13

SW Annotation Techniques and RTOS Modelling for Native Simulation of Heterogeneous Embedded Systems

Héctor Posadas, Álvaro Díaz and Eugenio Villar
Microelectronics Engineering Group of the University of Cantabria
Spain

1. Introduction

The growing complexity of electronic systems has resulted in the development of large multiprocessor architectures. Many advanced consumer products such as mobile phones, PDAs and media players are based on System on Chip (SoC) solutions. These solutions consist of a highly integrated chip and associated software. SoCs combine hardware IP cores (function specific cores and accelerators) with one or several programmable computing cores (CPUs, DSPs, ASIPs). On top of those HW resources large functionalities are supported.

These functionalities can present different characteristics that result in non homogeneous solutions. For example, different infrastructure to support both hard and soft real time application can be needed.. Additionally, large designs rely on SW reuse and thus on legacy codes developed for different platforms and operating systems. As a consequence, design flows require managing not only large functionalities but also heterogeneous architectures, with different computing cores and different operating systems.

The increasing complexity, heterogeneity and flexibility of the SoCs result in large design efforts, especially for multi-processor SoCs (MpSoC). The high interaction among all the SoC components results in large number of cross-effects to be considered during the development process. Additionally, the huge number of design possibilities of complex SoCs makes very difficult to find optimal solutions. As a consequence, most design decisions can no longer depend only on designers' experience. New solutions for early modeling and evaluating all the possible system configurations are required. These solutions require very high simulation speeds, in order to allow analyzing the different configurations in acceptable amounts of time. Nevertheless, sufficient accuracy must be ensured, which requires considering the performance and interactions of all the design components (e.g. processors, busses, memories, peripherals, etc.).

Static solutions have been proposed to estimate the performance of electronic designs. However, these solutions usually result too pessimistically and are difficult to scale to very complex designs. Instead, performance of complex designs can be more easily evaluated with simulation based approaches. Thus, virtual platforms have been proposed as one of the main ways to solve one of the resulting biggest challenges in these electronic designs:

perform software development and system performance optimization before the hardware board is available. As a result, engineers can start developing and testing the software from the beginning of the design process, at the same time they obtain system performance estimations of the resulting designs.

However, with the increase of system complexity, traditional virtual platform solutions require extremely large times to model these multiprocessor systems and evaluate the results. To overcome this limitation, new tools capable of modeling such complex systems in more efficient ways are required. First, it is required to reduce simulation times. Second, it is required to have tools capable of modeling and evaluating initial, partial designs with a low effort. For example, it is not acceptable to require complete operating system ports to initially evaluate different platform possibilities. Only when the platform is decided OS ports must be done, due to the large design effort required.

Virtual platform technologies based on simulations at different abstraction levels have been proposed, providing different tradeoffs between accuracy and speed. As early evaluation of complex designs requires very high simulation speeds, only the use of faster simulation techniques can be considered. Among them, simulations based on instruction set simulators (ISSs) and binary translation are the most important ones. However, none of them really provides the required trade-off for early evaluation.

ISSs are usually very accurate but too slow to execute the thousands of simulations required to evaluate complete SoC design spaces. ISS-based simulations usually can take hours, which means that the execution of thousand of simulation can require years, something not acceptable in any design process.

Simulations based on binary translation are commonly faster than ISSs. However, these solutions are more oriented to functional execution than to performance estimation. Effects as cache modeling are usually not considered when applying binary translation. Furthermore, this simulations also result too slow to explore large design spaces.

Additionally, in both cases, the simulation requires a completely developed SW and HW platform. Completely operational peripheral models, operating systems, libraries, compilers and device drivers are needed to enable system modeling. However, all these elements are usually not available early in the design process. Then, these simulation techniques are not only too slow but also difficult to perform. The dependence on such kind of platforms also results in low flexibility. Evaluating different allocations in heterogeneous platforms, different kind of processors and different operating systems is limited by the refining effort required to simulate all the options. Similarly, the evaluation of the effect of reusing legacy code in those infrastructures is not an easy task. As a consequence, faster and more flexible simulation techniques, capable of modeling the effect of all the components that impact on system performance, are required for initial system development and performance evaluation.

The solution described in this chapter is to increase the abstraction level, moving the SW simulation and evaluation from binary-based virtual platforms to native-based infrastructures. Using cross-compiled codes to simulate a platform in a host computer requires compulsory using some kind of processor models and a developed target SW platform. Thus, the simulation overhead provided by the processor model, and the development effort to develop the SW platform are items that cannot be avoided. On the

contrary, simulations based on native or host-compiled executions avoid requiring a functional processor model, since no binary interpretation is done. Furthermore, a complete SW platform is not required, since the native SW platform can be partially used.

Nevertheless, in order to accurately modeling the system behavior and its performance modelling, a set of additional elements have been included in the native simulation infrastructures. Capabilities for modeling the delay of the SW execution in the target processor, the operation of the different level of caches, the target operating system and the other components in the HW platform, have been added. In the literature, some partial solutions have been proposed to support some of the elements of this list. However, some other features have not been solved in previous approaches, such as the support of different operating systems. Additionally as most of the proposed works are partial proofs of concept, there is a lack of complete integrated solutions.

The modeling of the application SW and its execution time in the target platform is a key element in native simulation, since it is the part of the infrastructure with more impact both in the simulation speed and in the modelling accuracy. Thus, in order to enable the designers to adjust the speed/accuracy ratio according to their needs, different solutions for SW annotation are presented and analyzed in the chapter. All solutions enable very easily exploring the effect of using different processors in the system. Only a generic compiler for the target processor is used. No specific OS ports, linker scripts or libraries are required.

With respect to the operating system, a basic OS modeling infrastructure has been developed, providing the user the possibility of simulating code based on Linux (POSIX), uC/os-II and Windows. The model has been developed starting from an OS modelling infrastructure providing a POSIX API. This infrastructure has been extended to support at the same time the other two APIs. This is an important step ahead to the state of the art, since very few proposed infrastructures support real operating systems, and to the best of our knowledge none of them considers these different APIs.

The resulting virtual platforms are about two-three times slower that functional execution when caches are not considered, and about one order of magnitude slower when using cache models. Processor modeling accuracy in terms of execution times is lower than 5% of error and the number of cache misses has an error of about 10%.

2. Related work

The modelling and performance evaluation of common MpSoC systems focuses in the modelling of the SW components. Since most of the functionality is located in SW this part is the one requiring more simulation times. Additionally the evaluation accuracy of the SW is also critical in the entire infrastructure accuracy. SW components are usually simulated and evaluated using two different approaches: approaches based on the execution of cross-compiled binary code and solutions based on native simulation.

Simulations based on cross-compiled binary code are based on the execution of code compiled for a target different from the host computer. As a consequence, it is required to use an additional tool capable or reading and executing the code. Furthermore, this tool is in charge of obtaining performance estimations. To do so, the tool requires information about the cycles and other effects each instruction of the target machine will have in the system. Three different types of cross-compiled binary code can be performed depending on the

type of this tool: simulations with processor models, compiled simulation and binary translation.

Instruction set simulators (ISSs) are commonly used as processor models capable of executing the cross-compiled code. These simulators can model the processor internals in detail (pipeline, register banks, etc.). As a consequence, they achieve very accurate results. However, the resulting simulation speed is very slow. This kind of simulators has been the most commonly used in industrial environments. CoWare Processor Designer (Cowar), CoMET de VaST Systems Technology (CoMET), Synopsys Virtual Platforms (Synopsys), MPARM (Benini et al, 2003) provide examples of these tools. However, due to the slow simulation speeds obtained with those tools, new faster simulation techniques are obtaining increasing interest.

Compiled simulation improves the performance of the ISSs while maintaining a very high accuracy. This solution relies on the possibility of moving part of the computational cost of the model from the simulation to the compilation time. Some of the operations of the processor model are performed during the compilation. For example, decoding stage of the pipeline can be performed in compilation time. Then, depending on the result of this stage, the simulation compiler selects the native operations required to simulate the application (Nohl et al, 2002). Compiled simulations based on architectural description languages have been developed in different projects, such as Sim-nML (Hartoog et al, 1997), ISDL (XSSIM) (Hadjiyiannis et al, 1997) y MIMOLA (Leupers et al, 2099). However, the resulting simulation is still slow and complex and difficult to port.

The third approach is to simulate the cross-compiled code using binary translation (Gligor et al, 2009). In this technique assembler instructions of the target processor are dynamically translated into native assembler instructions. Then, it is not necessary to have a virtual model describing the processor internals. As a result, the SW code is simulated much faster than in the two previous techniques. However, as there is no model of the processor, it is a bit more difficult to obtain accurate performance estimations, especially for specific elements as caches. Some examples of binary translation simulators are IBM PowerVM (PowerVM), QEMU (Qemu) or UQBT (UQBT).

Although these techniques result in quite fast simulators, the need of modelling very complex system early in the design process requires searching for much faster solution. For example, the exploration of wide design spaces can require thousands of simulations, so simulation speed have to be as close to functional execution speed as possible. The previous simulation techniques require a completely developed SW and HW platform, which are usually not available early in the design process. Then, these simulation techniques are not only too slow but also difficult to perform. Additionally, the simulation of heterogeneous platforms, with different kind of processors and different operating systems is limited by the refining effort required to evaluate all the options.

In order to overcome all these limitations, native simulation techniques have been proposed (Gerslauer et al, 2010).

2.1 Native simulation

In native simulation, the SW code is directly executed in the host computer. Thus, it is not required any kind of interpreter. As a consequence, very high simulation speeds can be

achieved. However, in order to model not only the functionality but also the performance expected in the target platform additional information has to be added to the original code.

Furthermore, a model of the SW platform is also required. If the target operating system API is different than the native one, an API model is required to enable the execution of the SW code. A scheduler only controlling the tasks of the system model, not the entire host computer processes, specific time controller, or different drivers and peripheral communications are elements the SW infrastructure must provide.

Several solutions have been proposed for both issues in the last years.

2.2 SW performance estimation

Native simulation (Hwang et al, 2008; Schnerr et al, 2008; Bouchima et al, 2009) obtains target performance information from an analysis of the source code of the application SW to be executed. The common technique used to perform native simulations is to divide the code in fragments, estimate the time for each one of the fragments before the compilation process and annotate this information in the code. Usually basic blocks are used as code fragments because the entire block is always completely executed in the same way. Thus, basic blocks can be annotated as a single unit without introducing estimation errors. Such annotated code is then compiled and executed in the host computer, together with an infrastructure capable of capturing the timing estimations generated, and applying the corresponding delays to the simulation. As a consequence a timed model of the SW is obtained; a model which is ready to interact with other timed SW and HW components, to model the entire system.

Several techniques have been proposed to obtain the time information for each code fragment. These techniques can be divided in three main groups: pure source code estimations, estimations of intermediate code and cross-compiled code analysis.

Performance estimations based on source code analysis consider directly the C/C++ instructions of the basic block. They associate a number of cycles per instruction to each C operator. Using these values the total number of cycles required to execute each block is estimated. The associated time per instruction is obtained depending on the compiler and the target platform. Using simple mathematical operations, the number of cycles required to execute large sections of code is obtained (Brandolese et al, 2001; Posadas et al, 2004). Compared with the other two solution types described below, this solution is the most platform-independent one. No operational SW infrastructure for the target platform is required: no compiler, no operating system or libraries, etc. However, the other two solutions are more accurate, especially because no compiler optimizations can be considered in this one.

Estimations obtained from analysis of the intermediate code enable considering compiler optimizations, at least the optimizations that do not depend on the target instruction set. The basic idea is to identify the instructions of the basic blocks of the source code in the intermediate code. Analyzing the blocks in the intermediate code it is possible to obtain more accurate information than that obtained with the source level analysis. The main benefit obtained from using intermediate code is that the task of extracting the relationships among the basic blocks of the source code and the intermediate code is much simpler than with final cross-compiled code (Kempf et al, 2006; Hwang et al, 2008; Bouchima et al, 2009).

However, this technique presents several limitations. First, not all compiler optimizations can be analyzed. Second, the intermediate code is completely dependent on the compiler, so the portability of the solutions is limited. To solve those limitations, a few proposals for analyzing the cross-compiled binary code have been also presented.

Estimations based on binary code are based in the relationships between the basic blocks of the source code and the cross-compiled code (Schnerr et al, 2008). Since the code analyzed is the real binary that is executed in the target platform, no estimation errors are added for wrong consideration of the compiler effects. The problem with these estimations is how to associate the basic blocks of the source code to the binary code (Castillo et al, 2010). Compiler optimizations can provoke important changes in the code structure. As a consequence, techniques capable of making correct associations in a portable way are required.

Moreover, different efforts for modelling the effect of the processor caches in the SW execution have been proposed. In (Schnerr et al, 2008) a first dynamic solution for instruction cache modelling has been proposed. Another interesting proposal was presented in (Castillo et al, 2010). Additionally, also solutions for data cache modelling have been proposed (Gerslauer et al, 2010; Posadas et al, 2011).

This chapter proposes some solutions for making the basic block estimations, providing different ratios between speed and accuracy, always maintaining complete portability for its application to different platforms. Cache solutions provided in (Castillo et al, 2010) and (Posadas et al, 2011) have been applied to optimize the final accuracy and speed.

2.3 Operating system modeling

The second element required to perform a correct native simulation is the modeling of the SW platform. That is, it is required to model the operating system (Zabel et al, 2009; Becker et al, 2010). Concurrency support, scheduling, management of priorities and policies and services for communication and synchronization are critical issues in SW execution. Several solutions have been proposed to simulate SW codes on specific OSs. Some operating system providers include OS simulators in their SW development kits (ENEA; AXLOG). These simulators enable the development and verification of SW functionality without requiring the HW platform. However, these simulators only model the processor execution, without considering other elements of the final system. This limitation has two different drawbacks. First the simulators are not adequate for evaluating the system performance. Additionally, the simulation of the SW with application-specific HW components is not possible. As a result they are not adequate for its integration in co-design flows.

In order to obtain optimal HW/SW co-simulation environments with good relations between accuracy and speed for the early stages of the design process, it is necessary to develop models of RTOS based on high-level modeling languages. Several models based on SpecC (Tomiyama et al, 2001; Gerstlauer et al, 2003) and SystemC (Hassan et al, 2005; He et al, 2005; Schirner et al, 2007) have been proposed. However, most of these solutions have limited functionality and proprietary interfaces, which greatly complicate the modeling of real application SW codes (Gerstlauer et al, 2003; He et al, 2005; Yoo et al, 2002). Most of these models are limited to providing scheduling capabilities. Later a few models of specific

operating systems have been proposed (Honda et al, 2004; Hassan et al, 2005). However, these RTOS models were very light and with reduced functionality.

Given the need of providing more complete models for simulating MPSoC operating systems, the infrastructure presented in this chapter starts from a very complete operating system model based on the POSIX interface and the implementation of the Linux operating system (Posadas et al, 2006). This chapter proposes an extension of this work to support different operating Systems. The models of the common operating systems uC/OS and Windows APIs are provided. As a result, the increasing complexity and heterogeneity of the MpSoCs can be managed in a flexible way.

3. Previous technology

As stated above, one of the main elements in a system modelling environment based in native simulation is the operating system model. It is in charge of controlling the execution of the different tasks, providing services to the application SW and controlling the interconnection of the SW and the HW. For that purpose, a model based on the POSIX API is used. The model uses the facilities for thread control of the high-level language SystemC to implement a complete OS model (Figure 1). Threads, mutexes, semaphores, message queues, signals, timers, policies, priorities, I/O and other common POSIX services are provided by the model. This work has been presented in (Posadas et al, 2006).

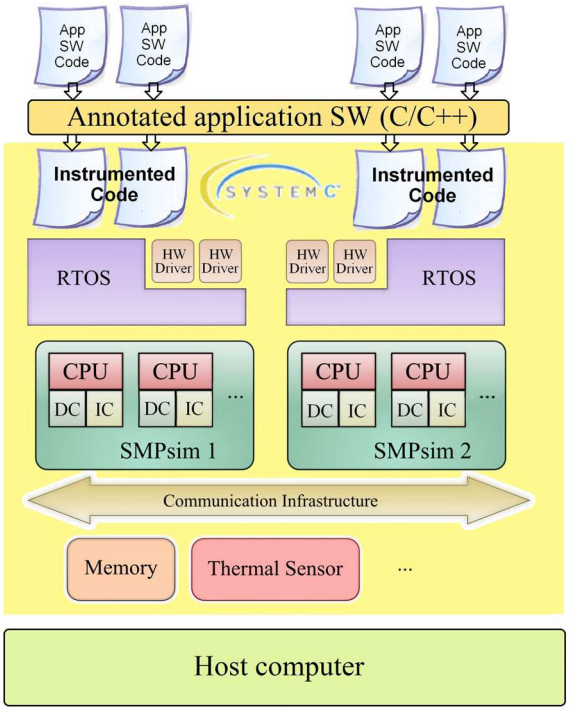


Fig. 1. Structure of the previous simulation infrastructure.

Special interest in the operating system model has the modeling of separated memory spaces in the simulation. As SystemC is a single host process, the integration of SW components containing functions or global variables with the same names in a single executable, or the execution of multiples copies of components that use global variables result in name collisions. To solve that, an approach based on the use of protected dynamic variables has been developed (Posadas et al, 2010).

However, the OS model is not only in charge of managing the application SW tasks. The interconnection between the native SW execution and the HW platform model is also performed by this component. For that goal, the model provides functions for handling interrupts and including device drivers following the Linux kernel 2.6 interfaces.

Additionally, a solution capable of detecting and redirecting accesses to the peripherals directly through the memory map addresses has been implemented. Most embedded systems access the peripherals by accessing their registers directly through pointers. However, in a native simulation, pointer accesses do not interact with the target HW platform model, but with the host peripherals. In fact, accesses to peripherals result in segmentation faults, since the user code has no permission to perform this kind of accesses. To solve that, these accesses are automatically detected and redirected using memory mappings ("mmap()"), interruption handlers, and code injection, in order to work properly (Posadas et al, 2009).

Furthermore, a TCP/IP stack has been integrated in the model. For that purpose, the open-source, stand-alone lwIP stack has been used. The stack has been adapted for its integration into the proposed environment both for connecting different nodes in the simulation through network models, and for connecting the simulation with the IP stack of the host computer, in order to communicate the simulation with other applications.

As a consequence, the infrastructure has demonstrated to be powerful enough to support the development of complete virtual platform models. However, improvements in the API support and performance modelling of the application SW are required. This work proposes solutions to improve them.

4. Virtual platform based on native simulation: goals and benefits

The goal of the native infrastructure is to provide a tool capable of assisting the designer during the initial design steps. More specifically, the infrastructure has been developed to provide the following services to the designers:

- Simulate the initial system models to check the complete functionality, before the platform is available, including timing effects.
- Provide performance estimations of the system models to evaluate the design decisions taken.
- Provide an infrastructure to start the refinement of the HW and SW components and their interconnections from the initial functional specification
- Work as a simulation tool integrated in design space exploration flows together with other tools required in the process

The first goal is to provide the designer with information about the system performance in terms of execution time and power consumption to make possible the verification of the

fulfilment of the design constraints. This verification can be performed in two ways. First, the infrastructure reports metrics of the whole system performance at the end of the simulation, in order to enable the verification of global constraints. This solution allows "black box" analysis, where designers can execute several system simulations running different use cases, to easily verify the correct operation in all the working environments expected for the system.

A second option enabled by the infrastructure is to perform the verification of the system functionality and the checking of internal constraints. These internal constraints must be inserted in the application code using assertions. For that purpose, the use of the standard POSIX function "assert" is highly recommended. The infrastructure offers to the designer functions that provide punctual information about execution time and power consumption during simulation. Using that functions, internal assertions can check the accomplishment of parameters as delays, latencies, throughputs, etc.

A second goal of the infrastructure is to provide useful information to guide the designers during the development process. The co-design process of any system starts by making decisions about system architecture, HW/SW partitioning and resource allocation. To take the optimal decisions the infrastructure provides a fast solution to easily evaluate the performance of the different solutions considered by the designer. Task execution times, CPU utilization, cache miss rates, traffic in the communication channel, and power consumption in some HW components are some of the metrics the designer can obtain to analyze the effects of the different decisions in the system.

Another goal of the infrastructure is to provide the designers with a virtual platform where the development of all the components of the system can start very early in the design process. In traditional development flows, some components, such as SW components, cannot start their development process until a prototype of the target platform is built. However, it increases the overall design time since HW and SW components cannot be developed in parallel.

To reduce the design time, it is provided a solution for HW/SW modeling where the design of the SW components can be started. To enable that, the infrastructure provides a fast simulation of the SW components considering the effects of the operating system, the execution time of the SW in the target platform and enabling the interaction of the SW with a complete HW platform model. Even, the use of interruptions and drivers can be modelled in the simulation. The execution of the SW is then transformed in a timed simulation, where the use of services such as alarms, timeouts or timers can be explored in order to ensure certain real-time characteristics in the system.

Furthermore, the simulation of the SW using a native execution improves the debugging possibilities. Designers can directly use the debuggers of the host system, which has a double advantage: first, it is not necessary to learn how to use new debugging tools; second, the correct operation of the debuggers are completely guaranteed, and does not depend on possible errors in the porting of the tool-set to the target platform. Additionally, designers can easily access to all the internal values of both the SW and HW components, since all are modelled using a C++ simulation.

In order to achieve all these goals, the infrastructure implements a modeling infrastructure capable of supporting complete native co-simulation. The infrastructure provides novel

solutions to enable automatic annotation of the application SW, a complete RTOS model, models of most common HW platform components and an infrastructure for native execution of the SW and its interconnection with the HW platform. Additionally, it is possible to describe configurable systems obtaining system metrics.

5. SW estimation and modeling

As a stated before, SW modeling solutions have become one of the most important areas of native simulation technology. The fastest possible execution of the system functionality is the direct compilation and execution of the code in the host computer. Thus, the goal is to provide a modeling solution capable of evaluating system performance, but maintaining a similar execution speed, as long as possible. Specially, the modeling solution has to overcome the three main limitations of functional execution with a minimum simulation overhead. First, functional executions do not consider any timing effect resulting of executing the code in the target platform. As a consequence, no performance information and no constraint checkings are available. Second, these executions cannot interact with the functionality implemented as HW components in the target platform. Thus, the simulation of the entire system functionality and the verification of the HW/SW integration are not possible. Finally, there is a problem when trying to execute a SW code developed for other OS APIs different from the native API.

To solve the first limitation, the solution proposed is to automatically modify the application SW in order to model performance effects. These performance effects include the execution of the code in the target processor core and the operation of the processor caches. The general solution applied for that modeling is based on estimating the effects during SW execution and apply them to the simulation, just before the points where the SW tasks start communications with the rest of the system, usually system calls. Four main solutions have been explored for obtaining the estimations: modified host times, the use of operator overloading and static annotation of basic-blocks at source and binary level. As a consequence, designers can modify the simulation speed and accuracy according to their needs on each moment.

The general annotation infrastructure enables using any of the estimation techniques with a virtual platform. Even, they can be combined in the same simulation. It depends on the method selected how to apply the estimated times for each SW component to increase the simulation time. The basic idea is to apply the estimated times when a system call is performed. This is caused because system calls are the points where communications and synchronizations are executed, that is, when SW tasks interacts with the rest of the system.

5.1 SW estimation based on modified host times

The first technique implemented is based on the use of the execution times of the host computer. As the time required for a processor to execute a code depends on the size of the functionality, there is a relationship between the time a SW execution takes in the host computer and in the target platform. Thus the idea is to run the simulation on the native PC getting the time required to execute each code segment. The estimated time costs of the components in the target platform are estimated by multiplying the time required to execute

in the host computer by an adjustment factor. This factor is based on the characteristics of the native PC and the target platform.

Unlike the other techniques presented below, this solution does not require the generation of annotated SW code. The original code is executed as it is, without additional sentences. Estimation and time modeling is done automatically when the system calls of the OS model are executed. The execution time of each segment is obtained by calling the function "clock_gettime ()" of the native operating system (Figure 2). To minimize the error produced by the other PC tasks, the simulation must be launched with the highest possible priority.

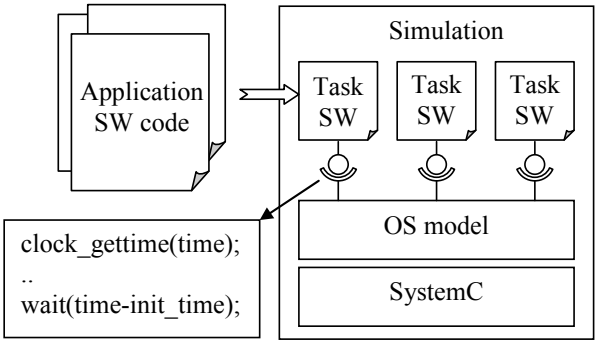


Fig. 2. Modeled by native time setting.

This solution has the advantage of being very fast, because no annotations increasing the execution time are needed. Nevertheless, a number of disadvantages hinder their use in most cases. First, we must be able to ensure that the simulation times obtained are really due to the execution of system code, and no caused by other parasite processes that were running on the computer. Second, the solution is not able to model cache behaviour adequately. Moreover, as only the execution time information can be obtained from the simulation, the transformations applied to obtain times of the target platform are reduced to a linear transformation. However, there is no guarantee that the cost of the native PC and the platform fits a linear relationship. On the contrary, the existence of different hardware structures, such as different caches, memory architectures or mathematical co-processors can produce significant errors in the estimation.

Summarizing, this solution is recommended only for very large simulations or codes where the accuracy obtained in performance estimations is not critical. Additionally, it is a good solution to estimate time of SW components that cannot be annotated. For example, some libraries are provided only in binary format. Thus, annotations are not possible since source code is not present. As a result, this solution is the only applicable of the four proposed.

5.2 SW estimation based on operator overloading

The estimation technique using operator overloading calculates the cost of SW as it progresses. Each operation executed must be accompanied by a consideration of the time cost it requires in the target platform. The temporal estimation of an entire SW code segment is obtained accumulating the times required to perform all the operations of a segment. This solution will avoid costly algorithms and static calculations, avoiding getting oversized

times, as in the case of techniques for estimating worst case (WCET), or the consideration of false paths. That way, the estimated time depends on exactly the code that is executed.

The solution relies on the capability of C++ to automatically overload the operators of the user-defined classes. Using that ability, the real functional code can be extended with performance information without requiring any code modification. New C++ classes (generic_int, generic_char, generic_float, …) have been developed to replace the basic C data types (int, char, float, …) . These classes replicate the behavior of the basic data type operators, but adding to all the operator functions the expected cost of the operator in the target platform, in terms of binary instructions, cycles and power consumption. The replacement of the basic data types by the new classes is done by the compiler by including an additional header with macros of the type:

"#define int generic_int"

A similar solution is applied to consider the cost of the control statements.

To apply that technique, a table with the cost of all the operators and control statements in the target platform must be provided by the user.

The operating mechanism of this estimation technique can be seen in Figure 3. First, the original code is modified by replacing the original data types of the SW by new classes overloaded. This is done automatically using compiler preprocessor C. The new classes are provided by the simulation infrastructure. There is a class for each basic data type, which stores the value of the data type and the cost of each operation for this operator. The resulting code is executed using the overloaded operators.

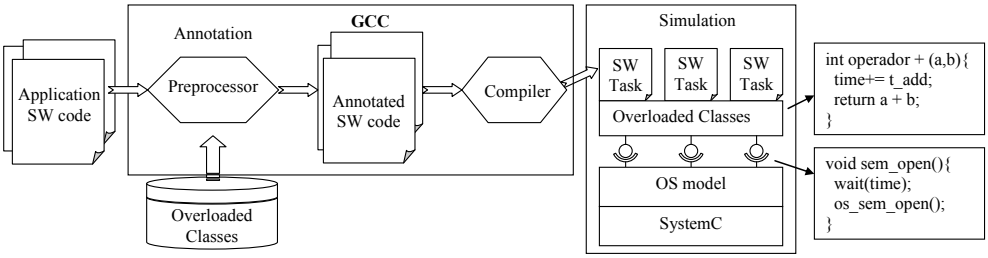


Fig. 3. Temporal model with operator overloading.

The original application code is compiled without any prior analysis or modification. Therefore, the operator overloading modeling technique is completely dynamic. All operations performed in the code are monitored by the annotation technique. This implies that the technique has enormous potential as a technique for code analysis. Studies on the number of operations, or monitoring data types of variables can be easily performed minimally modifying the overloading of operators.

This solution has demonstrated to be easy to implement, and very flexible to support additional evaluations, since all the information is managed dynamically, including the data values. Nevertheless, this solution has several limitations if the solely objective of the simulation is the estimation of execution times. Compiler optimizations are not accurately considered. Only, a mean optimization factor can be applied. Furthermore, the use of

operator overloading for all the data types implies a certain overhead, which slows down the simulation speed.

5.3 Annotation from source-code analysis

To obtain simulations with really low overhead, it is needed to move analysis effort from simulation to compilation time. Solutions based on static annotation divides the performance modeling in two steps. First, the source code is statically analyzed, obtaining performance information for each basic block of the source code. After that, this information is annotated in the code, and the cost of each basic block executed is accumulated during the simulation and applied at system calls.

As in the technique of operator overloading, this estimation technique is based on assigning a time cost to each C operator. The total cost of each segment of SW code is estimated by adding the time of the operators executed in the segment. The cost of each operator is calculated in the same manner as shown in the previous technique. As a consequence, the effects of compiler optimizations are difficult to estimate from the analysis of source code. For this reason, an adjustment factor can be provided to the simulation to consider improvements introduced by compiler optimizations. This factor is obtained comparing the sizes of SW code segments both optimized and not optimized.

For the static analysis, a parser based on an open-source C++ grammar has been implemented. The parser analyzes the source code, obtaining the number and type of operators used on each basic block, as long as the control statements at the beginning of each block. Using that information and the table with the cost of each operator used for the previous technique it is possible to obtain the cost for the entire basic block. Then, this cost is applied in the source code in the following way:

"segment_cycles += 120; segment_instructions += 20;"

As a result, the variables segment_cycles and segment_instructions accumulate the total cycles and instructions required to execute the entire code in the target platform. The complete sequence of tasks necessary to perform the estimation based on source code analysis is shown in the next figure.

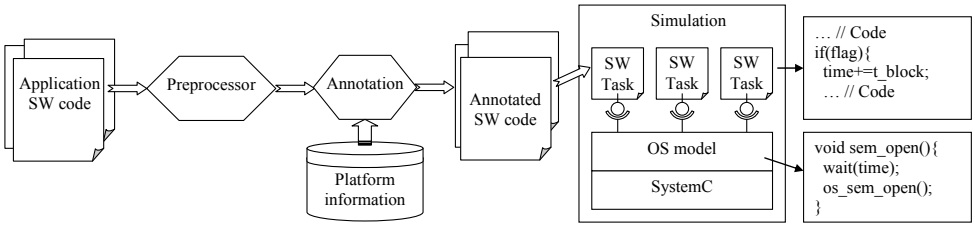


Fig. 4. temporal modeling with source-code analysis.

This solution requires more development effort than the operator overloading technique, especially for the implementation of the parser using the yacc/lex grammar. However, the simulation speed is really improved, achieving simulation times very close to the functional execution times (only two or three times slower). The main limitation of the technique is,

again, the impossibility of accurately considering the compiler optimizations, since no analysis of the compiler output is performed.

5.4 Source annotations based on binary analysis

The last solution proposed is capable of maintaining the qualities of the previous annotation technique, but providing more accurate results, including compiler optimizations. In this solution, the analysis of the source code is replaced by an analysis of the cross-compiled binary code. The use of compiled code instead of source code enables accurately considering all the effects of cross compiler optimizations. Once identified the assembler instructions corresponding to each basic block of the SW code, the number of instructions of the blocks and the cycles required to execute them are annotated in the source code.

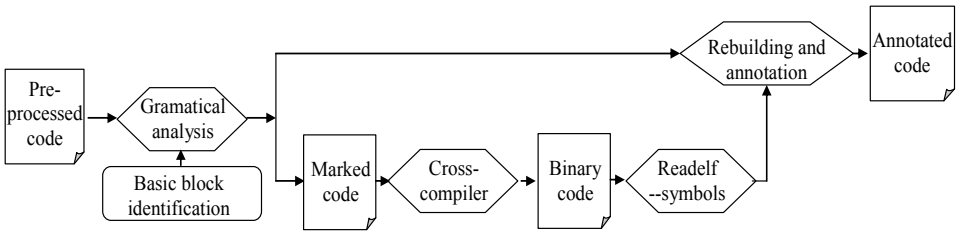


Fig. 5. Estimations with analysis of binary code.

However, estimations based on binary code usually present two limitations: first, it is difficult to identify the basic blocks of the source code in the binary code, and second, these solutions are usually very dependent on the processor. In order to build a simulation infrastructure fast and capable of modelling complex heterogeneous embedded systems, both issues have to be solved.

The correlation between source code and compiled code is sometimes very complex (Cifuentes) This is mainly due to results of the compiler optimizations as the reordering of instructions and dead code elimination. Furthermore, the technique should be easily portable to allow evaluation of different processors with minimal effort. To easily extract the correlation between source code and binary code, the proposed solution is to mark the code using labels. Both the annotation and identification of the positions of the labels can be done in a manner completely independent of the instruction set of the target processor. The annotation of labels in the code is a standard C feature, so it is extremely portable. Additionally, there are several standard ways to know the address of the labels in the target code, such as using the bin-utils or reading the resulting assembler code. Thus, the technique is extremely portable, and well suited to handle heterogeneous systems.

However, including compiler optimizations implies another problem. Compilation without optimizations enables easily identifying points in the binary code by inserting labels in the source code. However, the optimizations have the ability to move or even remove those labels. For example, if we insert a label in a loop, and apply an optimization of loop unrolling, the label loses its meaning. In order to avoid the compiler to eliminate the labels, they are added to the code of the form:

```
asm volatile("etiqueta_xx:");
```

The use of volatile labels forces the compiler to keep the labels in the right place. Thus, inserting labels at the beginning and end of each basic block we can easily obtain the number of assembly instructions of each basic block. The identification of basic blocks in the source code is made by a grammatical analysis. This grammatical analysis is done by a pre-compiler developed using "lex" and "yacc" tools, as in the estimation technique of source code analysis. This will locate the positions where the labels first and add annotations later.

Getting the value of the labels can easily be done using the command:

```
readelf –s binary_code.o | grep label_
```

The estimated time required to execute each basic block in the target platform is obtained by multiplying the number of instructions by the number of cycles per instruction (CPI) provided by the manufacturer. Although this solution carries a small error, such as not considering stops by data dependencies, it has the advantage of being fast and generic. To evaluate the behavior of a program on one processor, only a cross compiler for that processor is need. Libraries, operating systems or simulators as ISSs adapted specifically for the target platform are not required, resulting in a very portable and flexible approach.

However, with the introduction of volatile labels the compiler behaviour is still partially changed. Most of the optimizations, such as the elimination of memory accesses by reusing registers are correctly applied. But a few optimizations, with minor effects cannot be performed. Loop unrolling is not possible, although its use for processors with cache is unusual because it increases cache misses. The reordering of instructions to avoid data dependencies is also altered, but since the processor's internal effects are not modeled, this optimization has small effect on the estimation technique.

5.5 Cache modelling and pre-emption modeling

Nevertheless, the performance of the SW in the target platform does not only depend on the binary instructions executed. Processor caches also have an important impact on it. Common cache models are based on memory access traces. However, in native co-simulation no traces about the accesses in the target platform are obtained. As a consequence, new solutions for modeling both instruction and data caches have been explored and included in the infrastructure.

The modeling of instruction caches is based on the fact that instructions are placed sequentially in memory, in a place known at compilation time. Knowing the amount of assembler instruction for each basic block it is possible to obtain a relative address for the instructions with respect to the beginning of the "text" section of the "elf" file. This information is used as variables' address to access the cache model, instead of the real access trace. Additionally, the use of static structs has been applied in order to speed-up the simulation speed, achieving a similar error and overhead for instruction cache modeling than for the static time annotation (Castillo et al, 2010).

For data caches, the solution proposed uses corrected host addresses for each data variable used in the code. Additionally, global arrays handling information about the status of all the possible memory cache lines are used to improve the simulation speed maintaining the balance of the two previous techniques. The technique is described more in detail in (Posadas et al, 2011).

A final issue related to modeling the performance of the application SW is how to consider pre-emption. With the proposed modeling solutions, the segments of code between function calls are executed in "0" time, and after that, the time estimated for the segment is applied using "wait" statements. As a consequence, pre-emption events are always received in the "wait" statements. Thus, the segment has been completely executed before the information about the pre-emption arrives. As a consequence, the task execution order and the values of global variables can be wrong. In order to solve these problems, several solutions have been proposed in "Real-time Operating System modeling in SystemC for HW/SW co-simulation" (Posadas et al, 2005). The final solution applied is to use interruptible "wait" statements. This approach solves the problems in the task execution order. Additionally, it is considered that possible modifications in the values of global variables are not a simulation error but an effect of the indeterminism resulting of using unprotected global variables. In other words, it is not really an error but only a possible solution.

6. Operating system modeling

6.1 Support of multiple APIs

On of the main advantages of the underlying infrastructure selected to create the virtual platform infrastructure is the use of a real API. Since an implementation of a complete POSIX infrastructure is provided, most of the platforms based on Linux-like operating systems or other operating systems providing this API can be modelled. Then the infrastructure is able to support real software for a certain amount of platforms. However, other operating systems are used in embedded systems. As a really useful infrastructure has the goal of providing wide support in order to decide at the beginning of the design process the most adequate platforms for an application, support of other operating systems is recommended. Thus, in this work the extension of the infrastructure in that way has been evaluated. To do so, two different operating systems of wide use in embedded systems have been considered: a simple operating system and a complex one. As simple OS, uC/os-II has been selected. As complex OS, the integration of a win32 API has been performed.

6.1.1 Support of uC/os-II

µC/OS-II is a portable, small operating system developed by the Micrium company to be integrated in small devices. It is configurable and scalable, requiring footprints between 5 Kbytes to 24 Kbytes. This operating system provides a preemptive, real-time deterministic multitasking kernel for microprocessors, microcontrollers and DSPs. As a real-time kernel, the execution time for most services provided by µC/OS-II is both constant and deterministic; execution times do not depend on the number of tasks running in the application.

In order to easily implement the µC/OS-II API support the adopted approach has been to generate a layer on top of the existing POSIX API. Then, the implementation of the services only requires in most of the cases to adapt the interface of the µC/OS-II API to call a similar function in the POSIX infrastructure. Following that way, a list of 81 functions of the µC/OS-II API has been implemented. The following services have been implemented:

- Functions for OS management, such as starting the kernel, controlling the scheduler, or managing interrupts.

- Functions for task management, such as starting, stopping and resuming a task or modifying the priority
- Services for task synchronization: mutexes, semaphores and event flag groups.
- Services for task communication: message queues and mailboxes
- Memory management
- Time management and timers

As the POSIX infrastructure is quite complete, the task of generating this layer has resulted relatively easy. This demonstrates the validity of the infrastructure proposed to support other small operating systems.

6.1.2 Support of Win32

Although in the embedded system market Microsoft does not have the dominant position than in the PC (Laptop, Desktop and Server) market, the company through their Windows CE and Windows Mobile, now Windows Phone, holds an important market share which can even increase in the near future once Windows CE is offered under 'shared source' license and after the Nokia-Microsoft partnership. Thus, solutions to support of win32 API in a virtual platform modeling infrastructure results of great interest.

The proposed approach is to integrate virtualization of Win32 on the POSIX API of the performance analysis framework. As it is shown below, the overload of this approach is small. The virtualization framework is provided by the open-source code WINE. WINE is a free software application that aims to allow Unix-like computer operating systems to execute programs written for Microsoft Windows. WINE implements a Windows Application Programming Interface (Win32 API) library, acting as a bridge between the Windows application and Linux.

One of the reasons to use WINE is that, in accordance with the "Wine Developer's Guide", its architecture and kernel are based on the architecture and kernel of Windows NT, so that its behavior will be the same as most of the Windows operating systems, particularly those mostly used in embedded applications like Windows CE and Windows Phone.

Figure 6 shows in grey color the Windows NT architecture allowing the execution of Win32 application by the NT kernel. The white part of the Figure 6 represents the modules added for the construction of the Wine architecture.

Using the complete WINE architecture, the complete Windows NT architecture of Dynamic Link Libraries (DLL) is encapsulated by the WINE server and the WINE executable. The WINE executable virtualizes the underlying Unix kernel. For that purpose, additional DLLs and Unix-shared libraries are used.

The "WINE Server" acts as a Windows kernel emulator, executing the Win32 calls for thread creation, synchronization and destruction. It provides Inter-Process Communication (IPC). When a thread needs to synchronize or communicate with any other thread or process, is the Wine Server the handler of these actions making as an intermediary. The Wine server itself is a single and separated Unix process and does not have its own threading. Instead, it alerts whenever anything happens, such as a client having send a command, or a wait condition having been satisfied.

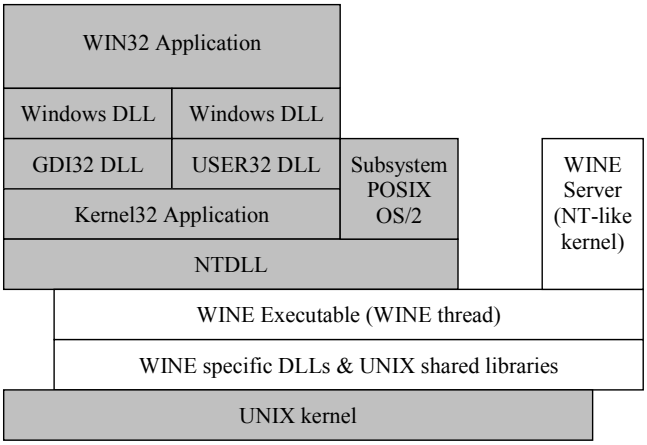


Fig. 6. Windows NT architecture + WINE Architecture.

The architecture of the integration of WINE on top of the POSIX model is shown in Figure 7. The most significant change from the WINE architecture of Figure 6 is the substitution of the POSIX subsystem, responsible for implementing the POSIX API functionality. In this way, the Win32 application is executed and its performance estimated by the native simulation infrastructure after the Win32 to POSIX translation.

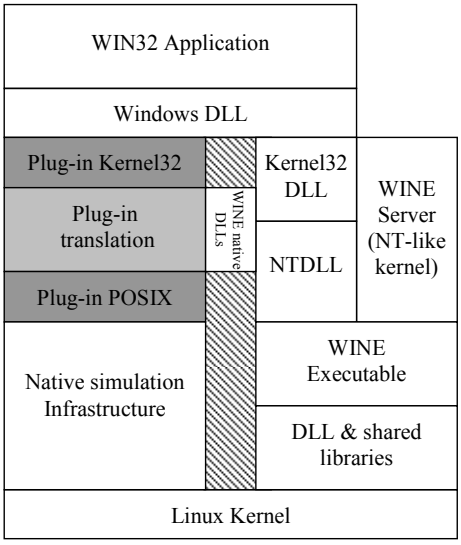


Fig. 7. Architecture of the WINE/native integration.

The WINE use is justified for the integration of WIN32 API in the native simulation framework. WINE allows us to abstract from the redeployment of Win32 functions for the execution in a POSIX system. Ideally, through this we can handle Win32's functions automatically by adding to our architecture the necessary libraries (DLLs).

However, when a simulation is being run, the user code can carry out calls to the API WIN32 functions. However, depending on which functions are being called, they are treated in two different ways. On the one hand, we have all those functions that are completely managed by WINE and that just need to be taken into account by native co-simulation in order to estimate the system performance in terms of execution times, bus loads and power consumption. On the other hand, there are other functions that are internally managed by the abstract POSIX native simulation kernel under the supervision of the WINE functions as they directly affect its kernel. The plug-in translation is responsible for these functions of thread creation, synchronization and destruction. When an API Win32 function is called, the plug-in analyzes and manages the handlers that have been generated by WINE. By default, the native WINE function is run, but in case the handle makes reference to a thread or object based on the synchronization of threads, it runs the translation to an equivalent POSIX function. In this way, the execution of these objects is completely transparent to the user.

As we said, part of the plug-in translation code is aimed at the internal management of the object's handles that are created and destructed in Wine as the user code requires. In the process of creating threads and synchronization objects, the code stores the resulting handle and the information that may be necessary for that regard. Thus, when any operation is performed on such handle, the plug-in can analyze and perform the necessary steps to carry out such operation.

The kind of services affected by such analysis are:

- Concurrency services (e.g. threads)
- Synchronization services (as semaphores, mutexes, events)
- Timing services (e.g. waitable timers)

In case that the handle belongs to any of the previous objects, it would be necessary to run the translation into an equivalent POSIX of the operation to be performed on this object so that it be performed by SCoPE correctly. Nonetheless, there are also other objects that are directly managed by the plug-in translation and do not require a previous analysis like Critical sections or Asynchronous Procedure Calls.

As shown in Figure 7, it is the “WINE Server” which acts as Windows kernel emulation, so that the thread creation, synchronization and destruction are performed through calls to this kernel. That is the reason why there is no literal translation for the behavior of these functions from the Win32 standard into the POSIX standard. An important contribution to this work and, therefore, an innovative solution to this problem, is the creation of a new code that is in charge of performing this task, maintaining the semantic and syntactic behavior of the functions of the affected Win32 standard. This is important in order to perform a translation by using only the calls to the POSIX standard functions, so that through the supervision of “WINE Server” our application is able to run those functions by respecting the Win32 standard at all times.

Finally, Graphics (GDI32) and User (USER32) libraries have been removed because they are not necessary in the functions currently implemented. As commented above, graphic interfaces are not supported yet as their modeling requires additional effort that is out of the scope of the current chapter. The user interface is not necessary when modeling usual embedded applications. Nevertheless, the proposed methodology for abstract modeling of complex OSs opens the way to solve this particular problem.

All the collection of functions of the API Win32 has been faithfully respected in accordance with the on-line standard of MSDN. To check it, a battery of simple tests has been developed to verify the correctness of some critical functions closely related with the integration of WINE with the simulation infrastructure. The tests generated include management of threads, synchronization means, file system functions and timers. The results have been compared with the same tests compiled and executed on a Windows platform (XP SP2 winver 0x0502) and in an embedded Windows CE platform, obtaining the same results in all the cases.

In the compilation process of a Win32 application in WINE, this one generated the scripts that are necessary to create a dynamic library from the application's source code, which is later loaded and run after the initialization process of WINE.

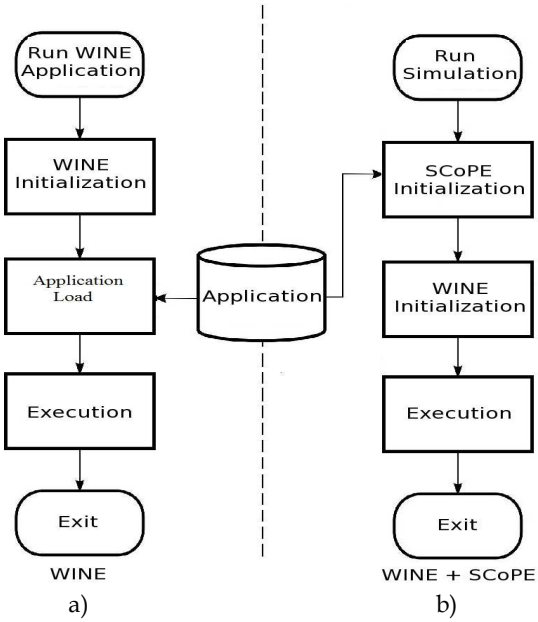


Fig. 8. WINE integration in the native simulation.

The process to generate a POSIX WINE executable from a Win32 application is shown in Figure 8-a. After WINE initialization, the scripts that are necessary to create a dynamic library from the application's source code are generated. Then, using these scripts, the application is loaded and executed. This application initialization and loading process is not compatible with the native co-simulation methodology.

The alternative process implemented is shown in Figure 8-b. The default initialization process of WINE is performed after the native co-simulation initialization process. The application is instrumented and loaded into the native simulation environment in this step. In order to support the parsing and back-annotation required by native co-simulation, it is necessary to integrate in the native co-simulation compiler the options required by WINE in order to recognize the application.

7. Results

Several experiments have been set-up in order to assess the proposed methodology. Firstly, simulation performance has been measured and compared with different execution environments of Win32 applications through small examples. Furthermore, a complete co-simulation case study has been developed showing the full potential of the proposed technology on a realistic embedded system design. After that some experiments have been performed to check the accuracy of the performance estimations.

7.1 Win32 simulation

In order to measure the simulation overhead of the proposed infrastructure, several tests focused on the use of OS services have been developed and instrumented. The tests have been carried out in four different scenarios, all on the same host computer:

- Proposed Win32 native simulation running on a native Linux platform (Fedora 11).
- WINE running on the same Linux platform.
- Windows XP SP2 running in a virtual machine (VirtualMachine 2.2.4) on the same Linux platform.
- Windows XP SP2 installed directly in the host.

The resulting execution times of the tests on the different scenarios are shown in Figure 9. As expected, the execution of Windows on a virtual machine is always slower than the OS directly installed in the host. Nevertheless, this is not the case when virtualising Windows with WINE. Results show that WINE can be faster than XP installed directly on the same host. This is not a surprising result and it has been already reported.

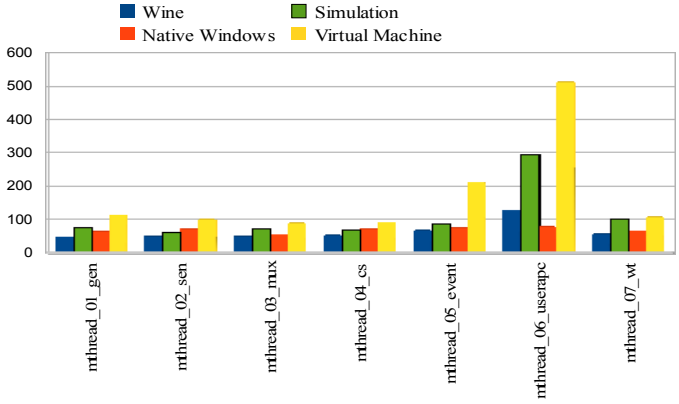


Fig. 9. Execution times.

As shown in Figure 9, native simulation is only 46% slower in average than WINE although the simulation is modeling execution times, data and instructions cache, memory and peripheral accesses, power consumption, etc. This result is coherent with the comparison figures between native simulation including performance estimations and functional execution. This explains why native simulation can be faster in some cases than functional execution on a Windows platform. This result shows the advantage of using WINE; we can

integrate native simulation on a virtualization of Windows, implementing most of its functionality and taking advantage of its fast implementation.

In order to assess the Win32 simulation technology in its final application of performance analysis of complex embedded systems including processing nodes using Windows, a heterogeneous system has been modeled, simulated and the performance figures obtained. The system is a low cost surveillance system taking low quality images from a camera at low speed (1 image per second) and coding and sending them through a serial link.

Apart from those simple examples, a complex example, a H.264 coder has been used for global correctness. This example makes an exhaustive use of calls to memory dynamic management functions, and there is also a writing of all the logs resulting from the codification when running. This part of the reference model has been modified so that the calls to the equivalent functions of the API Win32 are carried out in order to verify the correct operation of the plug-in this sort of operations. Dynamic memory management has been carried out through calls to the Global, Local and Heap memory management functions, and the file management through calls to the respective data input and output functions (e.g. CreateFile and WriteFile).

The system architecture is shown in Figure 10. It is composed of a Windows ARM node executing the H.264 coder, the camera taking the images, a memory where the input data are stored and the serial link taking the images and sending them out. The architectural exploration affects the selection of the most appropriate voltage-frequency and data and instruction cache sizes ensuring a CPU usage lower than 90% and a power consumption less than 1W.

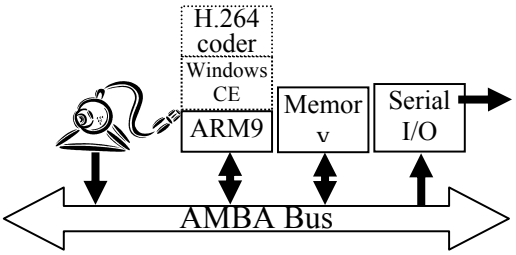


Fig. 10. Case study architecture.

Results of CPU usage and power consumption are shown in Figure 11. As can be seen, in this example, the size of the data and instruction caches do not affect too much the power consumption but the CPU usage.

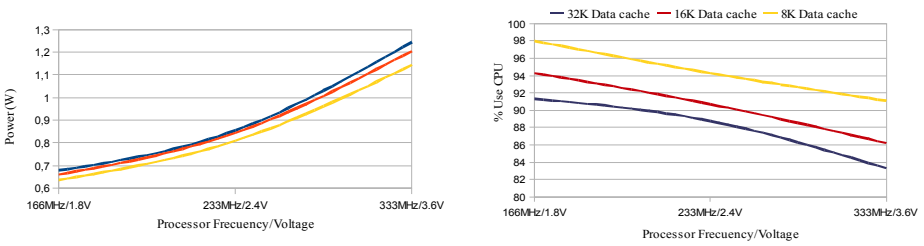


Fig. 11. CPU usage and Power consumption.

7.2 Win32 simulation performance

The proposed approach has been also applied to an ARM9 platform, in order to evaluate the accuracy of each on the techniques presented above. The ARM9 platform has been used to compare the estimation results of the different modeling solutions, in order to obtain the error when applied to one of the most popular processors in the embedded world.

As a summary of the final results achieved, the following tables show the estimation accuracy of the SW modelling, and the simulation times for a list of examples:

	Modified Host Time		Operator Overloading		Source Code analysis		Binary Code analysis	
	Error	Time	Error	Time	Error	Time	Error	Time
Bubble 1000	24.4	0.012s	14.8	0.75s	14.5	0.032s	12.5	0.030s
Bubble 10000	13.5	1.281s	3.5	81.6s	3.2	3.501s	0.01	3.486s
Vocoder	54.2	0.003s	24.2	0.41s	26.4	0.015s	18.3	0.014s
Factorial	34.5	0.013s	4.5	0.85s	4.1	0.042s	0.01	0.043s
Hanoi	47.9	0.082s	17.9	0.82s	16.9	0.271s	14.9	0.262s

Table 1. Comparison of estimation error (%) and simulation time for an ARM9 platform

As can be seen, the most accurate annotation technique is the solution based on the analysis of the binary cross-compiled code. After that, the technique based on source code analysis and the operator overloading are similar, since both rely on the same information (cycles of each C operator) and the same main source of error (optimizations). Finally, the modified host time is the less accurate one.

However, the technique of modified host tome is about 3 times faster than the annotation techniques based on code analysis, and more than 60 times than the operator overloading solution.

Finally, the results for cache modelling are shown in the next tables:

	Instruction Cache Misses					
	Without optimizations (-o0)			With optimizations (-o2)		
	Skyeye	Proposal	Error (%)	Skyeye	Proposal	Error (%)
Bubble 1000	15	16	6.66	6	5	16.67
Bubble 10000	25	27	8	7	7	0
Vocoder	8	7	12,5	5	4	20
Factorial	20	18	10	12	10	16.67
Hanoi	46074	46761	1.49	25842	28607	10.70

Table 2. Comparison of instruction cache misses ARM926t platform.

	Data Cache Misses					
	Without optimizations (-o0)			With optimizations (-o2)		
	Skyeye	Proposal	Error (%)	Skyeye	Proposal	Error (%)
Bubble 1000	126	127	0.80	126	126	0
Bubble 10000	5199772	5209087	0.18	5199310	5211595	0.24
Vocoder	375	500	33.33	375	500	33.33
Factorial	38	45	18.42	41	45	9.76
Hanoi	6018	5908	1.82	6026	5915	1.84

Table 3. Comparison of data cache misses ARM926t platform.

Summarizing, simulation speed-ups of two or more orders of magnitude can be achieved by assuming an acceptable error, below 20%.

8. Conclusions

In this chapter, several solutions have been developed in order to cover all the features required to create an infrastructure capable of obtaining sufficiently accurate performance estimation with very fast simulation speeds. These solutions are based on the idea of native co-simulation, which consists in the combination of native simulation of annotated SW codes with time-approximate HW platform models. All these techniques have been integrated in a simulation tool which can be used as an independent simulator or can be used integrated in different design space exploration flows.

The modeling solutions can be divided in two main groups: solutions for modeling in the native execution the operation of the application SW in the target platform, and a complete operative system modelling infrastructure. These solutions have been implemented as SystemC extensions, using the features of the language to provide multiple execution flows, events and time management.

The modeling of the application SW considers the execution times and power consumption of the code in the target platform, as long as the operation of the processor caches. Four different solutions for modeling the processor performance have been explored in the chapter (modified host times, operator overloading, annotation based on source code analysis and annotation based on binary code analysis), in order to find an approach capable of obtaining accurate solutions with minimal simulation overheads and as flexible as possible, to minimize the effort required to evaluate different target processors and platforms. As a result of the study, the annotation based on binary code analysis has demonstrated to obtain the best results with minimal simulation overhead. Additionally, the technique is very flexible, since only requires a cross-compiler for the target platform capable of generating object files from the source code. No additional libraries, ported operating systems, or linkage scripts are required. Additionally, it has been demonstrated that cache analysis for both instruction and data caches can be performed obtaining accurate results with adequate simulation times.

A POSIX-based operating system model has been also extended to support other APIs. Two different operating system APIs of wide use in embedded systems have been considered: a simple operating system and a complex one. Support for a simple OS, uC/os-II, has been integrated. As complex OS, the integration of a win32 API has been performed.

Summarizing, this chapter demonstrates that the SystemC language can be extended to enable the early modeling and evaluation of electronic systems, and providing important information to help the designers during the first steps of the design process. These extensions allow using a SystemC-based infrastructure for functional simulation, performance evaluation, constraint checking and HW/SW refinement.

9. Acknowledgments

This work has been supported by the FP7-ICT-2009- 4 (247999) Complex and Spanish MICyT TEC2008-04107 projects.

10. References

AXLOG, http://www.axlog.fr.

M.Becker, T.Xie, W.Mueller, G. Di Guglielmo, G. Pravadelli and F.Fummi, "RTOS-Aware Refinement for TLM2.0-Based HW/SW Designs", in DATE, 2010.

Benini et al, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC", Journal of VLSI Signal Processing n 41, 2005.

A. Bouchima, P. Gerin & F. Pétrot: "Automatic Instrumentation of Embedded Software for High-level HS/SW Co-simulation. ASP-DAC, 2009.

C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "Source-level execution time estimation of c programs," CODES 2001.

J. Castillo, H. Posadas, E. Villar, M. Martínez, "Fast Instruction Cache Modeling for Approximate Timed HW/SW Co-Simulation", 20th Great Lakes Symposium on VLSI (GLSVLSI'10), Providence, USA. 2010

C. Cifuentes. "Reverse Compilation Techniques". PhD thesis, Queensland University of Technilogy, 1994.

VaST Systems Technology. CoMET R. http://www.vastsystems.com/docs/CoMET_mar2007.pdf

CoWare Processor Designer, http://www.coware.com/products/processordesigner.php

ENEA: "OSE Soft Kernel Environment", in http://www.ose.com/products.

Gerstlauer, A. Yu, H. & Gajski, D.D.: "RTOS Modeling for System Level Design", Proc. of DATE, IEEE, 2003.

A. Gerslauer, "Host-Compiled Simulation of Multi-Core Platforms", Rapid System Prototyping, 2010

M. Gligor, N. Fournel, and F. Petrot, "Using binary translation in event driven simulation for fast and flexible MPSoC simulation", in CODES+ISSS, France, Oct. 2009.

G. Hadjiyiannis, S. Hanono & S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. Design Automation Conference, 1997.

M. Hartoog J.A. Rowson, P.D. Reddy, S. Desai, D.D. Dunlop, E.A. Harcourt & N. Khullar. "Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign". Design Automation Conference, 1997.

M.A. Hassan, K. Sakanushi, Y. Takeuchi and M. Imai: "RTK-Spec TRON: A simulation model of an ITRON based RTOS kernel in SystemC", Proceedings of the Design, Automation and Test Conference, IEEE, 2005.

Z. He, A. Mok and C. Peng: "Timed RTOS modeling for embedded System Design", Proceedings of the Real Time and Embedded Technology and Applications Symposium, IEEE, 2005.

S. Honda, T. Wakabayashi, H. Tomiyama and H. Takada: "RTOS-centric HW/SW co-simulator for embedded system design", Proceedings of CoDes-ISSS'04, ACM, 2004.

Y. Hwang, S. Abdi, D. Gajski. Cycle-approximate Retargetable Performance Estimation at the Transaction Level. DATE, 2008

T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, H. Meyr. "A SW Performance Estimation Framework for Early System-Level-Design Using Fine-Grained Instrumentation". DATE, 2006

R. Leupers, J. Elste, and B. Landwehr. "Generation of interpretive and compiled instruction set simulators". Asia and South Pacific Design Automation Conference, 1999.

A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr & Andreas Hoffmann, "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation", DAC, 2002

H. Posadas, F. Herrera, P. Sánchez, E. Villar, F. Blasco: "System-Level Performance Analysis in SystemC", Proc. of DATE, IEEE CS Press. 2004

H. Posadas, E. Villar, F. Blasco: "Real-time Operating System modeling in SystemC for HW/SW co-simulation", XX Conference on Design of Circuits and Integrated Systems, DCIS. 2005

H. Posadas, J. Adámez, P. Sánchez, E. Villar, F. Blasco: "POSIX modeling in SystemC", 11th Asia and South Pacific Design Automation Conference, ASP-DAC, 2006

H. Posadas, E. Villar: "Automatic HW/SW interface modeling for scratch-pad & memory mapped HW components in native source-code co-simulation", A. Rettberg, M. Zanella, M. Amann, M. Keckeiser & F. Rammig (Eds.): "Analysis, Architectures and Modelling of Embedded Systems", Springer, 2009

H. Posadas, E. Villar, Dominique Ragot, M. Martínez: "Early Modeling of Linux-based RTOS Platforms in a SystemC Time-Approximate Co-Simulation Environment", IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010

H. Posadas, L. Diaz, E. Villar: "Fast Data-Cache Modeling for Native Co-Simulation", Asia and South-Pacific Design Automation Conference, ASP-DAC, 2011

IBM PowerVM, http://www-03.ibm.com/systems/power/software/virtualization/

Qemu, http://www.qemu.org/

G. Schirner, A. Gerstlauer, and R. Dömer. "Abstract, Multifaceted Modeling of Embedded Processors for System Level Design". Asia and South Pacific Design Automation Conference (ASP-DAC), 2007.

J. Schnerr, O. Bringmann, A. Viehl, W. Rosenstiel. High-Performance Timing Simulation of Embedded Software. DAC, 2008

SkyEye web page, http://www.skyeye.org/index.shtml

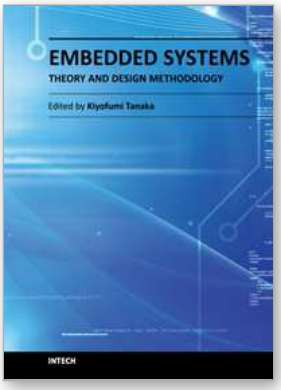
Synopsys, Platform Architect tool, http://www.synopsys.com/Systems/ArchitectureDesign/pages/PlatformArchitect.aspx

H. Tomiyama, Y. Cao and K. Murakami: "Modeling fixed-priority preemptive multi-task systems in SpecC", Proceedings of the 10th Workshop on System And System Integration of Mixed Technologies (SASIMI'01), IEEE, 2001.

UQBT, http://www.itee.uq.edu.au/~cristina/uqbt.html

S. Yoo, G. Nicolescu, L. Gauthier, A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in SoC design", Proc. of DATE, IEEE, 2002.

H. Zabel, W. Müller, and A. Gerstlauer, "Accurate RTOS modeling and analysis with SystemC", in "Hardware-dependent Software: Principles and Practice", W. Ecker, W. Mü ller, and R. Dömer, Eds. Springer, 2009.



Embedded Systems - Theory and Design Methodology
Edited by Dr. Kiyofumi Tanaka

ISBN 978-953-51-0167-3
Hard cover, 430 pages
Publisher InTech
Published online 02, March, 2012
Published in print edition March, 2012

Nowadays, embedded systems - the computer systems that are embedded in various kinds of devices and play an important role of specific control functions, have permitted various aspects of industry. Therefore, we can hardly discuss our life and society from now onwards without referring to embedded systems. For wide-ranging embedded systems to continue their growth, a number of high-quality fundamental and applied researches are indispensable. This book contains 19 excellent chapters and addresses a wide spectrum of research topics on embedded systems, including basic researches, theoretical studies, and practical work. Embedded systems can be made only after fusing miscellaneous technologies together. Various technologies condensed in this book will be helpful to researchers and engineers around the world.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821