Modeling Assembly Instruction Timing in Superscalar 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

This paper proposes an original model of the execution time of assembly instruction in superscalar architectures. The approach is based on rigorous statistical foundations and provides a methodology and a toolset to perform data analysis and model tuning. The methodology also provides a framework for building new trace simulators for generic architectures. The results obtained show a significant accuracy improvement over previous works.

1. INTRODUCTION

2. MATHEMATICAL MODEL

The basic assumption made in [**?**] and maintained in [**?**] concerns the *a priori* knowledge of instruction CPIs. In this case, the interlock–free timing of a program can be easily obtained by simply summing the CPIs of all the executed instructions. The interlock–aware timing can also be obtained including the statistical term representing the stall overhead associated to every single instruction. Stall overheads are obtained by suitably averaging the contribution deriving from the *dynamic* interaction between instructions falling within a trace window of fixed size. In superscalar architectures, the parallel execution of assembly instructions strongly influences both the actual CPI of an instruction and the number and type of possible interlocks. For example, when the three instructions γ_1, γ_2 and γ_3 are executed in an ideal pipeline the resulting CPI of each one is 1.0; In a superscalar architecture with three ideal pipelines in parallel, the resulting CPI is $\frac{1}{3}$. However, real processors are significantly different than ideal architectures and only a portion of the theoretical parallelism can be exploited. To this purpose, a *parallelism coefficient* has been introduced and defined according to a statistical analysis of the execution of actual programs on a given architecture. Indicating $t_n(\gamma_s)$ and $t_{oh}(\gamma_s)$ the number of clock cycles for nominal execution and the number of stalls of instruction γ_s and with p_s th parallelism coefficient, the estimated CPI can be expressed as:

$$CPI_{s,est} = p_s \cdot (t_n(\gamma_s) + t_{oh}(\gamma_s)) \quad (1)$$

It is interesting to note that the energy absorbed by each instruction is not influenced by parallel execution while average power w_s increases, as shown by the following relation:

$$w_s = \frac{e_s}{CPI_{s,est} \cdot \tau} \quad (2)$$

where τ is the clock period and e_s the absorbed energy. The following paragraphs describe how the parallelism coefficient p_s is defined and how it can be derived from a statistical analysis of program execution traces.

2.1 Instruction Set Taxonomy

In order to maintain the approach as general as possible, no specific architecture or set of architectures should be considered. Each architecture is, in fact, characterized by strongly different execution capabilities, leading to significant differences in the actual parallelism. A possible solution to this issue is to define a set of *general* classes to which instructions of a *specific* architecture can be assigned. The classification must account for the dynamic interaction between instructions with respect to both interlock effects and parallel execution.

DEFINITION 1. *The equivalence relation $\mathcal{R} \subseteq I \times I$:*

$s_i \; \mathcal{R} \; s_j \iff s_i$ *and* s_j *have similar dynamic behavior*;

defines a taxonomy $\mathcal{C} \in 2^I$ on the instruction set I as the partition induced by $\mathcal{R}$ on the instruction set I. The cardinality $|\mathcal{C}|$ of the taxonomy depends on the relation $\mathcal{R}$. The taxonomy $\mathcal{C}$ is thus formed by the classes c_i with $i \in [1; |\mathcal{C}|]$.

Definition 1 gives a way to obtain the taxonomy based on the equivalence relation $\mathcal{R}$. Nevertheless, $\mathcal{R}$ is still to be properly defined for each instruction set and architecture. Three approaches are possible:

Architectural The relation $\mathcal{R}$ is defined *a priori* and is based on the knowledge of both the instruction set and the architectural details.

Numerical The relation $\mathcal{R}$ is defined *a posteriori* based on the data extracted from simulation of the dynamic behavior of instrucitons.

Full The relation $\mathcal{R}$ is always false. In this case each instruction belongs to a different case, i.e. no classification is performed.

Section **??** shows and discusses the results obtained folloing these classification paradigms.

2.2 Model Definition

The model expressed by equation (1) depends on two parameters: the interlock overhead $n_{oh,s}$ and the parallelism coefficient p_s. The present work is based on a previous model [**?**] and extends it in order to fit superscalar microprocessors. The original model defined an *execution trace* Γ as an ordered list of instructions resulting from actual execution of a program. Let a trace Γ be:

$$\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_N\}, \quad \gamma_k \in I, \quad N > 0 \tag{3}$$

where N indicates the execution trace size. Instructions γ_k are then classified based on the relation $\mathcal{R}$ and the *membership function* can be defined accordingly as:

$$\langle k, i\rangle = \begin{cases} 1 & \text{if } \gamma_k \in c_i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

A timing overhead $t(\gamma_k)$, introduced by dynamic effects during the execution, can be associated to the instruction that has been stalled in order to resolve an hazard situation. Based on the classification imposed by $\mathcal{R}$, such overheads have to be collected and associated to instruction classes, rather than single instructions. This leads to the definition of a stochastic variable D_i whose density is given by the relation:

$$f_{D_i}(d) = \frac{\sum_{k=1}^{N} \delta_{t(\gamma_k)=d}\langle k, i\rangle}{\sum_{k=1}^{N}\langle k, i\rangle} \tag{5}$$

where N is suitably large[1] and δ is the Kronecker symbol. The overhead $n_{oh,s}$ can be obtained as the expectation value of D_i for the index i such that $I_s \in c_i$.

2.3 Parallel Execution Model

The parallelism factor can be estimated experimentally starting from the execution trace Γ and observing the instructions that are executed in parallel. Similarly to the computation of overheads, the parallelism coefficients are referred to instruction classes. According to such approach, the more instructions belonging to a given class c_i are executed in parallel, the lower the corresponding parallelism coefficient p_i is, and thus the lower the actual CPI for instructions of that class is also. To determine p_s it is necessary to know when a given instruction γ_k starts and ends executing. The notion of time is here intended as the number of clock cycles since the beginning of the execution. This is clarified by the following definition.

DEFINITION 2. *Let $t_{in}(\gamma_k)$ the starting time of a generic instruction $\gamma_k \in \Gamma$ and $t_{out}(\gamma_k)$ its ending time. The **time range membership function** of instruction γ_k with respect to class $c_i \in \mathcal{C}$ at time t is defined as:*

$$\lceil t, k, i\rfloor = \begin{cases} \langle k, i\rangle & \text{if } t_{in}(\gamma_k) \le t \le t_{out}(\gamma_k) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

[1]For a good approximation, $N \ge 10^6$.

where the values $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ are properties of the instruction γ_k with respect to a given trace Γ.

It is worth noting that the time range between $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ is given not only by the instruction latency but it also includes the inter–instruction overhead resulting from stalls or cache misses. When an instruction is stalled, in fact, it still occupies some resources. The time range membership function allows to know, at each clock cycle, which instructions are being executed. Starting from the time range membership function it is possible to aggregate values for each class.

DEFINITION 3. *The **class load function** represents the number of instructions belonging to class c_i being executed at time t. It is defined as*

$$\lceil t, i\rfloor = \sum_{k=1}^{N} \lceil t, k, i\rfloor \tag{7}$$

The class load function can be used to compute an instantaneous parallelism coefficient, defined as follows.

DEFINITION 4. *The **instantaneous parallelism coefficient** is defined as:*

$$p(t) = \frac{1}{\sum_{i=1}^{|\mathcal{C}|} \lceil t, i\rfloor} \tag{8}$$

where the sum is extended to all classes in the taxonomy.

It can be easily proved that $p(t) \in [1/N; 1]$ with N being the maximum number of instruction that the specific architecture is capable of handling in the same clock cycle. As an example consider a simple DLX–like 5-stage pipeline architecture: in this case $N = 5$ since when the pipeline is full all its stages are executing an instruction at every clock cycle. The instantaneous parallelism coefficient $p(t)$ must then be aggregated according to the selected taxonomy in order to obtain a per–class vision of the amount of parallelism that the architecture under analysis can actually exploit. The following definition introduces such concept.

DEFINITION 5. *The **class parallelism coefficient** is the time reduction factor associated to an instruction belonging to class c_i when executed in parallel with other instructions. It is modeled by the stochastic variable P_i, which is characterized by its density function:*

$$f_{P_i}(\hat{p}) = \frac{\sum_t \delta_{p(t)=\hat{p}} \lceil t, i\rfloor}{\sum_t \lceil t, i\rfloor} \tag{9}$$

where the summations are extended over all clock cycles needed for the execution of the trace Γ.

The parallelism coefficient p_s can be then assumed to be the expectation value of the stochastic variable P_i, that is:

$$p_s = E[P_i] = \sum_{p=0}^{1} p \cdot f_{P_i}(p) \qquad I_s \in c_i; \qquad p \in \mathbb{Q} \tag{10}$$

It must be noted that the variable $p \in \mathbb{Q}$ since it is computed as the ratio of two integer numbers and $0 \leq p \leq 1$ by definition.

3. TRACE SIMULATION

To calculate both overheads and parallelism coefficients a cycle–accurate simulation of the *timing behavior* of a given architecture is necessary. In particular, for each instruction γ_k actually executed, the following times must be determined:

- the starting time $t_{in}(\gamma_k)$,
- the ending time $t_{out}(\gamma_k)$,
- the nominal execution time $t_n(\gamma_k)$.

The execution time overhead $t_{oh}(\gamma_k)$ can be easily expressed in terms of these times, as the following equation shows:

$$t_{oh}(\gamma_k) = t_{out}(\gamma_k) - t_{in}(\gamma_k) - t_n(\gamma_k) \tag{11}$$

It is worth noting that the nominal execution time can be found in the processor datasheets while $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ can only be determined by performing a pseudo–simulation, i.e. a simulation that only accounts for the timing properties of the instruction while neglecting their functionality. This is possible thanks to the fact that the simulated instructions are taken from an execution trace, which is dynamic, rather than from the assembly code generated by compilation, which is, on the contrary, static. These considerations and the goal of being as independent as possible from a specific architecture have led to the development of a customizable pseudo–simulation framework, whose structure is shown in figure **??**.

Figure 1: Trace simulation flow

3.1 Microcompilation

The execution trace is first fed to the `atomic` microcompiler that translates each assembly instruction in a sequence of microinstructions. This step has the goal of decomposing the complex activity of a generic instruction into a sequence of basic activities involving only few basic operations. Table **??** exemplifies the microcompilation process by means of three examples. For the sake of clarity, let us briefly describe the meaning of the microinstructions in the first example. The two `read` are used to perform read requests to the integer register file, specifically of the the registers `%r0` and `%r1`. The microinstruction `require` indicates that the execution of the multiplication is performed by the integer ALU and requires 16 clock cycles[2]. Finally the `write` microinstruction indicates that register `%r2` is written. It is important noting that whenever a register needs to be written it must

[2]This is the nominal execution time $t_n(\gamma_k)$.

SPARCv8 Assembly	Microcode		
`mul %r0, %r1, %r2`	`read`	`0`	`regfile-int`
	`read`	`1`	`regfile-int`
	`require`	`16`	`alu-int`
	`write`	`2`	`regfile-int`
`ld [%r0,%r1], %r2`	`read`	`0`	`regfile-int`
	`read`	`1`	`regfile-int`
	`load`	`1`	`address`
	`write`	`2`	`regfile-int`
`fadd %f0,%f1, %f2`	`read`	`0`	`regfile-fp`
	`read`	`1`	`regfile-fp`
	`require`	`5`	`alu-fp`
	`write`	`2`	`regfile-fp`

Table 1: Microcode examples

be *locked*, for example at decode–time, to avoid subsequent instructions to use its content before the up–to–date value is actually present. The lock is then removed at a later stage, for example at retire–time. Other microinstructions work in a similar manner. As a further remark, it interesting to analyze the meaning of the `load` microinstruction. It is used to require a memory access and does not explicitly specify the number of clock cycles necessary to complete the operation. This intentional generality allows to model different memory hierarchies and thus permits to include in the interlock model all memory– and cache–related effects.

3.2 Behavioral simulation

The microcompiled trace is then fed to the behavioral simulator `tribes`. The simulator is composed of two main portions: a general purpose *simulation engine* and a set of custom, user–defined *functional units* and *resources* implementing the behavior of a specific architecture. According to this scheme, an architecture is composed of a set of functional units connected by *instruction buffers* and resources. The functional units communicate with the resources by means of *messages*. Figure **??** describes this architecture. Functional

Figure 2: Behavioral simulator structure

units and instruction buffers are always synchronized by the clock signal, while resources may or may not have an explicit notion of time. According to this scheme, instruction

traverse the different functional units which basically behave as dispatchers (determining the path) and schedulers (determining the timing). The simulation engine provides:

- Base classes from which specific functional units and resources can be built exploiting inheritance.
- Primitives for instantiating and connecting functionalities and resources by means of buffers and messages.
- The notion of a global discrete time managed by the simulation kernel.

Such framework is flexible enough to model a very large set of architectures and, at the same time, sufficiently standard to provide a wide range of capabilities that each specific simulator can exploit without modification.

3.3 Data analysis

The output of the behavioral simulator is an annotated trace, that is a list of annotated instructions. Each line of the output has the structure:

$$< class > \quad < t_{oh} > \quad < t_{in} > \quad < t_{out} >$$

and provides all data necessary for model tuning, which is performed by the stand–alone tool `tune`. This tool elaborates the input data and constructs, for each class i, the average parallelism coefficient p_i and the density function f_{D_i} of the timing overheads. It is worth noting that the tuning process requires very large trace files and for this reason the tool chain can be conveniently executed in a pipeline.

4. MODEL TUNING

The methodology and the simulation framework have been set up to model the microSPARC II Embedded Processor architecture. Figure **??** shows the structure of the architecture in terms of functional units, buffers and resources and highlights the possible paths of instruction through the various units. A detailed description of this architecture can be found in **??**. The simulator has been stimulated

benchmarks per la traccia (gcal, gzip, cpp2html, bc)(FP????)

Figure 3: microSPARC II simulator structure

5. EXPERIMENTAL RESULTS

6. CONCLUSIONS