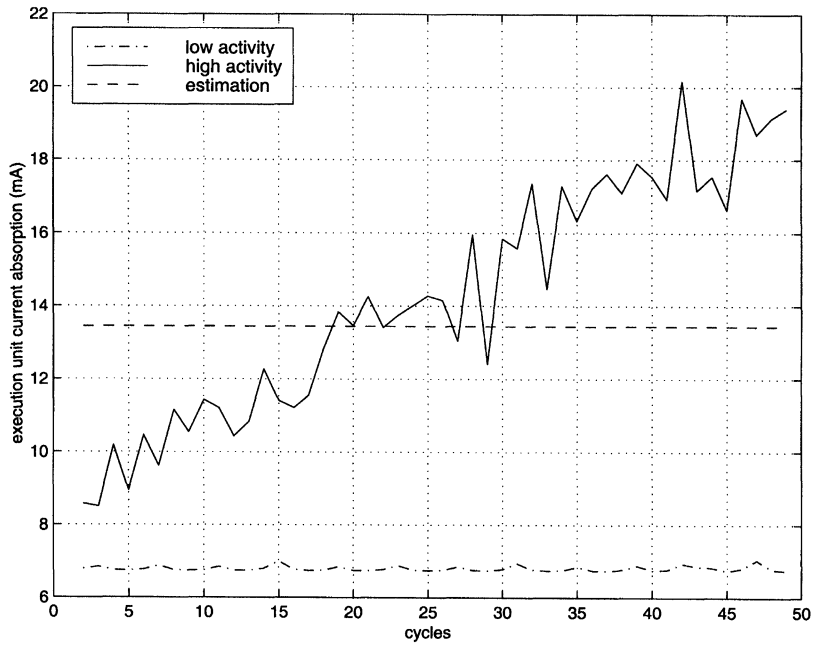

FIGURE 2 Average amount of power absorbed by the *execution unit* cycle by cycle during execution of a sequence of `add` instructions operating with null arguments and gradually increasing arguments.

(dashed and dotted curve) and gradually increasing arguments (continuous-line curve). The average amount of power absorbed per cycle ranges between 7 and 20 mA, a band of 13 mA that corresponds to 185% of the minimum absorption, while in terms of power there is an increase of 78% entirely due to the effect of the data. Use of the ACM to estimate this code (dashed curve) would mean a mean error per cycle of 98% in the low-activity test estimate and 20% in the increasing activity estimate.

In our case study, the data-dependency of power consumption is far from negligible. The range of variation of the amount of power absorbed during execution of an instruction with varying amounts of data is on average 150% more than the lower end of the range.

3. A DATA-DEPENDENT MODEL

In Section 2.1.3 it was shown that use of the ACM to estimate the average amount of power absorbed by an instruction leads to considerable error with simple processor architectures. It is therefore not appropriate to label an instruction with an average value: we need a model that is data-dependent as well [12]. In the following sections we will present a data-dependent estimation model as a generalisation of the ACM, simply separating the basic and inter-instruction costs from the contribution made by data. The model will be adapted in order to apply it to the ST20-C2P and will be validated by estimating some software tests and comparing the results with those obtained using the ACM.

3.1. Formulation

We will initially refer to the simple architecture of the core processor which represents our case study. Architectures of this kind are commonly used in embedded systems in which memory area and power consumption constraints are more important than performance. The model will then be extended to consider RISC architectures with a greater pipeline depth.

Given the sequence of instructions I_{i-1}, I_i and I_{i+1}, the average amount of power absorbed during execution of the instruction I_i can be estimated by using the following equation:

$$\begin{aligned} Cost(I_i) = {} & base_F^{(0)}(I_{i+1}) + int_F^{(0)}(I_i, I_{i+1}) + \\ & + base_E^{(0)}(I_i) + int_E^{(0)}(I_{i-1}, I_i) \\ & + f(data) \end{aligned} \tag{6}$$

Considering constant the power absorbed by the *fetch unit* ($base_F^{(0)}(I_i) = K \; \forall \; I_i$), given $base^{(0)}(I_i) = base_E^{(0)}(I_i) + K$ and indicating the inter-instruction costs in the execution and fetch stages as $(I_{i-1} \rightarrow I_i)_{back}^{(0)}$ and $(I_i \rightarrow I_{i+1})_{forw}^{(0)}$ respectively, we have:

$$\begin{aligned} Cost(I_i) = {} & base^{(0)}(I_i) + (I_{i-1} \rightarrow I_i)_{back}^{(0)} + \\ & + (I_i \rightarrow I_{i+1})_{forw}^{(0)} + f(data) \end{aligned} \tag{7}$$

The estimation function is seen as the sum of three contributions: that depending on the instruction being examined, the inter-instruction contribution (the sum of the effect of the change in configuration and the determination of the subsequent state) depending on the instruction preceding and following the one being executed, ad lastly the contribution made by the data. The three addends are filtered by their mutual effects: the basic and inter-instruction costs are determined in a state of null activity and the contribution of the data is filtered by the basic and inter-instruction costs.

3.1.1. Basic Cost in a State of Null Activity

The meaning of the terms appearing in Eq. (7) is made clearer by Figure 3, which shows part of the datapath of a generic processor, comprising three components: the registers, the datapath buses and the logical combiner blocks. The first contain the data to be processed, while the second act as channels to transfer the data to the combiner

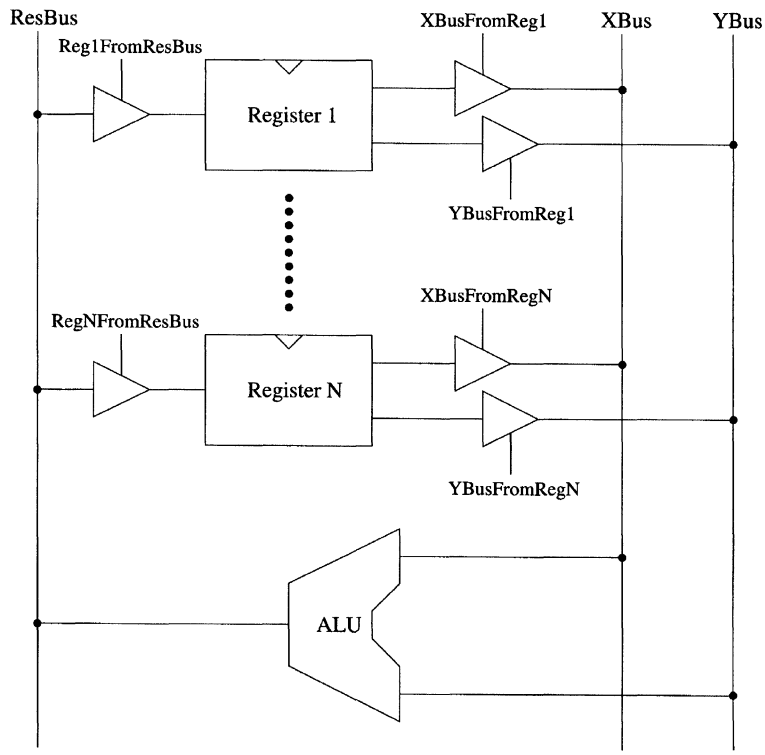

FIGURE 3 Part of the datapath of a generic processor.

blocks. Figure 3 shows N registers connected to the two datapath buses (*xbus and ybus*) by three-state buffers. *xbus* and *ybus* are connected to the ALU, the sub-blocks of which (adder, multiplier, shifter, comparator, *etc.*) perform arithmetical and logical functions on the data transported by *xbus* and *ybus*. All the blocks have the same input and all output a result. A multiplexer will only select the output of the correct block, loading the *resbus* with the result of the operation requested.

The basic cost of the instruction I_i in a null activity state ($base^{(0)}(I_i)$) is defined as the amount of power absorbed during execution of the instruction I_i when it is preceded and followed by the same instruction I_i and null activity is maintained on the buses and registers. As execution of an instruction translates into correct mapping of the registers on *xbus* and *ybus*, by maintaining activity on the registers null, the activity on the bus is also null.

For the ST20-C2P the basic null-activity cost was around 7.2 mA per instruction. The reason for this behaviour is mainly the absence of *operand isolation techniques*: during execution of an instruction all the blocks are working and producing a result; the only difference between two different instructions is selection of the output of one block rather than that of another. If, for example, we focus on the ALU block during execution of an add instruction, all the blocks (adder, comparator, shifter *etc.*) receive the same data and they all output a result, but only the adder output is selected. This therefore accounts for the similarity between the basic costs.

Table II gives the basic null-activity costs for 36 instructions. As expected, the null-activity basic costs for the various instructions are pretty much the same. The average deviation of the null-activity basic costs from the average value is less than 7%.

TABLE II Basic null-activity costs for 36 instructions (values in mA)

Instruction	Current (mA)	Instruction	Current (mA)
adc	6.8462	ldnlp	6.7553
add	6.8055	ls	7.1600
and	6.7426	lsx	7.4301
bcnt	6.7660	mint	8.1546
bsub	6.8478	nop	5.8913
cj	8.3284	or	7.1834
diff	6.7552	pop	8.4912
dup	6.8469	rev	6.8016
eqc	8.2784	shl	6.8044
gt	6.8644	shr	6.7221
gtu	8.4379	ssub	6.8047
lb	7.2378	stabs	7.0677
lbx	7.4530	stl	7.0202
ldabs	6.9215	sub	6.7981
ldc	6.7322	sum	7.0246
ldinf	8.9764	wsub	6.7306
ldl	8.0208	wsubdb	6.7326
ldlp	6.9280	xor	6.8684

3.1.2. *Null-activity Inter-instruction Cost*

The cost $(I_{i-1} \rightarrow I_i)^{(0)}_{back}$ is defined as the overhead on the basic null-activity cost of instruction I_i when the latter is preceded by the same instruction I_{i-1} and followed by the instruction I_i and activity on the registers and buses remains null. This excess is exclusively due to the effects of the change in configuration. Likewise, the cost $(I_i \rightarrow I_{i+1})^{(0)}_{forw}$ is defined as the overhead on the basic null-activity cost of the instruction I_i when it is preceded by the same I_i and followed by the instruction I_i and activity on the registers and buses remains null. This excess is exclusively due to the micro-controller that determines the subsequent state or, more generally, to the decoding logic.

Tables III and IV give the backward and forward costs for a number of pairs of instructions. The variation in backward costs as compared with the basic cost is on average less than 25%. For forward costs, on the other hand, the deviation is much greater: on average over 100%. Note also that the symmetry hypothesis put forward in [2] is not always valid in our case. The lack of symmetry in forward costs is due to glitches that occur at the micro-controller ROM inputs for particular pairs of instructions. See, for example, the forward costs for the pairs (*and*; *add*), (*and*; *sub*) and (*and*; *rev*), where this asymmetry is quite evident.

3.1.3. *Data Contribution*

The term in 7 indicated as *f*(*data*) takes into account the contribution of data towards the amount of power absorbed during execution of the instruction I_i. Execution of this instruction means activating the appropriate combiner blocks, the power absorbed by which depends on how their respective inputs vary. Let us indicate a metre whereby we can measure these variations with the term activity. The average power absorbed in a cycle by the generic block $Block_j$ depends on the activity of its inputs:

$$Current_{Block_j} = function(activity_j)$$

TABLE III Backward costs $(I_{row} \rightarrow I_{col})^{(0)}_{back}$. (values in mA)

	ldc	adc	ldl	and	ldlp	xor	shr	add	sub	rev
ldc	0	2.2358	2.8322	5.7584	1.7709	6.3066	4.1149	5.5035	10.4929	1.7647
adc	2.4086	0	3.5152	4.1824	2.5392	4.7116	5.6289	3.5985	8.8789	1.8869
ldl	4.3438	5.6238	0	6.4104	4.3830	6.7266	7.0199	6.5295	11.5839	2.6096
and	4.3188	2.7828	3.7732	0	6.2490	0.1607	2.4895	0.4680	5.7339	2.5358
ldlp	2.0811	2.9153	3.3912	6.3754	0	6.7066	6.3449	5.9925	10.9109	3.8214
xor	4.4088	2.9989	4.0482	0.2301	4.5960	0	2.7095	0.5129	5.9469	2.7532
shr	3.4078	5.2868	4.6662	2.7419	5.3490	2.8250	0	2.8203	8.1259	5.2404
add	4.2078	2.4870	3.7092	0.4657	4.2790	0.4109	2.7327	0	5.3209	2.5789
sub	7.3828	5.8088	6.5782	2.2959	7.8270	2.3966	4.5909	3.3885	0	5.3054
rev	2.5458	2.7739	3.6812	3.4064	2.6570	3.2926	3.9279	3.1623	8.5449	0

TABLE IV Forward $(I_{row} \rightarrow I_{col})^{(0)}_{forw}$. (values in mA)

	ldc	adc	ldl	and	ldlp	xor	shr	add	sub	rev
ldc	0	13.2088	1.9298	16.2518	9.7448	27.5108	19.5178	21.2688	25.1338	24.2858
adc	13.7068	0	15.5348	29.3388	9.6738	29.4628	21.9638	32.6258	27.8898	25.2718
ldl	2.5052	18.3832	0	16.1212	8.7242	25.7692	22.7282	21.5772	28.7982	27.4252
and	20.3214	29.2884	18.6504	0	22.7614	27.9834	11.1494	12.1174	17.8604	17.0164
ldlp	10.0940	9.6260	12.9450	22.9130	0	24.6580	17.7420	29.5460	34.7740	23.9830
xor	28.9366	30.6096	28.3086	27.4136	23.5486	0	23.1136	21.2956	25.6796	15.0986
shr	22.2029	21.1129	24.6759	10.7869	16.7359	23.5839	0	18.999	22.3789	7.9969
add	24.5395	34.4555	24.8345	34.7765	25.5405	34.8075	42.1455	0	9.4955	9.8205
sub	27.4709	27.7049	29.4309	47.2159	32.9449	37.6149	43.8209	9.8419	0	13.1249
rev	30.0924	28.0094	30.3144	41.8314	24.0494	30.5464	33.4654	10.5334	15.6304	0

Let:

$$Block^{(I_i)} = [Block_1^{(I_i)}, \ldots, Block_{NB_{I_i}}^{(I_i)}]$$

and

$$activity^{(I_i)} = [activity_1^{(I_i)}, \ldots, activity_{NB_{I_i}}^{(I_i)}]$$

respectively be the blocks activated during execution of the instruction I_i and the activity at the inputs of these blocks. We can then write:

$$f(data) = \sum_{j=1}^{NB_{I_i}} Current_{Block_j^{(I_i)}}(activity_j^{(I_i)}) \quad (8)$$

Adapting to the ST20-C2P, which has no operand-isolation functions, execution of an instruction I_i means activating all the blocks, so we have:

$$NB_{I_i} = NB \; \forall \; i$$

therefore substituting in 8 we get:

$$f(data) = \sum_{j=1}^{NB} Current_{Block_j}(activity_j) \quad (9)$$

Going back to Figure 3, it is clear that the activity at the inputs of a block depends on the activity on the datapath buses (which represent the inputs of the combiner blocks):

$$activity_j = function(activity_{Xbus}, activity_{Ybus})$$

The activity induced on the datapath buses depends on the mapping of the registers of the current and previous instruction. If we indicate the registers mapped on *xbus* and *ybus* when instruction I_i is executed as $RegX_i$ and $RegY_i$ respectively, we have:

$$activity_{Xbus} = function(RegX_{i-1}, RegX_i)$$
$$activity_{Ybus} = function(RegY_{i-1}, RegY_i)$$

So, in conclusion:

$$f(data) = function(RegMap_{i-1}, RegMap_i) \quad (10)$$

where

$$RegMap_j = [RegX_j, RegY_j]$$

represents the register values that instruction I_j maps on *xbus* and *ybus*.

Equation (10) can be generalised to account for other activity indexes such as activity on the address and data lines interfacing the memory. Proceeding in the same way, if we indicate the registers mapped on the memory address and data buses when instruction I_j is executed as $RegMA_j$ and $RegMW_j$ respectively, Eq. (10) still holds:

$$RegMap_j = [RegX_j, RegY_j, RegMA_j, RegMW_j]$$

3.2. Application to a Case Study

The model defined by Eq. (7) was applied to the ST20-C2P.

3.2.1. *Definition of f(data)*

It was found that the activity induced on the datapath buses (*xbus* and *ybus*) and on the *memaddr* bus was the greatest power consumer in the ST20-C2P. These buses were therefore chosen as *activity indexes*. To measure this activity, the *Hamming distance* between two successive configurations of the index being considered was chosen. This confirms the hypothesis that the power absorbed depends on the number of variations and is independent of the lines on which the variations occur. To define $f(data)$ a linear model was chosen in which the activity of the various indexes is weighted with a different coefficient.

Given these hypotheses, Eq. (10), as adapted for the C2P, becomes:

$$\begin{aligned} f(data) = {} & Hamming(xbus_{i-1}, xbus_i) \\ & \times weight_{xbus} + \\ & + Hamming(ybus_{i-1}, ybus_i) \\ & \times weight_{ybus} + \\ & + Hamming(memaddr_{i-1}, memaddr_i) \\ & \times weight_{memaddr} \end{aligned} \tag{11}$$

We will now describe the method used to determine the weights and confirm the linearity hypothesis.

3.2.2. *Determination of Weights*

By suitable loading the registers and knowing the register/activity index mapping, it is possible to construct sequences of instructions in such a way as to induce guided activity on the activity indexes. To calculate the weights appearing in Eq. (11) a test was constructed in such a way as to render the activity on two indexes fixed, making only one of them variable. In this way the contribution of the activity on one index is isolated and it is possible to define a relation that links the activity on the index with the power absorbed.

Let $\bar{c} \in \Re^{96 \times 1}$ be a column vector split into 3 components with 32 elements: the elements of the first component $\bar{c}(c(1:32))$ represent the power absorbed by the *execution unit* when there are $1,2,\ldots,32$ transitions on *xbus* and zero transitions on the remaining activity indexes. The elements of the second component of $\bar{c}(c(33:64))$ represent the power absorbed by the *execution unit* for $1,2,\ldots,32$ transitions on *ybus* and zero transitions on the remaining activity indexes. Likewise for the third component, *memaddr*.

Given:

$$\bar{c} = \begin{bmatrix} c_1 \\ c_2 \\ \ldots \\ c_{32} \\ c_{33} \\ c_{34} \\ \ldots \\ c_{64} \\ c_{65} \\ c_{66} \\ \ldots \\ c_{96} \end{bmatrix} \quad M = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ \ldots & \ldots & \ldots \\ 32 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 0 \\ \ldots & \ldots & \ldots \\ 0 & 32 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 2 \\ \ldots & \ldots & \ldots \\ 0 & 0 & 32 \end{bmatrix}$$

$$\overline{weight} = \begin{bmatrix} weight_{xbys} \\ weight_{ybus} \\ weight_{memaddr} \end{bmatrix}$$

we can write:

$$\bar{c} = M \times \overline{weight} \tag{12}$$

Equation (12) is a system of 96 equations with the three unknown values $weight_{xbus}$, $weight_{ybus}$ and $weight_{memaddr}$. By applying the least squares method it is possible to obtain the vector $\overline{weight}$

minimising the quadratic error.

$$\overline{weight} = (M^T M)^{-1} M^T \bar{c} \tag{13}$$

A more general method is to use a lookup table for each activity index. Each table is addressed by measuring the activity on the activity index being examined and gives the average amount of power absorbed.

3.2.3. *Linearity Approximation*

The adequacy of the linearity approximation linking the amount of power absorbed to the activity on the activity indexes is evident in Figure 4, which shows the power absorbed by the *execution unit* with varying amounts of activity on *xbus* and *ybus*. As can be seen, the activity induced on *xbus* leads to a greater power dissipation. The difference can be accounted for by the fact that *xbus* has more branches than *ybus* and propagates its activity to a greater number of blocks, thus entailing a higher cost per transition. It was also observed that the weight of $0 \rightarrow 1$ transitions was different from that of $1 \rightarrow 0$ transitions. Table V gives the costs per transition determined by the least squares method, discriminating between $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions.

Equation 11 was extended so as to treat $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions differently:

$$\begin{aligned} f(data) = {} & Hamm^{0\rightarrow 1}(xbus_{i-1}, xbus_i) \\ & \times weight^{0\rightarrow 1}_{xbus} + \\ & + Hamm^{1\rightarrow 0}(xbus_{i-1}, xbus_i) \\ & \times weight^{1\rightarrow 0}_{xbus} + \\ & + Hamm^{0\rightarrow 1}(ybus_{i-1}, ybus_i) \\ & \times weight^{0\rightarrow 1}_{ybus} + \\ & + Hamm^{1\rightarrow 0}(ybus_{i-1}, ybus_i) \\ & \times weight^{1\rightarrow 0}_{ybus} + \\ & + Hamm(memaddr_{i-1}, memaddr_i) \\ & \times weight_{memaddr} \end{aligned} \tag{14}$$

where the function $Hamm^{x\rightarrow y}(\bar{a}, \bar{b})$ gives the number of transitions $x \rightarrow y$ involved in passing from configuration $\bar{a}$ to configuration $\bar{b}$.

The activity indexes will foreseeably shift from the combinatorial elements towards the interconnection buses as the scale of integration increases. The wire-to-gate capacity ratio has gone from 3 for old technologies to 100 for new technologies.

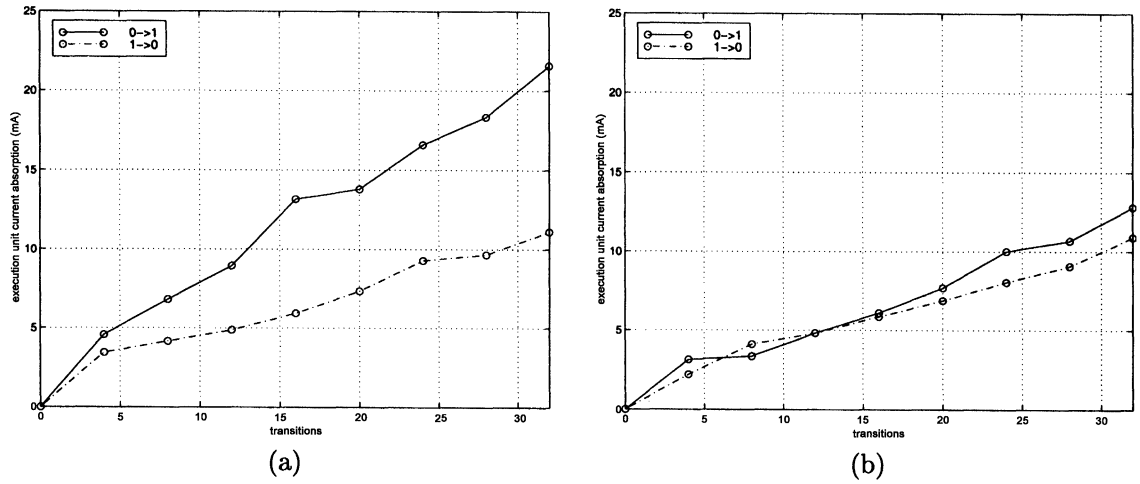

FIGURE 4 Current absorbed by the *execution unit* with varying amounts of activity on *xbus* in (a) and *ybus in* (b).

TABLE V Costs per transition determined by the least squares method (mA/transition)

	Transitions	
	$0 \rightarrow 1$	$1 \rightarrow 0$
xbus	0.6966	0.3665
ybus	0.3125	0.2000
memaddr	0.2500	

This means that the interconnection buses are of greater importance, in relation to power consumption, than combinatorial logic and so it is of fundamental importance to focus on the power dissipated due to activity on the buses.

3.2.4. *Determination of Glitches*

To apply Eq. (14) it is necessary to have precise knowledge of the transitions on the activity indexes. Let us consider the following code fragment:

```
ldc 0x00000000
ldc 0xffffffff
ldc 0x00000000
add
ldc 0xffffffff
ldc 0
```

Let us assume that we wish to determine the contribution of data, in terms of power absorbed, following execution of the instruction `ldc 0xfff ffffff`. To determine the activity on *xbus* and *ybus* Table VI shows the mapping of the registers on the datapath buses in relation to the instructions `ldc`

TABLE VI Mapping of the registers on the datapath buses

Instruction	*xbus*	*ybus*
ldc n	0	n
add	*AReg*	*BReg*

and `add`. From Table VII we obtain:

$$Hamm^{0\rightarrow 1}(xbus_{add}, xbus_{ldc}) = 0$$
$$Hamm^{1\rightarrow 0}(xbus_{add}, xbus_{ldc}) = 0$$
$$Hamm^{0\rightarrow 1}(ybus_{add}, ybus_{ldc}) = 0$$
$$Hamm^{1\rightarrow 0}(ybus_{add}, ybus_{ldc}) = 0$$

from which we conclude that according to Eq. (14) the data contribution is null.

The power measured during execution of the `ldc 0xffffffff` is 62 mA, while the estimated amount is 11 mA, thus giving an underestimation error of 82%. The estimation error was due to the way in which the transitions were counted. The enable signals of the three-state buffers for the mapping of the registers on the datapath buses come from the micro-controller. As the length of the paths is different, these selection signals arrive after the contents of the registers have changed.

Figure 5 shows the real transitions on *xbus* and *ybus*. During execution of the instruction `add`, *xbus* and *ybus* are loaded with the contents of the registers *AReg* and *BReg* respectively. On the rising edge of the next cycle the contents of the registers will be updated with the result of the add operation and the datapath buses will be loaded with the new contents of the registers *AReg* and *BReg*, generating 32 $0 \rightarrow 1$ transitions on *xbus* and 32 $1 \rightarrow 0$ transitions on *ybus*. After a delay T following the rising edge of the clock, the selection signals will map the registers *ConstantReg* and *0Reg* on *xbus* and *ybus*, thus determining another 32 $1 \rightarrow 0$ transitions on *xbus* and 32 $0 \rightarrow 1$ transitions on *ybus*.

Indicating the contents of the register that instruction I_{i-1} maps on *xbus* during execution of the instruction I_i as $RegX_{i-1}^{(i)}$, the total number of transitions on *xbus* when instruction I_i, preceded

TABLE VII Then the data contribution is null

Instruction	*AReg*	*BReg*	*CReg*	*xbus*	*ybus*
`add`	0	0xffffffff	0	0	0xffffffff
`ldc 0xffffffff`	0xffffffff	0		0	0xffffffff

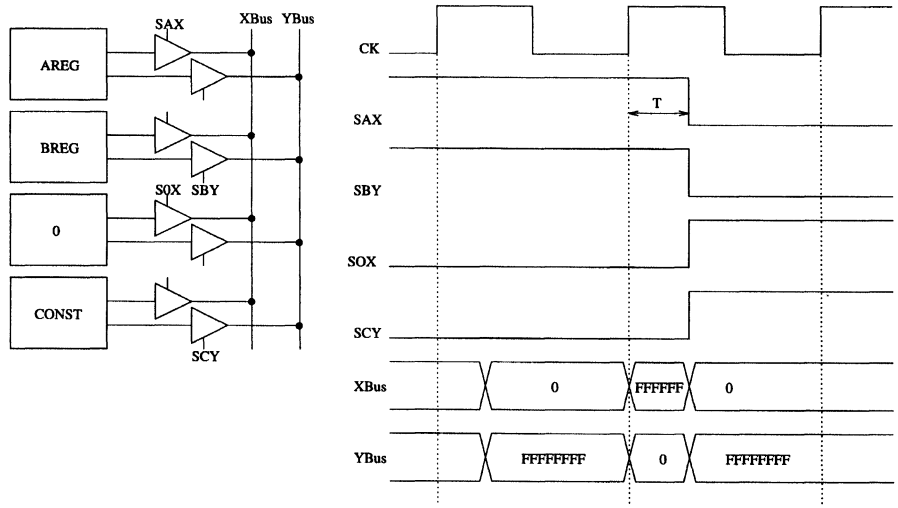

FIGURE 5 Real transitions on *xbus* and *ybus*.

by instruction I_{i-1}, is executed is:

$$Hamm(xbus_{i-1}, RegX_{i-1}^{(i)}) + Hamm(RegX_{i-1}^{(i)}, xbus_i)$$

Indicating:

$$\begin{aligned} transX^{a\to b}(I_{i-1}, I_i) &= Hamm^{a\to b}(xbus_{i-1}, RegX_{i-1}^{(i)}) + \\ &+ Hamm^{a\to b}(RegX_{i-1}^{(i)}, xbus_i) \\ transY^{a\to b}(I_{i-1}, I_i) &= Hamm^{a\to b}(ybus_{i-1}, RegY_{i-1}^{(i)}) + \\ &+ Hamm^{a\to b}(RegY_{i-1}^{(i)}, ybus_i) \end{aligned}$$

Equation (14) becomes:

$$\begin{aligned} f(data) = {} & transX^{0\to 1}(I_{i-1}, I_i) \times weight_{xbus}^{0\to 1} + \\ & + transX^{1\to 0}(I_{i-1}, I_i) \times weight_{xbus}^{1\to 0} + \\ & + transY^{0\to 1}(I_{i-1}, I_i) \times weight_{ybus}^{0\to 1} + \\ & + transY^{1\to 0}(I_{i-1}, I_i) \times weight_{ybus}^{1\to 0} + \\ & + Hamm(memaddr_{i-1}, memaddr_i) \\ & \times weight_{memaddr} \end{aligned} \quad (15)$$

3.3. Estimation Tests

Given the sequence of instructions I_{i-1}, I_i, I_{i+1}, the model used to estimate the average amount of power absorbed during execution of instruction I_i is:

$$\begin{aligned} Cost(I_i) = {} & base^{(0)}(I_i) + \\ & + (I_{i-1} \to I_i)_{back}^{(0)} \\ & + (I_i \to I_{i+1})_{forw}^{(0)} + \\ & + transX^{0\to 1}(I_{i-1}, I_i) \\ & \times weight_{xbus}^{0\to 1} + \\ & + transX^{1\to 0}(I_{i-1}, I_i) \\ & \times weight_{xbus}^{1\to 0} + \\ & + transY^{0\to 1}(I_{i-1}, I_i) \\ & \times weight_{ybus}^{0\to 1} + \\ & + transY^{1\to 0}(I_{i-1}, I_i) \\ & \times weight_{ybus}^{1\to 0} + \\ & + Hamm(memaddr_{i-1}, memaddr_i) \\ & \times weight_{memaddr} \end{aligned} \quad (16)$$

This model will be referred to as DDM (data dependent model). The DDM was validated and compared with the ACM by performing some

software estimation tests. These were divided into two classes: tests constructed ad hoc to highlight the features of the model proposed, and tests extracted from fragments of the codes of real applications. Each test was carried out using both the ACM and the DDM, calculating the average amount of power absorbed by the *execution unit* (real and estimated) per cycle (for the first 80 clock cycles) and the distribution of the mean percent error over the estimation cycle.

To show the influence of data on power consumption, a test was prepared comprising a sequence of 50 `add` instructions. This test was performed in two modes: low-activity (`sum_low`) and high-activity (`sum_high`). In the low-activity mode the 50 `add` instructions operate on null arguments, while in the high-activity mode the arguments gradually increase (see Section 2.1.3). In Figure 6 the solid line shows the real average power absorbed per cycle in the case of high and low activity, while the dashed line shows the estimated power absorption.

To test the robustness of the DDM to inter-instruction effects, a test (`rand_inst`) was built comprising a sequence of about 500 instructions extracted at random from the instruction set. The aim of this test was to create a preponderant inter-instruction effect which, as we have seen, is the greatest contributor to power consumption. The random distribution of the instructions avoids the presence of repeated sequences which might falsify the estimation results. Figure 7 gives the percent error distribution over the cycle using the ACM (a) and the DDM (b).

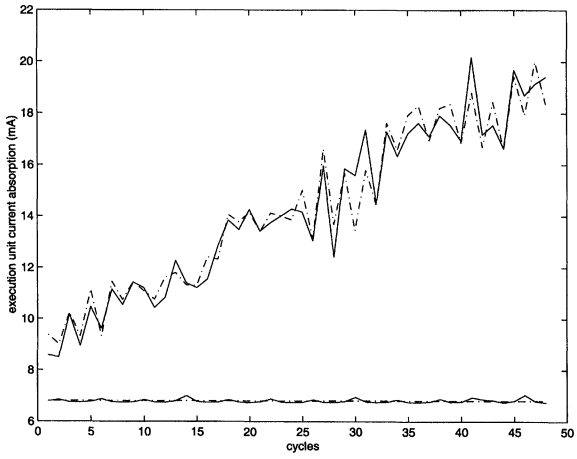

FIGURE 6 Real and estimated average power absorbed per cycle in the case of high and low activity for a test comprising a sequence of 50 `add` instructions.

Figures 8 and 9 shows percent error distributions using the ACM and the DDM for test which executes the sum (`mtx_sum`) and the product (`mtx_mul`) of two 3×3 matrices formed by random integer elements.

Figure 10 shows the real and estimated power absorbed by the execution unit in the first 80 cycles of execution of a test performing a digital Fourier transform (`dft`) on 16 signal samples. Figure 11 gives the percent error distributions over the cycle using the ACM (a) and the DDM (b).

Table VIII summarises the results of the tests described in this section. r and s respectively indicate the vectors of the real and estimated power per cycle. $E\{x\}$ indicates the average of the elements in the vector x. For each test carried out the table gives the average amount of power absorbed during execution, the percent estimation error and the average of the absolute error values obtained cycle by cycle. In all cases the error made using the DDM was lower than 6% in estimating the average power and less than 10% in the estimate per cycle. The accuracy of the ACM is not so easy to quantify as the error varies in a very wide range. For some highly data-dependent tests, the average error goes from 97.77% to 5.58% for varying amounts of input data. In the test estimate in which the average amount of power absorbed has a narrow range of variation for varying amounts of data, the average ACM estimate error is between 7% and 20% for the average amount of power absorbed and between 15% and 30% for the cycle-based estimate.

3.4. Generalisation for Pipelined Processors

The model described by 7 was specifically defined for our case study. In this section we will show that

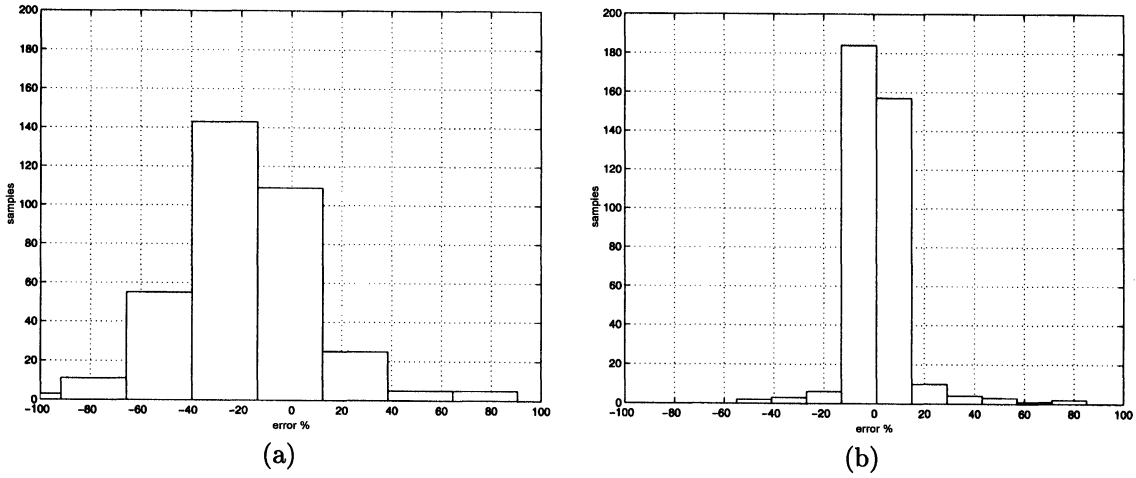

FIGURE 7 Percent error distribution over the cycle using the ACM (a) and the DDM, (b) for a test comprising a sequence of about 500 instructions extracted at random from the instruction set.

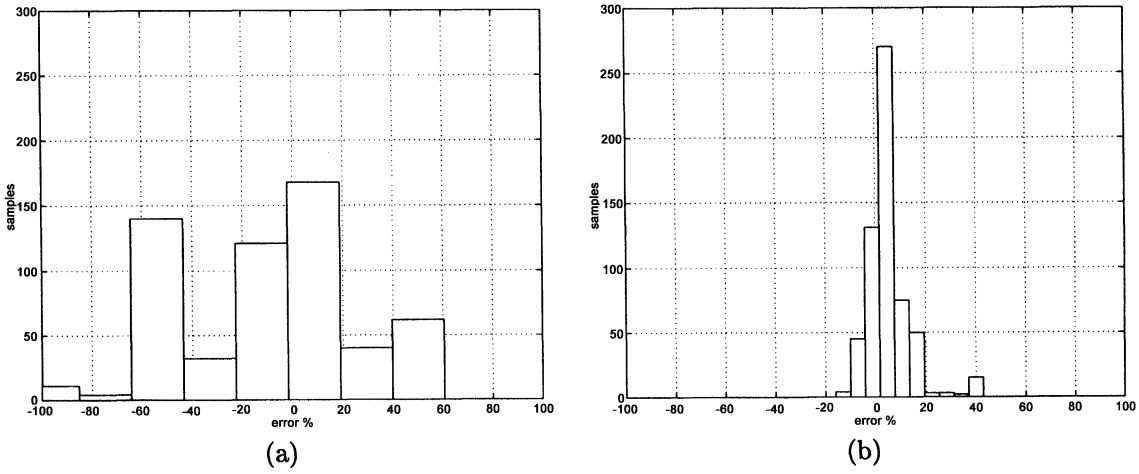

FIGURE 8 Percent error distribution over the cycle using the ACM (a) and the DDM, (b) for a test which executes the sum of two 3×3 matrices.

it is possible to generalise the model to apply it to generic RISC processors. Let us refer to the following simplifying hypothese:

- N pipeline stages (indicated as $S_1, S_2, \ldots, S_N$).
- Each stage processes instructions in a single clock cycle.
- Each instruction is processed by all the stages.

We will consider the pipeline configuration shown in Figure 12.

The average power absorbed per CK_M clock cycle is:

$$\begin{aligned} C_{CK_M} = {} & base_1^{(0)}(I_{N+1}) + int_1^{(0)}(I_N, I_{N+1}) + \\ & + F_1(\overline{activity_1}, I_{N+1}) + \\ & + base_2^{(0)}(I_N) + int_2^{(0)}(I_{N-1}, I_N) + \\ & + F_2(\overline{activity_2}, I_N) + \\ & + \cdots + \\ & + base_{N-1}^{(0)}(I_3) + int_{N-1}^{(0)}(I_2, I_3) + \end{aligned}$$

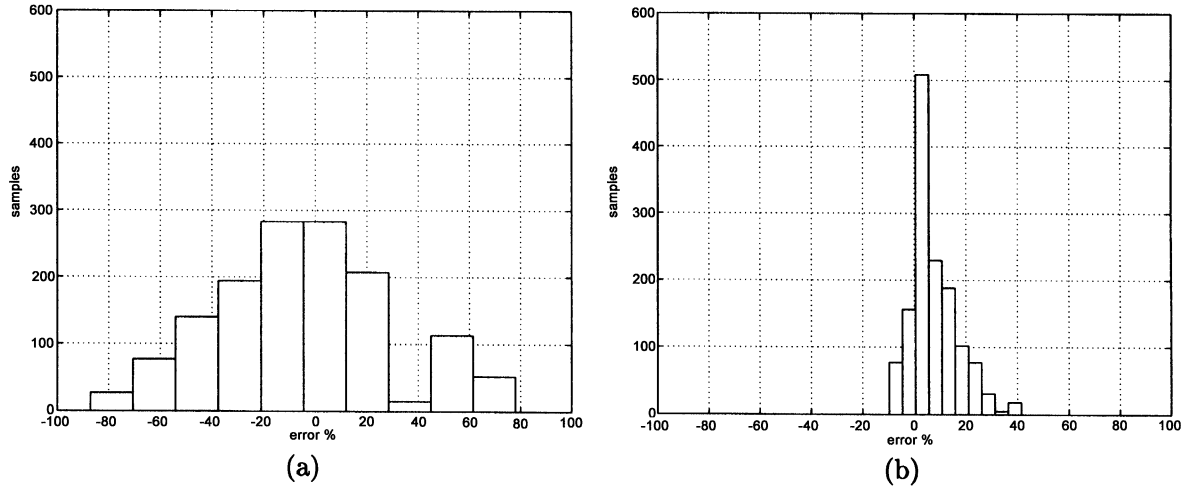

FIGURE 9 Percent error distribution over the cycle using the ACM (a) and the DDM, (b) for a test which executes the product of two 3×3 matrices.

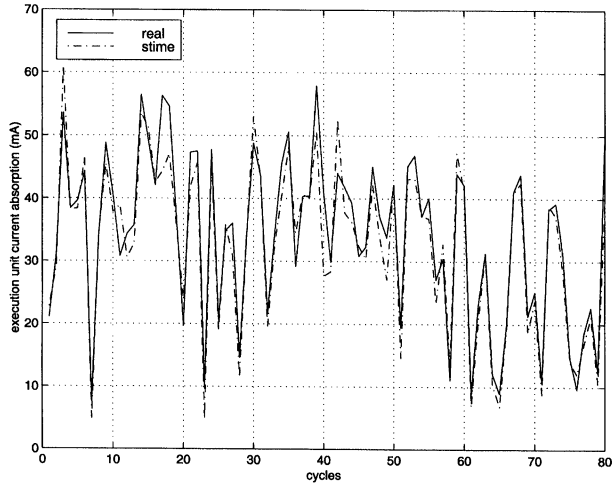

FIGURE 10 Real and estimated power absorbed by the execution unit in the first 80 cycles of execution of a test performing a digital Fourier transform (DDM).

$$+ F_{N-1}(\overline{activity_{N-1}}, I_3) + \\ + base_N^{(0)}(I_2) + int_N^{(0)}(I_1, I_2) + \\ + F_N(\overline{activity_N}, I_2) \qquad (17)$$

where:

- $base_S^{(0)}(I)$ is the power absorbed (per cycle) by stage S when it processes instruction I and when activity on the activity indexes of the stage remains null.
- $int_S^{(0)}(I, J)$ is the excess as compared with $base_S^{(0)}(I)$ (or $base_S^{(0)}(J)$) due to the fact that the stage before S was processing instruction J (I) while the current one I (J) is being processed. $int_S^{(0)}(I, J)$ represents the cost of the change in configuration for stage S and the following relation is assumed to hold:

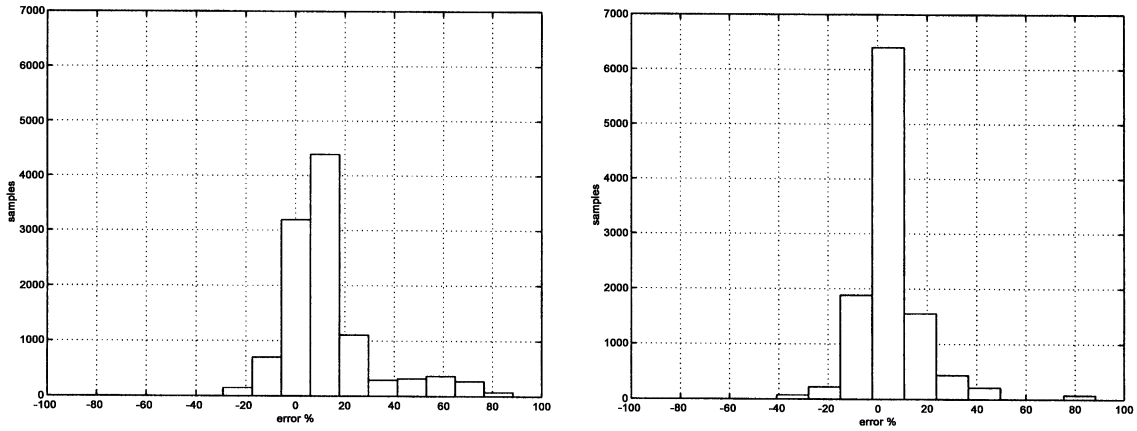

FIGURE 11 Percent error distributions over the cycle using the ACM (a) and the DDM, (b) of a test performing a digital Fourier transform.

TABLE VIII Results summary

Test	$E\{r\}$	$E\{s\}$	(mA)	$(E\{r\}-E\{s\}/E\{r\})$		$E\{\|(r-s/r)\|\}$	
name	(mA)	ACM	DDM	ACM	DDM	ACM	DDM
sum_low	6.79	13.44	6.80	97.77%	0.12%	97.79%	0.86%
sum_high	14.24	13.44	14.31	5.58%	0.5%	20.37%	3.47%
rand_inst	40.62	48.14	40.12	18.51%	1.24%	31.37%	6.44%
mtx_sum	31.97	37.83	30.97	18.32%	3.13%	31.44%	6.81%
mtx_mul	36.01	41.26	34.02	7.07%	5.53%	27.21%	8.61%
dft	32.07	28.57	30.92	10.91%	3.54%	15.13%	8.94%

Clock	S_1	S_2	...	S_{N-1}	S_N
CK_{M-1}	I_N	I_{N-1}	...	I_2	I_1
CK_M	I_{N+1}	I_N	...	I_3	I_2

FIGURE 12 Pipeline configuration.

$$int_S^{(0)}(I,J) = int_S^{(0)}(J,I) \tag{18}$$

- $F_N(\overline{activity_S}, I)$ represents the contribution of activity on the activity indexes for stage S. Activity in stage S is measured by the vector $\overline{activity_S}$ and can be defined, for example, as the Hamming distance between two subsequent configurations of the activity indexes for stage S. The contribution of activity is differently weighted according to the instruction currently being processed by stage S (so F_S also depends on instruction I).

Example In the execution stage the activity indexes can be the registers operating at the ALU block input. In this case:

$$\overline{activity_{exec}} = [activity_{oper_1}, activity_{oper_2}]$$

and for example:

$$\begin{aligned} F_{exec}(\overline{activity_{exec}}, I) \\ = activity_{oper_1} \times weight_1(I) + \\ + activity_{oper_2} \times weight_2(I) + \end{aligned}$$

where $weight_1$ and $weight_2$ weight the activity according to the instruction being executed. In *no operand isolation* architectures we will have $weight(I) = cost\ \forall\ I$.

3.4.1. *Basic and Inter-instruction Costs*

Constructing a test formed by repetition of an instruction I, so that the pipeline configuration is the one shown in Figure 13, it is possible to obtain the basic costs $base_S^{(0)}(I) \forall\ S \in \{1, 2, \ldots, N\}$. Likewise, constructing a test in which the sequence of instructions I and J is repeated, so that the pipeline configuration is the same as the one in Figure 14, we obtain the costs $int_S^{(0)}(I, J) \forall\ S \in \{1, 2, \ldots, N\}$.

3.4.2. *Determination of $F_S(\overline{activity_S}, I)$*

There remain to be determined the contribution of the data represented in Eq. (17) with the terms of type $F_S(\overline{activity_S}, I)$. First it is necessary to identify the activity indexes for each stage. It can reasonably be assumed that buses transferring data towards large combinatory blocks or buses interfacing the memory (with large capacity loads) represent general activity indexes.

This will be made clearer by referring to a specific case. Let us consider the classic 5-stage pipeline architecture of an MIPS or DLX. We will assume two activity indexes in the execution stage: the registers operating at the ALU input (interface between the decode and execution stages) and define an activity metric (*e.g.*, the Hamming distance between the configurations in these registers in two subsequent clock cycles). It is possible to construct a sequence of instructions in such a way as to induce a fixed activity on the activity indexes. To determine the relation between activity on one index and the amount of power absorbed it is necessary to construct a test in which activity on one index is made to vary continuously while that on the other is kept constant. The points obtained can be interpolated or approximated with the points of a straight line using the least squares method to establish a relation between activity on the index concerned and the power absorbed. Let us consider for example the instruction add r_d, r_{s1}, r_{s2}, the effect of which is to load the activity indexes with the contents of the registers r_{s1} and r_{s2}. Constructing a sequence of the type:

```
li r1, X1a; r1 := X1a
li r2, X2a; r2 := X2a
li r3, X1b; r3 := X1b
li r4, X2b; r4 := X2b
add r5, r1, r2; Oper1 = r1, Oper2 = r2
add r5, r3, r4; Oper1 = r3, Oper2 = r4
```

when the second `add` had been processed by the execution stage, the first activity index will pas from the configuration `X1a` to the configuration `X1b`, while the second index will pass from `X2a` to `X2b`. If we define the average amount of power absorbed by the execution stage and the activity on the activity indexes as C_{ex} and $activity_{ex}$ respectively, it is necessary to link C_{ex} and $activity_{ex}$ mathematically, *i.e.*, to determine a function f such that:

$$f(activty_{ex}) = C_{ex} + err$$

so as to minimise err. The function f can be determined in various ways. If the activity index under examination is a high-capacity bus, the activity can be measured *via* the Hamming distance between two subsequent configurations and the activity can be linked with the power absorbed with a proportionality constant. If the activity index is due to variation in the inputs of a combinatory block, it can be characterised using the power model proposed in [20] or by using a neural network to obtain the relation between the input switching activity and the power absorbed.

Clock	S_1	S_2	...	S_{N-1}	S_N
$CK-1$	I	I	...	I	I
CK	I	I	...	I	I

FIGURE 13 Pipeline configuration to obtain the basic costs $base_S^{(0)}(I) \forall\ S \in \{1, 2, \ldots, N\}$.

Clock	S_1	S_2	S_3	...	S_{N-1}	S_N
$CK-1$	I	J	I	...	J	I
CK	J	I	J	...	I	J

FIGURE 14 Pipeline configuration to obtain the costs $int_S^{(0)}(I, J) \forall\ S \in \{1, 2, \ldots, N\}$.

3.5. Characterisation: ACM *Vs.* DDM

As we have shown, the characterisation of both models (ACM and DDM) on the core processors of embedded systems requires the use of software analysis tools. Although the use of accurate simulation tools during characterisation guarantees better results in the estimation phase, it requires great calculation resources and long computation times (the lower the level at which they operate, the more accurate they are).

In this section the ACM and DDM will be compared in relation to the complexity of characterisation when, as in our case study, the silicon of the core to be characterised is not available and the only way to take the measurements is to use a software tool for power analysis. As a specific case we will consider characterisation of the instruction set of the ST20-C2P. Determination of *base*(*Inst*) requires the construction of a test in which the sequence:

```
ldc random
ldc random
ldc random
Inst
[Inst]
Inst
```

is repeated a significant number, N, of times to characterise the average. The `ldc` instructions are executed in a number of cycles proportional to the length in nibbles of the operand $\left(cycles(\texttt{ldc n}) = 1 + \frac{nibble(n)-1}{2}\right)$. If the data is uniformly distributed, a `ldc random` will be executed on average in 2 cycles. Let C_{Inst} be the number of cycles needed to execute the instruction *Inst*. The total number of cycles needed to calculate the basic cost of instruction *Inst* is therefore:

$$Cycles_{acm}(base(Inst)) = N \times (\underbrace{3 \times 2}_{3\ ldc} + \underbrace{3 \times C_{Inst}}_{3\ Inst})$$

To determine the inter-instruction costs, by repeating N times the sequence:

```
ldc random
ldc random
ldc random
Inst_j
[Inst_i]
Inst_i
```

the cost $(Inst_j \rightarrow Inst_i)_{back}$ is determined, whereas by repeating N times the sequence:

```
ldc random
ldc random
ldc random
Inst_i
[Inst_i]
Inst_j
```

$(Inst_i \rightarrow Inst_j)_{forw}$ is determined. We have:

$$\begin{aligned} &Cycles_{acm}((Inst_j \rightarrow Inst_i)_{back}) \\ &\quad = N \times (\underbrace{3 \times 2}_{3\ ldc} + + C_{Inst_j} + 2 \times C_{Inst_i}) \\ &Cycles_{acm}((Inst_i \rightarrow Inst_j)_{frow}) \\ &\quad = N \times (\underbrace{3 \times 2}_{3\ ldc} + + 2 \times C_{Inst_i} + C_{Inst_j}) \end{aligned}$$

Let M the number of instructions to be characterised, the number of cycles needed to determine all the basic and inter-instruction costs for the ST20-C2P is:

$$\begin{aligned} Cycles_{acm} = &\sum_{i=1}^{M} Cycles(base(Inst_i)) + \\ &+ \sum_{j=1}^{M}\sum_{k=1}^{M}[Cycles((Inst_j \rightarrow Inst_k)_{back})] + \\ &+ \sum_{j=1}^{M}\sum_{k=1}^{M}[Cycles((Inst_k \rightarrow Inst_j)_{forw})] \end{aligned} \tag{19}$$

We follow the same reasoning when the DDM is used. Determination of $base^{(0)}(Inst)$ requires the construction of a test of the following kind:

```
ldc C
ldc B
ldc A
Inst
[Inst]
Inst
```

in which the values A, B and C are fixed in such a way that execution of the framed instruction takes place in a state of null activity. $base^{(0)}(Inst)$ will be the average power absorbed during execution of the framed instruction. If we indicate the number of cycles needed to execute instruction $Inst$ as C_{Inst}, the total number of cycles needed to calculate $base^{(0)}(Inst)$ will be:

$$Cycles_{ddm}(base^{(0)}(Inst)) = 1 \times (\underbrace{3 \times 2}_{3\ ldc} + \underbrace{3 \times C_{Inst}}_{3\ Inst})$$

To determine the inter-instruction costs with null activity, we consider the following sequences:

```
ldc C
ldc B
ldc A
Inst_j
Inst_i
Inst_i
```

determining the cost $(Inst_j \rightarrow Inst_i)^{(0)}_{back}$, while from:

```
ldc C
ldc B
ldc A
Inst_i
Inst_i
Inst_j
```

we determine $(Inst_i \rightarrow Inst_j)^{(0)}_{forw}$. We get:

$$Cycles_{ddm}((Inst_j \rightarrow Inst_i)^{(0)}_{back} = 1 \times (\underbrace{3 \times 2}_{3\ ldc} + + C_{Inst_j} + 2 \times C_{Inst_i})$$

$$Cycles_{ddm}((Inst_i \rightarrow Inst_j)^{(0)}_{forw} = 1 \times (\underbrace{3 \times 2}_{3\ ldc} + + 2 \times C_{Inst_i} + C_{Inst_j})$$

The number of cycles needed to determine all the basic and inter-instruction costs with null activity is:

$$Cycles_{ddm} = \sum_{i=1}^{M} Cycles(base^{(0)}(Inst_i)) + + \sum_{j=1}^{M}\sum_{k=1}^{M}[Cycles((Inst_j \rightarrow Inst_k)^{(0)}_{back})] + + \sum_{j=1}^{M}\sum_{k=1}^{M}[Cycles((Inst_k \rightarrow Inst_j)^{(0)}_{forw})]$$

Then:

$$Cycles_{acm} \approx 2 \times N^2 \times Cycles_{ddm}$$

This requires some clarification. The number of cycles needed to characterise the ACM was calculated without including the cycles needed to characterise the term $f(data)$. In our case study, only two activity indexes were taken into consideration, so the number of cycles needed to characterise them is negligible as compared to those required to characterise the other terms. In addition, the absence of *operand isolation* mechanisms makes it possible to render $f(data)$ instruction-independent (see 3.2.2. in which about 96 measures were required to determine the costs per transition for the activity indexes considered and where each measure required the construction of a test of about 10 cycles). It should be pointed out that prediction of the number of cycles needed for the characterisation phase, as presented, is quite conservative. In reality, the number may be greatly limited if the instruction set is subdivided into classes of instructions (*e.g.*, memory access instructions, integer calculation, operations on bits, *etc.*) and only one instruction per set is characterised. This technique minimises characterisation times, but it makes the estimate less accurate.

4. A TOOL FOR SOFTWARE POWER CONSUMPTION ESTIMATION

Application of model 16 requires knowledge of the state of the processor cycle by cycle. By the state of a processor in a generic clock cycle we mean the contents of the registers, the logical levels of the

lines of the buses interfacing the memory and the datapath, and the state of the finite state machine implementing the micro-controller. Such detailed knowledge of the state of the processor can be obtained *via* VHDL simulation of the code to be estimated. The manufacturer of the processor will obviously not provide the VHDL models of the processor, so the client cannot run a VHDL simulation of the code to obtain a cycle-accurate trace and apply model 16.

A CPU emulator is an application that is always present in the package of development tools a CPU manufacturer provides his client with. A CPU emulator is a functional model of the CPU with which it is possible to trace the flow of instructions executed by the processor during the execution of a program. For each instruction executed, it gives the state of the registers and any memory access operations. An instruction is generally executed in a certain number of cycles, passing through a sequence of micro-states that is usually not fixed.

4.1. Binary Tree Representation

The finite state machine implementing the CPU micro-controller determines the subsequent state according to the current one and the value of certain signals. The VHDL code implementing the FSM of the micro-controller can be generalised as in Figure 15. So, given an initial state it is possible to reach several final states according to the value of certain signals. We can represent the states that can be reached from a certain state by means of a binary tree in which the nodes represent the conditions and the leaves the states reached. In the case described, the binary tree representing the state $state_i$ is made up of four nodes and four leaves. The nodes represent the conditions of the `IF` constructs while the leaves represent the possible subsequent states (Fig. 16). The path through the tree depends on whether the conditions defined by the nodes are met or not: having fixed a conditional node, the left-hand side path is taken if the condition is met, the right-hand side

```
CASE current_state IS
   ...
   WHEN state_i =>
      <operazioni>
      ...
      <operazioni>
      IF signal_i = TRUE THEN
        IF signal_j = TRUE THEN
          next_state <= state_j;
        ELSE
          next_state <= state_k;
        END IF;
      ELSE
        IF signal_k = TRUE THEN
          IF signal_l = TRUE THEN
            next_state <= state_l;
          ELSE
            next_state <= state_m;
          END IF
        ELSE
          ...
        END IF
      END IF
   ...
END CASE
```

FIGURE 15 VHDL code implementing the FSM of the micro-controller.

one if it does not. If, for instance, the signals $signal_i$ and $signal_j$ are both high, the subsequent state will be $state_j$.

4.2. Analysis of Conditions

Splitting up the instructions into the component micro-states can be achieved if it is possible to evaluate the conditions (at the nodes of the conditional tree) using the information supplied by the CPU emulator. These conditions refer to the contents of the registers, the datapath bus and the signals regulating the memory access protocol.

The conditions depending on the register contents can be directly evaluated as the trace file generated by the processor emulator provides information about the register contents. Information about the state of the datapath bus lines is not directly given in the trace file. It is, however,

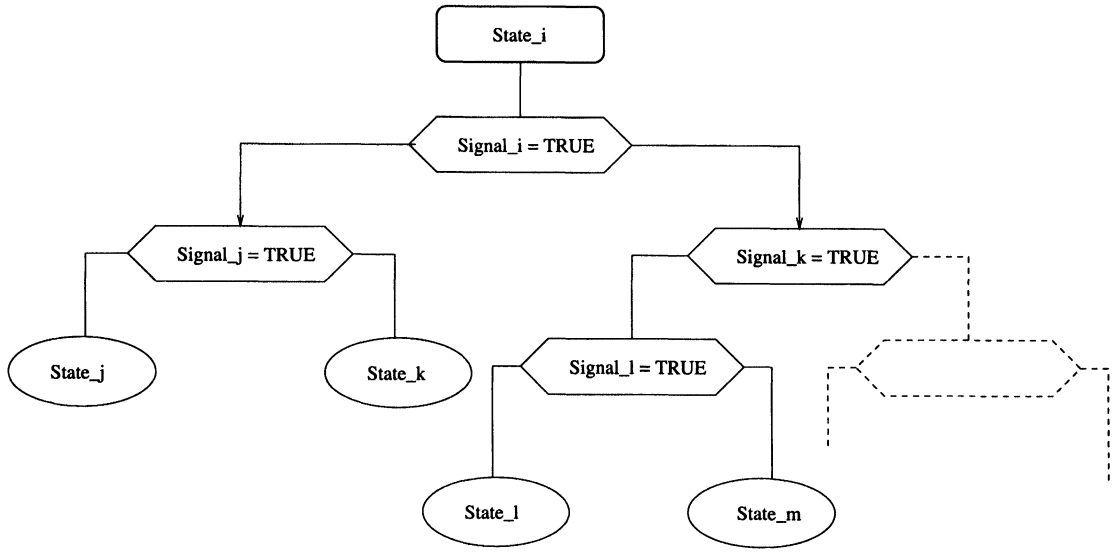

FIGURE 16 Conditional tree representing the state $state_i$.

possible to obtain it through knowledge of the current state and the register contents. The state, in fact, fixes the mapping between the registers and the datapath buses, so it is possible to obtain the word mapped on *xbus* and *ybus*. For instructions that do not map a register on *xbus* and *ybus* the contents remain constant due to the presence of the *bus keepers*. The problem of evaluating the conditions depending on the memory speed is solved by implementing a memory model that operates according to the protocol specified.

4.3. Generation of a Trace of the Micro-states

As we have illustrated, it is possible to split an instruction up into its component micro-states just by using the register contents and state information. Valuation of the conditions makes it possible to visit the conditional tree relating to the current state and reach the leaf representing the subsequent state. If this algorithm is applied to all the instructions appearing in the trace of a program, it is possible to generate a micro-state by micro-state trace from the instruction by instruction trace.

Figure 17 shows the estimation flow starting from the source code of the program. This is compiled to generate an image of the ROM, and is then executed *via* the CPU emulator that supplies a trace of the instructions executed together with the contents of the registers and information about memory access. State information in the form of a conditional tree, together with the instruction by instruction trace and memory model, allow us to generate a micro-state by micro-state trace. At this point we can apply the model defined by 16 to obtain a trace comprising power information.

4.4. Clarification of the Cycle-accurate Concept

Splitting the instruction up into their component micro-states does not mean generating a cycle-accurate trace. Fetch unit stalls may introduce waiting states that are difficult to pick up from the instruction trace. A waiting state is imposed when the fetch unit has stalled and the execution unit is ready to execute a new instruction in the next cycle. Picking up these effects form the instruction by instruction trace is a very complex operation that essentially depends on two factors:

- the memory speed, which determines the number of cycles needed to fill the instruction buffer with the new instructions to be executed.

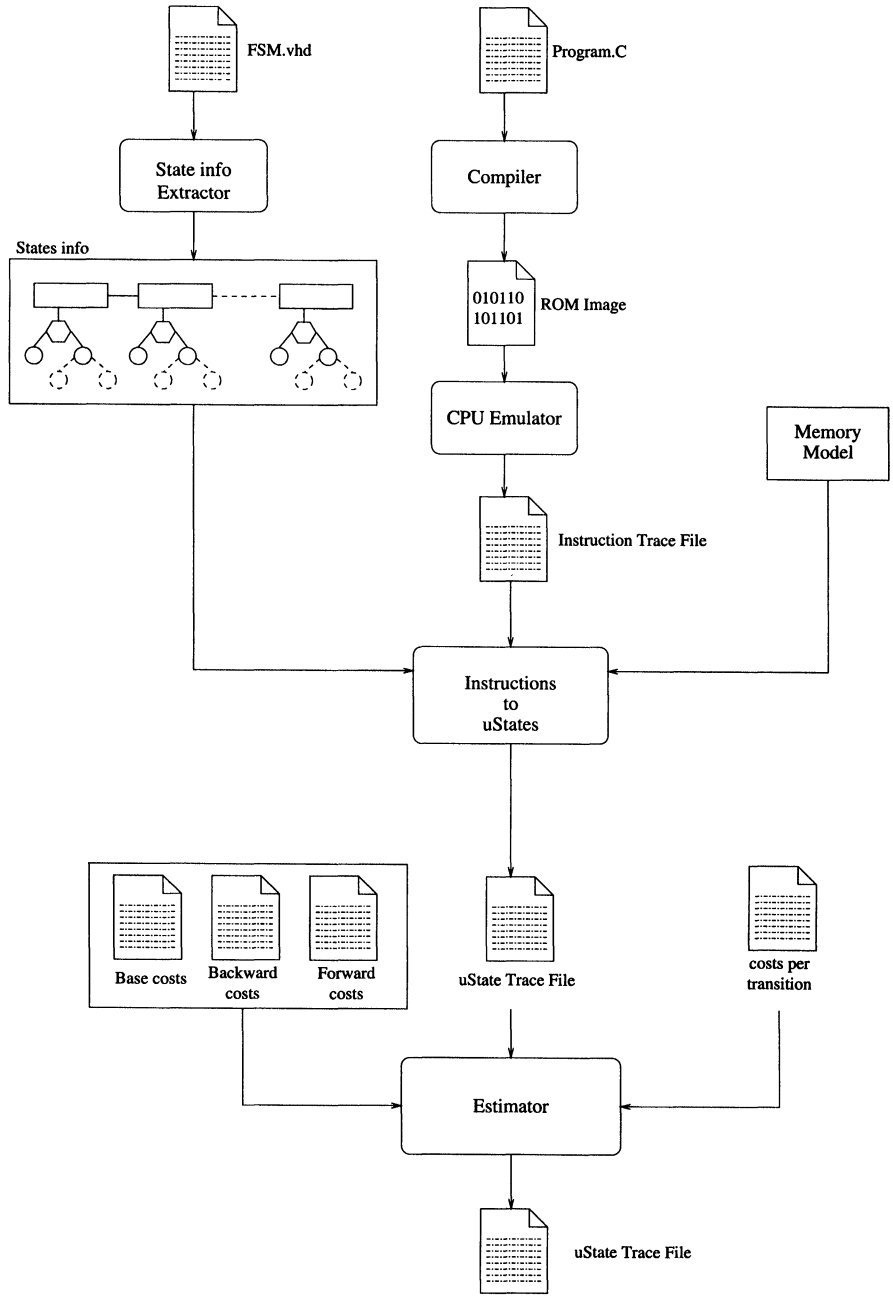

FIGURE 17 Estimation flow.

- The alignment of the words of the machine code of the instructions in the memory.

Estimation of the average amount of power absorbed during these waiting states also depends on a number of factors. As we have seen, a great contribution to the power absorbed in the ST20-C2P is made by activity at the micro-controller ROM inputs. In fetch unit stall cycles, the micro-controller ROM is stimulated with spurious data

on account of the direct connection between the fetch unit predec output and the micro-controller ROM inputs.

4.5. Validation of the Method

The tool was validated by executing some estimation tests and comparing the results with those obtained using PowerMill. In all the cases examined both the error in estimating the average amount of power absorbed and that in estimating the total power required were always below 5%. It was not possible to assess the tool in terms of error per cycle since stalls during the execution of the tests led to a loss of synchronisation between the trace extracted *via* the VHDL simulation and the one generated by the tool. It was only possible to perform this analysis on short sequences of code in which there were no stalls. The accuracy achieved is shown in Figure 18.

4.6. Best and Worst Case

The energy absorbed in executing any one program varies according to the data involved. A data-dependent model can be used to determine the minimum and maximum power absorbed to execute the program. These operational extremes can be estimated by exploiting model 7. Having established the software, the minimum power required can be estimated by neglecting the data contribution (*i.e.*, setting $f(data)=0$ for each instruction executed), whereas the maximum power required can be estimated by maximising the term $f(data)$, *i.e.*, maximising the activity on the activity indexes involved during execution of each instruction.

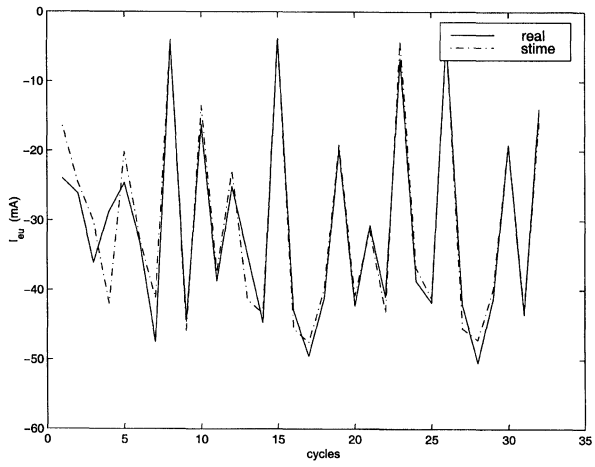

FIGURE 18 Real and estimated power absorbed in the first 32 cycles of execution of a test performing the sum of two 3x3 matrices.

TABLE IX Estimate of the average power absorbed when the term $f(data)$ is neglected and maximised. (values in mA)

Program	$E\{I\}$	$E\{I_{min}\}$	$E\{I_{max}\}$
`mtx_sum`	30.30	27.79	51.82
`mtx_mul`	34.31	28.53	55.41
`img-filter`	28.04	24.62	50.29

An example of the possible variation in the amount of power absorbed by a single program when the amount of data varies is given in Table IX. For each program the table gives the estimate of the average power absorbed when the term $f(data)$ is neglected and maximised. As can be seen, the data contribution alone can in principle cause a 100% difference in the average amount of power absorbed per cycle.

5. CONCLUSIONS

We have proposed a data-dependent model for estimation of the average amount of power a processor consumes cycle per cycle during execution of a program. Application of the model requires a trace of the program as input, *i.e.*, the flow of instructions executed and the state of the system as defined by the contents of the registers and information about memory access, if any. To determine the trace of the program it is necessary to execute it using, for example, a functional model of the processor. The need to execute the program to obtain the trace is not a limiting factor: generally, any model for software power consumption estimation requires knowledge of the flow of instructions executed. As the sequence of instruc-

tions executed is highly data-dependent (consider, for example conditional jumps), it is necessary to execute the program to be evaluated.

The software power consumption estimation models proposed in literature aim at minimising the mean estimation error over a time window which extends to the duration of the program. Our point of view, on the other hand, was to minimise the estimation error per clock cycle: in this case the data contribution cannot be overlooked (compare Figs. 2 and 6). The results obtained confirm the validity of the model: in all the tests carried out, the average error per cycle remained below 10%. The advantage of a data-dependent model over a data-independent one ([1]) or an instruction-independent one ([11]) is without doubt the greater accuracy of the estimate.

All three models give the same estimate of the total amount of power consumed during execution of a whole general-purpose program, but where a data-dependent model shows its strength is in the estimation of small fragments of code or highly input data-dependent applications (mathematical routines, DSP applications).

A data-dependent model can be used to characterise a given software application in power terms for varying amounts of input data. An example is given by a function for the application of an image filter: it can be characterised *via* a power cost depending on the graphical filter used. Another field of application for a data-dependent estimation model is determining the best case and worst case conditions for a program in terms of power: model 7 can be applied to each instruction in the program trace minimising or maximising the term $f(data)$.

References

[1] Vivek Tiwari, Sharad Malik and Andrew Wolfe,"Power Analysis of Embedded Software: A First Step Towards Software Power Minimization", *IEEE Transactions on VLSI Systems*, December, 1994.

[2] Tiwari, V., Malik, S., Lee, M. and Fujita, M.,"Power Analysis and Minimization Techniques for Embedded DSP software", *IEEE Transactions on VLSI Systems*, March, 1997.

[3] Tiwari, V., Malik, S. and Wolfe, A. (1996). "Instruction Level Power Analysis and Optimization of Software", *Journal of VLSI Signal Processing*, pp. 1–18.

[4] Tiwari, V. and Lee, M. (1998). "Power Analysis of a 32-bit Embedded Microcontroller", *VLSI Design Journal*, **7**(3).

[5] Tiwari, V., Ashar, P. and Malik, S.,"Technology Mapping for Low Power", *NEC CCRL Technical Report*, 1(92-C019-4-5509-2), October, 1992.

[6] Bahar, R. I. and Somenzi, F., "Boolean Techniques for Low Power Driven Re-Synthesis", *ICCAD-95: ACM/IEEE International Conference on Computer Aided Design*, Santa Clara, CA, November, 1995.

[7] Tiwari, V., Malik, S. and Ashar, P., "Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design", *IEEE Transactions on Computer-Aided Design*.

[8] Téllez, G. E., Farrahi, A. and Sarrafzadeh, M. (1995). "Activity-Driven Clock Design for Low Power Circuits", *International Conference on Computer Aided Design (ICCAD95)*, San Jose, Ca., USA, pp. 62–65.

[9] Chih-Tung Chen, C. T. and Kucukcakar, K., "An Architectural Power Optimization Case Study Using High-level Synthesis", *International Conference on Computer Design (ICCD '97)*. 12–15 October, 1997, Austin, Texas.

[10] Tiwari, V., Donnelly, R., Malik, S. and Gonzalez, R., "Dynamic Power Management for Microprocessors: A Case Study", *IEEE VLSI Design*, January, 1997.

[11] Russell, J. T. and Jacome, M. F., "Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors", *Proceedings of ICCD'98*.

[12] Sarta, D., Trifone, D. and Ascia, G., "A Data Dependent Approach to Instruction Level Power Estimation", *IEEE Alessandro Volta Memorial Workshop on Low Power Design*, Como Italy, 4–5 March, 1999, pp. 182–190.

[13] Benini, L., De Micheli, G., Macii, E., Poncino, M. and Quer, S., "Power Optimization of Core-Based Systems by Address Bus Encoding", *IEEE Transactions on VLSI Systems*, **6**(4), December, 1998.

[14] Jensen, F. and Tyagi, A., "Reduced Address Bus Switching with Gray PC", *Power Driven Architecture Workshop*, Barcelona, Spain, 1998.

[15] Musoll, E., Lang, T. and Cortadella, J., "Reducing the Energy of Address and Data Buses with the Working-Zone Encoding Technique and its Effect on Multimedia Applications", *Power Driven Architecture Workshop*, Barcelona, Spain, 1998.

[16] Benini, L., De Micheli, G., Macii, E., Sciuto, D. and Silvano, C., "Address Bus Encoding Techniques for System-Level Power Optimization", *IEEE Design Automation and Test Conference in Europe*, Paris, France, February, 1998, pp. 861–866.

[17] Stan, M. R. and Burleson, W. P., "Coding a Terminated Bus for Low Power", *Great Lakes Symposium on VLSI*, pp. 70–73, Buffalo, NY, March, 1995.

[18] Ramprasad, S., Shanbhag, N. R. and Hajj, I. N., "Coding for Low-Power Address and Data Busses: A Source-Coding Framework and Applications", *Eleventh International Conference on VLSI Design: VLSI for Signal Processing*, 4–7 January, 1998, India.

[19] Stan, M. R. and Burleson, W. P. (1999). "Bus-Invert Coding for Low Power I/O", *IEEE Transactions on VLSI Systems*.

[20] Qing Wu, Qinru Qiu, Massoud Pedram and Chih-Shun Ding, "Cycle-Accurate Macro-Models for RT-Level Power Analysis", *IEEE Transactions on VLSI Systems*, **6**(4), December, 1998.

[21] Givargis, T., Henkel, J. and Vahid, F., "Interface and Cache Power Exploration for Core-Based Embedded System Design", *Int. Conference on Computer Aided Design (ICCAD)*, pp. 270–273, November, 1999.
[22] Givargis, T. and Vahid, F., "Interface Exploration for Reduced Power in Core-Based Systems", *Int. Conference on System Synthesis*, December, 1998.
[23] Vahid, F. and Givargis, T. (1998). "Incorporating Cores into System-Level Specification", *Int. Symposium on System Synthesis.*
[24] Givargis, T., Vahid, F. and Henkel, J. (2000). "A hybrid approach for core-based system-level power modeling", *Asia and South Pacific Design Automation Conference.*
[25] "ST20C2 Core Instruction Set Reference Manual", ST-Microelectronics, *http://www.st.com/.*

Authors' Biographies

Giuseppe Ascia received the Laurea degree in electronic engineering and the Ph.D. degree in Telecommunications from the University of Catania, Italy, in 1994 and 1998, respectively.

In 1994, he joined the Institute of Computer Science and Telecommunications, University of Catania, Italy. His interests are artificial intelligence, soft computing, and hardware architectures.

Vincenzo Catania received the Laurea degree in electrical engineering from the University of Catania, Italy, in 1982.

Until 1984, he was responsible for testing microprocessor systems at SGS, Catania, Italy. Since 1985 he has been cooperating in research on computer networks with the Institute of Computer Science and Telecommunications, University of Catania, Italy, where he is an Associate Professor of computer science. His research interests include performance and reliability assessment in parallel and distributed systems, VLSI design, and fuzzy logic.

Maurizio Palesi received the Laurea degree in computer science engineering from the University of Catania, Italy, in 1999. He is currently working toward the Ph.D. degree at the same university.

In 1999, he joined the Institute of Computer Science and Telecommunications, University of Catania, Italy. His research interests include power estimation, low-power design methodologies and techniques, codesign techniques for embedded systems.

Davide Sarta received the Laurea degree in electronic engineering from the University of Catania, Italy, in 1994 and Master degree in Information Technology in 1997 at the Politecnico di Milano.

He has worked at STMicroelectronics, Catania, Italy, since 1997, where he is involved in design and verification of System-on-a-Chip. His research interests include techniques for power estimation and optimisation in digital systems.