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 Research Centre, Via R. Fucini, 2 – 20133 Milano, Italy
betrami@cefriel.it, {brandole,fornacia,salice,sciuto,trianni}@elet.polimi.it

ABSTRACT

This paper proposes an original model of the execution time of assembly instructions in superscalar architectures. The approach is based on a rigorous mathematical model 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 good accuracy paired with a satisfactory computational efficiency.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance and Reliability—*Performance Analysis and Design Aids*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*; I.6.5 [**Computing Methodologies**]: Simulation and Modeling—*Model Development*

General Terms

Languages, Performance

Keywords

Assembly-level analysis, Performance estimation, superscalar architectures

1. INTRODUCTION

The relevance of the software portion of an embedded system is steadily increasing and its impact is so large that many design flows are being studied in order to include a specific development section dedicated to software. While, within such flows, hardware synthesis and estimation and software compilation techniques have been studied for many years and are well established, few methodologies, and even fewer tools, are available for an early analysis of the performance of software programs. The problem of execution time estimation has been tackled, in literature, according to two orthogonal approaches. On one hand, accurate methods and tools have been developed performing a *dynamic* analysis strongly relying on proprietary instruction set simulators [5, 11, 10]. These approaches show good accuracy and satisfactory efficiency but severely lack generality and portability over different target platforms. On the other hand, *static* approaches [12, 9, 4] are often based on path analysis and typically provide worst-case estimates, which may significantly differ from typical execution times. Moreover, most of the techniques found in literature provide an estimate of the instruction execution count rather than the expected number of clock cycles. When considering superscalar architecture, the instruction count, or even the sum of the nominal execution times of all instructions, strongly differs from the actual timing due to parallelism that such architectures can exploit. This paper specifically addresses the problem of execution time estimation of the instructions on superscalar architectures. The proposed approach founds on a previous work [2] and extends it to explicitly account for parallel execution of instructions. This implies considering effects such as pipeline interlocks and memory-related effects. To tune the model for a specific architecture a *behavioral simulation engine* and a related SDK have been developed. The simulator has been specifically designed to model the widest possible range of architectures accounting for instruction-level parallelism features—such as instruction shelving, alignment schemas, out-of-order execution, branch prediction and register renaming—and memory hierarchies [14]. The results obtained after tuning are a *static* characterization of the considered instruction set from a timing point of view and can be used to estimate the actual execution time of a program without resorting to proprietary ISS. The paper is organized as follows. Section 2 introduces the foundations of the model and justifies the need for the behavioral simulation described in Section 3. Section 4 collects the results obtained both with the simulation framework (paragraph 4.1) and the estimation flow (paragraph 4.2). Finally, in Section 5 some conclusions are drawn and the ongoing and future work are briefly outlined.

2. MATHEMATICAL MODEL

When dealing with simple pipelined architectures the nominal execution time of a program can be easily calculated by summing the CPIs of all the instructions actually executed. In fact, the basic assumption made in [3, 2] concerns the *a priori* knowledge of instruction CPIs. Nevertheless, the actual execution time must also consider the effects of pipeline stalls that, in most cases, produce a severe devi-

ISSS'02, October 2–4, 2002, Kyoto, Japan.

ation from the ideal behavior. A more realistic estimate, i.e. the interlock-aware execution time, can be obtained by including a statistical term accounting for the stall overheads associated to every single instruction. Stall overheads are calculated by suitably averaging the contribution deriving from the *dynamic* interaction between instructions. In [3, 2] such overheads are evaluated by considering the possible interactions of all the instructions falling within a sliding window of fixed size. The window length is chosen to be equal to the maximum latency of an instruction in the pipeline since farther instructions do no influence each other. 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 s_1, s_2 and s_3 are executed in an ideal pipeline the resulting CPI is 1.0 for all of them. In a superscalar architecture with three ideal pipelines in parallel, the resulting CPI would be 1/3. However, real processors significantly differ from ideal architectures and only a portion of the theoretical parallelism can be exploited. To account for such deviation, a *parallelism coefficient* has been introduced and defined according to a statistical analysis of the execution of real-world programs on a given architecture. Indicating with $n(s)$ and $oh(s)$ the number of clock cycles for nominal execution and the number of stall cycles of instruction s and with $p(s)$ the parallelism coefficient, the estimated CPI is expressed as:

$$CPI_{est}(s) = p(s) \cdot [n(s) + oh(s)] \tag{1}$$

It is worth noting that the energy model defined in [2] can also be applied considering this new definition of CPI_{est}. The average power $w(s)$, however, increases, as shown by the relation:

$$w(s) = \frac{e(s)}{CPI_{est}(s) \cdot \tau} \tag{2}$$

where τ is the clock period. 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 has been 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 are assigned. The classification must account for the dynamic interaction between instructions with respect to both interlock effects and parallel execution.

DEFINITION 1. *Given an instruction set $\mathcal{I}$, the equivalence relation $\mathcal{R} \subseteq \mathcal{I} \times \mathcal{I}$:*

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

defines a taxonomy $\mathcal{C} \in 2^{\mathcal{I}}$ on the instruction set $\mathcal{I}$ as the partition induced by $\mathcal{R}$ on the instruction set $\mathcal{I}$. 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 have been envisioned:

Hazard The relation $\mathcal{R}$ is defined *a priori* and is based on the knowledge of the instruction set, the possible hazards and the architectural details

Full The relation $\mathcal{R}$ is always false. In this case each instruction represents a class of its own, i.e. no classification is performed.

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

Section 4 shows and discusses the results obtained using the first two classification paradigms.

2.2 Model Definition

The model expressed by equation (1) depends on two parameters: the interlock overhead $oh(s)$ and the parallelism coefficient $p(s)$. The present work extends a previous model, described in [2], in order to encompass superscalar architectures as well. The original model defined an *execution trace* Γ as a list of instructions resulting from the 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 by means of the relation $\mathcal{R}$ and the *membership function* is accordingly defined as:

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

The overhead $oh(\gamma_k)$, introduced by dynamic execution effects, is associated to the instruction that has been stalled in order to resolve a hazard situation. Based on the classification imposed by $\mathcal{R}$, such overheads have to be collected and associated to instruction classes. This leads to the definition of a stochastic variable D_i whose density is:

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

where N is suitably large[1] and δ is the Kronecker symbol. A good estimation of the overhead $oh(s)$ can, for example, be the expectation value of D_i:

$$oh(s) = E[D_i] = \sum_{x=0}^{\infty} x \cdot f_{D_i}(x) \quad \text{with} \quad s \in c_i, \; x \in \mathbb{N} \tag{6}$$

2.3 Parallel Execution Model

The parallelism coefficient 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 this approach, the more instructions $s \in c_i$ belonging to a given class are executed in parallel, the lower the corresponding parallelism coefficient $p(s)$ and $CPI_{est}(s)$ are. To determine $p(s)$ it is necessary to know when an 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.

[1]Experiments suggest that N should be greater than 10^7.

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:*

$$\lceil t, k, i \rfloor = \begin{cases} \langle k, i \rangle & \text{if } t_{in}(\gamma_k) \leq t \leq t_{out}(\gamma_k) \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

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

It is worth noting that the time range between $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ not only depends on the instruction latency but also includes the inter-instruction overhead resulting from stalls. 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:*

$$\lceil t, i \rfloor = \sum_{k=1}^{N} \lceil t, k, i \rfloor \quad (8)$$

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

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

$$p_t = \begin{cases} 1/\sum_{i=1}^{|\mathcal{C}|} \lceil t, i \rfloor & \text{if } \sum_{i=1}^{|\mathcal{C}|} \lceil t, i \rfloor \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

where the summation extends to all classes in the taxonomy.

Figure 1 clarifies these concepts with an example in which three functional units U_1, U_2 and U_3 execute eight instructions $\gamma_1, \ldots, \gamma_8$ belonging to the classes c_1, c_2 and c_3. The figure is composed of two parts: the upper portion shows the scheduling of instructions on each unit while the lower portion reports the values of $\lceil t, i \rfloor$ and p_t for the considered scheduling.

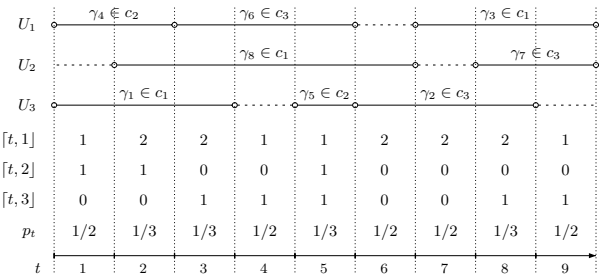

Figure 1: Example of parallelism computation

Consider, for instance, the clock cycle with $t = 3$ and with instructions γ_6, γ_8 and γ_1 being executed. The class load function $\lceil 3, 1 \rfloor$ is equal to 2 since $\gamma_8, \gamma_1 \in c_1$. Similarly, $\lceil 3, 3 \rfloor$ is equal to 1 since $\gamma_6 \in c_3$ and $\lceil 3, 2 \rfloor$ is 0 since no instructions of class c_2 are being executed. According to equation (9), the instantaneous parallelism coefficient p_3 is equal to $1/(2+0+1) = 1/3$. It can be proved that $p_t \in [1/M; 1] \cup \{0\}$ with M 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 [8]: in this case $M = 5$ since, when the pipeline is full, all its stages are executing an instruction at every clock cycle. In more complex architectures, where more pipelines are present and possibly share some of the stages, the computation of M becomes more sophisticated since the observation of the status of the single units of all pipelines is necessary. 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 formalizes this concept.

DEFINITION 5. *The* ***class parallelism coefficient*** *is a scale factor influencing the execution time of 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 the density function:*

$$f_{P_i}(x) = \frac{\sum_{t=0}^{\infty} \delta_{p_t = x} \lceil t, i \rfloor}{\sum_{t=0}^{\infty} \lceil t, i \rfloor} \quad (10)$$

where the summations actually extend only over all clock cycles needed for the execution of the trace Γ.

Referring again to the execution trace of figure 1, consider the density function $f_{P_3}(x)$. Since $p_t \in \{1/3,\ 1/2\}$ and thus $\delta_{p_t = x} = 1$ only when $x = 1/3$ or $x = 1/2$, then $f_{P_3}(x)$ is to be computed only for such values. In particular for $x = 1/3$:

$$\begin{aligned} f_{P_3}(1/3) &= \frac{\sum_{t=1}^{9} \delta_{p_t = 1/3} \lceil t, 3 \rfloor}{\sum_{t=1}^{9} \lceil t, 3 \rfloor} \\ &= \frac{0+1+1+1}{0+0+1+1+1+0+0+1+1} = \frac{3}{5} \end{aligned} \quad (11)$$

The same procedure leads to the result $f_{P_3}(1/2) = 2/5$. The parallelism coefficient $p(s)$, similarly to the instruction overhead, can conveniently approximated with the expectation value of the stochastic variable P_i, that is:

$$p(s) = E[P_i] = \int_0^1 x \cdot f_{P_i}(x) dx \quad \text{with} \quad s \in c_i,\ x \in \mathbb{Q} \quad (12)$$

It must be noted that $x \in \mathbb{Q}$ since it is computed as the ratio of two integer numbers and that $0 \leq x \leq 1$ by definition, thus the integral is computed according to the Lebesgue's definition of measure. Concluding the example, $p(s)$ for instructions in class c_3 is:

$$p(s) = \frac{1}{2} \cdot f_{P_3}(1/2) + \frac{1}{3} \cdot f_{P_3}(1/3) = \frac{2}{5} \quad (13)$$

3. BEHAVIORAL ANALYSIS

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, three quantities must be determined:

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

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

$$oh(\gamma_k) = t_{out}(\gamma_k) - t_{in}(\gamma_k) - n(\gamma_k) \quad (14)$$

While the nominal execution time $n(\gamma_k)$ can be found in the processor data-sheets, the times $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ can be determined by executing a *behavioral simulation*, i.e. a simulation that only accounts for the timing properties of the instructions while neglecting the functional and data-dependent behavior [13]. This is possible thanks to execution traces which account for these aspects by design. These considerations, and the goal of being as independent as possible from a specific architecture, have led to the development of a proprietary behavioral analysis tool-set organized according to the flow shown in figure 2.

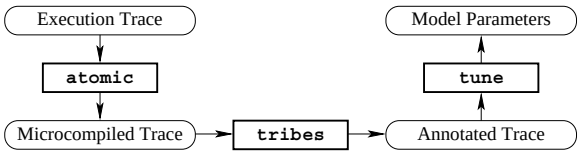

Figure 2: Behavioral analysis flow

The single steps of this flow and the intermediate results obtained are detailed in the following paragraphs.

3.1 Microcompilation

The execution trace is first fed to the `atomic` microcompiler that translates the architecture specific assembly into an expanded, loosely encoded micro-instruction set format called *microcode*. This step has the goal of decomposing the complex functionality of a generic instruction into a sequence of basic activities involving only few basic operations. Table 1 exemplifies the microcompilation process.

Table 1: Microcode examples

SPARCv8 Assembly	Microcode		
`umul %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`

Consider, as an example, the first instruction: the two `read`s are used to perform read requests to the integer register file `regfile-int`, specifically for registers `%r0` and `%r1`. The microinstruction `require` indicates that the execution of the multiplication is performed by the integer ALU `alu-int` 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 be *locked*, for example in the decode stage, to prevent subsequent instructions from using its content before the up-to-date value is actually present. The lock is then removed in a later stage. Other microinstructions work in a similar manner. As a further remark, it is 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 related effects.

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

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 3 describes this architecture.

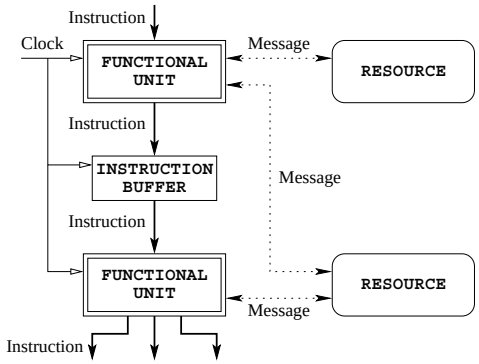

Figure 3: Behavioral simulator structure

Functional 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. A similar proprietary simulator (Asim) has been developed at Compaq [7]. However, neither sufficient details nor an implementation for a commercial architecture are currently available to allow a comparison with the framework presented in this paper.

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:

$$< instruction\_id > \quad < oh > \quad < t_{in} > \quad < t_{out} >$$

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 density functions f_{P_i} and f_{D_i} of the parallelism coefficients and timing overheads respectively. The results obtained in the tuning phase are then classified according to different equivalence relations (see Definition 1).

4. EXPERIMENTAL RESULTS

This section describes the experimental environment that has been set up to validate the simulator and to perform model tuning and verification of the estimation flow. Two different set of benchmarks have been used: one for the tuning of the model and the other for validating the results.

4.1 Simulator validation

The methodology and the simulation framework have been customized to model the microSPARC™-II Embedded Processor architecture. Figure 4 shows the internal structure of the behavioral simulator in terms of its basic components, i.e. functional units, buffers and resources. The figure also shows the paths of instructions through the various units. A detailed description of this architecture is out of the scope of this paper and can be found in [1]. To verify the correctness of the simulator engine, 12 benchmarks, taken from different application domains, have been simulated and the total estimated execution time has been compared with the actual execution time [6]. The actual execution time has been measured by running the benchmarks on the target platform configured to enable microstate accounting and to use high-resolution timers [16]. Each benchmark has been run with different sets of input data, leading to the overall results reported in table 2 both considering and ignoring memory effects. The delays caused by cache misses, write-buffer overflows and memory refresh have been currently determined separately using the SUN Microsystems proprietary simulator `uni_per` [15]. The overall average error obtained considering memory effects also is 4.19% with a standard deviation of 4.72%. This proves that the simulator has a satisfactory accuracy.

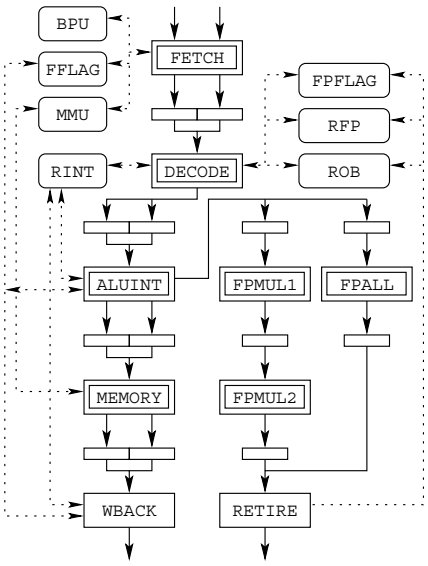

Figure 4: microSPARC-II simulator structure

Table 2: Simulation accuracy results

Code	Error (%) w/o Memory	Error (%) w/ Memory
`adpcm`	-10.47	-9.23
`gsm`	-11.87	-7.48
`lagrange`	-4.79	-3.01
`qsort`	-8.74	+1.39
`g723`	-4.20	-3.00
`fdct`	-10.68	-9.48
`crc16`	-0.83	+2.37
`md5`	-11.55	+5.15
`rle`	-2.79	-1.59
`bsort`	-2.64	-0.11
`matrix`	-34.27	-3.38
Overall	**9.33**	**4.19**

4.2 Estimation model validation

The toolset has then been used to determine the density functions f_{D_i} and f_{P_i} for each instruction using a subset of the benchmarks reported in table 2. For each instruction the expectation value of the variables D_i and P_i have been calculated both with *hazard* and *full* classifications. The traces used for tuning have been generated from the benchmarks `cpp2html`, `bc`, `gzip`, `mandel`, and `rasta`, for an overall trace length of approximately 1.5×10^8 instructions. This phase has led to two differently tuned models that have been been validated by estimating the execution time on all the benchmarks not used for the tuning phase. Figure 5 shows the results obtained by annotating the execution traces using the parallelism coefficients and the timing overheads resulting from the two considered classification schemes. In both cases the accuracy is more than satisfactory. The only exception is the `lagrange` benchmark which shows a much higher error. This is due to the fact that it executes thousands of times a very tight loop where branch folding occurs with a probability that is much higher than the average case. Nevertheless, the average relative error is as low as 13% without

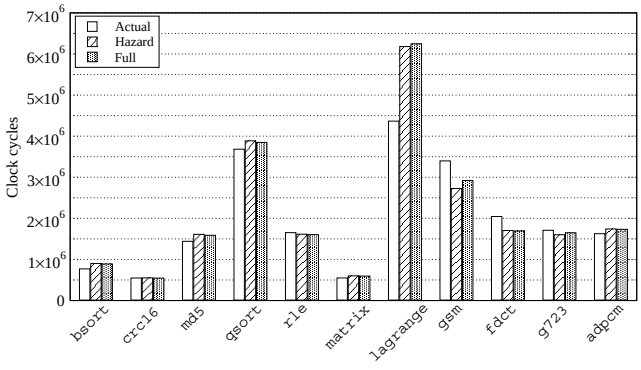

Figure 5: Accuracy of classification schemes

classification and 11% with the hazard-based one[3]. It is worth noting that using classification leads to an improved accuracy for two main reasons:

- the resulting model is less sensitive to the instruction trace used for tuning;
- the peculiar timing behavior of certain (possibly frequent) instructions is averaged with that of more regular ones, smoothing the effect of borderline cases.

4.3 Performance

The experiments have been performed on a dual-Pentium III 966MHz with 512MB RAM running Linux RedHat 7.2. The performance of all the processing phases, expressed in number of instructions processed per second, have been measured leading to the results summarized in table 3.

Table 3: Toolset performance

Phase	Tool	Performance
Instruction tracing	`bintrace`	1.9 Minst/sec
Micro-compilation	`atomic`	70 Kinst/sec
Behavioral simulation	`tribes`	4 Kinst/sec
Model tuning	`tune`	90 Kinst/sec
Estimation	`annotate`	140 Kinst/sec

Since the simulation flow involves the first three phases and since all tools can be concatenated in a single pipeline, the resulting simulation throughput is around 4 Kinst/sec. On the other hand, the estimation only requires instruction tracing which is much faster than estimation itself and thus does not impact on the estimation performance. The estimation flow is thus roughly 35 times faster than simulation. This justifies the construction of a model and the partitioning of instruction sets into classes.

5. CONCLUSIONS

The paper has presented a flexible approach to execution time estimation of assembly code for superscalar architectures. The starting point is a rigorous mathematical model previously developed by the authors and extended to include the effects of parallel execution and memory access. The model has proved sound and adequate. Its accuracy has been demonstrated by implementing a set of tools—configurable for different target architectures—leading to an average relative error around 11% for the estimation flow and below 5% for the simulation flow. The current efforts concentrate on the definition of a better classification scheme, the fine-tuning of the model parameters and the integration of a complete memory-hierarchy model in the simulation framework.

6. REFERENCES

[1] G. Beltrame. A Model for Assembly Instruction Timing and Power Estimation on Superscalar Architectures. Technical report, Cefriel Institute, March 2002.

[2] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, and V. Trianni. An assembly-level execution-time model for pipelined architectures. In *Proc. of Intnl. Conference on Computer Aided Design*, pages 195–200, San Jose, CA, November 2001.

[3] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. An instruction-level functionality-based energy estimation model for 32-bits microprocessors. In *Proc. of the 36th Design Automation Conference*, pages 346–350, 2000.

[4] K. Chen, S. Malik, and D. August. Retargetable static timing analysis for embedded software. In *Proc. of the 14th International Symposium on System Synthesis*, pages 39–44, 2001.

[5] T. Conte and C. Gimarc. *Fast Simulation of Computer Architectures*. Kluwer Academic Publishers, 1995.

[6] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proc. of the 28th Intnl. Symposium on Computer Architecture*, pages 49–58, 2001.

[7] J. Emer. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

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

[9] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, pages 552–559, 2000.

[10] M. Lazarescu, J. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *Proc. of the 6th Intnl. Workshop on Hardware/Software Codesing*, pages 65–69, 1998.

[11] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli. Software timing analysis using hw/sw cosimulation and instruction set siumulator. In *Proc. of the 6th Intnl. Workshop on Hardware/Software Codesing*, pages 65–69, 1998.

[12] S. Malik, M. Martonosi, and Y.-T. S. Li. Static timing analysis of embedded software. In *Proc. of the 33rd Design Automation Conference*, pages 147–152, 1997.

[13] S. S. Mukherjee, V. S. Adve, T. Austin, J. Emer, and S. P. Magnusson. Performance simulation tools. *Computer*, 35(2), 2002.

[14] D. Sima, T. Fountain, and P. Klaksuk. *Advanced Computer Architectures – A Design Space Approach*. Addison-Wesley, 1998.

[15] Sun Microsystems. microSPARC™-IIep source distribution. `http://www.sun.com`.

[16] Sun Microsystems. Prying into processes and workloads. Unix Insider, 4/1/98.

[3]Excluding the critical case of `lagrange`, the error drops to less than 8%.