Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks

Shunning Jiang Berkin Ilbeyi Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{ sj634, bi45, cbatten }@cornell.edu

ABSTRACT

Modern high-level languages bring compelling productivity benefits to hardware design and verification. For example, hardware generation and simulation frameworks (HGSFs) use a single "host" language for parameterization, static elaboration, test bench generation, behavioral modeling, and simulation. Unfortunately, HGSFs often suffer from slow simulator performance which undermines their potential productivity benefits. In this paper, we introduce Mamba, a new Python-based HGSF that co-optimizes both the framework and a general-purpose just-in-time compiler. We conduct a quantitative comparison of Mamba vs. traditional and emerging hardware development frameworks across both simple and complex designs. Our results suggest Mamba is able to match the performance of commercial Verilog simulators and is 10× faster than existing HGSFs while still maintaining the productivity of using a high-level language in hardware design.

1 INTRODUCTION

The increasing complexity of modern hardware has motivated design teams to augment or even replace traditional domain-specific hardware description languages (HDLs) with high-level general-purpose programming languages. The hope is that high-level languages can reduce time-to-solution by improving the productivity of design and verification. These approaches include: *high-level synthesis* (HLS), where a software-oriented program written in a high-level language is automatically *synthesized* into a low-level HDL implementation [10]; and *hardware generation*, where a hardware-oriented declarative or procedural description written in a high-level language is used to explicitly *generate* a low-level HDL implementation. Both approaches use powerful general-purpose language features to improve productivity including: strong static type systems and/or flexible dynamic type systems; object-oriented, generic, and functional programming paradigms; reflection and introspection; lightweight syntax; and rich standard libraries. While both approaches show promise, our focus in this work is on improving methodologies for hardware generation.

Early work in hardware generation focused on developing *hardware preprocessing frameworks* (HPFs) which use an ad-hoc intermingling of a high-level language and a low-level HDL (e.g., Scheme mixed with Verilog in Verischemelog [15], Perl mixed with Verilog in Genesis2 [23]). Unfortunately, mixed-language HPFs create a semantic gap, since they require simultaneously designing, verifying, and analyzing designs written in a high-level language (for parameterization, static elaboration, test bench generation) and a low-level HDL (for behavioral modeling). In an HPF, the high-level language usually uses basic string processing and is unaware of hardware semantics. True *hardware generation frameworks* (HGFs) address this semantic gap by completely embedding parameterization, static elaboration, test bench generation, *and* behavioral modeling in a unified high-level "host" language (e.g., Haskell in Lava [7], standard ML in HML [16], Scala in Chisel [4], Python in Stratus [5], PHDL [18]). However, HGFs must still generate a low-level HDL implementation for simulation, which prolongs the development cycle and creates a new kind of semantic gap between the high-level host language and the low-level HDL simulation. HDL simulation means designers are limited in the host-language features they can use for online debugging, instrumentation, and profiling. Designers must either manually write test benches in the low-level HDL or use a limited "generator-friendly" subset of the host language to implement test benches. These challenges have inspired completely unified hardware generation and simulation frameworks (HGSFs) where parameterization, static elaboration, test bench generation, behavioral modeling, *and* a simulation engine are all embedded in a general-purpose high-level language (e.g., Java in JHDL [6], Haskell in CλaSH [3], Python in MyHDL [11], PyRTL [9], Migen [19], PyHDL [13]). Our previous work on PyMTL demonstrated the potential for a Python-based HGSF to improve the productivity of hardware design and verification [17].

Section 2 compares the simulation performance of traditional HDLs, state-of-the-art HGFs, and emerging HGSFs. Our results suggest that while HGSFs can close the semantic gap present in other approaches, HGSFs also suffer from significantly slower simulation performance. For both small and large designs, highly optimized HGSFs are still typically 10× slower than HDL simulation. The highest performing HGSFs use: (1) general-purpose just-in-time (JIT) compilers that are not optimized for HGSFs [8]; or (2) highly specialized JIT-compiled simulators driven from the host language [9, 17]. Unfortunately, these techniques cannot completely close the performance gap, and/or they reintroduce the semantic gap (i.e., mixing the host language and JIT-compiled simulation). This in turn undermines the productivity benefits of using an HGSF.

In this paper, we introduce Mamba, a new version of PyMTL that has been carefully designed to close the performance gap in productive hardware development frameworks. *Our key insight is the need to deeply co-optimize the HGSF and the underlying general-purpose JIT compiler.* Section 3 provides background on state-of-the-art meta-tracing JIT compilers. Section 4 describes several novel JIT-aware HGSF as well as HGSF-aware JIT techniques, and then quantitatively compares their impact on multiple designs. Section 5 compares RISC-V single- and multi-core designs implemented using Verilog, PyMTL [17], and Mamba. Our results suggest Mamba is able to match the performance of commercial HDL simulators and

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196073

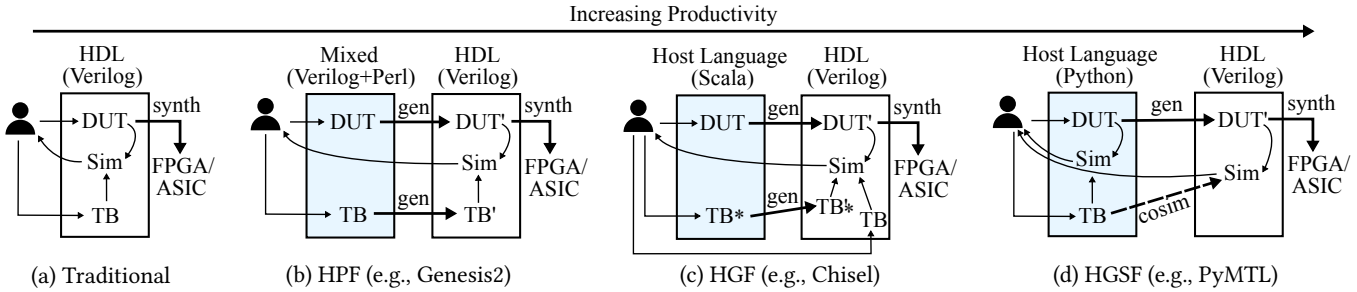


Figure 1: Qualitative Comparison of Hardware Development Workflows – (a) traditional HDL (e.g., Verilog); (b) hardware preprocessing frameworks (HPF) mix a high-level language and a low-level HDL (e.g., Perl mixed with Verilog in Genesis2 [23]); (c) hardware generation frameworks (HGF) use a single host language for parameterization, static elaboration, TB generation, and behavioral modeling, but must still use low-level HDL-based simulation (e.g., hardware generation with Scala in Chisel [4]); (d) hardware generation and simulation frameworks (HGSF) also use a simulation engine in the host language (e.g., hardware generation and simulation with Python in PyMTL [17]). DUT = design under test; DUT' = generated DUT; TB = test bench; TB* = TB with limited functionality; TB' = generated TB; Sim = simulation.

is 10× faster than existing HGSFs even when simulating more complex designs. While this paper explores these techniques within the context of Mamba, our work also sheds light on performance optimization opportunities in other HGSFs.

This paper makes three technical contributions: (1) we describe five JIT-aware HGSF techniques (static scheduling, schedule unrolling, heuristic topological sort, trace breaking, and block consolidation); (2) we describe two HGSF-aware JIT techniques (support for performance-critical data types and huge loops); and (3) we introduce Mamba which uses these techniques to match the performance of commercial HDL simulators while maintaining the productivity benefits of using a single host language for parameterization, static elaboration, test bench generation, behavioral modeling, and simulation.

2 COMPARISON OF HARDWARE DEVELOPMENT WORKFLOWS

In this section, we qualitatively and quantitatively evaluate four different kinds of hardware development workflows (see Figure 1). Our quantitative evaluation uses a 64-bit radix-4 iterative divider implemented at the register-transfer level (RTL) in six different hardware development frameworks (Verilog, Chisel [4], MyHDL [11], PyMTL [17], PyRTL [9], Migen [19]) with different kinds of simulators (e.g., ahead-of-time compiled, interpreted, JIT compiled). To ensure an apples-to-apples comparison, we implemented the iterative divider in each framework in a very similar way using a structural datapath and finite-state-machine control unit. Figure 2 shows the performance of simulating the divider using identical random inputs for 1B cycles assuming the divider is busy: (1) 100% of the time; and (2) only 10% of the time.

Hardware Description Languages – Figure 1(a) shows a traditional HDL workflow where the designer: manually writes both the design under test (DUT) and test bench (TB) in Verilog; compiles the DUT and TB into a simulator; uses the simulator to iteratively verify and evaluate the DUT; and eventually pushes the DUT through an FPGA/ASIC toolflow. Figure 2(a) shows the simulator performance of the hand-written Verilog for the iterative divider. CVS[1], one of the fastest commercial Verilog simulators, achieves 1.2–2.9M simulated cycles/second (CPS). Icarus, an open-source event-driven Verilog simulator [14], is well known to be relatively slow, however, Verilator, an open-source tool for translating synthesizable Verilog into a compiled C++ simulator [24], achieves an impressive 15–18M CPS. Verilator requires C++ TBs and significantly longer compile times on larger designs (e.g., several minutes); it is more often used for virtual prototyping as opposed to iterative development.

Hardware Preprocessing Frameworks – Figure 1(b) shows an HPF workflow using Genesis2 [23] where the designer: writes the DUT and TB in a mix of Perl and Verilog; uses Perl to preprocess the DUT and TB into pure Verilog; and then transitions to the traditional HDL workflow. HPF workflows have similar simulator performance to HDL workflows since they use the exact same HDL simulators. A critical weakness of HPFs is that the high-level language acts as a simple text preprocessor without any understanding of hardware semantics. As shown in Figure 1(b), the *iterative development cycle* (i.e., designer → DUT → generated DUT → simulation → designer) stretches across two languages further increasing the semantic gap.

Hardware Generation Frameworks – Figure 1(c) shows an HGF workflow using Chisel [4] where the designer: writes the DUT and TB in Scala using the Chisel library; executes the Scala program to generate a Verilog DUT and TB; and then transitions to the traditional HDL workflow. A key feature of HGFs is the ability to completely embed parameterization, static elaboration, TB generation, *and* behavioral modeling in a unified high-level "host" language. However, Chisel only supports rather limited "generator-friendly" TBs, so designers often manually write more sophisticated Verilog TBs. HGF workflows can also create a potentially frustrating semantic gap by stretching the iterative development cycle across multiple languages (e.g., Scala and Verilog). Figure 2(b) shows the simulator performance of the Chisel-generated Verilog for the iterative divider, which is comparable to HDL workflows as expected.

Hardware Generation and Simulation Frameworks – Figure 1(d) shows an HGSF workflow using PyMTL [17] where the designer: writes the DUT and TB completely in Python using the PyMTL library; uses Python-based simulation to verify and evaluate the DUT; and only transitions to the traditional HDL workflow to push the DUT through an FPGA/ASIC toolflow. A key feature of HGSFs is the ability to use a simulation engine written in the host language to drastically reduce the iterative development cycle and eliminate any semantic gap. The designer avoids crossing any language boundaries for development, testing, and evaluation, and can use the complete expressive power of the host language for

[1]Tool vendor anonymized due to license agreement.

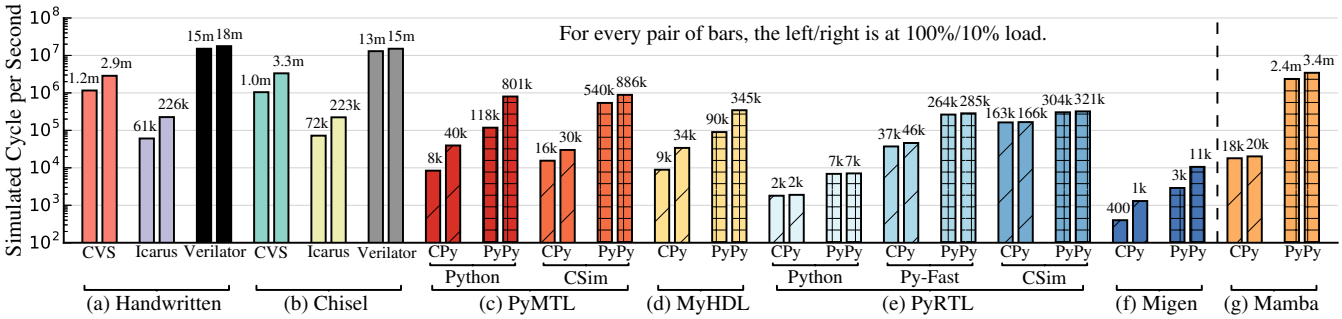


Figure 2: Quantitative Comparison of Hardware Development Workflows – Simulator performance for a 64-bit radix-4 iterative divider implemented at the register-transfer level. Results for identical random inputs for 1B cycles assuming the divider is active: (1) 100% of the time; and (2) only 10% of the time. Chisel = Chisel-generated Verilog; Handwritten = hand-written Verilog; CVS = commercial Verilog simulator; CSim = hybrid C/C++ compiled simulation; CPy = CPython. See Section 5 for details on the simulation platform.

verification, debugging, instrumentation, and profiling. Figure 2(c) shows the simulator performance of PyMTL for the iterative divider. Simulation using CPython, the reference Python interpreter, is 150× slower than CVS at 100% load. PyMTL uses an event-based simulator that dynamically schedules combinational blocks using an event queue so the average work per cycle is reduced under light load. PyMTL can improve performance by 14–20× using PyPy, a state-of-the-art JIT compiler for general-purpose Python programs [8]. PyMTL can further improve performance by translating RTL designs into Verilog, translating this Verilog into C++ with Verilator, compiling this C++ into a shared library, and then dynamically linking this library into the original PyMTL program. Overall, PyMTL is able to close the performance gap to less than 10× on this small design, although Section 5 suggests slowdowns of ≈10× are more reasonable for larger designs. Figure 2(d–f) shows the simulator performance of MyHDL [17], PyRTL [9], and Migen [19]. These Python-based HGSFs have their own unique approach to hardware modeling, but all three have dismal performance with CPython and relatively low performance even with PyPy. PyMTL and PyRTL's support for specialized JIT-compiled simulators produces modest performance improvements but also begins to reintroduce the semantic gap by requiring designers to at least on some level interact with multiple languages.

Other Approaches – SystemC [21], a set of C++ classes and macros for system-level design, is also an HGSF, but uses a less productive high-level language compared to Python-based HGSFs. As a result, SystemC is usually used for behavioral simulation and HLS, as opposed to RTL modeling and hardware generation, which is the focus of this work. Bluespec [20] uses a very different approach that combines a new HDL based on guarded atomic actions, limited HLS, and powerful static elaboration mechanisms. This work focuses on less radical approaches to improving the productivity of more traditional RTL design flows.

3 BACKGROUND ON META-TRACING JITS

Many of the high-level programming languages used in HGSFs are *dynamic languages*. These languages typically include: lightweight syntax; dynamic typing of variables; managed memory and garbage collection; rich standard libraries; interactive execution environments; and advanced introspection and reflection capabilities. These features are critical to the implementation of productive HGSFs, but these features are also the root cause of low HGSF performance. These languages traditionally use interpreters to implement a virtual machine that closely aligns with the language semantics, but as seen in Figure 2(c-f), interpreted code can be many orders-of-magnitude slower than statically compiled code. Dynamic languages use JIT-optimizing virtual machines to apply ahead-of-time (AOT) compiler techniques at run-time. Co-optimizing the HGSF and the JIT is the key to achieving peak performance while maintaining HGSF productivity benefits. In this paper, we co-optimize the HGSF and the PyPy meta-tracing JIT for Python [1, 8].

Tracing JITs – Tracing JITs start by interpreting the program and profiling the executed code to find frequently executed loops. Upon identifying a *hot loop*, the interpreter records the *trace* of the executed operations of one loop iteration. For better performance through type specialization, the trace also includes the concrete types of variables that were observed as the trace was recorded. This trace is then fed to the optimization engine to generate efficient machine code. Note that the trace is sequential and represents only one of the many possible paths. To ensure correctness, *guards* are placed at every possible point where another code path is possible, e.g., at conditional branches in the executed program or type checks to ensure the actual types match the recorded types. When a guard fails the execution immediately falls back to the interpreter and a new path may be traced and compiled starting from the failing guard if the guard has failed many times. A *bridge* is used to connect the original and new traces.

Meta-Tracing JIT – PyPy uses a meta-tracing JIT approach to build its tracing JIT compiler. Unlike a traditional tracing JIT compiler that records the executed operations in the application, the meta-tracing JIT compiler records the operations performed by the *interpreter as it interprets the application*. This approach separates the complicated JIT compiler machinery from the interpreter implementation and allows easily re-targeting the JIT compiler for other application languages or extensions. In PyPy's case, the interpreter is described in a statically typed subset of the Python language called RPython, and the RPython toolchain will *automatically* attach the meta-tracing JIT compiler to the interpreter. See [1, 8] for more details on the PyPy meta-tracing JIT.

JIT Warm-Up – A JIT compiler can spend significant time interpreting, analyzing, and tracing various code paths before actually generating JIT-compiled machine instructions for a frequently executed loop. This initial *JIT warm-up* is a key source of overhead, although hopefully this overhead is amortized by spending most of the steady-state execution time in JIT-compiled code.

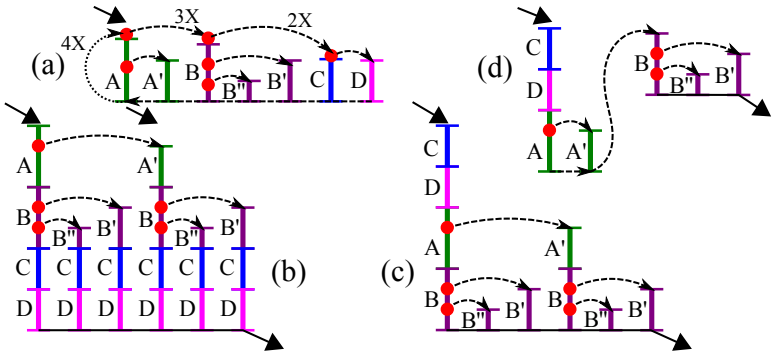


Figure 3: Meta-Traces of One Simulated Cycle – (a) event driven and static scheduling; (b) schedule unrolling; (c) heuristic topological sort; (d) trace breaking. A,B,C,D = traces of update blocks; red dots = guards that have bridges compiled from (connected by dashed arrows); A',B',B" = conditional paths in update blocks that result in bridges; 4X,3X,2X = how many times the jump occurs in a simulated cycle; solid arrows = entry from and exit to the cycle loop.

Technique	Divider	1-Core	32-core
Event-Driven	24K CPS	6.6K CPS	65 CPS
JIT-Aware HGSF			
+ Static Scheduling	13×	2.6×	1.1×
+ Schedule Unrolling	16×	24×	0.2×
+ Heuristic Toposort	18×	26×	0.3×
+ Trace Breaking	19×	34×	1.5×
+ Consolidation	27×	34×	42×
HGSF-Aware JIT			
+ RPython Constructs	96×	48×	61×
+ Support Huge Loops	96×	49×	67×

Table 1: Mamba Performance – The baseline is event-driven simulation in Mamba. Each row adds a new technique upon all previous ones. All results are with PyPy. CPS = simulated cycles per second.

4 THE MAMBA HGSF

Like earlier versions of PyMTL, Mamba supports behavioral modeling using concurrent structural composition, positive-edge-triggered update blocks, and combinational update blocks. Mamba's syntax is similar in spirit to PyMTL. Mamba also leverages Python's built-in reflection features, particularly abstract-syntax-tree (AST) self-parsing, to determine which variables are read or written in an update block. Sensitivity information is constructed based on readers/writers of the same variable in different blocks.

Figure 2(g) shows the simulator performance of Mamba for the iterative divider. At 100% load, Mamba is 2× faster than CVS, 20× faster than PyMTL, and 8× faster than PyRTL. The key to Mamba's performance is its co-optimization of the HGSF and JIT which results in a speedup of 124× speedup over CPython. Table 1 lists the five JIT-aware HGSF techniques and the two HGSF-aware JIT techniques and reports the incremental performance improvement of each technique. Table 1 includes results for the iterative divider from Section 2 and a simple single- and multi-core RISC-V design described in more detail in Section 5.

4.1 JIT-Aware HGSF

As a starting point, we implemented event-driven simulation in Mamba using a very similar technique to PyMTL. Table 1 shows the performance of event-driven Mamba simulation for the iterative divider. Like PyMTL, we use two nested loops: an outer loop for simulated cycles, and an inner loop over an event queue of combinational update blocks. Because each iteration of the inner loop is a different update block, the tracing JIT compiles a different trace for each of these update blocks. The tracing JIT will then insert a guard at the beginning of each trace to check if that trace is compiled for the called update block. Figure 3(a) illustrates this scenario using a cartoon representation of traces, guards, and bridges. Unfortunately, these guards create a pathological chain of bridges for the inner loop. Executing the n-th compiled update block will result in failing the first $n-1$ guards. In other words, the number of guard failures in an entire simulated cycle scales *quadratically* with the total number of update blocks, which becomes the scaling bottleneck. Small traces for each individual update block also prevents the compiler from performing escape analysis to remove unnecessary memory operations. Finally, enqueuing dependent blocks only when a signal's value changes requires an extra data-dependent check after every assignment. So while event-driven simulation can be efficient when most signals are stable, it can also create a perfect storm of challenges for tracing JITs. The JIT-aware HGSF techniques described in this section help mitigate many of these challenges.

Static Scheduling – Instead of event-driven simulation, Mamba statically schedules update blocks. While static scheduling has been shown to improve the performance of C++-based simulation frameworks [12, 22], we argue that static scheduling is particularly important in Python-based HGSFs for two reasons: (1) static scheduling avoids bridges due to data-dependent checks on every signal assignment; and (2) static scheduling paves the way for using additional techniques to increase the length of each trace. The Mamba execution semantics require each update block to be executed exactly once in each cycle. This enables a static fixed-order linear schedule to be generated at elaboration time. We leverage the sensitivity information to schedule the update blocks correctly: an update block that writes x should be scheduled before all blocks that read x. We use a topological sort to serialize the dependency graph into a total order of blocks. The topological sorting can succeed only if the directed graph is acyclic (DAG). Thus designers must not create inter-dependencies between combinational blocks. The inner loop simply iterates over the static schedule. Note that this does not change the meta-trace patterns in Figure 3(a); this simply changes the way in which execute the corresponding update blocks. Table 1 shows that this approach improves the performance by 1.1–13× over event-driven simulation. The concern for static scheduling is that all update blocks are executed regardless of activity. However, a tracing JIT can still optimize a hot path used under light load to improve performance. As shown in Figure 2, Mamba is 1.5× faster under 10% load vs. 100% load.

Schedule Unrolling – Static scheduling makes it possible to eliminate the pathological chain-of-bridge pattern in the inner loop by unrolling this loop into a sequence of update block calls. Table 1 shows that this improves performance by 1.2–9× compared to static scheduling without inner-loop unrolling for the divider and the 1-core design. Figure 3(b) illustrates how static scheduling and schedule unrolling get rid of the chain-of-bridge pattern but increase the overall trace length.

Heuristic Topological Sort – Unfortunately, schedule unrolling can create an *exponential number of bridges* due to data-dependent control flow within each update block. Every code path permutation due to control flow in update blocks (A/A' and B/B'/B" in Figure 3(b)) can create a new bridge. In other words, schedule unrolling introduces a new pathological pattern that can lead to serious performance degradation in larger designs (see 32-core in Table 1). To address this problem, we observe that there are multiple valid topological sorts for any given DAG, and each ordering can produce different guard/bridge behavior in our scenario. For example, Figure 3(c) illustrates an ordering with fewer guards and bridges (and a smaller instruction-cache footprint) compared to Figure 3(b). We use a heuristic to schedule update blocks with potentially more guards as late as possible. The stack used in the topological sort is replaced with a priority queue where each update block's priority is the number of `if`/`elif` statements in that block counted using AST self-parsing. Table 1 shows that this can improve the performance by 10–30% over basic schedule unrolling.

Trace Breaking – A large number of guards and bridges is still possible in more complex designs. To further control the number of guards and bridges, we use Python-level JIT hints to break long traces into multiple smaller traces. These application-level hints are provided by PyPy to control the JIT compilation process, and they can be used to prevent tracing in certain parts of the application. During the topological sort, we pack update blocks into a *meta-update block*. A meta-update block is a sequence of one or more update blocks that do not include any `if`/`elif` statements followed by a final update block which does include an `if`/`elif` statement. A meta-update block ends with a trace-breaking hint. This technique essentially limits the number of `if`/`elif` statements within any given trace (see Figure 3(d)). Table 1 shows this technique has a more significant impact on larger designs, e.g., improving performance by 5× for the 32-core design over heuristic topological sort.

Block Consolidation – Despite the techniques described above, the size of JIT-compiled code scales with the design size due to the nature of JIT compilation: the same update block from different instances is JIT-compiled individually. This problem is less prominent in static languages because different instances of the same module will likely reuse the same compiled assembly code. Block consolidation is a new technique that deduplicates different instances of an update block in a JIT trace. We modify the topological sort to identify different instances of the same update block and to then schedule these instances together. We group them into a new nested loop that iterates over these different instances by calling the same update block with different parameters in each iteration. Table 1 shows that large designs can significantly benefit from block consolidation, e.g., improving performance by 28× for the 32-core design over trace breaking.

4.2 HGSF-Aware JIT

The previous section described techniques to improve the performance of an HGSF when using a general-purpose meta-tracing JIT. In this section, we describe two techniques to improve performance by making the JIT specialized for the HGSF.

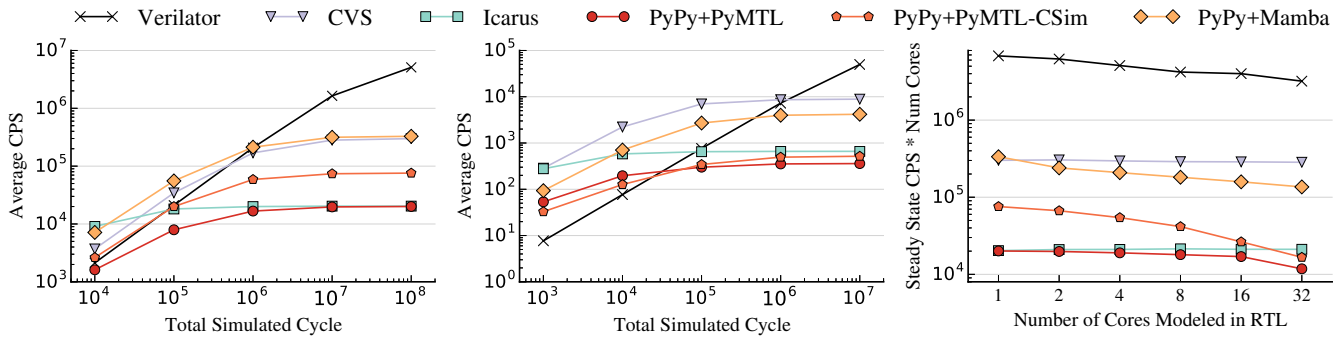
Meta-Tracing the Performance-Critical Constructs – Although the PyPy JIT compiler can run arbitrary Python code, native Python constructs may not be the best fit for HGSFs. For example, fixed-bit-width data types are used extensively in HGSFs, but they are not natively supported by Python. HGSF designers must emulate slicing and two's complement arithmetic using integer arithmetic. This increases warm-up time, requires redundant arithmetic operations, and creates excessive bridges due to dynamic type casting. We implement a fixed-bit-width data type in RPython as a proof of concept. Other performance-critical constructs (e.g., byte-addressable memory) can also be implemented in RPython. The key is the meta-tracing approach that enables writing Python-like code exactly once. We exploit the invariant that the bit-width of a signal does not change during simulation; RPython enables annotating the bitwidth as immutable. We are also able to directly manipulate the underlying integer arrays at the RPython level. These specializations significantly eliminate potential bridges. Table 1 shows that this technique improves the performance by an additional 1.5–3.5× on top of the JIT-aware HGSF techniques.

Support for Huge Loops – The techniques described in Section 4.1 improve performance but also often increase the total size of all traces. PyPy's VMProf tool is only useful for identifying Python-level bottlenecks, so we use the Linux *perf* tool to identify the microarchitectural implications of these larger instruction cache footprints. Experiments show that for the 8-core (1-core) simulation in Section 5, 3% (0.2%) of all instruction fetches incur an instruction TLB load, among which 22% (2.6%) are iTLB misses. The need for larger TLB reach motivates us to modify PyPy to allocate 2 MB huge pages for traces and to fall back to 4 KB pages if Linux's huge-page support is unavailable. As a proof of concept, the removal of excessive iTLB accesses (confirmed by *perf*) improved the performance of the 32-core design by 10% as shown in Table 1.

5 CASE STUDY: RISC-V MULTICORE

We present an apples-to-apples simulation performance comparison of 1–32 RTL RV32IM [2] five-stage cores implemented in-house in Verilog, PyMTL, and Mamba using a structural datapath and pipelined control unit. The cores run a parallel matrix multiplication application kernel using a lightweight parallel runtime. We simulate Verilog with CVS, Icarus, and Verilator, and we use PyPy for PyMTL, PyMTL-CSim, and Mamba. The multi-core does not include caches nor an interconnection network and is simulated with a behavioral test memory implemented in Verilog for CVS and Icarus, C++ for Verilator, and Python for PyMTL and Mamba. ASIC synthesis results show that each core can be implemented in around 10 K gates. This design is sufficient for exploring the scalability of various hardware development frameworks, and more complex system-on-chip designs are left as future work. The simulation platform includes an Intel E3-1240 v5 processor and 32 GB DDR4-2400 memory running Ubuntu 14.04 Server, gcc-4.8.5, PyPy-5.8, Verilator-3.876, and Icarus-11.0.

Compilation/Warmup – Figure 4(a) and (b) reflect the iterative development cycle for simulating a specific number of instructions. This includes all overheads: CVS, Icarus, Verilator, and PyMTL-CSim compile times; PyMTL and Mamba elaboration times; and PyMTL and Mamba JIT warmup times. Intuitively, the leftmost points (i.e., short simulations) are affected the most by these overheads. Overall, CVS and Icarus have relatively low compilation overhead (1–2 s for 1-core, 3 s for 32-core), whereas Verilator has larger compilation overhead (4–5 s for 1-core, 130 s for 32-core). PyMTL and Mamba have short elaboration times for one core (<1s) but longer elaboration times for 32 cores (6-8s). The JIT warmup overhead is difficult to quantify; both PyMTL and Mamba warm up within at most 10^5 simulated cycles, and the absolute warm-up time is shorter in Mamba compared to PyMTL.



(a) Simulating 1-Core (b) Simulating 32-Core (c) Steady State Performance

Figure 4: Performance of Simulating RISC-V Multicore – Each point in (a) and (b) is the average simulated cycle per second (CPS) taking compilation overhead into account; (c) captures the scalable performance: steady state CPS multiplied by number of simulated cores.

Performance – When simulating a 1-core system, Mamba executes 332K CPS which is slightly faster than CVS and significantly faster than the other frameworks. Mamba's 1-core performance is equivalent to 148K committed instructions per second. When simulating a 32-core system Mamba is 2.1× slower than CVS but again significantly faster than the other frameworks. Overall these results demonstrate that Mamba nearly matches the performance of CVS for both small and large designs for both short and long simulations. While Verilator can achieve impressive performance for long simulations, it can be difficult to amortize Verilator's long compile times for short simulations potentially precluding using Verilator in agile test-driven development.

Scalability – Figure 4(c) summarizes the steady-state performance of all frameworks with a gradually increasing number of simulated cores. We multiply the simulated cycles per second by the number of cores to reflect the simulation performance scaling with the size of design. A flat line indicates perfect scalability (i.e., a 2× larger design results in a 2× reduction in CPS). CVS and Icarus have good scalability, whereas Verilator appears to be less scalable. The source code size generated by Verilator scales up linearly with the number of cores, potentially harming the quality of C++ compilation. Mamba is faster than CVS at 1-core, and only 2× slower at 32-core. PyMTL scales better than PyMTL-CSim and Mamba, but its absolute performance is relatively low.

6 CONCLUSION

This paper introduced Mamba, a new version of PyMTL that has been carefully designed to close the performance gap in productive hardware development frameworks. Our key insight is the need to deeply co-optimize the HGSF and the underlying general-purpose JIT compiler. Several novel JIT-aware HGSF as well as HGSF-aware JIT techniques enable Mamba to match the performance of a commercial HDL simulator and to improve performance compared to prior HGSFs by 10×. While this paper explores these techniques within the context of Mamba, our work also sheds light on performance optimization opportunities in other HGSFs. The source code used in this paper is published at `https://github.com/cornell-brg/mamba-dac2018`.

ACKNOWLEDGMENTS

This work was supported in part by NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, AFOSR YIP Award #FA9550-15-1-0194, and a donation from Intel. The authors acknowledge and thank Derek Lockhart for his valuable feedback and his work on the original PyMTL framework. We also thank Christina Delimitrou for access to a high-performace Intel Xeon platform.

REFERENCES

[1] D. Ancona et al. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. *Symp. on Dynamic Languages*, Oct 2007.
[2] K. Asanovic et al. Instruction Sets Should Be Free: The Case for RISC-V. Technical report, UCB/EECS-2014-146, Aug 2014.
[3] C. Baaij et al. Cλash: Structural Descriptions of Synchronous Hardware Using Haskell. *Euromicro Conf. on Digital System Design (DSD)*, Sep 2010.
[4] J. Bachrach et al. Chisel: Constructing Hardware in a Scala Embedded Language. *Design Automation Conf. (DAC)*, Jun 2012.
[5] S. Belloeil et al. Stratus: A Procedural Circuit Description Language Based Upon Python. *Int'l Conf. on Microelectronics (ICM)*, Dec 2007.
[6] P. Bellows et al. JHDL-An HDL for Reconfigurable Systems. *Symp. on FPGAs for Custom Computing Machines (FCCM)*, Apr 1998.
[7] P. Bjesse et al. Lava: Hardware Design in Haskell. *Int'l Conf. on Functional Programming (ICFP)*, Sep 1998.
[8] C. F. Bolz et al. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, Jul 2009.
[9] J. Clow et al. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.
[10] J. Cong et al. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, Mar 2011.
[11] J. Decaluwe. MyHDL: A Python-based Hardware Description Language. *Linux Journal*, Nov 2004.
[12] J. P. Grossman et al. The Role of Cascade, a Cycle-Based Simulation Infrastructure, in Designing the Anton Special-Purpose Supercomputers. *Design Automation Conf. (DAC)*, Jun 2013.
[13] P. Haglund et al. Hardware Design with a Scripting Language. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2003.
[14] Icarus Verilog. `http://iverilog.icarus.com`.
[15] J. Jennings et al. Verischemelog: Verilog Embedded in Scheme. *Conf. on Domain-Specific Languages (DSL)*, Oct 1999.
[16] Y. Li et al. HML, A Novel Hardware Description Language and Its Translation to VHDL. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 8(1):1–8, Dec 2000.
[17] D. Lockhart et al. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.
[18] A. Mashtizadeh. PHDL: A Python Hardware Design Framework. Master's thesis, EECS Department, MIT, May 2007.
[19] Migen. `https://m-labs.hk/gateware.html`.
[20] N. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Jun 2004.
[21] P. R. Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. *Int'l Symp. on Systems Synthesis (ISSS)*, Oct 2001.
[22] D. G. Pérez et al. A New Optimized Implemention of the SystemC Engine Using Acyclic Scheduling. *Design, Automation, and Test in Europe (DATE)*, Feb 2004.
[23] O. Shacham et al. Rethinking Digital Design: Why Design Must Change. *IEEE Micro*, 30(6):9–24, Nov/Dec 2010.
[24] Verilator. `http://www.veripool.org/wiki/verilator`.