Techniques for Speeding Up Virtual Platform Simulators

Vishal Goel, Amit Nene

Presented by: Tor Jeremiassen

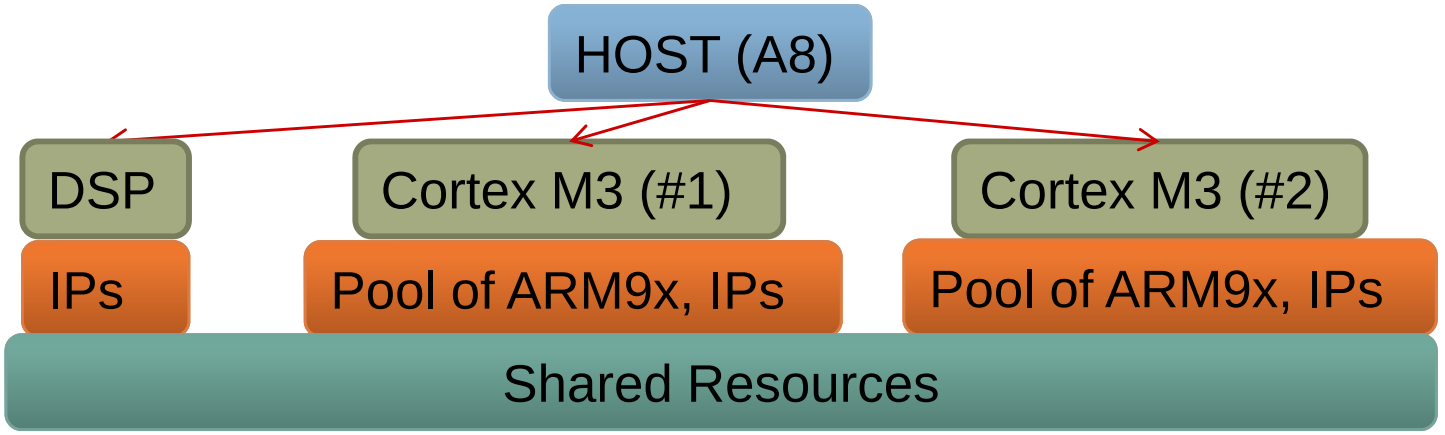
Texas Instruments

Overview

- Problem Statement
- Speedup Techniques
- Results
- Challenges related to SystemC and/or future work
- Conclusion and Q&A

DM816x Virtual Platform Simulator

- Highly integrated video SoC targeted at networked, HD video products
- 2 application level processors: CortexA8™ (©ARM Ltd) + TI DSP
  - 8 other ARM CPUs as slave/control processors
  - 100+ IP models including Programmable HD video image processing engines, HD display subsystem, Power/Clocking module, etc
- Role of CPUs
  - CortexA8 ARM manages high level OS such as Linux or WinCE
  - DSP runs audio functions & differentiating signal processing algorithms
  - Slave processors are used for encoding or decoding HD streams
- SW Architecture
  - Host ARM runs Linux and controls boot of DSP and 2nd level CPUs.
  - Second level CPUs control various IPs and control boot of next level CPUs
  - SW running on CPUs mostly independent from each other (DSP: audio, M3: video)



Problem Statement/ Challenges

- Speed of the Simulator
  - Expected:10s of MIPS at the system level
- End to End app
  - Linux Boot-up; followed by end to end real application:
    - video capture, multiple channels of video encode/decode and display of HD streams containing at least 30 frames
    - Needs to run in few hours.

Approaches Used

- Incremental Simulator Development
- Quantum Control
- Compiled simulation or Binary translation
- Higher-abstraction models
- Controlling clock to idling CPUs
- Record and Playback
- Virtualization of the OS File System
- DMI: Direct Memory Interface

Incremental Simulator Development

- A Virtual Platform is an integration of sub-system simulators,
  - Akin to the final software stack
- Most pre-Si users concerned w/only one or two CPUs/sub-systems
  - Sub-system models run faster and consume less memory
- Strategy
  - Deliver different integrations: sub-system sims; besides the complete VP sim.
  - Need: Simple and intuitive integration methodology to selectively enable/disable as well as change abstraction levels of components.
- Existing Integration methodology:
  - Simple text input for dynamic system construction: **`DSP=OFF; HOST=OFF; HDVICP=OFF;`**
  - Infrastructure handles construction of desired components
  - Loose connections (due to missing IPs) are tied-off to generate appropriate warnings if accessed during run time, without crashing the simulator.
    - *SystemCs bind() with "0+" ports ensures that the systemC engine does not complain about loose connections; BUT will crash simulation if used*
    - *Infrastructure should provide useful messages (such as, name of the IP that is missing) when the loose ports are accessed.*

QUANTUM Control

- Frequent synchronization (context switches) is very slow.
- Let each subsystem run for multiple cycles (similar to TLM-2 LT)
  - Functionally independent sub systems run at some higher quantum
  - Thus, Synchronization between sub system is maintained loosely.
- Impact of increasing the Quantum
  - Base assumption: Applications must be event based and not assume any timing in software. [Reasonable for most apps running over OS].
  - Performance: Applications ran faster with increasing quantum lengths up to about (Q~200), beyond that little performance gain was observed.
  - Functionality: Longer quantum lengths would cause applications to hang, indicating violations of apps timing/synchronization assumptions.
  - Apps do need synchronization. So increasing a quantum too much makes the app wait more, and may start decreasing performance.
  - Thus there is an upper bound beyond which quantum length increases have no or negative performance impact.
- Optimal Quantum for different apps for the same platform is different
- Challenge: Can Quantum Keeper be leveraged to dynamically determine (near) optimal quantum for a target application

Compiled simulation or Binary translation

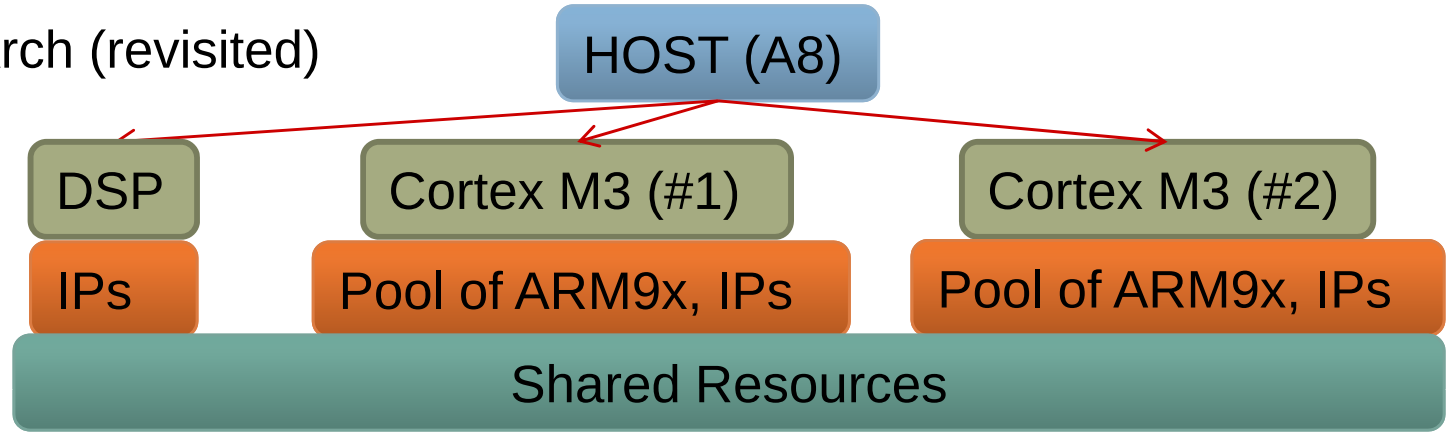
- Approach to improve speed of CPU models
- Map instructions of CPU model to Host CPU (x86) instructions
- In Standalone environment, Speed gains: ~10-30x.
- In Multicore SoC platform however, the gains are less impressive due to:
  - Small quantum (need for synchronization)
  - Overhead of memory transactions going via interconnect etc as against tightly coupled memories in standalone CPUs
- DMI can help
  - DMI or direct memory interface could reduce the memory transactions going out of the CPU boundary; maximizing the benefits of compiled simulation

Higher Abstraction models

- Maintain s/w programmer's view: No need to model behavior exactly matching H/w
- IP models that benefited from higher abstraction
  - Video encoding/decoding & Image processing IPs: Instead of modeling pixel-by-pixel model, entire frame level processing finished in one go
  - Bit exactness of Display Sub system IPs is compromised; anyways it could not be detected by bare eyes.
  - DMA engines: Instead of cycle by cycle transfers, block transfers
  - Functional Buses and Interconnect: Instead of modeling the complete bus protocols, used TL4 level buses
  - Blocking and bulk transfers instead of breaking/queuing the transactions into smaller chunks. E.g. Transfers as large as 4KB (Linux page size)
- Careabouts for raising abstraction
  - Some notion of timing must be maintained. E.g. DMA: transfer may happen immediately; but completion-notification should be delayed for some approximate cycles

Clock Control

- DM816x S/w Arch (revisited)



- Typical SWflow
  - Linux boots on Host ARM. During this, other CPUs are held in Reset;
  - Linux loads the OS images on DSP and other CPUs, and brings them out of Reset, which in turn, load the images for next level of CPUs
- Observations:
  - While slave CPUs are held in reset, clocking them is redundant and unnecessarily slow.
  - Similarly, while CPUs are in “Wait for interrupt” state; clocking can be avoided.
- Improving simulation speed for such a platform
  - We model the hardware equivalents of Clock/Reset module (PRCM)
  - Through programming PRCM, software itself takes care of controlling the clocking and execution of it’s slave CPUs.
  - Simulation Execution Engine understands that if a CPU is held in reset or clock is gated, no host-cycles are spent on simulating that CPU.

Record and Playback

- Next level of raising the abstraction by altogether removing an IP.
- Works when the s/w developer is debugging the top level software stack, and does not care about the lower most IP models behavior.
- How it works
  - Run the simulation one-time. Record the signals (In/out) of the sub-system, which user is not interested in debugging, as VCD trace.
  - During Playback, remove the targeted sub-system, and replace it with a model, which plays back the pre-recorded VCD trace.
  - Top level s/w can not distinguish between response from real IP and VCD trace playback module
- Example: Top s/w asking underlying models to do video encoding of a certain sequence of frames
- Approach is not for gaining speed; but to improve debug cycles for subsequent runs
  - Poor man's solution for save & restore.

Virtualization of the OS file system

- Very important feature for accelerating the s/w development for this chip.
- Is equivalent to providing higher abstraction model of Ethernet driver/SATA

Usage Pattern Without OS-FS virtualization:

1.Bundle the apps to be run on the Linux into the RAMDisk

2.Boot Linux: Load the RAMDisk and uboot on Host ARM & run the simulator.

3.Run the apps from /user/apps*

4.If app fails, rebuild the application with fixes, and **repeat all the steps** 1..3

Usage pattern With OS-FS virtualization

1.RAMDisk need not contain any app.

2.Boot Linux: Load the RAMDisk and uboot on Host ARM & run the simulator.

3.Map Host drive (e.g. C:/temp) to (say) “/mnt” older in Simulated Linux

4.Copy Apps to C:\temp.

5.Run the apps from /mnt.

6.If App fails, rebuild the app and goto step (4).

Virtualization of the OS file system (contd..)

- Mechanism:
  - Custom “IP model” mapped to reserved memory location.
  - OS linked with custom file system driver.
  - Future: Use Ethernet/NFS kind of model to enable same OS image on hw and simulator.
- Advantages
  - For any application failure, no need to bring down the simulator, or even simulated Linux.
  - Same application can be used for different input streams
    - Very important as input video streams with sizes of the order of MBs (or GBs) can be provided to the simulator without copy or pre-loading
  - Generated output on simulated Linux can be written to files, then moved to local drive for future reference.
- This feature also improves boot up time:
  - Smaller Ramdisk
  - No need to model full network stack or SATA.

DMI – Direct Memory Interface

- DMI between an IP and a memory model eliminates transactions across interconnects by providing direct pointers to memory.
- Usage in DM816x
  - "Local DMI" was used between IP models and local memories no pointers management, no invalidation required.
- Challenges:
  - Developing TLM2-DMI based infrastructure to efficiently manage DMI pointers with efficient memory usage.
    - Memory consumed by the infra is important because the same infra would be required at potentially every initiator:
      - E.g. in DM816x, 20+ masters are candidates for DMI
  - Search within DMI pointers: Efficient add/search mechanism across pointers to different sized blocks from different slaves (e.g. one IP/memory model manages 4KB pages; other: 64KB).
  - Ensuring correct "Invalidation of DMI pointers" wherever there are any dynamic address translation change (e.g. MMUs) between a slave(memory) and a master (CPUs/IPs)

Results

- Due to Sub system simulators and other improvements
  - Each s/w component could be developed independently and made available in timely fashion for s/w integration.
  - Faster simulations offered multiple iterations/debugging cycles/day.
- Due to fast platform simulation, users could run streams with HD frames pre-Si; rather than just using QCIF (smallest size) frames.
  - Porting the complete software stack for major use cases from simulator to hardware took less than 30 days.

Results (cont..)

USE CASE	SPEED
Bring-up of the 10-core Simulator in CCSv4 IDE (© TI)*	~30 seconds on Windows and Linux
Linux Boot time	~ 1 minute
Video Processing (Encode/Decode)	3-4 sec /QCIF H.264 decode 10-12 sec /QCIF H.264 encode 3-4 min /HD H.264 decode 10 min /HD H.264 encode
Using Playback	1 sec /QCIF H.264 decode 1 sec /QCIF H.264 encode 30 sec /HD Frame H.264 decode 30 sec /HD Frame H.264 encode
Display	3 seconds per HD frames
End to end run of the SW stack on the system simulator	200 D1 frames captured, processed & displayed in over 3-4 hours.

*CCSv4 (Code Composer Studio) is the Eclipse based latest debugger from Texas Instruments

Challenges related to SystemC / Future Work

- Dynamically deciding optimal quantum for various single CPUs or multi-CPU sub-systems – Quantum Time Keeper
- Zero-performance-overhead for idling CPUs (idle loop recognition)
- Efficient DMI pointers management/deployment
- Integration methodology support enabling incremental s/w develop.
- Additionally:
  - Leveraging multi-core hosts (e.g. Intel Dual/Quad core machine):
    - Simulation partitioning or finer grained parallelism within simulator.
  - Higher abstraction models
    - Further abstraction of models is possible, but not w/o having to abstract out the Hardware abstraction layer (HAL) which makes maintaining SW across pre and post Si a challenge.
    - Host based simulation: Use host OS instead of OS running on the simulator for Apps development / SW integration for Host CPU

Conclusion

- Incremental simulator development can speed up s/w development (sub-system models).
- Techniques like Idling clocks, optimal Quantum, Compiled Simulation, DMI, raising the abstractions helped improve simulation performance.
- Techniques like virtualization & playback improved debug cycles and indirectly contributed to (compensated for) simulation performance.
- SystemC with multi-threading/Multi-core support and infra pieces like dynamic quantum and solving practical problems in DMI will greatly improve SystemC based simulations.