RT.js: Practical Real-Time Scheduling for Web Applications

Christian Dietrich Stefan Naumann Robin Thrift Daniel Lohmann
Leibniz Universität Hannover
Hannover, Germany
{dietrich, naumann, lohmann}@sra.uni-hannover.de

***Abstract*—For billions of deployed browsers, JavaScript provides the platform-independent lingua franca that enabled the triumphal march of web-based applications. Originally intended for simple UI-event processing, JavaScript comes with an event-driven programming model, where event-callback functions are executed in strict sequential order. However, with applications getting more complex and tasks becoming more computation intensive, its first-come–first-served and run-to-completion semantic is hitting a limit, when reactions to user inputs are delayed beyond the human perception threshold. With the rise of the Internet of Things, this leads to friction-filled user experiences in everyday situations.**

With RT.js, we selectively introduce pseudo-preemption points into JavaScript functions and sequence the execution of event callbacks with well-known real-time scheduling policies, like EDF. Thereby, we provide a soft real-time abstraction that mitigates the described shortcomings of the JavaScript execution model without modifying the actual engine; making RT.js compatible with billions of devices. Applied to generated real-time task sets, we can almost eliminate the 30-percent deadline-miss ratios of baseline JavaScript at moderate costs. In a browser-based macro benchmark, we could diminish the influence of computation-intensive background tasks on the page-rendering performance.

***Index Terms*—Browser, JavaScript, Real-Time System, Preemption**

I. Introduction

JavaScript has evolved into the *lingua franca* of the internet for the development of cloud-supported, web-based applications. Popular examples include Office 365 or Skype, but there is also a myriad of stand-alone Electron apps that essentially ship their own browser to render and execute the application in a portable fashion on any modern operating system. This success continues into the *Internet of Things (IoT)* domain, where, according to a recent survey [1], JavaScript has now become the third-most employed language environment (after Java and C). It is typically employed in areas of "smart devices" that need to combine classic control tasks with a nifty user interface and cloud-based services, such as in the robotics and home automation domains.

The benefits of JavaScript for this domain are manifold, including platform independence, programmer compatibility, and a huge amount of reusable open-source libraries and frameworks, achieved by library versions of the browser ecosystem. The simple event-driven run-to-completion execution model is easy to grasp and integrate. All this reduces development costs and time to market.

However, originally intended for simple client-side UI-event processing, the JavaScript execution model hits its limits when applications get more complex and include computation-intensive tasks as well as tasks with (soft) real-time requirements. Due to the run-to-completion model, a task that does not finish "quick enough" does not only cause a sluggish user interface, but also reduces the responsiveness of control tasks of the smart device.

A. About this Paper

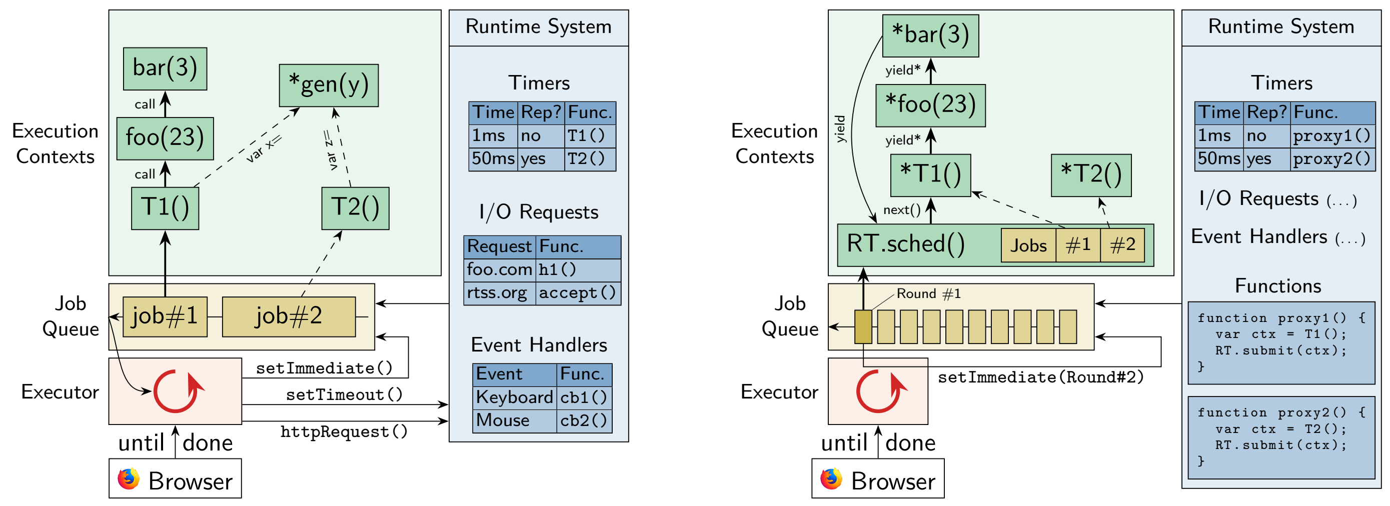
We propose RT.js, a framework that mitigates these issues by introducing preemption and priority-based scheduling into JavaScript. Combined with a real-time scheduling policy (such as EDF), the application can prioritize tasks based on deadlines and achieve (soft) real-time behavior. Our evaluation results show that for real-time workloads we thereby can reduce the deadline-miss ratio by an order of magnitude at a modest median run-time overhead of less than five percent.

We openly confess that the basic idea – to introduce preemptive scheduling in a previously only run-to-completion framework – is not new outside the JavaScript world. The transition from originally purely-event-driven-for-simplicity to multithreaded-for-reality on the longer term appears to be a recurring pattern in the rise of IoT and sensor net frameworks. Examples include Contiki [2], TinyOS [3] and, more recently, Arduino [4]. The main point of our approach is, however, its immediate and easy applicability to a large set of JavaScript applications by full backward compatibility. In particular, RT.js does not require any modifications to the employed JavaScript engine nor existing library code – it runs out of the box in every modern browser as well as the popular standalone Node.js engine. The key concept to achieve this goal is not to aim for full preemptability of the JavaScript engine, but instead exploit built-in JavaScript facilities to automatically transform JavaScript application and library code to become *pseudo preemptable*.

B. RT.js in a Nutshell

In a nutshell, RT.js introduces preemptable real-time scheduling into existing JavaScript code as follows:

This work was partly supported by the German Research Foundation (DFG) under grant no. LO 1719/3-1 and LO 1719/4-1.



Time	Rep?	Func.
1ms	no	T1()
50ms	yes	T2()

Request	Func.
foo.com	h1()
rtss.org	accept()

Event	Func.
Keyboard	cb1()
Mouse	cb2()

Time	Rep?	Func.
1ms	no	proxy1()
50ms	yes	proxy2()

```
function proxy1() {
  var ctx = T1();
  RT.submit(ctx);
}
```

```
function proxy2() {
  var ctx = T2();
  RT.submit(ctx);
}
```

(a) Baseline JavaScript

(b) JavaScript with RT.js

Fig. 1: JavaScript and RT.js Execution Models. Instead of registering and executing jobs directly, we submit the jobs, which were made generators, to RT.js for preemptive (yield) and prioritized execution; the JavaScript engine remains unchanged.

- Ahead of time, longer JavaScript calculations and functions are automatically sliced into subtasks by inserting `yield`-statements.
- Tasks are transformed into JavaScript *generator functions* – functions that can yield their execution job mid execution to be resumed later.
- The generator functions always yield to the RT.js scheduler, which selects the next (sub)task according to its priority or deadline and resumes this job.

Note that all parties yield the processor voluntarily, so technically the result is not *fully*-preemptive scheduling, but more *pseudo*-preemptive scheduling on the basis of interrupt-less cooperative scheduling. As the relinquishing `yield`-statements are forced into the code in a preprocessing step with reasonable frequency, cooperation becomes mandatory and no task can monopolize the CPU. The major benefit of this pseudo-preemptive scheduling is that it avoids interrupt synchronization issues and can be implemented within the existing JavaScript frameworks and engines – at a modest overhead of less than five percent. In particular, we claim the following contributions:

- We identify and quantify the problem of unprioritized and non-preemptable job processing that is inherent to the JavaScript execution model and grasp it in real-time terminology.
- With RT.js, we propose a method and implementation to introduce preemptability via AST transformations and allow for the prioritized execution of JavaScript callbacks with fixed-priority and EDF scheduling.
- We evaluate our prototypical implementation with micro benchmarks and a realistic macro benchmark, give guidelines for the fine-tuning of the scheduling policy, and demonstrate dramatically improved deadline-miss ratios and frame rates.

The rest of the paper is organized as follows. In Section II, we describe the JavaScript execution model and its conceptual problems. In Section III, we propose the RT.js approach and evaluate its impact in Section IV. In Section V, we discuss our results and give an overview about the related work in Section VI, before we close in Section VII.

II. JavaScript Execution Model

JavaScript was designed as a language for reacting to UI events, like keyboard presses, mouse clicks, or short-lived (periodic) timers, which result in changes to the *Document Object Model (DOM)* of the displayed webpage. These initial requirements heavily influenced the design of the language and resulted in its *event-driven* programming model, which lives off many short-running callback functions. Triggered by external or internal events, the JavaScript execution engine (see Figure 1a) executes callback jobs in a *first-come–first-served (FCFS)* manner in strict sequential order.

The lifetime of a JavaScript program starts when the browser finished the download of a code file and hands over control for the first time to the JavaScript engine. Executed in a fresh runtime instance, the program defines functions, issues I/O requests (e.g., additional HTTP requests), registers UI event handlers, and requests time-based function activations (e.g., `setTimeout()`). The runtime records the user-supplied callback functions, which will be invoked later on, for the registered events. When the program reaches the end of the file, control is handed back to the browser and only the global variables and the event–callback tables remain as the JavaScript-runtime state.

When an event occurs (e.g., a timer expires or key is pressed), the engine searches for a registered event handler, creates a job from the given callback function, and appends it to the *job queue*. For example, in Figure 1a, job #2 is the newest job and it was created as result of a periodic 50 ms timer. When the browser deems it appropriate, it hands over control to the JavaScript engine, which starts emptying the job queue by executing the oldest job first. The engine executes jobs in FCFS order in a run-to-completion fashion. These jobs perform calculations, define more functions, request other network resources, or register new event handlers. Furthermore, a job can directly re-insert another job into the queue with `setImmediate()`.

The execution of a job starts with the invocation of the callback function with an initial *execution context*, which holds its arguments (e.g., network response). On every function call, a new execution context is allocated and holds the function's arguments, the local variables, and a link to its calling context; we can think of execution contexts as heap-allocated stack frames. In Figure 1a, `bar(3)` is currently running in the context of job #1 and was invoked by `foo(23)`.

Unlike normal stack-allocated function-call frames, these execution contexts, which are independent from concrete jobs, allow for the usage of *generators*. This language feature resembles the continuation concept [5] from Lisp; it makes it possible to define functions that can return multiple times without losing their local state. On invocation, a generator function returns, instead of a result, an execution context, which can be stored in a variable (`x` in Figure 1a). We can continue this execution context (`x.next()`) and the generator's function body will proceed until it hits the next `yield` keyword. There, the engine records the next continuation point and hands back control to the caller of `next()`. With each yield, a return value and an indication of the generator's completion state gets passed to the caller.

Furthermore, JavaScript supports generator chaining with the `yield*` keyword. Thereby, a generator delegates control to a sub-generator for as long as that generator produces values and the parent generator only continues after the sub-generator has finished. Generator chaining is transparent for the caller of `next()` and the return-value sequence of the sub-generator is sliced into the parent's return sequence.

The JavaScript engine and its event-driven programming model were built from the ground up to be used asynchronously. This is necessary since the browser executes the JavaScript engine and the page rendering alternating in the same thread; a started job must complete before the renderer can regain control. Thereby, no synchronization is necessary between the renderer and JavaScript-induced DOM manipulations, or between two JavaScript jobs. However, if a job runs for too long, it stalls the whole rendering pipeline, and the responsiveness of the browser tab, or even the whole browser suffers. Therefore, we identify the first problem of JavaScript in the context of larger web applications:

Problem 1. *Long-running computations result in a reduced and jittering page-rendering frequency.*

Confronted with this problem, developers came up with an ad-hoc idea, which became a common practice [6]: The developer manually splits up the computation into multiple functions and chain their execution with `setImmediate()`. Since this practice harms the code readability and fosters the prevalent "callback-hell" [7] problem, ECMAScript 6 (ES6), which is the standard defining JavaScript, standardized not only generators but also asynchronous functions. Annotated with `async`, an asynchronous function does not directly return a value but only a promise object. At some later point, another `async` function can resolve the promise with `await` and extract the actual result. Technically, the invocation of an unresolved promise preempts the current execution context and enqueues a job that will calculate the promise result; afterwards, the preempted context is enqueued again with the promised result.

However, one problem remains, and gets even more relevant with `async`/`await`: in which order are jobs executed? While we described the JavaScript engine as having only one queue, real-world JavaScript implementations have multiple queues that are serviced by the *event loop* strictly in order [8]. This means that even if we directly reinsert a job into the queue with `setImmediate()`, the JavaScript engine will execute all pending timer callbacks and all pending I/O callbacks first, before our code gets executed. Furthermore, these queues are emptied in a FCFS fashion and the engine is agnostic to the importance of a given job. For example, it will execute the parsing of a large network response, without any mercy, until completion, even if our user eagerly waits for the response to her mouse click. Therefore, we identify the second problem of the JavaScript execution model:

Problem 2. *An application has no control over the execution order of jobs and cannot schedule them according to their relative importance.*

Taking a step back, we can interpret the browser ecosystem as a soft real-time environment: The periodic signal to render the page every, which triggers every 16.6 ms, and the human perception thresholds (< 100 ms feels instant) [9] introduce natural timing bounds for the response time. Therefore, we can grasp the JavaScript execution model in real-time terminology: The execution is already based on jobs that are scheduled non-preemptively in a (multi-level) FCFS order. Furthermore, we can see internal and external events as interrupt requests, where the event-handler tables are similar to interrupt-vector tables (see Figure 1a) and the registered callbacks are the interrupt-service routines. Each job runs with masked interrupts (no event detection) on a single processor (executor) until completion.

III. The RT.js Approach

With these problems in mind, we will apply two common techniques to JavaScript that are used by the systems community every day: preemption and scheduling of activities. With RT.js, we provide a real-time abstraction that provides soft–real-time properties, as far as possible in an interpreted language, and allow for the preemptive and prioritized execution of jobs with a bounded interrupt-detection latency. Thereby, we purely rely on standardized ES6 features (i.e., generators) and do *not* modify the execution engine itself. This makes RT.js – out of the box – compatible with all modern browsers and Node.js.

A. Programming Model

As an extension to the normal JavaScript programming model, we provide the possibility to selectively annotate long-running functions with `@preempt`. Executed with the RT.js machinery, these functions become preemptable at every function call and before each loop iteration in their own function body. Un-annotated library functions, even called from a `@preempt` function, execute in their regular fashion. With this controlled introduction of preemptability, we preserve most of the synchronicity guarantees of JavaScript since consecutive instructions (basic blocks), conditionals, and library calls still execute atomically. If further atomicity guarantees are required, RT.js also provides an API to erect further-reaching *non-preemptive critical sections (NPCSs)*.

In order to provide the necessary real-time parameters, like deadline and static priority, RT.js jobs are created from a JavaScript class that inherits from `Task` and provides a `@preempt` entry function, or created ad-hoc from a `@preempt` function and a parameter set. These jobs are submitted to the RT.js machinery, which schedules and executes them according to the chosen strategy. In our current implementation, we support *fixed-priority (FP)* and *earliest–deadline–first (EDF)* scheduling. In Figure 1b, we see how RT.js proxies the job execution: instead of waiting as callbacks in the JavaScript queue, a job resides in RT.js's (prioritized) queue and gets invoked by the scheduler.

Instead of registering her functionality as an event handler, the user registers a proxy function that submits an RT.js job. For example, in Figure 1b, `proxy2()` creates a T2 job every 50 ms and submits it to RT.js. If she wants to react to the return value of the submitted job, she can supply a callback function that gets invoked after the jobs have completed. During their execution, jobs can create and submit more jobs, erect critical sections, or install (periodic) timers. We also integrated RT.js with asynchronous functions: a `@preempt` function that resolves a promise with `await` is paused until the promise gets resolved.

After the developer has set up the event handlers and submitted the initial jobs, she starts the RT.js scheduler, which handles the execution of the supplied jobs and reinserts itself for another *round* of execution if work is to be done (see Figure 1b). While RT.js executes jobs, it regularly gives back control to the JavaScript engine to allow for page rendering and for the detection of new events. Furthermore, RT.js can be used alongside a normal event- and callback driven JavaScript program.

B. Preemption of Running Jobs

For our RT.js concept, it is essential that the scheduler is able to regain control from the running jobs before they finish their execution. However, unlike bare-metal hardware, the JavaScript engine provides no option to forcefully execute an OS function (e.g., via an interrupt) while a user-defined function is running. Therefore, we can never reach full preemptability if we want to leave the JavaScript engine unmodified; a highly desirable goal keeping in mind the billions of deployed

```
function counter() {
  let counter = 0, i = 0;
  for (; i < 1000000; i++) {
    counter += 1
  }
  return counter;
}
```

(a) Original Function

```
function* counter() {
  let counter = 0, i = 0;
  for (; i < 1000000; i++) {
    yield;
    counter += 1
  }
  return counter;
}
```

(b) ... with `yield`

Fig. 2: Preemption of JavaScript functions using generators.

JavaScript engines. Instead, we perform code transformation of the task's code (transpiling) ahead of time to include synchronous *preemption points (PPs)* at sensible points. For this, we mark the annotated functions as generators and insert regular `yield` statements, which results in the function giving up control cooperatively.

In Figure 2a, we see a minimal example of a long running function that will block the engine. When called, `counter()` increments its variables until the loop terminates and the function returns its result. In order to regain control from `counter()` while it is currently running, we mark it a generator (see Figure 2b, "*") and insert a `yield` at the beginning of the loop body. Thereby, `*counter()` gives back control to its caller on every loop iteration. While the generator is suspended, its execution context still holds the current counter value.

Since generators give back control to their direct invocation context, we have to treat function calls between annotated functions specially: Instead of a normal function call, we use generator chaining (i.e., `yield* counter()`) to hand over control from the parent to the child generator. For example, in Figure 1b, `T1()` is only continued after `foo(23)` and `bar(3)` reached the end of their function bodies. Thereby, we establish preemptability for all `@preempt` functions that are on an unbroken call chain of `@preempt` functions.

In order to achieve compatibility with existing JavaScript software, we also support the integration with asynchronous functions: Annotated functions can be `async` and are allowed to resolve a promise with `await`. In the transpilation process, we exchange the `async` with the generator annotation and replace every `await` keyword with a `yield` statement that hands the promise object to the scheduler. The scheduler removes the resolving job from the RT.js queue and unblocks it only after the promise got resolved.

RT.js also supports that a user manually inserts `yield` statements to introduce preemptability in a very controlled and explicit fashion. However, since for most applications, such a tedious and cautious control of preemptability is not required, we provide an automated tooling to introduce PPs. For this, we utilize a technique that is common practice in the JavaScript world (e.g., to translate between different language revisions): transpilation. In a source-to-source transformation, which happens ahead-of-time before the deployment, the *abstract syntax tree (AST)* of the program gets transformed according to predefined rules (see Table I). For our implementation, we used

Pattern	Replacement
`for(...) {...}`	`for(...) {`**`PP;`**` ...}`
`while(...) {...}`	`while(...) {`**`PP;`**` ...}`
`do {...} while(...);`	`do {`**`PP;`**` ...} while(...);`
`fn(...);`	**`PP;`**` fn(...);`
PP *(alternatives)*	`yield`
	`if (--budget == 0) yield`
`preemptFn(...);`	**`yield*`**` preemptFn(...);`
`await asyncFn(...);`	**`yield`**` asyncFn(...);`

TABLE I: Rules to introduce preemption points (PP)

the TypeScript[1] compiler [10], which provides an interface for user-defined source-to-source transformations.

In the transpilation process, TypeScript searches for all functions with `@preempt` annotation and makes them generators. Furthermore, it searches the function-body AST recursively for predefined patterns and introduces PPs (see Table I). The general rule is that PPs are added at the beginning of loop bodies and before function calls (to catch recursive loops). Thereby, we ensure that a job cannot monopolize the executor in an annotated function. In this step, we also handle calls to annotated (`preemptFn()`) and asynchronous (`asyncFn()`) functions.

Since PPs are hit frequently during the program execution and each `yield` statement comes with an overhead, there is a trade-off between responsiveness and induced system overheads. The more functions, from the task entry downwards, are marked with `@preempt` and the more PPs we introduce, the more frequent the scheduler regains control of the executor. However, since the hit frequency of a PP is unknown ahead of time, we cannot simply leave out every second PP without giving in to potential executor monopolization.

Instead, we introduce an integer-typed global budget variable that the scheduler fills with a constant before it hands control to a job. The budget is comparable to a thread quantum in systems like Windows or Linux in that it represents a (logical) time interval, in which the running thread does not need to relinquish the CPU. We alter the inserted code at the PPs, such that the budget variable is decremented on every visit and `yield` is only called if the budget hits zero. With the budget, we control the number of dynamically-executed `yield` statements, and can, thereby, adjust the trade-off between responsiveness and preemption overheads. In Section IV-C, we will investigate experimentally on an upper limit for the budget.

C. The RT.js Scheduler

With the transpiled task bodies, RT.js is able to hand control to a job for a budgeted number of PPs and it will regain the executor afterwards. However, we still have to schedule between different jobs and integrate the RT.js scheduler with the JavaScript execution model. For the scheduling, RT.js currently supports the fixed-priority/rate-monotonic and the (unicore) earliest-deadline–first scheduling policies [11].

[1]TypeScript is a JavaScript super set that supports optional typing.

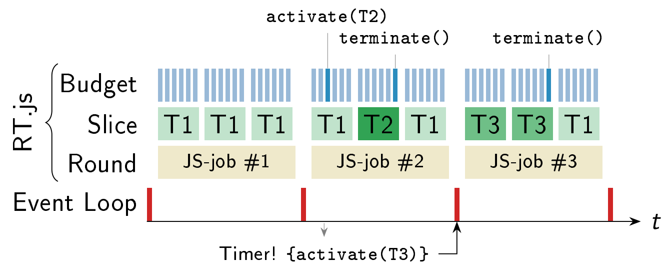


Fig. 3: Scheduling Scheme. T1 and T2 have the same priority, which is lower than T3's priority.

Thereby, especially the EDF scheme is well suited for use in the browser as the human-perception thresholds provide us with scenario-specific time bounds, where periods or minimal interarrival times are hard to give.

However, we still have to integrate the scheduler loop into the execution model (see Figure 3). For this, we chose a three-level execution schema that consists of *budgets* (PP-count bound), *slices* (time bound), and *rounds* (time bound). On the finest level, we continue the execution of a job for a job-specific *budget* of PPs. We execute multiple budgets in sequence, checking the current wall clock time after each budget, until the *slice* duration has expired. After each slice, we invoke the scheduler to switch between jobs with the same priority in a round-robin fashion. For example, in Figure 3, T1 and T2 have the same priority and we only switch to T2 after the first slice has expired. Multiple slices make up one *round*, which is a normal JavaScript job that checks, after each slice, if the round duration has expired. On expiry, we reinsert another round with `setImmediate()` into the JavaScript job queue if there is still a pending job and return control to the event loop. Thereby, we poll for events after each round and the round duration becomes the detection latency.

D. Further System Abstractions

a) Synchronization: Since RT.js adds concurrency to the application and the JavaScript consistency is weakened (see Section III-A), we have to provide additional synchronization mechanisms between jobs. Since we still promise that basic blocks and branches are executed without preemption, Boolean flags can provide some kind of protection. However, we also provide an API to establish NPCSs, where events are still detected, but no rescheduling is done between slices. In the future, we plan to incorporate more complex synchronization protocols, like the priority-ceiling [12] or the stacked-resource protocol [13], that interact with the scheduler and guarantee freedom from deadlocks.

b) Alarms: JavaScript-native timers (`setTimeout()`) have two essential drawbacks: (1) While the engine is active, timer activations, like all other events, are delayed until the current job has finished. (2) The browser only guarantees a minimum but no maximum latency before the callback is executed. Therefore, we provide an *alarm* API that eases both problems and provides more precise (periodic and non-periodic) timers.

For the implementation, we keep an ordered list of pending alarms with their absolute activation time. After each slice, we check the first alarm for expiration and call, directly from the scheduler loop, a user-defined callback that can, for example, submit a job. If the scheduler is currently not active, we fall back to the normal JavaScript timer mechanism to activate the callback function. Thereby, alarms can trigger while RT.js jobs are executing and the timer latency is bounded by the slice duration if RT.js is currently active.

IV. Evaluation

In the evaluation, we want to investigate the presented preemption mechanism and scheduling policy on three level of detail. First, we measure the cost of a single voluntary preemption via `yield` and look into the influence of calling `yield` only on every Nth preemption point. Second, we use periodic task sets from the real-time domain to inquire the benefits of scheduling JavaScript jobs instead of using the default FCFS policy. At last, we perform a macro benchmark in a web browser to show the improved responsiveness in a realistic usage scenario with simulated user input.

A. Evaluation Setup

We performed all benchmarks on an Intel Core i5-6400 CPU with 2.70 GHz and 32 GiB of main memory that runs Ubuntu 18.04. For the preemption-overhead measurements and for the execution of the generated task sets, we used Node.js v8.10.0, which is a standalone JavaScript runtime environment. For the macro benchmark, we used the Google Chromium 73.0.3683.86 and Mozilla Firefox 67.0 browsers. Since both Node.js and Google Chromium use the V8 JavaScript engine, we expect that measurements and qualitative conclusions are transferable between both environments. Mozilla Firefox uses the SpiderMonkey JavaScript engine.

B. Preemption Overhead

First, we want to quantify on the lowest level, how much overhead we introduce into a program if we insert preemption points with generators and by calling `yield`. This overhead measurement is comparable to measuring the dispatching overhead in a bare-metal RTOS or to the overheads of switching user-level threads. We explicitly measure only the preemption overheads and execute everything in this section *without* the RT.js scheduler.

We perform a micro benchmark on Node.js, where the workload function runs a tight counter loop (see Figure 2) for 262144 ($256 \cdot 1024$) iterations. For the baseline variant, we measure the runtime for executing the workload and divide it by the number of loop iterations. We executed the workload $n = 10000$ times and show the (arithmetic) mean of the run times and the standard deviation in Table II. For one loop iteration, Node.js requires about 1 ns.

For quantifying the `yield` overheads, we use the generator version of the counter loop (see Figure 2b). We resume the

[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

TABLE II: Microbenchmark of Javascript Generators. Baseline is a tight counter loop ($i \leq 256 \cdot 1024$) that was executed $n = 10000$ times. For the generator variants, a n-level deep generator-function call chain executes the tight loop and calls `yield` after each iteration.

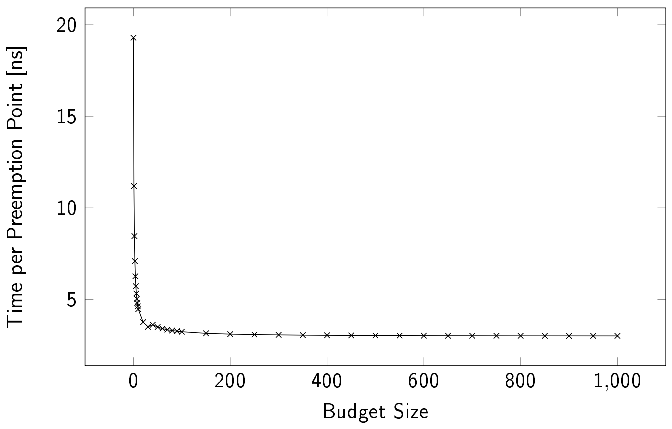


Fig. 4: Run-Time Overhead per Preemption Point. With a budget, the introduced preemption points invoke `yield` only on every Nth time. (n=10000, tight counter loop $i < 256 \cdot 1024$)

generator execution context in tight loop until it finishes and record the required time. Thereby, we measure, besides the counter calculation, one yield and one return-from-yield operation for each loop iteration. Furthermore, we want to quantify if the costs of a preemption increase if the workload nests invocation of generators with `yield*`. For this, we execute the loop directly in the called generator (1 Level), in a generator that is invoked with `yield*` from a generator (2 Levels), or if the loop is located on the third generator level (3 Levels). Thereby, we measure if the cost of PPs changes deep down the calling hierarchy.

From the results (Table II), we see that a preemption in the first generator level takes about 18 ns (per `yield`) if we subtract the nanosecond for the minimal workload. If we deepen the generator level, the preemption overhead grows by 20-30 ns per call-hierarchy level. Here, the increase indicates that the runtime iterates over the whole `yield*` chain instead of jumping directly to the invoker of `next()`.

As we have already discussed in Section III-B, we introduce a budget of PPs that a job can visit before it actually calls `yield`. Of course, this budget value is specific for each job and depends on the amount of code a job executes between two PPs. However, we want to find an upper bound for the budget where the overheads per PPs no longer decrease but only the scheduling latency grows. For this, we run the same tight counter loop (generator, 1 level), which is the worst-case scenario for the overhead of our pseudo-preemptive approach, with a varying budget size and show the results in Figure 4.

With the PP budget, we introduce additional computation, namely the decrease of the budget and the budget-exhaustion check, at each preemption point, while we reduce the `yield` frequency. The results show that the overhead drops rapidly with an increasing budget but flattens out if the budget exceeds 300. There, the budget check dominates the `yield` overhead and, on average, one preemption point costs around 3 ns. Therefore, a budget size of 300 provides the lowest possible cost per PP and we will use it as the default value for our evaluation.

C. Generated Task Sets

For an integrated view on the influence of RT.js, we perform a second evaluation with generated real-time task sets with varying utilization. While real-time task sets will not perfectly match the actual workload that arises in a web browser, they can still provide significant insights into the properties of the JavaScript engine itself, as well as the improvements that we get by using RT.js. As task model for this evaluation we use a set of independent and periodically activated tasks, whose deadlines are equal to their periods.

For the task-set generation, we used SchedCAT[2], which uses the Roger Stafford's randfixedsum algorithm [14] to produce task sets with a given utilization. We generated 1000 task sets, which consist of 15 tasks with a period of $[15\,ms, 5000\,ms]$ and a logarithmic-uniform distribution. We varied the utilization between 0.05 and 1.00 in increments of 0.05, such that 50 task sets were generated for each utilization step.

From the task-set description, we generate a baseline system, where released jobs are directly executed by the JavaScript engine in FCFS order. Each task executes (blocking) for its WCET and then hand back control to the JavaScript engine for handling further events. We run each system for 10 seconds and control the task activation explicitly: Before the run time, the benchmark generates a list of events that have to happen during the 10-second duration. From this event list, we calculate the absolute deadline for each job and use `setTimeout()` to release at the given time. With this event-list precalculation, we ensure that blocking the JavaScript runtime has no influence on release times and absolute deadlines.

For the RT.js variants, we make the task function preemptable by the described transpilation and execute the jobs with RT.js either with rate-monotonic/fixed-priority scheduling (FP-RM) or with earliest-deadline-first (EDF) scheduling. Task priority and absolute deadlines are derived from the period, which is also the relative deadline. The RT.js scheduler is configured to use a budget size of 300, a slice length of 1 ms, and a round length of 5 ms. For the RT.js variant, we use the supplied alarm mechanism (see Section III-D) to handle the event list.

[2]https://github.com/brandenburg/schedcat

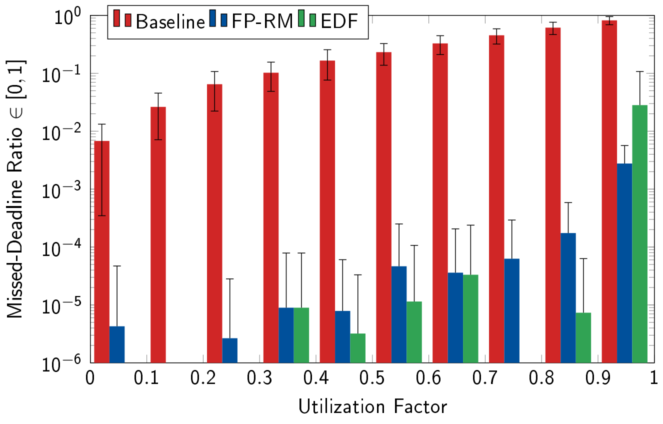


Fig. 5: Deadline-Miss Ratio for Generated Task Sets (n=1000). The task sets are binned according to their utilization (i.e., first bin is $[0, 0.10)$), and the bar (*logarithmic!*) indicates the mean (and standard deviation) over the ratio between released jobs and missed deadlines. The task sets were executed for 10 seconds.

We execute the three variants for each generated benchmark for 10 seconds, which also includes the $t = 0$ instance where all periodic tasks are activated at once. As the longest period is 5 seconds, each task is activated at least twice during the benchmark. For each job, we record the planned release time, the actual detection time, the job's actual computation time, the completion time, and the absolute deadline, which is derived from the planned release time. After the benchmark run, we count all missed deadlines and the overhead introduced by RT.js.

In Figure 5, we see a statistic of the ratio between released jobs and missed deadlines for the three variants. The systems are binned according to their planned utilization factor into 10 buckets (width is 0.10) and we use the average over the miss ratios for each system. As we can see, the miss ratio for the baseline system scales about linearly with the increasing utilization rate, while both RT.js variants perform better by (at least) one order of magnitude. Over the whole utilization range, the baseline system misses 29.6 % of all deadlines, while FP-RM misses 0.045 % and EDF misses 1.1 %. Although EDF theoretical promises to execute these task sets without deadline misses, the timing uncertainty of the engine and the overheads of RT.js still yield some deadline misses.

In Figure 6, we look at the run-time overhead for the scheduling and compare the job's computation time with the time that was spent in scheduler rounds. For the baseline variant, these numbers are, of course, equal as no overheads have to be paid. For the RT.js variants, we see that the overheads are highest for task sets with a small utilization but stay below +13.2 % for a bucket. Of course, for task sets with a small utilization, the overheads have a percentual higher value. Over all task sets, the median overhead for FP-RM is 4.4 % and 3.8 for EDF.

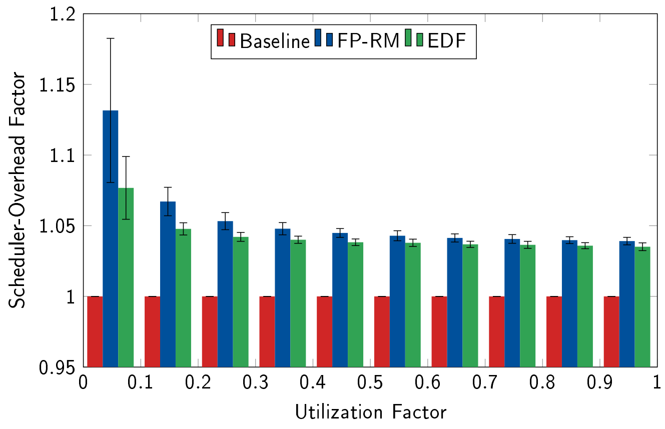
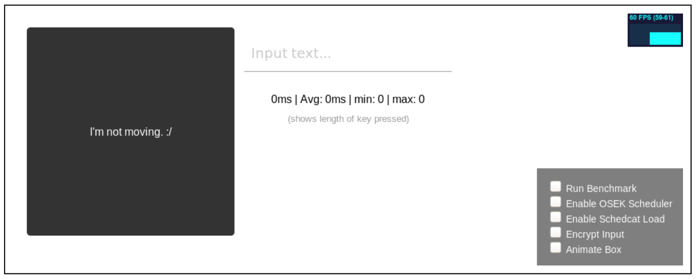


Fig. 6: Run-Time Overhead for Scheduling. With the same binning as in Figure 5, we show the average (and standard deviation) over run-time overheads (scheduler time/task time) of scheduling the task set instead of executing them in FCFS order.



```
0ms | Avg: 0ms | min: 0 | max: 0
```

Fig. 7: Screenshot of the Macro Benchmark

D. Macro Benchmark

Lastly, for quantifying the impact of RT.js on real-world web applications, we designed a benchmark scenario that resembles the requirements regarding reactivity under a high computation load. The benchmark consists of an HTML page with a moving gray box, a text input, a selection of options, a *frames-per-second (FPS)* graph (see Figure 7), and four JavaScript components that compete for computation time.[3] The scenario that we are mimicking is a user who inputs data into an animated web application, which runs different periodic and sporadic computation tasks. The goal of this benchmark is to measure the influence of the additional payload on the frame rate, and, therefore, on the user's experience.

The workload consists of two mandatory and two optional components:

1) The *box task* is released periodically via `requestAnimationFrame()` ($p = 16.6\,\text{ms}$, $d = 10\,\text{ms}$) and only moves the box between the left and the right border. Since its jobs complete fast, they never deplete their budget and, therefore, never call `yield`.
2) The *input task* ($d = 32\,\text{ms}$, sporadic) is triggered on text input (at the key-up event), records the input on the web page, and releases an AES job (if enabled).
3) The *AES task* ($d = 500\,\text{ms}$, sporadic) concatenates the input text 16 times, encrypts it 350 times, and writes the results to the JavaScript console. Its jobs complete normally in $[35, 75]\,\text{ms}$ and we set the yield-budget size to 10.
4) A set of SchedCAT-generated tasks with a combined utilization of 0.75.
   (n=15, $p \in [1, 53]\,\text{s}$, $d = p$, WCET $\in [15, 1815]\,\text{ms}$)

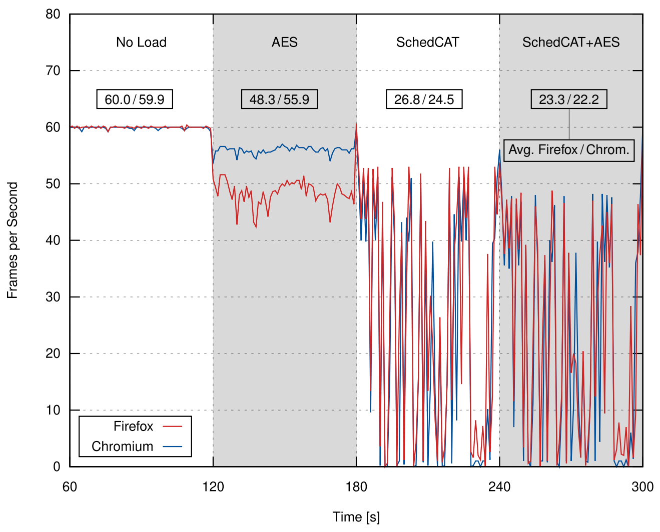
[3]This benchmark can be found at https://sra.uni-hannover.de/Research/rtjs/demo.html

The whole benchmark consists of 8 subtests that run for 60 seconds each. After an initial grace period of 60 seconds, we run the baseline benchmark, where the unmodified code is executed in a non-preemptive FCFS manner by the JavaScript runtime itself. In the first subtest ([60, 120] s), we only execute the box task and the input task with no additional background load. We simulate the user input with a timer that triggers every 500 ms and inputs one word from an English word list. In the following three subtests ([120,300] s), we activate different load scenarios: only the AES task, only the SchedCAT tasks, or all described tasks together. The task setup for each subtest is activated programmatically.

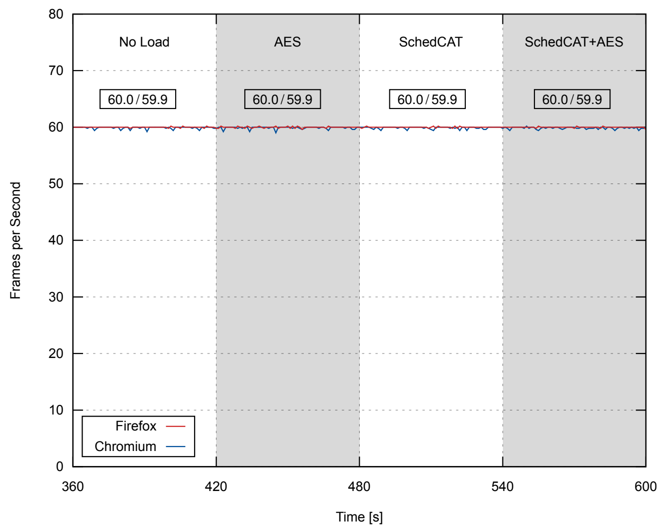
After the baseline test, we have an additional grace period of 60 seconds, such that we start the RT.js subtests at $t = 360\,s$. We execute the same scenario as for the baseline but with the RT.js scheduler enabled. Here, we use the EDF policy, a default yield budget of 300, and we give back control to the JavaScript event loop every millisecond. Only in these four subtests, we use the transpiled JavaScript code.

We run the described benchmark scenario in Firefox 67.0 and Chromium 73.0.3683.86. For measuring the frame rate, we use an additional `requestAnimationFrame()` callback that is not connected to the RT.js machinery. This callback records the frame rate during the benchmark and flushes them only after the benchmark has finished. If the JavaScript runtime is blocked at the time of the next animation frame, this callback gets executed late and the measured frame rate drops. We used this JavaScript-implemented FPS measurement since the browser-internal FPS graphing mechanisms did not provide adequate export mechanisms, configurable buffer sizes, and even skewed the FPS measurement for an idling system (Firefox). We executed the whole benchmark five times, reload the browser tab in between, and calculate the arithmetic average for each second.

Excluding the two grace periods, we show the measured FPS rates in Figure 8. Without the intense computation loads, we see that both systems do not hinder the page-rendering performance and reach the maximal value of 60 FPS. For the baseline JavaScript engine, we see how already the relatively short-running AES task, which introduces, on average, 40 ms of computation every 500 ms, introduces a noticeable FPS-rate decrease. We also see that the impact of these computations is much higher for Firefox than for Chromium. However, if we enable the SchedCAT tasks, which introduces a few long-running computations and has a utilization of 0.75, the frame



(a) FPS without RT.js scheduler (FCFS, non-preemptive)



(b) FPS with the RT.js scheduler (EDF, preemptive)

Fig. 8: Frames rendered per second in the macro benchmark

rate widely varies and regularly drops to zero. For the user, this results in an unusable web application, where the input field, as well as the animated box, freezes sometimes for several seconds.

By using the RT.js scheduler, the frame rate stabilizes and becomes an almost flat line. Due to the introduced PPs and the prioritized job scheduling, we could not measure any influence of the background computations on the page rendering. Thereby, the web application, although under heavy computation load, is able to react quickly to user input and produces visual feedback that is "smooth" for the user.

V. Discussion

A. Threats to the Validity of Evaluation Results

Regarding our evaluation results, we see the following threats to the validity of our results: First, we have no control about the JavaScript engine internals. For example, the just-in-time compilation could have optimized out some of our benchmark code. However, if we set the execution frequency (Section IV-B) for two integer increments (1 ns $\hat{=}$ 1 GHz) and for two user-level context switches (18 ns $\approx$ 55 Mhz) into the context of the evaluation machine, our results are in a reasonable range. Also, this lack of control poses no threat to the SchedCAT task sets as the exhaustion of WCET budgets was tied to actual progress of the physical time during the job execution.

Another concern could be that our benchmark tasks are not representative enough to reflect the code structure of actual applications. However, by using very tight loops with minimal bodies, we hit the worst case for our pseudo-preemptive rescheduling scheme as the ratio between computation code and PPs is highly unfavorable for us. We also cover a wide range of possible utilization factors and used an actual algorithmic payload (AES) in the macro benchmark.

B. Applicability

Regarding the generalizability, one could argue that manual annotation is cumbersome and error-prone. Even if the developer already uses RT.js, she could forget to annotate a long-running function. Since we only introduce PPs selectively, this could result in a single budget duration that exceeds its slice. However, RT.js could detect such excessively-long budgets automatically and warn the developer about it.

Furthermore, we could, in principle, also replace JavaScript jobs with RT.js jobs entirely: With a list of all runtime-API functions that take a callback for later invocation, we wrap each callback into a proxy function that submits an RT.js job. Together with making every user-defined function a generator, we would achieve that no job could ever monopolize the executor. However, this non-selective application of RT.js could lead to undesirable overheads due to deep `yield*` chains (see Table II).

C. Alternatives to Generators

In our development of RT.js, we also explored two alternative mechanisms to achieve preemptive and concurrent execution of functions: `async`/`await` and web workers. With the former, we can achieve the same preemption behavior by making all `@preempt` functions `async` and calling `await` on pseudo promises in every PP. The scheduler, then, resolves the promise of the currently running job which, again, waits after its budget is exhausted. However, in our measurements we saw that this alternative preemption mechanism performs poorly and achieves only about 700 preemptions per second. This performance decline stems from the subsequent reintroduction of jobs into *different* JavaScript queues that are handled subsequently by the event loop.

We also investigated on using web workers as job executors. Web workers, or service workers, are independent JavaScript runtime incarnations, which are supported by current browsers to offload computations. However, as they have some severe drawbacks, they are more orthogonal to RT.js than a competitor. Being independent from the main runtime, they share no state and all data transfer is done by explicit message passing; in particular, they cannot modify the DOM directly. Furthermore, they are quite heavy weight to start (≈40 ms, [15]) and require a separate JavaScript file as an entry point. For our approach, we can interpret web workers as separate processors and could spawn individual instances of RT.js, similar to a partitioned-scheduling real-time system.

D. *JavaScript Engine Optimizations and Improvements*

The proposed RT.js approach achieves preemption of jobs and prioritized scheduling by using injected generators to save and restore execution contexts. Thereby, RT.js can be applied to any existing JavaScript engine. Nevertheless, by only relatively small extensions to the JavaScript engine itself, it would be possible to boost RT.js' performance: We have seen that PPs are still expensive and get more expensive the deeper a `yield*` chain gets (Table II). This came to our surprise as we believe that `yield*` could, in principle, be implemented with the same overhead as `yield`. The top-level generator context could hold a pointer to the continued context, which is updated on every `yield*` and a pointer to the caller context of `next()`, so no `yield*`-chain walking is necessary.

In a similar direction, it would be highly beneficial for RT.js, to have actual coroutines [16] instead of having only generators. Both concepts differ in their possible return points: where a coroutine can continue its execution in any other coroutine, generators have to yield to their invoker. Therefore, our job switch involves two context switches (job→scheduler→job) instead of only one (job→job).

It would also be beneficial for RT.js to get an indication from the runtime, whether one of the JavaScript job queues contains at least one item. Thereby, we could drastically reduce the event-detection latency as we could end rounds early.

A deeper integration of RT.js into the JavaScript engine could be achieved by integrating prioritized job scheduling. As long as such an integration excludes the preemptability of jobs, the traditional execution model, and the accompanying atomicity guarantees, would remain backward compatible. In order to explore such a non preemptive but prioritized scheduling, one could exclude the transpilation step and only utilize the scheduler abstraction of RT.js.

VI. Related Work

A. *System Software for JavaScript*

With JavaScript still being the only universal language of the browser ecosystem, different projects provide transpilation from other languages to JavaScript. For example, Emscripten [17] automatically transpiles LLVM [18] IR code to JavaScript and provides, together with the C/C++ front end Clang, a complete transpilation chain from C/C++ to JavaScript. For the Go language, which has a built-in coroutines abstraction, GopherJS [19] provides the same service. Both transpilers replace blocking calls, like `sleep()` or reading from a Go channel, with `async` versions to implement passive waiting within jobs. For this, all functions that directly, or indirectly, call a blocking function, become asynchronous and the required `await` statements are inserted automatically. Thereby, both transpilers only provide the possibility of self-suspension of jobs, but leave out prioritized or preemptive scheduling. However, in principle, they can be combined with the orthogonal RT.js approach.

Powers et al. proposed Browsix [20], which brings standard Unix abstractions like the shared file system, processes, pipes, signals and sockets to the browser. The JavaScript-only framework provides the required Unix system calls and maps regular processes to web workers. With the help of Emscripten, they transpile large Unix programs, like $\TeX$, and execute them in the browser. Since Browsix uses web workers, they rely on the browser and the underlying operating system for preemption and scheduling. Thereby, Browsix cannot respect the relative importance of different jobs and they have to pay the communication overheads between the main thread and the web workers (see Section V-C).

The recently released WebAssembly [21] specification brings another, more low level, front end to the JavaScript engine. WebAssembly, which is already supported as an LLVM back end, defines a memory-safe virtual-machine interface that is supposed to be more load- and run-time efficient than JavaScript. While WebAssembly programs can directly manipulate the DOM, they have only implicit access to their call stack and there is no support for generators. In the specification, threads are only mentioned as a possible future extension; Google Chrome already supports an experimental version of WebAssembly threads [22], but there is no indication about prioritized thread execution. However, as WebAssembly functions can be called from JavaScript, they can be invoked as non preemptable leaf functions from RT.js jobs.

B. *From Events to (Pseudo-)Preemptive Scheduling*

RT.js integrates a thread concept with preemptive scheduling into an originally purely event-driven framework. As mentioned in the introduction, this transition of purely-event-driven-for-simplicity to multithreaded-for-reality appears to be a recurring pattern in the rise of frameworks addressing a new domain, such as sensor networks, cyber-physical systems, the Web 2.0, or the Internet of Things: Originally motivated by simplicity and resource constraints, Contiki [23] was quickly extended by protothreads [2], which are basically light-weight and explicit to employ function-local generators for C, but provide neither preemption nor scheduling facilities. TinyOS [24] got extended by TinyThreads [25], which implement full coroutines with FIFO scheduling, but no preemption, and TOSThreads [3], which provide preemption, but have to serializes all access to the TinyOS kernel by message passing for better backward compatibility – a goal that also has driven the design and implementation of RT.js. While Contiki and TinyOS target extremely constrained sensor nodes, Arduino [26] was intended as an easy-to-use platform for teaching and developing networked cyber-physical systems (a classical real-time domain) using a solely event-driven run-to-completion execution model. Here, Cheng, Li and West recognized the inherent limitations and presented Qduino [4], an RTOS that provides an extended Arduino API for threading and priority-based fully-preemptive scheduling, but replaces the existing Arduiono software stack. In contrast, RT.js is implemented for compatibility with existing JavaScript engines by exploiting JavaScript language features (i.e., generators) and the idea of pseudo-preemptive scheduling (i.e., forced cooperation by transpiler-inserted `yield` statements). A somewhat comparable approach to interrupt-less preemptive threading can be found in a workshop paper by Luc Bläser about an Oberon-based component model for embedded devices [27], in which "the compiler automatically inserts checks in the machine code, initiating preemption of a process if a certain execution time has passed."

VII. Conclusion

With RT.js, we presented a method to integrate (pseudo-) preemptive and prioritized job execution with the JavaScript execution model, while being compatible with unmodified JavaScript engines that implement the ECMAScript 6 standard. Thereby, we mitigate the impact of JavaScript's non-preemptive and first-come–first-served job-execution order on the user-input reaction times and improve the page-rendering performance.

In an upfront transpilation step, we introduce synchronous preemption points into the application and use JavaScript generators to regain control over the executor from an already started job. Combined with real-time scheduling policies, like earliest–deadline–first or fixed-priority, RT.js becomes a soft-real–time capable proxy layer between the JavaScript engine and the application jobs. By handing control back to the browser, we achieve bounded event-detection latencies and a high degree of reactiveness.

In the evaluation, we quantified the overheads of generator-based context switches to be lower than 20 ns and showed a drastic reduction of the deadline-miss ratios from, on average, 30 % to 0.045 % (FP) at a median runtime overhead of less than 5 %. For an RT.js-enabled web application, we achieved 60 FPS page-rendering although long-running and computing intense background jobs competed for the JavaScript executor.

The source code of RT.js is available at: `https://github.com/luhsra/RT.js`.

References

[1] The Eclipse Foundation, "2018 IoT developer survey results," 2018. [Online]. Available: https://www.slideshare.net/kartben/iot-developer-survey-2018

[2] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, Nov. 2006. [Online]. Available: http://dunkels.com/adam/dunkels06protothreads.pdf

[3] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan, "TOSThreads: Thread-safe and non-invasive preemption in TinyOS," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*. New York, NY, USA: ACM, 2009, pp. 127–140.

[4] Z. Cheng, Y. Li, and R. West, "Qduino: A multithreaded arduino system for embedded computing," in *Proceedings of the 36th IEEE International Symposium on Real-Time Systems (RTSS '15)*, 2015.

[5] J. C. Reynolds, "The discoveries of continuations," *Lisp Symb. Comput.*, vol. 6, pp. 233–248, Nov. 1993.

[6] MDN, "Intensive javascript," https://developer.mozilla.org/en-US/docs/Tools/Performance/Scenarios/Intensive_JavaScript, 2019, accessed: 2019-05-13.

[7] K. Gallaba, A. Mesbah, and I. Beschastnikh, "Don't call us, we'll call you: Characterizing callbacks in javascript," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2015, pp. 1–10.

[8] Node.js, "The Node.js Event Loop, Timers, and process.nextTick()," https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/, 2019, accessed: 2019-04-21.

[9] R. B. Miller, "Response time in man-computer conversational transactions." in *AFIPS Fall Joint Computing Conference (1)*, 1968, pp. 267–277.

[10] "Typescript," accessed: 2019-05-24. [Online]. Available: https://www.typescriptlang.org/

[11] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[13] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Systems Journal*, vol. 3, no. 1, pp. 67–99, Apr. 1991.

[14] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010, pp. 6–11.

[15] G. C. Marty, "How fast are web workers?" https://hacks.mozilla.org/2015/07/how-fast-are-web-workers/, 2015, accessed: 2019-05-23.

[16] M. E. Conway, "Design of a separable transition-diagram compiler," *Communications of the ACM*, vol. 6, pp. 396–408, Jul. 1963.

[17] "Emscripten," accessed: 2019-05-23. [Online]. Available: https://github.com/emscripten-core/emscripten/

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Washington, DC, USA: IEEE Computer Society Press, Mar. 2004.

[19] "Gopherjs - a compiler from go to javascript," accessed: 2019-05-24. [Online]. Available: https://github.com/gopherjs/gopherjs/

[20] B. Powers, J. Vilk, and E. D. Berger, "Browsix: Bridging the Gap Between Unix and the Browser," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 253–266, event-place: Xi'an, China. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037727

[21] "Webassembly specification - release 1.0," May 2019. [Online]. Available: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[22] A. Danilo, "Webassembly threads ready to try in chrome 70," Oct. 2018, accessed: 2019-05-29. [Online]. Available: https://developers.google.com/web/updates/2018/10/wasm-threads

[23] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki — a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, FL, USA, Nov. 2004.

[24] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Wireless Sensor Networks*, ser. Ambient Intelligence. Heidelberg, Germany: Springer-Verlag, 2005.

[25] W. P. McCartney and N. Sridhar, "Abstractions for safe concurrent programming in networked embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*. ACM, 2006, pp. 167–180.

[26] "Arduino - Homepage." [Online]. Available: http://arduino.cc

[27] L. Bläser, "A high-performance operating system for structured concurrent programs," in *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07)*. New York, NY, USA: ACM Press, Oct. 2007.