Optimizing Links for performance hungry applications

University of Edinburgh

20/Aug/2014 (version 1)
29/Aug/2014 (version 2)
28/Sep/2014 (version 3)

Dariusz Jędrzejczak
dariusz.jedrzejczak.work@gmail.com

version 3
[final]

Gist

This document describes various minor optimizations to Links JavaScript runtime (*jslib.js*) and their effectiveness. The overall performance of Links is assessed and possible optimizations are proposed and discussed.

The major part of the volume of this document are charts.

Observations

There's a lot of room for improvement. Simple optimizations increased performance siginficantly.

Problem

Garbage collector slowdowns.

Cause

Periodical collection of large amounts of garbage by the browser's garbage collector.

Proposed solutions

- Write applications in a way that avoids generating garbage. For Links it means enabling (more) ways to do it as well as optimizing the compiler and the runtime, so that less garbage is being generated.
- Optimize some Links' data structures and functions.
- Optimize JavaScript generated by the compiler.
- Implement a custom garbage collector/give the user more control over memory.
- Make the compiler generate code for a language that compiles to LLVM bytecode, then generate fast JavaScript from it using Emscripten[1] (or something similar). This would most likely also mean implementing a custom garbage collector.

1 http://en.wikipedia.org/wiki/Emscripten

The following charts illustrate the effectiveness of the optimizations:

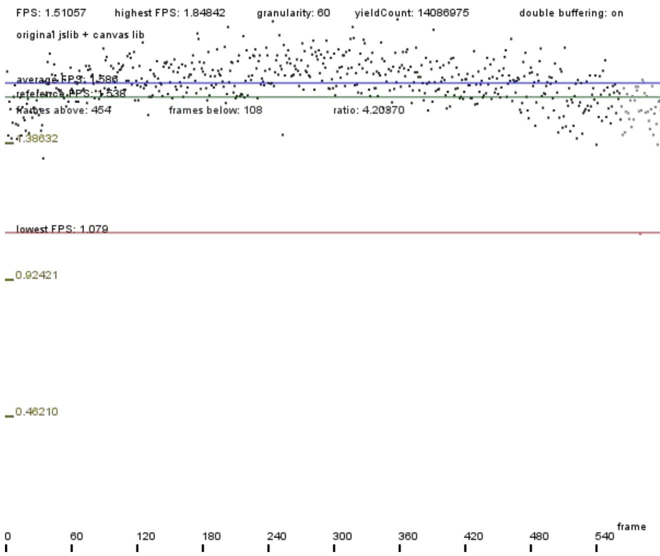


Chart 1: Before: average FPS = 1.5

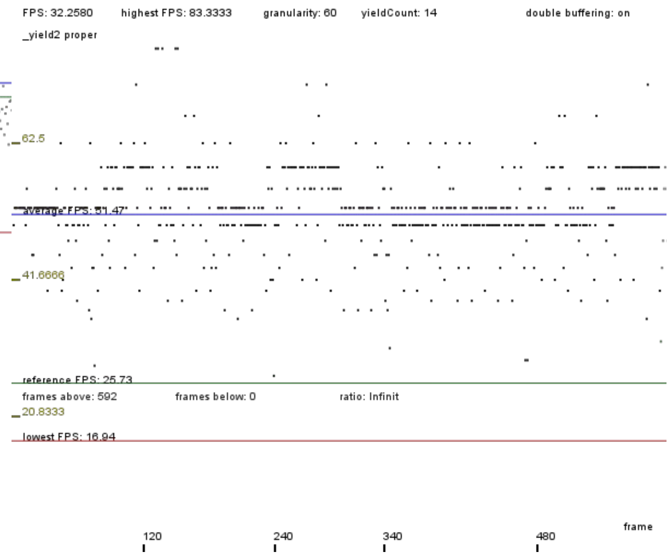


Chart 2: After: average FPS = 51

Contents

Gist....2
  Observations....2
  Problem....2
  Cause....2
  Proposed solutions....2
Contents....4
Introduction....5
  The benchmark application....5
  Files attached to this document....6
Basic optimizations....8
  The problem....9
  The unoptimized version....9
  First optimization – optimized _yield and _yieldCont....10
  Second optimization – faster setTimeout....11
  Third optimization – increasing _yieldGranularity....12
  Fourth optimization – turning off double buffering....13
  First two optimizations combined....14
  First three optimizations combined....15
  First two optimizations without double buffering....16
  Garbage collector's behavior....17
More optimizations and profiling....19
  JavaScript Optimizer....19
  Comparison with the native verison....23
  Profiling....27
  Other optimizations....28
Lists, equality and _yield optimizations....29
  Base....29
  Lists....30
    Linked list type....30
    Linked list type with native take and drop....31
    JavaScript linked lists....33
    JS lists with null....36
  Equality....37
  _yield....41
    Chromium....41
    Firefox....42
    Profiling....43
Conclusions....46
  Suggestions for future work....46

Introduction

This section describes how the performance measurements were carried out.

I tried to make sure that the measurements are done properly. Considering the short time in which all this was done, I might have missed something (most likely did), but nonetheless acquired data shows interesting things. The following paragraphs describe how the data was obtained.

The benchmark application

I wrote an application (see the file *performance-frozen.links* – attached to this document) in Links, which displays a chart of instantaneous FPS[2] for every frame (numbers of frames are on the X axis and the instantaneous FPS is on the Y axis).

The application is itself very resource consuming, though all it does is processing 600 samples of FPS data and drawing on the screen. I didn't make attempts at optimizing it, though, because that's irrelevant – it's enough that the same application is used unchanged for every measurement.

Links' debug mode was off during all tests (`debug=off` in the config file). I made sure not to have any extra applications running in the background while testing (aside from a text editor, file manager and a terminal emulator, which were running constantly). All tests were performed using Chromium 36.0.1985.143 on Arch Linux on an ASUS X53S laptop.

I generated hundreds of charts. For this document I selected a representative chart for each optimization.

The next section describes all relevant files that were used during measurements: the benchmark application as well as the runtime. Different versions of the runtime were produced by modifying the original. This allowed me to easily test different optimizations in isolation as well as in combination. For inspecting and comparing, I suggest using a diff tool on different versions of the files.

2 http://en.wikipedia.org/wiki/Frame_rate. *Instantaneous* means that it indicates how many frames could be processed in a second, if all frames in that second took as much time to process as the current frame. **Note**: I'm using the word *framerate* and *FPS* interchangeably throughout this document.

Files attached to this document

Various versions of the benchmark application are attached to this document as the following files:

- **performance-frozen.links** – the original benchmark application in Links (most charts in this document were generated by it)
- **performance-frozen-optimized.html** – a version of the original benchmark optimized by the Google Closure Compiler[3]
- **performance.html** – native JavaScript version of the benchmark application
- **BASE performance-frozen-lists.links** – performance-frozen.links with a custom list type defined in Links. This is the base for all files named **performance-frozen-lists.*
- **TAKE-DROP performance-frozen-lists.links** – uses JavaScript versions `lsTake` and `lsDrop` (which work like `take` and `drop`) defined in *JS lists 2 - map jslib.js*
- **JS LISTS 2 - MAP performance-frozen-lists.links** – uses a custom JavaScript linked list type (defined in *JS lists 3 - map jslib.js*)
- **JS LISTS 3 - MAP performance-frozen-lists.links** – uses a custom optimized JavaScript linked list type (defined in *JS lists with null - map jslib.js*)
- **specialized equality performance-frozen-lists.links** – uses specialized functions for testing equality (defined in *specialized equality jslib.js*)

Other files attached to this document:

- **lib.ml** – the original *lib.ml* + interface for canvas manipulation
- **lib 2.ml** – the above *lib.ml* + interface for manipulating linked lists and specialized equality functions
- **irtojs.ml** – the original *irtojs.ml* adjusted for an optimized version of `_yield`

3 https://developers.google.com/closure/compiler/

Various versions of the Links runtime (*jslib.js*)[4]:

- **original jslib + canvas lib.js** – the original (unoptimized) version of *jslib.js* which was used as a reference – I added only the interface for canvas manipulating functions to it. I used the *jslib.js* file from GitHub – from the version of *sessions* branch (last commit July 30), which my branch (*dariusz*[5]) was derived from.
- **optimized _yield and _yieldCont jslib.js** – the original with optimized versions of `_yield` and `_yieldCont` functions – the optimization removed any references to functions in the `DEBUG` namespace from the body of `_yield` and `_yieldCont` and made some other minor changes
- **setZeroTimeout jslib.js** – the original with all calls to `setTimeout` with the second argument of 0 replaced by a call to `setZeroTimeout`[6]
- **_yieldGranularity + 200 jslib.js** – the original with `_yieldGranularity` increased from 60 to 260
- **new base jslib.js** – the original with optimizations from *optimized _yield and _yieldCont jslib.js* and *setZeroTimeout jslib.js* combined
- **optimized yield + setZeroTimeout jslib.js** – same as previous
- **new base + _yieldGranularity + 200 jslib.js** – *new base jslib.js* with `_yieldGranularity` increased from 60 to 260
- **google closure jslib.js** – modified *new base jslib.js* with a function for invoking Chromium debugger; this file was used as part of the input to Google Closure Compiler; the whole input is attached in the file **google closure input.js**
- **JS lists 2 - map jslib.js** – adds a few functions for manipulating a linked list type defined in *JS LISTS 2 - MAP performance-frozen-lists.links*
- **JS lists 3 - map jslib.js** – defines a linked list type (based on the one in the Elm language[7]) and functions for manipulating it entirely in JavaScript. Used with *JS LISTS 3 - MAP performance-frozen-lists.links*
- **JS lists with null - map jslib.js** – *JS lists 3 - map jslib.js* with further optimizations of the linked list type
- **specialized equality jslib.js** – adds specialized JavaScript functions for testing for equality as (theoretically faster) an alternative to `LINKS.eq`
- **proper yield2 jslib.js** – adds an optimized version of _yield that required a slight change in the JavaScript generated by the compiler (*irtojs.ml* was adjusted for this optimization – it is attached to this document)

4 The names of the attached files approximately correspond to descriptions (if present) found on charts
5 https://github.com/slindley/links/compare/dariusz
6 Implementation from http://dbaron.org/log/20100309-faster-timeouts
7 http://elm-lang.org/elm-runtime.js

Basic optimizations

This section contains the generated charts with descriptions.

The chart-generating application works like this: every frame the highest and the lowest FPS is updated if needed. Every 600 frames (an *iteration*) the average FPS is calculated and collecting samples starts over. The samples from the previous iteration are marked with gray and the samples from the current iteration are marked with black.

Every chart consists of:

- X axis (frames): from 0 to 600, a mark every 60 frames
- Y axis (instantaneous FPS): from 0 to the highest registered FPS, marks at 25, 50 and 75% of the highest FPS
- Blue line – indicating the average FPS (calculated over 600 frames from the previous iteration)
- Green line – user-defined reference FPS. Can be moved with up and down arrow keys. Below this line are three values dependent on its position:
    - Frames above – how many samples calculated in this iteration lie above the reference line
    - Frames below – how many samples lie below the line
    - Ratio – the number of frames above the line divided by the number of frames below
- Red line – indicates the lowest instantaneous FPS
- Text at the top:
    - FPS – current instantaneous FPS
    - highest FPS – highest instantaneous FPS
    - granularity – the value of `_yieldGranularity` (constant)
    - yieldCount – current value of `_yieldCount`; in the charts without the first optimization this value is very high as it is incremented every time `_yield` or `_yieldCont` is called (eventually it overflows, which may cause an unplanned stack clear); after the optimization the value is reset when it reaches the value of `_yieldGranularity`
    - double buffering – indicates whether double buffering[8] is on or off[9]
    - description – the second line from the top describes the chart[7]

8 http://en.wikipedia.org/wiki/Multiple_buffering#Double_buffering_in_computer_graphics

9 See the last section of this document for details

The problem

There's a clear pattern in the optimized versions: every n frames the FPS drops significantly – this is caused bythe garbage collector collecting large amounts of garbage periodically[10]. The green line on every chart may be helpful in estimating the n.

The unoptimized version

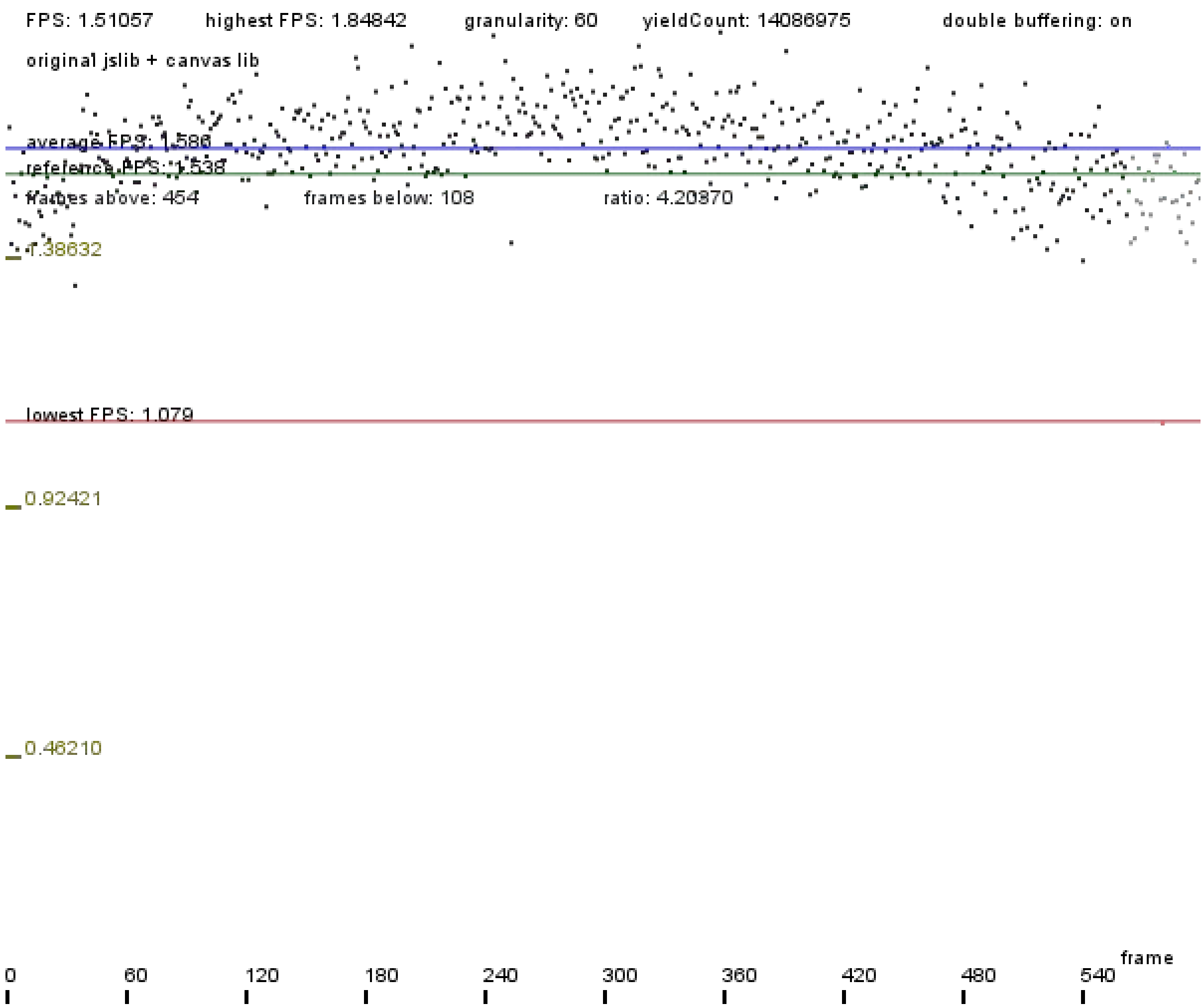


Chart 3: The unoptimized version

Average FPS ~ 1.6. This is the reference.

10 See the section: Garbage collector's behavior.

First optimization – optimized `_yield` and `_yieldCont`

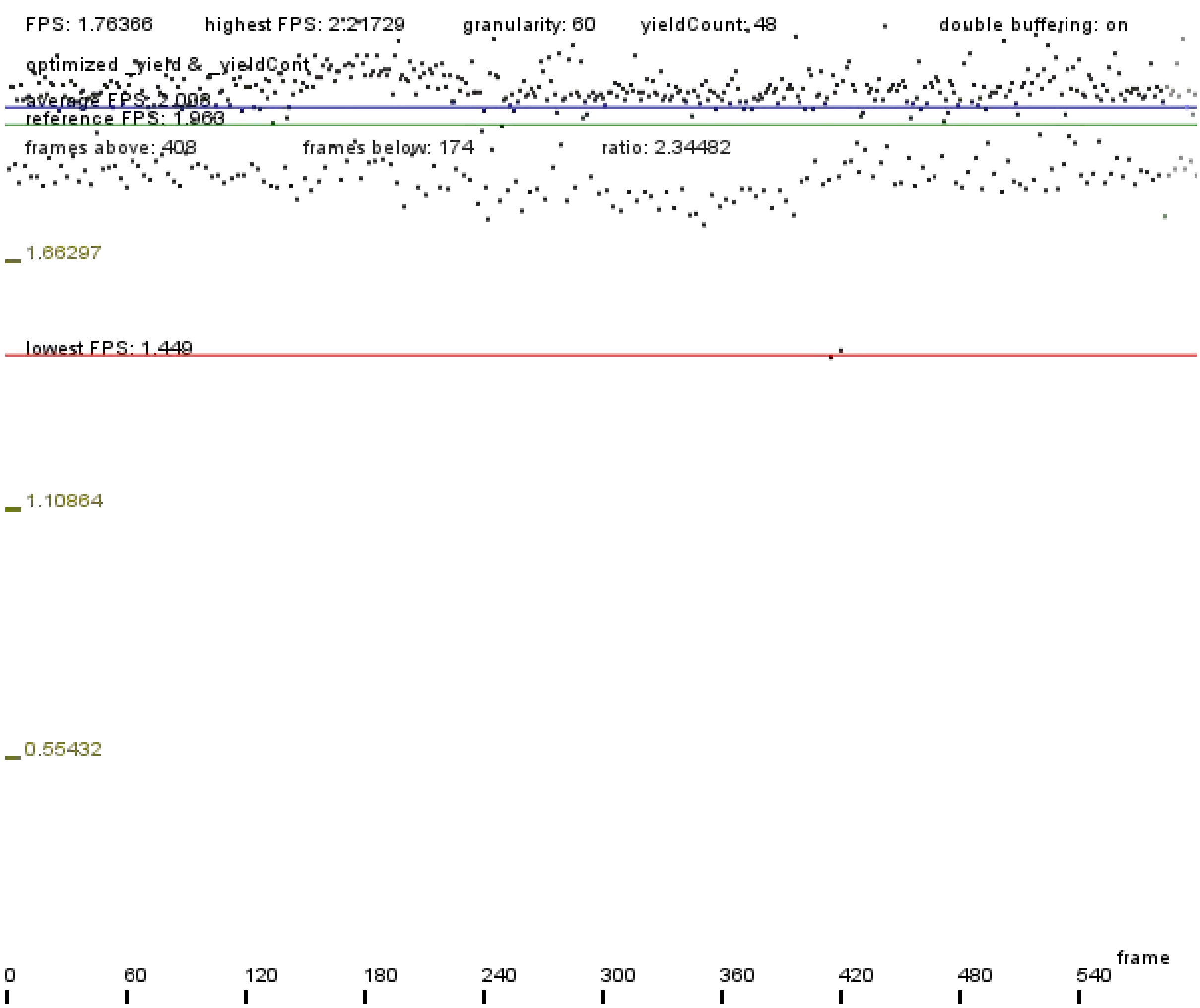


Chart 4: First optimization – faster `_yield*`

Average FPS ~ 2.

Removing some calls to debugging functions from the bodies of `_yield` and `_yieldCont` and getting rid of one modulo operation and one negation improved the performance a bit. In fact the improvement is much more significant than what comparing this chart to the previous may indicate, as we'll see when we combine this optimization with the next one.

We can see the pattern described in The problem: the framerate oscillates between some higher and lower value. It drops almost every third frame.

Second optimization – faster `setTimeout`

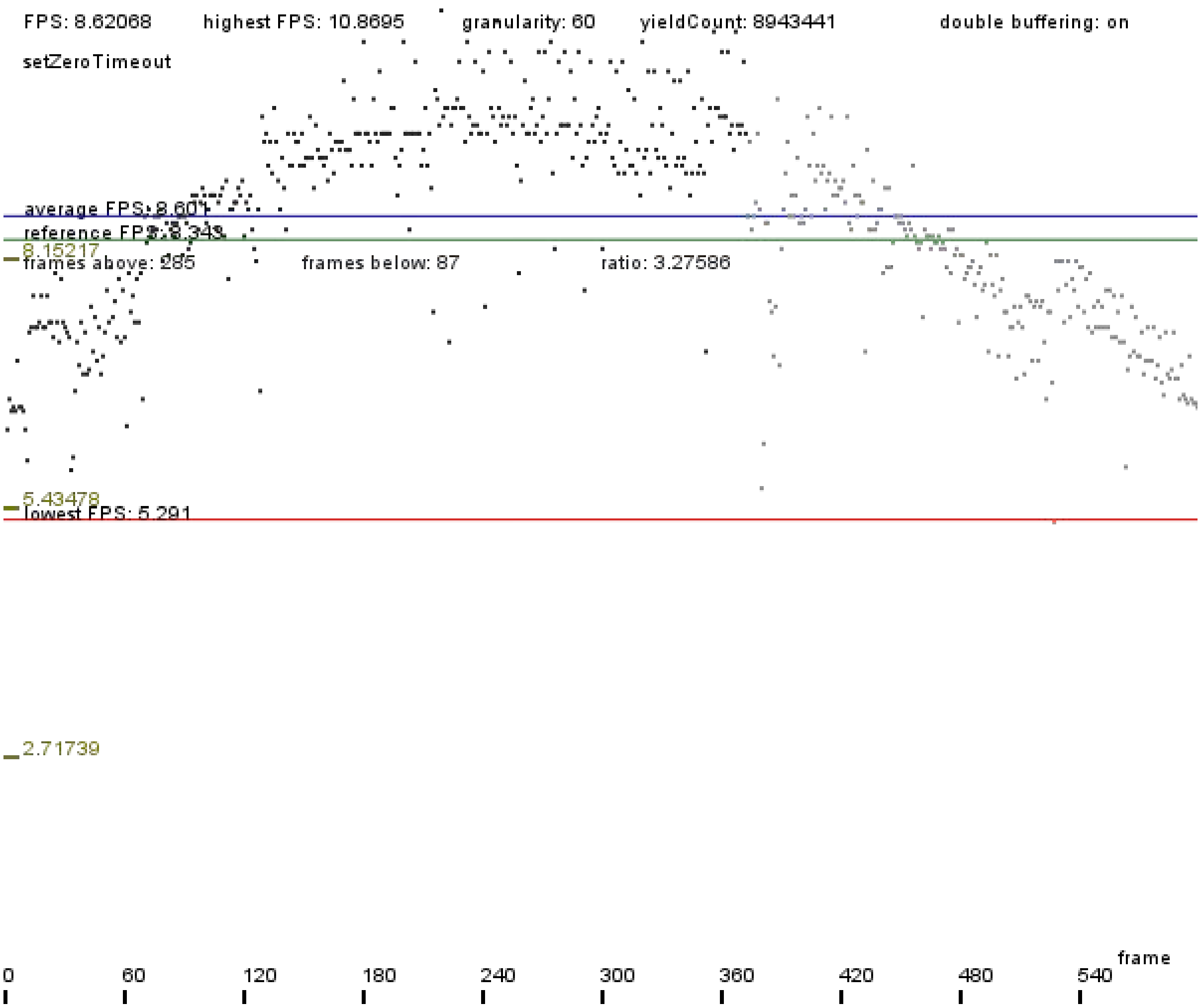


Chart 5: Second optimization – `setZeroTimeout`

Average FPS ~ 8.6.

All calls to `setTimeout` with the second argument of 0 were replaced by a call to `setZeroTimeout`[8] – a major optimization. `setTimeout` effectively has a minimum delay of about 4 ms[11]. `setZeroTimeout` doesn't have that limitation.

The oscillation of the FPS is more apparent when the `_yield*` optimization is applied – so not here. This is most likely because the amount of time spent yielding each frame is much longer without the optimization and garbage collecting time stays roughly the same.

11 https://developer.mozilla.org/en/docs/Web/API/window.setTimeout#Minimum.2F_maximum_delay_and_timeout_nesting

Third optimization – increasing `_yieldGranularity`

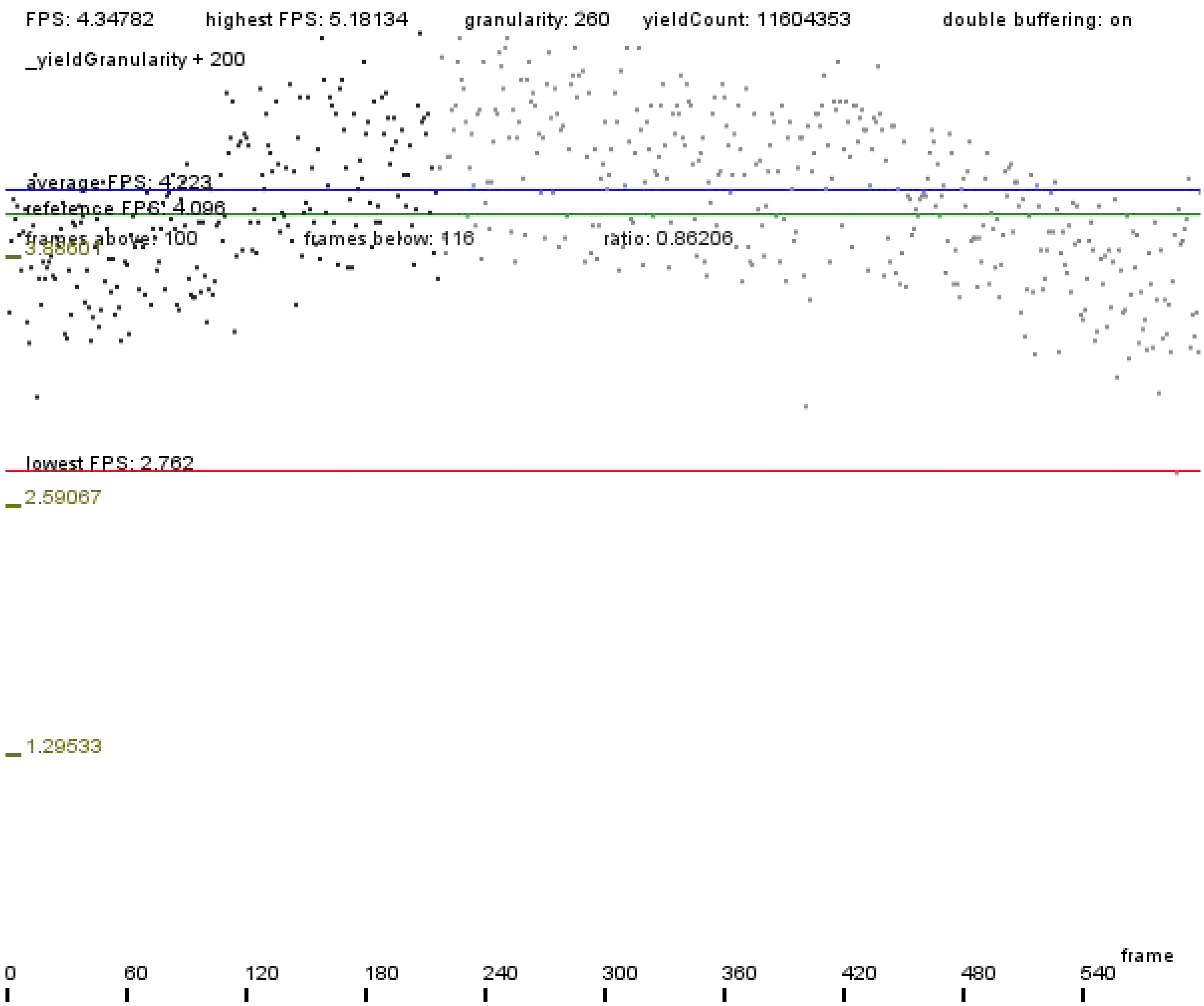


Chart 6: Third optimization – calling `setTimeout` less often

Average FPS ~ 4.2.

Increasing `_yieldGranularity` obviously has an impact on performance, as stack is not cleared so often (but this works up to a point[12] and the maximum value of `_yieldGranularity` differs between applications and browsers).

12 Because as `_yieldGranularity` grows the amount of garbage being accumulated also grows (see Chart 12)

Fourth optimization – turning off double buffering[13]

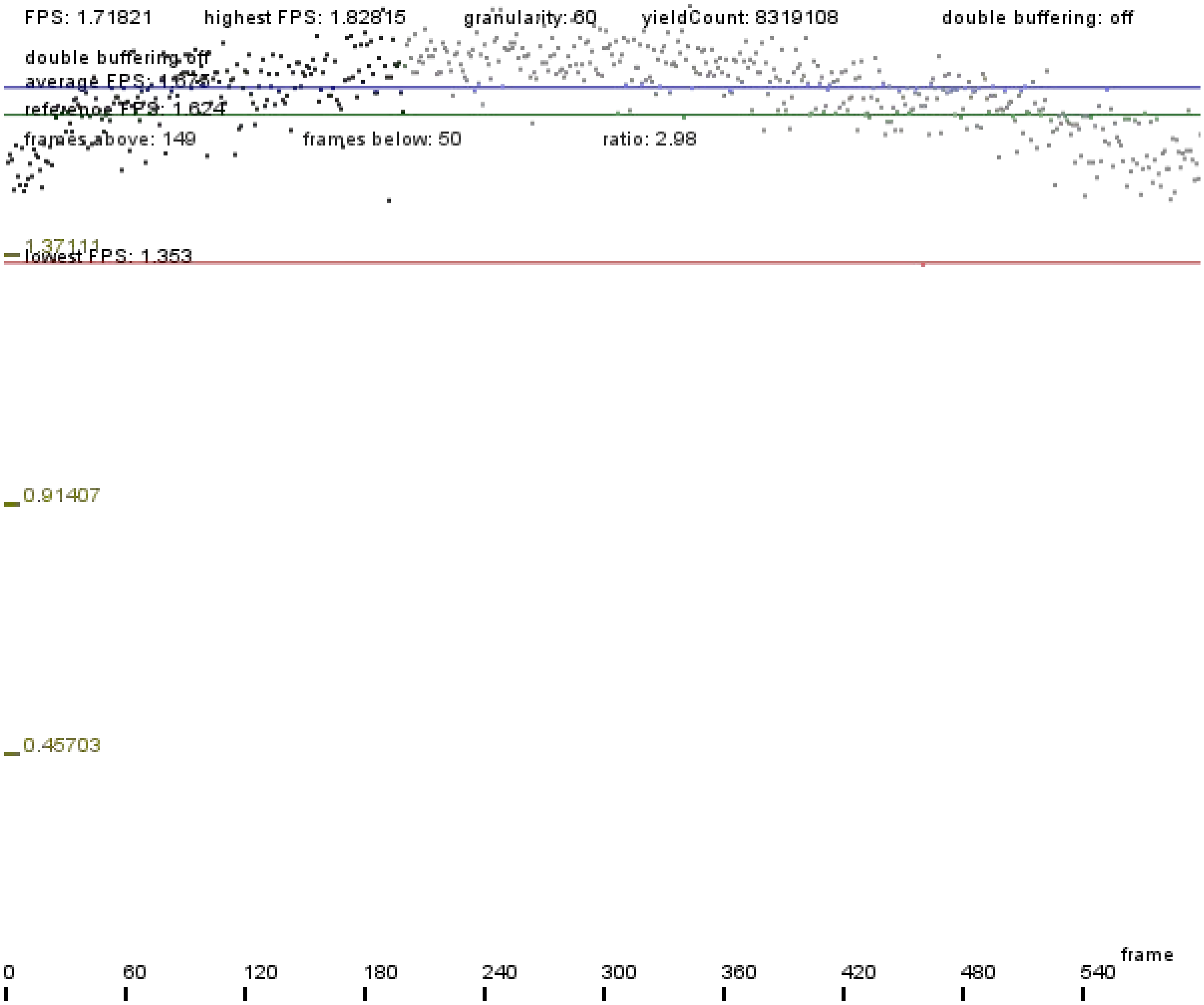


Chart 7: Fourth optimization – double buffering off

Average FPS ~ 1.7.

Almost no effect compared to the original. That's most probably because drawing is one of the least performance significant parts of the benchmark application. Doing similar tests with a Breakout clone in Links shows that turning off double buffering has, as expected, a significant effect.

13 Normally browsers do double buffering automatically (http://www.mail-archive.com/whatwg@lists.whatwg.org/msg19969.html), but it won't work for Links, most likely because the generated JavaScript for the drawing makes asynchronous calls all the time, so currently to avoid flickering of the canvas double buffering has to be always on.

First two optimizations combined

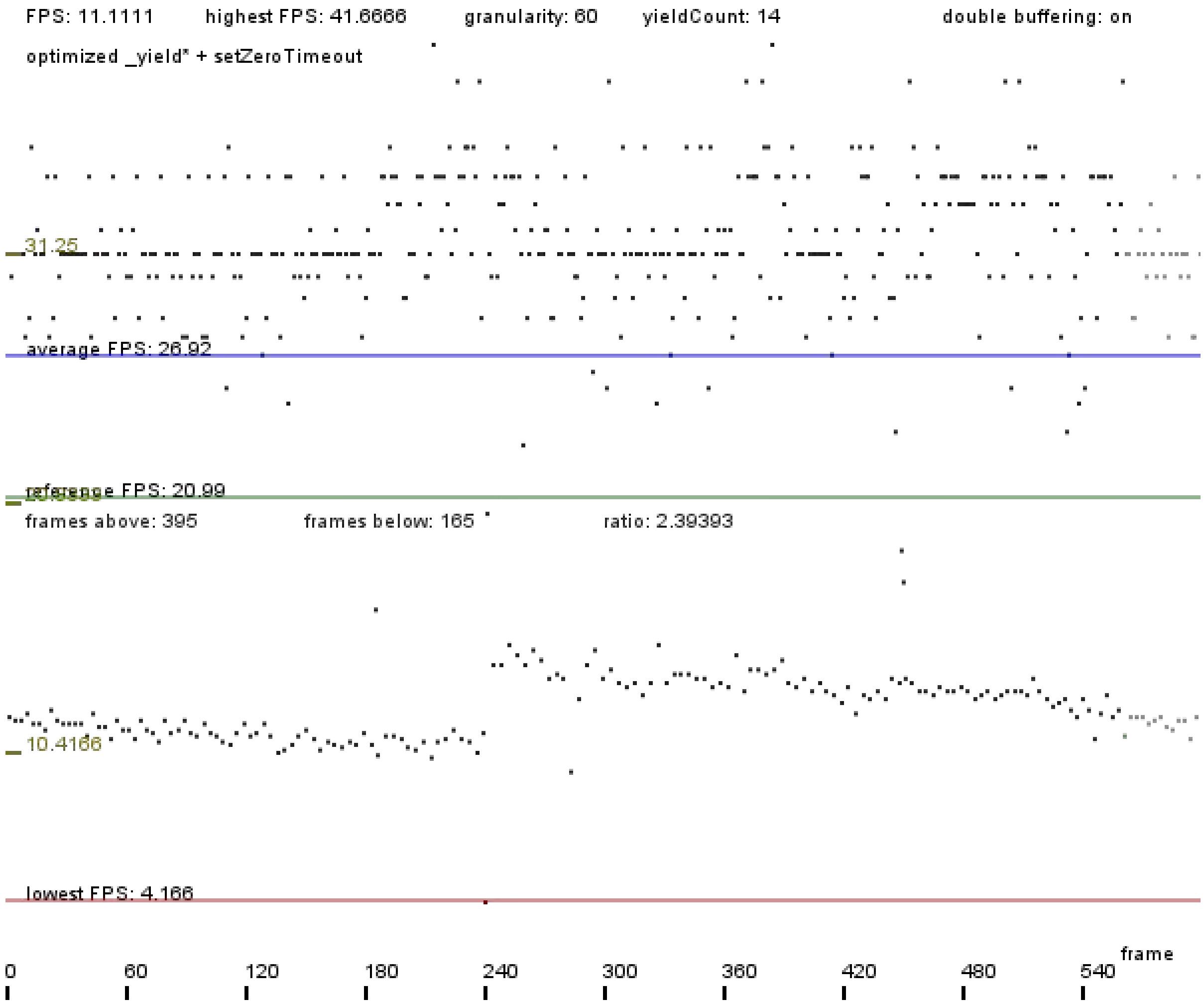


Chart 8: First two optimizations combined

Average FPS ~ 27.

Great improvement. The oscillation clearly apparent.

First three optimizations combined

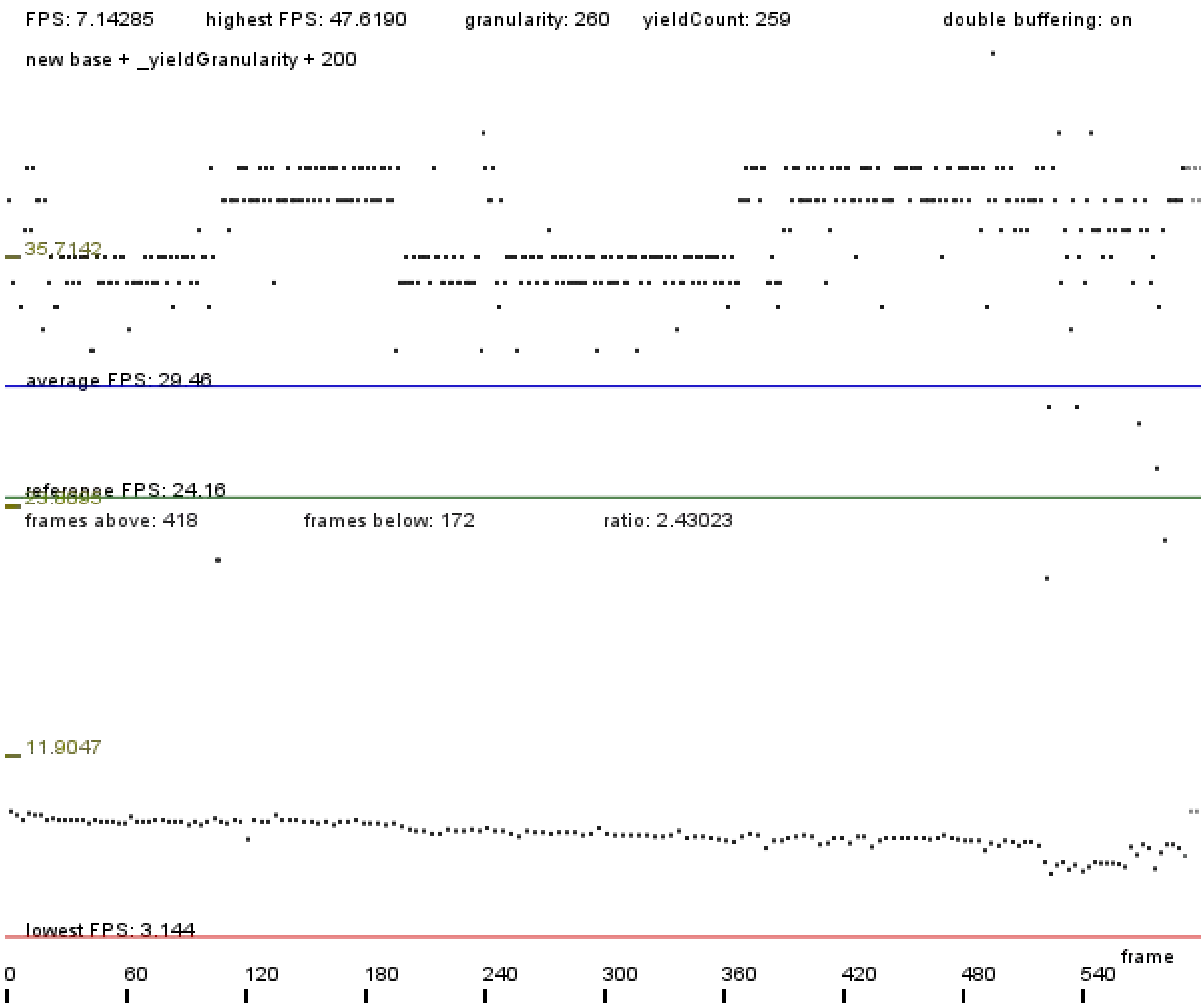


Chart 9: First three optimizations combined

Average FPS ~ 29.

An increase in `_yieldGranularity` bumps up the FPS a bit, but not that significantly.

First two optimizations without double buffering

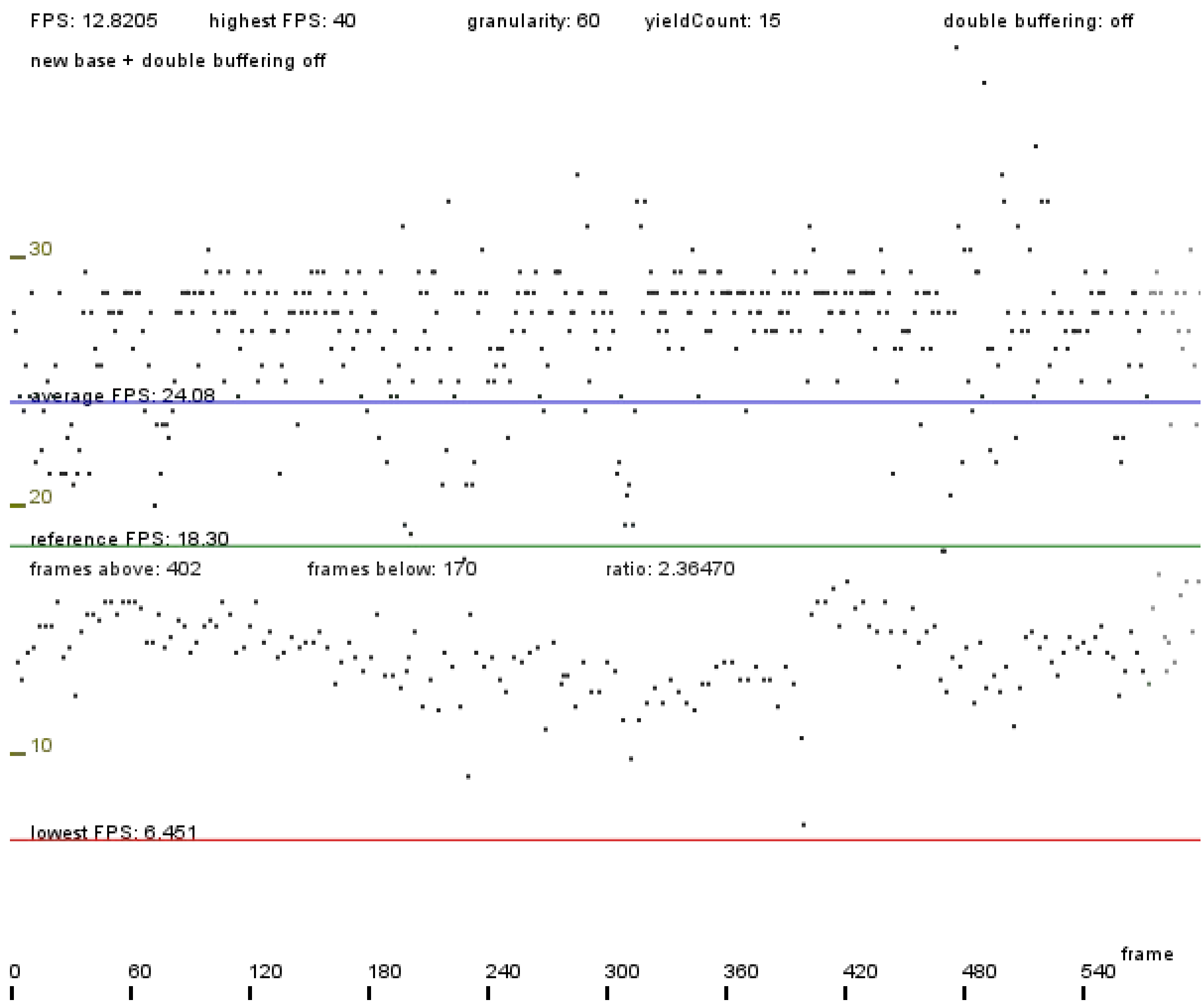


Chart 10: First two and the fourth optimization combined

Average FPS ~ 24.

Virtually no improvement over the two optimizations alone.

Garbage collector's behavior

The charts in this section were made using a modified version of the benchmark application – some variation of *performance2.links*.

To confirm that the GC is the cause of drops in FPS, I added a function that allows invoking Chromium's garbage collector on demand to Links. I put calls to this function after code that I thought was responsible for generating a lot of garbage – calling map on a big list (I forced 2 GC invocations per frame). This indeed, stabilized the framerate:

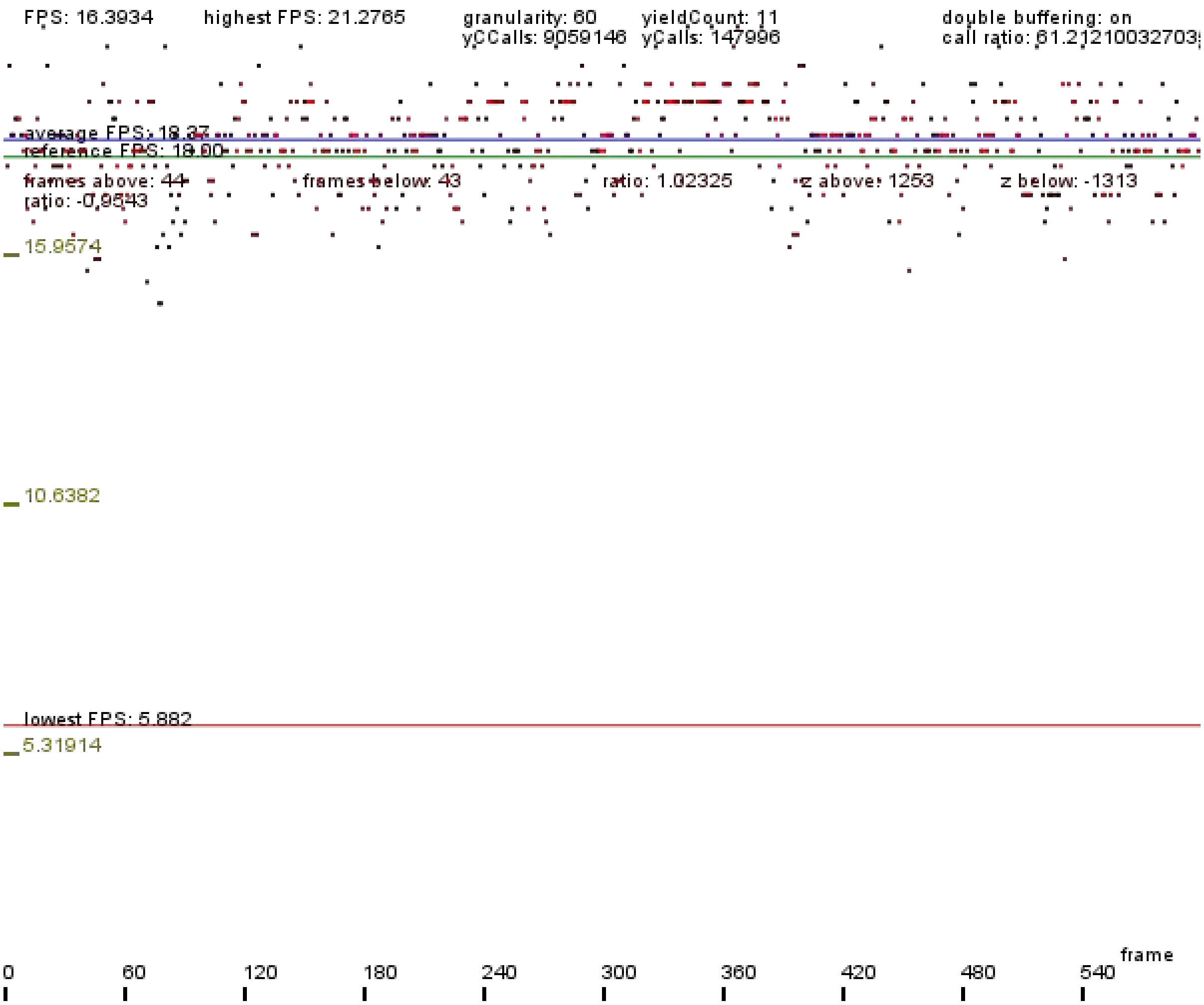


Chart 11: Invoking GC after mapping a function over a big list to clean up stabilizes the FPS; optimizing the implementation of map may significantly boost the performance

When I increased `_yieldGranularity` too much, the pattern on the chart changed – GC was invoked more often. Probably because the bigger the stack grows, the more garbage accumulates.

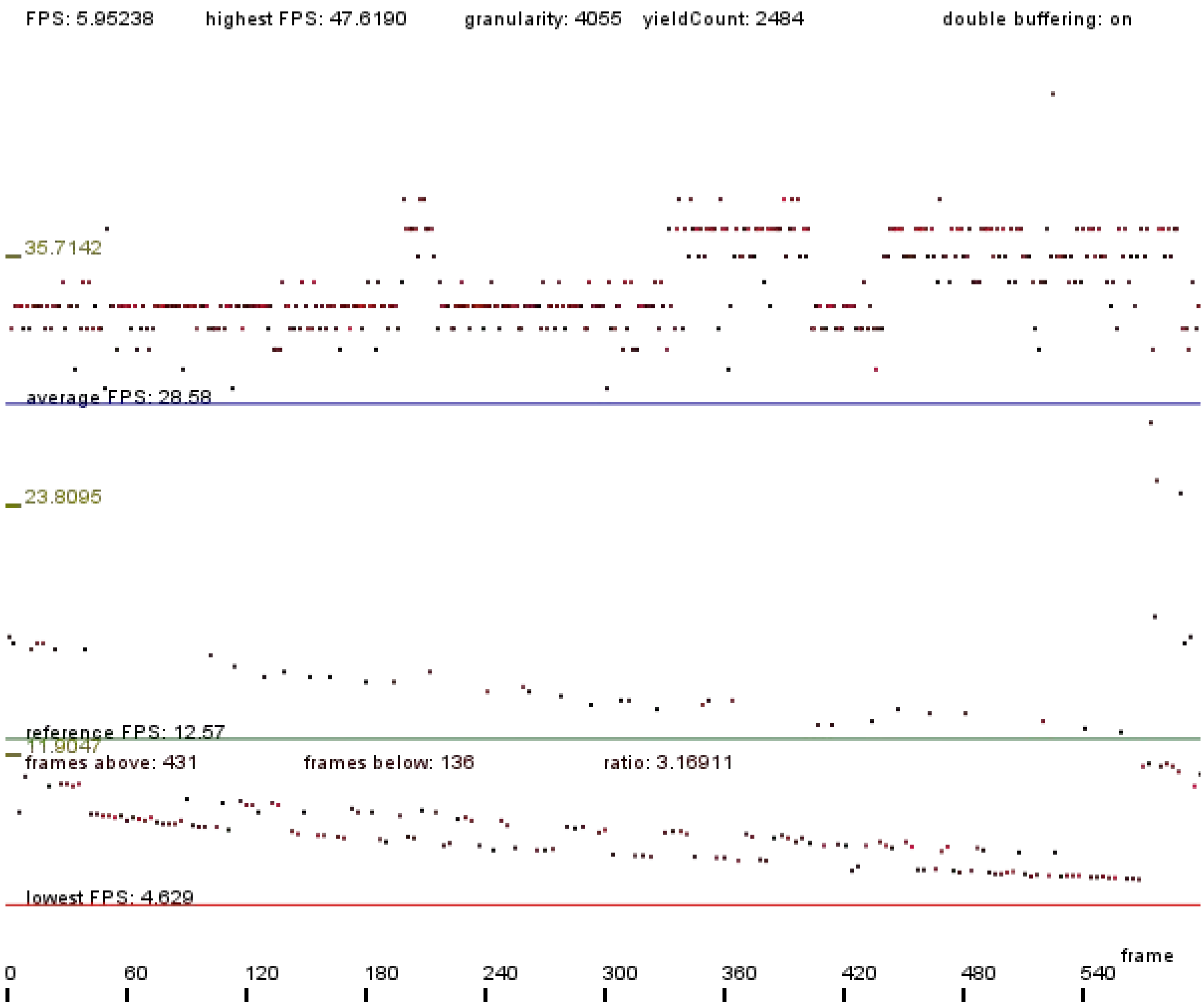


Chart 12: Too high `_yieldGranularity` results in more GC slowdowns

More optimizations and profiling

This section describes various other optimizations that I attempted. It also includes a comparison of the benchmark with a native JavaScript implementation.

JavaScript Optimizer

I tried running the runtime (*jslib.js*) and the JavaScript generated by the Links compiler through the Google Closure Compiler[14]. This required some modifications to *jslib.js*, but I eventually got it running (the final input to the Closure Compiler is attached to this document as *google closure input.js*). The effects are presented below:

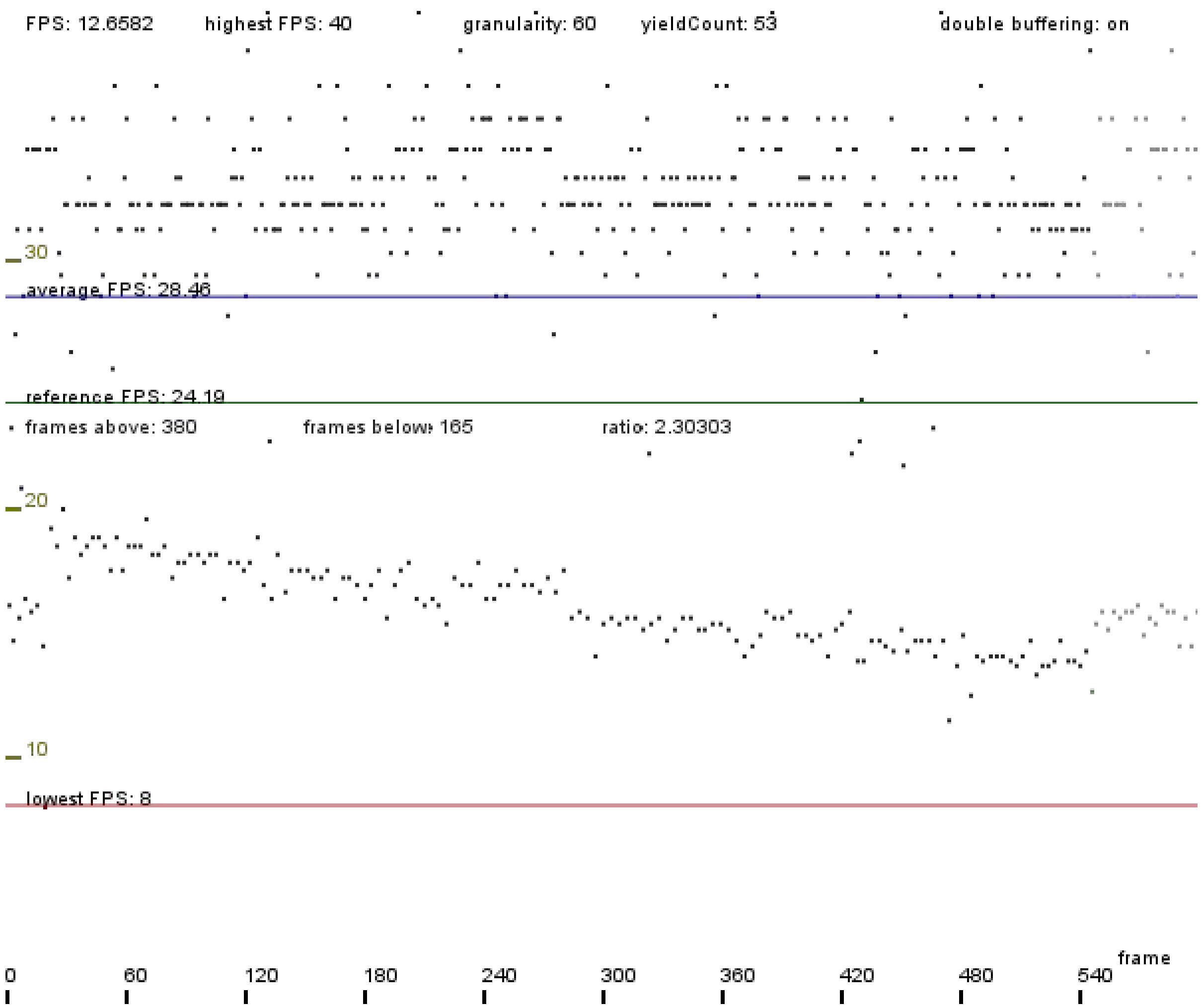


Chart 13: Performance of the ***unoptimized*** *code*

14 https://developers.google.com/closure/

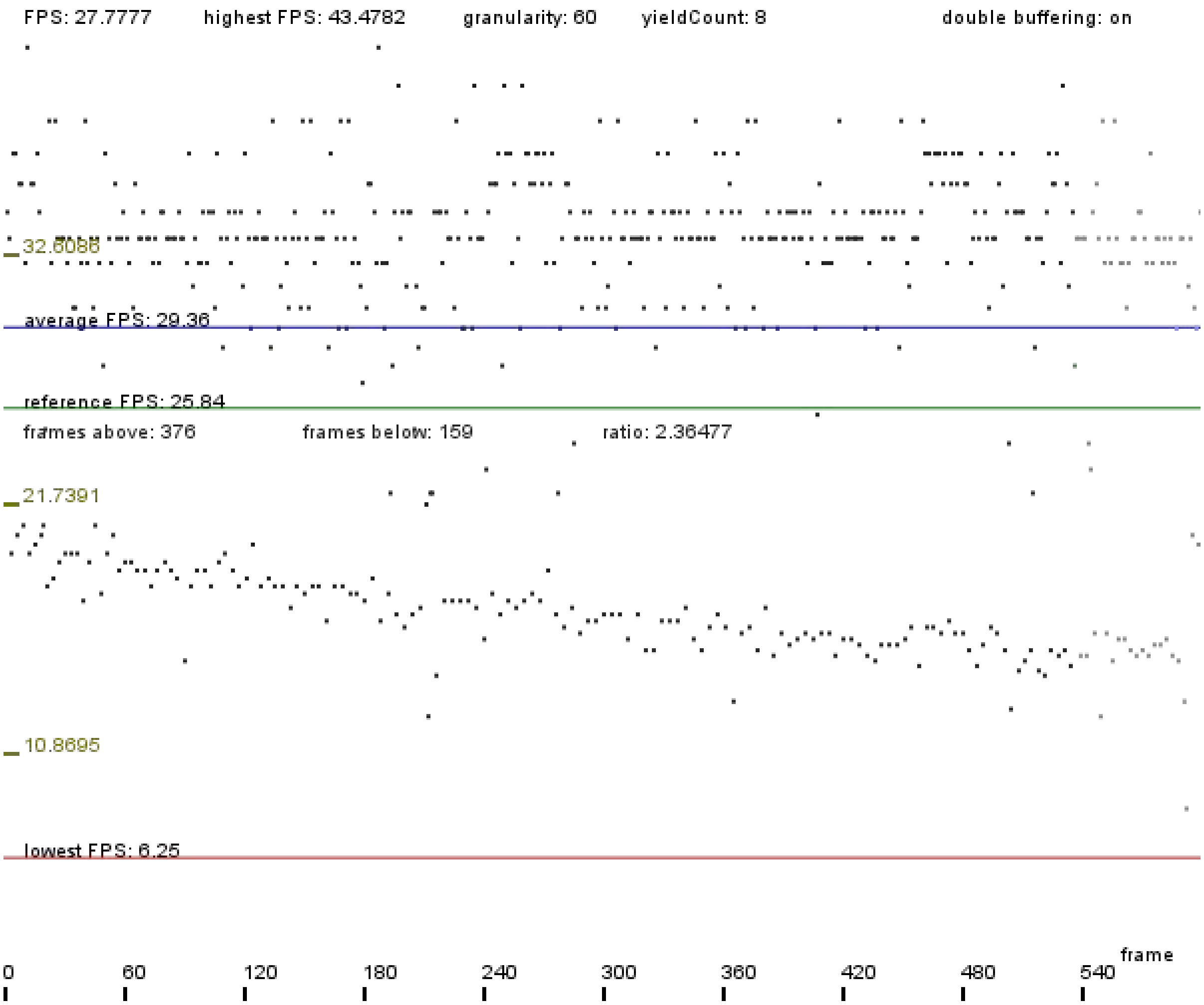


Chart 14: Performace of the code ***optimized*** *with the Closure Compiler*

We can notice a very slight improvement. I tested the code multiple times, also on Firefox. The effect was consistent.

Although this optimization didn't improve the performance significantly in this case, running the output of the compiler through an optimizer can be beneficial for bigger applications.

The next two charts present the comparison of profiling timeline plots (note the heap size – the blue line) between the original and the optimized code.

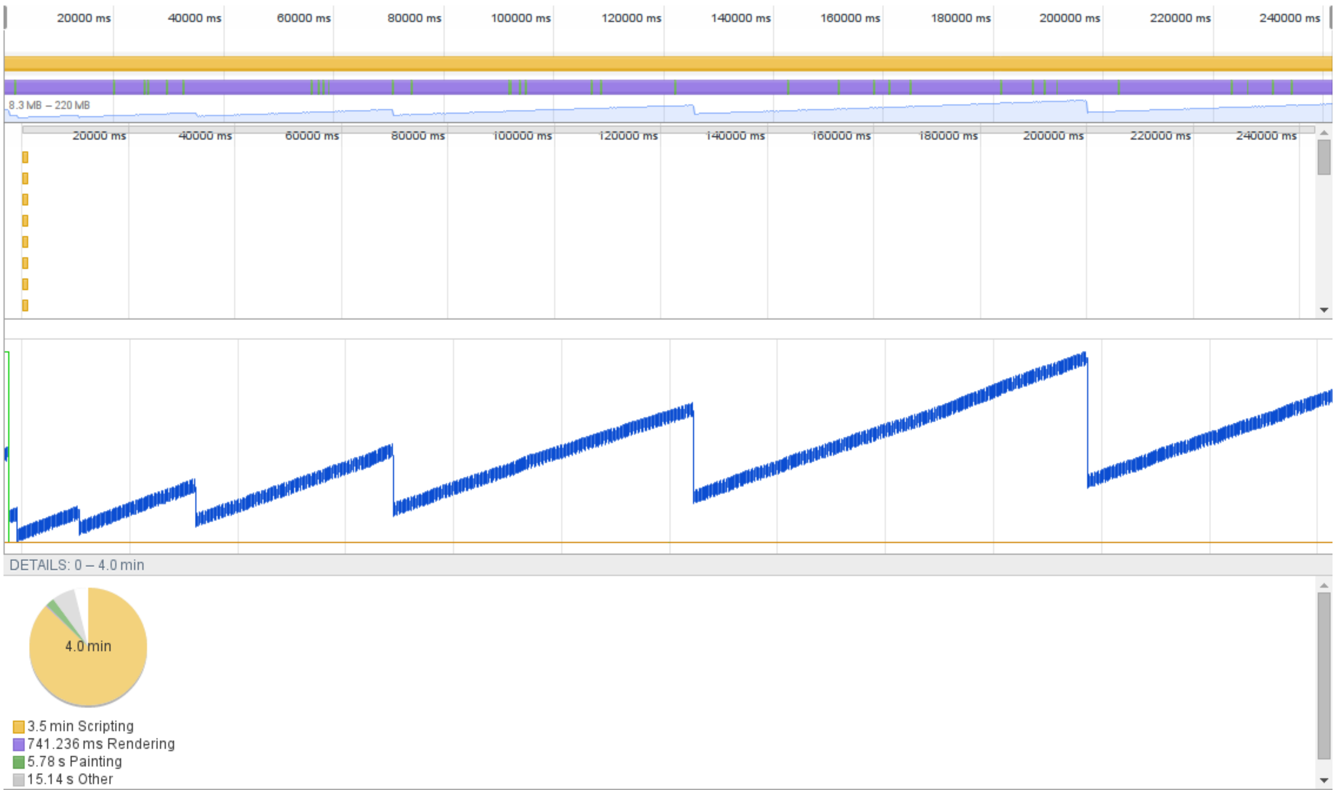


Chart 15: Chromium profiling timeline plot for ***unoptimized*** *code*

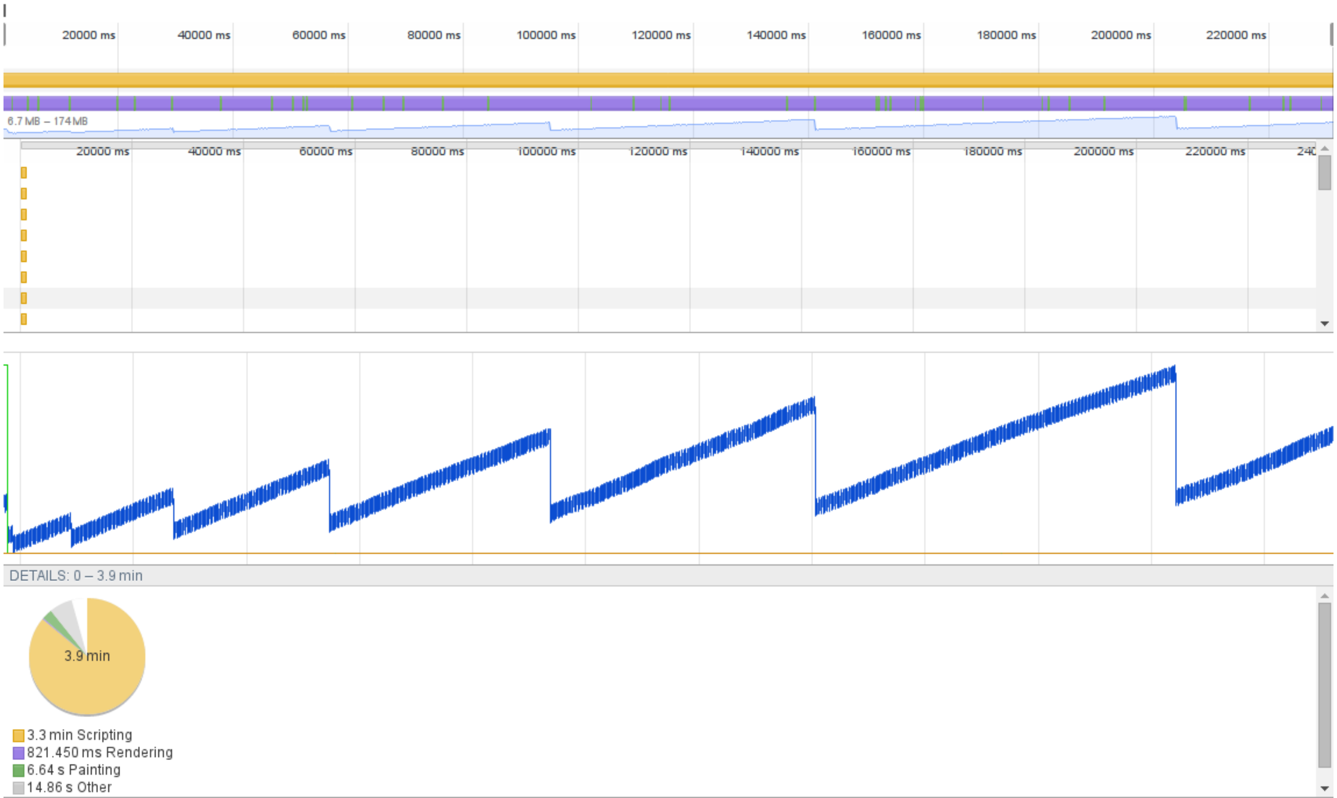


Chart 16: Chromium profiling timeline plot for code ***optimized*** *with the Closure Compiler*

Again, we see no significant difference. Perhaps a slight improvement.

Comparison with the native verison

It's also interesting to compare the previous timelines with a timeline for the native JavaScript version of the benchmark:

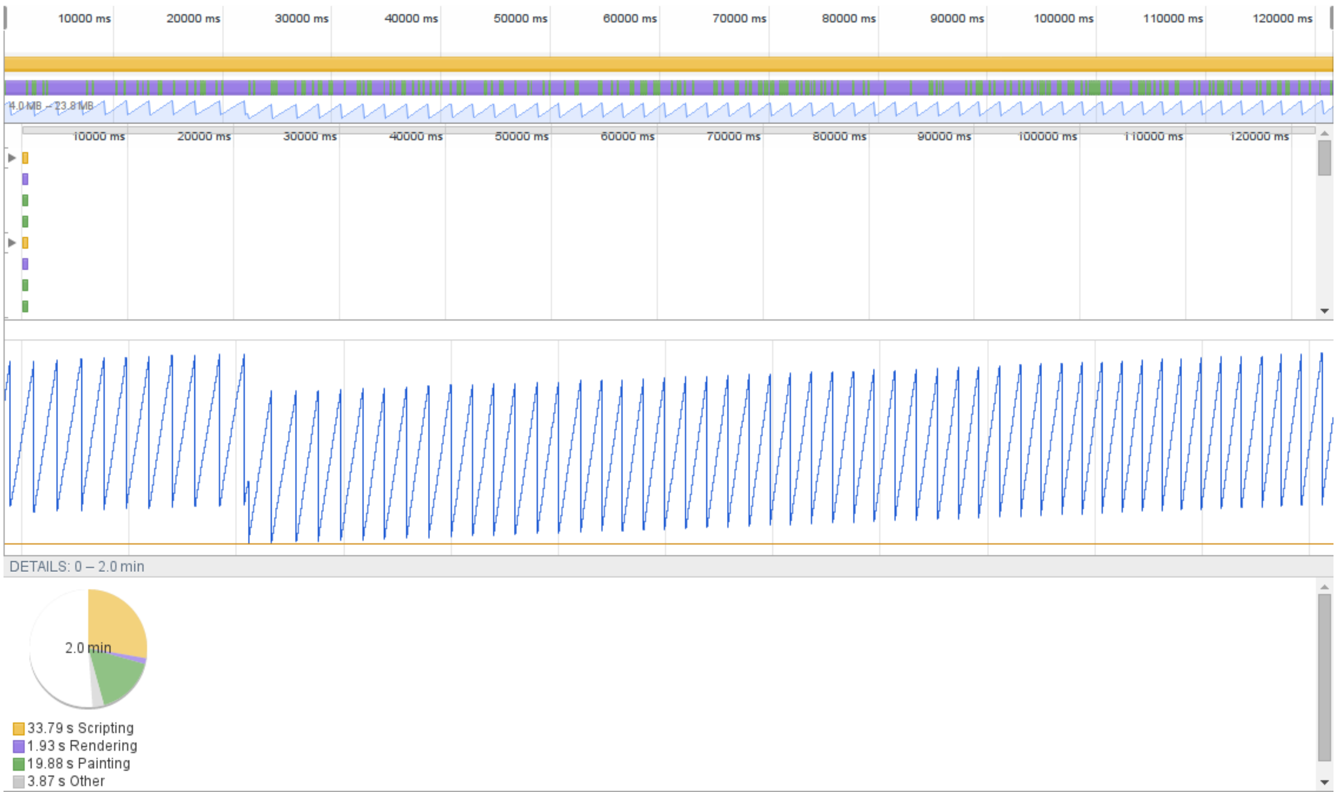
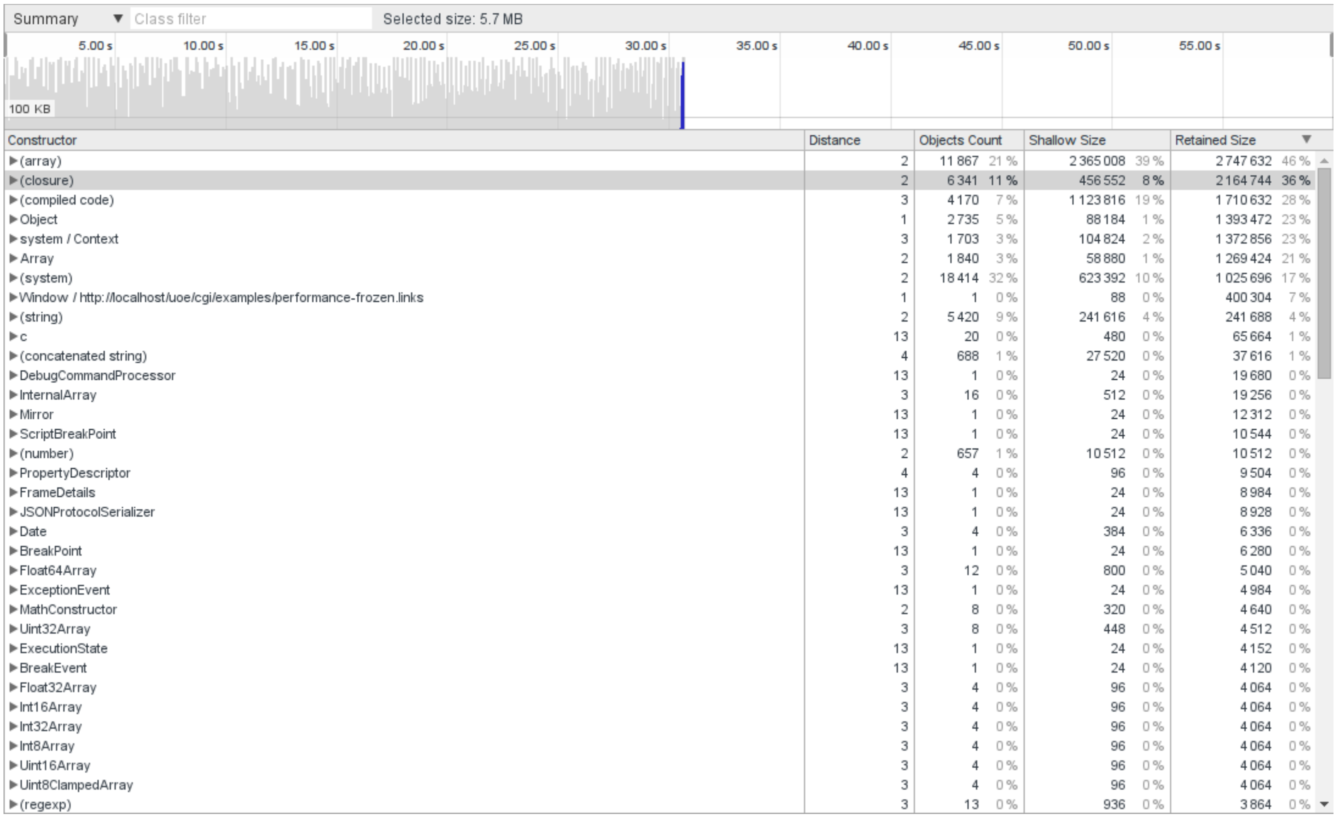


Chart 17: Timeline for the native JavaScript version

We see that much less garbage is being generated (about 4 times less garbage is being collected per GC event). The garbage collector slowdowns have no significant impact on performance in this case.

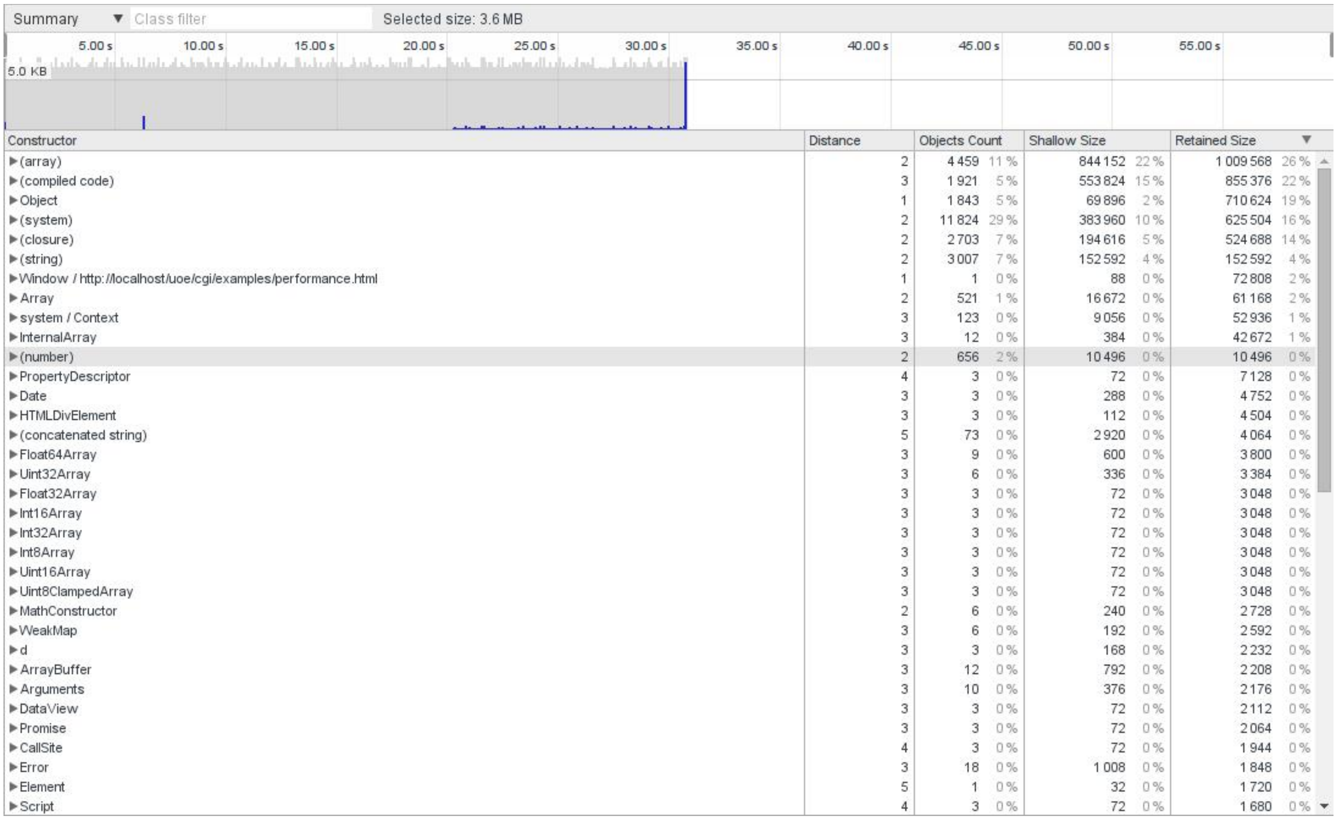
Compare also the heap allocation record:



Constructor	Distance	Objects Count		Shallow Size		Retained Size	
▶(array)	2	11 867	21 %	2 365 008	39 %	2 747 632	46 %
▶(closure)	2	6 341	11 %	456 552	8 %	2 164 744	36 %
▶(compiled code)	3	4 170	7 %	1 123 816	19 %	1 710 632	28 %
▶Object	1	2 735	5 %	88 184	1 %	1 393 472	23 %
▶system / Context	3	1 703	3 %	104 824	2 %	1 372 856	23 %
▶Array	2	1 840	3 %	58 880	1 %	1 269 424	21 %
▶(system)	2	18 414	32 %	623 392	10 %	1 025 696	17 %
▶Window / http://localhost/uoe/cgi/examples/performance-frozen.links	1	1	0 %	88	0 %	400 304	7 %
▶(string)	2	5 420	9 %	241 616	4 %	241 688	4 %
▶c	13	20	0 %	480	0 %	65 664	1 %
▶(concatenated string)	4	688	1 %	27 520	0 %	37 616	1 %
▶DebugCommandProcessor	13	1	0 %	24	0 %	19 680	0 %
▶InternalArray	3	16	0 %	512	0 %	19 256	0 %
▶Mirror	13	1	0 %	24	0 %	12 312	0 %
▶ScriptBreakPoint	13	1	0 %	24	0 %	10 544	0 %
▶(number)	2	657	1 %	10 512	0 %	10 512	0 %
▶PropertyDescriptor	4	4	0 %	96	0 %	9 504	0 %
▶FrameDetails	13	1	0 %	24	0 %	8 984	0 %
▶JSONProtocolSerializer	13	1	0 %	24	0 %	8 928	0 %
▶Date	3	4	0 %	384	0 %	6 336	0 %
▶BreakPoint	13	1	0 %	24	0 %	6 280	0 %
▶Float64Array	3	12	0 %	800	0 %	5 040	0 %
▶ExceptionEvent	13	1	0 %	24	0 %	4 984	0 %
▶MathConstructor	2	8	0 %	320	0 %	4 640	0 %
▶Uint32Array	3	8	0 %	448	0 %	4 512	0 %
▶ExecutionState	13	1	0 %	24	0 %	4 152	0 %
▶BreakEvent	13	1	0 %	24	0 %	4 120	0 %
▶Float32Array	3	4	0 %	96	0 %	4 064	0 %
▶Int16Array	3	4	0 %	96	0 %	4 064	0 %
▶Int32Array	3	4	0 %	96	0 %	4 064	0 %
▶Int8Array	3	4	0 %	96	0 %	4 064	0 %
▶Uint16Array	3	4	0 %	96	0 %	4 064	0 %
▶Uint8ClampedArray	3	4	0 %	96	0 %	4 064	0 %
▶(regexp)	3	13	0 %	936	0 %	3 864	0 %

Chart 18: Unoptimized version – heap allocations

We see that very large amounts of memory are being allocated and collected.



Constructor	Distance	Objects Count		Shallow Size		Retained Size	
▶(array)	2	4 459	11 %	844 152	22 %	1 009 568	26 %
▶(compiled code)	3	1 921	5 %	553 824	15 %	855 376	22 %
▶Object	1	1 843	5 %	69 896	2 %	710 624	19 %
▶(system)	2	11 824	29 %	383 960	10 %	625 504	16 %
▶(closure)	2	2 703	7 %	194 616	5 %	524 688	14 %
▶(string)	2	3 007	7 %	152 592	4 %	152 592	4 %
▶Window / http://localhost/uoe/cgi/examples/performance.html	1	1	0 %	88	0 %	72 808	2 %
▶Array	2	521	1 %	16 672	0 %	61 168	2 %
▶system / Context	3	123	0 %	9 056	0 %	52 936	1 %
▶InternalArray	3	12	0 %	384	0 %	42 672	1 %
▶(number)	2	656	2 %	10 496	0 %	10 496	0 %
▶PropertyDescriptor	4	3	0 %	72	0 %	7 128	0 %
▶Date	3	3	0 %	288	0 %	4 752	0 %
▶HTMLDivElement	3	3	0 %	112	0 %	4 504	0 %
▶(concatenated string)	5	73	0 %	2 920	0 %	4 064	0 %
▶Float64Array	3	9	0 %	600	0 %	3 800	0 %
▶Uint32Array	3	6	0 %	336	0 %	3 384	0 %
▶Float32Array	3	3	0 %	72	0 %	3 048	0 %
▶Int16Array	3	3	0 %	72	0 %	3 048	0 %
▶Int32Array	3	3	0 %	72	0 %	3 048	0 %
▶Int8Array	3	3	0 %	72	0 %	3 048	0 %
▶Uint16Array	3	3	0 %	72	0 %	3 048	0 %
▶Uint8ClampedArray	3	3	0 %	72	0 %	3 048	0 %
▶MathConstructor	2	6	0 %	240	0 %	2 728	0 %
▶WeakMap	3	6	0 %	192	0 %	2 592	0 %
▶d	3	3	0 %	168	0 %	2 232	0 %
▶ArrayBuffer	3	12	0 %	792	0 %	2 208	0 %
▶Arguments	3	10	0 %	376	0 %	2 176	0 %
▶DataView	3	3	0 %	72	0 %	2 112	0 %
▶Promise	3	3	0 %	72	0 %	2 064	0 %
▶CallSite	4	3	0 %	72	0 %	1 944	0 %
▶Error	3	18	0 %	1 008	0 %	1 848	0 %
▶Element	5	1	0 %	32	0 %	1 720	0 %
▶Script	4	3	0 %	72	0 %	1 680	0 %

Chart 19: Native JavaScript version – heap allocations

We see that the size of the heap is much smaller, there's much less memory objects being generated and there are no spikes (the reference line here is at 5 KB and in the previous chart it was at 100 KB and went up to 5 times higher than that so the amount of memory allocated is at times 100 times greater in Links version).

The chart produced by the JavaScript version (note that double buffering is off, because it's unneccessary; also the framerate is limited to 60 FPS – frames are drawn using `requestAnimationFrame`[15]):

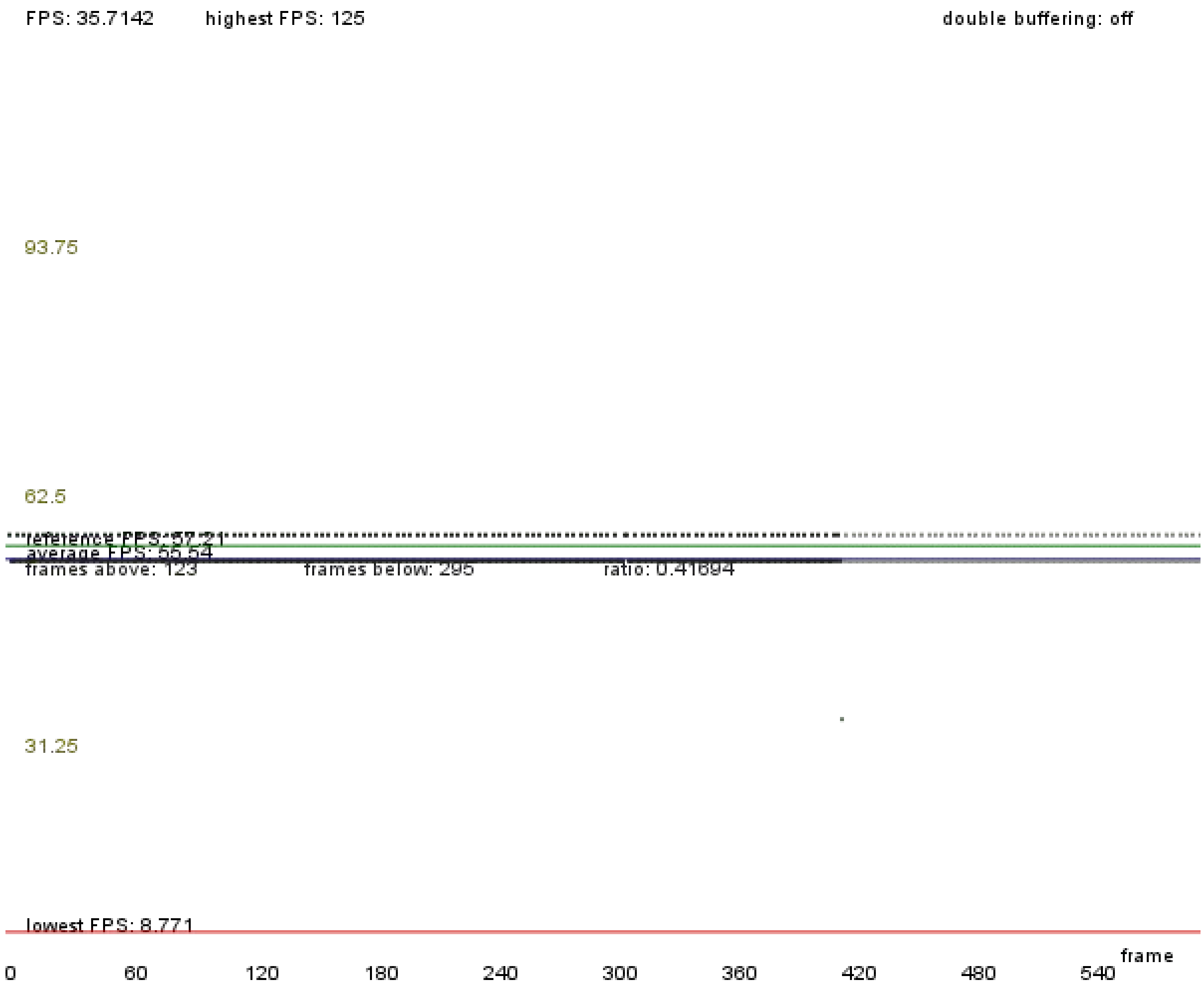


Chart 20: A chart generated by the JavaScript version of the benchmark

We see that in this case the FPS is stable – around 60, as expected.

15 https://developer.mozilla.org/en/docs/Web/API/window.requestAnimationFrame

Profiling

Profiling the execution time of any Links application gives similar results (this is a Firebug output for my Breakout clone):

_yield	72173	12.57%	3520.344ms	441966.667ms	6.124ms	0.069ms	64.442ms
LINKS</LINKS.eq	42105	8.31%	2327.104ms	6600.942ms	0.157ms	0.016ms	29.873ms
DEBUG</<.is_array	65154	7.52%	2104.892ms	3433.801ms	0.053ms	0.049ms	2.932ms
_yieldCont	59419	7.03%	1968.201ms	326011.922ms	5.487ms	0.049ms	64.531ms
___append	72172	5.42%	1516.437ms	1516.437ms	0.021ms	0.018ms	3.805ms
is_instance	65154	4.75%	1328.909ms	1328.909ms	0.02ms	0.019ms	0.531ms
_Concat	38344	4.43%	1239.774ms	2089.944ms	0.055ms	0.049ms	8.437ms
LINKS</LINKS.concat	38344	3.04%	850.171ms	850.171ms	0.022ms	0.019ms	8.385ms
_Cons	23690	2.72%	760.763ms	2051.792ms	0.087ms	0.08ms	8.471ms
concatMap	9767	2.36%	661.824ms	59392.895ms	6.081ms	0.311ms	57.352ms
DEBUG</<.is_unit	34230	2.26%	631.935ms	631.935ms	0.018ms	0.016ms	0.087ms
_tl	28668	2.04%	571.488ms	571.488ms	0.02ms	0.018ms	1.526ms
fold_left	9543	1.9%	531.448ms	50745.051ms	5.318ms	0.137ms	63.218ms
_hd	28668	1.89%	529.918ms	529.918ms	0.018ms	0.017ms	1.017ms
concatMap/<	9385	1.62%	452.956ms	54224.969ms	5.778ms	0.172ms	26.463ms
concatMap/</</</<	9385	1.56%	435.489ms	47863.003ms	5.1ms	0.172ms	34.252ms
map	5051	1.14%	318.988ms	29698.037ms	5.88ms	0.319ms	64.784ms
concatMap/</<	9385	1.14%	318.981ms	53585.898ms	5.71ms	0.106ms	26.394ms
concatMap/</</<	9385	1.13%	315.353ms	56259.066ms	5.995ms	0.106ms	57.473ms
zip	4628	1.11%	311.201ms	64499.496ms	13.937ms	0.527ms	62.428ms
foldStep_1873/</<	4501	1.04%	290.569ms	21938.159ms	4.874ms	0.272ms	17.764ms
zip/</<	4501	1.02%	285.234ms	62251.253ms	13.831ms	0.432ms	62.071ms
map/<	4788	0.83%	232.334ms	25953.619ms	5.421ms	0.173ms	23.582ms

Obviously `_yield` and `_yieldCont` take the most overall time and are called the most often. A lot of list operations means calling a lot of functions that manipulate them, which are also quite costly.

Functions that are potential candidates for optimization:

- `LINKS.eq` – the compiler should make use of type information and generate specialized code for comparing things, istead of just using a general runtime function
- `map` and other list manipulating functions (`take`, `drop`, `zip`, `hd`, `tl`...) – the way lists are implemented (right now they are just JavaScript arrays) should be changed and all functions operating on lists should be adjusted to that more efficient implementation; that would be a major improvement in performance

Other optimizations

Other optimizations I tried out were:

- Using a much (up to 10x) faster implementation of queues[16] – I tested that with `setZeroTimeout` (uses a queue for storing functions to be called) and with `_send` (`!` in Links' syntax) and `recv` (which use queues for process mailboxes), but it turned out to be not a big improvement, because of the generally small size of the queues (this faster implementation shows its advantages when used with bigger queues; for small ones the gain is cancelled out by the overhead);
  - Note: the current implementation of queues based on `unshift-pop` is up to 2 times slower[17] than the implementation based on `push-shift` (at least in Firefox and Opera – in Chromium the `unshift-pop` seems to generally be a bit faster)
- Removing calls to debug functions from various places (like `_send`), which did improve the performance significantly; any calls to debug functions in often used code obviously may add up to a significant slowdown; I mostly removed calls to these functions:
  - `DEBUG.assert`
  - `DEBUG.assert_noisy`
  - `_debug`
  - `_dumpSchedStatus` – a very significant slowdown, unneccessarily executing code containing loops even if not in debug mode, everytime `_send` is called
- Using `Date.now()` instead of `new Date().getTime()` in `_clientTime()`

16 http://code.stephenmorley.org/javascript/queues/

17 http://jsperf.com/queuing-push-shift-vs-unshift-pop

Lists, equality and _yield optimizations

This section describes various other successful optimizations.

Base

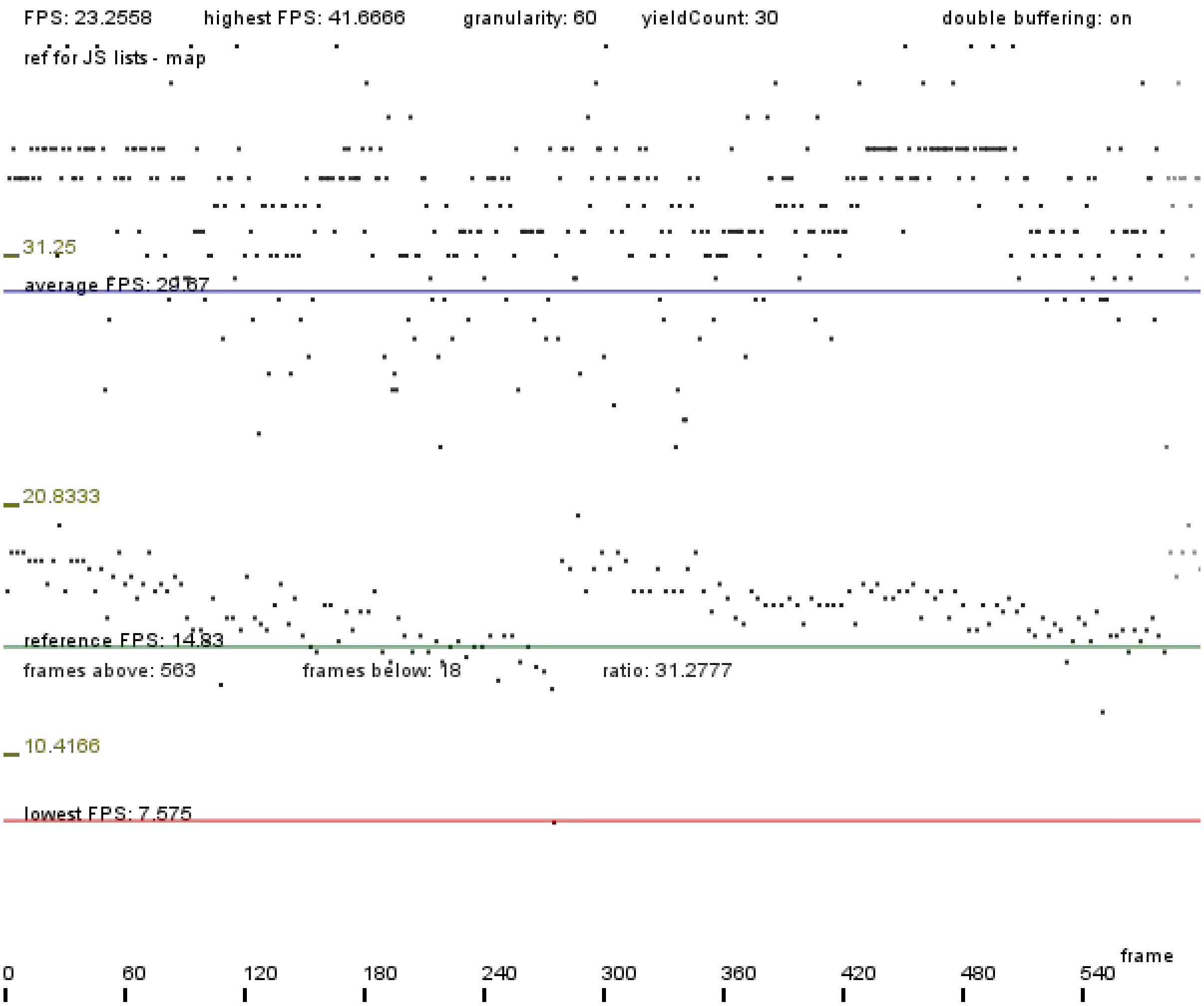


Chart 21: This is the reference for all charts that follow

The reference FPS for all optimizations described in the following sections is 30.

Lists

Linked list type

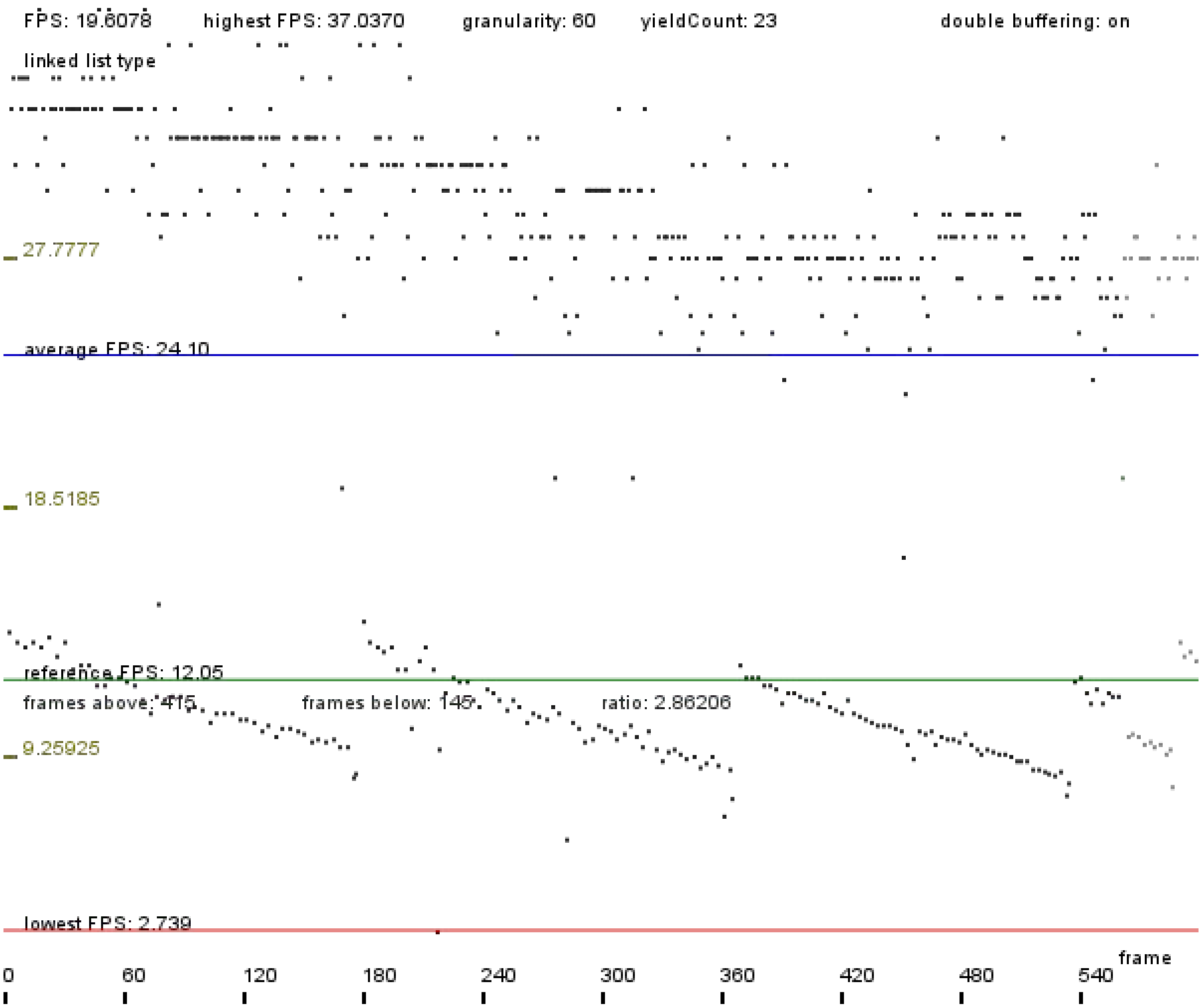


Chart 22: Linked list type

After replacing all Links lists with a custom list type, defined like so:

```
typename Ls(a) = mu l . [| Nil | Cn: (a, l) |];
```

we can see a drop in framerate and a lot more garbage being generated. Looks like the opposite of what we'd expect. There seems to be much more copying. Before I tried investigating how the generated JavaScript looks like and what is the exact reason for this copying, I checked the hypothesis that functions like `take` and `drop`, which might do some unneccessary copying, are problematic here. Then I encountered some strange behaviour (see next section) and moved on to a different approach.

Linked list type with native take and drop

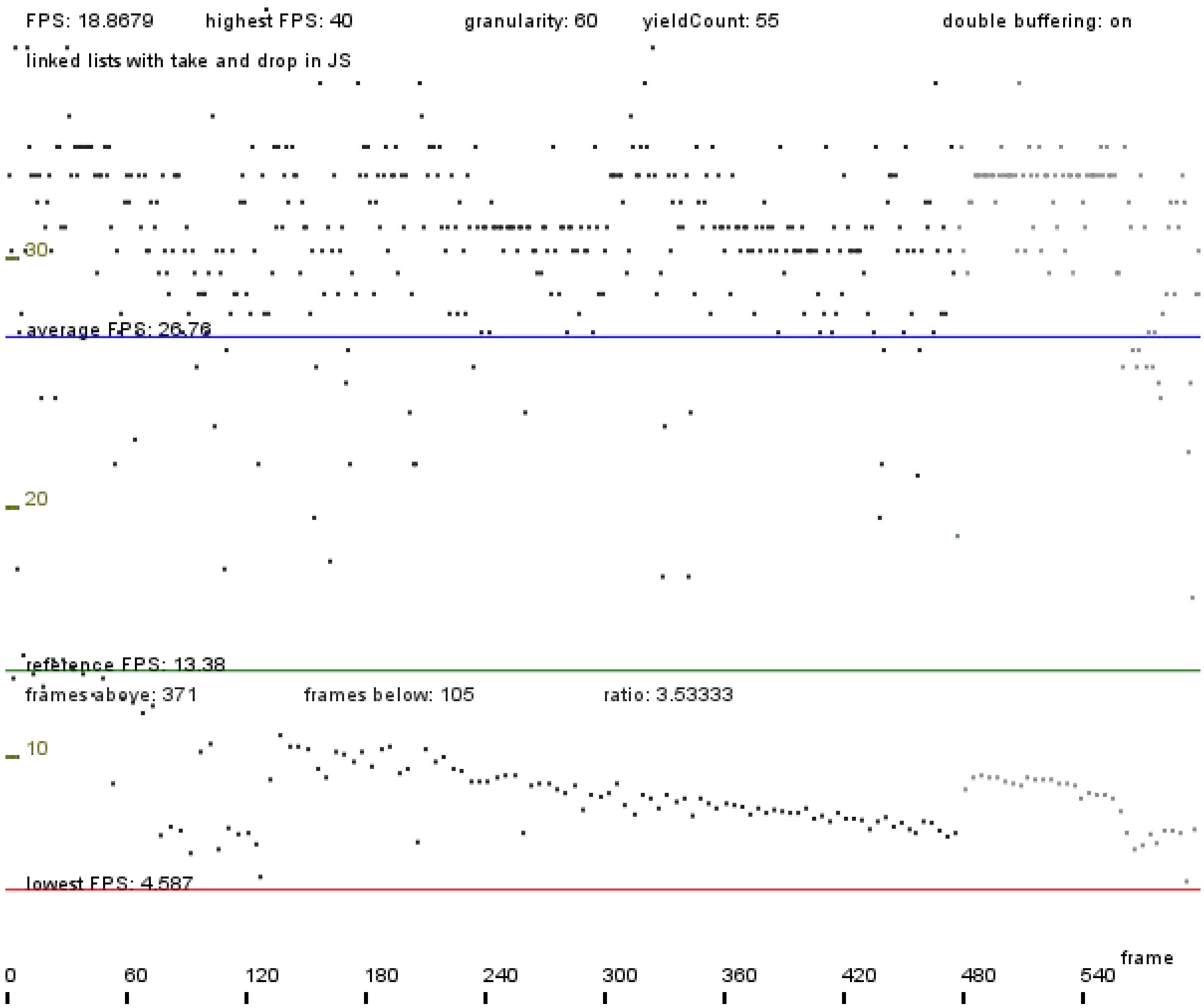


Chart 23: Linked lists with take and drop in JS

The most copying in the benchmark application seems to be caused by `take` and `drop` functions (for the custom list type defined in Links as `lsTake` and `lsDrop`). If we replace them by optimized JavaScript versions, we indeed see some improvement, but still overall the performance decreased in comparison with the reference.

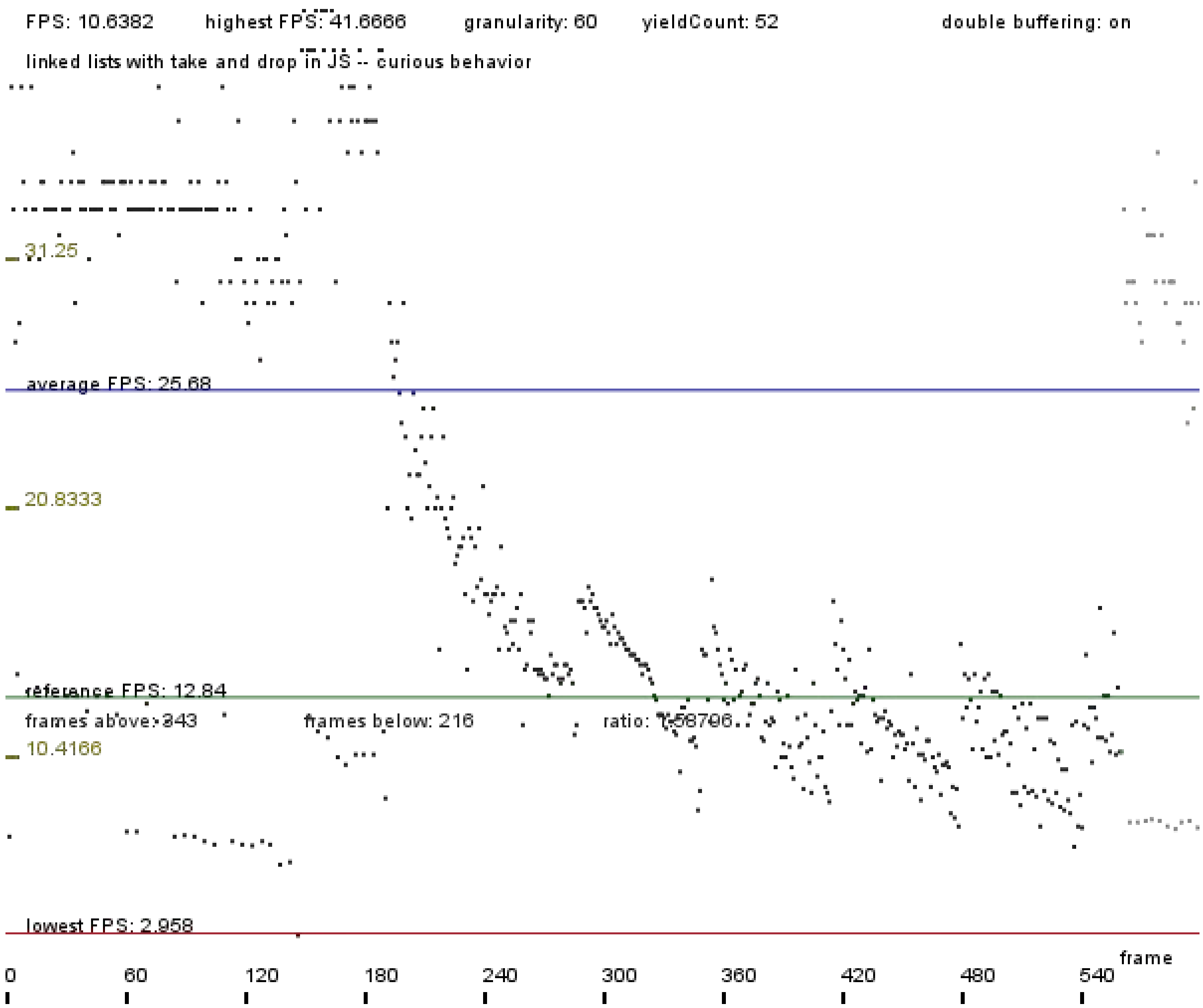


Chart 24: Linked lists with take and drop in JS – curious case

After over 8 iterations of the benchmark I also noticed this strange behaviour. I wasn't able to track down the cause, but I concluded that I will not rely on the list type defined in Links, but instead I'll implement my own, entirely in JavaScript.

JavaScript linked lists

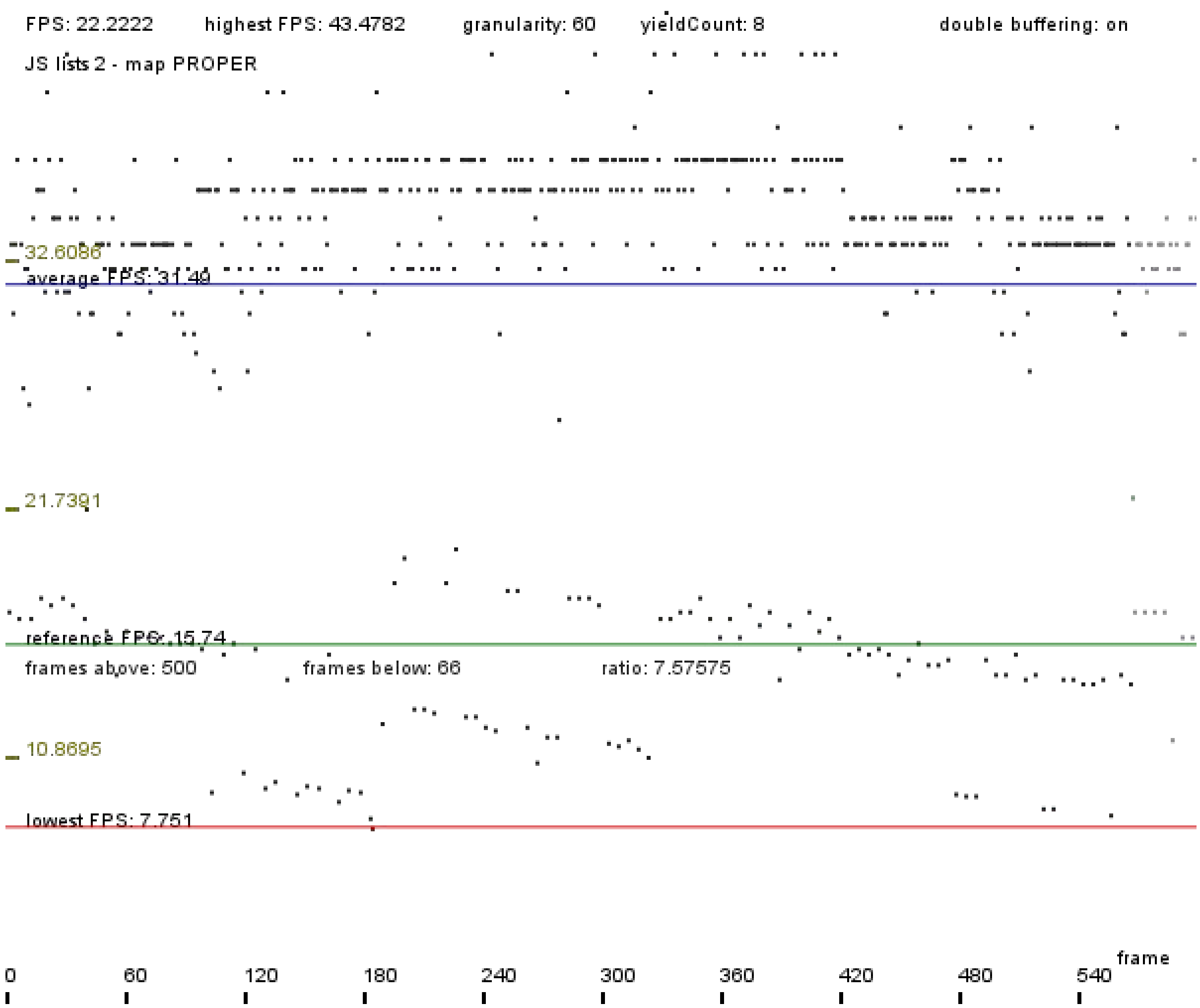


Chart 25: Linked lists in implemented entirely JS (except map)*

After replacing the list type with JavaScript implementation (I implemented all relevant list manipulating functions in JavaScript as well, except `map*`, which was implemented in Links as `lsMap*`), we see a small improvement. The framerate goes up, the garbage collection count decreases a bit.

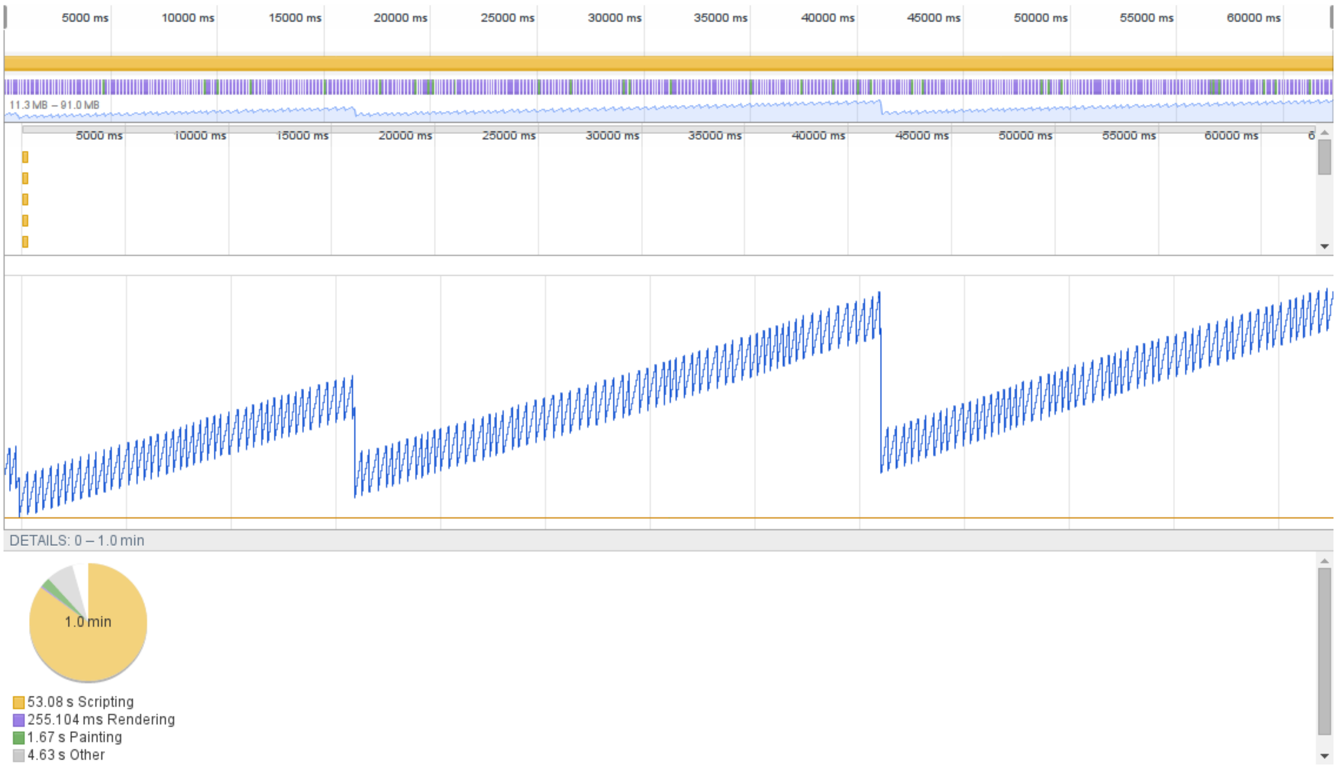
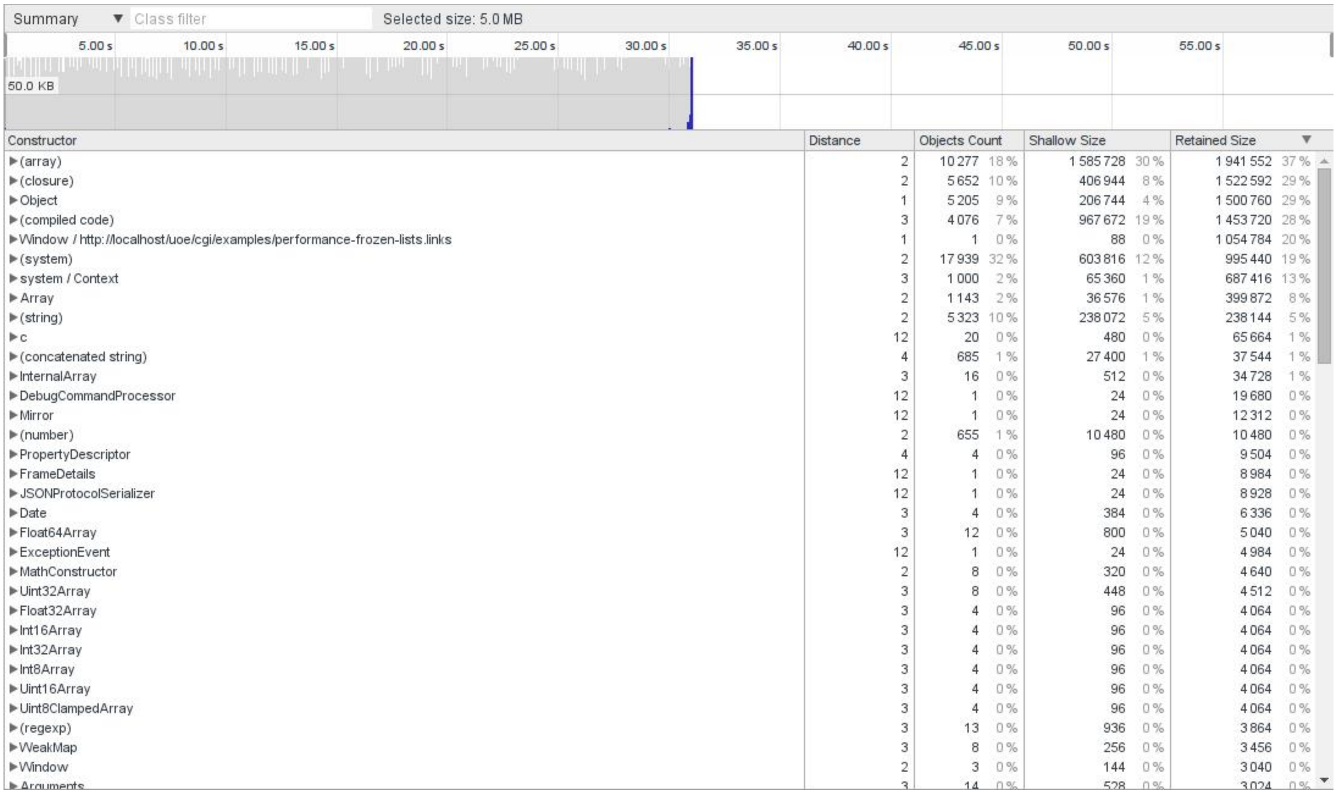


Chart 26: Timeline plot for native JavaScript linked lists – noticeable improvement

We can also see the improvement on the timeline plot – less garbage is being generated.



Constructor	Distance	Objects Count		Shallow Size		Retained Size	
▶(array)	2	10 277	18 %	1 585 728	30 %	1 941 552	37 %
▶(closure)	2	5 652	10 %	406 944	8 %	1 522 592	29 %
▶Object	1	5 205	9 %	206 744	4 %	1 500 760	29 %
▶(compiled code)	3	4 076	7 %	967 672	19 %	1 453 720	28 %
▶Window / http://localhost/uoe/cgi/examples/performance-frozen-lists.links	1	1	0 %	88	0 %	1 054 784	20 %
▶(system)	2	17 939	32 %	603 816	12 %	995 440	19 %
▶system / Context	3	1 000	2 %	65 360	1 %	687 416	13 %
▶Array	2	1 143	2 %	36 576	1 %	399 872	8 %
▶(string)	2	5 323	10 %	238 072	5 %	238 144	5 %
▶c	12	20	0 %	480	0 %	65 664	1 %
▶(concatenated string)	4	685	1 %	27 400	1 %	37 544	1 %
▶InternalArray	3	16	0 %	512	0 %	34 728	1 %
▶DebugCommandProcessor	12	1	0 %	24	0 %	19 680	0 %
▶Mirror	12	1	0 %	24	0 %	12 312	0 %
▶(number)	2	655	1 %	10 480	0 %	10 480	0 %
▶PropertyDescriptor	4	4	0 %	96	0 %	9 504	0 %
▶FrameDetails	12	1	0 %	24	0 %	8 984	0 %
▶JSONProtocolSerializer	12	1	0 %	24	0 %	8 928	0 %
▶Date	3	4	0 %	384	0 %	6 336	0 %
▶Float64Array	3	12	0 %	800	0 %	5 040	0 %
▶ExceptionEvent	12	1	0 %	24	0 %	4 984	0 %
▶MathConstructor	2	8	0 %	320	0 %	4 640	0 %
▶Uint32Array	3	8	0 %	448	0 %	4 512	0 %
▶Float32Array	3	4	0 %	96	0 %	4 064	0 %
▶Int16Array	3	4	0 %	96	0 %	4 064	0 %
▶Int32Array	3	4	0 %	96	0 %	4 064	0 %
▶Int8Array	3	4	0 %	96	0 %	4 064	0 %
▶Uint16Array	3	4	0 %	96	0 %	4 064	0 %
▶Uint8ClampedArray	3	4	0 %	96	0 %	4 064	0 %
▶(regexp)	3	13	0 %	936	0 %	3 864	0 %
▶WeakMap	3	8	0 %	256	0 %	3 456	0 %
▶Window	2	3	0 %	144	0 %	3 040	0 %
▶Arguments	3	14	0 %	528	0 %	3 024	0 %

Chart 27: Native JavaScript linked lists – less heap allocations

Looking at the heap allocation chart we notice that the reference line dropped from 100 KB to 50 KB and the heap size is a bit more stable in time.

JS lists with null

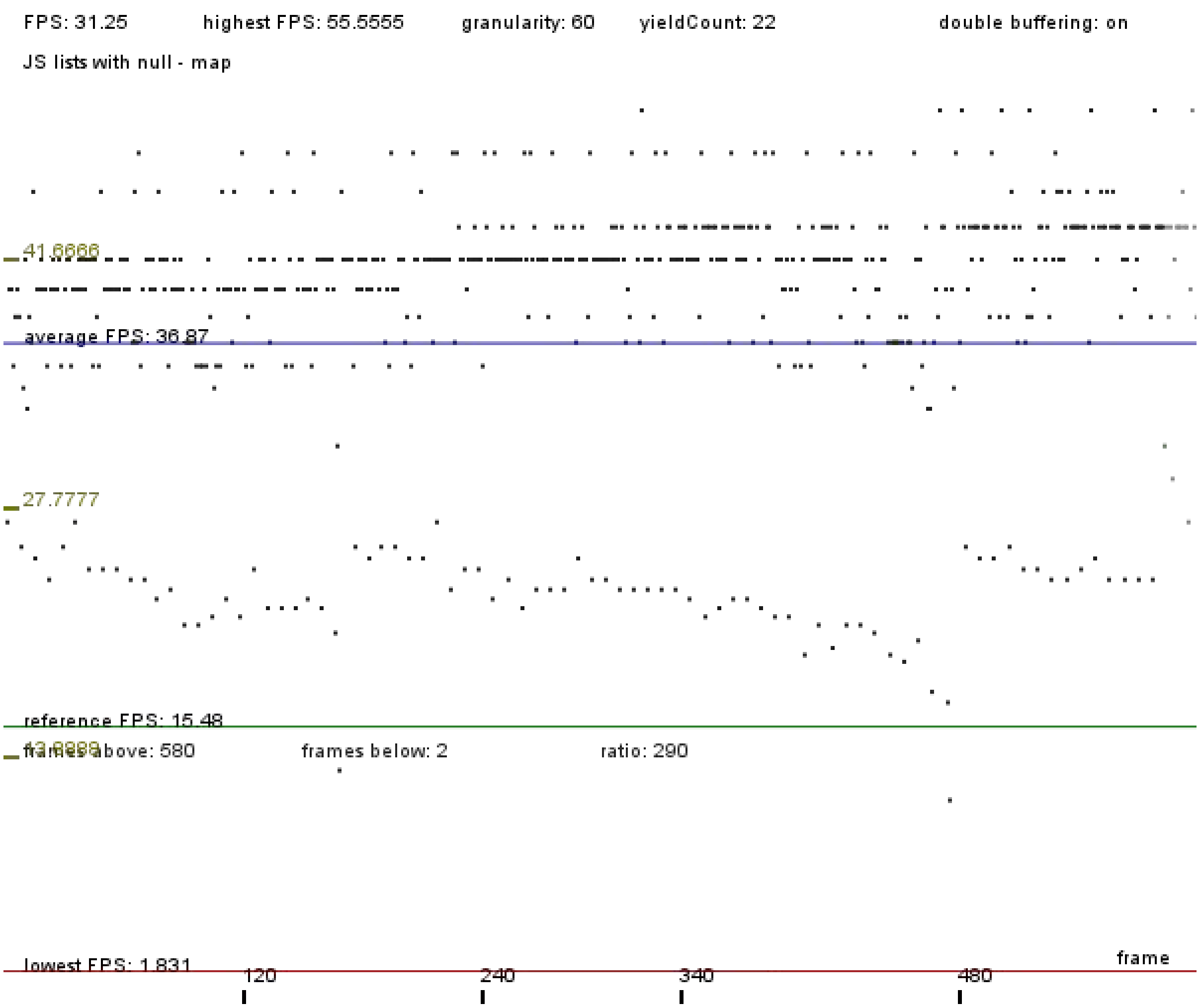


Chart 28: Further optimized native JavaScript linked lists

Simplifying and optimizing the native linked list bumped up the framerate a bit more. The final average FPS for this optimization is 37. There's less garbage collection and the low point of the oscillation went up a bit.

Equality

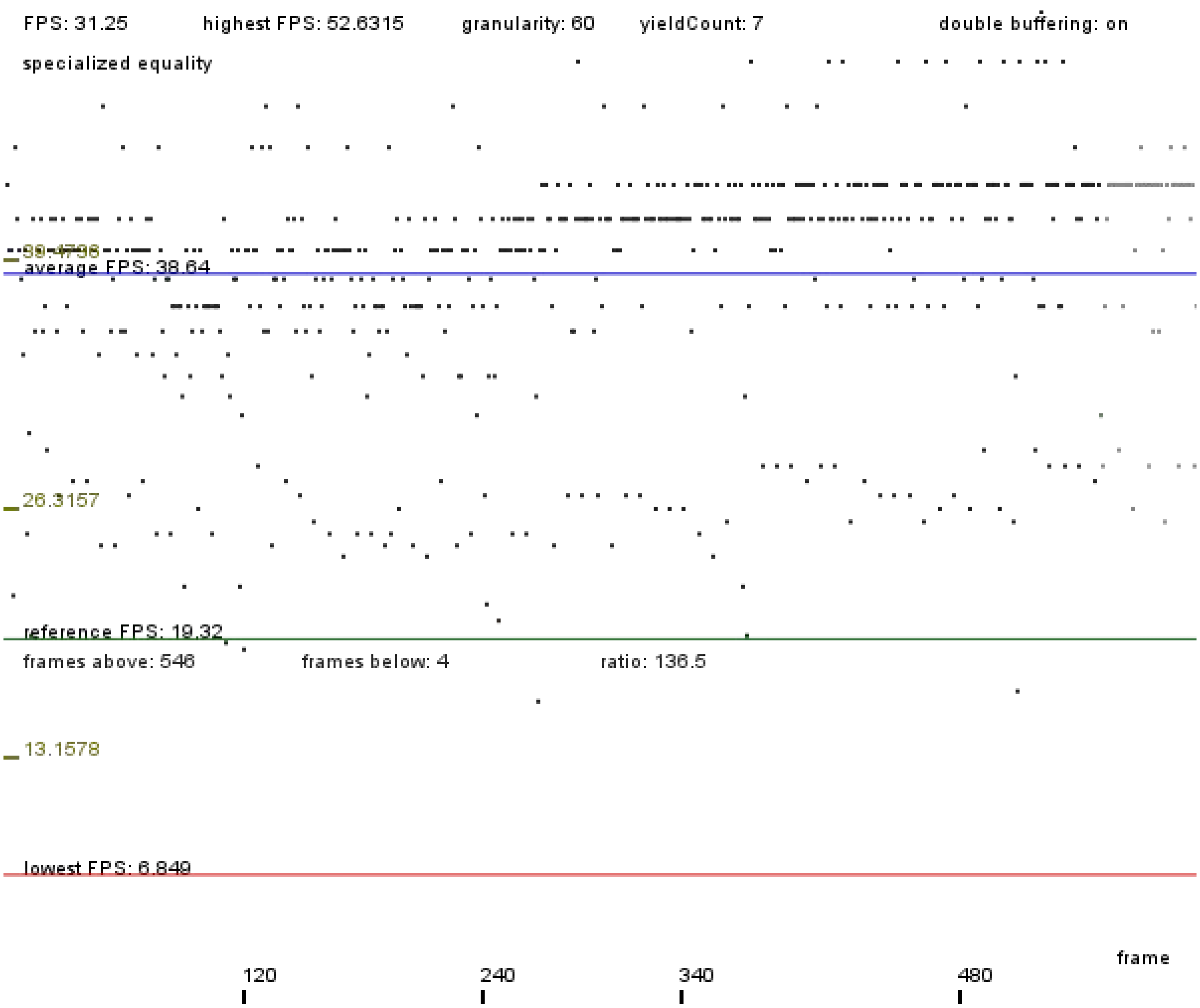


Chart 29: Using specialized functions for equality adds some more frames per second

Another optimization, which was applied on top of the previous one was replacing all the comparisons (==, etc.) in the code of the benchmark application with calls to specialized equality functions (which assume the type of the things being compared) implemented in JavaScript. This was supposed to reduce the overhead of `LINKS.eq`. And indeed, it turned out to be a slight improvement.

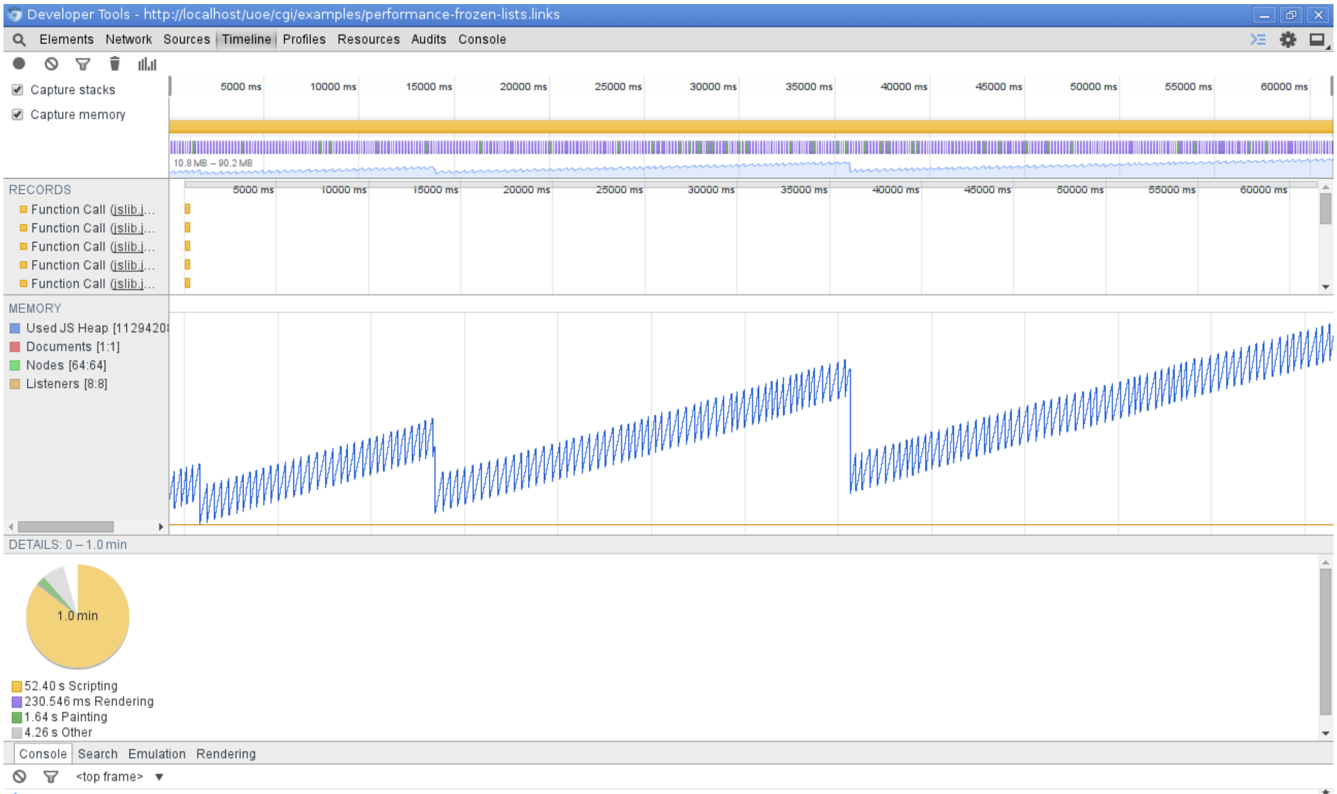


Chart 30: Timeline plot for the optimized native linked lists with specialized equality

The specialized equality functions of course didn't influence the heap size, but the last linked list optimization (for which I didn't show a timeline plot before) did decrease it a tiny bit.

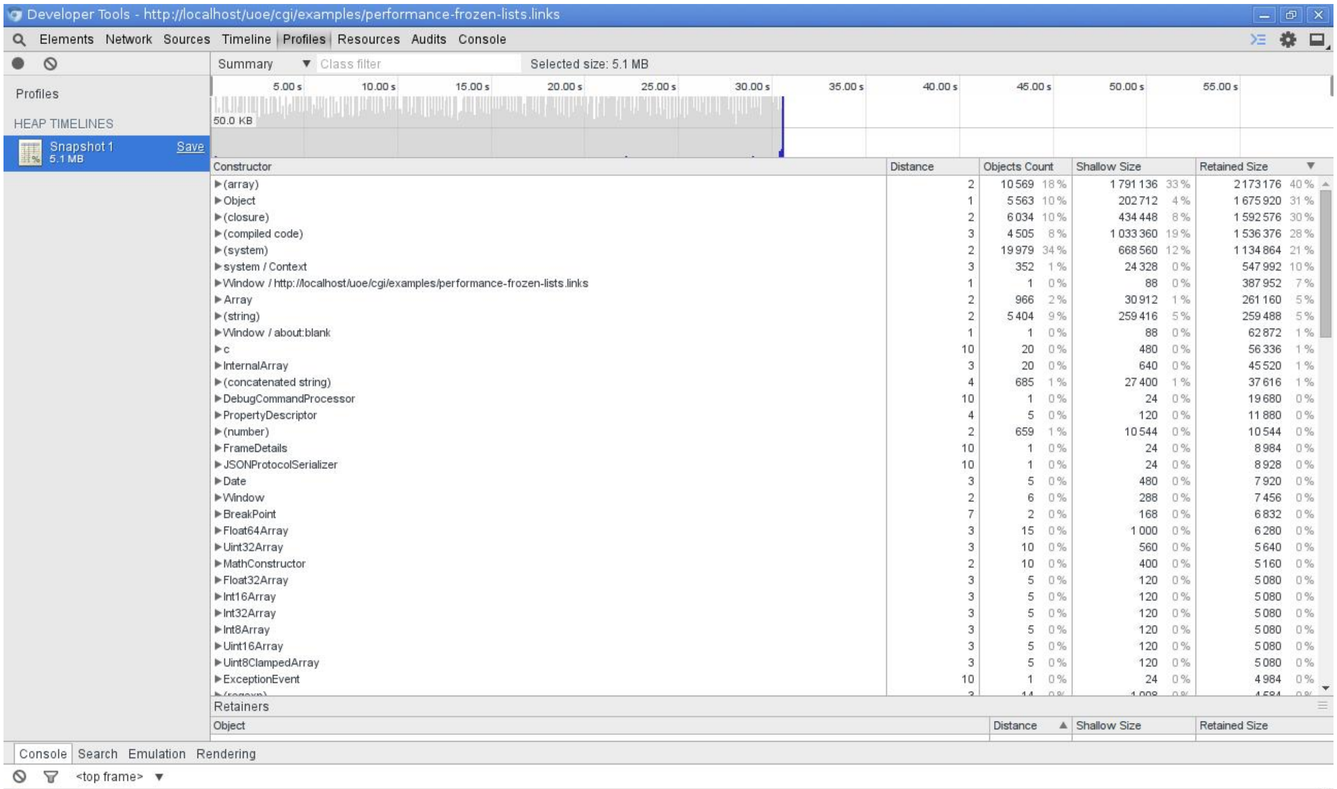
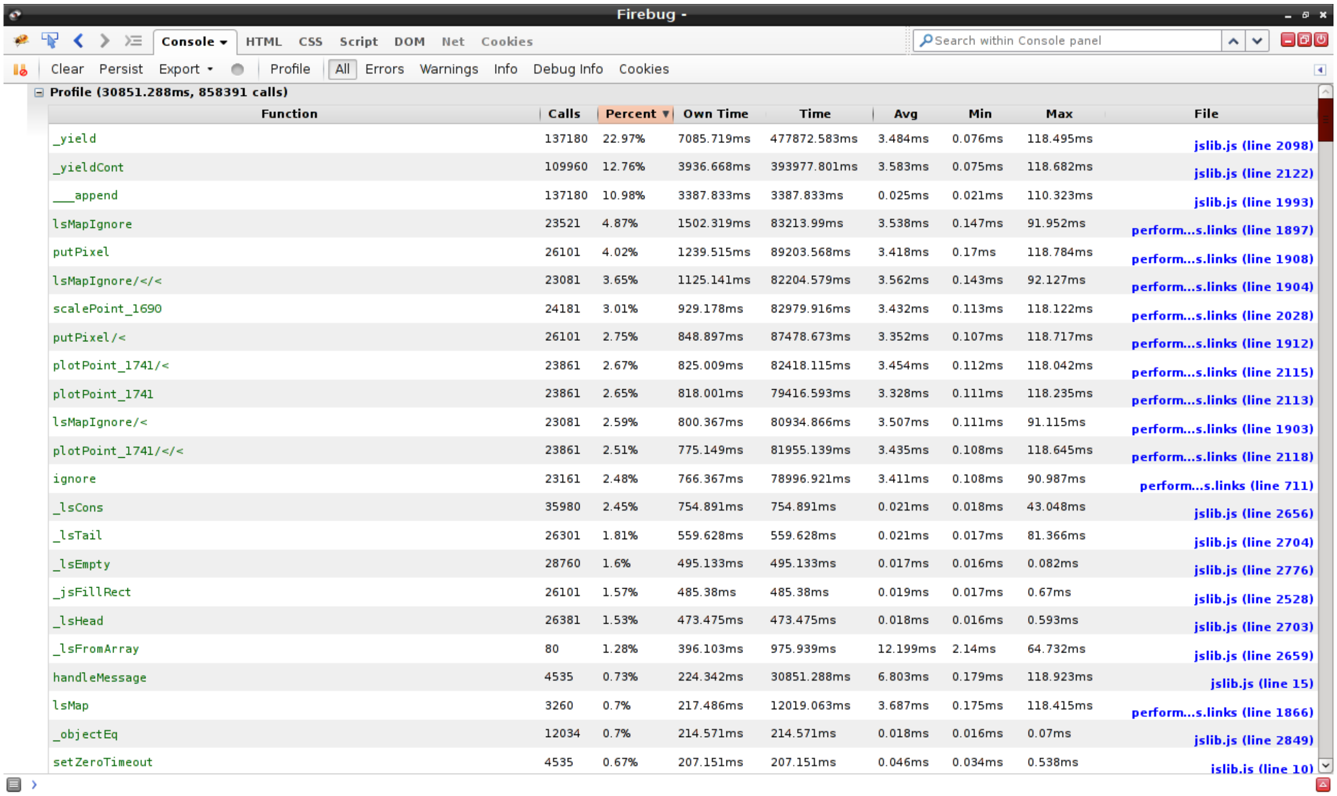


Chart 31: Heap allocations after the specialized equality optimization



Profile (30851.288ms, 858391 calls)

Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
_yield	137180	22.97%	7085.719ms	477872.583ms	3.484ms	0.076ms	118.495ms	jslib.js (line 2098)
_yieldCont	109960	12.76%	3936.668ms	393977.801ms	3.583ms	0.075ms	118.682ms	jslib.js (line 2122)
___append	137180	10.98%	3387.833ms	3387.833ms	0.025ms	0.021ms	110.323ms	jslib.js (line 1993)
lsMapIgnore	23521	4.87%	1502.319ms	83213.99ms	3.538ms	0.147ms	91.952ms	perform...s.links (line 1897)
putPixel	26101	4.02%	1239.515ms	89203.568ms	3.418ms	0.17ms	118.784ms	perform...s.links (line 1908)
lsMapIgnore/</<	23081	3.65%	1125.141ms	82204.579ms	3.562ms	0.143ms	92.127ms	perform...s.links (line 1904)
scalePoint_1690	24181	3.01%	929.178ms	82979.916ms	3.432ms	0.113ms	118.122ms	perform...s.links (line 2028)
putPixel/<	26101	2.75%	848.897ms	87478.673ms	3.352ms	0.107ms	118.717ms	perform...s.links (line 1912)
plotPoint_1741/<	23861	2.67%	825.009ms	82418.115ms	3.454ms	0.112ms	118.042ms	perform...s.links (line 2115)
plotPoint_1741	23861	2.65%	818.001ms	79416.593ms	3.328ms	0.111ms	118.235ms	perform...s.links (line 2113)
lsMapIgnore/<	23081	2.59%	800.367ms	80934.866ms	3.507ms	0.111ms	91.115ms	perform...s.links (line 1903)
plotPoint_1741/</<	23861	2.51%	775.149ms	81955.139ms	3.435ms	0.108ms	118.645ms	perform...s.links (line 2118)
ignore	23161	2.48%	766.367ms	78996.921ms	3.411ms	0.108ms	90.987ms	perform...s.links (line 711)
_lsCons	35980	2.45%	754.891ms	754.891ms	0.021ms	0.018ms	43.048ms	jslib.js (line 2656)
_lsTail	26301	1.81%	559.628ms	559.628ms	0.021ms	0.017ms	81.366ms	jslib.js (line 2704)
_lsEmpty	28760	1.6%	495.133ms	495.133ms	0.017ms	0.016ms	0.082ms	jslib.js (line 2776)
_jsFillRect	26101	1.57%	485.38ms	485.38ms	0.019ms	0.017ms	0.67ms	jslib.js (line 2528)
_lsHead	26381	1.53%	473.475ms	473.475ms	0.018ms	0.016ms	0.593ms	jslib.js (line 2703)
_lsFromArray	80	1.28%	396.103ms	975.939ms	12.199ms	2.14ms	64.732ms	jslib.js (line 2659)
handleMessage	4535	0.73%	224.342ms	30851.288ms	6.803ms	0.179ms	118.923ms	jslib.js (line 15)
lsMap	3260	0.7%	217.486ms	12019.063ms	3.687ms	0.175ms	118.415ms	perform...s.links (line 1866)
_objectEq	12034	0.7%	214.571ms	214.571ms	0.018ms	0.016ms	0.07ms	jslib.js (line 2849)
setZeroTimeout	4535	0.67%	207.151ms	207.151ms	0.046ms	0.034ms	0.538ms	jslib.js (line 10)

Chart 32: The time that all the calls to `LINKS.eq` took was slightly reduced by using specialized functions for comparison

Interesting to look at is also the Firebug plot showing execution time of the different functions. We see that obviously comparing objects (`_objectEq`) took the most time (the rest of the comparison functions are not in the picture, they took much less time). Also we see that `lsMapIgnore`, which is a custom native JavaScript function that works like map, but is only interested in side effects, so it doesn't have to construct and return a list, was a significant optimization. As were all the other list manipulating functions (`_ls*`).

_yield

The last and the most significant optimization was to the `_yield` function. It removed calls to `__append` and `apply` and replaced the three arguments to `_yield` with a single lambda argument. This optimization required modifying the code generated by the compiler (a few lines in *irtojs.ml* had to be added or adjusted). It's interesting to compare the charts generated by Chromium and Firefox – we can see the difference between garbage collectors in these browsers.

All in all the oscillation gap shrinked significantly and an average framerate of over 50 FPS was achieved. This optimization was applied on top of all the previous ones.

Chromium

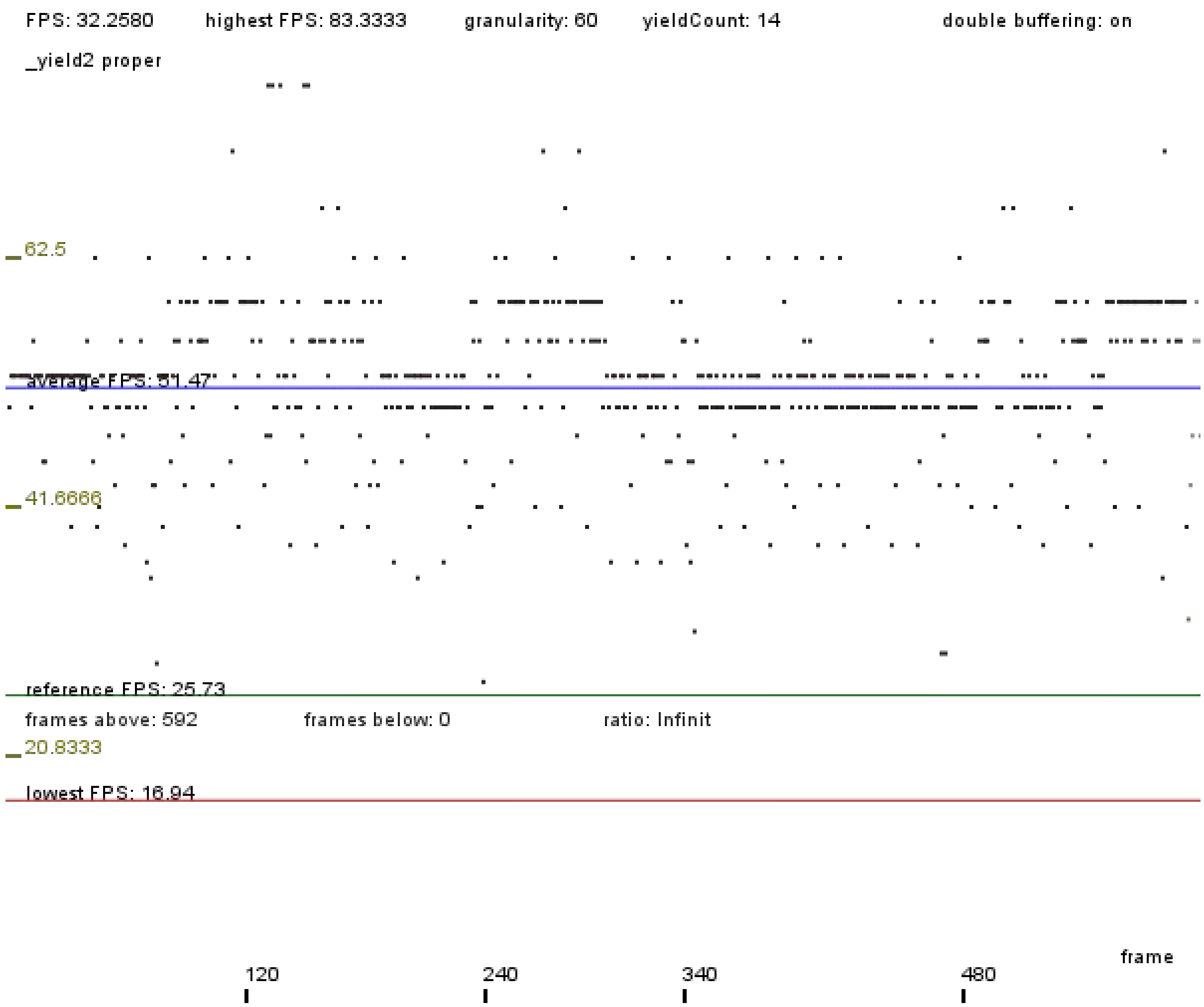


Chart 33: `_yield` optimization in Chromium

Firefox

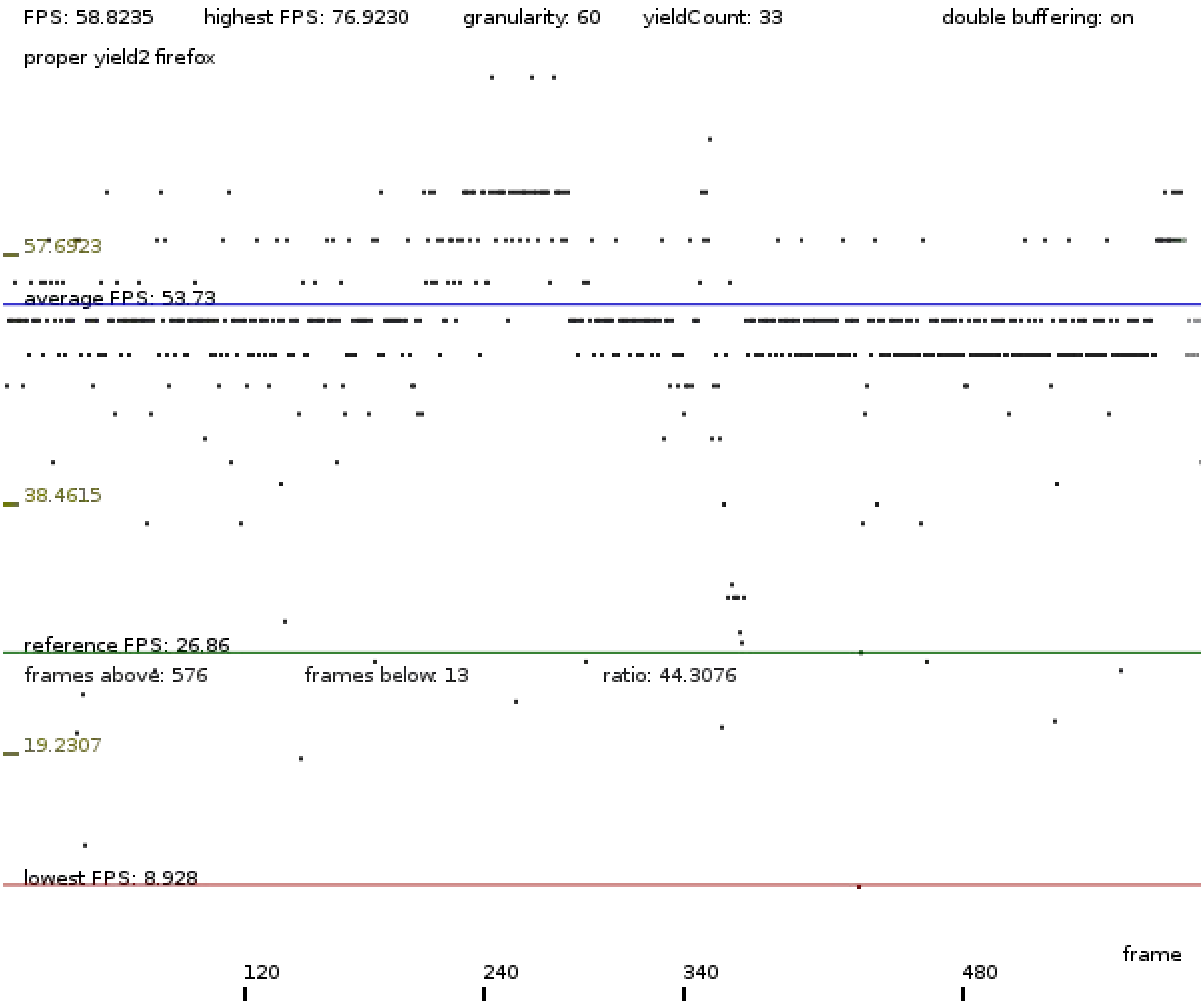


Chart 34: `_yield` optimization in Firefox

Profiling

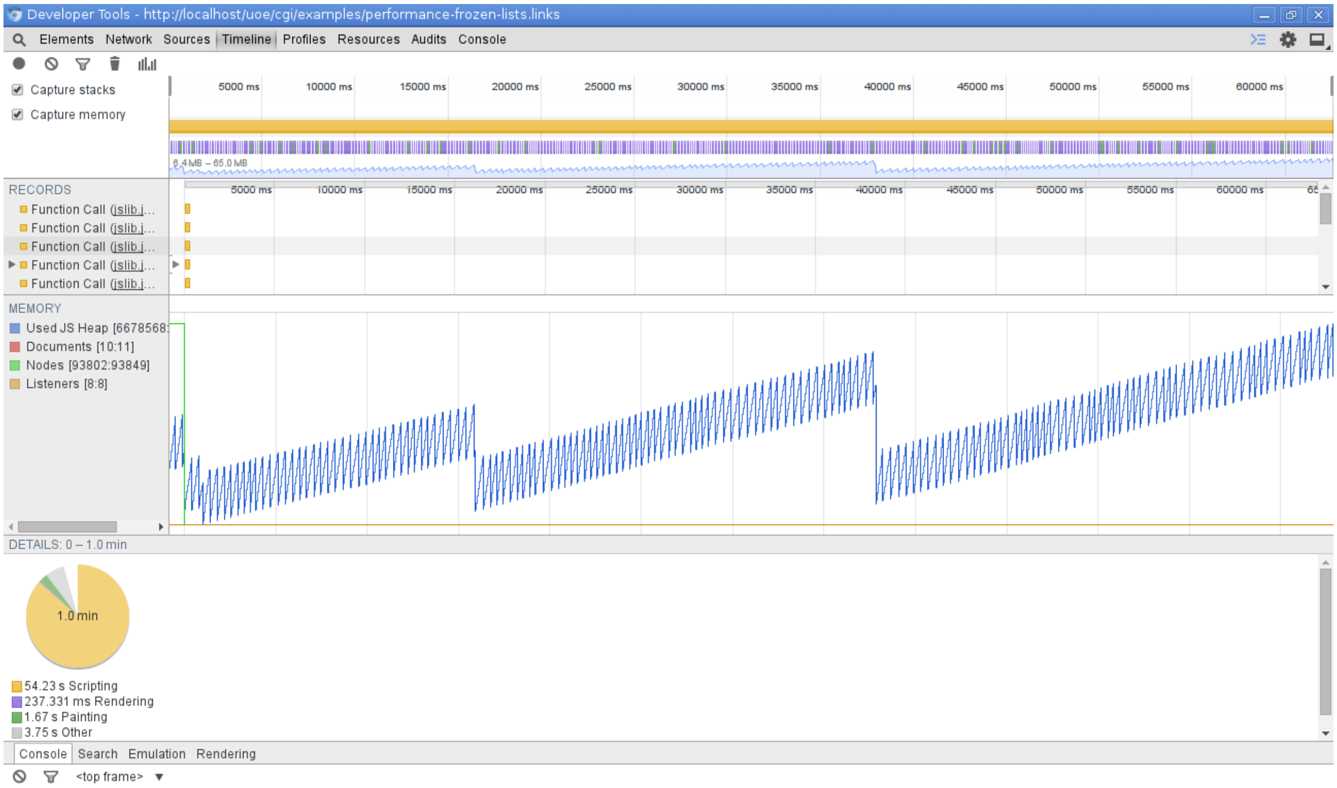


Chart 35: Timeline plot for the `_yield` optimization

The size of the heap in time decreased significantly.

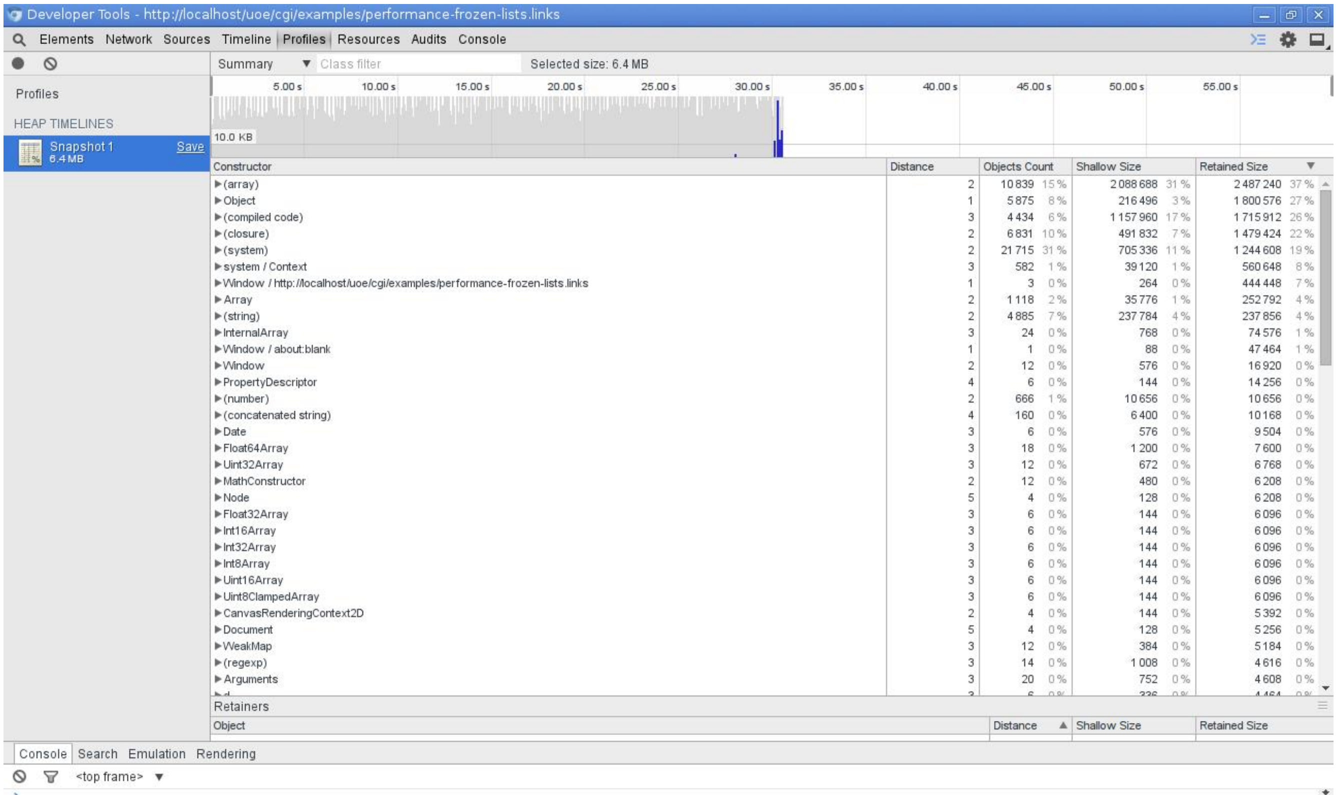
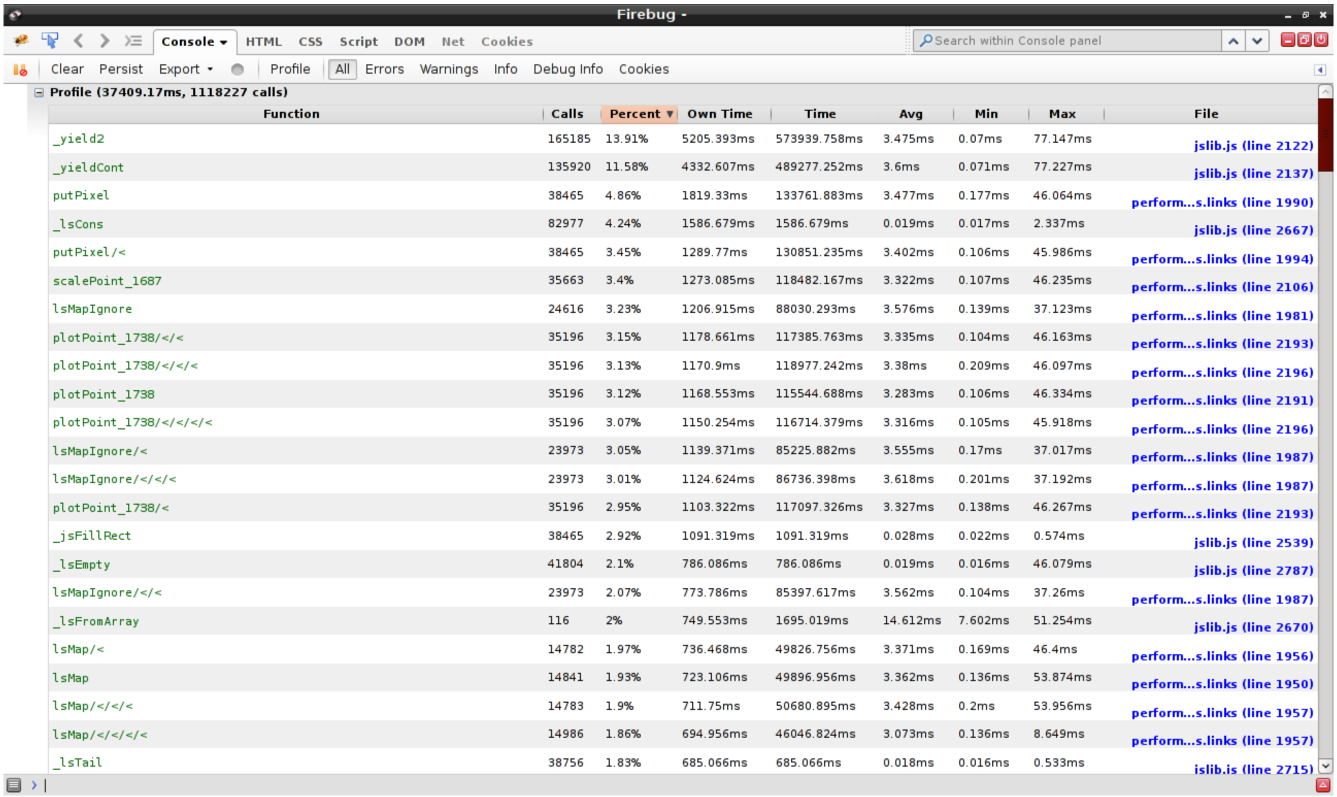


Chart 36: Heap allocations after the `_yield` optimization

We see that the reference size dropped further, from 50 KB to 10 KB.



Profile (37409.17ms, 1118227 calls)

Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
_yield2	165185	13.91%	5205.393ms	573939.758ms	3.475ms	0.07ms	77.147ms	jslib.js (line 2122)
_yieldCont	135920	11.58%	4332.607ms	489277.252ms	3.6ms	0.071ms	77.227ms	jslib.js (line 2137)
putPixel	38465	4.86%	1819.33ms	133761.883ms	3.477ms	0.177ms	46.064ms	perform...s.links (line 1990)
_lsCons	82977	4.24%	1586.679ms	1586.679ms	0.019ms	0.017ms	2.337ms	jslib.js (line 2667)
putPixel/<	38465	3.45%	1289.77ms	130851.235ms	3.402ms	0.106ms	45.986ms	perform...s.links (line 1994)
scalePoint_1687	35663	3.4%	1273.085ms	118482.167ms	3.322ms	0.107ms	46.235ms	perform...s.links (line 2106)
lsMapIgnore	24616	3.23%	1206.915ms	88030.293ms	3.576ms	0.139ms	37.123ms	perform...s.links (line 1981)
plotPoint_1738/</<	35196	3.15%	1178.661ms	117385.763ms	3.335ms	0.104ms	46.163ms	perform...s.links (line 2193)
plotPoint_1738/</</<	35196	3.13%	1170.9ms	118977.242ms	3.38ms	0.209ms	46.097ms	perform...s.links (line 2196)
plotPoint_1738	35196	3.12%	1168.553ms	115544.688ms	3.283ms	0.106ms	46.334ms	perform...s.links (line 2191)
plotPoint_1738/</</</<	35196	3.07%	1150.254ms	116714.379ms	3.316ms	0.105ms	45.918ms	perform...s.links (line 2196)
lsMapIgnore/<	23973	3.05%	1139.371ms	85225.882ms	3.555ms	0.17ms	37.017ms	perform...s.links (line 1987)
lsMapIgnore/</</<	23973	3.01%	1124.624ms	86736.398ms	3.618ms	0.201ms	37.192ms	perform...s.links (line 1987)
plotPoint_1738/<	35196	2.95%	1103.322ms	117097.326ms	3.327ms	0.138ms	46.267ms	perform...s.links (line 2193)
_jsFillRect	38465	2.92%	1091.319ms	1091.319ms	0.028ms	0.022ms	0.574ms	jslib.js (line 2539)
_lsEmpty	41804	2.1%	786.086ms	786.086ms	0.019ms	0.016ms	46.079ms	jslib.js (line 2787)
lsMapIgnore/</<	23973	2.07%	773.786ms	85397.617ms	3.562ms	0.104ms	37.26ms	perform...s.links (line 1987)
_lsFromArray	116	2%	749.553ms	1695.019ms	14.612ms	7.602ms	51.254ms	jslib.js (line 2670)
lsMap/<	14782	1.97%	736.468ms	49826.756ms	3.371ms	0.169ms	46.4ms	perform...s.links (line 1956)
lsMap	14841	1.93%	723.106ms	49896.956ms	3.362ms	0.136ms	53.874ms	perform...s.links (line 1950)
lsMap/</</<	14783	1.9%	711.75ms	50680.895ms	3.428ms	0.2ms	53.956ms	perform...s.links (line 1957)
lsMap/</</</<	14986	1.86%	694.956ms	46046.824ms	3.073ms	0.136ms	8.649ms	perform...s.links (line 1957)
_lsTail	38756	1.83%	685.066ms	685.066ms	0.018ms	0.016ms	0.533ms	jslib.js (line 2715)

Chart 37: Execution time after `_yield` optimization

Here, `_yield2` is the optimized version of `_yield`. Average execution time is 3.475 ms, compared to 6.124 ms before – almost twofold improvement, which is reflected in the framerate.

The final framerate, after all optimizations is 51 FPS. 34 times more than in the beginning. All these optimizations are pretty basic and there's room for a lot more.

Conclusions

For a summary of the document see the Gist section. For sources see the footnotes scattered around the pages of this document.

Suggestions for future work

- We see that a lot of the garbage was generated because of unneccessary copying of JavaScript arrays (which are used to represent lists). The optimized linked list type, which reduces that siginficantly should be polished and adapted to the language
- Specialized equality functions seem to improve the performance a bit as well, so they should replace `LINKS.eq`
- Another optimization to try out would be finding ways to reduce the frequency of context switching
- A lot of things I say here are from game development perspective, but anything relevant to that is applicable in other domains where performance is important
- For a smooth interaction with a game we should aim for a stable frame rate of at least 30 FPS (preferably 60 – standard in games)
- Good place to look for ready-made solutions are other languages, similar to Links, like Elm[18] and Opa[19]
- A handy thing for testing and comparing the performance of various JavaScript constructs is jsPerf[20]

18 http://elm-lang.org/
19 http://opalang.org/
20 http://jsperf.com/