An Assembly–Level Execution–Time Model for Pipelined Architectures

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

[§] Politecnico di Milano, Piazza L. da Vinci, 32 - 20133 Milano, Italy
[†] CEFRIEL, Via R. Fucini, 2 - 20133 Milano, Italy

Abstract

The aim of this work is to provide an elegant and accurate static execution timing model for 32-bit microprocessor instruction sets, covering also inter–instruction effects. Such effects depend on the processor state and the pipeline behavior, and are related to the dynamic execution of assembly code. The paper proposes a mathematical model of the delays deriving from instruction dependencies and gives a statistical characterization of such timing overheads. The model has been validated on a commercial architecture, the Intel486, by means of timing analysis of a set of benchmarks, obtaining an error within 5%. This model can be seamlessly integrated with a static energy consumption model in order to obtain precise software power and energy estimations.

1 Introduction

The peculiarities of target application fields of embedded computing (e.g., mobile systems), typically pose stringent area and energy constraints. The current trend towards high–levels of integration up to system–on–chip, is exacerbating the need of taking into account power requirement during the early stages of the design as well as throughout the entire verification flow. In addition, the *penetration* of software within the typical hardware/software architectures used in embedded systems, is steadily gaining importance but unfortunately efficient power aware compiling and estimation techniques are still a research topic not yet mature for the EDA arena. Previous approaches [1][2][3] propose a characterization of the power consumption of a given microprocessor based on the measurement of the average current absorbed by the core during the execution of long sequences of the same machine instruction. Power figures are then associated with assembly instructions, leading to an abstraction from architectural details of the microprocessor. Such approaches, though, still suffer a lack of generality since a new set of measurements is needed when a different processor is analyzed. A more general approach, proposed in [4], abstracts from the architectural level by determining a set of *functionalities* and by decomposing the computational activity of each instruction in terms of these functionalities. According to this model the energy absorbed by each instruction is computed as the weighted sum of the contributions of the functionalities. A tuning phase, based on a limited set of experimental data, allows associating to each functionality an average current absorption per clock cycle. It is worth noting that the overall energy consumption is strongly dependent on the number of cycles taken for the execution of assembly instructions. In [4] the timing is assumed to coincide with the nominal value reported in the processor data–sheets. This timing data, being purely static, are a sound starting point for a general energy model but disregard the delays introduced by the interlocks arising from a pipelined execution of the code. Limitations deriving from a static analysis have been studied in [5] and a solution, based on the models proposed in [1][2], has been presented. The extended approach, though, does not address the problem of the lack of generality. The aim of this work is to cope with the above limitations, providing a model capable of describing timing overheads due to inter–instruction effects in a formal and general way. The advantages of a *static* model with respect to a dynamic, simulation–based, approach are evident. The proposed strategy is based on a dynamic characterization—to be performed once and for all—of a given instruction set aimed at producing statically usable figures. To this purpose a sound and formally consistent statistical model has been developed and verified both theoretically and by comparison against actual timing measures. The methodology and related models are being implemented in a co–design flow that will enable accurate and efficient software power estimation [6]. This paper is organized as follows: Section 2 suggests a possible strategy to extend the framework described in [4] and details the mathematical model along with its statistical properties; the tuning and validation methodologies adopted are described in Section 3, where the experimental results obtained are reported. Some conclusions are summarized in Section 4.

2 Proposed Model

This section introduces the extension of the previously developed model [4], to cover also inter–instruction effects. For the purpose of producing a widely applicable *static* estimation of the timing overheads related to the interaction between instructions, a taxonomy of a generic instruction set has been proposed, including all possible hazards in an architecture–independent manner. This taxonomy is crucial to reduce the model complexity in order to make the statistical analysis feasible. Starting from such a taxonomy, a mathematical model for the estimation of the timing overhead caused by inter–instruction effects has been developed.

2.1 Problem Definition

As mentioned above, the model proposed in [4] provides a *static* estimation of the energy consumption of single instructions. According to [4] the energy dissipation e_s of an instruction s is:

$$e_s = \sum_{j=0}^{5} e_{s,j} = \left[\sum_{j=0}^{5} if_j \cdot a_{s,j} \right] \cdot V_{dd} \cdot \tau \qquad (1)$$

where if_j is the average current associated with the j-th functionality, V_{dd} is the power supply voltage, τ is the clock period and $a_{s,j}$ is a coefficient expressing the execution time spent by instruction s in the j-th functionality. The coefficients $a_{s,j}$ satisfy the relation:

$$\sum_{j=0}^{5} a_{s,j} = \text{CPI}_{s,\text{nominal}} \qquad (2)$$

stating that the time—expressed in clock cycles—spent by instruction s in all the functionalities corresponds to its average CPI[1]. It is worth noting that the average CPI used in this model is the nominal value and thus neglects all inter–instruction effects. The present work extends this static model and, in particular, focuses on inter–instruction effects related to pipelined execution. In pipelined processors instructions are executed with partial time overlap in order to minimize the average CPI. However, this execution scheme leads to some *hazard* conditions that have to be suitably managed in order to maintain the semantics of the original program. In some cases, it is necessary to stall the pipeline, consequently increasing the nominal CPI. The introduced temporal overhead results in an increase of the energy consumption that cannot be ignored. According to these observations, equation (1) can be extended by explicitly adding the temporal overhead $oh_{s,j}$, yielding:

$$e_s = \left[\sum_{j=0}^{5} if_j \cdot (a_{s,j} + oh_{s,j}) \right] \cdot V_{dd} \cdot \tau \qquad (3)$$

The actual execution time of instruction s is thus:

$$\text{CPI}_{s,\text{actual}} = \sum_{j=0}^{5} (a_{s,j} + oh_{s,j}) \qquad (4)$$

The aim of the model presented in the following paragraphs is to derive an accurate estimate for $\text{CPI}_{s,\text{actual}}$, while leaving to a subsequent analysis the task of splitting the total overhead on the different functionalities. The porposed model thus concentrates on the problem of timing estimation at an abstraction level that ignores finer–grained contributions. Since inter–instruction effects such as pipeline interlocks and cache misses are intrinsically *dynamic events*, a purely static analysis would lead to an oversimplification of the problem. Nevertheless, a static analysis is still viable if a characterization of the dynamic effects is available. Such information can be extracted once for each microprocessor considered and stored in a library. The procedure to derive the statistical dynamic figures is described in section 2.4. The proposed approach is thus based on a dynamic analysis of the whole instruction set aimed at a statistical characterization whose results, i.e. the $\text{CPI}_{s,\text{actual}}$, can then be statically used for the estimation process. An hazard is related to a particular sequence of instructions that flows into the processor pipeline. Based on this observation, the *execution trace* of a given code portion, i.e. the ordered list of all the instructions actually executed by the processor, turns out to be a simple means to dynamically analyze a given architecture with respect to interlock generation. The key idea behind the proposed model is thus to neglect the details of the specific architecture implementation and rather assume the dynamic behavior as its abstract representation. Paragraphs 2.2 to 2.4 formally describe the mathematical form of such an abstract view. The consistency of the model is theoretically proved and its accuracy is demonstrated against actual measurements in section 3.

2.2 Instruction Set Taxonomy

In order to maintain the approach as general as possible, no specific architecture or set of architectures should be considered. A simple solution consists in providing some general classes to be associated with architecture–specific instructions. Taxonomy classes are defined according to the type of hazard that an instruction may incur on, so that each instruction of a given instruction set belongs to the class that best represents its dynamic behavior. Three hazard types may arise [7]: *structural*, *data* and *control*, conventionally named *S-type*, *D-type* and *C-type* respectively in the rest of the paper. Their effect is a stall in the pipeline whose duration depends on the specific instructions causing the hazard. The idea of taxonomy is formally introduced by definition 1.

Definition 1 *Given an instruction set I, a* ***hazard–based partition****:*

$$\mathcal{I}_H = \{ I_{H,j} \,|\, j \in \{0,1\} \} \subset 2^I$$

distinguishes instructions that may cause H-type with $H \in \{S, D, C\}$ hazards and those that may not. The classes $I_{H,0}$ and $I_{H,0}$ are defined as:

$$\begin{aligned} I_{H,0} &= \{ s \in I \,|\, s \textit{may cause an H-type hazard} \} \\ I_{H,1} &= \{ s \in I \,|\, s \textit{may not cause an H-type hazard} \} \end{aligned}$$

[1] Clock–cycle Per Instruction.

and constitute a partition by construction.

This definition explains the concept that a hazard depends on an ordered pair of instructions: the former instruction is referred to as the *cause* of the hazard, while the latter is *stalled* in order to resolve the interlock situation. In this way, three different partitions on I can be identified: $\mathcal{I}_S$, $\mathcal{I}_D$ and $\mathcal{I}_C$; starting from these partitions, a taxonomy of an instruction set can be defined as:

Definition 2 *A **taxonomy** $\mathcal{C} \subset 2^I$ on the instruction set I is:*

$$\mathcal{C} = \big\{C_{i,j,k} \subseteq I \,|\, C_{i,j,k} = I_{S,i} \cap I_{D,j} \cap I_{C,k},\ i,j,k \in \{0,1\}\big\}$$

meaning that each $C_{i,j,k}$ is the intersection among 3 sets, each being chosen from a different partition. Thus $\mathcal{C}$ contains all possible combinations of the sets in $\mathcal{I}_S$, $\mathcal{I}_D$ and $\mathcal{I}_C$.

By definition, a partition covers the entire instruction set I, and its subsets are disjoint: in this sense, $\mathcal{C}$ is a partition of I. For the sake of simplicity, the taxonomy classes are renamed so that they have a single index. The conversion is made by assigning an integer index, corresponding to the binary conversion of the subscripts ijk, e.g. class $C_{0,0,0}$ is referred to as c_0, $C_{0,1,0}$ as c_2 and so on. According to the shorthand notation just introduced, the taxonomy can be rewritten as:

$$\mathcal{C} = \big\{c_h \subseteq I \,|\, h \in [0,7]\big\} \tag{5}$$

2.3 Model Definition

The model definition is intended to bring out a statistical characterization of interlock occurrence: to this purpose, it is necessary to estimate the probability of finding pairs of instructions and the probability with which such pairs generate interlocks. In this way, the underlying architecture can be neglected, and class–associated measures can be conveniently used. An *execution trace* Γ can be seen as an ordered set of instructions resulting from the execution of some real programs. Let a trace Γ be:

$$\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_N\}, \quad \gamma_k \in I, \quad N > 0 \tag{6}$$

where N indicates the execution trace size. The estimates of the probability of finding a taxonomy class pair in Γ are obtained by means of two operators, introduced by the following definitions.

Definition 3 *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 $\hat{w}$:*

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

The execution trace size N can always be assumed much larger than $\hat{w}$. In fact, practical distances are usually not greater than the pipeline depth, since farther instruction are almost independent in terms of interlock behavior. Considering that N is in the order of 10^6-10^7 for medium–sized programs, and pipeline depths for embedded processors are smaller than 10, the assumption $N \gg \hat{w}$ holds and does not cause loss of generality.

Definition 4 *The **membership function** of an instruction $\gamma_k \in \Gamma$ to $c_i \in \mathcal{C}$ is defined as:*

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

The membership function has the following property:

$$\sum_{i=0}^{7} \langle k, i \rangle = 1 \tag{7}$$

which is easily proved considering that an instruction must belong exactly to one class, since $\mathcal{C}$ is a partition. Previous definitions are combined to describe the following *event*:

$$c_i \overset{\hat{w}}{\dashv} c_j \quad \Leftrightarrow \quad \exists(k_1, k_2) : \begin{cases} \gamma_{k_1} \overset{\hat{w}}{\dashv} \gamma_{k_2} \\ \langle k_1, i \rangle = 1 \\ \langle k_2, j \rangle = 1 \end{cases} \tag{8}$$

meaning that there exists in Γ two instructions $\gamma_{k_1} \in c_i$, $\gamma_{k_2} \in c_j$, having distance $\hat{w}$. Events described above have to be characterized in a statistical manner. Given the nature of the system to be modeled a convenient solution is to adopt the frequency definition of probability, i.e. the ratio of the number of observation of a particular event and the total number of observations [8].

Definition 5 *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.

As a consequence, $\sum_{i=0}^{7} P(c_i) = 1$, due to the frequency definition of probability. Definition 5 can be extended to consider class pairs:

Definition 6 *The probability of finding class c_i and class c_j at distance $\hat{w}$ in the execution trace is:*

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

where N is suitably large[2].

Definitions 5 and 6 are tightly related, since:

$$\sum_{i=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j) = P(c_j) \tag{9}$$

$$\sum_{j=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j) = P(c_i) \tag{10}$$

[2] Due to the assumption that $N \gg \hat{w}$, the upper limit of the summation (the total number of class pairs) can also be approximated as $N - \hat{w} \approx N$.

These equations show that the sum of the probabilities of finding in the execution trace a pair of instructions starting (ending) with c_i (c_j) is equal to the probability of finding an instruction belonging to class i (j).

2.4 Interlock Model

Having obtained a characterization of the frequencies of class pairs in the execution trace, the next step is considering the interlocks they generate. Since different pairs of instructions represented by the same couple of classes, in general, have different interlock behaviors and latencies, the delay they possibly introduce in the execution must be accordingly modeled. To this purpose, it is useful introducing the following, non–analytic function:

Definition 7 *The* ***instruction pair delay*** *is the delay introduced by the execution of an instruction pair $\gamma_k, \gamma_{k+\hat{w}}$ at a distance $\hat{w}$ is given by the function $t(\gamma_k, \gamma_{k+\hat{w}}, \hat{w})$. The situation $t(\gamma_k, \gamma_{k+\hat{w}}, \hat{w}) = 0$ means that no interlock occurs.*

Translating this kind of information from specific instruction pairs in the execution trace to the statistical vision of classes leads to the introduction of *random variables*. Given a taxonomy class pair at a distance $\hat{w}$, the delay introduced in the execution is accounted for by a random variable, in order to consider all possible interlock delays of all instruction pair that the class pair represents.

Definition 8 *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 stochastic 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:

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

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). Interlocks associated with a single class interacting with any other class preceding it at a given distance $\hat{w}$ can also be considered:

Definition 9 *The* ***class delay*** *is the delay associated with the execution of a class c_i 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 the density function:*

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

Definitions 8 and 9 are bound by the following theorem:

Theorem 1 *The class delay density function $D_{j,\hat{w}}$ equals the sum of the pair delay density functions $D_{i,j,\hat{w}}$, weighted by the frequency of the corresponding pair, i.e.:*

$$f_{D_{j,\hat{w}}}(d) = \frac{\sum_{i=0}^{7} f_{D_{i,j,\hat{w}}}(d) P(c_i \overset{\hat{w}}{\dashv} c_j)}{\sum_{i=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j)} \tag{11}$$

Proof 1 *Applying definition 6 to definition 8 yields:*

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

Applying this result and relation (9) to the second term of (11) and simplifying $P(c_i \overset{\hat{w}}{\dashv} c_j)$:

$$\frac{1}{P(c_j)} \cdot \sum_{i=0}^{7} \left(\frac{1}{N} \cdot \sum_{k=1}^{N} \delta_{t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})=d}\langle k,i\rangle\langle k+\hat{w},j\rangle \right)$$

By swapping the order of summation:

$$\frac{1}{P(c_j)} \cdot \frac{1}{N} \cdot \sum_{k=1}^{N} \left(\delta_{t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})=d}\langle k+\hat{w},j\rangle \sum_{j=0}^{7}\langle k,i\rangle \right)$$

Applying the property of the membership function described in equation (7) and definition 5:

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

and the theorem is proved.

The importance of this theorem is twofold: on one hand, it proves the correctness of our model as already stated by relations 9 and 10; on the other hand, it is a means to estimate the class delays starting from pair delays, thus allowing a static estimation of dynamic inter–instruction effects. The statistic characterization of $D_{i,j,\hat{w}}$ and $D_{j,\hat{w}}$ is well represented by their first and second order moments: the expectation value and variance. The expected value $\mathrm{E}\left[D_{i,j,\hat{w}}\right] = \mu_{D_{i,j,\hat{w}}}$ of the discrete stochastic variable $D_{i,j,\hat{w}}$ is:

$$\mu_{D_{i,j,\hat{w}}} = \sum_{d=0}^{+\infty} d \cdot f_{D_{i,j,\hat{w}}}(d) \tag{12}$$

which can be transformed using definition 8 in:

$$\mu_{D_{i,j,\hat{w}}} = \frac{\sum_{k=1}^{N} t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})\langle k,i\rangle\langle k+\hat{w},j\rangle}{\sum_{k=1}^{N}\langle k,i\rangle\langle k+\hat{w},j\rangle} \tag{13}$$

This result comes from a property of the Kronecker's δ:

$$\sum_{d=0}^{+\infty} d \cdot \delta_{t(\gamma_k,\gamma_{k+\hat{w}},\hat{w})=d_0} = d_0$$

The obtained expectation value $\mu_{D_{i,j,\hat{w}}}$ corresponds to a weighted average of the delays introduced by the pairs of instructions represented by the class pair (c_i, c_j). The weights are the relative frequencies of the class pairs. Proceeding in a similar way, the variance VAR $[D_{i,j,\hat{w}}] = \sigma_{D^2_{i,j,\hat{w}}}$ of $D_{i,j,\hat{w}}$ can also be derived:

$$\sigma_{D^2_{i,j,\hat{w}}} = \frac{\sum_{k=1}^{N} t^2(\gamma_k, \gamma_{k+\hat{w}}) \langle k, i \rangle \langle k+\hat{w}, j \rangle}{\sum_{k=1}^{N} \langle k, i \rangle \langle k+\hat{w}, j \rangle} - \mu^2_{D_{i,j,\hat{w}}}$$

Similar definitions can be given for $D_{j,\hat{w}}$. The following theorem, whose demonstration is omitted, holds:

Theorem 2 *The moments of class pair delays and class delays are bound by the following relations:*

$$\begin{aligned} \mu_{D_{j,\hat{w}}} &= \frac{\sum_{i=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j) \cdot \mu_{D_{i,j,\hat{w}}}}{\sum_{i=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j)} \\ \sigma_{D^2_{j,\hat{w}}} &= \frac{\sum_{i=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j) \cdot \sigma_{D^2_{i,j,\hat{w}}}}{\sum_{j=0}^{7} P(c_i \overset{\hat{w}}{\dashv} c_j)} - \mu^2_{D_{j,\hat{w}}} \end{aligned} \tag{14}$$

This theorem expresses the relations between the moments of class pair delays and those of class delays.

3 Experimental Results

The mathematical model described in section 2 allows the estimation of the delay introduced by each instruction class due to inter–instruction effects. The model has to be tuned on real data and then validated in order to prove its effectiveness: this section describes the tuning and validation methodologies. The model tuning is based on the choice of a representative set of benchmarks on which to compute the distributions of the random variables used to build the static representation of dynamic effects. A number of execution traces have been generated, leading to an overall number of assembly code lines in the order of 10^7-10^8. It is worth noting that execution traces have been generated from real programs running on real data: this guarantees that they represent the effective dynamic behavior of a processor under real–life stimuli. However, an execution trace only describes the flow of instructions into the pipeline: any other dynamic information must be explicitly extracted from the trace. For the sake of generality, this task is performed in an architecture–independent manner. In particular, since hazards are specific *dynamic events*, a number of general *event models* have been developed, which altogether represent the dynamic behavior of a given architecture. A dynamic event is thus identified as a hazard when it matches one of the models characterizing the architecture. A tool has been implemented for the purpose of tuning the proposed model. Such tool fetches in a description of a given architecture, i.e. a library containing the instruction set taxonomy—obtained as described in section 2.2—the dynamic behavior representation—in terms of hazard models—and some other architecture–dependent data such as the pipeline depth and the branch prediction scheme. This data is used to build an internal representation of the processor under analysis. As a second step, one or more execution traces are parsed to derive the delay introduced by each taxonomy class pair. This phase leads to an estimate of the frequency of class pairs and a set of density functions characterizing the random variables used to statistically describe the interlock delays. Using the relation proved in theorem 1, these random variables are combined to obtain delay models associated to single classes which eventually allow the static analysis process. Once the statistical models of all classes have obtained, a validation phase is necessary. Validation is performed on the timing level by comparing the estimated execution times of a number of benchmarks with the corresponding actual timings, derived by reading—via assembly programs developed on purpose—the value of some hardware counters available on the target architecture. The validation consists of four main steps:

1. estimation of interlock–free timings;
2. estimation of the temporal overhead introduced by inter–instruction effects;
3. measurements of actual timings;
4. computation of the relative error between the estimated and actual timings.

This procedure has been applied on the five benchmarks described below[3], fed with five different data set:

`crc16` A 16-bit CRC on random 256-character strings;

`qsort` Quicksort of a vector of 100 integers;

`rle` RL encoding of random 128-characters strings;

`genprime` generates a prime number 3 digits long;

`md5` Message digest of 500-characters random strings.

These benchmarks cover a wide range of operations and are all processor–bound[4], and are thus suited for validation. The actual timings of these programs have been obtained on a Intel 486DX4 with a clock frequency of 100 MHz, running a minimal version of the RedHat7 Linux distribution. The results obtained are shown in table 3 which reports the average error and standard deviation for each benchmark for interlock–free and interlock–aware estimates with respect to the actual timings. The last row shows the overall average values. The interlock–free analysis results in a large underestimation of the execution timings, errors that are generally

[3] These benchmarks have not been used for model tuning.

[4] The size of data processed is such that the number of cache misses is reduced at a minimum.

	Relative error (%)	
Test case	interlock–free	interlock–aware
`crc16`	-22.1 ± 2.30	$+0.3 \pm 3.30$
`qsort`	-22.1 ± 0.51	-3.7 ± 0.71
`rle`	-17.9 ± 0.04	$+2.7 \pm 0.05$
`genprime`	-32.1 ± 2.13	-9.4 ± 2.70
`md5`	-26.5 ± 0.94	$+2.5 \pm 1.21$
Overall	-24.1 ± 5.15	-1.5 ± 5.06

Table 1: Interlock–free and interlock–aware errors

eliminated adding the inter–instruction overhead estimated by the proposed model. Even where the error remains high, the improvement is noticeable. It is worth noting that the proposed model still lacks of a comprehensive analysis of all the inter–instruction effect related to memory access, such as those due to cache misses. For the purpose of analyzing the cache miss impact, the `md5` benchmark has been run on larger strings, which cannot reside in the limited cache of the Intel 486DX4 processor. The resulting average errors, reported against the string length in table 2, confirm that the cause of the underestimates produced by the proposed model for large–sized strings is to be found in the missing contribution related to memory–dependent inter–instruction effects. These values, compared with those reported in table 3 confirm that the proposed model is much more accurate with respect to the interlock–free analysis, but it still has to be refined in order to adequately consider cache misses and other memory effects.

	Relative error (%)	
String size	interlock–free	interlock–aware
500	-26.5 ± 0.9	$+2.5 \pm 1.2$
100,000	-31.1 ± 2.5	-3.8 ± 3.5
300,000	-37.7 ± 1.0	-13.0 ± 1.4

Table 2: Cache miss impact on estimation accuracy for `md5`

4 Conclusions

The present paper proposed a general, processor–independet model for the estimation of the temporal overhead related to pipelined execution of assembly code. The model abstracts from the architectural details of specific processors and concentrates on the functional behavior of assembly instructions with respect to the interlocks they may cause. The validation presented in Section 3 concentrates on the effects deriving from data–path resource constraints. This is achieved by building a suitable set of testbenches that minimize the impact of the memory hierarchy on the actual execution time. In practice this means that the program code and the data it manipulates should be small enough to be contained in the cache memory of the considered processor. The accuracy provided by the model is more than satisfactory, leading to average estimation errors below 5%. Furthermore, the estimation process proved to be fast enough—tens of seconds to few minutes—to enable effective design space exploration. It is worth noting that the proposed approach can be used in conjunction with a separate model oriented at the characterization of the memory–related dynamic effects. Memory–dependent effects are currently under investigation and the preliminary results obtained suggest that their contribution to the total overhead can be accounted for independently. As a different alternative, an extension of the presented model can also be envisioned. Such an extension should incorporate memory–related effects into the mathematical framework already developed. This last approach is also currently under investigation.

References

[1] 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.

[2] 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.

[3] J. Russell and M.F. 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.

[4] 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.

[5] R. Sridhar and S. Kris. 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.

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

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

[8] A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, New York, NY, 1984.