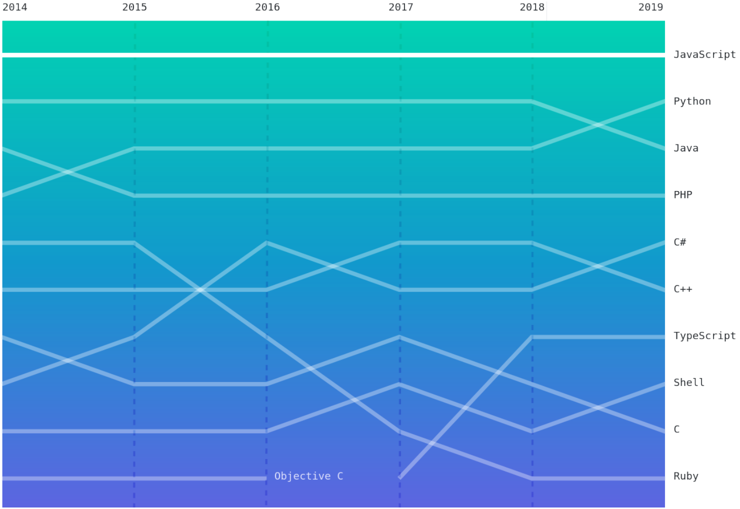
RT.js

Practical Real-Time Scheduling for Web Applications

Christian Dietrich, **Stefan Naumann**, Robin Thrift,
Daniel Lohmann

Leibniz University Hanover

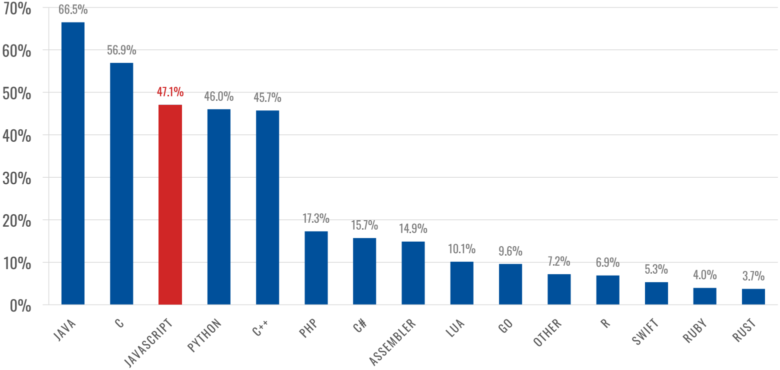
JavaScript: The Lingua Franca in the Webbrowser



Github. The State of the Octoverse, 2019

Overall Summary of Programming Languages

Which of the following programming languages, if any, do you use to build IoT solutions?

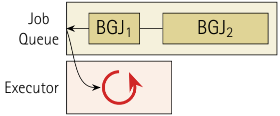


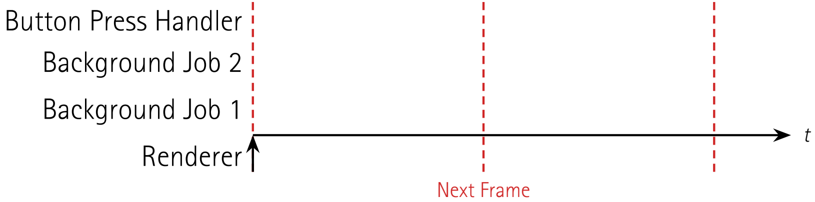
Eclipse IoT Working Group. IoT Developer Survey Results, 2018

IoT - Internet of Things

- on browser as frontend-language
- on Node.js as backend-language

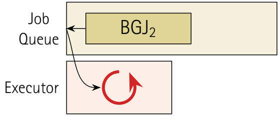
- single-threaded
- first-come-first-served
- run-to-completion

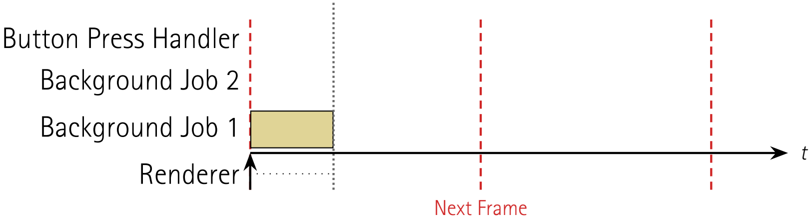




Scheduling in the JavaScript Executor

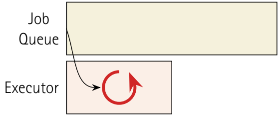
- single-threaded
- first-come-first-served
- run-to-completion

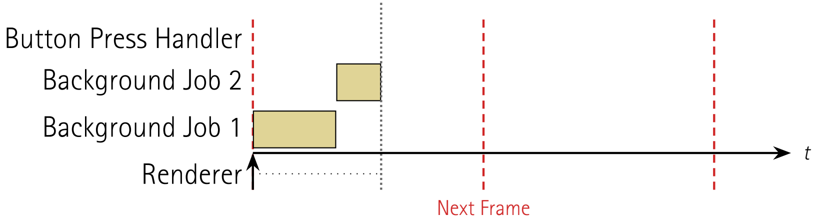




Scheduling in the JavaScript Executor

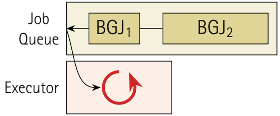
- single-threaded
- first-come-first-served
- run-to-completion

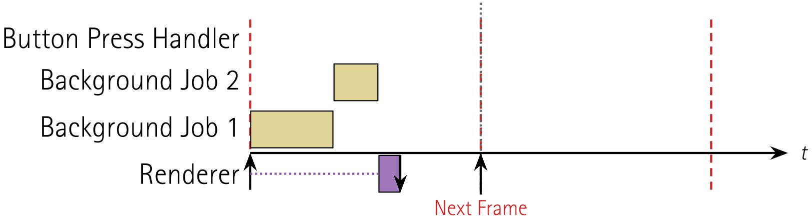




Scheduling in the JavaScript Executor

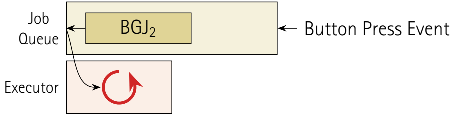
- single-threaded
- first-come-first-served
- run-to-completion

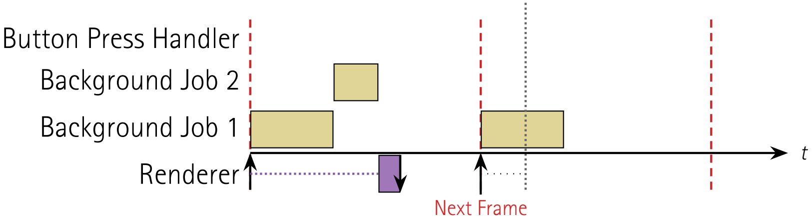




Scheduling in the JavaScript Executor

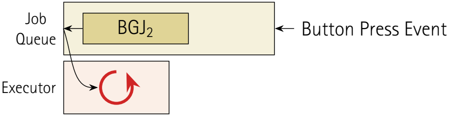
- single-threaded
- first-come-first-served
- run-to-completion

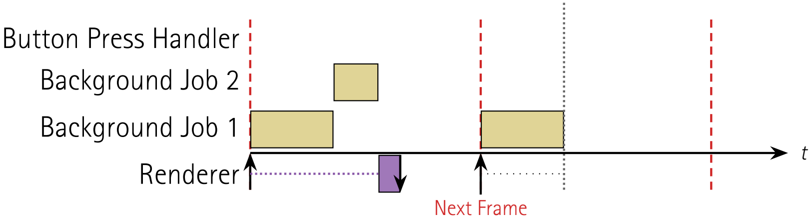




Scheduling in the JavaScript Executor

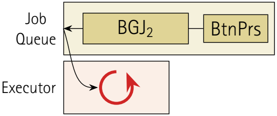
- single-threaded
- first-come-first-served
- run-to-completion

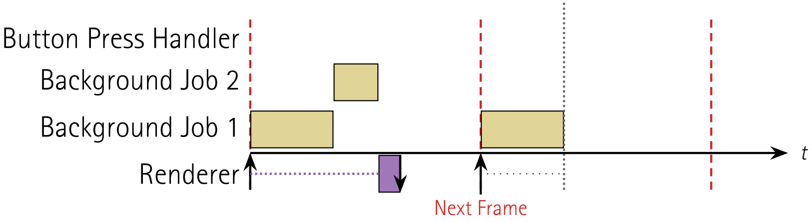




Scheduling in the JavaScript Executor

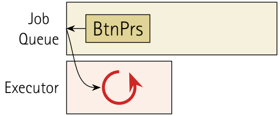
- single-threaded
- first-come-first-served
- run-to-completion

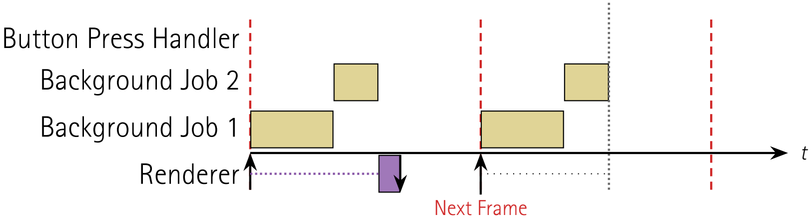




Scheduling in the JavaScript Executor

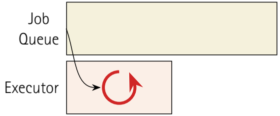
- single-threaded
- first-come-first-served
- run-to-completion

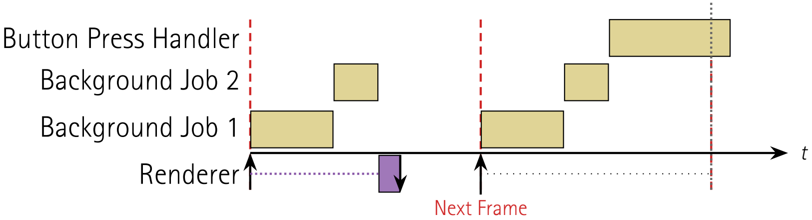




Scheduling in the JavaScript Executor

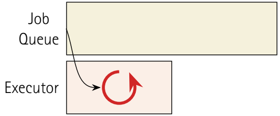
- single-threaded
- first-come-first-served
- run-to-completion

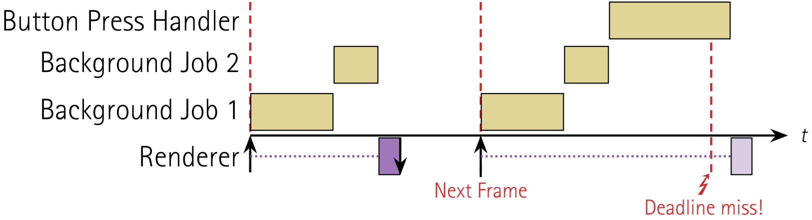




Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion





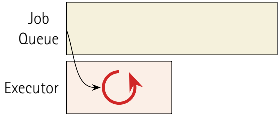
Problems in the JavaScript Execution Model

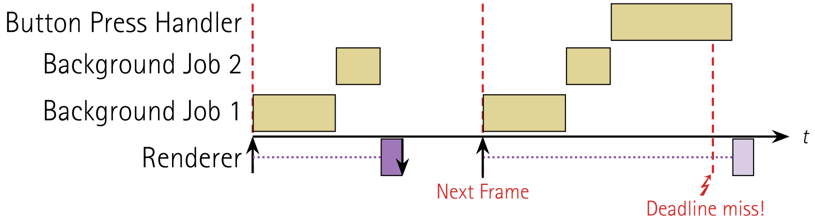
Problem 1: Long-running tasks block the executor

Long-running computations result in a reduced page-rendering frequency.

No Prioritization in the JavaScript Executor

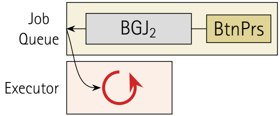
- single-threaded
- first-come-first-served
- run-to-completion
- no priorization possible

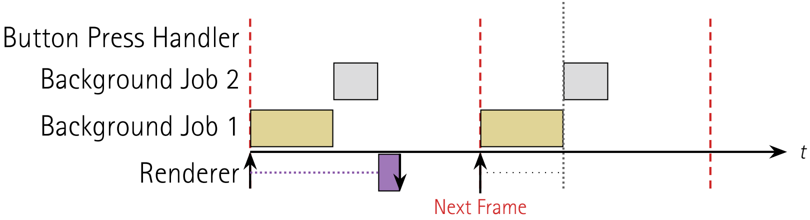




No Prioritization in the JavaScript Executor

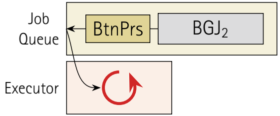
- single-threaded
- first-come-first-served
- run-to-completion
- no priorization possible

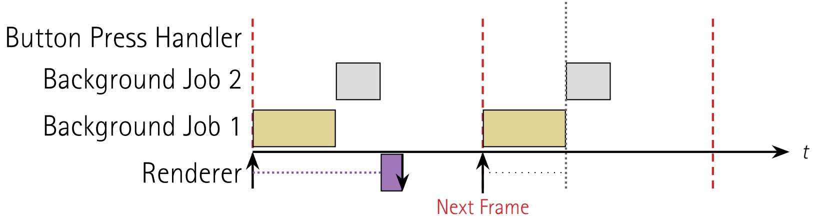




No Prioritization in the JavaScript Executor

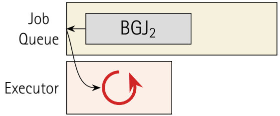
- single-threaded
- first-come-first-served
- run-to-completion
- no priorization possible

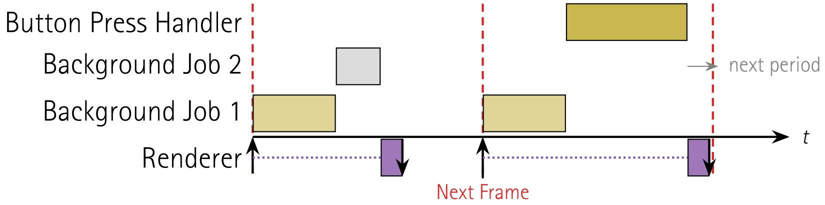




No Prioritization in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
- no priorization possible





Problems in the JavaScript Execution Model

Problem 1: Long-running tasks block the executor

Long-running computations result in a reduced page-rendering frequency.

Problem 2: No prioritization of tasks

An application cannot schedule it's jobs according to their relative importance.

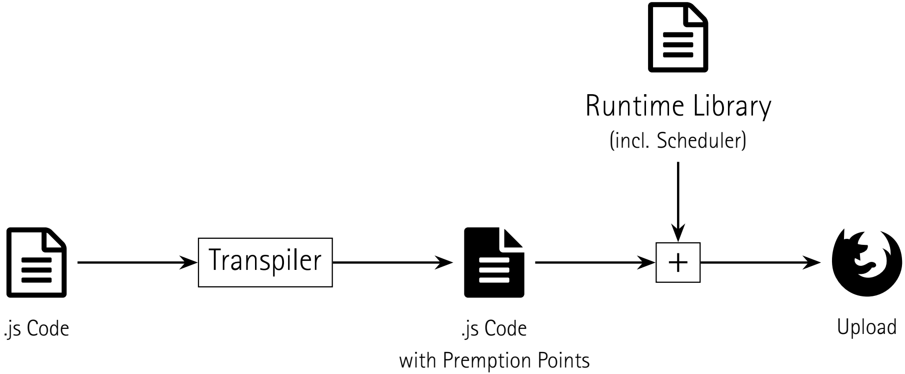
How would JavaScript solve it?

- asynchronous programming style discourages long running tasks
  - harder to read and maintain → *Callback Hell*[3]
  - not suitable in all cases

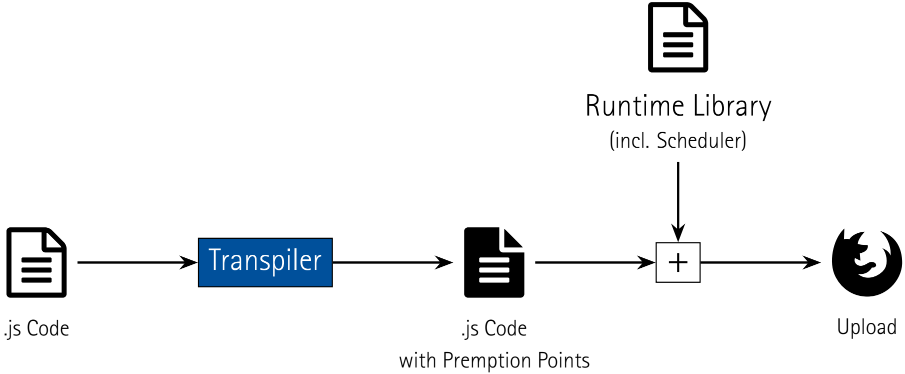
```
function cb () {
    console.log("Timer fired");
}

document.setTimeout ( cb, 4000 );
```

[3] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in javascript.
In *ESEM'15*, pages 1–10, Oct 2015

Proposing RT.js

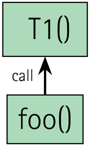


Proposing RT.js



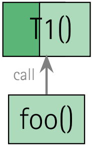
Generators as Preemptable Jobs

- Generator: Function, returns (`yield`) and keeps state
  - Compare: Iterator



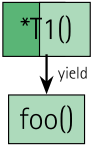
Generators as Preemptable Jobs

- Generator: Function, returns (`yield`) and keeps state
  - Compare: Iterator



Generators as Preemptable Jobs

- Generator: Function, returns (`yield`) and keeps state
  - Compare: Iterator



Generators as Preemptable Jobs

- Generator: Function, returns (`yield`) and keeps state
  - Compare: Iterator
- `yield` → Preemption Point

```
function T1 () {
    var i = 0;
    var result = 0;
    while ( i < 100000000 ) {
        // heavy computation
    }
    return result;
}
```

```
function *T1 () {
    var i = 0;
    var result = 0;
    yield
    while ( i < 100000000 ) {
        // heavy computation
        yield
    }
    return result;
}
```

Generators as Preemptable Jobs

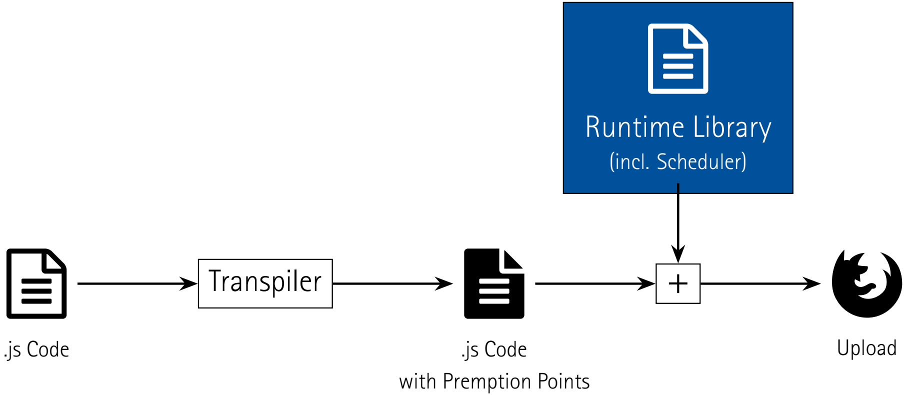
- Generator: Function, returns (`yield`) and keeps state
  - Compare: Iterator
- `yield` → Preemption Point

```
function T1 () {
    var i = 0;
    var result = 0;
    while ( i < 100000000 ) {
        // heavy computation
    }
    return result;
}
```

```
function *T1 () {
    var i = 0;
    var result = 0;
    yield
    while ( i < 100000000 ) {
        // heavy computation
        yield
    }
    return result;
}
```

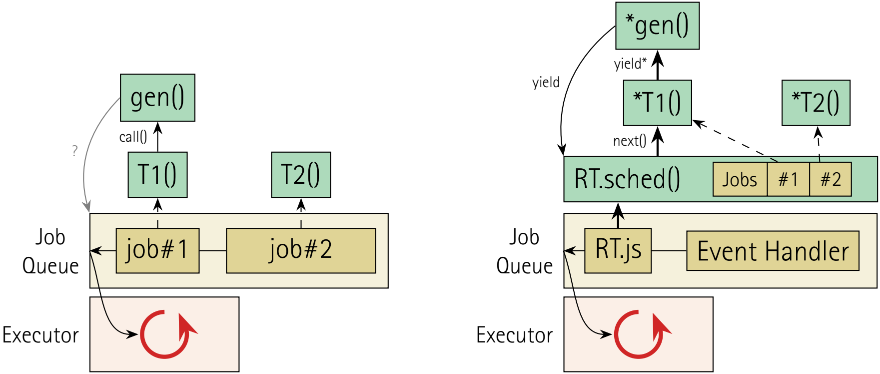
- **Automatically** add Preemption Points
- to decorated functions
  - in `for`, `while`, `do-while`-loops
  - before function calls

Proposing RT.js



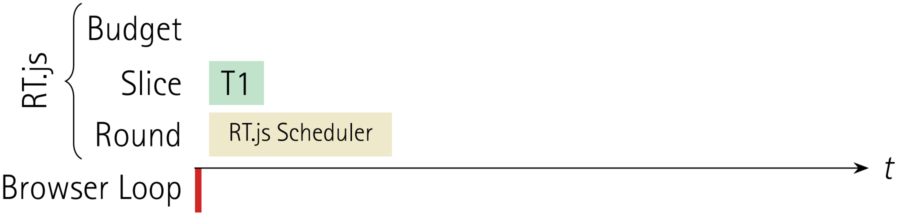
Scheduling Layer in JavaScript

- Manages Job queue itself, prioritizing



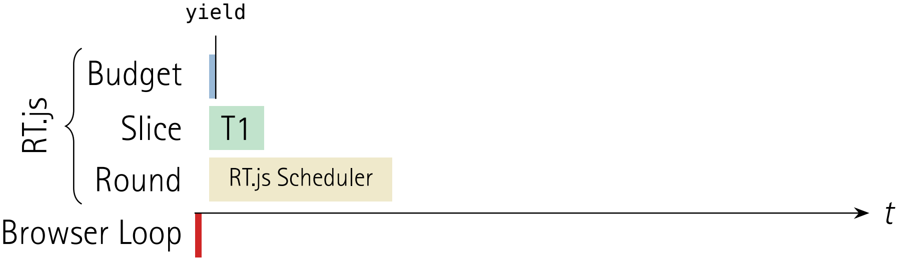
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



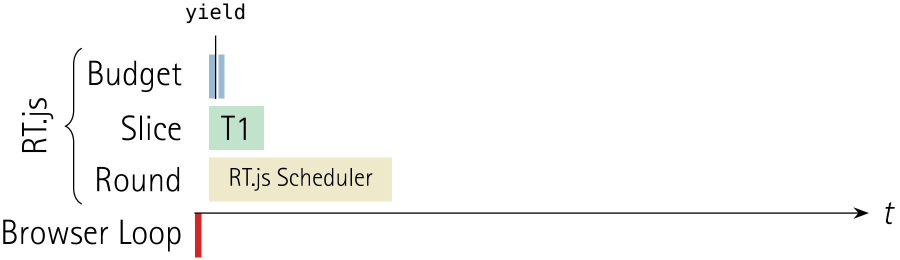
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



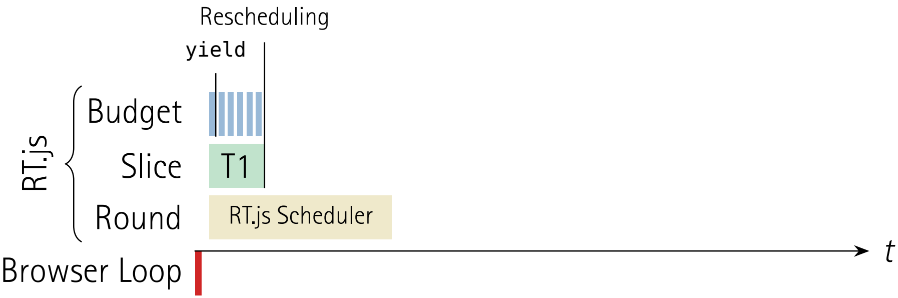
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



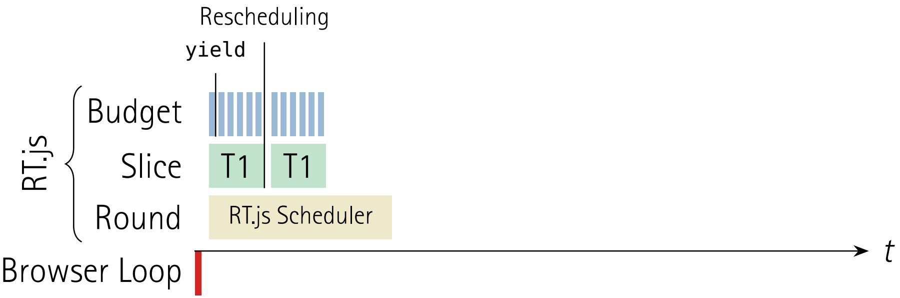
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



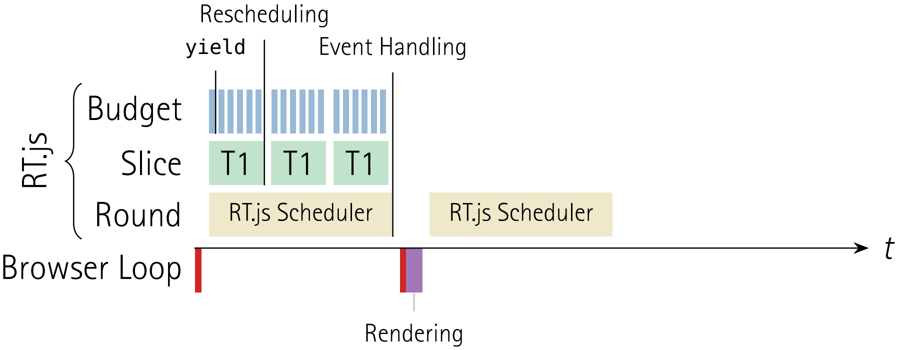
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



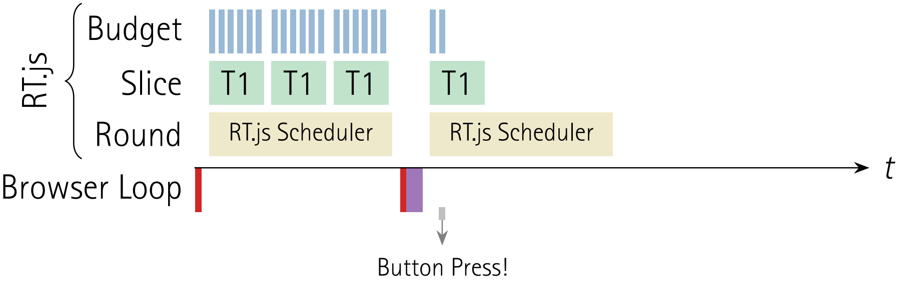
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



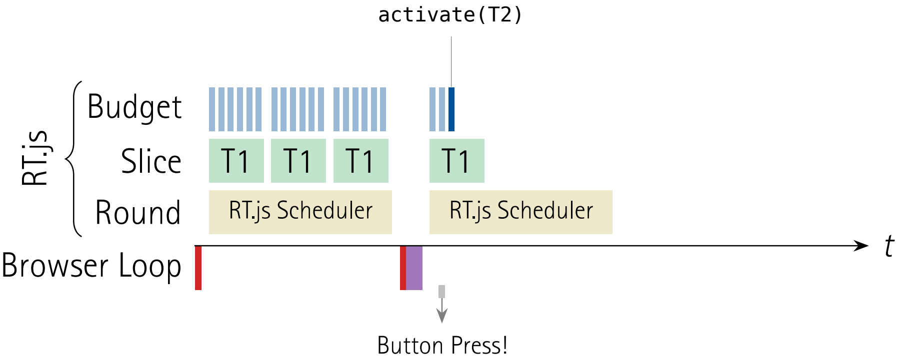
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



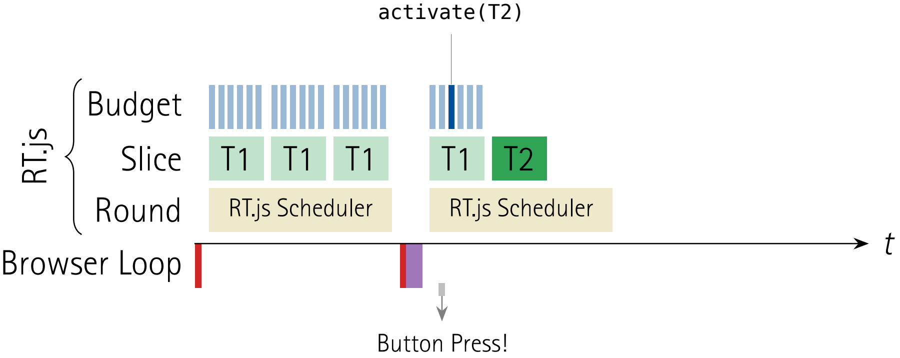
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



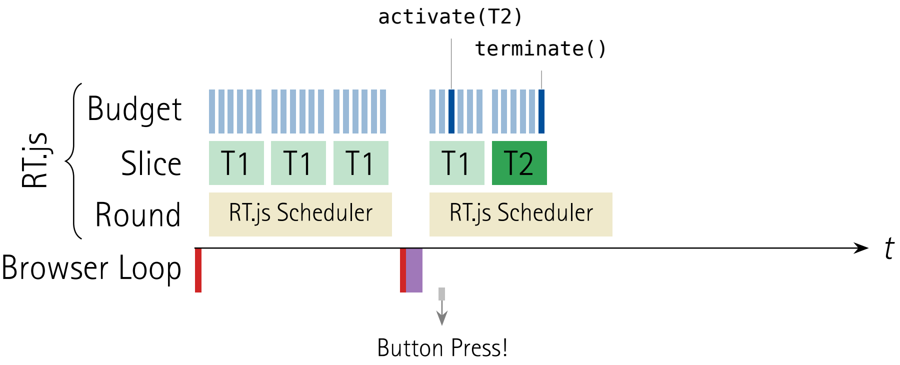
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



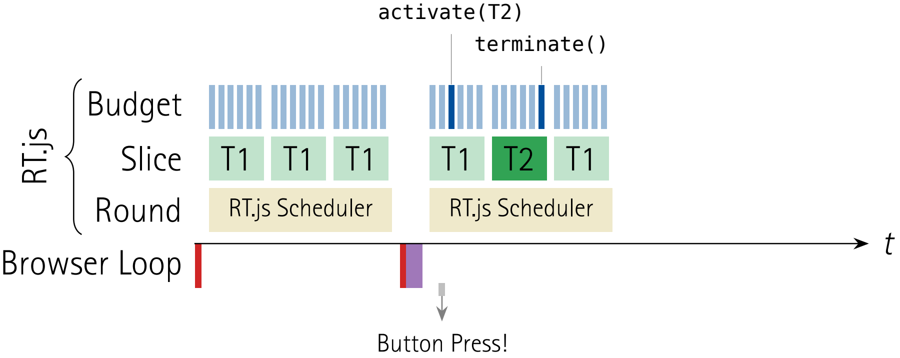
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



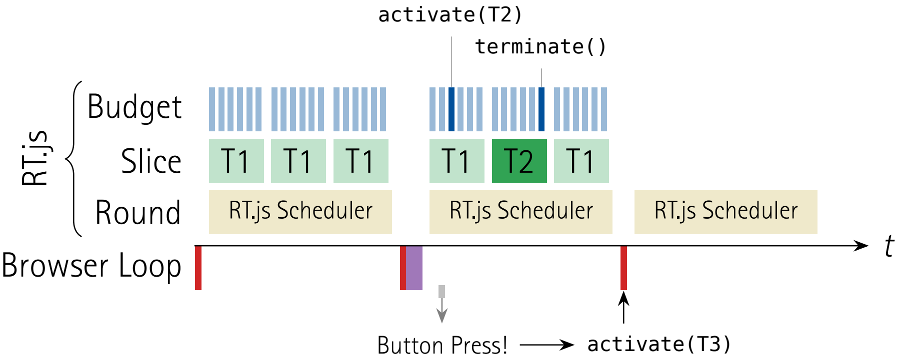
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



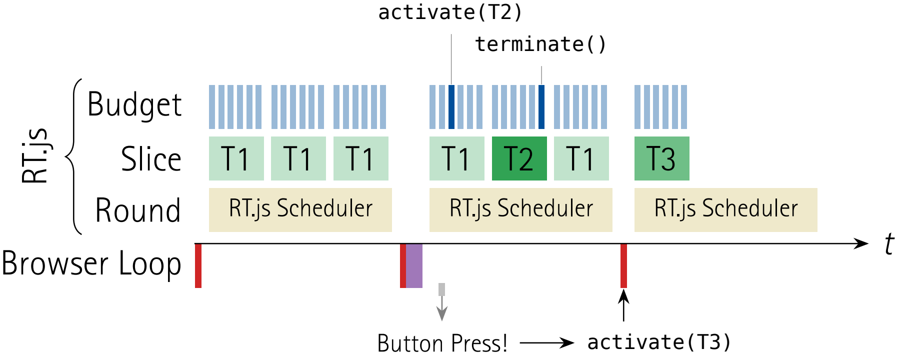
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



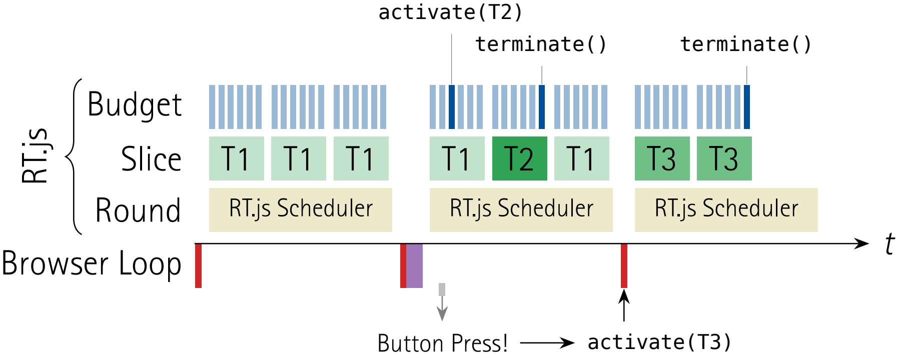
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



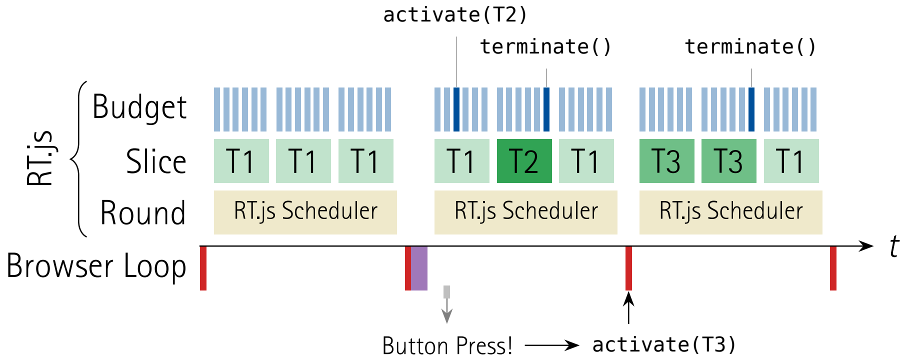
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$



Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$

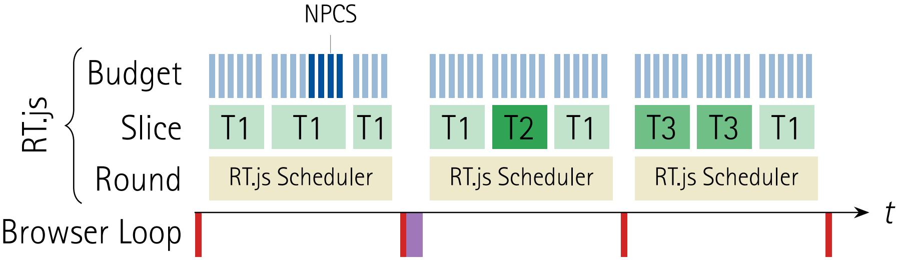


Synchronisation Guarantees

- single-threaded ✓
- non interleaving ✗
  - Synchronisation Primitives (Non Preemptive Critical Section)

Synchronisation Guarantees

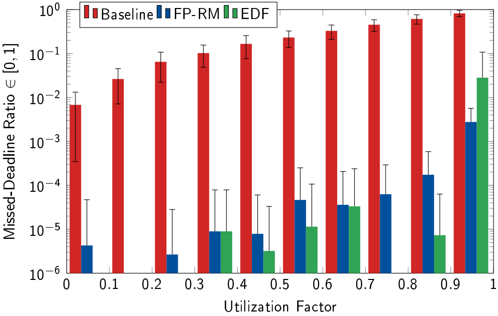
- single-threaded ✓
- non interleaving ✗
  - Synchronisation Primitives (Non Preemptive Critical Section)



Missed-Deadline Ratio of the Generated Task Sets

- 1,000 Generated task sets, with 15 tasks each
- Each jobs busy waits for its WCET
- Run on Node.js v8.10.0

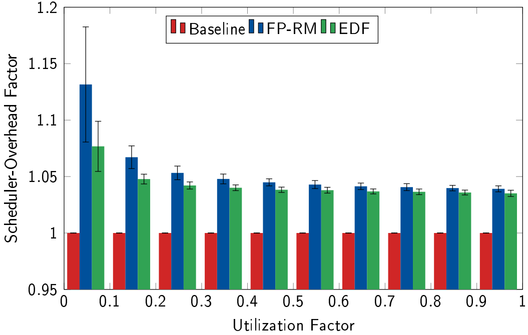
- FP-RM - Fixed Priority (Rate Monotonic)
- EDF - Earliest Deadline First



Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04

Scheduling Overhead

- Run on Node.js v8.10.0
- Scheduling overhead max +13.2 % (bucket average)
- Over all task sets, the median overhead for
  - Fixed Priority-Rate Monotonic (FP-RM) is 4.4 %
  - Earliest Deadline First (EDF) is 3.8 %



Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04

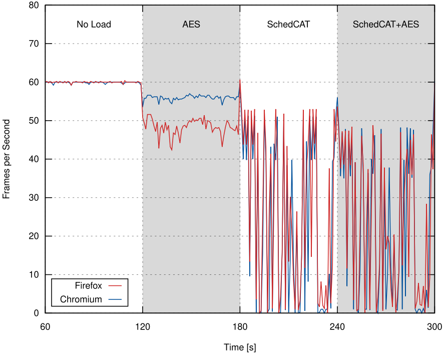
Makro Benchmark in the Browser

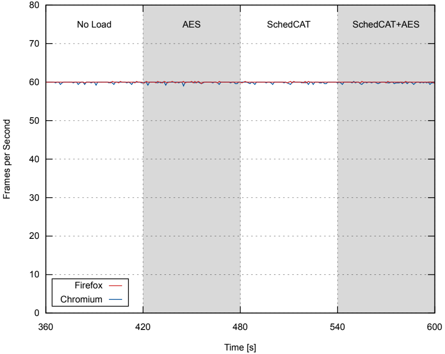
- A website, consisting of 4 components
  - Box Task, animates a box
  - Input Task, releases an AES-job
  - AES Task, encrypts input
  - A generated task set (u=0.75)

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04

Makro Benchmark

- Run on Firefox 67.0 and Chromium 73.0.3683.86





Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04

Conclusion

- RT.js allows prioritization of jobs
- Automatically introducing Preemption points
- The JavaScript engine was not modified
- Applicable to existing code and platforms

`github.com/luhsra/RT.js`

Microbenchmark: Overhead of yields

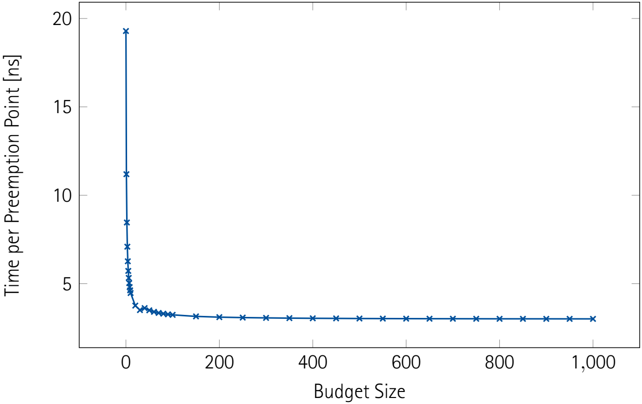
- Benchmarking the **yield**-statement on Node.js V8.10.0

```
function *genForLoop () {
    let counter = 0;

    for ( let i = 0; i < COUNT; i++ ) {
        yield;
        counter += 1;
    }

    return counter;
}

function *genFor2Loop () {
    let y = yield *genForLoop();
    return y;
}
```

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04

Microbenchmark: Overhead of `yields`

- Benchmarking the `yield`-statement on Node.js V8.10.0
- Upper bound for budget variable in tight loop at around 300



Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04

Microbenchmark: Overhead of `yields`

- Benchmarking the `yield`-statement on Node.js V8.10.0
- Upper bound for budget variable in tight loop at around 300

[ns]	Mean	Std. Dev.	Per Yield
Baseline	0.98	0.02	–
Generator (1 Level)	18.80	0.01	17.82
Generator (2 Level)	49.10	0.21	48.13
Generator (3 Level)	71.45	0.19	70.47

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04