A Fast, Inexpensive and Scalable Hardware Acceleration Technique for Functional Simulation

Srihari Cadambi
cadambi@nec-lab.com

Chandra S Mulpuri
chandi@nec-lab.com

Pranav N Ashar
ashar@nec-lab.com

C&C Research Laboratories
NEC, Princeton NJ 08540

ABSTRACT

We introduce a novel approach to accelerating functional simulation. The key attributes of our approach are high-performance, low-cost, scalability and low turn-around-time (TAT). We achieve speedups between 25 and 2000x over zero delay event-driven simulation and between 75 and 1000x over cycle-based simulation on benchmark and industrial circuits while maintaining the cost, scalability and TAT advantages of simulation. Owing to these attributes, we believe that such an approach has potential for very wide deployment as replacement or enhancement for existing simulators. Our technology relies on a VLIW-like virtual simulation processor (SimPLE) mapped to a single FPGA on an off-the-shelf PCI-board. Primarily responsible for the speed are (i) parallelism in the processor architecture (ii) high pin count on the FPGA enabling large instruction bandwidth and (iii) high speed (124 MHz on Xilinx Virtex-II) single-FPGA implementation of the processor with regularity driven efficient place and route. Companion to the processor is the very fast SimPLE compiler which achieves compilation rates of 4 million gates/hour. In order to simulate the netlist, the compiled instructions are streamed through the FPGA, along with the simulation vectors. This architecture plugs in naturally into any existing HDL simulation environment. We have a working prototype based on a commercially available PCI-based FPGA board.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids - *Simulation, Verification*

General Terms

Algorithms, Performance, Experimentation, Verification

Keywords

Functional simulation, hardware acceleration, VLIW, FPGA

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

1. INTRODUCTION

Functional simulation plays a critical role in determining the overall time-to-market of a design. The amount of functional simulation that designers have to perform before they incur the time and expense of manufacture is large. More than 60% of human and computer resources are used for verification in a typical design process, of which more than 85% are for functional simulation[2].

The need for faster functional simulation has been addressed to some extent by better simulators as well as specialized hardware emulators. Levelized compiled code simulators and cycle-based simulators specifically address the needs of functional verification by ignoring timing[9, 12]. They offer an inexpensive method of accelerating functional simulation. Specialized emulators, on the other hand, offer a considerable performance gain over software simulators, but at a much higher cost.

In this work, we present a novel and inexpensive hardware acceleration scheme for functional logic simulation. Functional simulation is a good candidate for hardware acceleration since it has considerable concurrency (or instruction-level parallelism) that cannot be exploited by traditional processors. We use a standard "off-the-shelf" PCI-board with a single FPGA as our accelerator. Instead of emulating the circuit directly on the FPGA, we introduce the notion of an intermediate *simulation processor* called SimPLE (Simulation Processor for Logic Evaluation). SimPLE is mapped onto the FPGA, while the netlist to be simulated is compiled to SimPLE. This makes the technique scalable, i.e., independent of the netlist size. Using the single FPGA board reduces the cost, while the FPGA itself offers flexibility in the SimPLE architecture.

Figure 1 compares our solution against software simulators and emulators. SimPLE, with its inexpensive hardware and software, costs the same as software simulators, but is orders of magnitude faster. In addition, the turn-around-time for simulating a design with SimPLE is similar to software simulators and much faster than emulators. Thus, it is a viable replacement or enhancement to software simulators and a cheaper alternative to emulators.

2. RELATED TECHNOLOGY

FPGAs are frequently used in emulators. A typical emulation board consists of several FPGAs interconnected to-

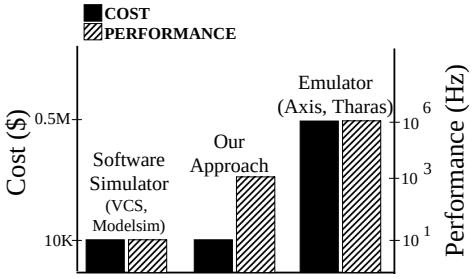

Figure 1: SimPLE vs. simulators and emulators.

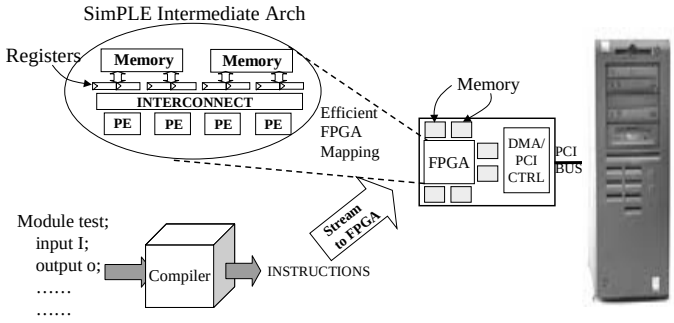

Figure 2: Simulating using SimPLE.

gether. [3, 4] demonstrated the Virtual Wires scheme to time-multiplex wires on physical pins connecting the FPGAs, thereby improving their utilization. However, such large emulation systems are neither cost-effective nor scalable. Several emulation vendors use multiple FPGAs and specially designed hardware within emulation systems costing hundreds of thousands of dollars.

Another approach to emulation is to time-multiplex large designs onto smaller FPGAs. The circuit is not emulated as a whole, but in portions: each portion fits inside the single FPGA, which is repeatedly reconfigured. While this does not have the high cost of the multi-FPGA solution, it's performance is adversely affected by the FPGA's reconfiguration overhead. Generic FPGAs dedicate only a small number of I/O pins for configuration purposes (for example, the SelectMap interface in Xilinx FPGAs [13]). Thus they have a very small configuration bandwidth resulting in significant delays during reconfiguration. Specialized FPGA architectures with extra on-chip storage for multiple configuration contexts have been devised to address this issue[10, 11]. However, such architectures are not commercially available.

3. OUR ACCELERATION SYSTEM

We solve the FPGA configuration bandwidth problem by introducing the notion of a virtual simulation processor called SimPLE. Netlists to be simulated are transformed into instructions for SimPLE by a compiler. These instructions use the data I/O pins of the FPGA and are therefore not affected by the small configuration bandwidth. Once it is configured on the FPGA, different netlists may be simulated on SimPLE using the instructions.

3.1 The Overall System

Our hardware acceleration system consists of a generic PCI-board with a commercial FPGA, memory and PCI and DMA controllers, so that it naturally plugs into any computing system. The board is assumed to have direct access to the host's memory, with its operation being controlled by the host. Thus, the host can direct DMA transfers between the main memory and the memory on the board, which the FPGA can access. Further, with our scheme, the board memory need only be single-ported with either the FPGA or the host (via the PCI interface) accessing it at any time.

Figure 2 shows our simulation methodology. The VLIW-stype instructions compiled for SimPLE are transferred to the on-board memory along with a set of simulation vectors using DMA. Each instruction represents a slice of the netlist. Executing all instructions simulates the entire netlist for one simulation vector. For each vector therefore, all instructions are streamed from the board memory to the FPGA after which the result vector is stored back in the board memory. If the SimPLE instruction is wider than the FPGA-memory bus on the board, it is time-multiplexed into smaller pieces that are reorganized using extra hardware on the FPGA. When all the simulation vectors are done, the result vectors are DMA'ed back from the board to the host. More simulation vectors may now be simulated if required. The host controls the entire simulation using an API.

In order to quantify the simulation speed, we define *user cycles*, *processor cycles*[10] and *FPGA cycles*. The FPGA cycle is the clock period of the FPGA with SimPLE configured on it. A processor cycle is the rate at which SimPLE operates. It is defined as the time taken to complete a single SimPLE instruction. Note that if the instruction is time-multiplexed (i.e., when the SimPLE instruction is wider than the FPGA-memory bus on the board), the processor cycle is larger than the FPGA cycle. Finally, a user cycle is the time taken to fully simulate the netlist for a single simulation vector, i.e., process all the instructions.

We can now quantify the simulation rate. Assume the SimPLE compiler produces N instructions for a netlist when targeting a SimPLE architecture whose instruction width is I_W. If the FPGA-memory bus width on the board is B_W and the FPGA clock cycle is F_C, then the user cycle U_C and simulation rate R are given by

$$U_C = N * \lceil \frac{I_W}{B_W} \rceil * F_C \quad (1)$$

$$R = \frac{1}{U_C} \quad (2)$$

Thus the simulation rate can be increased by reducing (i) the number of instructions produced by the compiler, (ii) the instruction width and (iii) the FPGA clock cycle.

If a very large circuit compiles to too many instructions that do not fit in the on-board memory, the instructions are broken up into smaller portions and DMA'ed separately. This affects the overall performance but maintains the scalability of SimPLE. By upgrading the on-board memory however, we can achieve scalability with no loss of performance. Reasonable amounts of memory allow very large netlists to be simulated: a board with 256MB of SDRAM, for instance, can hold all instructions for a 40-million gate netlist.

3.2 The SimPLE Architecture

SimPLE is based on the VLIW architectural model. Such an architecture can take advantage of the abundant inherent parallelism present in functional logic simulations. A template of SimPLE is shown in Figure 3. It consists of a

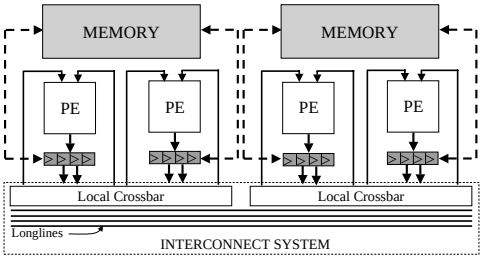

Figure 3: Architectural model of SimPLE.

large array of very simple interconnected functional units or processing elements (PEs)[1]. Each processing element is a 2-input look-up table (LUT) that can evaluate any 2-input Boolean function. The array of PEs can evaluate a large number of gates every cycle. In order to store intermediate signal values, SimPLE has a distributed register file system that provides considerable accessibility at high clock speeds. Each PE has its own register file, with a small number of read ports. Since the number of registers is limited by hardware considerations (FPGAs are not register-rich), there is a second-level of memory hierarchy in the form of a distributed memory system. Registers may be spilled into this memory, and loaded again later. The presence of multiple memory banks permits fast simultaneous accesses. Each memory bank services a set of processing elements and their register files. The PEs perform (i) a Boolean operation during which a result is written to their register files (ii) a memory store during which a register from the register file is spilled into memory and (iii) a memory load during which a value from memory is loaded into a register. Each register file has a single write port that may be used by the PE or the memory, a read port for memory stores and a small number of general read ports connected to other PEs by means of an interconnect system.

The interconnect system in SimPLE consists of local crossbars and long lines. A local crossbar connects a subset of PEs, i.e., it connects read ports of a PE's register file to the inputs of any PE within the subset. The long lines span the entire range of PEs, that is, they enable a register to be read by any PE. If there is a single local crossbar with no long lines, the interconnect system becomes a full crossbar switch. For this paper, we restrict ourselves to a single crossbar switch with a fixed latency.

The SimPLE architecture is parameterizable using the number of PEs, the register files (size and number of ports), the memory size and configuration (i.e., how many PEs each memory bank spans, etc.) and memory and interconnect latencies. All parts of SimPLE, including the memory, are mapped to an FPGA. The VLIW instruction format of SimPLE, along with further architectural details, is shown in Figure 4.

3.2.1 *Advantages of SimPLE*

SimPLE has the following advantages over software simulators and hardware emulators, whether FPGA-based or otherwise. (i) **Parallelism:** Parallelism present in functional simulations is effectively exploited by the wide processing datapath of SimPLE. (ii) **Register and Memory Access:** The architecture offers easy access to a large number of registers, much larger than what is possible in traditional CPUs. The memory banks are also within close proximity, permitting fast memory access times (1-2 FPGA cycles). (iii) **Configurability:** Since SimPLE is parameterizable, several different configurations may be premapped to FPGAs and stored in a library. Depending on the target FPGA or on the netlist at hand, an appropriate architecture may be chosen. (iv) **Scalability:** SimPLE is transparent to the size of the netlist, much like a software solution. A netlist is compiled into a set of instructions, any number of which may then be executed. Larger versions of SimPLE provide better performance, but smaller ones will still simulate the netlist. (v) **Configuration Bandwidth:** We avoid the small configuration bandwidths of FPGAs by using the data I/O pins for instructions.

[1]The PEs currently target gate-level netlists. We are in the process of extending them to RT-level netlists.

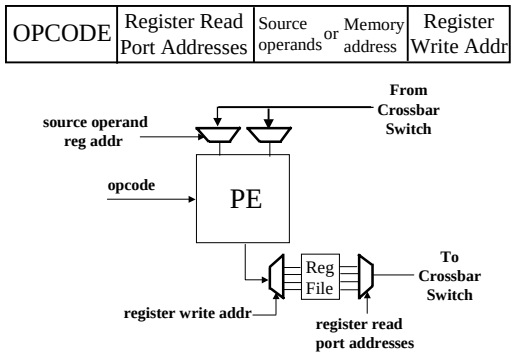

Figure 4: SimPLE instruction format.

4. THE COMPILER

Prior to simulation, a compiler translates netlists into SimPLE instructions. The compiler for SimPLE accepts as input a gate-level netlist described in structural verilog or the Berkeley Logic Interchange Format (BLIF) and the parameters of the target SimPLE architecture. The initial netlist is optimized using standard Boolean optimization techniques and then mapped to lookup-tables using FlowMap[5]. Subsequently, SimPLE instructions are generated. Instruction generation involves PE, register and routing resource allocation as well as memory management. We now focus on the instruction generation procedure.

4.1 The Scheduling Algorithm

The compiler schedules each operation to PEs and manages registers and memory according to the architectural constraints of SimPLE. It maps gates to PEs and wires interconnecting the gates to registers. The registers are allocated such that overall register usage is minimized. The inputs of all PEs are connected to source registers using the register ports and the interconnect system. When a register file is full, a register is selected to be spilled and stored into memory, to be loaded upon demand later. The scheduling algorithm is deterministic and fast, similar to [7].

At every stage of the compilation, the set of graph nodes that have their sources scheduled comprise the *ready-front*. Instruction generation proceeds by selecting nodes from the ready-front and packing them into instructions. Figure 5 shows the instruction generation process. If resources (i.e., PEs and registers) are available, a node is selected to be

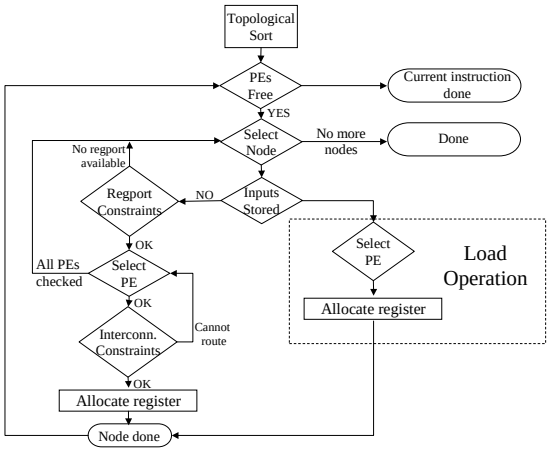

Figure 5: Scheduling and instruction generation.

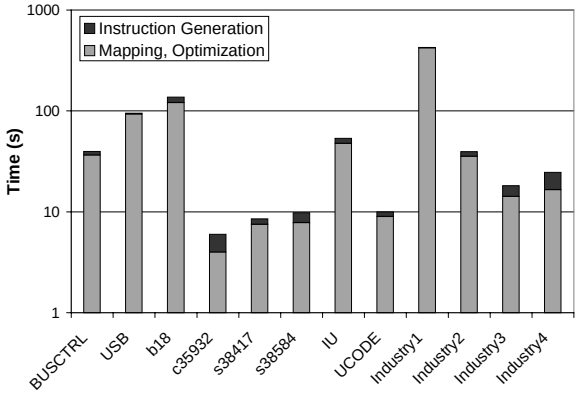

Figure 6: Compilation speed.

scheduled from the ready-front using a *selection heuristic*. Our heuristic selects a node N based on (i) its fanout and (ii) number of registers freed by scheduling N. A node with a large fanout opens up more possibilities for scheduling nodes in future time-steps, and is therefore prioritized, while prioritizing nodes that free a large number of registers is a simple, greedy strategy to minimize register usage[7]. On the other hand, if N has one or more inputs stored in memory, it's priority function is based on how many of the stored inputs' fanout are in the ready-front.

If no PEs are available to schedule a node, the current instruction is completed, and a new one started. If no registers are available, *store* operations are scheduled to spill registers. The node whose output is stored is chosen based on the number of its fanout in the ready front.

If the operands of a selected node correspond to registers that have been spilt, *load* operations are scheduled. Once the operands are in registers, the PE with the most free registers is selected to perform the operation. Subsequently, the operands are routed to the PE's inputs using the readports of register files and the interconnect system. Then a register is allocated for the output of the node. This continues until all PEs are occupied, after which new instruction is started.

4.1.1 Handling Registers Specified by the User

A register in the netlist needs to be handled in a special manner with our scheme. We simulate the behaviour of a register using memory loads and stores. Each register R is broken up into its input D and output Q. When an input to a gate G is Q, then the value that must be used to evaluate the gate is the value of the register from the *previous* user cycle (defined in Section 3.1). When the output of a gate G is connected to D, the value that must be written to the register is the value computed by G in the *current* user cycle. Thus, the SimPLE compiler transforms D into a memory load and Q into a memory store to the same memory location. Additionally, Q is constrained to be scheduled after D.

4.2 Compilation Results and Analysis

We analyze results using a combination of industrial, ISCAS and other representative benchmarks. For every result in this paper, we use 4 industrial benchmarks (Industry1-4), the integer and the microcode units of the PicoJava processor (IU and UCODE), and 6 large gate-level combinational and sequential netlists selected from ISCAS89, ITC99[6], and from common bus and USB controllers. The benchmarks range in size from 31,000 to 430,000 2-input gates.

Figure 6 illustrates the speed of our compiler on a 440MHz UltraSparc10. A fast compiler facilitates debugging, and is therefore important for simulation. We incorporate efficient data structures and make the instruction generation complexity approximately linear with the netlist size.

We also found that the storage requirement for intermediate values was very modest when using the SimPLE compiler. Further, we found that more than 2 register ports make little difference to the compilation efficiency when there are 32 or more processors. This is explained by the fact that all netlists are mapped to 2-LUTs during compilation, and sufficient parallelism exists with 32 processors to minimize overlap of values on the same processor (overlapping values on a single processor require the use of multiple readports). Henceforth we confine ourselves to SimPLE architectures with 2 readports.

5. FPGA SYNTHESIS

Prior to simulation, SimPLE must be configured onto the FPGA. This is done *only once*, after which an arbitrary number of simulations may be performed. The configuration bits for several SimPLE architectures may be produced beforehand and stored in a library. Thus, the time taken to place and route SimPLE on the FPGA does not affect the simulation speed. However, the FPGA clock speed does (Equation 2). Therefore, it is important to place and route SimPLE on an FPGA and achieve a high clock speed. This section describes our FPGA place and route procedure.

An HDL generator generates a behavioral description of SimPLE with a specific set of parameters, namely the number of processors, memory size, etc. It can also generate extra hardware to time-multiplex the SimPLE instruction if required. This description is synthesized using Synopsys' FPGA Express and mapped, placed and routed on a Virtex-2 FPGA using the Xilinx Foundation 4.1i.

5.1 FPGA Place and Route Methodology

Placement on an FPGA is extremely important in order to achieve good routability. It has been shown that correct placement of modules prior to routing can reduce congestion and enhance the clock speed considerably[8, 1]. We

SimPLE Arch		Clock Speed (MHz)	
Processors	Registers	No Macros	With Macros
32	32	61	124
32	64	56	120
48	32	48	112
48	64	43	112

Table 1: Effect of regularity-driven placement.

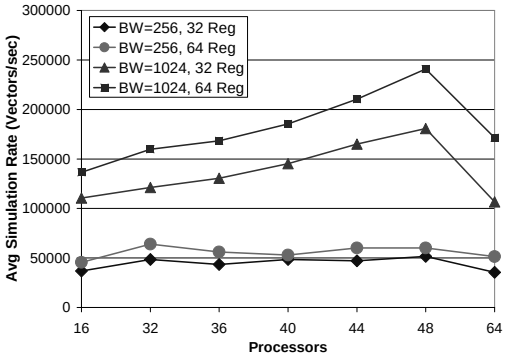

Figure 7: Simulation rate.

use a regularity-driven scheme to obtain a good placement. Every instance of SimPLE inherently has a high degree of regularity since the processing elements, memory blocks and register files are all identical to each other. The hierarchy of SimPLE, including all the regular units, is clear from Figure 3.

Our FPGA place and route methodology involves the following four steps: (i) identification of the best repeating unit in the design, (ii) compact pre-placement of the repeating unit as a single (relatively placed) hard macro, (iii) placement of the entire design using the macros and (iv) overall final routing.

From among the several macros possible in Figure 3, we experimentally found that the largest one (i.e., the top-level macro) was the best, since it had the best compaction ratio and relatively less IO. Once identified, a macro is synthesized, mapped to the FPGA CLBs and then placed. The overall description of SimPLE is instantiated in terms of the macro and then mapped, placed and routed. A more detailed description is beyond the scope of this paper.

Table 1 compares FPGA clock speeds with and without our macro strategy. All experiments were performed using the latest Xilinx Foundation 4.1i. We see clock speed improvements of up to 3x with our approach.

Using the FPGA clock cycle, along with the number of compiled instructions and the instruction width, we can compute the simulation rate using Equation 2. Figure 7 shows the simulation rate in vectors per second for various SimPLE architectures for two values of the width of the FPGA-memory bus on the board: 256 and 1024. The architecture with 48 processors is clearly the best when the FPGA-memory bus is 1024 bits wide. Wider architectures have wider instructions that need to be time-multiplexed more, and are therefore not necessarily better. With a smaller FPGA-memory bus width, several architectures were close. This indicates that the instruction width offsets gains provided by the wider architectures when the FPGA-memory bus width is small.

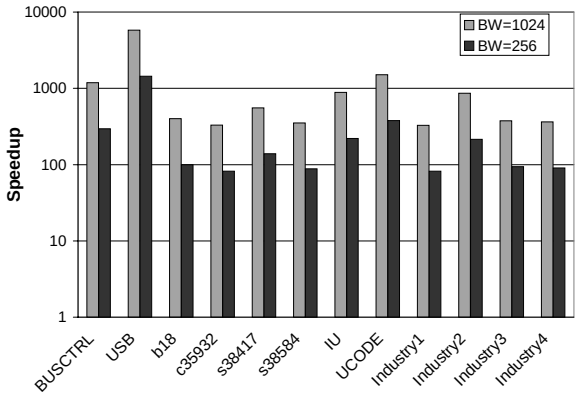

Figure 8: SimPLE vs. cycle-based simulation.

6. SIMULATION SPEEDUPS

In this section, we present actual speedups resulting from an implementation of SimPLE on a large Virtex-II FPGA as well as our first prototype on a generic board.

6.1 Speedup on Virtex-II

Based on the results in Section 5, we synthesized a version of the SimPLE processor with 48 PEs, 64 registers per PE, 2 register read ports per register file, a distributed memory system consisting of banks of 16Kbits each spanning two PEs, a memory word size of 4 bits and an interconnect latency of 2 on an 8-million gate Virtex-II FPGA (XV2V8000). We used Xilinx's Foundation tools as described in Section 5.

6.1.1 Comparison to Cycle-based Simulation

We used the publicly available cycle-based simulator Cyco, which produces straight line C code from structural verilog. We compiled and ran the C code on an UltraSparc 10 system with 1GB RAM containing a SparcV9 processor running at 440MHz. It may be noted that the time for compiling the generated C code is large (around a few hours). This is another advantage of SimPLE which has small compile times.

Figure 8 shows the speedup obtained by SimPLE with 48 processors and 64 registers running at 100MHz (restricted since most boards run at 100 MHz) over a cycle based simulator running on an UltraSparc 440MHz workstation. The right column for each benchmark indicates the speedup if the width of the FPGA-memory bus on the board is 1024 bits, while the smaller left column indicates the speedup for a FPGA-memory bus width of 256 bits. The speedups are 200-3000x for a FPGA-memory bus width of 1024 bits and decrease to 75-1000x for a FPGA-memory bus width of 256 bits.

6.1.2 Comparison to Event-driven Simulation

For this comparison, we used a commercial event-driven simulator with zero-gate delays on a 440MHz UltraSparc-10. Figure 9 shows the speedups obtained for the same benchmarks. The speedups are 100-10000x for a FPGA-memory bus width of 1024 bits and decrease to 25-2000x when the FPGA-memory bus width reduces to 256 bits[2].

[2]For examples with very low activity rates, event-driven simulation is faster than cycle-based simulation.

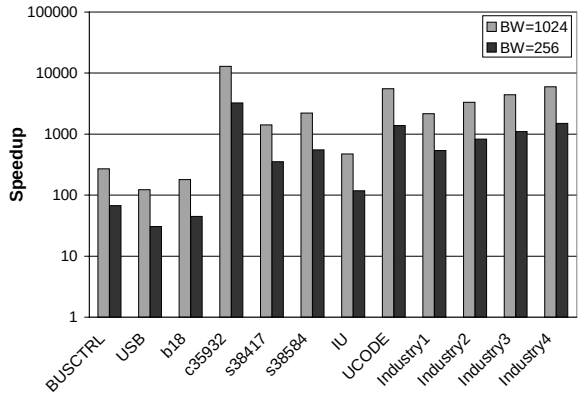

Figure 9: SimPLE vs. event-driven simulation.

6.2 Speedup using the prototype

We implemented a prototype using a generic FPGA board (ADC-RC1000) from AlphaData (www.alphadata.co.uk). The board had a Xilinx Virtex-E 2-million gate FPGA with an FPGA-memory bandwidth of 128 bits at 66 MHz. We have a fully working simulation environment that allows the user to compile and simulate a netlist, and view selected signals. We measured speedups obtained on the small prototype board for two designs. One was a 400,000-gate sequential benchmark, and the other a portion of the pipeline datapath of the PicoJava processor. For both of these, the prototype board was about 30x faster than the event-driven simulator, and 12x faster than the cycle-based simulator. These speedups are expected to increase by a factor of 6 with the Virtex-II board currently being targeted.

6.3 Where does the speedup come from?

The primary reasons for the speedups are (i) the parallelism (ii) large number of registers and memory in SimPLE (iii) high bandwidth between the FPGA and board memory and (iv) high FPGA clock speed. Superscalar processors typically execute 2-3 instructions per cycle. However, this does not usually translate to 2-3 gate evaluations per cycle; memory loads and stores when register files overflow, cache misses and long access times to main memory lower the simulation rate. In SimPLE however, the large number of registers reduces memory operations, while the array of processors permits more parallel gate evaluations than possible in a superscalar processor. The high bandwidth between the FPGA and the board memory allows quick transfer of the wide SimPLE instructions.

6.4 SimPLE in a larger system

SimPLE may be used as a transparent back-end to commercial software simulators. Portions of testbenches that consist of a large number of uninterrupted vectors may be simulated on SimPLE. The designer annotates such parts of a testbench, and compiles the design to the SimPLE architecture. During simulation, control switches from the software simulator to SimPLE using an interface such as Verilog-PLI. The SimPLE tools convert the testbench into vectors and download them (along with the compiled instructions) onto the accelerator board. Thus, SimPLE is best used as an enhancement for commercial software simulators.

7. CONCLUSIONS

In this paper, we introduced a novel approach to accelerating functional simulation. The key attributes of our approach are high-performance, low-cost, scalability and low turn-around-time (TAT). We reported speedups of up to 2000x over zero delay event-driven simulation and up to 1000x over cycle-based simulation on benchmark and industrial circuits while maintaining the cost, scalability and TAT advantages of software simulation.

8. REFERENCES

[1] J. Abke and E. Barke. A New Placement Method for Direct Mapping into LUT-based FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 27–36, August 2001.

[2] S. I. Assn. International Technology Roadmap for Semiconductors. *ITRS*, 1999. http://public.itrs.net.

[3] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[4] J. Babb, R. Tessier, M. Dahl, S. Hanano, D. Hoki, and A. Agarwal. Logic Emulation with Virtual Wires. In *IEEE Transactions on CAD of Integrated Circuits and Systems*, June 1997.

[5] J. Cong and Y. Ding. An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table based FPGA Designs. In *IEEE Transactions on CAD*, pages 1–12, January 1994.

[6] F. Corno, M. S. Reorda, and G. Squillero. RT-level ITC99 Benchmarks and First ATPG Results. In *IEEE Design and Test of Computers*, pages 44–53, July 2000.

[7] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *The 26th Annual Internation Symposium on Computer Architecture*, pages 28–39, May 1999.

[8] C. Mulpuri and S. Hauck. Runtime and Quality Tradeoffs in FPGA Placement and Routing. In *International Symposium on Field Programmable Gate Arrays*, pages 29–36, February 2001.

[9] E. Shriver and K. Sakallah. Ravel: Assigned-delay Compiled-code Logic Simulation. In *International Conference on Computer-Aided Design*, pages 364–368, November 1992.

[10] S. Trimberger. Scheduling Designs into a Time-multiplexed FPGA. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.

[11] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-multiplexed FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, February 1997.

[12] K. Westgate and D. McInnis. Reducing Simulation Time with Cycle Simulation. *Quickturn White Paper*, 2000. http://www.quickturn.com/tech/cbs.htm.

[13] Xilinx. Virtex-II 1.5v Field Programmable Gate Array: Advance Product Specification. *Xilinx Application Databook*, October 2001. http://www.xilinx.com/partinfo/databook.htm.