

High-Abstraction Level Complexity Analysis and Memory Architecture Simulations of Multimedia Algorithms

Massimo Ravasi and Marco Mattavelli

Abstract—An appropriate complexity analysis stage is the first and fundamental step for any methodology aiming at the implementation of today's (complex) multimedia algorithms. Such a stage may have different final implementation goals such as defining a new architecture dedicated to the specific multimedia standard under study, or defining an optimal instruction set for a selected processor architecture, or to guide the software optimization process in terms of control-flow and data-flow optimization targeting a specific architecture. The complexity of nowadays multimedia standards, 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 and 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. 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 reviews the *state of the art* of complexity analysis methodologies oriented to the design of multimedia systems and 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. This paper shows some examples of complexity analysis results that can be yielded with the tool and presents how the tools can be used at different stages of implementation methodologies.

Index Terms—Complexity analysis, computational complexity, data-exchange, virtual architecture simulation.

I. INTRODUCTION

THE continuous efforts of developing better performing multimedia standards, in terms of compression efficiency, quality of service and functionalities, yield large increases of the algorithms complexity. Here the term complexity is intended in a broader and more intuitive sense than its strict mathematical definition only considering the size of the algorithm minimal descriptions. More precisely, we are mainly interested in the various aspects and results of the run-time algorithm complexity metrics. Such metric results can be hardly evaluated, from the algorithm code itself because of its

size (i.e., complexity), and finally they are proportional to the development costs of software, hardware, or heterogeneous architectures aiming at minimizing specific implementation cost functions.

In current multimedia systems, multiple tasks, of quite different nature, coexist in the same application; more specifically, an application may need, at the same time, low-level computationally intensive tasks (e.g., signal processing) and high-level user interaction tasks (e.g., graphic user interfaces, touch screens). Depending on its type, each task can be more efficiently implemented on a specific device or architecture, yielding the need of heterogeneous systems, referred to as heterogeneous hardware/software systems [1], which may include more devices of different types and in which the different tasks can be implemented either in hardware [e.g., by means of application specified integrated circuits (ASICs) or field-programmable gate arrays (FPGAs)] or in software [e.g., running on general purpose processors or digital signal processors (DSPs)]. Heterogeneous architectures may be built by means of multiple independent devices [e.g., multiple integrated circuits (ICs)] or may reside on a single chip [e.g., systems-on-chip (SoC) and embedded systems].

The design of a system begins with an overall system specification and validation, firstly to clearly and completely define the functionalities the new system is meant for and, secondly, to provide a reference implementation in order both to verify that the system does correctly present the expected behavior and to test the system performance by the abstract algorithmic point of view [e.g., quality of service (QoS), compression rate, peak signal-to-noise ratio (PSNR), etc.] Besides, 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 reference descriptions—often referred to as “verification models”—by means of common programming languages such as C and C++, as confirmed by well known examples from standards such as the reference software for MPEG 2 [2], or MPEG 4 [3], [4] and JPEG2000 [5], where the reference description is provided by the standard definition itself.

Verification models written in common programming languages, thus become the main references for the design, at the place of the old textual descriptions, and are the true starting point for the design of new implementations. In a way, even though conceived as abstract system descriptions, verification

Manuscript received September 30, 2003; revised March 31, 2004.

The authors are with the Signal Processing Laboratory, École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland (e-mail: massimo.ravasi@epfl.ch; marco.mattavelli@epfl.ch).

Digital Object Identifier 10.1109/TCSVT.2005.846414

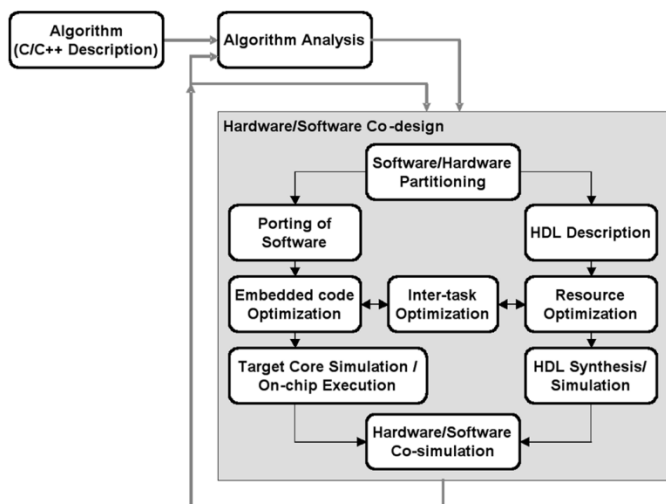


Fig. 1. Typical simplified design flow of a software/hardware heterogeneous system.

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, architecture, verification models are thus the starting point from which deriving all the necessary information for driving the first architectural design choices, as shown in Fig. 1.

The investigations trying to define the lower complexity bounds of the execution of standard multimedia algorithms on different candidate architectures are indeed based on reference software descriptions. Such approaches are typically limited to the analysis of some specific coding *modes* [such as discrete cosine transform/inverse discrete cosine transform (DCT/IDCT)], fractional-pel motion compensation, context adaptive binary arithmetic coding, loop filtering, etc.) that case-by-case constitutes the major processing load like, for instance, is reported in [6]. The objective of such analysis is to obtain approximate measures that can identify classes of candidate platforms for the actual implementations. In other cases the objective is to define appropriate tradeoffs between compression performance and implementation complexity or better cost of the implementation resources. The analysis and the comparisons are performed on optimized implementations of the reference software [7]. Another implementation approach, based on defining an optimized processor instruction-set for a specific multimedia algorithm, starts as well with a statistical analysis of the different coding modes. Such analysis is again performed on the reference software description [8]. All these three examples from the video multimedia field show that not only a specific complexity analysis is the starting point of any implementation process, but also that there is a lack of suitable tools assisting this fundamental stage. The heuristics and methods used in most of the work reported in literature present strong limitations, in general present a very limited portability, are often time- and resource-consuming or provide results with a low level of accuracy. In all cases the analysis requires a relevant amount of work that, besides being tedious and time consuming, is certainly error-prone when very large source codes are handled. In addition, for instance, the approach used in [6] neglects all

the overheads generated by the logic and functions outside the core function-set under analysis and assumes heuristic factors for the porting on different platforms ranging from factors of 2 up to 5 depending on the optimization level. Moreover, the analysis of the data-flow, possibly evaluating different cache hierarchies, is missing, while it may strongly affect the real implementation cost and performance [9].

The paper is organized as follows. Section II reviews the *state of the art* in complexity analysis and complexity metric measurements oriented to the video/multimedia field, Section III introduces an automatic integrated tool conceived for the complexity analysis and virtual exploration of the design-space for video/multimedia algorithms, the software instrumentation tool (SIT). Section IV presents some examples of the results obtainable in terms of both computational complexity, data-flow and storage analysis, and presents some possible evolutions, desirable improvements of the analysis capabilities and outlines some interesting features that constitute the subject of further work and research. Finally, Section V concludes the paper.

II. COMPLEXITY ANALYSIS AND DESIGN OF COMPLEX SYSTEMS

The very first step in the design of complex system is an exhaustive analysis of the system under study in order to fully comprehend its basic structure, to measure its complexity with the richest possible complexity metric, so as to discover the bottlenecks and the most critical constitutive blocks and to “explore” multidimensional design-spaces. The complexity analysis can be straightforward in the case of simple systems where a block diagram and some annotations can reveal all the necessary information for the design. Conversely, in the case of complex multimedia systems the preliminary complexity analysis can result to be a very hard task; a block diagram providing an overall view of the system may not be enough for a complete comprehension of the whole system; detailed information about all the constitutive blocks and about their theoretical complexity may be available, yet the overall system behavior and complexity may not be easy to be understood, especially when they strictly depend on the input data as in the case of multimedia systems [10]. Last but not least, most of the times the only available reference for the system to be designed is its software verification model, which may be composed by several thousand source-code lines and a manual analysis or partial code rewriting result to be very time-consuming tasks.

An important issue to take into account when the complexity analysis is based on the verification models is that the target architecture on which the system will be implemented will typically be different from that used to compile and run the verification models. For this reason, on one side the complexity analysis must be as independent as possible from the underlying simulation platform and from the compilation process (e.g., compiler optimizations) and, on the other side, it must provide the most accurate information for the implementation on the *virtual* target architecture.

Since the design of optimal solutions strictly depends on the preliminary complexity analysis of the system, the more complex the considered application is, the more crucial the

complexity analysis task becomes. For all these reasons, it is indeed obvious that automatic tools and integrated environments for complexity analysis are nowadays a fundamental need in the design of complex systems. Such tools not only have to ease the analysis task by relieving the designer of long annoying analyses by hand, but they also have to assure the reliability of the results of the analysis phase and to provide measures of candidate “virtual architectures” concerning both the computational and the data-transfer complexity.

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’s performance, or to estimate the I/O bandwidth and processing demands. Obviously, the choice of the complexity metrics depends both on the specific objectives of the evaluation and on the results obtainable with the chosen complexity analysis approach. In literature different complexity metrics have been considered and compared for specific implementation problems such as, for instance, motion estimation [11]. Complexity metrics can also be used in conjunction with quality metrics in order to jointly evaluate the tradeoff between complexity and performance. Several classes of *state-of-the-art* approaches to complexity analysis are briefly overviewed in the next subsections.

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 [12], [13]. 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 [12], [13]. 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. For video coding algorithms, for instance, strict worst case analysis can lead to results one or two orders of magnitude higher than the typical complexity values that can be measured on typical video sequences [14], [15]. 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 [16].

B. Profilers and Complexity Analysis at Instruction-Level

Instruction level profiling provides the number and type of processor instructions that a program executes at runtime. These

data give information on computational, control, and memory access costs and can be used for complexity evaluation, as well as for performance tuning of programs and algorithms. The constraints for an instrumentation tool are regarded as following: no restrictions for the programmer, no source code modifications, as well as high portability between different computer architectures, operating systems and compilers. Profilers (e.g., [17]) can provide two types of results: number of calls of a given section of a program, and/or execution time of that section. The program is first initialized with a series of calls to data collecting routines. These data are then interpreted to provide the overall results in terms of time spent in a function versus time spent during the calls to other functions. The information provided by profilers is only available at a relatively high level of abstraction, that is at a function level. 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 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 [18], a toolbox for optimizing memory I/O using geometrical model, which addresses memory related aspects of system-design, by supporting the data transfer and storage exploration methodology (DTSE) [19]. To take full advantage of the methodology, both in depth analysis and extensive transformations of the application’s program code are automatically performed by ATOMIUM. ATOMIUM operates at the behavioral level of an application, expressed in C. The output is a transformed C description, functionally equivalent to the original one, but typically leading to strongly reduced execution times, memory size and power consumption.

ATOMIUM allows designers to quickly identify memory related hotspots in the applications they are working on, such as:

- which data structures and arrays are, characterized by large data exchanges and with which functions;
- which functions, or function portions, require large memory access bandwidths;
- what is the run-time peak memory usage and when it occurs.

Other modules of the tool suite focus on the optimization of the storage bandwidth, of the storage size and of the address expression arithmetic. While ATOMIUM is a powerful tool for

data-transfer complexity analysis and optimization, it does not provide any means to perform a computational complexity analysis. Furthermore, the provided data-transfer analysis is based on a “flat” memory architecture model, which, for instance, does not allow taking into account the effects of introducing one or more cache memories in the memory hierarchy.

C. Hardware Description Languages and Hardware/Software Codesign Tools

Through synthesis and simulation, hardware description languages (HDLs) [20], [21] allow gathering very reliable results about the implementation complexity and performance of the system, because they can simulate the real final architecture. However, such results arrive too late in the design flow. The algorithms have to be translated from the general-purpose language specification of a verification model into an HDL description, implicitly implementing an underlying architecture. An almost complete rewriting of the HDL code might be necessary if it is realized that the *a priori* architectural choices are not appropriate for the algorithm at hand. In conclusion, a high-level measure of algorithmic complexity cannot be easily obtained by means of HDL descriptions.

Besides HDLs, there are tools which provide instruction-level simulation of DSPs or other type of embedded cores [22]–[26] allowing to estimate the performance of the implementation of an algorithm on a given target architecture. Other tools allow the designer to co-design and co-simulate heterogeneous embedded systems. They provide a more versatile framework in which it is possible to integrate hardware descriptions, software descriptions and instruction-level simulators, at different abstraction levels [27]–[37].

III. SIT FOR COMPLEXITY ANALYSIS

The approach, and associated tool called SIT, presented in this paper is has been developed with the goal of measuring the complexity of a specific implementation of an algorithm independently from the hardware architecture on which the software model is run. It is assumed that a software implementation of the algorithm, typically a software verification model, is available and that it can be run in realistic input data conditions. In other words, the interest is not only about the measure of the algorithmic complexity itself, but also about its dependencies under specific input data. This approach is in line with methodological approaches proposed for instance in [9] and [18], aiming at optimizing data transfers, memory bandwidths and storage requirements directly on algorithm specifications at high abstraction level.

Pure algorithmic complexity does not depend on any other factor than the algorithm description itself and the input data. Avoiding input data dependency would lead only to worst-case/best-case estimations, and these estimations, even though crucial for, e.g., real time control systems, in our opinion are not useful in the video/multimedia context.

The new approach of SIT [38] is based on 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 [39]. 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 source-code level, yet by means of simulations on real input-data. The verification model is considered as a sequence of abstract operations exactly as they can be seen while reading the source-code or, by the abstraction point of view, exactly as they were written when the application was being specified by means of the development of the verification model. 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 counted, therefore providing an exhaustive basis for computational complexity analysis. Besides, a customizable virtual memory architecture 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 instrumenting *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 technology can be summarized as follows.

- 1) Pure algorithmic complexity analysis at the highest possible abstraction level, that is at software programs’ source-code level. The analysis does not depend on the underlying platform or on the compilation but *only* on the source-code. This allows: 1) to provide complexity analysis results at the very beginning of the design cycle by performing the analysis directly on the verification models; and 2) to profile software programs at pure algorithmic level providing a reliable analysis basis for algorithmic optimizations.
- 2) Input-data dependent analysis. Nowadays applications cannot be studied statically: their behavior, hence their complexity, strictly depends on the processed input-data. Furthermore, the implementation of most applications is now based, rather than on the worst-case, on the Cost/QoS tradeoff, which implies the need of an input-data dependent analysis.
- 3) Completely automatic instrumentation process with no limitations on the ANSI C and K&R compliant C source code. SIT has been conceived to be an “easy-to-use” tool: the instrumentation of the source code appears to the user as a normal compilation process, without the need of modifying source files and makefiles or of having to type specific commands.
- 4) Fully customizable memory simulation, for a versatile data-transfer and storage analysis apt to explore different design-spaces in the memory architecture domain.
- 5) 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.

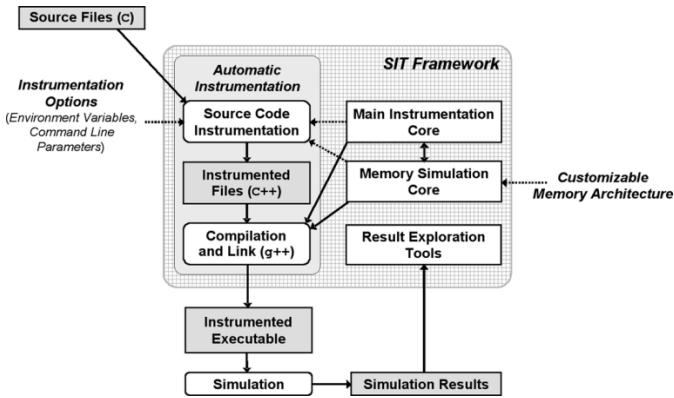


Fig. 2. SIT complexity analysis framework.

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 Fig. 2.

- 1) The *Main Instrumentation Core* and the *Memory Simulation Core* are the heart of SIT; they provide the necessary features for a correct instrumentation and simulation of the source code. The main instrumentation core is in charge of instrumenting all the data-types, operators, and statements of standard C code; since the C language is a constant, the main instrumentation core is constant as well and does not depend on any configuration settings. Conversely, the memory simulation core is based on a fixed set of interfaces, but it can be customized by defining different memory architectures; during simulation, the memory simulation core is driven by the main instrumentation Core.
- 2) The first step of the analysis is the *instrumentation of the C source code*. 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 C++ instrumented code is generated by means of the classes defined in the main instrumentation core and in the memory simulation core, and according to the instrumentation settings. The instrumented code is then compiled and linked with the instrumentation libraries, by means of GNU g++ compiler, generating eventually the instrumented executable.
- 3) 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 *Result Exploration Tools*.

A. Computational Complexity Analysis

During the execution of the instrumented version of the program, every executed operation is intercepted and a corresponding counter is incremented. The results are collected along two main axes: operations and data-types (e.g., operation “+” performed on “int” type). The set of intercepted operations is an extension of the 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) and the extra BOOL type, which does not belong to ANSI C standard; the BOOL type was introduced to explicitly differentiate the results for integer and logical expressions, since in C the *int* type is typically used in logical expressions. By counting all the operations performed during execution according to the bidimensional operations/data-types basis, SIT provides the finest grain information about the performed operations, which results in an exhaustive input basis for the computational complexity analysis. 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 traversing too (high execution-tree resolution).

Fig. 3 shows an example of computational complexity analysis results (the picture is a screenshot of SITView, the graphical result exploration tool 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 (ulong = unsigned long, pntnr = 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 and 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.

The main instrumentation core is in charge of providing the functionalities for the computational complexity analysis as well as the basic functionalities for the simulation, such as simulation initialization, execution tree tracing, result database management, and result file generation.

B. Data-Transfer and Storage Complexity Analysis

The data transfers and storage requirements, as discussed in the introductory and review sections of this paper, play a fundamental role in the evaluation of the algorithmic complexity of a system. In multimedia applications, for instance, 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 [9]. 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

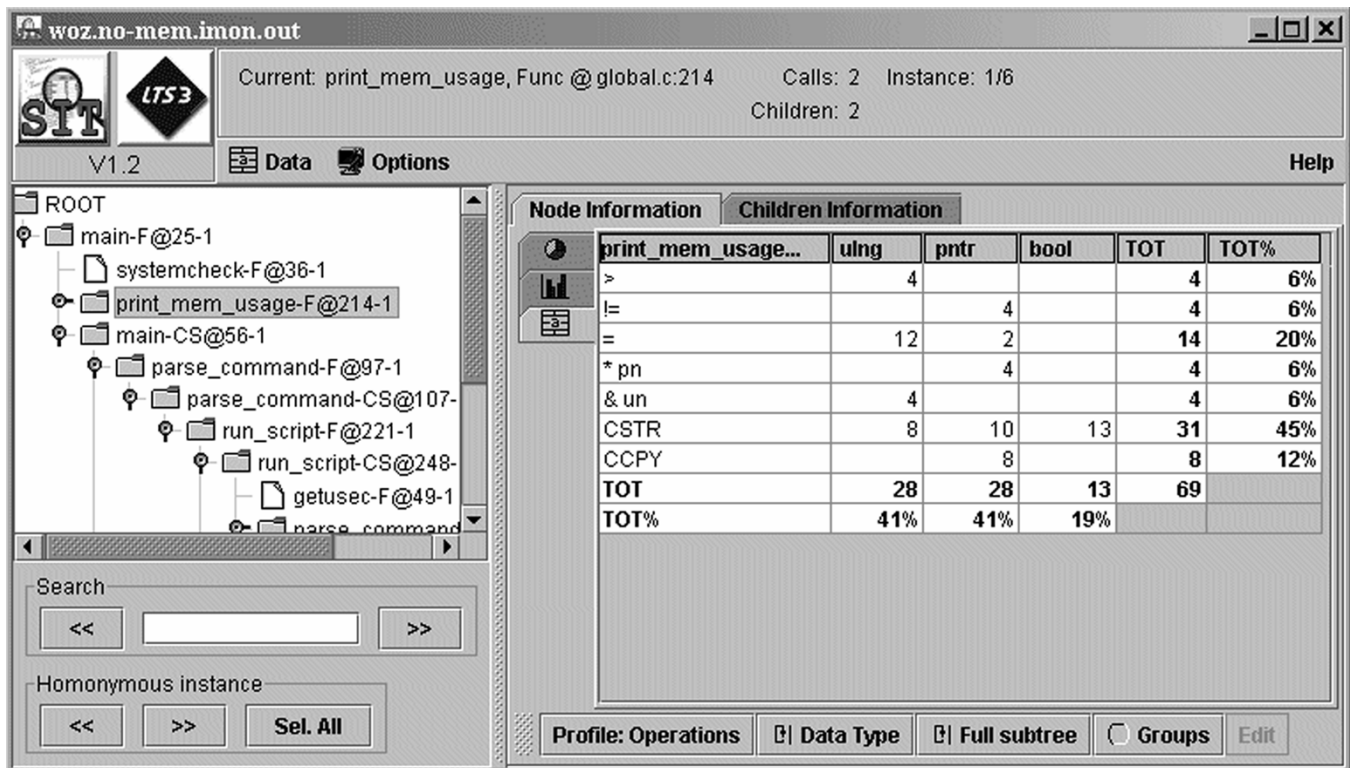


Fig. 3. Example of computational complexity analysis results.

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. It is an open framework, in the sense that it enables the simulation of freely customizable memory architectures, whose simulation results can be freely customized as well. The simulated memory architecture is composed of several memory models, each of them composed by different simulation modules (allocation managers, cache memories, storage memories—Fig. 4 shows as example of virtual memory architecture that can be simulated with SIT). A set of fixed interfaces is defined to specify the simulation modules, to “plug” them into each other, and to eventually specify a memory model; each single simulation module’s behavior and the number and type of results it produces at the end of the simulation, i.e., what is implemented “behind” the fixed interfaces, can be defined by the user with the highest possible degree of freedom.

Fig. 5 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

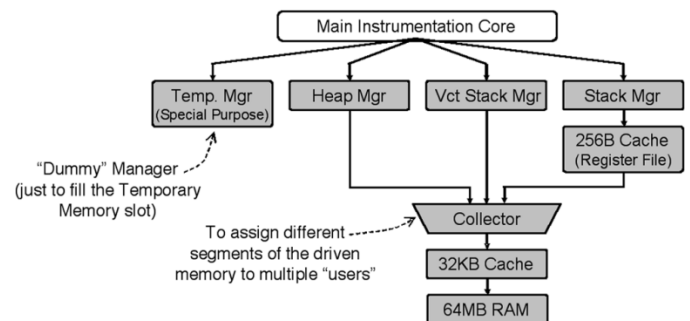


Fig. 4. Example of virtual memory architecture.

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.

The memory simulation core of SIT is based on the concept of memory model. A memory model is an entity in charge of the run-time simulation of the different functionalities of a memory hierarchy (Fig. 6), from the handling of the I/O and allocation requests to the final storage tasks, and of generating the corresponding data-transfer and storage complexity analysis results. The simulation of the virtual memory architectures relies on assigning variables and allocated memory segments to the memory models (“variable to memory” assignments, or *var-to-mem* assignments for short). A variable (or a dynamically allocated block) is assigned to a memory model so that during simulation all the I/O operations on that variable (or on that

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	

Fig. 5. Example of data-transfers and storage complexity analysis results.

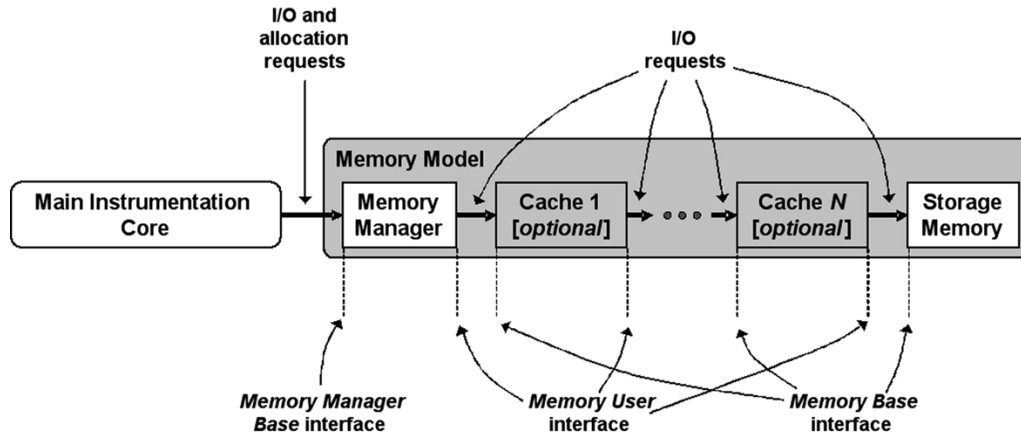


Fig. 6. Three basic interfaces for the specification of a memory model.

dynamically allocated block) drive the simulation of the corresponding memory model. Full support for any type of pointers, arrays, and complex data structures is provided, so that I/O operations are always correctly mapped onto the corresponding memory model, no matter through which combination of the aforementioned data-types the data-transfers are originated.

The virtual memory architectures are specified as sets of several memory models, as shown in Figs. 4 and 7. The simplest approach is to use independent memory models, whose behavior depends only on the commands coming from the main instrumentation core (Fig. 7). Nevertheless, nothing prevents from designing more complex memory architectures where the memory models communicate with each other (Fig. 4), or even full-custom memory simulation cores that are not built over the predefined structure based on memory models [Fig. 8(a)], therefore, further increasing the freedom in implementing custom data-transfer and storage complexity analysis cores. The variables can be assigned to memory models either automatically, by means of default *var-to-mem* assignments, or by specifying explicit *var-to-mem* assignments; in both cases, the *var-to-mem* assignments are taken care of during the instrumentation phase. Thanks to explicit *var-to-mem* assignments and to the possibility of using any number of memory models (Fig. 7, extended set), it is possible to focus the data-transfer analysis down on each single variable or allocated segment, e.g., to gather specific and detailed results for critical data.

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 instrumen-

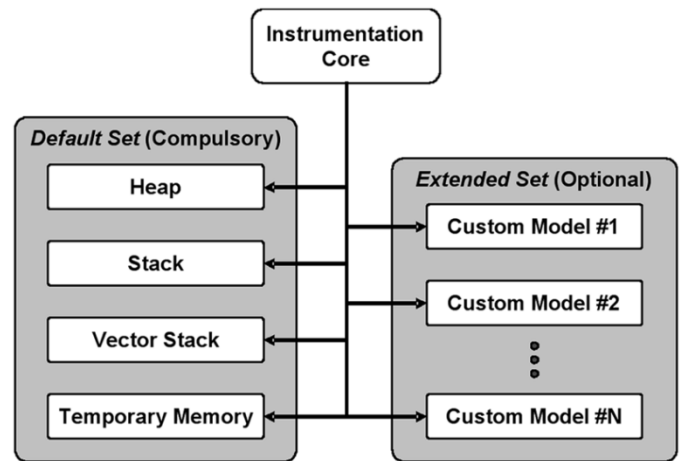


Fig. 7. Default structure of a simulated memory architecture is composed of the compulsory default set of memory models and of the optional extended set.

tation core and the memory simulation core [Fig. 8(b)]. 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 a validated and robust instrumentation and analysis framework for developing new simulation and analysis tools, as shown in Fig. 9. Possible solutions for developing new custom tools comprise modifying the instrumentation process (e.g., to add

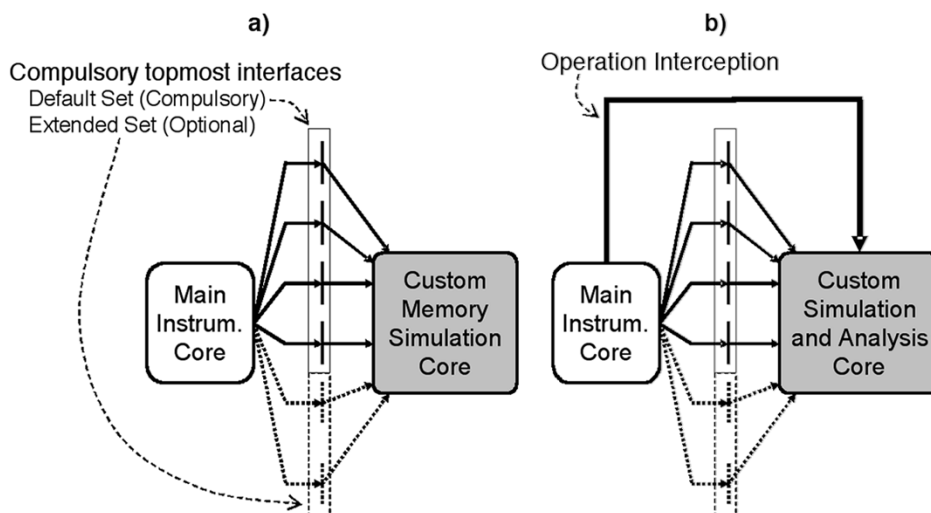


Fig. 8. Custom memory simulation cores and custom simulation and analysis cores. (a) Custom memory simulation core (without memory models). (b) Custom simulation and analysis core.

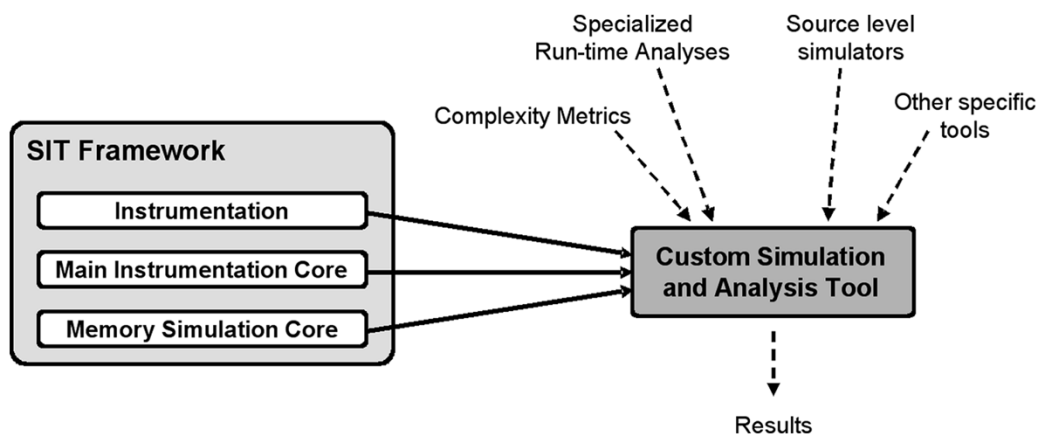


Fig. 9. SIT as a framework for building custom simulation and analysis tools.

extra instrumentation code), or inserting specific commands in the source code of instrumented types' classes (e.g., to drive the custom simulator whenever a given operation is intercepted by SIT), or developing specialized memory models (e.g., to make the custom simulation depend on I/O events). Obviously, any combination of the previous solutions is possible as well, enabling the development of complex run-time simulation tools, for complexity evaluations according to specific metrics. Two examples of custom simulation and analysis tools currently under development, built over SIT framework, are presented in next subsections.

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 [40].

C. Dynamic Critical Path Evaluation for Potential Operation Parallelism Estimation

The goal of the proposed critical path [41] evaluation methodology is providing an estimate of the potential operation parallelism at the early stage of the design, in order to be able to take

meaningful and efficient partitioning decisions and bring them to actual parallel implementations. A preliminary study on the evaluation of the critical path on the data flow execution graph (DFEG) of a C program has been carried out to validate an efficient dynamic critical path evaluation methodology [42]. This methodology is conceived for being implemented by means of SIT, as the dynamic critical path evaluation is based on run time operation and data transfer interception.

The critical path profiling is a metrics explicitly developed for parallel programs [43]. The critical path profile is a list of procedures and of the time each procedure contributed to the length of the critical path. Critical path profiling is a way to identify the component in a parallel program that limits its performance. It is an effective metric for tuning parallel programs and is especially useful during the early stages of tuning a parallel program when load imbalance is a significant bottleneck. It also helps to find out which components should be prioritized to complete the program execution in time. When a task has to be completed in a given time, the critical path analysis helps to focus on the essential activities to which attention and resources should be devoted.

The majority of already developed tools aim at the critical path profiling for tuning existing parallel programs executed

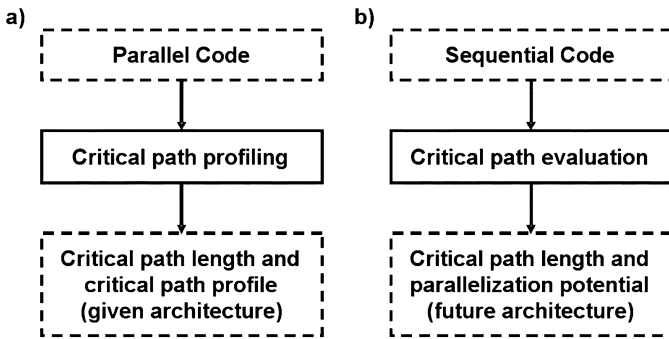


Fig. 10. Critical path profiling (a) of parallel code on event graphs versus critical path evaluation and (b) of sequential code on data dependences graphs.

on existing architectures [Fig. 10(a)]. The proposed new critical path model metrics allows finding out at which degree the different parts of a given algorithm can be potentially parallelized [Fig. 10(b)]. The critical path length and the complexity are computed by means of the same set of complexity weights. The maximum value of the critical path length equals the total complexity in the worst case, i.e., when a pure sequential data dependence exists among all operations. The ratio between the C code complexity and the critical path length is an index of the parallelization potential of the algorithm; actually this index is an estimate of the (weighted) number of operations that can be potentially executed in parallel, as it corresponds to ratio between the total (weighted) number of operations and the maximum (weighted) number of operations that have to be performed sequentially. The principles constituting the basis for the critical path model metric definition on a sequential C code are the following.

- 1) The critical path is defined on the C code's execution data-flow.
- 2) The critical path length and the system parallelization potential are defined in terms of C language basic operations and I/O operations complexity. The parameters of the machine executing the instrumented C code during evaluating the critical path are not taken into account.
- 3) In the definition of the critical path, the DFEG [42] is generated dynamically while running an algorithm on a real input-data set. The DFEG is used for the critical path definition instead of the traditional data flow graph.

This approach presents several advantages with respect to other methods.

- 1) It can be applied to large and complex software programs for which a static generation of the DFEG would not be feasible, since the number of nodes in the DFEG is equal to the number of operation performed during the execution.
- 2) It evaluates the critical path at run-time in real working conditions. Conversely, this is not possible with static methods based on DFG.
- 3) It allows critical path evaluations at operation level even in the case in which data-flows are obfuscated by the use of pointers and "anonymous" data (e.g., elements in an array).

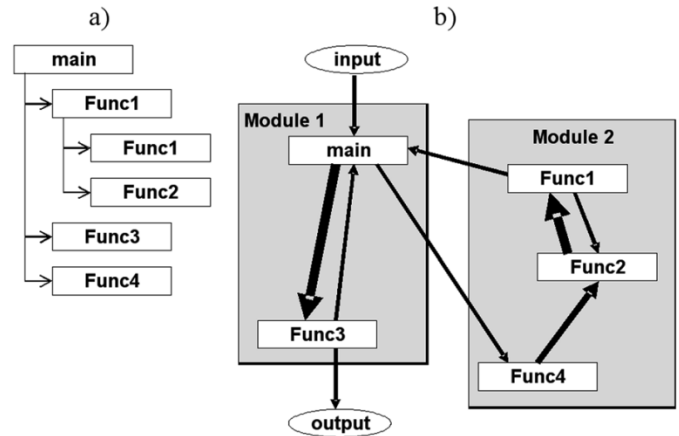


Fig. 11. Function-call tree (a) does not help detecting the actual data-transfer dependences functions the functions and grouping the functions in functional modules (b). (a) Function-call tree. (b) Functions, functional modules, and data-transfer dependences.

D. Automatic Measurement of Interfunction Data-Transfers for Explicit Assessment 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 among 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, as 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.

As shown in Fig. 11, explicit measurements of the interfunction data-transfers are a meaningful basis for high-level 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 interfunction 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.

A custom simulation core is currently being developed for automatically generating the interfunction data-transfer graph by means of the memory simulation and execution tracing capabilities of SIT. In the next phase, a module for automatic analysis of the interfunction data-transfer graphs will be developed for automatic detection of the functional modules.

TABLE I
CHARACTERISTICS OF THE TEST SEQUENCES

Test Case	Frame rate (fps)
Mother & Daughter	30
Foreman	25
Calendar & Mobile	15

TABLE II
MPEG-4 VM—PERFORMANCE

Test Case	Bitrate (kbps)	PSNR Y (dB)	PSNR U (dB)	PSNR V (dB)
Mother & Daughter	82	34.689	39.798	40.514
Foreman	167	30.912	35.197	35.620
Calendar & Mobile	195	23.182	28.405	27.256
<i>Average</i>	<i>148</i>	<i>29.594</i>	<i>34.467</i>	<i>34.463</i>

IV. EXAMPLES OF COMPLEXITY MEASURES PROVIDED BY THE SIT TOOL

This section presents some examples of the type, accuracy, and metrics of results produced by the computational complexity analysis with SIT. The examples are obtained using as reference model the generically optimized MPEG-4 codec (simple profile) constituting Part 7 of the standard currently under finalization [3], [44] and of the H.264 AVC codec, [45] constituting the Part 10 of MPEG-4 standard. The analysis reports the number of operations performed during program execution in real working conditions.

A. Testbench Definition

The reference software's used for this example of complexity analysis are the MPEG-4 Verification Model generically optimized for simple profile (OptSimple) version FPDAM1-020414 [44] and the JVT Joint Model JM 2.1 [45].

The used testbench consists of three reference sequences (Table I): Mother & Daughter is a low complexity head-and-shoulders sequence. Foreman has a medium complexity and Calendar & Mobile is a high complexity sequence, with different movements including rotations.

JVT JM has been tested with two different encoding settings:

- 1) "simple" settings: in this configuration JM performance is similar, in terms of PSNR and bit-rate, to that of MPEG-4 simple profile VM. This configuration is meant to compare the algorithmic complexity of the two codecs in similar working conditions.
- 2) "complex" settings: this configuration activates all JMs coding tools. The analysis shows how the improvement of JMs compression performance, due to the "complex" settings, increases the algorithmic complexity of the overall coding and decoding processes.

The tests were run on sequences of 100 frames. Tables II–IV show the coding results for MPEG-4 VM, JVT JM in "simple" configuration and JVT JM in "complex" configuration.

TABLE III
JVT JM, "SIMPLE" SETTINGS—PERFORMANCE

Test Case	Bitrate (kbps)	PSNR Y (dB)	PSNR U (dB)	PSNR V (dB)
Mother & Daughter	46	32.372	40.130	41.076
Foreman	141	30.900	38.313	39.718
Calendar & Mobile	243	26.537	31.928	31.261
<i>Average</i>	<i>143</i>	<i>29.936</i>	<i>36.790</i>	<i>37.352</i>

TABLE IV
JVT JM, "COMPLEX" SETTINGS—PERFORMANCE

Test Case	Bitrate (kbps)	PSNR Y (dB)	PSNR U (dB)	PSNR V (dB)
Mother & Daughter	13	34.213	40.145	41.238
Foreman	86	32.241	38.761	40.606
Calendar & Mobile	102	27.537	33.102	32.649
<i>Average</i>	<i>67</i>	<i>31.330</i>	<i>37.336</i>	<i>38.164</i>

TABLE V
ENCODERS—COMPUTATIONAL COMPLEXITY

Operations	MPEG4 VM # ops	JVT JM, "simple" settings		JVT JM, "complex" settings	
		# ops	Ratio vs VM	# ops	Ratio vs VM
Arithmetic	2.76E10	6.52E10	2.36	2.77E11	10.01
Comparison	7.58E09	6.31E09	0.83	6.16E10	8.12
Logical	1.15E08	2.46E08	2.15	8.99E09	78.43
Memory	2.72E10	5.71E10	2.10	5.36E11	19.71
<i>All ops</i>	<i>6.26E10</i>	<i>1.29E11</i>	<i>2.06</i>	<i>8.84E11</i>	<i>14.13</i>

This table shows the number of operations and their ratios vs the results obtained for MPEG-4 VM (average figures over three tests sequences)

B. Computational Complexity Analysis

This subsection presents an example of the results obtained by the tool that can be used for an in depth computational complexity analysis of the encoding and decoding of the three reference sequences. The results are collected separately for each operation and data-type (corresponding to C language's operators and data-types). For the sake of conciseness, in the following tables the results are presented independently of the data-types and are grouped in the following four sets of operations.

- 1) *Comparison operations*: they correspond to C operators "=", "!", ">", ">=", "<", and "<=".
- 2) *Logical operations*: they correspond to C operators "!", "&&", "||".
- 3) *Memory operations*: they correspond to C operators "=", "[", "– >" and the pointer dereferencing operator "*".
- 4) *Arithmetic operations*: all the other operations.

C. Encoders

Considering the results for encoding the three reference sequences, each of them in the three testbench cases, an example of a preliminary coarse-grain comparison of the three encoders is shown in Table V and Fig. 12. The computational complexity

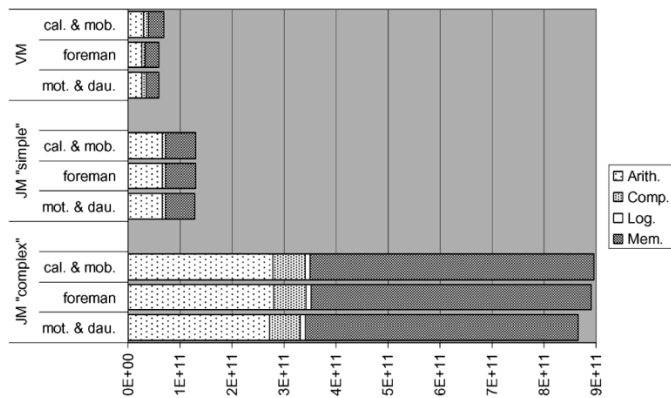


Fig. 12. Encoders—computational complexity.

TABLE VI
DECODERS—COMPUTATIONAL COMPLEXITY

Operations	MPEG4 VM	JVT JM, "simple" settings		JVT JM, "complex" settings	
	# ops	# ops	Ratio Vs VM	# ops	Ratio vs VM
Arithmetic	1.42E09	2.97E09	2.09	5.04E09	3.54
Comparison	4.33E08	8.80E08	2.03	1.59E09	3.67
Logical	2.33E06	3.33E07	14.30	3.37E07	14.48
Memory	1.52E09	2.64E09	1.74	3.51E09	2.31
All ops	3.38E09	6.53E09	1.93	1.02E10	3.01

This table shows the number of operations and their ratios vs the results obtained for MPEG-4 VM (average figures over three tests sequences)

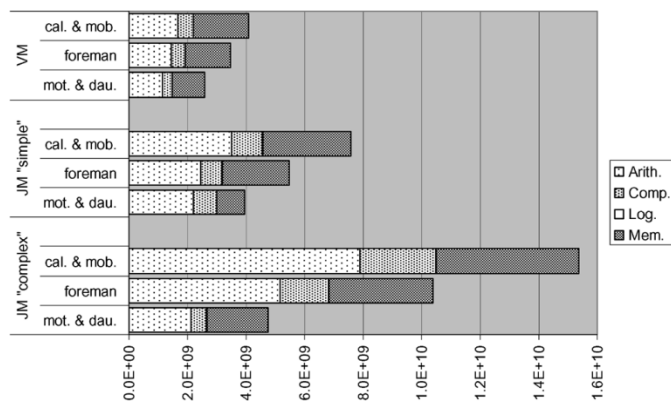


Fig. 13. Decoders—computational complexity.

of MPEG-4 VM is about half of that of JVT JM in "simple" configuration, while the cost of the performance gain of the JM in "complex" configuration (more specifically a reduction of the bit-rate of more than 50% and a PSNR gain of almost 2 dB) is an increase of more than one order of magnitude of the computational complexity.

D. Decoders

Similarly to the comparison described at previous paragraph for the encoders, Table VI and Fig. 13 and show the computational complexity analysis results for the decoders in the three cases. The computational complexity of the JM in "simple" configuration is about the double of that of the VM, as in the case of the encoder. Conversely, the computational complexity of the JM in "complex" configuration is about three times higher than that of the VM, while in the case of the encoder the increase is of

more than one order magnitude. Furthermore, the results clearly show that the computational complexity of decoding process, in all the three cases, is more sensitive to the visual complexity of the sequences than that of the encoding process. Finally, "arithmetic" and "memory" operations are the most frequent ones, as in the case of the encoders.

The results provided here constitute the overall summary of all codec executions, in reality the tools provides a full detailed analysis of both computational complexity and data-flow exchanges at all levels of the execution tree, thus enabling specific analysis based on "coding modes" comparisons, in order to explore implementation complexity/coding efficiency tradeoffs or other optimization goals.

V. CONCLUSION

This paper, after a brief review of the motivations and *state-of-the-art* approaches to complexity analysis for multimedia system design, has introduced a tool for the complexity analysis of C reference descriptions. 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 and performances of the algorithm under study on user configured memory architectures. Some examples of the richness of the run-time complexity metrics obtainable has also been provided as well as a description of the functionality, measures, and metrics extensions which will be possible with a further development of the SIT virtual simulator framework.

REFERENCES

- [1] P. V. Knudsen and J. Madsen, "Aspects of system modeling in hardware/software partitioning," in *Proc. 7th IEEE Int. Workshop on Rapid Systems Prototyping*, Thessaloniki, Greece, Jun. 1996, pp. 18–23.
- [2] "Information Technology—Generic Coding of Moving Pictures and Associated Audio Information—Part 2: Video," International Organization for Standardization, Tech. Rep. 13 818-2, 1994. ISO/IEC.
- [3] *Information Technology—Coding of Audio Visual Objects—Part 2 Visual*, ISO/IEC 14496-2 (MPEG-4).
- [4] *Information Technology—Coding of Audio Visual Objects—Part 10 Advanced Video Coding*, ISO/IEC, ISO/IEC International Standard 14 496-10 and ITU-T Recommendation H.264.
- [5] *JPEG 2000 Part 1 Final Publication Draft*, ISO/IEC JTC1/SC29/WG1 N2678, C. Boliek, C. Christopoulos, and E. Majani, Eds., Jul. 2002.
- [6] M. Horowitz, A. J. F. Kossentini, and A. Hallapuro, "H.264/AVC baseline profile decoder complexity analysis," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 704–716, Jul. 2003.
- [7] V. Lappalainen, A. Hallapuro, and T. Hamalainen, "Complexity of optimized H.26L video decoder implementation," in *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, Jul. 2003, pp. 717–725.
- [8] M. Berekovic, H. J. Stolberg, and P. Pirsch, "Multicore system—On-chip-architecture for MPEG-4 streaming video," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, no. 8, pp. 688–699, Aug. 2002.
- [9] L. Nachtergaele, D. Moolenaar, B. Vanhoof, F. Catthoor, and H. De Man, "System level power optimization of video codecs on embedded cores: A systematic approach," *J. VLSI Signal Process.*, vol. 18, pp. 89–109, 1998.
- [10] M. Mattavelli and S. Brunetton, "Implementing real time video decoding on multimedia processors by complexity prediction techniques," *IEEE Trans. Consumer Electron.*, vol. 44, no. 8, pp. 760–767, Aug. 1998.
- [11] P. Kuhn, *Algorithms, Complexity Analysis, and VLSI-Architectures for MPEG-4 Motion Estimation*. Norwell, MA: Kluwer, 1999, ch. 3.
- [12] Y. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 16, no. 12, pp. 1477–1487, Dec. 1997.

- [13] P. Pushner and C. Koza, "Calculating the maximum execution time of real time programs," *J. Real Time Syst.*, vol. 1, pp. 160–176, Sep. 1989.
- [14] S. Mallat and F. Falzon, "Analysis of low bit rate image transform coding," *IEEE Trans. Signal Process.*, vol. 46, no. 4, pp. 1027–1042, Apr. 1998.
- [15] M. Mattavelli and S. Brunetton, "A Statistical Study of MPEG 4 VM Texture Decoding Complexity," Tampere, XQXQXQ, Finland, Tech. Rep. M924, ISO IEC/JTC1/SC29/WG11, MPEG 4, Jul. 1996.
- [16] E. Kligerman and D. Stoyenko, "Real-time euclid: A language for reliable real time systems," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 9, pp. 941–949, Sep. 1986.
- [17] S. Graham, P. Kessler, and M. McKusick, "Gprof: A call graph execution profiler," in *Proce. Symp. Compiler Construction (SIGPLAN)*, vol. 17, Jun. 1982, pp. 120–126.
- [18] What is ATOMIUM?. [Online]. Available: <http://www.imec.be/design/multimedia/atomium/>
- [19] F. Catthoor, F. Balasa, K. Danckaert, E. De Greef, M. Eyckmans, F. Franssen, M. Janssen, S. Janssen, M. Miranda, L. Nachtergaele, H. Samsom, P. Slock, and S. Wuytack, "Optimization of global data transfer and storage organization for decreased area and power in data dominated real time processing systems," IMEC Internal Overview Report of Data Transfer and Storage Exploration Research, Nov. 1998.
- [20] *IEEE Standard for Verilog Hardware Description Language 2001*, IEEE Standard 1364 2001, Product SH94921 TBR, 2001.
- [21] *IEEE Standard VHDL Language Reference Manual 2002*, IEEE Standard 1076 2002, Product SH94983 TBR, 2002.
- [22] T. Ball and J. Larus, "Optimally profiling and tracing programs," *ACM Trans. Programming Languages Syst.*, vol. 16, no. 4, pp. 1319–1360, Jul. 1994.
- [23] R. Cmelik and D. Keppel, "SHADE: A fast instruction set simulator for execution profiling," in *Proc. 1994 ACM Conf. Measurement Modeling of Computer Systems*, 1994, pp. 128–137.
- [24] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus, "Wisconsin wind tunnel II: A fast and portable parallel architecture simulator," *IEEE Concurrency*, vol. 8, no. 4, pp. 12–20, Oct.-Dec. 2000.
- [25] A. D. Pimentel and L. O. Hertzberger, "Abstract workload modelling in computer architecture simulation," in *Proc. 24th ACM/IEEE Int. Symp. Computer Architecture*, Denver, CO, Jun. 1997, pp. 6–14.
- [26] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation," in *Proc. 1996 ACM SIGMETRICS Conf. Measurements and Modeling of Computer Systems*, May 1994, pp. 128–137.
- [27] P. Alexander, R. Kamath, and D. Barton, "System specification in rosetta," in *Proc. IEEE 7th Int. Conf. Workshop Engineering of Computer Based Systems Symp.*, Edinburgh, U.K., Apr. 2000, pp. 299–307.
- [28] G. Berry, "The foundations of esterel," in *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. Cambridge, MA, 1998.
- [29] CoWare N2C Design System [Online]. Available: www.CoWare.com
- [30] J. Davis II *et al.*, "Overview of the Ptolemy Project," Dept. EECS, University of California, Berkeley, CA 94720, Tech. Rep. UCB/ERL n. M99/37, Jun. 1999.
- [31] M. Eisenring, J. Teich, and L. Thiele, "Rapid prototyping of dataflow programs on hardware/software architectures," in *Proc. Hawaii Int. Conf. Systems Science*, Kona, HI, Jan. 1998, pp. 187–196.
- [32] R. Esser, "CodeSign—Concepts and tutorial," Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, Lausanne, Switzerland, 1996.
- [33] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real time executives: A seamless flow of graphs transformations," in *Proc. Formal Methods and Models for Codesign Conf.*, Mont Saint Michel, France, Jun. 2003, p. 123.
- [34] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau, "SPARK: A high level synthesis framework for applying parallelizing compiler transformations," in *Proc. 16th Int. Conf. VLSI Design*, Jan. 2003, p. 461.
- [35] RASSP Methodology—Version 2.0 (1995, Oct.). [Online]. Available: <http://www.eda.org/rassp/>
- [36] "SystemC v2.0.1 White Paper, Open SystemC Initiative," <http://www.systemc.org>, May 2001.
- [37] B. Tabbara, L. Lavagno, and A. S. Vincentelli, "Fast hardware software Co simulation using software synthesis and estimation," in *Proc. IEEE Int. High Level Design Validation and Test Workshop*, 1997, pp. 149–156.
- [38] M. Ravasi and M. Mattavelli, "High-level algorithmic complexity evaluation for system design," *J. Syst. Architecture*, vol. 48/13–15, pp. 403–427, May 2003.
- [39] P. Kuhn, *Algorithms, Complexity Analysis, and VLSI-Architectures for MPEG-4 Motion Estimation*. Norwell, MA: Kluwer, 1999, ch. 4.
- [40] M. Ravasi, M. Mattavelli, P. Schumacher, and R. Turney, "High-level algorithmic complexity analysis for the implementation of a motion-JPEG2000 encoder," in *Proc. Integrated Circuit and System Design*, 2003, pp. 440–450.
- [41] L. Liu, D. Du, and H. C. Chen, "An efficient parallel critical path algorithm," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 13, no. 7, pp. 909–919, Jul. 1994.
- [42] A. Prihozhy, M. Mattavelli, and D. Mlynek, "Data dependences critical path evaluation at C/C++ system level description," in *Proc. 13th Int. Workshop Power and Timing Modeling Optimization and Simulation*, Sep. 2003, pp. 569–579.
- [43] J. K. Hollingsworth, "Critical path profiling of message passing and shared memory programs," *IEEE Trans. Parallel Distributed Syst.*, vol. 9, no. 10, pp. 1029–1040, Oct. 1998.
- [44] Generically optimized MPEG 4 reference software (simple profile), Dept. of Electronics Engineering, National Chiaio Tung University, Taiwan, R.O.C. [Online]. Available: http://megaera.ee.nctu.edu.tw/mpeg/Optimized_Ref_Software/MoMuSys_FPDAM1_1.0_020414_nctu.zip
- [45] Reference Software JM 2.1 [Online]. Available: <http://bs.hhi.de/~suehring/tml/download/jm21.zip>



Massimo Ravasi received the degree in electrical engineering from Politecnico di Milano, Milano, Italy, and the Ph.D. degree for his work in the software instrumentation tool (SIT) project, focusing on the development of an automatic tool for high-level algorithmic complexity analysis for system design, in September 2003.

From October 1997 to March 1998, he was with the Laboratorio S.I.A., Politecnico di Milano, as a part-time collaborator in the field of teleaching systems. In April 1998, he joined the Signal Processing Laboratory (LTS) of the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, as a Research Assistant working on the development of a hardware JPEG 2000 codec in the Mitocoma project.



Marco Mattavelli was born in Milano, Italy, on July 18, 1961. He received his Diploma of electrical engineering degree from the Politecnico di Milano, Milano, Italy, in March 1987.

In 1988, he joined the Philips Research Laboratories, Eindhoven, The Netherlands, in the framework of EUREKA-95 (HDMAC) project. Main research activities regarded channel and source coding for optical recording and electronic photography. In 1990, he joined the CSA Philips Research Division of Monza, Italy, working on signal processing of TV and HDTV signals. In October 1991, he joined the Signal Processing Laboratory (LTS) of the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, where he received the Ph.D. degree in 1996 with the thesis: "Motion analysis and estimation: From ill-posed discrete inverse linear problems to MPEG-2 coding." At EPFL, he has been involved in various European projects of 4th and 5th framework (VADIS, COUGAR), research and didactic activities. In 1995, he was Visiting Researcher at the Center of Operational Research and Applied Mathematics, Cornell University, Ithaca, NY. In July 1996, he joined the Integrated System Laboratory (LSI) of EPFL where he has been involved in ATLANTIC, EMPHASIS, OCCAMM, CARROUSO, MOSES, and ENTHRONED European projects of 5th and 6th Framework, all projects dealing with specification and implementation of multimedia systems. He has been involved in several collaboration with industries and in the ISO/IEC JTC1/SC29/WG11 standardization activities (better known as MPEG), for which he is currently Chairman of the Implementation Study Group (ISG). For his work and contributions on the standardization of MPEG-4, he received the ISO/IEC Award in 1998 and in 2001. His major research activities and interests include architectures and systems for video coding, real-time multimedia systems, high-speed image acquisition and video processing, motion analysis and estimation, neural networks for image and signal processing, applications of combinatorial optimization to signal processing. He holds patents in the multimedia and video processing fields. He is the author or coauthor of more than 80 research papers and one book.