Proceedings of the 2010 Winter Simulation Conference
B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, eds.

SEAMLESS HIGH SPEED SIMULATION OF VHDL COMPONENTS IN THE CONTEXT OF COMPREHENSIVE COMPUTING SYSTEMS USING THE VIRTUAL MACHINE FAUMACHINE

Stefan Potyra
Matthias Sand
Volkmar Sieh
Dietmar Fey

University of Erlangen-Nuremberg
Martensstr. 3
Erlangen, 91058, GERMANY

ABSTRACT

Testing the interaction between hard- and software is only possible once prototype implementations of the hardware exist. HDL simulations of hardware models can help to find defects in the hardware design. To predict the behavior of entire software stacks in the environment of a complete system, virtual machines can be used. Combining a virtual machine with HDL-simulation enables to project the interaction between hard- and software implementations, even if no prototype was created yet. Hence it allows for software development to begin at an earlier stage of the manufacturing process and helps to decrease the time to market. In this paper we present the virtual machine FAUmachine that offers high speed emulation. It can co-simulate VHDL components in a transparent manner while still offering good overall performance. As an example application, a PCI sound card was simulated using the presented environment.

1 INTRODUCTION

With the ever growing integration scale and smaller manufacturing sizes, it becomes possible to develop increasingly complex hardware systems. Rising computational power also allows the software to become more elaborate. When developing computational systems comprised of both hardware and corresponding software, the need for testing arises. Classical approaches like manufacturing prototypes and testing the software on these are only feasible at late stages in the hardware design phase. Simulation is a common practice that aids in developing hardware before a prototype implementation exists. Simulating comprehensive systems in order to test the integration of the software with the system in question creates the need for a high simulation performance. Especially if applications in the context of complete operating systems need to get evaluated, the simulation is not trivial. Virtual machines provide the ability to simulate comprehensive computing systems in a very efficient manner, so that complete operating systems can be run on the simulated hardware. However current virtual machines lack the possibility to co-simulate hardware models.

FAUmachine (FAUmachine Team 2010) is a virtual machine written in the programming language C, which is highly customizable. A new feature of FAUmachine allows for seamless co-simulation of components modeled in VHDL. The methods used to combine the virtual environment with a classical discrete event oriented simulation as used in HDL simulations, and nevertheless to achieve a very good simulation performance will be described in this paper.

The remainder of the paper is structured in the following way: In Section 2 an overview about related work is presented. Section 3 describes the simulation engine used for the virtual machine. Section 4 gives an overview about the paradigm used to model chips, components and high level structures like entire PCs. How both structural and behavioral VHDL code can be co-simulated within the context of the virtual machine FAUmachine is presented in Section 5. Section 6 shows an illustrative example that uses co-simulation of VHDL code in the context of FAUmachine: We implemented a PCI sound card both in real hardware using a prototype card and simulated the VHDL code used for the prototype card within FAUmachine. Section 7 summarizes the presented work and gives an outlook on future work.

2 RELATED WORK

In the context of simulating computer systems, a lot of models exist already. Examples include the simulation of network interaction (Szymanski, Liu, and Gupta 2003), (Bauer et al. 2003), (Kamal 2004) or memory hierarchies (Over, Strazdins, and Clarke 2005), (Iyer 2003), (de Farias et al. 2001). Common to these models is that they are typically reduced to certain aspects of interest. Consequently, they contain a very high level of abstraction compared to real hardware.

Using hardware description languages (HDL), discrete event simulation can be used to simulate components and computer systems. Both commercial tools like ModelSim (Mentor Graphics Corp. 2007) and open source implementations exist ranging from tools to perform analysis (Wilsey, Martin, and Subramani 1998) up to complete simulation environments based on using time warp implementations (Radhakrishnan et al. 1998). To simulate complex environments, this approach is impractical though: Modeling complete systems using HDLs results in a huge conceptual effort. Moreover simulating complex systems in a cycle accurate manner using HDL simulators results in a very poor performance, rendering system tests extremely inexpedient.

In the context of virtual environments, many virtual machines exist next to FAUmachine. QEMU (Bellard 2005) and Bochs (Source Forge 2001) are based on the technique of emulation. VMware (VMware Inc. 2001) and KVM (Kivity et al. 2007) make use of hardware assisted virtualisation. User-Mode-Linux (User Mode Linux Core Team 2004) and Xen (Barham et al. 2003) utilize para-virtualisation. These tools provide to some degree the ability to simulate complete computer systems in a very efficient manner.

With the goal of hardware/software co-design in the context of complex systems in mind, the common virtual machines show the following disadvantages: Virtual environments that make use of para-virtualisation can only be used with operating system support, which often requires to use modified kernels that are suitable for para-virtualisation. Commercial operating systems without access to the source code lacking such support cannot be used within these virtual machines. Likewise it is debatable whether results that have been created by tests using para-virtualisation support are transferable to real hardware. Using hardware assisted virtualisation implies that the workload will access the real hardware of the host system as often as possible. While this yields an improvement in performance, it makes repeatable runs impossible: The main source of non-determinism stems from interrupts generated by the hosting hardware, which are by definition not repeatable.

When considering hardware-software co-design, automated tests are an important tool to avoid regressions in the design. The ability to run automated experiments is only present with QEMU in a very simplistic form.

Finally the main goal of many virtual machines is to run popular operating systems at the best possible performance. Hence the design of these virtual machines mimics real hardware only to provide the necessary interfaces that operating systems make use of. Details of real hardware are often neglected as long as the workload can still run properly. This abstraction improves the simulation speed, but makes it hard to impossible to use these virtual machines for hardware/software co-design.

One noteworthy piece of related work deals with coupling various simulation frameworks including a virtual environment (Hatnik and Altmann 2004). The hybrid framework is interconnected using TCP/IP sockets. The presented method applies loose coupling, which allows to combine components that have a moderate to large lookahead time and simulate these within complete systems. However close coupling, like e.g. connecting a PCI card modeled within VHDL is infeasible using said approach.

3 SIMULATION ENGINE OF FAUMACHINE

FAUmachine (FAUmachine Team 2010) is an open source virtual machine for standard PC hardware and components of the shelf. A variety of different operating systems can be run on FAUmachine without being modified, including Windows, Linux, Dos, Free-, Net- and OpenBSD. FAUmachine can simulate standard Intel CPUs and IDE controllers, NE2000- and Intel eepro100 network interface adapters but also peripherals such as networking hubs, serial terminals, or modems. FAUmachine provides the ability to inject faults into the simulated components. Interaction with the workload can be automated to perform a large number of experiment runs. A virtual user can be simulated who performs the interaction. The automation includes the possibility to inject faults as well. This way, early fault injection experiments can be run, for example, with focus on the interaction of fault-prone hardware components with fault-tolerating software modules. In order to achieve a high performance simulation, FAUmachine uses the techniques of a just-in-time compiler (Höxer, Waitz, and Sieh 2004). Using this approach, the overall performance of FAUmachine is about 5-20% as good as the performance of the corresponding host system. Two simulation modes are implemented within FAUmachine: Interactive mode provides a virtual environment useful for interactive testing. Deterministic mode allows experiment runs to be conducted in repeatable fashion (Sand, Potyra, and Sieh 2009). FAUmachine is implemented in the programming language C. This section describes the simulation paradigm used within FAUmachine.

3.1 Components, Signals and Buses

A standard PC consists of different components, like a CPU, memory modules, or a graphic adapter. Signals, buses and bridges connect these components. For example the CPU connects to the northbridge using the host bus. The northbridge is connected to the memory modules with the memory bus. It is also connected to the graphics card with the PCI bus, or in newer systems with the PCI-express bus.

FAUmachine tries to mimic real hardware in its modeling paradigm. Within FAUmachine, hardware components are modeled as individual simulators. No component has knowledge about the internals of any other component. Likewise no component knows about the structure of the entire simulated system. To nevertheless have components interact with each other, signals are used to interconnect them.

FAUmachine provides a number of basic signals of different types. For example the `boolean` signal type can be used to model simple low/high states. The interrupt request line connecting the programmable interrupt controller and the CPU is one example that uses the boolean signal type. Wired-or or wired-and signals can be implemented using the `std_logic` signal type. Components can connect to a signal, in case they want to be notified if a signal value changes. To achieve this, they provide a callback function. Each change of the value of a signal results in calls to the callback functions of every connected component.

Buses consist of many signal lines. The PCI-Bus consists of 32 address/data lines and a number of control lines. Let's consider that the northbridge would like to forward an IO-write request to the graphics adapter. This would mean that two updates to the 32 address/data lines have to be issued to transfer the IO-port and the corresponding data, plus a number of additional signal changes in the control lanes. As each change in the signal value will result in a function call, the overall simulation performance would be very poor. Hence FAUmachine doesn't model buses as a group of signal lines. Instead we use a form of abstraction: Buses always use a specific bus protocol. A bus cycle denotes an entire transaction using that bus protocol. For example the PCI-Bus can transfer IO-Read, IO-Write, Memory-Read, Memory-Write, Config-Read and Config-Write transactions as part of the PCI-protocol. In FAUmachine we are using an abstraction for these bus cycles as follows: We use the protocol of the bus to create a new signal type, for example a PCI bus type. Instead of individual signals – and callbacks for individual signals – there exists only one callback for each bus transactions. The transferred address, data, as well as necessary control information (e.g. did a device respond to an address) are part of the function signature of the callback. Hence the aforementioned IO-Write transaction will result in only one function call for every connected component instead of more than 64 function calls. This allows for a great improvement in simulation speed.

Updates of signal values result in immediate callbacks in connected components. As a consequence, no simulation time passes between the signal value change and the notification of the connected components. However to simulate PC-Hardware correctly, delays are inevitably necessary. For example the programmable interval timer (*PIT*) needs to generate an interrupt only after a certain interval has elapsed. FAUmachine's scheduler, which is described in more detail in Section 3.3, provides a function to achieve this: With `time_call_after` a callback function can be registered, that will be called after a given interval in simulation time. This way, FAUmachine mimics a discrete event simulation: The interrupt signal of the programmable interval timer can be updated from a function that is delayed using `time_call_after`.

FAUmachine also offers logical processes as a measure to implement state machines in a straightforward manner. A component can define a process by registering the process function to the scheduler. A process can be suspended and resumed by function calls to the scheduler. Logical processes are implemented using the concept of co-routines inside the scheduler.

3.2 CPU-Emulator

The main component of FAUmachine is the CPU-Emulator. Using discrete event simulation for it would result in a very poor performance. Hence the CPU-simulator uses a different simulation paradigm.

The CPU-simulator must execute the instructions of the workload. We'll call the workload "guest system" in the remainder of the paper. FAUmachine uses the technique of a Just-In-Time-Compiler to achieve high-speed emulation (Höxer, Waitz, and Sieh 2004). For each instruction of the guest system, a number of instructions are generated that can be executed in the host environment. The generated instructions simulate the behavior of the instruction of the guest system. This is done for a number of subsequent instructions of the guest system. The result is stored in a buffer. That way, buffer contents can be reused in case that the same instructions are executed in the guest system again, for example if the guest system's code contains a loop. Simulating a stream of instructions from the guest system now means to call the corresponding generated code.

In the case that a component is accessed, for example if the guest system contains an IO-instruction, the corresponding call to the bus access function is inserted into the generated code. After this call, a return instruction is added, so that the remainder of the CPU-simulator can then simulate results or can give back control to the scheduler in the case of need.

Executing a stream of instructions means that simulation time doesn't advance for the entire stream. If the stream contains a loop, this could mean that possibly the simulated CPU wouldn't see any change in simulation time for a very long period in wallclock time. To alleviate this, a check is inserted before each branch instruction occuring in the guest system. That check ends the loop in case an instruction limit has been reached.

The CPU-simulator is implemented as a logical process. Each logical process contains an instruction count register. The scheduler can set an initial limit of executed instructions with it. It is the duty of the logical process in question to decrement the instruction count register in a feasible way. The CPU-simulator uses this approach: If a stream of instructions was executed, the instruction count register is decremented by the number of simulated instructions. If the register turns negative, the CPU-simulator gives back control to the scheduler. That way all components can see a steady increase in simulation time.

3.3 Scheduler

The function of the scheduler is to schedule logical processes, to deliver timed events in the correct order and to advance simulation time. It can be configured to operate in two different modes: In "Interactive mode" FAUmachine provides a virtual environment. The simulation time is synchronized with wallclock time at discrete intervals. "Deterministic mode" provides the possibility to perform repeatable runs. Time synchronization with wallclock time is disabled in deterministic mode. For runs to be repeatable, other preconditions have to apply as well: There mustn't be interaction between the simulated system and the real world. This means for example that FAUmachine may not contact the real network, and may not accept input interactively from the user. Hence FAUmachine allows to simulate user behavior, as this becomes a necessity to perform experiment runs in the first place.

The algorithm of the scheduler for deterministic mode is straightforward: First, all scheduled callback events are called, if the time of the event is equal to or less than the simulation time. Next, the instruction limit for all logical processes is calculated: It is set to the minimum simulation time of the next event (if any) and a fixed upper bound *timeout*. Finally each logical process that is in the runnable state is executed and the simulation time is updated.

Interactive mode requires an additional synchronization of the simulation time to wallclock time. To achieve this, a time Δt between physical time and wallclock time is calculated. In case the physical time has advanced beyond wallclock time, the simulation is stalled for the calculated interval Δt. If $\Delta t$is positive, and hence the simulation is too slow compared to real time, the instruction limit of each logical process is adjusted: The factor $time\_shift$ is calculated at each iteration by dividing the time Δt through the interval of physical time that one iteration should consume. This interval is either the upper bound *timeout* or the simulation time of the next event. The instruction limit of each process is then adapted by right-shifting it by the factor $time\_shift$. Figure 1 illustrates the resulting behavior: The physical time for each iteration remains constant (unless an event should be scheduled). Adjusting the instruction limit of each logical process provides that the time consumed to run the process will approximately divide by the factor 2^{time_shift}. However in practice, simulating an interval of virtual time with the CPU emulator may need quite different amounts of real time. For example if the CPU is powered off or in a halt-state, the simulation is trivial and can be finished almost instantly. In contrast, if the CPU is performing calculations, the amount of real time needed to complete the simulation is much bigger. The aforementioned algorithm also synchronizes simulation time with wallclock time in this scenario. Since $time\_shift$ is readjusted in regular intervals, physical time seems to progress at the same rate as wallclock time. The only drawback is that logical processes like the CPU-simulator will perceive a variation in speed.

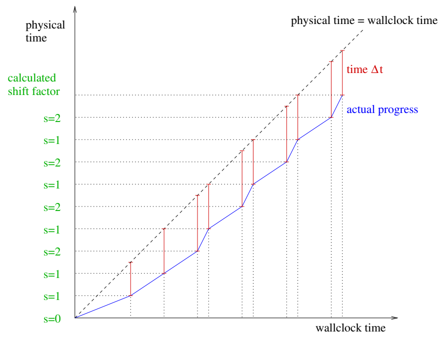

Figure 1: Perceived time progress with FAUmachine's real-time synchronization.

4 MODELING WITH FAUMACHINE

4.1 Hierarchical Structure

FAUmachine ships a number of simulators that model of the shelf hardware components typically found in a PC system. As already mentioned, the implementation of each component is strictly separated from each other. No simulator of a component has knowledge about the internals of any different component. Signals and buses are used to interconnect the different simulators.

Some components consist of sub-components. For example the motherboard is formed by many chips, some sockets and a large number of signals connecting these parts. As some of these chips might be used to build different motherboards, it makes sense to model them separately. Hence FAUmachine uses a hierarchical paradigm to model its simulators.

Generally each of FAUmachine's components consists of many chips, sockets and signals. A south-bridge nowadays contains a great amount of functionality distributed over many chips in former designs. In FAUmachine's chipset the south-bridge (Intel 82371AB) has an IDE controller with two IDE channels, one USB controller, two interrupt controllers, two DMA controllers, a real-time clock, and a programmable interval timer bundled in one chip. In the future even more functionality might get added to the south-bridge chip.

Each of these parts of the chip is independent from the rest of the chip. So in FAUmachine all these parts are modeled separately. This might be done by using sub-components and connecting them with signals. The problem is – speed. This approach would introduce far too many levels of signals.

In FAUmachine, each of these parts is called an "architecture", a construct similar to architectures in VHDL. The implementation is done using the C programming language. A file that implements an architecture is included in a component using the C preprocessor. If an architecture wants to change an output signal, it calls a function in the form `COMP_signalname_set(...)`. The C preprocessor replaces `COMP` with the name of the component. Inside a component a function for every included architecture must be implemented. Usually these functions either change signal values, or are used to directly forward a value to a different included architecture. Since a component represents one compilation unit together with all architectures that it uses, the compiler can inline these functions to improve performance.

Since modeling components, that contain a large number of sub-components in the C programming language is both time-consuming and prone to errors, the schematic editor of the *gEDA* suite (Hvezda et al. 2009) can be used to graphically combine components. In order to achieve this, every component exports its interface in a textual file. From that a symbol file that can be used with *gEDA*, the C header and a VHDL entity declaration are generated. This assures that a mismatch in interfaces

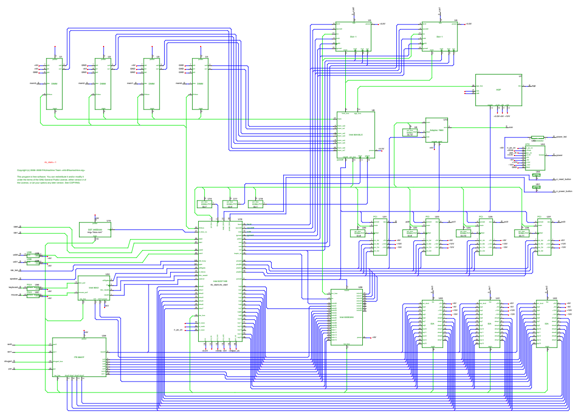

Figure 2: Layout of mainboard *GA_686dlx*.

between the C implementation and the textual interface declaration is detected at compile time. The motherboard is an example that is modeled graphically using a schematic. Figure 2 shows the layout of the mainboard *GA_686dlx* as modeled using the schematic editor. Components modeled using schematic files are converted to VHDL code.

4.2 Structural VHDL Descriptions

The structure of comprehensive computing systems is modeled using VHDL. FAUmachine ships with a number of predefined systems, but it is also possible to create a different one.

In order to obtain repeatable tests, user behavior must also be described. This can be done in VHDL as well. A few components make this possible: The pattern matcher can be used to observe images and text on the simulated monitor. A VHDL library implements the ability to control the mouse using this pattern matcher. That enables the simulated user to click on buttons or to wait until a text appears on the screen. Based on the observations, the user can react by typing text, by moving the mouse or by operating any other control that is part of the interface of the system.

Another precondition to make test runs repeatable is that the initial state must be identical for different runs. For example, in order to test the functionality of the software raid implementation in Linux (Potyra, Sieh, and Dal Cin 2007), a Linux system must be installed on the virtual hard drive and the software raid must be configured for the simulated system. Storing the initial state as the state of a fully configured system would create the need to store the entire state of the simulated hard drive. This is quite impractical though, since this would imply to save several gigabytes of data. Therefore we chose a different approach: The initial state for experiments uses empty hard drives. The first steps of an experiment are to perform the necessary setup of the workload under test by installing an operating system from install media and to configure it. That makes it even possible that other parties can reproduce experiment runs by only obtaining the files containing the structural setup and the user behavior, granted that the installation medium used is commonly available.

We have found, that experiments can often be performed in a sequential manner. For example installing an operating system usually consists of the repeated steps to wait for a text or screenshot to appear, and then to type a command or to press a button. Using VHDL to describe purely sequential tests brings in some overhead. Hence there exists a simplified language to describe user behavior within FAUmachine. A tool translates this simplified language to VHDL.

5 INTEGRATION OF VHDL WITHIN FAUMACHINE

As described in the previous section, high level components can be modeled in VHDL. Likewise it is possible to use VHDL to specify behavior. FAUmachine's components, signals and buses are implemented in the C programming language. This section will describe how VHDL structures can interact with the C implementation of FAUmachine. VHDL sources get translated by FAUmachine's VHDL compiler into an intermediate code. An interpreter written in C executes the intermediate code. The kernel of the interpreter uses a discrete event simulation approach. Instead of keeping track of an independent simulation time, it directly uses the simulation time that FAUmachine's scheduler provides.

5.1 Classification of Components

For each C component, there exists a VHDL description of the interface in the form of an entity declaration. The *foreign* attribute is used to denote that the entity refers to a C component. Additionally the name of the C component is stored in this attribute. With that knowledge, the VHDL compiler can decide, if an instantiated component refers to a component implemented in C, or if it is a VHDL component. The VHDL compiler annotates the intermediate code with that knowledge, so that the interpreter can instantiate a C component in the case of need. Instantiation is done by calling a glue-layer function. The signals of the `PORT MAP` statement form the arguments of this function. The name of the entity is passed as a string so that the glue function can decide which component to create. Since the interfaces of all component simulators is present in a text file (see Section 4.1) the function that maps entity names to the corresponding C function calls for instantiation is also generated automatically. This ensures that the mapping is consistent with the defined components.

5.2 Classification of Signals

Like components, signals present in VHDL descriptions also need to be categorized. This is done based on their usage. The usage is determined by accesses of the signal, that is reading the value or updating the value of a driver. Passing a signal to a component also denotes usage. If a signal is used solely from within VHDL, it is handled as a "native" signal. A native signal will be handled by the VHDL interpreter alone. Likewise, if a signal is used solely by foreign components, and is not accessed from VHDL context, then it is treated as a signal that is created from the C signal representation within FAUmachine. In that case, the interpreter doesn't allocate a signal on its own, but rather calls the corresponding function that creates a signal within the C context. The interpreter stores only the retrieved pointer to the signal instance in a variable. Component instantiation of a foreign entity can thus pass the pointer to the signal as parameter.

There exists yet a third category for signals: Signals that are accessed both from C components and from within VHDL context. These need to be handled specially. As a precondition, there must be a common type between the C implementation and the VHDL value. Internally the VHDL compiler stores all types either as an integral value, a floating point value or a pointer. Additionally the name of the base type is present as well. This name is passed to a glue function if a C signal should get created.

For mixed signals, a VHDL signal is created, and a C signal as well. Additionally one driver is created. The callback function that gets called if the C signal changes is connected to that driver. A change will therefore update the driving value. If the signal is written to from VHDL, additional care must be taken: For each VHDL driver, a special driver is created that is connected to the C signal, but not to the VHDL signal. This means, that resolution will *only* take place using the foreign signal. If both, the VHDL signal and the C signal performed resolution, and propagated the resolved value, then the signal could lock on a dominant value. Such a situation is shown in Figure 3. Figure 4 illustrates how signals shared between VHDL and C are implemented. However there is still another problem that must be taken care of: The VHDL language reference manual (1076-2002 2002) defines the exact steps of the simulation cycle. It is defined in such a way, that first all signals are updated from the corresponding values of the drivers, and next all processes are run, which could result in the change of driving values due to the change of signal values. This means explicitly that a change in a driving value will not directly lead to a change of another driving value, and that a change in a signal value will not directly lead to a change of another signal value. The propagation of a change of a signal value to the change of a different signal value happens at a discrete step. If the simulation cycle were implemented in a way, so that changes in C drivers result directly in callbacks to C components while also updating signals from the driving values, an update of a signal value could directly result in the update of a driving value, and hence the simulation would be incorrect: If the callback of a C signal is called to denote that the driving value changes, this implicitly denotes that simulation of the functionality inside a C component takes place. Calling this callback can hence cause – depending on the component in question – that the driver storing the driving value of a C signal is updated right away. To alleviate this, the simulation cycle in the VHDL interpreter was modified: First, all driver values are propagated to the corresponding signals. Next all drivers connected to C signals call the corresponding callback functions. Then all VHDL processes are run. Finally, it is checked if another δ-cycle should occur.

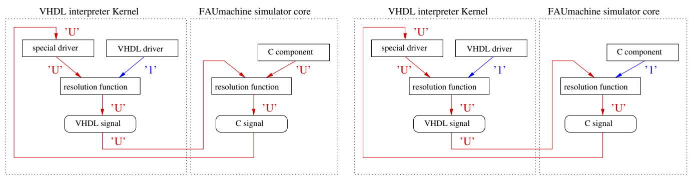

Figure 3: Possible deadlock when resolving a signal both in VHDL and C. Left side: Initial state resolves to *undefined*. Right Side: A change of the driver of the C component doesn't result in a state change due to the dominant *undefined* value.

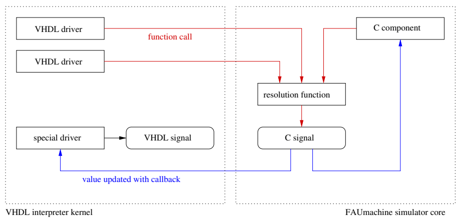

Figure 4: Approach to forward changes between signals shared in VHDL and C.

5.3 Converting Abstracted Bus Cycles

As discussed in the previous section, FAUmachine provides seamless integration of VHDL descriptions with its C components. However this only works for signals of simple types. To speed up the simulation, FAUmachine uses abstraction to model entire bus transactions (see Section 3.1). Buses that are implemented using this method cannot be accessed from within VHDL context. One possible way to overcome this shortcoming is to implement a bus in the way that allows both to connect to an abstracted interface and to a signal level interface. Depending on the connected receivers of bus events, either only abstracted accesses can be simulated, or a simulation of signal level changes can take place. We have implemented this version for the I^2C-Bus protocol: As soon as one component connects to an I^2C-Bus signal using the signal level interface, the simulation switches from an abstracted bus interface to boolean changes on individual signals. The implementation also interprets changes that reflect abstracted bus transaction. That way components that only implement the abstracted model will receive the expected callbacks. This method is feasible for the I^2C-Bus implementation, since this bus consists of only one data signal and one clock signal. Additionally the I^2C-Bus within FAUmachine is used to connect components, that typically transfer only a very small amount of data. One example of the use is to connect the monitor to the graphics card to transfer information about the display properties (so-called “EDID”). Actually the amount of transferred data is so sparse that the implementation was later changed to always simulate the bus in raw, since there was no measurable difference regarding simulation speed.

The aforementioned method provides only poor performance in the case that large amounts of data are transferred over a bus and many components are connected to it. To come around the necessity to simulate each bus transaction in detail, we propose a different approach: The logic to map bus cycles to individual changes of signals can be done in a component. We have implemented this approach for the PCI-Bus. As explained in Section 3.1, a callback reflecting a bus transaction is called for each component that is connected to the bus. Without additional logic, this doesn't directly result in an improved simulation performance. To speed up the simulation, the PCI-Bus signal iterates over connected components in order. As soon as the first component announces that it handles the bus request – which is implemented by the return value of the callback function – other connected components are no longer consulted. The information, which component responded to a request to a given address, is cached within the PCI bus signal. That way further requests to that address can be directly forwarded to the corresponding receiver. If the component that converts abstracted bus transactions to raw bus line events connects to the PCI bus signal, the aforementioned behavior results in a huge improvement in performance: As soon as all connected components have been cached in regards to the addresses they respond to, only transactions that are not handled by any other component need to get converted to raw bus cycles. Not using this approach results in a simulation speed that doesn't allow for interactive use.

5.4 Heuristics to Achieve High Simulation Performance

A big obstacle when simulating VHDL hardware that connects to the PCI-Bus is the clock signal of the bus. The PCI-Bus defines a frequency of 33 Mhz for its clock signal, later versions support even higher frequencies. Constantly simulating this changing clock signal significantly decreases simulation performance. The heuristic we chose to implement is to simulate changes in the clock signal only during accesses to the PCI-Bus. Hardware that is built in a way that it directly interacts to PCI-Bus events can be simulated correctly. However if the clock signal is used as a source that directly affects the internal state of the hardware in question, for example a counter, a simulation mismatch can happen. We will be investigating how to improve the currently used approach in the future.

Another heuristic used within FAUmachine concerns the simulation time. During one transaction on the PCI bus, simulation time should advance between each change of the clock cycle, and even between the data or an address has stabilized on the bus after an edge of the clock. This is highly impractical for FAUmachine's simulation framework: As advancing the simulation time implies to run any process in the runnable state (cf. Section 3.3), this would decrease the simulation speed. As a workaround, the simulation time is simply not changed when mapping a PCI bus transaction to signal level changes. This doesn't have an impact on the simulation in practice though: If the VHDL description can be synthesized, which is a precondition to create hardware from it, no delays are allowed in it in the first place, since delays cannot be transformed to gate level netlists by synthesizing tools.

5.5 Toolchain

Figure 5 gives an overview over the toolchain to run automated experiments including the simulation of hardware descriptions in VHDL: The sequential user interaction in a simplified language is first converted to VHDL by FAUmachine's tool `faum-gen-vhdl`. This VHDL file is combined with the structural description of the simulated system and the functional VHDL file that defines the behavior of a hardware component. The result is used as an input for the VHDL compiler of the FAUmachine project, which creates an intermediate code from it. The interpreter within FAUmachine can execute this code, thereby instantiating and connecting components of the simulated system. Furthermore the interpreter evaluates the hardware description.

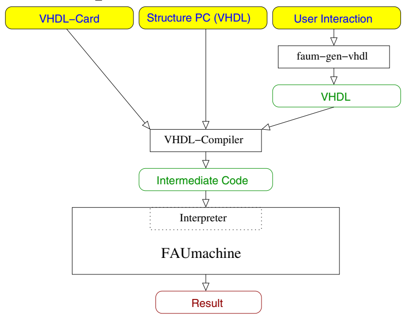

Figure 5: Toolchain used to run automated experiments with FAUmachine.

6 PCI SOUND CARD IN VHDL

To proof that the methods shown in this paper are applicable, we have developed a sound card that is connected to a PCI-Slot. This section will provide an overview about this example application. First we will describe how the developed VHDL card works on real hardware. Next we will show how the model was simulated within the FAUmachine framework.

6.1 Implementation of the Sound Card

The sound card consists of three parts: The PCI control interface connects to the PCI bus. It decodes accesses to the card, implements the necessary PCI configuration space and contains two 8-Bit general purpose Input/Output registers. Both registers are outputs of the PCI control interface. The next part of the sound card is the tone generator. It reads the output pins of the PCI control interface and generates tones in different frequencies, or no tone at all. The square waveform generated by the tone generator is supplied on one output port. More output ports can be used to adjust the volume and to denote the pitch. Finally the digital-analogue converter is implemented as an analogue circuit. It amplifies and smoothes the generated waveform. Both the PCI control interface and the tone generator are modeled using VHDL. To physically test the hardware, we used a prototype card containing the complex programmable logic device (*CPLD*) *iM4A3-512/160* from Lattice Semiconductors. It was programmed with the synthesized VHDL. The analogue circuit was implemented on the prototype board using RC members. Its tone output can be connected to a common loudspeaker.

To test the developed hardware we also implemented a program that performs a register test of the PCI interface at first, and then plays a tune.

6.2 Simulating the Sound Card within FAUmachine

Figure 6 depicts how the developed sound card is simulated within FAUmachine: The CPU issues bus requests on the host bus. The northbridge forwards them to the PCI bus. All of these requests use an abstracted model that reflects bus transactions. As described in Section 5.3, these transactions need to be converted to cycle accurate signal level changes. This is accomplished by the PCI converter. The PCI control interface connects to the PCI converter. The exactly identical VHDL model that was used for the implementation within the CPLD is reused here. To show that FAUmachine allows for seamless integration in both the direction C components to VHDL components and VHDL components to C components, the tone generator was reimplemented in C. The interface is identical to the interface of the VHDL implementation. The digital-analogue converter was also modeled as a C component. To perform automated tests with the simulated VHDL card, a script was created. It boots from a Live-CD, compiles the same test program that was used to test the real hardware implementation and executes it.

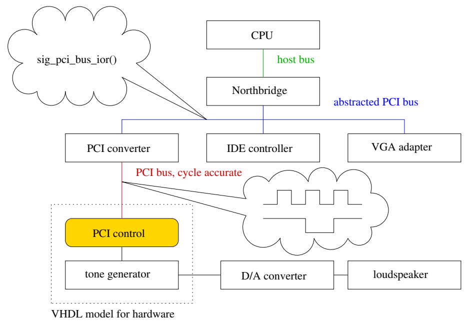

Figure 6: Sound Card Simulation within FAUmachine.

6.3 Results

Running the test program mentioned in Section 6.1 on the real hardware implementation plays a tune. Performing the automated experiment using FAUmachine described in the last section produces a tune that sounds identical. The only difference consists in an initial delay before the simulated sound is played. However the simulation speed is not the reason for this delay, but rather buffering done in the sound system. Hence we believe that the simulation is correct to that degree that it can aid development of real hardware, as it allows for testing in an complete environment.

We have also measured the overhead caused by the co-simulation of VHDL within FAUmachine. To achieve this, we have reimplemented the PCI control logic as a C component. We have then run the experiment four times, varying in using the VHDL or the C implementation, and varying in using interactive or deterministic simulation mode. Table 1 shows the run times of each variant and the calculated resulting overhead caused by the VHDL simulation. With the overhead ranging from 0.4

% to 2.4 %, we have demonstrated that our proposed method is suitable for VHDL co-simulation within a virtual environment while still providing a very high simulation speed.

Table 1: Overhead of VHDL Simulation.

Simulation Mode	**C**	**VHDL**	**Overhead**
interactive	126.9 s	127.4 s	0.4 %
deterministic	92.0 s	94.3 s	2.4 %

7 SUMMARY AND FUTURE WORK

FAUmachine provides a virtual environment. The structural setup is specified using VHDL, whereas the components of FAUmachine are implemented in C. We have shown that it is also possible to extent FAUmachine using components written in VHDL in a straightforward manner. Due to the heuristics used within the simulation framework, the resulting simulation speed is very high. As an example, we have implemented a PCI sound card on a prototype card. Performing an automated test run using FAUmachine both with and without VHDL co-simulation consumes almost the same amount of time. Simulating said example using FAUmachine also yielded no behavioral difference compared to the prototype implementation. Hence we believe that the simulation is correct to that degree that it can aid development of real hardware, as it allows for testing in an complete environment. It can also decrease the time to market if software needs to get developed together with the hardware, as software development can start without the need for a prototype implementation.

Simulating a clock signal with a high frequency, for example the PCI bus clock, in a timing exact manner significantly decreases simulation performance. To alleviate this, we used an heuristic which only simulates the clock signal when the bus is accessed. This has the drawback that hardware may not change its state based on changes of the clock signal alone. As an example, a continuously running counter cannot get simulated correctly. One item of future work is to find a method that allows for better simulation of clock signals while maintaining an adequate simulation speed.

Parallel and distributed simulation methods might enhance the simulation performance even further. The current approach uses sequential simulation only to avoid synchronization of several logical processes running in parallel. In particular simulating large clusters of PCs would greatly benefit from a distributed simulation. Future work is to evaluate methods present regarding parallel and distributed simulation and to adapt these to FAUmachine's framework where applicable.

REFERENCES

1076-2002, I. S. 2002. IEEE Standard VHDL language reference manual. Product No.: SH94983-TBR.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. Xen and the art of virtualization. In *SOSP*, 164–177.

Bauer, D., G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman. 2003. Simulation of large scale networks III: ROSS.Net: optimistic parallel simulation framework for large-scale internet models. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, 703–711: Winter Simulation Conference.

Bellard, F. 2005. QEMU, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, 41–46. Berkeley, CA, USA: USENIX Association.

de Farias, C. R. G., L. F. Pires, W. L. de Souza, and C. E. Morón. 2001. Specification and validation of a real-time parallel kernel using LOTOS. In *MASCOTS*, 7–14: IEEE Computer Society.

FAUmachine Team 2003–2010. FAUmachine. URL: http://www.FAUmachine.org/.

Hatnik, U., and S. Altmann. 2004. Using ModelSim, Matlab/Simulink and NS for simulation of distributed systems. In *PARELEC '04: Proceedings of the international conference on Parallel Computing in Electrical Engineering*, 114–119. Washington, DC, USA: IEEE Computer Society.

Höxer, H., M. Waitz, and V. Sieh. 2004. Advanced virtualization techniques for FAUmachine. In *11th International Linux System Technology Conference, Erlangen, Germany, September 7-10, 2004*, ed. R. Spenneberg, 1–12.

Hvezda, A. et al. 2009. geda project. URL: http://www.gpleda.org/.

Iyer, R. R. 2003. On Modeling and Analyzing Cache Hierarchies using CASPER. In *MASCOTS*, 182–187: IEEE Computer Society.

Kamal, A. E. 2004. Discrete-time modeling of TCP Reno under background traffic interference with extension to RED-based routers. *Perform. Eval.* 58 (2-3): 109–142.

Kivity, A., Y. Kamay, D. Laor, U. Lublin, and A. Liguori. 2007, July. kvm: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, 225–230.

Mentor Graphics Corp. 2007. ModelSim – a comprehensive simulation and debug environment for ASIC and FPGA design. URL: http://www.model.com.

Over, A., P. Strazdins, and B. Clarke. 2005. Cycle accurate memory modelling: A case-study in validation. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 85–96. Washington, DC, USA: IEEE Computer Society.

Potyra, S., V. Sieh, and M. Dal Cin. 2007. Evaluating fault-tolerant system designs using FAUmachine. In *EFTS '07: Proceedings of the 2007 workshop on Engineering fault tolerant systems*, 9. New York, NY, USA: ACM.

Radhakrishnan, R., D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey. 1998. An object-oriented time warp simulation kernel. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, 13–23. London, UK: Springer-Verlag.

Sand, M., S. Potyra, and V. Sieh. 2009. Deterministic high-speed simulation of complex systems including fault-injection. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*: IEEE.

Source Forge 2001. Bochs IA-32 emulator project. URL: http://bochs.sourceforge.org/.

Szymanski, B. K., Y. Liu, and R. Gupta. 2003. Parallel network simulation under distributed genesis. In *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation*, 61–68. Washington, DC, USA: IEEE Computer Society.

User Mode Linux Core Team 2004. User Mode Linux HOWTO. URL: http://user-mode-linux.sourceforge.net/UserModeLinux-HOWTO.html.

VMware Inc. 2001. VMware. URL: http://www.vmware.com/.

Wilsey, P. A., D. E. Martin, and K. Subramani. 1998. 8.4: Savant/tyvis/warped: Components for the analysis and simulation of vhdl. In *IVC-VIUF '98: Proceedings of the International Verilog HDL Conference and VHDL International Users Forum*, 195–201. Washington, DC, USA: IEEE Computer Society.

AUTHOR BIOGRAPHIES

STEFAN POTYRA is a member of the research staff at the chair of Computer Science 3 (Computer Architecture) of the Friedrich-Alexander-University Erlangen-Nuremberg. He obtained the diploma in computer science at the same university in 2007. His research interests include virtualisation and HDL simulation. His email address is <Stefan.Potyra@informatik.uni-erlangen.de>.

MATTHIAS SAND is a researcher at the chair of Computer Science 3 at the University of Erlangen-Nuremberg. He holds a doctor's degree in computer science. His research interests include model and simulation based evaluation and verification of the dependability properties of complex systems. He has been working for the last few years in the field of the assessment of complex computing equipment by using efficient virtualisation techniques combined with fault-injection. His email address is <Matthias.Sand@informatik.uni-erlangen.de>.

VOLKMAR SIEH is a member of the research staff at the chair of Computer Science 3 (Computer Architecture) of the Friedrich-Alexander-University Erlangen-Nuremberg. In 1998 he obtained a doctor's degree in computer science. Since several years Dr. Sieh is the project leader of the FAUmachine project (virtual machine with fault injection capabilities). His research topics are design and validation of models of hardware architectures for dependable high performance computing. His email address is <Volkmar.Sieh@informatik.uni-erlangen.de>.

DIETMAR FEY studied Computer Science and made his Ph.D. in the field of Optical Computing, both at the University Erlangen-Nuremberg. After his Ph.D. he was engaged as lecturer at the University of Siegen and Jena, Germany, for more than ten years. In Jena he held a professorship on Computer Engineering. Since 2008 he holds the Chair of Computer Architecture at the University of Erlangen-Nuremberg, Germany. His research interests are in the areas of parallel embedded processor architectures, heterogeneous parallel architectures, Cluster and Grid computing, and nanocomputing. His email address is <Dietmar.Fey@informatik.uni-erlangen.de>.