High Level Extraction of SoC Architectural Information from Generic C Algorithmic Descriptions

Marco Mattavelli and Massimo Ravasi

***Abstract*—The complexity of nowadays, algorithms in terms of number of lines of codes and cross-relations among processing algorithms that are activated by specific input signals, goes far beyond what the designer can reasonably grasp from the "pencil & paper" analysis of the (software) specifications. Moreover, depending on the implementation goal different measures and metrics are required at different steps of the implementation methodology or design flow of SoC. The process of extracting the desired measures needs to be supported by appropriate automatic tools, since code rewriting, at each design stage, may result resource consuming and error prone. This paper presents an integrated tool for automatic analysis capable of producing complexity results based on rich and customizable metrics. The tool is based on a C virtual machine that allows extracting from any C program execution the operations and data-flow information, according to the defined metrics. The tool capabilities include the simulation of virtual memory architectures.**

I. Introduction

The always increasing complexity of processing algorithms leads to the need of more and more intensive specification and validation tasks, and forces to perform these tasks at a high level of abstraction in order to minimize the cost and time of such preliminary design phase. It is a commonly adopted practice to write such abstract algorithm reference descriptions by means of common programming languages such as C and C++, as confirmed by well known examples from standards such as MPEG 4 [1] [2] and JPEG2000, where the reference description is provided by the standard definitions themselves.

In a way, even though conceived as abstract system descriptions, algorithmic models can be seen as real implementations over a generic virtual architecture, such virtual architecture being the chosen programming language. As for the successive system design over a real, possibly heterogeneous, SoC architectures, C algorithm descriptions also known as verification models, are thus the starting point for driving the first architectural design choices. A common methodology is to rewrite such algorithm description into *architectural C descriptions* where the C code *architecture* corresponds to the functional elements of the final architecture. The possibility of extracting architectural information from generic non-architectural code and then refine and validate the *architectural description* is very attracting because is permits to avoid wrong choices or to reduce the number and workload of redesign iterations.

Sec. II briefly reviews the *state of the art* in algorithmic complexity analysis and complexity metric measurements; Sec. III introduces an automatic integrated tool conceived for the complexity analysis and virtual exploration of the design-space for complex algorithms, called Software Instrumentation Tool (SIT). Section IV describes the measures obtainable for the computational complexity analysis, data-flow and storage analysis, the simulation of virtual architectures and outlines some possible evolutions, of the analysis capabilities. Section V concludes the paper.

II. Complexity Analysis and Design of Complex Systems

In literature several different ways have been proposed to measure the complexity of the building blocks of an algorithm and of their execution. Two main axes are typically recognized: the computational complexity analysis and the data-transfers and storage complexity analysis. The computational complexity represents the computational load that has to be sustained to perform a given task; it can be measured according to different metrics, such as number of times a given task has to be performed, number of operations or number of clock cycles. Similarly, the data transfer and storage complexity analysis may aim to measure the simple counting of I/O operations, or to estimate a cache performance, or to estimate the I/O bandwidth and processing demands.

A. Static Approaches

The methods based on a static analysis of the source code range from the simple counting of the number of operations appearing in a program up to sophisticated approaches determining lower and upper running time of a given program on a given processor [3]. While the simple counting technique provides a very accurate evaluation of the operations, it cannot handle loops, recursion and conditional statements except for some particular cases. Explicit or implicit enumeration of program paths can handle loops and conditional statements and can yield bounds on run-time best and worst case [3],. The main drawback of these techniques is that the typical real processing complexity of many algorithms heavily depends on the input data statistics while static analysis can only detect upper and lower bounds. Moreover, restricted programming styles such as absence of dynamic data structures, recursion and bounded loops are required so as to correctly perform a static analysis [4].

M. Ravasi and M. Mattavelli, are with the Signal Processing Laboratory 3, Signal processing Institute, Swiss Federal Institute of Technology of Lausanne (EPFL), CH-1015 Lausanne, Switzerland (e-mail: massimo.ravasi@epfl.ch marco.mattavelli@epfl.ch).

Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)
1098-8068/05 $20.00 © 2005 IEEE

B. Profilers and Complexity Analysis at Instruction-Level

The information provided by profilers is only available at a relatively high level of abstraction that is at a function level [5]. Since signal processing algorithms typically spend the majority of the time in a few functions, more details and reliable statistics about the processing operations executed by those functions are necessary to assess and understand the complexity of an algorithm. If only function-level information is provided, a complete rewriting of the program code, for instance to replace each elementary operation with a function call, is necessary to obtain accurate statistics of the executed operations. Profilers are well suited for program optimization tasks on a given specific architecture, as they measure, in fact, the time spent by parts of a program. Furthermore, the number of calls of a function can help the partial redesign of the program to reduce the number of function calls to costly functions.

The information gathered with profilers strictly depends on the underlying machine and on the compiler optimizations, while a complexity evaluation depending only on the algorithm itself is more appropriate for high-level SoC system design. For such reason, tools for profiling and optimization at very high abstraction level – i.e. at programming language level – are better suited for system design. An example of such tools is the ATOMIUM [6] tool-suite (A Toolbox for Optimizing Memory I/O Using geometrical Models), which addresses memory related aspects of system-design, by supporting the Data Transfer and Storage Exploration methodology (DTSE) [7]. ATOMIUM allows designers to quickly identify memory related hotspots in the algorithm such as data structures and arrays characterized by large data exchanges and functions, or function portions, requiring dominant memory access bandwidths as well as run-time peak memory usage. The provided data-transfer analysis is based on a *flat* memory architecture model, which does not allow taking into account the effects of introducing one or more cache memories in the memory hierarchy.

III. The Software Instrumentation Tool (SIT)

The approach, presented in this paper has been developed with the goal of measuring the complexity of a specific algorithm independently from the hardware architecture on which the software model of the algorithm is run. In other words this means to extract *architectural* algorithmic information from non-architectural and/or architectural C algorithmic description. This approach is in line with methodological approaches proposed for instance in [8] and [6], aiming at optimizing data transfers, memory bandwidths and storage requirements directly on algorithm specifications at high abstraction level.

The new approach of SIT [9] is possible by means of a breakthrough in the instrumentation/overloading technology enabling a complete detection of all C operators without any limitation in the way pointers and data structures are used **Error! Reference source not found.**. Such technology enables, besides a complete operator analysis, a full data-transfer analysis on any data structure providing design-oriented algorithmic complexity evaluations at pure unstructured source-code level. In a way, SIT can be seen as a virtual-machine for running C source code. The instruction set of this virtual-machine corresponds exactly to the set of C language operators and control-statements. By means of such virtual-machine, all the operations performed during the execution of the instrumented verification model are intercepted and processed, providing as result an exhaustive basis for computational complexity and architectural analysis. Besides such *operator* based analysis, customizable virtual memory architectures can be *plugged* into the virtual-machine extending the analysis capabilities to the data-transfer and storage domain. The current version of SIT is capable of instrument *any* C source code, independently of the chosen C dialect, allowing to analyze a software program *as-is*, without the need of tedious and error-prone work such as massive code rewriting or manual code instrumentation. The main innovations of SIT versus the *state-of-the-art* tool are:

- Pure algorithmic complexity analysis at the highest possible abstraction level: source-code level. The analysis does not depend on the underlying platform or on the compilation, but *only* on the source-code.
- Input-data dependent analysis, the implementation of algorithms is now based, rather than on the worst-case, on the Cost/Quality-of-Service trade-off, which implies the need of an input-data dependent analysis.
- Completely automatic instrumentation process with no limitations for ANSI C and K&R compliant C code.
- Fully customizable memory simulation, for a versatile data-transfer and storage analysis apt to explore different design-spaces in the memory architecture domain.
- The SIT *virtual-machine* is also a validated reliable framework for building on top of it other simulators and analysis tools, for different metrics and architectural explorations.

The schematic diagram of the main functional blocks constituting the SIT analysis framework and the blocks of the instrumentation and simulation process is shown in Figure 1. The whole instrumentation process, from the source files to the instrumented executable, is completely automatic: it appears to the end user as a normal compilation; it can be tuned by configuring specific instrumentation features, in several different ways (by means of environment variables, configuration files or command line options). The instrumented executable can be run on real input data, exactly as its native executable counterpart, to produce the complexity analysis results, which can be browsed and manipulated by means of an interactive GUI.

IV. Architectural Measurements and Metrics

A. Computational Complexity Analysis

The set of intercepted operations is an extension of C operator set: it comprises both explicit C operations (e.g., +, -, *, etc.) and implicit operations (e.g., implicit type castings

in expressions, variable constructions). Similarly, the data-type basis is an extension of the C data-types set, comprising C simple types (int, float, etc.), C derived types (pointers, vectors, structures and pointers to functions). Furthermore, results are collected along a third axis, the execution-tree; the user can choose if the nodes in the execution-tree correspond to the function calls (low execution-tree resolution, faster simulation) or if they include compound statements and basic-block (high execution-tree resolution).

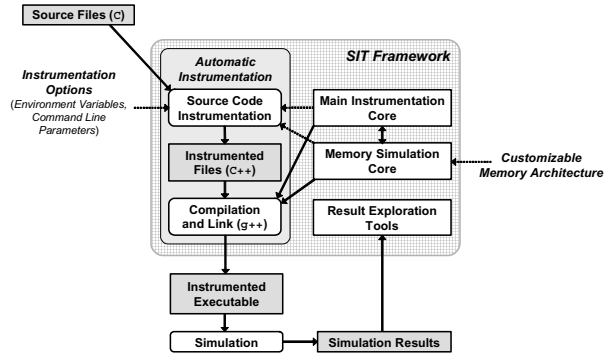

Figure 1. The "Software Instrumentation Tool" complexity analysis framework.

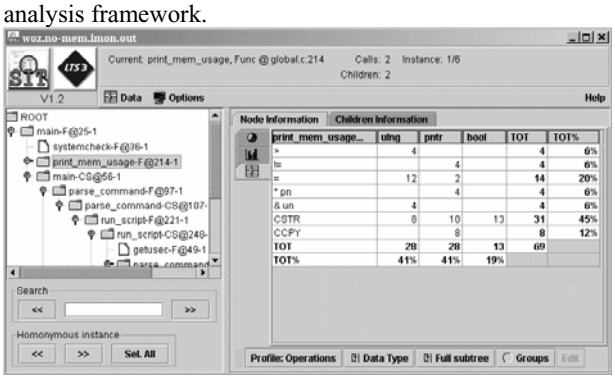

Figure 2. Example of computational complexity analysis results provided by the Interactive GUI.

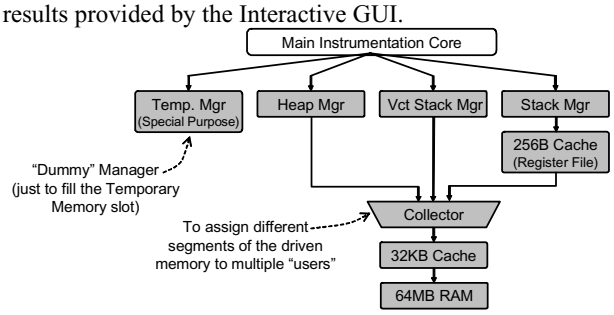

Figure 3. Example of virtual memory architecture.

Figure 2 shows an example of computational complexity analysis results (the picture is a screenshot of SITView, the GUI of SIT). On the left side, there is the execution tree, where the 'print_mem_usage' function is selected; since in this example the execution tree was traced at high resolution, two homonymous 'main' nodes are presented: the first is 'main-F' (two nodes above 'print_mem_usage') and corresponds to the actual 'main' function; the second is 'main-CS' (immediately below 'print_mem_usage') and corresponds to an inner compound statement of the 'main' function. On the right side, the numerical results of the computational complexity analysis are presented; the labels for the horizontal axis contain both C data types (ulng = unsigned long, pntr = generic pointer) and the extra BOOL type (bool label); the vertical axis presents the operation basis, where both explicit operations, i.e. >, >=, =, * pn (pointer dereferencing operator) and & un (unary & operator, returning the address of a variable) and implicit operations, i.e. CSTR (variable "construction") and CCPY (copy initialization in variable construction), can be identified.

B. Data-transfer and Storage Complexity Analysis

The data transfers and storage requirements play a fundamental role in the evaluation of the algorithmic complexity of a system for the design of a SoC. In data dominated algorithms most of the power consumption and bus load is due to data transfers and the optimization of these dominant costs is one of the most critical steps in the development of efficient and low-power implementations [8]. By intercepting memory accesses by means of *read* and *write* functions in instrumented types' C++ classes and by associating to the algorithm an underlying memory model, SIT enables the simulation of memory operations and the extraction of relevant information and measurements about memory performance, such as number of data-transfers, memory usage, cache hits and misses, etc.

The underlying memory architecture, for which measurements are required, can be easily specified aside without having to rewrite the algorithm source code. The Memory Simulation Core is the basic framework for memory simulation for data-transfer and storage complexity analysis. The simulated memory architecture is composed of several memory models, each of them composed by different simulation modules (allocation managers, cache memories, and storage memories). Figure 3 shows an example of virtual memory architecture that can be simulated with SIT. Figure 4 shows an example of memory simulation results. On the vertical axis, the different simulated modules can be identified, which in this case correspond to the simulation of three memory models (i.e. *Stack*, *VctStack* and *Heap*). It can be clearly seen that the results generated through the simulation vary according to the nature of a simulation module: the four labels *RHist*, *RMisses*, *WHits* and *WMisses* (Read/Write Hits and Misses) are specific for caches, the label *Alloc* is specific for allocation managers and the labels *PushSP* and (Push Stack Pointer) are specific for stack-like allocation managers. The results of the data-transfer and storage complexity analysis are collected along the same execution-tree basis as with the computational complexity analysis results.

main-F@25-1	Read	Write	RHits	RMisses	WHits	WMisses	Alloc	PushSP
Stack{RAM}	3.27e8	1.67e8						
Stack{TestCache:32:4}	8.77e9	4.80e9	8.51e9	2.59e8	4.70e9	1.04e8		
Stack{StackMgr}	7.37e9	3.40e9					2.16e8	3.49e8
VctStack{TestCache:256:64}	1.43e9	1.40e9	1.43e9		1.40e9			
VctStack{StackMgr}	3.31e7	4.12e6					3.27e3	3.49e8
Heap{RAM}	2.48e10	4.74e7						
Heap{TestCache:256:64}	4.03e9	2.88e7	2.48e9	1.54e9	2.34e7	5.39e6		
Heap{DynMgr}	2.35e9	2.81e7					6.90e6	

Figure 4. Example of data transfers and storage complexity analysis results.

The simulation and analysis capabilities of the custom

memory simulation cores can be further improved by fully interfacing directly with the Main Instrumentation Core – i.e. by bypassing the default interface between the Main Instrumentation Core and the Memory Simulation Core. More specifically, the Memory Simulation Core can be driven not only by the data-transfer and storage events, as in the default case, but also by the operation interception events. By this way, it is possible to design custom simulation and analysis cores, which may be targeted for other analyses than the data-transfer and storage complexity analysis or the computational complexity analysis only. That is, SIT can be easily reused as framework for developing new simulation and analysis tools.

Another interesting feature of the tool is the possibility of weighting all computational and memory based operators according to some specific target platforms. Accurate evaluations of the performance on the target platform are possible without the need of the actual porting of all or of some parts of the code [9].

C. Automatic Measurement of Inter-Function Data-Transfers for Explicit Statement of Data-transfer dependences among Functions and for Functional Modules Identification

A static analysis of a software program allows identifying the dependences among the various functions in terms of function call dependences. A dynamic analysis in real working conditions allows evaluating the *real* dependences among functions by explicitly detecting the actual function-call tree, with a noticeable improvement with respect to static analysis (e.g., by dead-branch detection, by faithful evaluation of recursive function-call branches and by explicitly taking into account dynamic dependences). Indeed, this analysis results to be of limited use for the system designer, as the data-transfer dependences between the functions cannot be derived from the study of the function-call tree. It is not uncommon that two or more functions exchange a great amount of data through a common buffer and yet they are *far* from each other in the function-call tree, possibly belonging to completely different branches. Furthermore, the functions in a verification model are often loosely related with the actual functional modules of the corresponding application, since several functions may contribute to provide the functionalities of a functional module Conversely, for the system designer it is very important to have an overall vision of an algorithm, of how it is composed by different modules and on how they interact with each other. Explicit measurements of the inter-function data-transfers are a meaningful basis for high-level SoC architectural optimizations. For programs composed by many nodes in the function-call tree, a bottom-up analysis of the function-call tree and of the inter-function data-transfer graph can easily help identifying the different functional modules by grouping the nodes in the call tree into groups with limited data-transfers toward the other modules.

Another simulation capability of SIT is to automatically generate the inter-function data-transfer graph by means of the memory simulation core.

a) Function-call tree b) Functions, functional modules and data-transfer dependences

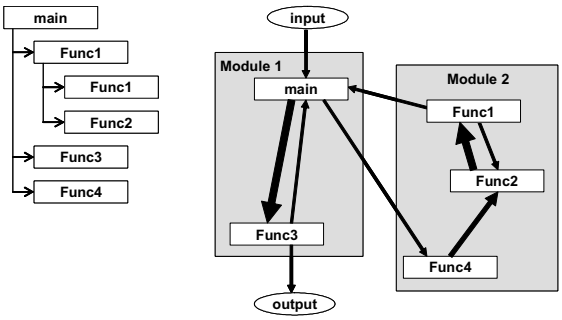

Figure 5. The function call tree (a) does not help detecting the actual data transfer dependences functions the functions and grouping the functions in functional modules (b).

V. CONCLUSIONS

This paper has presented a tool supporting complexity analysis of C algorithm descriptions for high level SoC architectural exploration. The tool is based on a breakthrough in instrumentation technology enabling the implementation of a C virtual simulator capable of measuring operators and data transfers during the execution of algorithms. Besides being completely automatic in the sense that no code rewriting is needed, the simulator can be configured to provide measurements on user configured memory architectures. Extensions of the metrics such as critical path measurements or other simulation capabilities obtainable using the SIT framework are not included here for brevity and can be found in [10][11].

REFERENCES

[1] ISO/IEC, “Information technology – Coding of audio visual objects – Part 2 Visual”, ISO/IEC International Standard 14496-2 (MPEG-4).

[2] ISO/IEC, “Information technology – Coding of audio visual objects – Part 10 Advanced Video Coding”, ISO/IEC 14496-10.

[3] Y. S. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration”, IEEE Trans. on Computer-Aided Design,of Int Circuits and Sys, vol. 16, pp. 1477-1487, Dec. 1997.

[4] E. Kligerman and D. Stoyenko, “Real–time Euclid: A language for reliable real time systems”, IEEE Transactions on Software.Engineering, vol. SE 12, pp. 941 949, September 1986.

[5] S. Graham, P. Kessler, and M. McKusick, “gprof: A call graph execution profiler”, in Proceedings of Symposium on Compiler Construction (SIGPLAN), vol. 17, pp. 120 126, June 1982.

[6] IMEC, “What is ATOMIUM?”, presentation page at the address http://www.imec.be/design/multimedia/ atomium/.

[7] F. Catthoor, et al, "Optimisation of global data transfer and storage organisation for decreased area and power in data dominated real time processing systems", IMEC Internal report, November 1998.

[8] L. Nachtergaele et al. “System Level Power Optimization of Video Codecs on Embedded Cores: A Systematic Approach”, Journal of VLSI Signal Processing, 18, pp. 89 109, 1998.

[9] M. Ravasi, M. Mattavelli,et al : "High-Level Algorithmic Complexity Analysis for the Implementation of a Motion-JPEG2000 Encoder", in "Integrated Circuit and System Design" Lecture Notes in Computer Science, LNCS 2799, pp 440-450, Springer September 2003.

[10] M. Ravasi, M. Mattavelli, "High-Level Algorithmic Complexity Evaluation for System Design", Journal of Systems Architecture, vol. 48/13-15, pp. 403-427, Elsevier Science B.V., May 2003.

[11] A. Prihozhy, M. Mattavelli, D. Mlynek, "Data Dependences Critical Path Evaluation at C/C++ System Level Description", Lecture Notes in Computer Science , LNCS 2799, pg. 569 579, Springer, September 2003.