J Sign Process Syst
DOI 10.1007/s11265-010-0562-x

Rapid Synthesis and Simulation of Computational Circuits in an MPPA

David Grant · Graeme Smecher · Guy G. F. Lemieux · Rosemary Francis

Received: 13 February 2010 / Revised: 31 July 2010 / Accepted: 9 November 2010
© Springer Science+Business Media, LLC 2010

Abstract A *computational circuit* is custom-designed hardware which promises to offer maximum speedup of computationally intensive software algorithms. However, the practical needs to manage development cost and many low-level physical design details erodes much of the potential speedup by distracting attention away from high-level architectural design. Instead, designers need an inexpensive, processor-like platform where computational circuits can be rapidly synthesized and simulated. This enables rapid architectural evolution and mitigates the risk of producing custom hardware. In this paper we present a tool flow (RVETool) for compiling computational circuits into a massively parallel processor array (MPPA). We demonstrate the CAD runtime is on average 70× faster than FPGA tools, with a circuit speed 5.8× slower than FPGA devices. Unlike the fixed logic capacity of FPGAs, RVETool can trade area for simulation performance by targeting a wide range in the number of processor cores. We also demonstrate tool scalability to very large circuits, synthesizing, placing, and routing a ≈1.6 million gate random circuit in 54 min.

D. Grant (✉) · G. Smecher · G. G. F. Lemieux
University of British Columbia, 2332 Main Mall,
Vancouver, BC Canada, V6T 1Z4
e-mail: daviddg@ece.ubc.ca

G. Smecher
e-mail: graeme.smecher@mail.mcgill.ca

G. G. F. Lemieux
e-mail: lemieux@ece.ubc.ca

R. Francis
University of Cambridge, 15 JJ Thomson Avenue,
Cambridge CB3 0FD, UK
e-mail: Rosemary.Francis@cl.cam.ac.uk

Keywords Circuit CAD · Field programmable gate arrays · Logic CAD · Software architecture · Software tools · FPGA-based design · Spatial computing

1 Introduction

To realize performance gains in many computationally intensive software algorithms, designers are implementing them in hardware as *computational circuits*. The end goal is often an ASIC or FPGA implementation to achieve the highest possible performance. This is being done for a wide range of algorithms, including molecular dynamics [1], fluid dynamics [2], video processing [3], financial modeling [4], ray tracing [5], and nuclear simulation [6]. Computational circuits are word-oriented and are often very large, requiring millions of gates.

Creating a computational circuit can be challenging and slow. A designer must repeatedly *synthesize* and *simulate* the circuit while debugging, improving, and verifying the design. For ASIC implementations, a correct design is important for avoiding costly re-spins. Many such circuits are also modelled in C or Matlab to ensure algorithmic correctness before the HDL is even attempted, further increasing design time. As circuit size increases, it takes longer to synthesize and simulate, reducing designer productivity, increasing time-to-market, and worsening the risk of missing a costly bug.

Having a fast Verilog synthesis and simulation flow reduces the need for a C or Matlab implementation in the design flow, further saving time. Instead, algorithmic correctness can be demonstrated with behavioural

Published online: 15 December 2010

Verilog, laying the groundwork for the final RTL implementation in Verilog.

Current solutions tend to offer either fast synthesis speed or fast simulation speed. Very high synthesis speeds (on the order of seconds) are achieved with compiled-code tools like Synopsis VCS by translating HDL into compiled C. Although these tools offer best-in-class simulation speed, simulating a hardware design on a high-performance processor still yields emulation rates of 1 MHz or lower. Parallel simulation may help, but simulation is still dominated by communication costs: one of the highest speedups reported is 13× for 32 processors [7]. Slow simulation speeds are a major obstacle for using these tools to design computational circuits.

In contrast, fast simulation speeds of 100 MHz+ can be obtained with FPGA devices. However, the synthesis time to map HDL to an FPGA can take hours or even days. Furthermore, if the HDL does not fit in the target FPGA, designers must resort back to simulation, buy a bigger device (if one exists), or partition the circuit into multiple FPGAs for testing. Slow synthesis and a strict capacity limit are major obstacles for using FPGAs to design computational circuits.

To address both the synthesis and simulation speed problems, we propose to compile Verilog to run on a massively parallel processor array (MPPA). Our tool flow, RVETool (Rapid Verilog Execution Tool), quickly synthesizes word-oriented computational circuits for RVEArch, our MPPA optimized for Verilog execution. RVETool can also target other MPPAs, such as Ambric Am2045 by Nethra Imaging Inc., at the expense of simulation speed.

The RVEArch architecture, first presented in [8], is an array of processors that uses high-speed pipelined interconnect and time-multiplexing to achieve a soft capacity limit, where capacity can be traded for performance. The key to accelerating simulation is more than just using additional processors; a low-latency, high-bandwidth NoC with fast core-to-core messages is necessary to overcome the communication bottlenecks in traditional parallel simulators. The pipelined and deterministically scheduled interconnect in RVEArch enables it to be even more efficient at emulating a circuit.

The objective of the toolflow, which is the topic of this paper, is to map a circuit onto RVEArch quickly and efficiently, resulting in high-speed emulation on the architecture. RVETool accepts behavioural Verilog and converts it to a high-level RTL that operates on words. To leverage the coarseness of the underlying architecture and improve synthesis speed, it does not break down all operations to a gate-level/bit-level netlist. We demonstrate that the tool can trade implementation area for speed in a coarse-grained MPPA, a tradeoff first demonstrated in fine-grained FPGAs with VEGA [9] and later with TSFPGA [10]. TSFPGA also added a modulo scheduling refinement, which we do not yet implement. In this paper, RVETool automatically compiles circuits to use between 1 and 1,024 processors. This soft capacity limit is essential for enabling the design of large, complex computational circuits.

This paper makes the following contributions:

1. It introduces a tool flow to quickly synthesize Verilog for an MPPA architecture.
2. It shows the platform (tools+architecture) can achieve fast synthesis (70× faster than FPGA CAD tools) and fast simulation (5.8× slower than an FPGA).
3. It demonstrates the tools can automatically scale a circuit to trade area for performance.
4. It shows the tool can synthesize very large circuits (≈1.6 million gates) in a reasonable amount of time (54 min).
5. It shows the architecture requires a reasonable amount of memory, approximately 16kB for each PE (data and instruction memory), for emulating circuits.

The first version of RVETool was presented in [11]. In this paper, we report improved CAD runtimes, improved $f_{\max}$ results with criticality-aware scheduling and a tail-to-head optimization, resource usage and a breakdown of longest-path delays. We also show that tool runtime scales well when processing very large circuits.

Section 2 presents related work on accelerating circuit simulation. Section 3 provides a brief overview of our execution model and architecture, and Section 4 details the tool flow for the architecture. We present the results of several experiments in Section 5, and conclude in Section 6.

2 Related Work

There are several general techniques for accelerating circuit simulation: compiled-code simulators, parallel simulators, and hardware-based accelerators.

Compiled-code simulators translate a circuit into a fixed program for native execution on a modern CPU (e.g., Pentium IV 3 GHz). Compilation is very fast compared to traditional FPGA CAD flows because

no placement or routing is required. Compilation can remain at a high level (rather than gate level) for additional speed. Execution can also be combined with event-driven simulation [12] to avoid updating parts of a circuit that do not change. However, the resulting program is single-threaded so the final simulation speed is still slow (on the order of 1 MHz). Verilator [13], VTOC [14], VBS [15], Symphony's VHDL Simili [16], Synposys VCS, and Cadence NC-Verilog are examples of compiled-code simulators.

Parallel simulators are concerned with the same fundamental task: simulating a circuit concurrently on multiple processors for maximum speedup. Parallel gate-level simulation is well researched [17–21] but speedups are usually less than 10 due to high interprocessor communication costs. PVSim [22] is a combined compiled-code and parallel simulator for up to 8 CPUs implemented with MPI. Compared to FPGAs, parallel simulators are up to six orders of magnitude slower [23, 24], making them prohibitive for verification of large systems. For parallel simulation with thousands of processors, a low-latency, high-bandwidth communication network is required.

Hardware based accelerators, like Cadence's Palladium [24] and Mentor Graphics VStation Pro [25], use processors or FPGAs for acceleration. These systems require a complete gate-level synthesis and emulate a design at roughly 2 MHz. Although this is slow compared to an FPGA, they can handle very large circuits. These simulators are also large and expensive: ranging in size from a mini-tower computer to a rack, they cost 0.4 to 10 million dollars [26]. Slow synthesis and prohibitive cost are significant barriers to using these devices.

Coarse-grain reconfigurable arrays (CGRAs) and massively parallel processor arrays (MPPAs) are potentially well-suited for speeding up simulation of computational circuits. As a result, our architecture bares great resemblance to these architectures. There exists a wide range of CGRAs (e.g. ADRES [27], PipeRench [28], MATRIX [29], Tartan [30], RaPiD [31], SCORE [32]) and MPPAs (e.g., Ambric [33], Tilera (based on RAW) [34]).

However, in contrast to these existing architectures, the RVEArch approach offers several improvements for increased efficiency of simulating computational circuits: no resources are used to implement a C or C-like programming model (e.g., no branch instructions or global memory), efficient implementation of logic gates like multiplexers (in the ALU) and bit-level signals (in a PLA, not yet supported), and concurrent execution of routing and processing tasks.

GPUs are often considered for high-performance computing, but current GPUs are designed for SIMD operations and have a large global memory optimized for coherent (structured) memory accesses. For circuit simulation, each core must execute a different program, so a true MIMD architecture is required. Poor core-to-core communications and high latencies from using unstructured data accesses throttle any potential speedups from a GPU [35].

3 Execution Model and Architecture

Computational circuits will be implemented in Verilog, or translated into Verilog from a high-level language like C. Rapid simulation requires a highly parallel, word-oriented platform with very low latency network-on-chip interconnect. The fast interconnect is critical because it must compete with the communication speed of the bare wires in the circuit it is emulating.

Our approach to implementing a computational circuit can be applied to almost any MPPA which supports processor-to-processor messaging, such as Ambric's. We view each processing element (PE) as containing a router and a core. In RVEArch they are separate hardware entities; in our Ambric implementation a single SRD processor implements both in software. Both are time-multiplexed and both follow a pre-programmed static schedule. The router is a 5×5 crossbar with registered outputs. Long-distance (pipelined) communication pathways are created by routing data through several PEs.

Using this time-multiplexed approach, the user clock is different than the system clock. Each PE router and core contain a schedule with exactly n instructions (the schedule length) to be executed in an infinite loop to implement the overall circuit. One pass of this schedule is equivalent to one user clock cycle, so if we assume RVEArch has a 1 GHz system clock, the user clock frequency would be $\frac{1}{n} \cdot 1$ GHz on RVEArch.

These pre-determined schedules mean the entire architecture is deterministic. This is similar to the GraphStep execution model [36]. It is the responsibility of the tools to schedule the code for each processor and for each router so that data is always in the correct place at the correct time. Non-deterministic delays, such as waiting for input data from an external device, must be handled at the user-circuit level. We are currently assuming the circuit uses a single clock domain. We believe this greatly simplifies the design of computational circuits, which is also the objective of this platform.

Multiple clock domains remains an issue for future work.

3.1 Idealized Ambric

We have used this execution model to target a slightly modified "idealized" Ambric architecture. We assume a 1 GHz clock instead of the 300 MHz clock used in the Am2045, and assume a single-cycle communication delay between neighbouring Ambric SRD processors.

We also assume that every instruction executes in a single cycle, that all RVE instructions are available on Ambric, and that sufficient memory exists to hold the entire schedule. These assumptions permit a fair comparison with RVEARch, where relatively simple changes have been made to make idealized Ambric more competitive. However, RVEArch and idealized Ambric still differ in their interconnect design, as modifying the Ambric interconnect to mimic the RVEArch interconnect would be a fundamental change to its design.

A single SRD processor implements both the router and core of our execution model. It uses lookup tables to determine which neighbouring SRDs to write to, then read from, then which core instruction to execute to complete a system clock cycle. Ambric uses blocking communication channels, requiring that all reads and writes are matched; our deterministic schedule gives us exactly this, so there is no risk of deadlocking the architecture.

In Section 5 we have used the output of RVETool to estimate the native-compiled code schedule length of each test circuit on this idealized architecture. The schedule length is calculated by summing the maximum number of reads and writes required across all PEs in each timeslot, and adding one to each total for the actual instruction execution.

3.2 RVEArch

RVEArch, shown in Fig. 1, distinguishes itself from other MPPAs in several ways:

- It uses a low-latency, high-bandwidth interconnect to connect only neighbouring PEs.
- It is intended to scale up to 100 × 100 PEs on a single chip by using a skew-tolerant clock distribution network.
- It contains a dedicated router.
- It contains a PLA to implement bit-level operations.

RVEArch uses a global clock, but PEs communicate only over short distances. Thus, while local skew

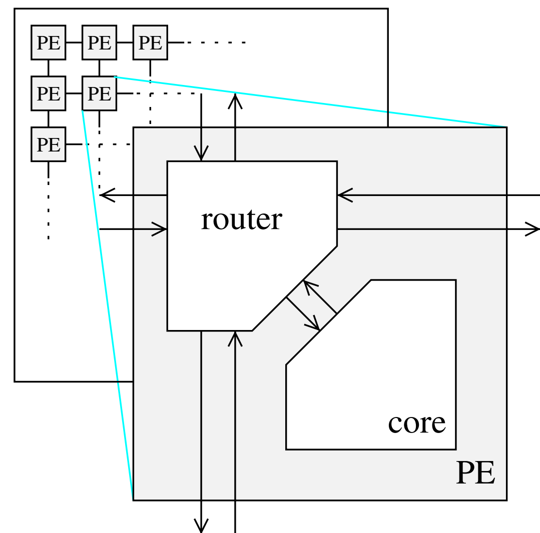

Figure 1 Architecture overview.

between all neighbouring PEs needs to be small and bounded, global skew requirements are somewhat relaxed and permit the use of a lower-energy clock distribution network. Since all PEs use the same clock source, the neighbours will still operate in synchrony. With this design, the architecture is readily scalable to high clock frequencies. Results presented in this paper assume a 1 GHz clock is realizable in 65 nm.

The RVEArch processor core is shown in Fig. 2. It is a simple processor with time-multiplexed ALU and data memory D to implement user-level circuit behaviour. There are no branch instructions, but conditional moves and multiplex (select) functions are supported. Node memory R is used to temporarily store local ALU results—emulating a wire for data used in the same user clock cycle and a flip-flop for data needed in the

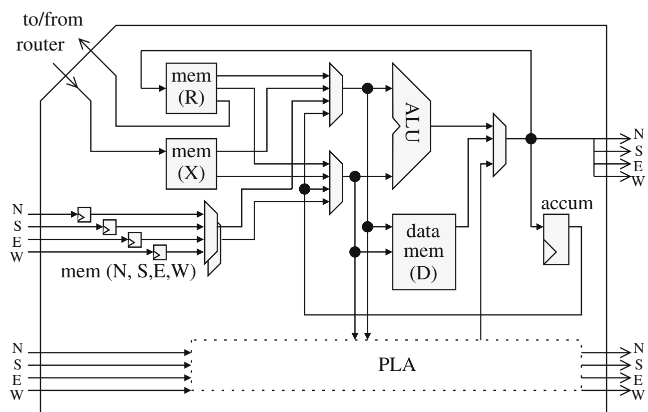

Figure 2 RVEArch PE core.

next user cycle. Likewise, PE memory X is used by the router to store data destined for this core from external PEs. There are also four buffered, direct links to adjacent PE cores (memories N, S, E, and W) which are used as a lower-latency alternative to the router for neighbour-only connections. These links are enabled in RVEArch for all results presented in this paper. The six PE memories (R, X, N, S, E, and W) represent wires or flip-flops in the user circuit, but from the RVEArch perspective these are implemented as register files of a traditional processor. These register files have separate read ports and write ports which require both an address (register number) and data. RVETool computes the register numbers automatically.

The dedicated router in each PE, shown in Fig. 3, allows all five router outputs to be assigned in a single cycle. The control is an instruction memory and decoder to set multiplexer select and register/write enable lines for the router in each cycle. One control word is read per time slot and it always advances to the next word (no jumping or branching). At the end of the schedule it restarts at offset zero. Similarly, the PE core in Fig. 2 also has an instruction memory and decoder that functions the same way, but it is not shown. We will show the separate router has a 3.8× performance advantage over the Ambric implementation where communication must be done serially and inline with the PE core computation. It is a key feature to keep communication costs low.

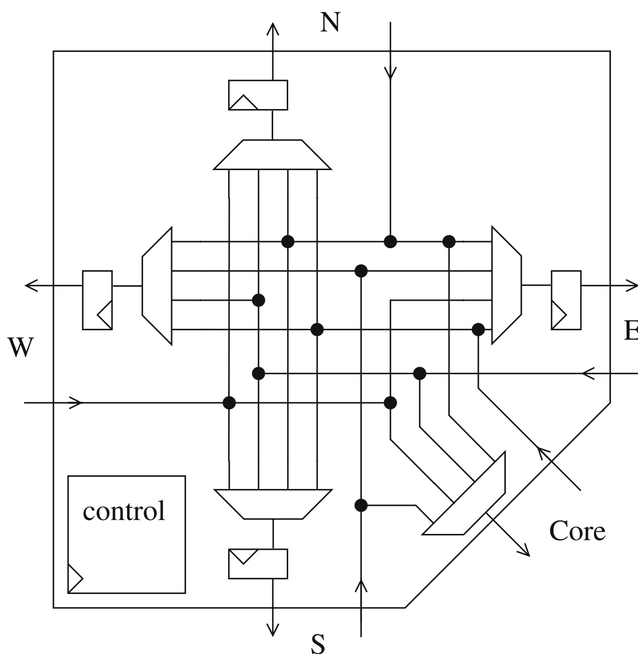

Figure 3 RVEArch PE router.

All buses are 32 bits wide, the data memory is fixed at 8kB (2,048 × 32-bit), and the instruction memory must be at least 8kB (1,000 × 59-bit, 40 bits for the PE core instruction plus 19 bits for the router control) to synthesize all benchmarks at 10× FPGA density. For all results presented in this paper, we allow RVETool to increase the number of entries in the PE memories as necessary, instead of enforcing usage limits. But based on the results, for our benchmarks over a wide range of architecture sizes, we found that 16 entries for the X and R memories, and 4 entries for the remaining memories is sufficient. We also assume all ALU operations including multiply are single-cycle.

Computational circuits may also contain single-bit (e.g. control) signals that map poorly to word-oriented processors. The PE core in Fig. 2 shows a PLA that is not time-multiplexed, which generates these signals. We are investigating the implementation of the PLA, so it has been omitted from all results in this paper. Instead, the tools currently implement single-bit logic using ALU instructions. (Few of our benchmark circuits use single-bit signals.)

4 RVETool Flow and Algorithms

This section describes RVETool, a tool flow which maps circuits onto RVEArch and other MPPA architectures. The input is a Verilog circuit, and the output is a configuration bitstream for the architecture or a simulator. The input circuit is also allowed to be many disjoint circuits, provided a common clock is used.

The tools use a graph representation of the circuit where graph nodes are circuit operations and graph edges are communication. The tool flow objective is to partition the circuit into clusters of executable code, one for each PE, and then schedule data movement and order code execution to reproduce the behaviour of the original circuit. At all steps, decisions favour minimizing the length of the overall schedule for the fastest possible simulation.

To test different parameters, an architecture file is used to specify the number of PEs, the width of buses, the size of each memory, and the resources in each PE (e.g., if the PE can perform I/O). These parameters act as constraints in the tool flow. In this paper we only vary the number of PEs.

RVETool is separated into four sub-tools: **Parallelize**, **Combine**, **Place**, and **Schedule**. Additionally, a **Simulate** step is used to mimic our target architecture. Each step is presented in the following sections.

4.1 Parallelize

The **Parallelize** tool parses the behavioural Verilog source and constructs an RTL graph representation of the circuit (including the control and data flow). All operations are left at a high-level and not elaborated to gates. It then performs graph legalization for execution on the target architecture.

The tool uses a modified version of Verilator [13] for parsing and graph construction. We allow Verilator to perform several processing steps and simple optimizations like module elaboration, dead code elimination, and constant folding. It also converts "free" hardware operations like bit-shifts or word-length truncations into shift and mask instructions. Verilator would normally generate a serialized C++ program for compilation and execution on the host processor, but we terminate it before it begins to serialize the graph.

The graph legalization is similar to technology mapping. Many operations in the Verilator output (e.g., the arithmetic and logic operations) map directly into PE instructions and are trivially converted. There are, however, other required graph transformations:

- Multiple writes to a single variable (a register, wire, or variable in the source Verilog) are mapped to a chain of multiplexers that feed a single write operation. This allows the computation of the written value to be (potentially) distributed among processors, while ensuring only one instance of the final value exists.
- Circuit inputs and outputs are mapped into I/O load and store operations. Later, the **Place** tool will restrict these operations to PEs with the I/O resource.
- User-instantiated memories, represented as array operations in the graph, are mapped to PE memory operations (load and store).
- Any node fanning out to a register is flagged as "end of cycle". If the node also fans out to a non-register it is duplicated first. All registers are then replaced with wires. This flag causes special treatment in the scheduler to recreate the expected clock-edge register behaviour.

4.2 Combine

The **Combine** tool groups operations in the graph into code clusters for each PE. **Combine** is similar to clustering in FPGA tools: wide nets and heavy communication nets are absorbed into a single cluster and kept off the communication network. But, instead of aiming to achieve fully packed clusters like a clustering tool, we want to combine code to varying degrees so that area (PEs) can be traded for performance. This soft capacity limit is demonstrated in Section 5.2.

A partitioning algorithm can achieve exactly this, so the tool uses hMETIS [37] to partition the graph using recursive bisection. To guide hMETIS, all nodes are assigned a weight of 1, except constant inputs which are assigned a weight of 0 so they can be placed in any cluster for free. All edges are assigned a weight equal to the bit-width of the variable on that edge. To ensure all operations involving a user memory reside in the same PE, load and store operations for a data memory are artificially connected with high edge weights to ensure they will not be separated.

4.3 Placement

The **Place** tool assigns code clusters to physical PEs while trying to keep the critical path as small as possible.

The problem is similar to the FPGA CAD placement problem, so we use VPR's [38] timing-driven annealing algorithm with a different cost function. The pipelined routing network in RVEArch means the delay between two nodes is equal to the Manhattan distance, not a propagation delay along a wire as in conventional FPGA CAD. The time-multiplexed PE cores introduce an additional level of complexity not found in FPGAs: two nodes within the same PE may be scheduled in timeslots far apart, causing additional critical-path delay. Unfortunately this delay is not known until scheduling is complete, so at this stage we assume it is zero.

The delay cost between two nodes i and j, in units of clock cycles, is:

$$\mathrm{delay}(i, j) = \begin{cases} 1 & i, j \text{ placed in same PE} \\ 1 & i, j \text{ placed in adjacent PEs} \\ 2 + mh(i, j) & \text{otherwise} \end{cases}$$

Where $mh(i, j)$ is the Manhattan distance between the PEs of i and j. For nodes in the same PE, the PE core must execute an instruction to produce the next value. For adjacent PEs, the value must be produced and communicated over a neighbour link, which requires no additional time. For distant PEs, there is a two cycle penalty to access the pipelined routing network, plus a number of cycles equal to the Manhattan distance to traverse it. The $delay(i, j)$ is used with a slack and criticality computation to calculate the *timing_cost* of the circuit, which is part of the placement cost function. The slack, criticality, and *timing_cost* computations are the same as in VPR [38].

In addition to timing cost, the VPR placement cost function uses a wiring cost, which we compute differently than VPR:

$$\text{wiring_cost} = \sum_{\forall i,\, j \in \text{circuit}} mh(i, j)$$

i.e., the Manhattan distance from every source to every sink. Since all communication links are time-multiplexed, the Manhattan distance prioritizes latency (performance) over interconnect utilization. This is different from traditional bounding-box minimization [39], where limited physical wiring forces wirelength to be more important than delay.

The final placement cost function is similar to VPR's:

$$\Delta C = \lambda \cdot \frac{\Delta\text{timing_cost}}{\text{previous_timing_cost}} + (1-\lambda) \cdot \frac{\Delta\text{wiring_cost}}{\text{previous_wiring_cost}} + \text{penalty}$$

The variable *penalty* is used to discourage illegal placements by adding 1,000 to the cost each time memory size is exceeded, unavailable PE resources are used, or too few/many PEs are used. The parameter λ is used to place more (or less) emphasis on the timing-driven aspects of placement; for this work, $\lambda = 0.9$.

At the end of placement, it is possible that several small, user-instantiated memories have been placed in the same PE. The **Place** tool assigns a base offset address for each user-instantiated memory, ensuring they do not overlap in PE data memory D. It then updates the relevant LOAD/STORE operations with this base offset. Individual memory addresses are assigned after scheduling. Note this step only handles user memories that are smaller than the PE data memory, so circuits that instantiate large user memories (larger than 8kB) cannot be compiled. The problem of splitting a large user memory across multiple PEs is left for future work.

4.4 Schedule

The **Schedule** tool orchestrates the overall execution of code and movement of data to reproduce the behaviour of the original circuit. It assigns each instruction to a timeslot in a PE core, and assigns each route-hop to a timeslot in the PE routers, resolving all routing collisions along the way.

The main loop of the **Schedule** tool is shown in Fig. 4. The algorithm is variation of list scheduling. The scheduler begins at *timeslot* = 1 and assigns as many nodes as it can across all PEs in that timeslot. It then moves on to the second timeslot, and so on. This timeslot-oriented approach ensures the scheduler is fast and is always making forward progress. NOP instructions are inserted in all timeslots that do not contain a circuit node after scheduling.

```
1:  ready_queue ← all nodes flagged "end of cycle" or
    nodes with no parent
2:  timeslot ← 1
3:  loop
4:    if is_empty(ready_queue) then
5:      if is_empty(next_queue) then
6:        return /* Scheduling complete */
7:      end if
8:      /* Swap queues, increase to next timeslot */
9:      ready_queue ← next_queue
10:     timeslot ← timeslot + 1
11:   end if
12:   /* Find a schedulable node */
13:   node ← dequeue(ready_queue)
14:   if not is_schedulable(node) then
15:     enqueue(next_queue, node)
16:     continue /* Restart loop */
17:   end if
18:   /* Create routes, record scheduled timeslot */
19:   schedule_routes(node)
20:   node.timeslot ← timeslot
21:   /* Increment sched. count in all children, enqueue
      any that
22:    * are now schedulable */
23:   for all node.children do
24:     child.sched_parent ← child.sched_parent + 1
25:     if child.sched_parent = |child.parents| then
26:       enqueue(next_queue, child)
27:     end if
28:   end for
29: end loop
30: /* Optional tail-to-head optimization */
31: tail_to_head()
```

Figure 4 Schedule tool main loop.

The is_schedulable(*node*) function checks whether *node* is schedulable in the current *timeslot*. It verifies that all of the following are true:

- The code position at *timeslot* is empty in the PE
- *node* may be scheduled as early as *timeslot*
- All PE core resources required by *node* are available
- All PE router resources required by the output of *node* are available for the first-hop of the route

Table 1 Synthesis/compile time, simulation speed, and density comparisons (RVE results average of 100 trials).

Circuit	Synthesis/compile time (s)				Best simulation speed (f_{max}, MHz)						Area at best speed		
	Vltr	MS	Stx-III	Amb, RVE	Vltr	MS	Amb	Stx-III	RVE no crit	RVE crit	Stx-III (ALMs)	RVE crit (PEs)	Density (ALMs/PE)
AES	3	1	148	6.33	2.61	0.480	4.8	559	29.9	20.9	191	224	0.9
pr	3	1	228	2.25	2.36	0.016	2.7	165	42.9	58.7	382	12	31.8
wang	3	1	182	2.49	2.33	0.015	8.0	158	43.5	58.4	442	24	18.4
honda	3	1	202	2.62	2.19	0.027	14.0	237	24.8	33.3	547	32	17.1
mcm	3	1	219	2.68	2.23	0.014	6.7	222	40.5	44.7	609	40	15.2
dir	3	1	372	3.04	1.66	0.012	8.5	183	15.1	19.2	1,084	48	22.6
FFT8	3	1	207	3.29	1.76	0.058	13.1	121	63.8	56.9	1,974	96	20.6
chem	4	1	477	5.00	0.49	0.013	12.5	176	26.8	29.7	2,278	48	47.5
ME	3	1	277	8.74	0.15	0.001	12.0	337	33.0	38.5	3,018	256	11.8
FFT16	3	1	790	5.04	0.73	0.022	4.4	237	42.6	38.3	4,678	192	24.4
geomean	3	1	272	3.7	1.27	0.02	7.7	218	33.9	37.2	970.2	62.4	15.5
FFT8-human									167	167		64	
ME-human									250	250		64	

The schedule_routes(*node*) function, described further in the next subsection, computes a series of route-hops from the node to all sinks, then assigns the route-hops to timeslots in the PE routers.

When a signal arrives at the destination PE, the destination node(s) are marked with the arrival timeslot; those nodes cannot be scheduled before the time of arrival.

The enqueue(*node*) function can optionally use the criticality computed during placement to order the nodes in the queue so the most critical nodes will dequeue first. This reduces the final schedule length and thus increases the f_{max} by ≈9.7%, on average. This improvement is shown between the two columns labeled 'RVE no crit' and 'RVE crit' in Table 1.

At the end of scheduling, memory accesses (to node memory and data memory) are assigned specific offsets using a greedy approach. At this point, the code for each router and core is ready to be packed into a single output bitstream.

4.5 Schedule—Routing

The routing algorithm is contained within the **Schedule** tool, and is called when needed. It computes a series of route-hops from a given source location to a PE sink. The routing problem is different from conventional CAD flows because the routing network is time-multiplexed, so the routing process must make temporal as well as the conventional spatial routing decisions. RVEArch also contains two routing resources: PE routers for moving data beyond the 4 immediate neighbours (requiring $2 + h$ cycles, where h is the number of hops), and single-cycle links for communicating between adjacent PEs.

For long-distance routes, the routing algorithm uses a simple horizontal-then-vertical routing strategy, routing each path without considering congestion or collisions.

When a routing conflict arises, the router holds the value in place for as many timeslots as necessary until a free timeslot is found in the next hop. In practice, however, we find there is little routing contention. This is shown in the results section in the last column of data in Table 5 labeled 'Hold'.

For neighbour communication, the router creates a route-hop that uses the neighbour-neighbour link, again ignoring congestion and conflicts. The **Schedule** tool assigns the link usage to a timeslot and directs the value to a specific offset of the N, S, E, or W node memory, or may delay the producing node if resources are unavailable.

4.6 Schedule—Tail-to-Head Optimization

The **Schedule** tool can implement time-borrowing between pipeline stages by moving some operations done at the tail-end of the schedule to the head. If all tail operations in the last timeslot can be moved to the head, the overall schedule length can be shortened by 1 cycle, resulting in a higher f_{max}.

We implemented this tail-to-head optimization as follows. Nodes are relocated until a node cannot be moved, e.g. because of unavailability of routing resources or the ALU. If all nodes are moved out of the last timeslot, the schedule length is decreased and all wrap-around routes are re-computed. Then, optimization tries to shrink the schedule again.

Testing shows this optimization improves the overall schedule length by an average of 1.11 cycles (see

Table 4). However, because of the complexity added to the architecture to support this optimization (needing to discard the output of the first run of instructions that are at the tail), we have opted to not include this optimization in any other results presented in this paper.

4.7 Simulation

The bitstream is given to a compiled-code architectural simulator to ensure functional correctness, or given to a statistics tool to extract performance data. All the results presented in this paper come from the statistics tool after a complete pass through the toolflow. The simulator runs sequentially on a PC, but uses its own thread scheduler to emulate parallel execution of the PEs. The simulator accepts input stimulus from vector waveform files and writes output waveform files. We verified the correctness of our entire toolflow by simulating all benchmark circuits (excluding the randomly-generated circuits) by comparing output waveforms against those produced by Quartus II.

5 Experimental Results

In the following sections, we evaluate RVETool and RVEArch. We begin in Section 5.1 by examining their overall ability to quickly synthesize and simulate computational circuits compared to traditional FPGA tools and software simulators. Next, in Section 5.2, we evaluate their ability to scale a circuit across a number of PEs, to better exploit parallelism than software simulators. In Section 5.3 we compare computer-generated bitstreams with hand-tuned ones, to explore the performance achievable by further refining our CAD tools. In Section 5.4 we explore the resources needed by RVEArch to implement the circuits. Finally, in Section 5.5, we investigate the synthesis speed of the tools with very large circuits (up to ≈1.6 million gates).

To evaluate each of these properties, we introduce several benchmarks written in Verilog:

- The **chem**, **dir**, **honda**, **mcm**, **pr**, and **wang** benchmarks [40] are dataflow– and DSP-style non-pipelined computational circuits described in behavioural Verilog.
- **AES** is a byte-oriented implementation taken from Altera's QUIP suite [41]. It was modified in several places to replace bit-shuffling computation with pre-computed table lookups to improve performance on the target word-oriented architecture. This is because the PLA in RVEArch, which would provide the desired high-performance bit-oriented operations, is not yet supported.
- **Motion Estimation** (ME) is an image-processing algorithm described in [8]; a block R of 16×16 pixels is swept against a 32×32-pixel reference image M, searching for the displacement which produces the lowest sum of absolute differences (SAD).
- **FFT8** and **FFT16** are 8- and 16-point complex FFTs respectively, implemented using a radix-2, decimation-in-time decomposition.

Of these benchmarks, ME and FFT16 are the largest, involving 3,609 and 1,192 circuit nodes, respectively. The other benchmarks' complexity varies from 90 nodes (wang) to 553 nodes (chem). A node is roughly equivalent to one arithmetic, logical, or routing operation up to 32 bits in size.

5.1 Synthesis and Simulation Speed

We first consider the synthesis speed and simulation speed of the toolflow and architecture, respectively. Table 1 shows the average of 100 compilations of both the time required to synthesize Verilog into a format suitable for simulation, as well as the corresponding simulation speeds for several platforms (all synthesis time results were gathered using the same PC):

- Verilator (abbreviated Vltr), translates Verilog into a C++ program run on a 2.8 GHz Intel Pentium IV PC,
- ModelSim 6.3d (abbreviated MS),
- Idealized Ambric (abbreviated Amb, see Section 3.1) at 1 GHz, code estimated from output of RVETool,
- Altera Stratix-III FPGA (EP3SE80F, speed −2; abbreviated Stx-III), using Quartus II 9.0,
- RVEArch without criticality (abbreviated RVE no crit) at 1 GHz, results previously presented in [11], and
- RVEArch with criticality (abbreviated RVE crit) at 1 GHz, as described in Section 4.

Figure 5 shows the best simulation speed results from Table 1 in graphical form. In all cases, the FPGA implementation runs faster than the alternatives, but it also takes a much longer time to synthesize. The RVEArch provides the fastest simulation results of the software-based alternatives, with a synthesis time comparable to Verilator that is roughly 70 times faster than the FPGA tools. The Idealized Ambric results show that RVETool can generate fast, scalable code

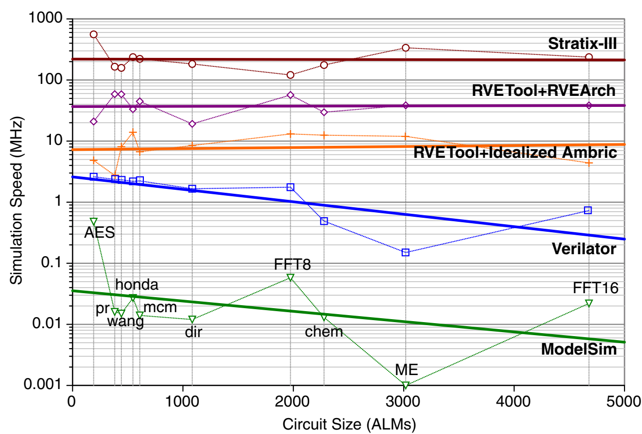

Figure 5 Simulation speed comparison.

for other MPPA architectures as well. Not only do the software platforms exhibit the poorest performance, they demonstrate poor scalability as the larger circuits tend to run more slowly. In comparison, RVEArch and Ambric maintain a relatively constant simulation performance as the circuit size increases.

The fast synthesis speed of RVETool is a combination of targeting a coarse-grained MPPA (less work for the tools), not synthesizing to gates (less work for the tools), and using algorithms that allow quality of results to be balanced against runtime.

The results of the AES benchmark are slightly anomalous. Quartus and ModelSim implement the table lookups we introduced for the bit-shuffling logic extremely well, resulting in high performance on those platforms. Verilator does not optimize these constructs as aggressively, resulting in average performance in both Verilator and on our architecture.

5.2 Platform Scalability and Density

In this section, we demonstrate the ability of the toolflow to trade the number of available processing elements (PEs) with the effective user clock rate (simulation speed), avoiding the hard capacity limits imposed by commercial FPGAs.

Figure 6a shows the simulation speed as the size of the array ranges from 9 to 256 PEs (3×3 to 16×16) this the complete dataset from which the RVE columns in Table 1 were created. When a small number of PEs are used, speedup is linear with the number of PEs. Outside the linear region, each benchmark exhibits an optimum PE allocation where its performance peaks. Beyond this peak, performance decreases as inter-PE communication paths lengthen.

Of the benchmarks shown in Fig. 6a, Motion Estimation (ME) performance peaks at a large number of PEs because it is a large circuit and it exhibits chiefly nearest-neighbour communications. Another large circuit, FFT16 scales similarly even though it exhibits more complex communication patterns. Also, the smallest circuits (wang and pr) achieve peak performance with the fewest PEs. In general, the number of PEs needed to achieve a benchmark's peak speedup tends to be correlated with circuit size. Real computational circuits would likely be much larger than any of these small benchmarks, so we anticipate the architecture is scalable far beyond 256 cores.

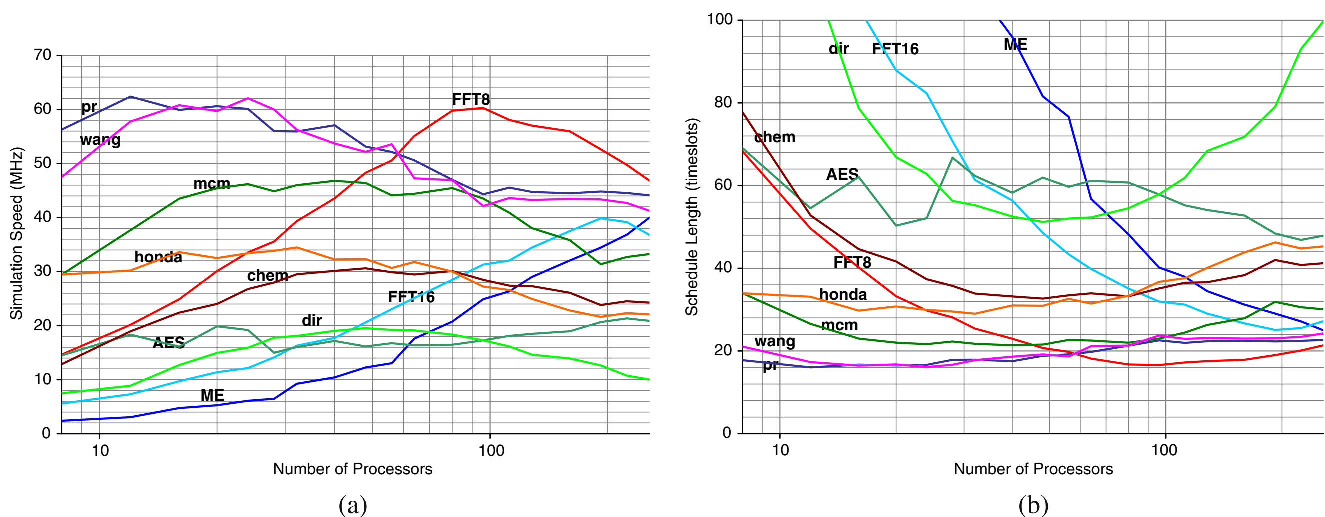

Figure 6 **a** Simulation speed (f_{max}) for benchmarks synthesized for 9-256 PEs. **b** Schedule length for 9–256 PEs.

Figure 6b shows the same results as Fig. 6a, but plots the schedule length rather than $f_{\max}$. The schedule length indicates the size of the instruction memory needed.

We roughly estimate the silicon area of one PE in terms of Stratix-III ALMs as follows. First, we estimated the largest Stratix-III die size as 850 mm^2 at 65 nm, which contains 135,000 ALMs [42]. Next, we budgeted 0.5 mm^2 for each PE in 65 nm. This allows room for several blocks of 16kB SRAM (0.05 mm^2 each [43]), a PE as powerful as a 32-bit ARM core with a multiplier (0.1 mm^2 [44]), and additional resources including routing. From this, we deduce that a PE is roughly equivalent in area to $\frac{135{,}000}{850/0.5} \approx 80$ ALMs.

From Table 1, at peak performance the RVE average number of PEs used is 62.4 and the average density is 15.5 ALMs/PE. This means that RVEArch offers $\frac{15}{80} = \frac{3}{16}$ the density of an FPGA when operating at peak clock speeds. However, RVETool can scale the implementation to any number of PEs. To improve density, Table 2 shows the simulation speed results of our architecture at a target density of 80 ALMs/PE (meaning that the circuit is implemented in the same silicon area as an FPGA). At the same density, the average speed is slightly less than 1/10th that of the FPGA. Continuing the scaling to reach 10× FPGA density (800 ALMs/PE), the average speed is 3.6 MHz. At this density, all circuits smaller than FFT8 are implemented on a single PE. While this is not ideal, it demonstrates that our tools can fold a design in space. It also shows an advantage of using a custom architecture. Even on a single PE, RVEArch achieves a higher $f_{\max}$ than the Verilator compiled simulation for many benchmarks. This is because, on average, Verilator uses 6.38 × 86 assembly instructions for each node (compiled with g++ −O3 −S, counted only instructions for the circuit, excluded comments, labels, and code included from the Verilator support libraries and macros) whereas RVEArch can implement each node in a single instruction which executes in a single cycle.

5.3 CAD Tool Efficiency

The simulation speed of each benchmark is determined by the benchmark itself, the CAD tools, and the architecture. To investigate how well our toolflow maps code onto each PE, we compare with the lowest-bound schedule length for all benchmarks, and also to hand-written code for two of the larger benchmarks (FFT8 and ME).

Table 3 compares the lowest possible schedule length with the actual schedule lengths used to compute the $f_{\max}$ results in Table 1. These bounds are determined from the maximum depth of the user circuit graph after the **Parallelize** tool, which means communication delays are excluded. On average, our results are just over two-fold worse than the lower bound. This indicates that, while there is room to improve our results, there are not order-of-magnitude improvements to be found. The two worse results, FFT8 and FFT16, are heavily pipelined—due to pipelining at each stage, the maximum graph depth is 4. However, registered results saved in a PE at the end of the schedule are subsequently used in a different PE at the beginning of the next iteration of the schedule. The results suggest it is difficult for the tools to identify and exploit spatial locality in these circuits.

Table 4 shows the reduction in schedule length for the tail-to-head optimization described in Section 4.6. The data was generated the same way as in Fig. 6 (100 trials of architectures from 9 to 256 PEs). The schedule

Table 2 Simulation speed at 80 and 800 ALMs/PE density.

		1× FPGA density 80 ALMs/PE		10× FPGA density 800 ALMs/PE	
Circuit	ALMs	Req'd PEs	Speed (MHz)	Req'd PEs	Speed (MHz)
AES	191	4	7.2	1	1.9
pr	382	4	40.4	1	1.1
wang	442	8	47.5	1	6.1
honda	547	8	29.4	1	1.3
mcm	609	8	29.4	1	8.7
dir	1,084	12	8.9	1	8.3
FFT8	1,974	28	35.6	4	7.8
chem	2,278	28	28.0	4	6.7
ME	3,018	40	10.4	4	1.2
FFT16	4,678	56	23.0	8	5.6
geomean		13.3	21.9	1.9	3.6

Table 3 Comparison of lower bound vs. actual schedule lengths (average of 100 trials).

Circuit	Lower bound SL	Actual best SL	Factor
AES	15	47.9	3.2
pr	10	17.0	1.7
wang	10	17.1	1.7
honda	19	30.0	1.6
mcm	11	22.4	2.0
dir	25	52.2	2.1
FFT8	4	17.6	4.4
chem	18	33.7	1.9
ME	18	26.0	1.4
FFT16	4	26.1	6.5
geomean			2.3

Table 4 Schedule length reduction for tail-to-head optimization (average of 100 trials).

Circuit	Avg. timeslots saved
AES	0.32
pr	1.68
wang	1.74
honda	2.29
mcm	2.36
dir	1.95
FFT8	1.22
chem	1.92
ME	0.09
FFT16	1.36
geomean	1.11

lengths were then compared to those in Fig. 6. It is encouraging that the optimization performs consistently, on average reducing the schedule length by 1.11 cycles. However, as mentioned earlier, this adds complexity to the architecture for a modest performance improvement. Hence, we have not used this optimization in any of the other results.

Figure 7 compares the scalability and performance of RVETool to a human-written implementation. The ME-human benchmark scales superlinearly because it is able to eliminate memory loads and stores by instead sending results to neighbouring PEs for processing. This figure demonstrates that RVEArch is able to scale aggressively, and that RVETool is able to track this scalability curve up to a certain point.

At peak performances shown, FFT8 and ME show performance gaps of roughly 2.8 and 4.8×, respectively, between handcrafted and tool-produced results. At this stage, the tools are all first-generation algorithms; the focus has been on infrastructure development and integration, not performance or quality of results. We hope to reduce this performance gap as we refine the algorithms.

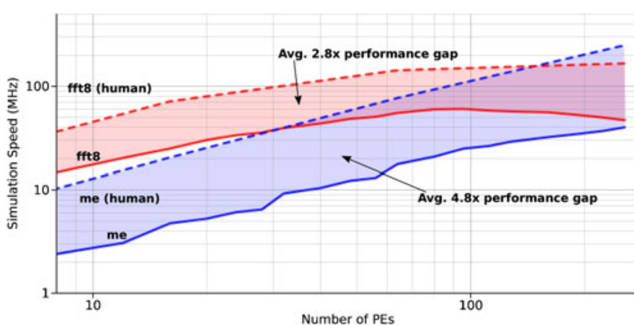

Figure 7 Performance of human- and machine-generated bitstreams.

5.4 Longest-Path and Resource Usage Analysis

The routing and compute resources for the longest paths in each circuit can tell us where our algorithms might be improved to reduce the overall schedule length. The length of a path in the scheduled circuit is the time required to traverse all compute and routing resources from inputs and registers, to outputs and registers. We use the longest path instead of the critical path because the critical path is difficult to define in a time-multiplexed environment where a non-critical path may be delayed (by the scheduler) until it appears to be critical, even though it isn't. The schedule length cannot be smaller than the longest path, so in that sense they are also critical.

Table 5 shows the longest-path analysis for the best-speed results from Table 1. All results are again the average of 100 trials. In the table, starting from the left is the schedule length of the synthesized circuit, and then the longest path. Next is the number of longest-paths because there is often more than one. The longest path will always be less than or equal to the schedule length; it can be lower in cases where an input does not immediately occur in the first timeslot due to scheduling decisions, or an output occurs before the last timeslot.

The next three columns in Table 5 show the number of routes of each type along the longest path. Local uses PE memory R to communicate, Nbour uses the neighbour memories N, W, E, or W, and Remote uses the routing network and memory X. The higher number of local routes shows that there is locality in the circuits, and the tools are finding it. The neighbour routes indicate that the tools are also scheduling some compute–and–move operations, saving routing timeslots.

The last five columns are the longest-path breakdown: the number of compute-only slots where an ALU is in use; the number of wait slots spent waiting for values to arrive or waiting for the ALU while it is busy with other computation; the number of compute-and-move slots (equal to the number of neighbour routes); the number of routing timeslots where progress is made; and the number of routing timeslots where a value is held due to a route conflict.

The number of compute timeslots and route timeslots are close, suggesting that the solution may benefit from more compute-and-route operations using the neighbour links to combine a compute and move into a single cycle. The almost-zero number of hold slots shows that our horizontal-then-vertical routing strategy combined with the abundance of routing resources means there are indeed few routing conflicts.

Table 5 Longest-path breakdown for best speed of each benchmark (average of 100 trials).

Circuit	Schedule length	Longest-path		Number of route-link types in longest path			Longest-path timeslots				
		Length (timeslots)	Number of paths	Local	Nbour	Remote	Compute		Compute &Route	Route	
							Compute	Wait		Route	Hold
AES	47.9	43.1	10.6	1.7	1.4	2.1	4.9	27.8	1.4	9.0	0.1
pr	17.0	17.0	9.7	4.6	1.4	0.7	6.2	7.6	1.4	1.8	0.0
wang	17.1	17.1	13.2	3.4	2.4	0.8	5.2	6.8	2.4	2.7	0.0
honda	30.0	30.0	31.8	7.5	3.5	1.0	9.5	13.5	3.5	3.5	0.0
mcm	22.4	22.4	17.6	4.5	1.6	1.1	6.6	9.7	1.6	4.5	0.0
dir	52.2	48.6	34.3	6.1	1.2	2.2	8.7	30.5	1.0	9.1	0.0
FFT8	17.6	15.9	3.7	1.2	0.4	0.4	2.0	10.6	0.3	3.3	0.0
chem	33.7	33.7	14.2	8.8	1.3	1.1	10.8	17.5	1.3	4.1	0.0
ME	26.0	26.0	24.0	9.6	1.5	0.0	10.6	13.9	1.5	0.0	0.0
FFT16	26.1	22.9	2.1	1.1	0.1	0.3	1.5	19.2	0.0	2.7	0.0

The number of wait cycles is larger than the number of compute cycles. Wait cycles are the extra cycles between the time an ALU operation is scheduled and the time of its most closely scheduled predecessor ALU operation. Some wait cycles are due to a ready operation waiting for the ALU, because it is busy servicing a large number of other operations that are also ready. Other wait cycles are due to an operation waiting for an operand to arrive over the routing network; in this case, the operand has already been computed but it must be transported. To reduce the number of wait cycles, it is possible to change the architecture by adding multiple ALUs per PE, or adding longer interconnect wires that span multiple PEs in a single clock cycle, or both. Future work will investigate these and other architectural options for reducing waiting time.

Table 6 shows the maximum memory entries used by the corresponding memory type in a compilation using 16 or 256 PEs. The memories would need to have the number of entries indicated here to successfully implement the benchmark without artifically inflating the schedule length to use less memory. Figure 8 plots the register usage for the chem benchmark over a range of architecture sizes. The other benchmark circuits show a similar trend. As expected, the maximum usage within a single PE decreases as the circuit is spread over space, showing the tools are distributing the work to more PEs.

The modest sizes indicated by Table 6 suggest that only small memories are required. Also, we note that the amount of instruction memory required in each PE is also small—for the peak performance results shown in Table 5, the schedule length is less than 50 words. These modest memory sizes partially demonstrate the feasibility of the architecture: from a memory perspective, it is possible to implement PEs that are both area-efficient and can run at a high speed of 1 GHz.

Table 6 Maximum node memory resource usage in a single PE.

Circuit	16 PEs			256 PEs		
	Local	Nbour	Remote	Local	Nbour	Remote
AES	14	4	8	6	2	5
pr	6	2	3	1	1	2
wang	5	3	3	1	1	3
honda	5	2	2	1	1	3
mcm	7	2	3	1	1	3
dir	36	4	20	6	4	7
FFT8	27	3	5	5	3	4
chem	12	3	8	4	2	4
ME	140	1	1	11	2	2
FFT16	77	4	9	13	4	4

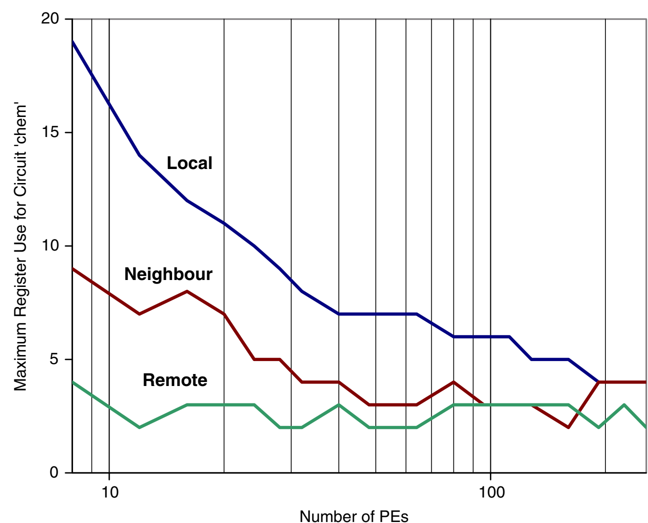

Figure 8 Maximum PE memory usage for the chem benchmark for architectures with 9–256 PEs.

5.5 Tool Scalability to Large Circuits

In this section we test RVETool's runtime on large circuits. Since large benchmark circuits are difficult to acquire, we randomly-generated *synthetic* benchmarks with a custom tool. The circuits are generated without any graph-depth or locality control, so they are not suitable for testing the quality of the tool output. For such testing, a more realistic random circuit generator would be required.

The circuits have 32 inputs, 16 outputs, and contain 1,000 to 50,000 nodes. Each node is a 32-bit operation (add, subtract, invert, and, or, xor, and multiply). To put this into perspective, if we assume the simplest case where each 32-bit operation requires 32 gates, the 50,000 node benchmark contains 1.6 million gates.

We also attempted to synthesize the circuits in Quartus-II v9.0 with the largest Stratix-III (EP3SL340F1760C2) to produce a rough comparison with FPGA area usage and runtime. The 1,000 node circuit required 22,066 ALMs (16.2% of the device) and synthesized in 9 min. The 2,000 node circuit used 64,748 ALMs (47.6%) synthesized in 34 min. The 4,000 node circuit used 152,338 ALMs (112%) and stopped after 80 min as it exceeded the size of the device. The 6,000 node circuit exhausted the 32-bit 4 GB memory limit and was forced to quit after 2 h. The 10,000 node circuit exhausted the memory after 13 h. The 50,000 node circuit ran for 24 h without finishing the analysis and synthesis phase (before placement and routing in Quartus). To be fair, Quartus II was probably attempting to do more optimizations than Verilator, so our randomly generated circuits may have been too unrealistic for it to handle.

Figure 9a shows the average synthesis time for RVETool when run with ten trials of the random benchmarks (a different random benchmark of the same size for each trial) compiled for an architecture of 1,024 (32×32) PEs. The total runtime curve is shown, plus a breakdown for each step in the toolflow. As the circuit size increases, the first two tool steps, **Parallelize** and **Place**, dominate runtime. For the 50,000 node circuit, the total synthesis time was 54 min.

To reduce the overall time, the **Parallelize** (parsing, elaboration, optimization, and DFG generation) and **Place** steps are the two most likely targets. However, it is unlikely that the parsing step can be improved much, except by reducing the optimization done. The **Place** step can be improved by reducing the inner-loop iterations of the annealer, at the expense of quality. The number of inner-loop iterations is the same as in VPR, $10 \times n_clusters^{1.3}$, which we have found to be a good balance between quality and speed. Increasing this causes longer run-times with little or no improvement in results, and decreasing it gives significantly poorer results. Beyond this, placement could be improved by changing to a fundamentally different approach, e.g. analytical placement. Alternatively, it was recently

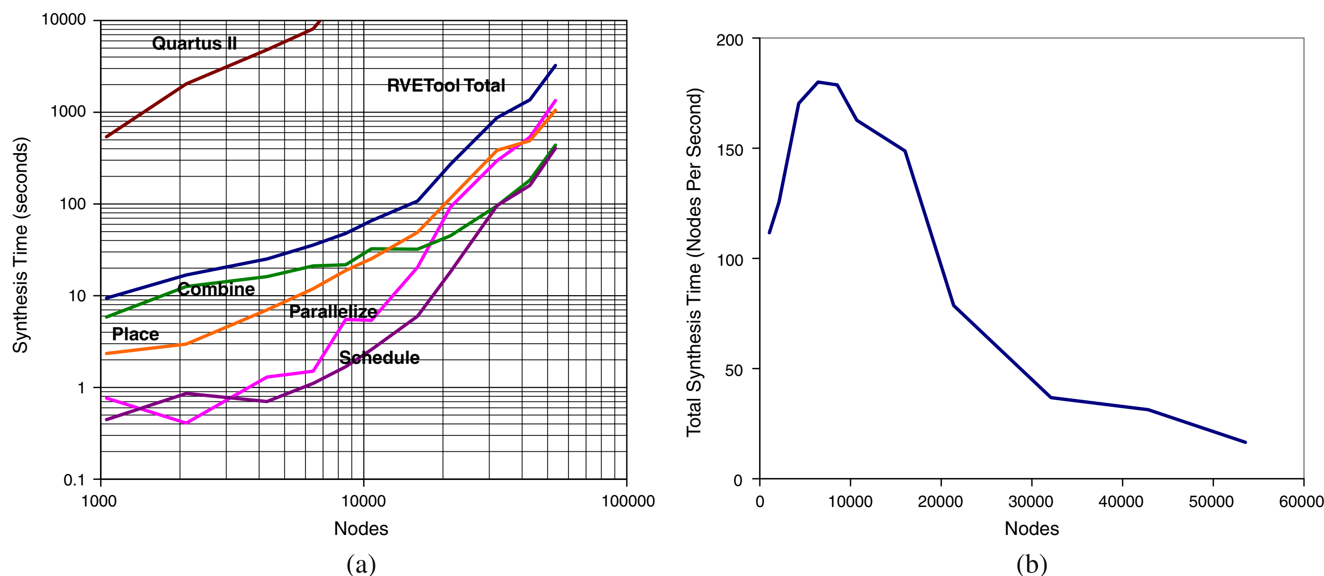

Figure 9 **a** Synthesis time for placing large random circuits on 1,024 PEs (32×32). **b** Synthesis time nodes-per-second for large random circuits.

shown that an MPPA is capable of greatly accelerated placement by self-hosting a parallel simulated-annealing algorithm [45].

Figure 9b shows the total synthesis time as a rate in nodes-per-second. The initial rise in rate, ending at ≈8,000 nodes, is caused by amortization of the tool overhead. Beyond this, algorithmic complexity catches up. Scaling to even larger circuits than shown here may require heuristics with better algorithmic complexity or with reduced quality of results.

6 Conclusions

In this paper, we presented a CAD toolflow (RVETool) that maps computational circuits expressed in Verilog onto an MPPA architecture (idealized Ambric) and a custom architecture (RVEArch). We evaluated the tools using a number of dataflow and DSP-type benchmarks, and demonstrated their performance relative to FPGAs (70× faster compilation, 5.8× slower simulation) and software simulators (on-par compilation, 29× faster simulation). The RVETool + RVEArch platform simulates computational circuits with better performance than software simulators, without incurring the long synthesis time of FPGA tools. RVEArch also shows a 3.8× performance improvement over the idealized Ambric architecture because of the separate router.

We explored the trade-off between density (number of PEs used) and simulation speed of RVEArch, demonstrating that it can avoid the hard capacity limit of FPGAs using time multiplexing. We also examined RVETool's ability to effectively distribute a circuit simulation across a number of PEs, and its ability to compile very large circuits in a reasonable amount of time. We have shown the maximum resource usage of the benchmarks are reasonable for implementation in an architecture.

As more algorithms are converted to computational circuits, a means for fast synthesis and fast simulation becomes even more important. Besides reducing design time and risk, designing a circuit at a behavioural level does not require as much hardware design experience. This allows software designers to participate in the hardware design (and testing) process, further encouraging the development of computational circuits.

Acknowledgements This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Equipment donations by CMC Microsystems and Ambric are gratefully acknowledged. The authors would also like to thank Deming Chen for providing several benchmark circuits as well as Wilson Snyder, Duane Galbi, Paul Wasson, and the many additional contributors to the Verilator open source tool.

References

1. Shaw D. E., et al. (2007). Anton, a special-purpose machine for molecular dynamics simulation. In *Proc. ISCA* (pp. 1–12).
2. Zygouris, V., et al. (2005). A navier-stokes processor for biomedical applications. In *Proc. SiPS* (Vol. 2–4, pp. 368–372).
3. Altera Corporation (2009). *Video and image processing example design*.
4. Tian, X., & Benkrid, K. (2009). American option pricing on reconfigurable hardware using least-squares Monte Carlo method. In *Proc. FPT* (Vol. 9–11, pp. 263–270).
5. Fender, J., & Rose, J. (2003). A high-speed ray tracing engine built on a field-programmable system. In *Proc. FPT* (Vol. 15–17, pp. 188–195).
6. Donev, A., et al. (2010). A first-passage kinetic Monte Carlo algorithm for complex diffusion-reaction systems. *Journal of Computational Physics, 229*(9), 3214–3236.
7. Boukerche, A. (2000). Conservative circuit simulation on multiprocessor machines. In *Proc. high performance computing* (pp. 415–424).
8. Grant, D., & Lemieux, G. (2008). A spatial computing architecture for implementing computational circuits. In *Proc. MNRC* (pp. 41–44).
9. Jones, D., & Lewis, D. (1995). A time-multiplexed FPGA architecture for logic emulation. In *Proc. custom integrated circuits* (pp. 495–498).
10. DeHon, A. (1996). *Reconfigurable architectures for general-purpose computing*. Master's thesis, Massachusetts Institute of Technology.
11. Grant, D., et al. (2009). Rapid synthesis and simulation of computational circuits in an MPPA. In *Proc. FPT* (pp. 151–158).
12. Lewis, D. (1991). A hierarchical compiled-code event-driven logic simulator. *IEEE Transactions on CAD, 10*(6), 726–737.
13. Snyder, W. (2007). Verilator-3.652. Available: www.veripool.com/verilator_doc.pdf.
14. Greaves, D. (2000). A verilog to C compiler. In *Proc. rapid system prototyping (RSP)* (pp. 122–127).
15. Ching, J. (2007). VBS project homepage. Available: www.flex.com/~jching/.
16. Symphony EDA (2008). VHDL simili v3.1 whitepaper. Available: www.symphonyeda.com/white_paper.htm.
17. Soulé, L., & Blank, T. (1988). Parallel logic simulation on general purpose machines. In *DAC* (pp. 166–171).
18. Bailey, M. L., et al. (1994). Parallel logic simulation of VLSI systems. *ACM Computing Surveys, 26*(3), 255–294.
19. Webber, D., & Sangiovanni-Vincentelli, A. (1987). Circuit simulation on the connection machine. In *DAC* (pp. 108–113).
20. Li, L., et al. (2003). DVS: An object-oriented framework for distributed Verilog simulation. In *Proc. parallel and distributed simulation* (p. 173).
21. Dong, W., et al. (2008). WavePipe: Parallel transient simulation of analog and digital circuits on multi-core shared-memory machines. In *Proc. design automation conference* (pp. 238–243).

22. Li, T., et al. (2004). Design and implementation of a parallel Verilog simulator: PVSim. In *Proc. VLSI design* (pp. 329–334).
23. Jaeger, J. (2007). Virtually every ASIC ends up an FPGA. Available: www.eetimes.com/showArticle.jhtml?articleID=204702700.
24. Cadence (2006). Incisive enterprise palladium series with incisive XE software (datasheet).
25. Mentor Graphics (2008). VStationPRO high-performance system verification (datasheet).
26. Goering, R. (2004). Cadence touts emulation/acceleration performance. Available: www.eetimes.com/showArticle.jhtml?articleID=51200173.
27. Mei, B., et al. (2003). ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. field-programmable logic and applications* (pp. 61–70).
28. Goldstein, S. C., et al. (1999). PipeRench: A coprocessor for streaming multimedia acceleration. In *ISCA* (pp. 28–39).
29. Mirsky, E., & DeHon, A. (1996). MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proc. FPGAs for custom computing machines (FCCM)* (pp. 157–166).
30. Mishra, S. C., & Goldstein, M. (2007). Virtualization on the Tartan reconfigurable architecture. In *FPL* (pp. 323–330).
31. Ebeling, C., et al. (1996). RaPiD—reconfigurable pipelined datapath. In *Proc. field-programmable logic and applications* (pp. 126–135).
32. Caspi, E., et al. (2000). Stream computations organized for reconfigurable execution (SCORE). In *FPL* (pp. 605–614).
33. Halfhill, T. R. (2006). Ambric's new parallel processor. *Microprocessor Report, 20*(10), 19–26.
34. Tilera (2007). Tile64 processor product brief. Available: www.tilera.com/pdf/ProBrief_Tile64_Web.pdf.
35. Perinkulam, A. S. (2007). *Logic simulation using graphics processors*. Master's thesis, University of Massachusetts Amherst.
36. deLorimier, M., et al. (2006). GraphStep: A system architecture for sparse-graph algorithms. In *FCCM* (pp. 143–151).
37. Karypis, G., et al. (1999). Multilevel hypergraph partitioning: Applications in VLSI domain. *IEEE Transactions on VLSI, 7*(1), 69–79.
38. Marquardt, A., et al. (2000). Timing-driven placement for fpgas. In *Proc. field programmable gate arrays* (pp. 203–213).
39. Betz, V., & Rose, J. (1997). VPR: A new packing, placement and routing tool for FPGA research. In *Proc. FPL* (pp. 213–222).
40. Srivastava, M. B., & Potkonjak, M. (1995). Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput. *IEEE Transactions on VLSI, 3*(1), 2–19.
41. Altera Corporation (2006). *Benchmark designs for the quartus university interface program (QUIP)*.
42. Altera Corporation (2007). *Stratix III device handbook*.
43. Agrawal, B., & Sherwood, T. (2006). Guiding architectural SRAM models. In *Proc. computer design (ICCD)* (pp. 376–382).
44. ARM (2009). Synthesizable ARM7TDMI^{TM} 32-bit RISC performance. Available: www.arm.com/products/CPUs/ARM7TDMIS.html.
45. Smecher, G., et al. (2009). Self-hosted placement for massively parallel processor arrays. In *Proc. FPT* (pp. 159–166).

David Grant received his BASc and MASc in Computer Engineering from the University of Waterloo, Canada in 2002 and 2004. He worked for a year at SlipStream Data, Inc. before starting his PhD. He is currently a PhD candidate at the University of British Columbia. His research interests include reconfigurable computing, massively parallel processing, and automatic compilation techniques for such systems.

Graeme Smecher completed his M. A. Sc. (honours, 2006) at Simon Fraser University, and graduated with a Master's in Engineering from McGill University (2009). His research interests include non-linear and statistical signal processing, with a focus on switching amplification. He currently consults for a cosmology laboratory at McGill University, where he designs FPGA-based readout electronics and firmware for the next generation of microwave telescopes.

Guy G. F. Lemieux received the B.A.Sc. degree from the Division of Engineering Science at the University of Toronto, and the M.A.Sc. and Ph.D. degrees in Electrical and Computer Engineering at the University of Toronto, Toronto, ON, Canada.

In 2003, he joined the Department of Electrical and Computer Engineering at The University of British Columbia, Vancouver, BC, Canada, where he is now an Associate Professor. He is co-author of the book *Design of Interconnection Networks for Programmable Logic* (Kluwer 2004). His research interests include FPGA architectures, computer-aided design algorithms, VLSI and SoC circuit design, and parallel computing.

Dr. Lemieux was a recipient of the Best Paper Award at the 2004 IEEE International Conference on Field-Programmable Technology.

Rosemary Francis completed her PhD in 2009 at the University of Cambridge. She developed a novel FPGA architecture with time-division multiplexed interconnect for the effective implementation of communication-centric systems, exploring both hard and soft network-on-chip designs. She is currently managing director of Ellexus Ltd, a software business serving the semiconductor implementation industry.