Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism

Mahyar Emami*
mahyar.emami@epfl.ch
EPFL
Lausanne, Switzerland

Sahand Kashani*
sahand.kashani@epfl.ch
EPFL
Lausanne, Switzerland

Keisuke Kamahori†
k-kamahori@g.ecc.u-tokyo.ac.jp
University of Tokyo
Tokyo, Japan

Mohammad Sepehr Pourghannad†
mspourghannad@ce.sharif.edu
Sharif University
Tehran, Iran

Ritik Raj†
ritik_r@ch.iitr.ac.in
Indian Institute of Technology Roorkee
Roorkee, India

James R. Larus
james.larus@epfl.ch
EPFL
Lausanne, Switzerland

ABSTRACT

The demise of Moore's Law and Dennard Scaling has revived interest in specialized computer architectures and accelerators. Verification and testing of this hardware depend heavily upon cycle-accurate simulation of register-transfer-level (RTL) designs. The fastest software RTL simulators can simulate designs at 1–1000 kHz, i.e., more than three orders of magnitude slower than hardware. Improved simulators can increase designers' productivity by speeding design iterations and permitting more exhaustive exploration.

One possibility is to exploit low-level parallelism, as RTL expresses considerable fine-grain concurrency. Unfortunately, state-of-the-art RTL simulators often perform best on a single core since modern processors cannot effectively exploit fine-grain parallelism.

This work presents *Manticore*: a parallel computer designed to accelerate RTL simulation. Manticore uses a *static bulk-synchronous parallel* (BSP) execution model to eliminate fine-grain synchronization overhead. It relies entirely on a compiler to schedule resources and communication, which is feasible since RTL code contains few divergent execution paths. With static scheduling, communication and synchronization no longer incur runtime overhead, making fine-grain parallelism practical. Moreover, static scheduling dramatically simplifies processor implementation, significantly increasing the number of cores that fit on a chip. Our 225-core FPGA implementation running at 475 MHz outperforms a state-of-the-art RTL simulator running on desktop and server computers in 8 out of 9 benchmarks.

*Equal contributors.
†Work done during EPFL internship.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0394-2/23/03.
https://doi.org/10.1145/3623278.3624750

CCS CONCEPTS

• **Hardware** → **Hardware description languages and compilation**; **Simulation and emulation**; **Testing with distributed and parallel systems**; *Hardware accelerators*; *Software tools for EDA*; • **Computer systems organization** → **Multicore architectures**; **Multiple instruction, multiple data**.

ACM Reference Format:
Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2023. Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3623278.3624750

1 INTRODUCTION

The long-anticipated end of Moore's Law and Dennard Scaling has dramatically increased commercial and academic interest in computational accelerators [4, 11, 15, 20, 21, 24, 31, 35, 38, 41–43, 46, 55]. As with any hardware artifact, accelerators require many iterations of design, debugging, testing, and software development. Detailed hardware simulation is at the heart of this activity, and a simulation's turnaround time and throughput can directly affect designer productivity and product quality.

Designers, however, face a dilemma. Software RTL simulators offer much faster turnaround and better visibility into hardware internals than FPGA prototypes. Simulation, however, runs far slower than hardware, which can be a bottleneck when simulating a large design, running a long execution, or performing extensive testing.

Since the advent of multicore processors, parallelism has been the preferred approach to improve software performance. RTL simulation seems to offer many opportunities to follow such an approach. For example, hardware description languages (HDL) like Verilog or VHDL [19] contain parallel constructs for describing independent hardware components that run in parallel and synchronize only at clock edges. RTL designs comprise many independent computation tasks. The designers, however, want circuits to run at high clock frequencies, which limits the number of gates between clock

edges. Consequently, realistic RTL designs comprise many *tiny* tasks. Modern multicore processors struggle with these fine-grain tasks because synchronization and communication are costly.

This work explores a different approach to increase RTL simulation speed. *Manticore* is a specialized architecture we designed and built for RTL simulation (i.e., a simulator accelerator). It uses the bulk-synchronous parallel (BSP [53]) execution model and static scheduling (i.e., *static BSP*) to eliminate the runtime overheads in communication and synchronization. Like MIT's Raw machine [54], Manticore relies entirely on its compiler to schedule resources and communication. Because RTL code rarely contains long divergent code paths, static scheduling is practical. The scheduled communication and synchronization run without runtime overhead, so fine-grain interactions among cores are efficient. In addition, static scheduling simplifies the Manticore processors, significantly increasing the parallelism possible on a chip.

Manticore's compiler accepts single-clock RTL designs and generates binary code for a Manticore accelerator. Compilation time is comparable to software compilers, offering software development-like turnaround and a fast simulation rate, especially useful for hours- to day-long simulations. We prototyped Manticore on an FPGA, and it outperforms Verilator [48] (the fastest open-source RTL simulator) running on top-of-the-line multicore general-purpose processors despite operating at a fraction of their clock speed. Hardware-accelerated simulation offers a way out of the dilemma posed above by optimizing "time to result." Small experiments and tests can run on a software simulator with rapid turnaround. More extensive experiments and tests can run on Manticore, with slightly slower compile times but much faster execution. And hardware prototypes can be reserved for full-system simulation, operating system bring-up, and software development.

The chief contributions of this work are:

- An application of the static BSP execution model to RTL simulation,
- The Manticore architecture that employs fine-grain parallelism to simulate RTL,
- A compiler that finds parallelism in RTL code and statically schedules it to run effectively on Manticore,
- A high-performance FPGA prototype of Manticore,
- An extensive evaluation comparing and analyzing the performance of Manticore against state-of-the-art software RTL simulation, and
- A demonstration of how to effectively exploit the fine-grain parallelism in RTL simulation.

The paper is organized as follows: §2 introduces RTL simulation taxonomy and how simulation is performed. §3 presents the static BSP execution model. §4 presents Manticore's architecture. §5 discusses a high-performance implementation of Manticore. §6 presents the compilation techniques used to exploit Manticore's hardware. §7 evaluates Manticore's performance and design decisions. §8 discusses limitations and future directions. §9 surveys closely related work. Finally, §10 concludes.

All of Manticore's components (hardware design, verilog frontend, backend compiler) are publicly available with an MIT license[1].

[1] https://github.com/ManticoreRTL

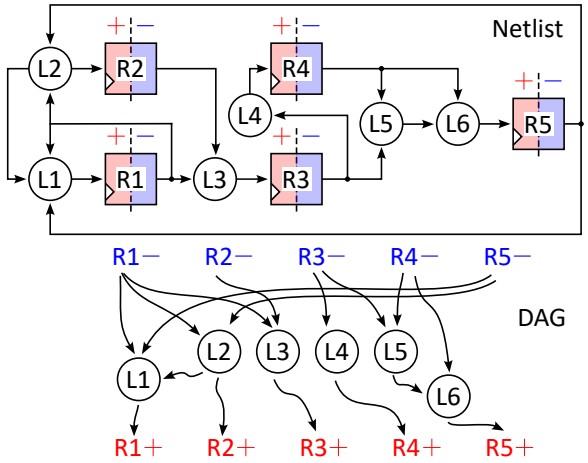


Figure 1: An example single-clock netlist (top) and its DAG representation (bottom). Circles represent gates and rectangles represent registers.

2 BACKGROUND

RTL simulation can be performed in two ways: *timing-accurate*, or *cycle-accurate*. Timing-accurate simulation fully models gate delays by timestamping value changes. By contrast, cycle-accurate simulation captures value changes only at clock edges. Early in the design process, when logic delays are unknown, cycle-accurate and timing-accurate simulations are similar.

Cycle-accurate simulators are implemented in two ways: *event-driven* or *full-cycle*. An event-driven simulator observes signals and *dynamically* schedules operations when values change, avoiding unnecessary re-evaluation of unchanging circuit elements. By contrast, full-cycle simulators use a fixed ahead-of-time schedule to ensure values are computed in the correct order.

This work focuses on full-cycle, cycle-accurate simulation. Full-cycle is generally faster than event-driven simulation despite the redundant evaluations since the cost of monitoring and scheduling events can outweigh the benefit of avoiding unnecessary execution [6].

2.1 Full-cycle simulation

Hardware description languages model circuits as a *netlist*. A netlist is a directed graph whose nodes are circuit cells (gates, registers, and memory banks) and whose edges are the wires connecting them. A netlist graph can be made acyclic by splitting the state nodes (e.g., registers) into a *next* and *current* value. For example, the top part of Fig. 1 contains a netlist in which circles represent gates and rectangles represent registers. The corresponding directed acyclic graph (DAG) at the bottom has the *next* and *current* values denoted by + and −, respectively.

Simulating RTL entails evaluating the netlist DAG while respecting precedence relations. A simulated cycle concludes when all *next* register values have been computed using the *current* register values. The *current* values are then updated from the *next* values,

and the process repeats. The DAG fully expresses the inherent parallelism of an RTL circuit as an evaluator can traverse independent paths in parallel.

3 THE STATIC BSP EXECUTION MODEL

This section describes *Static BSP*, a low-overhead execution model for parallel simulation. It is inspired by Valiant's bulk-synchronous parallel (BSP) execution model [53]. Fig. 2 depicts the components of static BSP. Like the original BSP model, ours consists of a system of networked processors that alternate between phases of local computation and cross-processor communication.

3.1 Runtime Synchronization Freedom

The original BSP model relied on a *runtime barrier* to synchronize processors at the end of communication (before they start a new computation phase). Static BSP replaces this barrier with a *compile-time schedule*. It requires hardware with a *deterministic* interface that permits a compiler to schedule computation and communication. The compiler uses delay operations (e.g., `sleep` in Fig. 2) to ensure all processes start the next phase synchronously.

3.2 Applying Static BSP to RTL Simulation

To parallelize RTL simulation, we partition the netlist DAG (bottom of Fig. 1) into multiple independent graphs by creating a DAG per sink node. The computation in each graph consumes multiple *current* register values and produces exactly one value in a *next* register. The DAGs are independent and can be evaluated in parallel. Once all DAGs are simulated, the newly computed *next* values become DAG inputs, and a new simulation cycle starts.

RTL simulation can be statically analyzed and scheduled because RTL code rarely has long-lived divergent code paths. This enables a conservative yet efficient schedule of code paths while maintaining determinism.

In the context of RTL simulation, we call a complete iteration of the computation *and* communication phases a *virtual cycle* (**Vcycle**). We do so to distinguish between *RTL cycles* (Vcycle) and the *clock cycles* of the processor that is running the simulation.

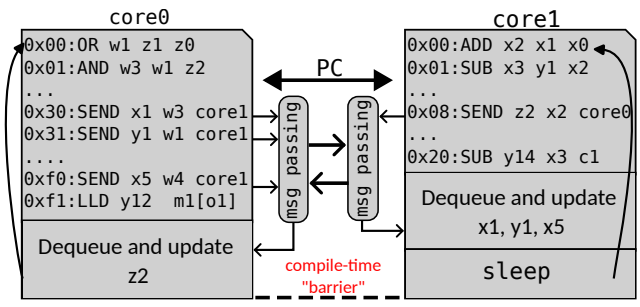


Figure 2: The static BSP execution model. Each core performs a local computation and then sends its result to the cores that need it for the next computation phase. Cores wait (with compiler-inserted `NOp`s) until all communication completes before starting new computation.

4 MANTICORE ARCHITECTURE

This section describes Manticore, whose deterministic runtime behavior satisfies the static BSP's requirements.

4.1 Key Ideas

We now list the salient Manticore features that support deterministic behavior.

- It consists of simple cores that communicate over a statically-scheduled network-on-chip (*NoC*) (Fig. 3).
- Cores communicate through *message passing* since shared memory's dynamic performance and communication makes static scheduling difficult.
- A compiler schedules how messages are routed between cores to ensure deterministic delivery.
- Instructions are *statically partitioned* across cores and stored in fixed-latency, on-chip memories, eliminating frontend stalls so long as each program partition fits a core's instruction memory.
- Manticore replaces branches with *predication* and executes all code paths.
- Each core accesses only the program state stored in its register file and local, fixed-latency scratchpad memory. This memory holds the small (few KiBs) on-chip memories typically found in RTL designs (FIFOs, etc.).
- If the program state does not fit in the scratchpads (e.g., processor caches or memories), a *privileged core* accesses an off-chip DRAM memory using a *global stalling* mechanism to ensure all cores and the NoC remain in lockstep.

4.2 Instruction Set

We briefly describe unconventional aspects of the ISA specific to RTL simulation.

Each core supports 32 programmable *functions*, which execute chains of bitwise logic operations with up to four inputs in a single cycle. E.g., consider the expression:

```
(a & 0xf) | b | (c & 0x3) | (d ^ 0x1)
```

with `a`, `b`, `c`, and `d` being operands[2]. A *single* custom instruction replaces these six instructions (see §6.2). Custom functions are programmed into a core during boot.

The `Expect rs1, rs2, eid` instruction raises an exception `eid` if the values of registers `rs1` and `rs2` differ. Exceptions can invoke services from the host processor (e.g., `$display`). Exceptions, like global memory accesses, stall the execution of all cores and the NoC until they are resolved. Instructions capable of globally stalling the execution are *privileged* and reserved for a single core, which permits an efficient implementation (see §5.3).

Each core has a scratchpad memory (up to 128KiB) for local load and store operations. Loads execute unconditionally, but stores are predicated. Global load and store (predicated) instructions are privileged and access large, off-chip memories using 48-bit addresses. From the perspective of a compiler, both global and local memory access have the same predictable latencies since the long off-chip latency is masked by stalling all cores and the NoC until a memory access completes.

[2]Taken from picoRV32, a multi-cycle RISC-V processor.

The producer of a value initiates communication with a Send instruction, which is the only way cores communicate. The Send `rt, rs, tid` instruction invoked by a core `sid` requests target core `tid` to update its register `rt` with the value of register `rs` from core `sid`. As Fig. 2 illustrates, Sends occur intermixed with computation, but the register updates are delayed until the end of a Vcycle.

5 MICROARCHITECTURE

This section describes Manticore's microarchitecture and its efficient FPGA implementation. We prototyped Manticore on a Xilinx UltraScale+ FPGA (an Alveo U200 datacenter FPGA card). The ideas, however, are general and apply to other FPGAs and ASIC implementations.

Fig. 3 depicts a sample 6-core Manticore grid. Manticore operates as a accelerator for a *host* (e.g., an x86 processor), which loads programs on Manticore and handles exceptions and termination. The host has full access to Manticore's DRAM and communicates with Manticore by reading and writing specific registers.

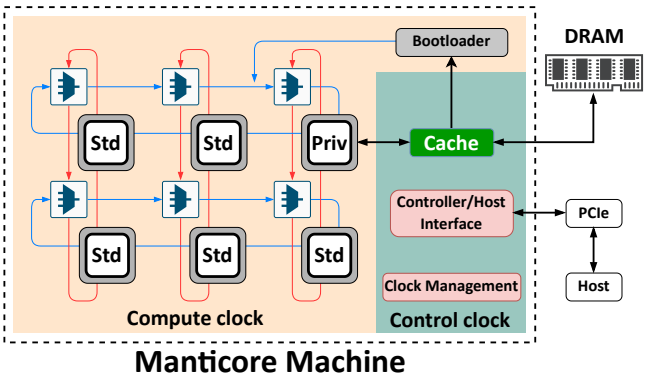


Figure 3: A Manticore grid of processors on a uni-directional 2D torus NoC. The cores and the NoC reside in the *compute* clock domain, while all other components reside in the *control* clock domain. The privileged core is connected to a cache and can access off-chip DRAM.

5.1 Pipeline Implementation

Each core is implemented with a simple 14-stage pipeline. The pipeline is simple because we remove expensive bookkeeping logic (e.g., interlocks and scoreboards) and delegate their function to the compiler. The logical pipeline consists of the usual five stages: fetch, decode, execute, memory access, and writeback. Each stage is internally pipelined to achieve a high clock frequency. A block diagram of the pipeline is in the Appendix (Fig. 15). RTL code contains structures of various bit widths that are typically narrower than a conventional processor's 32-bit word size. Manticore uses a 16-bit datapath to match the width of the FPGA's hard DSP units. This further simplifies the hardware and enables higher clock frequencies.

Instructions are fetched over two cycles from a dedicated instruction memory mapped to a 4096×64 URAM. URAMs are large 36 KiB on-chip memories.

Deep pipelines require a large register file to avoid stalls. Manticore provides a *2048-entry* register file that exposes all registers to the compiler to avoid expensive hardware renaming logic (similar to the Raw machine [54]). We implement the register file using BRAMs, configurable 4.5 KiB on-chip memories that support multiple addressing modes. We use a 2048×17 addressing mode where the lower 16 bits contain the register value, and the most-significant bit contains an *overflow bit* used by wide addition instructions. The size of the register file requires additional pipelining for reads. This makes decoding three stages long. Some instructions can read four values from the register file and write a single result. This requires four read ports and one write port, which BRAMs do not natively support. We use four write-mirrored and identical BRAMs to produce four values simultaneously.

The execute stage consists of two computational units pipelined over four stages. The ALU handles most standard instructions using a hard FPGA DSP. The custom function unit (CFU) consists of a small 32×256 memory made of LUTRAMs. LUTRAMs are FPGA primitives used for shallow memories. A 1-bit 4-input boolean function is canonically defined by the 16 bits of its truth table. Manticore's datapath is 16 bits wide, so we extend this idea to a 16-bit truth table using $16 \times 16 = 256$ bits of memory per function.

Scratchpads are mapped to a URAM, with two cycles to access and one cycle to reshape. We reshape a 4096×64 URAM into a 16384×16 memory by using byte-strobes on the write path and multiplexers on the read path.

5.2 Network-on-Chip

The cores communicate over a uni-directional torus NoC with buffer-less switching and dimension-ordered routing [26]. This design choice reduces routing congestion on the FPGA and supports a high clock frequency.

Switches do not queue messages and immediately route them. A switch drops an input message if the target link is busy. To avoid data loss, the compiler statically schedules communications. Manticore's deterministic execution makes it possible to predict link utilization at each cycle.

Links carry 27 bits of payload, and a few[3] bits to specify the target core address. Messages arriving at a core are queued and received when the core finishes a Vcycle (see Fig. 2). We use the instruction memory and its unused write port to implement the queue and save resources. An incoming message encodes an instruction, which is pushed at the end of the instruction memory. The core subsequently executes it like any other instruction when it reaches it (see Fig. 15 in the Appendix).

5.3 Global Stall

In our current implementation, the privileged core is connected to a 128 KiB direct-mapped, write-allocate, write-back cache backed by a DRAM bank. The cache is implemented using 4 URAMs. Accesses to the cache preemptively stall all cores and the NoC until completed, whether that access is a hit or miss. Therefore, from the compiler's point of view, a global memory access appears to all cores with fixed latency independent of DRAM latency.

[3]Varies based on the grid size, e.g., 8 bits for a 15×15 grid.

Implementing the stall by routing a global signal from the cache to each core would not scale to hundreds of cores. Instead, we take advantage of the FPGA's clock gating primitives to achieve this functionality. All parts of Manticore that operate in strict lockstep (the cores and the NoC) reside in the *compute* clock domain. The rest of the logic that deals with non-determinism reside in the *control* clock domain (see Fig. 3). The two domains are frequency-matched and phase-aligned. The logic in the control domain can *halt* or *resume* the compute clock with a global clock buffer as highlighted in Fig. 3.

We took great care in implementing the clock gating logic to minimize its effect on scalability. For instance, there is no logic delay from the clock enable signal to the clock buffer that receives it. The result is that clock gating logic is nearly independent of the number of cores.

With global clock gating, computation is frozen on a cache request and resumed once completed. The same mechanism is used to stall the compute domain when an exception occurs so that exceptions are precise. Control is then transferred to the host machine, and computation resumes at the host's command.

6 COMPILER

Manticore's hardware was co-designed with its compiler, responsible for extracting parallelism, custom function synthesis, and instruction scheduling. Fig. 4 sketches the compilation process. The compiler operates on two related intermediate representations (IR): netlist and lower assembly. Both use static single-assignment and can be interpreted in software. The lower interpreter is a full-fledged ISA simulator parameterized by the hardware configuration. We used the interpreters extensively to validate the compiler passes.

We derived our Verilog frontend from Yosys's [57]. We extended Yosys to support basic system calls, such as `$display` and `$stop`, required for simulation. After parsing the Verilog input, the frontend performs a few optimizations and emits netlist assembly. Because of the semantics of RTL code, instructions in netlist assembly are unordered and have arbitrary-width operands.

The backend orders the instructions and applies simple optimizations (dead code elimination, constant folding, and common sub-expression elimination). We then transform the netlist assembly

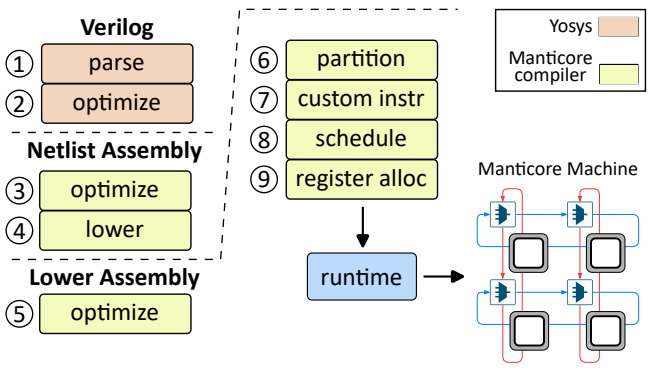


Figure 4: Manticore compiler. Frontend in red, backend in green. A host communicates with the Manticore accelerator through a runtime shown in blue.

instructions into an equivalent sequence of lower assembly instructions whose operands match Manticore's 16-bit data path. Initially, the lower assembly is a monolithic sequence of instructions (a single process). After further optimizations, the compiler partitions the instructions into multiple processes. The compiler then optimizes each process by fusing chains of bitwise logic instructions into custom instructions.

The final steps of compilation are scheduling and register allocation. Scheduling ensures that there are no data hazards in the pipeline by inserting `NOp` instructions to respect data dependencies. In addition, the `Send` instructions must be scheduled to ensure timely message delivery. The compiler then maps virtual registers to machine registers and emits binary code. The binary is then loaded into Manticore over the NoC by a runtime running on a host x86. See Appendix §A.3 for details.

The compiler is 18K lines of Scala. The Yosys Verilog frontend passes are about 2K lines of C++. The runtime is built on top of the Xilinx runtime library (XRT) with about 800 lines of C++ code.

6.1 Extracting Parallelism

Partitioning instructions across the cores is the most critical step to achieving good parallel performance. Despite the absence of runtime synchronization in Manticore, data movement is still costly and excessive communication will limit scalability. Our parallelization algorithm is aware of this cost and attempts to reduce NoC traffic while distributing work across cores such that each core executes roughly the same number of instructions.

The compiler parallelizes a single monolithic assembly process in two steps: (1) *Split* the monolithic process into a maximal number of tiny processes. (2) *Merge* the split processes so that the total number of processes does not exceed the number of available cores.

Splitting follows the approach described in §3.2 and illustrated in Fig. 1. The compiler first creates a DAG representing data dependencies in the monolithic process. It then uses a backward traversal to partition all nodes reachable from a data sink into an independent smaller process. In creating the parallel processes, the compiler ensures that instructions that access the same memory region (e.g., an unpacked array in Verilog) end up in the same process to avoid moving large amounts of data every Vcycle. Additionally, all privileged instructions must execute in the same process. Partitioning can duplicate DAG nodes across multiple cores, maximizing parallelism at the expense of increased computation.

If we view the maximal set of split processes as a graph whose nodes denote processes and edges denote communication, then merging is a graph partitioning problem. Existing partitioning tools [28, 47] assume a *linear* cost function, so merging two nodes would add their weight or cost. However, optimizations such as data sharing and duplicate code elimination make merging non-linear, so we required a heuristic algorithm.

The compiler estimates the execution time of a process as the total number of instructions it executes, including `Send`s, but excluding the `NOp`s used to schedule data hazards and received messages. A vital goal of merging processes is to avoid overloaded cores (i.e., forming stragglers) by equalizing the execution time of all processes. The compiler iteratively picks two merge candidates that minimize the increase in merged execution time. It starts from the

process with the shortest execution time and merges it with another process with which it communicates. Intuitively, by starting from the smallest processes and constructing larger ones, we can balance the execution time of the processes and simultaneously reduce communication (hence avoiding network contention).

Merging can continue even after reaching the number of available cores because it can reduce execution time. For instance, merging processes `p1` and `p2` that read a value produced by process `p3` could lower the execution time of `p3` because it executes one fewer `Send` instruction. Furthermore, since splitting the DAGs may have duplicate code, common sub-expression elimination in a merged process might reduce the number of instructions.

After the merge, the compiler assigns the process that contains privileged instructions to the privileged core.

6.2 Custom Function Synthesis

Manticore's instructions all have the same latency and programs are branch-free, so shorter programs are faster than longer ones. Custom function synthesis is the process of collapsing long chains of bitwise logic operations common in RTL simulation into a shallow sequence of 4-input custom functions. This process is conducted on each partitioned process independently. Instruction fusion borrows ideas from classical minimum-area, bit-level logic synthesis and applies them to word-level programs.

We start from a process' dependence graph and prune all non-logic vertices. This leaves us with a set of connected components, each containing only logic operations. We exhaustively extract all 4-input *maximum fanout-free cones* (MFFC) from each component using cut enumeration [17]. An MFFC is a tree rooted at a terminal instruction such that no intermediate result is used by an instruction outside the cone. Multiple MFFCs can represent the same function and differ only in their representation. We use logic equivalence checking to group all MFFCs by the function they compute.

Finally, we use a mixed-integer linear programming (MILP) formulation to maximize instruction savings by selecting the best set of non-overlapping MFFCs, while considering that some MFFCs are used at multiple places and yield more savings. Each MFFC is then replaced with a single custom function. The MFFCs' truth tables are used to configure each core's CFU at boot time.

6.3 Scheduling, Routing, and Register Allocation

The compiler uses a simple list-scheduling algorithm to schedule data hazards. It performs an abstract cycle-accurate simulation of one Vcycle using a model of a core's pipeline and the NoC. An instruction is scheduled when its predecessors (in the DAG) are scheduled and executed. Additionally, a `Send` instruction can be issued only when it will not collide with any other messages on its path. If we cannot issue an instruction in a scheduling step, the compiler delays it with a `NOp` instruction.

Because of the large register file, a simple linear-scan register allocator works well with practically no spills.

Furthermore, we optimize redundant register moves by allocating the same machine register to both the current and next values of an RTL register (e.g., in Fig. 1, `+` and `-` values use the same machine register when possible) [56].

7 EVALUATION

This section evaluates Manticore along several dimensions.

7.1 Fine-Grained Parallel Simulation

We first explore the motivation for a new architecture by studying the limits to fine-grained parallelism in RTL simulation on a general-purpose processor. We use a simple model of a simulator to find the relationship between simulation speed and computation granularity. In practice, simulator speed depends on the RTL design and details of the simulator's partitioning, optimization, and runtime. A fully accurate model is unnecessary if a simplified model offers an upper bound on any system, which we achieve with two simplifications:

- Ignore the data transfer among cores and focus exclusively on the synchronization *necessary* to coordinate data movement. BSP requires two synchronization points (barriers) per RTL cycle: one at the end of computation and another at the end of communication. These are the minimum synchronization needed to simulate an RTL cycle correctly. Verilator (our baseline RTL simulator; described in §7.3) also uses two synchronization points as a rendezvous for all tasks at the clock transitions in a cycle.
- As in full-cycle simulation, assume the number of machine instructions required to simulate one RTL cycle is independent of a design's state. This assumption also removes stragglers as a concern.

```
1  // N is the total number of instructions per cycle.
2  // P is the number of parallel threads.
3  // B is an arrive-await barrier.
4  // C is the number of RTL cycles to simulate.
5  // nonOpt() is an unoptimizable sequence of instructions:
6  //     a ^= (a+1); b ^= (b+1); c ^= (c+1); d ^= (d+1);
7  // Independent ops. to avoid read-after-write stalls.
8  for P parallel threads:
9      let localInstr be (N / P / instrInWhileLoop)
10     for C iterations:
11         // mock computation
12         while(localInstr > 0)
13             localInstr -= 1
14             nonOpt()
15         B.wait()
16         // mock zero-cost communication
17         B.wait()
```

Listing 1: Model of parallel simulation.

7.1.1 First Model. Listing 1 contains the initial model. The inner loop executes a set of independent arithmetic instructions to model the simulator's computation of an RTL cycle. The barriers at the end of this computation are necessary to synchronize the communication of newly computed values. These barriers execute when the model runs and contribute to its runtime cost. We measure the simulation rate (in kHz) in a strong-scaling experiment that increases the number of threads while keeping the total work constant. The dashed curves in Fig. 5 report the rates on desktop and server x86 systems (details in Table 2).

7.1.2 Second Model. Model 1 does not fully capture the behavior of a simulator since the `while` loop has a small instruction footprint that fits in an i-cache. RTL models are typically larger and incur cache misses. The fraction of a model that runs on a processor depends on the number of threads; hence, the i-cache performance

depends on parallelism. We fully unroll the `while` loop to capture this effect. The differences between the dashed and solid lines in Fig. 5 show that simulation speed decreases significantly because of cache pressure.

7.1.3 Discussion. This simple model corresponds to Verilator's performance (our baseline RTL simulation; described in §7.3). §7.6 contains measurements of Verilator running nine benchmarks. Fig. 6 shows that Verilator achieves a maximum speedup of 4× for the two benchmarks with the largest step size (see Table 3) and runs slower with multiple threads for smaller benchmarks.

Looking in detail, Fig. 5 identifies three regions of parallel operation:

- Small circuits (at most a few thousand instructions) running with very fine-grained parallelism. Each clock cycle is a small computation so that serial simulation can reach a few MHz. Parallel simulation introduces synchronization every 100–1000 instructions, and its cost causes a steep drop in performance between 1 and 2 processors in the top graphs in Fig. 5.
- As the size of a circuit increases, additional processors usually improve performance (middle graphs in Fig. 5). In this region, synchronization occurs every 2,000–20,000 instructions. Note that the performance benefits are limited, and eventually, the synchronization costs outweigh the benefits of splitting the computation further and the performance decreases. This region emphasizes the importance of serial performance; the EPYC processor lags behind the desktop processor, even with its many cores and large caches.
- Finally, with hundreds of thousands of instructions in an RTL cycle simulation, parallel execution is beneficial (bottom graphs in Fig. 5) since synchronization is infrequent. However, the overall rate is low because each cycle is costly. Many cores are needed to push the simulation speed into the 100 kHz range, and the simulation benefits from servers' higher core counts.

The figure also displays numerous inflection points where simulation performance decreases with increasing resources. These inflection points are particularly prominent in fine- and medium-grain simulation. They occur because additional processors reduce the work-to-synchronization ratio and increase the cost of a barrier.

The table in Fig. 5 reports the maximum speedups. Larger designs offer increased opportunities for speedup. The second model's speedups are better since its numerator (serial execution) suffers more from i-cache misses than the first model's smaller kernels. One data point (i7, 3.5M) shows that cache effects can produce super-linear improvement.

Manticore's unconventional design avoids these challenges and can scale its performance over hundreds of cores. The current prototype allows at most 4096 machine cycles between synchronization points (the instruction memory size). This puts Manticore in the top region of Fig. 5, where performance scalability is infeasible on a general-purpose computer. If we are to improve simulation performance through parallelism, adding more cores to general-purpose processors will also result in partitioned workloads that falls in the top region of Fig. 5.

Manticore, however, is limited by its total number of cores and clock frequency. To match the serial performance of a 4.6–4.9 GHz desktop processor, Manticore must overcome a 10× reduction in clock speed. Furthermore, general-purpose computers can execute 1–2.5 instructions-per-cycle (IPC). Manticore's simple processors execute a single instruction per cycle, have a narrower datapath, and support only simple instructions. Manticore can match the desktop processor's serial performance only if it can achieve a performance improvement of at least 10–25× by employing parallelism effectively.

7.2 Manticore FPGA

We first evaluate the physical design of Manticore's FPGA implementation. Table 1 reports the frequency achieved for various Manticore grid sizes. Smaller grids can operate at *very* high speeds (close to 500MHz). There is abrupt degradation at the 12×12 grid, explainable by the FPGA's physical layout. The U200's rectangular floorplan is divided into (1) a static shell connected to the PCIe bus and (2) a user logic region. The vendor immovably placed the shell at the center-right side of the chip. User logic has a C-shaped floorplan (Appendix §A.5 contains die-shots). With fewer than 160 cores, Manticore fits at the top of the chip, unperturbed by the shell. Additional cores surround the shell, which complicates timing closure. We significantly improved the quality of results by guiding the place-and-route tool through the floorplanning of designs with more than 160 cores. Details are described in the Appendix (§A.5).

Each core requires less than 0.021% of the U200's resources. The quantity of URAMs limits the number of cores to 398[4] [23] (the Appendix contains details).

Grid	8×8	10×10	12×12	15×15	16×16
Auto	500	485	480	395	180
Guided	–	–	500	475	450

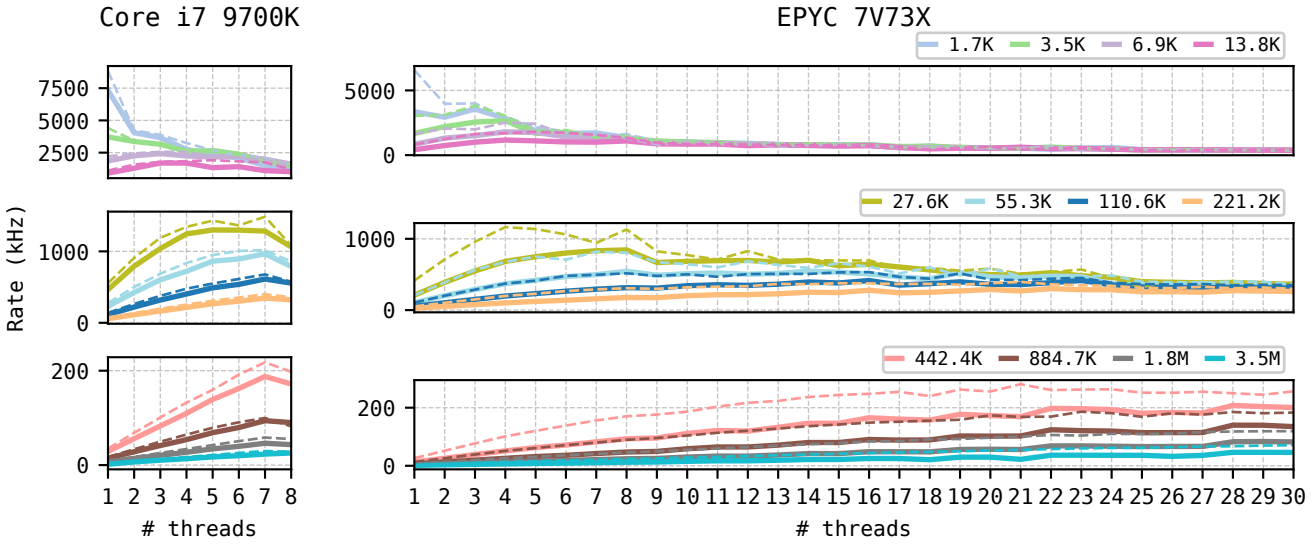
Table 1: Clock frequency (MHz) achieved on U200 using automatic and guided floorplanning.

7.3 Verilator Simulator

We use Verilator [48] as our baseline RTL simulator. Verilator is an open-source, full-cycle simulator widely used by academia and industry. It performs full-cycle simulation and is widely believed to run faster than commercial and other open-source simulators [48]. Verilator generates C++ code from an abstract syntax tree of inlined RTL code and highly optimizes it with branch prediction hints, short-circuitable branch conditions, and memory prefetch directives.

Verilator parallelizes RTL simulation by partitioning its DAG into *macro-tasks*, atomic units of work that run asynchronously. It then combines these tasks into larger units appropriate for multicore processors. Initially, each DAG node comprises its own macro-task. Verilator uses Sarkar's algorithm [45] to increase task granularity by combining macro-tasks that share an edge into a single task.

[4] Out of 800 available URAMs, the cache requires four.



granularity		1.7K	3.5K	6.9K	13.8K	27.6K	55.3K	110.6K	221.2K	442.4K	884.7K	1.8M	3.5M
i7	**model 1**	1.0	1.0	1.2	1.7	2.7	3.7	4.9	5.8	6.3	5.8	6.7	6.9
	model 2	1.0	1.0	1.3	1.8	2.8	4.2	5.3	6.0	6.5	6.6	6.6	14.5
epyc	**model 1**	1.0	1.3	1.5	2.2	2.8	3.9	5.1	7.6	10.7	14.2	18.2	21.7
	model 2	1.1	1.6	2.1	2.8	4.0	5.2	7.9	11.5	15.9	21.4	25.6	28.4

Figure 5: Measured simulated model speed on a desktop (left) and server (right). Dashed lines model only synchronization cost (model 1). Solid lines also include i-cache pressure (model 2). Each curve is labeled by the number of instructions in a simulation step. The table shows the maximum speedup of each model.

Combining nodes eliminates the communication of values between cores, but it can increase the critical path of the macro-task graph since the combined nodes execute sequentially. Verilator's algorithm merges the macro-task that yields the smallest increase in the critical path length. It does so until it reaches a heuristic threshold for the critical path. Verilator then statically assigns macro-tasks to a thread pool. At runtime, a macro-task starts running after its preceding macro-tasks complete execution. Atomic fetch-and-add operations (i.e., spin-locks) synchronize the macro-tasks, and barriers synchronize the tasks at the clock edges.

Verilator's parallel execution is not BSP since it uses fine-grain synchronization between tasks. However, in simulating a clock cycle, Verilator uses two synchronization points (final macro-tasks) as a rendezvous for all tasks, similar to the barriers in the model presented above in §7.1 and Manticore.

7.4 Test Environment

We used Verilator v5.006. Verilator recently added support for multiple clock domains and timing. Since Manticore does not yet support these, we disable timing in Verilator to avoid penalizing its performance. We evaluate Verilator's performance on an overclocked desktop and two servers with high core counts. Table 2 summarizes the key characteristics of the hardware platforms.

7.5 Benchmarks

We evaluate Manticore's performance using nine RTL workloads (the benchmarks are wrapped in simple, assertion-based Verilog test drivers):

- **bc** is a bitcoin miner [36].
- **mm** is a 16×16 integer matrix-matrix multiplier.
- **cgra** is a latency-insensitive, coarse-grained reconfigurable array of 64 floating-point processing elements.
- **vta** is an ML accelerator [33]. We use a larger[5] spatial implementation as the default configuration was too small to benefit from hardware acceleration. We also divide buffer sizes by 4 to fit in Manticore's scratchpads.
- **rv32r** consists of 16 in-order, pipelined RISC-V processors [29] communicating over a ring network.
- **jpeg** is a pipelined JPEG decoder [52].
- **blur** is a stencil computation accelerator [16].
- **mc** is a Monte-Carlo simulation stock option price evolution predictor with fixed-point arithmetic [50].
- **noc** is a 2D 4×4 uni-directional torus network-on-chip with wormhole routing and four virtual channels.

[5] `blockIn=64` and `blockOut=64` instead of 16.

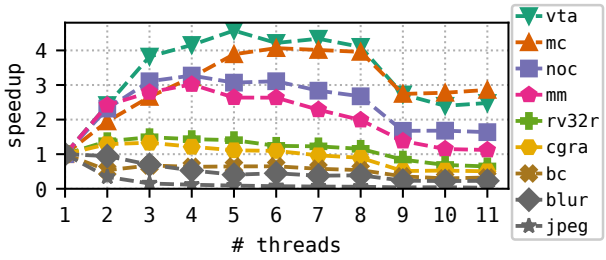


Figure 6: Verilator parallel scaling on EPYC 7V73X.

The benchmarks were sized to ensure their state fit in the Manticore on-chip scratchpads, so the compiler could accurately predict performance.

	Baseline (x86)			Manticore
	Verilator v5.006 (Feb 2023)			
HW	i7-9700K	Xeon 8272CL [2]	EPYC 7V73X [2]	Alveo U200
# cores	8	32	120	225
GHz	4.6–4.9	2.5–3.4	2.2–3.5	0.475
MiB	14.5	105.5	259.6	18.45
Date	Q4 2018	Q4 2019	Q1 2022	-

Table 2: Hardware platforms. # cores, GHz, MiB, and Date denote physical core count, clock frequency, SRAM capacity (e.g., cache), and release date. SRAM capacities for x86 are computed through `lscpu`.

7.6 Performance Comparison

First, we used the benchmarks to compare the Manticore prototype with Verilator. We disabled waveform dumps and unnecessary printing and enabled all optimizations in both Verilator (i.e., `-O3`) and Manticore (e.g., custom functions). We run each simulation for millions to billions of cycles to capture steady-state performance. Table 3 summarizes the simulation speeds achieved by Manticore and Verilator.

7.6.1 Verilator. We report both serial (**S**) and multithreaded (**MT**) simulation rates separately for each hardware platform. Multithreaded Verilator improves performance by up to 3.9× and 4.6× on desktop and server processors, respectively. Multithreading could not improve performance on the smaller benchmarks (e.g., **bc** and **jpeg**). For example, Fig. 6 shows the EPYC processor's scaling trends. At eight processors, all benchmarks have reached their scalability limit. Given the number of instructions in each step of the benchmarks, these results accord with the model discussed above in §7.1.

7.6.2 Manticore. The bottom half of Table 3 reports the simulation rates on a 475 MHz 225-core Manticore. It also reports speedups relative to Verilator's serial (×**S**) and multithreaded (×**MT**) performance. Manticore is consistently faster than Verilator, except for **jpeg**. This benchmark has the highest simulation rate in Verilator and the lowest in Manticore. The **jpeg** benchmark contains sizeable sequential data dependencies that cannot be parallelized[6]. Manticore's slow sequential performance hurts us on this serial benchmark. Parallelism improves **jpeg**'s single-core performance by only ≈17%. This marginal improvement cannot compensate for the single-core disparity between Manticore and x86.

Fig. 7 analyzes Manticore's scalability. The speedup numbers are predicted by Manticore's compiler instead of actual execution, since the compiler can accurately count cycles in the absence of off-chip memory accesses. The compiler reports a *virtual critical-path length (VCPL)*, the total number of instructions (including `NOps`) in the slowest core. VCPL is the number of Manticore machine cycles (i.e., FPGA cycles) required to simulate one RTL cycle. We consider the single-core VCPL as the baseline, however, on the prototype, single-core execution is, for most benchmarks, impossible since there is not enough space in a single core's instruction memory.

We see that Manticore continues to improve performance as the number of processors increases to 200–300. Unfortunately, this performance gain through parallelism must be weighed against the single-core/thread performance disparity between an x86 and Manticore. In other words, a large fraction of the gain goes into making up for the loss in single-core performance. However, the measurements demonstrate that, with appropriate architectural and

[6]Huffman table lookup is the bottleneck.

			vta	**mc**	**noc**	**mm**	**rv32r**	**cgra**	**bc**	**blur**	**jpeg**	***Geomean***
# instr. (k)			169	148	88	74	43	38	20	12	3	
# cycles			1M	1M	1M	1M	1M	1M	2M	5M	1B	
Verilator (kHz)	**i7**	S	41.3	33.9	41.4	43.9	96.6	152.0	599.0	726.7	4246	
		MT	160.2	127.2	80.5	83.0	141.8	146.2	354.4	362.0	700.7	
		×self	3.9	3.8	1.9	1.9	1.5	0.97	0.6	0.5	0.2	***1.19***
	xeon	S	32.4	26.6	37.1	34.7	97.3	136.8	462.7	532.6	3233	
		MT	94.9	68.9	41.5	52.3	73.3	74.3	190.6	186.1	590.6	
		×self	2.9	2.6	1.1	1.5	0.8	0.5	0.4	0.3	0.2	***0.79***
	epyc	S	32.1	29.7	32.4	31.6	109.2	126.0	550.2	430.5	3627	
		MT	146.9	120.8	106.0	95.2	162.7	167.8	370.6	406.9	1239	
		×self	4.6	4.1	3.3	3.0	1.5	1.3	0.7	0.9	0.3	***1.60***
Manticore (kHz)	225-core		278.1	423.0	293.6	567.5	221.0	421.5	1562	1015	214.2	
	i7	×S	6.7	12.5	7.1	12.9	2.3	2.8	2.6	1.4	0.05	***2.75***
		×MT	1.7	3.3	3.6	6.8	1.6	2.9	4.4	2.8	0.31	***2.38***
	xeon	×S	8.6	15.9	7.9	16.3	2.3	3.1	3.4	1.9	0.07	***3.37***
		×MT	2.9	6.1	7.1	10.8	3.0	5.7	8.2	5.5	0.36	***4.16***
	epyc	×S	8.7	14.2	9.1	18.0	2.0	3.3	2.8	2.4	0.06	***3.35***
		×MT	1.9	3.5	2.8	6.0	1.4	2.5	4.2	2.5	0.17	***2.07***

Table 3: Verilator and Manticore simulation performance. # instr. is the average number of x86 instructions (1000s) to simulate one RTL cycle. # cycles is the number of RTL cycles simulated to measure the simulation rate. For Verilator, the S and MT rows report the serial and multithreaded simulation performance in kHz. ×self is multithreaded speedup (wrt serial). For Manticore, we report simulation rates on a 225-core configuration, along with the speedup relative to the serial (×S) and multithreaded (×MT) runs of Verilator.

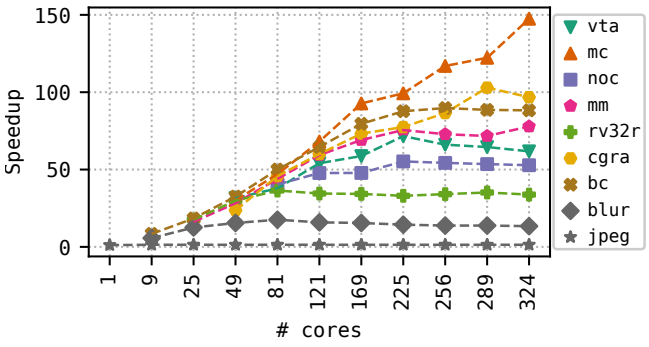


Figure 7: Manticore's multicore scaling.

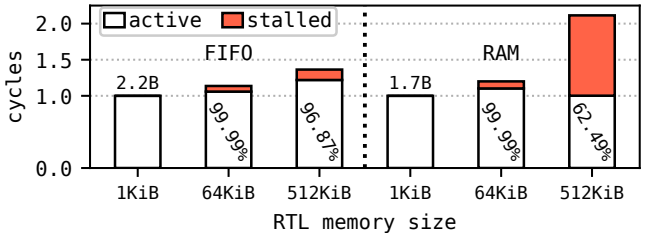


Figure 8: Number of machine cycles (lower is better) simulating a FIFO (left) and a RAM (right). Numbers are normalized to the cycles needed to run the 1 KiB design. Cache hit rate is denoted inside each bar.

compiler support for fine-grain parallelism, we can reach simulation speeds that are *unattainable* on a general-purpose architecture.

Finally, Manticore is not immune to Amdahl's law. If there is insufficient parallelism in the workload, then Manticore's scaling plateaus. Depending on the RTL design, this may happen early (**jpeg**) or late (**mc**).

7.7 Global Stall

We evaluate the cost of going off-chip with two RTL microbenchmarks running on a 1×1 Manticore grid at 500 MHz: (1) a FIFO, and (2) a RAM. The FIFO and RAM were sized at 1 KiB, 64 KiB, and 512 KiB. The FIFO reads/writes its memory sequentially, whereas the RAM accesses its memory with pseudo-random addresses (using a simple XOR-shift-128 generator). Each program runs for 16Mi Vcycles performing a load and store operation per Vcycle. We use hardware performance counters to log the total number of cycles, stalled cycles, cache hits, and cache misses. The 1 KiB configuration is a baseline for each microbenchmark since this memory fits in the scratchpad and incurs no global stalls. The 64 KiB represents a middle point where the state does not fit in the scratchpad but is entirely contained in the 128 KiB cache. Finally, the 512 KiB configuration corresponds to the scenario where the state is spread between the on-chip cache and off-chip DRAM. Fig. 8 demonstrates that large FIFOs have a high hit rate and are not stall-limited (i.e., FIFOs have excellent spatial locality). By contrast, randomly accessed RAMs run slower as the number of off-chip accesses increases. Finally, we observe that cache accesses come at a cost even if they hit since we conservatively stall the execution on every access.

7.8 Compiler Optimization

This section evaluates the compiler optimizations.

7.8.1 Communication-Aware Partitioning. The balanced partitioning algorithm (B) described in §6.1 merges the split processes while keeping communication costs low. As a baseline, we compare it against communication-oblivious, longest processing-time first partitioning (L) to observe the benefits of modeling communication. Both algorithms are heuristic and use the same cost estimation method but differ in their merge strategy. Furthermore, both algorithms are oblivious to the effects of instruction scheduling (after partitioning) as neither accounts for the NOps for data hazards and NoC contention.

Fig. 9 compares the two approaches for a 15×15 Manticore grid, with VPCL normalized to that of L. We divided the VCPL into the fraction of cycles in the straggler spent computing (compute), sending messages (send), or doing nothing (NOp). Modeling communication is beneficial as B significantly reduces the overall number of Sends (see Table 4), reduces the number of NOps in the straggler, and generally outperforms (except for **vta**) the communication-oblivious algorithm (L) while using fewer cores. The quality of partitioning significantly affects performance, as evident with **bc** and **mm**.

×1000	mm	mc	vta	noc	cgra	rv32r	bc	blur	jpeg
L	23.3	23.6	13.6	25.6	18.9	16.9	7.7	5.0	1.0
B	8.5	3.9	9.8	16.6	7.4	2.8	3.1	2.7	0.1
%	-63.6	-83.5	-28.0	-35.3	-60.6	-83.3	-59.5	-47.1	-94.1

Table 4: Send instructions (1000s) produced by longest processing-time first partitioning (*L*) and balanced partitioning (*B*).

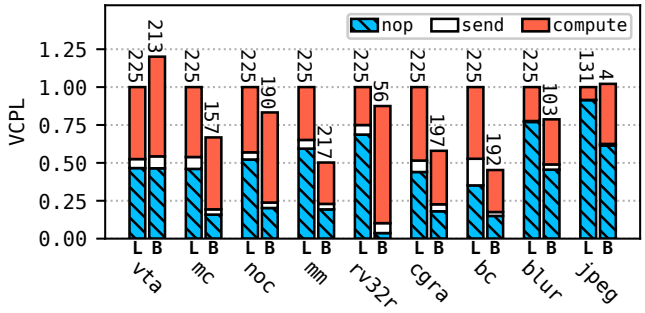


Figure 9: Comparison of the communication-oblivious, longest processing-time first algorithm (*L*) and the communication-aware algorithm (*B*) from §6.1 for a 15×15 grid. VCPL is normalized to the VCPL of L. The numbers above each bar are the number of cores.

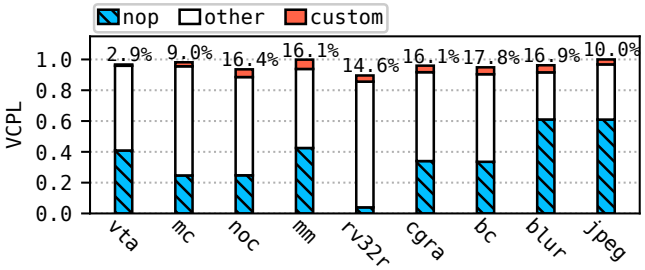


Figure 10: Savings in Vcycle due to custom instructions. The Vcycle is divided into three instruction types and normalized to not using custom functions. The numbers above each bar represent the reduction in non-NOp instructions over all cores.

7.8.2 Custom Instructions. We initially proposed custom instructions to compensate for the lack of instruction-level parallelism in Manticore's simple processors by exploiting bit-level parallelism seemingly abundant in RTL. Fig. 10 shows the VCPL of each benchmark normalized to the VCPL without custom instructions. The VCPL is divided into custom instructions, NOps, and other instructions. The numbers above each bar show the reduction in the total number of instructions over *all* cores (excluding NOps). This reduction is 2.9–17.8%, yet the VCPL (end-to-end) reduction is less than 10% for all benchmarks. Custom instructions reduce the *total* instruction count but may not reduce the path length of the straggler (e.g., in **jpeg**). Their small benefit comes with a small cost of one BRAM and tens of LUTs per core. Eliminating the custom instructions would not enable larger Manticore grids since the URAMs are the limiting resource.

7.8.3 Compile Time. The Manticore compiler is a prototype built in Scala for robustness. Its compile times can be several minutes (max. 16m). By contrast, Verilator compilations usually take less than a minute.

Despite its compilation time, Manticore offers a software development-like experience for longer simulations. For example, simulating 10B cycles of the **vta** on Manticore takes about 10 hours and 17 hours on the i7. In many cases, the extra compilation times are more than compensated by the increased speed.

7.9 Cost Analysis

For completeness, we provide a brief cost analysis using prices from Microsoft Azure. We estimate the cost of running a few billion simulation cycles in the cloud. Table 5 shows the Azure instances used in this analysis. We use the **D2 v4** instance with two virtual CPUs (vCPU) for serial simulation. For multithreaded simulation with Verilator, we use the **D16 v4** instance with sixteen vCPUs. Furthermore, we also consider the **HB120rs v3** instance as it lists RTL simulation as a use case. Renting individual cores on this instance is impossible; therefore, we consider this instance type for only parallel simulation. Unfortunately, renting an FPGA with a single vCPU in Azure is also impossible. The smallest instance is the **NP10s** with one Alveo U250 FPGA board and ten vCPUs, which makes the FPGA instance relatively expensive since we also pay for the unused cores.

All simulations finish in less than an hour for runs shorter than one billion RTL cycles. With hourly pricing and Verilator's sublinear speedup, serial execution would be the least expensive, followed by multithreaded D16 and Manticore, and finally the HB-series. However, the cost differences are small (few dollars at most). More realistically, we consider 1 and 10-billion cycle multiple-hour simulations by estimating the execution time using the simulation rates from Table 3 and then rounding to the next hour (Table 6). With longer runs, Manticore, in some cases, offers a lower cost than D2 and D16, despite its 2–18× higher base cost and unused resources.

Far more important, however, is the vast disparity in run duration. For the ten billion RTL cycle runs, Manticore finishes all of them in a long workday (13 hours). Multithreaded simulation requires up to two or more full days, while serial simulation can take most of a week. The productivity gain from several simulation runs per day dwarfs the minor cost savings from using small machines.

8 LIMITATIONS AND FUTURE WORK

Manticore explores providing architectural support to accelerate RTL simulation by using fine-grain parallelism. This paper focused

Instance	$/hour	Simulation
D2 v3 / Xeon 8272CL 2× vCPU	0.115	serial
D16 v4 / Xeon 8272CL 16× vCPU	0.92	multithreaded
HB120rs v3 / EPYC 7V73X 120× vCPU	4.68	multithreaded
NP10s / Alveo U250 + 10× vCPU	2.145	Manticore

Table 5: Hourly cost of Microsoft Azure instances [1].

			vta	mc	noc	mm	rv32r	cgra	bc	blur	jpeg
1B	D2	h	**8.58**	**10.45**	7.48	8.00	2.85	2.03	0.60	0.52	0.09
		$	**1.04**	**1.27**	**0.92**	**0.92**	**0.35**	**0.35**	**0.12**	**0.12**	**0.12**
	D16	h	2.93	4.03	6.70	5.31	2.85	2.03	0.60	0.52	0.09
		$	2.76	4.60	6.44	5.52	2.76	2.76	0.92	0.92	0.92
	HB	h	1.89	2.30	2.62	2.92	1.71	1.66	0.50	0.52	0.08
		$	9.36	14.04	14.04	14.04	9.36	9.36	4.68	4.68	4.68
	NP	h	1.00	0.66	0.95	0.49	1.26	0.66	0.18	0.27	1.30
		$	2.15	2.15	2.15	2.15	4.29	2.15	2.15	2.15	4.29
10B	D2	h	**85.83**	**104.52**	**74.77**	**79.96**	**28.54**	**20.30**	6.00	5.22	0.86
		$	**9.89**	**12.08**	**8.62**	**9.20**	**3.33**	**2.42**	**0.81**	**0.69**	**0.12**
	D16	h	**29.27**	**40.31**	**66.97**	**53.08**	**28.54**	**20.30**	6.00	5.22	0.86
		$	27.60	37.72	61.64	49.68	26.68	19.32	6.44	5.52	0.92
	HB	h	**18.91**	**22.99**	**26.21**	**29.17**	**17.07**	**16.56**	5.05	5.22	0.77
		$	88.92	107.64	126.36	140.40	84.24	79.56	28.08	28.08	4.68
	NP	h	**9.99**	6.57	**9.46**	4.89	**12.57**	6.59	1.78	2.74	**12.97**
		$	21.45	15.02	21.45	10.72	27.89	15.02	4.29	6.44	27.89

Table 6: Simulation cost using Microsoft Azure prices. Estimated runtime in hours (h) and cost ($). Embolden hours exceed one workday (8 hours). The lowest price is emboldened also.

on the technical aspects of our approach, which is still a prototype, not a complete "tool". Nevertheless, Table 3 shows a clear performance advantage of the Manticore prototype over a highly optimized software simulator for many examples. At this maturity level, Manticore is not a replacement for Verilator or other simulators. Much work is needed to bring Manticore to the same level of usability as Verilator, which has enjoyed more than a decade of active development.

Specifically, Manticore supports only single-clock designs and does not support most of SystemVerilog. Advanced language features, such as event control, are necessary for a complete simulator, especially for writing complicated test benches. Accurate timing control (i.e., not cycle-accurate) is incompatible with our approach and would be challenging to retrofit. However, multiple RTL clock domains could be supported by tracking clock activations independently at each core and conditionally enabling RTL clock domains in all cores.

Waveform debugging is an essential tool in a digital designer's arsenal. We have an initial design of hardware support for out-of-band waveform collection, but we leave its evaluation for future work.

Current Manticore compile times are longer than a conventional compiler. This is not an inherent limitation but a byproduct of building a research compiler that allows us to explore alternatives rather than a fast compiler. Nevertheless, the current compiler offers a faster time-to-result than parallel software simulation for even hour-long simulations.

Currently, most compilation time is spent partitioning the DAG. Partitioning is necessary for parallel RTL simulation, irrespective of the target hardware (e.g., x86 or Manticore). The higher degree of parallelism on Manticore makes partitioning slightly more expensive. We could close the gap between Manticore's and Verilator's compile times with some engineering effort. In addition, algorithms from research on high-quality, low-complexity partitioning could help in this step [10, 18, 30, 44, 58, 60].

Early in the project, we decided to build an FPGA prototype since fabricating an ASIC was not affordable, and simply simulating a Manticore processor would yield less information than constructing an implementation.

The FPGA implementation, however, is limited in its total number of cores and clock frequency. To match the serial performance of an x86 processor, Manticore must exploit parallelism to overcome a 10–25× performance loss due to its lower clock speed and IPC.

In addition to clock frequency, an FPGA has limited SRAM capacity. Our prototype can simulate up to ≈ 900k instructions (4096 instructions in each of 225 cores) with about 14.4 MiB SRAM for data and instruction. Modern accelerator chips contain 100s MiB of SRAM [3, 4, 25, 51], and hence an ASIC implementation of Manticore could easily avoid these limitations.

9 RELATED WORK

9.1 FPGA Prototypes and Emulation Platforms

FPGA prototypes achieve interactive simulation speeds by mapping RTL circuits to *gates* on an FPGA. Prototypes can run full software stacks for trillions of clock cycles but require significant engineering effort and lack visibility. FireSim [27] is an open-source FPGA prototyping platform, widely used as an *architectural* simulator for exploring RISC-V designs at datacenter-scale using cloud FPGAs. *Emulation platforms* are RTL simulators for very large designs [40]. They provide excellent visibility into hardware state by mapping RTL circuits to *instructions* that run on a processor grid. Interconnecting multiple custom processor grids (in a rack) generally greatly increases simulation capacity. However, commercial emulation platforms cost millions of dollars [22].

Although Manticore is implemented on an FPGA, its simulation runs in software (a program running on Manticore) rather than being mapped to an FPGA. Consequently, Manticore's compile times are a few minutes, whereas FPGA prototypes take hours to days to compile. Manticore is a first step towards an open-source alternative to commercial emulation platforms.

9.2 Parallel RTL simulation

There is considerable research on accelerating RTL simulation using parallelism, especially GPUs. Much of this work demonstrates significant speedups relative to commercial *event-driven* simulators. By contrast, Manticore is a *full-cycle* simulator, so it is not comparable to these systems. Most of this work focused on reducing the runtime overhead of monitoring value changes, for example, GCS [12–14] or Qian and Deng [39]. They improved simulation rates by orders of magnitude, to ≈ 5–37 kHz, over single-threaded commercial simulators. Manticore operates at rates exceeding 115 kHz (see Table 3).

RTLFlow [32] is a GPU-accelerated RTL simulator that exploits stimulus-level parallelism to speed up simulation by running many independent simulations on a GPU. RTLFlow improves execution speed by up to 40× over Verilator for many stimuli, but it runs an order magnitude slower than Verilator with a single stimulus. Manticore is faster than Verilator with a single stimulus.

Zhang [59] called for a renewal in GPU-accelerated RTL simulation research by leveraging recent advances in GPU-compute APIs designed for machine learning.

Nexus [8] is an FPGA-based open-source emulation platform. It uses an array of dynamically scheduled logic processors with an 8-bit data. Similar to Manticore, Nexus leverages FPGA LUTs to accelerate RTL simulation. However, unlike Manticore, Nexus does not have a standard ALU and hence *all* logic operations are emulated using the LUTs. At the time of this writing, Nexus does not have a functional compiler. Therefore we are not able to provide a quantitative comparison between Manticore and Nexus.

DyVe [49] is an event-based, cycle-accurate RTL simulator running on a custom array of many-core SoCs linked with a central FPGA. DyVe partitions the circuit graph by its primary outputs, then incrementally merges program regions that share the largest number of inputs. DyVe's performance numbers are based on whether a target's simulation code fits its processors' L1, L2, or SDRAM memories, so direct comparisons are impossible.

9.3 Sequential RTL Simulation

Most efforts in improving RTL simulation on CPUs focused on reducing the runtime overhead of event-driven simulation. ESSENT [6, 7] is a cycle-accurate simulator that employs a coarsened, conditional, singular, static (CCSS) execution model [6]. CCSS is a novel, hybrid approach that minimizes the overhead of runtime

checks in event-driven simulation, especially in the presence of low activity factors. ESSENT is single-threaded and accelerates simulation of RISC-V cores (CPUs have low activity factors) by 1.5–11.5× over Verilator. However, it is not clear how ESSENT performs with spatial designs that exhibit high activity factors, especially since it is single threaded [5]. Manticore's performance is independent of a design's activity factor.

Cuttlesim [37] is a cycle-accurate simulator for Kôika [9], a rule-based HDL derived from Bluespec Verilog [34]. Cuttlesim uses the high-level semantics of Kôika to generate C++ code optimized for sequential performance. It reports 2–3× faster simulation than the equivalent RTL code running serial Verilator.

9.4 Deterministic Acceleration

Manticore's design philosophy is similar to VLIW processors and other Raw machines [54]. A more recent example is Groq's ML accelerator [3, 4]. The Groq chip has deterministic hardware datapaths that enable precise reasoning and control by software. Like RTL simulation, machine learning exhibits rare long-lived divergent code paths, which makes static scheduling feasible.

10 CONCLUSION

RTL simulation is an essential aspect of hardware design, and improved simulation offers many benefits to hardware designers. Currently, a designer must choose between an FPGA prototype's long compile times and fast execution or an RTL simulator's fast turnaround and slow speed. RTL simulation is slow because even state-of-the-art simulators fail to improve their performance by exploiting the abundant fine-grained parallelism in RTL circuits due to the high communication and synchronization cost of modern processors.

This work presented Manticore, a prototype, hardware-accelerated RTL simulator. Manticore's processors expose a *deterministic* hardware interface that allows a compiler to *statically* schedule programs across hundreds of simple cores. This approach eliminates the costly runtime overhead of synchronization, which enables efficient parallel simulation of RTL circuits and allows hundreds of cores to fit on a single chip. Our prototype FPGA Manticore implementation consistently achieves better performance over a state-of-the-art software RTL simulator. The Manticore system demonstrates the actual performance benefits of exploiting fine-grained parallelism in RTL code to accelerate simulation. Its higher speed allows several long simulations per day, as opposed to several per week on a conventional computer, with a concomitant improvement in developer productivity.

REFERENCES

[1] Azure pricing calculator. https://azure.microsoft.com/en-us/pricing/calculator/.

[2] Virtual machine series. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/.

[3] Dennis Abts, Garrin Kimmell, Andrew C. Ling, John Kim, Matthew Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, John Thompson, Michael Bye, Jennifer Hwang, Jeremy Fowers, Peter Lillian, Ashwin Murthy, Elyas Mehtabuddin, Chetan Tekur, Thomas Sohmers, Kris Kang, Stephen Maresh, and Jonathan Ross. A software-defined tensor streaming multiprocessor for large-scale machine learning. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 567–580. ACM, 2022.

[4] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 145–158. IEEE, 2020.

[5] Scott Beamer. A case for accelerating software RTL simulation. *IEEE Micro*, 40(4):112–119, 2020.

[6] Scott Beamer and David Donofrio. Efficiently exploiting low activity factors to accelerate RTL simulation. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020.

[7] Scott Beamer, Thomas Nijssen, Krishna Pandian, and Kyle Zhang. ESSENT: A high-performance RTL simuator. In *Workshop on Open-Source EDA Technology (WOSET), at International Conference on Computer-Aided Design (ICCAD)*, 2021.

[8] Peter Birch. Open source FPGA-based emulation with nexus. In *Workshop on Open-Source EDA Technology (WOSET)*, number 1, 2022.

[9] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of bluespec: a core language for rule-based hardware design. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 243–257. ACM, 2020.

[10] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning, 2013.

[11] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 7:1–7:13. IEEE Computer Society, 2016.

[12] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. Event-driven gate-level simulation with gp-gpus. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pages 557–562. ACM, 2009.

[13] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. GCS: high-performance gate-level simulation with gpgpus. In Luca Benini, Giovanni De Micheli, Bashir M. Al-Hashimi, and Wolfgang Müller, editors, *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 1332–1337. IEEE, 2009.

[14] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. Gate-level simulation with GPU computing. *ACM Trans. Design Autom. Electr. Syst.*, 16(3):30:1–30:26, 2011.

[15] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 1–8. IEEE Press, 2018.

[16] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 77:1–77:6. ACM, 2014.

[17] Jason Cong, Chang Wu, and Yuzheng Ding. Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In Sinan Kaptanoglu and Steve Trimberger, editors, *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, FPGA 1999, Monterey, CA, USA, February 21-23, 1999*, pages 29–35. ACM, 1999.

[18] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, 1982.

[19] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon J. Davidmann. Verilog HDL and its ancestors and descendants. *Proc. ACM Program. Lang.*, 4(HOPL):87:1–87:90, 2020.

[20] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert J. Ou, Colin Schmidt, Samuel Steffl, John Charles Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 769–774. IEEE, 2021.

[21] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, 2019.

[22] Jim Hogan. Hogan compares palladium, veloce, eve zebu, aldec, bluespec, dini. online, apr 2018.

[23] Xilinx Inc. *Alveo data center accelerator card platforms.* Xilinx Inc., August 2022.

[24] A. Jahanshahi, R. Sharifi, M. Rezvani, and H. Zamani. Inf4Edge: Automatic resource-aware generation of energy-efficient CNN inference accelerator for edge embedded fpgas. In *2021 12th International Green and Sustainable Computing*

Conference (IGSC), pages 1–8, Los Alamitos, CA, USA, oct 2021. IEEE Computer Society.

[25] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking, 2019.

[26] Nachiket Kapre and Jan Gray. Hoplite: A deflection-routed directional torus noc for fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 10(2):14:1–14:24, 2017.

[27] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H. Katz, Jonathan Bachrach, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi, editors, *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 29–42. IEEE Computer Society, 2018.

[28] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(1):69–79, 1999.

[29] Donggyu Kim. riscv-mini. https://github.com/ucb-bar/riscv-mini.

[30] Krishnamurthy. An improved min-cut algonthm for partitioning vlsi networks. *IEEE Transactions on Computers*, C-33(5):438–446, 1984.

[31] Jiajie Li, Yuze Chi, and Jason Cong. HeteroHalide: From image processing dsl to efficient fpga acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 51–57, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. From RTL to CUDA: A GPU acceleration flow for RTL simulation with batch stimulus. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*, pages 88:1–88:12. ACM, 2022.

[33] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Q. Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[34] Rishiyur S. Nikhil. Bluespec system verilog: efficient, correct RTL from high level specifications. In *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*, pages 69–70. IEEE Computer Society, 2004.

[35] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P. Jouppi, and David A. Patterson. The design process for google's training chips: Tpuv2 and tpuv3. *IEEE Micro*, 41(2):56–63, 2021.

[36] Open-Source FPGA Bitcoin Miner. https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner.

[37] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. Effective simulation and debugging for a high-level hardware language using software compilers. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 789–803. ACM, 2021.

[38] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 13–24. IEEE Computer Society, 2014.

[39] Hao Qian and Yangdong Deng. Accelerating RTL simulation with gpus. In Joel R. Phillips, Alan J. Hu, and Helmut Graeb, editors, *2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, California, USA, November 7-10, 2011*, pages 687–693. IEEE Computer Society, 2011.

[40] Lauro Rizzatti and Charley Selvidge. Designing a modern hardware emulation platform. online, jan 2020.

[41] Kamil Rocki, Dirk Van Essendelft, Ilya Sharapov, Robert Schreiber, Michael Morrison, Vladimir Kibardin, Andrey Portnoy, Jean-Francois Dietiker, Madhava Syamlal, and Michael James. Fast stencil-code computation on a wafer-scale processor. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 58. IEEE/ACM, 2020.

[42] Bashar Romanous, Mohammadreza Rezvani, Junjie Huang, Daniel Wong, Evangelos E. Papalexakis, Vassilis J. Tsotras, and Walid Najjar. High-performance parallel radix sort on fpga. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 224–224, 2020.

[43] Vladimir Rybalkin, Jonas Ney, Menbere Kina Tekleyohannes, and Norbert Wehn. When massive GPU parallelism ain't enough: A novel hardware architecture of 2D-LSTM neural network. *ACM Trans. Reconfigurable Technol. Syst.*, 15(1), nov 2021.

[44] L.A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[45] Vivek Sarkar and John L. Hennessy. Compile-time partitioning and scheduling of parallel programs. In Richard L. Wexelblat, editor, *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*, pages 17–26. ACM, 1986.

[46] Kaz Sato and Clif Young. An in-depth look at google's first tensor processing unit (TPU), 2017.

[47] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *CoRR*, abs/2106.08696, 2021.

[48] Wilson Snyder. Verilator, accelerated: Accelerating development, and case study of accelerating performance. 2nd Workshop on Open-Source Design Automation (OSDA).

[49] Tobias Strauch. Combining simulation and FPGA based verification to an affordable and ultra-fast multi-billion-gate verification system. In *Proceedings of the 30th International Workshop on Rapid System Prototyping, RSP 2019, New York, NY, USA, October 17-18, 2019*, pages 22–28. ACM, 2019.

[50] Xiang Tian and Khaled Benkrid. Design and implementation of a high performance financial monte-carlo simulation engine on an FPGA supercomputer. In Tarek A. El-Ghazawi, Yao-Wen Chang, Juinn-Dar Huang, and Proshanta Saha, editors, *2008 International Conference on Field-Programmable Technology, FPT 2008, Taipei, Taiwan, December 7-10, 2008*, pages 81–88. IEEE, 2008.

[51] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics coprocessor provides up to 15, 000x acceleration on long read assembly. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 199–213. ACM, 2018.

[52] ultraembedded. High throughput JPEG decoder. https://github.com/ultraembedded/core_jpeg.

[53] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[54] Elliot Waingold, Michael B. Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew I. Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman P. Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[55] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput CNN inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pages 29:1–29:6. ACM, 2017.

[56] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 170–179. ACM, 2010.

[57] Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys/.

[58] Tao Yang and A. Gerasoulis. Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[59] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. Opportunities for RTL and gate level simulation using gpus (invited talk). In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*, pages 166:1–166:5. IEEE, 2020.

[60] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning, 2022.

A APPENDIX

This appendix contains supplementary material that elaborates on points raised in the body of this paper.

A.1 Verilator's scaling

Fig. 11 and Fig. 12 demonstrate Verilator's self-relative speedup on the Intel Xeon 8272CL and Core i7-9700K respectively.

A.2 Microarchitecture

Fig. 15 outlines the pipelined implementation of one core. The pipeline is 14 stages deep and is logically divided into the typical five functions: fetch, decode, execute, memory access, and writeback. The CFU is implemented as 16 LUTRAMs.

The pipeline's frontend (right side of Fig. 15) is responsible for receiving messages during simulation. Each message is translated on the fly to a `SET` instruction and is written into instruction memory. A `SET` instruction updates a register with an immediate value. A state machine controls the execution of the pipeline, which is kept in strict lock-step with all other cores. The compiler inserts sleep instructions to coordinate communication between cores. An additional state machine handles incoming messages from the bootloader (see §A.3.1) and fills the instruction memory before the simulation starts.

A.3 Runtime

Manticore's runtime is a program running on the host processor that takes a binary generated by the compiler and runs it on the Manticore hardware accelerator connected to the host. The runtime copies the program binary into FPGA DRAM, then instructs the hardware bootloader (see Fig. 3) to copy the program into the local instruction memories. While the code executes, the runtime continuously polls the hardware state registers to handle exceptions or terminate execution.

A.3.1 Bootloader. Bootloading starts with a soft reset that brings all cores to a *boot state*. The soft reset only changes a few state registers in each core; it does not reset the register files or the scratchpads.

Cores continuously push `NOps` through their pipelines and snoop the NoC for instructions when in the boot state. A hardware bootloader (see Fig. 3) module reads the program binary from DRAM and streams the instructions to each core in sequence. Cores store the incoming instructions in their instruction memory. The cores then wait for a message from the NoC that contains a core-specific countdown value. At that point, the cores initialize a local timer with their specific countdown and start execution when the timer counts down to 0. The countdown starts all cores simultaneously despite the non-deterministic time required to read a program binary from DRAM.

Fig. 15 shows the instruction stream format that a core receives. This stream consists of a header that encodes the number of instructions in a program and a footer comprised of three words:

(1) `EPILOGUE_LENGTH` denotes the total number of messages the core expects to receive at every Vcycle.
(2) `SLEEP_LENGTH` denotes the sleep period length (see `sleep` in Fig. 2).
(3) `COUNT_DOWN` is the last word received that initiates a countdown to the start.

A.3.2 Exceptions. Manticore's hardware design and execution model make it possible to pause the execution, perform some computation on the host, and resume the execution on the FPGA. An example can illustrate this. Consider the Verilog statement:

```
if (count != 0) $display("got %d", count);.
```

This statement is executed as a global store instruction predicated by the condition `count != 0` that stores `count` to the global memory. The `$display` system call is translated to an `EXPECT` instruction that throws an exception when `count` is non-zero. When Manticore raises the exception, the grid stalls globally, and the host flushes the cache and reads the value of `count` from the FPGA DDR memory. The runtime prints this value for the user. Currently, our compiler only supports basic Verilog system calls. We plan to support arbitrary DPI (Verilog Direct Programming Interface) calls through this mechanism. However, crossing the host-device boundary is very expensive and should be avoided as much as possible.

A.4 Verilog to Assembly

Listing 2 shows a simple Verilog module that prints a message to the console at every cycle. The equivalent representation in Manticore assembly using two processes is given in Listing 3. This code for each process is repeatedly executed until some exception is raised by the `EXP` instructions. When that happens, execution on Manticore freezes until the host processor take the necessary action, for instance print the message or terminate simulation (e.g., `$finish`) Each process performs and implicit loop that is padded with extra `nops` in way that all processes jump back to their starting position at the same time.

A.5 Floorplanning

The U200 is a large *multi-die* FPGA that contains three SLRs[7]. Inter-SLR connections are significantly more costly than intra-SLR ones. While Vivado can find an efficient floorplan of Manticore's torus structure when the full design fits in a single SLR, it fails to do so when SLR crossings are necessary (see Fig. 16).

We get around this limitation by using a floorplanning script to guide Vivado. Cores do not directly access the shell and communicate only with their corresponding switch through a pipelined path. Therefore, cores do not need to respect a torus topology, so we spread half the cores in the top SLR and the other half in the bottom SLR. However, the NoC switches must be connected in a torus structure, so we constrain them to the narrow rectangular region in the central SLR. We constrain Vivado to use a set of dedicated hard registers available for crossing SLRs for each core-to-switch connection. We also co-locate the privileged core, cache, bootloader, and clock control logic in the central SLR as they access the shell. Finally, we minimize clock skew between the compute and control clock domains by assigning the clock buffers and clock roots to the same clock region. These optimizations enable a 15×15 grid to run at 475 MHz.

[7] Super Logic Regions, Xilinx terminology for a single die.

```
module EvenOdd(input wire clock);
  reg [15:0] counter = 0;
  always @(posedge clock) begin
    counter <= counter + 1;
    if (counter[0] == 1'b0) begin
        $display("%d is an even number", counter);
    end else begin
        $display("%d is an odd number", counter);
    end
    if (counter == 20) begin
        $finish;
    end
  end
endmodule
```

Listing 2: Simple Verilog Example.

```
.p0: // privileged process
    // init $r0 = 0, $r1 = 1, $r3 = 0, $r5 = 0, $r6 = 0
    PRED $r1  // predicate mask for global store
    GST  $r4, cat{$r0, $r0, $r0} // global store $r2 in address 0
    // if ($r6) $display("%d is an even number", $r4);
    EXPECT  $r6, $r0, 0
    // if ($r6) $display("%d is an odd number", $r4);
    EXPECT  $r5, $r0, 1
    EXPECT  $r3, $r0, 2 // if ($r3) $finish
    // recv $r5, $r3, $r4 from p1
    // sleep
    // implicit jump  to p0

.p1:
    // init $r0 = 0, $r1 = 1, $r2 = 20, $r4 = 0
    // $r4 is the counter
    SLICE  $r3, $r4[0 +: 1] // $r3 = counter[0]
    SEQ  $r5, $r4, $r2 // $r5 = (counter == 20)
    ADD  $r7, $r4, $r1 // $r7 = counter + 1
    SEND p0.$r4, $r4 // p0.$r4 = counter;
    NOP  7;
    MUX  $r8, $r3, $r1, $r0 // $r8 = !counter[0]
    SEND p0.$r5, $r3 // p0.$r5 = counter[0] (delayed)
    SEND p0.$r3, $r5 // p0.$r3 = (counter == 0)
    MOV $r4, $r7     // counter = counter + 1
    NOP  7;
    SEND p0.$r6, $r8
    // implicit jump to p1
```

Listing 3: Manticore assembly code for Listing 2.

Bench	$\|E\|$	$\|V\|$	LoC	Compile Time		
				Manticore	Verilator	Verilator MT
vta	56142	7037	190818	15m29s	2m33	26s
mc	52330	9182	30353	12m57s	1m13s	16s
noc	114364	6927	39363	15m14s	3m23s	36s
mm	89102	6659	64963	8m38s	7m5s	2m55s
rv32r	60430	4497	31761	5m57s	1m56s	29s
cgra	57532	4615	104498	7m48s	2m15s	37s
bc	8135	4630	276	2m23s	40s	27s
blur	9649	751	3869	42s	22s	15s
jpeg	1005	131	6542	16s	7s	3s

Table 8: Manticore, single-thread compile Verilator and multithreaded compile Verilator compilation times. $|E|$ and $|V|$ respectively denote the number of edges and nodes in the graph obtained by splitting each benchmark into a maximal set of independent processes (see §6.1). LoC denotes the Verilog lines of code for each benchmark.

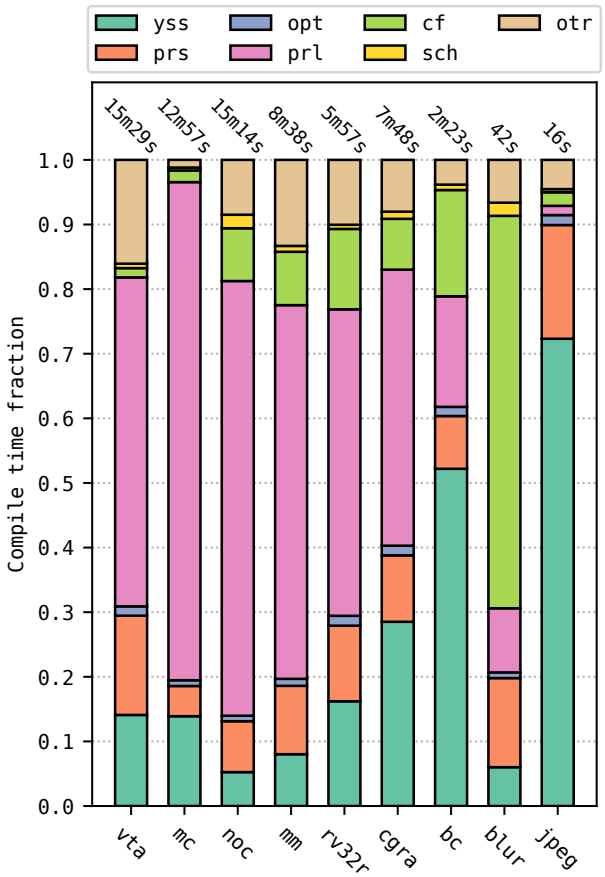


Figure 13: Breakdown of compilation time: Yosys (`yss`), assembly parsing (`prs`), basic optimizations (`opt`), parallelization (`prl`), custom function extraction (`cf`), scheduling (`sch`), others (`otr`).

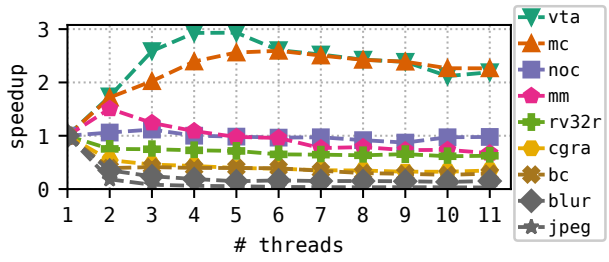
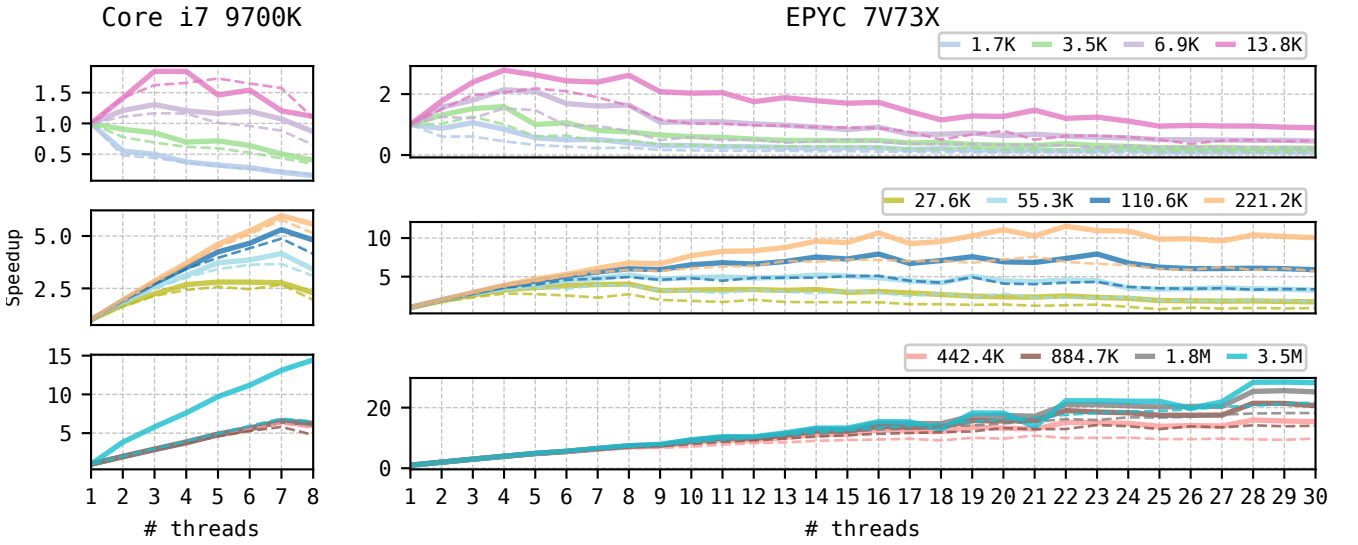


Figure 11: Verilator's parallel scaling on Xeon 8272CL.



(a) Core i7-9700K (b) EPYC 7V73X

	Instructions	1.7K	3.5K	6.9K	13.8K	27.6K	55.3K	110.6K	221.2K	442.4K	884.7K	1.8M	3.5M
epyc	model 1	[381, 6569]	[375, 3824]	[385, 2512]	[302, 1804]	[322, 1164]	[209, 820]	[105, 533]	[52, 396]	[26, 281]	[13, 186]	[7, 119]	[3, 71]
	model 2	[368, 3558]	[372, 2665]	[382, 1796]	[375, 1166]	[209, 845]	[105, 545]	[52, 415]	[26, 302]	[13, 208]	[7, 140]	[3, 84]	[2, 46]
i7	model 1	[1509, 8821]	[1527, 4411]	[1438, 2568]	[1106, 1913]	[555, 1485]	[277, 1016]	[139, 677]	[69, 400]	[35, 218]	[17, 100]	[9, 58]	[4, 30]
	model 2	[1134, 7336]	[1509, 3718]	[1610, 2429]	[916, 1692]	[464, 1299]	[232, 965]	[116,613]	[58, 345]	[29, 189]	[14, 94]	[7, 46]	[2, 26]

Figure 14: Measured parallel simulation speedup on a desktop processor (left) and server processor (right). Dashed lines model only synchronization cost. Solid lines also include i-cache pressure. Each curve is labeled by the number of instructions executed per simulation step. The table at the bottom, shows the [min., max.] simulation rates corresponding to each model.

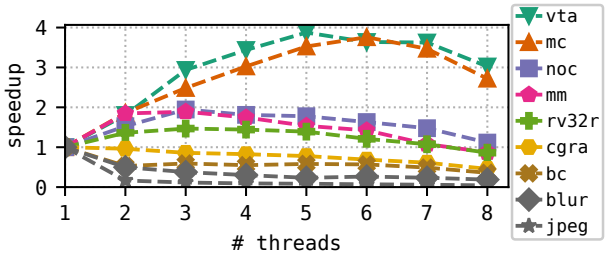


Figure 12: Verilator's parallel scaling on Core i7-9700K.

	LUT	LUTRAM	FF	BRAM	URAM	DSP	SRL
#	545	128	1358	4	2	1	102
%	0.05	0.02	0.05	0.19	0.21	0.01	0.02

Table 7: Resource utilization of a single core on the U200. Percentages are the faction of the total resources available on an U200.

A.6 Compile Time Analysis

Table 8 reports the compilation times of each benchmark for both Manticore and Verilator (single-thread and multithread). Manticore's compiler is written in Scala and runs on the JVM, whereas Verilator is written in C++. Fig. 13 contains a detailed breakdown of compilation steps and their contribution to the total time. Most of the compile-time is spent in parallelizing RTL code. We expect to improve compilation time (it is currently unoptimized).

A.7 FPGA resource utilization

Table 7 reports FPGA resource utilization for a single core. The dominant resource is the URAM, as two are required per core (one for the instruction memory and one for the local scratchpad). This limits the total number of cores on the U200 to 398. As all cores may not use their scratchpad memory, one optimization is a heterogeneous implementation where some cores lack a scratchpad and rely on only a large register file so that other cores can have more local memory. We leave heterogeneous processor grids to future work.

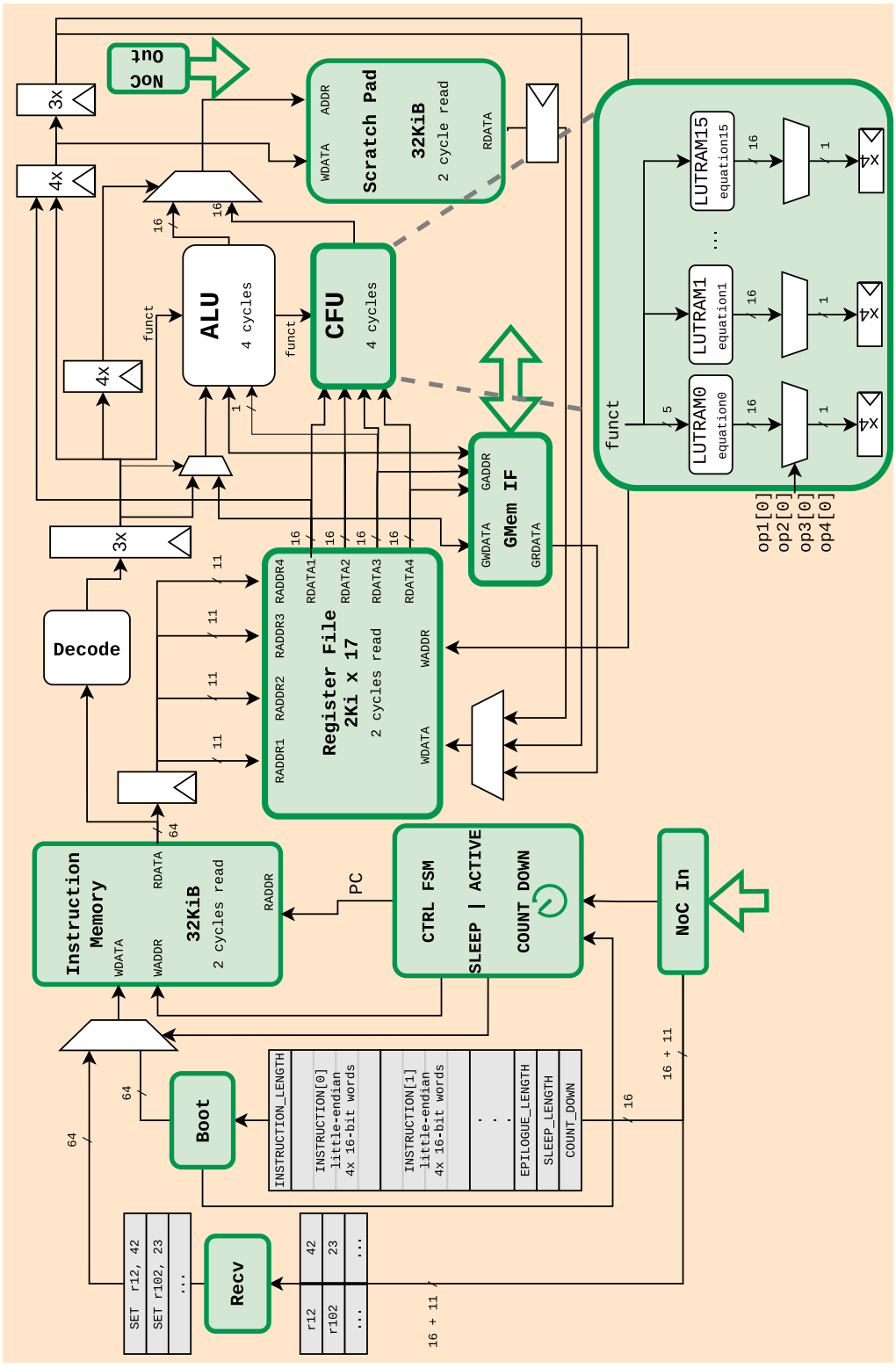


Figure 15: Microarchitecture of a core in Manticore's processor grid. Details are omitted for legibility.

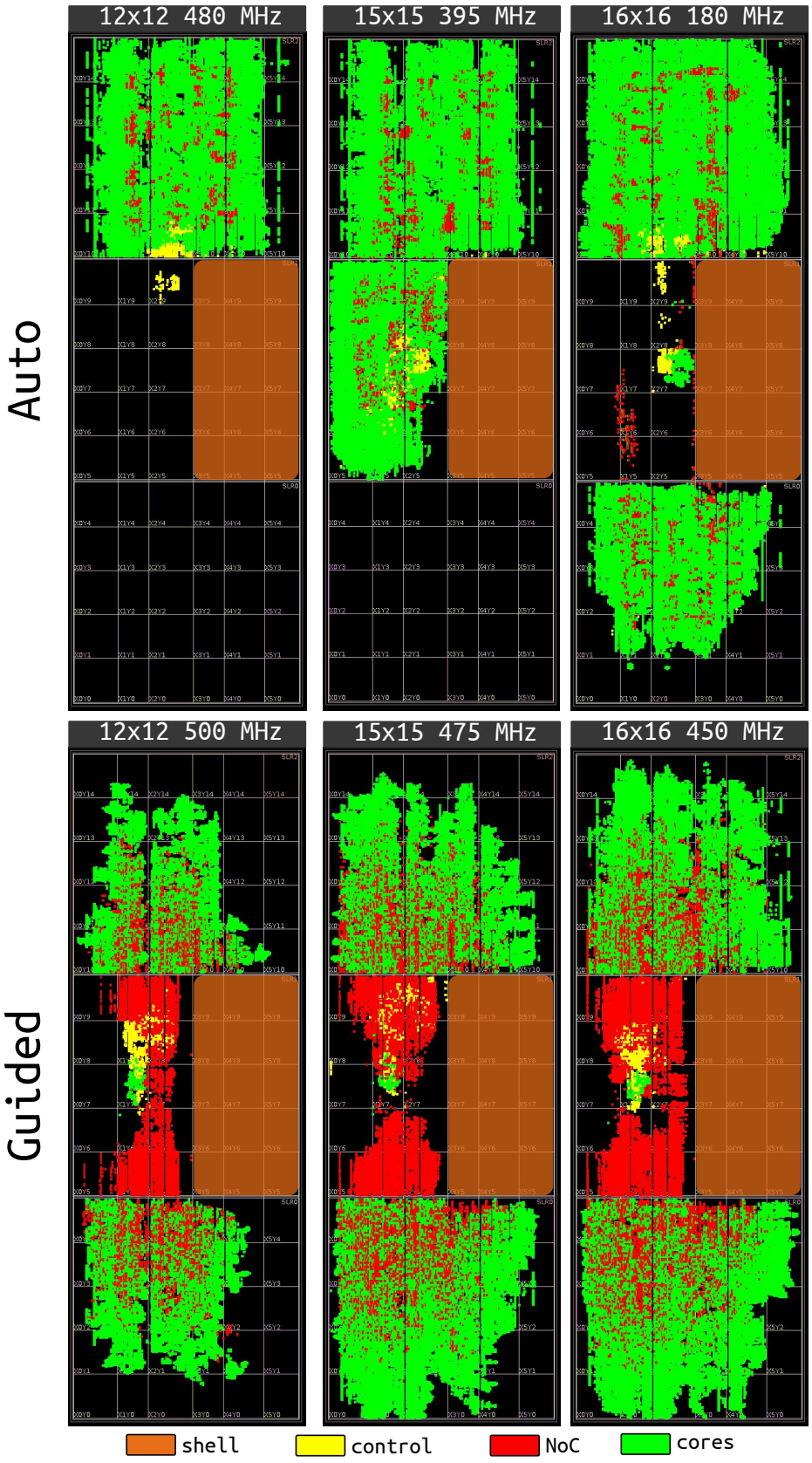


Figure 16: Floorplan of Manticore's physical implementation on U200. Vivado's automatic floorplanning is at the top and our guided floorplanning is at the bottom. The cores are colored in green, the NoC in red, and the control clock domain in yellow. The fixed shell is marked as an orange box in each floorplan. The clock speed is considerably higher with the guided floorplans.