TLMu

Edgar E. Iglesias
AXIS Communications AB
edgar.iglesias@axis.com
edgar.iglesias@gmail.com

June 10, 2011

Abstract

Using Transaction Level Modeling (TLM) is an increasingly popular way of doing early modeling of large systems. By using higher abstraction levels, it is possible to keep down the level of complexity and increase the speed of simulations. TLM enables the creation of virtual platforms that are well suited not only for early design exploration, but also for early software development.

When modeling large systems, a common requirement is the need for modeling CPUs. Even more, the need to model multiple CPU's of varying architectures, is increasing.

In this work we introduce TLMu, a modified version of QEMU suitable for integration with SystemC TLM-2.0 based systems. We show that it is possible to achieve high emulation speed, to model systems with multiple CPU cores of different architectures, to get rough performance estimations and to get good debugging facilities for software developers.

1 Introduction

QEMU is an emulator framework capable of cross running user-space linux applications across different architectures and also able to cross run complete operating systems across different architectures.

QEMU uses binary translation when emulating CPU's to achieve high emulation speed.

TLMu extends QEMU by allowing it to interact with device models provided by an external emulator. TLMu also makes it possible to have multiple CPU emulators in the same process, emulating different architectures (e.g an ARM and a MIPS core).

TLMu allows good flexibility in how systems can be partitioned. TLMu can use RAM both internal to the TLMu instance and also external (provided by the main emulation system). Different TLMu instances within an emulator can cross access each others RAMs and devices.

2 Heterogeneous CPU mix

When building QEMU, you build an emulator per architecture. TLMu wraps QEMU and a set of hooks into a set of libraries. A shared library, libtlmu-{arch}.so and a static libtlmu.a. Whenever a TLMu CPU core needs to be instantiated, the corresponding library is cloned and dynamically loaded into the emulator process.

All the TLMu CPU cores interface with the TLM-2.0 system through a TLM-2.0 module that exposes a set of TLM-2.0 sockets. See the TLMu documentation for more details.

3 Memory accesses

Memory accesses made by a TLMu core may either be handled internally (by the TLMu partial system) or they may be mapped to TLM-2.0 transactions. The TLM-2.0 module that bridges TLMu access to the TLM-2.0 side supports blocking transport and DMI, but not non-blocking transport.

When mapping RAM's from which the TLMu CPU may execute code from, a small limitation exists. The main emulator needs to tell the TLMu system in advance which externally mapped areas that are RAM. The reason for this is that internally, QEMU differentiates heavily between RAM's and devices.

It is also possible to make memory accesses from the main emulator (and thus from other TLMu instances) into the TLMu system. TLMu's TLM-2.0 wrapper provides a target socket for these accesses. Accessing RAM's that are provided by TLMu also support DMI, but they have no cost models for the moment.

4 Simulation speed

We ran a reference program that ran a computationally heavy operation (lets call it XOP) in software and also another program that accelerates the XOP operation by using an emulated hardware accelerator.

We emulate a CRIS CPU with a ROM, a RAM and the XOP accelerator.

The TLM-2.0 version of the emulator has the CRIS provided by TLMu and all the other devices provided by the TLM-2.0 system.

The QEMU only version of the system has all the devices built into a QEMU machine.

4.1 QEMU vs TLMu

Due to the fact that TLMu compiles the emulator system as position independent code, it is expected that TLMu will run somewhat slower than QEMU. Additionally, TLMu makes use of the SystemC/TLM-2.0 global quantum concept, causing further interruptions of the virtual CPU.

IO acccess, from QEMU to the TLM-2.0 world need to be translated and forwarded to SystemC, further adding to the execution times.

In the first test we compare QEMU vs TLMu with 10 rounds of XOP done in pure guest software, i.e without any use of the emulated XOP accelerator. TLMu was run with a 10us global quantum.

```
QEMU: 5.198s
TLMu: 5.257s
```

TLMu runs 1.1% slower. This is likely due to the overhead of PIC code and the quantum timer.

Next run, we compare emulation of XOP by use of the XOP accelerator.

```
QEMU: 3.454s
TLMu: 3.567s
```

TLMu runs 3.3% slower. Compared to the previous run, this adds the overhead of bus transaction translation into TLM-2.0 generic payload.

4.2 The global quantum

When using the loosely timed coding style, TLM-2.0 uses a global quantum to control the amount of virtual time a given process is allowed to run ahead of time. In general, the higher the value, the longer runs per process, the lower the amount of context switches and ultimately lower simulation times. On the other hand, as the quantum gets higher, the lower the accuracy in the timing of the emulation.

In some cases, loosing timing accuracy may significantly affect the total simulation time. For example, if a signal arrives late in time due to a high global quantum and that signal is meant to signal the end of simulation, we might end up simulating much more virtual time than would be needed if we had better accuracy (i.e lower global quantum). In the following runs we therefor compare real-time vs virtual time as well. We use 1ns global quantum as the reference for the speed comparisons.

```
1ns global quantum
Real time: 4.367 s
Virtual time: 0.4322 s
Realtime / virtual time: 10.104

10ns global quantum
Real time: 4.318 s (-1.1%)
Virtual time: 0.4322 s
Realtime / virtual time: 9.991 (-1.1%)

100ns global quantum
Real time: 4.225 s (-3.3%)
Virtual time: 0.4322 s
Realtime / virtual time: 9.978 (-3.3%)

200ns global quantum
Real time: 3.953 s (-9.5%)
Virtual time: 0.4322 s
Realtime / virtual time: 9.146 (-9.5%)

500ns global quantum
Real time: 3.716 s (-14.9%)
Virtual time: 0.4322 s
Realtime / virtual time: 8.598 (-14.9%)

1us global quantum
Real time: 3.642 s (-16.6%)
Virtual time: 0.4322 s
Realtime / virtual time: 8.427 (-16.6%)

10us global quantum
Real time: 3.567 s (-18.3%)
Virtual time: 0.4322 s
Realtime / virtual time: 8.253 (-18.3%)

100us global quantum
Real time: 3.559 s (-18.5%)
Virtual time: 0.4324 s (+0.05%)
Realtime / virtual time: 8.231 (-18.5%)

200us global quantum
Real time: 3.883 s (-11.1%)
Virtual time: 0.5578 s (+29%)
Realtime / virtual time: 6.961 (-31.1%)

1ms global quantum
Real time: 8.003 s (+83.3%)
Virtual time: 2.0620 s (+377%)
Realtime / virtual time: 3.881 (-61.6%)
```

As a note that is not really comparable, running the same thing on a cycle accurate gate level simulator:

```
Real time: ~11 hours (+906701%)
Virtual time: 0.43735613 s (+1.2%)
Realtime / virtual time: 90544 (+896020%)
```

5 Performance estimations

5.1 Performance estimations CPU

TLMu provides the main emulator system (e.g TLM-2.0) with accounting of it's execution time. The timing includes DMI access costs.

The CPU models provided by TLMu are high level models without internal details of the processor pipelines. Time is driven by the number of executed instructions. Another limitation in the performance model comes from how TLMu emulates CPUs, i.e from binary translation. Because TLMu only performs instruction fetches when translating code blocks and not when executing them, performance estimations may differ significantly from reality.

Despite these limitations, as seen in the previous section, with a 1ns quantum timer, TLMu manages to model the timing of the XOP operations with

98.8% accuracy at approximately 9000 times the speed compared to a cycle-accurate RTL simulator.

5.2 Performance estimations cache

TLMu does not provide modeling of caches. If such are needed, they can be provided by the TLM-2.0 system, but that will require TLMu to be used without any internal devices, e.g with all the devices provided by the TLM-2.0 system or by another TLMu instance. This will significantly decrease the performance of TLMu systems.

6 Software debugging

TLMu can be configured to provide a built-in GDB stub allowing debugger connections to be established to the individual CPU emulators. The current implementation has the GDB stub running in the same thread as the CPU core, meaning that in TLM-2.0 systems, only one CPU may be interactively controlled at a time.

It is also possible to generate per CPU execution traces.

References

[1] Fabrice Bellard **QEMU, a fast and portable dynamic translator**, 2005.