Abstract. In this paper, we present a simulation strategy that tries to combine the speed of the binary translation based ISSes with the accuracy of the event driven simulators. To have an accurate timing behavior for an instruction set simulator based on binary translation, we had to first solve timing issues in processor modeling, second define fast and precise cache models, and third solve the synchronization issues due to the different models of computation used in the ISSes and in the rest of the system. We have experimented our proposal using processors models provided by the QEMU framework to replace the existing ISSes and SystemC TLM as simulation environment for the whole platform.

1 Dynamic binary translation bases. QEMU

The binary translation represents the emulation of the instruction set of a processor by the instruction set of another processor using code translation. QEMU is a fast and portable emulator which emulates many architectures (X86, ARM, SPARC etc.) and runs on several architectures. To simulate guest systems, QEMU uses the dynamic binary translation. QEMU uses micro-operations as intermediate representation (IR) in the target to host code translation process. These micro-operations represent the instruction set of a simple 2-address abstract machine. This machine has three general purpose registers. For each of these micro-operations, a C function is hand coded and a unique integer identifier is associated with. These micro-operations are compiled to an object file.

Fig. 1 presents the QEMU simulation model. QEMU verifies if the sequence of target instructions starting at the address given by the program counter of the simulated processor has already been translated. In the case it was not translated before, the binary translation stage begins. The instruction corresponding to the program counter of the simulated processor is fetched from the target binary code. The fetched instruction is then decoded into several micro-operations whose identifiers are concatenated into a micro-operations identifier buffer. If the current instruction is not a branch instruction, the next target instruction is fetched and decoded. The binary translation stage ends after a branch instruction is decoded. The sequence of target instructions together treated by the binary translation stage forms a translation block. Once the block translation is finished, the tiny code generator generates a host function which is made of the
concatenation of the micro-operations compiled code corresponding to the identifiers stored in the buffer. The resulted host function is stored into a translation cache entry of the translation cache. Once the generated code is executed, the simulator verifies the existence of the translation corresponding to the new program counter. If the required translation already exists in the translation cache, it is directly executed.

2 Multiprocessor modeling

Our simulator is presented from the multiprocessor, time, frequency and energy modeling points of view.

The goal of the binary translation simulators is to be very fast and functionally correct, but not to provide information about hardware execution time. Therefore, the implementation of multi-processor platforms simply call the processors interpretation function one after another in a predefined order. The simulation order of the processors is always the same. As our goal is to estimate timings, we need to have a mean to control (and measure) the scheduling of the processor execution. To do so, the default behavior of the initial simulators must be modified, at the price of reducing the simulation speed.

2.1 ISS wrapping and connection

For each processor in the platform, we instantiate a SystemC module wrapper (iss_wrapper) as depicted in Figure 2. The execution of each processor is performed in the context of the SystemC process (SC_THREAD) of its wrapper. This way, the processors are simulated concurrently. It is necessary that each processor is simulated in the context of its own SystemC process, so that they are simulated independently under the control of SystemC.

In order to avoid the binary translation of the same target code for each simulated processor, the processors that may share the same translation cache are encapsulated into a SystemC module, called iss_group. To ensure that the host code generated for a translation block is correct for all processors in the group, the processors in a iss_group must be identical.
The processor wrappers are connected to an interconnect, through which they can communicate with other hardware components (traffic generator, timers, interrupt controller, memory, spinlocks *etc.*) also connected to it. All hardware components are implemented as SystemC modules. The interrupt lines of the different components (*e.g.* timers) are connected to the processor wrappers, which implement the interface between SystemC interrupt wires and the ISS. From the initial simulator, our approach uses only the processor models with, if required, their MMUs (Memory Management Unit). All other devices are externalized and implemented as SystemC modules for accuracy reasons.

### 2.2 SystemC synchronization points

In our model, a processor is simulated while it does not communicate with the world behind its caches and the initial simulator does not stop it. When an instruction/data cache miss or an I/O occurs, the processor simulation stops and the processor wrapper synchronizes itself with the rest of the SystemC platform by consuming the time (SystemC *wait* function) required by the real processor to execute the instructions simulated since the last synchronization.

So, unlike the cycle accurate simulators where the cycles of the ISSes are simulated concurrently, the proposed binary translation based ISS simulate a group of cycles before synchronizing with SystemC and allowing other ISSes to execute. The number of cycles in these groups may vary from one to many cycles, depending on the simulated target code. The simulation of these cycles takes zero SystemC time.

The synchronization also takes place after a predefined period of time without synchronization. For the target processor instructions designed for the synchronization of the software running on a SMP architecture, a SystemC synchronization should also be generated (*e.g.* exclusive load and store).

The processors simulation order depends on the time consumed by the processors at synchronizations. A synchronization condition may occur at any time during the simulation of a translation block (*e.g.* cache miss), thus the unscheduling does not respect anymore the border of the translation blocks.
3 Time modeling

The binary translation based simulators do not have a time notion for the simulated platform. For the timeout events required by the timer devices, they usually use different external mechanisms and sometimes asynchronous to the simulated platform. For instance, QEMU uses one of the host timers (hpet, rtc etc.), that generates an alarm signal on the host machine.

Instead of using the host time, our platform uses the SystemC simulation time. The timer interrupts are generated by the SystemC timer modules. For time accuracy of the processor model, a few changes have been made to the initial simulator in order to model the time required to execute instructions. These changes are performed by adding information when the target processor binary code is translated into the host processor binary code.

The first annotation tries to make simulated processors accurate from the point of view of the time internally required for instructions execution. When the target binary code is translated, we add a micro-operation before the micro-operations of each translated target instruction. The added micro-operation increments the number of cycles of the instructions that have been simulated since the last synchronization. The increment uses the number of cycles of the instructions given by the datasheet of the simulated processor.

Another modification applied to the initial simulator for improving the time accuracy consists of presence verification and loading of the required cache line into the instruction cache. The verification and the loading are done in a function called by a micro-operation inserted at the beginning of each translated translation block and, inside a translation block, before the first instruction of each instruction cache line. In case of an instruction cache miss, the currently simulated processor is synchronized with the rest of the SystemC platform by consuming the time required to execute the number of cycles obtained by the first annotation. After the synchronization, the processor wrapper sends a request over the interconnect for a burst transfer from memory of the implied cache line.

A similar modification refers to the main memory read/write data accesses. Each time a main memory location is read, its presence in the data cache is verified. The mechanism described for the instruction cache miss is also applied in the case of a data cache miss. The I/O accesses to SystemC modules also generate a SystemC synchronization followed by a request in the SystemC subsystem.

4 Experimental results

Table 1 presents the speedup and the accuracy of our simulator compared to a cycle accurate simulator using SOCLIB components and to a TLM simulator using interpretative SOCLIB ISSes and the rest of the hardware components from our simulator.

<table>
<thead>
<tr>
<th>Simulation speedup</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle accurate simulator</td>
<td>300X</td>
</tr>
<tr>
<td>TLM simulator using interpretive ISS</td>
<td>2X</td>
</tr>
</tbody>
</table>