

Chapter 2

M3-SCoPE: Performance Modeling of Multi-Processor Embedded Systems for Fast Design Space Exploration

Hector Posadas, Sara Real, and Eugenio Villar

Abstract Design Space Exploration for complex, multi-processor embedded systems demands new modeling, simulation, performance estimation tools and design methodologies. Recently approved as IEEE 1666 standard, SystemC has proven to be a powerful language for system modeling and simulation. In this chapter, M3-SCoPE, a SystemC framework for platform modeling, SW source-code behavioral simulation and performance estimation of multi-processor embedded systems is presented. Using M3-SCoPE, the application SW running on the different processors of the platform can be simulated efficiently in close interaction with the rest of the platform components. In this way, fast and sufficiently accurate performance metrics of the system are obtained. These metrics are then delivered to the DSE tools to evaluate the quality of the different configurations in order to select the best ones.

2.1 Introduction

System exploration with an optimum trade-off between performance and cost requires analyzing the performance of a large number of system configurations with a wide set of parameters, such as number and type of processors, memory architecture and sizing, mapping of SW tasks and suitability of communication infrastructure.

System simulation is a key design task widely used for design verification and evaluation. The main role of system simulation in embedded system design is to ensure the functional correctness of the design at the different abstraction levels. In Design Space Exploration (DSE), system simulation is used for performance analysis, providing to the exploration tool the required metrics such as delays, throughput, utilization rates, bandwidths, etc. Power consumption is becoming an additional, increasingly important metric to be estimated.

H. Posadas (✉)
University of Cantabria, Santander, Spain
e-mail: posadash@teisa.unican.es

Design Space Exploration requires auxiliary tools to provide the exploration engines with the metrics needed to evaluate the different system configurations. For evaluating complex HW/SW MPSoC systems [30], very flexible evaluation mechanisms are required. Static mechanisms are adequate to evaluate the effect of different parameter values in well known architectures. The analysis of internal processor components or cache configurations are examples in that context. However, to evaluate unknown or very flexible architectures, analysis methods based on mathematical equations are not applicable. Evaluation mechanisms based on simulation are then selected.


Simulation environments for DSE have to overcome several challenges. Mainly, these simulations require very fast speeds, considering the large amount of points to be simulated. Thus, modeling techniques need to evaluate all the configurations selected by the DSE tools without provoking additional delays. While HW simulation can be performed at different abstraction levels using appropriate languages such as VHDL, Verilog, System-Verilog and SystemC [58], efficient and sufficiently accurate SW simulation requires additional efforts. Electronic System Level (ESL) [4] has been proposed as an adequate abstraction level for complete system simulations [38]. At this level, there are three main methodologies used for SW simulation: Instruction-Set Simulation (ISS), virtualization with binary translation and native co-simulation.

The first solution, ISS-based HW/SW co-simulation, is the main industrial platform simulation technology supported by mature commercial tools offered by all the major vendors [40, 56]. Currently available commercial modeling and simulation tools are based on previous research activity in academia. In [7] a generic design environment for multiprocessor system modeling was proposed. The environment enables transparent integration of ISSs and prototyping boards. As an evolution of this work, in [8] a SystemC infrastructure was developed to model architectures with multiple ARM cores. This approach provides a set of tools that enables designers to efficiently design SW applications (OS ports, compilers, etc.). A fully operational Linux version for embedded systems was ported on the platform. Software simulation was based on an ISS for each processor, thus presenting the advantages and disadvantages commented above. Moreover, it cannot be easily used to evaluate platforms that do not contain ARM processors.

Several approaches have been proposed to improve the state-of-the-art of commercial tools (Fig. 2.1). One of them is the modification of the OS running over the ISS. As the OS is in fact the interface between SW applications and the rest of the system, it can be used to save simulation time. In [62], a technique based on virtual synchronization is presented to speed up execution of several SW tasks in the ISS. Only the application tasks run over the ISS. The OS is modeled in the co-simulation back-plane thus accelerating its simulation. As the OS execution time is only part of the total execution time, the gain is limited.

A recent technology proposed for SW simulation is using virtualization with binary translation. The most representative virtualization technology is QEMU [54]. SW emulation is based on virtualization. The binary code of the target processor is dynamically translated to the host executing the same functionality. In its original

Fig. 2.1 Approximate comparison of orders of magnitude for the different simulation mechanisms

Simulation type	Speed	Accuracy
Functional execution	100000	
Timed native co-simulation	10000	
Timed binary translation	1000	
ISS (instructions)	100	
ISS (cycle accurate)	10	
Pin accurate	1	

version, QEMU does not support execution time estimation for SW simulation, but some works has covered this aspect [20]. Providing QEMU with the required SW simulation capabilities is an active research area [6, 44]. Recently, Imperas has released its SW emulation technology as Open-Source [26].

Native co-simulation is based on the direct execution of the source code on the host. Simulation speed can be improved avoiding the use of processor models at the expense of some estimation accuracy in order to enable an efficient system evaluation [18]. Estimation errors of about 30–40% can be accepted for initial system assessments [29]. Using either static [5, 9, 24] or dynamic techniques [11, 31, 46] the SW execution times are estimated and annotated to obtain timed simulations of the application SW. A simple version of this technique has been implemented in a commercial tool [27]. This kind of analysis technique has proven to be useful for power consumption estimation [10, 34, 48] as well. Native simulation has shown its ability to estimate the number of cache misses together with execution time [14, 57].

An integral part of the embedded SW is the RTOS used. In the two binary simulation technologies commented above (ISS and virtualization), the complete embedded SW including the application SW and the RTOS are compiled together. In native simulation this is possible whenever appropriate modeling of the Hardware Abstraction Layer (HAL) functions is made [44]. The main advantage of this technology comes from the possibility to easily include all the Hardware-dependent Software (HdS), such as RTOS, drivers, etc. Whereas the main disadvantage is that one needs to develop, in advance, the complete embedded SW including the application SW, the chosen RTOS, drivers, etc. An additional limitation is that each RTOS and each microprocessor requires a specific HAL model.

An alternative for native SW simulation is based on using an abstract OS model. As the RTOS functionality is abstracted, this approach is faster than the HAL-based one. Several alternatives have been proposed, from generic [19, 22, 23, 63] to real OS models [21, 47]. The former technology is currently supported by some commercial tools [16, 17]. The latter approach has the advantage that, when efficiently exploited, a more accurate model of the underlying RTOS can be achieved. This additional efficiency is obtained by combining the source-code execution time estimation with the OS model providing more accurate results. The work in [12] provides an analysis of the impact of including the OS time in the overall system estimation.

These abstract RTOS techniques also dedicate a large effort to accurately integrate time annotation and OS modeling with HW/SW communication, especially for HW interrupt management. Very few of these approaches support a real OS Application Programming Interface (API) such as TRON or POSIX [21, 47]. The idea of integrating facilities for OS modeling in SystemC was proposed as a new version of the language some years ago by the SystemC consortium. However, OS modeling at system level proved to be a much more complex task than expected, becoming an active research area.

However, none of these modeling methodologies are aimed at complex Multi-Processor Systems-on-Chip (MPSoC) modeling. Current MPSoC modeling requires dynamic task mapping, drivers and interrupt management which are not covered by previous modeling techniques especially with the requirement of fast simulation speed during performance estimations and system dimensioning. Moreover, a framework is required that supports a complete model of the platform that can easily integrate new components, either an application-specific HW or a programmable processor. When the application SW code of each function has not yet been developed, the underlying simulation technology supporting native simulation can be used for high-level performance analysis [16, 17, 36].

In this chapter, M3-SCoPE, a framework for performance modeling of multi-processing embedded systems for fast DSE is described. The proposed system simulation technology includes abstract models of the RTOS and the multi-processing architecture that can easily integrate the application SW through the RTOS API (i.e. POSIX). The SW is annotated with estimations of the execution time and power consumption. The multi-processing architecture is connected through an abstract Transaction-Level Model (TLM) of the bus with the peripherals and application-specific HW components. Different nodes in the system can be connected through networks. The SW simulation technology includes novel features such as dynamic time estimation and cache modeling.

Additionally, to allow evaluating complete, heterogeneous systems in an efficient way, simulation tools require capabilities to automatically create the system models for the different configurations. On one hand, DSE tools do not have the capability to drive the creation of the platform models. On the other hand, need for manual recoding for each evaluation point results in too long exploration times. Several previous works have been proposed to solve or minimize the effort required for platform model creation. Automatic generation of system models oriented to specific target architectures has been proposed in [35, 60]. Other works have been focused on automating the exploration of component interconnection [33, 42, 61]. However, none of these approaches cover model generation for complete architectures in the general case.

To provide more generic techniques, TLM techniques based on system-level design languages like SystemC have been proposed [13, 25, 43, 55]. Green-socks is an open source infrastructure for distribution of TLM models [32]. In [45] a TLM framework for automatic system model generation is proposed. The framework receives a fixed system description and generates the executable system model. In [39, 51] TLM infrastructures are used to accurately estimate SW performance.

Some commercial tools [3, 16] can model designs at TLM level. The schematic entry tools simply provide a graphical interface for plugging existing database models together. These models are described and connected at the transaction-level. They also provide shell interfaces which allow modifying the characteristics of the system components. However, the system architecture is fixed and cannot be modified. An alternative solution to schematic entries for system description and model generation is using XML based descriptions. IP-XACT [28] standard describes an XML scheme for meta-data documenting Intellectual Property (IP) used in the development, implementation and verification of electronic systems. This scheme provides a standard method to document IP that is compatible with automated integration techniques. Several tools have been developed to support that integration [37, 41]. The resulting models can only configure certain parameters on the system components. However, modifiable platforms cannot be described through IP-XACT and modeled with these tools. Thus, the exploration of the best platform architecture cannot be performed with these tools.

In that context, the work presented here proposes a solution to describe modifiable architectures and automatically generate the corresponding system models. These models can be configured by modifying the parameters of the system components and also modifying the system architecture itself. This modeling capability will allow the DSE tools not only to find the optimal tuning of the system components, but also to optimize the system itself. System descriptions will be performed in a simple XML format, although the proposed solutions can be easily adapted to other XML descriptions.

2.2 Native Co-Simulation Infrastructure for DSE

As stated above, SystemC has been widely adopted for system modeling during last years. SystemC provides features to describe systems from RTL descriptions up to system-level models, in order to enable its application during all the first steps of the design flow. Furthermore, as SystemC is a library of C++, SW designers can integrate their C/C++ codes together with the HW platform descriptions to create models of the entire system. Then, the SW is compiled and executed natively in the host computer without requiring slow ISS models. As a consequence, this solution results in very fast simulation technology.

However, this solution, which can be used to verify the system functionality, presents several limitations in terms of additional performance estimations. The native execution of a SW code provides no information about the temporal behavior of the code in the target platform. Furthermore, this kind of execution of the embedded SW does not consider any kind of processor allocation nor the effects of an operating system. Finally, the HW/SW interfaces are not adequately modeled, so the effects of interrupts, device drivers or bus contentions are not taken into account.

SCoPE is a library that provides the extensions required by SystemC to enable the use of native SW execution together with SystemC HW descriptions in order to

obtain performance estimation of the entire system. These extensions include facilities to accurately estimate and model the temporal behavior of the application SW [52], to emulate the behavior of the operating system [47] and to enable realistic modeling of the HW/SW communication [50]. As a consequence, SCoPE is a powerful infrastructure for high-level system modeling during early system performance evaluation.

Nevertheless, an efficient modeling infrastructure for DSE requires additional features. Exploring the effects of different component frequencies, variable number of processors or different memory architectures also requires the capability to describe and automatically build the system models during the exploration. Common DSE tools are developed to select the experiments to be evaluated and to extract the best points from the Pareto curves. However, these tools have no intelligence to create the different system descriptions to be simulated. Current DSE tools are not capable of selecting, instantiating and connecting the components of the SystemC model depending on the selected configuration.

To solve this limitation, M3-SCoPE combines SCoPE features with an input/output interface where XML-based descriptions of highly configurable systems can be provided [49]. From these descriptions the library automatically creates the system model by instantiating and connecting all the components with its selected parameters. This capability avoids manual recoding during the exploration process, increasing the overall exploration efficiency. Additionally, the use of XML files enables run-time system modeling creation. Thus, no recompiling steps are required during simulations.

Summarizing, the main benefits provided by M3-SCoPE for the designers are the following:

- Capability of describing configurable system descriptions in XML format
- Automatic system model creation at run-time, avoiding recompiling steps
- Performance estimation (time and power) and annotation of the application SW
 - Modeling cache effects
 - Modeling compiler optimizations.
- Modeling the effects of the operating system and processor allocation
 - Scheduling, synchronization and communication
 - Memory space separation.
- HW/SW communication
 - Using device drivers and interrupts
 - Direct access to HW registers through pointers.
- Facilities to automatically estimate performance features from the HW components, to be delivered to the DSE tools.

Considering all these features, M3-SCoPE can be easily integrated in the DSE flow (introduced in Chap. 1) as shown in Fig. 2.2. For this integration two input XML files are defined to provide the system description: one XML file describing the system and its configuration options, called “*System Description*” file, and another XML file of pairs identifier-value, fixing the selected configuration for each experiment, called

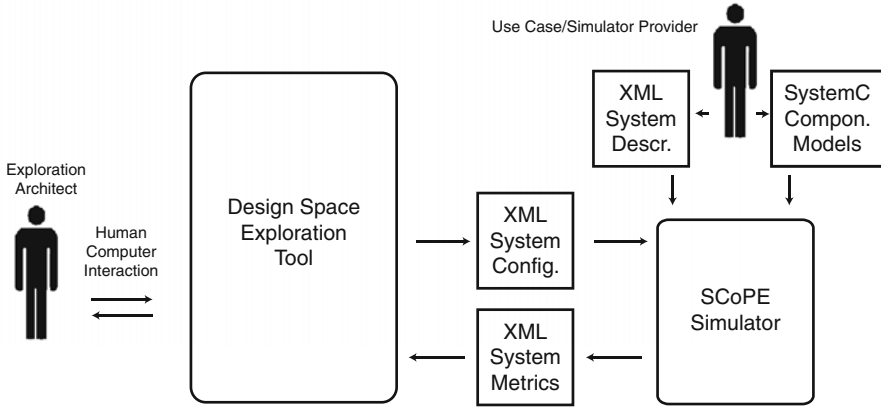


Fig. 2.2 Integration of the M3-SCoPE Simulator in the exploration flow. The system designer provides the configurable system description and the codes for all the system components. Then, DSE tool starts selecting configurations and receiving performance estimations from M3-SCoPE

“Configuration” file. An XML output file has been defined to return the simulation results. The external DSE explorer indicates the simulator which is the configuration to be analyzed each time by generating the corresponding *“Configuration”* file. The simulation tool interprets the file, builds the system model and performs the simulation. This means that no user interaction or model recompiling is required once the exploration process starts.

The tool generates an output file, when the simulation finishes. This file contains the values of the metrics obtained during the simulation. The output information is used by the DSE exploration architect to perform the search of the best solutions applying the RSM techniques.

2.2.1 Configurable XML System Descriptions

The XML System Description file includes information about the HW components, the HW architecture, and the SW tasks. A simple XML format has been developed to easily describe highly configurable platforms. The language guarantees fast model creation and efficient system simulation.

A simple example of an XML description using this language is shown in Fig. 2.3. To keep it simple, no configurable options have been added. The example proposes a system with a processor and a memory connected to a bus. A standard *“hello world”* application has been selected to execute in the processor. To describe a system with several platform architecture options and its configuration possibilities three XML mechanisms are provided by M3-SCoPE. All three mechanisms can be used simultaneously to describe highly configurable systems.

Fig. 2.3 Simple XML system description containing information about the HW components, its instances, the operating system, the SW applications and the task allocation. The description represents a simple system containing a processor, bus and memory, running a typical "Hello world" application

```

<HW_Platform>
  <HW_Components>
    <HW_Component category="bus" name="AMBA" frequency="200MHz" />
    <HW_Component category="processor" name="arm926t" frequency="200M"/>
    <HW_Component category="memory" name="Memory"
      mem_size="500MB" frequency="200MHz" mem_type="RAM" />
  </HW_Components>
  <HW_Architecture>
    <HW_Instance component="AMBA" name="my_bus" />
    <HW_Instance component="arm926t" name="my_proc" />
    <HW_Instance component="Memory" name="my_memory"
      start_addr="0x80000000" />
  </HW_Instance>
</HW_Architecture>
</HW_Platform>
<Application>
  <Functionality>
    <Exec_Component name="hello" category="SW" function="hello"/>
  </Functionality>
  <Allocation>
    <Exec_Instance name="Hello_world" component="hello"
      processor="my_proc"/>
  </Allocation>
</Application>

```

2.2.1.1 Configuration of the System Components

The first configuration option is used to tune the characteristics of the system components. The values of all parameters in the XML file can be replaced by identifiers when the parameter is a configurable one. For example, the bus frequency that was indicated in Fig. 2.3 (200 MHz) can be replaced by the identifier "FREQ".

To select a configuration, the values of all identifiers must be assigned in the System Configuration File. Thus, to perform different simulations it is only required to modify the value-identifier pairs in the System Configuration file (Fig. 2.2). Applying this solution, the simulation of each experiment required by the DoE is performed by substituting the identifiers of the configurable parameters by the selected values and creating the corresponding system model.

2.2.1.2 Replication of System Components

The second configuration option is to indicate the number of times a system component is replicated. To do so, a new XML "repeat" clause is provided. This clause defines the number of times the element is repeated, an index identifier and the initial index value. Figure 2.4 corresponds to an extension of the system description in Fig. 2.3 considering that the number of CPUs in the system can be set by the "CPUS" parameter. This parameter must be assigned in the System Configuration file.

The "repeat" clause can be used to replicate both single components and groups of components, copying complete parts of the system architecture. If the value is set to '0', the element is not placed in the system. This option is used to add or delete different components within the system, including modifying SW components, HW components and the communication infrastructures. As a consequence, different platform architectures can be described.

Fig. 2.4 XML description with configurable number of processors. Depending on the parameter “CPUS”, defined for each simulation by the DSE tool, the system model is created. Task allocation is modified depending on the number of processors

```
<HW_Platform>
  <HW_Components> ... </HW_Components>
  <HW_Architecture>
    <HW_Instance component="AMBA" name="my_bus" >
      <repeat number="CPUS" index="i" init="1">
        <HW_Instance component="arm926t" name="my_proc[%i]" />
      </repeat>
    <HW_Instance component="Memory" name="my_memory"/>
  </HW_Instance>
</HW_Architecture>
</HW_Platform>
<Application>
  <Functionality>
    <Exec_Component name="hello" category="SW" function="main" />
  </Functionality>
  <Allocation>
    <repeat number="CPUS" index="j" init="1">
      <Exec_Instance name="task[%i]" component="hello"
        processor="my_proc[%j]" />
    </repeat>
  </Allocation>
</Application>
```

Fig. 2.5 XML description with configurable HW architectures. The figure shows a two possible configurations for as an interconnection component: a bus or a Network-on-Chip (NoC)

```
<HW_Platform>
  <HW_Components> ... </HW_Components>
  <HW_Architecture name="arch1">
    <HW_Instance component="AMBA" name="my_bus" />
    ...
  </HW_Architecture>
  <HW_Architecture name="arch2">
    <HW_Instance component="NoC" name="my_noc" />
    ...
  </HW_Architecture>
</HW_Platform>
<Application> ... </Application>
<Simulation>
  < Implementation HW_Architecture="ARCH" />
</Simulation>
```

2.2.1.3 Selecting Complete Configurations

The third configuration option provided is to define several complete configurations and select one on each simulation. For example, in Fig. 2.5, two different HW architectures are described ("arch1" and "arch2"). The one selected for each simulation is defined in the "Implementation" clause. In this example, the architecture selected depends on the "ARCH" identifier. Its value must be set in the System Configuration file to "arch1" or to "arch2".

The system description mechanism allows dividing the system description in parts and exploring different combinations. Multiple HW component lists, HW architectures or SW allocations can be described to be explored by the DSE tool.

2.3 Modeling of SW Components through Native Simulation

2.3.1 Performance Modeling of Embedded SW

As mentioned before, native simulation is a very fast solution for functional modeling of SW components. However, to obtain estimations of the system performance the untimed functional executions must be transformed into timed SW models. To do so, the technique applied in M3-SCoPE has two steps. First, performance estimations of the SW code are done statically, adding time annotations to the SW code. Then, the code is executed, and these times are applied to obtain a timed simulation.

The simulation time of the application SW is estimated depending on three elements: the code, the compiler modifications and the cache effects. Additional times required by the processors accesses to the rest of the HW platform (memory or peripherals) are evaluated inside the HW platform models, and are not directly estimated by the SW modeling.


To consider the three elements, the code annotation is based on a basic-block solution. For each basic block, the time required for its execution, the power consumption and the required cache accesses are annotated. Using this basic-block information, M3-SCoPE obtains, during simulation, the total time, power and cache accesses by accumulating the values for all the basic blocks executed. Additionally, estimation of basic block metrics from a cross-compiled binary code allows to account for the compiler effects.

The estimated execution time is then applied to transform the native SW functional execution into a timed simulation. SystemC time annotation is performed by means of "wait (time)" sentences at communication and synchronization points, which correspond to system calls and I/O accesses. However, the use of standard "wait" clauses does not allow modeling preemption in the SW tasks, as the time is completely applied independently of additional events in the system, as interrupts. To solve that, the technique of pre-emptable waits is applied.

The application of this extra information involves a penalty in the simulation performance. Additional instructions require additional simulation time, which is opposite to the main requirement of DSE simulators: speed. As a consequence, these annotations must be minimal. The solution of basic-block annotation minimizes the problem. The use of global variables to accumulate time and power also helps to reduce the overhead (Fig. 2.6). But probably the most important solution applied is to move the cache address searches from the cache models to the code annotation, as it will be presented later.

In parallel to time estimations, the proposed techniques are able to estimate the power consumption required by the processor to execute the application software. The approach consists of assigning an average value of energy to each machine instruction. By analyzing the cross-compiled code, the library obtains the number of instructions for each basic block, and this value is back-annotated in the original source code along with time annotations.

Fig. 2.6 Example of extra code for annotating delays. At the end of each basic block the global variable “segment time” is increased with the block delay. At system calls the final time value is applied to the simulation



```

int a, b, c, i;
b = 2;
for (i=0; i<10; i++){
    c=a+b+i;
    a++;
}
i=0;
if (c>100) {
    b=0;
}

```

```

int a, b, c, i;
b = 2; segment_time+=10;
for (i=0; i<10; i++){
    c=a+b+i;
    a++;
    segment_time+=40; }
i=0; segment_time+=5;
if (c>100) {
    b=0;
    segment_time+=10; }

```

When running the simulation, this extra code is executed so we can estimate the number of machine instructions executed. By multiplying this value by the average energy per instruction provided in the XML files, an estimation of the total energy required by the processor is obtained. It should be noted that, while this approach is valid for all processors, the accuracy of the technique depends on the stability of the power consumption of the target core.

2.3.1.1 Basic-Block Time Estimation

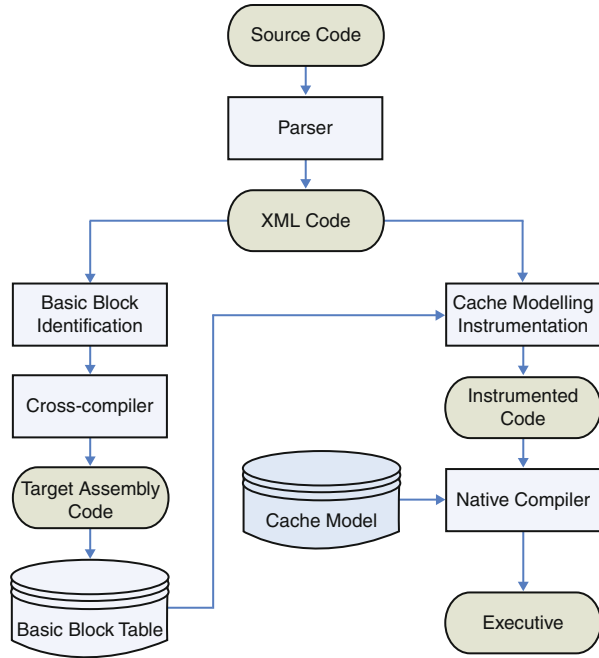
The estimation of the cost of each basic block in terms of time and power requires using cross-compiled code to take into account compiler optimizations. However, these compiler optimizations make it very complex to associate source code and generated binary code [1]. Complex reverse compilation techniques can be applied to correlate assembly blocks to source blocks [15]. However, this challenge is not always feasible due to advanced compiler optimizations.

In this work, a hybrid level technique for SW annotation is proposed: while basic block identification is performed at source level, characterization is obtained from assembly code. This strategy simplifies the characterization process and speeds up the analysis time.

To apply this solution to the DSE flow, a minimum effort from the designer is required. The cache model and its associated methods have been implemented as an independent library. Additionally, an automated instrumentation process performs all necessary annotations in source code to link software execution with the cache model. Figure 2.7 shows an overview of the cache estimation process, including basic block characterization.

Due to the rich syntax of source code, a C/C++ code parser has been developed in order to identify declarations, statements and expressions during the code annotation. The parser, based on a C/C++ YACC grammar, dumps the Abstract Syntax Tree (AST) to a file in XML format. This XML serves as input to the block identification and instrumentation processes. With the information included in the XML, the original C/C++ code is rebuilt adding time, power and cache information. Furthermore, this XML can be considered as an intermediate representation, independent from the source language. Any language could be estimated by simply creating a front-end for that language.

Fig. 2.7 Complete Annotation Process. The original code is analyzed with a C++ grammar. Then the code is built adding marks to identify the basic blocks and cross-compiled. The resulting code is analyzed to evaluate the metrics of each basic block, and the code is finally rebuilt adding this information



To identify basic blocks at source level, the proposed solution is to insert specific marks at the beginning and at the end of each basic block. This marked code is then cross-compiled, so the marks introduced are preserved in the target assembly code. The inserted marks looks like:

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

To prevent the compiler optimizations from deleting or moving such marks, they are declared volatile. Additionally, to keep the behavior of the original code, the asm instructions inserted consist simply of labels. This procedure guarantees that there is always a direct correlation between source and assembly blocks. However, as the effect of the compiler is constrained by the marks, the final binary code is not exactly the target one, so some errors are generated.

Compiler optimizations might affect both intra-block and inter-block behavior. Intra-block optimizations are considered in the characterization of the blocks from assembly code. This assembly code already includes both front-end and back-end optimizations. Inter-block optimizations are considered by delimiting the basic blocks at source level. Nevertheless, there are some compiler optimizations which cannot be accurately considered with this technique. Loop unrolling replicates the body of a loop statement in the assembly code, but from source point of view it is a unique block.

Nevertheless, in processors with instruction cache, loop unrolling is rarely employed, since the miss cost for each iteration significantly exceeds the jump cost.

Calls to inline functions from different points in the code have a similar problem, since the replicated code in the assembler file cannot be considered in the source instrumentation. If function inlining does not require replication of code (i.e. within a loop body), the technique works without limitations.

Summarizing, the error implied by this technique is minimal, and very adequate considering the accuracy required at the abstraction level.

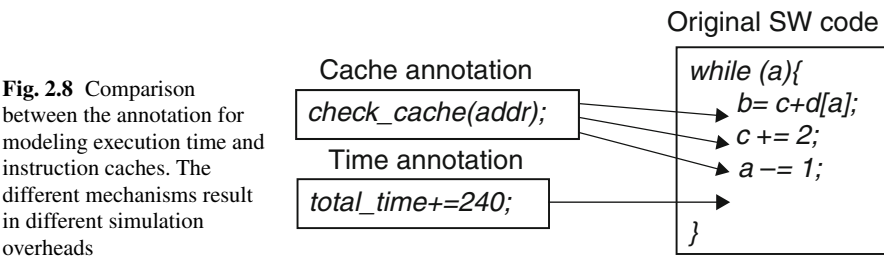
2.3.1.2 Cache Modeling

Cache memories have also a very important influence on SW performance. Thus, cache modeling is required to obtain accurate estimations. As it is well-known, cache memories consist of lines that are arranged in sets, depending on the degree of associativity. This value determines how many lines are grouped in each set. As an example, the ARM920T [2] instruction cache is 512-line size with a 64-degree of associativity.

When the processor requires the information placed in a certain address, it is required to search in all the possible locations for that address. The number of locations depends on the associativity degree. For high degrees, this task is a time-consuming process, which is really critical in the abstraction level required for efficient DSE. An example is shown in Fig. 2.8. While the overhead of the time annotation is only one additional code line, the cache access requires a function call on each line to check the instruction address, and a complex search within each call to check if the values are in cache or not. This drawback is then the main focus of the cache model to speed up simulation time.

The solution proposed to minimize the simulation overhead is based on three main improvements: search cache lines instead of single instructions, replace a list search by a static annotation and move the search from the cache model to the source code. This is shown in Fig. 2.9.

As instructions within a basic block are sequential, it is not required to search for each address in the cache. If one value (instruction or data) is in cache, all the instructions/data in the same line will be also there. Thus, the number of checks is extremely reduced. If a common cache line contains 8 values, cache checks are performed only 1/8th of the times the actual cache is checked. In practice, lines



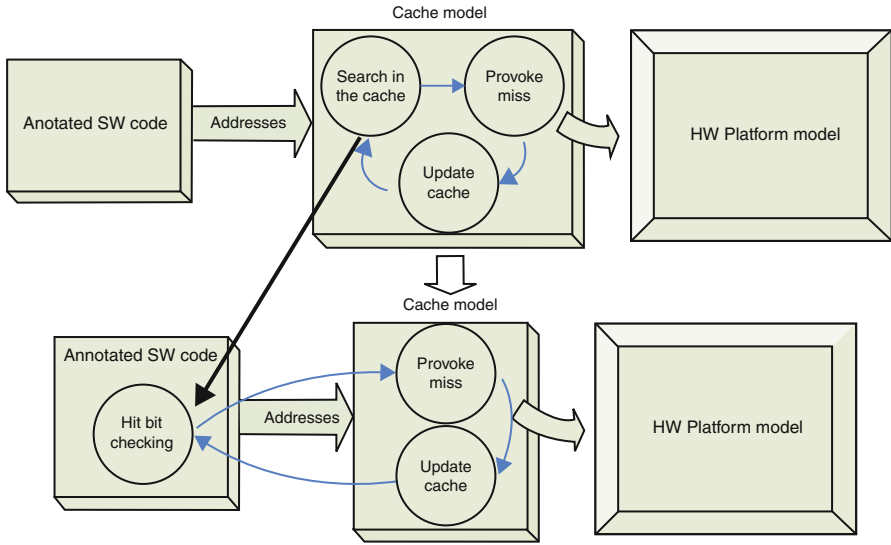


Fig. 2.9 The underlying concept applied to reduce of cache modeling overhead. The idea is to reduce the overhead by removing the function calls and searches done in the cache that are required when a new line is required. Instead, a bit checking is integrated in the software code annotation. Thus the cache model is only accessed at cache misses

containing instructions of different basic blocks are checked in all the blocks, so the reduction is slightly reduced.

The second improvement is even more important for complex caches. In order to avoid time-consuming “tag” searches, the dynamic search is replaced by a static “bit checking” solution. In the proposed model, a cache line is modeled as a structure which includes the target set at which it may be allocated and a flag which determines if a line is currently allocated in cache:

```
struct icache_line {
    char num_set;
    char hit;
}
```

The cache is modeled as a two-dimension array of pointers to cache line structures. The dimensions of the array depend on the physical characteristics of the cache to be considered (size, associativity and line size). All necessary methods for fetching or replacing data are also provided with the model. The caching mechanism consists of storing the addresses of the line structures in the array. An empty location in cache is modeled as NULL value.

The struct for each cache line is declared statically inside the SW code as part of the annotation, not in the cache model. This is done with the following line.

```
static icache_line line_124 = {0,0};
```

Thus, in order to check if the line is in cache or not it is enough to check the "hit" bit.

```
if (line_124.hit == 0)
    insert_line(&line_124);
```

This code is executed every time a basic block ends. The method used to model a cache miss, `insert_line()`, takes the address of the miss line and allocates it in cache. The allocation algorithm works as follows: the address of the line is used to find the target index. If there are free locations for this index, the address is simply recorded in the line pointers array and the hit flag of the line is set to true. If the index lines are full, the victim selection algorithm selects the victim line that must be expelled. The hit flag of the victim line is set to false through its pointer, and its location in cache is replaced by the new one. This way, the next time the victim line is required, the hit field will be false.

```
insert_line(icache_line *line) {
    icache_line *victim;
    victim = get_victim_line(line->set);
    if (victim != NULL) victim->hit = 0;
    line -> hit = 1;
    victim = line;
}
```

As a consequence of the entire process, tag searches are eliminated, and the cache is accessed only at misses, which is a small percentage of all the cache accesses. Thus, the cache modeling overhead is minimized. For data caches the solution applied is similar.

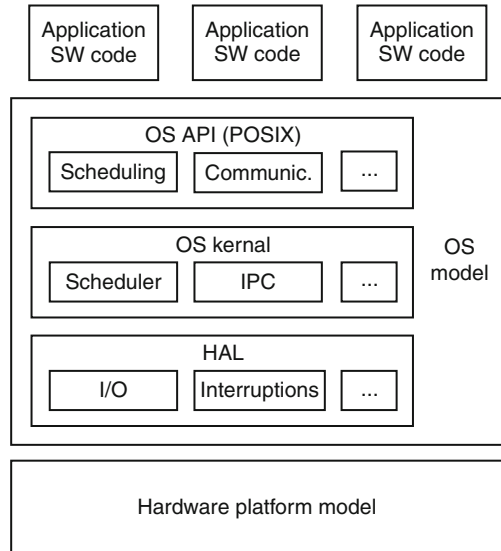
Note that no real caching of data is performed in our model during execution. As the SW code is executed natively, its execution does not depend on the behavior of the associated cache in the HW platform. Our cache modeling technique only models performance effects, not the actual HW behavior. This makes the technique faster than other cache simulation approaches.

2.3.2 RTOS Modeling

One of the most important points when transforming a high-level description into a real SW implementation is the inclusion of the OS. The SW code is gradually refined, mapping high-level facilities, such as communication and synchronization facilities, to the target OS facilities. Thus, the development infrastructure has to support the refined SW code, providing the target OS facilities inside the timed TLM model of the application SW.

The OS model included in M3-SCoPE is divided in a similar way to real OSs. An OS can be divided into several layers [59]. In the upper part, we can find the system call handlers that support the API. In the middle, the layers for internal OS management form what can be called the kernel core. At the lower level, there are

Fig. 2.10 Architecture of the OS model. The model is divided in three parts: the upper part contains the implementation of the services directly called by the application software. The middle part contains the model kernel, where all the internal OS services are implemented. Finally, the lower part is in charge of enabling the interconnection of the software with the rest of the simulation



the layers oriented to hide the HW platform details (Hardware Abstraction Layer, HAL) from the rest of the system, such as low-level functions to access the bus, the interruption management or the context switch infrastructure. Thus, the proposed OS model will provide support for all of these common OS sections, as can be seen in Fig. 2.10. In order to simplify the models, all the possible layers have been grouped into three layers, following the classification above. To model the execution time of the OS itself, a delay is associated to all system calls and some internal functionalities, such as context switches or interrupt handlers.

The kernel core is mainly in charge of process and thread execution management, memory management and task communication and synchronization. To model task scheduling characteristics, a new scheduler is placed on top of the original SystemC scheduler. The OS model scheduler controls task execution by maintaining only one unblocked thread per processor. This scheduler manages the priorities and policies as indicated in POSIX, integrating a two-level scheduler: process scheduling and thread scheduling. The scheduler is called when a task is blocked or unblocked as an effect of a system call, or when an interrupt resumes a task preempting the current running task.

This functionality has been implemented with use of the SystemC features. Each thread in the application SW is associated with a SystemC thread (`SC_THREAD`) running the corresponding code. In this way, the SystemC scheduler is in charge of managing the thread context switches. To model the OS thread scheduling the SystemC scheduler is hidden. All the SystemC threads are suspended with SystemC `wait` statements. When the scheduler selects one of these threads, a `notify` call is performed to awake the corresponding thread. When the thread is preempted or blocked, it returns to the `wait` statement and a new thread is scheduled. Thus,

the SystemC scheduler executes the only one runnable thread each time. This way scheduling is completely managed by the OS model scheduler through `wait` and `notify` statements. Note that SystemC `wait` statements are used for two purposes: thread blocking for scheduling purposes, as explained here, and for time annotations as explained in the previous section.

However, SW code refinement implies more than optimizing task scheduling characteristics. The proposed OS model provides a wide set of OS facilities, in order to allow the transformation of high-level descriptions into real executable SW codes. High-level features for communication, synchronization or time management have to be substituted by real OS features. In our proposal, a POSIX API has been implemented for communication (e.g. message queues or sockets), synchronization (e.g. semaphores, mutex or conditional variables), signals, timers, sleeps and many other functionalities. Thus, the designer can simulate the real SW code together with the rest of the system before moving to more accurate but time consuming simulation infrastructures, such as ISS-based simulations.

To support several independent OS models the OS model information has been encapsulated in a C++ class. Thus, modeling several OSs only require instantiating an OS class several times. Multiprocessor management is supported by maintaining as many running threads as processors. When a process is created or preempted, the presence of an idle processor is verified. When a processor is delivered, a new thread is scheduled to run on this processor. In this way, all processes are maintained active, and tasks are scheduled as soon as a processor becomes empty.

As all threads are SystemC threads, and the SystemC kernel activates the threads sequentially, there is no real parallelism during the simulation. Parallelism is only emulated. As a consequence, no additional mechanism is required to ensure safe concurrency inside the OS model, in contrast to real Symmetric Multi-Processing (SMP) OSs, where mechanisms such as *spinlocks* are critical to handle concurrency safety. Furthermore, the OS modeling technique is flexible enough to cover the most usual OS architectures. Each OS is associated to a memory space independently of the underlying processing architecture (bus or network). The complete MPSoC may contain as many memory spaces as needed. The HW platform model creation presented above easily enables that feature.

More complex is the modeling of memory space separation among processes of the same OS. As the SystemC simulation is a single host process, the management of memory spaces in the simulation requires additional mechanisms. Reusing names of functions and global variables among component codes or loading several instances of the same task is not possible without a memory space separation infrastructure. Nevertheless, this problem is not limited to processes of the same OS. As the entire simulation is a single host process, name duplication between SW process of different nodes, running in different OS, or between SW and HW components provokes the same error. A general solution has been implemented in M3-SCoPE. It will be presented hereafter when describing how all the system components are integrated in the SystemC model. Finally, the technology has also proven to be accurate enough to model dual General-Purpose/Real-Time Operating Systems, such as RTLinux. More information can be found in [52].

2.3.3 *Modeling of SW/HW Communication*

One of the most important features in native co-simulation is the interconnection between the native SW execution and the HW platform model. All accesses from the SW code to the HW platform must be detected and redirected, otherwise the SW will attempt to access the host peripherals, provoking multiple failures.

2.3.3.1 **HW/SW Communication Using Device Drivers**

The OS kernel and the API presented above are not enough to provide a complete OS model for software modeling and refinement. Although application code refinement is mostly supported, Hardware dependent Software (HdS) (such as drivers) is not supported. A more complete model, capable of managing interruption handlers and drivers, is required. The HdS is usually critical in embedded systems and its analysis at the early stages of embedded system development can provide large benefits during the rest of the project.

In order to perform correct HW/SW interface modeling it is required to have an OS model capable of managing the I/O accesses from the processor to the peripherals, and the interrupts the peripherals send to the processor. To manage the I/O, Linux-based driver support is provided. Linux standard functions for registering, accessing and managing drivers are integrated in the OS model, allowing the creation and use of Linux drivers.

Interrupts are generated from the HW platform model and received by the application SW modeling infrastructure, as explained in the previous section. When the timing effects have been established, the OS facilities for interrupt handling are called by the SW modeling infrastructure, calling the corresponding "wait" and "notify" statements. The standard or user-defined handlers associated with the interrupts are executed, performing the proper operations, and calling the corresponding drivers. The HW timer interrupt is of special interest. It is used to manage timers, sleeps, alarms and time-based scheduling policies, such as Round-Robin.

That way, HW/SW communication explicitly indicated in the code can be modeled. The I/O functions connect the native SW execution with the HW platform model. However, in embedded systems, HW/SW communication is not always so explicit. Peripherals can also be accessed by reading and writing directly the addresses of the peripheral registers. This communication is possible by using C/C++ pointers.

2.3.3.2 **Modeling of Direct I/O Accesses Through Pointers**

When the addresses of the peripherals are known by the SW developer and the OS allows that, peripheral registers can be directly accessed. For example, a device connected to a serial port can be accessed in the following way:

```
char *uart_addr = 0x80001004;
*uart_addr = A;
```

However, when the code is natively executed, this access will not work. Firstly, the host computer does not physically contain the required peripheral. As these pointer accesses cannot be easily detected statically, additional mechanisms are required to detect and redirect these accesses to the HW platform model at run-time.

Secondly, in native co-simulations the operating system prevents the application code to access specific HW addresses. When the SW code tries to access a fixed HW address, the memory management unit (MMU) will raise an exception as there is no physical address associated with the required virtual address. In an OS such as UNIX, this will result in a segmentation fault. The way to solve this problem is to force the operating system to create a page of virtual memory at the desired memory address. Thus, when the SW under simulation wants to read or write the HW values, values are correctly stored in the host memory.

To force the native operating system to create this memory page, the standard POSIX "mmap" function can be used. The "mmap()" function shall establish a mapping between a process address space and a file, shared memory object, or typed memory object. The format of the call is as follows:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

The required code is shown in Fig. 2.11. In that code, a file is created to store the information of the associated memory. It is important to note that the maximum size of the mapped memory is equivalent to the size of the associated file. As a consequence, if an empty file is used, no values can be read or written. The solution applied is to assign a size of "len" to the file before calling "mmap". To do so, the standard POSIX function "ftruncate" is used.

If the initial address does not correspond to the beginning of a memory page, special management is required. Memory pages always start from an aligned position. Thus the memory activated will start at the corresponding aligned address and will cover "len" bytes. To adjust the addresses, there are two possibilities. First the "offset" parameter can be used to indicate where exactly the mapped memory area must start. The second solution is to increase "len" with the offset of "addr". In the proposed code (Fig. 2.11) the second solution has been used. Furthermore, for debugging purposes, it is interesting to note that the values stored in the scratch-pad memory model can be shown by reading the associated file.

The solution is very effective when modeling scratch-pad memories in native co-simulations. It automatically allows executing SW code using fixed HW addresses with a negligible simulation overhead. As no cache misses or any other event is provoked internally by scratch-pad memories, only the ability of reading and writing values at these addresses is required. More specific details of internal scratch-pad operations are not handled at high level.

Fig. 2.11 Code for mapping the HW memory model

```
void initialize_periph (void *addr, int len){
    fd = open("tmp.txt", O_CREAT | O_RDWR, 0x01b6);
    ftruncate(fd, get_page_size());
    len += addr - page_aligned(addr);
    mmap(addr, len, PROT_READ | PROT_WRITE, MAP_FIXED, fd, 0);
}
```

Mapping HW peripheral accesses to a created memory page in native memory space results in another issue. These peripheral accesses from SW usually communicate particular event notification to the HW. Hence these HW peripherals must be notified about these HW access events. HW Peripherals are not designed to make polling of any variable. They react to read or write accesses from the system processors. Even in case of using a high-level model for the HW peripherals, it will also be based on such event-based communication. Whereas in technique presented in the previous section, the access is performed but the peripheral does not receive any event informing that a read/write operation has been performed in their registers.

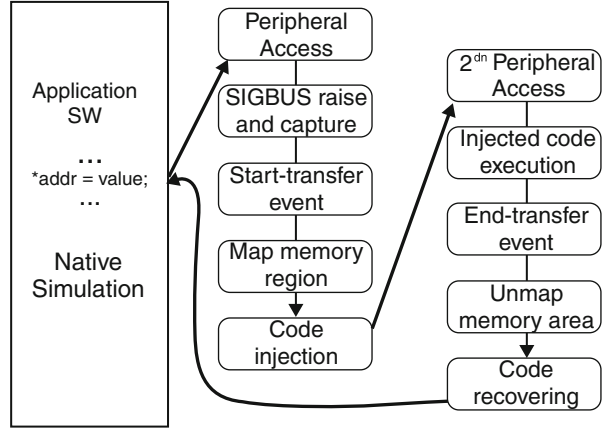
The only way to produce the event is not to map HW access to a created memory page and let the simulation crash. When the simulation is going to crash due to the segmentation fault, the error can be captured, solved and then the simulation can continue. The advantage of this approach is that the HW access is detected and the event can be sent to the peripheral model. When the SW tries to access an invalid memory address, the native operating system raises a SIGSEGV signal. This signal can be captured with an appropriate signal handler. This prevents the program to terminate, but this solution creates another challenge. During the signal handler, neither the access type (read/write) nor the value are known. Hence, the HW access cannot be performed at the signal handler.

To obtain the data, the memory remapping technique presented in the previous section is used. At the signal handler the memory mapping is activated and the code returns to repeat the pointer access. To perform a correct read access, the read transfer to the virtual platform is done first, updating the necessary memory address. Thus, when retrying the pointer read, the value obtained is correct. To perform a correct write access, the pointer access is performed first, and after that the new value is sent to the virtual platform.

Performing an "mmap" allows retrying the instruction, but once an access has been performed, the memory page is active and further accesses are not detected. To solve that, the memory page must be unmapped. However, when the code returns from the failed instruction, it continues normally without unpin the page. A possible solution is to create a parallel thread that wait for a certain time and then unmaps the page. However, this is an unsafe solution. There is no guarantee that there will be no more accesses before the unmap step, and even there is no guarantee that the unmap is done once the application SW code continue the native simulation.

To unmap the memory page properly, the SW code itself must do it. Just after the memory access is performed, the page must be unmapped. To do that, the original SW code must be modified. The solution applied is to dynamically inject code after the load/store assembler instruction that provoked the error. This injected code disables the memory page, re-establish original SW code and continues the execution. As a consequence, when the HW access is performed, the peripheral model is informed and the simulation status returns to the correct point to detect new accesses. Although the memory page is unmapped the data stored are not lost. The values are saved in the file associated to the memory page. The entire process is summarized in Fig. 2.12.

Fig. 2.12 Process for complete handling of HW accesses directly using pointers



Furthermore there is an another issue with the solution above. Detecting if a pointer access is a reading or a writing one is also complex. A possible solution is to disassemble the binary code of the instruction provoking the error, but this solution is not portable. Moreover, in x86 processors both reading and writing accesses are performed with "mov" instructions, so it is not easy to distinguish both. The portable solution is to force the system to raise different signals for read and write accesses. When executing an I/O pointer access, a SIGSEGV signal is obtained if the memory address has not been mapped. If the address has been mapped but the associated file has 0 size, a SIGBUS signal is raised. Thus at initialization the address is only activated for reading accesses with an empty file. Thus, a SIGSEGV raises at writing accesses (there is no writing permission) and a SIGBUS raises at reading accesses (there is no area in the associated file). This way one can differentiate between read and write HW peripheral accesses. More details on this scheme can be found in [50].

2.4 HW Platform Modeling

Once the software modeling has been defined, it needs to be connected to a HW platform for HW/SW co-simulation. For this purpose, an intermediate infrastructure is proposed. This infrastructure is placed in the OS model and in the target platform modeling facilities.

The SW modeling facilities presented above in Sect. 2.3 require a bus and, sometimes, a network interface to simulate a real communication of the SW with the rest of the platform. The bus model has to interact effectively and efficiently with the HW, thus a simple and flexible HW interface is integrated. This interface allows connection not only to generic HW components provided by the tool, but also to user-provided HW modules written in SystemC. For that purpose, three auxiliary HW models are needed to model a complete communication framework: a network

interface for node interconnection, a DMA for large data transfers and an abstract model of the physical memory to exchange data with the rest of the platform.

2.4.1 Bus Modeling

The techniques explained for SW estimation are always focused on fast performance estimation. TLM models are very efficient for this kind of applications. In order to take advantage of the simulation performance that TLM can provide, Programmer View with Timing (PVT) models are used instead of bus cycle-accurate models. These PVT models use behavioral descriptions with simplified communication mechanisms to perform fast simulations.

The bus model developed uses the TLM library for fast data transmission. In this way, communications are modeled by considering complete payload transfers instead of word-by-word transfers. The bus manages the simulation time of each transfer by considering parameters such as bus bandwidth, packet priority and size. However, the bus needs another component to complete its functionality – an arbiter. This module performs the transfer scheduling based on packet priority and delivery order. Therefore, the threads that try to access the bus and are not scheduled are blocked in a queue.

Another feature to take into account is the possibility to create bus hierarchy to cover more platform designs. The bus interface is able to connect not only buses with HW models, but also with other buses. This is implemented making use of the standard TLM interface to communicate different components. This interface can interconnect several modules transparently to the users.

2.4.2 HW Interfaces

M3-SCoPE provides several generic HW components, such as memory, bus, network or DMA. Additionally user-defined components can be integrated in the HW platform. In order to facilitate the integration of these user defined components, a base class is provided to be used as a wrapper. This wrapper base class is in charge of handling the TLM2 bus protocol management. The user needs to implement only the functionalities for read and write functions.

Additionally, the wrapper includes an automatic power estimation infrastructure. For each component it is possible to define parameters for the static power consumption and dynamic power consumption for read and write accesses. For read/write accesses it is possible to define two energy consumption values, one dependent on the number of accesses and the other one that considers the size of the data transfers. Applying all these parameters, M3-SCoPE automatically obtains an estimation of the component power consumption to be added to the consumption of the rest of system components. As all the HW component models (user-defined models and

components provided by M3-SCoPE) inherits this base class, the XML interface can automatically obtain power metrics of all the system components, and send that information to the DSE tool without manual intervention.

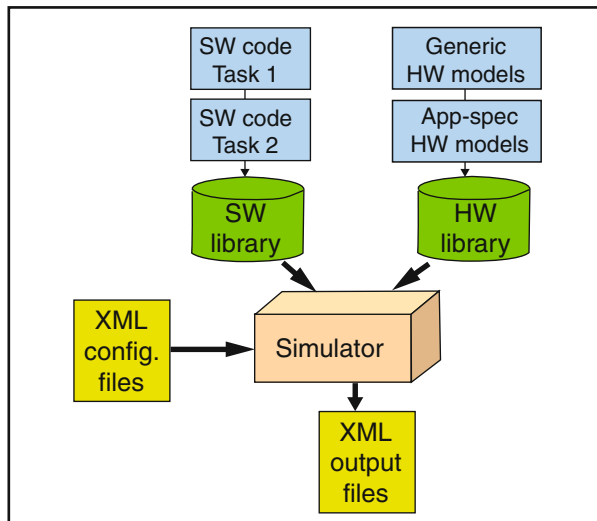
2.5 Automatic Generation of System Models

To simulate each one of the configurations selected by the DSE tool, different system models must be created. The model descriptions can be obtained by applying the values of the XML System Configuration file to the XML System Description. To generate the system model, the simulator dynamically creates instances of the required models and builds a platform model by interconnecting them as specified in the XML files. The complete process is shown in Fig. 2.13: first, the SW code of each process and the SystemC HW descriptions are compiled creating dynamic libraries. Then, the simulator loads these libraries and creates the system model. Finally, the model is simulated and performance results are obtained.

2.5.1 HW Platform Creation

To dynamically create the HW platform model, it is required to instantiate first the selected components, and then to create the interconnections between them. To do so, configurable models of the HW components are required. Response times, delays, area, mean power consumption, power for access, frequency, memory size,

Fig. 2.13 Flow proposed for automatically creating the design models. First the SW codes and the codes describing the functionality of the HW components are annotated and compiled creating binary libraries. At simulation time, the simulator receives the system description and uses the corresponding codes from that libraries to perform the simulation



IRQ or associated memory map addresses are some of the configuration possibilities. To instantiate a component in the system model, all these parameters must be set. Parameter values are obtained from the values indicated in the corresponding `HW_Instance` clause of the XML System Description file (Fig. 2.3). These parameters can be either defined as explicit values (“200MHz”, “500MB”) or identified as configurable values.

To set the parameters which are not specified in the `HW_Instance` clause, the simulator checks the `HW_Component` clause corresponding to the type of component instantiated (Fig. 2.3). Similar to the previous ones, these parameters can be fixed or configurable ones. Finally, if any of the parameters has not been fixed, the default values for this component model are applied.

Additionally, both the generic models provided by M3-SCoPE and the models provided by the users are provided as Dynamically-Linked Libraries (“.so” files). Using the host functions for dynamic loading, the components can be instantiated only requiring their names. After that, the instantiated HW components must be interconnected to create an executable system model. To simplify the interconnection work, TLM techniques have been used. TLM accurately describes the system communication architecture down to the level of individual read and write transactions. The use of transfers instead of signals reduces the complexity while automatically interconnecting the system components. To allow easy automatic interconnection of the system components, all component models have been created by using a generic template provided by the simulation engine. This template is oriented to ensure interface compatibility without limiting the component communication requirements. Ensuring that both ends of each interconnect have compatible interfaces, an automatic connection is possible.

Finally, to complete the HW platform generation, it is required to create the memory maps and to ensure correct interrupt delivering. Each time a component is connected to a bus, its associated memory area is integrated in the memory map, ensuring that it has not been used before. The solution is similar for network communication where network models require the node identifier in order to configure the internal routing protocols properly.

2.5.2 *SW Components Instantiation*

To create the SW infrastructure, OS models and SW tasks are finally added to the simulation as described in the XML files. OS models are provided by M3-SCoPE, while the application SW code must be provided by the user. OSs are mapped to a processor or group of processors (for SMP systems). SW tasks are associated to an OS, and thus mapped to a processor.

To integrate SW tasks in the simulation, SW code is annotated and compiled building a library which is added to the simulation. The annotated code provides the performance information required by an external DSE tool to perform the exploration.

Software tasks are defined in the XML System Description file indicating the name of the main function of the task. To load the main function, dynamic library management provided by OS is used, by calling the `dlopen` and `dlsym` function. Additionally, other parameters like the OS where it will run, priority, policy and the main function arguments can be defined. All these elements can be parametrized, so the DSE flow can explore the best configuration for the SW tasks.

2.5.3 Integration of Independent Component Codes

During the system model creation, multiple HW and SW components are integrated in a single SystemC simulation. When integrating different components in a system model, the models of each component can be incompatible for simultaneous integration in a single executable. If component models contain global variables and functions, name duplication problems can appear. This is especially important when integrating SW codes or algorithmic codes, where global variables are really common. If two components contain elements with the same names or a component is instantiated more than once, it will cause linking errors during compilation. Recoding of the system components can solve the problem, but it results in a very costly and error prone task.

To allow reusing names, two direct solutions have been considered in M3-SCoPE [53], considering “*direct*” as solutions without requiring recoding or any additional design effort. The first solution is to declare all functions and global variables as `static`. That way, the names are local to the object “.o” files. Then, when linking all the object files generated by applying this solution, the linkage is correct.

However, this solution presents four main limitations. These limitations will be overcome during next sections applying different approaches. The main problems are:

- First, it requires recoding, as it is required to add the `static` qualifier before the declarations. Thus, it cannot be called a “*direct*” solution.
- Second problem is that this solution is only applicable if each component is isolated in a single object file. If the system component description is divided in several object files, the functions and variables are not visible by the rest of the component object files. Considering that dividing functionality of SW processes and algorithms in C/C++ in several object files is usually considered a good programming practice, the solution is not recommended.
- Third problem appears when a component is instantiated several times. If a component using global variables is instantiated more than once, all instances will share global variables. This will generate interferences among the instances and will end in wrong system operation.
- Fourth problem is that this solution can only be applicable to internal functions and variables. For example, if we want to integrate several SW processes, all starting

with a `main` function, it is not possible to declare them `static`. Otherwise, the functions are hidden to the simulation kernel, and cannot be called.

Thus, a different solution avoiding these problems has been implemented. The first two problems can be solved by forcing separate visibility scopes for functions and variables on each component. The third problem requires modeling different memory spaces. Defining different scopes is not enough to share variables among different instances. Finally, the fourth problem can be overcome by providing an automatic way of defining and loading different entry points with the same names.

2.5.3.1 Achieving Separate Visibility Scopes

Visibility scopes can be defined by creating *namespaces* or *classes* in the code. However, these solutions are limited and require recoding. The solution recommended to implement the separate visibility scopes is to create shared libraries with all the application SW codes required for each SW process to be simulated. Shared libraries are libraries that are loaded by programs at run time. As the library is not copied into the executable file, the executable size is minimized.

Dynamic libraries can be created using the following commands. In the code, two source code files are compiled to create the corresponding object files, and then both object files are integrated to create a shared library file:

```
gcc -fPIC -c filea.c #create object file
gcc -fPIC -c fileb.c #create object file
gcc -shared -o libmylib.so filea.o fileb.o
```

Shared libraries can be used in a static or a dynamic way. When used statically, at link time, the linker searches through available libraries to find modules that resolve undefined external symbols. Then, the linker makes a list with the libraries required by the executable. When the program is loaded, start-up code finds those libraries and maps them into the program's address space before the program starts. The instruction used to create the executable is the following:

```
gcc main.c -o executable.x -L($Library_Path) -lmylib
```

When used dynamically, the library is unknown at link time. The library is selected and opened during the execution of the program. As a consequence, the linker command does not require a `-l` flag with the library names and paths.

To execute a program requiring shared libraries, the library must be in the library path. When using a library created specifically for an example, it is expected not to be in a standard library recognized directory. To add the library to the search list, the environment variable `LD_LIBRARY_PATH` must be updated properly.

To create separate visibility scopes, shared libraries can be used in both ways. The specific compiler options required for that purpose are related with the library creation, not with its use. By default, a function of a shared library always calls the symbol of the main program if it exists. To create separate scopes, the scope of all library elements (functions and variables) must be defined as local to the library.

Thus, when a function in the library calls another function or global variable, it uses the elements inside the library, instead of accessing the elements of the main program. Only when a name is not resolved inside the library, the main program or other shared libraries are accessed. To force the program to use the internal library elements, the library visibility has to be declared as `protected`. To do so, when creating the library, it is required to specify the compilation flag:

```
-fvisibility = protected
```

Using this flag, it is only required to indicate the entry points of each component description in the XML files to enter the visibility scopes. Once the starting function of the correct library is entered, all internal operations of the component description will be isolated within the library, and without having or producing external interferences.

2.5.3.2 Modeling Separate Memory Spaces

Providing different visibility scopes guarantees integrating independent component descriptions in a single executable. However, if a component is instantiated twice, global variables are shared. In real SW implementations, the memory space separation provided by the OS for each process solves the problem. Thus, the simulation framework must provide an equivalent solution.

Using different host processes to create the SystemC simulation is a possible solution to achieve that. However, inter-process communication mechanisms, and additional context switches required to perform the simulation makes the solution a bit complex to use.

A more easy solution to solve the problem of multiple instances is to create a copy of the shared library for each instance. It provides an automatic solution which is easy to implement. Storage requirements are increased as several copies of the files are created. Nevertheless, disk sizes of host computers are usually large enough.

Using dynamic links (`ln -s`) instead of copies is not a valid solution, as the linker uses the final node directly.

2.6 Simulation Results

As a simulator oriented to evaluate different configurations of software centric systems, the most important parameters required to demonstrate the validity of M3-SCoPE are estimation accuracy and simulation time. The estimation accuracy of the simulator covers two main aspects: the evaluation of the performance of the software code and the modeling of the effect the software execution has in the rest of the system. The first aspect can be checked by comparing the time estimated by the tool and by an ISS (e.g. Skyeye [56]). The second aspect can be evaluated verifying the amount of data injected by the software modeling in the system buses, which means checking the number of cache misses estimated by the tool and by the ISS.

	Instructions			Time		
	Skyyeye	M3SCoPE	Error %	Skyyeye	M3SCoPE	Speed-up
Bubble	5200180007	5200180007	0	4m45,53s	3,76s	x71
Factorial	2747041	2996535	9,1	1,28s	0,02s	x64
Hanoi	18481575	17695142	4,3	11,15s	0,11s	x100
Coder GSM	13466069	14066581	4,4	10,6s	0,09s	x117
H264	5601674012	5800347641	3,5	5m25,6s	3,92s	x80

Fig. 2.14 Analysis of accuracy and speed of the estimation technique. The results obtained with M3-SCoPE have been compared with SKyyeye [56], a no cycle-accurate ISS; the faster type of processor simulator

Example	Instruction Cache Misses			Data Cache Misses		
	M3-SCoPE	ISS (SkyEye)	Error (%)	M3-SCoPE	ISS (SkyEye)	Error (%)
Bubble	25	27	8	5204878	5199772	0.09
Hanoi	20	18	10	45	38	15.55
Factorial	8	7	12	500	375	25
GSM - 1 frame	3738	3663	2	660	670	1.49
GSM - 4 frames	15324	14814	3.4	2370	2452	3.46
GSM - 7 frames	26333	25730	2.3	4104	4235	3.19
GSM - 10 frames	37425	36624	2.2	5915	6026	1.84

Fig. 2.15 Accuracy of the cache models. The error is more representative in large examples, since in small examples, the number of errors is too small for comparisons

As can be shown (Fig. 2.14), the time estimation error of the native co-simulation technique is lower than 10%. Regarding the errors in the estimation of the number of misses, which has a high importance in bus and network modeling, the errors are in a similar range (Fig. 2.15).

In benchmarks with a small number of function calls, (i.e the bubble) the miss rate estimation errors is less than 1%, but when the number of calls to function is increased the miss rate grows too. Hanoi benchmark has a large percent of function calls in his instructions, his miss rate estimation error is 15% and in Factorial, which is an extreme code that only has function calls, the miss rate estimation error is 25%. This error is due to the big number of function calls, the difference in the compilers for different architectures produces that the number of registers saved in a function

were different. This error is important in specific cases, but in a larger codes it is minimized, as shown in the GSM coder example.

To demonstrate the modeling capabilities of all the presented features an the integration of the tool in a complete DSE process, an entire chapter is provided (Chap. 7), where M3-SCoPE is used to model a Power-Line Communication infrastructure.

2.7 Conclusions

SystemC has proven to be a powerful language for platform modeling. Nevertheless, its full exploitation for this purpose requires significant research activity in order to cover all the modeling requirements. The SCoPE framework has been described taking advantage of the language capabilities for system modeling and simulation. A methodology and associated library has been developed providing a complete OS functionality that can produce accurate timed simulations of the SW components. Power and timing estimations of the application SW running on the multi-processing platform in close interaction with the rest of the platform components can be obtained from the system simulation.

This chapter shows that native co-simulation can be applied successfully to this task. The performance analysis technology is fast enough to support efficiently the multiple runs required by DSE processes. To achieve this goal, M3-SCoPE has been developed with several improvements to SCoPE: cache modeling, direct I/O communication through pointers, memory space separation and dynamic platform creation from XML files.

This tool has been assessed on an industrial demonstrator, as shown in Chap. 7. The results demonstrate that the native co-simulation techniques proposed constitute a sufficiently accurate technique that is much faster than ISS-based simulation systems. This speed increase makes the technique optimal for fast system evaluation, covering the need to obtain effective metrics, which is necessary for efficient Design Space Exploration.

References

1. Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D.: Compilers: principles, techniques and tools. ED Pearson, (2007).
2. ARM, Advanced RISC Machines Holdings. Retrieved from www.arm.com
3. ARM Realview Development Suite (2005). Retrieved from www.arm.com/products/DevTools/RealViewDevSuite.html
4. Bailey, B., Martin, G., & Piziali, A.: ESL Design and Verification: A Prescription for Electronic System Level Methodology. Morgan Kaufmann (2007).
5. Balarin, F., Giusto, P., Jurecska, A., Passerone, C., Sentovich, E., Tabbara, B., Chiodo, M., Hsieh, H., Lavagno, L., Sangiovanni-Vincentelli, A. & Suzuki, K.: Hardware-Software Codesign of Embedded Systems: The POLIS Approach. Springer (1997).
6. Becker, M. Di Guglielmo, G., Fummi, F., Mueller, W., Pravadelli, G. and Xie, T.: RTOS-Aware Refinement for TLM2.0-based HW/SW Designs, Proc. of DATE, IEEE (2010).

7. Benini, L., Bertozzi, D., Bruni, D., Drago, N., Fummi, F., & Poncino, M.: SystemC cosimulation and emulation of multiprocessor SoC design, *Computer*, V.36, N.4, IEEE (2003).
8. Benini, L., Bogliolo, A., Menichelli, F. & Oliveri, M.: MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing*, V.41, N.2: Springer.
9. Bouchhima, A. , Gerin, P. & F. Petrot: Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation. *Proc. of ASP-DAC*. IEEE (2009).
10. Brandolese, C., Fornaciari, W. & Sciuto, D.: A Multi-level Strategy for Software Power Estimation, *Proc of ISSS*, IEEE (2000).
11. Brandolese, C., Fornaciari, W., Salice, F., & Sciuto, D.: Source-level execution time estimation of C programs. *Proc. of CoDes*, IEEE (2001).
12. Brandolese, C. & Fornaciari: Measurement, Analysis and Modeling of RTOS System Calls Timing, 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, 2008.
13. Cai, L. and D. D. Gajski: Transaction Level Modeling: An Overview. In *Proc. of CODES + ISSS03* (2003).
14. Castillo, J., Posadas, H., Villar, E., & Martinez M.: "Fast Instruction Cache Modeling for Approximate Timed HW/SW Co-Simulation". *Proc. of GSLVLSI*, ACM (2010).
15. Cifuentes, C.: "Reverse Compilation Techniques". PhD thesis, Queensland University of Technology (1994).
16. CoWare: Task Modeling and Virtual Processing Unit User's Guide (2009).
17. CoFluent: CoFluent Studio: System-Level Modeling and Simulation Environment (2009).
18. Gerin, P., Hamayun, M. and Petrot, F.: Native MPSoC Co-Simulation Environment for Software Performance Estimation. *Proc. CODES+ISSS*. ACM (2009).
19. Gerstlauer, A., Yu, H. & Gajski, D. D.: RTOS Modeling for System Level Design, *Proc. of DATE*, IEEE (2003).
20. Gligor, M., Fournel, N. & Petrot, F: Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. *Proc of Codes+ISSS*, 2009.
21. Hassan, M. A., Yoshinori, S. K., Takeuchi, Y. & Imai, M. RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC. *Proc of DATE*, IEEE (2005).
22. Hassan, M.A., Sakanushi, K., Takeuchi, Y. & Imai, M.: Enabling RTOS Simulation Modeling in a System Level Design Language. *Proc. of ASP-DAC*, IEEE (2005).
23. He, Z., Mok, A. & Peng, C.: Timed RTOS modeling for embedded System Design, *Proc. of RTAS*, IEEE (2005).
24. Hergenhan, A., Rosenstiel, W.: Static Timing Analysis of Embedded Software on Advanced Processor Architectures. *Proc. of DATE*, IEEE (2000).
25. Hwang, Y. , Abdi,S. & Gajski D.: Cycle-approximate Retargetable Performance Estimation at the Transaction Level, *Proc. of DATE*, 2008.
26. Imperas, <http://www.ovpworld.org>.
27. InterDesign Technologies, FastVeri (SystemC-based High-Speed Simulator) Product Overview, <http://www.interdesigntech.co.jp/english/>.
28. IP-XACT. The P1685 IP-XACT IP Metadata Standard. *Design & Test of Computers*, IEEE, Volume: 23, Issue: 4, On page(s): 316- 317, (2006).
29. ITRS - Design. International Technology Roadmap for Semiconductors. Retrieved from <http://www.itrs.net/Links/2007ITRS/Home2007.htm> (2007).
30. Jerraya, A. & Wolf, W. (Ed.). (2005). *Multi-Processor Systems on Chip*. Morgan Kaufmann.
31. Kempf, T., Karur, K., Wallentowitz, S. & Meyr, H.: A SW Performance Estimation Framework for Early SL Design using Fine-Grained Instrumentation, *Prof. of DATE*, IEEE (2006).
32. Klingauf, W., Gunzel, R., Bringmann, O., Parfuntseu, P. & Burton, M.: GreenBus - a generic interconnect fabric for transaction level modelling, *Proc. of DAC*. ACM (2006).
33. Lahiri, K. A. Raghunathan, and S. Dey: Efficient exploration of the SoC communication architecture design space. In *Proc. ICCAD'00* (2000).
34. Laurent, J., Senn, E., Julien, N. & Martin, E.: Power Consumption Estimation of a C-algorithm: A New Perspective for Software Design, *Proc. of LCR*, ACM (2002).

35. Lyonard, D., Yoo, S., Baghdadi, A. and A. A. Jerraya: Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. DAC'01 (2001).
36. Madsen, J., Virk, K., and Gonzalez, M.: A SystemC abstract real-time operating system model for multi-processing systems-on-chip. In A. Jerraya and W. Wolf. Multiprocessor Systems-on-Chip. Morgan Kaufmann (2005).
37. Magillem 4.0, <http://www.magillem.org>
38. Milligan, M.: The ESL Ecosystem - Ready for Deployment. Retrieved from http://www.esl-now.com/pdfs/eco_presentation.pdf (2005).
39. Moussa, I., Grellier, T. & Nguyen, G.: Exploring SW performance using SoC transaction-level modeling. Proc. of DATE. IEEE (2003).
40. Murray, B.: Virtual platforms - a reality check, part 2. SCD Source. <http://www.scdsource.com/article.php?id=66> (2007).
41. NAUET Design Assembler, <http://www.mataitech.com>
42. Ortega, R. B and G. Borriello: Communication synthesis for distributed embedded systems. In Proc. of ICCAD'98 (1998).
43. Pasricha, S., Dutt, N. & Ben-Romdhane, M.: Fast exploration of bus-based on-chip communication architectures, Proc. of CODES/ISSS. IEEE (2004).
44. Petrot, P.: Annotation within dynamic binary translation for fast and accurate system simulation. 10th International Forum on Embedded MPSoC and Multicore (2010).
45. Popovici, K., Guerin, X., Rousseau, F., Paolucci, and A.A. Jerraya: Platform-based Software Design Flow for Heterogeneous MPSoC. ACM Transactions on Embedded Computing Systems (2008).
46. Posadas, H., Herrera, F., Sanchez, P., Villar, E., & Blasco, F: System-Level Performance Analysis in SystemC. Proc. of DATE. IEEE (2004).
47. Posadas, H., Adamez, J., Sanchez, P., Villar, E., & Blasco, P.: POSIX modeling in SystemC. Proc. of ASP-DAC'06. IEEE (2006).
48. Posadas, H., Quijano, D., Villar, E. & Martinez M.: TLM interrupt modelling for HW/SW co-simulation in SystemC. Conference on Design of Circuits and Integrated Systems, DCIS'07 (2007).
49. Posadas, H., De Miguel, G. & Villar, E.: "Automatic generation of modifiable platform models in SystemC for Automatic System Architecture Exploration". Proc. of DCIS'09 (2009).
50. Posadas, H. & Villar, E.: Automatic HW/SW interface modeling for scratch-pad & memory mapped HW components in native source-code co-simulation. In the book, Rettberg, A. et al (Eds.): Analysis, Architectures and Modelling of Embedded Systems, Springer (2009).
51. Posadas, H., Castillo, J., Quijano, D., Villar, E., Ragot, D. & Martinez M.: SystemC Platform Modeling for Behavioral Simulation and Performance Estimation of Embedded Systems. In the book L. Gomes & J. M. Fernandes (Eds.): Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation, IGI Global (2009).
52. Posadas, H., Villar, E., Ragot, D. & Martinez M.: Early Modeling of Linux-based RTOS Platforms in a SystemC Time-Approximate Co-Simulation Environment. ISORC, IEEE (2010).
53. Posadas, H. & Villar, E.: Modeling Separate Memory Spaces in Native Co-simulation with SystemC for Design Space Exploration. Proc. of ARCS, 2PARMA workshop (2010).
54. Qemu, <http://wiki.qemu.org>.
55. Schirner, G. & Dömer, R.: Result Oriented Modeling, a Novel Technique for Fast and Accurate TLM, Transactions on Computer-Aided Design of Integrated Circuits, V.26, IEEE (2007).
56. SkyEye User Manual. Retrieved from <http://www.skyeye.org> (2005).
57. Schnerr, J., Bringmann, O., Viehl, A., Rosenstiel, W.: High-Performance Timing Simulation of Embedded Software. Proc. of DAC, ACM (2008).
58. SystemC, IEEE 1666 -2005 Standard LRM, <http://www.systemc.org/downloads/lrm>.
59. Tanenbaum, A.: Modern Operating Systems, 2 ED: Prentice Hall (2001).
60. Viaud, E., Pecheux, F. & Greiner, A.: An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles, DATE, IEEE (2006).

61. Wieferink, A., Leupers, R., Ascheid, G., H. Meyer, T. Michiels, A. Nohl and T. Kogel: Retargetable generation of TLM bus interfaces for MPSoC platforms. CODES+ISSS'05 (2005).
62. Yi, Y., Kim, D. & Ha, S.: Fast and time-accurate cosimulation with OS scheduler modeling. Design Automation of Embedded Systems, V.8, N.2-3: Springer (2003).
63. Yoo, S., Nicolescu, G., Gauthier L.G. & Jerraya, A.A.: Automatic generation of fast timed simulation models for operating systems in SoC design, Proc. of DATE, IEEE (2002).

Multi-objective Design Space Exploration of
Multiprocessor SoC Architectures

The MULTICUBE Approach

Silvano, C.; Fornaciari, W.; Villar, E. (Eds.)

2011, XXIV, 209 p., Hardcover

ISBN: 978-1-4419-8836-2