Appearing in *IEEE International Conference on Field-Programmable Technology* (FPT 2011), December 12–14, 2011

VLIW-SCORE: Beyond C for Sequential Control of SPICE FPGA Acceleration

Nachiket Kapre
Imperial College London
London, SW7 2AZ
nachiket@imperial.ac.uk

André DeHon
University of Pennsylvania
Philadelphia, PA 19104
andre@acm.org

***Abstract*—Many stand-alone, FPGA-based accelerators separate the implementation of a computation into two components – (1) a large parallel component that is realized as hardware on spatial FPGA fabric and (2) a small control and co-ordination component that is realized as software on embedded soft-core processors like an off-the-shelf Xilinx Microblaze (or host offchip CPU). While this hardware-software partitioning methodology allows the designer to lower design effort when composing the accelerator system, it introduces unnecessary Amdahl's Law bottlenecks and limits scalability. In this paper, we show how to avoid these limitations with VLIW-SCORE: a combination of a high-level parallel programming framework called SCORE and a custom, hybrid VLIW hardware organization. We demonstrate the benefits of this methodology for the SPICE circuit simulator when implementing the simulation control algorithms. With our spatial mapping flow we are able to improve performance by ≈30% (mean across circuit benchmarks) when compared to the Microblaze implementation for the Xilinx Virtex-6 LX760 FPGA. For complete application acceleration, we see an improved speedup from 1.9× for the Microblaze-based design to 2.6× for the hybrid, custom VLIW implementation when comparing a Xilinx Virtex-6 LX760 FPGA (40nm) with an Intel Core i7 965 CPU (45nm).**

I. Introduction

Hardware-software partitioning approaches to system design allow us to integrate parallel fabrics such as FPGAs with sequential processors such as Intel CPUs into a unified accelerator design. Modern FPGAs can be configured to implement embedded sequential processors like the Xilinx Microblaze [21] or the Altera NIOS [1] to reduce or even eliminate the dependence on host offchip CPUs. These embedded processors have a small area footprint on the FPGA fabric and allow us to devote most of the FPGA resources for implementing the core parallel computation. However, in many cases, this partitioning approach is not driven by fundamental mapping requirements but pragmatic considerations such as lower software development time. In this partitioning methodology, the designer offloads the sequential fraction of a large parallel computation to the embedded processor [15]. However, this can introduce unnecessary performance bottlenecks and limit smooth scalability of the design to larger FPGA capacities. It is possible to improve soft-processor performance through replication [13], customization [23] or micro-architecture extensions [4], [24]. However, these approaches are not well suited for implementing control components of computation as they offer a coarse granularity for customization and scaling.

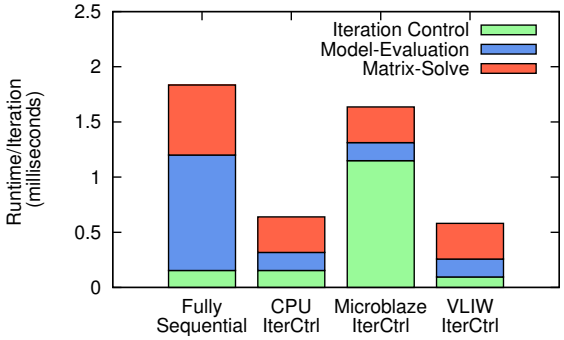

Fig. 1: SPICE Runtime Distribution for `s641` netlist (Virtex-6 LX760 Parallel Implementation)

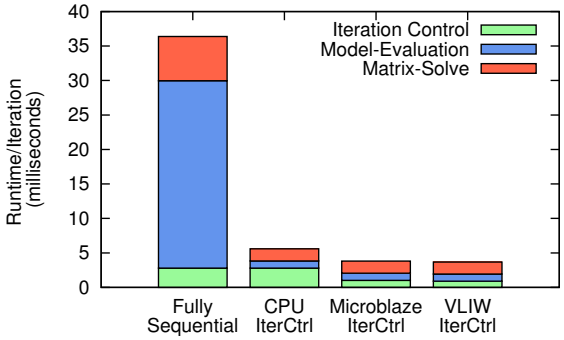

Fig. 2: SPICE Runtime Distribution for `r4k` netlist (Virtex-6 LX760 Parallel Implementation)

In this paper, we propose a unified spatial methodology based on VLIW-SCORE. We use the existing, high-level SCORE [3], [5] (Stream Computation Organized for Reconfigurable Execution) framework described in Section III and couple that to a custom, hybrid VLIW architecture described in Section III-C. This allows us to compose the complete accelerator system entirely on the parallel FPGA fabric without resorting to sequential offload or excessively burdening the programmer. We concretely demonstrate the benefit of this approach using the case-study of the SPICE Circuit Simulator. The SPICE algorithm, described later in Section IV, can be decomposed into compute-intensive Model-Evaluation and Matrix-Solve phases along with a small, sequential Iteration-Control phase. In our previous work, we have shown how to accelerate the compute-intensive components of SPICE using

978-1-4577-1740-6/11/$26.00 © 2011 IEEE

a custom VLIW architecture [11] for Model-Evaluation Phase and a dataflow architecture [12] for the Sparse Matrix-Solve phase.

To motivate the impact of implementation choice when mapping the Iteration-Control phase of SPICE on an FPGA, we show representative runtime breakdown for the `s641` SPICE netlist in Figure 1. In the "Sequential" column we observe that sequential runtime is dominated by the Model-Evaluation and Sparse Matrix-Solve computation while the Iteration Control is a small 7–8% fraction of total runtime. When we parallelize the Model-Evaluation and the Sparse Matrix-Solve phases on a Virtex-6 LX760, we notice that the Iteration Control is now a significant ≈25% portion of total runtime (see Column "CPU IterCtrl"). This suggests, that we must parallelize this phase to achieve high overall speedup. In Column "Microblaze IterCtrl", we observe that a Microblaze mapping is a poor implementation choice and results in lost speedup. In column "VLIW IterCtrl" we show the effect of implementing Iteration-Control phase on the a custom, hybrid VLIW FPGA fabric. We observe that the FPGA implementation offers the best overall performance for the complete simulator. When considering the additional `r4k` benchmark in Figure 2, we observe that the FPGA implementation is superior to the CPU-FPGA solution by 50%, emphasizing the merits of an application-customized, stand-alone accelerator. All our benchmarks exhibit characteristics between these two extremes shown in Figure 1 and Figure 2. This suggests we must consider stand-alone FPGA implementation methodologies to get the full potential speedup for the accelerator.

The key contributions of this paper include:

- Design and demonstration of a hybrid, custom VLIW architecture for implementing the control components of an FPGA-based SPICE accelerator.
- Development of a compilation flow for the VLIW architecture with a static VLIW scheduler.
- Development of a SCORE streaming library and code-generator for Xilinx Microblaze.
- Quantification of the speedups achieved by SPICE when considering: (1) CPU-FPGA partitioning, (2) Microblaze-based design and (3) a custom, hybrid VLIW mapping.

II. Related Work

We will now review some previous approaches for implementing control components of an FPGA accelerator. In CHIMPS [15], a C-to-FPGA compilation framework automatically extracts sequential traces from the computation and maps them to embedded soft-processors when offchip CPU transfer overheads are too large. However, for SPICE-like application requirements, the Microblaze mappings can lead to poor results as shown in Figure 1 and Figure 2. The RAMP Blue [13] project shows how to combine up to a thousand Microblaze soft-processors on a rack of FPGA boards to implement message-passing multi-processing fabrics. This granularity is unsuitable for implementing the control components if we desire balanced, overall speedups. And, this approach requires the programmer to manually partition the computation across multiple soft-processors. SPREE [23] shows how to improve performance of the soft-processor through low-level architectural modifications such as pipelining, multiplier designs, and novel shifters but delivers limited improvement (≈11% in some cases) that is insufficient for properly parallelizing SPICE. VESPA [24] and VEGAS [4] augment the NIOS soft-processors with configurable, flexible vector support to improve performance for data-parallel, regular computation. However, these architectures are unsuitable for SPICE since it has an insufficient auto-vectorization potential of 7% [8]. CUSTARD [6] provides efficient multi-threading support to a MIPS-based soft-processor and can deliver up to 5× improvements for media processing benchmarks with a 50% increase in area but only for integer-rich applications unlike resource-hungry, floating-point applications such as SPICE. Our SCORE-based approach provides a combination of automated, lightweight multi-threading as well as the customizability of the underlying hybrid VLIW datapath architecture to deliver scalable results for floating-point computation.

```
convergence(
param double reltol, param double abstol,
input double new, input double old,
output boolean cvg) {

        double max, diff, tol;

        state dfg(new, old):
                max=(new<old)? old:new;
                diff=new-old;
                tol=reltol*max+abstol;
                cvg = (tol<diff);
}
```

Listing 1: SPICE Convergence Detection in TDF

III. VLIW-SCORE Framework

As we saw in Figure 1 and Figure 2, the Iteration-Control phase only accounts for ≈7% of total sequential runtime. Our parallel implementation must take care to efficiently implement this portion to avoid an Amdahl's Law bottleneck. Simply mapping the sequential fraction of computation on the host CPU (offchip CPU) may introduce additional communication bottlenecks due to long round-trip latencies (*e.g.* PCIe latencies of ≈1–10μs). A C description of computation mapped to an embedded soft-processor can be equally limiting due to micro-architectural mismatch. Alternatively, we may choose to implement the computation in low-level VHDL or Verilog; however, they require careful cycle-level scheduling of irregular code for best results. Instead of relying on C or cycle-level RTL, we use a more suitable model for describing the control components – SCORE [3], [5]. SCORE is a streaming system architecture that allows natural expression of control-oriented, streaming computation and is well-suited for high-performance FPGA implementation. In general, it is a restricted subset of Hoare's Communicating Sequential Processes [9] (CSP) and an implementation of the Kahn model [10]. We now briefly review the SCORE framework.

```
convergence_compose(
param double reltol, param double abstol,
input double new, input double old,
output boolean cvg) {

        double cvg_stream;

        convergence(reltol, abstol, new, old,
                    cvg_stream);
        reduce_cvg(cvg_stream, cvg);
}
```

Listing 2: SPICE Convergence Composition in TDF

A. Description of Computation

A SCORE program consists of a graph of operators (compute) and segments (memory) linked to each other via streams (interconnect). Streams provide point-to-point communication and are realized as single-source, single-sink FIFO queues of unbounded length. SCORE operators are implemented as finite-state machine (FSM) operations that interact with the rest of the computation only through streams thereby preventing side-effects and decoupling the internal operation of each operator from each other. A state-machine description of a computation is a natural way to express the control-oriented, irregular operations in the SPICE Iteration-Control phase. The operations within a state can be described as a straight-line dataflow computation on stream inputs and local register values. SPICE convergence and timestep calculations in the Iteration Controller can be naturally expressed as data-parallel `reduce` and `map` functions which are amenable to simple SCORE operators with dataflow states. We also observe that each SCORE state will be evaluated using a deterministic function (logic AND) of the input stream states without peeking. This means that the operational semantics of the SCORE compute model are fully deterministic and independent of physical mapping substrate *i.e.* host CPU, embedded soft-processor, and spatial FPGA logic. We use the TDF [3] (Task Description Format) dataflow language for our expression. In Listing 1, we show a simplified, illustrative example from the SPICE Iteration-Control phase to show how one would describe computation in SCORE. Each SCORE operator declares its `input` and `output` streams. The `param` keyword allows the programmer to specify a late-bound constant in the code. Within the operator, computation is separated into `states` with optional sensitivity on input streams. Each stream can even carry `eofr` (end-of-frame) signaling to allow the operator to clear internal states if necessary and provide higher-level timing information to the processing. Multiple SCORE operators can be composed hierarchically as shown in Listing 2. This composition is a directed graph of operators connected through stream connections (later see Figure 6 for the SPICE Iteration-Control SCORE graph). We can test the correctness of the compiled SCORE code by using the multi-threaded C++ code-generation backend.

B. Parallelism

SCORE exposes different forms of parallelism within the computation at multiple levels of granularity. State transition operations and bit-level streaming operations can be customized to use spatial FPGA fabric (`fine-grained, bit-level parallelism`). Each SCORE state captures dataflow computation (`instruction-level parallelism, pipeline parallelism`) on stream inputs and local variables to generate stream outputs. Multiple SCORE operators can be processed in parallel (`thread-level parallelism`) if data is available in the stream inputs. For stateless SCORE operators it is possible to perform unrolling and replication to exploit data-level parallelism by exposing Instruction-Level Parallelism (ILP). For example, in Listing 1, we can unroll or replicate the stateless operator multiple times to improve performance.

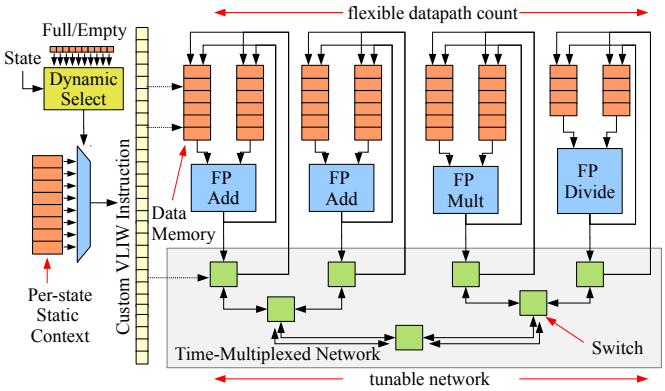

Fig. 3: Hybrid VLIW FPGA Architecture

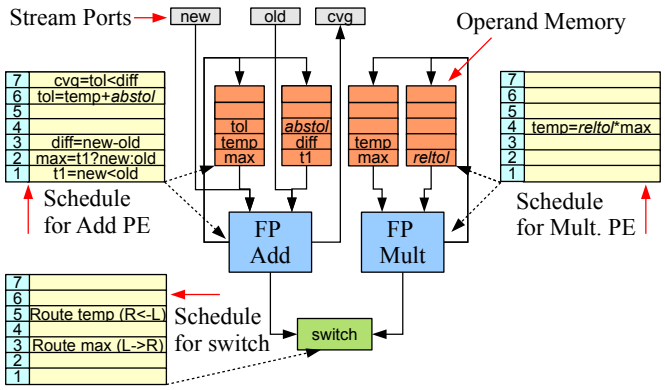

Fig. 4: Sample Schedule for Listing 1

C. Hybrid VLIW FPGA Architecture

Fully-spatial implementation of the Iteration Controller can result in unreasonable FPGA resource requirements. For sparse, irregular, control-oriented sequential computation, this spatial mapping will also result in mostly underutilized resources. If we are to implement this computation to maximize overall application performance on a single chip, we will have to *virtualize* (time share) the computation over limited FPGA capacity. To limit FPGA resource requirements while exploiting application-specific, customization potential of FPGAs, we develop a custom, hybrid VLIW FPGA architecture, shown in Figure 3. The resource-shared VLIW architecture

combines tight static scheduling with limited dynamic evaluation to fully implement the sequential computation. The VLIW architecture consists of a heterogeneous collection of floating-point datapaths coupled to high-bandwidth local memories and interconnected through a time-multiplexed communication network. A VLIW instruction for the architecture consists of read/write addresses for the input and output memories along with multiplexer select signals for the datapath. The time-multiplexed switch also contains configuration instructions that provide routing information to schedule communication between the input and output ports. We can efficiently store the VLIW configuration in on-chip memories (distributed RAMs).

SCORE Compatibility The SCORE framework [3] which was originally designed for rapidly-reconfigurable, page-based reconfigurable architectures like the HSRA [19] does not immediately match commercial FPGA capability. The Hybrid VLIW architecture shown in Figure 3 is a new implementation model for the SCORE framework. It is an effective backend target for TDF as it can simultaneously support both (1) streaming parallelism across SCORE operators and (2) dataflow parallelism within each SCORE state. The SCORE description allows a straightforward separation of computation into static and dynamic portions. The dataflow within a state can be considered static. For example, the code to compute `cvg` in state `dfg` of operator `convergence` in Listing 1 is a static dataflow expression. The data-dependent state transition and operator selection from multiple active SCORE operators over resource-shared VLIW hardware is considered dynamic. For example, we can fire state `dfg` in operator `convergence` of Listing 1 only when we dynamically detect data presence on both inputs. We compile static components to the custom VLIW datapaths and the dynamic components to the lightweight selection logic as shown in Figure 3.

VLIW-SCORE Compilation How do we compile this mix of static and dynamic SCORE processing to an FPGA effectively? We typically observe that we do not activate all SCORE operators uniformly. Certain portions of the graph are fired more often than others. Moreover, these graphs contain a diverse set of floating-point functions such as adds, multiplies, divides, square-roots. Not all functions are used equally either. Both of these observations help us engineer our custom compiler. We customize the number of VLIW engines and the mix of hardware functions in each VLIW engine in a manner that is proportional to the activation frequency of each SCORE operator as well as instruction distribution with each operator. For more details relevant to SPICE Iteration Control see Section V-B.

Static Mapping: We statically schedule the individual state expressions to ensure high utilization of VLIW resources. The static scheduler generates a VLIW configuration [7] for the execution that contains pre-computed datapath, memory and switch controls. For example, for the SCORE code shown in Listing 1, we show the custom datapath and a cycle-by-cycle static schedule for the operators and the switch in Figure 4. We can also map SCORE operators to separate VLIW engines interconnected by streams. For example, the `new` and `old` streams in Figure 4 connect to other VLIW engines.

Dynamic Mapping: We also compile dynamic state transition operations as well as SCORE operator selection to spatial FPGA hardware. We support lightweight spatial evaluation of state transition conditions as a function of stream full/empty signals. This dynamic logic then loads the pre-compiled static context for the active SCORE operator and state by simply computing an address offset for the context. In Figure 4, the static schedule is activated when the dynamic logic determines that the stream ports have valid data and space to write outputs.

D. Microblaze Architecture

We also map the computation to an embedded Microblaze soft-processor from the same high-level SCORE description. We develop a code-generator backend for SCORE to produce C code that is suitable for the embedded processor. The code-generator targets a stream library we developed for supporting stream-level operations on the Microblaze. The library transparently provides the same set of streaming interactions with other SCORE operators that are (1) running on the Microblaze, or (2) spatial datapaths connected through streaming FSL links. The Microblaze implementation interacts with the rest of the spatial FPGA fabric through FSL streams. The SCORE operators are implemented as light-weight PThreads managed by the embedded OS (Xilkernel [22]). Certain data-parallel interfacing computation is still implemented spatially connected to FSL links. We develop a lightweight runtime customized for the Microblaze to support operator and stream allocation along with buffer management.

IV. Application Case Study: SPICE

SPICE simulates the dynamic analog behavior of a circuit described by non-linear differential equations. SPICE circuit equations model the linear (*e.g.* resistors, capacitors, inductors) and non-linear (*e.g.* diodes, transistors) behavior of devices and the Kirchoff's Current Law at the different nodes and branches of the circuit. SPICE solves the non-linear differential circuit equations by computing small-signal linear operating-point approximations for the non-linear elements and discretizing continuous time behavior of time-varying elements until termination (① in Figure 5). 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 (② in Figure 5). It then solves for $\vec{x}$ using a sparse linear matrix solver in the **Matrix-Solve** phase (③ in Figure 5). The Model-Evaluation and Sparse Matrix-Solve phases dominant total SPICE runtime and have been accelerated using FPGAs previously in [11], [12].

The Iteration-Control phase manages two kinds of iterative loops: ① a loop for linearizing the non-linear elements of the circuit, and ② another loop for advancing the timestep of the

simulation. We show these loops in Figure 5. The Newton-Raphson algorithm is used to compute the linear operating-point for the non-linear devices like diodes and transistors with custom convergence conditions (ⓐ). Additionally, an adaptive time-stepping algorithm based on truncation error calculation (ⓑ) is used for handling the time-varying devices like capacitors and inductors. SPICE also supports a dynamic breakpoint processing logic for handling source transition timesteps in the voltage and current sources (ⓒ). The analysis state machines (ⓓ) implement the loop control algorithms for performing DC and transient analysis.

We now look at SPICE Iteration Control in greater detail. We show the high-level SCORE representation of the SPICE Iteration Controller in Figure 6. In the remainder of this section, we will explain the different computations within the SPICE Iteration-Control phase and attempt to understand performance characteristics that motivate a spatial FPGA mapping.

Convergence Condition (`converge` SCORE operator in Figure 6): The simulator declares convergence when two consecutive iterations generate solution vectors and non-linear approximations that are within a prescribed tolerance respectively (See Equation 1 and Equation 2). The closeness between the values in consecutive iterations is parametrized in terms of user-specified tolerance values: $reltol$ (relative tolerance), $abstol$ (absolute tolerance), and $vntol$ (voltage tolerance). In these equations, $\vec{V_i}$ or $\vec{I_i}$ represent the voltage or current unknowns in the i-th iteration of the Newton-Raphson loop. The convergence conditions compare the current solution vector in iteration (i) with the previous iteration ($i-1$). This is a purely data-parallel computation on the voltage and current vectors that can be trivially parallelized. We have previously shown simplified SCORE code for Equation 2 in Listing 1 and Listing 2.

$$|\vec{V_i} - \vec{V_{i-1}}| \leq reltol \cdot \max(|\vec{V_i}|, |\vec{V_{i-1}}|) + vntol \quad (1)$$
$$|\vec{I_i} - \vec{I_{i-1}}| \leq reltol \cdot \max(|\vec{I_i}|, |\vec{I_{i-1}}|) + abstol \quad (2)$$

Local Truncation Error (LTE) (`LTE` SCORE operator in Figure 6): Local Truncation Error is a local estimate of accuracy of the Trapezoidal approximation used for integration. The truncation-error-based time-stepping algorithm in `spice3f5` computes the next stepsize δ_{n+1} as a function of the LTE (ϵ) of the current iteration and a Trapezoidal divided-difference approximation (DD_3) of the charges (Q) at a few previous iterations (See Equation 3). For a target LTE, the Iteration Controller can match the stepsize to the rate of change of circuit quantities. If the circuit quantities are changing too rapidly, it can slow down the simulation by generating finer timesteps. This allows the simulator to properly resolve the rapidly changing circuit quantities. Alternately, if the circuit is quiescent (*e.g.* digital circuits between clock edges), the simulator can take larger timesteps for a faster simulation. The stepsize δ_{n+1} is added to the current timestep to advance the simulation as shown in Equation 4. The truncation error

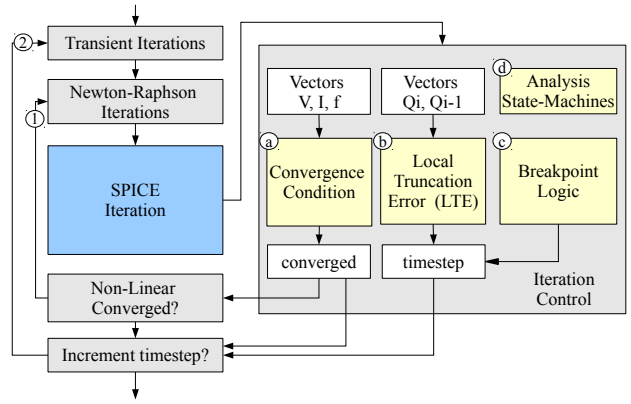

Fig. 5: Flowchart of a SPICE Simulator with emphasis on SPICE Analysis Control Algorithms

computation is also a data-parallel operation on the charge and current vectors across a few previous timesteps.

$$\delta_{n+1} = \sqrt{\frac{trtol \cdot \epsilon}{\max\left(\frac{|DD_3(Q)|}{12}, abstol\right)}} \quad (3)$$
$$t_{n+1} = t_n + \delta_{n+1} \quad (4)$$

Breakpoints (`breakpoint`, `accept` SCORE operators in Figure 6): The calculation of the timestep based on divided differences in Equation 3 assumes that the physical circuit quantities being approximated are continuously differentiable. However, when the source elements suddenly change value (*e.g.* Piece-Wise Linear sources), they introduce a discontinuity. SPICE stores these timepoints as breakpoints and forces a circuit evaluation at the breakpoint using a first-order backward-Euler integration. The breakpoint computation is specific to each circuit and data-dependent on the behavior of the circuit simulation. There is limited dataflow parallelism available when computing and updating breakpoints.

Analysis State Machines (`spicestmc`, `nistmc` SCORE operators in Figure 6): The loop control logic is managed by the SPICE analysis state machines. These state machines are responsible for organizing the simulation steps, handling error conditions, determining convergence and announcing termination. We separate these state machines into two parts: (1) a high-level controller `spicestmc` that manages the DC and transient analysis along with the timestepping algorithm (the outer-loop in Figure 5), and (2) the iteration controller `nistmc` that invokes Model-Evaluation and Sparse Matrix-Solve phases at the right time (the inner loop in Figure 5). The state machine evaluation sequence is specific to each circuit and data-dependent on the result of circuit behavior. This stage is difficult to parallelize and we extract limited dataflow parallelism (*i.e.* ILP) from state evaluations.

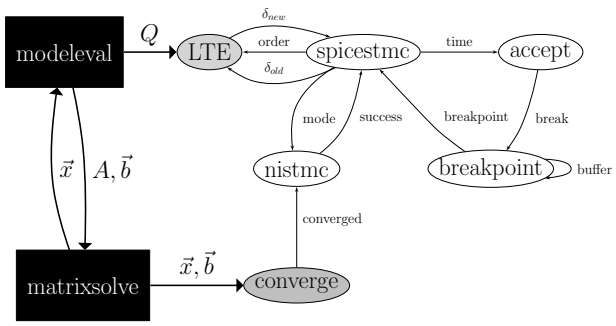

Fig. 6: High-Level SCORE Operator Graph for `spice3f5`

V. Methodology

We now describe our experimental methodology that enables comparing the different FPGA-based implementations of the SPICE Iteration-Control phase. We show a high-level representation of our experimental flow in Figure 7.

A. Verification Backend

To verify correctness of our implementation, we perform a functional SCORE simulation of the Iteration Control algorithms and compare our results with `spice3f5` as shown in Figure 7. We generate multi-threaded C++ code from the SCORE compiler to obtain a functionally-correct implementation of the SCORE description of Iteration Control. Then, we integrate the SCORE runtime into `spice3f5` to communicate relevant SPICE state to our SCORE implementation using Inter-Process Communication (IPC). We perform modular verification of individual SCORE operators by letting the `spice3f5` process handle the rest of the computation. We then empirically compare the sequence of visited states in the two processes to determine correct operation. A few cases involving floating-point rounding resulted in a mismatched state visit sequence but delivered the exact same timesteps and simulation results.

B. Mapping Flow

As discussed in Section III, we develop a new implementation model for SCORE based on resource sharing and static scheduling. We are now able to compile multiple SCORE states and even multiple SCORE operators onto the same set of VLIW datapaths if required. But how do we decide what is resource-shared? We choose an allocation of states and operators to custom, hybrid VLIW engines through a combination of compiler-driven and simulation-driven heuristic as shown in Figure 7.

To do this, we first count the number of state and operator activations corresponding to the SCORE operator graph of the Iteration Control computation from a `spice3f5` run. An activation corresponds to a state within that SCORE operator getting fired. This happens when the state transition condition is met *i.e.* stream inputs have valid data and stream outputs have sufficient spare capacity. In Table I, we show the dynamic activation counts for the different SCORE operators for a representative run. We observe that the `LTE` and `Converge` calculation dominate the activation counts. The activity measurement also allows us to determine the runtime of the two FPGA implementations by multiplying the state activations with the cycle count per state as shown in Figure 8.

TABLE I: SCORE Operator Activation Frequency for a simple Resistor-Capacitor-Diode circuit

Operator	Total Activations/ Iteration	Percent of Total
`converge`	1088465	64.394
`LTE`	601076	35.560
`accept`	299	0.017
`breakpoint`	48	0.002
`nistmc`	152	0.009
`spicestmc`	262	0.015

$$T = T_{clk} \cdot \left(\sum_{i \in operator} \left(\sum_{n \in state} active(i,n) * cycles(i,n) \right) \right)$$

Fig. 8: FPGA Cycles for Iteration-Control

Next, we measure the number of floating-point instructions and their types in the different SCORE operators as shown in Table II. These statistics are obtained from the optimized operation graphs generated by `tdfc`, the SCORE compiler. We observe that **If-Mux** and **Rest** instructions dominate total count but can be implemented cheaply on VLIW fabric. The floating-point **Add** is the next most frequent instruction which influences our VLIW datapath mix. We also note that we need only one **SQRT** floating-point operation and no other expensive floating-point functions (*e.g.* logarithm, exponential).

As shown in Figure 7, we first choose the number of VLIW engines and then choose a datapath-mix within each engine. We assign `LTE` and `converge` operators each to their own VLIW engines (12 datapaths each) scheduled using software-pipelining to exploit data-parallelism. We combine the rest of the Iteration Control computation into a single VLIW engine (4 datapaths) compiled using straight-line dataflow scheduling. We tabulate resource costs in Table IV.

C. Hybrid, Custom VLIW Backend

For the statically-scheduled implementation, we obtain the cycle counts from the scheduler for each state of every SCORE operator. We implement the data-parallel computation in `LTE` and `Converge` operators using the software-pipelined scheduler [11] with an unroll factor of 10. We implement the sequential state-machine logic in `nistmc` and `spicestmc` along with the `breakpoint` operators using a simple Dataflow scheduler (without any unrolling). We combine these two schedules to assemble the spatial implementation of the It-

TABLE II: Compiled Instruction Counts

Operator	Add	Mult.	Div	Sqrt	If	Rest	Total
`converge`	7	1	0	0	6	6	20
`LTE`	16	8	9	1	21	20	75
`accept`	81	2	1	0	56	80	220
`breakpoint`	95	2	1	0	110	122	330
`nistmc`	2	0	0	0	8	14	24
`spicestmc`	29	15	6	0	79	83	212
Total	230	28	17	1	280	327	733

Column Rest includes compare, bool, floor, ceiling, and other special func.

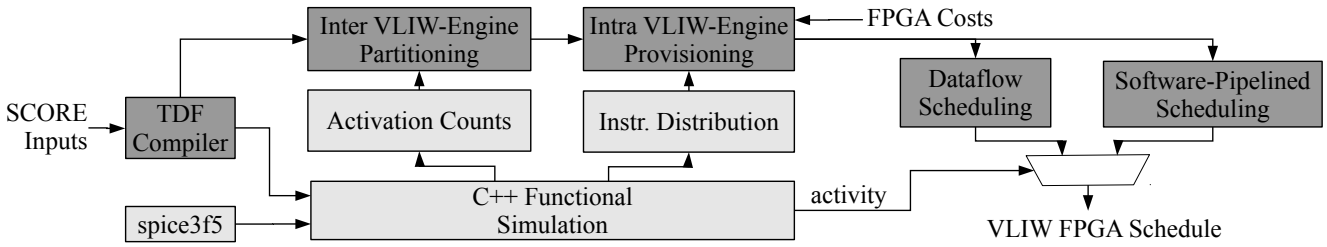

Fig. 7: VLIW Mapping and Verification Flow

TABLE III: Scheduled and Measured cycle counts (sum of all SCORE states)

Operator	Target	Scheduler	Cycles
converge	VLIW	Graphstep	16
LTE	VLIW	Graphstep	47
accept	VLIW	Dataflow	400
breakpoint	VLIW	Dataflow	422
nistmc	VLIW	Dataflow	124
spicestmc	VLIW	Dataflow	1406
accept	Microblaze	-	17271
breakpoint	Microblaze	-	15764
nistmc	Microblaze	-	5696
spicestmc	Microblaze	-	58555

eration Controller as represented in Figure 7. Presently, our statically-scheduled implementation operates at 200 MHz. We may be able to improve the frequency of our implementation but are currently limited by the frequency of the Xilinx Coregen double-precision floating-point divider. We use the scheduled cycle counts from our VLIW compiler to compute the total time required (see Table III).

D. Microblaze Backend

A Microblaze implementation of the data-parallel, floating-point intensive computation in LTE and Converge blocks will result in extremely poor performance that is substantially worse than what we present here. This is primarily due to poor double-precision floating-point support (10–100 cycles/-operation). Hence, we do not consider that implementation for our comparisons and implement these two data-parallel computations as VLIW engines. The Microblaze implementation only maps the SPICE analysis state-machine logic as illustrated in Figure 7. The Microblaze soft-processor along with supporting logic is designed to operate at 100 MHz while consuming 3734 slices of area [20]. The Microblaze uses low-latency onchip insruction memories connected over the PLB bus. Streaming data connections between the Microblaze processors is implemented using FSL links. We tabulate the cost model for these two designs in Table IV. We measure the number of Microblaze clock cycles to implement each state of every SCORE operator using a hardware counter (see Table III).

Finally, we compare both of these implementations with spice3f5 running on a 2.67 GHz Intel Core i7 965 under two scenarios. We first consider the Iteration-Control phase

TABLE IV: FPGA Resource Usage (Virtex-6 LX760)

SCORE Operators	VLIW Datapaths	Area (Slices)	Memory (BRAMs)
LTE (1)	12	10917	28
converge (2)	12	10389	9
breakpoint, nistmc, accept, spicestmc (3)	4	3644	6
Microblaze+Peripherals (4)		1504	16
Custom, Hybrid VLIW Total (1+2+3)		24950	43
Microblaze Total (1+2+4)		22810	53

running on the host CPU while the Model-Evaluation and Matrix-Solve phases are implemented on an FPGA PCI board. In this model, we count the cost of PCI transfer in our sequential runtime. To compute overall application speedups, we compare fully sequential implementation of spice3f5 with the FPGA implementation of the complete simulator. We measure runtime using the PAPI 4.0 [14] performance counters on a 64-bit Linux workstation running Ubuntu Lucid Lynx 10.04. We perform experiments with the SPICE simulator across circuits collected from Simucad [16] (RAM netlists), University of Michigan [17] (Clocktrees), UBC [18] (Wave-Pipelined Circuits) and IBM [2] (ISCAS 98 benchmarks).

VI. Evaluation

In this section, we characterize the speedup of the SCORE implementation of SPICE. Our studies show how we could avoid limiting overall speedup due to potential Amdahl's Law bottlenecks. We first show complete application-level impact of the different FPGA mappings while considering full-system partitioning to deliver a balanced overall design. We then explain the speedup behavior of Iteration-Control.

A. Application-Level Impact

We will first evaluate the impact of mapping the sequential fraction of SPICE under the three implementation targets on overall SPICE performance (1) host, offchip CPU (2) Microblaze and (3) Hybrid VLIW. We show speedups equations for these three implementation configurations in Figure 9. The composite SPICE FPGA accelerator is projected to use ≈90% of the total FPGA resources of the Virtex-6 LX760 with the Iteration-Control phase accounting for ≈17% of total resources. Remember, the Iteration-Control phase accounted for about 7% of sequential runtime but now accounts for a disproportionate 17% of area. This suggests that we pay a

$$Speedup_{hybrid_vliw} = \frac{T_{seq}(LTE + convergence + breakpoint + nistmc + spicestmc)}{T_{hybrid_vliw}(LTE + convergence + breakpoint + nistmc + spicestmc)} \quad (5)$$

$$Speedup_{microblaze} = \frac{T_{seq}(LTE + convergence + breakpoint + nistmc + spicestmc)}{T_{hybrid_vliw}(LTE + convergence) + T_{microblaze}(breakpoint + nistmc + spicestmc)} \quad (6)$$

$$Speedup_{cpu-fpga} = \frac{T_{seq}(LTE + convergence + breakpoint + nistmc + spicestmc)}{T_{hybrid_vliw}(LTE + convergence) + T_{seq}(breakpoint + nistmc + spicestmc)} \quad (7)$$

$T_{seq}(x) =$ Sequential CPU time, $T_{hybrid_vliw}(x) =$ Parallel VLIW time, $T_{microblaze}(x) =$ Sequential Microblaze time

Fig. 9: Speedup Calculation Equations for SPICE Iteration-Control Phase

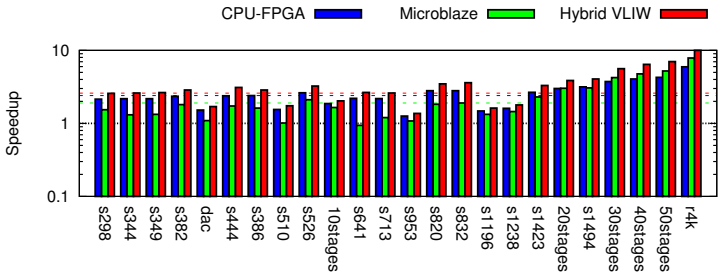

Fig. 10: Complete SPICE Application Speedups (Virtex-6 LX760 vs. Intel Core i7 965)

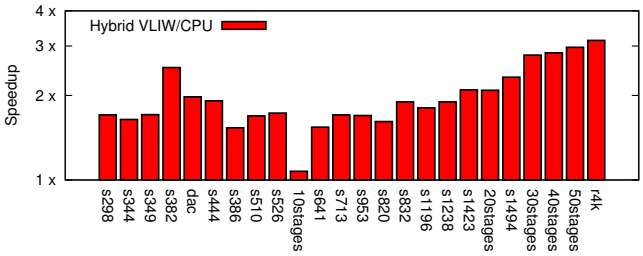

Fig. 11: Hybrid VLIW Speedups (12-datapath Virtex-6 LX760 vs. Intel Core i7 965)

higher area cost to accelerate the hard-to-parallelize sequential component of SPICE to avoid Amdahl's Law bottleneck.

B. Overall SPICE Speedups

In Figure 10 we summarize the speedups achieved for the complete SPICE simulator under the same three scenarios as before. The custom, hybrid VLIW implementation provides the highest overall mean speedups of 2.6×(1.3×–11.1×). The Microblaze implementation is outperformed by even the Sequential implementation of Iteration Control. We are able to achieve a mere 1.9× mean speedups (0.94×–7.8×) across our benchmark set if we implement the Iteration Controller on the Microblaze. This approach only delivers this speedup for very large circuits where the Iteration Control phase is a tiny fraction of total runtime. For the circuits in our benchmark set, even the naïve CPU-FPGA implementation of Iteration-Control phase deliver 2.4×(mean) speedup.

C. Iteration-Control Performance

Let us now understand the performance trends within the Iteration-Control phase for the Microblaze and VLIW mappings. We first compare the CPU and FPGA implementations of the Iteration-Control phase of SPICE as a stand-alone computation. In Figure 11, we plot the speedup achieved by our hybrid FPGA architecture over the sequential implementation on an Intel Core i7 965. We are able to accelerate this phase of SPICE by 1.07–3.3× across the benchmark set (mean of 2.12×). We deliver the higher speedups of around 3.3× for the larger circuit sizes. This is because the data-parallel evaluation of the `convergence` and `LTE` equations increasingly dominate at large circuit sizes.

Next, we consider a Microblaze implementation of the `state-machine` and `breakpoint`-processing logic. In this arrangement, the `LTE` and `Converge` operations continue to be implemented over custom, hybrid VLIW hardware. We compute speedup for this phase using the formula represented in Figure 9. The performance of this lightweight implementation is shown in Figure 12. Unfortunately, this implementation actually slows down the computation by as much as 30× for small circuits while delivering speedups of 2.9× for the larger benchmarks. In contrast, we can achieve ≈30% higher speedups (up to 3.3×) for all circuit sizes for the custom, hybrid VLIW FPGA architecture as shown earlier in Figure 11. The reasons for this slowdown include (1) lower clock frequency of the processor, (2) sequential nature of the processor architecture and (3) poor double-precision floating-point support (10s-100s of cycles/function).

Finally, in Figure 13, we show performance scaling with FPGA area of the complete Iteration Controller averaged across multiple benchmarks. As shown, in our current design configuration, we only allocate a small fraction (≈ 15%) of the entire Virtex-6 LX760 to this phase while devoting the rest of the of the area to Model-Evaluation and Sparse Matrix-Solve phases for maximum overall SPICE acceleration. With increasing FPGA capacity, our VLIW design can scale gracefully beyond the current configuration. This suggests that

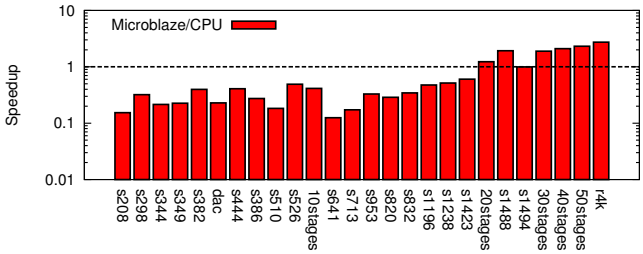

Fig. 12: Microblaze Speedups
(Virtex-6 LX760 FPGA vs. Intel Core i7 965

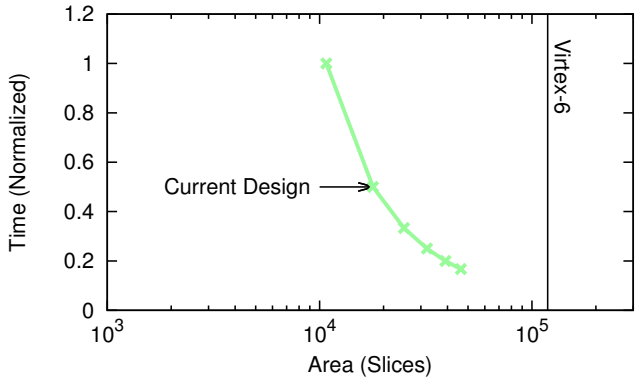

Fig. 13: Performance Scaling

there is an additional 2–3× worth of parallelism left in the Iteration Control phase that can be exploited with additional area.

VII. Conclusions

Applications like SPICE often contain control components (Iteration-Control phase) that make it challenging to fully parallelize the application on an FPGA accelerator. If we map these control components to embedded soft-processors like an off-the-shelf Xilinx Microblaze, our overall speedups are limited to a mean of 1.9× (max 7.8×). In contrast, if we use the SCORE framework to map the control components to a custom hybrid VLIW architecture, we can deliver higher mean speedups of 2.6× (max 11.1×). We are able to deliver these speedups through higher-level, parallel expression in SCORE that enables exploiting a combination of dataflow and coarse-grained parallelism in the computation and implementing this parallelism efficiently on a hybrid, VLIW substrate. In future work, we will examine the opportunity to generate stand-alone FPGA designs for a range of other applications.

VIII. Download

The source code for the SCORE framework is publicly available at: http://www.github.com/nachiket. We encourage the community to download, use and contribute to these tools. We will continue to add language refinements, extensions, and new backends into the public repository. We would also like to acknowledge the help of Eylon Caspi in our efforts.

References

[1] Altera. Nios II Processor Reference Handbook, 2010.

[2] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. *IEEE International Symposium on Circuits and Systems*, 3(May 1989):1929–1934, 1989.

[3] E. Caspi. *Design Automation for Streaming Systems*. Phd, University of California, Berkeley, 2005.

[4] C. Chou, A. Severance, A. Brant, Z. Liu, S. Sant, and G. Lemieux. VEGAS: soft vector processor with scratchpad memory. In *International Symposium on Field Programmable Gate Arrays*, pages 15–24. ACM, 2011.

[5] A. DeHon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems*, 30(6):334–354, Sept. 2006.

[6] R. Dimond, O. Mencer, and W. Luk. CUSTARD - A customizable threaded FPGA soft processor and tools. In *International Conference on Field Programmable Logic and Applications*, pages 1–6. IEEE, 2005.

[7] J. Ellis. *Bulldog: A compiler for VLIW architectures*. MIT Press, 1986.

[8] J. Hennesey and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kauffman, 2nd edition, 1996.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[10] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP CONGRESS 74*, pages 471–475. North-Holland Publishing Company, 1974.

[11] N. Kapre and A. DeHon. Accelerating SPICE Model-Evaluation using FPGAs. In *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 37–44. IEEE, 2009.

[12] N. Kapre and A. DeHon. Parallelizing Sparse Matrix Solve for SPICE Circuit Simulation using FPGAs. In *International Conference on Field-Programmable Technology*, pages 190–198, 2009.

[13] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P. Droz. RAMP Blue: A message-passing manycore system in FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 54–61. IEEE, Aug. 2007.

[14] P. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. *Proceedings of the Department of Defense High Performance Computing Modernization Program Users Group Conference*, pages 7–10, 1999.

[15] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig. Performance and power of cache-based reconfigurable computing. In *Proceedings of the International Symposium on Computer Architecture*, volume 37, page 395. ACM, June 2009.

[16] Simucad/Silvaco. BSIM3, BSIM4 and PSP benchmarks, 2007.

[17] C. Sze, P. Restle, G. Nam, and C. Alpert. ISPD2009 clock network synthesis contest. In *Proceedings of the International Symposium on Physical design*, page 149. ACM Press, 2009.

[18] P. Teehan, G. Lemieux, and M. Greenstreet. Towards reliable 5Gbps wave-pipelined and 3Gbps surfing interconnect in 65nm FPGAs. In *Proceeding of the International Symposium on Field Programmable Gate Arrays*, pages 43–52. ACM, 2009.

[19] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. HSRA: high-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the 1999 ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*, pages 125–134. ACM, 1999.

[20] Xilinx. Xilinx CoreGen Reference Guide, 2000.

[21] Xilinx. MicroBlaze Processor Reference Guide, 2010.

[22] Xilinx. OS and Libraries Document Collection, 2010.

[23] P. Yiannacouras and J. Rose. The microarchitecture of FPGA-based soft processors. *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 202, 2005.

[24] P. Yiannacouras and J. Steffan. VESPA: Portable, scalable, and flexible fpga-based vector processors. *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, 2008.