2009 17th IEEE Symposium on Field Programmable Custom Computing Machines

Accelerating SPICE Model-Evaluation using FPGAs

Nachiket Kapre
Computer Science
California Institute of Technology
Pasadena, CA 91125
nachiket@caltech.edu

André DeHon
Electrical and Systems Engineering
University of Pennsylvania
Philadelphia, PA 19104
andre@acm.org

***Abstract*—Single-FPGA spatial implementations can provide an order of magnitude speedup over sequential microprocessor implementations for data-parallel, floating-point computation in SPICE model-evaluation. Model-evaluation is a key component of the SPICE circuit simulator and it is characterized by large irregular floating-point compute graphs. We show how to exploit the parallelism available in these graphs on single-FPGA designs with a low-overhead VLIW-scheduled architecture. Our architecture uses spatial floating-point operators coupled to local high-bandwidth memories and interconnected by a time-shared network. We retime operation inputs in the model-evaluation to allow independent scheduling of computation and communication. With this approach, we demonstrate speedups of 2–18× over a dual-core 3GHz Intel Xeon 5160 when using a Xilinx Virtex 5 LX330T for a variety of SPICE device models.**

***Index Terms*—Spice, Analog Circuit Simulator, Spatial Computation, VLIW Scheduling, Loop Unrolling, Floating-Point**

I. Introduction

SPICE (Simulation Program with Integrated Circuit Emphasis) [1] is a circuit-simulator used to model static and dynamic analog behavior of electronic circuits. SPICE is part of the SPEC92 Floating-Point benchmarks [2] which is a collection of challenge problems for processors. Even today, accurate SPICE simulations of large sub-micron circuits can often take days or weeks (see Table I) of runtime on modern processors. Various other attempts at reducing these runtimes by parallelizing SPICE have met with mixed success (see Section II-D) SPICE does not parallelize easily on conventional processors due to the irregular structure of the computation, limited peak floating-point capacities and constraints due to scarce memory bandwidth.

Modern FPGAs contain thousands of configurable logic elements, hundreds of high-bandwidth, distributed on-chip memories and a rich interconnect. FPGAs are now large enough to support double-precision floating-point computation on a single-chip and can be customized to implement irregular floating-point datapaths. As a result, they are an attractive architectural candidate for accelerating SPICE.

When parallelizing SPICE, we must consider two phases of its operation: Matrix-Solve and Model-Evaluation. In this paper, we demonstrate how to parallelize the Model-Evaluation phase of SPICE using FPGAs; in future work we intend to parallelize the Matrix-Solve phase and integrate a complete SPICE simulator. The SPICE Model-Evaluation phase has high data parallelism consisting of thousands of independent

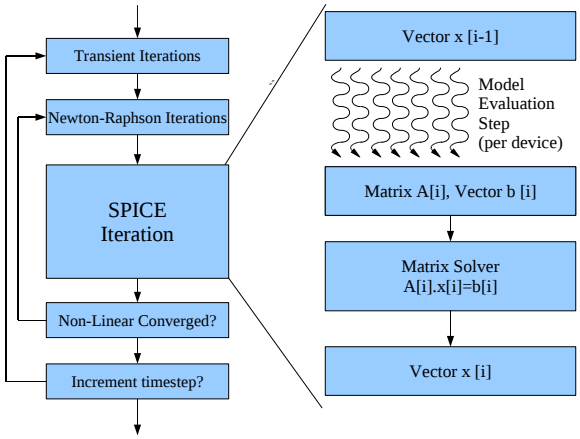


Fig. 1: Flowchart of a SPICE Simulator

device evaluations each requiring hundreds of floating-point operations. A completely unrolled spatial representation of the Model-Evaluation compute graph is too large to fit entirely on a single FPGA today (Table III) forcing us to either partition processing across several chips or virtualize it over the single chip. In this paper we estimate multi-chip implementations (Section III) and investigate architectures that virtualize computation over limited single-FPGA resources efficiently to provide speedups over a sequential implementation.

The key contributions of this paper include:

- Design and demonstration of single-FPGA implementations that accelerate model-evaluation for a variety of SPICE device models using IEEE double-precision floating-point arithmetic.
- A Verilog-AMS compiler for optimizing high-level device model descriptions with an extensible backend for targeting various computing architectures (*e.g.* FPGAs, GPUs).
- Quantitative comparison of two strategies for programming the FPGA using a VLIW architecture: Loop Unrolling and Software Pipelining with GraphStep Scheduling.
- Quantitative empirical comparison of SPICE model evaluation on the Intel Xeon processor and Virtex-5 FPGA.

II. Background

A. Structure of SPICE Model Evaluation

SPICE simulates the dynamic analog behavior of a circuit described by non-linear differential equations. SPICE circuit

978-0-7695-3716-0/09 $25.00 © 2009 IEEE
DOI 10.1109/FCCM.2009.14

equations model the linear (*e.g.* resistors, capacitors, inductors) and non-linear (*e.g.* diodes, transistors) behavior of devices and the conservation constraints (*i.e.* Kirchoff's conservation laws—KCL) at the different nodes and branches of the circuit. SPICE solves the non-linear circuit equations by alternately computing small-signal linear operating-point approximations for the non-linear elements and solving the resulting system of linear equations until it reaches a fixed point. The linearized system of equations is represented as a solution of $A\vec{x} = \vec{b}$, where A is the matrix of circuit conductances, $\vec{b}$ is the vector of known currents and voltage quantities and $\vec{x}$ is the vector of unknown voltages and branch currents. The simulator calculates entries in A and $\vec{b}$ from the device model equations that describe device transconductance (*e.g.*, Ohm's law for resistors, transistor I-V characteristics) in the **Model-Evaluation** phase. It then solves for $\vec{x}$ using a sparse-direct linear matrix solver in the **Matrix-Solve** phase. We illustrate the steps in the SPICE algorithm in Figure 1. The inner loop iteration supports the operating-point calculation for the non-linear circuit elements, while the outer loop models the dynamics of time-varying devices such as capacitors.

We make the following observations about the requirements and characteristics of the Model-Evaluation phase.

- At the start of the simulation, the simulator processes all the devices in the circuit to build A and $\vec{b}$. At subsequent timesteps, *only* the entries associated with the non-linear and time-varying elements change and must be recalculated.
- Each device in the circuit updates a constant number of entries in the matrix corresponding to its node terminals.
- For non-linear elements, the simulator must search for an operating-point using Newton-Raphson iterations. This requires repeated evaluation of the non-linear model equations multiple times per time-step.
- For time-varying components, the simulator must recalculate their contributions at each timestep based on voltages at several previous timesteps. This also requires a re-evaluation of the device-model in each timestep.

For circuits dominated by non-linear transistor devices, the simulator can spend almost half its time evaluating the device models (see "no parasitics" case in Table I; we generated datapoints in this table by running spice3f5 on an Intel Xeon 5160 using Simucad memory benchmarks [22]). For circuits dominated by linear parasitics (*e.g.* parasitic capacitances), simulation time may be dominated by the Matrix-Solve. Since we are ultimately interested in accelerating both Model-Evaluation and Matrix-Solve (See Section VIII), it is important to understand how far we can improve Model-Evaluation runtimes even in these cases where it is currently not the dominant percentage of runtime.

Furthermore, note that as transistor devices shrink in feature-size, the complexity of the device models required to simulate them correctly grows over time. Newer device models often have complexity 4–5× that of the classic `bsim3` model [10] (*e.g.* compare `psp` [11] and `bsim3` models in Table IV). This further motivates the need to accelerate device model evaluation to avoid paying a large modeling cost for future sub-micron circuit netlists. For example, when model evaluation time increases by 5×, the no parasitic `ram8k` will spend 80% of its time in model evaluation, and the parasitic case will spend 36%.

TABLE I: spice3f5 runtime distribution (Intel Xeon 5160)

Benchmark Circuits (bsim3)	**Model Eval. (seconds)**	**Matrix Solve (seconds)**	**Model Eval. (Percent)**
no parasitics			
`ram2k`	55	10	84
`ram8k`	237	87	73
`ram64k`	2005	1082	64
with parasitics			
`ram2k`	69	149	31
`ram8k`	300	2395	11
`ram64k`	2597	99487	3

B. Parallelism Potential

We enumerate the potential of parallelizing Model-Evaluation here:

- **Data Parallelism**: Each individual Model-Evaluation (*e.g.* for each transistor) within a timestep is completely independent.
- **Pipeline Parallelism**: Model-Evaluation operations can be represented as an acyclic feed-forward dataflow graph (DAG) with nodes representing operations and edges representing dependencies between the operations.
- **Specialization Potential**
  - **Static Workload**: The Model-Evaluation phase process all devices in the circuit in each timestep.
  - **Early Bound Graph**: The Model-Evaluation compute graphs are known entirely in advance and do not change during the simulation.
  - **Limited Diversity of Graphs**: Within a simulation, there may be very few unique device models active. (*e.g.* typically all transistors in a circuit will use same `bsim3` model).
  - **Parameterized Reuse of Graphs**: Individual device instances are customized using *parameters*. Typically the CMOS process determines most of these *parameters* leaving a handful of parameters which vary from device to device (*e.g.* W, L of a transistor).

C. Architecture Potential

There are several competitive architectural choices for accelerating floating-point applications (See Table II). These architectures exploit different forms of parallelism, support various programming models and require differing amount of programming effort. A full comparison between all architectures in Table II is beyond the scope of this paper (see Section VIII).

D. Related Work

Previous attempts [4], [5] to accelerate Model-Evaluation using FPGAs used a VLIW approach and required table-lookup model evaluation, trading off accuracy for capacity,

TABLE II: Peak Floating-Point Throughputs

Family	Intel Xeon	Xilinx Virtex-5	IBM Cell	NVIDIA GPU	AMD GPU	Clearspeed
Chip	5160	LX330T	PowerXCell8i	GTX-280	AMD 9270	CSX700
Technology	65 nm	65 nm	65 nm	65 nm	55 nm	90 nm
Clock	3 GHz	200 MHz	3.2 GHz	1.3 GHz	750 MHz	250 MHz
Double-Precision (GFLOPS)	12	11.4	102.4	74	240	96
Single-Precision (GFLOPS)	24	33	204.8	624	1200	96
Power	100 Watts	20–30 Watts	92 Watts	236 Watts	220 Watts	9 Watts

TABLE III: Estimated Speedup on Multi-FPGA Designs

Device Models	**FPGAs Required**			**Total Speedup**	
	Fully Spatial	Virtual Wires	No IO Limits	Fully Spatial	Virtual Wires
`bjt`	6	4	2	25	25
`diode`	1	1	1	26	26
`hbt`	41	9	6	264	132
`jfet`	1	1	1	21	21
`mos1`	7	4	2	10	10
`vbic`	14	4	3	67	67
`mos3`	28	4	3	53	53
`mextram`	850	64	52	602	120
`bsim3`	319	25	18	199	49
`bsim4`	107	16	12	223	111
`psp`	1250	64	61	664	110

in order to make implementation feasible. This FPGA implementation called Tina [5] used the Marc-1 reconfigurable board with 9 XC4005 FPGAs coupled to a discrete FPU to implement Model-Evaluation (speedup figures are unpublished). Our single-FPGA implementation exploits a different parallelization approach that exploits the significantly larger FPGA densities available today to provide speedups when compared to the latest generation processors without any lookup-table approximations.

[9] tries to parallelize existing SPICE Model-Evaluation code using OpenMP pragmas (no code modification) and shows limited speedups and scaling (saturates at 2× with 4 processors). [8] uses a multi-threaded implementation and demonstrates moderate speedups (5× with 8 processors) and decent scaling trends without sacrificing quality. [7] parallelizes transient simulations by optimistically evaluating multiple timesteps in parallel (3× with 8 processors) without specifically accelerating individual Model-Evaluations. Xyce is a highly-parallel simulator engineered for supercomputers that demonstrates good speedups (24× on 40 processors) only on sufficiently large circuits [6]. Recently, GPUs have been used to accelerate SPICE Model-Evaluation by an impressive 10×–50× (Double-Precision evaluation in [12]) and 32×–40× (Single-Precision evaluation in [13]). Our FPGA implementation exploits a different parallelization approach and high on-chip communication and memory bandwidth to deliver an order of magnitude or greater acceleration of full, double-precision floating-point Model-Evaluation using a single FPGA without sacrificing accuracy.

III. Multi-FPGA Designs

A fully spatial FPGA implementation of Model-Evaluation maps every operation in the computation to dedicated FPGA logic and uses FPGA interconnect to physically implement communication between the operations. With suitable pipelining of communication between the operations, we can start a new evaluation of the compute graph in each cycle. If cost is not a concern, this approach provides two to three orders of magnitude speedup over sequential implementations on an Intel Xeon 5160 3 GHz microprocessor as can be seen in Table III when compared to a Xilinx Virtex5 LX330T.

However, a fully-spatial implementation of the SPICE Model-Evaluation must be partitioned across multiple FPGAs. Since external FPGA IO is limited, we have to choose a multi-FPGA configuration that can accommodate communication over external IO without any serialization. We use VPR [14] to route inter-FPGA communication and estimate the minimum FPGAs required for an IO-limited mapping in Table III (Column Fully-Spatial). We can reduce cost at the expense of performance by serializing communication over external IO [15] (see Column Virtual-Wires in Table III). Both cases still require multiple FPGAs for the large Model-Evaluation graphs. Performance and design cost are dictated purely by our ability (or inability) to move data across external pins. Hence, we are motivated to consider affordable single-FPGA designs for the problem and avoid external IO entirely.

IV. Organizing Principles

We highlight the principles used to design an efficient single-FPGA architecture for Model-Evaluation.

1) Virtualization: We must virtualize the computation and the communication on finite hardware. The virtualized architecture consists of heterogeneous floating-point operators coupled to local, high-bandwidth memories and interconnected to other operators through a communication network (See Figure 2). This virtualization must be managed with minimum overhead to allow most of the resources to service the actual application.

2) Balanced Provision of Resources: Model-evaluation graphs contain a diverse set of floating-point operators such as adds, multiplies, divides, square-roots, exponentials and logarithms. Not all operators are used equally. We must choose an operator mix proportional to the frequency of their use since spatial implementations of floating-point operators can be quite expensive. We must also tune the interconnect richness to

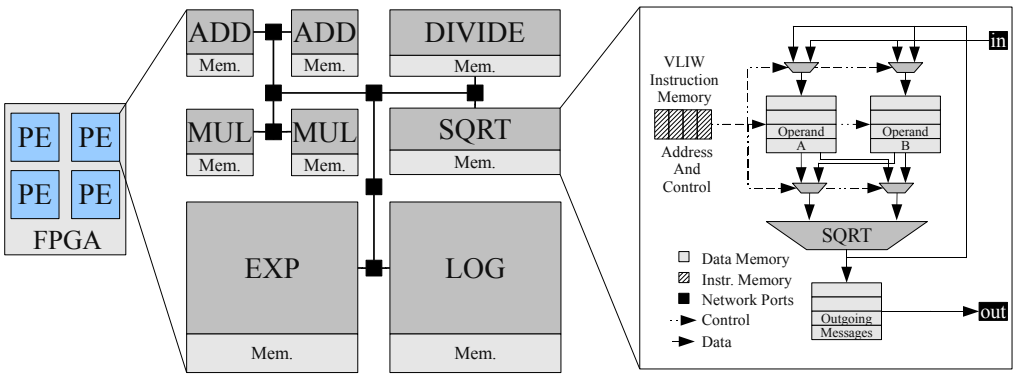


Fig. 2: An Example Parallel Architecture for SPICE Model-Evaluation

properly support communication traffic between the operators. For example, in Figure 2 we provision multiple adders and multipliers and just one each of the remaining operators within a Processing Element (PE). The FPGA consists of multiple PEs. Different device models will have different operator distributions and will require suitable operator mixes.

3) Static Scheduling: When we virtualize operation over finite hardware, we must co-ordinate access to the virtualized resources. The logic to mediate access to shared resources at runtime costs area and introduces additional latency into the execution. We minimize this overhead by statically scheduling the resources offline in VLIW (Very-Large Instruction Word [16]) fashion, thus avoiding runtime decisions entirely. The VLIW instruction for the shared operator consists of read/write address and control signals for the input and output memories along with multiplexer control signals for the datapath (See Figure 2). The time-multiplexed switch also contains configuration instructions that provide routing information to schedule communication between the input and output ports.

4) Scheduling to Maximize Resource Usage: An offline VLIW scheduler has access to the entire program graph and has a global view of available resources. In our virtualized architecture, we expose both the scheduling of floating-point instructions and communication between these instructions to the scheduler. A good scheduler will attempt to schedule the critical path in the program graph first and then try to fit the remaining operations into the idle slots left behind. However, model-evaluation graphs are irregular and do not have sufficient work to fill these slots. A key challenge is to expose work to use these slots productively.

V. VLIW Architectures on FPGAs

We consider two strategies for scheduling our VLIW architecture.

A. Conventional Loop Unrolling

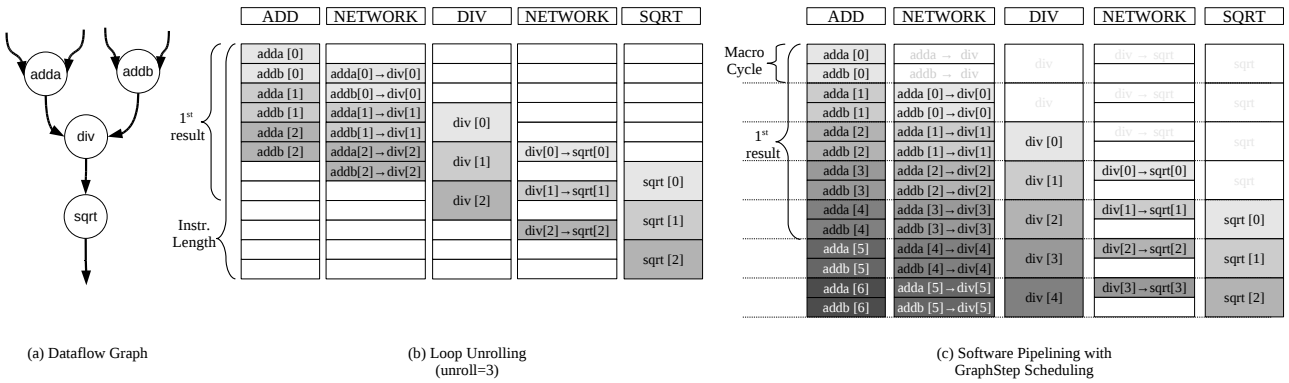
When scheduling single loop iterations on fully-pipelined hardware, the total number of active pipeline stages doing useful work may be limited. We can create additional work for the scheduler to fill these empty pipeline slots by unrolling multiple iterations of the loop. Loop Unrolling on Model-Evaluation graphs is possible with no increase in the critical path since iterations are independent of each other. This allows the scheduler to get better utilization of provisioned hardware resources. The per-iteration efficiency gains more than compensates for the slight increase in scheduling latency. For example, in Figure 3, the latency to get the output of a single iteration is 8 cycles. After unrolling 3 iterations, the total latency increases to 12 cycles, but the average per-iteration latency drops to 4 cycles. However, these efficiency gains come at the expense of increased memory cost for storing intermediate state and instruction context. We now need to store 12 VLIW configuration instructions instead of just 6 for the single-iteration case. Also, the intermediate state requirements increase from 3 registers to 9 registers. For 100 total iterations of example graph, this unrolled design will require $100 \times 4 = 400$ cycles. For 100 total iterations of example graph, this unrolled design will require $100 \times 4 = 400$ cycles. We empirically determine the extent of the unroll that provides the most judicious use of resources (Section VII).

B. Software Pipelining with GraphStep Scheduling

Software pipelining with Modulo Scheduling [17], [18] improves the per-iteration performance by initiating execution of successive loop iterations at a rate faster than their individual execution latencies (which in our case is the resource-constrained initiation interval) without requiring any unrolling. It overlaps execution of different portions of the loop in a single repetitive *macro-cycle*. The benefit of a software-pipelined schedule is that a single schedule is valid for all iterations thereby saving instruction storage costs. It does increase the amount of intermediate state for instructions communicating across macro-cycle boundaries. For example, in Figure 3 we need only 2 cycles to schedule all the instructions and communication between instructions in a macro-cycle (throughput is 1 result every 2 cycles) while the result of the first iteration is available after 5 macro-cycles (latency is 10 cycles). For 100 iterations of the example graph, our software pipelined design will require $(100+4) \times 2 = 208$ cycles which is a speedup of almost $2\times$ over loop unrolling example (note that the +4 accounts for the initial macro-cycles required to fill the scheduled pipeline).

For our scheduling problem:

- We must typically evaluate a large number of devices compared to depth of the graph (number of instructions along the critical path).



(a) Dataflow Graph

(b) Loop Unrolling (unroll=3)

(c) Software Pipelining with GraphStep Scheduling

Fig. 3: Comparison of the two VLIW Scheduling Approaches

- We have access to several on-chip distributed FPGA memories to store intermediate state.
- We have to generate a schedule for the switching network in addition to compute scheduling.
- We are able to provision interconnect resources as necessary to meet the demands of the application.

For data-independent loop-iterations, conventional modulo-scheduling will generate a schedule while obeying intra-iteration dependencies. Based on our observations, we propose a simplified scheduler described here:

- We distribute instructions to different operators with load-balancing and schedule all instructions on an operator **without** precedence constraints.
- Within a macro-cycle, all instructions are processed on the operators and all instruction dependencies are routed on the communication network in parallel with each other.
- We schedule data movement between operators concurrently but independently from the computation. This adds an extra macro-cycle of latency before the dependent instructions can see their inputs.
- We levelize the instructions in the compute graph based on an ASAP ordering of the instructions within an iteration. We retime the inputs to each instruction based on these levels and ensure they receive inputs from the correct iteration.

These simplifications allow us to densely pack both the floating-point operators and the communication network between these operators with high efficiency. We call this GraphStep scheduling since it is inspired by the GraphStep system architecture [19].

VI. Framework

We now explain the experimental framework we use in our experiments.

A. Verilog-AMS

Modern SPICE simulators accept a wide-variety of device models that cater to different designer requirements. Rather than manually rewrite each device model for each unique simulator interface, models are now released as simulator-independent Verilog-AMS code [20], [21]. We use open-source Verilog-AMS descriptions of a variety of devices available from Silvaco [22]. We developed a Verilog-AMS compiler that supports a subset of the Verilog-AMS language for device models [20]. We compile the device model equations into a flexible intermediate representation that allows us to perform analysis, optimization and code generation for different architectures easily. Our compiler currently performs simple dead-code elimination, mux-conversion, constant-folding, and identity simplification optimizations. It generates a generic feed-forward dataflow graph of the computation that is processed by architecture-specific backend tools.

TABLE IV: Optimized Instruction Counts

Device Models	Instruction Distribution					
	Add	Multiply	Divide	Sqrt	Exp	Log
bjt	22	30	17	0	2	0
diode	7	5	4	0	1	2
hbt	112	57	51	0	23	18
jfet	13	31	2	0	2	0
mos1	24	36	7	1	0	0
vbic	36	43	18	1	10	4
mos3	46	82	20	4	3	0
mextram	675	1626	397	22	52	37
bsim3 v3.2	283	634	122	9	8	1
bsim4 v3.0	222	286	85	16	24	9
psp	1345	2319	247	30	19	10

B. Tool Flow

The different FPGA organizations considered in this paper require a variety of mapping tools to implement the dataflow graphs on the system.

- For the fully-spatial implementations spanning multiple-FPGAs, we use a packing algorithm that assigns nodes of the graph onto FPGAs with user-supplied area and IO constraints. We use VPR [14] to place the packed instructions on the different FPGAs and calculate the minimum channel-width required to route the fully-spatial design. We then find the minimum system size necessary to permit a feasible fully-spatial design (See Table III).
- For the two VLIW designs, we share part of the mapping flow. We start by first deciding system size (*i.e.* number of floating-point operators) and partitioning the nodes based on locality using a high-quality partitioner MLPart [23]. We

TABLE V: FPGA Cost Model

	Area (Slices)	Latency (clocks)	Speed (MHz)	Ref.
Add	296	8	280	[26]
Multiply	611	9	237	[26]
Divide	1499	57	258	[26]
Square Root	822	57	282	[26]
Exponential	1022	30	200	[27]
Logarithm	1561	30	200	[27]
PE support logic	82	-	300	-
BFT T-Switchbox	48	2	300	-
BFT Pi-Switchbox	64	2	300	-
Switch-Switch Wire	32	2	300	-

then provision the number of hardware operators of each type according to instruction frequency and allocate them to partitions using need-proportional distribution [24], [25]. Each partition can process operations of a single operator type. We then reassign nodes paired with invalid operators to the nearest valid operator that is least occupied.

- For loop unrolling, we provide an unrolled graph to the partitioner/placer. Once instructions have been placed on proper operators, we then use a greedy list scheduler to assign those instruction to schedule slots on the operator. We use a priority function that prefers nodes along the circuit critical path. We schedule communication between the nodes using a greedy time-multiplexed router that uses A* routing. We developed this scheduler and router as part of the Graph Machine project [19], [28], [32].
- For the GraphStep scheduler, we separately schedule computation and communication. The compute scheduler simply assigns all instructions to consecutive scheduling slots on the fully-pipelined hardware operator. The communication scheduler routes every edge with A* routing without any precedence constraints.

C. FPGA Implementation

We use spatial implementations of individual floating-point *add*, *multiply*, *divide* and *square-root* operators from the Xilinx Floating-Point library in CoreGen [26]. For the *exp* and *log* operators we use FPLibrary from Arénaire [27] group. Neither of these implementations support denormalized (subnormal) numbers. We use the Xilinx Virtex 5 LX330T for our experiments. We limit our implementations to fit on a **single-chip** and use only on-chip memory resources for storing intermediate results. The time-multiplexed switches are a collection of multiplexers whose select bits are generated by a configuration context memory on each cycle. We pipeline the wires between the switches and between the floating-point operator and the coupled-memories for high-performance. You can find additional details of our time-multiplexed switches in [28]. We synthesize and implement a sample double-precision 8-operator design for the `bsim3` model on a Xilinx Virtex-5 device [29] using Synplify Pro 9.6.1 and Xilinx ISE 10.1. We provide placement and timing constraints to the backend tools and attain a frequency of 200 MHz (See Table V). Aggressive pipelining of *exp* and *log* operators should enable higher rates.

D. Sequential Baseline

We compile Verilog-AMS models into loop-unrolled, multi-threaded C-code for our sequential baseline comparison. We measure sequential performance on a dual-core 3 GHz Intel Xeon 5160 processor with a 4MB shared L2 cache and 16GB main memory running 64-bit Debian Linux. We use `gcc-4.3.3` (-O3) with either the GNU libm math library or the Intel MKL vector math library (with accelerated vector implementations of math functions) to compile device models. We use PAPI 3.6.2 [30] performance counters to measure runtimes and report runtime averaged across the 16384 device evaluations.

VII. Evaluation

In this section, we discuss the tradeoffs between the two VLIW FPGA architectures and compare their performance to a sequential mapping (Section VI-D).

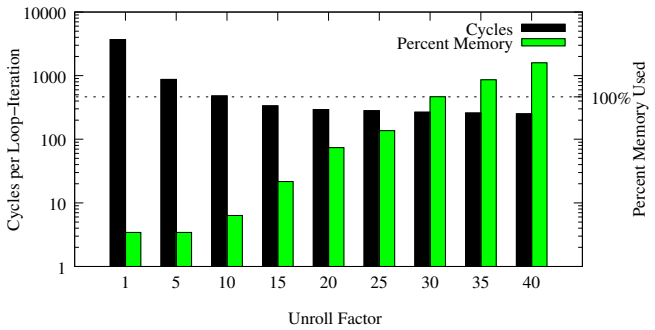
We define a Processing Element (PE) as a configuration of floating-point operators interconnected by a shared network (See Figure 2). A PE can have a variable number of floating-point operators and an FPGA can have multiple PEs. The smallest PE has one floating-point operator of each type (minimum 6 operators). There is no network between PEs. Both schedulers map the compute graphs to a PE. We measure cycles required per device evaluation as the average number of cycles required to run a single iteration on one PE divided by the number of PEs that can fit on that single-FPGA. We use a Butterfly Fat-Tree (BFT) topology for our mapping experiments, and we tune the bisection bandwidth available in the BFT by increasing the Rent parameter p (Bisection Bandwidth $IO = c \times N^p$). A network with a $p = 0$ has as much bandwidth as a ring while a network with a $p = 1$ is equivalent to a crossbar. The smallest BFT PE has 8 operators.

For the two scheduling strategies, we must pick the best design configuration for comparison with the sequential baseline. For Loop Unrolling, this requires choosing three parameters: extent of unroll (unroll factor), number of floating-point operators per PE and the Rent parameter (p) of the shared network. For GraphStep Scheduling, we must pick the number of floating-point operators per PE and the Rent parameter of the shared network.

A. Loop Unrolling

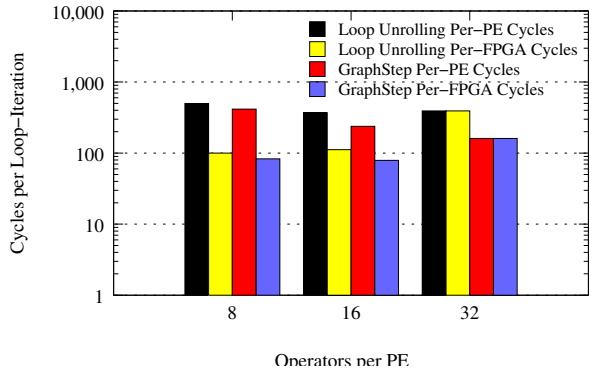
1) Extent of Unroll: In Figure 4, we show the impact of unroll factor on performance per-iteration for `bsim3` graphs when using 16 floating-point operators per PE and a network with a $p = 0.5$. As expected, we initially see an improvement in performance per iteration as we increase the unroll factor. After 20 unrolls, performance improvements start to diminish. By 35 unrolls, we exceed on-chip BlockRAMs capacity of a single FPGA (XC5VLX330T).

2) Number of Floating-Point Operators: In Figure 5, we see marginal improvement in performance per-PE when using larger PEs. When considering per-FPGA speedups, we find that it is best to use the smallest sized PE design (*i.e.* 8 operators per PE) and populate the FPGA with several instances



16 operators with 4 adders, 8 multipliers and 1 each of rest

Fig. 4: Impact of Unroll Factor for bsim3 ($p = 0.5$)



8 operator case has 5 PEs/FPGA with 2 add, 2 mult
16 operator case has 3 PEs/FPGA with 4 add, 8 mult
32 operator case has 1 PE/FPGA with 8 add, 18 mult, 3 div

Fig. 5: Impact of PE size for bsim3

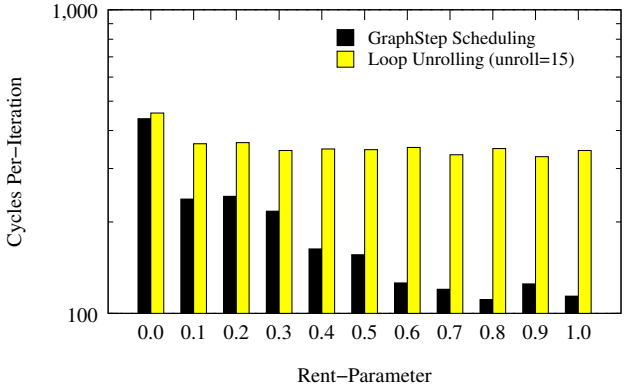
of this 8-operator PE instead of using the larger PEs. The scheduler is able to efficiently utilize available resources even at small operator counts.

3) Interconnect Richness: In Figure 6, we illustrate the impact of varying interconnect richness for an unroll factor of 15. We see that increasing interconnect richness results in no significant improvement in performance. The runtime is dominated by computation from the unrolling and dependencies within loops while communication requirements are limited.

B. Software Pipelining with GraphStep Scheduling

1) Number of Floating-Point Operators: In Figure 5, we see a significant improvement in performance per-PE as we scale to larger PEs. This is because the GraphStep scheduler separately schedules computation and communication and is able to generate a compact schedule. When considering per-FPGA performance, the 16-operator PE is the most efficient design and marginally beats the 8-operator design by a few cycles. Both these designs can fit multiple PEs per FPGA and lower the average number of cycles required.

Note that we do not count the prologue and epilogue costs of a software-pipelined schedule as circuit netlists are sufficiently large compared to depth of the compute graphs and on-chip memories are large enough to hold the dataflow graph IO (*e.g.* `ram2k` from Table I has 17000 transistors while the



32 operators with 8 adders, 18 multipliers and 3 dividers

Fig. 6: Impact of Interconnect Richness (bsim3)

`bsim3` model has only 100 levels. The 46K non-zeros in $\vec{A}$ and 4K rows in $\vec{b}$ and $\vec{x}$ can fit in $\approx 30\%$ of the BRAMs on the XC5VLX330T).

2) Interconnect Richness: In Figure 6, we observe that as we increase the Rent parameter of the network, we can improve performance by as much as a factor of 3 (this increases interconnect area from 3% to 8% since the floating-point datapaths account for most of the system area). A GraphStep scheduled design exposes all communication to an independent phase that increases the bandwidth requirements on the network. A richer network is better able to support this increased traffic demand.

C. Speedup Comparison

We compare the performance achieved on an Intel Xeon (with loop-unrolling and multi-threading) with that achieved on the best single-FPGA configuration using the two scheduling strategies applied independently as well as simultaneously. In Figure 7 we observe that Loop-Unrolling and GraphStep scheduling together provide the best speedup over a sequential implementation that either of them can provide independently. For most small designs (`bjt, diode, jfet, mos1`), Loop Unrolling offers better speedups than GraphStep scheduling since a larger degree of unrolling is possible and performance gets amortized across the unrolled iterations. For the larger device models, Loop-Unrolling can actually slow down evaluation (`mextram, psp`) as these large designs are harder to unroll and fit within a single-FPGA. The GraphStep design is better able to exploit limited hardware resources with efficient scheduling.

VIII. Future Work

We identify the following broad areas for additional research that can improve upon our current parallel design or extend its applicability.

- A parallel solution to the sparse **Matrix-Solve** phase is essential for achieving balanced total speedup for the SPICE application. [31] demonstrates a potential for at least 10× speedup for sparse-direct LU factorization.

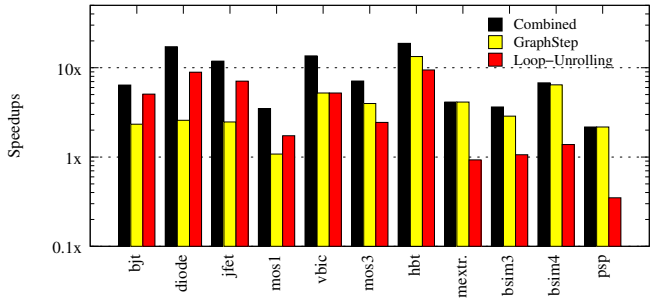


Fig. 7: Speedup Comparison (Virtex5 vs. Xeon 5160)

- Reduced-precision floating-point datapaths provide the potential to deliver even higher acceleration per FPGA. Additional work is needed to determine the precision required to achieve a given accuracy requirement.
- Table II suggests the latest NVIDIA GPU and IBM Cell processors have impressive raw double-precision performance. It will be useful to characterize the performance they can actually deliver for SPICE Model-Evaluation.

IX. Conclusions

A single FPGA can accelerate SPICE Model-Evaluation computation by 2–18× over sequential single-core microprocessor implementations. Fully-Spatial implementations of Model-Evaluation graphs can deliver two to three orders of magnitude speedups but require 10s–100s of FPGAs to provide that speedup. With limited on-chip FPGA memory capacities, Loop Unrolling of independent iterations is effective at exploiting parallelism only for small loop bodies. Software Pipelining with GraphStep scheduling can offer better speedups for larger loop bodies. Efficient single-FPGA schedules are possible when performing both Loop-Unrolling and GraphStep scheduling because we can separate the computation and communication phases of a loop-iteration and schedule multiple-iterations simultaneously.

References

[1] L. W. Nagel, "SPICE2: a computer program to simulate semiconductor circuits," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1975.

[2] SPEC, "Standard performance evaluation corporation," *Manassas, VA, USA*, 1995.

[3] A. Vladimirescu, D. Weiss, M. Katevenis, Z. Bronstein, A. Kfir, K. Danuwidjaja, K. Ng, N. Jain, and S. Lass, "A vector hardware accelerator with circuit simulation emphasis," in *24th Conference on Design Automation*, 1987, pp. 89–94.

[4] Q. Wang and D. M. Lewis, "Automated field-programmable compute accelerator design using partial evaluation," *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 145–154, 1997.

[5] M. van Ierssel, "Circuit simulation on a field programmable accelerator," Master's Thesis, University of Toronto, 1995.

[6] S. Hutchinson, E. Keiter, R. Hoekstra, H. Watts, A. Waters, R. Schells, and S. Wix, "The xyce parallel electronic simulator - an overview," *IEEE International Symposium on Circuits and Systems*, 2001.

[7] W. Dong, P. Li, and X. Ye, "WavePipe: parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," in *45th ACM/IEEE Design Automation Conference*, 2008, pp. 238–243.

[8] P. Lee, S. Ito, T. Hashimoto, J. Sato, T. Touma, and G. Yokomizo, "A parallel and accelerated circuit simulator with precise accuracy," in *Proceedings of the 7th Asia and South Pacific Design Automation Conference*, 2002, pp. 213–218.

[9] T. Weng, R. Perng, and B. Chapman, "OpenMP implementation of SPICE3 circuit simulator," *International Journal of Parallel Programming*, vol. 35, no. 5, pp. 493–505, Oct. 2007.

[10] P. Ko, J. Huang, Z. Liu, and C. Hu, "BSIM3 for analog and digital circuit simulation," in *Proceedings of the IEEE Symposium on VLSI Technology CAD*, 1993, pp. 400–429.

[11] G. Gildenblat, X. Li, W. Wu, H. Wang, A. Jha, R. V. Langevelde, G. Smit, A. Scholten, and D. Klaassen, "PSP: an advanced Surface-Potential-Based MOSFET model for circuit simulation," *IEEE Transactions on Electron Devices*, vol. 53, no. 9, pp. 1979–1993, 2006.

[12] A. M. Bayoumi and Y. Y. Hanafy, "Massive parallelization of SPICE device model evaluation on GPU-based SIMD architectures," in *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*. Cairo, Egypt: ACM, 2008, pp. 1–5.

[13] K. Gulati, J. F. Croix, S. P. Khatri, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, 2009, pp. 403–408.

[14] V. Betz, "VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs," <http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html>, March 27 1999, version 4.30.

[15] J. Babb, R. Tessier, and A. Agarwal, "Virtual wires: overcoming pin limitations in FPGA-based logicemulators," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp. 142–151.

[16] J. A. Fisher, "The VLIW machine: A multiprocessor for compiling scientific code," *Computer*, vol. 17, no. 7, pp. 45–53, 1984.

[17] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *SIGMICRO Newsl.*, vol. 12, no. 4, pp. 183–198, 1981.

[18] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1988, pp. 318–328.

[19] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon, "GraphStep: a system architecture for Sparse-Graph algorithms," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 143–151.

[20] L. Lemaitre, G. Coram, C. McAndrew, K. Kundert, M. Inc, and S. Geneva, "Extensions to Verilog-A to support compact device modeling," in *Proceedings of the International Workshop on Behavioral Modeling and Simulation*, 2003, pp. 134–138.

[21] B. Wan, B. Hu, L. Zhou, and C. Shi, "MCAST: an abstract-syntax-tree based model compiler for circuit simulation," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2003, pp. 249–252.

[22] "Open-Source simucad Verilog-A models," http://www.simucad.com, 2004.

[23] A. Caldwell, A. Kahng, and I. Markov, "Improved Algorithms for Hypergraph Bipartitioning," in *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2000, pp. 661–666.

[24] Y. Lin, "Recent developments in high-level synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 2, no. 1, pp. 2–21, 1997.

[25] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 768–781, 1989.

[26] "Xilinx core generator Floating-Point operator."

[27] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems*, vol. 31, no. 8, pp. 537–545, Dec. 2007.

[28] N. Kapre, N. Mehta, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet switched vs. time multiplexed FPGA overlay networks," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 205–216.

[29] Xilinx, *Virtex-5 Datasheet: DC and switching characteristics*, 2006.

[30] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: a portable interface to hardware performance counters," in *Proc. Dept. of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[31] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa, "Sparse LU decomposition using FPGA," in *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[32] Nikil. Mehta "Time-Multiplexed FPGA Overlay Networks On Chip," Master's Thesis, California Institute of Technology, Pasadena, 2006.