

# On MPSoC Software Execution at the Transaction Level

**Frédéric Pétrot**

Grenoble Institute of Technology

**Nicolas Fournel and Patrice Gerin**

Kalray

**Marius Gligor, Mian-Muhammad Hamayun,  
and Hao Shen**

Grenoble Institute of Technology

*Editor's note:*

This article presents a wide variety of techniques for realizing transaction-level models of the increasingly large-scale multiprocessor systems on chip. It describes how such models of hardware allow subsequent software integration and system performance evaluation.

—Zeljko Zilic, McGill University

■ **“TRANSACTION-LEVEL MODELING (TLM)** has proven revolutionary value in bringing software and hardware teams together using [a] unique reference model.”<sup>1</sup> This quote from Ghenassia and Clouard in 2005 summarizes the goals of efforts begun in the 1990s to raise SoC modeling abstractions above cycle accuracy. The abstractions were to provide simulation platforms suited to early verification of software running on complex SoC architectures before their availability as actual hardware platforms. This initial goal of performing early software integration is more relevant today than ever. Indeed, the continual increase in feature size has led the semiconductor industry to favor flexibility and redundancy through the massive integration of programmable components. Consequently, increasingly more code is being developed for all layers of the software stack.

Because of the amount of hardware and software being integrated and the level of detail required to check that the entire software stack runs correctly on the hardware, ensuring that the system performs correctly is difficult, if not totally unrealistic. The only viable, though not satisfactory, solution today is to simulate the entire system (operating system,

application, and so forth) to uncover as many software and hardware code problems as possible.

Transaction-level hardware-software simulation of multiprocessor SoCs (MPSoCs) requires handling software execution in some way. Thus, different strategies have been developed to pro-

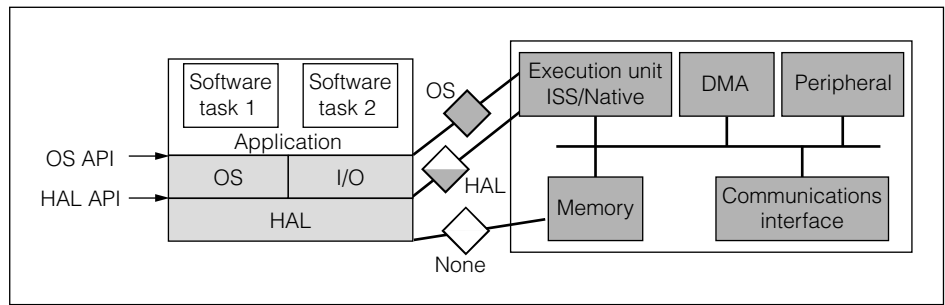
vide suitable environments for executing software code atop a virtual platform. These strategies are now quite mature, and they fall into a continuum between *raw instruction interpretation* of the cross-compiled code and *native execution* of the code on the host. Moreover, these strategies vary in their capabilities of representing the actual hardware, their simulation speed, and their simulation accuracy. In this article, we discuss these strategies, as we review the challenges involved in introducing software and hardware-software simulation in MPSoCs at the transaction level.

## Handling software in MPSoC simulations

The software that is finally embedded in an actual MPSoC platform is the entire set of software layers, from the application to the lowest-level assembly code (e.g., task switching and device configuration). To execute every single software instruction, the most straightforward strategy is *instruction-accurate interpretation*, which was introduced in the early 1960s.<sup>2</sup> By relying on the software APIs used in the application (e.g., standardized operating-system calls), performance gains are possible. Thus, it is feasible to natively execute the application code in a specific

hardware platform model and to totally avoid interpretation.

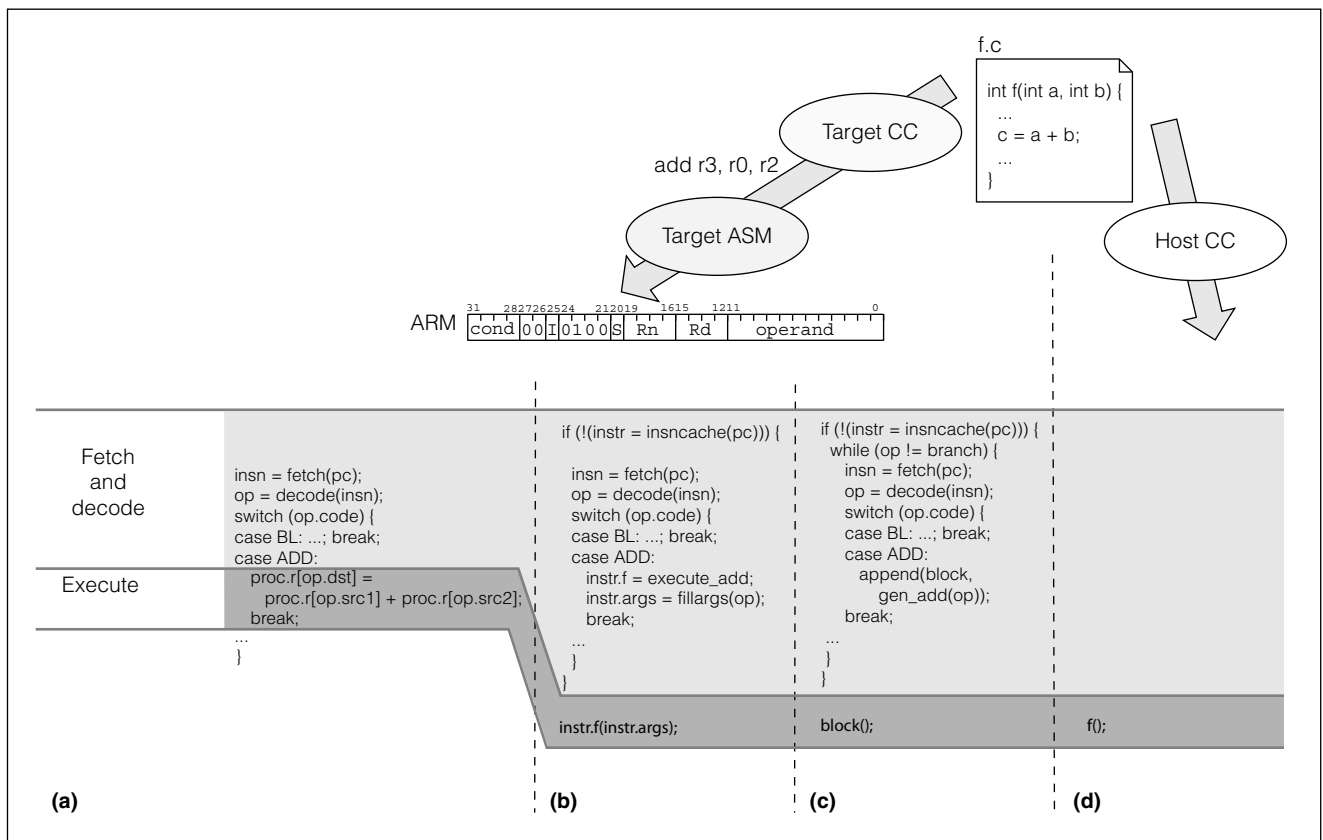
Figure 1 illustrates the integration of software within a TLM hardware architecture. The left side of the figure represents an application split in software layers, and the right side represents the TLM hardware on which the software runs. Depending on the software execution technique, the simulated software includes more or fewer layers. For the interpretation of cross-compiled code, all layers are included and loaded into program memory, so no abstraction is needed. For native approaches, the execution unit abstracts either the operating-system layer or the hardware abstraction layer (HAL),<sup>3</sup> by relying on their respective APIs. A wide variety of methods can be used for this software execution task (see Figure 2).



**Figure 1. Software integration. (DMA: direct memory access; HAL: hardware abstraction level; ISS: instruction-set simulator; None: no abstraction is needed; OS: operating-system level.)**

### Interpretive methods

Software interpretation is the process of transforming instructions from the target processor into instructions of the host processor. It always requires decoding an instruction and executing its behavior. But the time at which decoding is performed and the manner in which execution is handled can greatly impact performance. We now briefly review



**Figure 2. Software simulation techniques applied to the ARM instruction-set architecture (ISA): instruction-accurate interpretation (a), interpretive predecoding (b), dynamic binary translation (c), and native code execution (d). (ASM: assembly; CC: C language compiler.)**

the three techniques that are commonly used for interpretation. We assume the target code is in a memory array.

#### Instruction-accurate interpretation

The *decode-and-dispatch* method is the most basic, intuitive approach. The instruction's binary code is fetched from the array at an address function of the program counter and decoded according to the instruction format. Typically, the instruction is executed using one large switch statement on the opcode (*switch threading*, as in Figure 2a), or the opcode is used to index an array of function pointers (*call threading*).<sup>4</sup> The target processor is represented by a data structure containing its memorization resources, and the execution of the behavior reads and updates these resources. The instruction is decoded and its behavior executed every time the program counter points to it, leading to a nonnegligible overhead.

We can improve this approach by using *predecoding*, which builds a data structure that contains the instruction information in a host-machine-friendly way, in hopes that the instruction will be often reused and so amortize the cost of building this representation. Specifically, simulating the access to a register requires shifting and performing a logical AND operation on the instruction binary representation; and then accessing the array that represents the register file, using as the index the value that has been computed. Predecoding does something very similar the first time the program counter is reached, but instead of accessing the array, it saves the array slot's address into a pointer (Figure 2b). So, the next time the same program counter is reached, a simple dereferencing of the pointer allows access to the register. The predecoded instructions can be saved in a specific array, or simply in the processor's instruction cache model, if there is one. In this case, the predecoding occurs on a cache miss, and the data structure is always correct on a cache hit. Behavior execution still takes place one instruction at a time.

The pros and cons of instruction-accurate interpretation are as follows. It is easy to implement, can be made accurate, and supports simulation of the entire software layers. It also supports self-generating or modifying code (necessary, for example, to dynamically link shared libraries), provided the cache and memory models support coherency. However, it's extremely slow and not suited for extensive software-based validation.

#### Binary translation

Because nothing more can really be gained at the instruction level, the next step is to work at a coarser granularity. Ideally, implementing a static translation of the entire target program binary to the host binary would be the most efficient solution. However, this *static binary translation* (SBT) is unfortunately not typically a solution, because there are many computations and branches that use indirect accesses to the memory in which the values are known only at runtime (e.g., an array of dynamically assigned function pointers). Therefore, an intermediate granularity level between a single instruction and an entire program is a sequence of nonbranch instructions that end in a branch.

This level is appropriate because if we execute the first instruction, we are guaranteed to execute all instructions of the sequence. Technically, it is very close to the definition of a *basic block* used in compilation, but in a basic block no other instruction except the first one can be a branch target. So, in this case, this translation unit is called a *translation block*. This is precisely the choice that is made in *dynamic binary translation* (DBT),<sup>5</sup> which translates the target binary into the host binary on the basis of each particular translation block (see Figure 2c). As in predecoding, the translation blocks are cached and reused when the program counter reaches a block's entry address, thus amortizing the translation overhead.

The pros and cons of dynamic binary translation are as follows. It is very efficient and is the technology of choice for virtualization. In such cases, speed is the ultimate goal, so there is no notion of accuracy compared to the actual processor. However, DBT is conceptually complex and is challenging to implement.

#### Semihosting execution methods

Independent of pushing DBT to its performance limits, gaining yet more performance requires raising the software's abstraction level. Indeed, software is usually organized as functions that call other lower-level functions. This organization is well-accepted, and even often standardized (either in reality or de facto).

In a typical Unix application, the application calls the standard library routines, which are themselves built atop the Unix system calls. So, if the operating system is well-designed, it relies on a

HAL to access the hardware. *Semihosting execution*, a strategy that had been democratized by the end of the 1980s, and initially intended to give access to the host resources, consists of interpreting all the target instructions as usual, except the system calls. In semihosting execution, the simulator (rather than jumping to the trap handler) checks the system call number (usually given in a machine register), transforms its arguments in the target so that they're compatible to the host, and performs the call to the host-machine-equivalent system call. The result of the call is then transformed into target-compatible information. This strategy can be easily employed for system calls. Moreover, at a slightly higher complexity cost, it can also be applied for the standard libraries or even at the HAL.

The pros and cons of semihosting execution are as follows. Owing to the cost of fully interpreting a complex function or a system call, this approach is efficient and leads to high gains in simulation speed. But its main drawback is in performance evaluation, in which the time spent on host functions and its side effects (communication, cache state, etc.) can, at best, be only roughly guessed.

The semihosting technique can be used by any interpretive method to accelerate the execution of the functions belonging to standard libraries. As such, it is a technique that deserves to be known, but it does not provide a differentiating advantage to any of the interpretive methods. Therefore, we will not specifically address it in the rest of this article.

### Native execution methods

The most efficient way of executing software is to compile it directly on the host machine (see Figure 2d). This poses some difficult problems because of the potential use of hard-wired addresses, and requires that the entire code be in a high-level language (i.e., not assembly code).

The pros and cons of native execution are as follows. Simulation is fast and well-suited for cases in which software is well-structured and in an early phase. For performance evaluation, many researchers have proposed ways to insert timing annotations into

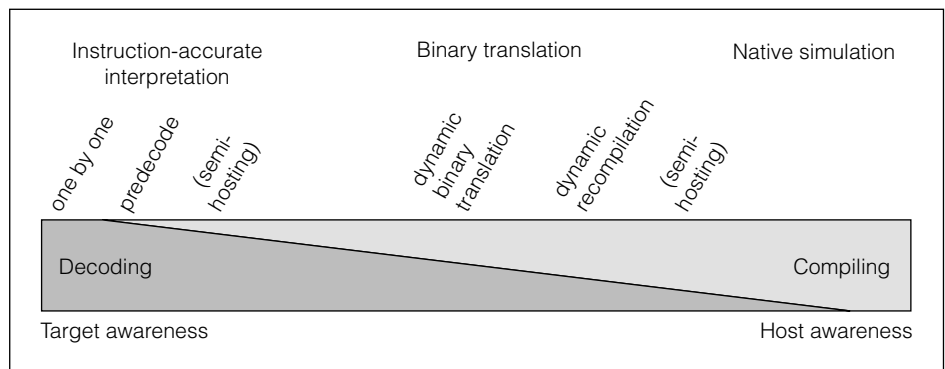
the source code to estimate execution time, but timing annotations cannot deal with compiler optimizations, because they change the code's structure. The impact of execution on caches and communication is also unpredictable.

### Comparison of simulation strategies

Figure 3 shows how software can be run in simulators. Even though there is not an infinite number of methods, the simulation strategies can be placed along an execution continuum, from *totally independent of the host but totally dependent on the target* on one end to *totally independent of the target but totally dependent on the host* on the other end.

Intuitively, we can expect the simulation speed to increase from left to right, and the simulation accuracy to decrease from left to right. Also, using semihosting calls and native execution makes it possible to abstract the knowledge of the underlying hardware so that only the software stack's upper layers (typically, the application or higher operating-system parts) need be available. This lets us simulate hardware-software systems in which both hardware and software can be in some preliminary stages, and thus is quite useful at the transaction level.

Table 1 compares the four software execution methods in terms of simulation speed, simulation accuracy, and development time, and makes it possible to execute the entire cross-compiled software. Simulation accuracy is given in terms of instruction count and wall clock time. The development time is the time taken to build an execution unit of the corresponding type from scratch (labeled "First" in the table), or starting from an existing infrastructure



**Figure 3. Simulation strategies along an execution continuum based on target awareness and host awareness.**

**Table 1. Rough characteristics of the software simulation approaches.**

Simulation approach	Simulation speed	Simulation accuracy		Development time		Full software execution
		Instruction count	Time	First	Reuse	
Instruction-accurate interpretation	Very slow	Very accurate	Very accurate	Simple	Easy	Yes
Interpretive predecoding	Slow	Very accurate	Very accurate	Complex	Easy	Yes
Dynamic binary translation	Fast	Not accurate	Not accurate	Very complex	Hard	Yes
Native code execution	Very fast	Not accurate	Not accurate	Very complex	Very easy	No

(labeled “Reuse” in the table). “Full software execution” refers to the capability of executing any code, including self-modifying code.

### Integration into a TLM environment

Here, we focus on integrating the three software execution strategies into a TLM environment that supports the notion of time and targets multiprocessor platforms, and we present existing solutions to enhance simulation accuracy. (Predecoding is an optimization of instruction-accurate interpretation, so we do not address that separately here.) We assume the implementation uses SystemC.

#### Instruction-accurate interpretation

Instruction-accurate ISS (instruction-set simulator) technology is popular in TLM environments. This is the simplest technology to carry out, but it suffers from a low simulation speed and, even if optimized with call threading or predecoding, remains a bottleneck for MPSoC simulation.

The integration of these simulators into TLM environments is simple for two reasons. First, all accesses outside the processor model (memories or device registers) are explicit and exact in the model and can easily be forwarded to cache, interconnect, and peripheral models. Second, all accesses use target addresses. Thus, the complete simulation platform handles the same address space, and the target-to-host address translation is a simple mask inside the models to index host-allocated data.

Making instruction-accurate interpretation more accurate in terms of performance estimations is relatively straightforward; it involves adding details of the underlying processor architecture in the simulation model. Doing so can reflect the behavior of the processor’s internal pipeline by implementing, for example, delay slots or register lock-down mechanisms in the ISS source code.

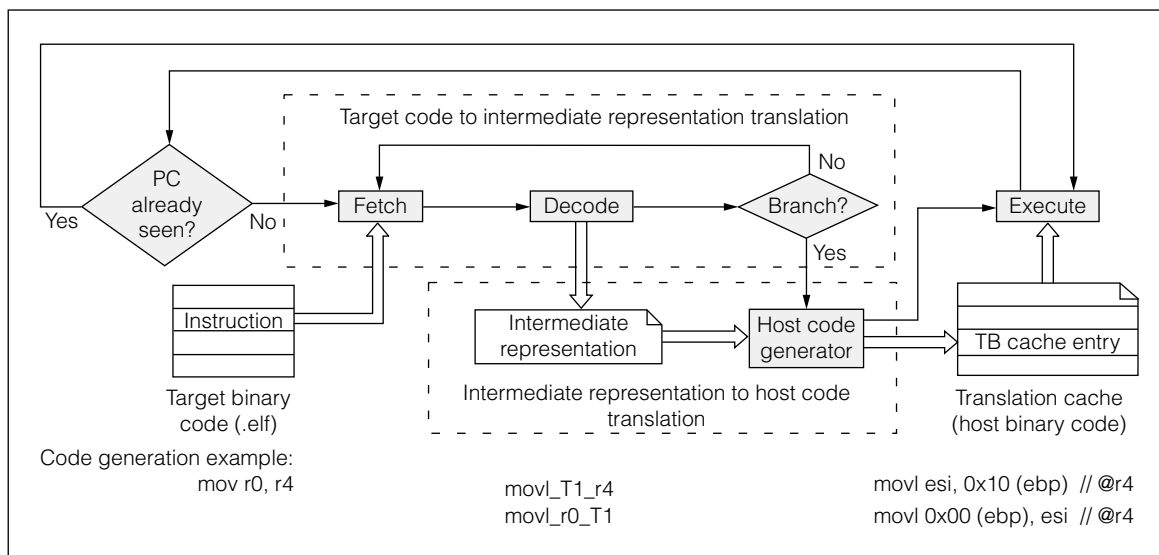
Timing annotation is not difficult with this technology, but each added architectural detail is an extra burden on the already-low simulation speed.

#### Dynamic binary translation

As of today, few efforts have been made to use DBT in a TLM simulation environment. Most of these efforts have used QEMU,<sup>6</sup> a processor emulator designed for virtualization that uses a portable dynamic translator to provide DBT. Two main strategies have been proposed to integrate QEMU and SystemC-TLM. In the first strategy, a bridge between the two simulation engines is built. In the second strategy, QEMU becomes a common SystemC module via a wrapper. Before discussing these integrations strategies further and the timing precision they can provide, let us first consider the DBT mechanisms, using QEMU.

Figure 4 shows the complete emulation process using DBT. As Figure 2c shows, this process separates execution from translation. Indeed, the granularity of each of these two tasks is the translation block (TB), a sequence of nonbranch target instructions. The translation task involves transforming code suited for one machine (a TB) into code having the same behavior but suited for another machine (a *translation code block*, or TCB). The execution task executes each TCB. The TCBs are stored in a cache to allow reusing translations, thus amortizing their cost over multiple executions. The simulation engine’s main loop is simple: it checks the existence of a TCB for the current program counter in the cache, calls the translation task if no TCB exists, then executes the TCB and loops back to the first step.

The translation process in turn comprises two steps: translation and code generation. The first step transforms the target instruction sequence into a sequence of micro-operations of the intermediate representation. This transformation is done on an



**Figure 4. Dynamic-binary-translation simulation model. (PC: program counter; TB: translation block.)**

instruction basis, meaning instructions are fetched one by one and decoded to produce the corresponding sequence of micro-operations. If the instruction is a branch or jump, this step ends, and the second step, the host code generation, begins. This step generates host instructions for all micro-operations produced in the previous step.

At this stage, each target instruction can be translated in several micro-operations, which can in turn generate multiple host instructions. There is no correlation between the original target instruction flow and the executed host instructions other than pure functional correctness. Figure 4 gives a simple example of binary translation of an ARM instruction into x86 instructions, using the QEMU intermediate representation.

A bridging-integration technique keeps the simulation engine of QEMU intact and adds the capability for integrating SystemC models to it.<sup>7</sup> With this technique, a large part of the single-processor system is simulated using QEMU, and only a small part of the platform is modeled in SystemC. However, the difficulty is in controlling the evolution of time for the two simulation engines. The QEMU part of the system advances at its pace, and, through event sharing, awakes the SystemC model when needed. The timed events produced by the SystemC model go into the QEMU event queue. QEMU then runs until either the first SystemC event occurs, or a transaction targeting the SystemC model awakes the SystemC model again.

The simulation of MPSoCs with this integration scheme relies only on the ability of QEMU to emulate multiprocessor systems. QEMU performs this emulation via a static round-robin scheduling scheme in which each processor executes one or several translation blocks in turn. This totally breaks the software execution's concurrency and cannot guarantee the order of accesses to the SystemC hardware models. One way to solve this issue is to reduce the part of the system modeled by QEMU so that only one processor is modeled by the QEMU process, and then use the SystemC engine to manage concurrency. This integration is based on the bridging strategy and relies on interprocess channels to have both engines communicate.

The second integration technique uses QEMU as a standard SystemC module. In this case, the main infinite loop of the QEMU engine is broken to allow start or pause of the software emulation. The engine is then wrapped in a module that guarantees the synchronization of the two engines by calling the SystemC `wait()` function, so that the SystemC simulator kernel can have other hardware models progress in time. The last proposition is based on a two-level wrapping scheme that permits translation cache sharing between different instances of the QEMU ISS. Using this sharing scheme reduces the cost of translation even more and improves simulation speed.

Some of these approaches also target timing-accurate estimations. In such a case, nonfunctional micro-operations are introduced at translation time

in the intermediate representation to enrich the generated code performing the software's functional execution with instructions that, for example, count the necessary cycles for target instructions execution or call target cache models.<sup>8</sup> This technique even allows the modeling of dynamic voltage and frequency scaling (DVFS) mechanisms via a simple conversion of cycles to time based on the current frequency.

The inclusion of these annotations, provided they have the right accuracy level, lets binary-translation-based ISSs reach a precision similar to instruction-accurate ISSs.

#### Native software simulation

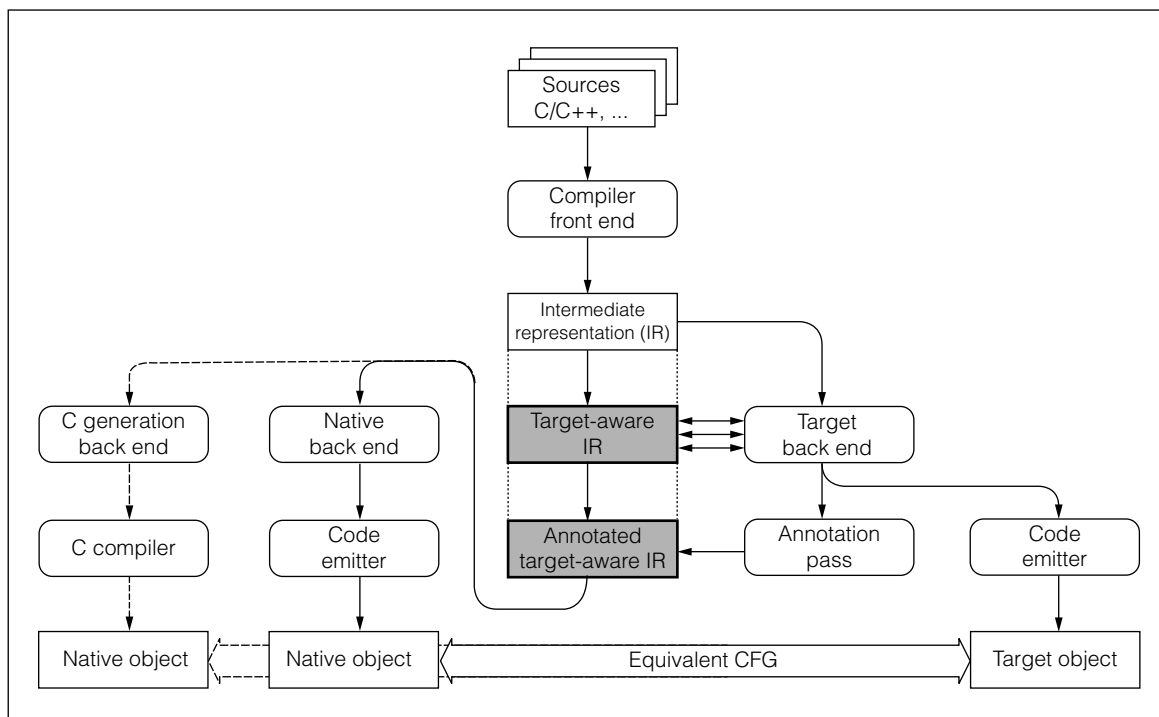
In its most general formulation, native software simulation targets the direct execution of software code on the host machine by using a wrapper to connect to an event-driven simulation environment. The initial proposals suggest encapsulating the application code into a TLM module, as if it were implemented as a hardware IP block using a single thread.<sup>9</sup> These solutions are simple but suffer from two severe drawbacks. First, the simulated code supports a very limited form of parallelism: coroutines. Second, because the software code executes in a hardware module, all data by the software is indeed allocated within the simulator process address space and not inside the target platform's memory. There is no way for a platform device to access a buffer allocated by the software if the buffer is not visible to it. Even if the buffer is visible to the platform device, the addresses of the simulated platform have no relationship to the addresses of the simulator process. This approach is clearly not suitable for supporting legacy code; thus, other proposals have emerged to solve both issues.

Seeking better concurrency support has led to the real-time operating system (RTOS) modeling approach.<sup>10</sup> The aim of this approach is to provide an implementation of a lightweight operating system using the simulation environment's event-based primitives, so each software task becomes a hardware module. The modeled RTOS relies on the hardware simulator's scheduler rather than that of the RTOS, even though some have proposed changing the hardware simulation kernel for this purpose. Some simple ad hoc kernels have been modeled using this approach, but the associated experiments required rewriting the application code, thus preventing this approach's use on legacy code.

Neither approach targets dynamic task creation or thread migration support between processors, both of which occur on symmetric multiprocessing (SMP) platforms that have recently appeared in SoCs. A few approaches have been proposed to solve this issue. These approaches rely on the definition of a thin HAL that must be used for all hardware-related accesses. This layer is implemented in a simulator wrapper that includes a simulator thread per processor, called a processing element (PE). Each HAL function call is performed in the context of its PE, assuming all PEs belonging to the wrapper share the operating-system code and data structures. Because the context-switching function belongs to the HAL, software thread migration is possible.

Correct handling of memory accesses is another challenge facing native software simulation. Native compilation of software implies the concurrent existence of two distinct, incompatible memory mappings: the platform memory mapping (which is defined by the hardware designers and used by the platform interconnect's address decoder at simulation time) and the simulator memory mapping (which is shared between the simulator process and the natively compiled software stack). Hardware-software interactions, such as programming a direct memory access (DMA) module with the address of a buffer allocated by the native software, simply will not work if this issue is ignored. Two main classes of solutions have been proposed: address remapping and address space unification.

Simple remapping techniques perform address conversion between the target address space and the simulation process address space for I/O accesses identified by the use of specific primitives. Remapping does not solve the issue of external access to natively allocated buffers, however. More complex remapping strategies rely on the fact that a host operating-system exception will be raised when a bad virtual address is accessed by the software.<sup>11</sup> The idea is to use the host operating system to mark the set of memory pages that are valid in the target platform as invalid in the host. Any access to these pages will raise an exception that the simulator can trap and handle. There could be performance ramifications of this technique if many exceptions are raised, and the technical aspects of handling overlaps between both memory spaces remain a problem.



**Figure 5. Annotation strategies. The normal use of the intermediate representation is the target object code generation (right side of the figure). Because it represents the actual control flow of the target program after all optimizations, it can also be used, once annotations have been inserted, to emit host object code that will behave exactly the same as the target object code (left side of the figure). (CFG: control flow graph.)**

Unification relies on the use of a unique memory mapping for the software and hardware components of the native simulation environment. The simulator process mapping is selected for this purpose because it is also the one used by the simulated software stack. Unification requires modifying the base addresses of memory sections and device registers in the software stack at the simulation's start time via mandatory external symbols in the hardware models. The drawbacks are the modification of the simulation models to build the unified memory space, the addition of a specific linking stage that is visible to the user, and the necessity for an operating-system port on the native HAL—all of which add significant overhead.

Native approaches can provide application and operating-system timing information by embedding simulator wait statements into the software code. However, most compiler optimizations lead to executable code whose execution paths are not isomorphic to those of the source code. Source-level annotation is, therefore,

inherently inaccurate, giving at best rough run-time estimates.

One solution to guarantee the isomorphism of the execution paths is to decompile the target binary into C and annotate it accordingly. However, because some semantic information is lost at code generation time, handling all address-related operations is a complex process.

More innovative approaches rely on the intermediate representations used by retargetable compilers.<sup>12</sup> Such representations, usually some forms of bytecode, contain all the semantic information and are used throughout the compilation process, including the optimization phases (see Figure 5).

The idea is to run the compiler as if the code is to be generated for the target, including compiler optimizations, but rather than emitting assembly for the target (right side of Figure 5), either emit it for the host (continuous path on the left side of the figure) or generate C and compile it for the host (dashed path on the left side of the figure). An annotation pass analyzes the target code and inserts calls to an

annotation function at the beginning of each basic block in the target-aware intermediate representation (IR). This annotated IR is then used to generate a native binary object, which has a control flow graph (CFG) equivalent to that of the target object and can be simulated on the host machine. The values to be added as annotations depend on the processor architecture for the instructions that do not access memory or I/Os, and on the platform for those that do.

As in binary translation, annotations make it possible to reach an accuracy close to that of instruction-accurate ISSs. The main source of inaccuracy comes from the host HAL calls, for which only static timing can be estimated.

**THE CHOICE OF ONE** software execution strategy over another is mainly dictated by the type of information that the entire MPSoC simulation is expected to provide and the maturity level of software and hardware development. Native simulation is well-suited to the development of the software stack's upper layers, for which it can provide accurate performance results at a high simulation speed by using fairly complex annotation strategies. Cache and memory system time models are coarse because some accesses are hidden in the lower software layers and because at least some accesses use host addresses. Adding timing annotation leads to a simulation speed reduction of approximately one order of magnitude. Native simulation relies on an implementation of the lowest software layers by a simulation model, so it cannot cope with assembly-level software nor execute self-modifying code.

DBT can execute an entire software stack, thus making it possible to obtain accurate performance estimates for this stack. Moreover, memory subsystem modeling can be detailed because all accesses are identified and target addresses are used. The simulation speed is close to the raw native simulation speed when no memory hierarchy is modeled; it is around two orders of magnitude lower when exact cache models are used.

In the context of processor design, when the instruction-set architecture (ISA) evolves, instruction-accurate interpretation will be far easier to use, both for adding instructions and debugging the software that makes use of them. However, the simulation speed, still several times lower than the most accurate DBT solution, will

intrinsically limit its use to low-level software validation. ■

## References

1. F. Ghenassia and A. Clouard, "TLM: An Overview and Brief History," *Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, F. Ghenassia, ed., Springer, 2005, pp. 1-22.
2. M.V. Wilkes, "The Growth of Interest in Microprogramming: A Literature Survey," *ACM Computing Surveys*, vol. 1, no. 3, 1969, pp. 139-145.
3. K. Popovici and A. Jerraya, "Hardware Abstraction Layer—Introduction and Overview," *Hardware-Dependent Software: Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, eds., Springer, 2009, pp. 67-94.
4. J.R. Bell, "Hardware Abstraction Layer: Introduction and Overview," *Comm. ACM*, vol. 16, no. 6, 1973, pp. 370-372.
5. L.P. Deutsch and A.M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proc. 11th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, ACM Press, 1984, pp. 297-302.
6. F. Bellard, "QEMU: A Fast and Portable Dynamic Translator," *Proc. USENIX Ann. Technical Conf. (ATEC 05)*, Usenix Assoc., 2005, pp. 41-46.
7. M. Monton, J. Carrabina, and M. Burton, "Mixed Simulation Kernels for High Performance Virtual Platforms," *Proc. Forum Specification and Design Languages (FDL 09)*, IEEE Press, 2009.
8. B. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, ACM Press, 1994, pp. 128-137.
9. R.K. Gupta, C.N. Coelho Jr., and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," *Proc. 29th Design Automation Conf. (DAC 92)*, IEEE CS Press, 1992, pp. 225-230.
10. A. Gerstlauer, H. Yu, and D. Gajski, "RTOS Modeling for System Level Design," *Proc. Design, Automation and Test in Europe Conf. (DATE 03)*, vol. 1, IEEE CS Press, 2003, pp. 130-135.
11. H. Posadas and E. Villar, "Automatic HW/SW Interface Modeling for Scratch-Pad and Memory Mapped HW Components in Native Source-Code Co-simulation," *Analysis, Architectures and Modelling of Embedded Systems: IFIP Advances in Information*

*and Comm. Technology*, vol. 310, no. 9, 2009, pp. 12-23.

12. Z. Wang and A. Herkersdorf, "An Efficient Approach for System-Level Timing Simulation of Compiler-Optimized Embedded Software," *Proc. 46th Design Automation Conf. (DAC 09)*, ACM Press, 2009, pp. 220-225.

**Frédéric Pétrot** is a professor in the Department of Computer Science and Engineering at the Grenoble Institute of Technology and heads the System Level Synthesis Group at TIMA Laboratory in Grenoble, France. His research interests include MPSoC simulation, architectural enhancement, and programming. He has a PhD in computer science from Université Pierre et Marie Curie, Paris.

**Nicolas Fournel** is a research engineer at Kalray, a startup company in Grenoble, France, that designs massively parallel multicore SoCs. He completed the work described in this article while working as a researcher in the System Level Synthesis Group at TIMA Laboratory. His research focuses on high-speed simulation and low-level software for integrated multiprocessor systems. He has a PhD in computer science from Ecole Normale Supérieure de Lyon.

**Patrice Gerin** is a research engineer at Kalray, a startup company in Grenoble, France, that designs massively parallel multicore SoCs. He completed the work described in this article while he was pursuing a PhD at Grenoble University. His research interests include simulation, validation, and profiling of MPSoC

architectures. He has a PhD in electrical engineering from Grenoble University.

**Marius Gligor** is a researcher in the System Level Synthesis Group at TIMA Laboratory under the auspices of the Grenoble Institute of Technology. His research interests include modeling and simulation of embedded systems and energy-saving algorithms at the software level. He has a PhD in electrical engineering from Grenoble University.

**Mian-Muhammad Hamayun** is pursuing a PhD in computer science at Grenoble University (under the auspices of the Grenoble Institute of Technology). His research interests include simulation and modeling of embedded systems using native techniques for performance estimation in MPSoC contexts. He has an MS in computer science from Joseph Fourier University, Grenoble, France.

**Hao Shen** is a researcher in the System Level Synthesis Group at TIMA Laboratory under the auspices of the Grenoble Institute of Technology. His research interests include fast simulation and virtualization technologies for embedded-system and heterogeneous multiprocessor architectures with related software support. He has a PhD in electrical engineering from Grenoble University. He is a member of IEEE.

■ Direct questions and comments about this article to Frédéric Pétrot, Laboratoire TIMA, Institut Polytechnique de Grenoble, 46, Avenue Félix Viallet, 38031, Grenoble, France; frederic.petrot@imag.fr.



## Silver Bullet Security Podcast

In-depth interviews with security gurus. Hosted by Gary McGraw.

[www.computer.org/security/podcasts](http://www.computer.org/security/podcasts)

Sponsored by  SECURITY & PRIVACY digital

**build  
your  
career**  
IN COMPUTING



Is your **career**  
**foundation**  
**solid?**

**Get the building blocks you need.**

Take your career to the next level in software development, systems design, and engineering with:

- Article collections from the IEEE Computer Society
- Materials from Harvard Business School Publishing
- Computer discounts
- Online courses and certifications

**Our experts. Your future.**

[www.computer.org/buildyourcareer](http://www.computer.org/buildyourcareer)

# RAISE YOUR STANDARDS

## Software Development



# Get Certified

*"Having the CSDP helped me strengthen our software quality process, reducing our production support costs by 40%."*

Phanindra Mankale  
CSDP

The CSDP certification is intended for mid-career software development practitioners and is based upon 11 Knowledge Areas (KAs) of the internationally-recognized Software Engineering Body Of Knowledge (SWEBOK) Guide.

#### Key benefits of CSDP Certification:

- 1 Practitioners:** Demonstrates your proficiency of established software development practices.
- 2 Employers:** Provides a method to build a more disciplined approach to software development practices.
- 3 Industry Recognition:** The CSDP was developed by the IEEE Computer Society, an international leader in the software engineering profession, in conjunction with key academic and industry leaders.

To learn more about our programs and how they can help your organization, visit us at  
[www.computer.org/getcertified](http://www.computer.org/getcertified)