CC4CS: A Unifying Statement-Level Performance Metric for HW/SW Technologies

V. Stoico[1], V. Muttillo[1], G. Valente[1], L. Pomante[1], F. D'Antonio[2]
[1]Università Degli Studi Dell'Aquila - Center of Excellence DEWS, L'Aquila, Italy
{vincenzo.stoico}@student.univaq.it, {vittoriano.muttillo, giacomo.valente}@graduate.univaq.it, {luigi.pomante}@univaq.it
[2]Thales Alenia Space, Via Campo di Pile, L'Aquila, Italy, {fausto.dantonio-somministrato}@thalesaleniaspace.com

I. INTRODUCTION

In the last thirty years there has been an exponential increase of the spread and evolution of information technology. In this respect, it is certainly underlined the spiraling of embedded systems. The presence of such systems in everyday life is constant and often almost invisible. Moreover, the adopted design methodology is of critical importance during the development of an embedded system. Unfortunately, such methodologies usually lack generality and can be very effort and time consuming, especially when working at a low level of abstraction. For this reason, working on a higher abstraction levels (i.e. system-level) is needed and early performance estimation is a fundamental step. One of the most common metric for computer performance analysis is MIPS (*Million Instructions Per Second*) [2] because it is normally available directly on data-sheet. MIPS metrics measures millions of assembly instructions executed per second, and it can be useful for comparing two processors with the same ISA (*Instruction Set Architecture*) but it is pointless in comparing ones with different micro-architectures.

In such a context, the objective of this work is to analyze the usefulness of a metric related to C programming language statements. This kind of metric, called CC4CS (*Clock Cycles for C Statement*), is defined as the ratio between the number of clock cycles required by the target processor to run an application and the number of executed C statements.

For this purpose, a framework that helps to calculate this kind of metric for a given program has been realized. Additionally, such a framework is also able to automatically generate large amounts of constrained random inputs and to evaluate statistics on the metric. By analyzing the data, it is possible to validate the metric with respect to the performance of a target processor. Summarizing, such a framework allows to easily evaluate CC4CS in a repeatable manner. The working process has been defined by looking at the CC4CS definition. The framework exploits an *Instruction Set Simulator* (ISS) and the simulation permits to calculate the number of clock cycle needed to execute the program while the number of executed C statements is obtained performing a profiling on the host architecture.

II. STATE OF THE ART

In order to evaluate the number of clock cycles required by the target microprocessor to run an application, several methods and tools are available in literature. A first approach is a direct timing measurement on real microprocessor through an external HW/SW profiling system (e.g. Rapitime [1]).

Another method that can be used is a target microprocessor simulation. The simulation can be both hardware and software. The hardware simulation can be realized by usage HDL tools (e.g. both Intel Altera and Xilinx company offers an integrated environment with their software suite). 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 microprocessor virtualization.

With respect to the approaches previously listed, this work focuses on the realization of a framework that executes specific benchmarks on different ISS technologies in order to provide a metric (CC4CS) able to help designers to early estimate the performance of a software application on different target processors.

III. FRAMEWORK IMPLEMENTATION

To validate the evaluation process and the metric, the framework has been tested on an Intel 8051 (core) and 3 main phases of the working process have been applied.

Inputs Generation: it is based on a module that automatically generates constrained random inputs for a given benchmark function. The module needs to know which kind of parameters the function requires. For this purpose, the programmer defines 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 find its name and to find out proper data for the function. For each parameter, the user is asked to insert a range for meaningful values (min and max) and then the number of values to be randomly generated. In case of a function that requires more than one variable, the Cartesian product of generated values is provided. For each produced combination a header file is created that contains the values of a single combination. At the end, the input generator creates the directory that contains all the header files.

Profiling on the host architecture: it is based on a procedure that counts the number of C statements executed. This value is obtained performing a profiling of the program. To have this task done, the GCov [4] profiler has been used. First of all, the program is compiled using GCC [5] 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. Then, the compilation process triggers the creation of a *.gcno* file and generates also the corresponding *.gcda* file. To complete the task, the gcov command is executed. To obtain the number of C statements executed, a sum of the single timing numbers is then performed.

Execution and metric evaluation on target processor: it is based on a procedure that calculates the number of clock cycles used by the target processor to execute the input/function pairs. The execution has been done with a software simulation of the processor by using an Instruction Set Simulator (ISS). In this work the core of the 8051 microcontroller has been considered as target platform. The Intel 8051 microcontroller [7] is built around an 8-bit CPU. The adopted memory model is the Harvard one, i.e. the core accesses to data and instructions by using 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 it has been developed by the *Dept. of computer Science of the University of California* [3]. 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 these characteristics, it has been chosen as the reference ISS for the evaluation of the CC4CS for 8051 microprocessor. The functions composing a benchmark have been compiled, with the SDCC (*Small Device C Compiler*) [6] compiler. SDCC is free open source C compiler suite designed for 8 bit processors. 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 perform the simulation. This kind of file is generated by SDCC. To do a proper simulation, during the compilation two options were 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 generates an .ihx file that is to .hex file using the *packihx* command. At the end, the ISS is executed. It generates a file that contains information about the simulation. After the simulation, the framework is ready to evaluate the metric and some statistics on the base of all the inputs generated for the different functions. These calculations are made with a program that returns two files containing metric values, for each input, and statistics on the sample.

IV. CC4CS ESTIMATION AND ANALYSIS

To validate the CC4CS metric some preliminary tests has been executed. A benchmark composed by 10 algorithms has been used. Some preliminary results are shown in Table 1. The metric has been evaluated with respect to 10.000 input files per function. For each single function, different data types have been considered (*int8*, *int16*, *int32*, and *float*) because the performance of each software changes with respect to the dimension of data since the microcontroller is based on a 8-bit CISC CPU core with a 8-bit ALU. Furthermore, with float data type the values of CC4CS are increasing with respect to the other values due to the lack of an FPU and to the HW architecture registers size.

TABLE I. CC4CS MEASURED USING 10.000 INPUT DATA SET PER FUNCTION (100.000 EXECUTION)

Method	Min	AM[a]	SD[b]	90[c]	95[d]	Max
Int8	58	117,8	47,4	170	176	410
Int16	80	161,4	67,5	265	297	453
Int32	104	227,9	88,7	354	400	760
Float	4	537,7	267,6	969	1173	1301

[a.] *AM: Arithmetic Mean, [b]SD: Standard Deviation, [c]90: 90th percentile, [d]95: 95th percentile*

V. CONCLUSION AND FUTURE WORK

In this work a new metric called CC4CS has been presented. A framework that allows to measure and estimate this metric has been implemented and tested on a benchmark composed of some representative functions (e.g. *Bellman Ford, Banker's Algorithm, Matrix Multiplication* etc.). The 8051 microcontroller HW architecture has been selected as reference and used to validate the framework environment and to evaluate the CC4CS metric. Future works involve the use of different ISSs to evaluate CC4CS on more processors (ARM, LEON, NIOS II etc.). Then, some other analysis and considerations related to the HW characteristics (registers and memory size, register binding, cache and pipeline interferences, ISA architecture etc.) of the processors will be done to improve accuracy of the metric. Finally, it is worth noting that, since this work avoids reasoning about assembly code related to C statements (i.e. it is based only on C code profiling and target execution time), it will be extended to evaluate CC4CS also for C functions directly implemented in HW by means of *High Level Synthesis* techniques. In other words, CC4CS will be used as an early unifying statement-sevel performance metric for HW/SW co-design methodologies (in particular to support system-level timing HW/SW co-simulations).

ACKNOWLEDGMENTS

This work has been partially supported by the ECSEL RIA 2016 MegaM@Rt2 and AQUAS projects.

REFERENCES

[1] RapiTime, Automated performance measurement on-target timing analysis tool, https://www.rapitasystems.com/products/rapitime, Accessed 26 April 2017.

[2] D.J. Lilja, Measuring Computer Performance, A Practitioner's Guide, Cambridge University Press, New York, USA, 2000.

[3] Dalton Project: 8051 microcontroller, University of California, http://www.ann.ece.ufl.edu/i8051/, Accessed 26 April 2017.

[4] GCov Profiler, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, Accessed 26 April 2017.

[5] GCC GNU Compiler Collection, https://gcc.gnu.org/onlinedocs/gcc, Accessed 26 April 2017.

[6] SDCC, http://sdcc.sourceforge.net/doc/sdccman.pdf, Accessed 26 April 2017.

[7] M. A. Mazidi, J. G. Mazidi, R. D. McKinlay The 8051 Microcontroller and Embedded Systems, 2nd Edition, Prentice Hall, 2005.