Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

ELSEVIER

Journal of Systems Architecture 48 (2003) 403–427

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

High-level algorithmic complexity evaluation for system design

Massimo Ravasi *, Marco Mattavelli

Signal Processing Laboratory (LTS3), Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne, Switzerland

Received in revised form 20 February 2003; accepted 20 February 2003

Abstract

The increasing complexity of processing algorithms has led to the need of more and more intensive specification and validation by means of software implementations. As the complexity grows, the intuitive understanding of the specific processing needs becomes harder and harder. Hence, the architectural implementation choices or the choices between different possible software/hardware partitioning become extremely difficult tasks. Moreover, it is also desirable to understand and measure the algorithm complexity at the highest possible level near to the *algorithmic level* so as to be able to take the more appropriate actions. Automatic tools to perform such analysis become nowadays a fundamental need.

In this paper, the requirements of a suitable algorithmic complexity evaluation technology are discussed, with a particular emphasis to the problem of the analysis of multimedia systems and signal processing algorithms. A brief review about limitations and weaknesses of existing tools is given, specifying the characteristics of ideal "complexity evaluation systems". A new approach is described, called here *Software Instrumentation Tool*, SIT, yielding an automatic software tool able to extract information not depending on the simulation platform, keeping into account specific input data and resulting in a good and useful measure of the desired *high-level* algorithmic complexity.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Complexity analysis; System design; Software/hardware partitioning; Software instrumentation

1. Introduction

1.1. Goals

The evolution of digital silicon technology enables the implementation of signal processing algorithms that have reached extremely high levels of complexity. This fact, among others, has two relevant consequences for the system designer. The first is that processing algorithms cannot be specified in ways other than developing a reference software description. The second important consequence is that the understanding of the algorithms and the evaluation of their complexity have to be derived from such software

* Corresponding author. Tel.: +41-21-693-69-80; fax: +41-21-693-46-63.
E-mail address: massimo.ravasi@epfl.ch (M. Ravasi).

1383-7621/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S1383-7621(03)00038-9

description. As consequence of the greatly increased complexity, the generic intuitive understanding of the underlying processing become a less and less reliable design approach. Besides, considering the shortening of time to market, it is not possible to design a new processor from scratch without a massive investment and a group of hundreds of motivated engineers [12]. Considering that, in many cases, the complexity of the processing is also heavily input data dependent, the system designer faces a very difficult task when beginning the design of a system architecture aiming at efficiently implementing the processing at hand.

This difficulty is evident when considering for instance the case of software/hardware co-design for system on chip integration. Fig. 1 shows a typical design flow for this implementation case. All the relevant information needs to be extracted by the software description that might be constituted by several thousands of computer program lines. Indeed, the analysis of the complexity of single *functions* does not give any information without the knowledge of the interconnection, occurrence and actual use of all *functions* composing the algorithm. Some other traditional styles of design such as complexity analysis based on *pencil and paper* or *worst case processing* applied to some portions of the algorithm, not only become more and more impractical for the effort required, but can also results in very inaccurate results for not taking into account the correct dependency of the complexity from the input data to be processed.

The results of this problematic preliminary analysis are then used for the software/hardware task partitioning. This is the initial step of the design flow where the final step is a full blown simulation of the resulting optimized software/hardware embedded system. All these steps involve considerable efforts, and due to a lack of precise initial information, erroneous preliminary task partitioning is done, generally leading to inefficient or sub-optimal design results. For this reason, costly iterations through the design process are needed to achieve good results.

It can be noticed that for software/hardware co-design, i.e. for synthesis and simulation with hardware description languages, instruction-level simulation and software optimization on embedded processors [2,4,23,28,38] and the overall modeling, design and simulation of heterogeneous systems [5,7,9,10,34,35,37], a large variety of tools is available at all levels. Conversely no suitable automatic tools are available to assist the fundamental task partitioning stage or to gather detailed and reliable information on the complexity of the algorithm for optimizing the implementation, starting from the generic software description.

All these considerations, although relevant for most of signal processing implementation problems, become fundamental for video–audio and multimedia coding, where the last generation of compression standards (i.e. MPEG-2 [6], MPEG-4 [8]) reaches a very high level of complexity that is also extremely

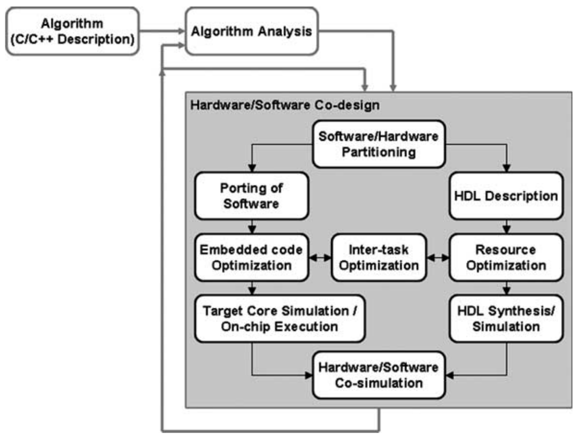

Algorithm (C/C++ Description)
Algorithm Analysis
Hardware/Software Co-design
Software/Hardware Partitioning
Porting of Software
HDL Description
Embedded code Optimization
Inter-task Optimization
Resource Optimization
Target Core Simulation / On-chip Execution
HDL Synthesis/ Simulation
Hardware/Software Co-simulation

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

sensitive to the encoder optimization choices and strongly data dependent. The content of the video–audio material and the coding options used to compress the data produce, indeed, results with very large ranges of complexity. Hence the analysis of the decoding process for a compressed bit-stream must be performed on a solid statistical basis. Such analysis is feasible only by means of automatic tools able to measure all the relevant aspects of the algorithm complexity, when the algorithms are applied to process meaningful input data sets.

Since, obviously, an in-depth understanding of the algorithm complexity remains a fundamental issue in any system design process, for the lack of suitable tools at this level it is not a surprise that system designers nowadays face great difficulties in extracting information about the complexity and structure of algorithms. For instance questions such as: *how many operations*, of *which type*, on *which type of data*, using *how many memory accesses*, *which processing functions and type of data* are necessary to correctly perform the algorithm, are definitely not easy to be answered. However, they are fundamental for the design of efficient processing architectures that aim to match the processing requirements. Fig. 2 illustrates that having this information in advance and as a reliable support to the software/hardware task partitioning and task optimization can reduce or even eliminate the need of costly and time consuming re-design iterations.

Obviously, having precise and reliable information about the process that has to be implemented, the initial architectural decisions and/or software/hardware task partitioning step can be greatly facilitated. Decisions can be drawn from algorithmic complexity evaluations based on inter-module bandwidth, shared memory bandwidth, operation and data type statistics. The same type of analysis is also useful for other system optimization tasks such as memory and power dissipation minimization that require several methodological steps starting from a generic algorithm specification [3,26]. An automatic tool supporting the designer skills is the solution to the main drawback of such approaches, constituted by the efforts and design time to accomplish the necessary steps.

Unfortunately, measures of these quantities on specific general-purpose hardware architectures used as simulation support might not be useful to understand the real processing needs and could be, for some aspects, even misleading. Measures of algorithmic complexity are needed at a *pure* algorithmic level. Information based on an analysis at assembly language level after the compilation on specific hardware architectures with all related compiler optimizations and specificities are certainly much less useful and relevant at the beginning of the architectural design flow.

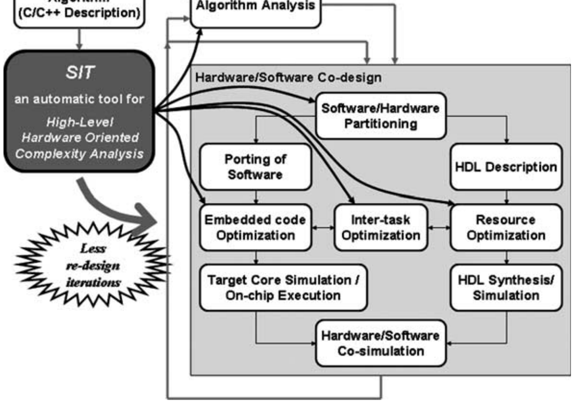

Algorithm (C/C++ Description)
Algorithm Analysis
SIT
an automatic tool for
High-Level Hardware Oriented Complexity Analysis
Hardware/Software Co-design
Software/Hardware Partitioning
Porting of Software
HDL Description
Embedded code Optimization
Inter-task Optimization
Resource Optimization
Target Core Simulation / On-chip Execution
HDL Synthesis/ Simulation
Hardware/Software Co-simulation
Less re-design iterations

Fig. 2. One goal of the SIT approach presented in this paper is to provide the right information for a "tool assisted" initial software/hardware partitioning and task optimization, reducing or even eliminating the need of re-design iterations in the design process loop.

2. State-of-the-art approaches to complexity analysis

Depending on the specific goals of the desired complexity analysis to be performed, very different approaches and tools can be chosen [16]. These can be roughly classified into five categories.

- *Profilers*, modifying the program to make it produce run-time data [11,13,17].
- *Compilers*, applying result-equivalent code replacements [1].
- *Static methods*, getting information from the source code, such as lines of code or basic blocks for instance. For these methods state of the art solutions rely on annotation at high-level programming language for explicit or implicit enumeration of program paths so as to determine lower and upper bounds of running time over a given processor [18].
- Descriptions by means of hardware description languages such as *VHDL* or *Verilog* enable other aspects of the complexity evaluation.
- Hardware specific tools, providing computational information to some extent.

2.1. Profilers

Profilers 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. Making this kind of modification of a program, great care has to be taken in order to avoid that the execution time of the data collecting part of the instrumented program influences the resulting statistical information about the algorithm. Most of the time, the data collecting routines are designed to run in a fixed and constant time, or the time consumed by this collection function is also given in the evaluation statistic. However, the information provided by such profilers is only available at a relatively high level, in other words 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 the algorithm. If only function-level information is provided, a complete rewriting of the program code replacing each elementary operation with a function call is necessary to obtain accurate statistics of the executed operations. If the timing information is available, an appropriate rewriting of the code could also enable the estimation of the relative cost of the considered operations. Only a few profilers are able to provide some relative timing information on a per source-code-line basis. This information is collected, in a statistical way, observing the program counter register of the processor at regular time intervals, and then mapping the memory locations to the corresponding source code lines. This information can be placed at a lower level than function level but it is less reliable, since it is obtained on a statistical basis. Operations that are frequently executed are accurately described, because the statistical evaluation is performed on a large set of samples. Less frequently used functions may lead to erroneous information. Furthermore, it is up to the user of the tool to figure out which operations are executed at high computational costs, basing the analysis on the statistical data. The automatic part of the tool only leads to a line of source code and not to simple operations. Therefore, profilers are really 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. However, 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. At the beginning of the design cycle a generic software specification of the algorithm is available and the goal is implementing it on a suitable architecture and not getting measures on general purpose computers with no relation with the final software/hardware implementation.

2.2. Compilers

Compiler technology allows performing sophisticated software analysis that is then used for speed or memory optimization. A compiler typically, as its last step, is able to modify the code, achieving some level of optimization. For instance, it can analyze and modify a program to reduce the number of operations by rearranging code parts, or to reduce the memory accesses by optimally using the processor internal registers. Compiler technology also includes *data flow analysis*, which can extract constant expressions from frequently executed code sections, for example, and *control flow analysis*, which can replace a sequence of statements by an equivalent one, i.e. producing the same result. Such technology, however, does not lead to an absolute complexity measure. It enables the comparison between two parts of code, measuring a *relative* complexity. A section of code is *rewritten* into a better one (faster and/or smaller in size), which is then put into the final program. The main drawback of such approach is that, so as to get good results, all the features of the underlying processor architecture have to be taken into account. If the goal is to get complexity results at the highest level of abstraction, without considering any specific architecture, this approach cannot lead to the desired results.

2.3. 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 [18,29]. 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 [18,29]. The main drawback of these techniques is that the real processing complexity of many algorithms heavily depends on the input data while static analysis depends only on the algorithm. For video coding, for instance, strict worst-case analysis can lead to results one or two orders of magnitude higher than the typical complexity values [19,20]. Consequently the range best case-worst case is so wide that results are meaningless. No useful indications can be extracted about the typical processing needs, which on the contrary can be better determined by including into the analysis statistical considerations and bounds on the input data. Moreover restricted programming styles such as absence of dynamic data structures, recursion and bounded loops are required [15]. This means, in many cases, the need to rewrite the program. Although video and multimedia processing can also be considered as *real-time* applications, their characteristics differ largely from real-time control applications that are the main field of these static approaches. Another serious drawback is the fact that while the high-level language is used to provide annotation, the final analysis is generally performed at the assembly language level thus implicitly accounting for the host processor system.

2.4. Hardware description languages and hardware specific tools

Hardware description languages (HDL) have now become very popular tools for the design and description of electronic systems. Automatic synthesis tools are able to generate circuit descriptions corresponding to the algorithms described in HDL. Through synthesis and simulation, such languages allow gathering very reliable results about the implementation complexity and performance of the described algorithm. However, such results, which are extremely useful for system design, arrive too late in the design flow. The algorithms have to be translated from the general purpose language specification into HDL, 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 HDL, there are tools which provide instruction-level simulation of DSPs or other type of embedded cores [2,4,23,28,38], 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 [5,7,9,10,34,35,37]. The whole system may be efficiently simulated to measure its performance, yielding reliable results useful in the optimization tasks whether for each single block or for the communications between different blocks. Another important advantage of these tools is that co-design and co-simulation ease the software/hardware partitioning and re-partitioning tasks, thus enabling quick system specifications and quick system re-design. Although these tools increase the overall productivity reducing both the design time and the number of re-design iterations, they do not cover the gap between the pure software specification of an algorithm and a system specification for a heterogeneous implementation of the same algorithm. Algorithms are becoming more and more complex and their specification and verification have to be performed at a very high level of abstraction, usually with common programming languages such as C and C++. Because of the previous reasons, such software verification modules are not meant to provide straight information for a hardware or software/hardware implementation of the algorithm itself or not even for another pure software implementation optimized for a specific core. Given a software verification module, the designer has to analyze it in order to fully understand it in depth, focus on critical points, discover bottlenecks, rewrite parts of the algorithm to optimize them for a specific implementation, etc. No automatic tool is available to help the designer at this very first stage and the first implementation choices strictly depend on the experience and skill of the designer. Because of the increasing complexity of algorithms and consequently of the increasing dimension of the corresponding software descriptions, decisions based on experience need more and more time to be taken and become less and less reliable. Hence the need of automatic tools able to perform a preliminary hardware oriented complexity analysis of the algorithm starting from a pure software specification, in order to drive the first implementation choices and reach an optimal solution with less re-design iterations.

3. Problem statement

A more precise statement of the desired algorithmic complexity analysis can be expressed as follows. The complexity of a specific implementation of an algorithm has to be measured *independently* of the underlying hardware architecture. It is assumed that a software implementation of the algorithm is available and that it can be run in realistic input data conditions. The goal is then to measure the complexity of the algorithm whose performance can be data dominated. In other words, the interest is not only about the measure of complexity of the algorithm itself, but also about its dependencies under specific input data. This approach is fully in line with methodological approaches proposed for instance in [24,25] aiming at optimizing data-transfer and memory bandwidths at a high-level description of the algorithm.

Pure algorithmic complexity does not depend on any other factor than the algorithm itself and the input data. Avoiding input data dependency leads only to worst-case best-case estimations, and these estimations, even though crucial for e.g. real-time control systems, are not of concern here. Both real-time and non real-time signal processing and image and multimedia processing are targeted, for which strict “worst case analysis” is not adequate [21,22]. Furthermore, the complexity evaluation must not depend on the type of hardware or compiler technology used for the evaluation. The only constant is the specific software implementation itself of the algorithm and the desired measure is the number of operations occurring during its execution, without taking into account the different ways to produce machine instructions for this particular program. The process of compiler optimization can, however, be used to accelerate the evaluation of the number of operations, but it must not interfere with it.

Given the number of operations **O** occurring during the execution of an algorithm A, the algorithmic complexity C_A is then defined, without loss of generality, as

$$C_A = f(\mathbf{O})$$

where $f(\cdot)$ is a mapping function. The way the complexity C_A is defined depends on the use of the complexity in the design process. It could be a single number, for a very high-level comparison of algorithms, but it will obviously be of smaller dimension than **O**. In practice, having a single number is not very useful, as it has already been shown in the world of benchmarks like MIPS and MFLOPS. Higher dimensions for C_A can be chosen to represent the aspect of complexity needed by the design phase. Thereafter, a mapping function $f(\cdot)$ can be defined. $f(\cdot)$ is a mapping from a set of measures, indicating algorithmic behavior, to a set of requirements, indicating what is important for the task. Therefore a universal mapping function $f(\cdot)$ cannot be provided.

It is then clear that the kind of information provided by C_A will heavily depend on the definition of $f(\cdot)$ but, more importantly, the reliability of this complexity information will directly depend on the reliability of the values in **O**. The goal is the faithful evaluation of the algorithm's operations **O**.

In software/hardware co-design, for instance, the most important issues can be classified into four categories:

1. type of operations (addition, multiplication, etc.);
2. type of data (integer, floating point, fixed point, etc.);
3. memory architecture;
4. memory accesses/bandwidth.

The goal being to provide a good insight in what an algorithm needs to be performed, **O** is decomposed into three components:

$$\mathbf{O} = \{O_{\text{ops}}, O_{\text{data}}, O_{\text{mem}}\}$$

where O_{ops} represents the number of operations per type of operation, O_{data} the number of operations per data type and O_{mem} the number of memory access operations. This is the most fine grain information that can be extracted from an implemented algorithm without having to take into account specificities of the underlying architecture.

Previous work by Shaw [32] on worst-case analysis for time-bounds estimation at the programming language level has turned out to be inadequate. In defining time-bounds on the different constructs of the language, they could estimate time-bounds for subroutines, and finally a whole program. Worst-case analysis has shown to be inadequate because of the difficulty of predicting time-bounds in a high-level language independently of the context in which it appeared and independently of the compiler and the target processor. Estimation of the number of operations does not suffer from this, because the measured quantity (operations) depends only on the algorithm and on the input data with which the algorithm is executed.

To be complete, and because algorithms are usually sequences of smaller steps, the complexity information should also contain information about the algorithm's logical organization. Therefore, function calls and function relation information should be a constitutive part of the complexity evaluation. This enables also the measuring of interactions between logical parts of the algorithm. For example, in a straight implementation of the discrete Fourier transformation (DFT), it could be interesting to know how many operations are spent in computing the transform coefficients versus the number of operations used to compute the basis function coefficients W_n^k. This indication is also important for algorithm optimizations, and leads to the use of lookup tables in the case of the DFT.

4. The proposed instrumentation approach

4.1. Concept

The implementation of the *Software Instrumentation Tool* (SIT) is based on the concept of the instrumentation of all the operations that take place during the execution of the software. Instrumenting code by C++ operator overloading has been already proposed in literature, but it has always been considered an approach presenting severe practical and functional limitations [16]. Major drawbacks were considered the applicability only to C++ program, the impossibility to instrument pointers and other data types such as structures and unions, resulting into not accurate analysis of data transfer oriented operations, and to an extensive manual rewriting of the original code. In the approach presented in this paper all known functional and practical limitations of the operator overloading approach have been solved. All type of operators for all type of data can be registered and assigned to a specific group of *counters*. The current version of *SIT* is able to instrument a `C` program by translating it into a corresponding `C++` program by means of an automatic tool: both programs have the same behavior but, by substituting `C` simple types with `C++` classes and by substituting all `C` operations with `C++` overloaded operators, standard `C` operations performed during the execution of the program can be intercepted and counted, along with other implicit operations such as memory accesses and data type conversions. This approach has the great advantage that no code rewriting is necessary to obtain high-level algorithmic complexity information. Moreover, associating an appropriate memory model to the processing makes SIT a complete simulation tool for fast architectural evaluations.

The actual instrumentation of a `C` program is schematically represented in Fig. 3. By changing the system executable search path, the standard `gcc` compiler is replaced by the *SIT's gcc*, a script that is in charge of controlling the overall instrumentation process, from source files to instrumented executable generation. The instrumentation process is completely automatic and it appears to the user exactly as a normal compilation with no need of modifying existing source files and makefiles or typing any special command. First original `C` source files are instrumented by *instrumenting gcc* (`igcc`) which translates each `C` source file into its corresponding instrumented `C++` version. The instrumented files are then compiled by means of standard `g++` and finally linked with system and SIT's libraries to produce the instrumented executable, which can be executed so as to process the corresponding input data. During execution, the *instrumented* version of the program registers every executed operation (explicit and implicit) and increments the corresponding *counter*, possibly using user defined contexts. Those *counters* can be merged in any form in order to represent the information in a more compact or detailed form depending on the user desires. By default, this grouping is based on the function call tree of the program. Facilities are provided to make the counting process time or data dependent. For instance, a program that codes frames of a video sequence might get a

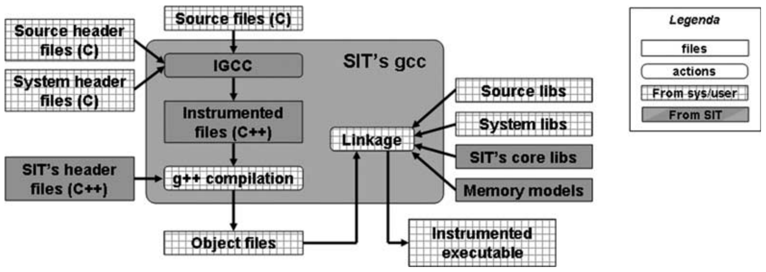


Fig. 3. Schematic diagram of the instrumentation process.

counter group for each frame. Each *counter* group would then include all functions needed to decode a specific frame, so that a complexity analysis on a frame-by-frame basis is possible.

SIT has basically two distinct goals:

- Providing complexity measures to compare different implementations of a given algorithm (abstract point of view).
- Given the complexity measures at previous point, helping to find out which architectures could help most in solving the different tasks (mapping and fast simulation point of view).

4.2. Operation count

SIT is able to count all types of operations, along with the data types they are applied to.

Table 1 shows the data types for which operations are counted: while there is a strict correspondence between C simple types and the corresponding instrumented types, the other data types have to be treated separately. In the first group, each instrumented type corresponds to a standard C simple type; the second group accounts for all the operations on pointers and vectors in general; structures and unions need to be instrumented as well, in order to manage the corresponding instrumented members contained in them, but all the operation counting is deferred to the instrumented members themselves; the last data type in table, BOOL, was introduced to count separately the operations in boolean expression because by a hardware point of view they are operations on bits and not on words like in C.

Exploiting C++ operator overloading SIT is able to intercept and count all the operations performed on the objects of the classes shown in Table 1. This means that the tool is able to count the operations performed on native C types in the original C description of the algorithm under analysis. A specific operator

Table 1
The instrumented C++ data types: all native C data types are substituted with C++ classes able to intercept and count, with their overloaded operators, the explicit and implicit operations performed during program execution

Group	Native C type	Instrumented type	C++ implementation of the instrumented type
Simple types	long double	LDBL	Classes
	double	DBL	
	float	FLT	
	unsigned long long	ULLINT	
	signed long long	LLINT	
	unsigned long	ULINT	
	signed long	LINT	
	unsigned int	UINT	
	signed int	INT	
	unsigned short	USINT	
	signed short	INT	
	unsigned char	UCHAR	
	signed char	CHAR	
Pointers and vectors	*Pointers*	Pointer<IT, OT>	Template classes
	Pointers to function	FPointer<OFP>	
	Vectors (any dimension)	Vector<IT, OT, SZ>	
Structures and unions	structs unions	*STRUCT*	Classes by means of *Template-like* macros
Boolean type	*Not defined*	BOOL	Class

The BOOL type was introduced to count boolean operations separately.

overloading over all possible combinations of operators and input data types is the key for a correct and reliable instrumentation and complexity analysis, because it allows both to preserve the native C behavior, explicitly implementing all the implicit type casting occurring in operations, and to count separately all the different operations:

- arithmetic operations (+, -, /, etc.);
- binary operations (&, |, <<, etc.);
- assignment operation (=), eventually combined with previous operations (+=, &=, etc.);
- prefix and postfix increment operations (++, –);
- comparison operations (==, <, <=, etc.);
- boolean operations (&&, ||, !);
- pointer dereferencing operations (*, [], ->);
- pointer arithmetic and assignment operations (+, -, ++, =);
- pointer comparison (==, <, <=, etc.);
- type casting, both explicit and implicit;
- memory I/O and allocation;

To get the most fine grain information about the executed operations in a system, the operation count updates one counter for each (*operation*; *data type*) couple. This leads to the possibility of an easy mapping onto any architecture, providing meaningful information at a high abstraction level.

4.3. Memory simulation and data-transfer and storage analysis

In complexity evaluation of systems, the memory bandwidth plays a fundamental role. 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 [24,25]. By intercepting the accesses to memory by means of read and write functions in instrumented classes and associating to the algorithm an underlying memory model, SIT is able to simulate memory operations and extract relevant information and measurements about memory performance, such as memory usage, cache hits and misses, data flows, etc.

The underlying memory architecture for which measurements are required can be easily specified aside without limitations and without rewriting the algorithm source code, thus avoiding the main drawback of systematic approaches to system design [3,26]. The simulated memory architecture is defined by means of *Memory Models* (see Fig. 4). The typical structure of a memory model consists of three types of simulation blocks:

- *Memory Manager*. It defines the allocation policy associated to the memory model (e.g. dynamic allocation, stack, etc.). It implements the interface between the memory model and the Simulation Core. During simulation it receives allocation and I/O commands from the Instrumentation Core and drives the rest of the hierarchy of simulation blocks in the memory model.

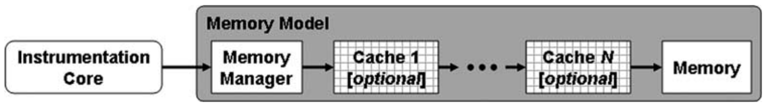


Fig. 4. Structure of a memory model for memory simulation. A memory model is composed by a Memory Manager, a Memory and an optional hierarchy of Caches.

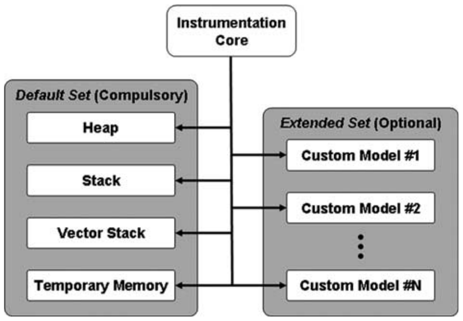


Fig. 5. The simulated memory architecture is composed by the compulsory Default Set of Memory Models and the optional Extended Set.

- *Memory*. It simulates the storage memory in the memory model and it is the last block in the simulation block hierarchy. Mainly it defines the size of the simulated memory and receives I/O commands from the parent simulation block in the hierarchy.
- *Cache*. An optional hierarchy of caches can be defined between the memory manager and the memory. A cache receives I/O commands from the parent simulation block and, according to the received I/O commands and the chosen cache policy, generates I/O commands to drive the child simulation block.

A standard set of interfaces is defined for implementing all the blocks in a memory model. These interfaces are meant to provide a standard framework to implement the different simulation blocks, plug them into each other according to the scheme of Fig. 4 and define the desired memory models to be used during simulation. Apart from the standard interfaces, the implementation of each block is completely free, allowing the user to define custom simulation blocks, with custom features and simulation results. That is, it is possible to simulate (partially or completely) different memory architectures, with different allocation policies and cache hierarchies, and compare the respective performances.

A variable (or a dynamically allocated block) is assigned to a memory model so that during simulation all the I/O operations on a variable (or on a dynamically allocated block) drive the simulation of the corresponding memory model. Pointers are the most critical entities in memory simulation, because they might point to "any" address and consequently it is not straightforward to detect to which memory model the pointed address is associated. The memory simulation core guarantees that all pointer operations are correctly mapped onto the "pointed" memory model, independently of the pointed address.

The simulated memory architecture must include at least four *Memory Models*, referred to as *Default Set of Memory Models* (see Fig. 5), for default assignments of variables and allocated blocks to memory models ("var-to-mem assignments" for short):

- *Heap*. All the variables dynamically allocated in the heap, e.g. typically large buffers, are assigned to this model. Its allocation policy must be dynamic.
- *Stack*. All the static and automatic variables, not of vector type, are assigned to this model. Its allocation policy must be stack-like.
- *Vector Stack*. It is the counterpart of *Stack* model for variables of vector type. Possibly *Stack* and *Vector Stack* can be implemented with one model only. The reason for using two different models for the stack is that two different types of variables are typically assigned to these models: counters, flags and temporary results to the *Stack* model, large buffers to *Vector Stack* model.

- *Temporary memory*. All the results of the operations, "temporary variables", are assigned to this model. This class of variables typically represents registers or even simple wires and this is the reason why they are associated to a different model for. Its allocation policy is free.

The user can specify, by means of specialized comments in the source code, different var-to-mem assignments. Thanks to these custom assignments, the user can assign one or more variables to other memory models than the ones in the default set, for instance to gather specific simulation results for sensitive data. The optional memory models for custom var-to-mem assignments are referred to as *Extended Set of Memory Models* (see Fig. 5).

5. Technical approach

5.1. Translation from C to C++

To count the operations, a specific code that intercepts the calls to all operations as they occur in the original program is needed, without interfering with the actual processing under evaluation.

The `C++` norm [33] states that any standard *ANSI-C* program is also a valid `C++` program. This forms the first requirement. A class can be built to behave in the same way of the corresponding `C` type. Replacing simple types, pointers, structures and unions with classes, it is possible to intercept, through operator overloading, all operations performed on data, here including both explicit `C` operations (sum, multiplications, etc.) and implicit operations (memory accesses).

Special care has to be taken when using classes to replace the original simple data types. With respect to `C` simple types, instrumented `C++` classes have some extra bytes of information, both because of the functionalities of *SIT* and because of some additional overhead for class management (e.g. classes' ids). Since many operations strictly depend on native `C` representation of data in memory (e.g. pointer type conversions, union management, etc.), the instrumented classes have to completely stick to such representation and therefore the data memory and the overhead memory must be managed separately. In this way, not only the same behavior between the original and the instrumented codes is guaranteed, but the exchange of data between instrumented and non instrumented libraries, such as standard `C` libraries is also possible.

Up to a few exceptions, the translation from a `C` code to the corresponding `C++` instrumented code consists in replacing the original declarations of variables and functions with new declarations using the instrumented classes and their constructors. Pointers and vectors are instrumented by means of `C++` templates generating different classes depending on the different data type they point to. The main exceptions concern structure and union declarations and boolean expressions. For each structure and union, classes with specific constructors and operators must be defined in order to manage correctly the members in different data structures; such specialized classes are defined with macros and with explicit code generation for specialized constructors and operators. Since boolean operations have to be counted separately from other operations on `int` type, despite of `C` implementation of boolean expressions, the `BOOL` instrumented type has been defined and during the conversion from `C` to `C++` this data type is explicitly managed in expressions, while for the other data types no change in the expressions is required.

5.2. Simple types

The list of all implemented simple types is shown in the first section of Table 1. The main constraint that has to be taken into account is that the set of operations must have the same effect on simple data types as on their class representation. While `C++` overloaded operators have the same precedence as the built-in

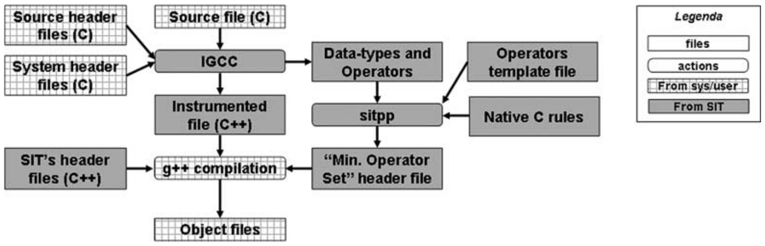


Fig. 6. `Igcc`, `sitpp` and the instrumentation of a source file. Besides instrumenting the source file, `igcc` generates also the list of the required data types and operators, which is processed by `sitpp` to generate the header file with the "Minimum Operator Set" for optimized compilation of the instrumented file.

equivalent ones, there is no constraint on their returned data type that, to stick to native C behavior, has to respect the automatic promotion rules occurring in C operations on simple types. Given that the return type, according to the promotion rules, depends both on the operator and on the type of the operands, a different overloaded operator for any legal combination of {*first operand type*, *binary operator*, *second operand type*} and of {*unary operator*, *operand type*} has been implemented. In this way not only the promotion rules can be respected, choosing the correct return type for each overloaded operator, but also any overloading ambiguity in compilation can be avoided, because any legal operation can be mapped directly onto one of the previous legal combinations. The main drawback of this choice is that some thousands of operators have to be implemented, resulting in a very large source code that, consequently, would be difficult to maintain and would compromise the performance of the compilation of instrumented code, both in terms of compilation time and memory usage. Both these two drawbacks have been overcome by creating a preprocessing tool, called *SIT preprocessor* (`sitpp`), able to automatically generate the source code for instrumented classes as described in the following points (see Fig. 6):

- *Operators Template File*. Instrumented classes and their overloaded operators are defined in a template source code of few hundred lines, much more easy to maintain than a code of many thousand lines.
- *Native C Rules*. The Operator Template File is expanded according to native C rules, e.g. available data types, available operations, promotion rules, etc., so that all the resulting classes and their overloaded operators are guaranteed to behave exactly as their native C counterparts.
- *Data types and Operators*. During instrumentation of each source file, `igcc` generates, besides the instrumented file, the list of necessary data types and operators. The information in this list is used by `sitpp` as a filter for the expansion of the Operators Template File, so that for each source file a header file is adaptively generated, which contains only the minimum set of operators specifically necessary for that source file (*Minimum Operator Set*). This minimizes the resources needed for the compilation of the instrumented files.

In order to allow data exchange between instrumented and non-instrumented libraries, operators must be provided to convert instrumented variables into their corresponding native values and vice versa. Constructors and assignment operators created by `sitpp` provide all the coercion paths from non-instrumented types to instrumented types. Problems can appear trying to implement coercion operators from instrumented types to non-instrumented types for implicit type castings when passing instrumented parameters to functions. As a side effect, they may cause overloading ambiguities in conversions between instrumented simple types and pointers and, furthermore, the implicit conversion for which they had been thought results to be useless when calling functions with variable argument lists, because in this case the compiler does not know which appropriate type conversion has to be implicitly called. It has been decided not to implement such operators to avoid overloading ambiguities and to perform such conversion by explicitly calling a conversion member function implemented for all instrumented types; `operator( )` was chosen for this task because no `C` data type uses this operator. In the translation from `C` to `C++` this operator must be explicitly called on instrumented data whenever they are passed as arguments to functions.

Here is an example showing how the following section of `C` code

```
int a = 10;

int function1(int param) {
 return param+1;
}
void function2(int param) {
  int a;
  a = function1(param);
  a+ = function1(10);
  printf("%i\n", a);
}
```

is translated into the corresponding instrumented `C++` code

```
INT a(10);                         // Type substitution in variable
                                   //   declarations
int function1(int SIT_param) {     // Function prototypes are never instrumented
                                   //   for compatibility issues
                                   //   with native libraries:
  INT param(SIT_param); {          //   parameters are instrumented
                                   //     inside the function
    ENTERFUNCTION("function1");    // Macro to register the function and trace
                                   //   the accesses to it
    return (param+1)();            // Explicit conversion to native C type
  }
}
void function2(int SIT_param) {
  INT param(SIT_param); {
    ENTERFUNCTION("function2");
    INT a;
    a = function1(param());      // Explicit conversion to native C type for
                                 //   parameters in function call
    a+ = function1(10);          // No change needed for non-instrumented parame-
                                 //   ters
  printf("%i\n\",                // Explicit conversion to native C type allows
         a()                     //   calling native library functions
         );
}
```

5.3. Pointers and vectors

Pointer operations and pointer arithmetic, as well as vector operations, are often used to implement iterations over data and to structure the data in a convenient way. The count of their operations is therefore critical to a sensible complexity analysis of a program.

Native pointers implement a limited set of operations and among them some specific ones, like dereferencing or bracketing to access an element in the array. Native pointers can be defined for any type, but the complexity of the operation is always the same. Therefore, only pointer operations are counted generically, and not depending on the data type they are working on.

As pointers apply to any type of data, C++ templates were chosen for their implementation. Instrumented pointers need information both about the pointed instrumented type, to use the corresponding instrumented operators when needed, and about the corresponding pointed native type, to know the real size of the data they point to and manage memory exactly as native C does. The `Pointer` template needs then two parameters `<class IT, class OT>` where `IT` and `OT` are, respectively, the instrumented and the original pointed types. Even though vectors and pointers share mostly the same behavior, some operations on vectors strictly depend on the number of elements in the vector itself and for this reason a specialized `Vector` template has been written, needing one parameter more (`<class IT, class OT, size_t SZ>`). Pointers to functions represent a special kind of pointers, with no instrumented type to point to. Actually, even an instrumented function is nothing more than a standard C++ function taking or returning native data types. The template `FPointer<class OFP>` instruments pointers to functions and `OFP` parameter represents the original pointer-to-function type. These three templates—possibly nested to create multidimensional vectors, pointers to pointers, pointers to vectors, etc.—allow to instrument all pointer and vector types and their operations and to manage memory exactly as their native C counterparts do.

The conversion from C to C++ is limited, as with the simple type, to change variable declarations and to add explicit type castings from instrumented to non-instrumented types:

C code

```
int *p = NULL;
long **pp;
int v[5];
long vv[3][4];
char (*f)(float);
...
printf("%p\n\", p);
```

C++ code

```
Pointer< INT, int> p(NULL);
Pointer< Pointer<LINT, long>, long*> pp;
Vector< INT, int, 5> v;
Vector< Vector < LINT, long, 4>, long[4], 3> vv;
  // The OT (original type) parameter of outer template is long[4] because
  //  long vv[3][4] is a "vector of 3 vectors of 4 long"
FPointer< char(*)(float)> f;
  // The parameter OFP (original function pointer) is a native C/C++
  //  pointer to function
...
printf("%p\n\", p());
  // Explicit type conversion through operator()
```

5.4. Structures and unions

The instrumentation of structures and unions needs specific classes for each original data structure like pointers and vectors. Unfortunately, these instrumented classes cannot be implemented by means of templates because they need specialized member declarations, constructors and operators depending not on the structure data type itself but on its member list. More precisely, an instrumented structure is composed by instrumented members that need to be constructed one by one, and this cannot be carried out with templates. The problem was solved with macros that mimic the features of templates while still allowing to add the required specialized code.

To stick to native `C` memory representation, the instrumented members are constructed by means of the addresses of their non-instrumented counterparts. The data of the instrumented structure are held in a non-instrumented structure. The instrumented members hold no data but simply intercept accesses and operations on the members of the non-instrumented structure. Unions are instrumented exactly like structures: their instrumented members being initialized with the addresses of their non-instrumented counterparts, data superposition of unions is automatically implemented without any change with respect to structure instrumentation.

The instrumentation of these data structures is rather more complicated than the instrumentation of previous data types but it is limited only to data type declarations and explicit conversions from instrumented to non-instrumented data structures.

Without getting into detailed explanations about the instrumented code, the following example shows how the declaration of a structure type and the instantiation of a variable of that structure type

```
struct myStruct {
  int i;
  char c;
  float *fp;
};
struct myStruct ms;
```

are instrumented:

```
struct myStruct{
  int i;
  char c;
  float *fp;
};
// Structure instrumentation by means of SIT_STRUCT_STEPxxx macros
//   and members' instrumentations
SIT_STRUCT_STEP1(SIT_type_myStruct, struct myStruct)
SIT_INT i;
SIT_CHAR c;
SIT_Pointer<
    SIT_FLT,
    float
> fp;
SIT_STRUCT_STEP2(SIT_type_myStruct, struct myStruct)
:SIT_STRUCT_STEP2_L(i)
```

```
,SIT_STRUCT_STEP2_L(c)
,SIT_STRUCT_STEP2_L(fp)
SIT_STRUCT_STEP3(SIT_type_myStruct, struct myStruct)
SIT_STRUCT_STEP3_L(i)
SIT_STRUCT_STEP3_L(c)
SIT_STRUCT_STEP3_L(fp)
SIT_STRUCT_STEP4(SIT_type_myStruct, struct myStruct)
SIT_type_myStruct ms;
```

Structures and unions with bit fields are not supported in the current SIT implementation.

5.5. Boolean type

By the hardware point of view, boolean data are 1 bit data while `C` manages such data and their expression using the word-sized `int` type. Since the goal of *SIT* is helping the designer figuring out the complexity of an algorithm, it is more correct to use a specific type, different from `int`, for boolean data and operations to get a more reliable complexity measure. For this reason, the `BOOL` instrumented type, without a corresponding standard `C` counterpart, has been introduced.

To properly take advantage of the `BOOL` type, an extra explicit type casting must be added each time a variable has to be converted in a boolean value, that is with boolean operators (& &, `||`, `?:`) or in conditional statements (`if`, `while`, `for`):

```
x = a?v1: v2;
if(a) { }
```

is instrumented as

```
x = BOOL(a)?v1: v2;
if(BOOL(a)) { }
```

The `!` operator (logical *NOT*) needs no explicit type casting because it is implemented as member of any instrumented class and the type casting can be defined inside the operator itself. While `C++` allows to overload logical operators && and `||` and the type casting could then be implemented inside the operator as with operator `!`, these operators can not be implemented as overloaded operators because of the strict boolean evaluation rule to which they must obey according to `C` standard. E.g. in a logical expression involving the '`||`' operator (logical *OR*), the left operand is always evaluated while the right operand is evaluated *if and only if* the first evaluated to false. The same mechanism applies to the '&&' operator (logical *AND*). Overloading the operator in an instrumented class, the behavior of the operator would change. For instance, the expression

```
char* p = NULL;
if (p && *p = ='a') { }
```

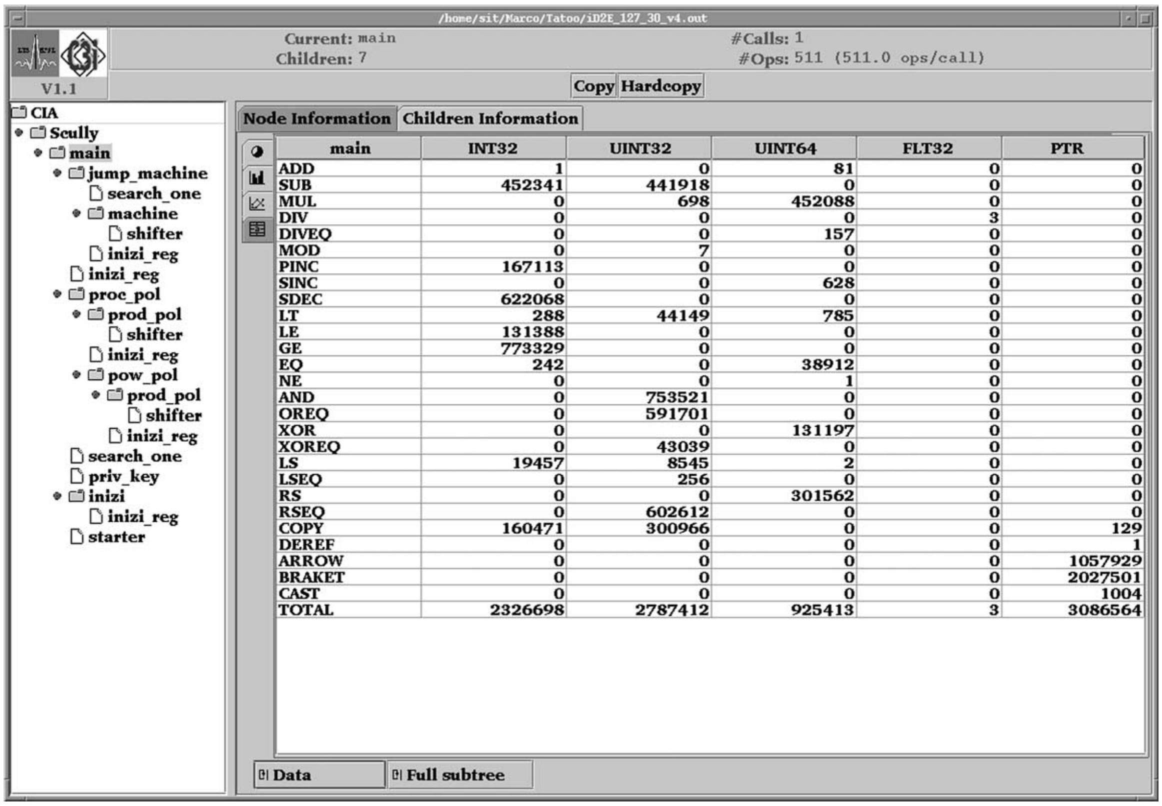
which checks if the pointer is `NULL` and *only* if it is not the case performs the comparison operation, would invariably make the instrumented program abort if the pointer is equal to `NULL`, because calling overloaded `operator&&`, not ruled by any strict boolean evaluation, would first result in evaluating *both* its arguments. Logical operators && and `||` are therefore instrumented as extending the corresponding native expression by calling a boolean function responsible only for the counting, but still using native `C` operators:

```
Pointer<CHAR, char> p = NULL;
if (count_AND() && BOOL(p)
&& BOOL(*p = ='a')) { }
```

where `count_AND()` is defined to count the logical *AND* operation and to return `1` (*TRUE*) to keep on evaluating the remaining part of the expression.

5.6. Complexity database construction

The complexity information is based on the counts of the basic operations. This information is produced during the execution of the instrumented program and is then stored in a file, the complexity database. To keep the structure of the procedure call tree, the counts are grouped by type of operations, data types and functions. However, for improved compaction of the information, no time information is kept, as the counts of operations are summed up through time. Keeping this information would mean keeping one record per operation. Such a database would be too large to be handled off-line. Functions that are called from different points in the program are handled separately, so that their contribution to the different branches in the call tree can be evaluated accurately. Similarly, recursive functions increase the depth of the call tree, so as to keep the maximum information available. The use of the resulting database is the same as for other tools used in analysis and optimization (e.g. `gprof` [11,13]). The execution of an instrumented program produces a file, which is self-contained in that it has operation counts, call tree and function names in it. The exploration of this file is then possible by means the graphical tool described in the following section.



Current: main
Children: 7
#Calls: 1
#Ops: 511 (511.0 ops/call)

```
CIA
  Scully
    main
      jump_machine
        search_one
        machine
          shifter
        inizi_reg
      inizi_reg
      proc_pol
        prod_pol
          shifter
        inizi_reg
        pow_pol
          prod_pol
            shifter
          inizi_reg
      search_one
      priv_key
      inizi
        inizi_reg
      starter
```

main	INT32	UINT32	UINT64	FLT32	PTR
ADD	1	0	81	0	0
SUB	452341	441918	0	0	0
MUL	0	698	452088	0	0
DIV	0	0	0	3	0
DIVEQ	0	0	157	0	0
MOD	0	7	0	0	0
PINC	167113	0	0	0	0
SINC	0	0	628	0	0
SDEC	622068	0	0	0	0
LT	288	44149	785	0	0
LE	131388	0	0	0	0
GE	773329	0	0	0	0
EQ	242	0	38912	0	0
NE	0	0	1	0	0
AND	0	753521	0	0	0
OREQ	0	591701	0	0	0
XOR	0	0	131197	0	0
XOREQ	0	43039	0	0	0
LS	19457	8545	2	0	0
LSEQ	0	256	0	0	0
RS	0	0	301562	0	0
RSEQ	0	602612	0	0	0
COPY	160471	300966	0	0	129
DEREF	0	0	0	0	1
ARROW	0	0	0	0	1057929
BRAKET	0	0	0	0	2027501
CAST	0	0	0	0	1004
TOTAL	2326698	2787412	925413	3	3086564

Fig. 7. Results for `main` function with key length = 127.

5.7. "SITview" graphical user interface

The exploration of the complexity database is done via the special graphical tool "*SITview*" developed along with *SIT*. All illustrations of graphical results in this paper are screenshots of this displaying tool. The information display is organized in *views*, each view being a combination of flags for the selection of the type of graph (bar, pie, table), the grouping of counts (current function, full sub-tree), the graph display (counts vs. data types, data types vs. counts) and the comparison level (current node, current node's children nodes). All these flags may be changed through the interface. This tool displays several types of information (see for example Fig. 7), all extracted from the complexity database file:

- Global information, as the current function name, database name, and the number of operations per call in the current function.
- The call tree, shown on the left, which presents multiple branches with the same label in case a function is recursive, or is called from different parent functions (e.g. shifter).
- The graphical representation of the currently selected call tree node and view (function `main` in Fig. 7).

The comparative evaluation of several runs of the same program or of different programs is possible, as several databases may be loaded at the same time into the displaying tool. For external evaluation of this information, exportation of the numbers as well as printing facilities are also available.

6. Results

This section reports two examples of the complexity analysis with SIT. The first example gives a global overview about the results that can be collected with SIT and how they were used to extract useful information about an encryption system. The second example focuses on the results of the memory simulation and data transfer analysis on an image processing algorithm.

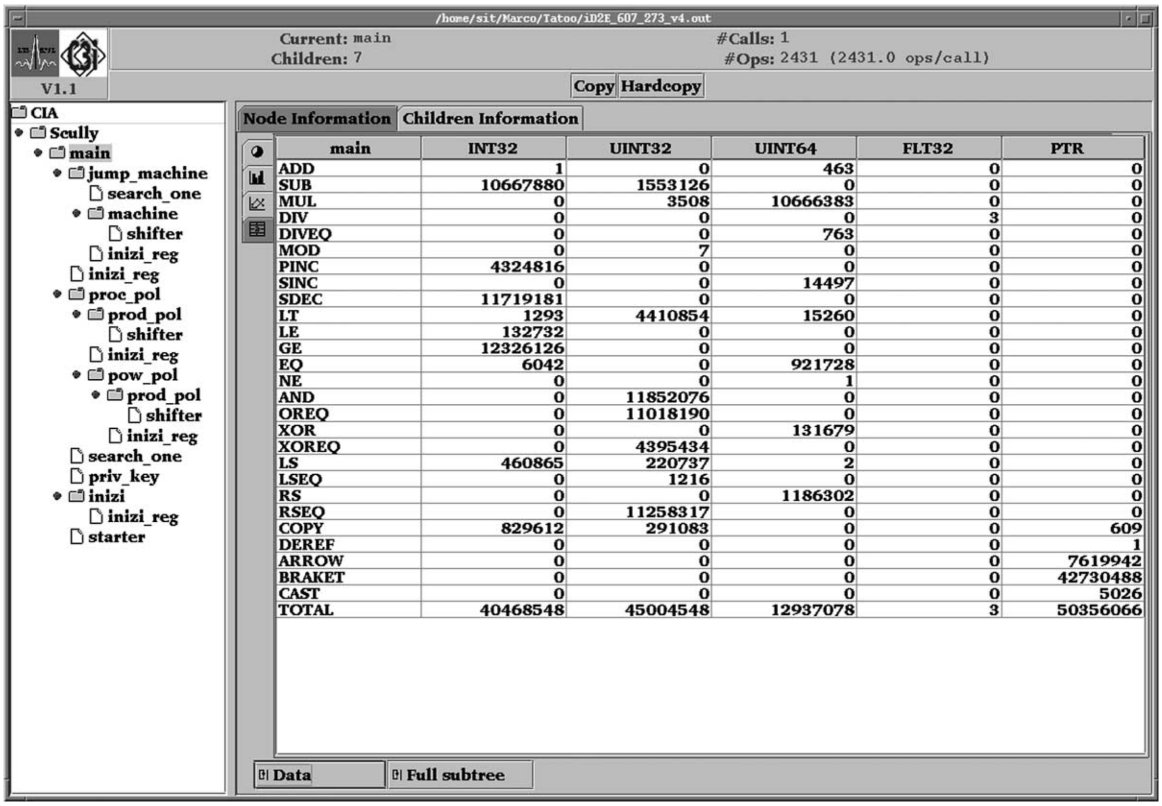
6.1. Application example no. 1: analysis of an encryption system

An example of the application of SIT is provided by the complexity analysis of a public key cryptosystem based on Diffie–Hellman algorithm [30,31]. This algorithm is based on the exponentiation of two numbers in finite Galois Fields. The exponentiation operation is reduced to recursive polynomial reductions and recursive shifts on state registers. The generically optimized `C` implementation is based on dynamic data structure and recursion.

Figs. 7 and 8 report the results of the overall number and type of operations for the public key generation in case of two different key bit-lengths. On the left, the program function call tree is shown and, on the right, the number of operations for the selected node in the tree (function `main` in this figure). The program includes recursive functions and dynamic data structures.

The exploration of the call tree immediately shows the different complexity of specific functions versus the two different key-lengths as depicted in Figs. 9 and 10. It is very clear that the function `prod_pol` is the most sensitive function to the length of the key.

These results have been obtained without any rewriting of the code, thus in case of software/hardware partitioning, the generically optimized `C` code is ready for embedding on a host system, while hardware co-processing options can be rapidly evaluated. By means of the *SIT* it has been possible to rapidly determine the key length boundary for which the calculation of used polynomials is more convenient than accessing memories storing pre-computed polynomials. Another information extracted was the estimation of the variance of the various operation numbers versus keys containing different fractions of ones and zeros.



```
/home/sit/Marco/Tatoo/iD2E_607_273_v4.out

Current: main                         #Calls: 1
Children: 7                           #Ops: 2431 (2431.0 ops/call)

CIA
• Scully
  • main
    • jump_machine
        search_one
      • machine
          shifter
        inizi_reg
      inizi_reg
    • proc_pol
      • prod_pol
          shifter
        inizi_reg
      • pow_pol
        • prod_pol
            shifter
          inizi_reg
      search_one
      priv_key
    • inizi
        inizi_reg
      starter
```

main	INT32	UINT32	UINT64	FLT32	PTR
ADD	1	0	463	0	0
SUB	10667880	1553126	0	0	0
MUL	0	3508	10666383	0	0
DIV	0	0	0	3	0
DIVEQ	0	0	763	0	0
MOD	0	7	0	0	0
PINC	4324816	0	0	0	0
SINC	0	0	14497	0	0
SDEC	11719181	0	0	0	0
LT	1293	4410854	15260	0	0
LE	132732	0	0	0	0
GE	12326126	0	0	0	0
EQ	6042	0	921728	0	0
NE	0	0	1	0	0
AND	0	11852076	0	0	0
OREQ	0	11018190	0	0	0
XOR	0	0	131679	0	0
XOREQ	0	4395434	0	0	0
LS	460865	220737	2	0	0
LSEQ	0	1216	0	0	0
RS	0	0	1186302	0	0
RSEQ	0	11258317	0	0	0
COPY	829612	291083	0	0	609
DEREF	0	0	0	0	1
ARROW	0	0	0	0	7619942
BRAKET	0	0	0	0	42730488
CAST	0	0	0	0	5026
TOTAL	40468548	45004548	12937078	3	50356066

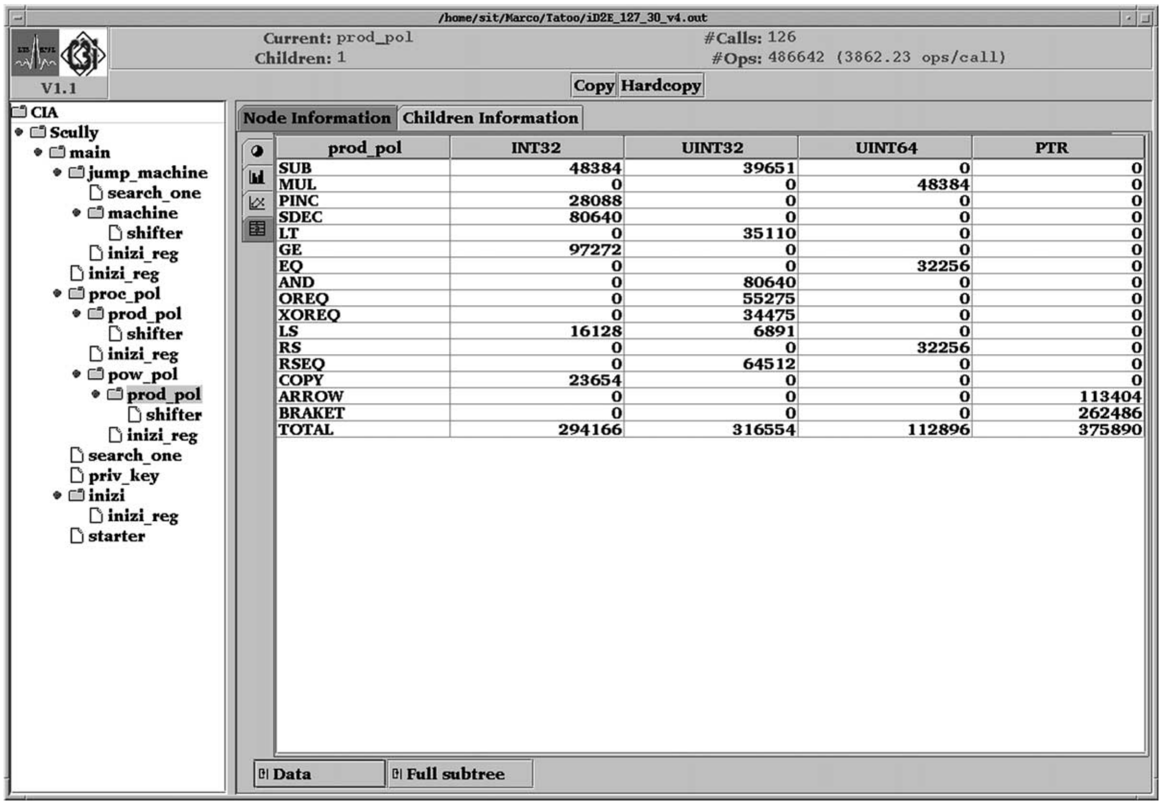
Fig. 8. Results for `main` function with key length = 607.

Conversely strict worst case–best case bounds obtained by path enumerations, but not corresponding to reasonably probable conditions, would lead to extremely large intervals.

6.2. Application example #2: analysis and implementation of an image processing algorithm

For several real time image-processing applications a fast and adaptive image acquisition stage is the key feature to achieve the required speed for the application requirements. In general, with sensors and applications based more and more on high resolution and super high resolution image content, the acquisition stage and the basic image processing capabilities become fundamental so as to reach real-time performance. CMOS sensors are very attractive because they allow adapting the acquisition to the processing as retina does [14,27]. This section shows the result of the data transfer analysis with SIT of an image processing algorithm, and its implementation on an embedded co-processor for an intelligent camera based on a CMOS sensor.

The algorithm was simulated over a memory architecture with a 512 bytes cache for the *Heap* Memory Model (see Section 4.3). By finding out the nodes of the execution tree where the cache had a good performance, that is the nodes with a high locality on the processed data, the parts of the algorithm eligible for data transfer optimization were easily identified. As shown in Table 2, a peak in cache performance was found in function "AdaptativeBinarization", an implementation of Niblack algorithm for local adaptive binarization [36], where 93% of the read operations resulted in a cache hit.



/home/sit/Marco/Tatoo/iD2E_127_30_v4.out

Current: prod_pol #Calls: 126
Children: 1 #Ops: 486642 (3862.23 ops/call)

```
CIA
  Scully
    main
      jump_machine
        search_one
        machine
          shifter
        inizi_reg
      inizi_reg
      proc_pol
        prod_pol
          shifter
        inizi_reg
        pow_pol
          prod_pol
            shifter
          inizi_reg
      search_one
      priv_key
      inizi
        inizi_reg
      starter
```

prod_pol	INT32	UINT32	UINT64	PTR
SUB	48384	39651	0	0
MUL	0	0	48384	0
PINC	28088	0	0	0
SDEC	80640	0	0	0
LT	0	35110	0	0
GE	97272	0	0	0
EQ	0	0	32256	0
AND	0	80640	0	0
OREQ	0	55275	0	0
XOREQ	0	34475	0	0
LS	16128	6891	0	0
RS	0	0	32256	0
RSEQ	0	64512	0	0
COPY	23654	0	0	0
ARROW	0	0	0	113404
BRAKET	0	0	0	262486
TOTAL	294166	316554	112896	375890

Fig. 9. Results for `prod_pol` function with key length = 127.

A second simulation with a different memory architecture, revealed that a smaller cache of 128 bytes was worthless, being the data transfers almost unaffected by the presence of the cache (see Table 3). The optimal cache size was therefore expected to be between 128 and 512 bytes.

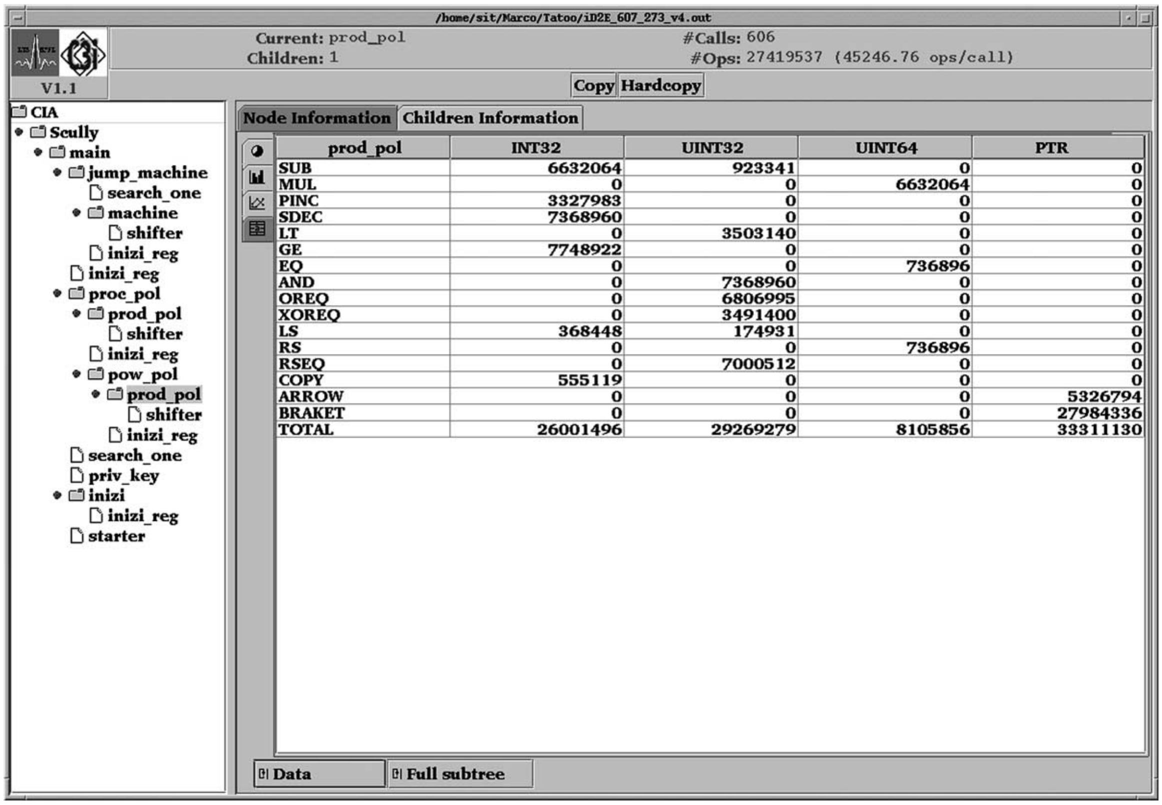
Following the preliminary analysis with SIT, an optimized hardware implementation of the local adaptive binarization algorithm was derived, with small local cache memories (MB blocks in Fig. 11) reducing the total bandwidth from the image buffer. The total size of the MB blocks is 320 bytes, in line with the expected range.

7. Conclusions

Current trends in algorithm design lead to complex schemes, most of them having to be specified and verified by generic software implementations. The intuitive understanding of the underlying processing and the comparison of their respective complexity are becoming a hard task for the system designer.

This paper has shown the importance of this fact and given an overview of existing complexity evaluation tools. Since all of them present serious drawbacks, an automatic tool is needed to assist the designer in the implementation of the considered algorithm at high abstraction level.

In the development of the SIT tool a particular attention has been devoted to some fundamental requirements. The first is to avoid the rewriting of the generic code describing the algorithm for extracting



/home/sit/Marco/Tatoo/iD2E_607_273_v4.out

Current: prod_pol #Calls: 606
Children: 1 #Ops: 27419537 (45246.76 ops/call)

V1.1

```
CIA
• Scully
  • main
    • jump_machine
        search_one
      • machine
          shifter
        inizi_reg
      inizi_reg
    • proc_pol
      • prod_pol
          shifter
        inizi_reg
      • pow_pol
        • prod_pol
            shifter
          inizi_reg
      search_one
      priv_key
    • inizi
        inizi_reg
      starter
```

prod_pol	INT32	UINT32	UINT64	PTR
SUB	6632064	923341	0	0
MUL	0	0	6632064	0
PINC	3327983	0	0	0
SDEC	7368960	0	0	0
LT	0	3503140	0	0
GE	7748922	0	0	0
EQ	0	0	736896	0
AND	0	7368960	0	0
OREQ	0	6806995	0	0
XOREQ	0	3491400	0	0
LS	368448	174931	0	0
RS	0	0	736896	0
RSEQ	0	7000512	0	0
COPY	555119	0	0	0
ARROW	0	0	0	5326794
BRAKET	0	0	0	27984336
TOTAL	26001496	29269279	8105856	33311130

Fig. 10. Results for `prod_pol` function with key length = 607.

Table 2
Memory simulation results for "AdaptiveBinarization" function with a cache memory of 512 bytes

AdaptiveBinarization	Read	RHits	RHits%	RMisses	RMisses%
Heap{DynMgr}	12′907′316				
Heap{TestCache: 512}	12′907′316	12′001′076	93.0	906′240	7.0
Heap{RAM}	906′240				

The very good performance of the cache reveals that the data transfers within this function can be optimized.

Table 3
Memory simulation results for "AdaptiveBinarization" function with a cache memory of 128 bytes

AdaptiveBinarization	Read	RHits	RHits%	RMisses	RMisses%
Heap{DynMgr}	12′907′316				
Heap{TestCache: 128}	12′907′316	114′232	0.9	12′793′084	99.1
Heap{RAM}	12′793′084				

This cache does not optimize the data transfers as the 512 bytes cache.

measurements. The second is the ability to instrument any algorithm written in C, without the typical limitations on operators and data type of classical state of the art approaches based on C++ operator

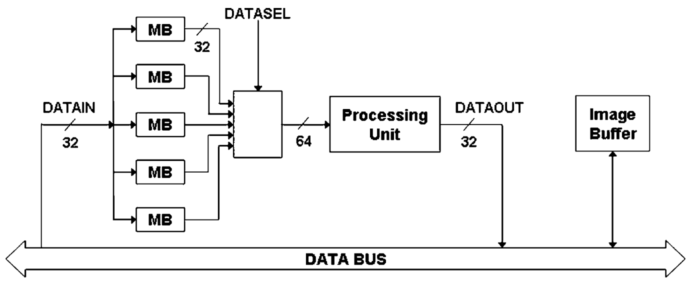


Fig. 11. Hardware module for Niblack algorithm for Local Adaptive Binarization. integrated on an FPGA Virtex II XC2V1000. Each MB is a memory block of 16 words of 32 bits, in charge of caching the input data in order to minimize the data transfers from the DATA BUS to the module.

overloading. With this purpose a technology that translates C it into an *instrumented* C++ code, which intercepts and counts all the operations, has been developed. The set of operations and data types has been taken from the definition of C itself. The *SIT* tool is capable of producing, in a multistep process transparent to the user, an executable that can be run as the original program, producing the same output plus an additional data-base consisting of the complexity analysis results about the explicit and implicit operations performed during the processing of the input data.

Finally, the analysis capability of the tool for detecting critical implementation issues has been shown by means of two examples of true design cases.

Further work is currently devoted to improve SIT functionality, in particular:

- for the development of a module for measuring the critical path and, consequently, providing an estimate of the potential operation and data parallelism;
- for the measurement of the working set in order to estimate the memory size requirements at different levels in the memory hierarchy (registers, caches, RAMs).

Another extension under study is the development of a tool to instrument C++ code thus supporting any system specification based on C and/or C++ modules.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques and Tools, second ed., Addison-Wesley, Reading, MA, 1988.
[2] T. Ball, J. Larus, Optimally profiling and tracing programs, ACM Transactions on Programming Languages and Systems 16 (4) (1994) 1319–1360.
[3] J. Bormans, T. Gijbels, L. Nachtergaele, Initial assessment of the video VM 5.0 Memory Requirements, ISO/IEC JTC1/SC29/WG11 MPEG97/M1914, Bristol, April 1997.
[4] R. Cmelik, D. Keppel, SHADE: a fast instruction-set simulator for execution profiling, in: 1994 ACM Conference on Measurement and Modeling of Computer Systems, 1994.
[5] CoWare Corporation, CoWare N2C Design System. Available from <www.coware.com>.
[6] I.D. 13818-2, Information technology—generic coding of moving pictures and associated audio information—part 2: Video, Technical Report, International Organization for Standardization, 1994.
[7] J. Davis II et al., Overview of the Ptolemy Project, ERL Technical Report UCB/ERL No. M99/37, Department of EECS, University of California, Berkeley, CA 94720, July 1999.
[8] T. Ebrahimi et al., Dynamic coding of visual information, technical description JTC1/SC2/WG11/M0320, mpeg-4, International Organization for Standardization ISO/IEC, October 1995.
[9] S. Edwards et al., Design of embedded systems formal models validation and synthesis, Proceedings of the IEEE 85 (3) (1997).

[10] M. Eisenring, J. Teich, L. Thiele, Rapid Prototyping of Dataflow Programs on Hardware/Software Architectures, in: Proceedings of HICSS'98, the Hawaii International Conference on System Science, Kona, Hawaii, January 1998, pp. 187–196.

[11] J. Fenlason, R.M. Stallman, Documentation for GNU gprof profiler, 1993. Available from <http://www.gnu.org/manual/gprof-2.9.1/html_mono/gprof.html>.

[12] L. Geppert, T. Perry, Transmeta's magic show, IEEE Spectrum 37 (5) (2000) 27–33.

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

[14] Y. Ho Jung, J. Seok Kim, B. Soo Hur, A. Moon Gi Kang, Design of real-time image enhancement preprocessor for CMOS image sensor, IEEE Transactions on Consumer Electronics 46 (1) (2000) 68–75.

[15] E. Kligerman, D. Stoyenko, Real-time euclid: a language for reliable real time systems, IEEE Transactions on Software Engineering SE-12 (Sept) (1986) 941–949.

[16] P. Kuhn, Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation, Kluwer Academic Publisher, Dordrecht, 1999, pp. 61–81.

[17] P. Kuhn, Instrumentation tools and methods for MPEG-4 VM: review and a new proposal, Technical Report M0838, ISO/IEC, March 1996.

[18] Y.S. Li, S. Malik, Performance analysis of embedded software using implicit path enumeration, IEEE Transactions on Computer-Aided Design, of Integrated Circuits and Systems 16 (Dec) (1997) 1477–1487.

[19] S. Mallat, F. Falzon, Analysis of low bit rate image transform coding, IEEE Transactions on Signal Processing 46 (Apr) (1998) 1027–1042.

[20] M. Mattavelli, S. Brunetton, A statistical study of MPEG-4 VM texture decoding complexity, Technical Report Doc. M924, ISO-IEC/JTC1/SC29/WG11 MPEG-4, Tampere, Finland, July 1996.

[21] M. Mattavelli, S. Brunetton, Implementing real-time video decoding on multimedia processors by complexity prediction techniques, IEEE Transactions on Consumer Electronics 44 (Aug) (1998) 760–767.

[22] M. Mattavelli, S. Brunetton, D. Mlynek, Real-time implementation of object based video coding, IEEE Transactions on Circuit and Systems for Video Technology (Special Issue on Object Based Coding) to appear.

[23] S. Mukherjee et al., Winsconsin Wind Tunnel II: a fast and portable parallel architecture simulator, Workshop on Performance Analysis and Its Impact on Design—PAID, June 1997.

[24] L. Nachtergaele, F. Catthoor, B. Kapoor, S. Janssens, D. Moolenaar, Low power data transfer and storage exploration for H.263 video decoder system, IEEE Journal on Selected Areas in Communication 16 (1) (1998) 120–129.

[25] L. Nachtergaele, F. Catthoor, B. Kapoor, S. Janssens, D. Moolenaar, Low power storage exploration for H.263 video decoder, VLSI Signal Processing, San Francisco, California, November 1996, pp. 115–124.

[26] L. Nachtergaele, D. Moolenaar, B. Vanhoof, F. Catthoor, H. De Man, System-level power optimization of video codecs on embedded cores: a systematic approach, Journal of VLSI Signal Processing 18 (1998) 89–109.

[27] F. Paillet, D. Mercier, T.M. Bernard, Second generation programmable artificial retina, in: Proceedings Twelfth Annual IEEE International ASIC/SOC Conference, 1999, pp. 304–309.

[28] A.D. Pimentel, L.O. Hertzberger, Abstract workload modeling in computer architecture simulation, in: Proceedings of the 24th ACM/IEEE International Symposium on Computer Architecture, Denver, USA, June 1997.

[29] P. Pushner, C. Koza, Calculating the maximum execution time of real-time programs, Journal of Real-Time Systems 1 (Sept) (1989) 160–176.

[30] A. Romeo, M. Mattavelli, A hardware oriented analysis of cryptographic systems for multimedia applications, in: Proceedings of EUSIPCO 2000, Tampere Finland, September 2000.

[31] A. Romeo, G. Romolotti, M. Mattavelli, D. Mlynek, Cryptosystem architectures for very high throughput multimedia encryption: the RPK solution, in: Proceedings of ICECS99, International Conference on Electronics, Circuits and Systems, Cyprus, 1 September 1999, pp. 261–264.

[32] A.C. Shaw, Reasoning about time in higher-level language software, IEEE Transactions on Software Engineering 15 (July) (1989) 875–889.

[33] B. Stroustrup, The C++ Programming Language, second ed., Addison Wesley, Reading, MA, 1994.

[34] Synopsys Corporation, Designing complex digital communications for systems on a chip. Available from <www.synopsys.com/products/dsp/digital_br.html>.

[35] B. Tabbara, L. Lavagno, A. Sangiovanni-Vincentelli, Fast hardware-software co-simulation using software synthesis and estimation, in: IEEE International High Level Design Validation and Test Workshop, 1997.

[36] O.D. Trier, A.K. Jain, Goal-directed evaluation of binarization methods, IEEE Transactions on Pattern Analysis and Machine Intelligence 17 (12) (1995) 1191–1201.

[37] P.V. Knudsen, J. Madsen, Aspects of system modelling in hardware/software partitioning, in: Proceedings of the 7th IEEE International Workshop on Rapid Systems Prototyping, Thessaloniki, Greece, June 1996, pp. 18–23.

[38] E. Witchel, M. Rosenblum, Embra: fast and flexible machine simulation, in: Proceedings of the 1996 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems, May 1994, pp. 128–137.

Massimo Ravasi was born in Lecco, Italy, on 16 December 1968. In December 1997, he received the degree in electrical engineering (undergraduate field: electronic systems) from Politecnico di Milano. From October 1997 to March 1998 he worked as a part-time collaborator at Laboratorio S.I.A. at Politecnico di Milano. In April 1998, he joined the Signal Processing Laboratory (LTS3) as a research assistant working on the development of a hardware JPEG 2000 codec in the Mitocoma project.He is now working for his Ph.D. in the SIT project on the development of a tool for high-level algorithmic complexity analysis for system design.

Marco Mattavelli received his Diploma of Electrical Engineering from the Politecnico di Milano, Milano, Italy in March 1987. Then he joined the "Philips Research Laboratories" of Eindhoven. Main research activities regarded channel and source coding for optical recording electronic photography, and signal processing of TV and HDTV signals. Since October 1991 he joined the "Signal Processing Laboratory" (LTS) of the "Swiss Federal Institute of Technology" (EPFL) where he got his PhD degree in 1996. Then he joined the Integrated Systems Laboratory of EPFL where he is currently Scientific Advisor. Main research interests currently are: architectures and system for video coding, the application and implementation of combinatorial optimization techniques for image analysis, and tools for the aid to architecture design of complex systems. He is also involved in ISO-MPEG standardization activities where currently chairs the MPEG Implementation Studies Group.