CC4CS: A Unifying Statement-Level Performance Metric for HW/SW Technologies (*DSD 2017*)

Innovative Off-The-Shelf ESL Performance Metric: Statistical Analysis in the SW Domain (*EPEW 2017*)

Vincenzo Stoico, Vittoriano Muttillo, Giacomo Valente, Luigi Pomante[1] and Fausto D'Antonio[2]

[1] Università Degli Studi Dell'Aquila, Center of Excellence DEWS, Via Giovanni di Vincenzo, 16/B, 67100 L'Aquila
{vincenzo.stoico}@student.univaq.it
{vittoriano.muttillo, giacomo.valente}@graduate.univaq.it
{luigi.pomante}@univaq.it
[2] Thales Alenia Space, Via Campo di Pile, Nucleo Industriale di Pile, L'Aquila, Italy
{fausto.dantonio-somministrato}@thalesaleniaspace.com

Abstract. Outlining the general characteristics of embedded systems is an arduous task because the design of such kind of systems is heavily influenced by functional and non-functional requirements, and it is based on quite complex design flows. So, there is the need to define a general methodology able to support the designers during high-level phases so that they can perform very early analysis before dealing with low-level ones. Such a methodology, to be effective, should take into account performance estimation and ESL HW/SW timing co-simulation. So, the goal of this paper is to present a high abstraction-level (i.e. statement-level) performance metric, able to be unifying for HW and SW, in order to speed-up very early analysis and design space exploration, and to early identify the more promising architectures for different application domains. In particular, the paper analyzes the proposed metric from a statistical point of view, in order to evaluate its meaningfulness and accuracy when exploited in the SW domain.

Keywords: Performance evaluation; Timing measurement; Early analysis; Metrics; Simulation.

1 Introduction

In the last thirty years there has been an exponential increase of the exploitation of embedded ICT technologies. In fact, the presence of embedded systems in everyday life is ever more considerable and, at the same time, invisible. Due to their HW/SW heterogeneity and the critical importance of non-functional (a.k.a. extra-functional) constraints, for such kind of systems the adopted HW/SW co-design methodology is a key factor for a successful development. In such a context, early HW/SW performance estimation and HW/SW timing co-simulation are always fundamental steps. One of the most important metrics for SW performance estimation is MIPS (Million of Instructions per Second) [REF] because it is normally available directly on the micro-processor data-sheet (it is evaluated offline and immediately available, i.e. it is in some way Off-The-Shelf). MIPS metric can be useful for comparing different micro-processors with the same (or very similar) ISA (Instruction Set Architectures) but it is pointless in comparing different ones.

In such a context, the objective of this work is to analyze the features and the meaningfulness of a performance metric (called CC4CS) that would be similar to MIPS, with respect to Off-The-Shelf availability, but related to a higher-level of abstraction. In fact, CC4CS considers high-level programming language statements (i.e. statement-level) instead of assembly instructions and so it is usable to directly com-pare different Processor Technologies [REF]: processors built to execute a given ISA (General Purpose Processors, GPP; Application Specific Processors, ASP) and processors built to directly execute (i.e. NO ISA involved) applicative functions (Single/Specific Purpose Processors, SPP). It is so clear that an Off-The-Shelf performance metric suitable for both HW and SW technologies (i.e. HW/SW unifying) is an ideal one for the very early steps of an ESL (Electronic System Level) HW/SW Co-Design Methodology (e.g. [REF]). In particular, this paper focuses on SW domain, by performing a statistical analysis of CC4CS evaluated for two famous processors: 8-bit microcontroller (ASP) Intel 8051 [REF] and 32 bit RISC (GPP) LEON3 [REF]. The final goal is to evaluate proposed metric meaningfulness and accuracy..

2 Definition of CC4CS

The proposed metric is related to C programming language statements so it is called CC4CS (Clock Cycles for C Statement). The choice of the C language is motivated by the following three reasons: it is the most used language for embedded SW development; it is very similar to SystemC [REF], one of the most used specification languages for HW/SW Co-Design (in particular, when focusing on SystemC Synthesizable Subset, it is likely that the results related to the C language are still valid); the most diffused HLS (High Level Synthesis) tools are able to realize SPPs that implements an algorithm specified in C/SystemC language. So, CC4CS (independently from the specific processor technology) is defined as follow:

Def. For a given processor X, CC4CS(X) is the average number of clock cycles needed to processor X to execute a generic C statements

A first clarification is due with respect to the concept of “generic C statement”: in this work it is generally intended as “something that ends with a semicolon”. However, it is possible to refer also to other views (e.g. [REF]) or directly to the way in which some of the tools used to evaluate CC4CS itself consider a C statement. As described later, from a practical point of view, this work refers to the way a very common profiling tool as GCOV [REF] performs the statements counting.
Another clarification is related to the fact that such a metric will be for sure influenced by the used compiler or HLS tool. Some ways to manage this issue could be: to specify

also the used tools (possibly giving rise to a set of CC4CS for each processor); to report the average of the results obtained by using the most diffused tools; to report only the results related to the most diffused one. At this point, it is clear that CC4CS, as defined above, will be affected by relevant errors, but this can be acceptable by keeping in mind the following aspects: it is the only way to have a MIPS-like Off-the-Shelf metric; it can be applied to every Processor Technologies; it is intended to be used for very early performance analysis; different (more costly) approaches are possible later by focusing only on the most interesting processors. Moreover, as de-scribed in the next sections, CC4CS can be also characterized by a set of values related to min, max, and standard deviation (or by a statistical distribution).

It is worth noting that, the goal of this work, is to obtain some starting results to reason about, and to evaluate their quality, while trying to avoid as much as possible any offline/online analysis of the statements composing the given C functions. Other approaches, based on the possibility to properly characterize a C function to improve CC4CS, are for sure of interest but will be considered in future works as an opportunity to obtain more accuracy at more cost. Finally, it shall be also very clear that it is important trying to avoid any reasoning about the possible assembly code related to C statements since it could affect the "HW/SW unifying" feature of the metric. In fact, it could prevent to firstly evaluate CC4CS for SPP generated by means of High Level Synthesis techniques since they don't rely on assembly instructions execution. However, more detailed analysis are still possible. For example, a more costly unifying approach is presented in [REF] while not-unified analysis can be introduced in the co-design flow after the system-level, e.g. after the HW/SW partitioning step. At that point, once the design space has been greatly reduced, could be also feasible to really compile/synthesize functions for possible targets to obtain very accurate data; an approach that is not feasible at system-level, when possible processor technologies are simply too many to be managed in such a way.

3 A Framework for CC4CS Evaluation

Starting from the definition provided before, it is clear that to evaluate CC4CS for a given processor there are needed several steps, so depicting a methodology that shall be supported by proper tools to automate the process. In fact, considering a single C function, CC4CS is the ratio between the number of clock cycles required by the target processor to run the function and the number of executed C statements (1).

$$CC4CS = \frac{Clock\_cycles\_required}{Number\_of\_C\_Statements\_executed} \quad (1)$$

So, to make the metric meaningful it is needed, at least, to: define a set of relevant C functions to be used as a benchmark for all the processor technologies; for each benchmark function to identify a way to stimulate it by means of relevant inputs data sets; to identify a tool to perform profiling in order to count the number of executed C statements for each input; to identify tools to compile/synthesize the C function for the target processor and to simulate its execution in order to obtain the number of needed clock cycles. Naturally, such steps have to be applied for each different processor considered in the co-design flow. However, it is worth noting that it is an offline one-shot task since CC4CS, once evaluated, will be available "for free" for next projects that rely on the same processors. So, to support CC4CS evaluation, a proper framework has been realized. Additionally, such a framework is also able to evaluate statistics on the metric. By analyzing the data, it is so possible to validate the metric with respect to the performance of a target processor. Summarizing, such a frame-work allows to easily evaluate CC4CS in a repeatable manner. The following sections summarize the main features of the generic implemented framework while focusing more on the parts dependent by the processors considered in this work: 8-bit micro-controller (ASP) Intel 8051 [REF] and 32 bit RISC (GPP) LEON3 [REF].

3.1 Input generation

In order to validate the framework, a module that generates inputs automatically for a given benchmark has been created. At this step, the benchmark is composed by some algorithms that has been used in order to evaluate in the right manner the CC4CS.

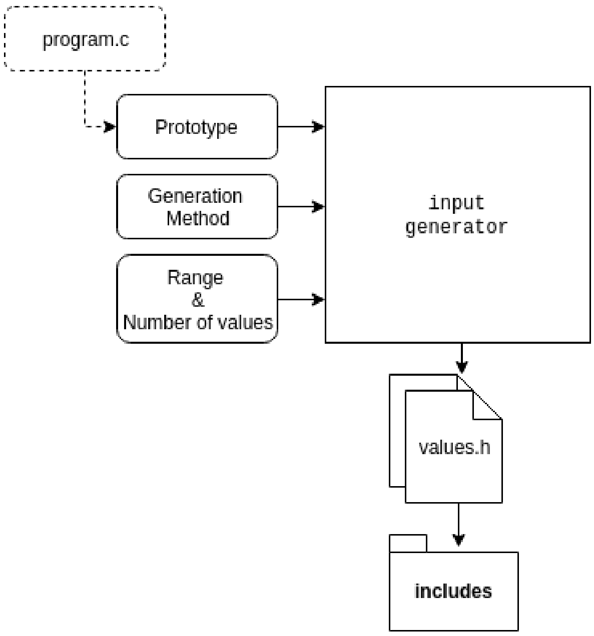


Fig. 1. Overview on inputs generation process

The benchmark will be presented in the next Section. So the CC4CS measurement method has been done with all the function and relative input generated. Each input were stored in a header file that was included in the function at execution time.

The module needs to know what kind of parameters the function requires. For this purpose, the programmer must define the prototype of the implemented function. The prototype contains the function name and the name and type of each parameter. The input generator parses the prototype file to found its name and to find out proper data for the function. For each parameter, the user is asked to insert a values range (min and max, to cover the large data size range as possible) and then the number of values that will be generated for this. The created values will have value between min and max numbers indicated in the range.

The generation of values may be done using two different approaches: a *random* approach or a *mixed* one. The first trivially consist in random generation of every value. The second one, is partially systematic. In this case, for variables that aren't array was calculated a *step* with a specific formula (2). The number_of_values variable specified in the formula is the same number inserted by the user.

$$\frac{1}{Number\_of\_values\ -1} \tag{2}$$

Starting from min , the step is incremented by 1 until the values of an evaluated expression (3). This kind of approach was defined partially systematic because in case of an array, the values will be generated randomly.

$$X = \{\min + (\max - min) * (step + 1)\ \forall i \mid min < X < max\} \tag{3}$$

In both approaches, in case of function that requires more than one variable, the Cartesian product of generated values has been done for every parameters. For each combination produced will be created a header file that contains the values of a single combination. At the end, the input generator creates the directory that contains every header file. An overview of the whole process has been shown in Fig. 1. After this step, it needs a profiling method to find the correct number of clock cycles related to the number of statements C of the software application.

3.2 Profiling on the host architecture

After the inputs generation phase, a procedure to count the number of statement executed was needed. This value was obtained performing a profiling of the program. To have this task done, the GCov [REF] profiler has been used. First of all, the program was compiled using GCC \[REF] and *-fprofile-arcs* and *-ftest-coverage* compilation flags. These flags tell the compiler to generate additional information needed by GCov to make a correct profiling. The first flag allows the generation of a *.gcda* file that contains additional information for each branch of the program while the second one adds information to count the number of times a statement has been executed. Compilation will trigger the creation of a *.gcno* file and generate also the corresponding *.gcda* file. To complete the task, the gcov command has been executed. The profiling will be done correctly only if the above described files were generated and reachable. To obtain the number of C statements executed, a sum of the single timing numbers has been performed. Fig. 2 shows the profiling technique used in this work.

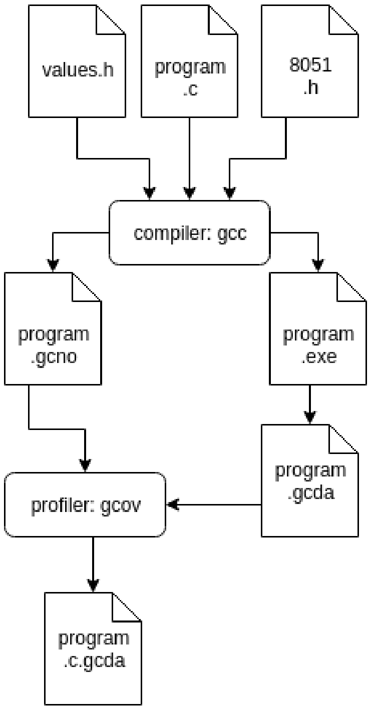


Fig. 2. Overview on profiling phase

3.3 Execution and metrics measure on target microprocessor

The last data to calculate CC4CS metric is the number of clock cycles used by the target microprocessor to execute the program. The execution has been done with a software simulation of the microprocessor using an Instruction Set Simulator (ISS).

- **Intel 8051 microcontroller.** In this work the 8051 microcontroller [REF]has been considered as first target platform. The Intel 8051 microcontroller is built around an 8-bit CPU. Architectural model used is the Harvard Architecture, and therefore it parts data and instruction by the use of two memories and two buses; indeed 8051 presents a PROM non-volatile memory which contains program instruction and a RAM memory for data, furthermore it presents an 8-bit Data Bus and a 16-bit Address Bus. I8051 registers are 8-bit registers. ALU works with 8-bit words and is provided with an accumulator register and communicates with four I/O 8-bit ports. The University of California has developed a project centered on 8051 microprocessor, which provides a number of tools useful for simulating C code on Intel 8051 microprocessor. The project name is Dalton and was developed by the *Dept. of computer Science of the University of California* [REF]. The Dalton Instruction Set Simulator (ISS) allows a user to simulate programs written for the 8051 and provides statistics on instructions executed, instructions executed per second, execution cycles required by the 8051, and average instructions per second for an 8051 executing

the same program. For this characteristics, it has been chosen as the reference ISS for the measurement of the CC4CS for 8051 microprocessor. The functions was compiled, with the SDCC (Small Device C Compiler) [REF] compiler. SDCC is free open source C compiler suite designed for 8 bit Microprocessors. The entire source code for the compiler is distributed under GPL and has extensive language extensions suitable for utilizing various microcontrollers and underlying hardware.
The Dalton ISS needs a .hex to do the simulation. This kind of file was generated with SDCC. To do a proper simulation, during the compilation two options was specified: --mmcs51 and --iram-size 128. The first one refers to the family of the microprocessor while the second to the dimension of the internal ram. The compilation will generate an .ihx file that was converted to .hex file using the packihx command. At the end, was executed the ISS. It generates a file that contains information about the simulation. After the simulation, the framework is ready to calculate the metric and some statistics on all input generated for the functions. These calculations are made with a program that returns a two files containing metric values, for each input, and statistics on the sample. An overview of this phase has shown in Fig. 3.

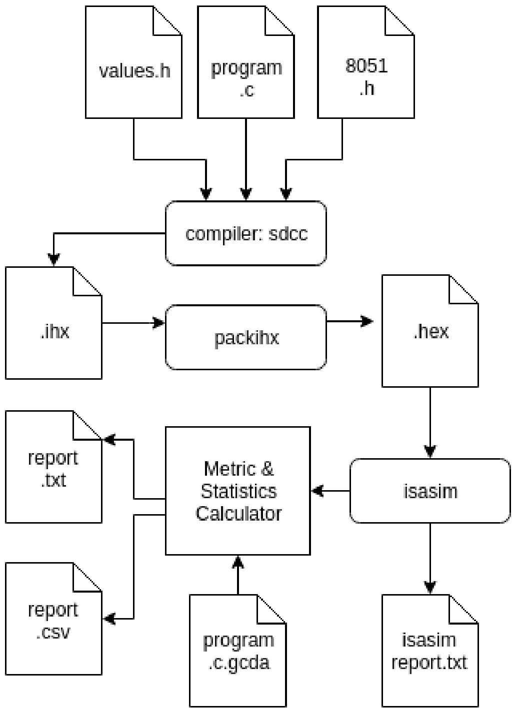


Fig. 3. Overview on execution phase on 8051

LEON3 Microprocessor. The second target platform is the LEON3 microprocessor. LEON3 [REF] is a 32-bit synthesizable soft-processor that is compatible with

SPARC V8 architecture: it has a seven-stage pipeline and Harvard architecture, uses separate instruction and data caches and supports multiprocessor configurations. It represents a soft-processor for aerospace applications. *Cobham Gaisler* offers *TSIM System Emulator* as an accurate emulator of LEON3 processors. A free evaluation version of TSIM/LEON3 is available on Gobham website [REF], but it does not support code coverage, configuration of caches, memories and so on. Anyway, it has been chosen as the reference ISS for first analysis since it provides the information needed to evaluate CC4CS. The LEON3 version has a default simulated system clock of 50 MHz. The evaluation version of TSIM/LEON3 implements 2*4 KiB caches (not removable), with 16 bytes per line with *Least-Recently-Used* (LRU) replacement algorithm. It has 8 register windows, a RAM size of 4096 KiB and a Rom size of 2048 KiB. By default, TSIM/LEON3 emulates the FPU. Benchmark functions have been compiled, with the *Bare-C Cross-Compiler* (BCC) for LEON3 processors [REF]. It is based on the *GNU* compiler tools and the *Newlib* standalone C-library. BCC is composed by *GNU GCC C/C++ compiler 4.4.2* [REF], *GNU Binutils 2.19.51* [REF] and *Newlib C-library 1.13.1* [REF]. After the simulation, the framework is ready to calculate the metric and some statistics by considering all the inputs used to stimulate the functions. An overview of this phase is shown in Fig. 4.

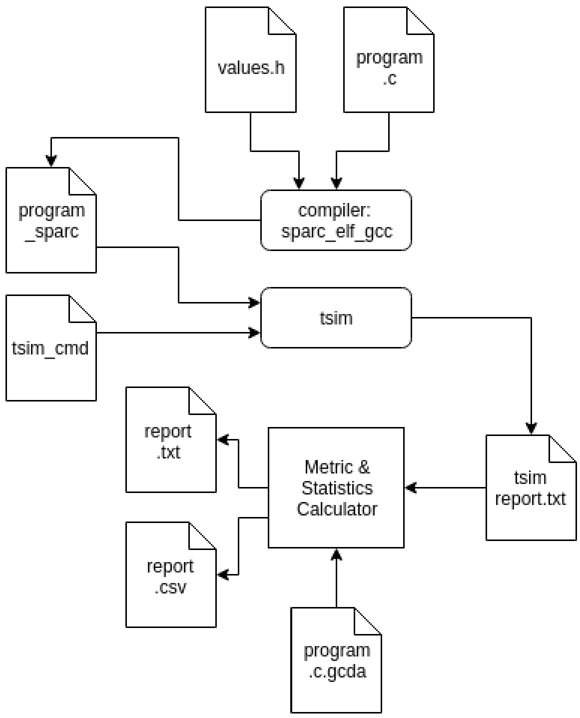


Fig. 4. Overview on execution phase on LEON3

4 Evaluation of CC4CS in the SW Domain – PART I

To validate the CC4CS metrics some tests has been executed. A benchmark composed by 10 algorithms has been used (i.e. C functions) in order to evaluate in the right manner the CC4CS. The functions which compose the benchmark and their brief description has been listed below:

- *Quicksort*: the sorting algorithm that follows the divide et impera approach. The algorithm recursively divides the input array until many small 1-length arrays was obtained. An array and its length have been passed as parameters.
- *Mergesort*: another sorting algorithm that follow the divide et impera approach. This also divides the input array in subsequences. This time we suppose that the subsequences are already sorted and so it's enough to choose always the minimum value between two subsequences that are comparing.
- *Matrix Multiplication*: an algorithm that does the matrix multiplication multiplying rows by columns.
- *Kruskal*: used to find the minimum spanning tree of a non-oriented graph that does not contains negative edges. It is a greedy algorithm and, in this case, the greedy choice consists in taking always the edge with minimum cost between among those available.
- *Floyd-Warshall*: calculates the distances between all pairs of vertices of a weighed graph with no negative loops. The costs of the edges may be negative value as long as these are not part of a negative loop.
- *Dijkstra*: calculates the minimum paths from a starting node x towards all nodes accessible by x. The graph must be oriented, can contain loops and must have edges with positive costs. This algorithm uses the concept of relaxation in order to obtain distances.
- *Breadth First Search* and *Depth First Search*: two algorithms for traversing a graph. In the first function the nodes that must be visited are inserted in a queue while in the second one in a stack.
- *Banker's Algorithm*: used in the operating systems to avoid deadlock situations during the allocation of resources to a process.
- *A**: a graph-searching algorithm that identifies a path from an initial node x to a final node y. Is similar to the dijkstra algorithm that for each node takes into account all possible directions and then chooses the one with lower cost. A* avoids to visit all edges connected to a node using a heuristic function that estimates the cost to the destination node.

4.1 CC4CS evaluation and analysis on 8051

Some results has shown in Table 1 and Table 2. The metrics was evaluated respect to 1.000 input files per function (with a total amount of result equals to 10.000) and 10.000 per function (with a total amount of result equals to 100.000).

Table 1: CC4CS MEASURED USING 1.000 INPUT DATA SET PER FUNCTION (10.000 EXECUTION) ON 8051

Method	Min	Max	AM	SD	Var	GM	GSD	85%	90%	95%	SE	RSE*
int8 t random	59	375	117,5112	44,9216	2020	110,7159	1,3964	165	170	176	1,42	0,01
int8 t mixed	59	473	120,9421	59,6024	3550	111,6812	1,4472	163	171	190	1,884	0,02
Int16 random	82	493	162,0144	64,8757	4210	151,0983	1,4393	213	267	297	2,051	0,01
Int16 mixed	82	537	163,5745	69,1658	4780	151,9858	1,4479	210	249	303	2,187	0,01
int32_t random	106	473	223,0118	87,569	7670	207,0883	1,4672	332	345	402	2,769	0,01
int32_t mixed	106	534	224,5114	86,9124	7550	209,051	1,456	331	343	402	2,748	0,01
float random	4	1322	526,5646	271,6848	73800	457,8767	1,8494	704	975	1198	8,591	0,02
float mixed	4	1324	548,8937	275,413	75900	480,5383	1,8135	845	984	1218	8,709	0,02

Table 2: CC4CS MEASURED USING 10.000 INPUT DATA SET PER FUNCTION (100.000 EXECUTION) ON 8051

Method	Min	Max	AM	SD	Var	GM	GSD	85%	90%	95%	SE	RSE*
int8 t random	58	410	117,8384	47,451	2250	110,7924	1,3991	160	170	176	1,5	0,01
int8 t mixed	58	472	118,4134	50,9183	2590	110,8222	1,4115	161	170	176	1,61	0,02
Int16 random	80	453	161,3708	67,5097	4560	149,8673	1,452	217	265	297	2,134	0,01
Int16 mixed	80	594	167,364	70,0199	4950	155,2539	1,4595	220	271	300	2,214	0,01
int32_t random	104	760	227,9238	88,6997	7870	211,5321	1,473	338	354	400	2,804	0,01
int32_t mixed	104	723	227,4784	88,9847	4900	211,1882	1,4698	336	354	401	2,813	0,01
float random	4	1301	537,6769	267,6368	71600	466,4769	1,9073	818	969	1173	8,463	0,02
float mixed	4	1301	544,8065	267,0008	71300	479,102	1,7971	827	977	1187	8,443	0,02

*AM: Arithmetic Mean, GM: Geometric Mean, SD: Standard Deviation, GSD: Geometric Standard Deviation, SE: Standard Error, RSE: Relative Standard Error

For the single functions, different variables data type has been considered (int8, int16, int32, and float) because the performance of each software change respect to the dimension of data since the microcontroller is based on a 8-bit CPU, a 8-bit ALU and a complex instruction set computer (CISC) instruction set.

Furthermore, with float data type the values of CC4CS is increasing respect to the other value about the lack of FPU and the HW architecture registers size. The frequency graph is shown in Fig. 5, Fig. 6, Fig. 7 and Fig. 8. From the plot it is possible to see that the spread of bars is increasing and distributed in a certain specific mode. In particular, changing the data type, the minimum and the mean data center moves to the right and decrease the maximum frequency respect to bins range. This kind of distribution is under investigation about University of L'Aquila.

ULTERIORE ANALISI DATI.

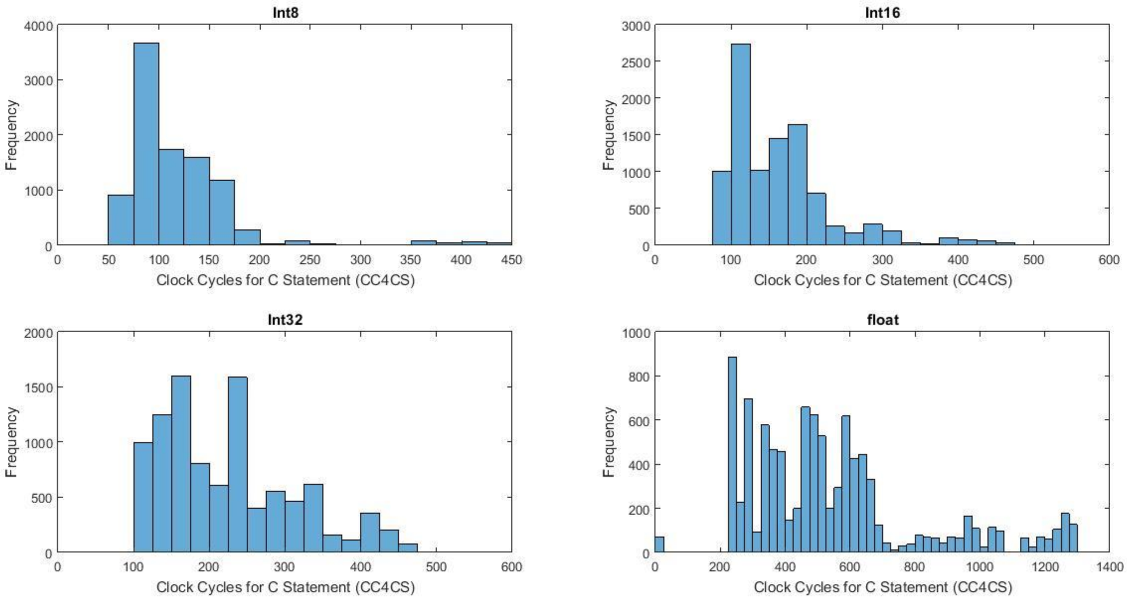


Fig. 5. CC4CS Frequency for processor 8051, method **mixed** and **10.000** total execution.

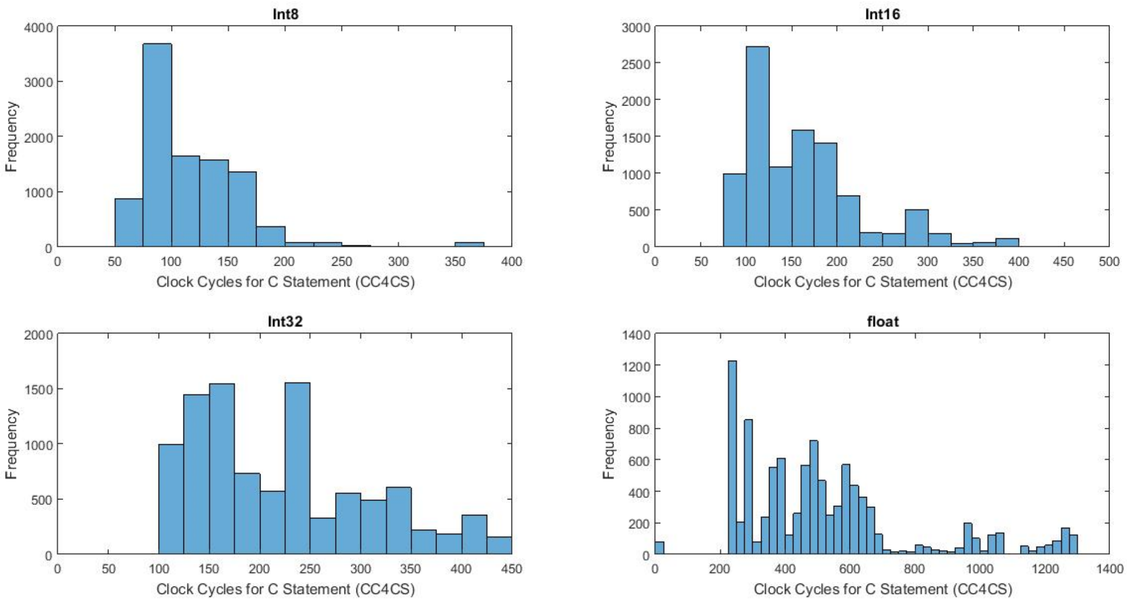


Fig. 6. CC4CS Frequency for processor 8051, method **random** and **10.000** total execution.

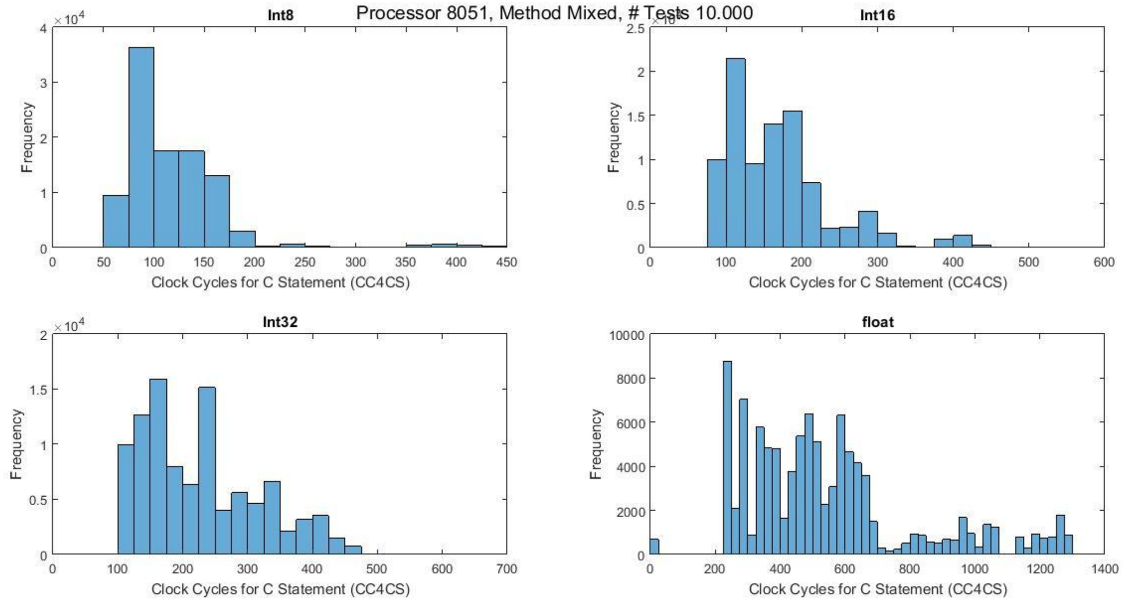


Fig. 7. Frequency of CC4CS for processor 8051, method **mixed** and **100.000** total execution.

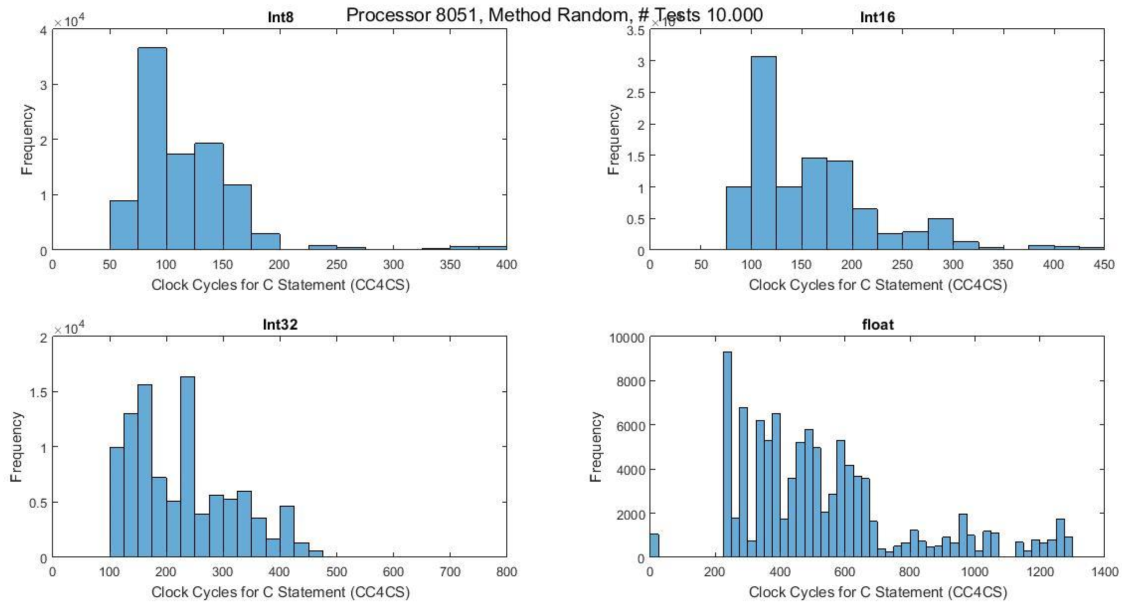


Fig. 8. Frequency of CC4CS for processor 8051, method **random** and **100.000** total execution.

4.2 CC4CS evaluation and analysis on LEON3

Some results has shown in Table 3 and Table 4. The metrics was evaluated respect to 1.000 input files per function (with a total amount of result equals to 10.000) and 10.000 per function (with a total amount of result equals to 100.000).

Table 3: CC4CS MEASURED USING 1.000 INPUT DATA SET PER FUNCTION (10.000 EXECUTION) ON LEON3

Method	Min	Max	AM	SD	Var	GM	GSD	85%	90%	95%	SE	RSE*
int8 random	11	2197	193	304	9,26E+04	90	3,3565	371	536	721	9,6244	5%
int8 mixed	11	2197	212,2466	395,8797	1,57E+05	88,4669	3,4652	345	493	722	12,518	6%
int16 random	12	2194	291,9572	401,5222	1,61E+05	149,1118	3,2279	537	644	1322	12,697	4%
int16 mixed	12	2196	290,6082	419,9791	1,76E+05	145,9891	3,2232	545	690	887	13,280	5%
int32 random	23	2194	437,1239	512,0694	2,62E+05	258,8093	2,8036	786	1047	2053	16,193	4%
int32 mixed	22	2194	418,3451	491,9029	2,42E+05	255,9912	2,6739	691	864	2053	15,555	4%
float random	28	2200	481,7072	516,9927	2,67E+05	305,9818	2,5861	817	1326	2058	16,348	3%
float mixed	28	2200	440,3998	502,9913	2,53E+05	282,065	2,5035	693	867	2058	15,905	4%

Table 4: CC4CS MEASURED USING 10.000 INPUT DATA SET PER FUNCTION (100.000 EXECUTION) ON LEON3

Method	Min	Max	AM	SD	Var	GM	GSD	85%	90%	95%	SE	RSE*
int8 random	11	2197	186	321	1,03E+05	85	3,3224	333	447	620	10,164	5%
int8 mixed	11	2197	185,9409	328,8503	1,08E+05	85,3025	3,2828	334	426	585	10,399	6%
int16 random	12	2196	291,6597	396,273	1,57E+05	153,5396	3,1886	529	631	1047	12,531	4%
int16 mixed	12	2196	276,5413	407,2217	1,66E+05	140,2027	3,1897	512	594	819	12,877	5%
int32 random	22	2194	441,6121	521,7224	2,72E+05	258,255	2,8203	726	1277	2053	16,498	4%
int32 mixed	22	2194	450,7706	517,624	2,68E+05	275,356	2,6992	721	907	2193	16,368	4%
float random	26	2200	484,2454	535,8412	2,87E+05	304,0376	2,6084	781	1326	2058	16,944	3%
float mixed	27	2200	468,1861	519,6861	2,70E+05	301,6297	2,5148	723	1045	2198	16,433	4%

*AM: Arithmetic Mean, GM: Geometric Mean, SD: Standard Deviation, GSD: Geometric Standard Deviation, SE: Standard Error, RSE: Relative Standard Error

ANALISI DATI DA FARE.

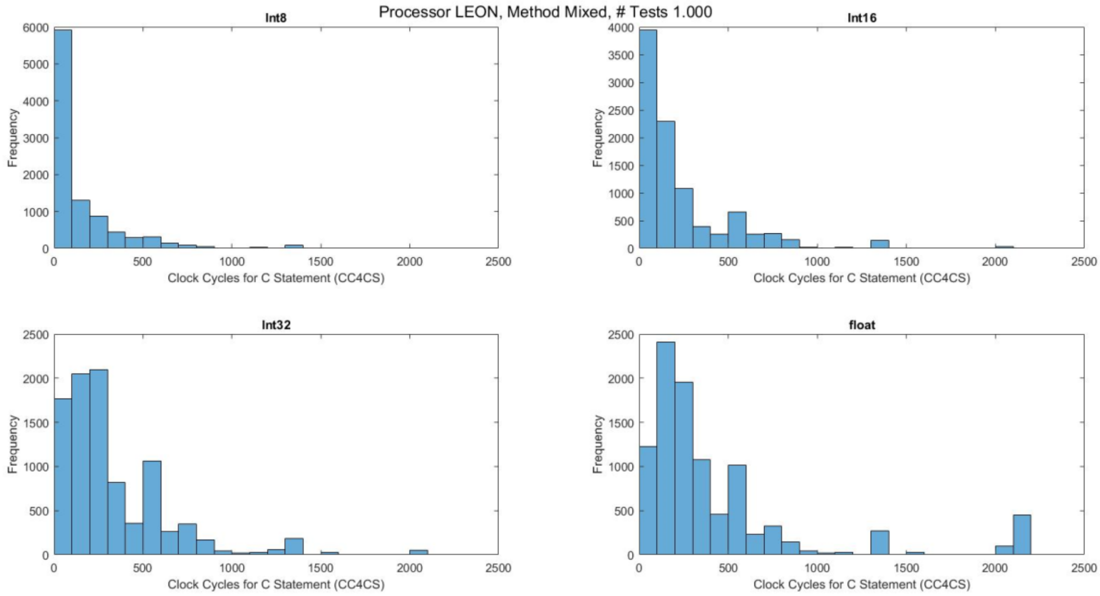


Fig. 9. CC4CS Frequency for processor LEON3, method **mixed** and **10.000** total execution.

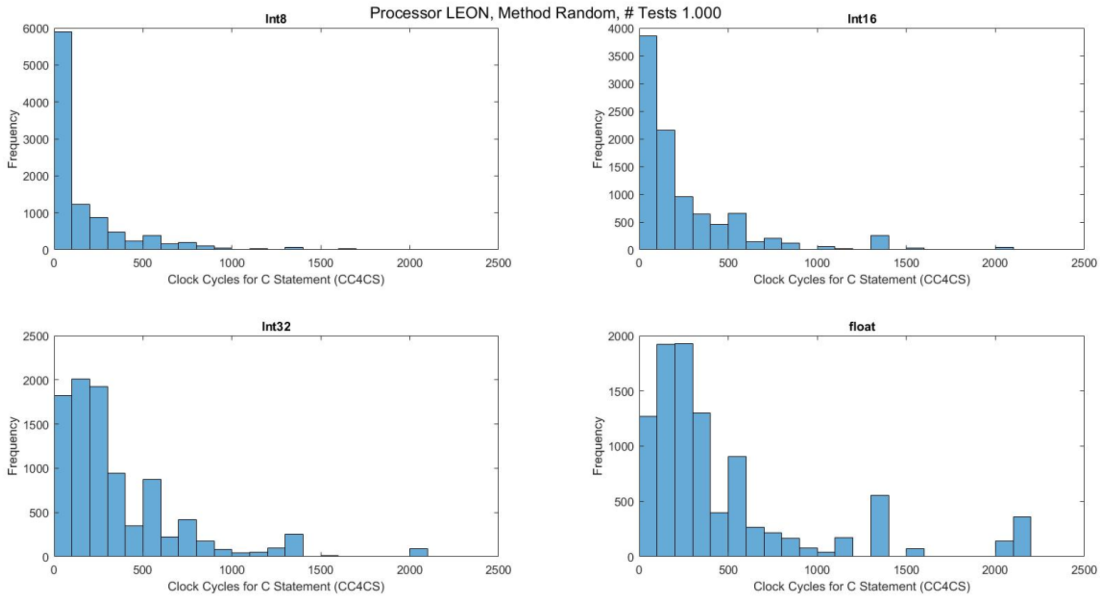


Fig. 10. CC4CS frequency for processor LEON3, method **random** and **10.000** total execution.

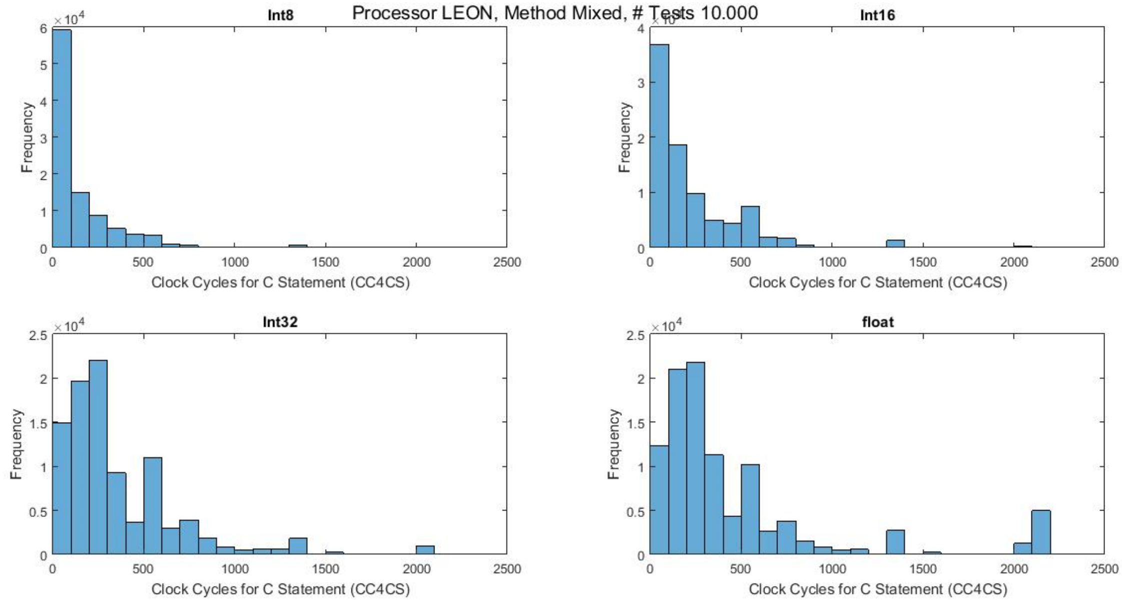


Fig. 11. CC4CS frequency for processor LEON3, method **mixed** and **100.000** total execution.

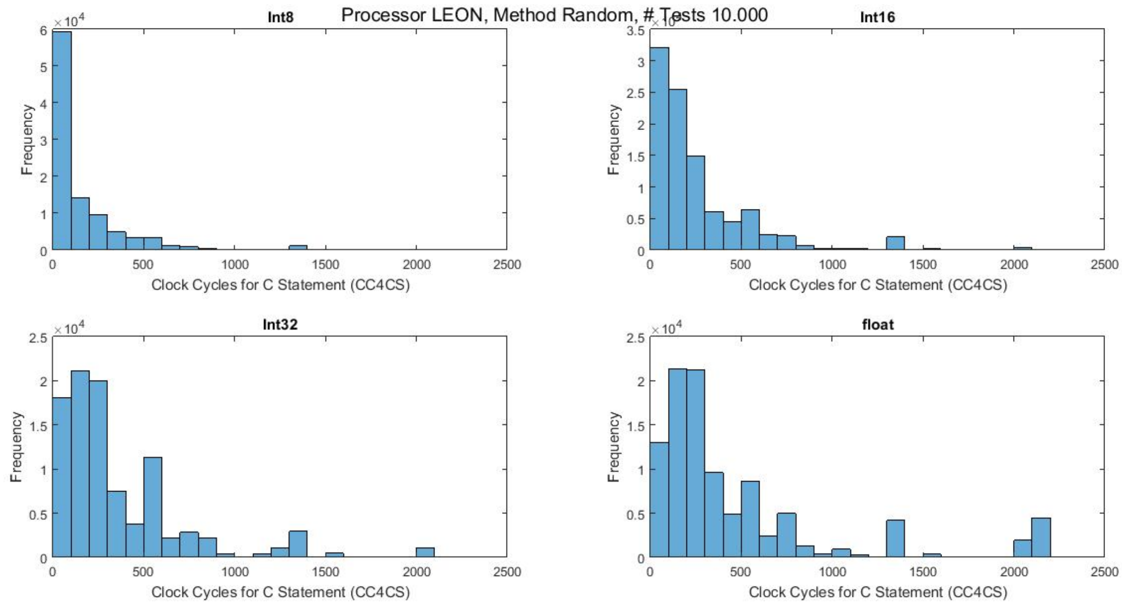


Fig. 12. CC4CS frequency for processor LEON3, method **random** and **100.000** total execution.

5 CC4CS Validation

To validate the CC4CS metric, a set of 5 other function was used to evaluate the errors related to such kind of estimation. The framework has been used to calculate the real execution time and the number of statements C of the single function and, then, the results has been analyzed offline. This functions and their brief description has been listed below:

- *Selection Sort*: divides the input list into two parts, the subset of items already sorted, and the subset of items remaining to be sorted that occupy the rest of the array.
- *Insertion Sort*: builds the final sorted array one item at a time.
- *GCD*: the classical greatest common divisor algorithm.
- *Binary Search*: finds the position of a target value within a sorted array.
- *Bellman Ford*: computes shortest paths from a single source vertex to all of the other vertices in a weighted graph.

5.1 CC4CS Validation on 8051

Starting from this functions, some statistics are shown in Table 5. The values of CC4CS are taken from Table 1 and Table 2 (the worst value respect to the specific data type). The statistics refers to the execution of all the functions considered for validating the CC4CS metric. The high value associated to float data type is related to the standard deviation of CC4CS, and its standard error, that is much bigger than the other data types (the distribution of the data occurrence makes the percentile interval wider than others).

Table 5: ERROR STATISTICS CONSIDERING EXTERNAL TARGET FUNCTION

Method	RMSE AM	PRMSE AM	RMSE GM	PRMSE GM	Min - 85%	Min - 90%	Min - 95%	Min - Max
int8 random	2.90 ms	42.2%	2.50 ms	37.4%	9.38%	8.92%	5.74%	0.00%
int8 mixed	2.99 ms	41.8%	2.55 ms	36.5%	6.54%	6.30%	4.20%	0.00%
int16 random	1.35 ms	22.9%	1.43 ms	21.9%	18.12%	4.60%	4.40%	0.00%
int16 mixed	2.65 ms	23.1%	2.16 ms	21.4%	17.08%	1.52%	1.24%	0.00%
int32 random	1.84 ms	21.0%	1.99 ms	24.3%	12.92%	10.00%	7.20%	0.00%
int32 mixed	2.03 ms	22.3%	2.18 ms	25.1%	14.06%	12.00%	10.00%	0.00%
float random	1.05 ms	76.6%	0.92 ms	60.2%	0.94%	0.00%	0.00%	0.00%
float mixed	1.10 ms	79.2%	1.01 ms	62.5%	3.20%	0.00%	0.00%	0.00%

* RMSE: Root Mean Squared Error, PRMSE: Root Mean Squared Percentage Error, AM: Arithmetic Mean, GM: Geometric Mean

Fig. 13 shown four target function, with all data types set to int8, with their real timing execution compared to MIN and 95 Percentile CC4CS interval lower and upper bound, and the arithmetic mean, geometric mean and 85 percentile line plot. Other analysis and consideration are under investigation, but from this preliminary result it is possible to say that the CC4CS, whereas should be compliant with specific data types, allows to predict, with a relative low margin of error, the execution time of C function on specific HW platform, with a less effort and a quickly, repeatable and smart measurement flow based on a robust framework that is independent from the target software, hardware platform and final embedded implementation.

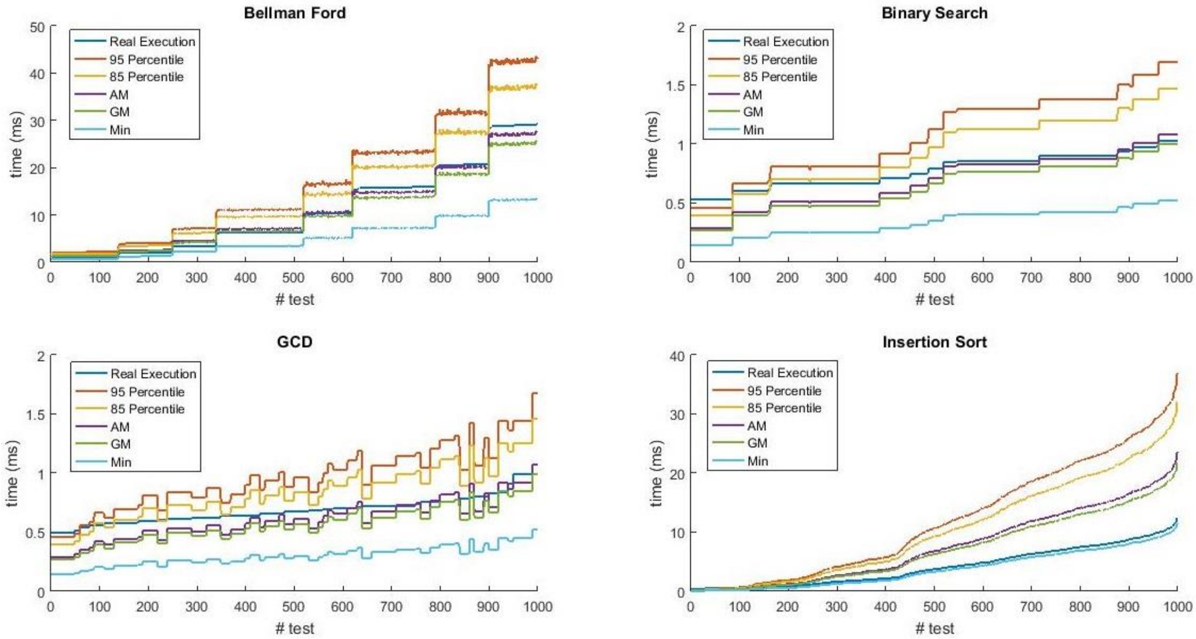


Fig. 13. Int8_t data type function validation of CC4CS time interval execution estimation on 8051

ALTRE ANALISI DA FARE.

5.2 CC4CS Validation on LEON3

Starting from this functions, some statistics are shown in Table 6. The values of CC4CS are taken from Table 3 and Table 4 (the worst value respect to the specific data type). The statistics refers to the execution of all the functions considered for validating the CC4CS metric. The high value associated to float data type is related to the standard deviation of CC4CS, and its standard error, that is much bigger than the other data types (the distribution of the data occurrence makes the percentile interval wider than others).

Table 6: ERROR STATISTICS CONSIDERING EXTERNAL TARGET FUNCTION ON LEON3

Method	RMSE AM	PRMSE AM	RMSE GM	PRMSE GM	Min - 85%	Min - 90%	Min - 95%	Min - Max
int8 random								
int8 mixed								
int16 random								
int16 mixed								
int32 random								

int32 mixed								
float random								
float mixed								

* RMSE: Root Mean Squared Error, PRMSE: Root Mean Squared Percentage Error, AM: Arithmetic Mean, GM: Geometric Mean

ALTRE ANALISI DA FARE.

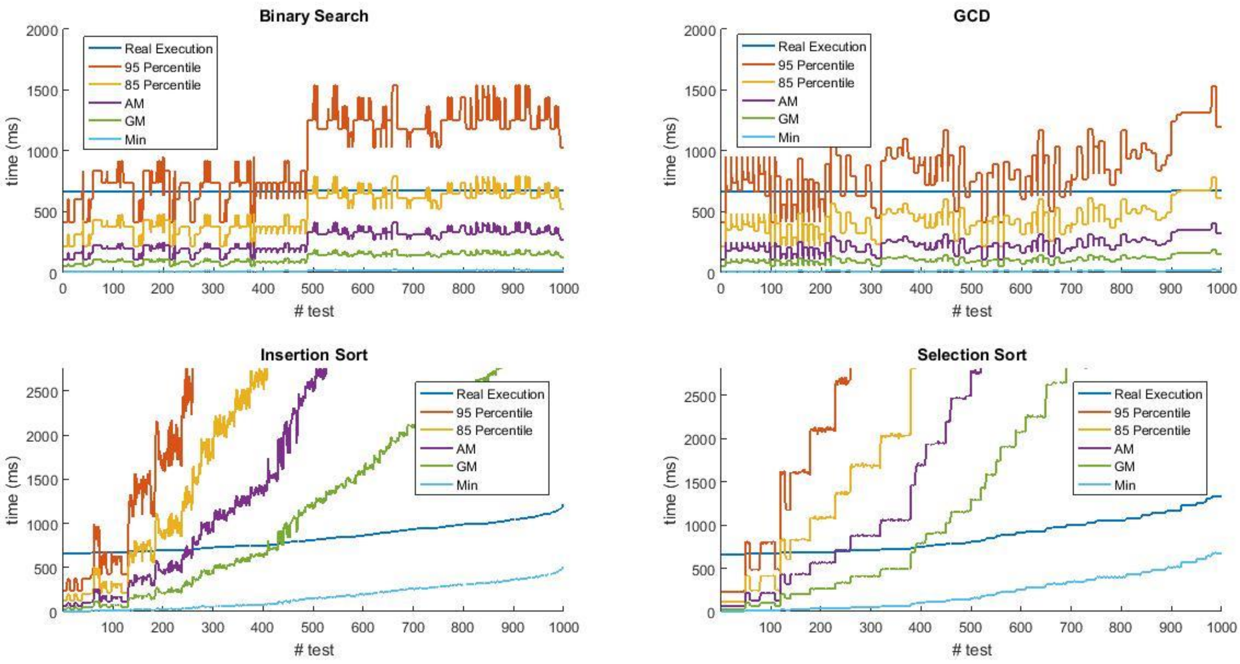


Fig. 14. Int8_t data type function validation of CC4CS time interval execution estimation on LEON3

ALTRE ANALISI DA FARE.

6 Evaluation of CC4CS in the SW Domain – PART II

To evaluate and analyze the CC4CS metric, some tests has been performed on a benchmark composed of 15 algorithms (i.e. C functions). Each benchmark function has been executed by using about 1.000 (with a total amount of run near to 14.907). Moreover, for each benchmark function, different data types have been taken into account (i.e. *int8*, *int16*, *int32*, and *float*) since, mainly depending on the internal architecture bit-width of the considered processor, the execution time is severely affected by data dimension. The functions composing the benchmark are: *Quicksort*, *Mergesort*, *Matrix*

Multiplication, Kruskal, Floyd-Warshall, Dijkstra, Breadth First Search, Depth First Search, Banker's Algorithm, A, Selection Sort, Insertion Sort, GCD, Binary Search*, and *Bellman Ford*.

6.1 CC4CS evaluation and analysis on 8051

Table 1 and Table 2 show the main results related to 8051. It is immediately possible to note that, with float data type, the values of CC4CS is considerably increased with respect to the other values. This is mainly due the 8051 bus/registers size and to the lack of a FPU.

The frequency graph (normalized to unit length) is shown in Fig. 3. From the plot it is possible to see that the spread of bars is increasing and distributed in a specific mode (it looks like a right-skewed distribution). In particular, by changing the data type, the minimum and the mean data moves to the right and increase the maximum frequency respect to bins range.

Table 7. CC4CS measured on 8051.

Data Type	MIN	85%	90%	95%	MAX
Int8_t	59	195	232	600	777
Int16	82	267	313	619	750
Int32	106	395	427	854	1016
Float	4	1032	1210	1631	2527

Table 8. CC4CS Statistics on 8051

Data Type	AM	SD	Variance	GM	GSD	SE	RSE[1]
Int8_t	155,2	139,3	1,94E04	125,5	1,766	1,14	0,7 %
Int16	200,6	134,1	1,79E04	173,8	1,631	1,10	0,5 %
Int32	290,6	178,6	3.19E04	253,1	1,646	1,46	0,5 %
Float	611,7	466,6	2,18E05	489,4	1,989	3,82	0,6 %

Fig. 3 shown also all the fitted valid parametric probability distributions to data. For this purpose, *Matlab Statistics Toolbox* [21] supports a long list of distributions. Starting from this tool, all valid parametric distributions have been fit to the data and sorted using *Bayesian Information Criterion* (BIC) metric, used also to compare the goodness of the fit.

[1] **AM**: Arithmetic Mean, **SD**: Standard Deviation, **GM**: Geometric Mean, **GSD**: Geometric Standard Deviation, **SE**: Standard Error, **RSE**: Relative Standard Error

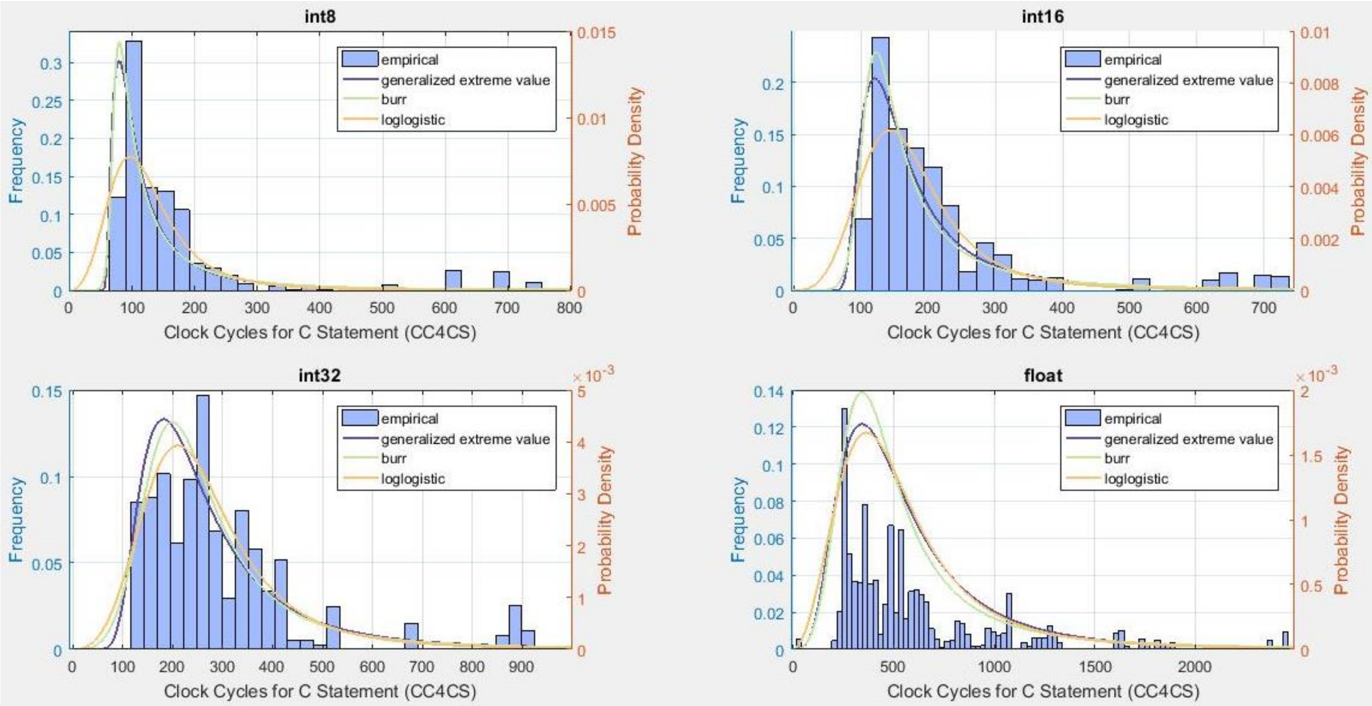
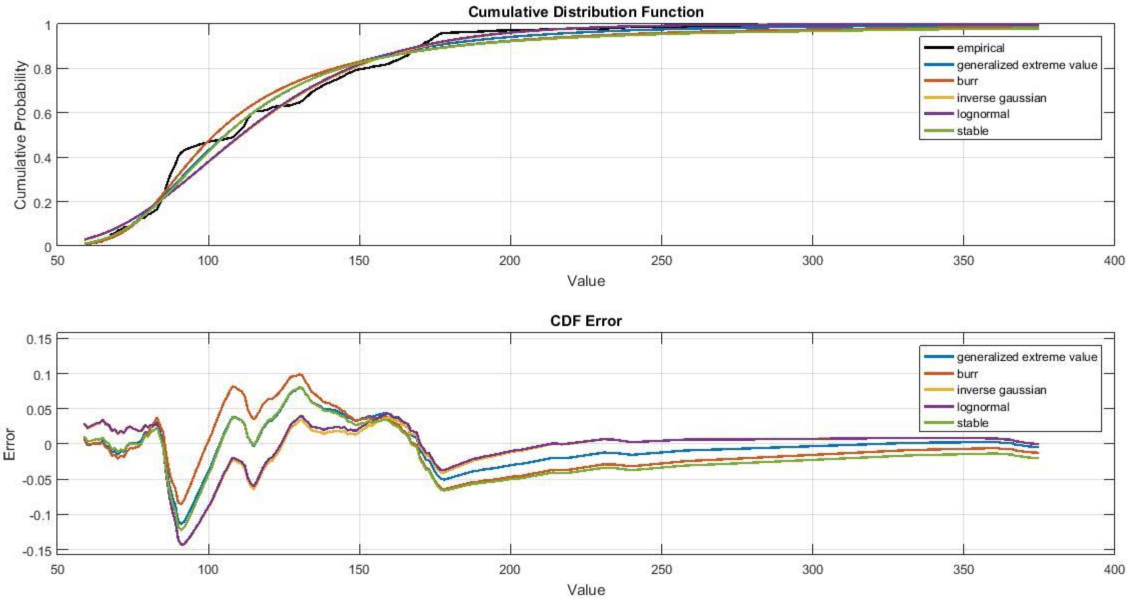


Fig. 3. Frequencies and Probability Distribution Functions of CC4CS on 8051

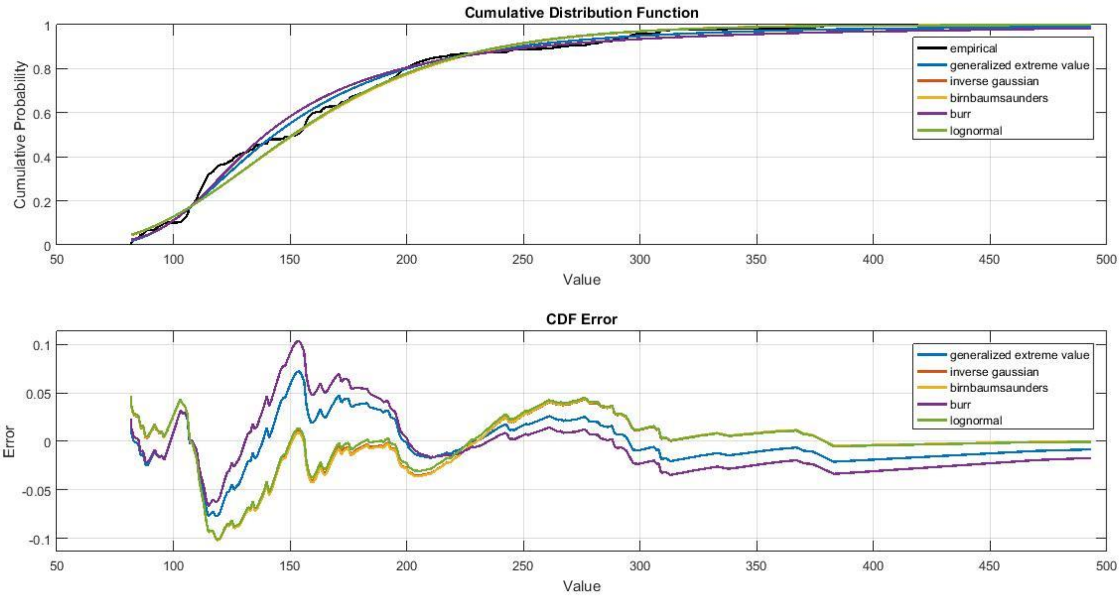
The 3 best continuous parametric distribution that fit to CC4CS data histogram are listed below:

- *Generalized Extreme Value* (GEV): developed within extreme value theory. This is often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. Its parameters are: int8 (k=0.6824; σ=34.0359; μ=93.2761), int16 (k=0.4584; σ=49.3502; μ=137.4673), Int32 (k=0.3203; σ=86.8787; μ=204.4474) and float (k=0.2864; σ=219.7836; μ=394.7265);
- *Burr Distribution*: for a non-negative random variable. The fitting parameter estimates are: Int8 (α=71.3167; c=14.4377; k=0.1200), Int16 (α=109.6685; c=9.4754; k=0.2145), Int32 (α=200.2810; c=4.5224; k=0.5787), Float (α=345.1090; c=3.8340; k=0.5080);
- *Log-Logistic Distribution*: for a non-negative random variable used for events whose rate increases initially and decreases later. The fitting parameters are: Int8 (μ=4.7532; σ=0.3040), Int16 (μ=5.1045; σ=0.2637), Int32: μ=5.5085; σ=0.2790, Float (μ=6.1639; σ=0.3591);

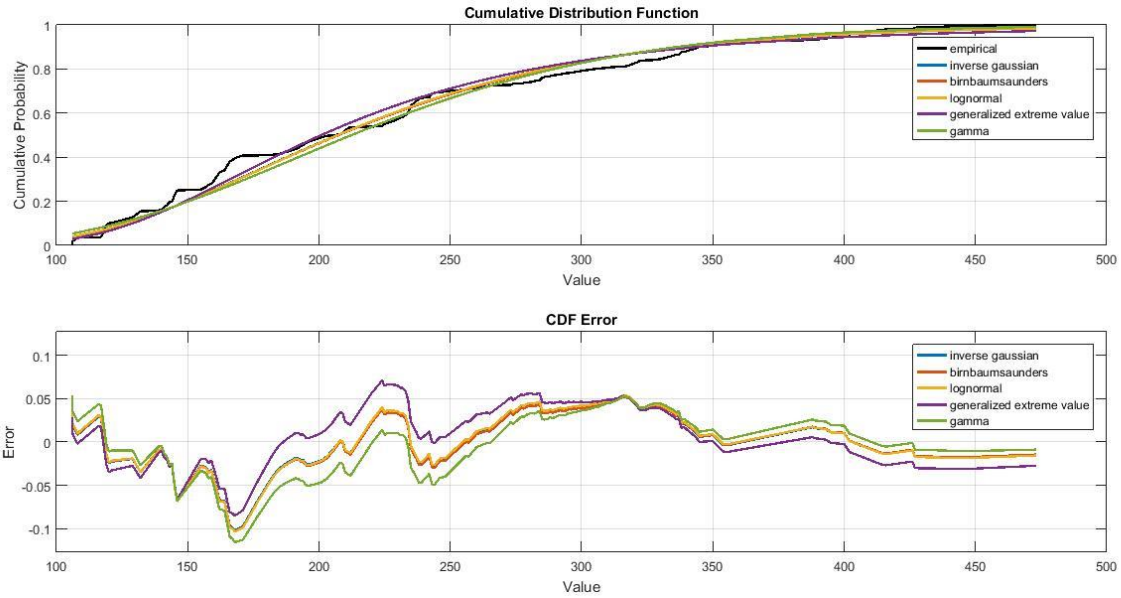
Fig. 4 shows the empirical *Cumulative Distribution Functions* (CDF) compared to all the fitted CDF. In this plot, it can be seen that the error corresponding to fitting distribution is under 5 % with respect to int8, int16 and int32 data types.



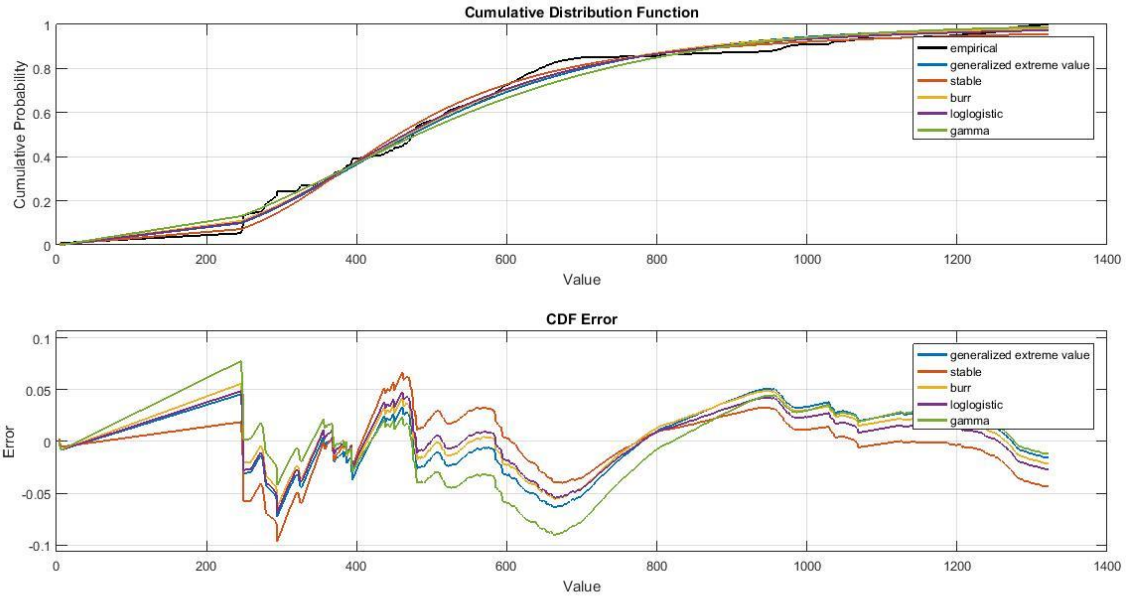
(a) Int8_t on 8051



(b) Int16_t on 8051



(c) Int32_t on 8051



(d) Float on 8051

Fig. 15. CDF on 8051 for int8 (a), int16 (b), int32 (c) and float (d) reference data type

6.2 CC4CS estimation and analysis on LEON3

Table 9 and Table 10 show the main results related to LEON3.

Table 9. CC4CS measured on LEON3.

Data Type	MIN	85%	90%	95%	MAX
Int8_t	11	926	1118	1276	2197
Int16	12	1070	1273	1515	2194
Int32	23	1277	1512	2053	2194
Float	4	1326	1524	2057	2200

Table 10. CC4CS Statistics on LEON3

Data Type	AM	SD	Variance	GM	GSD	SE	RSE[2]
Int8_t	323,6	452,6	2,05E05	124,6	4,098	3,71	1,2 %
Int16	429,5	516,2	2,66E05	206,5	3,561	4,23	1,0 %
Int32	596,5	578,9	3.35E05	354,6	2,976	4,74	0,8 %
Float	691,8	616,1	3,79E05	436,4	2,795	5,05	0,7 %

The frequency graph (normalized to unit length) and all the fitted valid parametric probability distributions are shown in Fig. 5. The behavior of the graph is similar to the 8051 plot (like a right-skewed distribution), but it is a bit more dispersed (maybe for cache presence). Further analysis will be done in future works. Anyway, the 4 best continuous parametric distribution to CC4CS data set are listed below:

- *Birnbaum–Saunders distribution* (BSD): used extensively in reliability applications to model failure times. The distribution has been characterized by: Int8 (β=142.1388; γ=1.6548), Int16 (β=207.0865; γ=1.4697), Int32 (β=339.0557; γ=1.2188), Float (β=417.5565; γ=1.1337);
- *Inverse Gaussian distribution*: also known as the Wald distribution, the inverse Gaussian is used to model non negative positively skewed data. The fitting parameter are: Int8 (μ =323.6872; λ=70.2918), Int16 (μ =429.4923; λ=129.1252), Int32 and Float not considered;
- *Generalized Pareto distribution*: often used to model the tails of another distribution. It has been characterized by: Int8 (k=0.9646; σ=97.2822; θ=11.0000), Int16 (k=0.4774; σ=245.6095; θ=12.0000), Int32 (k=0.0203; σ=561.9403; θ=23.0000), Float (k=-0.1777; σ=787.7083; θ=28.0000);

[2] **AM**: Arithmetic Mean, **SD**: Standard Deviation, **GM**: Geometric Mean, **GSD**: Geometric Standard Deviation, **SE**: Standard Error, **RSE**: Relative Standard Error

- *Log-Normal distribution*: distribution of a random variable whose logarithm is normally distributed and so it is closely related to the normal distribution. The parameters are: Int32 (μ =5.8710; σ=1.0906), Float (μ =6.0787; σ=1.0278), int8 and int16 not considered;

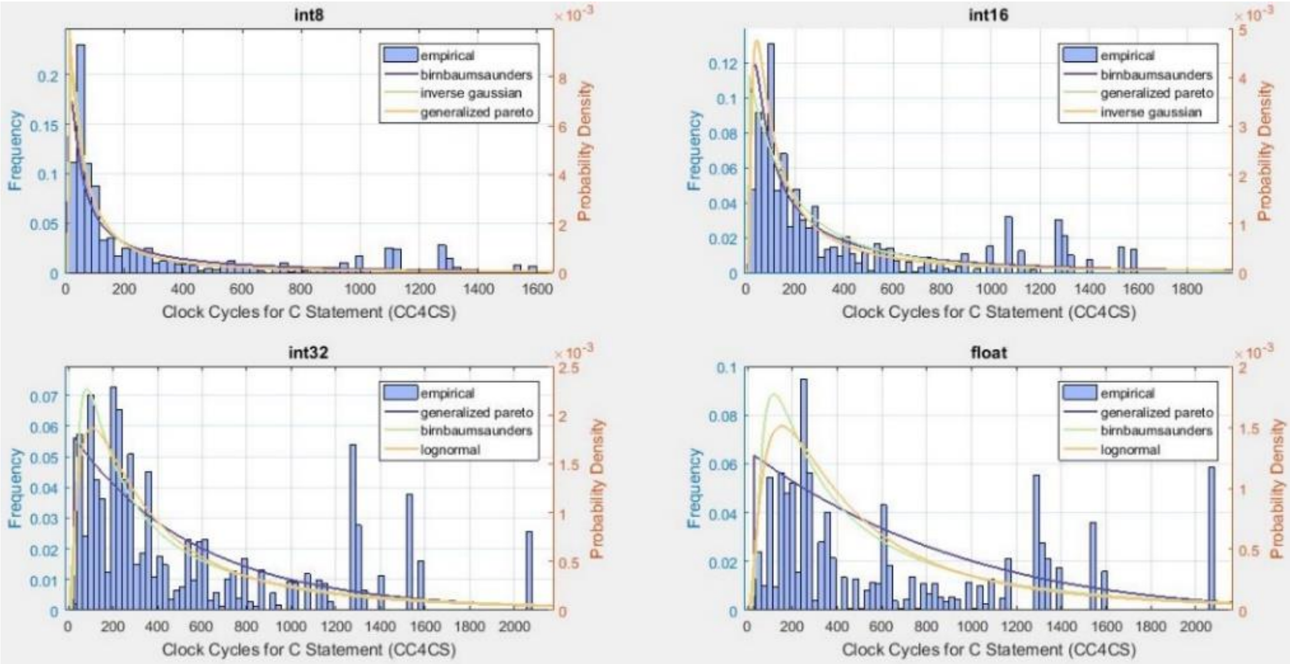
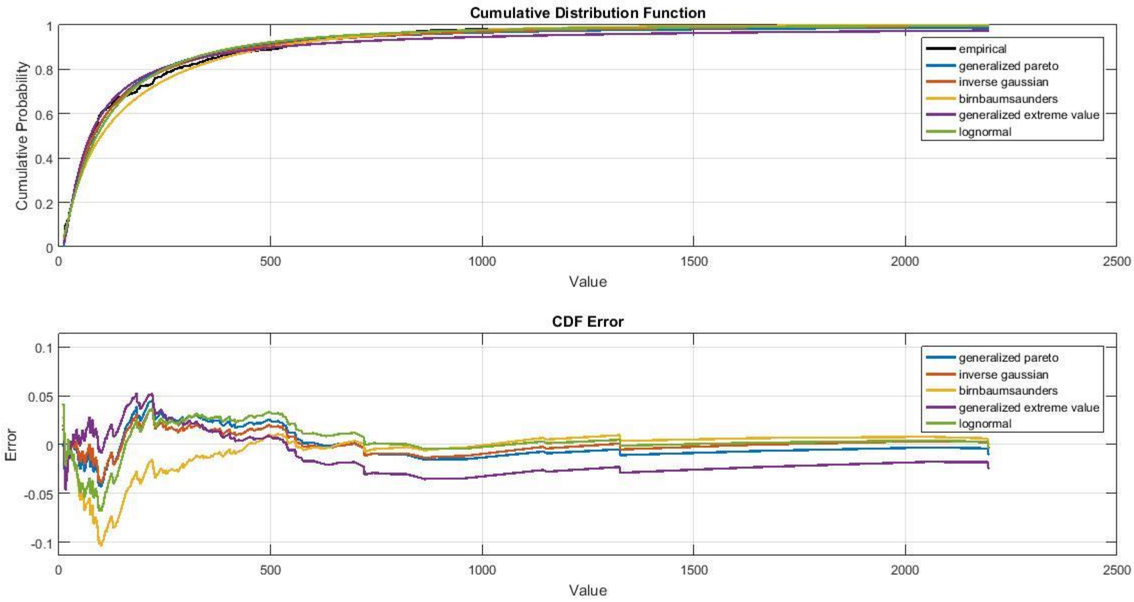
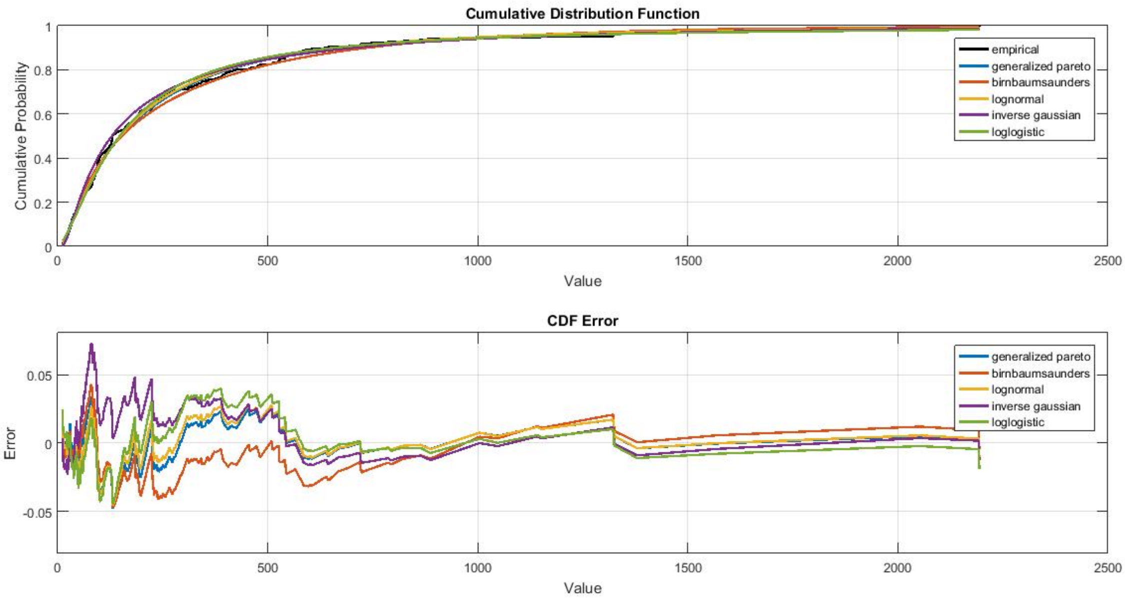


Fig. 16. Frequencies and Probability Distribution Functions of CC4CS on LEON3

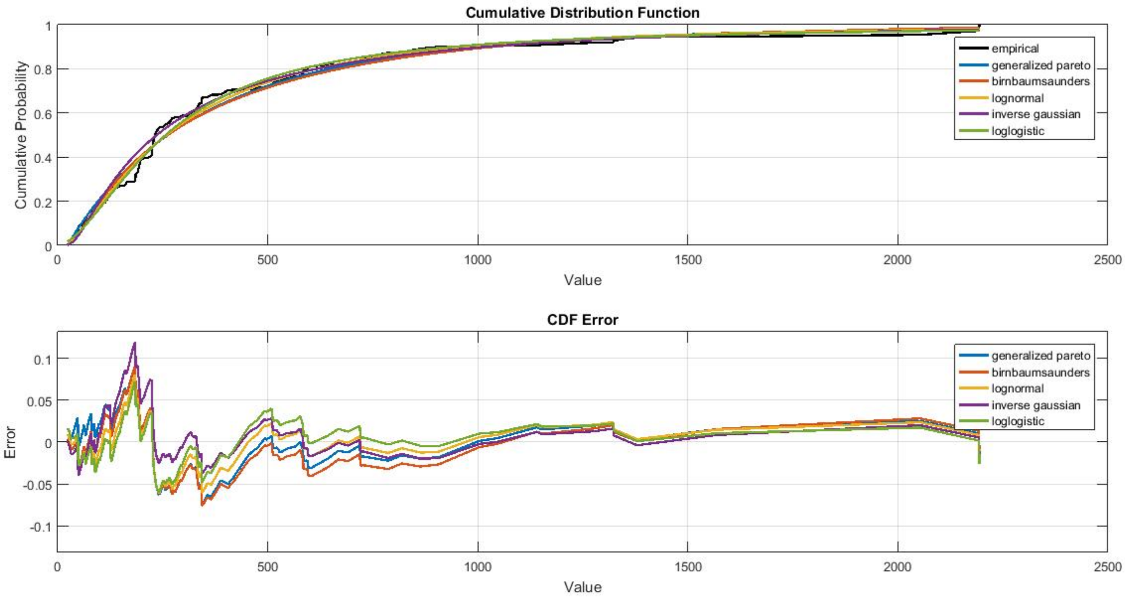
Fig. 6 shows the empirical Cumulative Distribution Functions (CDF) compared to all the fitted CDF. In this plot it can be seen that the error corresponding to fitting distribution is under 10 % with respect to reference data types. In Section V the precision of the estimation has been analyzed using standard error and confidence interval.



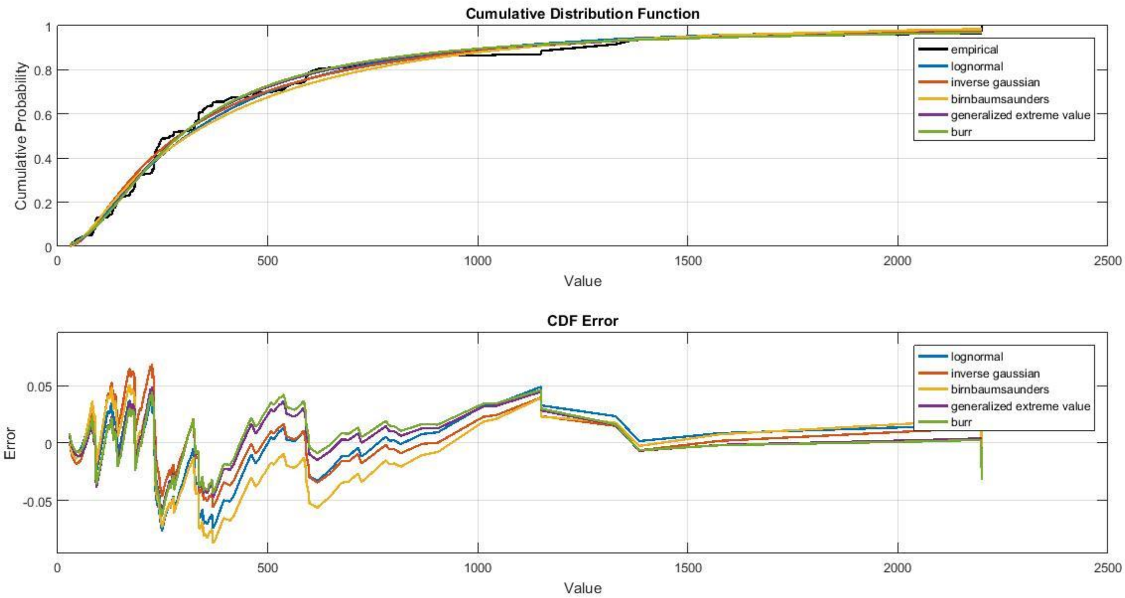
(a) Int8_t on LEON3



(b) Int16_t on 8051



(c) Int32_t on 8051



(d) Float on 8051

Fig. 17. CDF on 8051 for int8 (a), int16 (b), int32 (c) and float (d) reference data type

7 Statistical Analysis

Table 5 and Table 6 shown the Bayesian Information Criterion (BIC) metric used to evaluate the goodness of fit of the specific probabilistic distributions on 8051 and LEON3.

Table 11. CC4CS BIC goodness of fit on 8051.

Reference Data Type	Generalized Extreme Value	Burr	Log-Logistic
Int8_t	1.6139e+05	1.6157e+05	1.6669e+05
Int16	1.6987e+05	1.7012e+05	1.7269e+05
Int32	1.8421e+05	1.8471e+05	1.8507e+05
Float	2.1145e+05	2.1178e+05	2.1224e+05

Table 12. CC4CS BIC goodness of fit on LEON3.

Reference Data Type	Bimbaum Saunders	Inverse Gaussian	Generalized Pareto	Log-Normal
Int8_t	1.9317e+05	1.9326e+05	1.9361e+05	-
Int16	2.0583e+05	2.0667e+05	2.0660e+05	-
Int32	2.1757e+05	-	2.1756e+05	2.1829e+05
Float	2.2198e+05	-	2.2170e+05	2.2269e+05

To quantify the precision of the estimates, standard error has been first used. It has been computed from the asymptotic covariance matrix of the maximum likelihood estimators from the fitted probabilistic distribution. The standard error results are shown in Table 7 and Table 8 (for 8051 and LEON3 respectively).

Table 13. CC4CS fitting standard error estimation on 8051.

	Generalized Extreme Value			Burr			Log-Logistic	
Data Type	**k**	**σ**	**μ**	**α**	**c**	**k**	**μ**	**σ**
Int8_t	0.0092	0.3338	0.3126	0.2625	0.3456	0.0035	0.0041	0.0020
Int16	0.0082	0.4342	0.4656	0.5764	0.2135	0.0064	0.0037	0.0018
Int32	0.0083	0.7138	0.8366	2.0803	0.0665	0.0163	0.0040	0.0019
Float	0.0053	1.6477	1.9616	4.2470	0.0645	0.0147	0.0051	0.0025

Table 14. CC4CS fitting standard error estimation on LEON3.

	Birnbaum Saunders		Inverse Gaussian		Generalized Pareto			Log-Normal	
Data Type	**β**	**γ**	**μ**	**λ**	**k**	**σ**	**θ**	**μ**	**σ**

Int8_t	1.4060	0.0096	5.7109	0.8173	0.0176	1.7257	0	-	-
Int16	1.9303	0.0085	6.4400	1.5013	0.0145	3.9720	0	-	-
Int32	2.8222	0.0071	-	-	0.0122	8.2454	0	0.0090	0.0063
Float	3.3084	0.0066	-	-	0.0117	11.221	0	0.0084	0.0060

It is possible to note that standard errors related to 8051 fit parameters is not overly large, while on LEON3 is much greater and it depends on different HW processors architecture (i.e. cache, pipeline, external memory access and so on), mostly for generalized Pareto. After this error analysis, one of the most important parameter in the distribution fitting and statistical analysis is the confidence interval. Table 9 and Table 10 shows the confidence interval with respect to reference data types and distribution. This parameters can be used to estimate the mean of the execution time of a specific C function using the CC4CS value of the processor.

Table 15. CC4CS confidence interval on 8051.

	Generalized Extreme Value			**Burr**			**Log-Logistic**	
Data Type	**k**	**σ**	**μ**	**α**	**c**	**k**	**μ**	**σ**
Int8_t	0.6740	33.066	93.2399	70.460	14.026	0.1076	4.7529	0.2997
	0.7103	34.421	94.4827	71.481	15.430	0.1206	4.7696	0.3080
Int16	0.4424	48.506	136.554	108.54	9.0662	0.2023	5.0971	0.2602
	0.4744	50.208	138.379	110.80	9.9032	0.2273	5.1118	0.2673
Int32	0.3040	85.490	202.807	196.24	4.3940	0.5477	5.5007	0.2753
	0.3366	88.289	206.087	204.40	4.6546	0.6115	5.5163	0.2828
Float	0.2760	216.57	390.881	336.88	3.7096	0.4799	6.1539	0.3543
	0.2967	223.03	398.571	353.53	3.9626	0.5377	6.1740	0.3639

Table 16. CC4CS confidence interval on LEON3.

	Birnbaum Saunders		**Inverse Gaussian**		**Generalized Pareto**			**Log-Normal**	
Data Type	**β**	**γ**	**μ**	**λ**	**k**	**σ**	**θ**	**μ**	**σ**
Int8_t	139.364	1.6308	311.43	69.063	0.921	94.704	11.0	-	-
	144.846	1.6683	333.56	72.272	0.989	101.47	11.0	-	-
Int16	203.303	1.4529	416.87	126.18	0.449	237.95	12.0	-	-
	210.869	1.4864	442.11	132.07	0.506	253.52	12.0	-	-
Int32	333.524	1.2049	-	-	0.004	546.01	23.0	5.854	1.078
	344.587	1.2327	-	-	0.044	578.33	23.0	5.889	1.103
Float	411.072	1.1208	-	-	-0.20	766.02	28.0	6.062	1.016
	424.040	1.1467	-	-	-0.15	810.02	28.0	6.095	1.039

Other considerations and analysis are under investigation. The main goal of this work is to present a preliminary analysis of this innovative metric, useful to early evaluate processors performance and to early choose the processor technologies so reducing design space exploration time.

8 Evaluation of CC4CS in the HW Domain

The synthesized benchmark used in [REF CEDA1] are extracted from the C-language CHStone benchmark suite [REF], with the remainder being from DWARV. The selected functions originate from different application domains, which are control-flow, as well as data-flow dominated as they aim at evaluating generic (nonapplication-specific) HLS tools. An important aspect of the benchmarks is that input and golden output vectors are available for each program. Hence, it is possible to “execute” each benchmark with the built-in input vectors, both in software and also in HLS generated RTL using ModelSim. The RTL simulation permits extraction of the total cycle count, as well as enables functional correctness checking.
The work [REF CEDA1] performed two sets of experiments to evaluate the compilers. In the first experiment, they executed each tool in a “push-button” manner using all of its default settings, which they refer to as *standard-optimization*. The first experiment thus represents what a user would see running the HLS tools “out of the box.” They used the following default target frequencies: 250 MHz for BAMBU, 150 MHz for DWARV, and 200 MHz for LEGUP. For the commercial tool, they decided to use a default frequency of 400 MHz. In the second experiment, they manually optimized the programs and constraints for the specific tools (by using compiler flags and code annotations to enable various optimizations) to generate *performance-optimized* implementations. Table 17 and Table 18 show the CC4CS evaluated from the result taken in [REF CEDA1].

Table 17: STANDARD-OPTIMIZATION PERFORMANCE RESULTS

Function	Statements C	Commercial		Bambu		DWARV		LegUp	
		Cycles	**CC4CS**	**Cycles**	**CC4CS**	**Cycles**	**CC4CS**	**Cycles**	**CC4CS**
adpcm (int32)	36806	27250	0,74036	11179	0,3037	24454	0,6644	7883	0,2141
aes_dec (int32)	11568	5461	0,472	2766	0,2391	2579	4,4854	7367	1,5702
aes_enc (int32)	11580	3976	2,9124	1574	7,357	5135	2,2551	1564	7,404
gsm (int16)	7334	5244	0,715	2805	0,3824	6866	0,9361	3966	0,5407
mips (int32)	8853	4199	0,4743	4043	0,4566	8320	0,9397	5989	0,6764
bellman_ford	1804	2838	1,5731	3218	1,7838	2319	1,2854	2444	1,3547
sha (int8)	165493	197867	1,1956	111762	0,6753	71163	0,43	168886	1,0205
blowflsh (int8)	105290	101010	0,9593	57590	0,5459	70200	0,6667	75010	0,7124
dfadd (float)	2262	552	0,244	404	0,1786	465	0,2055	650	0,2873
dfdiv (float)	1088	2068	1,9007	1925	1,7693	2274	2,09	2046	1,8805
dfsin (float)	14368	57564	4,0064	56021	3,899	64428	4,4841	57858	4,0268
dfmul (float)	872	200	0,2293	174	0,1995	293	0,336	209	0,2396
jpeg (int32)	962612	994945	1,0335	662380	0,6881	748707	0,7777	1128109	1,1719
motion	8570	ERR	ERR	127	0,0148	152	0,0177	66	0,00077
sobel	21095723	2475541	0,1173	3641472	0,1726	3648547	0,1729	1565741	0,0884
satd	178	87	0,4887	36	0,2022	54	0,3033	42	0,2359

Table 18: PERFORMANCE-OPTIMIZED RESULTS

Function	Statements C	Commercial		Bambu		DWARV		LegUp	
		Cycles	CC4CS	Cycles	CC4CS	Cycles	CC4CS	Cycles	CC4CS
adpcm (int32)	36806	12350	0,3355	7077	0,1922	9122	0,2478	6635	0,5372
aes_dec (int32)	11568	3923	0,3391	2585	0,2234	2579	0,2229	4847	0,419
aes_enc (int32)	11580	3735	0,3225	1485	0,1282	3282	0,2834	1191	0,1028
gsm (int16)	7334	3584	0,4888	2128	0,2901	7308	0,9964	1931	0,2632
mips (int32)	8853	4199	0,4743	5783	0,6532	8320	0,9397	5989	0,6764
bellman_ford	1438	2607	1,8129	4779	3,3233	2319	1,6126	1036	0,7204
sha (int8)	165493	124339	0,7513	51399	0,1897	71163	4,2887	81786	0,4941
blowflsh (int8)	105290	96460	0,9161	57590	0,5469	70200	6,6672	64480	0,6124
dfadd (float)	2262	552	0,244	370	0,1635	465	0,2055	319	0,141
dfdiv (float)	1088	2068	1,9007	1374	1,2628	2846	2,6158	942	0,8658
dfsin (float)	872	200	0,2293	162	0,1857	293	0,336	105	0,1204
dfmul (float)	14368	57564	4,0064	38802	2,7005	90662	6,3099	22233	1,5473
jpeg (int32)	962612	602725	0,6261	662380	0,6881	706151	0,7335	1182092	1,228
motion	8570	ERR	ERR	127	0,0148	122	0,0142	66	0,0007
sobel	15100132	2475541	0,1639	3641402	0,2411	3648547	0,2416	No Val	No Val
satd	135	27	0,2	36	0,2666	54	0,4	42	0,3111

ALTRE ANALISI DA FARE.

Table 19. CC4CS measured on HW with Standard- Optimization performance results.

Data Type	MIN	AM	GM	MAX
Commercial	0,1173	1,137464	0,758158999	4,0064
Bambu	0,0148	1,17924375	0,468344662	7,357
DWARV	0,0177	1,253125	0,650089529	4,4854
LegUp	0,00077	1,339010625	0,483072385	7,404

Table 20. CC4CS measured on HW with Performance-Optimized results.

Data Type	MIN	AM	GM	MAX
Commercial	0,1639	0,85406	0,538482912	4,0064
Bambu	0,0148	0,69188125	0,33433658	3,3233
DWARV	0,0142	1,6322	0,639132411	6,6672
LegUp	0,0007	0,535986667	0,281288968	1,5473

ALTRE ANALISI DA FARE.

9 State of the art

In order to evaluate/estimate the time required by a target microprocessor to run an application, more methods and tools are available both commercially and in the literature.

A first approach is *direct timing measurements* on real processors through an external HW/SW profiling system. RapiTime [2] represents a tool that performs a timing measurement based analysis on real devices: it collects measurements taken from the system at runtime, and joins them with a static description of the software, in order to delineate the worst case path that leads to WCET. The measures are constituted of timestamps related to execution of the code, taken after a source code instrumentation step.

Another method often used is *target processor simulation*. The simulation can be both hardware and software. The hardware simulation can be realized by means of HDL tools. *Intel Altera* and *Xilinx Company* offers an integrated environment with their software suite. *Xilinx Vivado Design Suite* [3] (with *Vivado HLS and Simulator*) consists of a development environment for hardware description in VHDL and Verilog which allows to simulate and analyze described hardware behavior. *Altera Quartus II* [4] is a design software which enables the developer to compile designs, perform timing analysis and synthesize HW/SW solution on FPGA environment. Software simulation can be done with target processor models that execute a cross-compiled binary on the host. This procedure can be implemented through ISSs or processor virtualization. In order to collect timing measurements with this kind of simulation, two approaches are tipically used: *trace-based simulation* and *native simulation*. These methodologies are similar and both use instrumented code in order to obtain information on execution time. The first approach is implemented by *Lauterbach Microprocessor Development Tools* [5] that present a set of software API, de-buggers, logical analyzer and simulators for a large set of HW architecture available on the market. In particular *TRACE32 Instruction Set Simulator* [6] allow designer to test their solutions on different processor technologies. A simulator that works using a *native simulation* is VIPPE [7] and it permits high-level estimations of the software execution time on a virtual target platform.

Finally, OFFIS (*Institute for Information Technology*) has proposed a methodology based on an instrumentation approach that can be used to statically annotate the timing behavior and resource usage of applications in a context of mixed-criticality cyber physical systems [8]. The annotation has been taken by the use of *SocRocket transaction-level modeling framework* [9] that simulates the processors realized by Aeroflex Gaisler [10].

With respect to the approaches previously listed, this work focuses on the realization of a framework able to execute specific benchmarks on different ISS and HDL tools. Such a framework allows the evaluation of a metric to help designers to very early estimate the performance of software applications on different processor technologies. The final effect is the improvement of the development flow of HW/SW Co-Design methodologies.

10 Conclusion and future work

In this work a new metric called CC4CS has been presented. Moreover, a framework that allows to measure and to estimate this metric has been implemented and tested on a benchmark composed of some representative C functions. 8051 and LEON3 processors have been selected as reference targets and used to validate the framework environment and to evaluate CC4CS. Some preliminary results shows a low errors values and a promising estimation methodology. Some statistical analysis and tests have been performed in order to check the CC4CS distributions fitting data and the goodness of such estimations have been evaluated. Some other analysis and considerations related to the HW characteristics (registers and memory size, cache and pipeline interferences, ISA architecture etc.) will be done in the next future, also related to different European research projects and HW/SW Co-Design tools. In particular, some future works involves the use of different ISS and HDL tools to evaluate CC4CS on different processors technologies (i.e. GPP: ARM, MIPS32, NIOS II, etc.; SPP: Spartan3, Virtex7, etc.).

It is worth noting that one of the main goal of this work is to avoid reasoning about assembly code (related to C statements) in order to be able to evaluate CC4CS also for C programs directly implemented in HW by means of High Level Synthesis techniques. methodologies.

11 ACKNOWLEDGMENT

This work has been partially supported by the Artemis-JU AIPP 2013 EMC2 and the ECSEL RIA 2016 MegaM@Rt2 and AQUAS projects.

References

1. D.J. Lilja, Measuring Computer Performance, A Practitioner's Guide, Cambridge University Press, New York, USA, 2000
2. RapiTime, https://www.rapitasystems.com/products/rapitime, Accessed 26 April 2017.
3. Xilinx Vivado, https://www.xilinx.com, Accessed 26 April 2017.
4. Altera Quartus, https://www.altera.com, Accessed 28 April 2017.
5. Lauterbach Development Tools, http://www.lauterbach.com/, Accessed 26 April 2017.
6. Trace32 ISS, http://www2.lauterbach.com/pdf/simulator api.pdf, Accessed 28 April 2017.
7. L. Diaz, E. Gonzalez, E. Villar and P. Sanchez, ”VIPPE, parallel simulation and performance analysis of multi-core embedded systems on multicore platforms,” Design of Circuits and Integrated Systems, Madrid, 2014, pp. 1-7
8. K. Gruttner, A. Herrholz, P. A. Hartmann, A. Schallenberg, and C. Brunzema,OSSS - A Library for Synthesisable System Level Models in SystemC(TM) - The OSSS 2.2.0 Tutorial, Sep. 2008.
9. SoCRocket, Transaction-Level Modeling Framework for Space Applications, https://socrocket.github.io/, Accessed 28 April 2017.
10. LEON3 Processor, http://www.gaisler.com/, Accessed 28 April 2017.
11. GCov Profiler, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, Accessed 26 April 2017.
12. GCC GNU Compiler, https://gcc.gnu.org/onlinedocs/gcc, Accessed 26 April 2017.
13. M. A. Mazidi, "The 8051 Microcontroller & Embedded Systems", Pearson Education Asia, India, 2003
14. Dalton Project, http://www.ann.ece.ufl.edu/i8051/, Accessed 26 April 2017.

15. SDCC, http://sdcc.sourceforge.net/doc/sdccman.pdf, Accessed 26 April 2017.
16. TSIM Leon3 Simulator, http://gaisler.com/doc/tsim-2.0.23.pdf, Accessed 26 April 2017.
17. LEON (BCC), http://www.gaisler.com, Accessed 26 April 2017.
18. GNU Binutils, https://www.gnu.org/software/binutils/, Accessed 26 April 2017.
19. Newlib C-library, https://sourceware.org/newlib/, Accessed 26 April 2017.
20. SystemC, http://accellera.org/downloads/standards/systemc, Accessed 26 April 2017
21. Matlab Statistics Toolbox, https://it.mathworks.com/ , Accessed 26 April 2017
22. F. Vahid,T. Givargis, "Embedded System Design: A Unified HW/SW Introduction (ESD)", John Wiley & Sons 2001
23. L. Pomante, "Electronic System-Level HW/SW Co-Design of Heterogeneous Multi-Processor Embedded Systems", River Publishers, 2016
24. M. A. Mazidi, J. G. Mazidi, R. D. McKinlay (Author) "The 8051 Microcontroller and Embedded Systems (2nd Edition)", Pearson, 2005
25. M. Siegesmund, "Embedded C Programming Techniques and Applications of C and PIC MCUS (1st Edition)", Newnes, 2014