Design Document for JavaScript

Tentative Model for JavaScript Traces

$\langle e \in Event \rangle = \mathbb{Z}^* \cup \{e_g\}$

$\langle t \in Type \rangle ::= \mathbf{S} \mid \mathbf{M}\ i \mid \mathbf{W}\ i$

$\langle \tau \in Trace \rangle ::= \mathbf{newEvent}\ e\ t \mid \mathbf{begin}\ e \mid \mathbf{end}\ e \mid \mathbf{link}\ e_1\ e_2 \mid \mathbf{triggerOpAsync}\ op\ e_1\ e_2 \mid \mathbf{triggerOpSync}\ op\ e$

$\langle op \in Operation \rangle ::= \mathbf{open}\ p\ f \mid \mathbf{close}\ f \mid \mathbf{hpath}\ p\ m \mid ...$

$\langle m \in Eff \rangle ::= \mathbf{produce} \mid \mathbf{consume} \mid \mathbf{expunge}$

Figure 1: The Model for JavaScript Traces.

The model of JavaScript traces is shown in Figure 1. A trace entry can be one of the following constructs:

- **newEvent** t: This construct creates a new event of type t. Note that every event is uniquely described by a positive integer. Every type $t \in Type$ can be one of **S**, **M** i, and **W** i. The events of type **S** have the highest priority among any other event. On the other hand, the events of type **M** i have higher priority than those of type **W** i. Also, an event with type **M** i has higher priority than an event whose type is **M** j if $i < j$. Finally, events whose type is **W** i have higher priority than the events of the same type **W** i (# dimitro@theosotr: this sentence is not clear - please clarify. Is this about the same type – FIFO?). This model is flexible enough so that we can describe all the peculiarities of JavaScript regarding the scheduling of different events, i.e., promises, all timers (`setTimeout()`, `setImmediate()`, `nextTick()`), and asynchronous I/O.
- **link** e_1 e_2: This expression links event e_1 with e_2. This expression reveals the causal relation between two events, i.e., the event e_1 causes the creation of e_2.
- **triggerOpAsync** op e_1 e_2: This construct describes that the operation op is *asynchronously* executed inside the context of the event e_1 and it is associated with the event e_2. Note that an operation $op \in Operation$ can be one of the constructs described in the syntax of FSTrace (recall the Puppet paper).
- **triggerOpSync** op e: This constructs shows that the operation op is *synchronously* executed inside the event e.
- **begin** e: The execution of the event m begins.
- **end** e: The execution of the event m ends.

Given a sequence of traces $< \tau_1, \tau_2, \ldots >$ along with their semantics we can infer: (1) the happens-before relations between different events, and (2) conflicting operations, i.e., operations that access the same resource and at least one of them writes on it.

Example

Consider the following JavaScript program that examines a data race on the file `/foo`.

```
const fs = require('fs');

fs.writeFileSync("/foo", "data");

setImmediate(() => {
  fs.exists("/foo", (exists) => {
    if (exists) {
      fs.readFile("/foo", (err, data) => {
        if (err) {
          return;
        }
        console.log("Data: " + data);
      });
    }
  })
});

setTimeout(() => {
  fs.unlink("/foo", () => {});
});
```

Given the program above, we can produce the following traces.

[
1: **begin** e_g *//The event corresponding to the global object*
2: **triggerOpSync** (**open** `"/tmp"` **write** 3) e_g *//fs.writefilesync("/foo", "data")*
3: **triggerOpSync** (**close** 3) e_g *//fs.writeFileSync("/foo", "data")*
4: **newEvent** e_1 W_1 *//setImmediate(...)*
5: **link** e_g e_1
6: **newEvent** e_2 W_2 *//setTimeout(...)*
7: **link** e_g e_2
8: **end** e_g
9: **begin** e_1
10: **newEvent** e_3 W_3 *//fs.exists(...)*
11: **link** e_1 e_3
12: **triggerOpASync** (**hpath** `"/tmp"` **consume**) e_1 e_3
13: **end** e_1
14: **begin** e_2
15: **newEvent** e_4 W_3 *//fs.unlink(...)*
16: **link** e_2 e_4
17: **triggerOpASync** (**hpath** `"/tmp"` **expunge**) e_2 e_4
18: **end** e_2

19: **begin** e_3
20: **newEvent** e_5 W_3 *//fs.readFile(...)*
21: **link** e_3 e_5
22: **triggerOpASync** (**open** `"/tmp"` **read** 3) e_3 e_5
23: **triggerOpASync** (**close** 3) e_3 e_5
24: **end** e_4
]

From the analysis of the traces, we identify the following conflicting operations:

Trace 12 and Trace 17: The trace entry at line 12 consumes the path `/foo` while the trace at line 17 expunges the same file.

Trace 17 and Trace 22: The trace entry at line 17 expunges the file `/foo` while the trace at line 22 reads the contents of the file `/foo`.

We observe that the operation at line 11 takes place inside the event e_1, the trace 17 is executed in the context of the event e_2, and the trace 22 is executed as part of the event e_3. After interpreting that sequence of traces, we presume that there is no happens-before relation between the events e_1 and e_2, and e_2 and e_3. Therefore, a race detector would report those conflicts as possible data races.

Generating Traces

We need to investigate three different ways for generating JavaScript traces of a particular program, that is, static instrumentation of source code (JavaScript), dynamic instrumentation of binary code, and static instrumentation of native code.

Source Code Instrumentation

Node.js offers a built-in module named `async_hooks` [1] that provides user with hooks whenever an asynchronous resource (e.g., timer, promise, asynchronous I/O, etc.) is created, executed or destroyed. A unique ID is assigned to every asynchronous resource and `async_hooks` also provides the context in which every asynchronous resource is created. For example, in the program above, the `async_hooks` module tells us that the `setTimeout()` function is called at top-level code, while the `fs.exists()` resource is created inside the callback of the `setImmediate()` timer.

The implementation is straightforward: we simply need to produce the appropriate trace entries at every hook provided by the `async_hooks` module. Note that the `async_hooks` module does not keep track of synchronous calls (e.g., `writeFileSync()`). Also, `async_hooks` does not provide information about which I/O calls are triggered each time. Therefore, we could use `Jalangi` [2] (dynamic analysis framework for JavaScript) in parallel with `async_hooks` to track those calls or identify them through kernel monitoring through `strace`.

Pros:

- Easy to implement

Cons:

[1] https://nodejs.org/api/async_hooks.html
[2] https://github.com/Samsung/jalangi2

- It is JavaScript-dependent, it cannot be used in other domains.
- We need to use two different tools (e.g., `async_hooks` and `Jalangi`) to produce the appropriate traces.

Dynamic Binary Instrumentation

We could perform dynamic binary instrumentation through popular tools like Valgrind [3] or DynamoRIO [4]. The idea is to instrument every function entry and produce the corresponding traces. For example, if a call to `uv_fs_open()` is made (which is the `libuv` function for performing an asynchronous `open()` to a file), we could generate a **triggerOpAsync** (**open**) trace. Through dynamic instrumentation, we are able to inspect all function calls including calls from `V8`, `Node.js`, `libuv`, and `libc`. Therefore, we need to indentify all those native (C/C++) calls that are relevant to JavaScript's events and asynchronous I/O operations.

We could proceed as follows: we could create a Valgrind plugin [5] (similar to Callgrind) for keeping track of relevant function calls (along with their arguments). Alternatively, we could use the DynamoRIO API for wrapping and replacing functions [6].

Pros:

- This approach is generic and it can be used for tracing applications of other domains.
- There is no need (hopefully) to perfom any instrumentation to Node.js or JavaScript source code.
- No need to rebuild the framework.

Cons:

- Dynamic instrumentation imposes a significant overhead to the application.
- Harder to implement compared to the source code instrumentation.

Static Instrumentation

Through static instrumentation, we can instruct the compiler to call our own hooks before every function entry. For example, compiling a C/C++ program with the `-finstrument-functions` gives us two hooks for every function call: one before function entry and one after function exit. The signature of those hooks is the following:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

The arguments of those hooks correspond to the address of the function that is instrumented and the address of the call site. As a result, we do not have access to the arguments with which the function is called. One workaround is to inspect the stack in order to retrieve those arguments as described here. Similarly, we could create our own GCC or LLVM plugin to instrument the code as we want.

[3] http://valgrind.org/
[4] http://dynamorio.org/
[5] http://www.valgrind.org/docs/manual/writing-tools.html
[6] http://dynamorio.org/docs/group___drwrap.html

Pros:

- This approach is generic and it can be used for tracing applications of other domains.
- It does highly affect the performance of the application.

Cons:

- We have to re-compile the framework (i.e., Node.js) to add the instrumented code (e.g., by specifying the `-finstrument-functions`).
- Static instrumentation is not able to track the dependencies of a program. For example, we are not able to instrument a function of a shared library like `libc`.
- Harder to implement compared to the source code instrumentation.