A Time-Travel Debugger for Web Audio Applications

Andrew Thompson
Electronic Engineering & Computer Science
Queen Mary University of London
andrew.thompson@qmul.ac.uk

György Fazekas
Electronic Engineering & Computer Science
Queen Mary University of London
g.fazekas@qmul.ac.uk

Geraint Wiggins
Department of Computer Science
Vrije Universiteit Brussel
geraint@ai.vub.ac.be

ABSTRACT

Developing real-time audio applications, particularly those with an element of user interaction, can be a difficult task. When things go wrong, it can be challenging to locate the source of a problem when many parts of the program are connected and interacting with one another in real-time.

We present a time-travel debugger for the Flow Web Audio framework that allows developers to record a session interacting with their program, playback that session with the original timing still intact, and step through individual events to inspect the program state at any point in time. In contrast to the browser's native debugging features, audio processing remains active while the time-travel debugger is enabled, allowing developers to listen out for audio bugs or unexpected behaviour.

We describe three example use-cases for such a debugger. The first is error reproduction using the debugger's JSON import/export capabilities to ensure the developer can replicate problematic sessions. The second is using the debugger as an exploratory aid instead of a tool for error finding. Finally, we consider opportunities for the debugger's technology to be used by end-users as a means of recording, sharing, and remixing ideas. We conclude with some options for future development, including expanding the debugger's program state inspector to allow for in situ data updates, visualisation of the current audio graph similar to existing Web Audio inspectors, and possible methods of evaluating the debugger's effectiveness in the scenarios described.

CCS Concepts

•**Applied computing** → *Sound and music computing;* •**Software and its engineering** → **Software maintenance tools;**

Keywords

Time-travel debugging, Reverse debugging, Web Audio API, Declarative programming, Exploratory programming, Interactive audio applications

Web Audio Conference WAC-2021, July 5–7, 2021, Barcelona, Spain.

1. INTRODUCTION

A traditional debugger allows the programmer to mark a breakpoint that pauses program execution and lets them inspect the program's state. The programmer can insert multiple breakpoints and step forward to the next one by resuming the program up until that point. Reverse debugging builds on this idea by allowing the programmer to step backwards from a breakpoint to uncover the cause of an error. Time-travel debugging provides both capabilities, allowing the programmer to step forward and backwards in time. These debuggers provide greater context when debugging, allowing the developer to understand how a program got to a particular state and observe changes to that state over time.

Time-travel debugging in JavaScript has been an area of interest for some time. The Jardis debugger is a low-level time-travelling debugger for the Node.js runtime that works by hooking into the existing event loop and performing regular snapshots of the entire program's state [2]. The debugger also supports so-called "postmortem debugging, " allowing a full program recording to be captured and reconstructed after the fact. Full-program snapshots are streamed to a remote server, where they can later be loaded on another machine.

A similar tool for the browser, Timelapse, was developed by Burg et al [3]. Like Jardis, Timelapse intercepts low-level events in the runtime's event loop. It differs in its record/replay strategy, choosing to recreate events during playback instead of capturing entire program snapshots.

While Jardis and Timelapse rely on platform-specific extensions to work (Node.js and WebKit, respectively), McFly presents a method portable across all spec-compliant browsers [15]. Although implemented as an extension to the Microsoft Edge browser, they note that McFly's architecture builds on existing Web standards. This debugger captures "checkpoints" of program state and configurable intervals. In doing so, McFly supports step-backwards commands at *interactive* speeds where neither Jardis nor Timelapse can.

For debugging Web Audio applications specifically, the literature is less rich. Adenot provides comprehensive notes on the performance characteristics of various audio nodes and tips on how to improve audio processing performance [1]. Mozilla maintained the now-deprecated Web Audio Editor [10] and Google offers a browser extension for their Chrome browser titled the Web Audio Inspector [5]. Both tools provide a visual graph corresponding to the current Web Audio graph, letting developers get an overview of the signal chain and inspect the parameters of individual nodes. Develop-

ers may also edit individual parameters in the Web Audio Editor, and these changes are reflected in real-time. Choi developed Canopy, a testing and verification Web app for Web Audio code [4]. Like the others, this tool provides a graphical view of the audio graph and provides capabilities for audio recording, exporting, and waveform inspection.

We can see these tools, while undoubtedly helpful, primarily focus on visualising the audio graph, improving performance, or analysing signal processing code. In other words, these tools can provide insight into whether a program is working correctly or not but are largely incapable of determining how or why that program is incorrect.

2. TIME-TRAVEL DEBUGGING

The time-travel debugging systems we have mentioned share some common shortcomings, particularly when considering their use in *audio* application development. These systems only support one browser, either using a browser-specific extension API as seen in McFly or by building a new browser entirely as in Timelapse. It is important to note that different browsers often have different implementations of the same API, including the Web Audio API [8], so cross-browser testing is critical in ensuring the application will work for all users. Crucially, none of these systems explicitly support the Web Audio API.

We present a time-travel debugger for Web Audio applications that attempts to address these issues. The most important features of this debugger are listed below:

- **The audio context remains running while the debugger is active.** This means the audio output is updated in real-time when stepping between actions in the debugger.
- **Timing information is preserved** between actions bringing support for accurate timed playback of a sequence of events.
- **JSON import/export** for session recordings allows for postmortem debugging.
- **Cross-platform support**: the debugger is available in any browser that supports the Web Audio API.

We have implemented our time-travel debugger as part of the Flow framework, a declarative framework for interactive Web Audio applications based on the Model-View-Update architecture [14]. Applications written in Flow have a single source of application state, known as the model. Changes to this model are handled by a single update function that receives the current model and an action describing some event (e.g. the user clicked a button or a tick from a timer) and returns a new model in response. That is then passed to separate functions for rendering the HTML and producing a Web Audio graph (figure 1).

2.1 The Flow Framework

As a simple introduction to the Flow framework and the style of application development it promotes, we first present a simple "mouse theremin" whereby an oscillator's frequency and amplitude are controlled by mouse position.

First, we begin with an initial model. For such a simple program, we only need the model to store the oscillator's current frequency and amplitude.

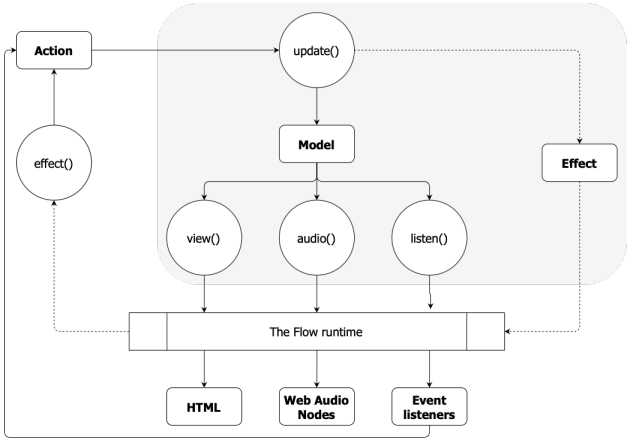

Figure 1: The Flow framework's MVU architecture. Functions are contained in circles; data are bolded and contained in rectangles. Arrows mark the flow of data in a program, and a dotted line indicates an optional flow. The grey shaded area represents the pure core of a Flow application; this code should be free of side effects.

```
function init () {
    return {  freq: 440, amp: 0 }
}
```

In Flow, updates to the program's state may only happen in the user-defined update function. The runtime calls this function whenever an event is fired that produces an action.

```
const MouseMoved = ({ x, y }) =>
    Action('MouseMoved', { x, y })

function update ({ action, payload }, model) {
    switch (action) {
        case 'MouseMoved': {
            const { x, y } = payload
            return {
                freq: x,
                amp: y / window.innerHieght
            }
        }

        default: return model
    }
}
```

Action-producing event listeners are returned from a user-defined listen function. The Flow runtime is responsible for managing the registering and deregistering of event listeners. Instead, the user-defined listen function returns an array of simple data descriptions of event listeners.

```
function listen (model) {
    return [
        DOM.Event.mousemove('window', MouseMoved)
    ]
}
```

Finally, we define an audio function called the current model and returns an object describing the audio graph.

As with event listeners, the Flow runtime is responsible for creating, updating, and removing Web Audio nodes from the graph. Crucial to the operation of our time-travel debugger is this notion that the audio graph is simply a function of a program's state. If the user code were to manage Web Audio nodes directly, the debugger would have to perform complex and expensive reflection of an audio context and its nodes to determine what has changed and when.

```
function audio (model) {
    return [
        osc([ frequency(model.freq) ], [
            gain([ amp(model.amp) ], [
                dac()
            ])
        ])
    ]
}
```

We have omitted the user-defined view function from this example and the necessary imports of the framework packages for brevity. A complete walkthrough of a Flow application is available online[1].

An essential property of Flow applications is their determinism. For any initial starting model m and a fixed sequence of actions s, the application will produce the same audio graph and DOM tree. Flow can reasonably guarantee this property because the application code is expected to be written in a pure functional style. Side effects are performed by Flow's runtime, with the result of those effects being passed to the user-defined update function as an action in the same way that interaction events are handled. Flow's time-travel debugger works by hooking into the runtime and capturing incoming actions and their resultant models. Additionally, these snapshots are paired with an accurate timestamp relative to the program's start time.

Because producing actions is the only way to respond to events or return from side effects, capturing these actions provide us with a simple yet powerful method of recording a program's history.

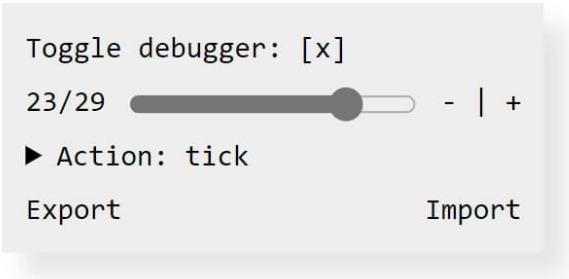

Figure 2: The collapsed debugger view shows information such as the number of actions captured in the recording as well as the number and name of the current action. Buttons exist to import an existing JSON recording or export the current recording as JSON.

Developers may then enable the debugger. Doing so unregisters all event listeners in the application (including timing and audio-related events and pending asynchronous operations such as HTTP requests) and allows the developer to step backwards and forwards in time. This is achieved through a debugging window that appears whenever an application is running in debug mode (figure 2) that can be expanded to show the current application model as in figure 3.

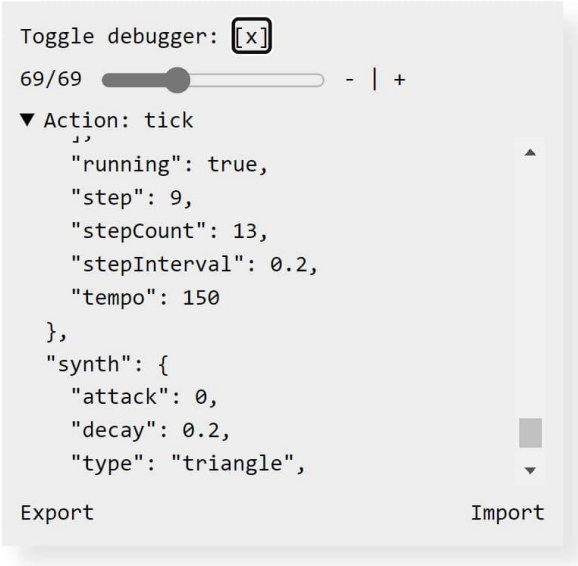

Figure 3: The debugger's expanded view shows the entire application model.

When stepping to any action, the model associated with that action is passed to the user-defined audio and view functions and then rendered. This approach differs significantly from existing time-travel debugging solutions that capture as much information as possible to recreate the program state themselves. The debugger can work this way because the model is always the single source of truth for audio and visual output; side effects are managed by the runtime and must make their way through the model-view-update architecture if their result is to be observed.

Notably, the audio context remains running while the debugger is enabled. This means the runtime faithfully renders audio graphs produced by stepping through actions. There is little difference between the actions created while the program is running and those produced while the debugger is active from the runtime's perspective. This powerful feature gives developers a new perspective when debugging; not only can they inspect program state for errors, they can also *listen* for them.

2.2 Scheduling Updates

Of course, not all changes to the audio output will happen immediately after an action is received. The debugger must accurately reproduce scheduled audio events such as audio parameter automation even when an action is stepped to at some arbitrary point in time.

In a typical Web Audio application a developer may schedule something to happen one second from *now* using the `AudioContext.currentTime` property.

[1] https://flow-lang.github.io/examples/00-polysynth

```
osc.freq.setValueAtTime(440,
    context.currentTime + 1
)
```

Doing so means *now* is entirely relative to when the code was run. Flow calls the user-defined audio and view functions after every call to update and diffs their result, using that information to update the internal stateful audio graph and DOM. Because Flow encourages a pure functional style of programming, it is expected that the same audio graph will be produced for any given model regardless of *when* that audio graph is produced.

Flow and the debugger side-step issues related to relative time in a few ways. First, developers cannot directly access the underlying AudioContext and thus cannot query the current time. Instead, actions are always accompanied by an audio timestamp of when they were created, which developers can store in their application model if they need to.

Second, time-based events can be subscribed to in the same manner as user interaction events. We have already demonstrated the `DOM.Event.mousemove` listener, and below is an event listener that triggers an action at a fixed interval.

```
Audio.every('Tick',
    500, // milliseconds
    time => Action('Tick', { time })
)
```

Because time is made explicit in Flow, the debugger can properly handle scheduled audio events regardless of when an action is stepped to in the timeline. Because timestamps accompany every action, the debugger can inspect any time values stored in the application model and calculate their time relative to the action time. When the user then retriggers that action during debugging, we can use these relative time values to schedule events relative to the new *now*.

Consider a model that stores three time values in response to some `Tick` action.

```
function update ({ action, payload, now }, model) {
    switch(action) {
        case 'Tick':
            return {
                a: now,
                b: now + 1,
                c: Time.absolute(2)
            }
    }
}
```

If this action is triggered at audio time 2.5, then the time values stored in the model would be as follows:

```
{ a: 2.5, b: 3.5, c: 2 }
```

The calculated relative time values would then be:

```
{ a: 0, b: 1, c: -0.5 }
```

If we then enabled the time-travel debugger and trigger this action sometime in the future, at audio time 10, we can use the relative time values to substitute the values stored in the model with new ones relative to the current time. This substitution happens before the model is passed to the audio function but does not affect the model passed to the view function.

2.3 Timed Playback

For many applications, the time-travel debugging we have described so far would be sufficient. However, for audio applications, *when* an action was received is often just as important as what action it is. Sometimes it may be necessary to step through sequences of actions, or indeed the entire recording, with the timing between each action preserved.

Audio scheduling is a well-discussed topic in the Web Audio community. Although the Web Audio API itself does not provide a mechanism for accurate scheduling of code beyond a handful of specific cases, several libraries and solutions have been developed [16, 12, 13] to remedy this. Flow's time-travel debugger uses a simple interval timer and a look-ahead window to schedule timed playback of events.

For other Web Audio applications, the jitter inherent in using `setInterval` is often too significant to use as a timing source on its own. For Flow, this is not the case. Because audio time is always explicit, the scheduler can be optimistic with its reporting of the current time such that if the current time satisfies the following:

```
currentTime >= targetTime - lookahead
```

The scheduler can simply dispatch the appropriate action with the *target time* instead of the current time. Because the user-written audio code can only depend on the model for timing information, the jitter present in `setInterval` is inconsequential. The importance of when audio code is run is replaced by the importance of what time is stored in the model.

For example, consider a sequence of three actions with the audio timings `[ 0, 0.5, 0.75 ]`. Assuming we start from the first action, when the timed-playback feature of the debugger is enabled, the first action is immediately dispatched, and the audio scheduler is used to schedule the following action for audio time 0.5. If we use a look-ahead value of 0.1, we can compensate for jitter and dispatch the action optimistically if the time is, for example, 0.45. Importantly, the time will be reported as 0.5 regardless of if the action is dispatched prematurely, and so any timing-sensitive scheduling is unaffected.

This timed playback is a powerful feature not afforded by other Web Audio development tools and allows the developer to take a much more holistic approach to debugging. Complex interactions between the user interface and the audio code can now be captured and accurately recreated, uncovering potential subtle bugs or issues.

3. USE CASES

Below, we describe three use-cases for the time-travel debugger we have developed. The first is one of error recreation; making use of Flow and the debugger's determinism to import a session created by an end-user or another developer and uncover the same bug(s) as them. The second is using the debugger as a tool for exploratory programming: recording a session interacting with the program, modifying the audio processing code, and listening to the new output. The final use-case considers how end-users could use the tool instead of developers to create and share recordings as a creative exercise rather than an engineering one.

3.1 Error Recreation

The primary use of time-travel debuggers is to debug tricky bugs resulting from a specific sequence of events or interactions. So-called *Heisenbugs* are difficult to debug with traditional debugging tools as the mere act of attempting to reproduce or measure the bug can cause it to go away [6]. The presented time-travel debugger allows developers to record a session interacting with the application, discover a bug, and then export that recording for later testing or sharing with other developers.

Our time-travel debugger works alongside native debugging tools. Developers can first step through a high-level timeline of the application's history, observe changes to the program state, and listen for audio bugs or unexpected behaviour. When they have identified the problem or need more fine-grained information about a specific function call, they can insert debugger statements for more detailed inspection.

The Flow architecture further facilitates the ability for the developer to effectively "zero in" on a problem, and the initial guesswork in placing breakpoints or console logs is replaced with a more holistic approach. First, developers step through a recording to identify when a bug or issue occurred. Then, knowing which action was dispatched for the potentially broken code, the developer can *then* use traditional debugger breakpoints or console logs in functions called in that specific branch of the update function. Simply knowing what actions were dispatched and when can already be helpful information when attempting to debug a problem.

3.2 Experimentation

Although time-travel debuggers are typically used to reproduce and locate errors, there is also the potential for their use as aids in exploratory programming. Exploratory programming is a method of software development defined by Kery and Myers as having two main properties. The first is that code is written as a means of experimentation or prototyping different ideas, and the second is that the goal is open-ended and evolving [7]. So-called "debugging into existence" is the practice of using tooling such as a debugger, inspector, or compiler to rapidly experiment with code. Learning how and why particular pieces of code might break before moving on to further experimentation [11].

Exploratory programming is a particularly common practice in audio and visual programming, where the correctness of a program's output is subjective and less well-defined [9]. Subtle tweaks to parameters in a complex audio graph can often profoundly impact the audio output, but it can be challenging to understand whether changes in sound are attributed to changes in the audio graph or how the interface was interacted with.

Programmers can use the time-travel debugger during this experimentation and tweaking stage, creating the user interface with the appropriate event listeners set up and then recording a session to capture a specific sequence of interactions. Now, the developer can export this recording, tweak some parameters or modify the audio graph, and reload the recording, using the debugger's timed playback to accurately recreate the session every time.

Developers can then be sure that changes to the audio output have only occurred because of their audio processing code and not because of subtle differences in interaction.

3.3 Sharing

Finally, we should consider how end-users can use the technology employed by the time-travel debugger beyond development contexts. We have seen that session recordings are encoded as JSON and contain enough information to be accurately recreated at a later point in time. It is not difficult to see how such technology could be used to share recordings among *users* of an application instead of the developers. These recordings could be used in place of audio recordings and would come with several advantages.

Session recordings encoded as JSON are likely to be smaller than the equivalent audio recording, even when considering compression. They also have the benefit of being interactive. When one user shares a recording with another, that second user could interact with the application using the recording as a starting point.

As presented, the time-travel debugger is not entirely suitable for such a use-case, but providing a developer-facing API for creating and loading session recordings would be a promising first step.

4. DISCUSSION

We have presented a time-travelling debugger for Web Audio applications using the Flow framework. Such a tool allows program execution to be paused and a program's history to be stepped forwards and backwards in time. Compared to existing debuggers, our tool does not suspend audio processing, allowing developers to recreate and *listen* for bugs. Finally, our debugger allows for accurate timed playback of a portion or entirety of a program's history.

Although the debugger we have presented here is functional, there are still some considerations to keep in mind and opportunities for future work. We have traded a dependency on a specific browser or platform for a dependency on a specific framework. The Flow framework imposes a specific application architecture and encourages a declarative pure functional style of programming that makes porting existing Web Audio applications to the framework potentially challenging. A browser *and* framework-agnostic solution would be desirable, but it is not immediately clear how that could be achieved.

One significant caveat with tying the debugger to the Flow framework is that the framework's determinism, and thus the debugger's reliability, relies in no small part on the assumption that developers write their application code using Flow's effects system to perform side effects. Doing so guarantees that effectful operations such as performing HTTP requests or getting the current time from the audio context produce an action passed to the update function, effectively isolating application code from side effects. Ultimately, developer code is still JavaScript, and so the framework itself cannot prohibit developers from writing effectful code or making use of global mutable variables.

Similarly, loading a recording into an application that differs from the version when recorded can potentially cause problems, specifically if an application no longer handles actions stored in the recording. Indeed, the debugger does not verify if the current application is suitable for the imported recording at all, meaning a recording can be loaded into an entirely different application.

5. FUTURE WORK

The potential to share session recordings as an alternative to audio recordings or application presets is an exciting avenue for future investigation. Currently, the time-travel debugger is only available when the application is initialised with a special debug flag. Given the tool's focus on debugging and developer exploration, the debugger is implemented as part of the framework's runtime and not exposed to the developer. By providing an API for the debugger's functionality, application code could start, stop, import, and export session recordings programmatically without presenting the debugging GUI to end-users of the application.

We plan to expand the debugger's explorer of the application model to allow for in situ modification of primitive values. For example, if the model for a simple synthesiser contained a field for the current frequency, the developer could enable the debugger and modify that value from the debugger's GUI without modifying the source code and reloading the page. This functionality would further support the debugger as a tool for exploratory programming, giving developers an easy way to tweak values and listen to their effect on the audio output without jumping between the running application and their code editor.

Additionally, we hope to add support for audio graph visualisation similar to the Mozilla Audio Editor or Chrome's Web Audio Inspector extension. The Flow framework uses a virtual audio graph, similar to the VDOM approach used by popular front-end frameworks such as React or Vue. Briefly, this means the audio graph is represented as a simple data structure, and the framework's runtime is responsible for turning that data structure into the actual stateful Web Audio nodes. Visualising such a data structure is more straightforward than visualising a standard Web Audio graph as it does not require modifying existing audio nodes to detect changes, creations, or deletions.

Finally, it would be insightful to perform some user evaluations on the debugger to verify its effectiveness as a debugging and exploratory programming tool and identify more specific areas for improvement. We intend to look towards human-computer interaction and the psychology of programming to guide the design of such evaluations.

6. ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council and the Arts and Humanities Research Council (grant EP/L01632X/1), ESPRC and AHRC Centre for Doctoral Training in Media and Arts Technology at Queen Mary University of London, School of Electronic Engineering and Computer Science.

7. REFERENCES

[1] P. Adenot. Web audio api performance and debugging notes. https://padenot.github.io/web-audio-perf, 2017.

[2] E. T. Barr, M. Marron, E. Maurer, D. Moseley, and G. Seth. Time-travel debugging for javascript/node. js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1003–1007, 2016.

[3] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484, 2013.

[4] H. Choi. canopy. https://github.com/hoch/canopy, 2020.

[5] Google. Web audio inspector. https://github.com/google/audion, 2017.

[6] M. Grottke and K. S. Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.

[7] M. B. Kery and B. A. Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29. IEEE, 2017.

[8] V. Lazzarini, S. Yi, and J. Timoney. Web audio: Some critical considerations. In *Proceedings of the VI Ubiquitous Music Workshop*, 2015. http://eprints.maynoothuniversity.ie/9421/1/web audio criticism.pdf.

[9] N. Montfort. *Exploratory programming for the arts and humanities*. MIT Press, 2016.

[10] Mozilla. Web audio editor. https://developer.mozilla.org/en-US/docs/Tools/Web_Audio_Editor, 2020.

[11] M. B. Rosson and J. M. Carroll. Active programming strategies in reuse. In *European Conference on Object-Oriented Programming*, pages 4–20. Springer, 1993.

[12] N. Schnell, V. Saiz, K. Barkati, and S. Goldszmidt. Of time engines and masters an api for scheduling and synchronizing the generation and playback of event sequences and media streams for the web audio api. 2015.

[13] J. Sullivan. Alternatives to lookahead audio scheduling. In *Proceedings of the 2nd Web Audio Conference (WAC)*. Georgia Institute of Technology, 2016.

[14] A. Thompson and G. Fazekas. A model-view-update framework for interactive web audio applications. In *Proceedings of the 14th International Audio Mostly Conference: A Journey in Sound*, pages 219–222, 2019.

[15] J. Vilk, E. D. Berger, J. Mickens, and M. Marron. Mcfly: Time-travel debugging for the web. *arXiv preprint arXiv:1810.11865*, 2018.

[16] C. Wilson. A tale of two clocks - scheduling web audio with precision - html5 rocks, 2015.