A Multi-Level Strategy for Software Power Estimation

C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, D. Sciuto
Politecnico di Milano, P.zza L. da Vinci, 32 - 20133 Milano, Italy
{brandole,fornacia,pomante,salice,sciuto}@elet.polimi.it

Abstract

In this paper a comprehensive methodology for software power estimation is presented. The methodology is supported by rigorous mathematical models of power consumption at three different levels of abstraction. The methodology has been validated in a complete framework developed within the TOSCA co-design environment.

1 Introduction

Embedded system development requires fine tuning of a number of specific constraints, since such applications strive for high volumes and there is a pay-off for size, power and speed optimization techniques. The current pervasiveness of microprocessor-based architectures, is enforcing the importance of concurrently design (co-design) hardware and software. A typical need is the possibility of working with models at different levels of granularity and accuracy, to enable fast exploration of alternative designs and to deal with the presence of partially characterized components, such as the microprocessors. Furthermore, the steady shifting toward nomadic applications is increasing the importance of analyzing power issues even during the software development process [2, 3, 4]. Recent analysis and optimization techniques[1], focused on specific aspects of the software power consumption, such as I/O management, memory requirements, system bus traffic optimization and instruction–set optimization but, according to our best knowledge, no one is attempting a systematic power/performance–aware analysis flow for the software. The goal of this paper is to describe how the different activities of the TOSCA hw/sw codesign flow, have been extended to enable power analysis of the software. In particular, due to space reasons, this paper includes only the description of the different models developed to power characterize the software and two examples of their practical use. The paper is organized as follows. Section 2 describes the compilation steps we are considering for the software and the related abstraction levels. Section 3 presents the analysis models we developed to characterize in power the instruction set of a microprocessor and how this information are processed to back-annotate the results towards the upper abstraction levels. Section 4 show how the identified model can be assembled into a comprehensive power estimation flow allowing the designer to navigate between different levels of abstraction and estimation accuracy. Finally Section 5, reports some experimental results obtained by considering two commercial microprocessors showing the achievements of the proposed methodology in terms of accuracy and analysis speed.

2 Software Compilation

The complete design flow presented in this paper has been developed with two main goals: on one hand, to provide a tool to compare the power consumption of different algorithms on the same microprocessor, on the other hand, to help a designer in the choice of the best suited microprocessor under performance and power constraints. To allow the comparison of the power requirements of the same algorithm over different microprocessors, the compilation process have to be retargetable. To this purpose, an intermediate representation based on the pseudo–assembly language VIS (*Virtual Instruction Set*) has been introduced [7]. The compilation flow is composed of two steps:

- *Compilation:* from source code to VIS language.
- *Mapping:* from VIS code onto target assembly.

Most of the complexity of the compilation is hidden in the first step, making thus the second phase much simpler and time–effective. Throughout the whole compilation flow, track is kept on the transformations being performed, to allow back–annotation of the information across the different levels of abstraction. Figure 1 depicts the compilation flow.

2.1 Compilation

The translation from the source, high–level, language to VIS is an actual compilation. The source code is parsed

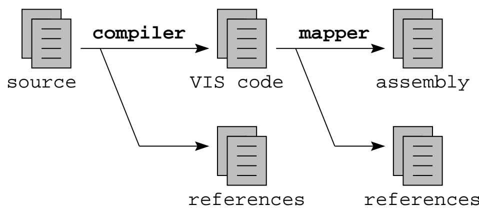

Figure 1. The compilation flow

and a syntax–tree is built in memory: all subsequent operations are performed on this internal model. First of all, expressions and conditionals are analyzed to determine the number of temporary variables necessary for calculations; these variables will then be associated either to registers, to the stack or to the local frame. With these information annotated on the syntax–tree, op–code selection and code generation can be performed. Note that the result of this compilation is a virtual assembly and is thus not related to any existing architecture. In particular, the number of registers available is not known during this phase (unless a specific target architecture has already been envisaged) and can thus be either selected by the designer or left unspecified. If a specific register–file size is selected, register binding is performed while, if it is left unspecified, all registers that the algorithm requires are allocated. The last compilation step adds to the purely "functional" code, the system calls necessary to manage concurrency and communication, whenever present in the specification.

2.2 Mapping

The transformation of the VIS code into the target assembly is a *mapping* process based on mapping *libraries*. A mapping library is a collection of *rules*, each specifying how one or more VIS instruction are to be translated into the target assembler. Mapping rules are written in C, with the support of a number of functions and macro definitions, and compiled into a dynamic library. The *mapper* kernel loads and parses the VIS code, links the selected mapping library and applies the suitable rules to produce as output the assembly code.

3 Power Estimation Models

At the end of the compilation flow, three representations, at different abstraction levels, are available. At assembly level, on one hand, the representation is extremely detailed and allows a very accurate calculation of the power consumption, on the other hand the raw power figures are hardly usable by the designer. At source level, the limited detail results in a less precise, but much more readable and useful, power estimate. The VIS level represents a trade–off between these two boundaries.

The methodology presented in the following considers all the three levels and provides models and tools to perform estimates with the desired level of detail. This section presents the models on which the power estimation is based.

3.1 Assembly–level model

The basic idea is to characterize each assembly instruction and addressing mode with an average, i.e. data independent, power consumption figure. Since the clock frequency and power supply voltage are known, power is often specified as an average current drawn per clock cycle.

Under this assumption, the energy requirements of a given assembly code can easily be derived once the specific instruction set is completely characterized. It is worth noting that, currently, only few processor vendors can provide this type of information and even fewer disclose them. Deriving the current absorption of each instruction, with its different addressing modes is a lengthy process that requires measuring the current drawn by the microprocessor core during execution of long sequences of the same instruction with varying data. Furthermore, setting up a suitable measurement environment is not trivial since typical development boards rarely provide access to the power supply pins of the core and have thus to be modified and, in addition, costly equipment is necessary to perform current measures at frequencies as high as 40 to 200 MHz or even more. To overcome all these problems, the model briefly presented in the following has been developed.

The model is based on a functional analysis and decomposition of the activities performed by a microprocessor as it executes a specific instruction. The key idea behind the model is the concept of *functionality* that is a set of activities, aimed at a specific goal, involving, partially or totally, one or more architectural units of a generic microprocessor. Functionalities must be either time–disjoint or space–disjoint or both. Two functionalities F_1 and F_2 are *time–disjoint* if they operate in different clock cycles; they are *space–disjoint* if they stimulate different architectural units. Under these constraints, the execution of an instruction can be modeled as the combination of a certain number of functionalities. A detailed analysis has led to set of functionalities of table 1.

As an example consider the Intel 80486DX instruction `ADD R3, (R2)+`: the op–code uses $F\&D$ and $A\&L$, the destination operand `R3` uses $WrReg$ and the source operand `(R2)+` uses $Ld\&St$, $A\&L$ and $WrReg$. The completion of the instruction stimulates thus $\{F\&D, A\&L\} \cup \{WrReg\} \cup \{Ld\&St, A\&L, WrReg\}$ $= \{F\&D, Ld\&St, A\&L, WrReg\}$.

Associating a current if_j to the j-th functionality, the en-

Functionality	Activities
$F\&D$	Fetch and decode
Br	Branch, calls
$WrReg$	Register writing
$A\&L$	Arithmetic and logic
$Ld\&St$	Load, store and stack

Table 1. A possible functional decomposition

ergy absorbed by the processor core executing the instruction s can be expressed as:

$$e_s = V_{dd} n_{ck,s} \tau i_s = V_{dd} n_{ck,s} \tau \sum_{j=1}^{k} if_j \cdot a_{s,j} \quad (1)$$

where $k = 5$ is the number of functionalities and $a_{s,j}$ is a coefficient that specifies whether functionality j is involved in the execution of instruction s. To derive the currents if_j a *learning–set* $\mathcal{S}_L$ of power–characterized instructions can be used [5]. Defining the matrices $\mathbf{IN} = \{i_s n_{ck,s}\}$, $\mathbf{A} = \{a_{s,j}\}$ and $\mathbf{IF} = \{if_j\}$, equation (1) can be rewritten as:

$$\mathbf{IN} = \mathbf{A} \times \mathbf{IF} + \mathbf{R} \quad (2)$$

where $\mathbf{R}$ is a residual vector. Solving equation 2 for $\mathbf{IF}$ in the least square sense (which implies neglecting the residual) gives an estimate of the currents:

$$\widehat{\mathbf{IF}} = (\mathbf{A}^T \times \mathbf{A})^{-1} \times \mathbf{A} \times \mathbf{IN} = \mathbf{A}^* \times \mathbf{IN} \quad (3)$$

Substituting the estimate $\widehat{\mathbf{IF}}$ into equation 2 allows deriving an estimate of the power consumption of instructions not in $\mathcal{S}_L$. To verify the correctness of this model, its statistical properties must be derived and in particular the used *estimator* (the least square method) must be proven unbiased. Under the hypothesis of a gaussian residual $G(0, \lambda^2)$, the expectation value and the variance of the parameters are:

$$\text{E}[\widehat{\mathbf{IF}}] = \mathbf{IF} \quad (4)$$

$$\text{VAR}[\widehat{\mathbf{IF}}] = \lambda^2 (\mathbf{A}^T \times \mathbf{A})^{-1} \quad (5)$$

Since λ^2, the input variance of the residual, is not known it has to be substituted by its estimated value:

$$\hat{\lambda}^2 = ||\widehat{\mathbf{IN}} - \mathbf{IN}||^2 / (m - k) \quad (6)$$

where m is the number of samples and k is the number of parameters. To verify the gaussian noise hypothesis, a $Z_0.95$ test can be performed: the null hypothesis is accepted if the mean value of the residual falls in the interval $\pm 1.96 \hat{\lambda}^2 / \sqrt{m}$.

This model allows extrapolation of the power figures of the whole instruction set, based on a limited number of measures (at least 10–15). Even when no measures are available at all, it is still possible to compare different algorithms or source codes with respect to their power consumption. An accurate analysis of the currents absorbed by different microprocessors during the execution of a number of instructions has revealed that though the absolute value of the currents varies in a wide range (from 5–15 mA to 400–600 mA) their relative values, with respect to a reference instruction, lay in a much narrower range (1.0 ± 0.2). This suggests that, using these relative values and a set $\mathcal{P}$ of microprocessors, a single general model can be derived. The relative current is defined as:

$$i_{rel,s} = \frac{i_s}{i_{ref}} = \sum_{j=1}^{k} \frac{if_j}{i_{ref}} a_{s,j} = \sum_{j=1}^{k} if_{rel,j} a_{s,j} \quad (7)$$

For the generic q-th processor of $\mathcal{P}$, characterized by $\mathbf{A}_q$ and $\mathbf{IN}_{rel,q} = \{i_{s,rel} \cdot n_{ck,s}\}$, the following equation holds:

$$\mathbf{IN}_{rel,q} = \mathbf{A}_q \times \mathbf{IF}_{rel,q} + \mathbf{R}_{rel,q} \quad (8)$$

where $\mathbf{R}_{rel,j}$ is, again, a residual vector. Solving the system in the least square sense yields:

$$\widehat{\mathbf{IF}}_{rel,q} = \mathbf{A}_q^* \times \mathbf{IN}_{rel,q} \quad (9)$$

The general model should depend on a unique set of parameters $\mathbf{IF}_{rel}$, rather than the processor–specific parameters $\mathbf{IF}_{rel,j}$, and thus the model becomes:

$$\mathbf{IN}_{rel,q} = \mathbf{A}_q \times \mathbf{IF}_{rel} + \mathbf{R}_{rel,q} \quad (10)$$

Combining equation 9 and 10 gives:

$$\widehat{\mathbf{IF}}_{rel,q} = \mathbf{A}_q^* \times \mathbf{IN}_{rel,q} = \mathbf{IF}_{rel} + \mathbf{A}_q^* \times \mathbf{R}_{rel,q} \quad (11)$$

Adding up equation (11) for all indices q corresponding to the p processors, and dividing both sides by p, yields:

$$\frac{1}{p} \sum_{q=1}^{p} \widehat{\mathbf{IF}}_{rel,q} = \mathbf{IF}_{rel} + \frac{1}{p} \sum_{q=1}^{p} \mathbf{A}^*{}_q \times \mathbf{R}_{rel,q} \quad (12)$$

Equation 12 indicates that an estimator of the parameters of the general model can be the *average* of the estimated parameters of each processor in the set $\mathcal{P}$. The statistical properties of the residual $\mathbf{R}_{rel,q}$ and of the chosen estimator are discussed in [5]. By applying the same method used for the single–processor case, the expression for the variance can be derived and results:

$$\text{VAR}[\widehat{\mathbf{IF}}_{rel}] = \frac{1}{p^2} \sum_{q=1}^{p} \text{VAR}[\widehat{\mathbf{IF}}_{rel,q}] \quad (13)$$

The meaning of this last equation is that by increasing the number p of considered processors, the variance of the parameters of the general model decreases.

3.2 VIS–level model

The methodology presented in this section can be applied to derive the power characterization of the VIS instruction set for any target processor. A VIS instruction is

defined by a) *op-code*, the type of operation; b) *addressing mode*, the type of operand, and c) *operand value*, the value of the operands; The *class* of an instruction is defined by its op-code and the addressing mode of its operands, but ignoring the value of operands. For each instruction *class*, thus, a set of instructions can be built, varying the value of the operands. As an example, consider the VIS instruction `MOVE.W #16, +5(R0)`: the op-code is `MOVE.W`, the addressing modes of the two operands are *immediate* (`#16`) and *indirect* (`+5(R0)`) and the values are 16 for the first operand and the couple (5, R0) for the second. It is not rare that instructions of the same *class* map to different assembly codes and are thus characterized by different power consumption [6].

As an example consider the ARM7TDMI microprocessor: an immediate constant can be loaded into a 32-bit register directly if and only if it falls in the range 0—255; when the immediate value is greater than 255, its low and high bytes must be loaded into the register separately, suitably shifting the register content after the first load. To properly account for these differences *all* the possible instructions of a given instruction class must be analyzed. Let $\mathcal{I}$ be an instruction class and $i_j \in \mathcal{I}$ a generic instruction with specific operands values. Using the estimation flow described in section 4, all instructions in $\mathcal{I}$ can be annotated with the actual timing $t_j = t(i_j)$ and average current $c_j = c(i_j)$.

To derive single values $t(\mathcal{I})$ and $c(\mathcal{I})$ for the VIS instruction *class* $\mathcal{I}$, four different approaches have been adopted:

- **Different translations only.** The class timings and currents are computed by averaging only the values t_j and c_j corresponding to different translations.
- **Different figures only.** Timings and currents are computed by averaging only the t_j and c_j that are numerically different.
- **Complete uniform.** Timings and currents are computed averaging *all* the values t_j and c_j. This choice neglects the semantics of instructions, assuming a uniform distribution of instructions within a class.
- **Complete weighted.** Timings and currents are computed by averaging all the values t_j and c_j, *weighted* with their relative frequencies obtained analyzing a large set of benchmarks. The previous case, as mentioned, ignores the semantics of the instructions. This hypothesis can be removed considering the fact that some values are more likely to be used than others. The measured relative frequency of each instruction is thus considered an estimate of its probability.

Experiments performed on a large set of benchmarks have led to the results summarized in table 2. The table reports the relative errors obtained by applying the four approaches described. The comparison of the results shows that the two best methods are the *complete uniform* and the *complete weighted*, the latter being slightly more accurate.

Method	Error
Different figures only	7.21%
Different translations only	4.31%
Complete uniform	3.14%
Complete weighted	2.71%

Table 2. Relative errors

3.3 Source–level model

At source–level, the degree of detail available is limited but even a rough power estimate may be very helpful to the designer. At this level of abstraction the current drawn by the microprocessor during the execution of an assembly instruction can be considered constant. Under this assumption, a power characterization can be derived by calculating the time needed to complete the execution of a given code. The model outlined in the following addresses this problem. The time $T(I)$ consumed by the complete execution of a generic instruction I can be expressed as:

$$T(\overline{I}) = cpi(\overline{I}) \cdot T_{SW} \tag{14}$$

where the function $cpi(\cdot)$ denotes the number of clock cycles and T_{SW} the clock period. This concept can be generalized to a process[1] γ introducing the new function $cpp(\cdot)$:

$$T(\gamma) = cpp(\gamma) \cdot T_{SW} \tag{15}$$

The two functions $cpi(\cdot)$ and $cpp(\cdot)$ are acronyms of *Clock–cycles Per Instruction* and *Clock–cycles Per Process*, respectively. Let P_i be a generic microprocessor and $\mathcal{IS}_i$ its instruction set. Let then $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$ be a set of p processors supporting instructions with the same maximum number of operands (typically one, two or three). The generic instruction set $\mathcal{IS}_i$ can be partitioned into a fixed number c of predefined *instruction classes* $\mathcal{IC}_{i,j}$ performing similar operations, such as data transfer, load/store, branch, etc. The instruction classes must satisfy these relations:

$$\mathcal{IS}_i = \bigcup_{j=1}^{k} \mathcal{IC}_{i,j} \tag{16}$$

$$\mathcal{IC}_{i,j_1} \cap \mathcal{IC}_{i,j_2} = \emptyset \; \forall j_1, j_2 \in [1; k] \tag{17}$$

Instruction sets of different processors may significantly differ: for this reason a specific processor may have one or more empty instruction classes. Two instructions $I_1 \in \mathcal{IS}_1$ and $I_2 \in \mathcal{IS}_2$ belonging to different instruction sets are said to be *compatible* if and only if:

$$\exists j \mid I_1 \in \mathcal{IC}_{1,j} \wedge I_2 \in \mathcal{IC}_{2,j} \tag{18}$$

[1] In this context the term *process* is used to indicate a generic part of the source code of an application or algorithm.

Considering all the p processors in $\mathcal{P}$ and their instruction sets $\mathcal{IS}_i$, it is possible to define a number k of *compatible instruction classes* satisfying the following relation:

$$\mathcal{CIC}_j = \begin{cases} \emptyset & \text{if } \exists i \mid \mathcal{IC}_{i,j} = \emptyset \\ \bigcup_{i=1}^{p} \mathcal{IC}_{i,j} & \text{otherwise} \end{cases} \tag{19}$$

These new instruction classes collect all the instructions of different processors that are compatible in the sense that all the instructions in the same class perform equivalent operations. The union of all $\mathcal{CIC}_j$ classes can be thought of as a generic instruction set denoted as $\mathcal{CIS}$ or *Compatible Instruction Set*.

Let $\mathcal{CIC}_j = \{I_{j,1}; I_{j,2}; \ldots; I_{j,N_j}\}$ be the j-th compatible instruction class and N_j its cardinality. We can determine two instructions $I_{U,j}$ and $I_{L,j}$ in each $\mathcal{CIC}_j$ such that their *cpi* are maximum and minimum, respectively:

$$I_{U,j} = MAX_{n=1}^{N_j}\ cpi(I_{j,n}) \tag{20}$$

$$I_{L,j} = MIN_{n=1}^{N_j}\ cpi(I_{j,n}) \tag{21}$$

The two instructions $I_{U,j}$ and $I_{L,j}$ represent the bounding cases for the j-th instruction class. Consider now a generic instruction $\overline{I}$ executed in $cpi(\overline{I})$ clock cycles. If $\overline{I}$ belongs to the $\overline{j}$-th compatible instruction class then an upper–bound to its execution times is $cpi(I_{U,\overline{j}})$ and, similarly, a lower–bound is $cpi(I_{L,\overline{j}})$. If $\overline{I}$ does not belong to any of the compatible instruction classes, then there exists no single instruction in the compatible instruction set that can perform the same operation. Its functionality must thus be obtained by combining more than one instruction in $\mathcal{CIS}$.

The upper and lower bounds for the instruction $\overline{I} \in \mathcal{CIS}$ can thus be formally defined introducing the following two functions:

$$cpimax(\overline{I}) = cpi(I_{U,\overline{j}}) \mid \overline{I} \in \mathcal{CIC}_{\overline{j}} \tag{22}$$

$$cpimin(\overline{I}) = cpi(I_{L,\overline{j}}) \mid \overline{I} \in \mathcal{CIC}_{\overline{j}} \tag{23}$$

Thus far, only single instructions have been considered while the microprocessor architecture has been neglected. In particular, up to now we did not consider the number of available registers, which is known to strongly influence the timing properties of the software. To account for the different number of registers available on different architectures the technique described in the following has been adopted.

Consider the source code γ_{src} of a process γ, two ideal microprocessors P_2 and P_∞, identical with respect to all their characteristics except the number of registers. Let the microprocessor P_2 have 2 register and P_∞ have an unlimited number of registers, pre–loaded with all necessary data. The source code, compiled for the two microprocessors, will result in two different assembler codes:

$$\Gamma_2 = [I_{2,1};\ I_{2,2};\ \ldots\ ;\ I_{2,M_2}] \tag{24}$$

$$\Gamma_\infty = [I_{\infty,1};\ I_{\infty,2};\ \ldots\ ,\ I_{\infty,M_\infty}] \tag{25}$$

composed of M_2 and M_∞ instructions, respectively. On this basis it is possible to define two bounding values for the timing of the process γ as two functions:

$$cppmax(\gamma) = \sum_{s=1}^{M_2} cpimax(I_{2,s}) \tag{26}$$

$$cppmin(\gamma) = \sum_{s=1}^{M_\infty} cpimin(I_{\infty,s}) \tag{27}$$

These conditions are referred to as worst–case and best–case, respectively. The same source code γ_{src}, compiled on an actual processor $P_i \in \mathcal{P}$, results in an assembly code $\Gamma_i = [I_{i,1};\ \ldots\ ;\ I_{i,M_i}]$ whose actual timing is given by:

$$cpp(\gamma) = \sum_{s=1}^{M_i} cpi(I_{i,s}) \tag{28}$$

An estimate $cpp_{est}(\Gamma_i)$ of $cpp(\Gamma_i)$ of the timing of process γ compiled for the generic processor P_i is:

$$cpp_{est}(\gamma) = cppmin(\gamma)^{\alpha} \cdot cppmax(\gamma)^{(1-\alpha)} \tag{29}$$

The value of α depends on a number of factors: the specific process, the compiler used, the compilation options, etc. To derive a good estimate of this parameter, benchmarking is necessary. Let α_r be the value corresponding to the process γ_r and consider different frameworks $f_{SW} \in \mathcal{F}_{SW}$ (compiler, options, etc.). An estimate $\alpha_{est,r}$ of α_r can be obtained by minimizing the square error:

$$\epsilon_r^2 = \sum_{f_{SW} \in \mathcal{F}_{SW}} \left[cpp(\gamma_r) - cpp_{est}(\gamma_r)\right]^2 \tag{30}$$

The results of benchmarking can then be combined to give the overall estimate α_{est} of α according to the following, simple, equation:

$$\alpha_{est} = \frac{1}{N_\gamma} \cdot \sum_{r=1}^{N_\gamma} \alpha_{est,r} \tag{31}$$

where N_γ is the number of processes considered.

Benchmarking has been performed on a large set of etherogeneous source codes, yielding $\alpha = 0.75$.

4 Power Estimation Flow

The models described in the previous section have been implemented in a complete power estimation flow. The flow operates at all the three levels of abstraction and allows different estimation paths, as depicted in figure 2. In the figure, solid arrows represent power estimation processes and are performed at a specific level of abstraction (source, VIS or assembly), while dashed arrows indicate back–annotation processes. To obtain a power characterization of the source code, three different paths are possible:

- **Fast.** An estimate is derived directly from the source code, based on the model described in Section 3.3. The accuracy that can be expected is within a 15–20% error and the estimation time is less than a second.

- **Intermediate.** The source code must first be compiled to VIS then estimation is performed according to the model described in Section 3.2. The power figures at VIS level can be finally back–annotated to the source code. The accuracy is within a 3–6% error and the time necessary for the complete process (compilation, estimation and back–annotation) is 1–5 seconds.

- **Accurate.** This path allows a very accurate estimate but requires compiling the source code into VIS and then mapping the VIS code to target assembly. On the assembly code an estimation can be performed based on the model described in Section 3.1. Data collected at assembly level can be back–annotated to the VIS and finally up to the source code. The accuracy obtained with this procedure is within 1%.

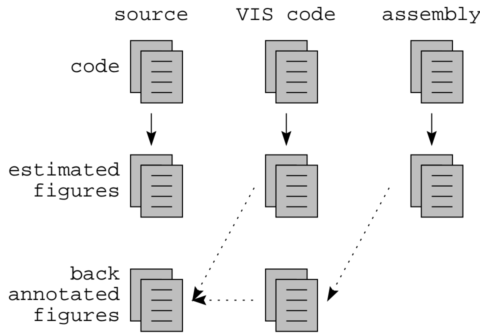

Figure 2. The estimation flow

The computation times reported refer to a source code of approximately 100 lines and errors are calculated with respect to the results obtained with a power aware instruction–set simulator.

5 Results and Conclusions

The models and the flow presented in this paper have been applied to a wide variety of examples and to an industrial application leading to the results summarized in tables 3 and 4. The largest software considered is a commercial 16-channel link controller ILC16 developed at Italtel R&D Labs. The source code, written in OCCAM2, is 2840 lines long and is composed of 54 procedures. This controller makes extensive use of concurrency and blocking unidirectional point–to–point communication with *rendez-vous* semantics. The OCCAM2 language has been selected because it naturally allows the definition sequentiality as well as concurrency at process level and provides an abstract communication paradigm based on blocking channels. The source code has been compiled for two target processors: Motorola MC68000 and Arm Ltd. ARM7TDMI in Thumb (low–power) mode [6]. The resulting assembly codes are roughly 89000 and 75000 lines long. The flow has been run on a single–processor Sun Enterprise 250 under Solaris 7.

Processor		Assembly	VIS	Source
ARM7	s	5.98	6.15	7.08
	mA	11.36	11.35	10.99
	mJ	224.48	230.65	256.77
MC68K	s	16.99	18.13	19.04
	mA	13.85	13.50	13.20
	mJ	776.52	807.69	829.38

Table 3. Estimates for the ILC16 design

Processor	Accurate	Intermediate	Fast
ARM7	37.57 s	22.07 s	3.91 s
MC68K	53.17 s	29.96 s	4.12 s

Table 4. Run times for the ILC16 design

The time figures reported in table 3 refer to a run with typical input data. The presented methodology still neglects some dynamic effects such as cache misses and pipeline stalls/refills. These problems are currently under investigation and some refinements to the source–level model are being studied. Nevertheless, the results obtained are encouraging and the accuracy achieved is acceptable for fast design–space exploration.

References

[1] E. Macii, M. Pedram, F. Somenzi, *"High-Level Power Modeling, Estimation, and Optimization,"* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 11, 1998.

[2] P.W.Ong and R.H.Yan, *"Power-conscious software design: a framework for modeling software on hardware,"* Proc. of 1994 IEEE Symposium on Low Power Electronic, pp. 36-37, San Diego, CA, Oct. 1994.

[3] V. Tiwari, S. Malik and A. Wolfe, *"Power Analysis of Embedded Software: a First Step towards Software Power Minimization,"* IEEE Transactions on VLSI Systems, Vol. 2, No. 4, pp. 437-445, Dec. 1994.

[4] V. Tiwari and M.T.-C. Lee, *"Power analysis of a 32-bit Embedded Microcontroller,"* VLSI Design Journal, 1996.

[5] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto *"An Energy Estimation Model for 32-bit Microprocessors,"* DAC2000, Los Angeles, CA, June 2000.

[6] PEOPLE ESPRIT project n.26769, Deliverable D1.2.1.

[7] Online documentation of the PEOPLE ESPRIT Project *http://www.cefriel.it/Eda/Projects/*