Full-System Simulation

(Extended Abstract)
Jakob Engblom

Abstract—**Full-System Simulation is a technology where the hardware of a computer system is simulated at such a level of detail that the complete real software stack can be executed. It has wide applicability in the development and research of computer systems, especially embedded systems. This talk gives an overview of the full-system simulation and some examples of its industrial and academic applications.**

I. INTRODUCTION

SIMULATION is a method of scientific and engineering inquiry that is based on the idea of building a *model* of a system, and then performing *experiments* on this model. Provided that the model has a good fidelity to the system being modeled, the results from the simulation experiments can be used to predict the behavior of the real system.

Simulation is used in all fields of science and engineering. For example, modern cars are virtually crash tested in computer simulation in order to build safer cars. Weather forecasts are prepared by simulating the evolution of current weather patterns into the future.

Computers can also be used to simulate computer systems. This introspective application of computers is incredibly useful, across all fields of computer hardware and software development, from initial architecture, through software development, to end-of-life maintenance.

A key concept is *full-system simulation*, meaning models that encompass *all* of a computer system. Including the processor core, its peripheral devices, memories, and network connections. With such technology, it is possible to simulate a whole computer system with its complete software stack, which opens up new possibilities in the field of computer-system simulation [1][2][3][4].

Full-system simulation has not been feasible until recently, thanks to two long-term technology trends. One is the fact that cheap PCs have become as fast as any other computer system; it used to be that in order to simulate a large server you needed a large server, but this is no longer the case [2]. The other trend is that simulation technology has improved the efficiency of simulation by orders of magnitude. The net result is that the simulation cost per hour of target time has come down by four orders of magnitude over the past 25 years [1]!

II. FULL-SYSTEM SIMULATION

The idea behind full-system simulation is very simple: *model the behavior of the hardware* of a system at a sufficient level of detail that all the *real software* can run (Figure 1); at a level of abstraction appropriate to simulating complete systems containing multiple processors and machines, potentially connected across simulated networks.

A. Scope and Abstraction

The size of the system (i.e. the scope) that can be simulated depends on the level of abstraction chosen in the model. The more detailed the model is, the smaller the simulated system has to be. For full-system simulation, the most appropriate abstraction is the instruction set and control register level. This level has the advantage of being the best documented and most stable layer in a system.

Working at this level makes it possible to simulate networks of *hundreds of simulated machines* using a few tens of host machines. Using a more detailed model, like the RTL level simulations common in hardware design, would decrease the simulation speed to an unusable level. Going to higher levels of abstraction, like operating system APIs, would lose too much information about the system to make it useful. Also, maintaining an API-level simulation is very expensive since it is a much broader and faster-changing interface compared to the instruction set level.

B. Processor

In most cases, the most complex part of a computer system is the main processor core(s). Instruction processing also uses most of the execution time of the simulation, and thus, creating a fast model of the instruction set is key to good simulation performance. The current state of the art allows
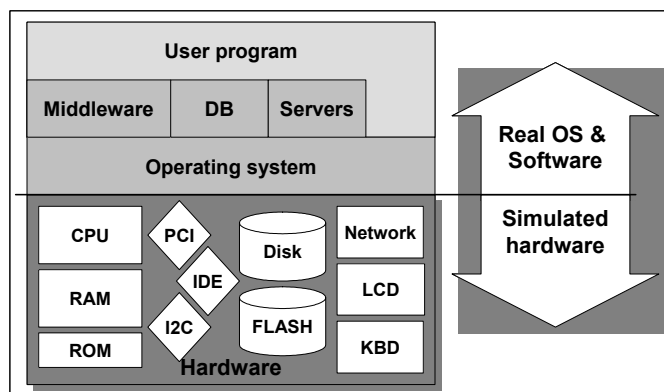
Fig. 1. Full-System Simulation is based on simulating the hardware while running all the real software of a system, with sufficient speed to execute real workloads.
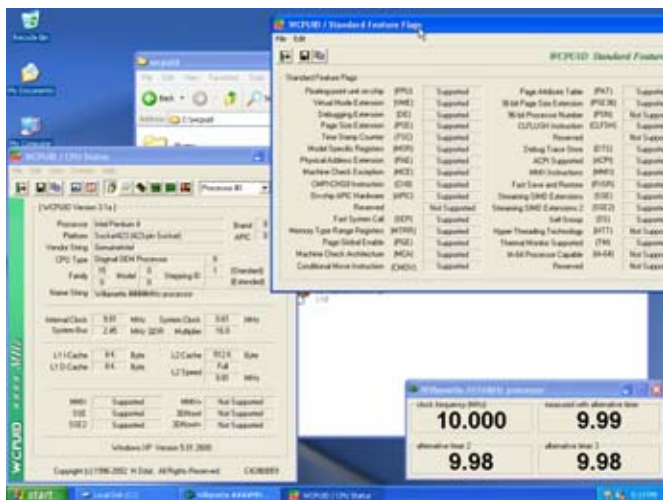
Fig. 2. This screenshot shows Virtutech Simics presenting all the model-specific registers of a Pentium 4 to the diagnostic program wcpuid. The clock frequency meter is THGClock, which measures the speed of the CPU clock using timers. This also gives a consistent result.

instruction set simulation to reach speeds in the hundreds of million of instructions per second, which is sufficient to execute large real-life workloads.

In order to run all the software of a system, the processor model must implement both the user-level and supervisor-level interfaces of the processors, as well as the memory-management unit and various low-level machine registers. Anything that can be seen from the software has to be modeled. Figure 2 gives an indication of the level of detail required when modeling processors of the x86 line.

Another important consideration is that the results of all instructions have to be bit-exact. A difference in computation results compared to a real machine is not acceptable.

For most systems, it is reasonable to use a fairly simplistic model of instruction timing, typically one cycle per instruction. If necessary, this model can be extended with more precise timing. For example, just adding the timing of the cache system provides a fairly useful timing model for most embedded systems, and computer architecture researchers will often create complete microarchitectural models of a processor. However, the more detail that is added, the slower the simulation will run.

C. Devices

The defining difference between full-system simulation and traditional instruction-set simulation is the modeling of devices. Device models are necessary in order to get meaningful software to run on the simulator. Timers, network interfaces, PCI bridges, SCSI interfaces, graphics devices, serial ports: there is a large number of different devices which are present in a computer system that must be modeled.

Device models can be either transaction-oriented or bit level. A *transaction-oriented* model handles each interaction (typically, a write to or read from the interface registers of the devices by a processor) to a device as a unit: the device is presented with a request, computes the reply, and returns it in a single function call. This is a very efficient model.

A *bit-level model* instead models the actual bit patterns traveling over buses and pins in the computer. Instead of a single transaction, each cycle on the bus is played out. The device model will read the address and data lines, wait for some cycles, and then put its reply on the bus connecting it to the CPU. This style of modeling results in much lower performance, since the simulation needs to switch context more often, and more computation work is required for the same result.

Of course, it is possible to create hybrids between the two extremes. One quite common approach is to let most of a system work at the transaction level, modeling only a small part of the system at the bit level. This offers an efficient way to drive a detailed model of a device with real traffic from a full system. At the point where the transaction level and the bit level meet, special conversion code is inserted. The conversion is quite simple, since most of the information needed to create bit-level signals is contained in the transactions. Usually, only detailed timing needs to be added in the form of a clock signal.

One should note that transaction-oriented modeling is enabled by the trend towards more abstract and less timing-dependent device interfaces. As noted in [5], in the 1970s, software and hardware often depended on precise timing to function together. Thanks to the trend that processors, devices, and buses are developed separately, software interfaces to hardware are becoming more abstract, thus enabling efficient device modeling.

D. Software

The key goal of full-system simulation is to let the simulator run all the real software of a system, from firmware and devices drivers, the operating system, to databases, middleware, and application programs. That the complete software stack is used in the simulation enables many exciting applications, as detailed in Section III.

Note that the user-level applications often can be executed on less complete simulators using API-level simulation of the operating system, but as workloads become more dependent on the operating system, such simulation will start to miss important effects [3].

E. Stimuli

Given that a good model of the hardware exists, and that the software stack is available and runs on this model, we need to find a way to provide stimuli to the system. Good stimuli are crucial to obtaining sensible results from the simulation, results that are relevant in the real world.

In some cases, just executing the software is a sufficient source of stimuli. This is the case, for example, when porting operating systems. Just doing a boot of the system provides a good way to check the correct function of the software being investigated. No external input is needed in this process.

For interactive systems, the simulation can rely on mapping the simulated user interface devices (screens, keyboards, mice, touch screens, etc.) to real devices.

Models of networked systems can include full machine models of both sender and receivers, or they can be interfaced to real networks. Abstract load generators can also be employed to generate traffic without using full machines.

When it comes to executing benchmarks like SPEC or TPC in the simulation, we might get performance issues if a detailed microarchitectural model is used. Especially processor design and computer architecture research is problematic, since these tasks by nature require very detailed processor models to be used. In such cases, we can use sampled execution or scaled workloads in order to get acceptable simulation times [2][3].

III. HANDY FEATURES OF SIMULATION

Since simulation is "just software", it offers many advantages compared to a real machine:

1) *Configurability*. Any machine configuration can be used, unconstrained by available physical hardware.
2) *Controllability*. The execution of the simulation can be controlled arbitrarily, disturbed, stopped, and started.
3) *Determinism*. A simulation is completely deterministic [1] (provided it is properly programmed).
4) *Globally synchronous*. Multiple processors, multiple devices, multiple machines in a network: all can be stopped instantly in a simulation, and a global snapshot of the state inspected.
5) *Checkpointing*. The state of the simulation can be written to disk and restored in an instant.
6) *Availability*. Creating a new machine is just a matter of copying the setup. There is no need to procure hardware prototypes or development boards.
7) *Inspectability*. The complete state of the simulated machine can be investigated without disturbing the execution.
8) *Sandboxing*. The simulation environment offers a perfect sandbox, out of which no code or data can escape unless explicitly allowed.

So we see that in many ways, simulation is really better than the real thing!

IV. INDUSTRIAL USE OF FULL-SYSTEM SIMULATION

A. Computer Architecture

Detailed simulation of processors has been a mainstay of computer architecture research and development for the past few decades [3]. Mainstream use is still simulation of single processors with none or few peripheral devices. However, as computer systems become more complex, including multiple processors and sophisticated peripherals, the scope of simulation has to expand apace. Many interesting workloads (like transaction processing and web servers) also feature significant amounts of operating system interaction. Thus, full-system simulation is becoming increasingly important.

B. Computer System Development

Full-system simulation is also used in the development of *computer systems*, not just components. By using full-system simulation, designers of complex computer systems such as servers, flight controllers, or network routers can build *virtual prototypes*. Virtual prototypes are used to model and analyze the system configuration, making it possible to create a more efficient overall system.

Furthermore, the virtual prototypes are used by software teams to get an early start on software development for the new system. As illustrated in Figure 3, the software teams can start working long before hardware is available (even in prototype form). This allows critical tasks like device driver and firmware development, and operating system bring-up to proceed in parallel with the hardware development, which can save many months of time to market for a moderately complex system. The need to program a computer before the system is completed was foreseen already in 1946 by Alan Turing [6], so it is really an old need that can finally be satisfied in a reasonable way thanks to full-system simulation.

A good example of the value of virtual prototyping is AMD's AMD64 architecture. When the processors and systems were finally launched in hardware form in 2003, operating systems like Linux and Windows were already running on the processors. This was thanks to a seeding program using full-system simulation which had been going on since 2001, giving AMD a much broader software support at launch than would otherwise have been possible.

C. Hardware/Software Cosimulation

Full-system simulation is also used in cosimulation, where some part of a system (a device, a processor, or maybe a bus controller) is simulated in detail at the RTL level. By simulating as much as possible of the system at a higher level of abstraction, simulation speed is greatly increased compared to simulation of a whole system at the RTL level.

Note that the slowest simulated component of a system will determine the overall simulation speed. However, if the slowest component is clocked at a lower rate than other parts (like a 100 Mhz bus connected to 1000 Mhz processors), the impact can be minimized. Intelligent and minimalist choices of what to simulate in detail is crucial to obtaining good overall simulation performance.

With increased speed, bigger software runs can be made. Thanks to the completeness of the simulated systems, cosimulation can include the effects of operating systems. Overall, this means that more realistic simulations can be performed, which increases the value of the co-simulation.

D. Network Research and Development

By allowing real operating systems and workloads to run on the simulated machines, full-system simulation opens up new vistas of network research. Today, most network simulations focus on the behavior of the network, while modeling the network nodes as simple synthetic traffic generators. Using full-system simulation, it is possible to replace synthetic traffic with real traffic. To model a client-server scenario, both the client and server machines are simulated, and real interactive sessions played out between them. This makes it

possible to investigate the interaction between applications, servers, and the network. And all parts of the equation can be changed arbitrarily. The effect of optimizations to network stacks, device drivers, and actual network interface hardware can be investigated in detail in a simulated environment.

Networks in simulation are also much easier to trace and inspect than real networks. Every packet sent is available for inspection, without affecting the functionality of the system. Disturbances and faults can be injected into a system with ease and precision, since there is no need to actually disturb physical links.

An interesting special case is the testing of large network configurations. In most cases, building a very large test network is prohibitively expensive. Here, full-system simulation allows hundreds of network nodes to be simulated using a handful of standard PCs or server machines. This leads to very large cost savings, as well as an increase in product quality and developer productivity thanks to better testing, easier setup and reconfiguration, and troubleshooting abilities.

E. Software Development

Simulators are being used in software development, for a number of different purposes. As discussed above, virtual prototypes are used to develop low-level software for new systems. Once the hardware is stable and no longer in the prototype stage, the simulation model really becomes *virtual hardware*. Such virtual hardware behaves identically to the real hardware, and is used as hardware replacements.

By making the hardware virtual, replication of the hardware becomes easy and very cheap. This means that more developers can be given direct access to the hardware in question, without the expense of buying or manufacturing extra target systems. Especially for expensive custom hardware systems, this gives great savings.

Virtual hardware comes with all the benefits of simulation: by using the checkpointing, determinism, and inspection abilities, software bugs are easier to repeat and fix.

For low-level software like interrupt handlers and operating systems, simulation offers a superior debugging environment. Hardware accesses can be accurately traced, the state of components, like configuration registers and processor TLBs can be inspected, and interrupt handlers can be single-stepped.
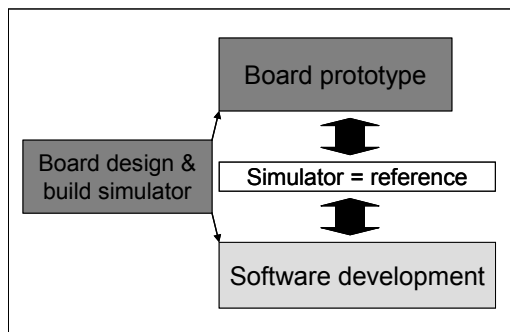

Fig. 3. Parallelization of software and hardware development using full-system simulation. The simulator is designed as part of the system design, and handed over to the software team at the same time as the hardware group starts building the hardware prototypes. This methodology can save many months of time to market for complex computer systems.

All of which is impossible on a real machine.

F. Fault Injection

Simulation affords the user complete control over the simulated system, which makes fault-injection testing very effective and efficient. Thanks to the controllability, faults can be injected at any point in a system: processor register, device registers, memory contents, bus traffic, network packets: everything is available for transient or permanent modification. Thanks to the control over timing, injected faults can be precisely repeated, which allows for regression testing in the presence of faults.

G. Legacy System Support

Long-lived computer systems used in commercial and aerospace applications have a tendency to far outlive the commercial lifespan of their components. This creates problems for the developers maintaining the software base: development boards will break, and replacements will be impossible to buy. To solve this problem, full-system simulation is used to create virtual development boards with a far greater lifespan than the hardware. A virtual model can easily be ported to run on successive generations of development workstations, irrespective of the hardware availability of the actual target systems.

V. SUMMARY

This extended abstract has given a quick overview of full-systems simulation technology, the ideas behind it, and its industrial and academic application areas. Full-system simulation is an old idea whose time has finally come, thanks to the prevalence of cheap and powerful computers, and a maturation of the technology.

REFERENCES

[1] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, Bengt Werner, "Simics: A Full System Simulation Platform", *IEEE Computer,* Feb 2002.

[2] Alaa R. Alameldeen, Milo M.K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Daniel J. Sorin, Mark D. Hill and David A. Wood, "Simulating a $2M Commercial Server on a $2K PC", *IEEE Computer,* Feb 2003.

[3] Kevin Skadron, Margaret Martonosi, David I. August, Mark D. Hill, David J. Lilja, Vijay S. Pai, "Challenges in Computer Architecture Simulation", *IEEE Computer*, August 2003.

[4] H. Shafi, P. J. Bohrer, J. Phelan, C. A. Rusu, and J. L. Peterson, "Design and validation of a performance and power simulator for PowerPC systems", *IBM Journal of Research and Development*, Vol 47, No 5/6, September/November 2003.

[5] Steven E. Gemeny and Michael W. Gemeny, "Ground System Planning for Long Duration Space Missions Helped by Lessons Learned Resurrecting Obsolete Computers", The John Hopkins University Applied Physics Laboratory, 2003.

[6] Andrew Hodges, "Alan Turing: the Enigma", page 326, ISBN 0-09-911641-3, Random House, London, 1992.