

Artifact Evaluation and Reproduction of the Calyx Hardware Compiler

Shaurya Raswan, sraswan@ucsd.edu

ABSTRACT

This paper explores and evaluates the Calyx hardware compiler framework. Originally introduced by Nigam et al. at ASPLOS 2021 [1], the Calyx intermediate language presents a novel approach to hardware accelerator design by separating control flow from structural datapaths to enable high-level, predictable optimizations. With its modern Rust-based implementation, the compiler utilizes strict software paradigms to enforce safe, predictable hardware generation. This enables data-level parallelism that standard High-Level Synthesis (HLS) tools often cannot detect, while strictly preventing structural hazards. I will present an artifact reproduction of the original Calyx evaluation, focusing specifically on cycle-level latency improvements across standard, unrolled, and systolic array benchmarks. By utilizing the 2026 Calyx toolchain and open-source Verilator simulations, I will assess the improvements and shortcomings of the framework over the past five years. I will also detail the resolution of legacy driver mismatches, deprecated static-timing passes, and the impact of new structural hazard assertions on legacy benchmarks. Despite these ecosystem shifts, I successfully validated the foundational latency optimizations and performance trends claimed in the original work.

1. INTRODUCTION

As the scaling of general-purpose processors slows, the computing industry has increasingly turned to specialized hardware accelerators to meet the performance demands of modern applications, particularly in domains like machine learning and data processing. Traditionally, designing these accelerators required writing low-level Register Transfer Level (RTL) code, such as Verilog, which is notoriously tedious and error prone. While High Level Synthesis (HLS) tools like Xilinx Vivado HLS have attempted to bridge this gap by compiling C/C++ to hardware, they often hide structural hardware allocation and control flow. This makes it difficult for researchers to build new DSLs that compile to silicon.

To address the limitations of traditional HLS, Nigam et al. introduced Calyx at ASPLOS 2021 [1]. Calyx is an intermediate language (IL) and compiler ecosystem specifically designed to serve as a versatile compilation target for hardware designs. The structure of Calyx seems heavily inspired by the type safety invariants and liveness optimizations that The Rust Programming Language [2] provides. It separates each hardware design into two halves: a *structural datapath*

(containing physical components like registers, memories, and adders) and *imperative control flow* (dictating when components execute using software-centric constructs like `seq`, `par`, and `while`). This separation makes it easier for frontend languages like Dahlia and systolic array generators to emit hardware and allows the Calyx compiler to perform parallelism optimizations before lowering the code to Verilog.

This report presents an independent artifact evaluation and partial reproduction of the original Calyx paper. The original study measured both physical hardware area (via proprietary Xilinx tools) and execution latency. I used open-source Verilator simulations to verify the fundamental performance claims of Calyx’s optimizations. I tested static timing scheduling and loop unrolling across PolyBench linear algebra kernels and systolic arrays.

I tried reproducing this 2021 hardware artifact using a 2026 compiler ecosystem, which presented challenges. The modern Calyx compiler, implemented in Rust, enforces significantly stricter memory safety and structural hazard guarantees than its predecessor. In this paper, I documented the necessary adaptations required to bridge this five-year gap. These include resolving mismatches between the legacy fud driver and modern environments, translating deprecated monolithic optimizations (specifically `-d static-timing`) into modern equivalent passes (`-d static-promotion`), and analyzing how structural hazard assertions (like multiple assignment to memory ports) impacted legacy unrolled benchmarks. I also uncovered important latency differences between the 2021 evaluation and this 2026 iteration that highlights a design shift of Calyx.

However, in order to understand the adaptations of the modern Calyx compiler, understanding the fundamental background behind Calyx and HLS toolchains is important.

2. BACKGROUND AND RELATED WORK

2.1 Rust for Parallelism

The increasing complexity of parallel architectures has sparked a need for frameworks that can offer high performance and rigorous structural safety. In recent years, Rust has emerged as a foundational tool in this, bridging the gap between high-level software abstractions and low-level systems execution. By enforcing strict memory and thread safety rules at compile time, Rust enables fearless concurrency and zero-cost abstractions that eliminate data races without sacrificing execution speed [3]. However, this comes at a caveat of

predictable parallelism, which is highly volatile when using unsafe Rust.

Calyx uses similar ideas to this fearless concurrency with the accelerator structure and semantics it attempts to enforce. A perfect example is Calyx’s register sharing via live-range analysis (Section 5.2 in [1]). This will be discussed further later.

This paradigm shift to memory safety is being validated across many parallel computing standards. For instance, recent implementations like NPB-Rust have demonstrated that Rust can effectively match the performance of traditional C and Fortran in the NAS Parallel Benchmarks, while significantly reducing memory-related vulnerabilities inherent to legacy systems programming [4].

This paradigm is also being applied to scale parallel workloads across distributed architectures. By using Rust’s type system, researchers have successfully introduced static verification to the Message Passing Interface (MPI), preventing complex communication and structural hazards at compile time [5]. Similarly, in the embedded and accelerator space, frameworks such as Ariel-ML utilize embedded Rust to orchestrate computing parallelization for neural networks on heterogeneous multi-core microcontrollers. This showcases Rust’s viability for low-level parallel execution even in resource-constrained environments [6].

Beyond software-defined parallelism, the boundary between hardware and software co-design is being redefined. The development of emerging hardware interfaces, such as Compute Express Link (CXL) architectures [7], relies heavily on predictable, safe control flows to manage complex memory models and customized accelerators [8].

The Calyx compiler framework sits at the intersection of software-enforced safety and hardware-level parallelism. Like the efforts to statically verify concurrent software systems, Calyx tries to use Rust to map concurrency guarantees into hardware generation. By strictly separating control flow from structural datapaths, Calyx applies modern compiler passes to enforce safe accelerator design.

2.2 Hardware Accelerators

Calyx provides the intermediate representation necessary to compile high-level algorithms into specialized hardware.

Within accelerator design, data-level parallelism is important. For example, Single Instruction, Multiple Data (SIMD) architectures achieve this by applying a single operation across multiple data points simultaneously, making them very effective for vectorized workloads [7].

Systolic array architectures have become the standard for dense matrix multiplication and convolution operations, which are integral to LLMs and machine learning. By passing data through a tightly coupled network of processing elements without needing to access main memory for intermediate results, systolic arrays maximize data reuse and throughput.

2.3 Memory System Safety

The shift toward heterogeneous accelerators and disaggregated memory introduces reliability challenges, particularly as data traverses high-speed interconnects like UCIe or UALink [7]. Hardware must remain resilient against transient faults, such as memory soft errors or a single processing

element (PE) failures. By using Calyx’s control flow, future designs could statically eliminate structural hazards and use error correction and recovery mechanisms directly into the hardware generation pipeline.

3. MOTIVATION

My motivation stems from my interest in the intersection of parallel programming, computer architecture, and systems-level safety. I am gravitated toward Rust as a tool for managing complex concurrent systems. Through recent projects on evaluating and benchmarking large language models in Rust-native frameworks like Burn against traditional PyTorch stacks, I have explored how Rust’s strict concurrency and memory paradigms offer distinct advantages. I am also interested in the compiler construction aspect of Rust. For instance, I have built a JIT compiler with Rust using dynasm and subsequently extended the CSE 131 curriculum as a TA alongside Professor Joseph Politz. As the computational demands of machine learning continue to drive accelerators, I hope to see frameworks like Calyx grow due to its benefits. It is incredibly fascinating to see modern hardware compilers handle structural hazards and extract data-level parallelism. I hope to help Rust become a prominent, foundational standard in the next generation of AI architecture.

4. DESIGN AND IDEAS

4.1 Language Semantics

A component in Calyx is similar to a function in software or a module in Verilog. It encapsulates a specific hardware structure and its corresponding execution schedule. Each component defines an interface of untyped input and output ports of specified bit-widths.

```
component loop_compute(in_val: 32)
  -> (out_val: 32)
{
  cells { ... }
  wires { ... }
  control { ... }
}
```

***Snippet 1:** The outer architectural wrapper defining the component interface. The internal logic is split strictly into structural (cells/wires) and temporal (control) blocks.*

The `cells` instantiate the physical subcomponents, such as registers, memories, or ALUs, like declaring variables or objects.

The `wires` establish continuous assignments between the ports of these cells. No port can have multiple active assignments in the same clock cycle, as this would result in undefined hardware behavior/structural hazard.

```
cells {
  a_reg = std_reg(32);
  b_reg = std_reg(32);
  x_reg = std_reg(32); // Loop counter
  add = std_add(32);
  cmp = std_lt(32);    // Comparator
```

```

}
wires {
    // Prevents overlapping drivers
    add.left = cmp.out ? a_reg.out;
    add.left = !cmp.out ? b_reg.out;
}

```

Snippet 2: Declaring the physical hardware units. The wires section shows a continuous, guarded assignment where the adder's left input is multiplexed based on the comparator's output.

The groups are unique to Calyx for safe hardware scheduling. A group encapsulates a set of wire assignments, keeping them completely inactive by default. Multiple different groups can contain assignments to the exact same hardware port without violating the unique driver rule since groups are never allowed to execute simultaneously. This reminds me of Rust's Aliasing XOR Mutability (AXM) which is enforced by its ownership + borrow checker.

Calyx creates lifetimes for hardware logic in this way. The group asserts a done signal to notify the compiler that its computation is safely completed. Calyx uses `safe Rust` to determine hardware interaction in this way, which is unique in how Rust hardware interaction is usually always unsafe inherently.

```

wires {
    group assign_one {
        x_reg.in = 66;
        assign_one[done] = x_reg.done;
    }
    group assign_two {
        x_reg.in = 77;
        assign_two[done] = x_reg.done;
    }
    group check_cmp {
        cmp.left = x_reg.out;
        cmp.right = 100;
        check_cmp[done] = 1'b1;
    }
    // ... compute_a and compute_b defined ...
}

```

Snippet 3: Encapsulating assignments into isolated lifetimes. Even though `assign_one` and `assign_two` both drive `x_reg.in`, the groups are never active concurrently.

The control section defines the imperative, software-like scheduler that orchestrates when each group is active. Because the structural hazards are safely encapsulated inside the groups, the user can freely use high-level control flow operators to manage execution.

Some examples are `seq` (for strictly sequential execution) and `par` (for explicit parallel execution). For `par`, Calyx guarantees the listed groups will execute concurrently, automatically generating the underlying FSM and synchronization logic necessary to wait for every parallel group's done signal before advancing.

```

control {
    seq {

```

```

        assign_one; // x_reg = 66
        while cmp.out with check_cmp {
            seq {
                par {
                    compute_a; // Update a_reg
                    compute_b; // Update b_reg
                }
                assign_two; // x_reg = 77
            }
        }
    }
}

```

Snippet 4: The execution schedule tying everything together. It initializes the state sequentially, evaluates the condition group (`check_cmp`), and safely executes `compute_a` and `compute_b` in perfect parallel synchronization before iterating.

This explicit declaration of parallelism is where Calyx outperforms traditional HLS, in my opinion. Standard HLS tools often fail to infer parallelism properly, resulting in opaque bottlenecks that are notoriously difficult to debug. Having an explicit `par` block feels inherently similar to defining explicit concurrent threads or asynchronous tasks in software. The issue with modern HLS frameworks is the guesswork, which is an issue the authors specifically highlight with Vivado HLS. They cite that the closed-source nature of Vivado makes it difficult to understand if it automatically infers systolic arrays from loop nests.

4.2 Calyx Compilation to Hardware

The compiler optimizes and lowers the high-level Calyx intermediate representation into synthesizable RTL code, such as Verilog. This process relies on **interface signals**, which define a calling convention that exposes the component's execution schedule. Calyx applies a series of passes that ultimately result in a flat list of guarded assignments with no remaining control statements or groups.

- **GoInsertion:** Guards all structural assignments within a group using the group's `go` interface signal, ensuring they only activate when actually called.
- **CompileControl:** Generates latency-insensitive FSMs to orchestrate control operators (like `seq`, `par`, `while`).
- **Sensitive:** Compiles control statements into groups using the latency-sensitive FSMs to optimize execution time. Specifically targets groups tagged with the "static" latency attribute. This is important for later!
- **RemoveGroups:** Inlines the uses of interface signals and entirely eliminates the group abstractions, flattening the architecture.
- **Lower:** Translates the resulting control-free, flattened program directly into synthesizable RTL.

5. IMPLEMENTATION DETAILS

In order to evaluate Calyx, we need to understand the optimizations it does. Calyx implements 3 optimizations:

resource sharing, live-range-based register-sharing, and cycle latency inference.

5.1 Resource Sharing

Calyx uses its explicit execution schedule to build a conflict graph. This models modern day register allocation problems, which are NP-Hard. Frameworks like LLVM have many different register allocation algorithms that resolve the conflict graph to minimize register use [9].

- **Conflict graph:** Nodes represent groups and edges denote that those groups might execute simultaneously (for example, if groups are in the same par block).
- **Greedy coloring:** Greedy graph coloring over the conflict graph happens to allocate shareable components. If two groups share an edge, they cannot share the same component/register.
- **Group rewriting:** Then it applies local rewrites to the wiring inside the groups based on the new mapping. The rewriter does not need to reason about component usage outside of the isolated group due to Calyx’s guarantees.

5.2 Register Sharing via Live-Range Analysis

Calyx implements a live-range analysis pass to understand how long data needs to be preserved.

- **Parallel control flow graphs (pCFGs):** We have p-nodes to represent entire par blocks, which represent all children execute simultaneously and all written data in the children is visible afterward.
- **Calculating read and write sets:** The compiler over-approximates the registers that are read and written.
- **Computing liveness:** Using a standard data-flow formulation, we see the live ranges of each register.
- **Conflict graph:** This is a new conflict graph where nodes are registers and edges are overlapping live ranges.
- **Greedy Coloring (again):** Like resource sharing, it does greedy coloring over this graph to multiplex multiple variables into the smallest possible number of physical registers.

5.3 Cycle Latency Inference

The way Calyx translates its control schedules into hardware is dictated by the availability of timing information. Calyx uses both latency-insensitive and latency-sensitive compilation within the same design to optimize.

Latency-Insensitive Compilation (Default): By default, the `CompileControl` pass generates latency-insensitive hardware. It builds FSMs that strictly wait for the done signal from one group before advancing the go signal of the next. This handshake mechanism is robust but has a performance penalty. Coordinating these interface signals requires additional hardware logic that wastes execution cycles.

Latency-Sensitive Compilation (Optimized): To reclaim this lost performance, we can provide exact timing information by annotating groups with a "static" attribute, specifying the exact number of clock cycles the computation requires.

Then the `Sensitive` pass opportunistically takes over. It makes an optimized FSM with a simple self-incrementing counter. This FSM blindly asserts the go signals of sequential groups for their exact specified durations. If we mix timed and untimed groups, Calyx falls back to latency-insensitive handshakes only when necessary.

In the Results section, I will evaluate the effects of this latency-sensitive compilation. By benchmarking standard, unrolled, and systolic array designs, I will analyze how the ecosystem’s transition from legacy static-timing passes to modern static-promotion affects cycle-level performance.

5.4 Targets for Evaluation Setup

Two compilers target Calyx:

- A PE-parametric systolic array generator. Calyx IL determines the data movement and computation schedule.
- **Dahlia [10].** This is a front-end programming language for accelerator design. It has a backend targeting HLS toolchains.

The implementation of this artifact evaluation is based on the official Calyx evaluation repository [11]. However, the original 2021 study and evaluation repository relied on a specific snapshot of the compiler. My reproduction utilizes the 2026 version of the Calyx infrastructure, necessitating significant adaptations to the driver and simulation pipeline.

5.5 Environment

My evaluation environment is an ARM-based Apple Silicon architecture (macOS) and an Ubuntu 64-bit VM, requiring a native build of the Calyx compiler. I utilized the Rust toolchain to compile the core calyx binary. For the frontend, I compiled the Dahlia compiler [10] using the Scala Build Tool (sbt) to process the .fuse PolyBench benchmarks.

Below is the environment I used:

```
- GNU parallel 20260222
- Verilator 5.046 2026-02-28
- python3 3.13.3, pip 25.0.1, numpy 2.4.3,
  pandas 3.0.1, seaborn 0.13.2,
  matplotlib 3.10.8, jupyterlab 4.5.6
- jq 1.8.1
- vcdump 0.1.2
- vivado v2019.2, vivado_hls v2019.2
- calyx (commit 052ab41)
- fud (commit 052ab41 from calyx repository)
- dahlia (commit bd68b13)
- scala 3.3.1
- javac 25.0.2
```

All of our versions are very recent. Our primary calyx version is incredibly new (accessed March 12, 2026). The original study used `futil`, which was the 2021 version of Calyx. `fud` is also a legacy driver, as using the updated `fud2` was difficult.

I utilized **Verilator** as the primary execution engine. The pipeline transforms high-level Dahlia code into Calyx IR, lowers it to Verilog, and finally compiles it into a native C++ simulator to extract exact clock cycle counts.

A primary challenge in the implementation was the architectural shift in the Calyx driver. While the modern ecosystem

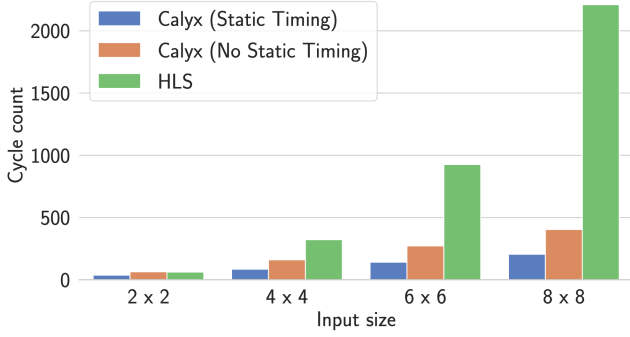


Figure 1: Figure 7a from paper. 2021 Systolic Array Normalized Cycle Counts

has moved toward fud2 (a Rust-and-Ninja-based driver), the 2021 evaluation scripts are hardcoded for the legacy Python-based fud driver. To maintain compatibility with the original bash scripts, I performed a dual-driver installation. I utilized the uv Python package manager to install the legacy fud tool within a virtual environment, manually configuring it to point to the modern Rust calyx binary.

5.6 Resolving Compiler Pass Deprecation

As the Calyx compiler evolved, several monolithic optimization passes were refactored. I encountered a critical failure when attempting to disable the `static-timing` pass, as that specific identifier has been deprecated in favor of a more granular suite of static scheduling passes. Through a comparative analysis of the 2026 compiler’s `pass-help` utility, the flag previously known as `static-timing` is now primarily managed by the `static-promotion` pass. I modified the evaluation scripts to utilize the `-d static-promotion` flag, successfully restoring the ability to generate the "latency-insensitive" baseline required for Figure 6c from the paper. Our Results will further explore how the new Calyx infrastructure changes latency sensitivity.

5.7 JSON Structure and Output Parsing

The simulation output format has transitioned from a flat JSON structure to a wrapped hierarchy. The original jq filters in the evaluation scripts (targeting `.TOP.main.clk`) failed to parse the modern vcdump output. I refactored the data extraction logic to use `.toplevel.main.clk`, which sums the individual cycle counts to calculate total benchmark latency.

I encountered strict structural hazard assertions in the 2026 compiler. In the unrolled benchmarks, the modern compiler correctly identifies multiple simultaneous assignments to the same memory port, which is a hazard that was handled more loosely in 2021. This only impacted `gramschmidt.fuse`, which still returned a latency value.

6. RESULTS

6.1 Systolic Array Latencies

First we will evaluate systolic array designs that will be lowered to Calyx. Calyx provides a Python generator to make

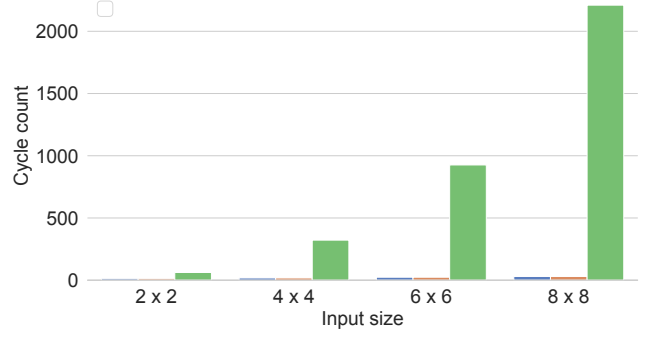


Figure 2: Recreation of Figure 7a. 2026 Systolic Array Normalized Cycle Counts.

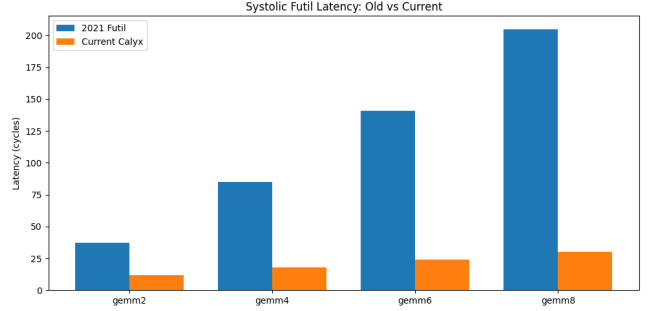


Figure 3: 2021 Futil (sensitive) vs. 2026 Calyx Systolic Array Timings

specified systolic array size designs.

In recreating Figure 7a (shown in Figure 1), I observed that disabling the modern `static-promotion` pass yielded identical cycle counts for the systolic array benchmarks, resulting in a 1.0x speedup between the latency-sensitive and latency-insensitive Calyx configurations. This reveals a fundamental evolution in the Calyx ecosystem. The 2021 compiler relied on a dedicated pass to optimize dynamic systolic components. The Calyx IR was later updated to support native static control constructs (like `static seq`, `static repeat`) [12]. The modern systolic frontend (built with Python) natively emits strict, statically-scheduled Calyx IR by default. Therefore, with the modern iteration of Calyx, it is impossible to generate the unoptimized latency-insensitive baseline as the hardware is optimally scheduled by construction.

As show in Figure 2, the updated Caylx compiler heavily outperforms the static timing values of the 2021 version. Since we are using the same Vivado version, the HLS timings were the same. In addition, the latency-insensitive values of Figure 2 are the same as the latency-sensitive values, however, as stated previously, it is not possible to get true values with no static timing with the systolic array generator anymore.

This aggressive optimization and improvement is most likely due to Calyx’s native static control constructs. While the original compiler relied on promoting dynamic groups, this often made minor cycle penalties during FSM state transitions. The 2026 toolchain uses native static control constructs to make counter-based scheduling, which eliminates

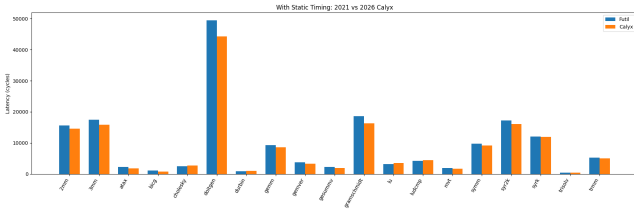


Figure 4: Sensitive/static timing for 2021 Futil vs. 2026 Calyx.

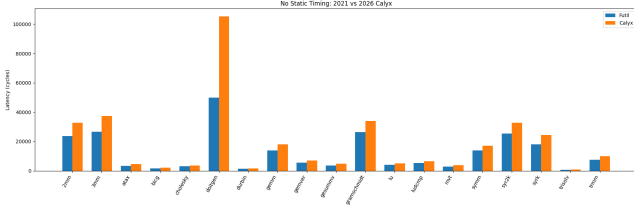


Figure 5: No static timing for 2021 Futil vs. 2026 Calyx.

dead cycles between control groups and optimizes memory pipelining. The systolic array generator can inherently execute perfectly on the theoretical $3N$ lower bound.

6.2 Dhalia Program Latencies

Next we use ‘.fuse’ Dhalia programs to test whether the latency from the Dhalia-to-Calyx compiler in 2011 LOC of Scala is the same as before. Using Calyx’s extensions to the Dhalia compiler, I used the Dhalia specification backend to generate Calyx from lowered Dhalia.

As seen in Figure 4, the 2026 iteration of Calyx has sensitive timing information that is quicker than the 2021 evaluation. The geometric mean speedup is 1.0762x. This is due to the new optimizations of the 2026 Calyx compiler and IR.

However, in Figure 5, we see no static timing information performs worse in the new version of Calyx, with a 0.7584x geometric mean speedup. This is possibly due to the stricter performance guarantees and conservative non-static passes of Calyx. The native static static control constructs were probably more widely adopted, making non-static optimization not a focus anymore.

When recreating Figure 9c from the paper, we see that the normalized speedup is much better with the 2026 version of Calyx. However, this is due to how the non-sensitive timings are much worse than before, making the delta in sensitivity speedup much higher.

Despite the non-static latency slowdown of Calyx, both of these target compilations still highlight the speedup that Calyx has over HLS overall.

7. CONCLUSION

I successfully reproduced and evaluated the core cycle-level latency claims of the ASPLOS 2021 Calyx framework, bridging a five-year gap in compiler evolution.

My evaluation revealed that Calyx has fundamentally matured from a compiler relying heavily on post-generation analysis to one that gives frontends native, precise control. In the systolic array benchmarks, the introduction of native static

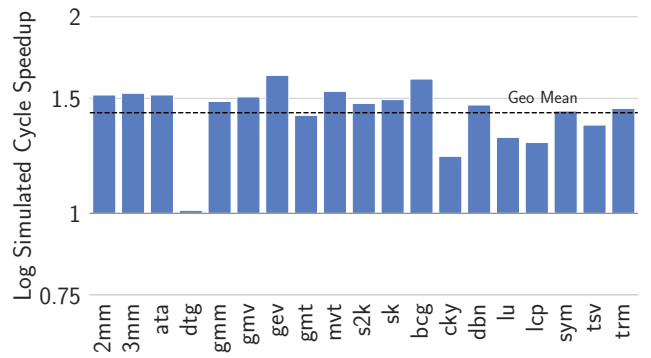


Figure 6: Figure 9c from paper. Cycle Speedup with 2021 Futil.

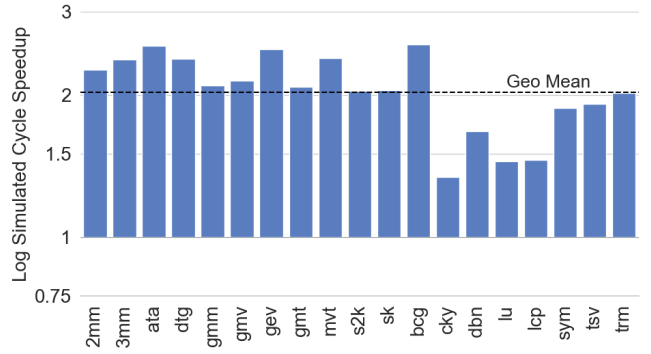


Figure 7: Recreation of Figure 9c. Cycle Speedup with 2026 Calyx.

control constructs (like `static seq`) allowed the modern generator to bypass dynamic handshaking. This eliminated FSM state transition overhead, allowing the 2026 compiler to achieve the bound of the $3N$ limit. This makes legacy static-promotion passes obsolete for natively static designs.

For generic algorithmic workloads compiled via Dhalia, the results demonstrate a prioritization of statically scheduled performance over legacy dynamic control. The optimized, latency-sensitive Calyx designs outperformed their 2021 counterparts (achieving a 1.07x geometric mean speedup), the unoptimized, latency-insensitive baseline regressed (0.75x speedup compared to 2021). This divergence highlights modern Calyx’s strictness. Memory safety guarantees and structural hazard assertions have conservative penalties on dynamic fallback logic. The dramatic latency speedups observed in the 2026 iteration are due to a higher performance ceiling for static designs and a stricter, slower baseline for dynamic ones.

Despite the difficulty of correcting the evaluation scripts and the deprecation of older compiler passes, the foundational claims of the original Calyx paper hold true. The separation of structural datapaths from imperative control flow continues to enable hardware optimizations that still outperform HLS. The modern Calyx iteration delivers on those promises with more strictness, safety, and cycle-level efficiency for timing sensitive workflows than its predecessor.

8. FUTURE WORK

First, future work must reintegrate physical resource estimation into the modern pipeline. Integrating modern synthesis tools (or open-source equivalents like Yosys) can show how native static control impacts the physical utilization of LUTs, DSP slices, and block RAM.

More investigation can be done about Calyx’s resource sharing mechanisms. Future studies should analyze how Calyx handles port multiplexing, and whether the strictness introduces any physical routing overhead when multiple control groups attempt to share the same hardware registers.

In addition, the Dahlia (.fuse) benchmarks evaluated in this report rely on generic, dynamic control flow that still encounters Calyx’s static-promotion pass. Modifying the Dahlia language and compiler to natively emit modern static Calyx IR constructs (like `static seq` and `static while`) could bypass the conservative dynamic baseline. This is similar to how the systolic array generator operates.

9. REFERENCES

- [1] R. Nigam, S. Thomas, Z. Li, and A. Sampson, “A language and compiler view of hardware design,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’21*, (New York, NY, USA), pp. 584–597, Association for Computing Machinery, 2021.
- [2] The Rust Project Developers, *The Rust Reference*. The Rust Foundation, 2025.
- [3] J. Abdi, G. Posluns, G. Zhang, B. Wang, and M. C. Jeffrey, “When is parallelism fearless and zero-cost with rust?,” in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2024.
- [4] E. M. Martins, L. G. Faé, R. B. Hoffmann, L. S. Bianchessi, and D. Griebler, “Npb-rust: Nas parallel benchmarks in rust,” *arXiv preprint arXiv:2502.15536*, 2025.
- [5] N. Iqbal and J. Brown, “Enhancing type safety in mpi with rust: A statically verified approach for rsmapi,” *arXiv preprint arXiv:2509.10803*, 2025.
- [6] Z. Huang, K. Schleiser, G. Myung, and E. Baccelli, “Ariel-ml: Computing parallelization with embedded rust for neural networks on heterogeneous multi-core microcontrollers,” *arXiv preprint arXiv:2512.09800*, 2025.
- [7] J. Zhao, “CSE 240B: Parallel Computer Architecture - Syllabus and Lecture Materials.” University of California, San Diego (UCSD) Course Website, 2026.
- [8] R. Gioiosa *et al.*, “Hardware-software co-development for emerging cxl architectures,” in *Proceedings of the International Symposium on Memory Systems (MemSys ’25)*, 2025.
- [9] A. Fried, M. Stemmer-Grabow, and J. Wachter, “Register allocation for compressed isas in llvm,” in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC ’23)*, (New York, NY, USA), ACM, 2023.
- [10] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, “Predictable accelerator design with time-sensitive affine types,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [11] C. Developers, “Calyx Evaluation: Artifacts and Benchmarks.” GitHub Repository, 2026. Accessed: March 20, 2026.
- [12] The Calyx Project, *Calyx Documentation: Static Timing*. Cornell University, 2023. Accessed: 2026-03-20.
- [13] R. Mishra, “High-performance machine learning inference systems with rust: A review of techniques, tools, and optimizations,” *International Research Journal of Engineering and Technology (IRJET)*, vol. 11, pp. 639–644, Nov 2024.
- [14] S. Basu, “Rust as a hardware description language,” in *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE ’24)*, 2024.
- [15] S. Basu, “Rhdl: Rust as a hardware description language,” in *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE ’25)*, 2025.
- [16] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, “The case for writing a kernel in rust,” in *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys ’17)*, 2017.
- [17] R. Tang, M. Wang, X. Sun, L. Huang, Y. Huang, and X. Ma, “Converos: Practical model checking for verifying rust os kernel concurrency,” in *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC ’25)*, 2025.
- [18] Z. Yu, L. Song, and Y. Zhang, “Fearless concurrency? understanding concurrent programming safety in real-world rust software,” *arXiv preprint arXiv:1902.01906*, 2019.
- [19] E. Holk, M. Pathirage, A. A. Chauhan, A. Lumsdaine, and N. D. Matsakis, “Gpu programming in rust: Implementing high-level abstractions in a systems-level language,” in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013.