Chapter 4

EMBEDDED SOFTWARE DEVELOPMENT
Through The TLM Approach

Eric Paire
STMicroelectronics, France

With special participation of Kshitiz Jain, Marc Harbonne, Maxime Fiandino, and Michel Bruant.

Abstract: Early embedded software development, covering coding, testing, integration and validation, is one of the most important targets of TLM platform methodology. This chapter describes mainly the close relationship between the TLM platform and the software running on it. The description illustrates how the software can benefit greatly from the early TLM platform availability. Reciprocally, hardware developers can also benefit from the early feedback on their design when used by the software developers. The TLM platform can therefore be considered as the meeting point between hardware and software development teams.

Key words: software; Operating Systems; firmware; device drivers; application; protocol stack.

1. INTRODUCTION

Nowadays, no hardware design of a system-on-chip is worth developing without any software to exercise its functions. The trend of “the smaller the better” in SoC design concept has rapidly pushed the role of software into prominence during SoC hardware design process. While hardware aspects are getting very tough to handle due to the ever-rising SoC complexity, the weight of software aspects becomes more and more important in the overall system to manage new hardware functionalities and to replace certain hardware features.

This chapter highlights the brand-new role of software in conjunction with TLM platforms. It underlines the core idea of how system embedded software and TLM platforms could enhance and enrich each other in their respective missions.

The conventional design approach allows a significant amount of the software being developed, compiled and tested before any strict form of the hardware platform is made available. Only a specific part of software could be developed when the detailed information tightly associated with the hardware is accessible in the form of RTL or emulation platform. This part is usually the toughest and longest to test and debug. Unfortunately, software developers are always bound to wait quite long for such hardware platform in order to validate their development work. This is not only a costly time loss, but also an inefficient cooperation between hardware and software designers for lack of a common development base.

Despite the somewhat opposed design philosophies between hardware and software fellows, current SoC complexity is urging these two worlds to work together in a new way leading to concurrent hardware/software design. Time-to-market reduction and cost saving will be the successful culmination of such parallel hardware/software design.

The idea of hardware/software co-design and co-implementation can be realized through a unique reference -*the TLM platform*-. Indeed, TLM platforms provide adequate and accurate hardware information for software designers much earlier than the conventional platforms such as RTL platforms. This information must be sufficiently accurate for software designers to start developing, testing, and debugging the software code closely associated with the hardware *without* pointless delay following the initial software development. In parallel, hardware designers can develop RTL platforms aimed at timing-accurate simulations, which are eventually employed for logic synthesis.

By the time the RTL design is complete, the software will have already been thoroughly verified on TLM platforms. The software design is thus ready to be integrated with the RTL hardware platform for system validation within a much shorter time than the traditional approach. As a result, sound and solid concurrent engineering is achieved through the unique reference of TLM platform.

A closer study clearly reveals that software running on TLM platforms can be classified into different categories according to their relationships with the hardware platform. This chapter will discuss extensively on the software categories ranging from design requirements to the mutual expectation of benefits between software and its hardware counterpart. Lastly, the chapter will draw a conclusion on how close collaboration between hardware and software developers could lead to a virtuous circle.

2. SOFTWARE TARGETED FOR TLM PLATFORM

Throughout the development of a new SoC platform, various teams participating in the hardware design are always interested in running software programs on the platform. Be it any team varying from RTL design to functional verification and integration, early software execution means early catching of hardware or software problems. More essentially, executing software on the target platform helps to identify any potential mismatch between software and hardware designs.

In spite of its very attractive advantages, getting ready the software for early phases of SoC design cycle should never be done at any inappropriate cost of software development. The software should be executed on a development platform that is as close as possible to the final hardware platform. That will increase the probability of software reuse on the target platform with very little or virtually no modification on the subsequent hardware platforms. Such reuses trim down not only the overall software development time, but also the cost of refining software for these platforms.

A key parameter of developing the software targeted at running on TLM platforms is the immediate usability of the software in the current hardware design process. It is not quite convincing to claim a software piece being developed early in a project *useful* if that software piece could only be validated on a later hardware platform. The software must be tested on the target hardware platform while it is being developed. To bring the software and hardware design in parallel, they must be managed in tandem for scheduling smooth meeting points that optimize their mutual enhancements.

Running software programs on TLM platforms may appear easier than what it could really be for several reasons listed below:

1. TLM platforms are *not* real hardware platforms but abstract models for new platforms or IPs under design. To reach optimal uses of TLM platforms, software adaptations might be necessary.
2. TLM platforms have diverse modeling varieties. Each model might involve subtle adjustments in the software to adapt for non-fully covered features such as interrupt request (IRQ) or input/output (I/O).
3. Software compilations might require specific coding rules for proper program-runs in certain simulated environment of TLM platforms, for instance, compilations for handling timing issues on inexactly timed platforms.

All these reasons seem coercive on the software development using TLM. These good reasons, however, will definitely lead to efficient software coding and better code quality if they are appropriately practiced.

2.1 Adequacy of Software and TLM Platform

2.1.1 TLM Platform Accuracy and Availability for Software

The software development through the TLM approach depends closely on the modeling level of the corresponding TLM platform, which directly reflects the level of accuracy of the target hardware platform.

TLM platforms not reaching a minimal level of the functional behavior of the real platform may mislead designers to an erroneous software development by masking certain mistakes or bugs. The harmful consequence would be giving the wrong impression that the software is validated and ready to run on the real hardware platform. If a TLM platform poorly simulates the final hardware, very few software programs will be able to run correctly on it. It may miss testing critical features for hardware validation. The amount of time spent in such software development will be wasted and hence a higher global time-to-market.

On the contrary, it is sometimes unnecessary to have all design features simulated in TLM platforms if the whole process of concurrent hardware/software engineering is not significantly improved. Consider the following situation: Running natively compiled software on a timing-accurate TLM platform will *not* give any clue to the final software performance on the target platform. For such case, instead of developing timing-accurate TLM platforms, it could be easier to insert annotations obtained from cross-compilation into natively compiled software codes for studying software performance. Such annotations provide accurate statistical timing information without considering hardware features like cache, memory management unit (MMU) or write buffer, which could heavily influence the software performance in simulation.

Executing a software program on various functional TLM platforms has resulted remarkable outcomes. As an example, running a JPEG decoding program either on a PentiumIV with 1Mbyte of internal cache or on an ARM926EJ-S with 16Kbyte of internal cache may yield vastly different performance results of latency and throughput. The results of executing the software on TLM platforms help to better analyze various aspects of the hardware and software relationships. The software efficiency and correctness on the simulated hardware or hardware modifications for facilitating software development are examples of such potential improvements.

More importantly, running software on functional TLM platforms brings mutual benefits to the two working parties:

- *Hardware Developers*

  A live picture of how software programs utilize TLM hardware interfaces for real applications, which subsequently helps to improve the functional view of IPs on hardware platforms.
- *Software Developers*

  A live picture of how TLM hardware IPs react when software programs are executed on TLM platforms, which subsequently helps to improve the software implementation.

Indeed, these mutual benefits require not only the appropriate modeling choices of TLM platforms tailored for varied software design purposes, but also the proper manner of developing software in the right perspective of TLM platforms available at different design phases. Such careful matching of software development with TLM platforms is what we mean by the "adequacy of software and TLM platforms", which aims at optimizing the software development through the TLM approach.

2.1.2 Layering Software in TLM Platforms

To achieve such optimization, the software should be developed in progressive layers corresponding to the different levels provided by TLM platforms for simulation. This idea is illustrated by the development of a software driver for a UART sending and receiving characters on a given platform. The coding approach normally begins with a character-by-character interface, although a direct memory access (DMA) can be used on the final target platform. In the early design phase, an added-value feature like DMA may not be available yet in the hardware platform; besides, adding DMA in the TLM platform may cause some undesirable time delay in simulation. Most of all, it might be inefficient to use DMA for handling just a few characters because more management of registers and more software managing I/O blocks will be involved for the same number of interrupts. Thus, it is best at this point to start developing the driver without supporting DMA.

The good practice of the "layered" software coding through the TLM approach is strongly recommended. This concept is illustrated in the example of splitting a UART driver development into five phases as described in Figure 4-1. In the figure, each phase is represented by a task box. The size of each task box reflects roughly the relative amount of work dedicated for that particular phase with respect to the overall development.

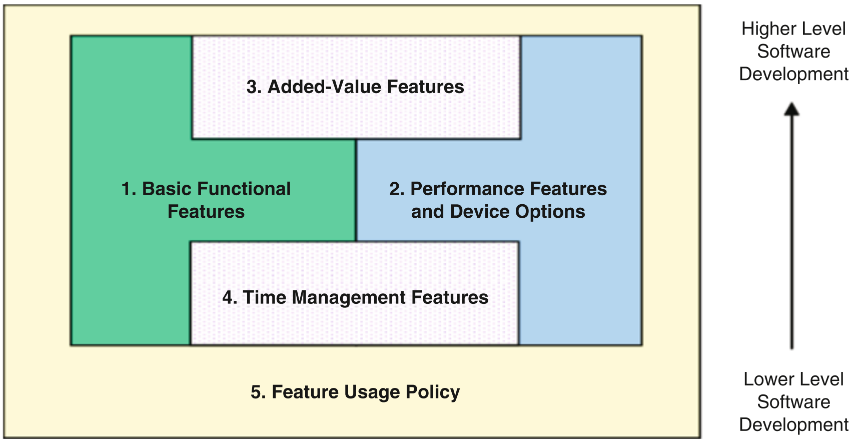
3. Added-Value Features
1. Basic Functional Features
2. Performance Features and Device Options
4. Time Management Features
5. Feature Usage Policy
Higher Level Software Development
Lower Level Software Development

Figure 4-1. Layered Software Development

1. *Development of basic functional features.*
   In this example, first phase focuses on developing a functional UART driver managing simply character-by-character I/O interface. It is fast to be developed for an early interface testing.
2. *Development of performance features and device options.*
   Second phase develops performance features and options of the UART driver such as DMA access and cache management. Usually, these features can be easily inserted within the static conditional compilation.
3. *Development of added-value features.*
   To build a complete functional UART driver, all added-value features are developed in third phase; for instance, sleep/wake-up mode or performance counters. These features may be essential to help software designers in developing application software at higher level.
4. *Development of time management features.*
   Fourth phase concentrates on developing time management features of UART driver such as those for sleep mode or I/O completion delayed interrupts, which are dynamically configurable through external parameters. These features are typically very close to hardware view.
5. *Development of feature usage policy.*
   The final phase of "layered" software development determines the policy of how and when all the optional and performance features should be strategically employed. The mechanism of using all the features developed in the four phases earlier is carefully refined in this phase.

From the performance's point of view, the cost of TLM transactions is not very dependent on the amount of the data transmitted. In the example of the UART driver development, sending the entire text of a message using a DMA will be much faster than using a character-by-character I/O. Although the layered approach is valuable for the software development, using character-by-character I/Os in TLM platforms is inefficient due to their very long testing time: Assume that a character I/O takes N register accesses in the UART IP, i.e. each character will require N TLM transactions. Suppose that for every UART access, the DMA makes M register accesses. If the DMA is enabled, each DMA access to the UART IP can include any number of character I/Os, which will require only M TLM transactions. This will certainly utilize TLM platforms much more efficiently. Therefore, such performance features should be considered early enough in the design cycle to increase TLM platform overall efficiency.

To conclude, there are three rules to respect for the optimal software development and execution on TLM platforms:

1. *Do not develop software too much in advance*. It is not worth developing the software for hardware features available very late or prone to change in the future. Time saving may turn out to be worthless or extra delay may occur when hardware pieces are available or modified later because of the adaptation time.
2. *Organize software development tightly coupled with hardware design in layers adapted to IP functions*. Basic but complete features should be clearly separated from optional parts. These features should be incrementally tested in phase with their addition in TLM platforms.
3. *Give priority in developing performance features and device options for better software performance on TLM platforms*. If this is not appropriately done, software developers may not make the most efficient use of TLM platforms (they may probably get discouraged to use the TLM platform due to its slowness).

2.2 Analyzing Software on TLM Platform

As presented in Chapter 2, TLM methodology offers two distinctive models of the hardware platform for software development, namely untimed TLM (PV) and timed TLM (PVT). The current section focuses on how the software should be adapted for running on different models of TLM platforms.

Practical software properties will be provided throughout this section to demonstrate the global software quality improvement that could be brought by each TLM model. Such improvements will be compared to what RTL models and real chips can do for the software development today.

2.2.1 Functional Accuracy

TLM platforms are designed to provide an accurate *functional* view of the final hardware platform so that any software with correct *functional* behavior will be able to run on them. This may not include running the software with some non-functional aspects of the hardware platform such as real speed, linear time, or event ordering.

On top of the layered software development explained earlier, writing the software that is independent of any timing or event ordering issues is another good coding practice reinforced by TLM. For example, assume that an I/O starts with a register-write. The associated software should be ready to receive the I/O completion event at any time starting from the return of register-write operation. The same sort of the software functional behavior can sometimes occur on real chips because of I/O errors or suspended instructions due to interrupt handling. An untimed TLM platform, however, can offer the same advantage at much earlier availability!

Another example of analyzing the software functional behavior is described hereafter. Imagine that a software code reads some data from an always-ready source and writes it to a sink. In the real life, the sink will take some time to handle the data before it is ready to consume more data. Meanwhile, that extra delay will allow the software to perform other tasks. In untimed TLM platforms, the sink may accomplish the task instantly or in very small simulation time. The software will thus be ready to keep getting data from the source and passing it to the sink. If the software is not able to handle such behavior, it will spend all its time moving data from the source to the sink but nothing else! Running such software on TLM platforms will give the wrong impression that the functional behavior of either hardware or software is incorrect.

As long as the TLM platform is functionally correct, it will provide an absolute time reference with strict event ordering although it may not be time-accurate. Indeed, the root of the problem above is writing the software with the assumption that the sink will take enough time to handle its data to allow other tasks being scheduled. Two methods can handle this situation properly:

1. Let the software manage its tasks in the round-robin such as simple executive runtime.
2. Let the software handle the I/O management on an event basis. It will require some software adaptations for TLM platforms. The same problem may still occur if the software has too many events to

manage. This method, however, helps to handle certain rare real life situations that are probably never really tested in real chips.

This example clearly illustrates how the software should be adapted for the chosen model of TLM platforms for an appropriate analysis of the software functional accuracy.

2.2.2 Global Time Accuracy

The global time accuracy of TLM platforms is not an easy aspect to handle. The reason is that a system should be able to run even if it is *not* time-accurate. Since timing is very often an important feature for the software, untimed TLM platforms cannot completely ignore the timing behavior. Instead of implementing the full timing, events are strongly ordered within each IP. There is *no* global order for events occurring in different IPs, meaning that delays between event occurrences of different IPs are not accurate.

Implementing the global time accuracy in the software is not particularly difficult. The software, however, must be ready to manage this behavior proficiently. It is a bad coding practice to assume the order of two event occurrences in a system. For example, a timeout should be programmed to occur anytime after its scheduling without assuming that it may not occur before something else.

The major difficulty of implementing the global time accuracy in the software is the task management based on timing but not on event, for instance, time-sliced scheduling of Operating Systems. Such implementation is usable only if the software can ensure that a task is able to complete a sufficient amount of work before a time-slice. The system could otherwise be reduced to switch from task to task with little or no time to perform anything useful in between! In this case, the software may appear functionally correct but the execution result could be too far from the expectation of software developers. Software cannot do much to solve this problem. Rather, the hardware platform should give some hints on the time evolution such as estimates of software time expenses. When running software on untimed TLM platforms, software developers should somehow be ready to see some unexpected timing behavior of their programs.

In contrast, it is quite a different matter to handle the global time accuracy of the software on *timed* TLM platforms. Such platforms are able to provide the global time accuracy, i.e. a strong ordering of events for the entire platform. The software can thus be executed more accurately with respect to its timing behavior, including timeout, time-slice, or delay required by platform IPs in handling I/O events. Unavoidably, such timing accuracy is paid by a much less efficient software execution because there are more events to manage compared to those in untimed TLM platforms.

The global time accuracy of a given platform depends very much on the way of how IPs are implemented in the platform. If all IPs comply with the timed TLM constraints, the entire platform will be globally time-accurate. Software programs may run only with approximate timings on the hardware platform in cases where certain IPs are not timed TLM compliant, or native software compilation or non time-accurate ISS is employed. Nevertheless, it could be interesting to test the software in environments that are different from the final timed platform.

Obviously, it is more understandable to develop and test the software on timed TLM platforms with fine-grain timings than on untimed TLM platforms with approximate timings. The most suitable choice for analyzing the software behavior related to the global time accuracy is of course the timed TLM platform. Software programs, however, should run correctly without any code modifications on both untimed and timed TLM platforms.

2.2.3 Protocol-Timing Correctness

When an external component is connected to a SoC, software developers need to program the relative timings correctly for eliminating any potential communication hazards. This is probably one of the trickiest problems to solve in the software because its failure cannot be easily detected on RTL hardware platforms. The symptom of such problem is typically an unstable system that works properly for some time, but crashes suddenly with no warning signs.

Timed TLM platforms are the best spot to uncover such programming errors. For example, PVT platforms can effortlessly reveal insufficient wait states for accessing a memory IP by comparing the number of wait states programmed by the software to its internal characteristics. To do so, the PVT memory controller validates if the time amount required by the memory access is coherent with the number of wait states programmed. If the wait states are insufficient, the memory IP can send a notice thanks to the timing information held by TLM transactions.

The concept explained in the example above, by analogy, applies to any other external controllers connected to SoC platforms via standard industrial buses such as I^2C, CAN, I^2S, SPI, and so on. Once the first prototype board around a SoC platform is built, it is usually too late to fix an external protocol-timing problem where platform controllers and external devices sharing the same protocol fail to communicate. A "quick and dirty" way to overcome such hardware problems is to modify the software, which unfortunately results in, most of the time, reduced performances and functionalities.

Protocol collision management is another protocol-timing test that can easily be set up thanks to TLM platforms. Some simple bus protocols such as CAN or I^2C are designed to solve collision issues by forcing a master to be a slave, which will consequently change the behavior expected by the software. Protocol collision is a very difficult software behavior to test because forcing collision on hardware is a tough procedure that usually requires special hardware to test all potential cases. Although a bus-cycle accurate platform can set up all types of collisions, timed TLM platforms are sufficient to set up global collision required by software developers at an earlier availability. In addition, the input of the TLM platform could be programmed to show such specific hardware behavior. Thus, it can provide software developers with the ability to validate the actual software behavior on demand.

2.2.4 Resource Overflow

With the advent of SoC, software developers have somewhat lost a little of the control they used to have over the unexpected limit reached by performance. Consider the following case of resource overflow: a 100Mbps Ethernet controller together with a fast CPU can sustain an Ethernet flow close to the theoretical limit, particularly for full duplex mode without collision on wire. If the theoretical limit is far from being reached, software developers can use a packet analyzer to examine the packets received by Ethernet driver from the controller. They might sadly notice that the packet is surprisingly in coherence with the speed announced by the application. The only solution is to analyze deeper the packet flow between its input in the Ethernet controller, and the interruption signaling for its availability in the memory.

In general, it is extremely hard to peek at the activities going on inside a SoC. But, there are so many hardware items involved in the packet management (IP, DMA, buses, caches, etc) that it is almost impossible to easily detect any bandwidth bottleneck. Resource overflow, on top of this difficulty, is very often hidden by some hardware limitations in bandwidth, access priority, etc. All these factors make this specific problem a real tough job to fix for software developers. In addition, RTL platforms are not exactly the right solution due to their performance limitation.

A good tactic to cope with resource overflow will be employing TLM platforms because they provide adequate details and hints to guide software developers in locating the problem. Timed TLM platforms optimize timing measurements to avoid all hardware contentions in accessing resources on the platform. As a result, it is easier to get the best performance measurements especially for cases where cycle-accurate ISS is applied. If the performance is satisfactory, software developers can proceed with a bus-cycle accurate platform, which gives results on the miscellaneous hardware contentions that the system has to face for this particular test. With all these results, software developers will be able to locate the problem of resource overflow.

2.2.5 Performance Profiling

The foremost interest of executing software programs on TLM platforms is of course getting the software running on the target platform. Once the software gets up running properly, the next goal will be collecting early performance results before the final hardware is available. Performance measurements are not only based on timings, but also start with non-timing counters such as the volume of transactions exchanged by IPs. This job can be accomplished adequately by untimed TLM platforms.

Untimed TLM platforms, however, cannot do much to obtain timing results. Attempting this on hardware platforms may not be the best choice because the measurement software itself could modify the overall timing of the platform. Since measurement mechanisms are embedded in the IPs, timed TLM and RTL platforms are both capable of evaluating timing results without altering the overall timing of a given platform. Obviously, timed TLM platforms are better options than RTL platforms for performance profiling thanks to their usual earlier availability.

Inconveniences may arise in common practices of performance profiling. Frequently, software needs to be modified to obtain profiling results. The measurement software is thus intrusive on the system platform. Sometimes, the profiling procedure could be dreadfully time-consuming or the external hardware required for extracting profiling results from a platform may not be available all the time.

Through timed TLM platforms, however, all these inconveniences are straightforwardly resolved. Since measurement mechanism is embedded in the platform IPs, performance profiling is *independent* of any software running on the platform. That will greatly reduce the workload of software developers.

The example of latency profiling gives a better idea of how helpful timed TLM platforms could be for software performance profiling. Latency is very hard to finely measure when the software is running on the hardware platform. Such difficulty is particularly bitter for real-time systems that are extra-sensitive to latency issues. Timed TLM platforms, nevertheless, can run real-time software without any modification to conduct profiling such as building the histogram of interrupt latency. Therefore, a software developer can get fine and accurate results without any modification of software, just by extracting the right profiling from its timed TLM platform.

2.2.6 Hardware Utilization

Running software on TLM platforms grants the ability to detect whether software makes the *right* use of hardware platforms. Additional non-functional code can be embedded in TLM platforms to validate if hardware is utilized properly as expected by its design. Although hardware could tolerate certain bad or poor utilization by software, the resulting effects of such use are sometimes likely out of software expectations.

Consider the example of UART transmit-character register. Under normal practices, it is not permissible to push another character into this register if the previous character is not yet consumed. The hardware, however, allows software to freely write characters in this register as many times as it wants, without any effect on the IP behavior. Most of the time, overwriting character in such manner is a programming error. TLM platforms can help to verify the same sort of programming errors without much effort. As a result, software developers can obtain reliable hints on the potential programming errors in the software.

TLM platforms also provide interesting results about the software utilization of particular hardware features. For the same register in the last example, certain UART IPs allow software to push another character in the register while the current one is being transmitted. This is a special feature to reduce the latency between the end-of-transmit interrupt and the availability of the next character to be transmitted, which software developers are invited to use as much as possible. Internal counters can easily be enabled to measure how frequently this hardware feature is used by the software. Following the simulation, a statistical listing for the utilization of special hardware features can be provided. Based on the list, software developers can learn better about the hardware utilization by their software implementations, whereas hardware developers can see the actual utilization of hardware features in real cases.

2.2.7 Conclusion

After discussing on how untimed and timed TLM platforms can help software developers, Table 4-1 summarizes and compares the different kinds of software behavior that can be studied at different modeling levels.

At first glance, the summary may mislead to the conclusion that bus cycle-accurate (BCA) platforms give the best software support. This could probably be true if the overall platform performance and setup work are *not* considered. This is the reason why these two criteria usually determine the interest level of using a TLM platform model for running, testing, and debugging software before RTL and real hardware platforms are available.

If these criteria are considered, BCA is certainly not the best option because both untimed and timed TLM still provide faster performance than BCA, and are usually set up and integrated much quicker. Although RTL is the slowest for performance and construction, its vital hardware simulation capabilities make it necessary to be constructed (normally after TLM platforms). Concisely, TLM platforms are the most compelling models for running and testing software before the real chip is available on silicon wafer.

Table 4-1. Software Behavior Observed at Different Modeling Levels

Software Behavior	PV	PVT	BCA	RTL	Silicon
Functional Accuracy	Yes	Yes	Yes	Yes	Yes
Global Time Accuracy	No	Yes	Yes	Yes	Yes
Protocol-Timing Correctness	No	Yes	Yes	No	No
Resource Overflow	No	Yes	Yes	Yes	No
Performance Profiling	Yes/No	Yes	Yes	Yes	Yes/No
Hardware Utilization	Yes	Yes	Yes	No	No
Accurate Concurrency	No	No	Yes	Yes	Yes

PV = Untimed TLM BCA= Bus-Cycle Accurate
PVT = Timed TLM RTL= Register Transfer Level

Notice that the accurate concurrency is a behavior listed in Table 4-1 without being discussed earlier. This is a critical behavior to analyze when two or more IPs try to access concurrently the same platform resource like bus or DMA. Such concurrency is part of the functional accuracy that can be implemented in TLM platforms. The accurateness of such concurrent collision, however, is not handled by TLM because it requires cycle accuracy to manage the interactions and requests of platform IPs.

2.3 Software Environments of TLM Platform

Running software on TLM platforms depends not only on the platform design, but also on the different environments in which the software will be handled. There are four major TLM software environments, which will be discussed in the coming sections:

- *Software Development Environment*
  Describe how software is produced and debugged.
- *Software Execution Environment*
  Describe how software is executed on TLM platforms.
- *Software Integration Environment*
  Describe how software is integrated into TLM platforms.
- *Software Simulation Environment*
  Describe how software gets input data and puts output data.

As depicted in Figure 4-2, these four software environments correspond very well to the famous V-diagram for the life cycle of software. Each of the environments prepares the necessary setting for performing the different software work at various phases.

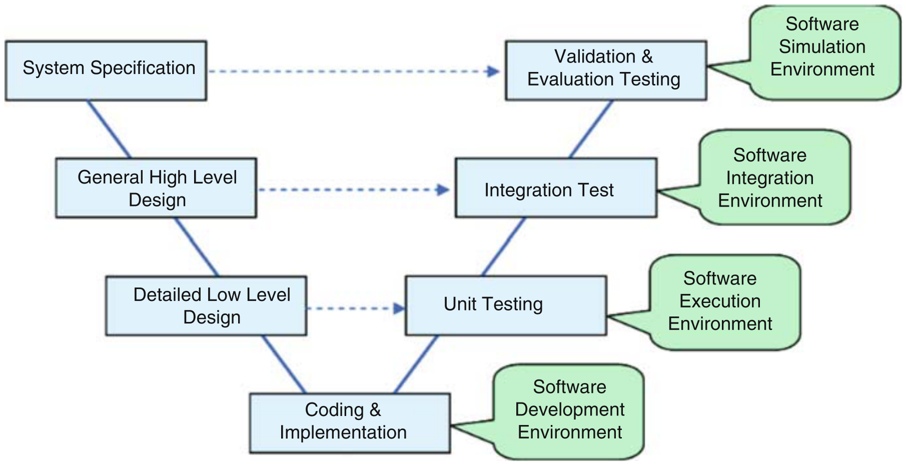
System Specification
Validation & Evaluation Testing
Software Simulation Environment
General High Level Design
Integration Test
Software Integration Environment
Detailed Low Level Design
Unit Testing
Software Execution Environment
Coding & Implementation
Software Development Environment

Figure 4-2. Relating TLM Software Environments in V-Diagram

2.3.1 Software Development Environment

TLM offers the great advantage of having a simulated hardware platform that can be either natively compiled for faster speed or cross-compiled for binary compatibility and higher accuracy. This dual compilation capability therefore provides two development environments to software coding and implementation.

The *cross-compilation* development environment requires embedding a model for the targeted processor (usually called an ISS) in the TLM platform. The software can then be compiled for the actual target processor and simulated by the ISS. The software is thus isolated from the TLM platform execution by the host system.

The *native-compilation* development environment merges the execution of the software with the execution of TLM platform IPs by the host system. The software is link-edited with the TLM platform simulation code and executed as part of the complete platform process; the main characteristic is that software shares the same address space as the platform simulation code itself.

TLM software development environment relies heavily on the decision made for software integration. Different integration methods require different integration tools, for instance, native integration necessitates different tools from cross-integration. According to the opted integration method, the appropriate development tools must be applied; and that will determine the software development environment.

Certain development tools, however, remain the same for either native or cross integration. A handful of examples include editors, source code generators, and particular compilation suites such as those using GNU tools. Sometimes, it is even *compulsory* to keep the same tools. Consider the example of GNU tools: If GCC and binutils are used, source code must be compiled exactly the same manner in either native or cross environment. The reason is that different compilers may actually require code adaptations due to their different specific syntax or extensions.

Using two different development environments (and thus two different integration environments) reinforces software portability, especially if both have different compilers. Not only can the code quality be improved by porting the software on two distinct environments, but more potentials problems can also be uncovered through different code compilations.

Conversely, software may undergo the side effect of being sensitive to certain processor aspects listed below due to using two different central processing units (CPU) in its development environment:

1. *Endianness*. The software must be ready to support any endianness (little, big, reverse, cross, etc) if the two processors (native and cross) have different ones.
2. *Assembler*. If the software embeds assembly codes as C extension, the same function ought to be available for both processors; it should otherwise be replaced by a functionally equivalent but less performing C code.
3. *Self-modifying code*. If the embedded software uses self-modification, a similar feature must be made available in the native environment.
4. *Data alignment and size*. If the embedded software relies on specific data alignment and size, then the software must provide all used compilers with these requirements.

5. *Addressing features*. If the software relies on specific addressing features imposed by the final processor, they must be implemented by any potential native processor.

In the software development chain, post-compilation tools for debugging and profiling could be very different between native and cross-compilation. Software debuggers, in particular, can be totally different. The native debugger controls the running platform directly whereas the cross debugger controls the platform indirectly via a client-server architecture. Showing too many tiny details of the TLM platform to software developers is an additional problem of the debugger in native environment. It could be very confusing for those developers who wish to debug their software but not the hardware. The native debugger should then be adapted to display only necessary information to software developers.

Compared to debugging, software profiling on TLM platform is quite a different matter. It is only worthwhile for special cases as follows:

1. *Profiling conducted on natively compiled TLM platforms*. Although the results can be very different from the final platform, it gives some valuable hints on the behavioral performance of the platform during early development phase, such as access counters. Calling graphs might also be extracted in such profiling for early performance and execution path analysis.
2. *Profiling conducted on cross-compiled timed TLM platforms*. Such profiling provides the very first idea of software profiling with coarse-grain timing before RTL hardware platform is available.

2.3.2 Software Execution Environment

TLM software execution environment is determined according to the adopted development environment. Software reaching this phase should be ready to be executed for unit testing, either as native compilation or as cross-compilation with an ISS.

With such performance-reducing factor as ISS overhead in cross-execution or hardware emulation, native execution is certainly the fastest execution environment. This is nonetheless not always a true statement because timing issues in natively compiled codes are totally different from those in cross-compiled codes. For an example, a different timing in hardware could probably cause such an overhead that the software is paralyzed, or it could probably run the software correctly in the native mode but masking some stubborn bugs that would only be visible under ISS execution!

One of the assumptions held in the previous example is that the compiler chain produces correct code in both native and cross cases. Running native codes can help nothing in debugging cross-assembled parts or coprocessor specialized instructions. In addition, certain data representations cannot be compiled because they are unavailable on native platforms, for instance, floating-point representations.

The toughest challenge in software execution is the memory mapping of the software. It is quite straightforward for software cross executed with an ISS. The software simply runs in the memory space defined by the ISS, i.e. the memory zone perceived by the software in the platform. The situation, however, becomes trickier in native execution. The software is bound to run in the memory space defined by the local host, which could be different from the one programmed in the software for the final hardware platform.

Consequently, the software needs to be relocated into this different memory zone. Addresses of memory layout might need to be translated to addresses not used by the underlying host system. Some software adaptations are required to remap cross-compiled hardware addresses into natively-compiled addresses without flaw. There is something similar to implement for register accesses. The reason is that they are not simple memory-mapped read and/or write accesses as in the cross-compiled environment, but requiring some modifications to fit the actual bus modeling schema.

The register access remap should never be regarded as useless overhead, but rather as a good software coding practice. It allows the re-definition of hardware register accesses via generic read and/or write macros according to different compilation modes. Native compilation paves the way for software developers towards the first functional view on the final hardware platform; meanwhile, it enables the implementation of valuable portability features in software.

Among all the possible native execution environments, the operating system (OS) emulation deserves a special hat's off. Its goal is to abstract the interface between the OS and the hardware platform to set up a native environment. In this environment, applications can run natively on the OS layer while the OS itself can run natively as well on the hardware platform. Since the CPU used in the host machine is more powerful than the one in the real hardware, such setting can reach very high performance by running software on the simulated platform much faster than on the real hardware platform.

2.3.3 Software Integration Environment

TLM software integration environment provides the right setting to perform integration tests for a given system. It is not a simple task to determine how TLM software should be integrated into a hardware platform, especially when multiple solutions exist. One of the solutions is to incorporate the embedded software into the simulated hardware. It suggests that the software interacts with the hardware in terms of reading or writing data. These interactions are simulated as software actions on TLM platforms. For example, a hardware IP register access is interpreted as calling the right function in the IP module of TLM platform to simulate the access.

When modifications are necessary, it is preferable to change the software instead of the hardware for the reasons of cost, time, and workload. Therefore, it is sometimes desirable to separate software from hardware. An alternative solution could be compiling software for the target CPU and simulating IP accesses through an ISS.

Bear in mind that performance is one of the main criteria for using TLM platforms. In the alternative solution, performance is yet a problem because ISS is not as fast as native CPU. If performance is the main consideration, the most appealing solution could be native execution. The software must then be link-editable in TLM platforms, and that could probably be a source of diverse problems. Some of the possible problems are listed below:

1. TLM platform is link-editable through some external libraries that must be compatible with those of the software. If they use different or incompatible versions for the same library, the integration will fail because the same symbol may cover different functions.
2. If the software defines external symbols that collide with those of TLM platform libraries, the same problem as in (1) will occur.
3. The software is obliged to compile with the definitions of TLM platform that could potentially collide with those of the software.
4. The software may use process resources such as signals, memory mapping, and file descriptors in an incompatible manner with those on TLM platforms.

This list is non-exhaustive but enough to show the lurking problems that could appear anytime during the integration process. It is therefore hard to decide beforehand if native execution is feasible, although it may appear attractive in performance. Anyway, a potential solution always exists, i.e. integrating software into a cross-compiled environment where the software runs independently of its hardware platform.

2.3.4 Software Simulation Environment

Once it has managed to execute and integrate correctly on the target platform, TLM software will proceed to the software simulation for validation and evaluation testing. The software usually cannot run alone in the simulation environment because the entire board holding the SoC is involved; meaning that some external input and output data flows are required to conduct such simulation in a real environment.

A simple way to establish connections between the platform and the external world is to input/output data of platform IPs from/to local host files. Its greatest advantage is the easy setup that enables software to run test samples promptly from the local host files. Such reference samples will really be handy for debugging algorithm or platform behavior of certain final code, say protocol decoding.

Connecting with local host file is not always sufficient. It is interesting to connect IPs with real devices in certain cases; for instance, interfacing a card reader IP with a serial line, or bringing the actual character protocol into an UART IP to allow testing software on emulated hardware that is connected to real hardware. Such "real" connection can also be employed for buses like Ethernet or USB through the host system devices.

Another interesting aspect of the TLM simulation environment is its ability to report the input/output of hardware multi-media to the host. Consider the following example. If the software is designed to use an LCD of a given size, it is quite straightforward to map the LCD on the host graphical window. That allows debugging the exact contents provided to users without needing to write a single line of code, which is anyway not reusable on the final hardware platform. Essentially, this aspect is the most remarkable difference of TLM simulator from an emulator that really entails interfacing with the software.

The greatest interest of exporting the simulation environment out of the platform is to provide total flexibility in the way of connecting the platform to the external world. Defining standard interfaces for internal IPs is a corollary of giving software developers such flexibility in the simulation. With this flexibility, software developers can simulate their design with the external world in any way they wish (including incompatible simulations), and to any extent they wish (up to the complete simulation). The platform with such interfaces needs not to embed any external input/output devices such as graphical windows to simulate serial communication. As a result, the platform is more portable from one system to another because the communication will be standardized via an open socket protocol.

2.4 Conclusion

TLM platforms provide software developers with a brand-new interesting methodology to test the software in a hardware simulation environment. Since the simulation is pure software, it is possible to set up different environments depending on the characteristics required by the platform, including accuracy, performance, connection to the external world, and so forth. In fact, TLM has filled up the gap between software and hardware developers. A bridge is now constructed between these two teams to enable each of them to observe from their own perspective how their development work is used by another team.

2.4.1 TLM Impact on Software Development

TLM platforms provide software developers with a hardware base to develop and more importantly, to test their software long before any pure hardware emulation is available. This is particularly helpful for the new hardware IPs on which no software has ever been ported or written yet. The major advantage of such early software development and testing in the SoC design cycle is to reveal any potential problem between hardware and software prior to their delivery.

Developing software that can be simulated immediately on the target platform is certainly beneficial. It helps to produce better software implementations in terms of portability and hardware utilization. In general, TLM reinforces good practices in software development process.

Based on TLM platforms, software developers can fully focus on the coding targeted for the final hardware platform without building any temporary dummy (and sometimes costly) hardware platforms. The software can be simulated at different accuracy levels on TLM platforms in the different environments required by the software developers. Such conveniences grant software designers ample freedom to perform their job without waiting keenly for the first hardware platform.

2.4.2 TLM Impact on SoC Design Flow

The overall SoC design flow has to be reconsidered when using TLM. This is essentially the foremost impact of TLM on the SoC development. A TLM platform is regarded as the first hardware prototype wherein software developers can execute their code. Even a partially complete TLM platform can interest software developers because it can already help debugging their code up to a certain extent.

In brief, TLM can significantly alter the conventional manner of how a system-on-chip is constructed by creating more positive interactions between hardware and software fellows. A veritable hardware/software co-design will therefore be achieved through TLM approach.

Another appealing advantage of TLM is the cost. The number of a given TLM platform can be multiplied as many as the host machines that it can use for running. Consequently, the number of software developers being able to use this particular TLM platform is potentially unlimited at a given time. This advantage can rarely be provided by a typical hardware prototype such as emulator due to the cost issues. Naturally, more engineers will be able to work on a SoC project in its early design phase based on TLM platforms.

Figure 4-3 illustrates the time phases of the TLM-oriented hardware and software development. During the development of the untimed TLM hardware platform, a huge functional part of software programs can be developed. Once the untimed platform is ready, software designers can start testing the written software on this platform. Certain time-level features can be added to the software codes based on the untimed platform. Through observing the software execution on the untimed platform, we can improve not only the codes but also the untimed platform. Meanwhile, hardware designers continue their work in conceiving the timed TLM hardware platform. Once it is done, the further developed version of software codes will be executed and tested on the timed platform. Based on the timing information on the timed platform, software designers can further develop the software for the hard timing parts. Such software execution helps not only to improve the software code but also the timed hardware platform. At the same time, hardware designers keep on their job to conceive RTL hardware platform. Note that as the RTL hardware is ready, the software will have already been well tested on untimed and timed TLM platforms. Such "almost-final" software applications will be able to run quickly on the RTL platforms to reveal some hidden stubborn bugs.

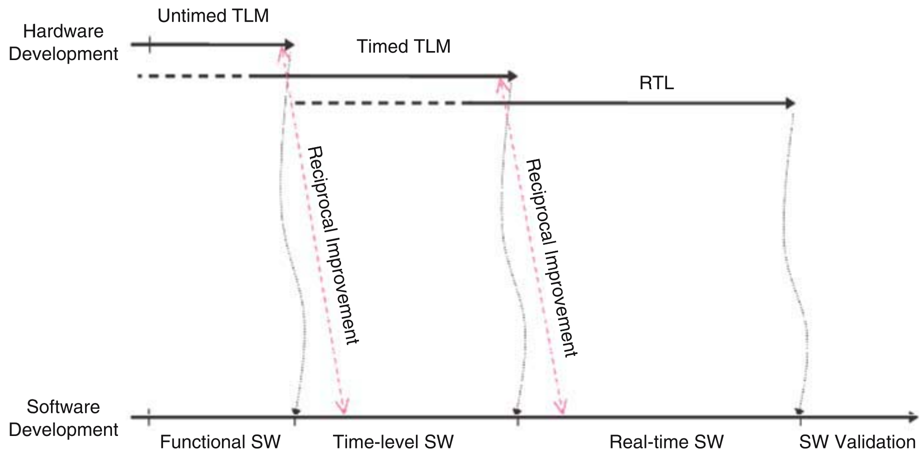

Figure 4-3. Time Phases of TLM-oriented HW/SW Development

The interactive design between hardware and software teams enhances the whole system design by visualizing their work to each other in a transparent manner. Hardware designers can observe how the software program utilizes the hardware platform while software designers can see how the hardware platform reacts to the software execution.

2.4.3 Illustration of Software on TLM Platforms

After reviewing various aspects of the relationship between software and TLM platforms, it is worth our time to discuss in details about the development of different software families based on TLM platforms in the rest of this chapter. The discussion will lay emphases on the objectives of using TLM platforms, TLM-based development and execution approaches along with illustrations of practical examples.

From an architectural point of view, software can be arbitrarily split into three layers as depicted in Figure 4-4. Each layer has a particular relationship with the hardware, and thus with TLM platforms.

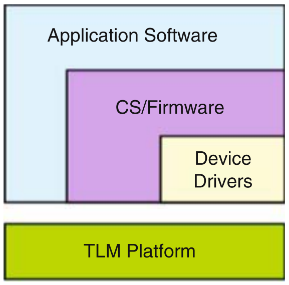

Figure 4-4. Software Families Developed on TLM Platform

3. TLM-ORIENTED DEVICE DRIVERS

3.1 Introduction to Device Driver

Device driver is the closest software level to TLM platform as shown in Figure 4-4 earlier. The key role of device drivers is to abstract low-level peripheral details to represent a generic programmable interface comprising a number of predefined functions. Device drivers should be the only entity accessing peripheral resources such as registers or shared memory.

A common method of accessing peripherals is via register accesses. Usually, registers are gathered into a unique I/O memory area reserved for specific IP accesses. Since their behavior is peripheral-dependent, register accesses must be correctly implemented in TLM platforms with the accurate functions. Another way of accessing peripherals is by means of shared memory. A memory zone is reserved in TLM platforms for data exchanges between peripherals and device drivers. Such data exchanges are performed within the structures defined by the peripherals.

3.2 Purposes of TLM in Device Driver Development

3.2.1 Unit Test Development

One of the very fundamental purposes of device drivers is to develop unit tests for a given IP on a TLM platform. Such device drivers run simple tests to assure the proper implementation of platform IPs. The degree of correctness tested by them depends on the types of the underlying TLM platforms, for instance, it is out of scope to test timing issues of an IP on the untimed TLM platform.

A device driver may cover more than a single IP if a DMA is coupled with the IP-under-test. In that case, the DMA will be tested as well but only for its interactions with that particular IP-under-test, i.e. the device driver can only conduct partial DMA testing.

3.2.2 Non-Regression Test Development

Device drivers can be developed as simple software for performing non-regression tests on TLM platforms. In the early phases of TLM IP development, it is vital to run device drivers on the TLM models to verify their correctness. As the design develops gradually into TLM models and becomes more complex, running the existing device drivers can be considered as a good non-regression test suite, which can verify that the additional new features work properly without distorting the old features.

Non-regression tests are usually totally independent of whether there is an embedded processor or not within the platform. Thus, the same tests are portable on the different platforms integrating the same IPs. This is a great advantage to validate quickly the reutilization of IPs on various platforms. The only characteristics to modify from one platform to another will be the base I/O address and the interrupt mapping.

3.2.3 OS/Firmware Device Driver Development

The term “device driver” is indeed derived from the semantics of OS/Firmware. It represents a piece of software developed specifically to be inserted into another piece of software that is more complex, i.e. the OS/Firmware itself. The purpose of this extra software piece is to isolate low-level management of IPs in an independent module with some externalized interface.

Device drivers serving for such purpose do not run alone as in the two previous cases, but rather in an environment with some constraints that will impose a particular way to use IPs. These constraints enforce a conventional manner of software coding, which may potentially improve the way that hardware is programmed.

Despite some attempts to standardize the interfaces, device drivers are usually not portable from one OS/Firmware to another; hence leading to different ways of using a given hardware.

3.2.4 Experimentation of New Hardware Features

Another interesting purpose of device drivers is to exercise new hardware features for experimenting their different aspects such as programming ease, performance improvement, programming examples, etc. Such experiments can be quickly set up on TLM platforms to test tiny modifications on the hardware before the real alterations.

Since device drivers are final software pieces of larger models like OS/Firmware, they can be modified independently from the rest of the whole system to include new hardware features. It is therefore very easy to rapidly set up a model for hardware developers to build their intended design, and subsequently exercise this new design under a realistic software execution.

As a result, hardware developers are able to verify the correctness as well as the resulting effects of their tentative design under the real scenario of software run.

3.3 Approach to Device Driver Development

This section focuses on the different approaches to developing device drivers. General rules of writing software targeted for TLM platforms are presented in section 2.1. For device driver software, the methodology of layered software development remains valid for its development and testing. Some additional aspects that deserve special attention will be explained extensively in this section.

3.3.1 Interrupt vs Polling Management

Reporting occurrences of interrupt events within an IP is normally managed by setting a particular bit of the IP status registers. Optionally, the IP may forward a signal to an interrupt controller that will in turn monitor the CPU interrupt line. From the angle of software, there are two methods of managing IP events:

1. *Synchronous Programming*. Polling (i.e. reading continuously) the bit reserved for interrupt in the status register until the right value is obtained.
2. *Asynchronous Programming*. The software executes standard procedures. Under interrupt occurrences, it is diverted to execute a handler that has been previously associated to the interrupt. At the end of the handler, it will simply continue execution of the procedure at the point it has been interrupted.

The two methods are not really independent in TLM platforms. The first method issues a TLM transaction whenever the status register is read or accessed. It thus induces a lot of overhead especially if the interrupt event takes quite some simulation time to occur. This is the appropriate choice for coding interrupts in unit test software because there is normally no other software running than unit testing.

Waiting for an interrupt as described in the second method is very close to the real situation on the real platform. It leaves no impact on TLM platforms because the IP will initiate a transaction when the real interrupt is routed to the interrupt controller. This method is suitable for testing the interrupt mechanism of a system. It is particularly useful for device drivers as they need to continue other tasks while waiting for the interrupt event to occur.

The software should take into account some unexpected behavior that could probably be induced by TLM platforms. One of the common examples is the approximate timing estimated by the untimed TLM platform, which delays certain event occurrences. Asynchronous interrupt programming assumes that the hardware will notify event occurrences with sufficient delay, which allows the software to perform some useful job while waiting in background for the interrupt. The consequence is that the software may not be able to do anything or even spend more time than expected in the interrupt handler if the interrupt occurs too quickly. The same problem, however, will not arise in the timed TLM platform since the timing is an absolute reference, i.e. events setting an interrupt will consume the required time amount before their occurrences.

Therefore, polling should be applied as much as possible in the untimed TLM platform instead of asynchronous interrupts. However, if the asynchronous interrupt modeling is required on untimed TLM platforms, interrupts must be expected to occur at any time. They can even occur in the same instruction of the I/O that starts an interrupt, which can rarely happen in the real life.

3.3.2 Time Management

Unlike interrupt controllers, certain IPs such as real-time clock, watchdog or timer deal directly with time management. The software written for such IPs must be aware of the time events like time-slicing, time-out or time-count for running on TLM platforms.

Timing is locally accurate on the untimed TLM platform. From the software point of view, events are locally ordered within a given IP. Consider the following example: a given IP with two timers programmed for sending interrupts at different dates will always send interrupt events in the well-coordinated order. Now, consider two distinct IPs with a timer in each. Even if the two timers are programmed in the same manner, IP events could occur in any order because both IPs are completely independent from each other with unspecified relative timing approximation.

Luckily, it is quite uncommon to depend on the relative timings between different IPs to run a software program correctly. For instance, time-out is usually implemented on top of a timer by the software so that it can be ordered continuously. Although the simulated time difference remains unpredictable, it brings no problem since the software usually relies on time order but not on time difference.

Testing scope is quite restrictive for timing aspects on the untimed TLM platforms as the timing accuracy is not really measurable. Low-level design is thus reduced to validating the functions of interrupt and status indicators. The timed TLM platform, however, offers larger capabilities in terms of timing testing.

Another important point on timing is time-slicing. When a dummy or buggy C program executes a "`for(;;) continue;`" sequence, it will keep looping forever. It is always possible to stop this loop by sending an interrupt, e.g. character typed or time-out, which can divert the execution from the loop and eventually stop the loop. Running such programs on an ISS is well handled by TLM platforms. The untimed TLM platform manages this program by advancing its timeline from time to time, even if the ISS does not require any I/O on the IPs. For example, the time progression can take place when the ISS runs an I/O access; the internal SystemC scheduler can then be called freely to move forward the timeline. For the timed TLM platform, the rule is much stricter since timing accuracy is required. The ISS must access the internal SystemC scheduler (even for nothing) in order to let other IPs running their codes at the right scheduled time.

The approach is totally different for natively compiled applications as they are integrated into the execution environment of TLM platforms. Bear in mind that TLM threads are *non-preemptive*. If any thread happens to loop, no other thread can preempt it from looping and the TLM simulation will just loop forever. To let other threads run, a special thread layer such as OS emulation can be of great help by simulating multiple OS threads within the same SystemC thread. An alternative solution is inserting some calls to the internal "`sc wait()`" function at the right locations. This function will essentially give a chance to the system to progress its simulation. Such situation is one of the very few circumstances where the software must cooperate directly with the TLM platform.

To conclude, software running on the TLM platform, especially when natively-compiled, must be capable of handling unpredictable time management.

3.3.3 Performance-Accelerating Hardware Features

It is a general comment that TLM platforms do not simulate hardware fast enough. Although this is always a personal perception, such moderated simulation speed might actually be very useful to detect some problems that may appear unobvious on fast-simulating platforms such as the real hardware platform.

As a matter of fact, the reduced simulation speed is frequently a “bug amplifier”. A subtle bug occurring for a very short time period could probably be invisible during the simulation on the hardware platform. The same bug, however, may turn into a disaster in a TLM simulation and thus much easier to be detected and fixed.

Suppose that a driver for a slow-communication IP does not use a DMA correctly. The real hardware platform works so fast that it may conceal this problem on regular uses. The problem can only be revealed by an integration test where other IPs are involved to use the slow-communication IP intensively. The system will give an abnormal response time that serves as an indicator of such problem. A TLM simulation, on the other hand, shows the abnormal response time immediately because such problem will give the character-by-character output (1 character per transaction) instead of the message-by-message output (N characters per transaction) that should normally be provided by the DMA use. Since the overhead of a transaction is not negligible, a unit test is usually sufficient to uncover the poor programming of the DMA.

The similar problem can be encountered in cache programming. If the cache is badly used or unused, the number of accesses to the TLM memory will be unacceptably high. Software developers will consequently notice a bus overhead rapidly, and thus identify a cache-related bug.

Therefore, the moderated simulation speed on TLM platform provides users with an early detection of misused features. This is extremely helpful for revealing those directly related to the overall system performance but hidden in a small local area for a long time. Indeed, these are very tough features to detect because they appear functionally correct. It is the reason why some software programmers may not see the advantage of TLM “bug amplifier” right in the beginning. Once they get more acquainted with TLM, they will definitely find this characteristic rewarding.

3.3.4 Peripheral Error Management

Another critical piece of device driver software is the management of peripheral errors. In common practices, this software piece is only ranked as secondary level of importance because the priority is always given to programming the regular peripheral uses. Unfortunately, the quality of a low-level code like device driver is not in the regular working parts, but rather in the error management and recovery.

Through modifying specific values in the setting, debuggers are used to “set up” and reproduce an error to facilitate the analysis of a particular fault in details. This method, however, will get a little cumbersome when an error comes directly from hardware devices. Too many registers will have to be set up in debuggers for such bugs. Some manual intervention or script-writing in debuggers is even required for certain cases. Consequently, such errors become extremely difficult to regenerate or reproduce “correctly and accurately”, for instance, in non-regression tests.

Let us consider the error management of the Ethernet controller. Under normal working conditions, the Ethernet driver is not in charge of any errors. For high system load, the driver must nonetheless face plenty of severe conditions such as input errors, buffer underflow, out-of-buffer, etc. Under these conditions, the driver may decide to reinitialize the Ethernet controller while a simple recovery procedure could be sufficient. This technique works most of the time but it may result in catastrophic performance consequences. For this reason, it cannot give good quality software although it functions correctly.

Such hardware-related error management is a real pain for software developers. It consumes much time in understanding and coding yet brings too little visible functionality to the software. Most of all, testing errors that practically never occur in a real system is too huge a challenge. Hardware developers do have hardware devices to reproduce specific errors easily. However, these devices may not be available for software developers. Even if particular hardware test sequences can be set up, they will not be suitable for software error management.

TLM platforms are sound solutions for handling peripheral error management in device drivers. Software developers can simply inject data from the external world into the platform IPs to reproduce specific IP hardware errors. This error injection helps to test the behavior of device drivers when the error actually appears. Since the error is managed by the software, error sequences can be produced in the IPs as many times as required for running the error testing at high level of confidence.

3.3.5 Native Compilation

Native compilation is the fastest TLM simulation system for software. Although irresistibly attractive, it must nevertheless be employed with meticulous care for a number of potential pitfalls. In particular, software codes must respect the underlying restrictions rooted in the fact that the software is link-editable with TLM platform codes, i.e. TLM platform codes will be embedded together with software for running.

The most obvious restriction is the non-exclusive use of shared resources. Software must never “monopolize” common resources shared with TLM platforms such as heap memory, signals, file descriptors, etc. For instance, the signal handlers from software codes should never replace but add onto those already existing in TLM platform codes; in the same sense, the allocation order of file descriptions should never be deduced from the one of the underlying OS algorithm.

Software codes must never be based on libraries or software compilation tools that are incompatible with those required by the TLM platform codes. A simple example can be illustrated by GCC compiler. It is well known that the GCC-2.95 release is incompatible with the GCC-3.x release for C++ programs due to changing of name mangling algorithm. If a software program compiled with GCC-2.95 is link-edited with TLM platform codes compiled with GCC-3.1, the link-edit will fail indicating that a problem exists or worse, the link-edit will seemingly succeed but the execution will crash without any obvious reason.

In the same line of idea, another interesting point is dealing with threads. Threads used in a software program must be compatible with those used in TLM platform; besides, they must respect the reentrancy programming constraints of TLM platforms. In other words, only threads compatible with SystemC runtime are allowed for TLM-oriented software codes because TLM platforms are based on SystemC runtime. For example, only one OS thread is permissible in the OSCI runtime, which restricts uses of SystemC threads and those simulated within a SystemC thread. In addition, the software thread scheduling has to be compatible with the one used in SystemC. The reason is that SystemC functions are not required to be implemented as reentrant; for instance, the current thread scheduling of OSCI runtime is neither reentrant nor thread-safe.

Debugging natively compiled software is much more complex as it is based on the SystemC runtime. There are two major difficulties. First, software developers may perceive codes out of their control, i.e. TLM platform procedures called when their own code access IP registers. Stack frames can be quite confusing as well because it may not be easy to locate the frames at the exact spot where the software really starts. Second, software developers may not see all of their threads if their codes are multi-threaded. The reason is that their threads are embedded in SystemC threads, which may not be visible to debuggers, e.g. the current case for OSCI runtime. Today, debuggers are not much adapted yet for certain non-standard environments such as multi-thread wherein hardware and software simulations are mixed.

Despite all these pitfalls, most of our low-level software runs perfectly well in the native execution environment. In fact, such pitfalls or constraints appear mostly in very high-level software that will be discussed later on. In a nutshell, native compilation is a simple method to start working out low-level codes. It also assists in rising code portability because the same code should run in cross compilation as well where no such constraints apparently exist.

3.4 Examples of TLM-oriented Device Drivers

Without any practical examples, all the approaches described earlier could probably be too theoretical to digest. Let us zoom in on the details of some low-level software already running on TLM platforms through our development work.

3.4.1 SPI Controller Test

The Synchronous Peripheral Interface (SPI) is a very popular protocol widely used in the industrial environments to enable data exchange between a micro-controller and an external peripheral. Instead of plugging a given peripheral directly on a system bus, it is much easier to connect them through a serial interface whose major advantage is the reduction of communication pins. The SPI protocol is founded on the data exchange initiated by a master to a slave at a clock rate determined by the master itself. At each clock signal, the slave must be ready to receive a bit and send out another.

SPI controller tests involve two strictly distinct parts: testing SPI master and/or slave. Data exchange is the principal of testing SPI controller. A fixed set of data must be provided to the SPI controller for exchanging between the master and slave sides, the aim of which is to validate the SPI behavior.

Let us take a closer look at testing an SPI master role (SPI slave role will have a similar testing line). In such test, no SPI slave device is utilized. Instead, it is replaced by a file containing data to be exchanged with the SPI master. The SPI master is exercised by software actions; it also receives the input data from another file holding exactly what it expects to receive. When the software sends a data item such as a byte or something larger to the SPI master, the TLM IP of SPI master controller will read the next data item potentially being sent from the data file representing the SPI slave device. The data read from the latter will then be placed in the registers of SPI master as if it was received in the real situation. Depending on how the software is programmed, the TLM SPI master may update its registers after storing this data.

By comparing the data received from both master and slave sides (more precisely, from their respective data files), the tests of sending/receiving SPI data are carefully conducted by the TLM SPI master controller. Complex data exchanges can certainly be set up, for instance, those including DMA or end-of-transmission interrupt. The validation of SPI data exchanges helps to justify not only the correct functioning of the IP, but also helps to verify the right software programming of the IP registers. With a successful validation of SPI data exchanges, the same software should result in the same test behavior on the real hardware IP that is available later (provided that the SPI slave is correctly simulated by the data file).

Testing a given IP is unfortunately not only limited to its functional tests especially when the IP is synchronized with the external world. In particular, it is impossible to test if the clock programming fits in as required by the slave since the test is not timed. The IP test set will be incomplete if there is no synchronization between the master and slave. If the master acts too fast, the slave will not be able to respond in time. However, the master will still sample the data line coming from the slave to deduce the value transmitted by the slave. This deduction could be incorrect if the timing is wrong. Although the timing programming may be validated statically on an untimed TLM platform, this may not be sufficient.

For that reason, there are two conditions to fully test an SPI IP. First, a timed TLM platform is most of the time compulsory. Second, a mechanism allowing the simulated slave to analyze the timed master responses is required for validating the correct timing of the master. This example illustrates how and when different TLM platform implementations should be employed for various purposes.

3.4.2 I^{2}C Controller Test

The inter-integrated circuit (I^{2}C) bus is a bi-directional two-wire serial bus providing a communication link between integrated circuits. The main difference of I^{2}C from SPI is that I^{2}C supports multi-master mode: I^{2}C allows multiple master devices to connect on the same bus to start the communication at the same time. The collision is resolved electrically, and only one master remains the master of the communication.

I^{2}C is much more complex than SPI. SPI slave is equivalent to SPI master except for the clock generation, whereas I^{2}C slave and I^{2}C master exchange control information such as address, acknowledge, and start/stop right on the bus. This is a sophisticated feature needing software for testing.

For this reason, it is essential to have at least two devices on the bus for testing I^2C: a master and a slave. TLM platforms normally provide a single I^2C controller that represents a single device on the bus, which supports either multi-master mode or exclusive master-or-slave mode. Unlike SPI controller, the second I^2C device cannot be replaced by a file because the control information exchanged on the bus will not be tested. The simplest solution is therefore setting up another I^2C controller on TLM platforms *exclusively* for testing. It generates and validates the required bus control information, and its mapping is done on unused addresses.

Once the two devices are properly set up, the I^2C controller test can start from any mode. The major challenge of such test is to synchronize the test software precisely between two similar collaborative IPs. While sending information from one of the IPs, another IP must be controlled for its correct receiving of whatever previously sent by the first IP. Polling is not a good tactic because both IPs must be polled concurrently, but interrupts from either IP could arise in any order.

The testing schema describe above is insufficient to test all I^2C features. For instance, the feature of *master arbitration lost*[1] in the multi-master mode can only be tested when both IPs agree to set up the same testing condition. Then again, this set up cannot be done by regular register I/O. The same problem may arise in testing all communication errors such as non-acknowledge testing.

Just like SPI controller test, I^2C controller test is capable of testing many interesting functional features of the IP before the hardware is ready. It helps to show that the platform runs correctly under standard conditions. This is essentially the first step towards getting a validated TLM platform for running real software programs, particularly real device driver codes.

3.4.3 PrimeXsys UART Linux Driver

The ultimate goal of TLM platforms is *not* developing test software for IPs, but running actual device drivers that will be used on the real hardware platform. Certainly, all events especially errors cannot be triggered to occur as exactly as under real-life conditions. They will however be tested under standard conditions, thus representing the actual behavior most of the time.

An excellent illustration for this concept is the behavior analysis of a real device driver running on a TLM platform. Theoretically, a device driver should run correctly without any modification in the cross-compilation

[1] A feature with an I/O starts functioning as a master, but the I/O will become a slave when another master wins the exclusive access to I^2C bus (Arbitration).

mode. Let us study the Linux device driver for the UART on the ARM PrimeXsys platform. This driver is initially compiled with the rest of Linux for the ARM PrimeXsys platform. It is then booted on an untimed TLM platform without any modification but some additional error messages for testing purposes.

The experiment shows several interesting side effects. First, the output of messages is very slow. The usage message of *ls* command takes longer than the booting of Linux kernel to display. A closer examination reveals that the DMA is not configured by default for the UART even if the codes are identical. The driver is functionally correct except that an important feature, i.e. DMA, is missing. Although coded, this missing part is not visible enough on the real ARM PrimeXsys platform. In contrast, TLM platforms manage to "amplify" this problem because the missing DMA changes the behavior of TLM platforms significantly. Software using buffered C runtime stdio output routines such as printf, putc, and puts, running on TLM platforms with DMA display a message per transaction while those without DMA () display a character at a time.

Once the problem is fixed, the DMA is enabled in the platform. Yet, the performance still does not show the expected results. All messages are output very quickly but a noticeable delay occurs between usage messages of the Linux *ls* command. Another problem is then identified in the TLM code of UART IP. The added delays in the output for simulating the programmed UART baud rate actually slow down the entire simulation process of UART driver, the reason of which is the ARM platform has nothing else to do but displaying messages. By removing these delays, the UART driver can finally give satisfactory performance results. Indeed, this "discovery" is interesting because time characteristics must be simulated (even on untimed platforms) but with flexible and careful adaptation.

To sum up, running the UART Linux driver on the ARM PrimeXsys platform demonstrates the following benefits of TLM platforms:

1. Device drivers can run without any modification in cross-compilation.
2. Missing performance features can be detected without any special tests on TLM platforms (e.g. DMA).
3. Poorly coded software is immediately revealed by running real software on TLM platforms.

3.4.4 Native Device Driver

Device drivers are not only coded for running in the cross-compilation mode. Running them in the native mode can be equally beneficial for software developers as long as certain coding rules are well respected.

Performance and code portability are the two chief advantages. Software programs can run much faster in the native mode than in the cross-compilation mode, hence higher performance. They can also be compiled on a machine with different constraints such as data alignment, byte order, and language basic types to increase the code portability.

Embedding the software codes of TLM platforms is a very distinctive characteristic of the native environment. It obliges the respect of the host execution rules, including reserved addresses, dynamic loading, name space pollution, etc. These obligations can be very tough barriers to deal with when developing huge software pieces. It is not straightforward to execute both software programs and TLM platforms nicely in the same environment. Beware that all these problems may arise during a project, although huge amounts of code have already been ported in such environment.

The first rule for developing native device drivers is to facilitate the contact point between software codes and TLM platforms. It is usually achieved by using IP register accesses. Such contact in cross-compilation is simply the simulation of a foreign instruction at a given I/O address, which is translated by the ISS into a TLM transaction. TLM platforms just need to issue the I/O and the corresponding results will be given back to the software by the ISS during the simulation of the same instruction. The software in native compilation, conversely, must issue the required TLM transaction by itself. Therefore, the compilation of IP register accesses will need to be transformed into a TLM transaction at the lowest software cost.

Wrapping in macro register accesses is recommended as a good software coding practice. It is particularly useful for increasing software portability onto those systems needing special instructions to access I/O spaces such as I386. Such macro wrapping facilitates the definition of a separate set of macros for the native mode, hence leading to highly portable software. The macro wrapping cannot be applied if register accesses are coded as mapped address dereferences. The reason is that the address range in this case is more likely forbidden to be used in the host execution environment. The initial solution is thus code modification, which may entail additional time delay in the software development exclusively for native compilation.

Another point of attention is accessing the memory shared between the software and hardware for data representation. When everything is ready to be analyzed in an IP, the hardware normally expects to download from the shared memory some data that is already formatted by the software. A good example is a DMA scatter/gather list. Such data representation is a real complex problem because:

1. The simulated hardware may have different byte ordering from the host system.

2. The simulated hardware may align data in a way incompatible with the alignment rules of the host system.
3. The software may use different data types for cross and native compilations, e.g. the "long" type of C language.

There are several good practical rules applicable to solving this situation nicely, which are similar to those used for IP register accesses:

a) Always use fixed length data types so that the field length definitions are not ambiguous, e.g. "int32_t" type of C-99 language.
b) Always access shared data with macros that can be redefined correctly in case of incompatible byte ordering or alignment.

Name conflict is a much tougher problem to solve. Fortunately, it is not something that happens very frequently. This sort of conflict occurs when the software uses an external name that is already defined by the TLM platform. By some chance, the link-editor may detect this as an error. There is however a slight risk that the link-editor may merge it quietly at the same location in the common data segment. That will very likely lead to concurrent use of the same memory location by two modules without relationship. It is then easy to imagine the kind of errors provoked by this bogus situation. Such problems can arise either in static or dynamic link-edit where search results of external names are often hidden by high-level functions..

Sadly, there are not too many solutions for this problem. To cope with it, avoid such naming conflicts by prefixing (or using different name spaces) the external names and minimize using global variables. Name conflict is not a problem directly related to TLM platforms, but it is often encountered by software designers developing huge software pieces.

Concisely, a very important point here is that TLM imposes good software coding practices to prevent some tricky problems from happening during the earliest stage of the development.

4. TLM-ORIENTED OS/FIRMWARE

4.1 Introduction to OS/Firmware

Recall Figure 4-4 shown previously, the software family located above device drivers covers OS (Operating System) and Firmware.

OS is a higher-level software family responsible for integrating all lower-level software pieces to set up a coherent view of the hardware management. Such responsibility is generally entitled to Operating System or Executive

Runtime, which presents a programming interface to higher-level applications.

A key difference exists: Operating System shares CPU time between a large variable set of tasks that are scheduled only when they have something to do, whereas Executive Runtime shares CPU time between a small fixed set of tasks that are called at regular intervals for testing event occurrences and performing potential job. A common point between them is the ability to manage the conflicts of resource accesses for an optimal use of the hardware platform.

Operating System is essentially a piece of complex software for managing task preemption and switching, hardware interrupt dispatches, collaboration between low-level device drivers, etc. The task management role of Operating System is even more distinct when it provides real time functions. Executive Runtime, on the contrary, is considered as a simple task scheduler that neither has potential preemptions between tasks nor interrupt handling; and it has virtually no overhead for task switching. Both of them are of course relatively far from each other in terms of functionality, but they will be considered and described collectively as a single entity called *OS* to cover the two task areas aforesaid.

Another group of interesting software in this family is *Firmware*. It is the software piece responsible for driving some processing parts embedded on the hardware platform. Firmware usually receives and manages specific jobs from an external entity. It therefore plays a mid-level role by unloading some jobs that can be managed locally from the CPU. Such role can be endorsed by running some software on a digital signal processing (DSP) unit to control certain IPs directly for high-level data exchanges with the CPU.

4.2 Purposes of TLM in OS/Firmware Development

Using TLM platforms throughout the development of OS/Firmware serves an important objective: it integrates all lower-level device drivers and executes them in parallel to detect software (potentially hardware) problems related to the interactions of multiple data flows. Such problem detection is either a direct mode by sharing a device driver between two data flows, or an indirect mode where the activities on a data flow prevents another data flow from being correctly managed.

4.2.1 Integration Test Suite Development

As discussed earlier, unit tests are developed for testing a given IP individually (two IPs are required occasionally). Such tests are much limited to testing a single IP without testing the rest of the platform IPs simultaneously. If all unit tests of a platform are pulled together, it is possible to set up an integration test suite provided that the unit tests are developed to run as either standalone tests or concurrent tests in common with other tests.

Therefore, it could be quite easy to set up an integration test suite based on the available unit tests. For a given IP, all of its unit tests can be serialized to form a set of unit tests reserved for this IP. Such unit test set for all IPs can then be run collectively at the same time to exercise all IPs concurrently, hence leading to a proper integration test suite.

This approach, nevertheless, is not that straightforward for untimed TLM platforms. Due to their untimed characteristic, setting an I/O transfer via a register write is virtually immediate. Thus, no I/O parallelism can have effect in the set of IPs under test. Moreover, an interrupt will be triggered as soon as a register access is completed if the interrupt mode for I/O completion signaling is used. It is possible to chain all these tests in order to run them one after another. Yet, this is still not quite a real integration test suite.

The missing part is an executive runtime that can run one test after another. It should avoid running the entire test set of a given IP right after running the entire test set of another IP. A better way to handle this is running interleaved tests in order to exercise all IPs more frequently.

On the contrary, an integration test suite can be set up very nicely on timed TLM platforms. The reason is that each I/O consumes some time to signal its completion, and that consequently allows running multiple tests for different IPs in pseudo-parallelism. The term “pseudo” signifies the fact that I/O completion time is accurate but its progression is without cycle accuracy.

Integration tests should also be in charge of testing arbitration, i.e. hardware conflict resolution. The typical examples of such conflict are two concurrent interrupts or I/O bus accesses. Interrupt conflicts can be validated on timed TLM platforms whereas bus access conflicts can only be tested on BCA platforms.

In brief, integration tests are merely some test set on untimed TLM platforms; they provide much more interesting results on timed TLM platforms; and finally give solid outcome on BCA platforms. Therefore, TLM platforms should be considered as initial test platforms where integrations tests are executed and debugged before other further accurate platforms are made available.

4.2.2 Quick Application Software Evaluation

An attraction for using TLM platforms in OS/Firmware development is the *quick* set up of a complete system integrating hardware and OS, which aims at evaluating external higher-level application software. Throughout a new SoC development, it is always a challenging mission to validate if the design meets the software requirements until the software really runs on it. The early availability of TLM platforms allows software developers to get a precise image of the final hardware platform for running software ahead of time.

Today, the standardization of interfaces and low-level services such as Windows, POSIX, OSEK, and iTRON has facilitated the implementation of such interfaces on top of a number of Operating Systems. As a result, higher-level software can be ported much easier from one platform to another. Combined with TLM platforms, these interfaces and low-level services offer high-level software developers a complete system that is ready to support high-level software. Depending on their levels of accuracy, TLM platforms serve extensively as evaluation systems that support major OS available today.

A complete system consisting of OS, device drivers, and TLM hardware platforms is essentially the very first integrated system accessible to high-level software developers. Such a complete system holds several important characteristics as follows:

1. It is not restricted for large deployment since it is purely software; the number of systems available for using is thus not limited to just a few fragile hardware boards.
2. It implements a realistic system platform whose accuracy depends on the accuracy of TLM components and the software integrated.
3. It provides a platform with a coherent behavior of all integrated platform parts, i.e. both hardware and software.

4.2.3 Closed Integrated Software Module

During the development of OS/Firmware, TLM platforms also serve the purpose of employing black box tests and pure binary software codes.

TLM platforms are established as accurate representations of some existing or upcoming real platforms Thus, they must support binary software codes intended for running on the real platforms in a transparent and reliable manner. Software developers count a lot on this feature to prove not only the accuracy of TLM platforms, but also the correctness of their software with respect to the real platforms.

Note that this particular characteristic is only necessary for cross-compiled platforms because it is worthless to set up binary software for some platforms that will never exist. A prototype can hardly provide the same feature described here because it will never be accurate enough to run binary software codes as a black box test. This is also hindered by other reasons such as netlists, definitions of IP registers, component timings, etc.

To conclude, running pure binary software codes on TLM platforms have two ultimate goals:

1. Validate the TLM platforms when the real hardware already exists with the same software.
2. Validate the software provided as binary codes with some extensions, which are developed for supporting additional hardware features on a new platform compatible with some existing ones.

4.3 Approach to OS/Firmware Development

OS or Firmware developed for running on TLM platforms are not directly related to the hardware simulation. They should consequently be less sensitive to certain low-level details implemented in TLM platforms. Bear in mind that OS/Firmware is a special software layer responsible for the employment policy of the mechanisms defined by lower software layers. This is exactly where meticulous care must be taken to handle the capabilities of TLM platforms correctly. The approaches to developing OS/Firmware mainly focus on how to get a complete hardware/software system to run efficiently without wasting simulation performance in useless tasks.

4.3.1 Active Waiting Loop Avoidance

The foremost software quality is being able to utilize the underlying hardware at the optimal level. This is nevertheless quite a tricky game to play with on a simulation platform such as TLM. Since the hardware parts of a simulation platform are merely some simulated models with various accuracies, certain programming techniques may appear inefficient in the simulation run although they may work reasonably on the real hardware. The main reason is that some trade-off between performance and accuracy must be made on simulation platforms.

The active waiting loop is an example of the programming techniques difficult to be adapted on simulation platforms. On the real platform, such software will loop perpetually through a list of awaited events until one of the events finally occurs. The same software technique runs in the same way on TLM platforms but less efficiently, as the hardware event will only occur when the software allows it to occur. The software run indeed prevents the parallel run of TLM platforms, which is supposed to raise the subsequent event waited by the software.

This situation is similar to a deadlock but not fatal because the software might be preempted sometimes. Rather, active waiting loops are considered unproductive since the system cannot evolve during their execution. Keep in mind that events are driven by the hardware speed on the real hardware platform for which the real time is continuous, whereas events are driven by *no* previous activity on TLM platforms for which the simulated time is discrete.

Therefore, it will be wise to have useless software suspended until the next hardware event occurrence. The software should then loop again until it finds the occurred event. Since such software loop is sensitive to hardware events, it is not easy to program it transparently for either the real or TLM platforms.

Actually, new requirements for low-power consumption on SoC have helped to solve this tricky problem. Under this concept, any software with nothing constructive to perform will simply switch to the low-power mode to wait for the next event. Such switches are handled by some hardware interactions on TLM platforms, which can subsequently advance the system to the following event in line. Essentially, this is what will really occur on the real and TLM platforms. The low-power feature therefore avoids the active waiting loops and enables the same binary software to be used on both platforms with equal efficiency. Without the problem of active waiting loops, TLM platforms are once again ready to drive software towards the better use of the underlying hardware.

4.3.2 Hardware Interrupt Management

Interrupt management is another problem similar to the active waiting loop. Normally, starting an interrupt I/O on the real platform takes some time. The software will not just wait for the completion of the I/O event by doing nothing. Instead, it will try to perform some other useful jobs while the hardware processes the I/O work. This is feasible for the software only if the hardware takes long enough to notify interrupt events. As discussed earlier in the interrupt management for device drivers, receiving interrupts too early could be unfavorable as the software may not be able to perform other useful jobs or may even spend more time in handling the interrupts. This is particularly true for the management of input device events arriving at unknown (or very high) frequency.

Looking at the whole picture of a system design, a given software should run equally well either on the real or TLM platforms. Similarly, polling and interrupt modes implemented in the software should be valid for both platforms despite their different performance behavior.

Recall that OS/Firmware involves more than a single device driver. There is a potential side effect of using immediate interrupt notices. It may happen that the software reads data from a hardware source that always acquires ready to-be-consumed data. On the real and timed TLM platforms, acquiring data consumes some time that the software can run other tasks while reading data. However, on untimed TLM platforms, it occurs that interruptions related to I/O completion are somewhat instantaneous following their respective I/O activation, which may lead to a kind of apparent execution starvation for other processes. Polling is therefore the better solution for handling interrupts on untimed platforms.

To prevent such misbehavior that the software cannot avoid by itself, certain safeguards must be provided in TLM platforms. These safeguards will serve as the guidelines to standardize the software codes running on different platforms, because software designers develop their software based on the TLM platforms provided to them.

4.3.3 Native vs Cross Execution Environments

Native versus cross software compilations are discussed in sections 3.3.5 and 3.4.4 to describe the compilation nature of low-level software like device drivers. Regarding higher-level software such as OS/Firmware, software developers must consider carefully some different behaviors introduced by its execution environment.

Performance is always much faster on a native platform than on a cross platform. This is the reason why most of the SoC developers tend to use native compilations for their TLM platforms. Such high performance, however, is not as easy to reach on OS/Firmware layers due to certain processor-specific features that are tough to cope with; among them the most noticeable ones are the virtual memory management, the code and/or data cache management, the execution and interrupt paths, which are clearly under the responsibility of OS/Firmware.

Another important concern for the execution environment is the strategy of software debugging. On the real hardware, software developers debug by either plugging in additional hardware pieces to control program execution or embedding software debuggers in the OS/Firmware codes. The most common solution is based on the Joint Test Action Group (JTAG) and some hardware extensions such as In-Circuit Emulator (ICE) or User Debugging Interface (UDI) for a complete execution control, while the latter is based on the low-level CPU control.

JTAG and other hardware-based solutions are not practical for TLM platforms because they are too dependent on the hardware availability. On one hand, embedded software debuggers can only be supported in cross-compiled environments, but on the other hand, they can benefit from the precise debugging information via a debugging server that can stop ISS at the granularity of an instruction. This is relatively much easier compared to the real platform on which software debuggers must trap current execution flow on the real processor by exception, hence causing an intrusive debugging process. The direct impact of these differences is that the debugging strategies might not be similar between the real and TLM platforms. For native platforms, the only debugging solution is using host debuggers with some proper adaptations as explained earlier in section 3.4.4.

4.3.4 Virtual Memory Management

An important software aspect dedicated to OS/Firmware management is the Virtual Memory Management, namely the handling of the mapping between physical pages (the ones actually in memory), and their association in the standard addressing schema. This is usually achieved either via a Memory Management Unit (MMU) or via a TLB Translation Look-aside Buffer (TLB), internal to the CPU.

Under the cross-execution, the ISS is responsible for such management, and the TLM platform only receives accesses to physical addresses. This is however a costly management as each memory access must be translated from virtual to physical addressing space, which relies on search tables. The main advantage is that the accuracy of the management is high, and that all MMU-related traps are notified to the OS.

Since performance is the main problem of memory management, then it could be useful to take native compilation into consideration. The main problem now is to get the maximum possible accuracy in order to be in a simulation environment providing enough realism for software development.

The first thing to remember is that the embedded OS and firmware usually have nothing to do with on-demand page swapping (i.e. the ability to put unused memory pages on secondary memory such as disk drives). The memory management is restricted to the ability of running multiple applications within a finite amount of memory, which is more of a memory placement problem rather than virtual memory management problem: if all of the applications are not able to fit into the available memory, then the system cannot work safely.

This is the reason why for OS/Firmware coming with source code, the most efficient approach seems to replace virtual memory management by the dynamic placement of all the processes in the virtual memory of the TLM platform execution. The main advantage is the performance gain, as there is no overload of translating virtual addresses to physical ones (the code and data addresses are relative to a register and allocated memory areas being directly accessed). The main drawbacks, on the other hand, are the non-protection of a process data against erroneous accesses (by itself or by other processes).

This solution can be applied with Linux, for which µClinux is the right and most efficient solution. The reason is that OS provides the same API not only to user-mode applications, but also to dynamically loadable kernel modules such as device drivers, file systems, etc. Such pieces of software may be almost completely and transparently tested and debugged within a MMU-less environment, and then integrated into a cross-integrated TLM platform with complete MMU support for final validation.

4.4 Examples of TLM-Oriented OS/Firmware

The current section provides a brief description on some OS/Firmware examples already running on TLM platforms through our development work. These practical examples illustrate how some approaches described earlier are used.

4.4.1 Quick Setup of Integration Test Suite

As soon as a TLM platform is set up, the immediate subsequent step is running unit tests for every IP of a platform. These unit tests must be conducted IP per IP to evaluate the correct functioning of each IP. Once the first step succeeds, the next step is running integration tests to validate the integration of all IPs in the TLM platform. This step is particularly important for validating the areas that necessitate arbitration for handling concurrent accesses to shared resources such as bus or interrupt controller.

Since parallelization is required, an integration test must be written to deal with more than a single IP at a time. Moreover, integration tests need a test harness to optimize the pipelined execution of all IP unit tests and to exercise all IPs in parallel on a given platform. The test harness referred to here is indeed an Executive Runtime that schedules each task to manage a single IP. It may also include interrupt management to validate the right functioning of the shared interrupts. Another advantage of the Executive Runtime is its ability to handle multiple IP tests such as the I^2C controller test described in section 3.4.2. Test codes for each platform IP is placed in a different task scheduled by the Executive Runtime in order to retain the paradigm of managing an IP per task.

A test set should be arranged to complete properly because an Executive Runtime does not really stop by itself. It is vital to ensure that there is definitely nothing else to execute in order to terminate the integration test correctly. A good practice here is to let the underlying SystemC runtime "discover" the simulation completion by having no more events to handle. In this way, the simulation will exit without returning to the software; hence avoid having the Executive Runtime to deal with the test termination.

4.4.2 OS on ARM PrimeXsys Wireless Platform

The ARM PrimeXsys Wireless Platform (PWP) is an extendable development platform whose description is open to the public[2]. Software developers can understand this complex platform much better by running a simulation on the TLM platform of ARM PWP.

Booting an OS on PWP aims at verifying the software behavior on both simulation and real platforms. It could be really challenging to boot an OS binary image on a cross-compiled platform if a bug appears in the OS code execution. The core architecture assembler is the only accessible debugging level, which is unfortunately not so obvious to get information from. Bugs found in this type of simulations are usually related to some subtle behavior differences between the real and simulated hardware. If the source codes are unavailable, tracking such bugs becomes much tougher.

The first right approach is to run an OS whose source codes are available, for instance, running Linux on PWP. In general, an ISS-embedded debugger is aware of the OS object format and thus capable of conducting symbolic debugging. If the debugger recognizes the structure of the internal threads, the whole OS execution structure can be exposed through the debugger interface. As a result, the debugging process becomes much easier to control. This process is a little more difficult in the case where MMU is handled by Linux because the debugger must understand the virtual memory translation. In this case, it is easier to start running the platform with uCLinux, i.e. the Linux without MMU port. Once the PWP TLM platform is extensively exercised by this OS, it is time to get some binary OS such as Windows or Symbian running on the platform.

Booting OS on TLM platforms is advantageous. It gets ready a complete platform for software developers to design high-level software. In addition, it helps to find missing or incorrect OS codes that are virtually invisible on

[2] Available at http://www.arm.com/products/solutions/PrimeXsysPlatforms.html

native platforms. A real-life example is the public code of PrimeXsys Linux for the ARM926EJ-S platform. By comparing the TLM simulation to the one of real platform, two errors are found in the public code. We notice that the bootstraps for cache enabling as well as the UART accesses to DMA are too slow on TLM platforms while they appear apparently correct on the real platform. This example shows the direct advantage of TLM simulations.

4.4.3 Native OS Emulation on Video Platform

The video platform is an interesting platform type to learn more about TLM concepts. As the video flood flows from one IP to another, the central processor only acts as a director without a straight view of the video flood. IPs are generally complex hardware pieces that run some firmware on top of an internal micro-controller.

Simulating precisely all IPs on a TLM video platform is not quite feasible because not all IP models are made available quick enough for software developers. It is possible to build accurate TLM models of these complex IPs and integrate them as a single TLM platform. The resulted performance, however, is very likely to be slow. Bear in mind that one of the primary goals of video platforms is the pipelining of the video flood. Video components are therefore designed to run in parallel. Each of them must also run its associated code accurately, and that will consume many CPU resources. Consequently, the sum of such consumption will result in a very slow platform.

Running software on a simulated platform is nevertheless still very important for software debugging. A different approach must be adopted to cope with the problem of slow simulations. The proposed idea is splitting a given platform into simpler but meaningful hardware pieces for software development.

In other words, if the software perceives a specific IP block and its associated firmware as a black box, then a single TLM model is conceived to represent the whole IP block including the associated firmware. Sometimes, software developers may need to debug both software and firmware on the same platform. In that case, software developers must set up the platform in such a way that the internal IPs and their associated firmware are exposed.

The benefit of such platform setup is the close simulation of the IPs whose associated software requires debugging, while maintaining the overall platform performance at an acceptable level through keeping other IPs as efficient black boxes.

Native emulation is the suitable solution for working with the firmware of complex IPs because it retains both performance and functional accuracy of IPs. In addition, debugging the whole platform in native emulation allows grabbing an image of the OS running on the central processor and the firmware running on the IPs of interest. Therefore, it is feasible to analyze the interactions between all the software running on the different IPs at the same time, or even in the same debugger session.

Having TLM models as black boxes of IP hardware and their associated firmware may have some impact on the OS drivers managing these IPs. If the firmware is encoding/decoding a well-established algorithm such as MP3 or MPEG4, it will be quite straightforward to set up an IP model running the similar code but with different interfaces. There are two solutions for the OS driver:

1. adapting the driver to the simulation interface, i.e. a fast solution;
2. adapting the simulation software to the IP interface, i.e. an accurate solution.

Both solutions are valuable approaches that should be applied at different phases of the SoC design process, depending on the expectation of the software development. The accurate solution, however, should be retained once it is available because it allows debugging the actual OS driver.

5. TLM-ORIENTED APPLICATION SOFTWARE

5.1 Introduction to Application Software

Although the two lower-level software families i.e. device drivers and OS/Firmware bring interesting results, the ultimate goal of software developers is to get the final application running on TLM platforms as soon as possible. This high level is probably the easiest to set up on TLM platforms because it has very little or no relationship with the actual hardware programming.

Building and running a complete application on TLM platforms is an ambitious objective to achieve. In common practices, applications normally run on a prototype of hardware board whose central SoC is only a part of the board. Today, the implementation cost of hardware prototypes is skyrocketing due to the explosive SoC complexity. Running applications on a much more complete platform like TLM is therefore getting increasingly attractive for SoC developers. The TLM platform is not only a hardware platform that is accurate enough and available significantly earlier, but it is also able to easily outperform the equivalent RTL platform.

By running the critical software parts interacting frequently with the lower-level software, software developers should gain enough confidence that the application is ready to be integrated as soon as the prototype of the final platform is available. The critical software parts already layered and packaged as independent libraries such as protocol stacks, data stream decoders, all sorts of data filters, graphical environments are among the most essential parts to be verified in this manner.

5.2 Purposes of TLM in Application Development

5.2.1 Final Validation Test Development

Once integration tests are completed, validation tests can be started. A validation test exercises a given platform in the environment that it is specifically designed for running in. The principal idea is to define a set of representative test scenarios intended for the final platform execution, i.e. the highest level of tests that a platform must go through for its validation.

Validation tests are crucial in assuring the accuracy of the integration of TLM platforms. These tests aim at demonstrating that TLM platforms behave exactly as they are designed. If TLM platforms show their high fidelity to the hardware platforms, the same validation tests should provide the same level of confidence on the real hardware platforms.

Since accuracy is a characteristic that becomes more critical close to the end of SoC development, a validation test will emphasize more on the accuracy of a platform than on its performance. For this reason, running validation tests can sometimes be very time consuming. The focus of validation tests is on the whole software rather than on the TLM platform because the objective is to try out the TLM platform in the real environment.

It is vital to have validation tests running in a simulation environment that is as accurate as possible in order to test the internal behavior of the platform. Thus, TLM platforms must interact with the external world at the highest possible accuracy. This requirement is less strict for the previous two lower-level software families because their main purpose, i.e. testing TLM platforms, is different from the one of validation tests for high-level software. As such, validation tests can be considered as part of the interoperability tests.

5.2.2 Performance Experiment

While the OS/Firmware is sufficient to demonstrate that a platform is functional by itself, the application software is intended for providing additional feedback from TLM platforms. More precisely, it places TLM platforms in a real application environment wherein non-functional results can be obtained from the whole system.

Such non-functional results can be considered altogether as performance results, covering timed profiling (e.g. latency, speed, throughput, deadline), untimed profiling (e.g. counter, contention, bottleneck), and other factors such as power consumption estimate, security evaluation, fault tolerance, resource footprint, etc. Without TLM platforms, it is almost impossible to assess these performance factors accurately since the whole application will not run in a realistic environment.

Running application software on TLM platforms is a good occasion to conduct a brief benchmarking for the internal behavior of a platform or an IP, i.e. behavior due to the fine-grain impact from certain hardware features on the platform. Transactional analyses in this realistic benchmarking serve as accurate sources for significant decision-making in SoC development. To perform such accurate simulations, timed TLM platforms are compulsory.

5.2.3 Impressive Demonstration

Running high-level software is not only useful for hardware and software developers, but it is also valuable for other professionals involved in the SoC project. A lively demonstration of a given application can be surprisingly rewarding for marketing crew as well as final users.

An impressive demonstration of the whole system is the real foundation of communication for marketing crew. The demonstration is intended not only to prove the platform compliance with the requirements, but also to illustrate the impact of pulling all the requirements into the entire system. Final users, on the other hand, can start validating if the platform fits their design requirements without waiting for the first prototype. If the result is negative, there is still ample of time margin to modify the platform before the real hardware advances. After all, TLM platforms are simply some software pieces that can be easily altered.

5.3 Approach to Application Software Development

This section provides some general advices to develop application software revolving around TLM methodology.

5.3.1 Provision of Realistic Environments

Developing a specific application targeted at an embedded system on a real chip is quite a different matter from developing the same application for running on a workstation or application server. The reason is that the embedded system is a *real* environment with plenty of constraints in terms of computing resources, bounded memory size and less powerful CPU. On the other hand, the workstation or application server protects the software developers from this realistic environment with their own software environment. Such *unreal* environment may tolerate certain programming pitfalls that could become threatening on the real embedded system. Therefore, application developers must be aware of this potential discrepancy and design their software tailored for the optimal use of the computing resources.

A significant part of the application software can be developed natively on workstations or servers without TLM platforms. Although well fitted on this untargeted foreign environment, the resulted software often triggers a disaster on the target platform. Several reasons can explain this situation. In general, the larger the SoC project, the more difficult the anticipations will be. Collected below are among the most common examples:

1. non-executable application due to the excessive memory consumption by the application itself on the target system;
2. system crash due to some leakage in the resource consumption;
3. inefficient codes due to a slower processor;
4. deadlocks of multi-threaded applications due to different scheduling.

Looking at all these potential problems, it is absolutely critical to get ready a realistic environment for embedded applications as soon as possible. Through a realistic environment, such problems can be detected in the early phases of the application software design ranging from development to execution and simulation. TLM platforms offer a 3-in-1 solution that covers the three environments for bringing realistic effects, i.e. development, execution, and simulation environments.

The *development* environment is the set of applications and libraries necessary for building and debugging a given embedded software. Its early use within the application design process enables the early detections of compilation problems, missing target library functionalities, and resulted image size to be loaded on the target platform.

The *execution* environment is the set of resources involved in the embedded software execution. This is the heart of what TLM platforms have to offer to software developers. Running applications on such platforms give software developers a realistic idea of any potential execution problems hidden in their applications.

The *simulation* environment is the set of external devices connected to TLM platforms. Its mission is to provide software developers with a realistic external simulation that will exercise the whole system hardware and software. Essentially, this environment simulates the application in the setting that it expects to run in. Such simulation is important for observing the realistic aspects of the embedded system that depend very much on the data exchanges with the external world.

5.3.2 Attention to Application Performance

The amount of software to be run at the application level is huge. It is very likely to run on many loosely coupled IPs and split into a central application software with multiple firmwares around. Therefore, performance becomes a core issue for using TLM platforms in order to extract quickly interesting results. If the accurate timing is one of the major expected results, native compilation is of little help.

Running multiple loosely coupled software pieces in parallel provides software developers with the ability to parallelize them easily. Although platform IPs may seem independent from the angle of software, TLM platforms impose a certain level of serialization among them because they interact altogether at the hardware level. Consequently, a very accurate TLM platform from the hardware perspective may become much less efficient in running an application software. It is therefore extremely important to choose the accuracy level of each TLM component very carefully for a system integration.

Nevertheless, keep in mind that the host processor is usually much more powerful than the simulated hardware (for the majority of the simulated IPs except the central processor). The global performance of the TLM platform thus correlates with the ISS performance. Such correlation is also true for IPs running a firmware on a little embedded micro-controller turning at low speed. If the TLM platform embeds an ISS for this micro-controller, then the ISS performance will be high thanks to a simple code emulation as well as the high relative speed between the micro-controller and the host processor. Watch out: the firmware running on this IP could be an active loop waiting for an event to be raised by another IP, and that may cause the system temporarily doing nothing constructive for the platform evolution. Of course, this is another matter on a timed TLM platform as the relative performance of IPs are already taken into consideration.

To conclude, the challenge of running an entire application on a platform may turn out to be much tougher than expected. The challenge has no close relationship with the hardware, but rather in validating the integration of all software pieces. It may lead to coherent yet antagonistic directions from the point of view of application performance.

5.3.3 Control on TLM-Specific Code Amount

TLM and real platforms provide software developers with different but complementary advantages. TLM platforms, however, are not set up to run huge applications due to a costly price to pay for the high accuracy, i.e. low performance. Yet, there is still a real interest to run huge software slowly on

TLM platforms: obtain a level of internal observability that is almost impossible to get from a real platform.

When a TLM platform is stopped, all of the platform components are stopped at the same time in the same state even though they are not tightly coupled. This is a very advantageous situation to examine the component states in details. Such convenience is not easily reachable on a real platform because everything is fit into a chip, which is only accessible through some internal complex and indirect tracing mechanisms such as JTAG.

Software developers must refrain from writing too many software specific to TLM platforms. This can be helpful at the early stages of a project in order to make use of native compilation. However, it becomes less and less useful as the project advances because the platform will normally be more and more complete while adding new functionalities. Having such code at late phases might be mainly for catching some subtle bugs. It will not be reused on the final hardware and thus can be considered worthless to be developed.

Bear in mind that TLM platforms are pure software. If the real hardware is available, it will certainly be more efficient to debug an entire application on it than on the TLM platform (provided that lower software layers are sufficiently debugged beforehand on the TLM platform). Although subtle bugs can be discovered faster on the real hardware, TLM platforms still merit a vital role at this stage to continue testing certain software parts separately from the whole system view.

In brief, software developers must *wisely* determine the software amount developed particularly for running high-level software on TLM platforms, knowing that it is preferable to target final applications on the real platforms for the reasons of debugging efficiency and code reusability.

5.4 Examples of TLM-Oriented Application Software

Typical applications running on TLM platforms are those interoperating different IPs through an embedded processor or micro-controller. Particular applications running on a single IP can be interesting for demonstration but not really valuable for debugging.

5.4.1 Multi-Processor Platform Application

A multi-processor SoC platform (MPSoC) is a platform that embeds more than a single processor of the *same* type on which the system workload is distributed. Other main parts on MPSoC platforms include communication channels and potential managers to handle the multiple processors.

MPSoC are extremely complex platforms with more than one CPU to run the software that manages the whole platform, which consequently leads to a distributed management model. Such hardware implementations could be too complex to understand for software developers. Rather, they should try to comprehend the TLM model of the platform and figure out how to get it run. That will focus software developers on the problems related in running an application on these platforms, instead of understanding how the platforms work.

Communication is another interesting point to employ TLM models of MPSoC platforms. The reason is that the complex software running on MPSoC platforms always tries to split its workload on all platform resources for an optimal utilization. It is therefore fundamental to have an excellent control over the communication and data exchanges in order to master such platforms. The unified view and global fine-grain control of each processor through TLM platforms allow software developers to retrieve and analyze the MPSoC platform behavior easily. These analyses can be accomplished without overlooking the subtle platform management normally handled by software such as cache coherence with DMA and shared memory.

Typical MPSoC platform applications are generic parallel applications in which various tasks such as graphical encoding/decoding, network routing, scientific computations can be executed *indifferently* on any processor. The goal of using TLM platforms for such applications is not really running the computations, but rather validating through small examples that the workload is well distributed among all of the processors. It is easier to conduct this sort of validation on the slow but accurate TLM platforms than on the real hardware whose activities are much more difficult to control.

5.4.2 Centralized Multi-Architecture Platform Application

A multi-architecture platform denotes a platform that embeds multiple processors of *different* types. The overall workload is distributed among all of the processors but these processors, unlike MPSoC platforms, do not play the same role. Software must be split into pieces dedicated to each type of processors.

Multi-architecture platforms are usually called for applications dealing with the parallel handling of multiple data flows whose management is centralized on one or multiple processors of the *same* type. Examples for these platforms include telephone, set-top box, and complex audio platforms.

Since each processor is of different type on multi-architecture platforms, heterogeneity and synchronization are unsurprisingly the most difficult parts to master. Indeed, communications in multi-processor platforms can be considered as a particular case of communications in multi-architecture platforms. High-level management applications are responsible for optimal distribution of the instantaneous workload to the right processor at the right timing. TLM platforms provide a real advantage by giving a high-level view of different software pieces running on different processors.

The DSP plays a particular role as a multi-architecture platform. DSPs are not general-purpose processors but they can be continuously reloaded with a new program to perform a new function required at a given time. Debugging DSPs is exceptionally complex because the software running on some of the processors changes constantly. This characteristic could be a potential source of bugs that is not easy to detect and fix. To handle the communication between a CPU and a DSP, software developers need to obtain a coherent view of the software distributed on heterogeneous processors all the time.

5.4.3 Pipelined Multi-Architecture Platform Application

The pipelined multi-architecture platform is another kind of multi-architecture platforms. This is a platform whose IPs are integrated in such a way that a data flow will stream from a specific IP to another specific IP through the whole platform.

The role of the CPU on a pipelined multi-architecture platform is similar to the one of the centralized one, i.e. it organizes the data flow and manages the external events. Typical platform examples are multi-media decoders.

Getting ready the application software as soon as possible for such platforms enables the flow design validation and the early revealing of any potential bottlenecks that may threaten the overall platform performance. These platforms usually consist of multiple IPs with micro-controllers that run a firmware not modifiable until the next platform reset. The overall software is split into multiple pieces that are loaded into their own target IP processors, which are independent from each other except for their synchronization.

TLM platforms can simulate this type of complex platforms for validating not only the IP-dependent software individually, but also the overall application that subsequently allows careful checking of certain platform aspects during debugging process.

6. CONCLUSION

Running software on TLM platforms leads to splitting software in three layered categories that correspond to the three principal software-testing layers: unit test, integration test, and validation test. For each layer, TLM platforms offer software developers the simulated platform that they need for executing and debugging their software. The choice of the accuracy level of TLM platforms is crucial, as the software will appear to run less efficiently on a more accurate platform. Figure 4-5 recalls the idea of relating different software families and environments in the V-diagram of software testing.

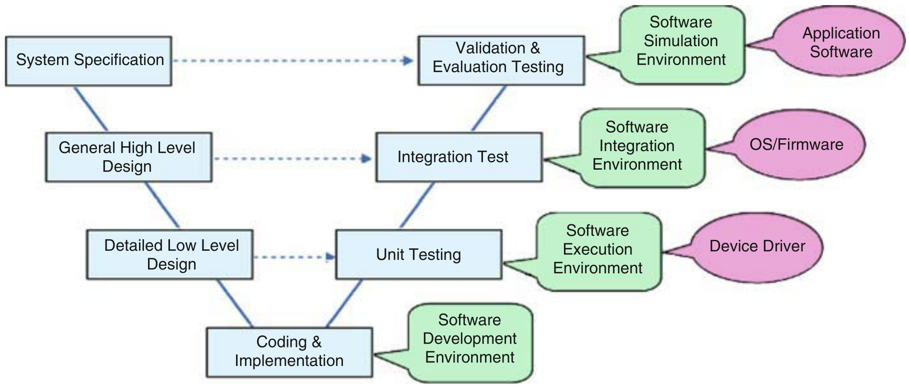
System Specification
Validation & Evaluation Testing
Software Simulation Environment
Application Software
General High Level Design
Integration Test
Software Integration Environment
OS/Firmware
Detailed Low Level Design
Unit Testing
Software Execution Environment
Device Driver
Coding & Implementation
Software Development Environment

Figure 4-5. Software Families and Environments vs Software Testing

Developing a complete system based on TLM platforms can significantly improve the methodology and the schedule of the hardware and software design. Software developers are able to test their codes on simulated but accurate hardware platform long before the real hardware prototype is ready. Even after the real hardware is made available, TLM platforms continue to bring software developers important information that is not obtainable from the real hardware. Essentially, TLM platforms play the central role in hardware/software development as a common exchange platform between these two development teams.

TLM platforms are also considered as the software bug amplifiers. They reveal a more general behavior of the hardware, which is normally not easily accessible to software developers. This advantage reinforces good software practices for software families ranging from low-level codes to software architecture layering.

In conclusion, TLM platforms shorten the global time-to-market and raise the overall quality of SoC projects. Uncovering software and hardware bugs long before the real platform availability considerably trims down the cost of bug fixing. Last-minute patches can then be avoided most of the time, giving software developers more time to work on performance issues.