Multiprocessor System-on-Chip Prototyping Using Dynamic Binary Translation

18

Frédéric Pétrot, Luc Michel, and Clément Deschamps

Abstract

Dynamic binary translation is a processor emulation technology that allows to execute in a very efficient manner a binary program for an instruction-set architecture A on a processor having instruction-set architecture B. This chapter starts by giving a rapid overview of the dynamic binary translation process and its peculiarities. Then, it focuses on the support for SIMD instruction and the translation for VLIW architectures, which bring upfront new challenges for this technology. Next, it shows how the translation process can be enhanced by the insertion of instructions to monitor nonfunctional metrics, with the aim of giving, for instance, timing or power consumption estimations. Finally, it details how it can be integrated within virtual prototyping platforms, looking in particular at the synchronization issues.

Acronyms

DBT	Dynamic Binary Translation
ILP	Instruction-Level Parallelism
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
MMU	Memory Management Unit
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multi-Processing
SSA	Static Single Assignment

F. Pétrot (✉)
Université de Grenoble Alpes, Grenoble, France
e-mail: frederic.petrot@univ-grenoble-alpes.fr

L. Michel • C. Deschamps
Antfield SAS, Grenoble, France
e-mail: luc.michel@antfield.fr; clement.deschamps@antfield.fr

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_20

TB	Translation Block
TLM	Transaction-Level Model
VLIW	Very Long Instruction Word
VP	Virtual Prototype
WAR	Write-After-Read

Contents

18.1 Introduction

Virtual Prototype (VP) serve different purposes, and depending on these purposes, the acceptable "accuracy *vs* speed of simulation" trade-off is very different. More than two decades ago, Transaction-Level Model (TLM) was introduced as a way to abstract time and data and clearly decouple computations from communications. Compared to Register Transfer Level (RTL) which focuses on implementation and targets on accuracy, TLM aims at giving a high-level view of the system so that it is possible to quickly take design decisions. What has been intuited at that time is now recognized as an actual solution for early software development and design space exploration of hardware/software systems [5]. The system-on-chip industry has seen the value of having models sitting in between RTL and fully analytical formulas and has adopted this kind of modeling strategy quite rapidly [15].

In the context of an ever-increasing number of programmable cores in integrated devices, the simulation of the software part of a system becomes a critical issue. Even though TLM is well suited for hardware design, it says nothing about the way the software that runs on the hardware part of the system should be executed. Several strategies can be thought of, ranging from interpretive instruction-set simulation of the cross-compiled target binary code to the native execution of host compiled code using the Operating System (OS) calls as simulation callbacks [24]. Figure 18.1 summarizes the main strategies used for software execution on top of transaction-level hardware.

The three first software execution approaches can be classified as interpretive ones.

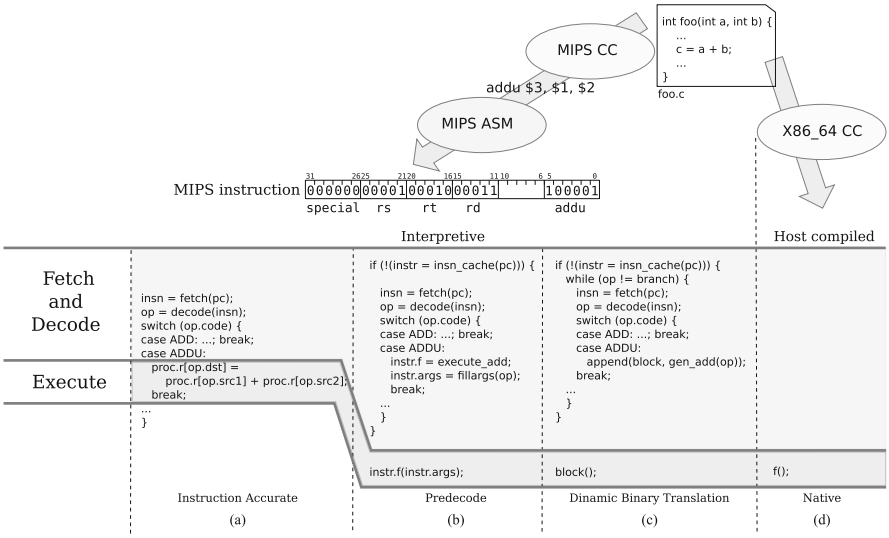

Fig. 18.1 Major software execution strategies in TLM environments

Instruction accurate interpretation (Fig. 18.1a) is the textbook method for executing cross-compiled code. Each instruction is read from memory, decoded, and executed. Decoding can be done using a big `switch`-like control structure or a functions pointer table [4].

Predecode (Fig. 18.1b) is based on the same principle, but the instructions are fetched and decoded, and placed in a cache in which they are identified by their program counter (`pc`) [22]. Then, if the `pc` matches a cache entry, the instruction is directly executed through a function call. Using a decode cache brings up new issues as it must be kept up to date when some code region is modified. Some applications heavily rely on dynamic compilation, so it is necessary to handle it with efficiency. Incidentally, this cache can be the actual simulation model of the processor instruction cache, as the high hit rate it usually has ensures that few fetch/decode will be redone without necessity. Furthermore, provided the simulated instruction caches ensure coherency, multiprocessor systems work out of the box with this approach.

Dynamic Binary Translation (DBT) pushes the limits further by fetching and decoding an entire sequence of code ended by a branch at once, translating it into host code with identical behavior, and caching the result [11]. The whole sequence is then executed atomically, avoiding to check per instruction the presence of the `pc` in the cache. As a translation cache is used, the same issues as predecode arise, but then using the model of an instruction cache is not possible as there is no such thing as a single target instruction in DBT.

The last approach, introduced here only for completeness, takes a fully different angle, as the high-level code to be executed is directly compiled for the host. Many

different approaches have been proposed, as detailed in ▸ Chap. 19, "Host-Compiled Simulation" of this book. Anyhow, these native or host-compiled approaches need to access the hardware and, at some point, rely on an operating system or hardware abstraction layer API to implicitly let the simulator execute the hardware models. A clear limitation of these strategies is that it is hardly possible to handle self-modifying code or dynamic code generation; however, not all applications need it.

A current trend in system-level simulation is to use DBT to execute target code and TLM to model hardware [1, 16, 23]. Indeed, a desirable goal is to define a framework which provides a way to virtually prototype full hardware/software multiprocessor systems with an entire software stack, including the operating systems and the device drivers. As this requires to execute the cross-compiled binary code of all software layers at high speed, dynamic binary translation is the more suitable software execution technique.

On the one hand, modern DBT engines take their root in virtualization, an effort that took place in computer science to make possible the execution of several OS in isolation concurrently on the same processor [28], based on the *virtual machine* technology developed in the early 1960s [7].

On the other hand, due to the constraints of consumer system integration (form factor, packaging, power, heat, etc.), each application or class of application still calls for a specific circuit in order to fit into the performance/power budget. The Multi-Processor System-on-Chip (MPSoC) design approaches aim firstly at clearly separating computation from communication, using interfaces that are standardized, allowing to quickly exchange one IP, including processors, by another. Secondly, they aim at producing figures of merits, such as code sequence run times and interrupt latencies, used bandwidth on an interconnect, even energy or power information, depending of the architectural choices. Due to the increase in number of programmable cores and software in the near to come SoCs, the availability of virtual platforms providing a structural view of the system and fast application and OS code execution with a reliable accuracy is an important issue.

If modularity is of primary importance in MPSoC design, it is not the case for virtualization which targets a single one-shot hardware platform with an as high as possible execution speed. Although the goals of the hardware/software cosimulation and virtualization may seem very different, at the end of the day the way to achieve these different goals are similar.

The rest of the chapter is devoted to clarifying the necessary points to build an operational optimized MPSoC simulator which makes use of DBT as software execution engine.

18.2 Dynamic Binary Translation Basics

Full virtualization allows the execution of an operating system, called guest, on top of another operation system, called host, without any modification. One of the main issues in virtualization is the execution of the privileged guest operating

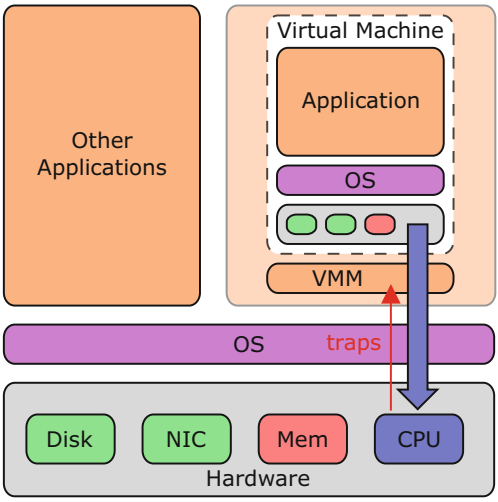

Fig. 18.2 Trap and emulate based virtualization

system instructions since the simulator in running on the host operating system as an unprivileged application. Another delicate issue is the interception of I/O operations of guest operating system. The classical solution for full virtualization, called trap-and-emulate, is presented Fig. 18.2. It assumes the direct execution of the simulated machine binary code on the host processor; therefore, the host processor and the target processor (the one on which the guest operating system is run) must be identical. This solution is based on a virtual machine monitor (VMM) which takes control whenever a trap caused by a privileged operation executed in the unprivileged context of the virtual machine occurs. However, not all processors are fully virtualizable [25], typically because some privileged instructions executed in user mode do not trap. To work around this problem, the binary translation approach using DBT, as depicted Fig. 18.3, can be used. This approach has no constraints concerning the host and target processor types, as all instructions of the guest operating system binary code, including the privileged ones, are replaced by unprivileged instructions and/or system calls.

Apart from virtualization, historically DBT has also been used for transparently running legacy software compiled for a processor on either a new version of this processor or an entirely different processor. Apple used this technology when transitioning from the PowerPC architecture to the x86 one (Rosetta by Transitive Technologies, now acquired by IBM). When Intel introduced the IA64, DBT from x86 code was also applied [2]. While the former case translates from scalar to scalar architectures, the latter one does a scalar to Very Long Instruction Word (VLIW) architecture translation, which requires a more complex process.

Figure 18.4 presents the general principle of binary-translation-based simulators. The simulator starts by verifying if the current `pc` of the simulated processor has already been encountered. If not, the *binary translation* stage begins.

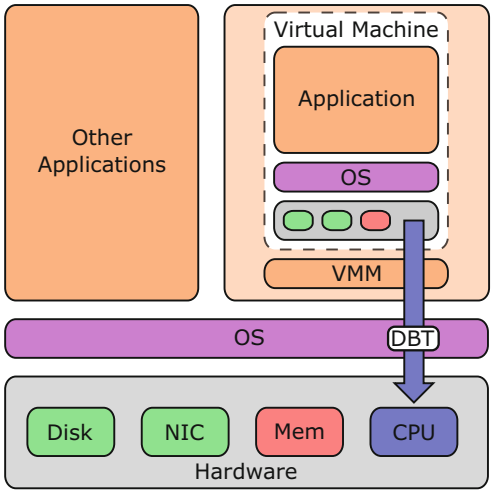

Fig. 18.3 Dynamic binary translation based virtualization

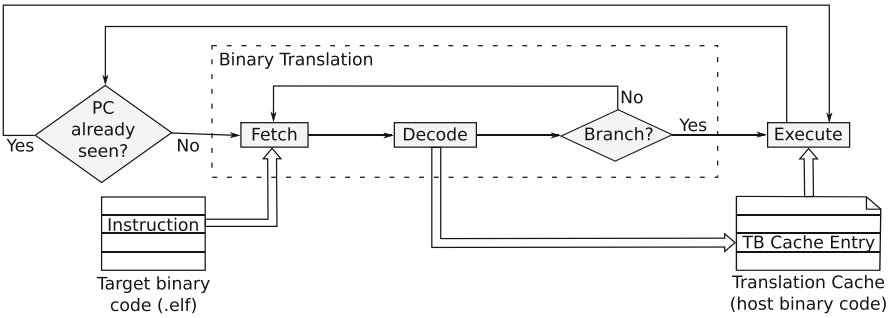

Fig. 18.4 Binary translation principle

The instruction corresponding to the program counter of the simulated processor is fetched from the target binary code. The fetched instruction is then translated into several host instructions. If the current instruction is not a branch instruction, the next target instruction is fetched and decoded. Otherwise, the binary translation stage ends. The sequence of instructions treated by the binary translation stage at once forms a Translation Block (TB). The host instructions generated for a translation block are grouped together and stored into the translation cache and are ready to be executed to simulate the corresponding target instructions behavior. Once executed, the simulator has stepped forward by one TB, and the simulated `pc` has evolved accordingly. The simulator then verifies the existence of the translation corresponding to this new program counter value. If it already exists in the translation cache, it is directly executed. The idea is that the price paid for host code generation will be amortized as the translated code sequences are usually executed many times.

The notion of translation block is similar in spirit to the notion of basic block used by compilers, but they are not identical. Even though translation blocks and basic blocks end after a branch instruction, there are other conditions that end only the translation blocks or only the basic blocks. As an example, a translation block is also ended at the page boundary of the target processor, because the DBT engine must ensure that the page is mapped in memory. Other conditions for ending a translation block include instructions generating exceptions (e.g., undefined instruction), change of the execution mode of the target processor, etc. By definition, the basic blocks are also ended before the instructions which are the target of jump or branch instructions.

A binary translation simulator would generate in this case a new translation block starting at the jumping address. So, an instruction can be part of several translation blocks, but only of a single basic block.

Given the simulation context we are interested in, it is required to be able to support different types of target processor architectures on different types of host architecture, even though simulation will very likely take place on a x86_64 machine. Given t target processor types and h host processor types, the approach just described leads to the development of $t \times h$ translators to allow all target processors to run on all the host processors.

Due to the complexity of writing a translator, the principle of retargetable DBT has been proposed [28, 29]. Instead of being translated directly to host code, each target instructions is first translated to a bytecode (also often called intermediate representation or IR) common to all targets. The virtual instructions of the bytecode, that we call *microoperations*, are then translated to host code. This way, target and host translations are independent. Adding a new target processor requires "only" a translator of the instruction set of that processor to the bytecode, no matter the number of hosts on which the new target processor can be simulated. By adding a translator from the bytecode to a new host processor, all target processors can be simulated on the new host processor. So, the number of translators is now $t + h$. The principle of these simulators, given Fig. 18.5, is close to that presented in Fig. 18.4. The target instruction decoder generates the bytecode corresponding to

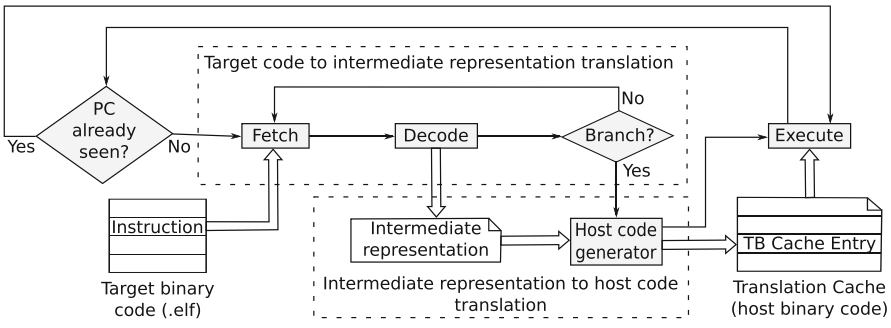

Fig. 18.5 Retargetable binary translation principle

Target code (MIPS)	Description	Generated micro-operations (QEMU)	Generated host code (x86_64)

`sltiu v1,v0,23`	v1 ← v0<23 ? 1 : 0	`mov_i32 tmp0,v0`
		`movi_i32 tmp1,$0x17`
		`setcond_i32 v1,tmp0,tmp1,ltu`
			`mov %ebx,0x74(%r14)`
			`cmp $0x17,%ebp`
			`setb %bpl`
			`movzbl %bpl,%ebp`
			`mov %ebp,0xc(%r14)`

Fig. 18.6 Microoperations and host code generated while translating a target instruction

the translation block under consideration. Then, the *host code generator* produces the host code corresponding to that translation block for the host processor.

Figure 18.6 is a simplified illustration of the translation process on a MIPS instruction, the intermediate bytecode is a 3-address Instruction-Set Architecture (ISA) close to the one of QEMU [12], and the target is x86_64. The generated bytecode uses a mix of processor architectural registers, e.g. `v1`, and of temporaries, here `tmp0` and `tmp1`. The architectural registers represent a part of the processor state and belong to a larger structure called the *environment* in QEMU. Their value survives between translation blocks, while temporaries live only inside a TB. Once translated as host code, these elements become offsets within the environment, pointed to by the host register `%r14` in QEMU for x86_64.

When performing translation, some instructions require complex operations that can hardly be produced by generating host instructions at run time. For example, when doing a load instruction, it is necessary to access the Memory Management Unit (MMU) model to determine whether the addressed page is mapped in memory. If not, a page fault exception must be raised. This leads to complex operations, like traversing the page table, which are done using a lot of code. Therefore, instead of generating the whole code dynamically (which is hardly possible and an overkill), the DBT engines generate calls to specific functions (called helpers) to handle these instructions.

Even though instruction interpretation is the core of dynamic binary translation, quite a lot of housekeeping is necessary to actually obtain a running simulator.

- First and foremost, as the guest OS expects a memory management unit, it is necessary to simulate it. It is also necessary to produce the expected page faults including the protection flags, etc. Page boundaries are handled at translation time; flags are handled on memory accesses.
- Second, self-modifying code must be supported. This is especially true as now, even in embedded environments, many just-in-time compilers are used (just think of Javascript in a browser or of Android). The performance issue related to supporting dynamic code generation in DBT is considered as of primary importance [17].
- Third, the translation cache has to be managed. A cache, by nature, has a finite size. The placement of the newly generated translation blocks and the replacement policy in case of overflow have to be determined.

- Fourth, multiprocessor systems must be supported. A basic implementation simply simulates the processors one after other in a predefined order by calling their execution function. The simulation of a processor is suspended, and the simulation of the next one (e.g. round-robin algorithm) resumed when an interrupt or an exception occurs. However these events can only occur at the border of a translation block. Other solutions can be thought of to better mimic parallelism, as explained in Sect. 18.5.
- Finally, many optimizations are possible. Chaining is a classical optimization which links a block to its successor using a jump instruction without going back to the DBT engine when possible. The identification of the mostly used paths, called hot paths, and the optimized retranslation of these paths can be beneficial. It can also be counterproductive, as it requires accounting in the translation blocks and time for optimized retranslations. Finding the right balance is known to be hard [12].

18.3 Support for Non-scalar Architectures

Dynamic binary translation is usually used for running target code for a scalar architecture on a scalar host. However, MPSoCs target specific markets with tight power and area constraints and therefore embed specialized processor extensions or processors with unusual architectures. The goal of this section is to briefly present how such features can be efficiently supported by DBT.

18.3.1 Support for SIMD Instructions

Single Instruction, Multiple Data (SIMD) instructions perform parallel operations on multiple data (Single Instruction, Multiple Data (SIMD)). There are today multiple ISA extensions providing SIMD instructions to general purpose CPUs. For performing parallel operations on multiple data, an SIMD instruction performs the operation (or sequence of operations) on registers interpreted as array of values. This array of data can have a variable number of values of various size, for example, a 128 bits wide register can be viewed as two 64 bits, four 32 bits, eight 16 bits, or sixteen 8 bits values. On top of that, the variety of the operations applied to the data is huge. It ranges from the classical arithmetic operation (add, sub, shift, ...) to saturated or rounding arithmetic. Among this large range of instructions, each SIMD instruction set represents a unique subset choice made by the processor architects. SIMD extensions can also integrate some exotic instructions such as polynomial multiplication or pixel distance computation that have no equivalent in other SIMD ISA.

DBT engines focus on efficiency for the most often used instructions, following the adage "make the common case fast and the uncommon case correct." As the SIMD extensions are rarely relevant in general purpose computing, most translators use helpers to execute their behavior even though all host processors include SIMD instructions.

People in companies developing processors [20] have looked at this issue as the number of different SIMD extensions for different processor generations (e.g., MMX, 3DNow!, SSE1, ..., AVX for Intel) is huge, so having legacy code being able to benefit from the most efficient version available on the actually running processor has a clear value for some applications [26].

Replacing the helpers by dynamically generated code which uses the host SIMD instructions opens the interesting question of the appropriate intermediate bytecode for representing SIMD instructions [21]. The main two constraints to which one may think for this bytecode are:

- limiting the number of new IR microoperations in order to limit the burden on the code generator and the overall performances of the binary translator,
- adding enough microoperations in the IR to allow a wide coverage of both target and host SIMD instruction sets.

Indeed, adding too many microoperations will tend to the addition of one microoperation per SIMD instruction. This will not solve the problem since the code generator (the second phase of DBT) will have a heavy work to do to produce the inlined optimized code for each of the SIMD microoperations. Conversely, if not enough microoperations are added, the semantic of all SIMD instructions will not be expressed, and the translator will have to perform this translation using many non-SIMD microinstructions. A simple way to extend the IR is to choose a set of microoperations which is close to the intersections of the more widely available SIMD instruction sets. The IR microoperations will be 3-address operations since it is the most general case and allows to represent the 2-address versions easily, whereas the reverse is not true.

As opposed to scalar DBT, finding instruction equivalence in SIMD DBT has to take care of the SIMD specificities: parallelism and register interpretation. The following section illustrates these peculiarities with concrete examples of translation from ARM NEON instruction set to Intel MMX/SSE.

Direct mapping between instructions: in the presence of an exact equivalence between a target SIMD instruction and an host SIMD instruction, the behavior of the SIMD DBT is quite similar to the one of the scalar DBT. This case can be called a *direct mapping*. The main difference between scalar and SIMD *direct mappings* lies in the fact that it is necessary to guaranty that there is the same level of parallelism between the two instructions, i.e. the same interpretation of registers (couple of number and size of the values).

This case is widely applicable on arithmetic operations of SIMD instruction sets. Figure 18.7 illustrates the DBT of an ARM Neon `vadd.i16` into an Intel MMX/SSE `paddw`. The IR microoperation used to propagate the parallelism is named `simd_128_add_i16` and represents the SIMD instruction performing 8 parallel adds on 16-bit values in 128-bit registers.

Table 18.1 gives some examples of *direct mappings* between the ARM Neon add instructions and the Intel SSE ones. These examples are only 128-bit adds, but equivalent mapping can also be found for 64-bit instructions and for other arithmetic instructions such as `sub`, `and`, `or`, and `xor`.

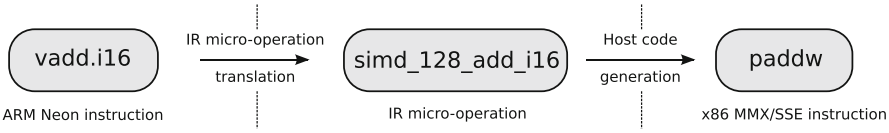

Fig. 18.7 Direct mapping between `vadd.i16` Neon instruction and `paddw` MMX/SSE instruction

Table 18.1 Mapping between addition instructions

Operation	Neon instruction	MMX/SSE instruction
Add 8 bits	`vadd.i8 Qd, Qn, Qm`	`paddb xmm1, xmm2`
Add 16 bits	`vadd.i16 Qd, Qn, Qm`	`paddw xmm1, xmm2`
Add 32 bits	`vadd.i32 Qd, Qn, Qm`	`paddd xmm1, xmm2`
Add 64 bits	`vadd.i64 Qd, Qn, Qm`	`paddq xmm1, xmm2`

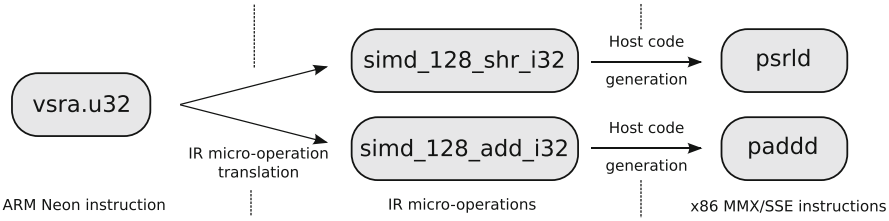

Fig. 18.8 The `vsra` Neon instruction is translated into two IR microoperations

No direct mapping: in a less favorable case, there exists no direct mapping between instructions of the instruction sets. Most of the cases, this lack of mapping is due to a lack of generality of the operations performed by the target SIMD instruction. In that case it is only of little interest to have a microoperation in the IR for that instruction. The strategy in such cases is to split the target SIMD instruction in more elementary operations already present in the IR. This technique is once more identical to the one used in scalar DBT, but more parameters have to be taken into account during the process, i.e. parallelism level and registers interpretation.

Figure 18.8 gives an example of this situation with the translation of the ARM Neon `vsra.u32` instruction which performs a right shift on operands and accumulate the shifted results in the output register. This SIMD instruction is translated into two elementary IR microoperations `simd_128_shr_i32` and `simd_128_add_i32`. The code generator can then find an equivalent for each microoperation, i.e. `psrld` and `paddd`.

Exceptional case: a third and least favorable case is finally possible. This situation occurs quite rarely, but due to the way instructions have been chosen for integration in the SIMD instruction sets, it can be encountered. It happens when an SIMD instruction of the target can be translated into a corresponding IR microoperation, but no equivalent is available in the host SIMD instruction set. As shown Table 18.2, all versions of the shift are available in ARM Neon

Table 18.2 Mapping between left shift instructions

Operation	Neon instruction	MMX/SSE instruction
shl 8 bits	`vshl.i8 Qd, Qm, #imm`	N/A
shl 16 bits	`vshl.i16 Qd, Qm, #imm`	`psllw xmm1, xmm2`
shl 32 bits	`vshl.i32 Qd, Qm, #imm`	`pslld xmm1, xmm2`
shl 64 bits	`vshl.i64 Qd, Qm, #imm`	`psllq xmm1, xmm2`

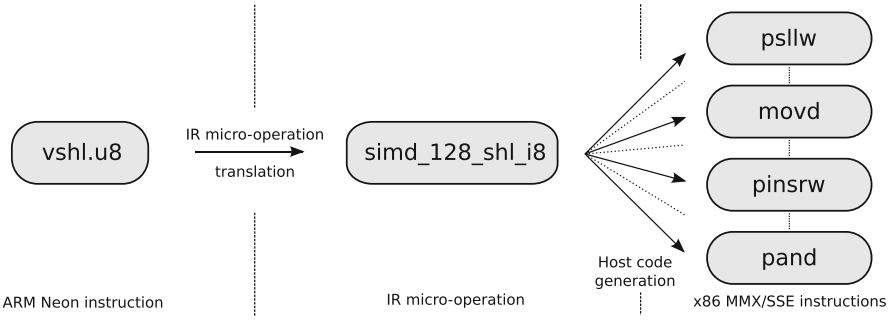

Fig. 18.9 The left shift microoperation is translated into multiple MMX/SSE instructions

SIMD instruction set. This is even true for all SIMD instruction sets that have been analyzed for this study, except for the Intel SSE SIMD instruction set. As it can be realized from this table, there exists no instruction for shifting 8-bit values. As this operation is present in all other instruction sets, it is present in the IR.

The code generator has then to solve this situation by generating multiple host instructions, as shown Fig. 18.9. The example given in this figure is for the translation of a 8-bit logical left shift emulated by a 16-bit version.

Comparison instructions: as far as comparisons are concerned, PowerPC Altivec, Sparc VIS, MMX/SSE, and Neon instruction sets provide the result for each element in the output operand, whereas the MIPS DSPASE sets flags. Because of this unbalanced distribution, a reasonable choice is to define microoperations producing their results in the output operand.

18.3.2 Support for VLIW Architectures

VLIW are not uncommon in the embedded space; indeed several recent many-core architectures are VLIW based [10, 14, 18], as they provide a high computing vs. power efficiency. The VLIW idea is to make a processor simple yet powerful by having the compiler provide the Instruction-Level Parallelism (ILP) explicitly in the execute packet, i.e., the set of instructions to be executed concurrently. The VLIW implementation choices are mainly trade-offs regarding simplicity of the design against compiler complexity, the major one being bypasses and stalls against register update latencies (also called delay slots).

18.3.2.1 VLIW Specificities

VLIW have specificities which render the VLIW to scalar DBT process particularly hard. The main characteristic of the execute packet is that multiple member instructions are meant to be executed in parallel. Some may use a register as input operand, while another one might use it as an output operand. The correct behavior is to feed the input operands with values the register had before the execute packet begin, which is not trivial in the DBT context.

The next characteristic concerns the latency of arithmetic and logic instructions, which may be greater than one cycle. Instead of stalling the pipeline until the availability of the result in the destination register, the compiler has the responsibility to ensure that an instruction depending on this result will not be scheduled before the end of the delay. It also means that all instructions executed in between can read this register to get its old value, and it is even possible to write it, as the actions will occur after the corresponding latency, as long as there is no multiple writers on the same destination register at the same cycle (otherwise, the result is undefined).

The last specificity is related to branches. Branch instructions have also delay slots, which means that instructions following a branch will be executed irrespective of the branch outcome.

18.3.2.2 VLIW DBT Extension Principles

The DBT has to handle the fact that multiple instructions are executed in parallel which may have destination registers being source registers of others in the same execute packet. The Write-After-Read (WAR) dependencies must be fulfilled to obtain correct behavior against execute packet semantic. The systematic solution to this problem is to introduce a new copy of a register each time it is overwritten. This strategy is similar to the Static Single Assignment (SSA) [9] form used in compilation when considering target registers as variables.

Applied to the intra-execute packet WAR dependency, this leads to at most two living versions of the register in each execute packet, one corresponding to the old value and one corresponding to the updated value.

Although this solution clearly solves the WAR problem, it does not solve the issue of instructions delay slots. The solution is again relatively simple: the old version of a register must be kept and used in case of a read until all delay slots of the instruction have been consumed. This results in keeping possibly alive more than two versions of the same register at the same moment, which is clearly not the case in conventional SSA.

The number of living versions of the registers increases, but fortunately, this number is bounded to a reasonable amount, which is the maximum instruction latency among all instructions plus 1. Indeed, the worst case is a sequence of largest latency instructions writing to the same register. In that case, one living version must exist for each instruction executed between the execution of the first occurrence and the availability of its result, plus one version for the previous value.

Finally, the branches have to be handled in two phases. Firstly, the code responsible for the computation of the branch target address is generated when

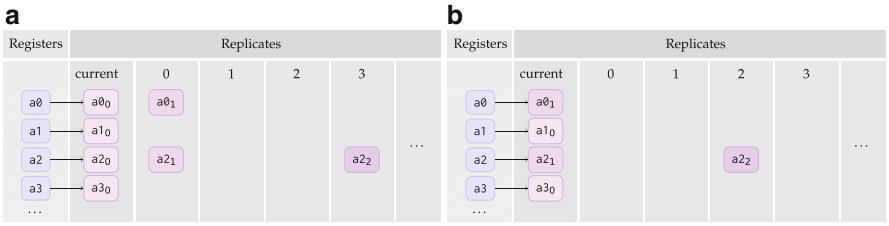

Fig. 18.10 Reading and writing registers replicates

encountering the branch instruction. Secondly, a TB ending instruction is produced after the delayed instructions and `pc` is updated accordingly.

18.3.2.3 TB Entry and Exit States

Using the above-described techniques (plus others not described here but similar in spirit, to handle, e.g., predicated instructions), the DBT translation phase can produce a sequential bytecode using registers replicates accomplishing the exact functional behavior of the source VLIW code.

For implementation purposes, each register points to the head of a queue. Reading the register value is done accessing the head, while the registers replicates (created when an instruction writes the register) are inserted in the position representing their latency. Figure 18.10 illustrates the idea. At first, replicates 0 are created in case a read occurs. Then, depending on the execute packet instructions, two 0 cycle latency instructions writing registers 0 and 2 and a 3 cycle latency instruction writing register 2 Fig. 18.10a, the replicates are created. After the translation of an execute packet, the queues are shifted left one position (see Fig. 18.10b), the leftmost replicate being kept if no value overwrites it, and the useless replicates are freed.

Due to this translation time renaming strategy, the working version (currently used versions) of the registers, although known, is unpredictable. More precisely, as TB are translated independently, the working versions when leaving a TB are unknown to the newly entered TB. Indeed, when a TB has several different predecessors, there is no way to guaranty that the working version will be identical at the exit of all the preceding TBs, and the translator does not even know if a TB has more than one predecessor when performing the translation.

The solution for handling this need of TB independence is to define a canonical entry and exit working register set for TBs. In that way, once translated, the TB will be reusable from all TBs pointing to it. The canonical state will be composed of the first replicate of each register, in which are mapped the first version of each register at the beginning of TB translation.

A further, this time dynamic, complication may arrive: in some cases, the delay slots cross the border of a TB. In that case, an external mechanism needs to be set up to handle these delay slots. This mechanism needs to be external to the translator because of the TB independence requirements. Indeed, the translator loses

the information about delay slots when exiting a TB, and thus only a run-time mechanism can manage the delay slots in that case.

This mechanism records the pending delayed registers with their current cycle delay when exiting the current TB at run time. The next TB executed consults the recorded data during its first execution cycles (bounded by the maximum possible latency) to check if there are some registers needing to be updated as they reach their latency. This can be done through helper calls inserted by the translator.

18.3.2.4 Complexity of the Modifications

From a pure functional point of view, all modifications needed to implement the VLIW DBT are part of the translator. The code generator does not strictly need to be modified to handle these new features.

The modifications needed are first a change from a simple array of registers in which each cell represents a target register to an array of these registers in which a line represents all the replicates of one target register. This change requires to modify all mechanisms at translation time to allocate the correct replicate each time an assignment to a target register occurs.

The delay slot handling, delaying use of a newer replicate, can be modeled using a simple queue, in which future versions of registers can be inserted at the corresponding delay. At the end of each cycle, all delayed register versions progress of one cycle in the delay queue. All these computations occur at translation time. Complexity is once again limited.

The handling of the canonical state implies a modification of the translator but impacts the generated code. Indeed, the easiest way to return to the canonical state at the end of a translation block is to generate instructions that move current working replicates of registers to the ones defined in the canonical state. This solution has a limited complexity on the translator but has the unfortunate side effect of increasing the TB size and thus its execution time.

Finally, the handling of delay slots crossing TB edges is a pure run-time mechanism. This mechanism has to be identical for all TB to be valid for all sequences of TB. The amount of generated code for this purpose is not huge, as it consists of generating helper calls to propagate correctly the register updates. Even though the functions themselves are quite straightforward since they only handle registers updates through a run-time delay queue similar to the one described before, the run-time overheads necessary to set up and perform these calls are quite large compared to a simple sequence of generated host instructions.

18.4 Annotations in Dynamic Binary Translation

Although DBT is a very efficient technique for instruction interpretation, it is not, in its usual form, suited to performance evaluation of software, making the virtual platforms built on top of it unsuited to design space exploration of hardware/software systems. Indeed, the translation process produces host code

Target code (MIPS)	Generated micro-operations (QEMU, simplified)	Generated host code (x86_64, simplified)

`sltiu v1,v0,23`	`ld_i32 tmp0,env,$0x48c`
	`add_i32 tmp0,tmp0,1`
	`st_i32 tmp0,env,$0x48c`
	`mov_i32 tmp1,v0`
	`movi_i32 tmp2,$0x17`
	`setcond_i32 v1,tmp1,tmp2,ltu`
		`mov 0x48c(%r14),%eax`
		`add $1,%eax`
		`mov %eax,0x48c(%r14)`
		`mov %ebx,0x74(%r14)`
		`cmp $0x17,%ebp`
		`setb %bpl`
		`movzbl %bpl,%ebp`
		`mov %ebp,0xc(%r14)`

Fig. 18.11 Generation of annotated code

whose behavior must reproduce the one of the target code, and there is no provision to estimate any kind of information (time, power, etc.) related to the execution of a translated block. However, as translation generates code, it is possible to produce nonfunctional code which aim at doing, during execution, some specific activity [6,30]. These nonfunctional pieces of code are called *annotations*. The first goal of these annotations is to make the simulated processors accurate from the point of view of the time internally required for instructions execution.

The place at which these annotations are exactly placed depends on the overall virtual prototyping approach, but it can be either before the first functional instruction of a translation block, before or after a specific instruction, or after the last functional instruction of a translation block.

Assuming the annotation targets a rough estimation of the software execution time (excluding cache and memory effects), a first approach consist of generating, before the translation of the instruction, an addition on a global variable, as illustrated Fig. 18.11, which can be compared with Fig. 18.6 to measure the overhead due to annotation.

In this case, the number of cycles is a field at offset `0x48c` in the environment.

The value of the field is incremented by the number of cycles associated to the instruction, quantity typically found using a look-up table that contains the information copied from the processor datasheet. Sometimes the number of cycles required by an instruction depends on values available only when the instruction is executed. For instance, the number of cycles required by a multiplication instruction may depend on the values of the operands, which may differ from an execution to another. In that case, specific code has to be generated to add the corresponding difference to the preceding value. Since entering a translation block guarantees that all instructions it contains will be executed, a wiser way, that does more at translation time but less at execution time, consists of generating code at the end of the translation block that adds to a variable the value accumulated during the translation of the whole translation block.

Annotations can also be used for power estimation, or for totally different purposes, such as fault injection for code analysis [3] or generation of traces for analysis [8].

18.4.1 Cache Modeling Strategies

A modification which greatly improves the time accuracy (at the cost of execution speed) consists of modeling the instruction and data caches. This section deals with level-1 (L1) caches, as the situation with upper-level caches depends on the structure of the memory hierarchy. Indeed, if the level-2 (L2) caches are private, then the same approach as for L1 cache, described below, should be used. However, if they are shared, then a global state must be visible, so that correct updates take place. The simulation of the L2 cache can either be a component at TLM level or a shared structure within the DBT engine. In the former case, it takes benefit from the event-driven nature of the TLM simulator to synchronize with the rest of the system as any other component would do, at the cost of more synchronizations within the simulator. In the latter case, the relative progression of all hardware models must be considered carefully.

For the instruction cache, the access to the models occurs thanks to a helper called through a microoperation inserted at the beginning of each translated translation block and, inside a translation block, before the first instruction from each instruction cache block. As exact target instruction addresses are known at translation time (even for dynamically generated code since it is translated at run time), this can always be done at relatively low cost.

Data caches using write-through or write-back policies can also be modeled for main memory read/write data accesses. Each time a main memory location is read or written, its presence in the data cache is checked and the proper action (return value, fetch block, write-allocate or write update) taken. Here also the addresses are exact, so the modeling in terms of hits and misses by a simple array of addresses can be faithful.

For set associative caches, the replacement policy may be difficult to reproduce exactly, e.g., when a set is chosen at random with a generation of the random number depending on a free-running counter, but this is not DBT specific.

Figure 18.12 shows how the annotations are inserted to handle the computation of cache misses. To limit the code size, pseudo-code instead of actual code is used. In this figure, the first column is the instruction address, the second column the generated code without annotations, and the third column the generated code with annotations. Assuming a 4 32-bit words long cache block, it is necessary to check the presence of the instruction in the cache only when the 4 lower bits of the address are equal to zero, which is done by a call to the `insn_cache_verify()` helper when the program counter has this property. Read and write accesses, even though handled specifically by the translator back-end to simulate the MMU and actually access the memory and peripherals, are simple microoperations in the front-end. To model the L1 data cache, helper calls (`read_access()` and `write_access()`) are added.

When modeling shared memory processor subsystems, a further issue is cache coherency. Two solutions can be thought of for maintaining coherent data or

Insn addr	Target code (pseudo-code)	Generated code (pseudo-code)	Generated annotated code (pseudo-code)

10	insn1_reg_operation	host_insn1_for_insn1
		.....
		host_insnN1_for_insn1
			insn_cache_verify(0x18);
			nb_cycles += cpu_datasheet[insn1];
			host_insn1_for_insn1
			.....
			host_insnN1_for_insn1

14	insn2_load_from_addr1	host_insn1_for_insn2
		.....
		host_insnN2_for_insn2
			nb_cycles += cpu_datasheet[insn2];
			read_access(addr1);
			host_insn1_for_insn2
			.....
			host_insnN2_for_insn2

18	insn3_reg_operation	host_insn1_for_insn3	nb_cycles += cpu_datasheet[insn3]; .....
1c	insn4_reg_operation	host_insn1_for_insn4	nb_cycles += cpu_datasheet[insn4]; .....

20	insn5_store_x_to_addr2	host_insn1_for_insn5
		.....
		host_insnN3_for_insn5
			instr_cache_verify(0x20);
			nb_cycles += cpu_datasheet[insn5];
			write_access(addr2, x);
			host_insn1_for_insn5
			.....
			host_insnN3_for_insn5

Fig. 18.12 Cache modeling through annotation

instruction (necessary to support dynamically generated code) within the caches. The brute force one simply ignores the cache protocol and traverses all cache models when a write occur to check for the presence or absence of the written address in the cache. When the number of processor is small, this traversal is quick enough to be acceptable. To do so, the run time of the DBT engine simply needs to maintain a list of caches accessible by all CPUs. However, when the number of processor increases, the traversal time becomes unacceptable, and the solution is then to implement (even in a simplified form) a cache coherence protocol. Indeed, assuming n is the number of writes and p the number of processors, traversing all caches is in $O(p \times n)$, while a hardware cache coherence protocol would lead to performance in $O(k \times p)$, with $k \ll 10$ in average since the number of sharers of a data is usually low. This behavior can be mimicked efficiently by accessing a hash table indexed by the hashed address whose entries contain the address as key and a pointer to the array representing the cache which caches the address as data. The tipping point between both solutions depends on the implementation details, but it seems clear that for many-core architectures, the latter one is more efficient.

As can be seen, adding cache models increases the size of the generated code and requires more complex handling of the memory accesses at execution time, which leads in any case to an important degradation in simulation speed. Given the fact that is it possible to count load and store, higher, typically analytical, models can also be used when accuracy can be trade-off for speed.

18.4.2 Modeling Branch Predictors

Even though not that many embedded processors today include branch predictors, it is worth taking a look at how it can be modeled in DBT-based simulation as the influence of this microarchitectural feature on out-of-order execution performance is major. At first, modeling a branch predictor seems easy: the branch outcomes are known, so updating a set of branch history tables (as in the TAGE [27] predictor or its descendants, current state-of-the-art predictors) to predict a future branch outcome is straightforward. However, branch predictors use large global tables, shared by all branches in the execution flow. As opposed to caches, these tables are accessed one after the other at what looks like random indexes (computed as hashes of branch history and branch instruction address). As far as simulation is concerned, repeated accesses to random places lead to poor program locality and, in consequence, poor simulation performance. Experiments have shown slowdowns of 1.5× to 15× as compared to raw DBT execution [13]. To limit this overhead, a performance/accuracy trade-off can be made by defining models which predict the behavior of the branch predictor. The principle consists of transforming the tables of the reference architecture, which are global, i.e., shared between all branches, into information local to each branch. This leads to allocating the table entries only when needed and enhances locality, thus limiting the number of cache misses when simulating the predictor model.

Taking TAGE as an example, the bimodal table, as can be seen Fig. 18.13, is reduced to a 2 bit counter which can be stored in the local data of each branch. Then, for the tagged tables, two parts are distinguished, the tables themselves and the way they are indexed. Concerning the tables, a simple solution is to use a single tagged table, ignoring history length. In the same way as for the bimodal table, only

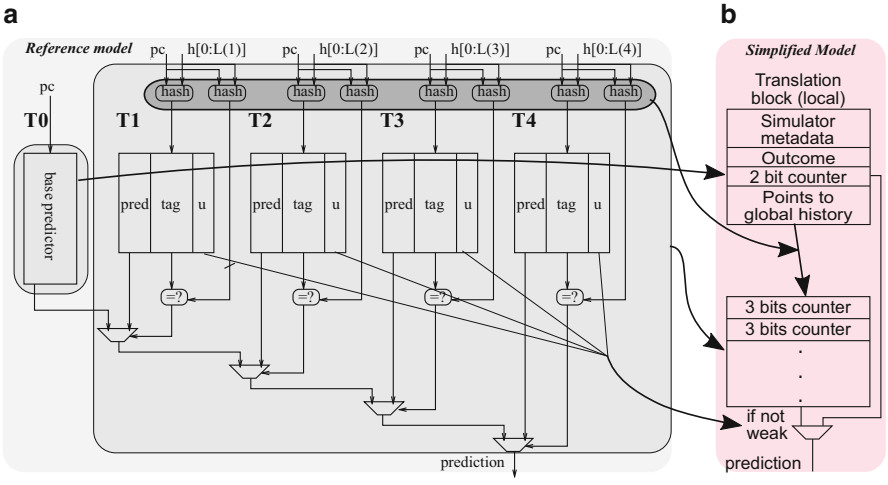

Fig. 18.13 (**a**) Reference [27] and (**b**) simplified models for a TAGE branch predictor

the potentially reachable entries, i.e., the ones concerning already seen branches, can be allocated locally for each branch. This results in a local table that is indexed by the global history, which, in the reference architecture, was hashed with the program counter (PC) of the branch.

The choice of the entry is then naturally simplified as there is only one, local, tagged table to choose from.

The confidence state of the entry in the tagged table is used to select between the tagged table and the bimodal table. If it is "weak," the prediction will be given by the bimodal table; otherwise it will be given by the tagged one. This should ensure that, as in the reference architecture, tagged prediction is chosen only when the tagged entry is "sure" of its prediction. The tag of the tagged table is not useful anymore, as entries concerning one branch are now specific to it. In other words, aliasing, i.e., multiple branches pointing to the same entry, already very unlikely thanks to tags, is impossible in the simplified model.

The resulting model uses memory local to the current basic block, which tends to enhance locality, but it also uses more memory, as information shared due to aliasing is now duplicated. The information used by the simplified predictor is similar to that of the reference architecture: the outcome of the current branch, its own data, a global history of branch outcomes, and, finally, the structure of the already executed code (to store data per branch). This model produces identical predictions to the TAGE architecture in average 95% of the time for a 5% execution time overhead. Even though 95% identical predictions may seem a good achievement, the impact on the execution time may be of importance, so finer models can also be of interest.

As for caches, a wide range of models can be used for taking into account branch prediction, and it boils down to produce annotations at the end of the basic blocks, as shown Fig. 18.14. The `bp_model` helper is called with the address of the branch instruction (`PC`) and the outcome of the branch (`bcond`).

18.5 Integration with TLM Simulations

Integrating DBT-based Instruction-Set Simulator (ISS) in a TLM simulation environment can be done in two steps, as depicted Fig. 18.15. First, all processors belonging to a Symmetric Multi-Processing (SMP) subsystem are grouped together

Target code (MIPS)	Generated micro-operations (QEMU, simplified)
`...`	`mov_i32 tmp0,t3`
`beqz t3,0x8000024c`	`movi_i32 tmp1,$0x0`
`nop`	`setcond_i32 bcond,tmp0,tmp1,eq`
	`movi_i64 tmp2,$bp_model`
	`call tmp2, PC, bcond`
	`movi_i32 tmp1,$0x0`
	`brcond_i32 bcond,tmp1,ne,$0x1`
	`...`

Fig. 18.14 Annotation to model branch prediction

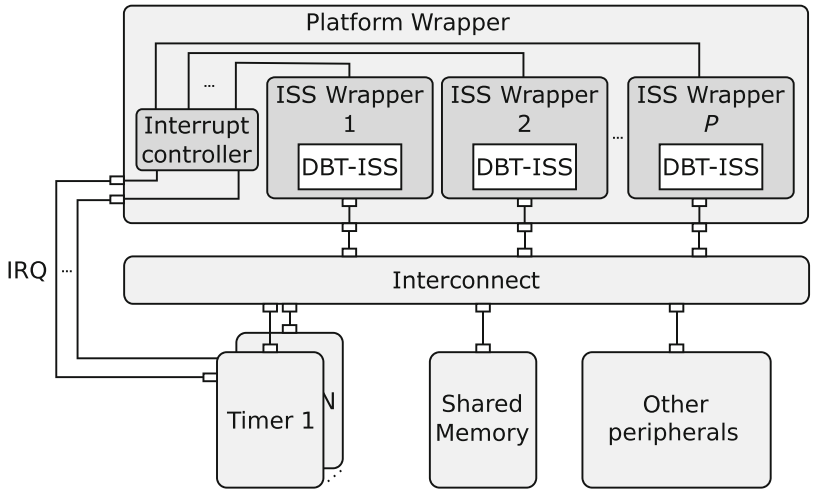

Fig. 18.15 Binary translation based simulator – SystemC simulation platform

in a module that can be instantiated by the TLM simulator (*platform_wrapper*). It is assumed that all processors are identical and have identical cache geometries and that they have a shared view of the memory. This allows to share the translation cache between them, avoiding retranslation on one processor of code migrating from another processor in SMP systems and an efficient implementation of cache coherency.

Second, the platform wrapper instantiates a module wrapper for each processor (*iss_wrapper*). The execution of each processor is performed in the context of the process of its wrapper. This way, the processors are simulated concurrently by the TLM simulator time-sharing scheduler. The platform wrapper is useful for managing the common aspects shared by the processors (e.g., inter-processor interrupt management, platform specific registers, interrupt controllers etc.).

The platform wrapper is connected to an interconnect, through which it can communicate with other hardware components (memory, timers, DMA engines, frame buffer, etc.) also connected to it. All hardware components are implemented as TLM modules.

From the initial DBT platform, the platform uses only the processor models with, if required, their MMUs. All other devices are externalized and implemented as TLM modules. This allows to enforce the notion of IP and reuse, thanks to the TLM principles, whereas devices in DBT are described in very ad hoc manner and use shortcuts hardly acceptable when the simulation is also used for design space exploration purposes. The main memory is also implemented as one or more TLM modules. For accessing TLM models other than the main memory, a few ranges of the simulated processor physical addresses are mapped as I/O addresses in the processor wrapper. The I/O requests from the simulated processors are then transformed by the processor wrappers into TLM requests, using the protocol understood by the interconnect. Memory, on the other hand, is accessed through

a direct memory interface. This avoids relinquishing the CPU to the TLM part of the simulation and thus saves a lot of time.

18.5.1 Precision Levels

Depending upon the accuracy one expects from the simulation, four trade-offs can be made regarding memory accesses.

The first approach does not implement caches and uses the main memory internally allocated by the DBT engine. The time required for executing the number of cycles corresponding to the instructions simulated is consumed using the *wait* function of the simulator. In this configuration, it is considered that the memory is always available for all processors, without any cycle cost for accessing it. The communication with other peripherals is performed by sending requests over the interconnect. The time consumed for these accesses is composed of the time consumed by each TLM components involved in the transmission and the reception of the request packets. In this case, a simulated processor synchronizes with the rest of the TLM platform only when an I/O operation is executed and when that processor is unscheduled by the DBT simulator. Due to the reduced number of synchronizations, large pieces of translated code are executed without interruption. As a result, the simulation will be very fast (close to the DBT alone). The accuracy in this case will be low as the cache effects and the time required to communicate with the main memory over the interconnect are not accounted for. Since all memory accesses are done without going through the interconnect, there is no need for an explicit support for cache coherent mechanisms.

The second approach relies on the caches being implemented only from the hit/miss point of view, while the main memory of the initial DBT engine is still used. As opposed to native simulation or compiled simulation presented in ▸ Chap. 19, "Host-Compiled Simulation", dynamic binary translation uses the exact addresses for instructions fetch and data accesses. However, the target instructions are fetched from memory only once, for translation, before their first execution. The simulator always executes the generated binary host code stored in the translation cache. So, to accurately account for these accesses, a model of the cache is needed. Both data and instruction caches can be modeled as pure directories, so that an array access (with the proper tag, index, and offset) indicates if the instruction would in reality be in the cache. A cache miss issues a TLM *wait* for a time precomputed to be required to load a cache line, without actually sending the request over interconnect. As for the previous approach, the I/O operations involve the interconnect and other TLM hardware models. The time corresponding to the simulated cycles is consumed at the beginning of the next synchronization. In this case, the processors are synchronized with the TLM simulator when a cache miss occurs, an I/O is executed or when they are unscheduled by the DBT engine. The simulation speed for this configuration is reduced a lot because of the large number of synchronizations produced by the cache misses. As the precomputed time is consumed directly in the cache model, a single timed event is generated for each cache miss. This is not much, considering

that the transfer of a single byte over the interconnect requires more than 10 timed and untimed events. The accuracy increases by using a precomputed average value for the time required for a memory transfer over the interconnect. However, the interconnect load, hardly guessable as it is highly nonlinear when congested, is not taken into account.

The third approach is an extension of the second one, in which, instead of consuming the precomputed time when the cache misses occur, the consumption of time is postponed until the next synchronization produced by an I/O operation or by the normal unscheduling of a processor. At the synchronization moment, the sum of the precomputed times required by all write accesses and cache misses that have occurred since the previous synchronization is consumed. This way, the number of synchronizations is reduced, increasing simulation speed. The chances of preventing other processors to modify a variable waited on by the current simulated processor are higher in this configuration compared to the previous one because of the small number of synchronizations. This may have a negative impact on simulation speed as more cycles have to be simulated by a polling processor and has a negative impact on simulation accuracy.

The fourth and most accurate approach fully implements the caches and uses an external TLM memory module as main memory. In this case, in addition to the directory, the caches also have their data part. However, the data of the instruction cache is ignored. The instructions needed for translation are searched directly in the memory module, without issuing a TLM *wait*. The loading price will be paid by the instruction cache when the generated code is executed from the translation cache, as explained Fig. 18.12. In all cases, a cache miss issues a request over the interconnect. The simulation speed for this configuration will be even slower, because the requests and responses pass through all the components that are required for the transfer.

18.5.2 TLM Synchronization Points

In this DBT+TLM integration approach, a processor is simulated as long as it does not communicate with the world behind its caches and the DBT engine does not stop it. When an instruction/data cache misses or an I/O occurs, the processor simulation stops, and the processor wrapper synchronizes itself with the rest of the platform by consuming the estimated time required by the real processor to execute the instructions simulated since the last synchronization. In case no such event occurs, in order to limit the divergence between the different processors' execution, a synchronization can be forced after a predefined period of time without synchronization. For the target processor instructions designed for synchronization of the software running on a SMP architecture (e.g., exclusive load and store, compare, and swap), a synchronization should also be generated.

The processors' simulation order depends on the time consumed by the processors at synchronizations. A synchronization condition may occur at any time during the execution of a translation block (e.g., cache miss); thus unscheduling does not anymore always occur at the boundary of a translation block. As the DBT engine

unschedules the processors only at the translation block border, it is necessary to save their "execution context" before synchronization and restore it afterward.

18.5.2.1 TLM Synchronization After Long Intervals Lacking in Synchronization

Due to the fact that simulated processors do not synchronize at each memory access, if two or more simulated processors read/write from/to the same memory address, the instructions executed by these processors may differ from those executed on a cycle accurate platform. A write to an address should invalidate the corresponding cache line in all caches but the one of the writing processor, and these other processors should see the new value at their next read from that address. Because of the direct access to memory, the write is visible by the rest of processors before it happens in the simulated timeline, if the writing processor is simulated before the reading one. If the processors' simulation order is inverted, the reading processor does not see the effect of writing until it synchronizes and the writing processor executes the writing code.

Processor unscheduling after a predefined time without synchronization is needed even for cases when a processor waits in a loop for a simple variable to be changed by another processor or any initiator of the system or even by an interrupt handler on the same processor. This kind of loops would also prevent the interrupts to occur for that processor because the interrupt pending flag is set during synchronization. For example, for computing the processor speed, Linux waits in a loop for the *jiffies* variable to be incremented by the timer interrupt handler. The condition of unscheduling due to lack of synchronization is verified at the beginning of each translation block. The time period for this unscheduling condition determines the maximum lag of the interrupts.

18.5.2.2 TLM Synchronizations Caused by Target Synchronization Instructions

The threads of a software application usually synchronize together. A spin lock is an example of a software synchronization mechanism. The lock and unlock function of the spinlocks are usually implemented using exclusive load and store instructions.

Figure 18.16 presents an example of software running on two processors and using a spinlock for the software synchronization. This figure shows what would happen if the simulator would not generate a synchronization for this type of target instructions or if the spinlock functions would not be implemented using exclusive access instructions.

Figure 18.16a presents the execution on a real hardware. The first processor ($P1$) locks a spinlock at $t1$, at $t2$ a cache miss occurs, at $t4$ $P1$ releases the spinlock, and at $t6$ it executes an I/O operation. The second processor tries to lock the spinlock ($t3$) just before $t4$; it actually obtains the lock at $t4'$ and releases the spinlock at $t5$.

The execution on our platform in the case when the simulator would not generate synchronization for the exclusive accesses is depicted in Fig. 18.16b. Considering that $P1$ is first scheduled for simulation, it locks the spinlock at $t1$ without being unscheduled (the spinlock is placed in the main memory), but at $t2$ it is unscheduled

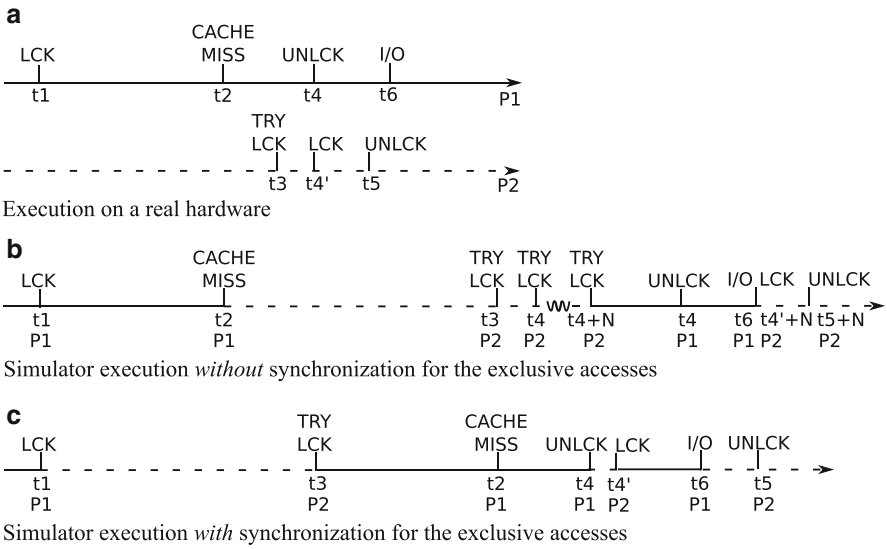

Fig. 18.16 Simulation behaviors based on the spinlock implementation

for synchronization before loading the cache line. $P2$ is now scheduled and at $t3$ it begins trying to take the lock. $P2$ reads in an infinite loop the spinlock locked value from the main memory.

After the predefined time without synchronization, $P2$ would be however unscheduled at time $t4 + N$. Then, $P1$ is scheduled, it unlocks the spinlock, and then it is unscheduled for synchronization before the I/O operation. $P1$ will be able now to take the spinlock, but it has oversimulated by time N.

The simulation behavior when spinlocks functions use exclusive access functions and the synchronizations that are generated for them are presented in Fig. 18.16c. In this case, the processors synchronize before each lock and unlock. $P1$ synchronize at $t1$ and $P2$ is scheduled. $P2$ is unscheduled before its first lock attempt. $P1$ synchronizes at $t2$, but it is rescheduled because $P1$ is more advanced in simulation time ($t3 > t2$). At $t4$, before releasing the lock, $P1$ synchronizes and it is unscheduled ($t4 > t3$). Between $t3$ and $t4$ (the simulation time of $P1$), $P2$ synchronizes and it is rescheduled at each attempt to lock the spinlock. After $t4$, $P1$ is scheduled and it releases the lock and it is simulated until $t6$. $P2$ gets the lock at $t4'$ (immediately after $P1$ has released it) and release it at $t5$.

18.6 Concluding Remarks

Dynamic binary translation provides a real increase in performance as compared to instruction accurate instruction-set simulators. This enhancement comes at the price of a much greater implementation complexity and, intrinsically, less capabilities to

monitor precisely nonfunctional properties of the software. The progress toward the integration of more and more cores on SoC makes it however a must for achieving hardware/software simulation at acceptable speed.

To act as an independent processor simulator, DBT must be integrated into standard event-driven simulation environments, e.g., SystemC. Then, it must support processors with non-scalar architectures and possibly more efficiently scalar processors [19], by resorting to run-time optimizations including dynamic recompilation. And finally, it should be capable of integrating models of processor specific microarchitectural details so that software performance evaluations (mainly timing and power) can be done.

References

1. Aarno D, Engblom J (2014) Software and system development using virtual platforms: full-system simulation with wind river simics. Morgan Kaufmann, Waltham
2. Baraz L, Devor T, Etzion O, Goldenberg S, Skaletsky A, Wang Y, Zemach Y (2003) Ia-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on itanium®-based systems. In: Proceedings of the 36th annual IEEE/ACM international symposium on microarchitecture, pp 191–201
3. Becker M, Baldin D, Kuznik C, Joy MM, Xie T, Mueller W (2012) Xemu: an efficient qemu based binary mutation testing framework for embedded software. In: Proceedings of the tenth ACM international conference on Embedded software. ACM, pp 33–42
4. Bell JR (1973) Threaded code. Commun ACM 16(6):370–372
5. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the 1st IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis. ACM, pp 19–24
6. Cmelik B, Keppel D (1994) Shade: a fast instruction-set simulator for execution profiling. In: Proceedings of the 1994 ACM SIGMETRICS conference on measurement and modeling of computer systems, pp 128–137
7. Creasy RJ (1981) The origin of the VM/370 time-sharing system. IBM J Res Dev 25(5): 483–490
8. Cunha M, Fournel N, Pétrot F (2015) Collecting traces in dynamic binary translation based virtual prototyping platforms. In: Proceedings of the 2015 workshop on rapid simulation and performance evaluation: methods and tools. ACM, p 4
9. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK (1991) Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 13:451–490
10. de Dinechin BD, Ayrignac R, Beaucamps PE, Couvert P, Ganne B, de Massas PG, Jacquet F, Jones S, Chaisemartin NM, Riss F, Strudel T (2013) A clustered manycore processor architecture for embedded and accelerated applications. In: IEEE high performance extreme computing conference. IEEE, pp 1–6
11. Deutsch LP, Schiffman AM (1984) Efficient implementation of the smalltalk-80 system. In: 11th ACM SIGACT-SIGPLAN symposium on principles of programming languages, pp 297–302
12. Duesterwald E, Bala V (2000) Software profiling for hot path prediction: less is more. ACM SIGARCH Comput Archit News 28(5):202–211
13. Faravelon A, Fournel N, Pétrot F (2015) Fast and accurate branch predictor simulation. In: Proceedings of the design automation and test in Europe conference. ACM, pp 317–320
14. Flamand E (2009) Strategic directions towards multicore application specific computing. In: IEEE/ACM conference on design, automation & test in Europe, pp 1266–1266

15. Ghenassia F, Clouard A (2005) TLM: an overview and brief history. In: Ghenassia F (ed) Transaction level modeling with SystemC: TLM concepts and applications for embedded systems. Springer, Dordrecht
16. Gligor M, Fournel N, Pétrot F (2009) Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In: Proceedings of the 7th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, Grenoble, pp 71–80
17. Hawkins B, Demsky B, Bruening D, Zhao Q (2015) Optimizing binary translation of dynamically generated code. In: Proceedings of the 13th annual IEEE/ACM international symposium on code generation and optimization. IEEE Computer Society, pp 68–78
18. Lethin R (2009) How vliw almost disappeared-and then proliferated. IEEE Solid-State Circuits Mag 1(3):15–23
19. Leupers R, Eeckhout L, Martin G, Schirrmeister F, Topham N, Chen X (2011) Virtual manycore platforms: moving towards 100+ processor cores. In: Design, automation & test in Europe conference & exhibition (DATE), 2011. IEEE, pp 1–6
20. Li J, Zhang Q, Xu S, Huang B (2006) Optimizing dynamic binary translation for simd instructions. In: Proceedings of the international symposium on code generation and optimization, pp 269–280
21. Michel L, Fournel N, Pétrot F (2011) Speeding-up simd instructions dynamic binary translation in embedded processor simulation. In: Proceedings of the design, automation & test in Europe conference, pp 277–280
22. Mitchell JG (1970) The design and construction of flexible and efficient interactive programming systems. PhD thesis, Carnegie-Mellon University, Pittsburgh
23. Monton M, Carrabina J, Burton M (2009) Mixed simulation kernels for high performance virtual platforms. In: Forum on specification & design languages, pp 1–6
24. Pétrot F, Fournel N, Gerin P, Gligor M, Hamayun MM, Shen H (2011) On mpsoc software execution at the transaction level. IEEE Des Test Comput 28(3):32–43
25. Popek GJ, Goldberg RP (1974) Formal requirements for virtualizable third generation architectures. Commun ACM 17(7):412–421
26. Rohou E, Williams K, Yuste D (2013) Vectorization technology to improve interpreter performance. ACM Trans Archit Code Optim (TACO) 9(4):1–22
27. Seznec A, Michaud P (2006) A case for (partially) tagged geometric history length branch prediction. J Instr Lev Parall 8:1–23
28. Sites RL, Chernoff A, Kirk MB, Marks MP, Robinson SG (1993) Binary translation. Commun ACM 36(2):69–81. doi:10.1145/151220.151227
29. Ung D, Cifuentes C (2000) Machine-adaptable dynamic binary translation. ACM SIGPLAN Not 35(7):41–51
30. Witchel E, Rosenblum M (1996) Embra: fast and flexible machine simulation. ACM SIGMETRICS Perform Eval Rev 24(1):68–79