PERFORMANCE-ORIENTED WAVE DIGITAL CIRCUIT EMULATION

Jatin Chowdhury and Mark Rau

Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, USA
chowdsp@mit.edu

ABSTRACT

Wave Digital Filters are a circuit-modeling paradigm well-suited for reusable software implementation, but existing software implementations often incur significant overhead due to run-time abstractions and data layout constraints. This paper presents a performance-oriented toolchain for implementing Wave Digital circuit models based on static code generation. The toolchain consists of a declarative circuit description language, a compiler that generates circuit simulation code with minimal persistent state and no run-time abstraction, and a minimal runtime library implementing specialized circuit components as Wave Digital Filters. Performance measurements across several test circuits demonstrate that the generated models consistently outperform existing implementations, and achieve near-ideal performance relative to a theoretical execution bound.

1. INTRODUCTION

Wave Digital Filters (WDFs) are a paradigm for simulating the behaviour of analog circuits [1], and are widely used in the development of virtual analog (VA) audio effects and synthesizers [2]. Since Wave Digital circuit models are often deployed in real-time audio applications, their run-time efficiency is a primary concern.

WDFs are naturally modular, allowing circuit models to be constructed from symbolic representations of individual circuit elements (resistors, capacitors, etc.) and their interconnections (series, parallel, etc.). This property has motivated the development of several software libraries that implement these WDF building blocks as reusable components. However, such approaches also introduce computational overhead through abstraction, indirection, and limited opportunities for compile-time optimization.

This paper presents a code generation approach for implementing WDF circuit simulations. Given a textual description of the circuit's topology, the proposed system generates specialized executable code that avoids most of the overhead associated with existing WDF frameworks. Models of several circuits were created using the code generation system as well as several existing WDF libraries and the performance of those models was evaluated on two modern consumer CPUs. A novel performance metric is introduced to evaluate the performance of each circuit model relative to a theoretical baseline. Results demonstrate that the proposed approach achieves near-ideal run-time performance, while also providing additional insight into further optimization opportunities for WDF implementations.

The rest of the paper is organized as follows: §2 discusses previously published WDF software implementations; §3 describes the implementation of the novel system; §4 presents performance measurements comparing the novel system to prior work as well as the theoretical baseline; §5 concludes.

Copyright: © 2026 Jatin Chowdhury et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, adaptation, and reproduction in any medium, provided the original author and source are credited.

2. PRIOR WORK

Published in 2016, `RT-WDF` is a modular WDF software library written in `C++` [3]. The library is designed using an object-oriented approach, with WDF elements and adaptors implemented in a polymorphic class hierarchy. This design allows circuit models to be constructed and modified at run time, but incurs computational overhead due to virtual dispatch, pointer indirection, and potentially duplicated per-object state. Additionally, the library design encourages users to allocate the data for each circuit component independently, which may negatively impact cache locality and overall run-time performance.

`wdmodels` implements WDFs in the Faust programming language [4]. Faust is a functional meta-programming language for expressing signal processing algorithms that can be transpiled into several target languages, including `C++`. When constructing a Wave Digital circuit model with `wdmodels`, the circuit topology is fixed at translation time, as any change to the model requires modification of the Faust source code. This approach enables efficient generated code for WDF circuit simulations, however, the user has limited control over memory layout and low-level implementation details in the resulting output.

Developed between 2020 and 2022, `chowdsp_wdf` is a `C++` WDF framework that supports both run-time and compile-time circuit construction [5]. Run-time models are constructed using dynamic polymorphism, while compile-time models leverage template meta-programming to reduce abstraction overhead. In practice, the compile-time approach typically outperforms `RT-WDF` and `wdmodels`. However, even in the template-based design, the circuit structure is encoded as a hierarchy of types representing specific circuit-level constructs, which constrains data layout and limits further optimization.

Several other works have published source code for performing WDF circuit simulations of specific individual circuits, or using programming languages intended for non-real-time use cases (e.g. `MATLAB` or `Python`) [6, 7, 8].

2.1. Object-Oriented Design and Performance

In this work, object-oriented design is defined as "a compile-time hierarchy of encapsulation that matches the domain model" [9].

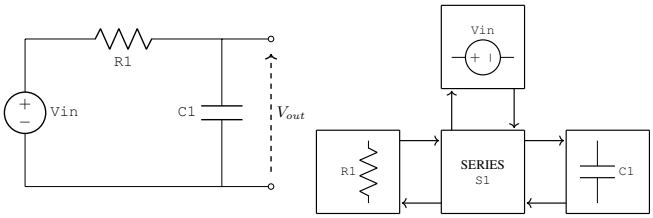

Figure 1: *Left: Circuit schematic for a simple RC lowpass circuit. Right: The same circuit shown as a WDF connection tree.*

For WDFs, this design pattern leads to the encapsulation of data for individual circuit elements and adaptors into discrete "objects", which are then composed to form a complete circuit model. Both `RT-WDF` and `chowdsp_wdf` follow this approach.

For real-time circuit simulation, such a hierarchy introduces two primary inefficiencies. First, much of the data stored in each object does not need to persist from one sample to the next, yet is retained as part of the object's state. Second, many stored values are redundant or shared across elements of the same circuit. These inefficiencies increase per-sample memory traffic, degrade cache locality, and limit the compiler's ability to optimize the resulting code. An ideal implementation would store only the minimal persistent state required by the circuit model and organize that data for efficient sample-by-sample computation.

3. IMPLEMENTATION

3.1. Design Goals

The toolchain presented here is named `wdf_compiler`. The design of the toolchain is guided by four primary goals:

- Generate numerically precise WDF circuit models
- Eliminate overhead due to run-time abstractions
- Minimize persistent state and memory traffic
- Exploit compile-time knowledge of the circuit topology and parameters for further optimization

`wdf_compiler` is available as open-source software under the MIT license. [1]

3.2. Compiler Architecture

The `wdf_compiler` system consists of a declarative WDF circuit description language, a static compiler, and a small WDF runtime library.

3.2.1. Circuit Description Language

The WDF description language is a domain-specific language (DSL) used by `wdf_compiler`, and contains four "sections": `circuit`, `inputs`, `outputs`, and `meta`.

WDF circuit models can be simply described by a "connection tree" between the circuit components and adaptors [2]. The `circuit` section of the DSL is a textual representation of a WDF circuit model written in the form of a connection tree, where each circuit element is declared, followed by the declarations of each of the "child" elements following from that element.

[1] `https://github.com/Chowdhury-DSP/wdf_compiler`

The `inputs` section describes which circuit elements are treated as input sources for which the user will need to provide a voltage or current value. The `outputs` section describes the circuit elements for which the user may want to read out the voltage or current across that circuit element as an output of the model.

Finally, the `meta` section describes additional information that `wdf_compiler` will take into account when generating the resulting implementation code. This information includes:

- A "namespace" to use for the generated code (if the generated code is in `C++`)
- Additional source files to include/load in the generated code (important for circuits with "custom" elements)
- Specific variants of the `calc_impedances` method to generate (to be discussed in the next section)

Figure 1 shows the circuit schematic and WDF connection tree representation for a simple RC lowpass filter circuit, and lis. 1 shows a complete description for the circuit in the WDF description language. The `circuit` section declares a voltage source (`Vin`) as the "root" of the connection tree, followed by a series adaptor (`S1`), resistor (`R1`), and capacitor (`C1`). The resistor and capacitor values are specified using metric prefixes for convenience (e.g., `1k` for a 1 kΩ resistor). `wdf_compiler` supports standard metric prefixes ranging from pico- (10^{-12}) to Mega- (10^{6}). The remainder of the circuit description specifies `Vin` as the circuit input and the voltage across `C1` as the output, and defines the namespace `rc_lowpass` for the generated code.

```
circuit:
IVs(Vin);
   Series(S1);
      R(R1, 1k);
      C(C1, 1u);

inputs:
Vin

outputs:
V:C1

meta:
namespace:rc_lowpass
```

Listing 1: *The RC lowpass filter from fig. 1 described in the WDF description language.*

3.2.2. Static Compiler

The primary component of the `wdf_compiler` toolchain is an executable of the same name that takes a circuit description as input, and generates source code for a Wave Digital model of that circuit as output. The `wdf_compiler` executable is written in the `Jai` programming language and can generate output code in `Jai`, `C++`, `C`, or `Rust`.[2] The generated code contains three data types (`Params`, `Impedances`, and `State`), and two sets of methods (`calc_impedances` and `process`).

[2] `Jai` is a statically-typed compiled language with an emphasis on software performance and quality, developer productivity, and fast compilation speed. At the time of writing, `Jai` is not yet publicly available. While the authors regret that most readers of the paper will not be able to compile and modify `wdf_compiler` for themselves (until `Jai` is publicly released), `Jai` was chosen because it was the best available tool for creating the highest-quality software system in the shortest amount of time.

The `Params` data type contains numerical representations of the component values for each circuit component (for example, the resistance of each resistor in the circuit and the capacitance of each capacitor in the circuit). If the circuit description provides a static value for a circuit component, the generated code may denote that value as a compile-time constant (i.e. `constexpr` in `C++`).

The `Impedances` and `State` data types contain all the data required for computing the output of the circuit model given some input. The data in `State` will be both read from and written to by the `process` method, while the `Impedances` data will only be read from. `Impedances` typically contains values including the impedances and conductances of various components in the circuit as well as coefficients required for series and parallel adaptor "junctions." `State` typically contains persistent values such as the state of a Wave Digital capacitor.

```
struct Params {
   static constexpr float R1_value = 1.0e+03;
   static constexpr float C1_value = 1.0e-06;
};

struct Impedances {
   float S1_pr;
};

struct State {
   float C1_z;
};

void calc_impedances(Impedances& impedances,
                     float fs, Params params) {
   // Computing impedance for: R1;
   const auto R1_R = params.R1_value;
   // Computing impedance for: C1;
   const auto C1_G = 2 * params.C1_value * fs;
   const auto C1_R = 1 / C1_G;
   // Computing impedance for: S1;
   const auto S1_R = R1_R + C1_R;
   const auto S1_G = 1 / S1_R;
   impedances.S1_pr = R1_R * S1_G;
}

float process(State& state, Impedances imp,
              float Vin) {
   auto C1_b = state.C1_z; // C1 reflected
   auto R1_b = 0; // R1 reflected
   auto S1_b = -(R1_b + C1_b); // S1 reflected
   auto Vin_a = S1_b; // Vin incident
   auto Vin_b = 2 * Vin - Vin_a;// Vin reflected
   auto S1_a = Vin_b; // S1 incident
   auto R1_a = R1_b - imp.S1_pr
               * (S1_a - S1_b); // R1 incident
   auto C1_a = -S1_a - R1_b + imp.S1_pr
               * (S1_a - S1_b); // C1 incident
   state.C1_z = C1_a; // C1 state update

   // C1 voltage
   auto v_C1 = (C1_a + C1_b) * 0.5;
   return v_C1;
}
```

Listing 2: *The `C++` code generated by `wdf_compiler` for the RC lowpass filter circuit described in lis. 1.*

The `calc_impedances` methods take in the circuit's parameters (i.e. `Params`) and update the `Impedances` data according to the parameter values, as well as the sample rate. By default `wdf_compiler` will generate a single `calc_impedances` method that updates all `Impedances` values for the entire circuit. If specified in the `meta` section of the circuit description, additional `calc_impedances` methods will be generated to update only the `Impedances` values corresponding to certain circuit components. These additional methods can be used to reduce redundant computation in cases where one or more of the circuit's parameters change more frequently than the others.

Finally, the `process` method takes in the `Impedances` and `State` data, as well as the input quantities for a given timestep, and returns the output at that timestep. Again, the `State` data may be modified, but the `Impedances` data will not be modified.

Listing 2 shows the `C++` code generated by `wdf_compiler` for the circuit description shown in lis. 1.

3.2.3. WDF Modeling

As with the prior work mentioned in §2, `wdf_compiler` implements wave digital circuit models using voltage waves, and the corresponding elements defined in terms of voltage waves [2]. Stateful circuit elements (i.e. capacitors and inductors), are discretized using the trapezoidal rule by default. The system also provides an implementation of a wave digital capacitor discretized using the alpha transform, as described in [10].

3.2.4. WDF Library

The `wdf_compiler` toolchain natively supports a wide range of common circuit components and adaptors, including resistors, capacitors, inductors, series and parallel adaptors, and polarity inverters. However, there are many WDF elements that may benefit from user-level customization (e.g. models of nonlinear components that employ mathematical approximations), or may require custom implementations for each circuit in which they are used (e.g. $\mathcal{R}$-Type adaptors [11] or neural network models of specific components [12, 13, 14, 15]).

As such, `wdf_compiler` also supports WDF models containing "custom" circuit elements and provides a minimal library with support for several commonly used custom elements. The library includes models of diodes and anti-parallel diode pairs as derived in [16] using an approximation of the Lambert $\mathcal{W}$ function derived in [17], as well as a model of a triode vacuum tube as derived in [18]. The library also includes helpers for implementing $\mathcal{R}$-Type adaptors as custom elements, leveraging SIMD instruction sets (Intel SSE and ARM NEON) for improved performance.

3.2.5. Tests

The `wdf_compiler` toolchain also contains a test suite to ensure that the Wave Digital models generated by the toolchain compile correctly and generate accurate outputs. The test suite contains a variety of circuits including simple lowpass and bandpass filters, simple diode circuits, a common cathode amplifier circuit similar to the example shown in [18], the "tone stack" circuit from the '59 Fender Bassman [19], a Baxandall-style EQ circuit [20], and several more. The test suite covers code generated in `C++`, `C`, `Jai`, and `Rust`, as well as multiple data types, including "native" single- and double-precision floating-point types, and (in `C++`) a user-defined SIMD data type from the `XSIMD` software library.[3] Across all test circuits, the maximum error when compared with

[3] `https://github.com/xtensor-stack/xsimd`

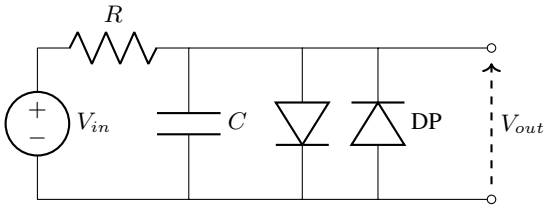

Figure 2: *Circuit schematic for the diode clipper circuit used for testing and performance measurements.*

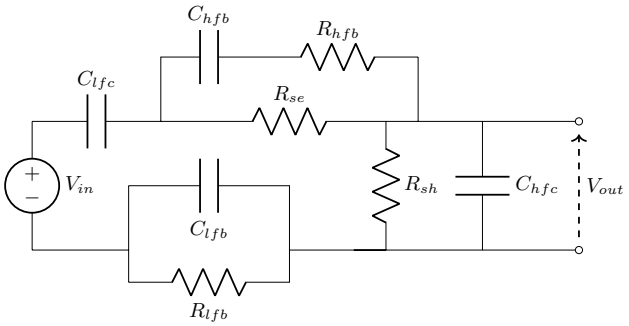

Figure 3: *Circuit schematic for the pre-amp EQ circuit used for testing and performance measurements.*

`chowdsp_wdf` was 7.12×10^{-5}, well within the expected tolerance for single-precision floating-point arithmetic.

3.3. Generating Optimized Code

The `wdf_compiler` toolchain uses several strategies to attempt to generate optimal code. First, the generated processing code is all within a single function (when no custom components are used), avoiding the use of expensive run-time abstractions such as virtual function calls, and allowing the compiler to aggressively optimize the processing code as a single unit. Second, the generated code is able to minimize the number of state variables used by the model. For instance, in the example presented in lis. 2, only one "state" variable and one "impedance" variable is present in the generated code.[4] By contrast, an implementation of an equivalent circuit model implemented with `chowdsp_wdf` contains approximately 20 persistent variables.

Another advantage of generating standalone processing code, rather than relying on a software library is that the standalone code can be more easily analyzed using tools including Compiler Explorer[5] and the LLVM Machine Code analyzer[6]. The use of these tools enabled further performance improvements in the generated code, for example, the use of 2-multiply series adaptors (see §8.1).

3.4. Circuit Reduction

At times, multiple circuit components can be replaced with a single circuit component (e.g. two resistors connected in series is equivalent to a single resistor with a resistance value equal to the sum of the resistors being replaced). In these cases, an optimal Wave Digital circuit model should use the variant of the circuit with fewer components, in order to avoid unnecessary computations. Similarly, the authors of `chowdsp_wdf` previously found that merging commonly found combinations of circuit components (e.g. a resistor and capacitor in series) into a single WDF element can improve the performance of a Wave Digital model [21].

To achieve optimal performance, `wdf_compiler` will attempt to "reduce" a given circuit by combining circuit elements wherever possible. Note that in the case where the voltage or current across a given circuit element is listed as one of the outputs of the circuit, that element will not be combined with any other circuit elements, since the desired output quantity may no longer be easily computable after the reductions have taken place. Despite this limitation, the circuit reduction process leads to significant performance improvements for some circuit models (see §8.2).

3.5. Dynamic Circuit Construction

Although `wdf_compiler` requires the circuit topology to be fixed at compilation time, it is still possible to dynamically construct a circuit model via "hot-reloading". A prototype audio plugin has been constructed as an example of this approach. [7]

The "hot-reloading" scheme works as follows: Given a text file containing a circuit description written in the WDF description language, the plugin will "watch" the file for changes. When the file is modified, the plugin invokes the `wdf_compiler` to generate a new file of `Jai` source code that implements the Wave Digital circuit model described in the text file. The plugin then invokes the `Jai` compiler to compile a dynamic library, which can be loaded by the plugin. Finally, the plugin forwards its audio callback to a method in the dynamic library so that the given audio stream is processed through the circuit model.

This approach allows circuit models to be constructed and modified dynamically, while still enjoying the performance benefits of the code generation approach. The full reloading process is typically completed in well under 1 second. Attempts to implement a similar system using `wdmodels` or `chowdsp_wdf` resulted in significantly slower reloading due to the slower compilation times associated with `Faust` and `C++` compilers (especially when attempting to compile heavily-templated `C++` code [22]).

4. MEASUREMENTS

4.1. Benchmark Setup

In order to validate the proposed approach, a series of performance benchmarks were constructed to compare the performance of the `C++` code generated by `wdf_compiler` with three existing alternatives: `RT-WDF`, `wdmodels`, and `chowdsp_wdf`. For WDF models constructed with `wdmodels`, the `Faust` code was transpiled to `C++`. The `chowdsp_wdf` circuit models were constructed as compile-time models using the library's template meta-programming interface. All `C++` code was compiled with Clang version 20.1.6, using compiler flags `-std=c++20 -O3`.

Four test circuits were selected: an RC lowpass filter (see fig. 1), a diode clipper (fig. 2), a custom-designed pre-amp EQ circuit

[4] Note that the impedance variable is not required to be persistent, although it may be convenient in some cases.

[5] `https://compiler-explorer.com`

[6] `https://llvm.org/docs/CommandGuide/llvm-mca.html`

[7] `https://github.com/jatinchowdhury18/wdf_compiler_plugin`

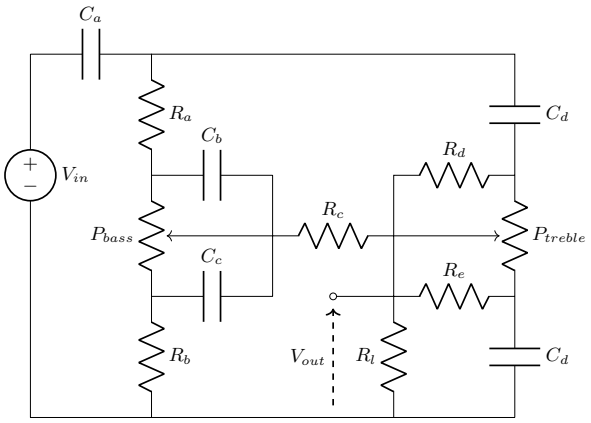

Figure 4: *Circuit schematic for the Baxandall EQ circuit used for testing and performance measurements.*

Table 1: *Performance measurements on an Apple M1 CPU. The fastest implementation of each circuit model (in cycles/sample) is shown in bold.*

Circuit	Framework	ns/sample	instr./sample	cycles/sample	instr./cycle
RC Lowpass	`RT-WDF`	22.54	251.02	72.07	3.48
RC Lowpass	`wdmodels`	6.02	17.01	19.04	0.89
RC Lowpass	`chowdsp_wdf`	6.97	18.01	22.09	0.82
RC Lowpass	`wdf_compiler`	1.37	12.00	**4.42**	2.71
Pre-Amp EQ	`RT-WDF`	107.66	1007.10	346.18	2.91
Pre-Amp EQ	`wdmodels`	22.91	83.02	73.30	1.13
Pre-Amp EQ	`chowdsp_wdf`	20.71	56.01	66.33	0.84
Pre-Amp EQ	`wdf_compiler`	18.42	66.02	**59.18**	1.12
Diode Clipper	`wdmodels`	116.52	566.93	374.53	1.51
Diode Clipper	`chowdsp_wdf`	41.05	102.04	131.78	0.77
Diode Clipper	`wdf_compiler`	35.08	76.02	**112.79**	0.67
Baxandall EQ	`RT-WDF`	183.01	2028.18	588.16	3.45
Baxandall EQ	`wdmodels`	18.54	163.02	59.23	2.75
Baxandall EQ	`chowdsp_wdf`	20.52	156.01	65.55	2.38
Baxandall EQ	`wdf_compiler`	11.35	55.01	**36.56**	1.50

(fig. 3), and a Baxandall-style EQ circuit (fig. 4). The RC lowpass filter was chosen as a simple circuit comprised of only linear circuit components. The pre-amp EQ was chosen as a more complicated linear circuit that does not require an $\mathcal{R}$-Type adaptor. The diode clipper was chosen as a circuit with a nonlinear component. Finally, the Baxandall EQ was chosen as a linear circuit that does require an $\mathcal{R}$-Type adaptor. The diode clipper circuit model was not implemented with `RT-WDF` since the library does not provide a "native" diode model.

Previous work (e.g. [4, 5]) evaluates run-time performance by measuring the time required by the model to process a given number of audio samples. While timing measurements are useful for comparing relative performance, they can suffer from measurement noise and provide limited insight into overall performance.

For each implementation, uniform random single-precision floating-point samples were processed by each circuit model implementation, and four measurements were collected[8]:

- Processing time (ns / sample)
- Instructions / sample
- Cycles / sample
- Instructions / cycle

While the processing time and cycles/sample measurements are roughly correlated, the cycles/sample measurements exhibited less variance between runs. Instructions/sample and instructions/cycle indicate how effectively each model minimizes overhead and utilizes the CPU's execution pipeline. Measurements were collected on two devices: a 2021 MacBook Pro with an Apple M1 Pro CPU, and a Beelink GTR7 with an AMD Ryzen 7 Zen 4 CPU.

4.2. Performance Score

Along with comparing the performance of circuit models constructed with `wdf_compiler` to existing alternatives, each implementation is also evaluated relative to a theoretical performance bound. For a given circuit implementation, a "performance score" may be calculated as follows.

[8]Event counters on MacOS were implemented using code borrowed from Daniel Lemire [23]. On Windows, a custom event counting system was implemented, inspired by previous work including Mārtiņš Možeiko's `miniperf` [24] and Casey Muratori's `pmctrace` [25].

First, the operations required by the circuit model are determined from the circuit topology, assuming no "circuit reductions" are performed. Next, given the CPU on which the circuit is being measured, instruction tables [26, 27] are used to estimate the number of cycles required to compute one output sample. Two estimates are computed. The first (L) represents the cycles-per-sample assuming the CPU is latency-bound, while the second (T) represents the cycles-per-sample assuming the CPU is throughput-bound. For superscalar CPUs, L is typically greater than T. Given the measured cycles-per-sample (M), the performance score (S) is then defined as

$$S = \frac{L - M}{L - T}. \tag{1}$$

Note that $S = 0\%$ and $S = 100\%$ correspond to the ideal latency-bound and throughput-bound performance respectively. Additionally, if the implementation is not able to achieve at least the ideal latency-bound performance, the score will be negative. Note that `wdf_compiler` is capable of achieving scores greater than 100% by using circuit reductions to eliminate operations that would otherwise be necessary.

The performance score was computed for implementations of the RC lowpass and pre-amp EQ circuit models. No performance score was computed for implementations of the diode clipper and Baxandall EQ circuit models, since both of those models relied on the use of custom components whose operation counts cannot be derived directly from the circuit topology.

4.3. Results

Tables 1 and 2 show the measurement results for each circuit model and implementation on the Apple M1 and AMD Zen 4 CPUs. The `wdf_compiler` was the fastest implementation for all four circuit models across both CPUs.

It should be noted that while `wdmodels`, `chowdsp_wdf`, and `wdf_compiler` all use the diode model derived in [16], `chowdsp_wdf` and `wdf_compiler` use D'Angelo et al.'s approximation of the Lambert $\mathcal{W}$ function [17] while `wdmodels` uses a less performant iterative solver [4]. `wdmodels` is unable to use the D'Angelo et al. approximation, since it relies on bit-level operations that are not supported by the `Faust` language.

Table 2: *Performance measurements on an AMD Zen 4 CPU. The fastest implementation of each circuit model (in cycles/sample) is shown in bold.*

Circuit	Framework	ns/sample	instr./sample	cycles/sample	instr./cycle
RC Lowpass	`RT-WDF`	20.31	268.06	99.85	2.68
RC Lowpass	`wdmodels`	2.98	14.51	15.03	0.97
RC Lowpass	`chowdsp_wdf`	6.44	38.02	31.73	1.20
RC Lowpass	`wdf_compiler`	1.21	9.51	**6.06**	1.57
Pre-Amp EQ	`RT-WDF`	99.98	1132.51	505.42	2.24
Pre-Amp EQ	`wdmodels`	15.92	128.05	80.29	1.59
Pre-Amp EQ	`chowdsp_wdf`	18.96	136.05	95.77	1.42
Pre-Amp EQ	`wdf_compiler`	12.77	83.03	**63.35**	1.31
Diode Clipper	`wdmodels`	90.17	354.14	389.10	0.91
Diode Clipper	`chowdsp_wdf`	28.31	126.24	141.86	0.89
Diode Clipper	`wdf_compiler`	22.84	96.07	**113.12**	0.85
Baxandall EQ	`RT-WDF`	160.94	2406.47	805.60	2.99
Baxandall EQ	`wdmodels`	14.66	261.04	72.95	3.58
Baxandall EQ	`chowdsp_wdf`	22.17	199.19	112.25	1.77
Baxandall EQ	`wdf_compiler`	8.58	95.02	**43.62**	2.18

Table 3: *Performance scores for the RC lowpass and pre-amp EQ circuits on the Apple M1 CPU.*

Circuit	`RT-WDF`	`wdmodels`	`chowdsp_wdf`	`wdf_compiler`
RC	-82.4%	60.6%	52.3%	**99.9%**
EQ	-128.6%	61.4%	66.3%	**71.3%**

Tables 3 and 4 show the performance scores for the RC lowpass and pre-amp EQ circuit model implementations on the Apple M1 and AMD Zen 4 CPUs respectively. While `RT-WDF` was unable to achieve a positive score, the other libraries were able to achieve scores above 50% in most cases. `wdf_compiler` was able to achieve a performance score above 70% for the pre-amp EQ circuit, and above 99% for the RC lowpass circuit, implying that the circuit model implementations can approach the theoretical limit for run-time performance. `wdf_compiler`'s implementation of the RC lowpass circuit was able to achieve a score greater than 100% on the AMD Zen 4 CPU, implying that the system's circuit reductions were able to reduce the model's required operations to the point that the theoretical throughput-bound performance of the unreduced model was surpassed.

4.4. Discussion

By treating WDF circuit model implementation as a code generation problem, `wdf_compiler` is able to aggressively simplify the model's data layout and processing operations, yielding better run-time performance than existing WDF implementations in most cases, as well as near-optimal performance compared to a theoretical ideal. While the `wdf_compiler` circuit models typically require fewer instructions-per-sample than the other implementations, they sometimes achieve fewer instructions-per-cycle. This result implies that further performance gains may be achievable by generating code that makes more efficient use of the CPU's execution pipeline.

5. CONCLUSION

This paper presents a performance-oriented toolchain for implementing Wave Digital circuit models via static code generation.

Table 4: *Performance scores for the RC lowpass and pre-amp EQ circuits on the AMD Zen 4 CPU.*

Circuit	`RT-WDF`	`wdmodels`	`chowdsp_wdf`	`wdf_compiler`
RC	-133.8%	80.9%	38.7%	**103.6%**
EQ	-189.9%	69.3%	59.9%	**79.7%**

Given a declarative description of a circuit topology, the system generates specialized processing code with minimal persistent state and no run-time abstraction.

Benchmark results show that the generated circuit models consistently outperform existing WDF implementations and, in many cases, approach theoretically ideal performance on modern CPUs.

Future work includes further improving instruction-level efficiency and using `wdf_compiler` as a platform for further experiments with Wave Digital circuit modeling. In particular, the data layout and processing code generated by `wdf_compiler` could perhaps be tuned for a specific CPU through an iterative process using direct experiments or simulations to analyze the performance of the model on the given CPU. Additionally, further research has already begun to explore automated translation of a circuit "netlist" into the WDF description language.

6. ACKNOWLEDGMENTS

Thanks to Daniel Lemire, Mārtiņš Možeiko, and Casey Muratori for their work on event tracing, which inspired the performance tests in this paper.

7. REFERENCES

[1] A. Fettweis, "Wave Digital Filters: Theory and Practice," *Proceedings of the IEEE*, vol. 74, no. 2, pp. 270–327, Feb. 1986.

[2] K. J. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters," Ph.D. dissertation, Stanford University, June 2016. [Online]. Available: https://searchworks.stanford.edu/view/11891203

[3] M. Rest, W. R. Dunkel, K. J. Werner, and J. O. Smith, "RT-WDF - A Modular Wave Digital Filter Library With Support For Arbitrary Topologies And Multiple Nonlinearities," in *Proc. Int. Conf. Digital Audio Effects (DAFx-16)*, Brno, Czech Republic, Sept. 5–9, 2016, pp. 287–294.

[4] D. Roosenburg, E. Stine, R. Michon, and J. Chowdhury, "A Wave-Digital Modeling Library for the Faust Programming Language," in *18th Sound and Music Computing Conference (SMC-2021)*, June 2021.

[5] J. Chowdhury. (2022) chowdsp_wdf: An Advanced C++ Library for Wave Digital Circuit Modelling. Available: https://arxiv.org/abs/2210.12554.pdf.

[6] U. Zölzer, Ed., *DAFx: Digital Audio Effects*, 2nd ed. Chichester, UK: John Wiley & Sons, 2011.

[7] G. Anthon, X. Lizarraga Seijas, and F. Font Corbera, "PY-WDF: an Open Source Library for Prototyping and Simulating Wave Digital Filter Circuits in Python," in *Proc. of the 26th Int. Conference on Digital Audio Effects (DAFx-23)*, Copenhagen, Denmark, Sept. 2023.

[8] A. Barrera, X. Lizarraga-Seijas, and F. Font, "Modeling the Pultec EQP-1A with Wave Digital Filters," in *Proceedings of the 21st Sound and Music Computing Conference (SMC 2024)*, Porto, Portugal, July 2024.

[9] C. Muratori, "The Big OOPs: Anatomy of a Thirty-Five Year Mistake," in *Better Software Conference 2025*, July 2025.

[10] F. G. Germain and K. J.Werner, "Design Principles For Lumped Model Discretisation Using Möbius Transforms," in *Proc. of the 18th Int. Conference on Digital Audio Effects (DAFx-15)*, Trondheim, Norway, Sept. 2015.

[11] K. J. Werner, A. Bernardini, J. O. Smith, and A. Sarti, "Modeling Circuits With Arbitrary Topologies and Active Linear Multiports Using Wave Digital Filters," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, pp. 4233–4246, 2018.

[12] J. Chowdhury and C. J. Clarke, "Emulating Diode Circuits with Differentiable Wave Digital Filters," in *Proc. of the 19th Sound and Music Computing Conference (SMC-22)*, Jun. 2022.

[13] C. C. Darabundit, D. Roosenburg, and J. O. Smith, "Neural Net Tube Models for Wave Digital Filters," in *Proc. of the 25th Int. Conference on Digital Audio Effects (DAFx-22)*, Vienna, Austria, Sept. 2022.

[14] O. Massi, A. I. Mezza, R. Giampiccolo, and A. Bernardini, "Deep Learning-Based Wave Digital Modeling of Rate-Dependent Hysteretic Nonlinearities for Virtual Analog Applications," *EURASIP Journal on Audio, Speech, and Music Processing*, Dec. 2023.

[15] R. Giampiccolo, G. Casanova, O. Massi, and A. Bernardini, "Wave Digital Modeling of Circuits with a Single Variable-μ Pentode based on Neural Networks," in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025.

[16] K. J. Werner, V. Nangia, A. Bernardini, J. O. Smith, and A. Sarti, "An Improved and Generalized Diode Clipper Model for Wave Digital Filters," in *139th Convention of the Audio Engineering Society, AES*, Oct. 2015.

[17] S. D'Angelo, L. Gabrielli, and L. Turchet, "Fast Approximation of the Lambert W Function for Virtual Analog Modelling," in *Proc. of the 22nd Int. Conference on Digital Audio Effects (DAFx-19)*, Birmingham, U.K., Sept. 2019.

[18] R. Giampiccolo, S. D'Angelo, A. Bernardini, and A. Sarti, "A Quadric Surface Model of Vacuum Tubes for Virtual Analog Applications," in *Proc. of the 26th Int. Conference on Digital Audio Effects (DAFx-23)*, Copenhagen, Denmark, Sept. 2023, pp. 296–303.

[19] D. T. Yeh and J. O. Smith, "Discretization of the '59 Fender Bassman Tone Stack," in *Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06)*, Montreal, Canada, Sept. 2006.

[20] P. J. Baxandall, "Negative-Feedback Tone Control," *Wireless World*, pp. 402–405, Oct. 1952.

[21] J. Chowdhury, "Combining WDF Elements," https://raw.githubusercontent.com/Chowdhury-DSP/chowdsp_wdf/439ae0/notes/combining_wdfs.pdf, June 2023, accessed: 2026-03-11.

[22] D. Mivelli, "Analyzing and Reducing Compilation Times for C++ Programs," Master's thesis, Linköping University, Department of Computer and Information Science, Software and Systems, June 2022.

[23] D. Lemire, "Code used on Daniel Lemire's blog," https://github.com/lemire/Code-used-on-Daniel-Lemire-s-blog/blob/ddb082/2023/03/21/performancecounters, March 2023, accessed: 2026-03-11.

[24] M. Možeiko, "`miniperf`," https://gist.github.com/mmozeiko/bd5923bcd9d20b5b9946691932ec95fa, October 2024, accessed: 2026-03-11.

[25] C. Muratori, "Real-time PMCs on Windows with ETW," https://www.computerenhance.com/p/real-time-pmcs-on-windows-with-etw, November 2024, accessed: 2026-03-11.

[26] A. Fog *et al.*, "Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs," *Copenhagen University College of Engineering*, vol. 93, 2011.

[27] D. Johnson. (2021) Apple M1 Microarchitectural Research. Available: https://dougallj.github.io/applecpu.

8. APPENDIX

8.1. 2-Multiply Series Adaptor

A WDF 3-port series adaptor is defined as follows [1] :

$$\begin{aligned} R_p &= R_1 + R_2 \\ b_0 &= -a_1 - a_2 \\ b_1 &= -\frac{R_1}{R_p}a_0 + \frac{R_2}{R_p}a_1 - \frac{R_1}{R_p}a_2 \\ b_2 &= -\frac{R_2}{R_p}a_0 - \frac{R_2}{R_p}a_1 + \frac{R_1}{R_p}a_2. \end{aligned} \tag{2}$$

where the a_n, b_n, and R_n are the incident voltage wave, reflected voltage wave, and port impedance (measured in Ohms) at port n.

Previous WDF literature (e.g. [2]) has noted that the above expression can be re-written in a form that requires a single multiplication, for example:

$$\begin{aligned} b_1 &= a_1 - \frac{R_1}{R_p}(a_0 - b_0) \\ b_2 &= -(a_0 + b_1). \end{aligned} \tag{3}$$

WDF circuit models often implement the 3-port series adaptor in 1-multiply form, with the goal of reducing the number of operations required to perform the adaptor computations, and specifically to reduce the number of multiplications [5].

However, the 1-multiply form shown above requires that every part of the computation occur sequentially, since most operations depend on the result of a previous operation. On superscalar CPUs (and specifically on modern CPUs that contain multiple execution slots capable of executing floating-point math operations), it may be beneficial to perform the series adaptor computation in such a way that multiple operations can be performed simultaneously, even if that results in more total instructions or even more multiplications.

Table 5: *Performance measurements of a WDF RC lowpass filter implemented with either a 1-multiply or 2-multiply series adaptor. Measurements were performed on an Apple M1 CPU.*

	ns/sample	instr./sample	cycles/sample	instr./cycle
1-mul	6.51	14.83	19.53	0.76
2-mul	5.01	13.59	15.09	0.90

Table 6: *Performance measurements for the test reductions circuit on an Apple M1 CPU. The fastest implementation of each circuit model (in cycles/sample) is shown in bold.*

Framework	ns/sample	instr./sample	cycles/sample	instr./cycle
`chowdsp_wdf`	40.51	185.04	129.94	1.42
`wdf_compiler`	10.69	27.01	**34.48**	0.78
`no_reduce`	34.99	122.03	112.55	1.08

Consider the following 2-multiply form of the adaptor:

$$\begin{aligned} b_1 &= a_1 - \frac{R_1}{R_p}(a_0 - b_0) \\ b_2 &= -a_0 - a_1 + \frac{R_1}{R_p}(a_0 - b_0). \end{aligned} \tag{4}$$

In this form there is no dependency between b_1 and b_2, allowing them to be computed in parallel. By default, `wdf_compiler` generates code to compute a 3-port series adaptor in this 2-multiply form, however this behaviour can be overridden via program argument if the user prefers the 1-multiply form (for example, if the generated code is going to be deployed on a CPU with only a single execution slot capable of floating-point arithmetic).

A comparison was performed on an Apple M1 CPU, measuring the performance of a WDF model of an RC lowpass filter circuit using either the 1-multiply or 2-multiply series adaptor (with circuit reductions disabled). Table 5 shows the measurement results. Note that in this case, the 2-multiply formulation results in fewer operations-per-sample (although this will not always be the case), as well as more operations-per-cycle, resulting in fewer cycles-per-sample and faster overall computation.

It should be noted that introducing certain compiler optimization flags (e.g. GCC and Clang's `-fassociative-math`) may result in the compiler introducing this kind of optimization automatically, or otherwise re-ordering operations to better utilize the CPU's execution pipeline.

8.2. Circuit Reductions Performance Impact

An additional test circuit model was constructed to specifically test the "circuit reduction" process described in §3.4. Table 6 shows the results of performance measurements on an Apple M1 CPU for three implementations of the test circuit: one using `chowdsp_wdf`, one using `wdf_compiler` with reductions enabled, and one using `wdf_compiler` with the `-no_reduce` option to disable circuit reductions. With reductions enabled, `wdf_compiler` performs 16 total reductions on the circuit.

Compared to the full circuit, `wdf_compiler`'s implementation of the reduced circuit uses fewer than 25% of the instructions-per-sample and fewer than 50% of the cycles-per-sample.

Table 7: *Performance measurements on an Apple M1 CPU for an RC lowpass filter with different data types. The fastest implementation of each circuit model (in cycles/sample) is shown in bold.*

Framework	Data Type	ns/sample	instr./sample	cycles/sample	instr./cycle
`wdmodels`	`double`	6.15	19.00	19.06	1.00
`chowdsp_wdf`	`double`	7.08	18.00	22.04	0.82
`wdf_compiler`	`double`	1.37	12.00	**4.42**	2.71
`chowdsp_wdf`	`batch<float>`	14.11	40.01	43.30	0.92
`wdf_compiler`	`batch<float>`	2.20	10.00	**7.05**	1.42

Table 8: *Performance measurements on an Apple M1 CPU for models of three test circuits generated in different programming languages. All measurements are given in nano-seconds/sample. The fastest implementation of each circuit model is shown in bold.*

Circuit	`C++`	`C`	`Jai`	`Rust`
RC Lowpass	**1.38**	1.58	2.38	2.50
Pre-Amp EQ	**18.38**	19.96	19.01	19.29
Baxandall EQ	11.36	**11.00**	24.70	25.24

8.3. Performance Measurements for Other Data Types

Performance measurements for the RC lowpass filter circuit was repeated using two additional data types: `double` and `batch<float>` from the `XSIMD` library. The `double` measurement was performed with the `wdf_compiler`, `chowdsp_wdf`, and `wdmodels` implementations. The `batch<float>` measurement was performed with the `wdf_compiler` and `chowdsp_wdf` implementations.

Table 7 shows the results of the performance measurements for these additional data types on an Apple M1 CPU. In the double-precision tests, `wdf_compiler` outperforms the other libraries, although notably, both the `wdf_compiler` and `chowdsp_wdf` implementations outperform their single-precision counterparts. In the SIMD tests, `wdf_compiler` is more than 6 times faster than `chowdsp_wdf`.

8.4. Performance Measurements for Other Languages

Since `wdf_compiler` is capable of generating circuit models in different programming languages, a performance measurement was performed across all supported languages. Three circuits were tested: the RC lowpass filter, pre-amp EQ, and Baxandall EQ. `Rust` code was compiled using `rustc` version 1.93.1 with `opt-level=3`. `Jai` code was compiled with the compiler beta version 0.2.026, using the `-optimized` flag and the LLVM backend. Table 8 shows the results of the performance measurements. The generated `C++` code was the most performant in most cases, however, the generated `C` was the fastest implementation of the Baxandall EQ circuit model. The generated `Rust` code was the least performant in all cases.