Fine Grain Analysis of Simulators Accuracy for Calibrating Performance Models

Marco Lattuada, Fabrizio Ferrandi

Politecnico di Milano - Dipartimento di Elettronica ed Informazione
Via Ponzio, 34/5 – 20133 – Milano (Italy)
{lattuada,ferrandi}@elet.polimi.it

***Abstract*—In embedded system design, the tuning and validation of a cycle accurate simulator is a difficult task. The designer has to assure that the estimation error of the simulator meets the design constraints on every application. If an application is not correctly estimated, the designer has to identify on which parts of the application the simulator introduces an estimation error and consequently fix the simulator. However, detecting which are the mispredicted parts of a very large application can be a difficult process which requires a lot of time.**

In this paper we propose a methodology which helps the designer to fast and automatically isolate the portions of the application mispredicted by a simulator. This is accomplished by recursively analyzing the application source code trace highlighting the mispredicted sections of source code. The results obtained applying the methodology to the TSIM simulator show how our methodology is able to fast analyze large applications isolating small portions of mispredicted code.

I. Introduction

Performance evaluation is a critical aspect of an embedded system design flow since designers have to assure that final solution will meet the timing constraints. To evaluate the performances of an architecture on one or more applications, three main families of methods exist: measuring them directly on the real platform, estimating them by using simulators or estimating them by using mathematical models.

The first solution is not an applicable option most of the time because the platform is developed parallel to the software. On the other hand, several simulation based methods for estimating the performance of a multiprocessor system-on-chip architectures exist which provide a different trade-off between efficiency and accuracy [1]. Analyses based on mathematical models are usually faster but less accurate when compared to cycle accurate simulators. For these reasons, during first stages of the design flow, mathematical models may be preferred to speed-up the design space exploration. On the contrary, to refine the solution during following phases, simulators may be preferred for their accuracy.

Accuracy guaranteed by cycle accurate simulators is usually very high; however, the estimation error introduced on a single application can be much higher than the average. Moreover, errors which can be negligible considering a whole application, can become very relevant when considering its single parts. In particular, an estimation error of few percentage points on a critical part of the application will not be acceptable in final stages of a design flow, so we need to improve the accuracy of the simulator itself. Tuning a simulator or the machine model on which it is based is not a trivial task since single relevant estimation errors may be caused by corner cases. Identifying and isolating which are the parts of the application which stimulate these corner cases can help the simulator designer to improve the accuracy of the simulator.

In this paper we propose a methodology for helping the debugging and the tuning of generic cycle accurate simulators by identifying the portions of the C source code of an application whose performances are mispredicted.

The main characteristics of the methodology are:

- it is based on the recursive partitioning of the application execution trace at source code level; the complete execution trace is not needed since trace portions are dynamically computed by the methodology itself;
- the finer granularity at which the analysis can work is at C statement level (lower granularity like the assembly level is out of the scope of this work);
- the only performance measure which the simulator and the real system must provide is the overall execution time of modified versions of the application.

The paper is organized as follows: Section II presents the Related work, Section III describes the proposed methodology, Section IV shows the experimental results of the application of this methodology while Section V draws some conclusions and outlines some future works.

II. Related work

In the last years several methodologies for automatic generation of simulators have been proposed. These methodologies start from a high level description of the target machine and use this information to build the simulator. The precision of the machine description highly influences the accuracy of the produced simulator. Identifying errors in the machine description requires stimulation of the parts of the performance model where the errors are present. For this reason, several strategies have been proposed for choosing the set of benchmarks used to test the accuracy of the simulator. Black et al.

978-1-42447074-7/10/$26.00 ©2010 IEEE

[2] propose a taxonomy of the possible errors introduced in performance models and present a methodology which aims at exploiting them. This is accomplished by automatically generating benchmarks sets for stressing different portions of the models. However, this type of benchmarks is not able to detect all the possible errors of a model, so some hand-written benchmarks are added to test suites; however, even with this addiction the complete coverage of all the possible errors is not guaranteed.

Desikan et al. [3] propose to reduce the testing suite to only handwritten tests: they present a test case for validating the Alpha 21264 using a suite of 21 microbenchmarks. Also in this case they can not assure that all possible errors of the simulator are covered by the testing suite. Using only this type of benchmarks can be harmful: they can not stimulate errors which occur in complex situations and they can be biased if written by the simulator designer. Indeed, if the designer has ignored a particular condition during simulator design, he likely does not write a test case which stimulates it.

Pimentel et al. [4] propose a semi-automatic methodology for tuning parameters of performance models. Applications are described using Kahn Process Networks: the methodology tunes the parameters of both the single component performance models and of the whole system. The disadvantages of this approach consists of having to model the application as a Khan Process Network. Moreover, the single component performance models must be linear and the methodology tunes them considering only a single application.

III. The proposed methodology

The methodology proposed in this paper is based on multiple binary search: the execution trace of an application is recursively partitioned and analyzed at source code level to identify mispredicted portions of code. The methodology is proposed for the analysis of applications written in C code, but it can be very easily extended to support other programming languages. It is completely automatic, but it can be driven by the designer by tuning two parameters:

- `error`: the maximum relative error which the designer considers tolerable; `error` is defined as

$$error = \frac{|t_{real} - t_{estimated}|}{t_{real}}$$

  where t_{real} and $t_{estimated}$ are the cycle execution times measured on the real system and estimated by the simulator;
- `size`: the minimum size of a piece of code of the application whose analysis is considered significant; how to measure the size of a piece of code can be decided by the designer (e.g.: measured execution cycles, number of source code statements); through this parameter the designer controls the granularity level of the analysis.

The recursive analysis of the methodology works on *Code Regions* but, before defining what a Code Region is, we have to introduce some definitions about traces at source code level. We define:

```
    double a[90], b[90];

    int main()
    {
       int i, temp;
A:     for(i=0; i < 90; i++)
       {
B:        if(!(i%2))
C:           even(a, i);
          else
D:           odd(a, i);
       }
E:     printf("%f\n", a[89]);
F:     for(i=0; i < 90; i++)
       {
G:        temp = 1 << ((i-1)%32);
H:        if(i%2)
I:           b[i] = a[i]*a[i] + temp;
          else
J:           b[i] = temp;
       }
K:     printf("%f\n", b[89]);
L:     printf("%f\n", a[89]+b[89]);
M:     return 0;
    }

    void even(double * array, int i)
    {
N:      array[i]=sin(i);
    }

    void odd(double * array, int i)
    {
O:     array[i]=cos(i);
    }
```

Fig. 1: Source code of functions `main`, `even` and `odd`

- $Instr$: the set of C statements of an application;
- $s = I1, I2, \ldots, IN$: a sequence of C statements;
- $pred_s(I)$: the predecessor of I in sequence s;
- $succ_s(I)$: the successor of I in sequence s.

Given a single application execution, we define the only complete sequence of executed operations as *application source code trace* Tr. For each statement $I \in Instr$, we annotate each its occurrence in Tr with a progressive number; in this way each annotated element of Tr is unique.

The application source code trace Tr of the example presented in Figure 1 is tabled in Table I(a). We report only the first and last iteration of both the loops. Each new line corresponds to the start of a new loop iteration or of a new function execution, each column corresponds to all the executions of a particular statement. The sequence Tr can be

TABLE I: Example of Code Regions
$Tr_{function}$ are the function source code traces, Id is the identifier of the sequence, P is how the sequence is composed, $IsCr$ reports if the sequence Cr_{Id} is a Code Region; if Cr_{Id} is also a Loop Code Region, Lr is its identifier.

(a) The application source code trace of the example of Figure 1.

Tr_{main}	Tr_{even}	Tr_{odd}
A_0 B_0 C_0		
	N_0	
A_1 B_1 D_0		
		O_0
A_{89} B_{89} D_{44}		
A_{90} E_0		
F_0 G_0 H_0 J_0		
F_1 G_1 H_1 I_0		
F_{89} G_{89} H_{89} I_{44}		
F_{90} K_0 L_0 M_0		

(b) Examples of sequences.

Id	P	$IsCr$	Lr
Cr_1	$A_0B_0C_0$	Yes	$l_1(0,0)$
Cr_2	$A_0B_0C_0$ A_1B_1	Yes	
Cr_3	$A_0B_0C_0$ $A_1B_1D_0$	Yes	$l_1(0,1)$
Cr_4	$A_0B_0D_0$	No	
Cr_5	$A_1B_1D_0$	Yes	$l_1(1,1)$
Cr_6	$A_{90}E_0$	Yes	
Cr_7	C_0N_0	No	
Cr_8	N_0	Yes	$l_0(0,0)$

obtained by simply reading the Table from left to right and from first row to last row.

Given an application source code trace Tr, a *function source code trace* Tr_f is the projection of Tr on the source code statements which belong to the function f. Since each element of Tr is unique, also each element of Tr_f is unique. The Tr_f of a function f of the application of Figure 1 can be obtained by reading the sequence of Table I(a) ignoring the column not belonging to f.

Given a function source code trace Tr_f, a Code Region Cr is a contiguous sub-sequence $J1_{j1}, J2_{j2}, ..., JN_{jn}$ extracted from it. Since each element of Tr_f is unique, we can identify a Code Region by its boundaries.

Examples of Code Regions are presented in Table I(b). Cr_4 is not a Code Region since it is not contiguous; Cr_7 is not a Code Region since C and N belong to different functions.

A subset of Code Regions is the set of *Loop Code Regions*. Before defining them, we have to introduce some notations about loops:

- $L_f = \{l\}$ is the set of loops present in the source code of function f identified by a modified version of the Sreedhar-Gao-Lee algorithm [5];
- l_0 is the outmost loop of the function corresponding to the whole function; it is introduced to generalize the loop structure: all the other loops of the function are nested in it;
- $\forall l \in L_f, i = H(l)$ identifies the first statement of the header of loop l.

In function `main` of Figure 1, we have two `for` loops identified as l_1 and l_2: $A = H(l_1)$ and $F = H(l_2)$

A Code Region $Cr = [I1_n, J1_o]$ is a Loop Code Region Lr if its statements correspond to exactly one or more iterations of a loop:

$$\exists l : I1 = H(l) \wedge J1_o = pred_f(I1_p)$$

The first condition states that the Loop Code Region must start with the first statement of a loop l; the second condition states that the first statement after the Loop Code Region must be the starting of a new iteration of the same loop l. We identify this Loop Code Region as $Lr_l(n, o)$: l identifies the loop, n and o the first and the last iteration which belong to the Loop Code Region.

Examples of Loop Code Regions are presented in column Lr of Table I(b). Cr_1, Cr_3 and Cr_5 are Loop Code Regions since they start with A_j ($A = H(l_1)$), they end with C_0 ($C_0 = pred(A_1)$) and D_0 ($D_0 = pred(A_2)$). Cr_8 is a Loop Code Region because of the outmost loop l_0 we have introduced.

Given the definition of Code Region, we now describe the methodology which in pseudo-code is shown by Algorithm 1.

The input of the methodology is the boundaries of the Code Region $Cr = [start_{Cr}, end_{Cr}]$ to be analyzed (initial input is the boundaries of the whole application: the first and last statement). The output is the set of mispredicted Code Regions. The methodology does not need to have the complete input Code Region nor does it require to compute it: starting from the boundaries of Cr, it dynamically computes all the $I \in Cr$ it needs.

Since the analysis is recursive, we have to describe its two main components: the *Base Case* and the *Recursive Step*.

The conditions under which the *Search* algorithm reaches the Base Case are the following:

- ⓐ the error in the analyzed Code Region Cr is not significant (Line 2); how the error is computed will be described in Section III-B;
- ⓑ Cr is too small to be analyzed (Line 4);
- ⓒ Cr is considered atomic and so not splittable (Line 11);
- ⓓ Cr is a call to an already examined function or to a library call (Line 6);
- ⓔ $Cr = Lr_l(o, p)$ (Line 13) and loop l has already been examined (Line 15).

During the Recursive Step, the methodology, after checking that no termination condition has been met, splits the current Cr into two halves (Line 21). How the splitting point pair $Ml - Mr$ is computed will be described in detail in Section III-A. Having gotten the splitting points, the analysis continues recursively on the two new Code Regions obtained (Lines 22

Algorithm 1 Pseudo-code of *Search*

```
1:  function SEARCH(start_Cr, end_Cr)
2:      if GETERROR(start_Cr, end_Cr) ≤ error then
3:          return Ø
4:      else if GETSIZE(start_Cr, end_Cr) ≤ size then
5:          return Cr
6:      else if ISOLDCALL(start_Cr, end_Cr) then
7:          return Cr
8:      else if ISNEWCALL(start_Cr, end_Cr) then
9:          Called ← GETCALLED(start_Cr, end_Cr)
10:         return SEARCH(Called.begin, Called.end)
11:     else if NOTSPLITTABLE(start_Cr, end_Cr) then
12:         return Cr
13:     else if ISLR(Cr) then
14:         loop ← GETLOOP(Cr)
15:         if loop ∈ visited_loops then
16:             return Cr
17:         else
18:             visited_loops ← visited_loops ∪ loop
19:         end if
20:     end if
21:     Ml, Mr ← GETMIDDLE(start_Cr, end_Cr)
22:     left ← SEARCH(start_Cr, Ml)
23:     right ← SEARCH(Mr, end_Cr)
24:     return left ∪ right
25: end function
```

and 23).

A. GetMiddle: Splitting a code region

Where a code region is split is a crucial step of this methodology since it determines the number of iterations of the analysis and so the overall execution time of the methodology. The solution of partitioning a Code Region into two performance equivalent regions has been rejected because of its requirements. Indeed, to implement it, we need two types of information which can be not available: the complete application source code trace and an estimation model of the target.

On the contrary, the criterion we selected is cheaper in terms of information required and more focused on the main aim of this methodology which is to provide the simulator designer a fast and automatic method to isolate the portion of the application which is mispredicted by the simulator. Dynamic execution of the application is not the only information used by analysis: we combine it with the Control Flow Graphs of each function for driving the exploration of the application, providing more readable results to the designer.

Given a Code Region Cr, first of all analysis checks if all its statements belong to the same basic block or not. In the first case, it checks the number of function calls present in Cr:

- ⓕ zero function call: said N the number of the statements of Cr, it is split after the $\lfloor N/2 \rfloor$ statement;
- ⓖ one function call, which is also the only statement of the Code Region; if the called function has already been examined, the analysis stops (Termination rule ⓓ), otherwise it proceeds recursively on its function source code trace (Line 8);
- ⓗ one function call, which is the last statement of the sequence: the region is split immediately before the call;
- ⓘ one function call, which is not the last statement of the Code Region; it is split immediately after the call;
- ⓙ $N > 1$ function calls: the region is split after the $\lfloor N/2 \rfloor$ function call.

If statements of Cr belong to more than a basic block and Cr is an iteration, or part of an iteration, of a loop l_i, analysis checks the loop contained in it:

- ⓚ Cr does not contain any loop l_j nested in l_i and it is composed by statements belonging to N basic blocks: Cr is split after the last operation of the $\lfloor N/2 \rfloor$ Basic Block;
- ⓛ Cr contains a single nested loop l_j and starting of Code Region is also the header of loop l_j: Cr is split after the last iteration of loop l_j contained in Cr;
- ⓜ Cr contains a single nested loop l_j and starting of Code Region is not the header of loop l_j: Cr is split before the first iteration of l_j contained in Cr;
- ⓝ Cr contains $N > 1$ nested loop: Cr is split before the first iteration of the $\lfloor N/2 \rfloor$ loop contained in Cr.

If Cr is composed exactly by $N > 1$ iterations of a loop l_i:

- ⓞ Cr is split at the end of the $\lfloor N/2 \rfloor$ iteration among the ones contained in the Code Region.

Finally, there are some types of Code Region which are not treated by `GetMiddle` since none of the previously presented bisection rules (ⓕ to ⓝ) can create them starting from Tr. These types are:

- ⓟ Cr contains parts of two different iterations of the same loop;
- ⓠ Cr contains parts of two or more iterations of different loops.

Figure 2 shows an example of application of this methodology: it shows the search tree built during the analysis of the application of Figure 1 when statements D, E_0, G_1, K_0, L_0, N, are mispredicted. The analysis proceeds in depth first order; the label of an intermediate node corresponds to the rule applied in that point of the search. The dotted parts of the tree represent chains of applications of rule ⓞ which have been omitted. Each leaf is a Code Region which has been examined and not split: plain line boxed leaves are the mispredicted ones. The union of all the leaves is equal to the application source code trace.

B. GetError: Measure the simulator error on a code region

The simulator error is computed comparing the performance estimation produced by the simulator with the measure of

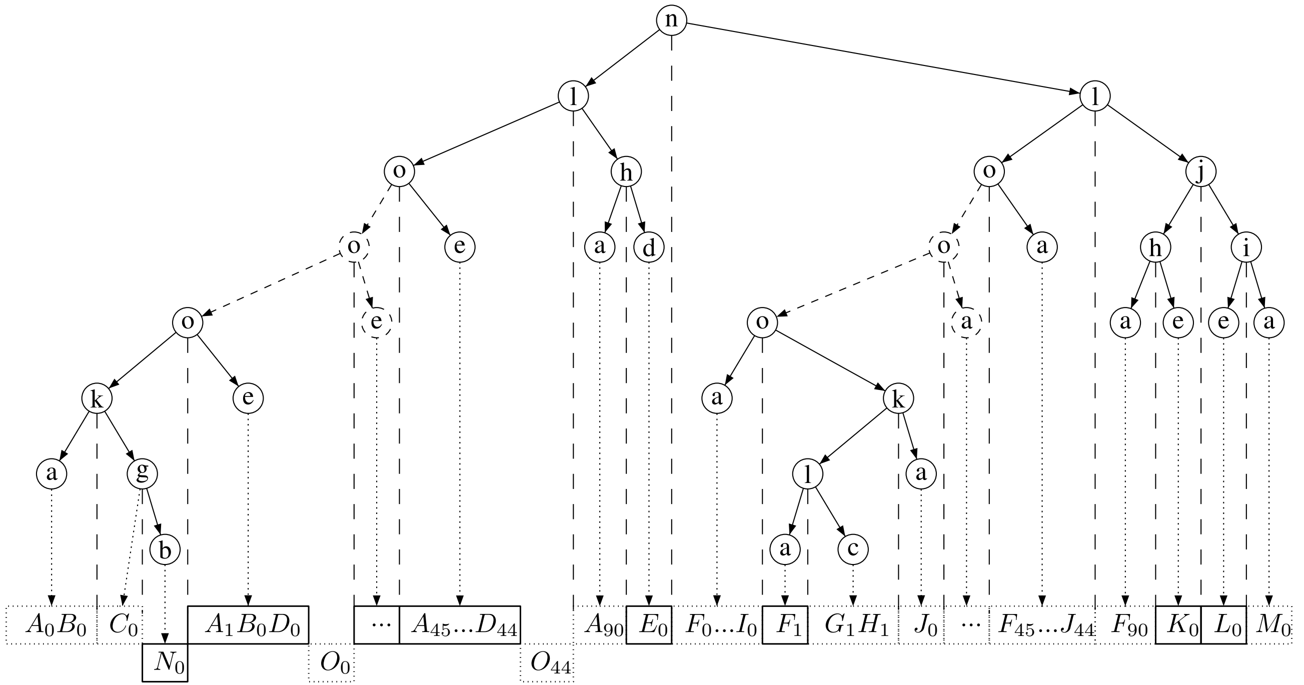

Fig. 2: Search tree produced by application of the methodology to application of Figure 1. The label of each intermediate node corresponds to the rule applied. Leaves correspond to Code Regions which have not been split during analysis.

```
///Last statement of code region
<statement>;
if(__counter-- == 0)
   exit();
```

Fig. 3: Example of added instrumentation

the real platform. Since not all the simulators and platforms allow getting the execution time at an arbitrary point of the execution, we propose a profiling technique based on code instrumentation to retrieve these data. In particular, we instrument the source code to stop the application execution in a given point of the application. This type of profiling can be exploited since we are working at source code level: working at assembly level would require a more complex target-dependent instrumentation technique. The execution time of the Code Region is computed as the difference between the execution times of the application stopped after and before the Code Region itself. An example of the added code is shown in Figure 3.

This type of instrumentation provides two advantages: generality and lightness. Indeed, in this way we guarantee the generality of the methodology since we only require that the platform and the simulator provide the overall execution time of an application execution. Moreover, since the instrumentation is limited to a single point, we reduce its perturbation on the application performance. However, if the simulator or the platform provides a method for directly measuring the execution time of an arbitrary portion of code, it can be exploited.

All the execution times collected during the search are cached. In this way, in all the iterations of the methodology but the first one, we only have to perform a single run of the simulator and of the platform. Indeed, if the current Code Region is the left (right) part of its parent, we have already measured its starting (ending) time which is the same of the parent.

C. *Methodology simplifications*

Some simplifications have been introduced in the proposed algorithm to speed-up the methodology.

The first one consists in assuming that if the estimation error on a Code Region Cr is non significant, the simulator correctly estimates the performance of every part of the Code Region itself (Termination rule ⓐ). This assumption is not necessarily true: a simulator can overestimate some parts of this Code Region and underestimate others producing an overall null error.

An example of this situation is shown in Table II. Estimation error on Control Region Cr_{11} is 0%, but the two Control Regions Cr_9 and Cr_{10} which compose it are mispredicted.

To be sure that simulator correctly estimates all the single parts of the application, we should exhaustively examine the application.

TABLE II: Example of effects of combination of Estimation Errors

Cr	Sequence	Execution Time		Error
		Real	Simulated	
Cr_9	A_0	10	5	50%
Cr_{10}	B_0	5	10	100%
Cr_{11}	A_0B_0	15	15	0%

The next simplification, implemented by termination rule ⓔ, is the analysis of only the first mispredicted iteration of a loop and not of all ones (Line 15 of the Algorithm 1). This simplification reduces the complexity of the analysis of a loop body executed n times from $O(n)$ (exhaustive search of all iterations in the worst case) to $O(\log n)$ (binary search of the first mispredicted iteration). Since the number of runs on simulator and platform is linearly proportional to the number of SEARCH invocations, this simplification can reduce in a relevant way the overall execution time of the methodology.

The cost paid for this simplification is less accurate results: we still identify all the mispredicted Code Regions, but some of them can be bigger than necessary: they can also contain correctly estimated Code Regions. For example, in the mispredicted Code Regions produced by the analysis shown in Figure 2, we have $Cr = [[A, 1][D, 1]]$ instead of $Cr = [O, 0]$. However, since each execution of the methodology correctly identifies and isolates at least one error for each loop, the successive applications of methodology and fixings of the simulator allow to completely debug the simulator.

Last simplification is quite similar, but it concerns function calls: only the first mispredicted execution of a function is examined instead of all ones (Termination rule ⓓ). Also in this case, to guarantee the isolation of all the existing misprediction errors, the designer can debug the simulator by repeating execution of methodology combined with simulator fixing.

IV. EXPERIMENTAL RESULTS

We have verified the proposed methodology by integrating it in PandA [6], a framework for the Hardware/Software codesign based on the GNU GCC compiler [7]. The framework has been used to verify the methodology by analyzing the accuracy of the TSIM simulator [8] on a set of benchmarks extracted from the DSPStone [9], MediaBench [10] and PowerStone [11]. TSIM is a cycle accurate simulator of the LEON3 processor [12], a 32-bit microprocessor compliant with the SPARC V8 ISA. Based on LEON architecture originally designed by the European Space Agency, it is currently developed by Gaisler Research licensed under GNU GPL license.

The simulator results have been compared with the ones produced by a single processor LEON3 System synthesized on a Virtex-5 XUPV5 Board. The benchmarks have run both on the real system and on the simulator without operating system to exclude its non-deterministic effects which can invalidate the analysis. All the benchmarks have been compiled with BCC 4.4.1, a GNU GCC based cross-compiler with a simple bare-C run-time with interrupt support. To avoid any non-deterministic effect we have automatically excluded from the profiling all the system call functions such as `printf`. Indeed this type of function has proved to have a sensible variance in execution time also when run on a system without operating system. Finally, we have had to exclude from the analysis all the benchmarks which read input data from files because of the operating system absence.

The results of the analyses are reported in Table III. All the benchmarks have been tested compiling them without optimization (O0 level) and with maximum optimization (O3 level). The analyses have been performed with the following parameters: `error` has been set to 5%, `size` to $1,000$ cycles and we consider as not splittable a basic block without function call since in almost every case is smaller than `size`.

The results concerning benchmarks which are correctly estimated with both optimization levels have not been reported in the Table. For each mispredicted benchmark we report its size in terms of source code (*Lines Numbers*) and of overall number of Basic Blocks in the Control Flow Graphs (*Static BB*). We also report the size of the execution trace of the application expressed in terms of number of Basic Blocks which compose it. For both the optimization level the overall error (*Error*) on the application and the number of Basic Blocks identified as mispredicted (*Wrong BB*) are reported. Finally we report the recursions of the methodology (*Recur.*) and the overall time (*Time*) required to execute it on a Intel Quad-Core Xeon X5355 (2,33 GHz, 4 MB L2 cache per couple of cores) with 8 GB RAM.

The detailed results about the identified mispredicted Code Regions can be used to identify the conditions under which TSIM may introduce an estimation error. For example, in the *fir2* and *wavelt* benchmarks, most of the estimation error has been introduced on a basic block composed by a long sequence of multiplies and sums of elements of globally declared array. In the case of *complex update* set of benchmarks, the error seems to be caused by the use of the complex arithmetic during data access. The misprediction error on the *hanoi* benchmark, instead, seems to be due to the presence of a recursive function.

To show how the methodology parameters influence its performance and results, we analyze the *jpeg* application with different combination of *error* and *size*. The results of these analyses are presented in Table IV.

The Table shows how the trade-off between the results precision and the performance of the analysis is controlled by *error* and *size*. For example, set the *error* to 1%, the reduction of the *size* from $10,000$ to 100 removes 26 Basic Blocks, which were false positive, from the set of the identified mispredicted Basic Blocks. On the other hand, set the *size* to 1000, the reduction of the tolerable *error* from 2.5% to 1% add two new Basic Blocks to the set of the mispredicted.

V. CONCLUSIONS

In this paper we present a methodology for fast isolating the estimation errors introduced by a simulator on a C application. The methodology is based on the recursive analysis of the

TABLE III: Results of the analysis of the TSIM performance estimation

				O0 Optimization				O3 Optimization			
Name	Lines Number	Static BB	DynamicBB	Error (%)	Wrong BB	Recur.	Time (s)	Error (%)	Wrong BB	Recur.	Time (s)
bcnt	95	8	$3.80 \cdot 10^1$	14.07	2	4	17	13.69	2	4	19
blit	102	25	$4.01 \cdot 10^3$	12.37	17	3	116	13.43	3	12	52
complex multiply 1	58	2	$2.00 \cdot 10^0$	19.29	2	3	44	15.76	1	3	36
complex multiply 2	70	12	$7.00 \cdot 10^1$	15.76	1	4	16	9.29	3	7	23
complex update 1	78	2	$2.00 \cdot 10^0$	5.07	1	3	17	1.40	0	1	3
complex update 2	81	2	$2.00 \cdot 10^0$	5.56	1	3	17	1.71	0	1	3
convolution	73	9	$7.10 \cdot 10^1$	7.49	2	9	30	8.14	3	7	23
fir2	118	11	$1.73 \cdot 10^3$	9.54	3	13	60	9.40	3	13	45
fir2dim 1	149	38	$7.21 \cdot 10^2$	6.80	2	22	74	7.31	3	38	131
fir2dim 2	152	46	$8.23 \cdot 10^2$	11.75	5	38	129	13.04	7	59	213
g3fax	653	57	$3.31 \cdot 10^5$	6.19	1	6	30	4.77	0	1	6
gamma	41	15	$1.73 \cdot 10^2$	7.38	1	2	30	4.01	0	1	15
hanoi	122	26	$1.81 \cdot 10^8$	2.26	0	1	150	5.95	4	14	570
iir biquad N sections 1	117	11	$1.33 \cdot 10^2$	6.75	2	6	43	12.52	2	14	47
iir biquad N sections 2	119	23	$1.34 \cdot 10^2$	19.62	2	13	54	7.58	4	16	54
iir biquad one sections 1	94	2	$4.00 \cdot 10^0$	18.54	1	7	23	1.78	0	1	3
iir biquad one sections 2	86	2	$4.00 \cdot 10^0$	18.20	1	7	23	1.49	0	1	3
jpeg	931	155	$4.74 \cdot 10^5$	4.02	0	1	6	6.01	1	6	32
lms 1	156	18	$1.36 \cdot 10^2$	14.10	3	10	33	13.20	4	13	12
lms 2	156	11	$8.20 \cdot 10^3$	22.43	4	13	43	7.36	5	14	50
main16 bit reduct	125	80	$4.01 \cdot 10^2$	6.41	4	17	3	5.55	4	17	3
main1024 bit reduct	149	40	$7.22 \cdot 10^4$	5.86	8	36	297	0.28	0	1	12
main1024 inspca	125	37	$3.12 \cdot 10^4$	6.45	8	32	263	0.14	0	1	10
matrix 1	131	20	$3.65 \cdot 10^3$	15.18	4	19	3	10.38	4	20	4
matrix 2	131	38	$3.22 \cdot 10^3$	11.91	5	19	3	10.30	3	14	4
matrix1x3 1	99	14	$6.20 \cdot 10^1$	9.31	2	11	23	7.11	2	10	23
matrix1x3 2	76	7	$3.30 \cdot 10^1$	5.01	2	6	34	5.41	2	6	12
n complex update 1	90	8	$1.05 \cdot 10^2$	8.80	3	11	36	7.82	4	10	34
n complex update 2	89	14	$1.03 \cdot 10^2$	6.99	3	11	37	9.06	4	10	34
n real updates 1	70	8	$1.05 \cdot 10^2$	9.27	3	11	36	8.78	4	10	35
n real updates 2	69	14	$1.03 \cdot 10^2$	2.28	0	1	37	22.89	3	11	35
pi	680	81	$2.00 \cdot 10^6$	5.08	1	11	594	0.12	0	1	65
real update 1	71	3	$2.00 \cdot 10^0$	8.92	1	2	13	1.59	0	1	4
real update 2	66	2	$2.00 \cdot 10^0$	6.61	1	2	13	1.38	0	1	4
SearchGame	456	185	$5.04 \cdot 10^7$	10.00	1	21	572	1.20	0	1	52
startup	193	33	$5.79 \cdot 10^2$	5.64	4	31	84	5.20	4	31	107
wavelt	155	63	$8.86 \cdot 10^2$	7.42	10	44	157	7.43	9	37	148

TABLE IV: Analysis results on *jpeg* benchmark (O0 optimization) with different parameters.

`error`	`size` (cycles)	Wrong BB	Recur.	Time (s)
1%	100	13	53	75
	1000	15	49	60
	10000	39	29	32
2.5%	100	13	46	50
	1000	13	31	37
	10000	23	18	22
5%	–	0	1	6

application execution trace at source code level. We do not need the complete execution trace: in each iteration of the methodology, only the points needed for the analysis are dynamically computed. The results show that our approach can be used to fast analyze also large applications in a few iterations. Moreover, by tuning the methodology parameters, the designer can choose the desired trade-off between the analysis precision and its execution time. Future works will consist of improving the methodology precision in presence of multiple loop iterations and multiple mispredicted function calls.

REFERENCES

[1] Joshua J. Yi, Resit Sendag, David J. Lilja, and Douglas M. Hawkins. Speed versus accuracy trade-offs in microarchitectural simulations. *IEEE Trans. Comput.*, 56(11):1549–1563, 2007.

[2] Bryan Black and John Paul Shen. Calibration of microprocessor performance models. *Computer*, 31(5):59–65, 1998.

[3] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 266–277, New York, NY, USA, 2001. ACM.

[4] Andy D. Pimentel, Mark Thompson, Simon Polstra, and Cagkan Erbas. Calibration of abstract performance models for system-level design space exploration. *J. Signal Process. Syst.*, 50(2):99–114, 2008.

[5] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 18(6):649–658, 1996.

[6] PandA framework, http://trac.ws.dei.polimi.it/panda.

[7] GNU Compiler Collection. GCC, version 4.3.

[8] Tsim erc32/leon simulator, www.gaisler.com.

[9] V. živojnović, Juan M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *ICSPAT '94*, 1994.

[10] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[11] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In *ISLPED '00*, pages 241–243, New York, NY, USA, 2000. ACM.

[12] Leon3 sparc v8 processor, www.gaisler.com.