Dynamic Modeling of Inter–Instruction Effects for Execution Time Estimation

G. Beltrame[†], C. Brandolese[§], W. Fornaciari[§], F. Salice[§], D. Sciuto[§], V. Trianni[§]

[§] Politecnico di Milano, Piazza L. da Vinci, 32 - 20133 Milano, Italy

{brandole,fornacia,salice,sciuto,trianni}@elet.polimi.it

[†] CEFRIEL, Via R. Fucini, 2 - 20133 Milano, Italy

beltrami@cefriel.it

ABSTRACT

The market for embedded applications is facing a growing interest in power consumption issues: this work is intended to provide a new model to estimate software–level power consumption of 32-bit microprocessors. This model extends previous ones by considering dynamic inter–instruction effects that take place during code execution, providing a static means to characterize their energy consumption. The model is formally sound: it is conceived for a generic architecture and it has been preliminary validated on the Intel486™ architecture.

1. INTRODUCTION

While there has been a significant research effort in power estimation techniques and in low power design tailored for hardware systems, no EDA tools are available to help hardware/software embedded systems designers [4]. The main obstacle is an efficient analysis of the CPU power consumption, necessary to take into account also the software components during design–space exploration, while avoiding to rely with architectural or even layout–level simulation of the microprocessor. To fill such a gap, strategies working at the instruction–level recently appeared in literature. In fact, having a power model of assembly instructions is a value added for designers, since the increasing complexity of embedded systems software is evident and the need of early prototyping of embedded systems, in particular in terms of power consumption, is becoming a must. In [12][13][11] *a priori* knowledge of the current drawn by an instruction is obtained by executing an infinite loop of the target instruction in order to average out fine–grained fluctuations. These approaches are strongly processor–dependent and normally the statistical significance of power figures is not taken into account. A different approach, working on the concept of early *virtual* prototyping of the software for different target CPU cores has been proposed in [1]. This methodology abstracts from the architectural level and focuses on the *functionalities* involved during instruction execution. The resulting functional model decouples the execution time of an instruction from its average power consumption. This allows a *static* (data independent) characterization of each instruction in terms of the energy consumed together with a statistical validation of the resulting software power model. These assumptions are the basis for the work presented in this paper, which concentrates on timing aspects. To consider also the presence of *dynamic* inter–instruction effects such as pipeline interlocks or cache misses, which typically lead to an additional energy consumption not to be neglected in a overall system–level perspective, the previous approaches should be extended. In fact, even if estimates are accurate for the static microprocessor model, the introduction of dynamic inter–instruction effects may cause severe strays from reality for the entire system. The importance of this problem has been recognized in [10], where an instruction–level power model that considered dynamic effects is presented. The solution is based on a modification of the model proposed in [12][13] to obtain a more precise estimate for the base costs. Basically, the authors separated instructions with the same opcode but different addressing modes and added a statistical analysis of cache and pipeline interlock overheads. Unfortunately, this solution is still not general, in the sense that it needs measures for every processor it has to be applied to. The goal of this paper is to overcome the above limitations, providing a general model to describe interlock overheads for different types of processor cores, to complete the information provided via *static* analysis. This model is going to be implemented in a co-design flow in order to obtain a truly accurate and efficient software power estimation tool [9]. This paper is organized as follows: Section 2 defines the problem of considering inter–instruction effects in the scope of a static analysis; Section 3 introduces the mathematical and statistical models; Section 4 describes the methodology for model tuning, and Section 5 presents some experimental results. Finally, some conclusions are drawn in Section 6.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

2. PROBLEM DEFINITION

In this work, the model proposed in [1] is extended, taking into account inter–instruction effects related to pipelining. Interlocks are generally related to the execution of a particular sequence of instructions, thus they correspond to *dynamic* events. Taking into account such effects implies

either a dynamic analysis, which requires an excessive computational complexity, or an accurate characterization of the computations made by the processor, which leads to a loss of generality [1].

When dealing with inter–instruction effects it is not possible to avoid a dynamic analysis. To recognize a hazard arising from the execution of a sequence of instructions, it is necessary to model the pipeline behavior of the target architecture and the instruction flow in the pipeline, thus leading to an excessive computational complexity. But if an *a priori* statistical characterization of the inter–instruction behavior of each instruction is known, then the static approach can still be applied: this correspond to a static representation of dynamic effects. The proposed approach is based on a *dynamic analysis* aimed at deriving a statistical model of the delay introduced by inter–instruction effects. Such model is then used *statically* during the estimation of the timing and energy requirements of each instruction. The dynamic analysis is made *once* for each target architecture, obtaining statistical information about the delay introduced in the execution time of each instruction due to pipeline interlocks. This temporal overhead is then combined with the purely static model [1], allowing an extension into the energy consumption domain, as detailed in the conclusion of Section 3.2. As mentioned above, the paper focuses on inter–instruction effects arising from a pipelined execution of the code. In this work pipeline stalls related to caches and memories are deliberately neglected. When a hazard is detected, the pipeline might be stalled for a given number of clock cycles. Three hazard types may arise [3]:

Structural hazards are due to limitations in the data–path, which cannot support certain instruction sequences.

Data hazards are related to the semi–parallel execution of the code, which may lead to an incorrect ordering of read/write operations.

Control hazards are related to the branch execution, since the pipeline generally needs to be stalled until the target address is calculated.

The present work analyzes the pipeline behavior of the two target architectures, namely the microSPARC™-IIep [6][7] an the Intel Embedded486™ [5]. Based on this analysis a general abstract model of the interlock behaviour has been derived and applied to a different microprocessor, as reported in Section 5. In general, a hazard may be associated with a couple of instructions occurring at a given distance in time; the penalty introduced by the resulting pipeline stall depends on both the specific hazard and the distance between the instructions. The former instruction in the pair is the cause of the interlock[1] while the latter the stalled one. For this reason the temporal overhead is conventionally associated with the latter. The proposed model estimates a statistical temporal overhead to be associated to an instruction pair and folds it on single instructions, as it will be presented in section 3. The estimates are obtained by means of a dynamic analysis carried out on a significant benchmark set; to this purpose, the *execution trace* of the chosen benchmarks is used. The trace carries the information of the instruction flow into the pipeline. The model is dynamic in the sense that it is tuned on execution traces rather than on the assembly source code but, once tuned, it is used statically to obtain the desired instruction characterization. However, an execution trace does not explicitly contain all the dynamic information on the processor state. For this reason, it is necessary to extract the implicit information from the trace, as it will be explained in section 4.

[1]In some uncommon cases the actual cause of a stall is a particular sequence of instructions rather than a single instruction.

3. MATHEMATICAL MODEL

For the purpose of producing a static estimation of the delay introduced by inter–instruction effects, a *taxonomy* of instruction sets has been proposed, which describes all possible hazards in an architecture independent manner. This taxonomy is essential to reduce model complexity and to allow a computationally feasible statistical analysis. Based on such a taxonomy, a mathematical model for the estimation of the temporal overhead caused by inter–instruction effects is then introduced.

3.1 Model Definition

The instruction set taxonomy provides some general architecture independent classes to be associated with architecture specific instructions. These classes are defined according to the type of hazard that an instruction may cause. Each instruction s of a given instruction set I is assigned to the class that best represents its dynamic behavior. Since three hazard types are possible, the taxonomy contains a maximum of eight classes c_h, each representing the set of instructions that may cause the same hazard type. The taxonomy $\mathcal{C}$ can be defined as:

$$\mathcal{C} = \{c_h \subseteq I | h \in [0,7]\}$$

To define the mathematical model of the temporal overhead, an assembly execution trace must be considered. Let Γ be an execution trace, i.e. an ordered sequence of instructions:

$$\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_N\}, \quad \gamma_k \in I, \quad N, k \in \mathbb{N} \tag{1}$$

where N represents the execution trace size. To estimate the probability of finding a taxonomy class pair in the trace Γ, two operators have been introduced according to the following definitions.

DEFINITION 1. *The* ***distance*** $w(\gamma_{k_1}, \gamma_{k_2})$ *between two instructions* γ_{k_1} *and* γ_{k_2} *is defined as the difference* $|k_2 - k_1|$. *The following notation is used to point out that two instruction* γ_{k_1}, γ_{k_2} *occur at a distance* $\hat{w}$:

$$\gamma_{k_1} \overset{\hat{w}}{\dashv} \gamma_{k_2}, \quad \hat{w} = w(\gamma_{k_1}, \gamma_{k_2})$$

The considered distances $\hat{w}$ should not be greater than the pipeline depth, i.e. 5-15, since farther instruction are almost independent in terms of interlock behavior.

DEFINITION 2. *The* ***membership function*** *of instruction* $\gamma_k \in \Gamma$ *to* $c_i \in \mathcal{C}$ *is defined as:*

$$\langle k, i \rangle = \begin{cases} 1 & \text{if } \gamma_k \in c_i \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Definitions 1 and 2 can be then combined to introduce the concept of *event* as:

$$c_i \overset{\hat{w}}{\dashv} c_j \Leftrightarrow \exists (k_1, k_2) : \begin{cases} \langle k_1, i \rangle = 1 \\ \langle k_2, j \rangle = 1 \\ \gamma_{k_1} \overset{\hat{w}}{\dashv} \gamma_{k_2} \end{cases} \tag{3}$$

meaning that there exists in Γ two instructions $\gamma_{k_1} \in c_i$ and $\gamma_{k_2} \in c_j$ that occur at distance $\hat{w}$. Let us now introduce a statistical characterization of the events described above by examining the presence of particular classes of instructions in the execution traces. To this purpose, the frequency definition of probability is used [8].

DEFINITION 3. *The probability of finding class c_i in the execution trace is:*

$$P(c_i) = \frac{1}{N}\sum_{k=1}^{N}\langle k, i\rangle$$

where N is suitably large[2].

From this definition the relation $\sum_{i=0}^{7} P(c_i) = 1$ can be easily proved. Let us now consider class pairs statistics.

DEFINITION 4. *The probability of finding class c_i and class c_j at distance $\hat{w}$ in the execution trace is*

$$P(c_i \overset{\hat{w}}{\dashv} c_j) = \frac{1}{N}\sum_{k=1}^{N}\langle k, i\rangle\langle k+\hat{w}, j\rangle, \quad N \gg \hat{w}$$

where N is, again, suitably large[3].

Definitions 3 and 4 are strictly linked; in fact, the following relation holds:

$$P(c_j) = \sum_{i=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j), \tag{4}$$

This relation proves the consistency of the two definitions and thus, in turn, the soundness of the presented model.

3.2 Interlock Model

Thus far, a characterization of the frequencies of pairs of classes in the execution trace has been obtained while the properties of the interlocks that might arise have been neglected. It is worth noting that different pairs of instructions have different interlock behavior and latency, even if they are represented by the same couple of classes. In order to maintain both generality and accuracy it is thus necessary to consider the interlocks as produced by instructions pairs, and then to aggregate these figures up at class level. To do this it is necessary to abandon the exact, analytic view of the delay overhead introduced by single instruction pairs and rather consider such delays as random variables. The function $t(\gamma_k, \gamma_{k+\hat{w}}, \hat{w})$ that returns the *delay* introduced by the execution of an instruction pair $\gamma_k, \gamma_{k+\hat{w}}$ at a distance $\hat{w}$ is introduced. A delay of zero means that no interlock occurs. In order to take into account each possible interlock given a pair of classes at a distance $\hat{w}$, it is necessary defining the delay introduced by inter–instruction effects as a random variable.

DEFINITION 5. *The **class pair delay** is the delay introduced by the execution of a class pair (c_i, c_j) at a distance $\hat{w}$ and is modeled by the random variable $D_{i,j,\hat{w}}$. This variable is characterized by its density function:*

$$f_{D_{i,j,\hat{w}}}(d) = \frac{\sum_{k=1}^{N}\delta_{t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})=d}\langle k, i\rangle\langle k+\hat{w}, j\rangle}{\sum_{k=1}^{N}\langle k, i\rangle\langle k+\hat{w}, j\rangle}$$

where N is suitably large and $\delta_{t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})=d}$ is the Kronecker symbol, defined as:

$$\delta_{t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})=d} = \begin{cases} 1 & \text{if } t(\gamma_k, \gamma_{k+\hat{w}}, \hat{w}) = d \\ 0 & \text{otherwise} \end{cases}$$

Given i, j, $\hat{w}$, and d, $f_{D_{i,j,\hat{w}}}(d)$ represents the relative frequency of d-delay interlocks with respect to the class pair (c_i, c_j). Interlock latency associated with a single class—rather than a pair of classes—turns out to be particularly useful when a fast estimation, performed at source level, [2] is required.

Definition 6 formally introduces the idea of class delay.

DEFINITION 6. *The **class delay** is the delay associated with the execution of a class c_j paired with any other class at a distance $\hat{w}$, and is modeled by the stochastic variable $D_{j,\hat{w}}$. This variable is characterized by its density function:*

$$f_{D_{j,\hat{w}}}(d) = \frac{\sum_{k=1}^{N}\delta_{t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})=d}\langle k+\hat{w}, j\rangle}{\sum_{k=1}^{N}\langle k+\hat{w}, j\rangle}$$

Definitions 5 and 6 are bound by the relation:

$$f_{D_{j,\hat{w}}}(d) = \frac{\sum_{i=0}^{7} f_{D_{i,j,\hat{w}}}(d) P(c_i \overset{\hat{w}}{\dashv} c_j)}{\sum_{i=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j)} \tag{5}$$

stating that the density function of the class delay $D_{j,\hat{w}}$ is equal to the sum of the density functions of the pair delays $D_{i,j,\hat{w}}$, weighted by the frequency of the corresponding pair. The importance of this relation is twofold: on one hand, it stresses the formal correctness of the model, as already stated by equation 4; on the other hand, it provides a means to estimate the class delays starting from class pair delays. Class delays can be used to obtain an energy consumption measure, by integrating them into the model described in [1]. According to the cited model, the energy consumption of an assembly instruction is expressed as the average current drawn by the processor during its execution and can be calculated as:

$$\mathbf{IN} = \mathbf{A} \times \mathbf{IF} \tag{6}$$

where **IN** is a vector collecting the total currents of all the instructions $s \in I$, **IF** is a vector storing the average currents per clock cycle associated to a predefined set of processor abstract functionalities and **A** is a matrix accounting for the execution times of each instruction s with respect to these functionalities. Since the main effect of interlocks is an increase of the execution time, a natural way to extend the power model is to add an overhead to the matrix **A**. The new model maintains the same algebraic form provided that **A** is substituted with $\mathbf{A}' = \mathbf{A} + \mathbf{OH}$, where **OH** stores the class overheads, suitably distributed over the different functionalities.

4. MODEL TUNING

This section presents the methodology adopted to tune the proposed model and to provide the estimates for the temporal overhead introduced by inter–instruction effects.

[2]The trace length N of a medium–sized assembly program is about $10^6 - 10^7$ instructions.

[3]Since $N \gg \hat{w}$ the upper limit of the summation $N - \hat{w}$, i.e. the total number of pairs, can be conveniently approximated with N.

Since timing and power effects are decoupled, as pointed out by equation (6), a first tuning phase can be performed considering timing properties only. A validation in terms of energy consumption is out of the scope of this paper. The model proposed in section 3 has been implemented by a tool to be integrated in a co–design environment for evaluating the energy consumption of embedded systems. Such a tool should consider different target architectures and thus its implementation must be as modular as possible. As mentioned above, hazards can be generally considered as particular *events* that occur during the code execution. The tool implementation consists of two portions:

- an architecture dependent inter–instruction model represented by a *list*. When a hazard condition arises (the event), it is detected by matching a corresponding *event model* in the list. This is the implementation of the delay function $t(\gamma_k, \gamma_{k+\hat{w}}, \hat{w})$ defined in section 3.2. Different architectures are characterized by means of a specific configuration file;
- an architecture–independent module responsible of providing a general representation of the assembly, and deriving the density functions associated with stochastic variables.

Estimation of the stochastic variables requires the three steps described below (see figure 1):

- produce an execution trace of a significant number of selected benchmarks;
- parse the assembly code and build an architecture–independent representation;
- check the hazards conditions and derive the distributions of the stochastic variables.

The first step of the tuning process is the generation of the execution trace which carries the dynamic information of the instruction flow into the processor pipeline. The main problem is the selection of a suitable set of benchmarks to be traced. Once a number of such execution traces has been created, the estimated densities of the random variables are assumed to represent a good statistic characterization of the temporal overhead associated with each instruction class pair at a given distance[4]. Creating a suitable execution trace is not trivial at all: the aim is to observe the real dynamic behavior of the pipelined execution, thus it is necessary to obtain data corresponding to a realistic computation. In fact, as pointed out in section 3.1, it is important not only to detect hazards, but also to know the actual probability of finding an instruction pair that flows in the pipeline at a given distance. The adopted approach consists thus in generating the execution traces of real programs operating on real data sets, in such a way to cover a sufficiently wide range of application typologies. The benchmarks used cover typical applications such as image processing, text manipulation and networking. Figure 2 shows the probability of finding all class pairs in the selected execution traces, and the corresponding aggregate values, calculated using equation (4). These values are assumed to represent all the available knowledge on the dynamic behaviour of typical programs. This assumption is justified by the fact that

[4]Distance is a fixed parameter

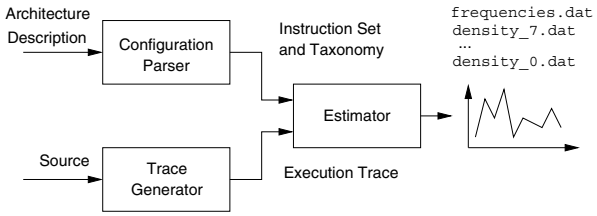

Figure 1: Model tuning execution flow

the obtained data do no exhibit any dependence on the specific benchmarks used. The size of the execution trace is in the range of 10^6-10^7 code lines, i.e. large enough to provide a satisfactory accuracy of the estimates. The second step of the tuning process is aimed at creating a general representation of the given architecture, with respect to both its instruction set and its dynamic behavior. The importance of this process is twofold: one one hand, it allows to apply the same methodology and perform an estimation of the dynamic behavior on different architectures; on the other hand, it is intended to be compatible with the static energy estimation model [1], which will be extended by considering inter–instruction effects. The last step is responsible of detecting hazards operating on an abstract representation of the execution trace. To this purpose, abstract models of the possible hazard conditions must be built. As an example, consider a typical RAW hazard of the Intel Embedded486™: the pipeline stalls for one clock cycle when an instruction that writes a register is immediately followed by an instruction that reads the same register; the following Boolean expression is the abstract model for this specific hazard:

$$(op_1 \equiv *) \wedge (op_2 \equiv *) \wedge ((rd_1 = rs_2[1]) \vee (rd_1 = rs_2[2])), \quad (7)$$

where op_1 is the op–code of the former instruction in the pair, op_2 the one of the latter instruction, rd_i is a register written by instruction i in the pair and $rs_i[j]$ is the j^{th} source register of instruction i; the '$*$' is a wildcard, standing for any operation. This expression evaluates to true when

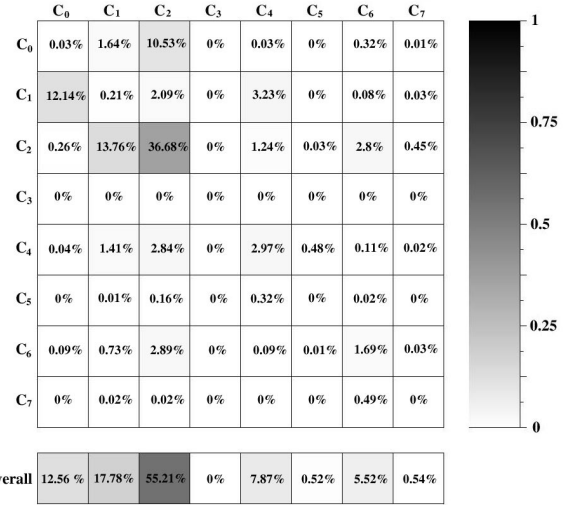

Figure 2: Class pair distance probabilities and aggregate class values

an interlock occurs. For the sake of clarity, consider the following instruction pair:

```
lea  0x4(%ebx,%esi,4),%eax
mov  %eax,0x8078eec
```

The pair will match the model, allowing the recognition of the interlock. For every instruction pair in the execution trace a delay value associated with a class pair can be computed; this process results in a number of random variables describing the statistical overhead introduced by each class pair. Using equation (5) these data can be folded on single classes, obtaining a set of random variables which can be statically used to estimate the timing requirements of each instruction. Table 1 shows the average value of the estimated temporal overheads for all classes, in clock cycles.

Table 1: Class temporal overhead introduced by inter–instruction effects

Class	Average delay	Class	Average delay
c_0	0.433673	c_4	0.515101
c_1	0.185519	c_5	0.108412
c_2	0.484779	c_6	0.499436
c_3	0.000000	c_7	0.042164

Once the random variables have been derived, two different estimation approaches can be devised. A first coarse analysis can be done by averaging the values of the class pair random variables and fold them to single classes. A finer analysis, on the other hand, might exploit all the information contained in the random variables by collecting and combining density functions rather than average values, leading to a statistical characterization of code portions or even entire functions.

5. EXPERIMENTAL RESULTS

The tuning of the model has been made by choosing $\hat{w} = 1$ because at this distance more hazards may arise. As figure 2, points out, class c_3 never appears and has thus an overall zero frequency in the code: in fact, such a class represents all the instructions that may cause both data and control hazards. This condition is never satisfied in the Intel486™. As table 1 shows, the average overhead introduced by each instruction class is generally significant. The model should be validated at two different levels: energy and time. In fact, the translation from the temporal overhead introduced by inter–instruction effects to the actual energy consumption associated with them introduces an additional level of abstraction that need to be confirmed by experimental data. Three solutions have been proposed:

- use an *Instruction Set Simulator* (ISS) which can produce exact timings for both activities of instruction execution and interlock management. This method refers only to timing measures and is only viable if a cycle–accurate instruction set simulator is available;
- simulate the code execution on a given architecture by means of its RTL–HDL model. Though such a solution offers a better accuracy, it is hardly applicable due to the scarce availability of RTL models of processor architectures and to the extremely high computational requirements;
- compute the timing on a test code directly during its execution, overriding delays due to the operating system and to interrupts. Though slightly less accurate, this approach is easily feasible.

The last approach has been adopted and the result obtained are presented in the remaining part of this section. The validation methodology has been applied as follows (see figure 3). A test program is first compiled, then executed and disassembled. The static data in the disassembled code is then combined with the dynamic information derived from the execution trace and then fed to an *annotation tool* which estimates—in an interlock–free manner—the timing of the code. The annotation is based on an *a priori* knowledge of the CPIs of every instruction, which typically depends on its operation code and its addressing modes. The same execution trace is analyzed by a dynamic effects estimator, which produces the total temporal overhead associated with the pipeline interlocks. Interlock–free timing data and temporal overheads are then added to obtain an overall estimate of the test program timing. The estimation of the interlock tempo-

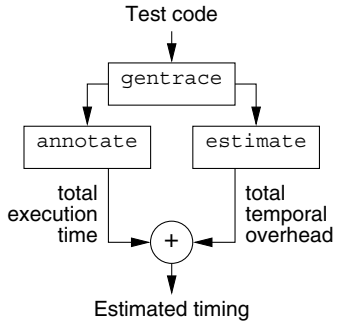

Figure 3: Code execution timing estimation

ral overhead is performed based on the knowledge of the random variable associated with each instruction class: every instruction is contained in a taxonomy class, and thus inherits the class average temporal overhead which contributes to the total value. In this way, the contribution of the different classes is weighted by the actual frequencies of instructions in the traced code. The result is a more precise estimate of the timing that is then compared with the actual execution time. Five integer benchmark programs—not belonging to the tuning set—have been used to validate the model:

1. `genprim` generates a 3-digits prime number;
2. `rle` computes the run–length encoding of a string;
3. `crc16` computes the 16–bit CRC on strings;
4. `qsort` implements Quicksort for integers;
5. `md5` computes the digest of a given string.

Table 2 shows the relative errors obtained. The interlock–free static analysis obtained with the annotation tool has an average error around -24.1%, the reason being that it underestimates the clock cycles needed by the execution of each instruction. Considering the inter–instruction effects as well results in a considerable improvement: the error reduces to an average of -1.5%, indicating a much better overall estimation of the dynamic behavior of the processor.

Table 2: Relative errors of the interlock–free and interlock–aware models

Test case	Relative error interlock–free	interlock–aware
`genprim`	-32.1%	-9.4%
`rle`	-17.9%	+2.7%
`crc16`	-22.1%	+0.3%
`qsort`	-22.1%	-3.7%
`md5`	-26.5%	+2.5%
Overall	-24.1%	-1.5%
Overall (absolute)	24.1%	3.7%

It is important to remark that the applied analysis still ignores dynamic effects resulting from cache misses. For this reason all the benchmarks have been tailored in order to avoid or limit the number of cache misses. An accurate analysis of the results obtained has confirmed that the higher negative error obtained for `genprim` is mostly due to the presence of some uncommon complex addressing modes, currently ignored by the annotation tool.

6. CONCLUSIONS

In this paper a methodology for analyzing dynamic effects in pipelined architectures has been proposed. It is based on a rigorous mathematical model exhibiting satisfactory statistical properties. The results obtained suggest that the model is promising, while still in the early stages of validation. In particular the methodology has been applied to real cases and the estimated execution times have been proved to be much more accurate than those obtained neglecting timing dynamic components related to inter–instruction effects. Estimation errors are in the range of a few percent and are biased towards underestimation, as the model predicts. Such improved accuracy will enable a more precise energy estimation of embedded software programs. The timing model has been effortlessly integrated in a more general power estimation framework and preliminary results on the accuracy of the extended power model are under investigation. Future work will consider the integration of cache miss analysis or, more generally, of memory–related inter–instruction effects. The application of a better validation methodology and of a finer model tuning is currently in progress.

7. REFERENCES

[1] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. An instruction–level functionality–based energy estimation model for 32-bits microprocessors. In *Proceedings of 37th IEEE Design Automation Conference*, pages 346–351, Los Angeles, CA, June 2000.

[2] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source–level execution time estimation of c programs. In *Proceedings of International Workshop Hardware Software Codesign*, Copenhagen, Denmark, April 25-27 2001.

[3] J. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, II edition, 1996.

[4] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1061–1079, November 1998.

[5] Intel Architecture Software Developer's Manual vol. 1. Technical report, Intel Corporation, 1997.

[6] The SPARC Architecture Manual, version 8. Technical report, Sun Microsystems, 1990.

[7] microSPARC™-IIep user's manual. Technical report, Sun Microsystems, 1997.

[8] A. Mood, F. Graybill, and D. Boes. *Introduction to the theory of statistics*. McGraw–Hill, New York, NY, 1988.

[9] PEOPLE. (Power Estimation for Fast Exploration of Embedded Systems). Technical Report D3.3.1, ESPRIT-ESD project n.26769, 1998.

[10] S. Ramalingam and K. Schindler. Instruction level power model and its application to general purpose processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 753–756, 1998.

[11] J. Russell and M. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings of ICCD'98, International Conference on Computer Design*, pages 328–333, Austin, TX, October 1998.

[12] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.

[13] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of the Intel 486DX2. Computer Engineering Technical Report No. CE-M94-5, Princeton University, June 1994.