Time-Travel Debugging for JavaScript/Node.js

Earl T. Barr
University College London, UK
e.barr@ucl.ac.uk

Mark Marron
Microsoft Research, USA
marron@microsoft.com

Ed Maurer
Microsoft, USA
edmaurer@microsoft.com

Dan Moseley
Microsoft, USA
Dan.Moseley@microsoft.com

Gaurav Seth
Microsoft, USA
Seth.Gaurav@microsoft.com

ABSTRACT

Time-traveling in the execution history of a program during debugging enables a developer to precisely track and understand the sequence of statements and program values leading to an error. To provide this functionality to real world developers, we embarked on a two year journey to create a production quality time-traveling debugger in Microsoft's open-source ChakraCore JavaScript engine and the popular Node.js application framework.

CCS Concepts

•Software and its engineering → Software testing and debugging;

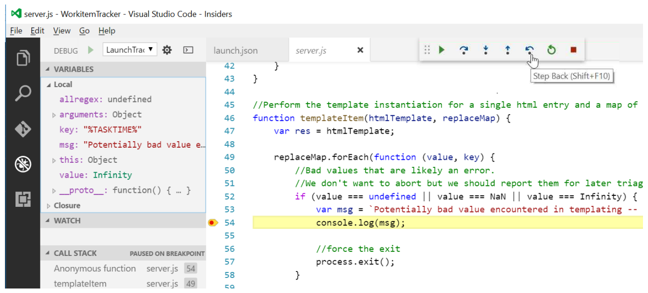
Keywords

Time-Travel Debugging, JavaScript, Node.js

1. INTRODUCTION

Modern integrated development environments (IDEs) provide a range of tools for setting breakpoints, examining program state, and manually logging execution. These features enable developers to quickly track down localized bugs, provided all of the relevant values remain on the call-stack and the root cause of the failure is close to where the error manifests itself. However, they provide only rudimentary support to iteratively track down more difficult errors: developers often resort to repeatedly setting breakpoints and rerunning the program. The dynamic nature of JavaScript and the extensive use of callbacks/promises in Node.js can make this process especially tedious and time-consuming.

Time-traveling debuggers offer an attractive alternative to the iterative process of debugging nontrivial errors using a traditional debugger. Instead of requiring developers to stop debugging, set a new breakpoint, and then reproduce the desired execution to hit that breakpoint, time-traveling debuggers allow a developer to simply press a button to step-back in a program's execution. We present JARDIS, a time-traveling debugger for JavaScript; it provides a rich set of functionality for recording and navigating the execution history of a program including:



```
allregex: undefined
arguments: Object
key: "%TASKTIME%"
msg: "Potentially bad value e…
this: Object
value: Infinity
__proto__: function() { … }
```

```
Anonymous function   server.js 54
templateItem         server.js 49
```

```
    }
}

//Perform the template instantiation for a single html entry and a map of
function templateItem(htmlTemplate, replaceMap) {
    var res = htmlTemplate;

    replaceMap.forEach(function (value, key) {
        //Bad values that are likely an error.
        //We don't want to abort but we should report them for later triag
        if (value === undefined || value === NaN || value === Infinity) {
            var msg = `Potentially bad value encountered in templating --
            console.log(msg);

            //force the exit
            process.exit();
        }
```

Figure 1: Visual Studio Code with extra time-travel functionality (Step-Back button in top action bar) at a breakpoint.

- Options for both using time-travel during local debugging and for recording a trace in production for postmortem debugging or other analysis (Section 2.1).
- Reverse-Step Local and Dynamic operations (Section 2.2) that allow the developer to step-back in time to the previously executed statement in the *current* function or to step-back in time to the previously executed statement in *any frame* including exception throws or callee returns.
- Reverse to Callback Origin operation (Section 2.3) that reverses time to the line that *registered* the currently executing callback *e.g.* the call to `setTimeout(...)` where the currently executing function was registered.

In combination with the low-overhead imposed by JARDIS (under 2% runtime increase in Section 4), these features provide a system that is suitable for use as both the *default* debugger for Node.js applications and for collecting execution histories from in-production deployments. Thus, JARDIS represents a major advance in state of the art debugging tools and provides an enabling technology for related research areas including interrogative debugging, automated fault localization, and performance profiling.

2. FEATURES

JARDIS provides a range of features for debugging programs using JavaScript/Node.js and improves developer productivity in both local and postmortem debugging scenarios[1].

2.1 Local Debugging and Offline Recording

JARDIS starts by augmenting the conventional graphical debugger, including its variable and call stack displays, breakpoints, and single-stepping, with the ability to reverse step through code execution via its **Reverse-Step** button as shown in Figure 1.

[1]A video demo of JARDIS is available at [12].

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983933

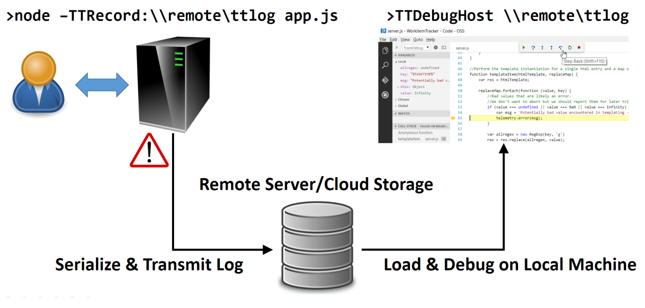

>node -TTRecord:\\remote\ttlog app.js
>TTDebugHost \\remote\ttlog
Remote Server/Cloud Storage
Serialize & Transmit Log
Load & Debug on Local Machine

Figure 2: Remote recording with a telemetry service, followed by postmortem debugging at full fidelity.

JARDIS also supports postmortem debugging, as shown in the workflow in Figure 2. In this scenario, the developer runs their application under JARDIS's record mode when deployed to their users, given their consent. When an error occurs, the execution trace is sent to the specified location for debugging later. The collected trace has sufficient information to replay the execution of the program exactly, including any non-determinism, and show the value of any JavaScript program variable that the developer might be interested in. Thus, the debugging experience is the same as if the developer had a debugger attached to the remote Node.js process when the trace was collected.

To record the trace, the developer simply invokes Node with an additional command line flag, `-TTRecord:[uri]`, where the `uri` parameter indicates the location (a file or remote server) to save the trace. By default, we continuously record approximately the most recent 2–4 seconds of execution in a ring buffer and save the data if an unhandled exception occurs, a call to `process.exit` is made, or a dump is explicitly requested by the the application. All of these parameters can be adjusted as desired.

Later, the developer can load and debug the serialized trace by invoking the replay host with `TTDebugHost [uri]` and attaching a debugger. In Figure 2, Visual Studio Code [23] is configured to do this and execute the code/trace to the point where the exception is thrown. When this point is reached, the debugging experience is virtually identical to the standard local debugging scenario: the developer can inspect any value of interest, view the stack frame, set breakpoints, step forward/back in the code, and even evaluate expressions in the watch window.

2.2 Reverse-Step Operations

The first type of reverse-step operation, and most commonly useful operation, provided by the JARDIS tool is a reverse-step (`rs`) to the previously executed line in the *current* call frame. This is simply the reverse version of the forward step provided by existing graphical and command line debuggers. The next type of reverse operation provided by JARDIS is reverse-step dynamic (`rsd`) which acts as the reverse version of step-into operations in existing debuggers. This operation steps-back in time to the previously executed statement in *any frame*.

To illustrate the various use cases for these operations and how they differ, consider the code in Figure 3. If the current breakpoint is at line 14, labeled `bp1`, then the `rsd` operation reverse-executes to the return statement of the previously called method (line 2 in `bar`) while the `rs` operation reverse-executes to line 13. When the current position is the result of an exception, line 10 labeled `bp2`, the `rsd` operation reverse-executes to the throwing statement at line 2 in bar, while the `rs` operation reverse-executes to line 8. In all other cases, the behavior of `rsd` and `rs` is the same.

```
 1  function bar(x) {
 2    return x.g;
 3  }
 4
 5  function foo() {
 6    var v = undefined;
 7    try
 8    { bar(v); }
 9    catch(e)
10    { /*bp2*/ console.log('fail' + e); }
11  }
12
13  bar({ g: 10 });
14  /*bp1*/ console.log('success');
15
16  setTimeout(foo, 10);
```

Figure 3: Reverse-Step (rs) *vs*. Reverse-Step Dynamic (rsd).

2.3 Callback Reverse to Origin

JavaScript and Node.js code use callbacks (also called promises or continuations) extensively. Debugging this code can be challenging as the control flow is difficult to reconstruct: code registered at one point in a program's execution is invoked by the event loop later, sometimes much later. When a developer is at a breakpoint in some callback code, they may not, therefore, know where or why this callback code was registered, what the program state was at that time, or why the callback was was eventually invoked.

To address this challenge, JARDIS provides its reverse to callback origin (`rcbo`) operation. This operation allows the developer to travel back from the currently executing callback to the point in time when the callback was registered. To illustrate this operation, consider the code in Figure 3. If the current breakpoint is at line 10, labeled `bp2`, then the `rcbo` operation reverse-executes to the `setTimeout` on line 16 where the callback to `foo` was registered.

3. IMPLEMENTATION

JARDIS is part of Microsoft's ChakraCore JavaScript engine [7] and a fork of the Node.js runtime environment [18]. The overall system design is a record-replay-snapshot architecture [2, 10, 16]. The snapshot algorithm is based on [2] with the record-replay work done in the JsRT hosting API layer [14] that Node.js uses to interface with the JavaScript runtime. The snapshot code and the record-replay code involves approximately 20 kloc in the ChakraCore codebase. We added less than 100 lines of code to the Node.js codebase. Source code is available online [7, 18].

As Figure 4 shows, a Node.js process consists of the ChakraCore JavaScript engine and the Node.js runtime hosting environment. The ChakraCore engine executes the application JavaScript code, while the Node host manages all interactions with the environment including, file and network IO, callback scheduling, and OS resources. The Node host interacts with the ChakraCore engine via the JsRT API, which allows the Node host to create/inspect objects, invoke JavaScript functions, and create JavaScript function wrappers that call back into native host code when invoked.

3.1 Record and Replay

To record the full range of events needed to replay the execution of the JavaScript component of the program, we added logging code to three subsystems in the Node/ChakraCore codebases:

1. The JsRT API implementations. For each API, we add code to record the kind of operation, the argument parameters, and the return value. At replay time, JARDIS uses them to

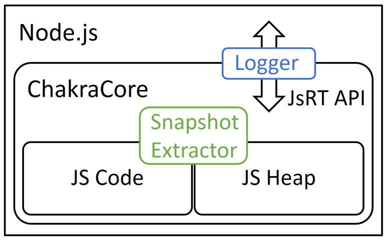


Figure 4: Overview of the Node.js/ChakraCore runtime and the architecture of JARDIS in the system.

re-execute the needed actions in the ChakraCore engine *e.g.* loading JavaScript code from files, calling JavaScript functions, or creating/modifying JavaScript objects.

2. Internal ChakraCore code paths that involve non-deterministic data, such as getting the current Date/Time, the invocation of external native code, or where enforcing deterministic execution was impractical. Property enumeration is an example where forcing a deterministic order is impractical.
3. Node host actions that bypass the JsRT API and directly modify `BYTE*` buffers for efficiency, as well as cases where the host has diagnostic information that ChakraCore cannot access, such as event loop status or callback dispatch dependencies (Section 3.3).

When invoked, the replay host `TTDebugHost` loads the specified log file, finds the first snapshot in the log, inflates the program state from this snapshot, and then begins replaying events from the log until hitting a breakpoint. Some of the events in the log correspond to external actions (event categories 1 and 3) and are directly replayed by the host. For example, we may have recorded that the Node runtime accessed a property of an object, created an `ArrayBuffer`, and then directly initialized the contents of the buffer with data from the network. To replay the original execution correctly, the replay host must explicitly simulate all of these actions. Other events in the log, event category 2, are replayed implicitly by the ChakraCore engine during its execution. During the original run, for example, the JavaScript `Date.now` function records its result into the log; at replay time, the ChakraCore engine simply reads this value from the log and returns it.

To allocate and record each event with a very low overhead, we use a bump-pointer allocator with fixed-size 40 byte entries and the occasional entries that need more space can allocate a second buffer. As JavaScript extensively dynamically loads and evaluates code, JARDIS's logger maintains a second log to record the URI origin of code, whether the code was loaded from a file, via eval, or `new Function`, and the code's source and associated internal sourceId information.

3.2 Snapshots

JARDIS takes snapshots of the program state at regular intervals. These snapshots allow a developer to jump between points in time in the recording, avoiding replay from the start of the log, and allow a developer to start a recording at any point during a program's execution. The snapshot extraction is based on Barr and Marron's work [2] with modifications needed to support unique features of the JavaScript object model, such as support for dynamically modifiable type layouts and the range of specialized representations for JavaScript builtin types.

Node.js is built around an event loop that dispatches JavaScript actions from internal worklists. This loop is an ideal point for performing snapshots. In it, the JavaScript execution stack is empty, reducing the size of the program state we need to capture and eliminating the need to snapshot call-frames and associated invocation data. Further, we can check if additional work is pending and delay taking a snapshot or, if the event loop is idling, preferentially take a snapshot early.

SnapShot :	{URI, Rnd, Roots, Code, Types, Vals, Objs}
URI :	URI of root .js file
Rnd :	Seed for PNRG — obviates recording each value
Roots :	[vId, ...]
Code :	[codeId, ...]
Types :	[{typeId, protoId, jsKind, flags, [SlotInfo ...]}, ...]
SlotInfo :	{pid, attrib}
Vals :	[{vId, jsKind, data}, ...]
Objs :	[{vId, typeId, Slots, ObjData ....}, ...]
Slots :	Array of void*, interpreted using tag bits
ObjData :	Pointer to data specific to a builtin type

Figure 5: Snapshot Representation.

Figure 5 shows the textual snapshot representation. The snapshot starts with standard information. The *URI* for the root code file and the value of the PNRG seed at snapshot time. The *Roots* are a list of *vId* values which are simply unique integer identifiers for each JavaScript root in the heap. Next is the *Code* list which contains *codeId* values that map to code load information in the log and indicates the code loaded into the program at this point.

The next set of components in the snapshot describe the layouts of the rest of the data in the snapshot. The *Types* component provides information on the associated prototype (*protoId*), the enum value (*jsKind*) indicates if this is a special builtin type and which one, *flags* for `frozen`/*etc.* info, and a list of *SlotInfo* entries. Each *SlotInfo* contains information on a single entry in a slot array that describes where each propertyId maps to in the object's slot array and an attribute value containing `enumerable`/*etc.* information.

Finally, we have the actual data from the program. The *Vals* list contains information for all the primitive values including strings, numbers, the special `null`/`undefined` values, *etc.* These values have the enum value *jsKind* and `void* data` that is interpreted as needed for each of the primitive *jsKind* tags. The objects refer to a *typeId* that gives us a *Type* entry that is used to interpret the *Slots* array and the *ObjData* pointer. The *Slots* array is an array of `void*` pointers which are interpreted via tag bits that indicate if the values are tagged integers (and can be cast directly) or pointer-based values that we treat as *vId* values (and can lookup in the snapshot to find the corresponding data).

3.3 Callback Dependence Tracking

To implement the reverse to callback origin (`rcbo`) operation, we require three pieces of information: 1) when callbacks are registered, 2) when they are dispatched, and 3) if they are canceled at some point. In the Node.js architecture, the runtime host manages this information in its internal event loops where the ChakraCore cannot directly inspect it. Thus, we modified the APIs that register callbacks to wrap and tag the registered callback function with the current index in the event log idx_r. When this wrapped function is dispatched from an event loop, we record the register index idx_r in the event log entry for the function invocation. When a developer uses the `rcbo` operation, we simply scan the event log to find the enclosing function invocation event log entry, get the idx_r value, and then time travel back to that point in the execution.

Table 1: Performance overheads when running JARDIS in record mode. *Process Size* is the baseline memory used by the Node.js process without JARDIS. The *Time* and *Space* overheads are the increase in runtime and memory seen when running the benchmark with JARDIS set to recording mode; the † symbol marks when the recording and baseline execution times were indistinguishable. *Log Size* is the size of the compressed (2–4 second) log when written to disk after the run completes.

Program	Code	Process Size	Time Overhead	Memory Overhead	Log Size
gbemu	11 kloc	45 MB	2 %	56 %	7.3 MB
navier	0.5 kloc	17 MB	1 %	71 %	6.3 MB
raytrace	1 kloc	60 MB	1 %	27 %	4.5 MB
dash	18 kloc	12 MB	†	83 %	5.2 MB
tasks	96 kloc	73 MB	2 %	74 %	8.9 MB
twitter	41 kloc	42 MB	†	55 %	7.8 MB

4. EVALUATION

We focus our evaluation on the performance overhead when running a Node.js application in record mode. Our benchmarks consist of the compute/memory intensive Octane [19] workloads gbemu, navier, and raytrace modified to run in Node.js as well as more traditional Node.js workloads of a code status dashdoard (dash) [9], a task tracking application (tasks) [21], and a twitter bot (twit).

We configured Node.js to take snapshots approximately every 2 seconds and keep the most recent 2 snapshots to retain approximately the last 2–4 seconds of execution. We setup all the benchmarks to run for 4–8 seconds to reduce measurement noise and ensure we fill the TTD log. We ran our experiments 10 times (with highest/lowest values discarded) on a Intel Sandy Bridge class Xeon processor at 3.6 GHz with 16 GB of RAM and a SSD drive.

Table 1 shows that, for both the compute intensive and the application based benchmarks, the recording overheads are exceptionally small. For all benchmarks, the runtime overheads are under 2%. For two of the application benchmarks, the runtimes for the baseline version of Node and JARDIS with recording were indistinguishable. For the application benchmarks, the Node worklist was frequently empty during execution allowing JARDIS to take snapshots during the idle period. Even in the compute bound benchmarks, where there are no idle periods, the snapshot cost is a very small part of the overall runtime. Despite creating 1000's of events in some benchmarks, the cost to record each action — a bump allocation and writing a few fields — is trivial compared to the work to dispatch and execute the action.

The memory overheads in Table 1 are larger, ranging from 27% to 83%, due to the fact that the log and uncompressed snapshots are kept in memory. Despite keeping up to 2 snapshots of the *entire* JavaScript program state in the log at a time, the memory overhead is less than 100% for all our benchmarks, indicating that the live object based snapshots are substantially smaller than what a naive process memory page snapshot would produce. The *Log Size* column shows that these snapshots and the log compress remarkably well, ranging from 4.5 MB to 8.9 MB.

5. RELATED WORK

Work on time-travel debugging has a long history and a wide variety of methods for implementing the functionality have been proposed. The most straight-forward mechanism for implementing a TTD system is to only trace events in the program and, usually, to replay from the start of execution based on the logged data. Current systems employing this approach include [3, 5, 8, 11]. An alternative to tracing the entire program execution is to intermittently take snapshots of the program state and then, on demand, restore, and replay as done in this work and others [2, 10, 16, 20, 22, 24].

Previous work has employed call-stack stitching [6] (or long stack traces [4]) to ease debugging asynchronous calls. This approach copies the call-stack when a callback is registered and appends it to the call-stack when the call-back is invoked. Unfortunately, copying a call-stack is expensive and can keep references alive. Further, the copies are also shallow so heap allocated values may be modified after the copy is made leaving the debugger information in an inconsistent state.

A number of research areas depend on, or would benefit from the availability of, a low cost method for obtaining the execution history for a program [1, 13, 15, 17]. The cost of producing the history, at the needed level of detail, is a limiting factor in the applicability of all these techniques. Thus, the ability to efficiently record snippets of a program's execution, and replay offline later while tracking exactly the required information, is an enabler for further work in these areas.

6. CONCLUSION

As shown, the open source JARDIS tool provides a rich set of functionality for recording, then inspecting and navigating the execution history of a program in a debugger. Additionally, JARDIS's extremely low recording overhead represents a major opportunity for research and tools that rely on collecting execution histories from in-production deployments. JARDIS represents a major advance in state of the art debugging tools and is an enabling technology for research areas including interrogative debugging, automated fault localization, and postmortem diagnostics.

7. ACKNOWLEDGEMENTS

We would like to thank Nikhil Khandelwal and Rob Paveza for their initial drive to start this project in Chakra. Akrosh Gandhi, Sandeep Agarwal, Arunesh Chandra, and Fiona Fung who provided extensive help working in the Chakra codebase and managing the project. Edgardo Zoppi and John Vilk both contributed during internships at Microsoft Research. Priyank Desai, James Lissiak, and Rob Lourens helped add prototype Time-Travel support to the F12 and Visual Studio Code debuggers.

8. REFERENCES

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 1993.

[2] E. T. Barr and M. Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *OOPSLA*, 2014.

[3] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006.

[4] Long stack traces for bluebird promises. http://bluebirdjs.com/docs/features.html#long-stack-traces.

[5] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *UIST*, 2013.

[6] Stack stitching in Chrome DevTools. http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/.

[7] ChakraCore JavaScript engine source code. https://github.com/Microsoft/ChakraCore.

[8] Chronon v3.10. http://chrononsystems.com.

[9] Code status dashboard application code. https://github.com/mrkmarron/WorkItemTrackerDemo.

[10] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. In *Parallel and Distributed Debugging*, 1988.

[11] GDB v7. http://www.gnu.org/software/gdb/news/reversible.html.

[12] Jardis demo video. https://youtu.be/EJqo3j9we0A.

[13] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE*, 2007.

[14] JsRT API documentation. https://msdn.microsoft.com/en-us/library/dn249673(v=vs.94).aspx.

[15] Y. P. Khoo, J. S. Foster, and M. Hicks. Expositor: Scriptable time-travel debugging with first-class traces. In *ICSE*, 2013.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.

[17] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *ICSE*, 2008.

[18] Node with ChakraCore source code. https://github.com/nodejs/node-chakracore.

[19] Octane 2.0 Benchmark. https://developers.google.com/octane/.

[20] Mozilla rr tool. http://rr-project.org/.

[21] Task tracking application code. https://github.com/dreamerslab/express-todo-example.

[22] UndoDB v3.5. http://undo-software.com.

[23] Visual Studio Code (Insiders Build). https://code.visualstudio.com/.

[24] P. R. Wilson and T. G. Moher. Demonic memories for process histories. In *PLDI*, 1989.