Accelerating RTL Simulation with Hardware-Software Co-Design

Fares Elsabbagh
MIT CSAIL
Cambridge, MA, USA
farese@csail.mit.edu

Shabnam Sheikhha
MIT CSAIL
Cambridge, MA, USA
shab@csail.mit.edu

Victor A. Ying*
MIT CSAIL
Cambridge, MA, USA
victory@csail.mit.edu

Quan M. Nguyen
MIT CSAIL
Cambridge, MA, USA
qmn@csail.mit.edu

Joel S. Emer
MIT CSAIL
Cambridge, MA, USA
emer@csail.mit.edu

Daniel Sanchez
MIT CSAIL
Cambridge, MA, USA
sanchez@csail.mit.edu

ABSTRACT

Fast simulation of digital circuits is crucial to build modern chips. But RTL (Register-Transfer-Level) simulators are slow, as they cannot exploit multicores well. Slow simulation lengthens chip design time and makes bugs more frequent.

We present ASH, a parallel architecture tailored to simulation workloads. ASH consists of a tightly codesigned hardware architecture and compiler for RTL simulation. ASH exploits two key opportunities. First, it performs *dataflow execution* of small tasks to leverage the fine-grained parallelism in simulation workloads. Second, it performs *selective event-driven execution* to run only the fraction of the design exercised each cycle, skipping ineffectual tasks. ASH hardware provides a novel combination of dataflow and speculative execution, and ASH's compiler features several novel techniques to automatically leverage this hardware.

We evaluate ASH in simulation using large Verilog designs. An ASH chip with 256 simple cores is gmean 1,485× faster than 1-core Verilator, and it is 32× faster than parallel Verilator on a server CPU with 32 complex cores, while using 3× less area.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; • **Hardware → Electronic design automation**.

KEYWORDS

simulation, register-transfer-level, hardware acceleration, dataflow execution, speculative execution, domain-specific architectures.

ACM Reference Format:
Fares Elsabbagh, Shabnam Sheikhha, Victor A. Ying, Quan M. Nguyen, Joel S. Emer, and Daniel Sanchez. 2023. Accelerating RTL Simulation with Hardware-Software Co-Design. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614257

*Victor A. Ying began working at Tenstorrent after this work was done.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614257

1 INTRODUCTION

Fast simulation of digital circuits is crucial for effective digital design. Current processors and SoCs integrate hundreds of complex components, including cores, accelerators, and memory hierarchies. Simulating these systems is needed to explore their design space and verify their correctness. In particular, verification dominates chip design cost [24]. Unfortunately, existing simulators are slow. This increases design time, limits the efficiency of the resulting design (due to insufficient design exploration), and makes hardware bugs more likely (due to insufficient verification coverage).

Simulation can happen at different levels of abstraction. In this work, we focus on accelerating *Register-Transfer-Level (RTL)* simulation, i.e., the precise simulation of a hardware design written in a Hardware Description Language (HDL) like Verilog. RTL simulation is a common bottleneck in chip design, as it is needed to debug an evolving implementation and evaluate and improve its performance, power, and area.

Prior work has built software and hardware techniques to accelerate RTL simulation. Unfortunately, these techniques have major drawbacks. The fastest software simulators, like Verilator [56], are purpose-built optimizing compilers that translate RTL code into an efficient software program (e.g., in C++) [8, 46, 56]. But simulators scale poorly across CPU cores, because parallelism is *fine-grained*: work must be split into small *tasks* with few operations each. But these tasks have many data dependences and need frequent synchronization. This causes high overheads on conventional multicores, which erase most of the benefits of parallelization (Sec. 2).

Since simulators are so slow, chip designers often resort to specialized *emulators*, which map a design to hundreds to thousands of FPGAs [13, 63] or specialized processors [9, 12]. Emulators work at a lower level of abstraction (logic gates), which limits their efficiency; they are large and expensive; and they suffer very long compile times, *days to weeks* for large designs [23]. This limits emulators to final integration testing, when the design is nearly finished and bug-free. By contrast, simulators compile designs in minutes, so they are the only option for most of the design cycle, when changes are frequent (Sec. 2.4). Thus, we focus on accelerating software simulation, retaining its fast compilation while achieving dramatic speedups.

We propose the ASH (Accelerator of Simulated Hardware) system, a carefully co-designed architecture and compiler for RTL simulation. We present two ASH variants that target the key acceleration opportunities in RTL simulation:

Dataflow execution with DASH: RTL simulation is naturally expressed as many tasks whose outputs drive the inputs of other tasks, so it is a perfect match for dataflow (i.e., data-driven) execution [15]. Dataflow execution has the potential to expose abundant parallelism, as each task can run as soon as its inputs have been produced. Unfortunately, RTL simulation fails to scale on current multicores because it needs small tasks to expose sufficient parallelism, but small tasks have prohibitive overheads on multicores that negate the benefits of parallelism.

DASH (Dataflow ASH) tackles this challenge by providing novel hardware mechanisms for dataflow execution of small tasks. We extend a multicore with hardware to orchestrate dataflow execution: DASH hardware gathers inputs and dispatches tasks to cores after inputs are available. DASH implements a novel *prioritized* dataflow execution that orders tasks to reduce in-flight tasks, avoiding the overheads of prior dataflow architectures [17, 40, 62]. DASH also features a simple, partitioned memory system without cache coherence specialized to the needs of simulation.

Selective event-driven execution with SASH: In most digital designs, only a small fraction of the signals change each cycle [7, 16, 31], so simulating the whole design every cycle is wasteful. *Selective execution*—simulating *only* the active fraction of the design—is more efficient. Prior work has proposed *event-driven* simulators and architectures [1, 14, 19, 20, 26] where work happens through events that may schedule other events for later execution. Unfortunately, these prior systems lack support for dataflow execution, so running dataflow tasks selectively incurs costly synchronization through memory that negates the benefits of selective execution.

SASH (Selective event-driven ASH) tackles this challenge by extending DASH with selective execution, running *only* tasks whose inputs change during a given cycle. Selective execution introduces *dynamic* data dependences: each task receives inputs only from producers that have run, but not from skipped ones. SASH handles these dependences with *speculative execution*, running each task with its received inputs and using old values for non-received ones (i.e., speculating that these inputs will not change this cycle). A late-received input causes the task to abort and re-execute.

ASH compiler: The ASH compiler builds on Verilator and introduces novel techniques to parallelize RTL code automatically and efficiently, place tasks and data to minimize communication, and effectively use DASH's dataflow scheduling hardware as well as SASH's speculative execution features.

In summary, ASH leverages classic compiler and architecture concepts, including dataflow and speculative execution, and contributes new techniques to make their combination efficient and use them automatically. While prior systems used either task-level dataflow [17, 75] or speculation [1, 18, 27], ASH is the first to combine them, achieving good scalability and avoiding ineffectual work.

We evaluate ASH in simulation, on four large Verilog designs that include CPU cores, GPU cores, and accelerators. Our contributions make RTL simulation scale to hundreds of cores. A 256-core ASH system is gmean 1,485× faster than serial Verilator on a system with a single (simple) core, and it is gmean 32× faster than parallel Verilator on a commercial CPU with 32 complex cores while using 3× less area.

2 MOTIVATION AND BACKGROUND

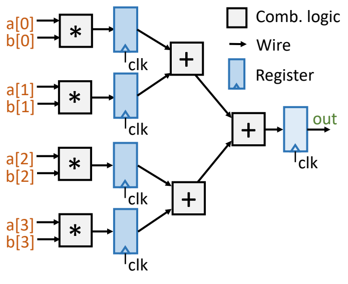
In this section, we first introduce the key characteristics of RTL simulation. We then present its two key optimization opportunities and quantify the limitations of existing systems, including parallel simulators and FPGA emulators.

2.1 Understanding RTL simulation

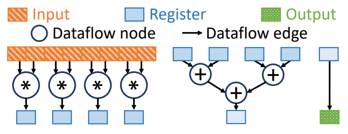
RTL simulation functionally evaluates a digital circuit written in an HDL like Verilog. RTL simulation precisely evaluates how signals and state elements (e.g., registers) evolve over clock cycles. For instance, Fig. 1a shows a 2-stage pipeline that computes the dot-product of two 4-element vectors. In an RTL simulation, a testbench feeds inputs each simulated cycle, and compares the outputs (and maybe other signals) against a reference model. It may also evaluate performance (e.g., cycles taken on a long computation) and power (by measuring component activities). Thus, RTL simulation verifies digital designs and analyzes their efficiency.

In this work, we focus on simulating *synchronous* circuits, by far the dominant design style (other styles, like asynchronous circuits [11], are rarely used). These systems comprise combinational logic and clocked registers. (For simplicity, we assume that all registers are driven by one clock, but our techniques also work with multiple clock domains [70].)

Dataflow graphs: RTL simulation of synchronous circuits can be abstracted as the execution of a *dataflow graph*, where nodes represent combinational logic and edges represent data values. Fig. 1b shows the dataflow graph for the circuit in Fig. 1a, with **inputs** on the top and the only **output** on the bottom. The same **registers** are shown on both top and bottom, as they are read and written every cycle. RTL simulation evaluates each node in the dataflow graph once per simulated cycle in an order consistent with its dataflow edges.



(a) Example 2-stage pipeline.



(b) Dataflow graph for (a).

Figure 1: RTL simulation of synchronous circuits can be abstracted as a dataflow graph.

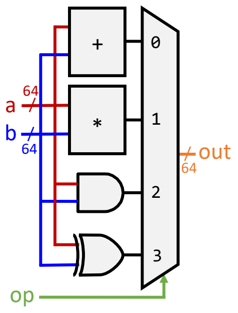
RTL simulators: Modern RTL simulators like Verilator [56], ESSENT [8], RepCut [71], and Cuttlesim [46] are *compilation-based*: they translate HDL code to a dataflow intermediate representation (IR), then translate the IR to a software program that simulates the circuit. For example, Fig. 2 shows how an ALU is translated from Verilog to efficient C++. Translation leverages the semantic similarities of HDLs and software languages, e.g., arithmetic operations are directly translated to C++ arithmetic operations, not bit-level operations.

RTL simulation admits two key optimizations: executing the dataflow graph in parallel, and executing only the nodes whose inputs change. We quantify the shortfalls of existing systems, showing the need for hardware support.

```
typedef bit [63:0] Word;
typedef enum {ADD, MUL, AND, XOR} AluOp;

module alu(input Word a, input Word b,
     input AluOp op, output Word out);
  always_comb begin
    case (op)
      ADD: out = a + b;
      MUL: out = a * b;
      AND: out = a & b;
      XOR: out = a ^ b;
    endcase
  end
endmodule
```

(a) Verilog RTL code for a simple ALU.



(b) ALU hardware.

```
class alu {
 // Inputs
 uint64_t a, b;
 uint8_t op;
 // Outputs
 uint64_t out;

 void eval() {
  if (op == 0) out = a + b;
  else if (op == 1) out = a * b;
  else if (op == 2) out = a & b;
  else if (op == 3) out = a ^ b;
 }
};
```

(c) Generated C++ code for (a).

Figure 2: Fast RTL simulators compile HDL code (e.g., Verilog) to a software program.

2.2 Parallel execution needs small tasks

RTL simulators can parallelize execution by leveraging the dataflow graph, which exposes the available parallelism: independent paths can be evaluated in parallel. However, fine-grained dependences cause serialization: for instance, in Fig. 1b the adder tree can be partially parallelized, but two inputs feed into the final adder. In practice, combinational paths have tens of nodes, with edges fanning in and out, creating complex dependences that make parallel execution challenging.

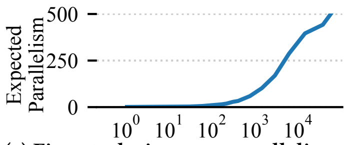
The granularity of scheduling is a key aspect of parallelization. Each *node* in the dataflow graph performs a tiny amount of work, often just a single assignment. On current multicores, communication costs would make individually scheduling these nodes extremely expensive. Instead, dataflow nodes are first *merged* into larger *tasks*. Then, this *task-level dataflow graph* is scheduled across cores.

Merging nodes into larger tasks introduces delicate tradeoffs between work-efficiency and parallelism. On the one hand, larger tasks are more work-efficient. On the other hand, merging too much causes two problems: it *reduces parallelism* and *creates tasks with many inputs and outputs*, which are harder to schedule. For instance, merging the four independent multiplier nodes in Fig. 1b reduces parallelism by 4× and produces a task with 8 inputs and 4 outputs.

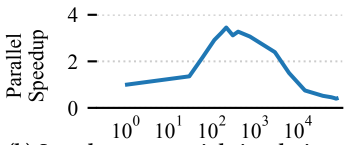
The problem is that existing multicores force simulators to create huge sequential tasks that sacrifice most of the parallelism. This happens because communication and synchronization are very expensive: communicating a single value across cores is done through shared memory, and costs hundreds of cycles. Tasks must have thousands of operations to amortize these costs. But larger tasks also have more dependences, inducing needless serialization that tanks parallelism.

To quantify this problem, we use Verilator [56] to compile an RTL simulator for Chronos, a graph-processing accelerator with 128 processing engines (see Sec. 8 for methodology details). Verilator iteratively merges dataflow nodes into large tasks. It tries to avoid lengthening the *critical path*, i.e., the costliest chain of tasks from initial values (inputs or registers) to final values (outputs or registers). Merging nodes can produce a costlier task on the critical path, so Verilator stops merging when the *parallelism*, i.e., the ratio of total cost of all tasks to critical path cost, reaches a threshold.

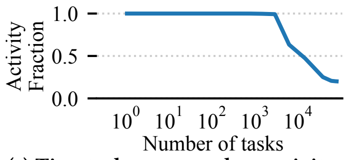
Fig. 3a shows how parallelism grows with the number of tasks. The x-axis uses a logarithmic scale to show the wide range of options: from serial simulation using one task, to restricting Verilator to only do merges that avoid hurting the critical path, which produces 74 K tiny tasks. Fig. 3a shows that parallelism is plentiful, but only with many small tasks: merely achieving 100× parallelism requires about 2000 tasks.



(a) Finer tasks increase parallelism.



(b) Speedup over serial simulation.



(c) Tiny tasks uncover low activity.

Figure 3: Effects of varying task size for Verilator. More tasks (x-axis) means finer grain.

Fig. 3b shows how this potential parallelism translates to performance. It reports speedup over serial Verilator across the range of task counts (and granularities) on a 32-core AMD Zen 2 CPU. Verilator schedules tasks statically to a fixed number of threads (a good strategy, as task costs are known at compile time). We adapt Verilator to allow varying threads and merging level independently, and report the performance of the best thread count for each merging level.

Unfortunately, Fig. 3b shows that parallel execution yields limited speedups: the best performance is for 203 tasks, which is only 3.5× faster than serial simulation despite an expected parallelism of 18.7. Such limited speedups are typical for Verilator's multithreaded simulations [57, 58]. This is because, even with 203 relatively large tasks, each task has only about a microsecond worth of work, and synchronization costs dominate. These costs make speedups plummet with smaller tasks. DASH's support for dataflow execution avoids these costs, enabling small tasks with high parallelism.

2.3 Selective execution needs small tasks

Digital systems have low activity factors: a small fraction of the logic switches each cycle [7, 16, 31]. Simulators can exploit this by evaluating only the nodes whose inputs change.

Effective *selective execution* demands small tasks. Fig. 3c reports the fraction of work performed by *active* tasks, averaged across simulated cycles. A task is active and must execute if any of its inputs has changed since it last executed. With small tasks, active tasks make up only 20% of work. But larger tasks make it more likely that some logic within each task is active, making inactive tasks rare. Below 2000 tasks, the activity factor is pegged to 100%.

Prior work has developed serial simulators that exploit low activity factors through conditional execution [8]. But the inability of parallel simulators to deal with small tasks makes selective execution ineffective, and to our knowledge, state-of-the-art parallel RTL simulators are non-selective [57].

In parallel simulation, skipping inactive tasks adds complexity because any active task must now know which inputs change and which stay the same. Prior work has proposed *conservative* and *optimistic* parallelization approaches. With conservative parallelization, as done in the Chandy-Misra-Bryant (CMB) algorithm [14], each inactive task must send *null messages* to indicate lack of changes in its outputs. This approach still traverses the entire dataflow graph each cycle—it skips only the work within each task, *but not the communication.* For the tiny tasks needed in RTL simulation, this is inefficient.

Time Warp [25, 26] avoids null messages and conservative synchronization by using an optimistic approach: tasks execute speculatively, assuming that no other inputs will be received, and an input received out of order causes the task to abort and re-execute. But Time Warp has no concept of dataflow execution: each task carries one argument from a single simulated object to another. Thus, while prior accelerators used this optimistic approach in other domains [1, 20, 27], they lack the hardware support needed to manage speculative execution of *dataflow tasks.* As we later quantify (Sec. 10), ASH is the first system with sufficient hardware support to make selective execution profitable in parallel RTL simulation.

2.4 Emulation versus simulation

Hardware emulators are the only platforms capable of running billion-gate designs at MHz speeds. Emulators work by mapping gates directly to hardware elements, either FPGAs or specialized gate processors. However, their long compile times make emulators ill-suited to crucial parts of the chip design process, such as design space exploration and prototyping, when RTL changes are extremely frequent.

Current FPGA-based emulators include Cadence's Protium [13] and Synopsys's ZeBu [63]. Their compilation times span from days to weeks, as partitioning, placing, and routing a design across many FPGAs is a very complex process. Processor-based emulators use a large collection of ASICs instead of off-the-shelf FPGAs: Cadence's Palladium [12] uses a massive array of Boolean processors; and Mentor's Veloce Strato [37] uses custom FPGAs tailored to emulation. (Sec. 10.3 describes additional systems.) ASIC-based systems have lower compilation times (hours to days) and better debugging capabilities than FPGA-based ones, but they are slower and more expensive. Thus, emulators are restricted to late in the design process, when changes are rare.

Simulation is the best choice for evaluating RTL designs at high speeds and at large scales. To concretely quantify its advantages, Table 1 shows the times to compile, and then run, Chronos in *(1)* pure software simulation, *(2)* SASH, and *(3)* emulation on 2 FPGAs. Simulated duration is important, because different activities demand different durations: one million cycles (~ 1 ms) suffices for short tests, one billion cycles (~ 1 s) enables thorough performance evaluation, and one trillion cycles (~ 20 min) enables running complex software. During RTL development, 1M-1Bcycle tests are common.

To benchmark emulation, we use a system with two large Alveo U250 FPGAs [74] directly connected through 200Gbps links. Chronos does not fit in one FPGA. By manually partitioning the design and applying Virtual Wires [6], we achieve a 1.4 MHz speed, limited by inter-FPGA latency. After days of tuning, this design compiled in *13 hours* (running both FPGA compiles in parallel). Commercial emulators can automate partitioning but at the cost of longer compile times, so this compilation time is generous to emulators.

System	Compile time	Sim. speed	Time to simulate		
			1M cycles	1B cycles	1T cycles
SW sim.	2 mins	11 KHz	3.9 mins	1.1 days	3 years
SASH	2 mins	414 KHz	2.4 mins	43 mins	28 days
FPGA ×2	13 hours	1.4 MHz	13.2 hours	13.4 hours	8.7 days
SASH/FPGA			332×	18.9×	0.3×

Table 1: Simulation vs. FPGA emulation.

SASH has a somewhat lower simulation speed than emulation, but compiles Chronos in *2 minutes.* This enables a much shorter design and debug cycle than emulation: in the time it takes to carry out a billion-cycle FPGA emulation run, engineers could have done nearly 20 SASH runs.

3 SYSTEM OVERVIEW

Fig. 4 shows an overview of the ASH system. ASH consists of a co-designed hardware architecture and RTL compiler. The architecture efficiently supports the two key optimizations identified in Sec. 2: it scales simulation to many cores using dataflow execution, and it avoids needless work through selective execution. The RTL compiler leverages these features to achieve high performance.

ASH hardware is designed from the ground up to support fine-grain tasks. The ASH chip consists of multiple tiles, and each tile has several cores and a *task management unit* that queues and dispatches tasks to cores. We design two variants of ASH, and present them in stages: DASH (Sec. 4) supports task dataflow specialized to simulation workloads, and SASH (Sec. 5) adds speculation to DASH to achieve selective execution. ASH does not use global coherence; it performs all inter-tile communication through task arguments; and it performs task-driven instruction prefetching (Sec. 6) to cope with the high instruction footprints of RTL simulation.

Our ASH compiler implementation is based on Verilator, a state-of-the-art Verilog/SystemVerilog simulator [56]. Verilator supports parallelization on shared-memory multicores, which as we have seen, achieves limited speedups (Sec. 2.2). We thus contribute new compiler techniques that automatically divide work into fine-grain dataflow tasks, partition tasks and their data across tiles, and modify these tasks for selective, speculative execution. We explain these compiler contributions as we introduce DASH and SASH.

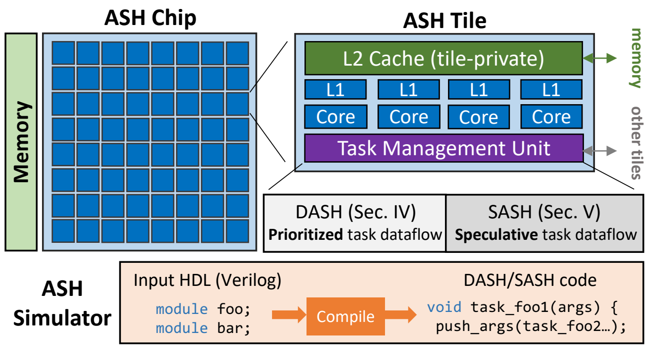


Figure 4: ASH system overview.

4 PRIORITIZED DATAFLOW EXECUTION WITH DASH

This section presents the DASH system, which implements dataflow execution but not selective execution—all tasks run in every simulated cycle. We first present DASH's execution model and ISA; then describe its hardware implementation; and finally explain how RTL compiles to DASH.

4.1 Execution model and ISA

All computation in DASH happens through *tasks*. A task is a unit of code—a function in our implementation. Tasks can have one or more input *arguments*, which are produced by other tasks, and can produce arguments for other tasks.

Dataflow execution: Hardware provides native support for *dataflow execution* of tasks. A task enqueues arguments for other tasks with a push_args instruction. Hardware buffers the arguments for each task; the task becomes *ready* when all its arguments are available. Ready tasks are queued and dispatched to cores for execution. Cores execute each task to completion, and stall if no further tasks are available.

Hardware support for dataflow execution avoids communication through shared memory: a task enqueues output arguments from registers, and when a task starts execution, its input arguments are placed in registers.

Prioritized execution and timestamps: *Prioritized execution* is DASH's key distinguishing feature. Previous dataflow architectures (Sec. 10.2) execute the dataflow graph in an unordered fashion. This often produces an excessive number of in-flight task arguments, requiring large storage structures for arguments [3, 75].

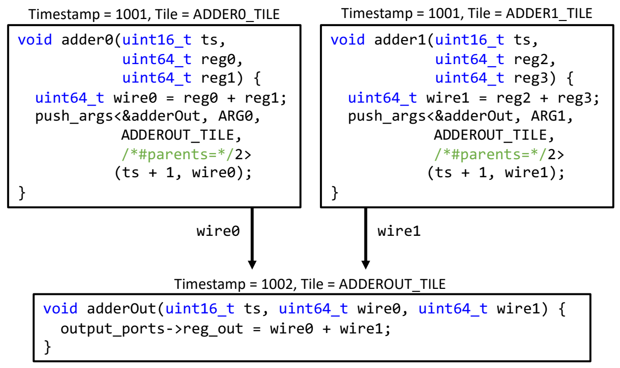
Instead, DASH prioritizes tasks for execution: software assigns an integer *timestamp* to each task, and hardware dispatches ready tasks for execution in timestamp order. The DASH compiler assigns timestamps to prioritize work in the critical path and keep a small memory footprint. Thus, DASH has much lower footprint than prior dataflow architectures, and a simpler implementation (Sec. 9.3).

Timestamps also differentiate invocations of the same task. Each task invocation has a different timestamp (as each task runs once per simulated cycle), and timestamps enable their arguments to coexist. This allows overlapping the execution of successive cycles, increasing parallelism.

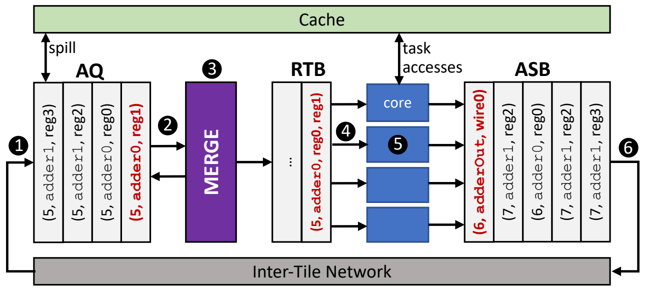
Memory: Although task arguments are passed through registers, tasks can also access memory. Supporting data accesses serves two purposes. First, some simulated structures, like SRAM memories, have a lot of state. It would be inefficient to encode all this state in dataflow edges; it is better to store this state in memory and have dataflow nodes that access it. Second, hardware limits the number of arguments per task (our implementation allows up to five 64-bit arguments). Tasks that exceed this limit must store some inputs in memory (Sec. 4.3.4).

Task mapping: While memory stores some state, there is no global shared memory, and *all* global communication happens through task arguments. To enforce this restriction, each task is mapped to run on a fixed tile, and all memory accesses to the same data must happen from same-tile tasks.

Argument enqueue ISA: The push_args instruction conveys the task's metadata: function pointer, timestamp, tile id, and the total number of arguments. push_args may also convey one or more argument values and their indices.


Timestamp = 1001, Tile = ADDER0_TILE
```
void adder0(uint16_t ts,
            uint64_t reg0,
            uint64_t reg1) {
  uint64_t wire0 = reg0 + reg1;
  push_args<&adderOut, ARG0,
            ADDEROUT_TILE,
            /*#parents=*/2>
           (ts + 1, wire0);
}
```
Timestamp = 1001, Tile = ADDER1_TILE
```
void adder1(uint16_t ts,
            uint64_t reg2,
            uint64_t reg3) {
  uint64_t wire1 = reg2 + reg3;
  push_args<&adderOut, ARG1,
            ADDEROUT_TILE,
            /*#parents=*/2>
           (ts + 1, wire1);
}
```

Timestamp = 1002, Tile = ADDEROUT_TILE
```
void adderOut(uint16_t ts, uint64_t wire0, uint64_t wire1) {
  output_ports->reg_out = wire0 + wire1;
}
```
Figure 5: Task code for the adder tree in Fig. 1.



1. AQ receives reg1 for adder0 at ts 5 and inserts based on timestamp priority.
2. Merge unit dequeues reg0 and reg1 for adder0 at ts 5.
3. If args missing, wait in the merge unit; otherwise, merge and send to RTB.
4. RTB dispatches adder0.
5. adder0 runs, produces wire0, pushes an arg descriptor for adderOut at ts 6.
6. ASB sends arg descriptors to appropriate tiles.

Figure 6: DASH tile, with descriptors from Fig. 5.

Fig. 5 shows this interface, with code for Fig. 1's adder tree. Tasks adder0 and adder1 can run in parallel, each enqueueing arguments for the adderOut task to sum.

4.2 DASH hardware implementation

DASH extends each tile with a Task Management Unit (TMU) shown in Fig. 6 that consists of two components: an Argument Queue (AQ) and an Argument Send Buffer (ASB).

Each execution of push_args creates an *argument descriptor*, which includes the task metadata (function pointer, tile id, timestamp, and number of parents) and task arguments. This descriptor is sent to the tile where the task is mapped, and queued in the AQ. The TMU merges queued descriptors of ready tasks, and dispatches them for execution.

The ASB is a simple FIFO that buffers produced argument descriptors and autonomously sends them to their destination tiles. The ASB makes push_args asynchronous: the core continues execution and does not stall while the descriptor is sent. ASBs use a push message protocol with NACKs and exponential backoff to handle full AQs in destination tiles.

Each tile's AQ stores in-flight argument descriptors for tasks mapped to the tile. Along with the argument descriptors, the AQ also has a priority queue sorted by the pair (timestamp, task function pointer). This enables quickly finding the argument descriptors for the lowest-timestamp queued task. We implement this fixed-capacity priority queue using a pipelined heap [1, 10] that can pop one descriptor per cycle.

The TMU's *merge unit* consumes argument descriptors from the AQ in priority order, and merges them into tasks. The merge unit is similar to a reservation station [69] in an OOO core: it has a small number of entries (16 in our implementation). Each entry gathers the arguments for a task: the first descriptor for a task allocates a new entry, and later arguments for the task are merged in the same entry. After all the arguments are merged into one entry, the entry is freed and the resulting ready task is queued for execution at a core, in a small *ready task buffer*, as shown in Fig. 6. Ready tasks may execute out of order, while tasks with missing arguments wait in the merge unit. Unlike reservation stations, since tasks may arrive and allocate entries out of order, the merge unit may need to evict an unready task back to the AQ to allow an earlier-timestamp task to be merged. These evictions are very rare (Sec. 9.3).

The AQ has a small and fixed size (512 descriptors in our implementation). When the AQ fills up, a simple FSM spills high-timestamp descriptors to memory and later returns these descriptors to the AQ in timestamp order. This prevents high-timestamp descriptors from starving low-timestamp ones.

Prioritized execution keeps DASH's TMU small and efficient. Prior dataflow architectures implemented large *wait-match* units that stored arguments for unready tasks and performed expensive lookups to merge them and find ready tasks [17, 40, 62]. Instead, DASH uses tiny merge units similar to reservation stations, which work almost as well as unbounded ones because most lowest-timestamp tasks have all arguments available (Sec. 9.3). Moreover, prior dataflow architectures hinder memory footprint: they execute the dataflow graph in an unordered fashion, so produced arguments may not be consumed for a long time. Instead, prioritization produces arguments in the right order, so modestly sized AQs suffice to keep spills rare.

4.3 DASH compiler

We build DASH's compiler by modifying Verilator [56], a state-of-the-art compiler that transforms Verilog RTL into an efficient parallel C++ simulator. We reuse its frontend, which translates Verilog to a dataflow-style IR. We then heavily modify its backend to produce DASH code from this IR. Fig. 7 shows our backend's compiler passes, which we describe next.

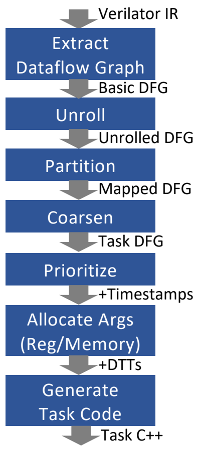


Figure 7: DASH compiler passes.

4.3.1 Increasing parallelism with the unrolled dataflow graph. Existing RTL simulators parallelize execution cycle by cycle, and use a *single-cycle* dataflow graph like the one introduced in Fig. 1b. In this representation, wires are converted to dataflow edges, but registers, inputs, and outputs are *in memory*. This representation limits parallelism in two ways. First, it induces write-after-read dependences within each simulated cycle: every read to a register must be performed before the register is updated. For example, in Fig. 1b, the adders in the second pipeline stage must read their registers before the multipliers in the first stage write them. Second, the single-cycle dataflow graph makes it hard to parallelize across cycle boundaries because it assumes that registers are up-to-date at the start of each simulated cycle.

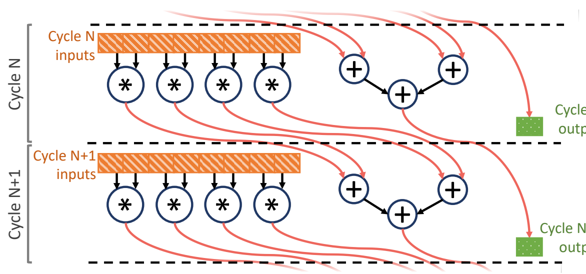


Figure 8: The unrolled dataflow graph turns registers into cross-cycle edges (shown in red) to expose more parallelism.

To overcome these limitations, we contribute the *unrolled dataflow graph*. This representation *turns registers into cross-cycle dataflow edges*. Fig. 8 shows this representation, distinguishing between **same-cycle edges** (wires) and **cross-cycle edges** (registers). This representation avoids write-after-read dependences and allows overlapping computation from different simulated cycles, exposing more parallelism.

4.3.2 Mapping and coarsening dataflow nodes. DASH first *partitions* the unrolled dataflow graph across tiles. Since sending arguments across the chip consumes bandwidth and energy, we try to both minimize cross-tile communication and balance work across partitions. In addition, nodes that access the same data in memory (e.g., an SRAM array, as explained in Sec. 4.1) are restricted to the same tile. Finally, we map nodes that incur (rare) system tasks, like I/O, to a fixed tile, which lets us use standard libraries despite the lack of cache coherence.

We use METIS [29] to perform this partitioning, minimizing the sum of cross-partition edge costs while keeping the sum of node costs per partitions roughly balanced. We estimate a node's cost as the number of instructions within it and an edge's cost as the bits of its argument descriptor.

The compiler then *coarsens* the partitioned graph, producing larger tasks from small dataflow nodes. We modify Verilator's merging pass (described in Sec. 2.2) to prevent merging tasks from different partitions, prioritize merging tasks outside the critical path, and limit task size.

After these passes, we have a *task dataflow graph* where each task is mapped to a tile, and memory accesses are local to a tile: *all cross-tile communication is through descriptors.*

4.3.3 Prioritizing tasks. DASH next assigns timestamps to tasks, prioritizing their execution within and across cycles. Within a cycle, each task has a *depth d*: the longest chain of tasks that produces an argument for the task, measured from the start of the cycle. The *cycle depth D* is the length of the deepest task chain in a cycle. For instance, assuming that each node in Fig. 8 is a task, the multipliers and initial adders have $d = 0$, the final adder has $d = 1$, and the cycle has $D = 2$. Each task instance is given $timestamp = D \cdot cycle + d$, where *cycle* is the current cycle.

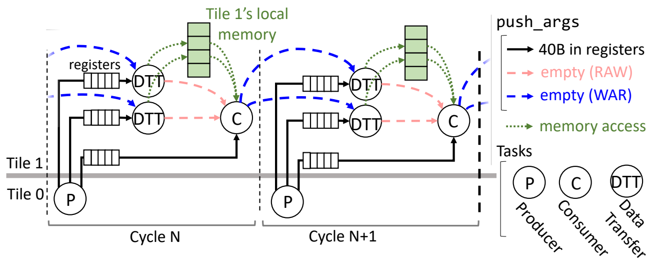


Figure 9: Example showing how DTTs transfer values that do not fit in register arguments from producers to consumers, and how consumers read them from memory.

4.3.4 Allocating arguments to registers and memory. DASH hardware allows each task to receive arguments in up to five 64-bit registers. In most cases, all arguments fit in registers; non-fitting values are read from memory.

If producer and consumer are on the same tile, the producer simply writes the non-fitting values to memory (at a pre-allocated location specific to the consumer task), and the consumer later reads them. But since ASH does not have global shared memory, when the producer is on a different tile, values are communicated through auxiliary *data-transfer tasks* (DTTs). Fig. 9 shows DTTs in action: a producer must pass a large 120-byte value. It passes 40 bytes directly to the consumer through arguments, and creates two DTTs to ship the remaining 80 bytes. Each DTT takes 40 bytes of arguments from the producer, and is sent to the consumer's tile, where it writes these bytes to a fixed memory location. Each DTT has a **dataflow edge** to the consumer task (with no arguments) to preserve RAW dependences.

ASH hardware also limits the number of direct parents (producer tasks) of each task (to 8 in our implementation). To address this, the compiler emits a *fan-in* task tree for tasks with many arguments. Similarly, if a produced argument has many consumers, the compiler emits a *fan-out* tree to pass these values in parallel and reduce inter-tile communication.

Storing some arguments in a single memory location introduces write-after-read (WAR) dependences (like we saw in Sec. 4.3.1 for single-cycle dataflow graphs). For instance, in Fig. 9, the *next-cycle*'s DTTs have a **WAR dependence** with the consumer. For simplicity, we add cross-cycle edges to the task graph to respect WAR dependences, using a `push_args` without arguments from the reader to the writer. We do this *selectively*, adding a WAR edge only if the dependence isn't already covered by existing edges (often, a chain of RAW edges already forces the writer to run after the reader).

The resulting task dataflow graph obeys hardware limits while preserving two invariants: it encodes all dependences as dataflow edges, and localizes all memory accesses.

Finally, the compiler generates C++ code for this task graph, producing a DASH program.

5 SELECTIVE DATAFLOW EXECUTION

DASH runs all tasks in the dataflow graph, but many tasks are ineffectual because only a fraction of the logic switches each cycle. SASH leverages low activity factors by running *only tasks whose inputs have changed* on each cycle.

SASH extends DASH hardware to skip inactive tasks and to run tasks *speculatively*, using optimistic parallelization as described in Sec. 2.3. We first describe the changes needed for selective execution, then those needed for speculation.

5.1 Selective execution

In SASH, parents only push arguments that are different from previous cycles. This requires simple hardware changes, shown in Fig. 10. First, a task must be able to detect a change in their produced output compared to previous cycles. Second, since each task may only receive *some* of its arguments, it must fill in the missing arguments with their previous values.

Ineffectual arguments are filtered: SASH statically allocates a per-task, hardware-managed, in-memory *output argument buffer* that holds the latest arguments the task has pushed. This buffer has finite size because we restrict each task to push at most 8 descriptors, and each push has at most five 64-bit registers. Each `push_args` reads the output buffer and compares the new and previous arguments. If they differ, the output buffer is updated and an argument descriptor is pushed to the ASB. If they match, no action is taken.

Task dispatch does not wait for all arguments: In SASH, every task that has received at least one descriptor is *ready*. The TMU's merge unit now does *not* store multiple tasks: it merges the descriptors for the lowest-timestamp task and dispatches it for execution.

Dispatched tasks fill missing arguments with old values: Since tasks with inactive parents have missing arguments, SASH maintains a per-task, in-memory *input argument buffer* that records the latest arguments of the task. When a task is dispatched, before starting execution, the input argument buffer is read to retrieve the task's missing arguments, and then updated with the latest values.

Handling in-memory arguments: When a task consumes inputs from memory (Sec. 4.3.4), two types of descriptors carry no arguments: those that encode read-after-write (RAW) dependences (e.g., between DTTs and consumer in Fig. 9) and those that encode WAR dependences (e.g., between consumer and next-cycle DTTs in Fig. 9). When a descriptor has no arguments, `push_args` takes an `isRAW` flag to distinguish them. Argumentless RAW descriptors are not filtered, as they encode a memory dependence (in this case, software checks whether the in-memory value is the same, and skips pushing the descriptor if so). WAR descriptors are used in speculative execution, described next.

5.2 Speculative execution

SASH dispatches tasks even if some of their arguments are missing. Given enough parallelism and appropriate timestamps, most missing arguments arise because their producer task was skipped or filtered them. But occasionally, an argument is missing because it is *late*, so using its old value is incorrect. To preserve correctness, SASH runs tasks speculatively: on a late-arriving argument, the task and all its consumers are aborted, and the task is re-executed with the new argument.

SASH implements speculative execution by adapting techniques from from Chronos [1] and Time Warp [26]. Note that, unlike most speculative architectures (e.g., transactional memory), SASH does

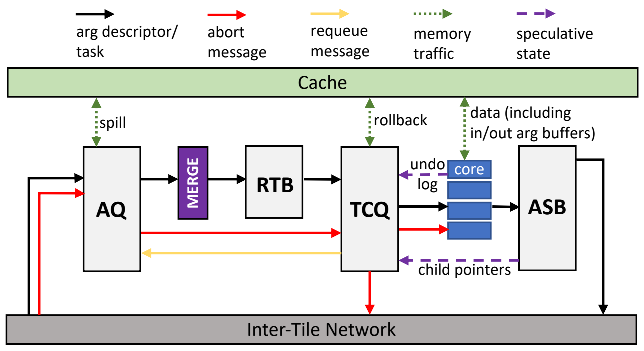


Figure 10: SASH tile.

not detect misspeculation through memory accesses. This results in much simpler hardware.

Speculation and rollback: SASH uses a per-tile *task commit queue* (TCQ) that holds the speculative state of uncommitted tasks. Each dispatched task is allocated an entry in the TCQ.

SASH detects conflicts when descriptors arrive at the tile. The TCQ is checked to find if a task with the same timestamp and function pointer was dispatched. If so, the task is aborted and requeued to the AQ, along with the new argument.

WAR descriptors are handled differently: they undergo conflict detection as described above, but are not enqueued in the AQ and are immediately discarded. This is because WAR descriptors do not encode a true data dependence, they ensure that writers do not clobber earlier readers. By conflict-checking but discarding the descriptor, we enforce its WAR dependence but prevent an ineffectual task from running.

SASH adopts eager versioning with undo logs to handle memory accesses, following Chronos's hardware-only implementation: on a store, the core updates memory in-place, and pushes the old value to a small in-memory undo log. If the task aborts, memory values are restored from the undo log. The undo log is discarded when the task commits. Note that updates to output and input argument buffers are also logged.

Finally, SASH implements cancellation of pushed argument descriptors as in Time Warp. When a task aborts, in parallel with restoring memory, its TCQ deletes each argument descriptor that the aborted task had pushed, sending messages to perform this deletion if the argument descriptor is in a remote AQ. If the argument descriptor was used by a dispatched task, that task instance is aborted. The AQ overflow mechanism from DASH is modified to use the spill policy from Chronos, spilling only argument descriptors whose parents are not speculative.

High-throughput commits: To make commits fast, each TCQ can buffer tens of finished tasks per core, and the TMU determines which tasks are safe to commit in bulk, amortizing commit overheads over many tasks. Specifically, we use the Virtual Time protocol from Time Warp: periodically (every 100 ns in our implementation) each tile sends the timestamp of its earliest unfinished task to a global arbiter, which finds the minimum such value across all tiles and broadcasts it. TCQs then commit all tasks with timestamps lower or equal than this minimum. We follow Chronos's Virtual Time implementation: tiles and arbiter communicate through the network, and each round of the protocol takes tens of cycles.

6 TASK-DRIVEN INSTRUCTION PREFETCHING

Beyond dataflow and selective execution, ASH includes a simple optimization to cope with high instruction footprints: large hardware designs execute multiple megabytes of code per simulated cycle. On conventional systems, this causes frequent instruction cache (icache) misses that limit performance [5].

To avoid this problem, we add a simple *task-driven instruction prefetching* mechanism that fetches the task's code into the core's icache ahead of execution. We add a small Ready Task Buffer (RTB) to each core. The task unit pushes tasks to this RTB eagerly, so the RTB holds the task that runs *after* the currently running task. The core starts fetching a task's code into the icache as soon as the RTB receives it. To do this accurately, we include the task function's size (up to 8 cache lines) in some of the unused high-order bits of the task pointer.

Task-driven instruction prefetching virtually eliminates icache stalls and avoids overfetching, because tasks have a few tens of instructions each and execute most of them (branches do not usually skip whole cache lines).

7 HARDWARE COSTS

We evaluate an ASH implementation with 256 scalar in-order cores—for RTL simulation, using many simple cores that are closely integrated is preferable to using fewer complex cores, as we will see later. We use a 2-level memory hierarchy with 1 MB tile-private L2s and quad-channel DDR5 memory, shown in Fig. 4 and with parameters given in Table 3.

Component	Area (mm^2)
256 cores	45.1
64×1MB L2s	39.3
4×Mem ctrl+PHY	25.0
64×SASH TMU	5.6
Total	**115.0**

Table 2: Area breakdown.

Table 2 shows the estimated area of the SASH system in a 7nm process, which takes a modest 115mm^2. We estimate core area using scaled-down Atom Bonnell cores [77], and other components are measured from Alder Lake-S [51] (built in Intel 7). We estimate SASH's area requirements (DASH would be simpler) by synthesizing its key components based on the Chronos open-source RTL [67] using yosys [73] on FreePDK45 [39] and scaling to 7nm. SASH adds 45 KB of state per tile and adds 4.8% to chip area, a modest overhead.

8 METHODOLOGY

We evaluate DASH and SASH using a simulator based on Swarm's simulator [2, 27, 76], which is execution-driven using Pin [36, 43]. We use detailed timing models for cores, caches, main memory, on-chip network, and all ASH features. We simulate all task and speculation overheads.

Baselines: We compare with Verilator running on two systems: a 32-core AMD Zen2 CPU (Threadripper 3975WX) at 3.5 GHz (also used in Sec. 2); and a simulated multicore with the same parameters in Table 3, but no ASH features and a shared, coherent LLC (instead of tile-private L2s).

Benchmark hardware designs: We simulate the four large, complex, and diverse hardware designs in Table 4:

Cores	256 scalar in-order cores in 64 tiles (4 cores/tile), 2.5 GHz, x86-64 (base) ISA, scoreboarded, stall-on-use, 8B-wide 2-stage ifetch, 2-level bpred with 512×10-bit BHSRs + 1024×2-bit PHT, 3-stage decode, 1-stage issue, 2-5-stage backend, 16-entry ld/st queue
L1 caches	16 KB L1D & 16 KB L1I per core, 8-way, 2-cycle latency
L2 caches	1 MB per tile, 16-way, inclusive, 9-cycle latency
NoC	16×16 mesh, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [72]), 2.5 GHz
DRAM	4 controllers at chip edges, 120-cycle minimum latency
ASB	64-entry FIFO per tile for new argument descriptors
AQ	512 entries per tile to hold unmerged argument descriptors
Merge	16-entries per tile to merge descriptors, DASH only
RTB	1 entry per core holding merged task for dispatch
TCQ	512 entries per tile for SASH to track speculating tasks, global Virtual Time computed every 100 ns

Table 3: Parameters of the evaluated systems.

(1) Vortex GPU system: We use the open-source Vortex GPU [68] with 32 cores using 2 lanes and 4 warps each, running an OpenCL kernel that performs vector addition.
(2) Chronos domain-specific accelerator: We use the Chronos open-source framework [67] to generate an accelerator for Dijkstra's algorithm for shortest weighted paths. We simulate a 128-PE system traversing a 128 × 128 grid.
(3) Chronos RISC-V manycore: To represent manycore systems, we use another Chronos configuration to generate 128 simple VexRiscv cores [60] instead of specialized PEs.
(4) NTT wide functional unit: We evaluate a pipeline that computes Number Theoretic Transforms (NTT). NTTs are similar in structure to FFTs but use modular arithmetic; NTT functional units are a key component of Fully Homomorphic Encryption accelerators [32, 52, 53]. We follow CraterLake's NTT unit design [53], which we implement from scratch. This unit performs 256-point NTTs using a 256-wide, 8-stage pipeline with 1024 modular multipliers.

These designs have diverse activity factors (Table 4) and are large, requiring multi-FPGA emulation (from 2 FPGAs for Chronos/PE, as we saw in Sec. 2.4, to 6 FPGAs for NTT).

RTL compiler implementation: Our implementation adds 12 K lines of code (LoC) to Verilator, and the whole compiler takes 86 K LoC of C++. We retain Verilator's fast compilation times: each design takes 7.2–140s to compile (Table 4).

Verilator is a full-featured Verilog/SystemVerilog simulator, which matches or outperforms state-of-the-art commercial simulators like VCS [57, 58]. Verilator has fewer features than commercial simulators, e.g., no support for 4-state logic, VHDL, or private IPs. However, these are not ASH limitations: ASH provides general-purpose cores, and could accelerate other simulators.

Sampling: Each RTL simulation runs for many instructions (often trillions), so we use a sampling-based approach following SimPoints [55]. First, we functionally execute the whole RTL simulation to record activity factors. Then, we select three cycle ranges representative of the full program. We fast-forward to the start of each range, warm up for 3 simulated cycles, and gather results for the remainder of the range. The methodology ensures that our results have the same activity factor as the full run. Even with sampling, each simulation runs many millions of instructions.

9 EVALUATION

9.1 ASH widely outperforms baseline systems

Table 5 reports simulation speeds. Each row reports results for a system, and each column shows the speed for a specific benchmark design. Speeds are in KHz, i.e., thousands of simulated cycles per second. The last rows report SASH's speedups over the baselines. Overall, SASH is 32× faster than Zen2 despite taking 3× less area than Zen2, and it is 21× faster than the simulated 256-core baseline.

Parallel Verilator has limited scalability, so for the baselines, Table 5 reports both their 1-core and best parallel speeds, showing the best thread count on each cell (e.g., Zen2's best Vortex result is with 12 threads). DASH and SASH always improve with more cores, so Table 5 reports only their 256-core speeds. The Zen2 system achieves a gmean speed of 19.6 KHz, whereas the simulated baseline multicore achieves 29.9 KHz. Note that, for the simulated results, smaller systems have fewer tiles and thus less LLC. Thus, at 1-core, Zen2 widely outperforms the simulated baseline mainly due to having much more cache; but on the best results, which have a similar amount of LLC, the simulated baseline outperforms Zen2. This happens because Verilator makes limited use of out-of-order cores: its large code footprint and poor branch predictor performance limit IPC below 1.0, so using simpler cores is preferable.

DASH and SASH achieve high simulation speeds: 263 KHz and 636 KHz gmean, respectively. DASH widely outperforms the baselines because it *(1)* exploits more parallelism; *(2)* avoids synchronization overheads due to hardware support; and *(3)* reduces memory stalls due to tile-private caches, distributed tasks, and prefetching. SASH further outperforms DASH due to selective execution. SASH's improvement over DASH is highly correlated with the design's activity factor (Table 4).

Looking across benchmarks, Table 5 shows that Vortex and the two Chronos variants achieve similar simulation speeds, e.g., around 10 KHz on Zen2. SASH has more pronounced differences across designs due to varying activity factors (Vortex's activity factor is only 7%, whereas Chronos/RV's is 15%). The NTT benchmark is quite different from the others. First, despite being the largest design in terms of area, NTT is the fastest to simulate and has a lower code footprint (Table 4), as it is dominated by arithmetic operations. This highlights a key advantage of simulation over emulation: on arithmetic-heavy or structured benchmarks, a single instruction (e.g., a multiply) can perform the functionality of many gates. In these cases, simulating RTL code using CPU instructions is especially advantageous over compiling it to gates and emulating it. Second, the NTT benchmark has a near-100% activity factor, representing a worst-case scenario for SASH vs. DASH. The fact that SASH and DASH perform similarly on this benchmark shows that SASH performs selective execution efficiently.

9.2 Architectural analysis

We now present more detailed data to understand the performance and efficiency of DASH and SASH, and evaluate the impact of our contributions.

Scalability: Fig. 11 shows the speedup of all simulated systems as we scale from 1 to 256 cores. Larger systems have more tiles and thus more total cache capacity (scaling this way keeps the area ratio between cores and cache fixed).

Name	Description	Nodes	Edges	Tasks	% DTTs	Task Edges	Parallelism	Activity Factor	1-core Cycles	Code Footprint	Compile Time
Vortex	Full-system GPU	245.5K	414.7K	86.3K	14.9 %	167.0K	672	7.1 %	8.6M	10.1 MB	70.6s
Chronos/PE	Graph accelerator	297.1K	635.0K	82.7K	57.4 %	207.0K	407	17.4 %	8.4M	16.2 MB	140.5s
Chronos/RV	RISC-V manycore	375.9K	733.2K	65.5K	51.3 %	184.0K	415	15.0 %	8.9M	13.7 MB	134.1s
NTT	Cryptographic accelerator	29.4K	41.7K	7.4K	0.0 %	11.8K	558	97.0 %	0.8M	0.7 MB	7.2s

Table 4: Main characteristics of the benchmark hardware designs.

System \ Design	Vortex	Chronos /PE	Chronos /RV	NTT	gmean
Zen2 t=1	3.8	3.1	2.8	101.0	7.6
Zen2 Best	t=24 14.8	t=16 10.7	t=12 9.2	t=1 101.0	19.6
Baseline t=1	0.2	0.3	0.2	2.6	0.4
Baseline Best	t=64 25.6	t=64 19.3	t=64 18.8	t=16 86.0	29.9
DASH 256-core	201.3	108.1	133.0	1,644.7	262.7
SASH 256-core	686.5	413.9	340.5	1,689.2	635.8
SASH/Zen2 Best	46.5×	38.6×	36.8×	16.7×	32.4×
SASH/Baseline Best	26.8×	21.5×	18.1×	19.6×	21.3×

Table 5: Workload simulation speeds (in KHz) and speedups.

Compared to serial Verilator running on one core, 256-core DASH is gmean 617× faster, and SASH is gmean 1,485× faster (up to 2,877× on Vortex).

Fig. 11 shows that speedup increases linearly throughout the range for DASH, showing that it leverages the plentiful parallelism in these benchmarks. SASH also shows good scalability, though the slope of the speedup curve drops somewhat between 128 and 256 cores for applications with a low activity factor (Vortex and the Chronos variants). This is because SASH performs less work per cycle, which tends to reduce dynamic parallelism (the ratio of active work to critical path).

Cycle breakdowns: Fig. 12 gives more insight into these results by showing how cores spend cycles for SASH systems of different sizes. Each plot reports results for a different benchmark, and each bar reports the number of cycles summed across all cores, normalized to the cycles of the 1-core system (lower bars are better, and a height of 1.0 denotes a speedup equal to the number of cores). Each bar is broken down into the cycles that cores spend *(1)* running tasks that later **commit**, *(2)* running tasks that later **abort**, and *(3)* **idle** because the AQ is empty or the TCQ is full.

Fig. 12 shows two key trends. First, the number of committed cycles *decreases* as the system size grows, with a sharp knee around 16 or 64 cores. This happens because L2 misses and memory stalls fall as the system grows: each tile executes a smaller fraction of the tasks, and the tasks' working set fits into the L2 caches for the larger systems. This causes IPC to grow from about 0.2 at 1 core to about 0.6 at 256 cores. Specifically, these models have both a large code footprint and a substantial data footprint; this speedup arises because the model's *data* starts fitting in-cache (instruction footprint is even larger, but those are prefetched, as we'll see next).

Second, Fig. 12 shows that most time is spent running committed tasks. Aborts are rare, and the slight scalability drop at 256 cores in Vortex and Chronos/RV stems from idle cycles due to lack of work.

Energy breakdowns: Fig. 13 shows the chip energy consumed by the 256-core DASH, SASH, and by the 256-core baseline system

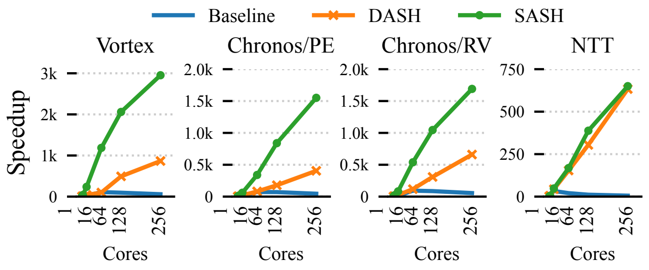


Figure 11: Speedups of DASH and SASH over serial Verilator on the 1-core simulated baseline as cores grow from 1 to 256.

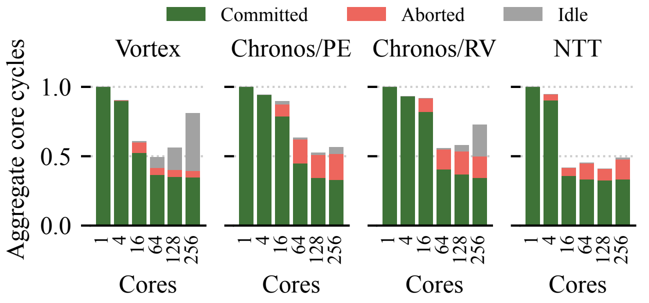


Figure 12: Breakdown of core cycles for SASH on 1–256 cores.

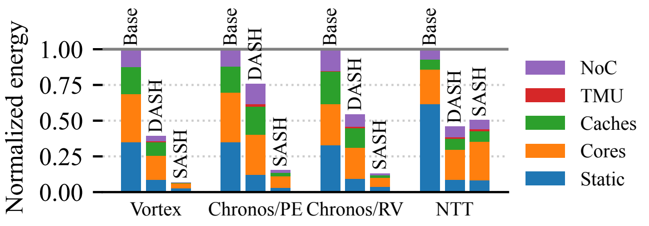


Figure 13: Energy breakdown for 256-core baseline, DASH, and SASH.

at its best-performing thread count (at 256 threads, the baseline is often dominated by spin-waiting and far less efficient). We model core and network energy with McPAT at 22 nm and scale to 7 nm. Caches are modeled at 7 nm using FinCACTI [54]. We also use these tools to model energy consumed by the components of the Task Management Unit (TMU, Sec. 4.2 and Sec. 5.1). Power stays under 60 W across all systems and applications.

DASH consumes less energy than the baseline, thanks to executing fewer instructions and completing faster (which reduces static energy). SASH's selective execution further reduces energy for all benchmarks but NTT, where the near-100% activity factor shows that speculative selective execution has modest energy costs. TMU costs are small, and most task-related energy is spent sending argument descriptors through the NoC.

9.3 Impact of ASH features

Prioritized dataflow: ASH's prioritized dataflow execution has two benefits over prior dataflow architectures: simplifying hardware and keeping memory footprint in check (Sec. 4.2). We now analyze these benefits.

Fig. 14 shows DASH's gmean performance as the number of entries in the merge unit grows, relative to the default (16 entries). While small merge units incur some degradation (e.g., 8 entries is 3% slower), the default is sufficient, and a merge unit with unbounded entries is only 3% faster. Thus, prioritized tasks enable dataflow execution with minimal resources (each tile's merge unit is much smaller than the issue queue of a modern OOO core).

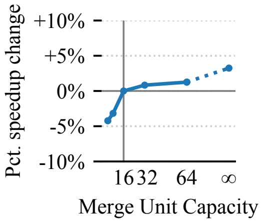


Figure 14: Sensitivity to DASH merge unit size.

Unlike in OOO cores, arguments arrive at a tile out of order, and a late-arriving task might evict a higher-timestamp task from the merge unit. Such evictions are rare: one per 500 dequeues on average (thousands of cycles elapse between evictions).

Fig. 15 compares the memory footprint of prioritized vs. unordered (conventional) dataflow. Unordered dataflow execution runs each task as early as its inputs become available. This causes tasks within short paths in the simulated design (the common case) to run ahead of critical paths, producing arguments that will not be consumed for a long time. Fig. 15 shows that this is a major problem: conventional dataflow increases average data footprint by 16.8× gmean and by up to 47× (Chronos/PE) over prioritized dataflow. With data footprint in the tens of megabytes (Table 4), this blowup would cause substantial main memory traffic, hindering performance.

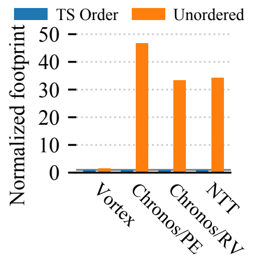


Figure 15: Prioritization reduces memory footprint.

Instruction prefetching: So far, all the results we have presented include task-driven instruction prefetching. Fig. 16 shows the speedup that prefetching achieves on SASH (DASH results are similar). Each bar shows gmean speedup across benchmarks for a specific core count. Smaller systems have greater need for prefetching because less code fits on chip. At 1 core, most code is evicted every simulated cycle. At 256 cores, Vortex, Chronos/PE, and Chronos/RV still have 3-6% of L1 instruction cache misses also miss in the L2, chiefly due to L2 cache conflicts. Thus, prefetching code offers speedups across system sizes (e.g., 1.9× at 256 cores). We observe a peak memory bandwidth of 114 GB/s at 256 cores (on Chronos/PE), with code making up the vast majority of memory traffic.

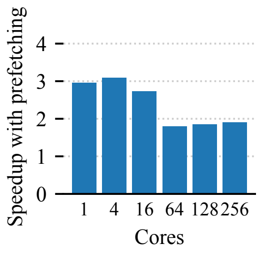


Figure 16: Speedup of task-driven instruction prefetching for SASH on 1–256 cores.

Queue utilizations: Fig. 17 reports the average occupancies of the per-tile Argument Queue (AQ) and Task Commit Queue (TCQ) on the 256-core SASH system. Each group of bars reports results for a single benchmark. Each structure holds up to 512 entries. AQ occupancy corresponds to the average number of in-flight arguments per tile, and TCQ occupancy is the average number of uncommitted tasks per tile. The modest AQ occupancy shows that prioritized dataflow execution works well: thanks to ordering tasks by timestamp, we execute the dataflow graph in an order that incurs a modest number of in-flight arguments that comfortably fit in the AQs (three of the designs never spill argument descriptors to memory at 256 cores, and Vortex has minimal spilling). The TCQ occupancy shows that tiles exploit out-of-order execution, running ahead of the earliest unfinished task to extract speculative parallelism.

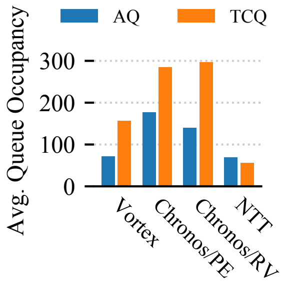


Figure 17: Average queue occupancies per benchmark in 256-core SASH.

Factor analysis: Fig. 18 shows gmean speedup vs. parallel Verilator as we add features, quantifying the benefits of our hardware and compiler techniques. *+hw df* uses hardware dataflow execution within each cycle. *+unroll* uses our unrolled dataflow graph instead, which passes register values through dataflow edges and increases parallelism. *+mapping* uses our task partitioning and coarsening algorithm instead of Verilator's; this configuration is DASH. Finally, *+selective* adds selective execution; this is SASH. Fig. 18 shows that each of our contributions yields substantial performance gains.

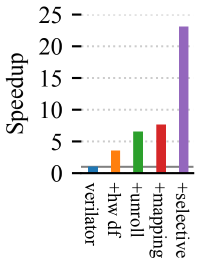


Figure 18: Factor analysis.

10 RELATED WORK

We first discuss prior architectures with task-level speculative and dataflow execution, and compare them with SASH and DASH, then discuss other simulation-related techniques.

10.1 Prior task-level speculative architectures

Chronos [1] and Swarm [27] are the closest architectures to SASH. Swarm is a shared-memory multicore with hardware support for small ordered atomic tasks. Tasks can access arbitrary memory, and cache coherence is used to detect conflicts, similar to other speculative systems like HTM [21, 22, 38] and TLS [47, 59, 61]. Chronos avoids cache coherence by mapping each task to a fixed tile and providing hardware support for mutual exclusion. These systems target graph analytics and other domains, including non-RTL event-driven simulation, as they are good match for Time Warp's [26] optimistic parallelization.

SASH's speculation support borrows heavily from Chronos (Sec. 5, Sec. 7). However, these systems are poorly suited for RTL simulation because *they do not support dataflow execution*: each task

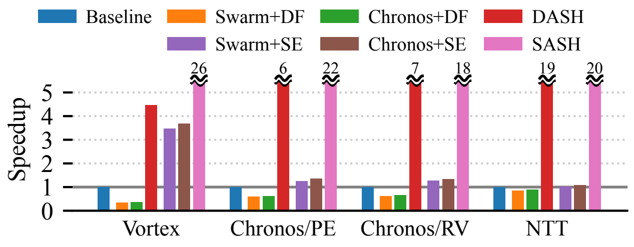


Figure 19: Speedup over best Verilator multicore run.

has a single parent and can run the moment it is created. This requires implementing dataflow execution in software, which adds substantial overheads.

To quantify these overheads, we simulate 256-core Swarm and Chronos systems with similar parameters to ASH. Since these systems lack hardware support for dataflow execution, we modify our compiler to do dataflow synchronization in software. Each producer spawns a task for the consumer. All tasks except the last write their arguments to memory and finish. The last task gathers all arguments and runs the consumer once. For selective execution, we perform output filtering in software.

Fig. 19 compares the performance of the baseline multicore, Swarm and Chronos with dataflow non-selective (DF) and selective (SE) execution, DASH, and SASH. Each group of bars shows, for one benchmark, the speedup of each scheme vs. the best-thread baseline. Swarm-DF is *slower* than the baseline, because the lack of dataflow support adds large overheads: instruction footprint grows by about 2×, and branches and memory accesses become more frequent, limiting IPC. Swarm-SE has limited improvements over Swarm-DF, and is significantly faster only in Vortex, where the activity factor is lowest. This happens because selective execution in software adds even more overheads. Chronos shows similar trends to Swarm: whereas Swarm uses a coherent cache hierarchy (like the multicore baseline), Chronos uses private caches and avoids coherence (like ASH); this makes Chronos somewhat faster than Swarm. But these systems lack *hardware support for dataflow execution, which is crucial* to accelerate RTL simulation: DASH/SASH are gmean 12.5×/12.9× faster than Chronos-DF/SE, respectively.

10.2 Dataflow architectures

Dataflow execution [15] runs operations as their inputs become available, instead of following a fixed operation sequence. Dataflow execution is a general principle to exploit fine-grain parallelism, and there is rich prior work applying it to a wide range of architectures [15, 17, 40–42, 44, 62, 75].

Task-superscalar [17] is perhaps the closest to DASH. It performs task-level dataflow execution, with dedicated on-chip structures to track in-flight arguments and dispatch tasks. As we have seen (Sec. 4.2), DASH's key distinguishing feature is *prioritized* dataflow execution: DASH gives each task a timestamp, and follows this priority at runtime. Sec. 9.3 showed that this addresses the key challenges of dataflow architectures. First, prior systems require large on-chip structures to track arguments, e.g., multi-megabyte eDRAMs in Task Superscalar [17], or add metadata to main memory as in I-Structures [4]. They also perform costly associative lookups to find ready operands. Instead, DASH performs dataflow execution on a tiny window of tasks (the number of merge unit entries). Second, unordered dataflow systems often produce many live in-flight arguments, sacrificing locality [3]. Sec. 9.3 showed order-of-magnitude footprint blowups in in-flight arguments, which ASH avoids.

10.3 Hardware emulators

Rizzatti [48–50] reviews the rich history of hardware emulation, including key technical and commercial developments. IBM's YSE [45] and EVE [9] were the first specialized architectures for emulation, including ASIC processors to simulate 4-input logic gates and storage elements. Early commercial emulators combined tens to thousands of off-the-shelf FPGAs [6, 30, 34], interconnected with a low-latency network, and software to partition and map large circuits across FPGAs. Current processor- and FPGA-based emulators, described in Sec. 2.4, all follow this template.

ASH fundamentally differs from hardware emulators in many ways: *(1)* Emulators synthesize RTL to *gates*, which forces the use of specialized processors or FPGAs; instead, ASH compiles RTL to general-purpose code, where each instruction can simulate many gates. *(2)* Emulators directly map gates to hardware elements so emulator size limits the size of the emulated system; instead, ASH simulates large systems by running tasks over time. Consequently, *(3)* emulators are large and expensive, with costs in the millions of dollars, whereas ASH systems can be small. *(4)* Due to these design choices, emulators suffer from long compilation times, as Sec. 2.4 showed. Finally, *(5)* emulators do not leverage selective execution, and cannot avoid ineffectual work.

10.4 Batched GPU-accelerated simulation

RTLFlow [35] accelerates RTL simulation on a GPU by *batching* different tests, i.e., running them in a single group. Batching enables using vectors to represent each signal (with each vector element dedicated to a different test), and makes simulation well-suited to a GPU. Batching works well on small circuits when running thousands of homogeneous tests: RTLFlow breaks even with Verilator when batching 1K simulations, and is up to 40× faster with 64K simulations.

However, RTLFlow has several limitations. First, large RTL designs have large amounts of state, and batching blows up that state, making simulation memory-bound and eventually overwhelming memory capacity. For example, the Chronos/PE design has a data footprint of 30MB (this includes registers, on-chip memories, and the active pages of simulated main memory, which we optimistically assume are the same across all batched simulations). Batching 1K simulations, RTLFlow's break-even point with Verilator, would require 30GB of memory, roughly the size of an A100 GPU; 64K simulations would require 1.9TB. Second, many use cases for RTL simulation don't use thousands of homogeneous tests. For instance, test programs often have vastly different lengths, and batching requires running for the duration of the longest program. And debugging an evolving RTL design often requires tracing a single long run. By exploiting parallelism within a simulation, ASH avoids the footprint issues of batching and supports use cases where simulation latency is important.

10.5 Other simulation accelerators

Prior work has also proposed several FPGA-based simulation accelerators, but these systems do not target RTL simulation. RAMP Gold [65] and FAME [66] simulate a multicore system by offloading core simulation to FPGAs, time-division multiplexing cores as needed, and use a high-level timing model for the memory system. RAMP Blue [33] emulates distributed message-passing architectures. These systems are restricted to multicore designs.

Diablo [64] and FireSim [28] emulate datacenter-scale systems and offer scalable multi-FPGA emulation. However, these systems target simulating large collections of loosely connected systems, and use conservative (CMB-style) parallelization. Thus, while these systems work well for scale-out systems (e.g., many simple in-order cores connected through Ethernet), they cannot simulate a large RTL model that does not fit in an FPGA. In addition, adapting existing designs to these systems requires manual changes to the RTL.

11 CONCLUSION AND FUTURE WORK

RTL simulation is a key bottleneck in chip design, and existing simulators and emulators have serious drawbacks. By leveraging classic parallelization approaches (dataflow and speculative execution) and through several novel techniques, we have dramatically accelerated RTL simulation while retaining its fast compilation time and flexibility.

DASH and SASH open exciting avenues for future work. Though we have focused on RTL simulation, our combination of dataflow and selective execution may be more broadly beneficial, both for other types of simulation (e.g., microarchitectural), and for workloads beyond simulation. Our compiler-based techniques may extend to these domains, making it possible to map their designs automatically. Also, we have focused on a single-chip system, but our techniques are general and should apply to other architecture styles. For example, ASH could be implemented on a multi-FPGA system using soft cores to accelerate simulation on existing infrastructure.

ACKNOWLEDGMENTS

We are grateful to all who have supported and given feedback on this work, including Serge Leef, Sung-Kyu Lim, James Wilson, Darrell Teegarden, Julian Warchall, Hyun Ryong (Ryan) Lee, Axel Feldmann, Yifan Yang, Nikola Samardzic, Xingran (Maggie) Du, Aleksandar Krastev, Nithya Attaluri, Clément Pit-Claudel, Thomas Bourgeat, and Lauro Rizzatti. This work was supported in part by DARPA under contract N00014-21-1-2960. Shabnam Sheikhha was supported in part by an MIT EECS graduate fellowship. This research was, in part, funded by the U.S. Government. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*.

[2] Maleen Abeydeera, Suvinay Subramanian, Mark C. Jeffrey, Joel Emer, and Daniel Sanchez. 2017. SAM: Optimizing Multithreaded Cores for Speculative Parallelism. In *Proc. of the 26th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-26)*.

[3] Arvind. 2005. Passing the token. In *Proc. of the 32nd annual Intl. Symp. on Computer Architecture (ISCA-32)*.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-Structures: Data Structures For Parallel Computing. *ACM TOPLAS* 11, 4 (1989).

[5] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proc. of the 46th annual Intl. Symp. on Computer Architecture (ISCA-46)*.

[6] Jonathan Babb, Russell Tessier, Matthew Dahl, Silvina Zimi Hanono, David M Hoki, and Anant Agarwal. 1997. Logic emulation with virtual wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 6 (1997).

[7] Scott Beamer. 2020. A Case for Accelerating Software RTL Simulation. *IEEE Micro* 40, 4 (2020), 112–119.

[8] Scott Beamer and David Donofrio. 2020. Efficiently exploiting low activity factors to accelerate RTL simulation. In *Proc. of the 57th Design Automation Conf. (DAC-57)*.

[9] Daniel K Beece, G Deiberg, Georgina Papp, and Frank Villante. 1988. The IBM engineering verification engine. In *Proc. of the 25th Design Automation Conf. (DAC-25)*.

[10] Ranjita Bhagwan and Bill Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. of the IEEE Infocom 2000.*

[11] Janusz A Brzozowski and Carl-Johan H Seger. 1995. *Asynchronous circuits*. Springer.

[12] Cadence. 2015. Palladium Z1 enterprise emulation platform. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/palladium-z1-ds.pdf, archived at https://perma.cc/MD6F-EYGQ.

[13] Cadence. 2019. Protium X1 enterprise prototyping platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/protium.html.

[14] K. Mani Chandy and Jayadev Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM* 24, 4 (1981).

[15] Jack B Dennis and David P Misunas. 1975. A preliminary architecture for a basic data-flow processor. In *Proc. of the 2nd annual Intl. Symp. on Computer Architecture (ISCA-2)*.

[16] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proc. of the 38th annual Intl. Symp. on Computer Architecture (ISCA-38)*.

[17] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2010. Task Superscalar: An Out-of-Order Task Pipeline. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-43)*.

[18] Richard Fujimoto. 1989. The virtual time machine. In *Proc. of the 1st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*.

[19] Richard Fujimoto. 1990. Parallel discrete event simulation. *Commun. ACM* 33, 10 (1990).

[20] Richard M. Fujimoto, Jya-Jang Tsai, and Ganesh C. Gopalakrishnan. 1992. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Trans. Comput.* 41, 1 (1992).

[21] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture (ISCA-31)*.

[22] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture (ISCA-20)*.

[23] William N.N. Hung and Richard Sun. 2018. Challenges in large FPGA-based logic emulation systems. In *Proc. of the 2018 Intl. Symp. on Physical Design (ISPD)*.

[24] IBS data on IC design costs. 2018. As Chip Design Costs Skyrocket, 3nm Process Node Is in Jeopardy. https://www.extremetech.com/computing/272096-3nm-process-node.

[25] David Jefferson. 1985. Virtual time. *ACM TOPLAS* 7, 3 (1985).

[26] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger, and Steve Bellenot. 1987. Distributed Simulation and the Time Warp Operating System. In *Proc. of the 11st Symp. on Operating System Principles (SOSP-11)*.

[27] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48)*.

[28] 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 Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system

simulation in the public cloud. In *Proc. of the 45th annual Intl. Symp. on Computer Architecture (ISCA-45).*

[29] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998).

[30] Ubaid R. Khan, Henry L. Owen, and Joseph L.A. Hughes. 1993. FPGA architectures for ASIC hardware emulators. In *Proc. of the Sixth Annual IEEE Intl. ASIC Conf. and Exhibit.*

[31] Donggyu Kim, Jerry Zhao, Jonathan Bachrach, and Krste Asanović. 2019. Simmani: Runtime power modeling for arbitrary RTL with automatic signal selection. In *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-52).*

[32] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *Proc. of the 49th annual Intl. Symp. on Computer Architecture (ISCA-49).*

[33] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz. 2007. RAMP Blue: A message-passing manycore system in FPGAs. In *Proc. of the 2007 intl. conf. on Field Programmable Logic and Applications (FPL).*

[34] Helena Krupnova and Gabriele Saucier. 2000. FPGA-based emulation: Industrial and custom prototyping solutions. In *Proc. of the 10th intl. conf. on Field-Programmable Logic and Applications (FPL).*

[35] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2023. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *Proc. of the 51st Intl. Conf. on Parallel Processing (ICPP-51).*

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Notices* (2005).

[37] Mentor/Siemens. 2017. Veloce Strato. https://eda.sw.siemens.com/en-US/ic/veloce/strato-hardware/.

[38] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: Log-based transactional memory. In *Proc. of the 12nd IEEE intl. symp. on High Performance Computer Architecture (HPCA-12).*

[39] Nangate Inc. 2008. The NanGate 45nm Open Cell Library. http://www.nangate.com/?page_id=2325.

[40] Rishiyur S. Nikhil and Arvind. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. on Computers* 39, 3 (1990).

[41] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44).*

[42] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous Von Neumann/dataflow execution models. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (ISCA-42).*

[43] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. 2005. Controlling Program Execution through Binary Instrumentation. *SIGARCH Computer Architecture News* (2005).

[44] Yale N Patt, Wen-mei Hwu, and Michael Shebanow. 1985. HPS, a new microarchitecture: Rationale and introduction. In *Proc. of the 18th annual workshop and symp. on Microprogramming and Microarchitecture (MICRO-18).*

[45] Gregory F Pfister. 1982. The Yorktown simulation engine: Introduction. In *Proc. of the 19th Design Automation Conf. (DAC-19).*

[46] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. 2021. Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers. In *Proc. of the 26th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI).*

[47] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. 2005. Thread-level speculation on a CMP can be energy efficient. In *Proc. of the Intl. Conf. on Supercomputing (ICS'05).*

[48] Lauro Rizzatti. 2015. Hardware emulation: Three decades of evolution. Part I. https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-11_issue-1/articles/stream/hardware-emulation-three-decades-of-evolution_vh-v11-i1.pdf, archived at https://perma.cc/F3DU-U6ZK. *Verification Horizons* 11, 1 (2015), 26–27.

[49] Lauro Rizzatti. 2015. Hardware emulation: Three decades of evolution. Part II. https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-11_issue-2/articles/stream/hardware-emulation-three-decades-of-evolution-part-II_vh-v11-i2.pdf, archived at https://perma.cc/XB4N-C7MS. *Verification Horizons* 11, 2 (2015), 40–42.

[50] Lauro Rizzatti. 2015. Hardware emulation: Three decades of evolution. Part III. https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-11_issue-3/articles/stream/hardware-emulation-three-decades-of-evolution-part-iii_vh-v11-i3.pdf, archived at https://perma.cc/BK4D-NAJX. *Verification Horizons* 11, 3 (2015), 15–18.

[51] Efraim Rotem, Yuli Mandelblat, Vadim Basin, Eli Weissmann, Arik Gihon, Rajshree Chabukswar, Russ Fenger, and Monica Gupta. 2021. Alder Lake Architecture. In *IEEE Hot Chips 33 Symposium (HotChips-33).*

[52] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-54).*

[53] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data.. In *Proc. of the 49th annual Intl. Symp. on Computer Architecture (ISCA-49).*

[54] Alireza Shafaei, Yanzhi Wang, Xue Lin, and Massoud Pedram. 2014. FinCACTI: Architectural analysis and modeling of caches with deeply-scaled FinFET devices. In *Proc. of IEEE Computer Society Annual Symposium on VLSI (ISVLSI).*

[55] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X).*

[56] Wilson Snyder. 2003. Verilator. https://www.veripool.org/verilator/.

[57] Wilson Snyder. 2018. Verilator 4.0: Open Source Simulation Goes Multithreaded. In *The Open Source Digital Design Conference (ORConf).*

[58] Wilson Snyder. 2020. Verilator, Accelerated. In *2nd Workshop on Open-Source Design Automation (OSDA).*

[59] Gurindar S Sohi, Scott E Breach, and TN Vijaykumar. 1995. Multiscalar processors. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture (ISCA-22).*

[60] SpinalHDL. 2018. A FPGA friendly 32 bit RISC-V CPU implementation. https://github.com/SpinalHDL/VexRiscv.

[61] J. Gregory Steffan and Todd C. Mowry. 1998. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proc. of the 4th IEEE intl. symp. on High Performance Computer Architecture (HPCA-4).*

[62] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The WaveScalar architecture. *ACM Transactions on Computer Systems (TOCS)* 25, 2 (2007).

[63] Synopsys Inc. 2018. ZeBu Server 4. https://www.synopsys.com/verification/emulation/zebu-server.html.

[64] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanovic, and David Patterson. 2015. DIABLO: A warehouse-scale computer network simulator using FPGAs. In *Proc. of the 20th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX).*

[65] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. 2010. RAMP Gold: an FPGA-based architecture simulator for multiprocessors. In *Proc. of the 47th Design Automation Conf. (DAC-47).*

[66] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. 2010. A case for FAME: FPGA architecture model execution. In *Proc. of the 37th annual Intl. Symp. on Computer Architecture (ISCA-37).*

[67] The Chronos FPGA Framework to accelerate ordered applications. 2020. https://github.com/SwarmArch/chronos/.

[68] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-54).*

[69] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967).

[70] Ray Turner. 2004. A primer on processor-based emulation. EETimes, https://www.eetimes.com/a-primer-on-processor-based-emulation/.

[71] Haoyuan Wang and Scott Beamer. 2023. RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning. In *Proc. of the 28th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII).*

[72] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-chip interconnection architecture of the Tile Processor. *IEEE Micro* 27, 5 (2007).

[73] Claire Wolf. 2014. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[74] Xilinx. 2019. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[75] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. 2013. Hybrid Dataflow/Von-Neumann Architectures. *IEEE Trans. on Parallel and Distributed Systems* (2013).

[76] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. 2020. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. In *Proc. of the 47th annual Intl. Symp. on Computer Architecture (ISCA-47).*

[77] Rumi Zahir. 2012. Medfield smartphone SOC Intel® Atom Z2460 processor. In *IEEE Hot Chips 24 Symposium (HotChips-24).*