A Generic Tool-Set for SoC Multiprocessor Debugging and Synchronization

Andreas Wieferink, Tim Kogel,
Rainer Leupers, Heinrich Meyr
Integrated Signal Processing Systems
Aachen University of Technology
Aachen, Germany
wieferink,kogel,leupers,
meyr@iss.rwth-aachen.de

Achim Nohl, Andreas Hoffmann
CoWare, Inc.
San Jose, CA, USA
achim.nohl,andreas.hoffmann@coware.com

Abstract

Current and future SoC designs will contain an increasing number of programmable units. To be able to tailor and debug these processors in their system context at the highest possible overall simulation speed, we propose a methodology and the necessary tooling for a multiprocessor debugging environment which allows a flexible runtime trade-off between observability and simulation speed.

This approach has been applied on a complex SoC case study.

1 Introduction

Programmable processor cores plus their corresponding application software form a central component of most of today's embedded SoC designs in different areas like telecom, automotive, and consumer electronics. The ever increasing computational power silicon offers, already now and even more in the future, will be filled mainly by software blocks because of the much higher flexibility and the easier reusability compared to hardwired ASIC blocks.

To develop an optimal SoC design, for every software block an individual trade-off between computational efficiency and remaining flexibility should be evaluated. This leads to heterogenous SW blocks ranging from general purpose microprocessors cores, powerful DSP kernels up to optimally tailored Application Specific Instruction set Processors (ASIPs).

It is widely accepted that simulation of the SW blocks in the system context is compulsory for coping with the ever increasing complexity. In the feasible future of platform based design, simulation will be the major workhorse for all phases in the design flow: first, during the architecture exploration phase, to determine and develop the optimal programmable HW platform as well as, second, during the full embedded SW (eSW) development phase.

Especially for the initial design phases, it is absolutely necessary to simulate the whole system as early as possible to avoid expensive redesign loops. Thus, fast HW/SW cosimulation, including an efficient communication modeling, is compulsory. Even more, to get a maximum of valuable information from a running simulation, it must be possible to select the degree of observability dynamically.

To enable user friendly debugging and online profiling of the eSW and its platform, the user should always - at simulation runtime - have the possibility of getting the full SW centric view of

an arbitrary SW block. All other SW blocks that are currently not considered should still run at maximum speed.

This paper describes a methodology and the necessary tooling to offer such a runtime trade-off to the user.

The following section presents related work on the field of HW/SW cosimulation. The third section then presents the concept of retargetable simulator frontends and how they are applied on single processor standalone simulations. The technique to keep all SW blocks of a multiprocessor SoC simulation synchronized to each other independent of the user's debugging behavior is topic of the fourth section. Then, in section five, the additional tooling is described that enables the user to switch his debugging scope arbitrarily during runtime.

Having these tooling libraries available, the sixth section then describes how they are applied on a complex real world example.

Finally, the paper is completed by a short conclusion.

2 Related Work

In the wide area of HW/SW cosimulation, a lot of research and development work has already been done.

Commercially available HW/SW cosimulation tools like Seamless [16] and Eaglei [5] provide a coupling of processor models to HDL simulators. Since the latter are very slow, these environments can mainly be used for interface verification, but they are not eligible for architecture exploration nor embedded software development.

The new generation tools like CoWare N2C [13], CoCentric SystemStudio [4] or handcrafted coupling using TLM SystemC [17] enable a faster simulator coupling since the HW can be simulated on a higher level of abstraction. But so far these tools do not provide comfortable eSW debugging capabilities: either there is no observability of the SW blocks possible at all or the debugger GUI of the processor vendor becomes the permanent bottleneck for system simulation. The tool-set presented in this paper solves this shortcoming for these environments.

Some research work already has been done on the trade-off between accuracy and simulation speed (e.g. [9]). In contrast, our scope is the trade-off between observability and simulation speed.

High simulation speed is absolutely necessary but not the only need for a relevant system simulation environment. We also need retargetability for the processor debugger and simulator to be able to integrate user defined SW modules as well.

Retargetable SW debuggers like ddd [7] can cope with SW running on different CPU types, but since they only focus on C-Source Debugging, the abstraction level is too high to debug issues related to the processor architecture the SW is running on.

Retargetable processor simulators have been dealt with in several publications already. There are multiple processor description languages like EXPRESSION [1], Sim-nML(FSim) [11], ISDL (XSSIM) [6] and MIMOLA [14]. In principle, a multiprocessor debugging environment as presented here could be created on top of those who provide a cosimulation interface and a bus interface. Our work is based on the LISA Processor Design Platform [2]. The generated simulators offer very flexible cosimulation facilities, can simulate very fast, and enable efficient architecture exploration by versatile runtime profiling capabilities.

Mentor's XRAY [12] debugging environment already contains the possibility of multiprocessor debugging even with different processor targets. Their main scope is debugging already existing boards and processors. For host simulation, XRAY supports only a few fixed simulator backends.

This makes it valuable for later design steps using these fixed and already existing processor architectures. But architecture exploration applying user defined retargetable processors is not possible here.

In contrast, the multiprocessor debugging tool-set presented here especially supports the early embedded processor design steps by a) supporting any user defined or predefined processor model and b) providing runtime profiling information for optimally evaluating the applied processor architecture as well as the application software.

3 Retargetable Standalone Simulation

Our flexible multiprocessor debugging environment is based on retargetable standalone simulators, satisfying several constraints. This section presents the necessary or at least recommended features of a well suited underlying processor simulator environment.

3.1 ISS Requirements

Any Instruction Set Simulator (ISS) to be applicable here must strictly be separated into two parts: an architecture specific simulator backend and a generic, mostly graphical debugger frontend (Fig. 1). The simulator backend has to encapsulate all its functionality into a well defined architecture independent Application Programming Interface (API). The debugger frontend can then - either directly or via Inter Process Communication (IPC) - access the backend through the API to dynamically retarget to the respective backend architecture and display its state. The API functions can be separated into two main groups: a) simulation control and b) processor observation/manipulation.

These requirements are fulfilled by state of the art ISS like the ARMulator [3]. Automatically generated retargetable processor simulator backends as provided by the LISATek tool-suite [10] [13], meet these constraints for arbitrary processor architectures and for multiple abstraction levels (i.e. instruction accurate, cycle accurate).

3.2 Observation/manipulation API with Generic Resource Access

The observation/manipulation part of the remotely accessed API contains methods for generic resource access, breakpoint and watchpoint handling, profiling data access etc. The generic resource access mechanism allows the frontend GUI (Graphical User Interface) to adapt its resource windows and memory display to the respective processor hardware.

When opening the architecture, the debugger frontend initially does not know anything about the simulator backend. A set of API functions allows the frontend to get to know the number of resources and then to request the type and properties for each of them. Memory blocks, General Purpose (GP) registers, pipeline registers, ports: all these types can be managed by the same uniform resource data structure received through the backend's API. Having this information, the debugger's windows (Fig. 1) can be configured: every memory resource is added as an additional tab to the memory window, the general purpose registers and pipeline registers are appended to the respective window. These windows can display and edit the current resource state as well as manage watchpoints.

The disassembly window always highlights the current instruction and allows the user to set and reset breakpoints. If debug information is provided by the current application executable, a source code window offering the same breakpoint features is created as well.

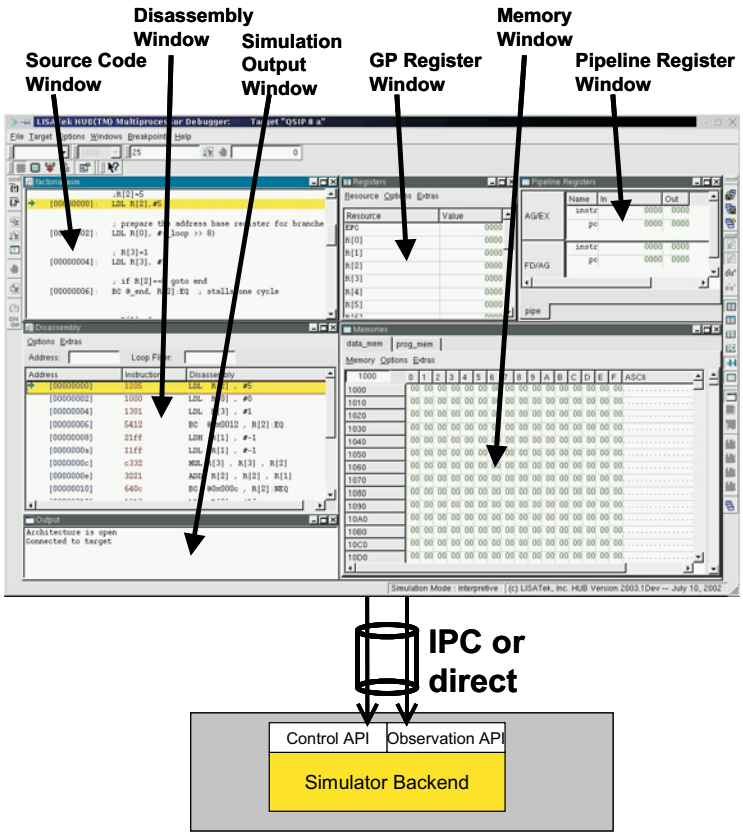

Figure 1. Standalone simulation with debugger GUI

If the respective features are enabled in the current simulator backend, further windows for application and architecture profiling, application and architecture debugging can be created, or alternatively, the existing windows are extended. A state of the art debugger frontend also enables the user to modify the displayed values and can manipulate the current state of the processor simulator backend accordingly.

3.3 Control API

The control part of the remotely accessed API allows the user to manually or automatically invoke and stop processor simulation cycles on several degrees of granularity. For standalone processor simulation, several simulation control modes are known. The main distinction can be made between interactive and non-interactive mode.

In this context *interactive mode* means the user has full control over the respective processor simulator. The control can regularly be handed back to the user on several levels of granularity: clock cycles, assembly instructions or source code instructions (i.e. C/C++ source). Simulation is continued only on an explicit new invocation by the user.

In contrast, *non-interactive mode* here means the processor simulator does not need any special user input, but is solely invoked automatically by the system environment it is embedded into. There are several non-interactive modes as well, differing only in the degree of observability still possible: automatic frontend display refresh on completion of every instruction, frontend display refresh only

on demand, or the simulator running in the background with no frontend connected at all.

A state of the art debugger enables free choice of the simulation mode at any time, depending on the user's current interest. Besides a manual switch between the simulation modes, this also can be initiated automatically e.g. if a breakpoint is hit while being in non-interactive mode, the simulator is turned to interactive mode automatically.

4 Multiprocessor Synchronization

In this section, we generalize the single-debugger-frontend - single-simulator-backend constellation to multiple debugger frontends and multiple processor simulator backends.

4.1 Goal

For user friendly multiprocessor debugging, the user should be able to open as many instances of a generic debugger frontend which - at simulation runtime - can be retargeted to those processors the user selects.

To keep the potential for highest possible simulation speed in a complex SoC simulation, the data exchange between the processors and their respective SoC environment - which is always necessary independent of the user's system observation - must be highly optimized. This can only be guaranteed if all processor simulators reside within one executable on the host such that no slow Inter Process Communication (IPC) is necessary for this task.

IPC is applied if and only if the user decides to observe or take control over a certain processor simulator.

4.2 System Integration Concept

By encapsulating all ISS functionality into a unified API, instantiating any number of ISS types within the same simulation executable is possible simply by linking the respective architecture libraries to the system simulator. This guarantees the high simulation speed: no IPC is necessary for communication within the system simulation if no frontend, i.e. user interaction, is desired (Fig. 2).

On the other side, the user may decide to observe and control the execution of a certain processor. For that purpose, he can create an IPC connection between a generic frontend instance and the processor backend to access its API (Fig. 3). This IPC connection transfers Control and Observation/manipulation information (CaO).

Thus, the remote retargetable debugger frontend instance offers all observability features for multiprocessor simulation as known from standalone simulation. Even resources external to the processor simulator backend like peripheral registers and external memories can be visualized and modified through the observation/manipulation API.

For good observability, every frontend can request backend data to display the current processor state without influencing the remaining processors or the system simulation synchronization. Analogously, using the manipulation API, we can modify the state of a processor simulator backend without directly influencing the others. For user controllability, in contrast, we need some more sophisticated synchronization between the user control coming from the frontend and the system simulation control coming from the simulation scheduler (Fig. 3).

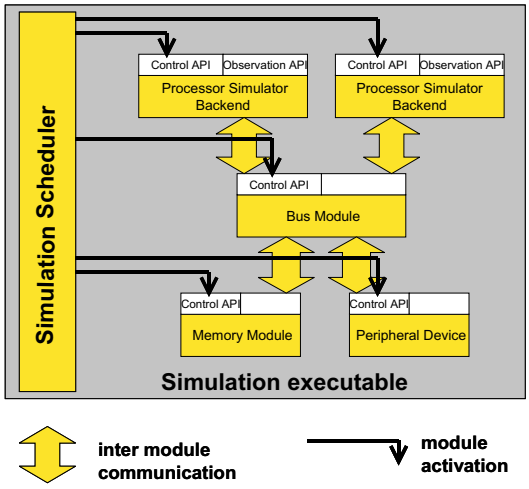

Figure 2. System simulation executable without debugger GUI

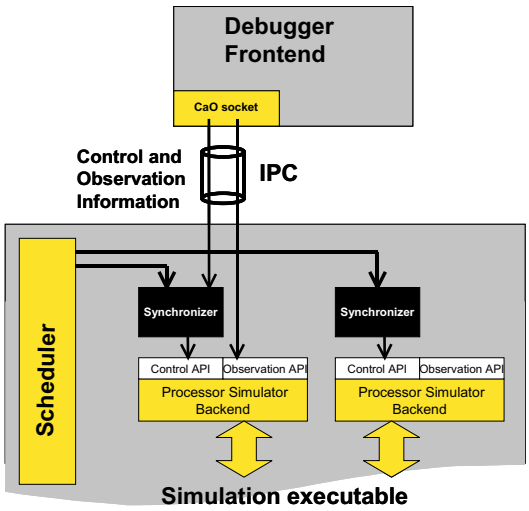

Figure 3. Synchronizing user and scheduler control

4.3 The Synchronizer Unit

The standalone simulation control modes described in section 3 are available for every processor core during initial development. When system integration starts, it is reasonable still being able to choose all the above interactive and non-interactive simulation modes individually for each processor of the system.

The black boxes in Fig. 3 indicate the problem that 2 instances can access every Control API and need to be synchronized.

The control flow for the most common synchronizing mode is sketched in Fig. 4. From the upper side, the control is handed over from both sources: The simulation scheduler at the left; the debugger frontend (respectively the user) at the right. Both instances have to cope with the situation that the call can be blocking, i.e. it takes some time until control is handed back to the respective unit.

The synchronization is done such that the debugger frontend only gives permission to actually execute cycles on the backend simulator. Depending on the user command, this permission can be for one cycle, an entire assembly instruction, or even more.

When the simulation scheduler activates the processor module, first it is checked if a frontend is connected at all (Fig. 4). If this is not the case, the simulator backend can be forwarded a cycle since it is running in the background now. If a breakpoint or watchpoint is hit, all frontend instances can be notified about that.

Otherwise, if a debugger frontend is connected on scheduler's activation, it has to be checked if the frontend already gave permission to execute the next cycle. If this is not true, the execution thread blocks until this permission is available. Then the processor cycle actually can be invoked. After that, it is checked if the permission given by the frontend is expired. This decision depends on the kind of permission the frontend provided. If the permission is expired, or if a breakpoint is hit, or if the frontend should refresh its display, then the frontend instance is notified about that.

In all of the above cases, the control is handed back to the simulation scheduler so it can invoke the other system modules, i.e. the remaining processor backends, in the same manner.

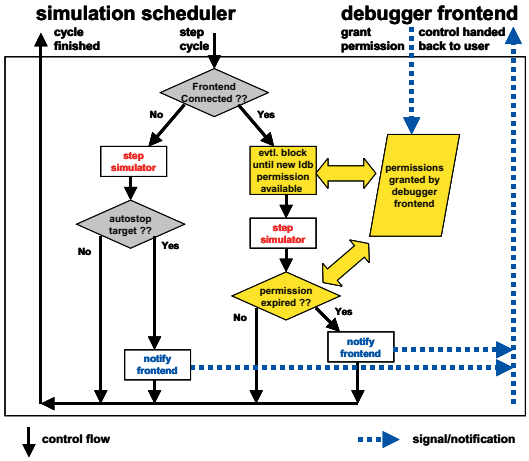

Figure 4. Synchronizer control flow

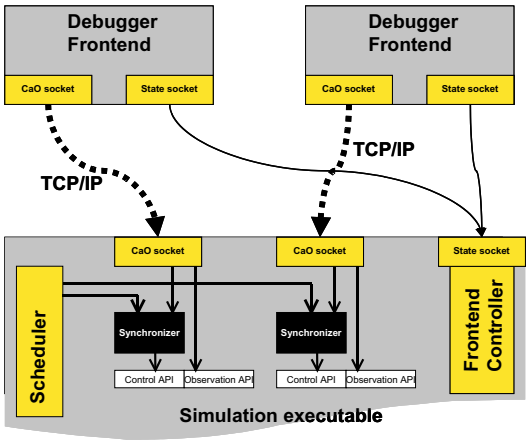

Figure 5. Dynamic Connect Debugging

4.4 Multiprocessor Simulation Modes

By attaching such a synchronizer device to every processor simulator backend, all simulation modes known from standalone simulation still are available for multiprocessor simulation and can be selected independently for each processor. If one processor is in interactive mode, the whole system simulation is blocked as soon as new user input is expected at the respective frontend. If there are multiple processor backends in interactive mode at the same time, the control is passed to the respective debugger frontend in turn. Even if no processor simulator backend is in interactive mode at a time, all breakpoints or watchpoints, independent in which processor, can be considered and then can stop system simulation.

5 Dynamic Connect

For any point in time during simulation and individually for every processor core, the user should choose dynamically the optimal trade-off between good observability/controllability and high simulation speed. For enabling this dynamic connect capability, the tooling described above needs some extensions.

5.1 TCP/IP

The synchronizing mechanism described above manages both situations: a debugger frontend being connected or the simulator backend running without user interaction.

To switch between these two states at runtime it is necessary to dynamically create and destruct the CaO IPC connection. By choosing TCP/IP for the IPC, implemented using the platform independent QT library [19], we can launch the frontend instances not only on a different host, but even on a different host operating system (OS).

5.2 State Information Exchange

To manage the information how to setup and tear down the TCP/IP connections, a global connection management is necessary (Fig. 5).

For that purpose, a frontend controller is added to the simulation executable. The controller maintains global information about all simulator backends that should be published to all frontend instances: host/port information to connect to the CaO socket of the respective processor simulator, the current availability for dynamic connect, as well as global backend information like current processor load. Additionally some system simulation control panel can be managed by this device.

The frontend controller offers this information to a simulation state socket, which is permanently connected to every debugger frontend. By that, every frontend instance can display the current global state of every processor backend, provide a system simulation control panel, and provide the user with the ability to connect to the CaO socket of any simulator backend currently available.

5.3 Socket watching

To any time, the debugger frontends must have the possibility of accessing the simulation executable's TCP/IP sockets, even if only for refreshing the display.

Thus, all sockets of the simulation executable must be observed all the time, independent if the system simulation is running, a processor backend being in interactive mode is blocking, or some other component, e.g. an external HW simulator or an external system simulation control panel is holding the control. In the latter case, the sockets can not necessarily be served by the main event loop. For still being able to integrate the simulator backends into a third party system simulation environment (e.g. SystemStudio provided by Synopsys), it is necessary to make the dynamic connect backend thread safe to be able to operate it from within a separated thread or directly from the host operating system.

In terms of Fig. 4, the control flow at the right side, which serves one CaO socket of the simulation executable, is possibly not done from within the main thread that executes the system simulation itself, but by an additional socket watcher thread.

6 Case Study

This methodology and tooling has been applied on an IP forwarding platform design, containing three programmable units (Fig. 6). For developing these ASIPs, the LISA tool-suite provided by LISATek [10] has been applied. For system integration, Synopsys' CoCentric SystemStudio [4] has been used.

6.1 The system architecture

IP forwarding is a central part of the IP network infrastructure. This challenging application domain combines sophisticated functionality for QoS support with highest performance requirements: at OC-192 wire speed (corresponds to 10Gbit/s) the timing budget for the processing of an 64Byte packet is as short as 52ns.

IP Forwarding with QoS Support requires the following functional blocks (Fig. 6): The *Parser* performs sanity checks on the incoming packet and provides the following units with a unique packet descriptor (PD), which holds all the relevant header information of the respective IP packet. The *Route Lookup Unit* (RLU) performs forwarding of IP Packets based on the longest match table search algorithm. The forwarding decision is based upon destination IP address and the routing table. The *Classifier* classifies incoming packets into Classes of Service (CoS), so the packets are processed according to their negotiated Quality of Service parameters by the following blocks. The *Meter* measures the IP packet rate and drops packets exceeding the negotiated traffic characteristics

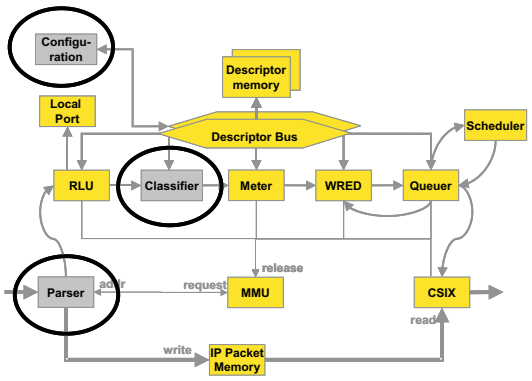

Figure 6. IP Forwarding Platform

more observability

more complex system ↓ / processor cycles per second	no observability run free / disconnected	disassembly + source code updated each cycle	full display incl. memory + registers updated each cycle
Standalone Parser	1 600 000	100	100
3-Processor Subsystem	22 000	2.6	1.6
Complete IP Forwarding Platform	6 000	2.4	1.4

Figure 7. Simulation Performance

to protect the succeeding queuer unit from greedy traffic streams. The *WRED* unit drops additional packets employing the weighted RED algorithm [15] to avoid throughput degradation in a TCP friendly manner due to congestion. The *Queuer* stores IP packets according to their CoS until they can be forwarded to the CSIX unit. The *Scheduler* decides on the basis of the priority and the actual fill status of the packet queues in the queuer unit. Finally the *CSIX* unit segments IP packets into fixed size packets in compliance with the standardized CSIX bus protocol [18] to interface the switch fabric. The *Memory Management Unit (MMU)* performs the memory allocation and deallocation of the IP- and PD-memory. The *Configuration* Unit sets up the lookup tables and configures the other system modules, according to their remaining flexibility.

Having modeled these modules on a high abstraction level according to the GRACE++ [8] methodology, they could be debugged and profiled using the CoCentric SystemStudio Tools&Monitors [4], the GRACE++ token MSC debugger [8] and the GNU DDD SystemC debugger [7]. As a result of the system level profiling, and driven by flexibility needs, it has been decided to develop application specific processors for the three system blocks *Configuration*, *Parser* and *Classifier*. In this section, we exemplarily describe the development of the *Parser* unit, applying the methodology and tooling presented in this paper.

6.2 Parser NPU Development

To get an optimally tailored Network Processing Unit (NPU) quickly, it is a good idea to start with an already existing generic NPU processor model and to refine it by adding special instructions needed for the respective task.

For the *Parser* unit, to build up the packet descriptor, we need to do a lot of bit masking and shifting to arrange the data fields. Thus, a new instruction has been added that can do right that within one cycle: cutting out a bit sequence from a source field, and copy it into a different bit location of a target register. This new processor instruction can quickly be verified using the standalone debugger (Fig. 1) by loading a short test application.

After the tailored NPU model has been created and roughly verified this way, the *Parser* application software development can begin. According to the already existing abstract system model of the *Parser* unit, the application software is created in assembly. It can roughly be verified in the standalone simulator as well. The necessary data and stimuli normally coming from the system environment, like IP packet contents or packet arrival signaling, can be included into the application image or entered manually using the simulator's observation/manipulation API.

6.3 Parser System Integration

As soon as the *Parser* application seems to run standalone correctly, along with the necessary stimuli getting more and more complex, it is time to integrate the SW module into the system context. For this purpose, the existing processor simulator backend is configured to use external buses for system environment dependent tasks. Thus, the same application software tested above now works in system context (Fig. 2).

Connecting the retargetable debugger frontend to the *Parser* processor (Fig. 3) during system simulation allows user friendly debugging and verification: as long as the *Parser* is not working correctly in the system context, both can be debugged using the debugger frontend: the processor model as well as the application code. Also the handshaking between 2 system blocks, e.g. between the *Configuration* processor and the *Parser* that is about to be configured can be observed very detailed by connecting a debugger instance even to both processors concurrently when necessary.

Connecting the retargetable debugger frontend to the *Parser* processor also makes valuable run-time profiling data visible to the user for effective architecture exploration. E.g., as the built-in loop counter indicated, the *Parser* processor often was unsuccessfully polling on the bus for the arrival of a new IP packet. The bus monitor provided by the system simulation environment confirmed that: There was a lot of unnecessary bus traffic caused by polling for a new packet. So the NPU processor model was extended for interrupt capabilities, and a respective interrupt service routine was added to the application code.

Independent if verification or profiling make some changes necessary: major modifications to the system block may initially best be checked standalone, but very soon the realistic system simulation environment will be compulsory for meaningful simulation again.

6.4 Simulation Performance

The simulation performance that could be retrieved on an Intel Celeron 1300 MHz host for different configurations is outlined in Fig. 7.

From top to bottom, the overall system complexity increases: The *Parser* processor running standalone (Fig. 1) is shown in the first table line. The simulation mode is interpretive with all profiling features enabled. The system profiled in the second line additionally contains the two other processors as well as a common bus and the shared data memory. Here the processors already run from within SystemStudio with the processor debugger frontend being connected via TCP/IP. The last line deals with the whole system shown in Fig. 6. The remaining system modules are of a much higher level of abstraction.

From the left to the right, the degree of observability increases. The first configurations give the user no observability and almost no controllability (except for interrupting the running simulation). The second and third column's configurations update the *Parser* debugger frontend display after completion of each instruction. In the second column this is only done for the source code and disassembly window; in the third column the resource windows are updated as well.

By comparing the first column to the others, it can be seen that the performance penalty for the automatic frontend refresh is quite high. But this is no problem at all since the user could follow the GUI's highlighted disassembly and source code lines only for low simulation throughput of a few cycles per second anyway. And the user can switch back to full simulation speed indicated in the respective left column at any time. It may even be senseful to slow down the standalone simulation debug mode intentionally to allow the user easier following the standalone simulation.

As expected, the simulation performance also decreases for more complex systems. In the first

column, the decrease only depends on the system complexity itself. Especially the SystemC simulation scheduler necessary from the second system configuration on significantly slows down the overall performance. In the second and third column, an additional performance loss compared to standalone simulation occurs since the frontend display can only be refreshed using slow IPC. But even for the complete SoC model, the speed is still high enough to allow user friendly debugging.

7 Conclusion

Together with the debugging and profiling facilities for abstractly modeled SoC modules and SystemC HW blocks (CoCentric SystemStudio Debugging & Monitors [4], GRACE++ token MSC debugger [8], GNU DDD SystemC debugger [7]), complex systems can be coped with conveniently on all levels of abstraction down to cycle accuracy using the presented multiprocessor debugging environment.

References

[1] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[2] A. Hoffmann, T.Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A novel methodology for the design of application specific instruction-set processors using a machine description language. *IEEE Transactions on Computer-Aided Desig of Integrated Circuits and Systems*, 20(11):1338–1353, November 2001.

[3] ARM. *http://www.arm.com*.

[4] CoCentric System Studio. *Synopsys, http://www.synopsys.com*.

[5] Eaglei. *Synopsys, http://www.synopsys.com*.

[6] G. Hadjiyiannis and S. Hanono and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.

[7] GNU Data Display Debugger. *http://www.gnu.org/software/ddd*.

[8] GRACE++ Project. *http://www.iss.rwth-aachen.de/Projekte/grace/index.html*.

[9] K. Hines, G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proceedings of the Design Automation Conference (DAC)*, 1997.

[10] LISATek. *http://www.lisatek.com*.

[11] M. Hartoog J.A. Rowson and P.D. Reddy and S. Desai and D.D. Dunlop and E.A. Harcourt and N. Khullar. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.

[12] Mentor XRAY. *Mentor Graphics, http://www.mentor.com/*.

[13] N2C. *CoWare, http://www.coware.com*.

[14] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, Jan. 1999.

[15] S. Floyd, and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):379–413, 1993.

[16] Seamless. *Mentor Graphics, http://www.mentor.com/*.

[17] T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[18] The Network Processor Forum. *founded by CSIX/CPIX members in 2001 http://www.npforum.org*.

[19] The QT Library. *http://www.trolltech.com*.