Optimizing Links for performance hungry applications

University of Edinburgh

20/Aug/2014 (version 1)
29/Aug/2014 (version 2)

Dariusz Jędrzejczak
dariusz.jedrzejczak.work@gmail.com

version 2
[final draft]

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.

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 as well as the problem described above:

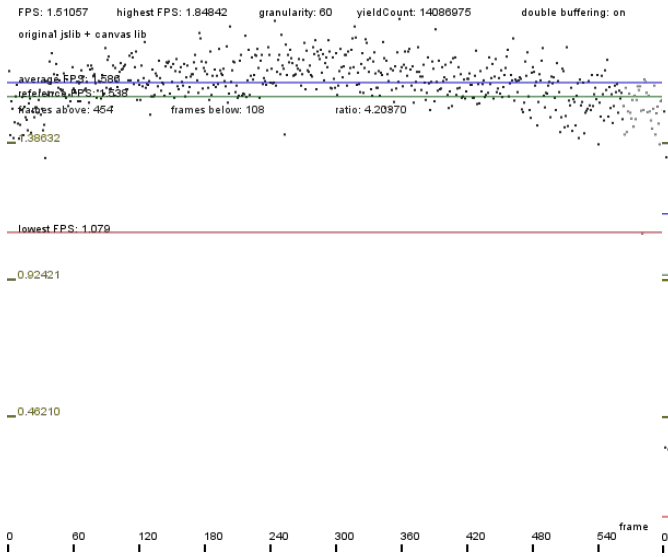


Chart 1: Before: average FPS = 1.5

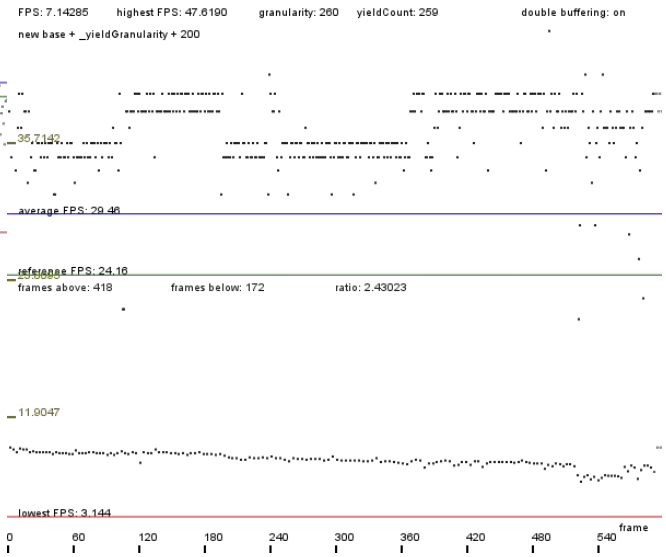


Chart 2: After: average FPS = 29

Contents

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 some stuff 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.

I generated over 100 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.

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 Links runtime (*jslib.js*)[3] are attached to this document as the following files:

- **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*[4]) 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`[5]
- **_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**

3 The names of the attached files approximately correspond to descriptions (if present) found on charts

4 https://github.com/slindley/links/compare/dariusz

5 Implementation from http://dbaron.org/log/20100309-faster-timeouts

Various versions of the benchmark application:

- **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[6]
- **performance.html** – native JavaScript version of the benchmark application
- **performance2.links** – a working/improved version of *performance-frozen.links*. Not up to date

Other files attached to this document:

- **lib.ml** – the original *lib.ml* + interface for canvas manipulation

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

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[7] is on or off[8]
  - description – the second line from the top describes the chart[6]

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

8 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[9]. The green line on every chart may be helpful in estimating the *n*.

The unoptimized version

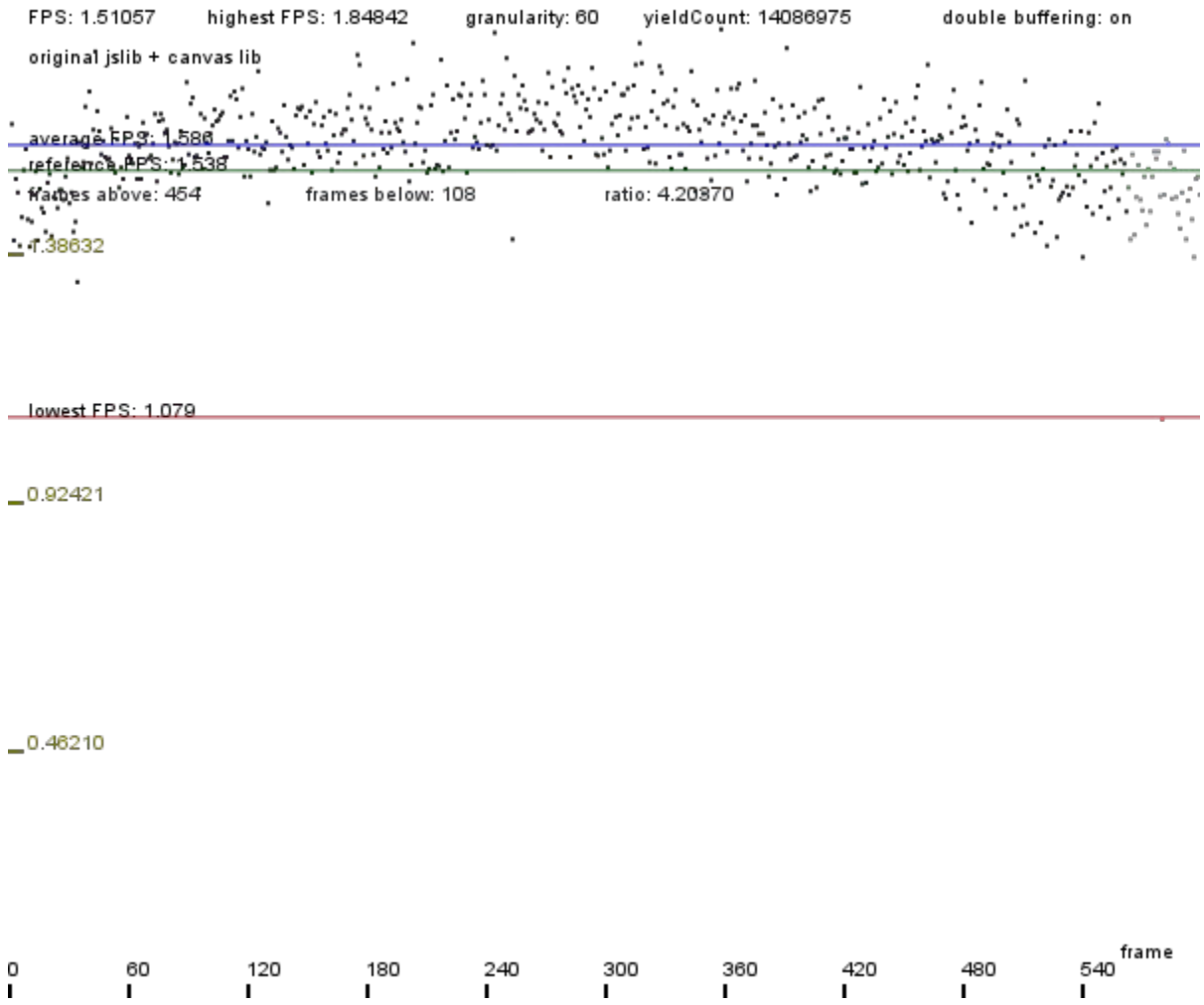


Chart 3: The unoptimized version

Average FPS ~ 1.6. This is the reference.

9 See the section: Garbage collector's behavior.

First optimization – optimized `_yield` and `_yieldCont`

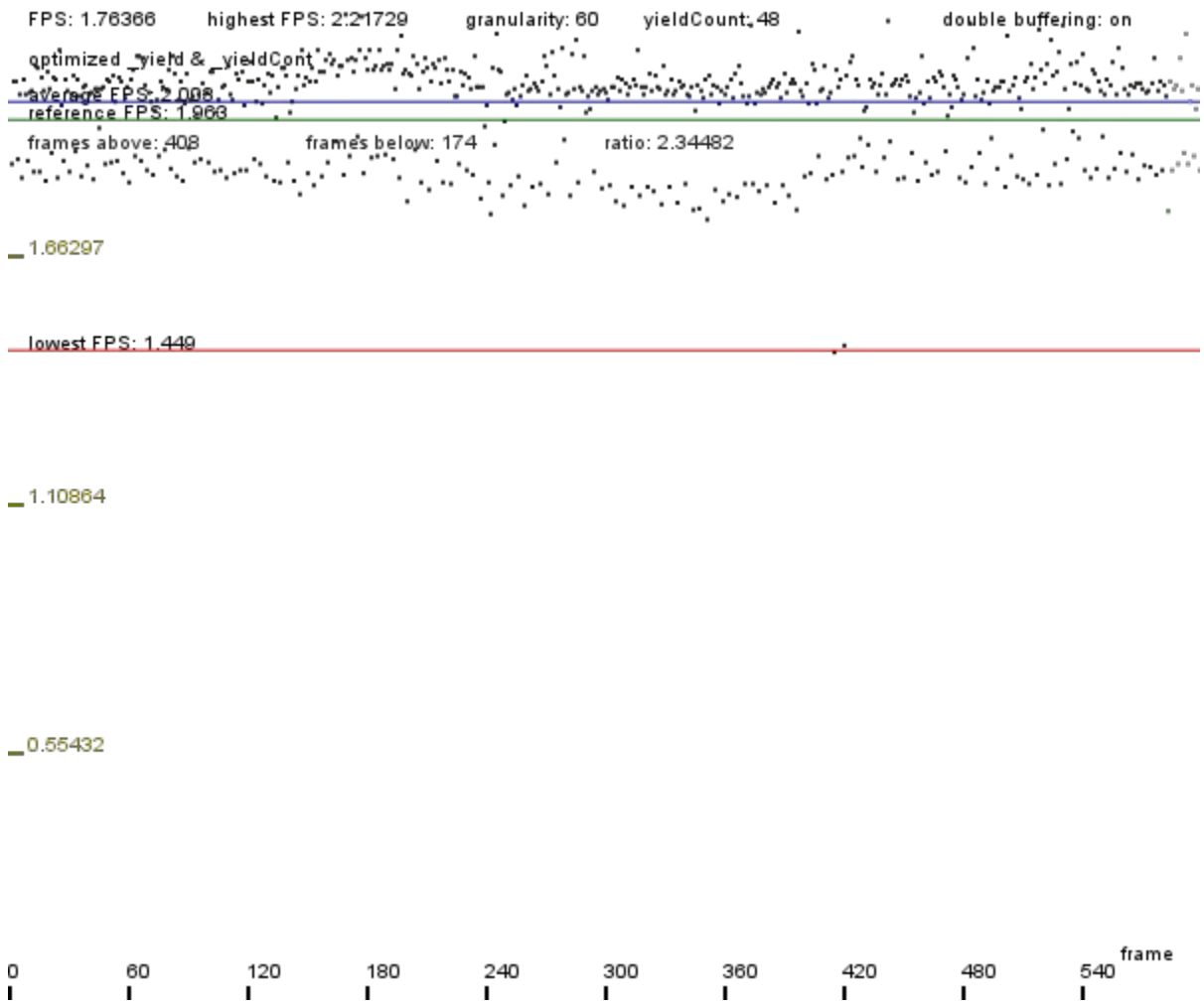


*Chart 4: First optimization – faster `_yield`**

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`

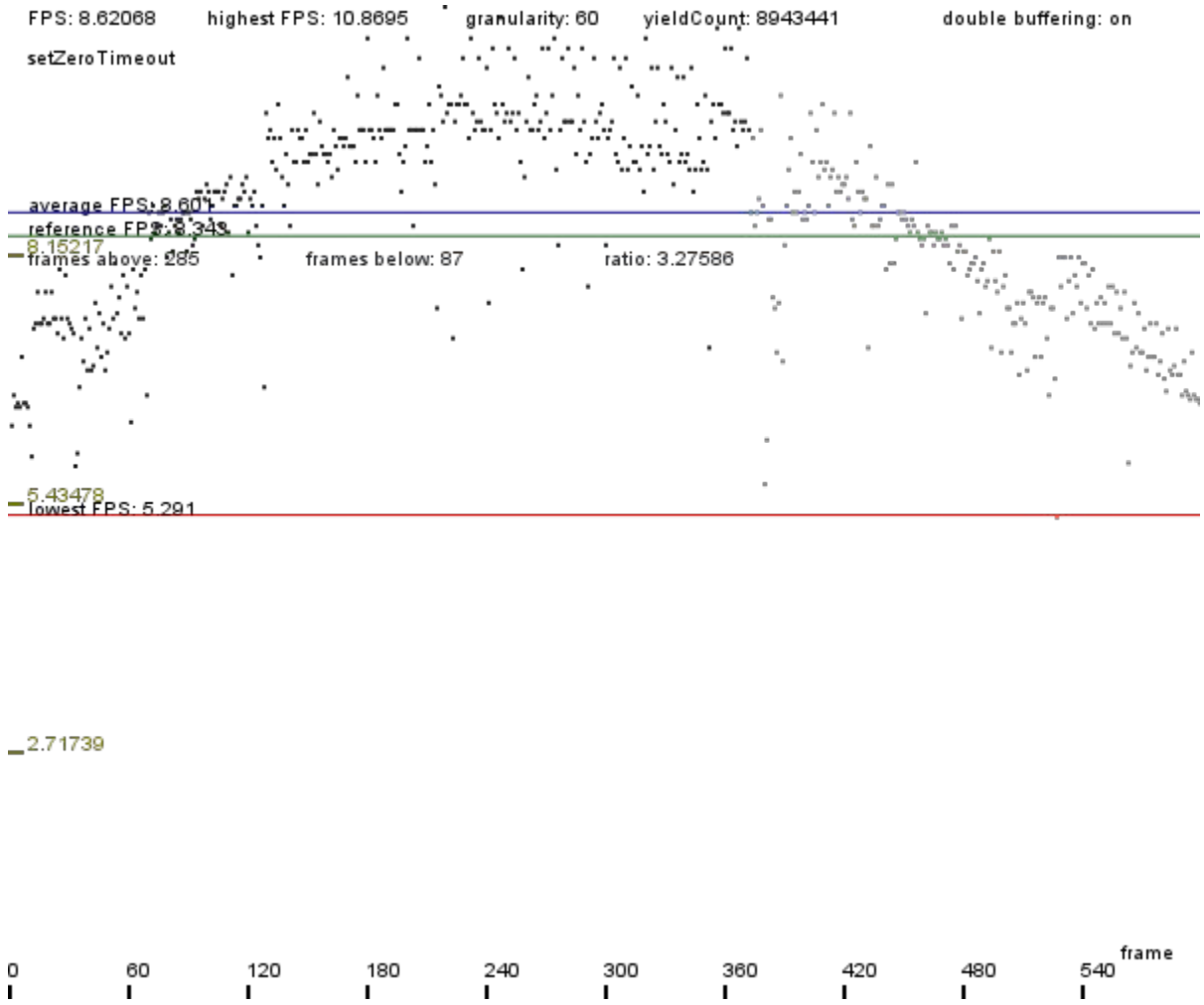


Chart 5: Second optimization – `setZeroTimeout`

Average FPS ~ 8.6.

All calls to `setTimeout` with the second argument of 0 were replaced by a call to `setZeroTimeout`[7] – a major optimization. `setTimeout` effectively has a minimum delay of about 4 ms[10]. `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.

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

Third optimization – increasing `_yieldGranularity`

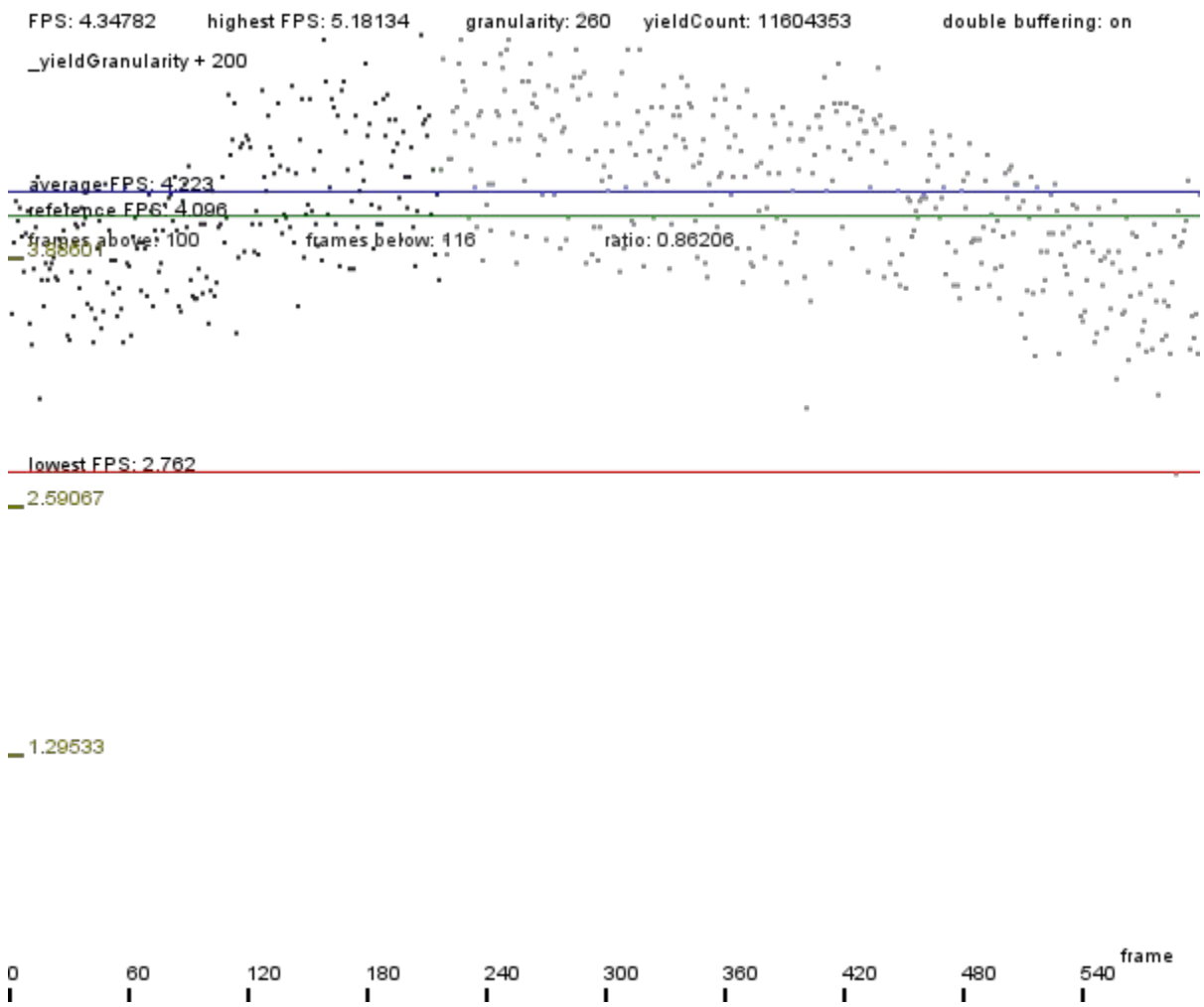


Chart 6: Third optimization – calling `setTimeout` less often

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[11] and the maximum value of `_yieldGranularity` differs between applications and browsers).

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

Fourth optimization – turning off double buffering[12]

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.

12 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

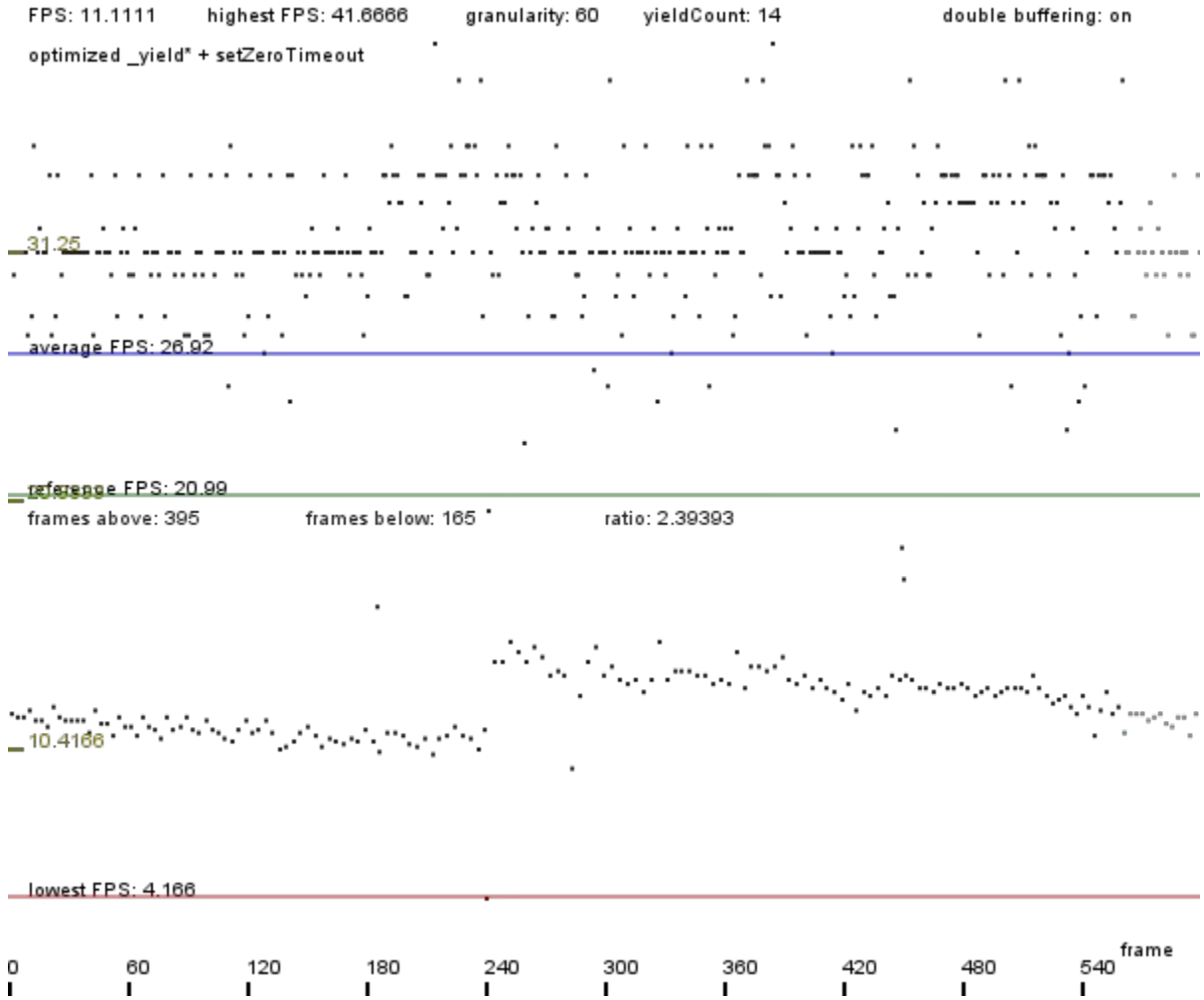


Chart 8: First two optimizations combined

Average FPS ~ 27.

Great improvement. The oscillation clearly apparent.

First three optimizations combined

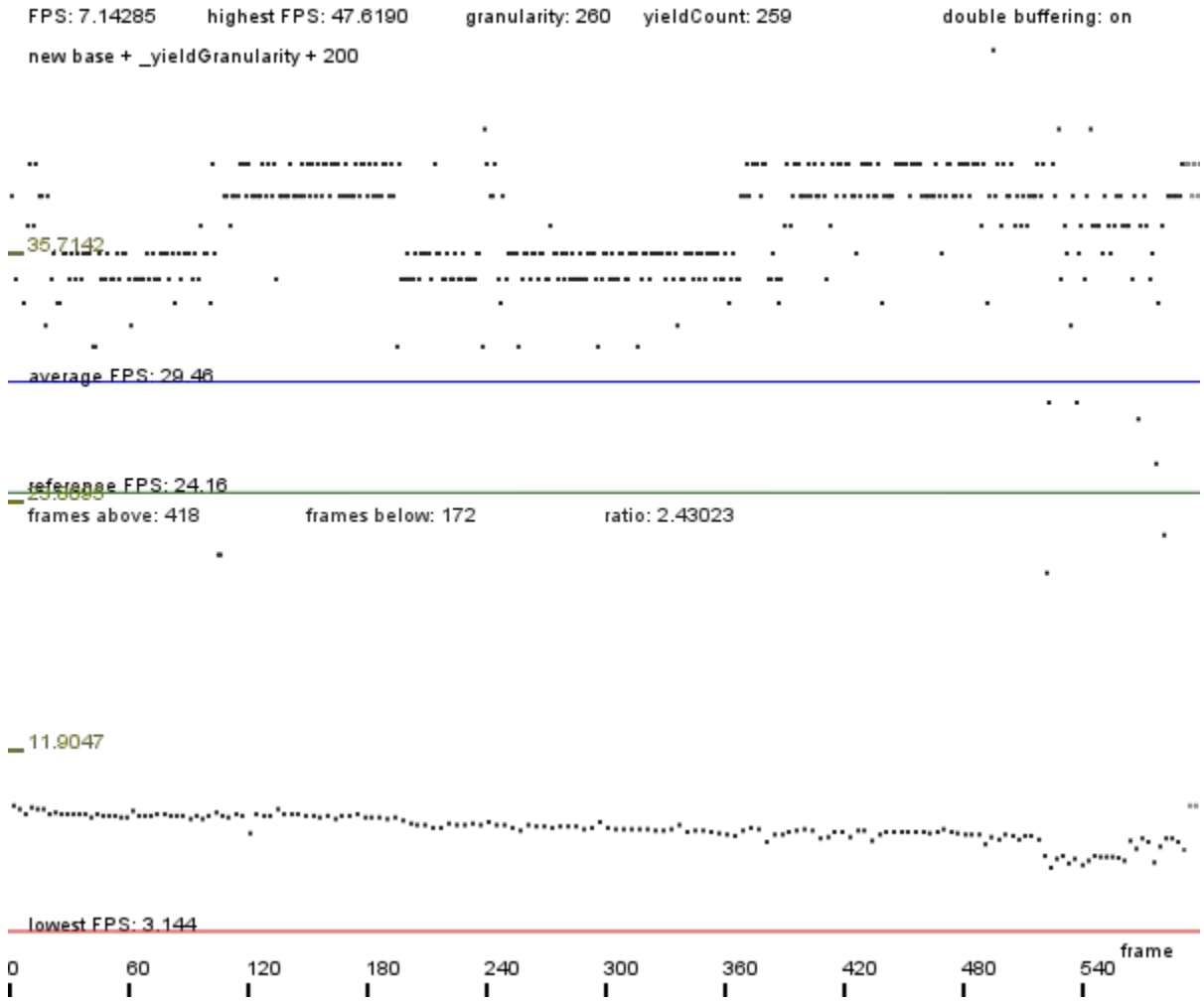


Chart 9: First three optimizations combined

Average FPS ~ 29.

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

First two optimizations without double buffering

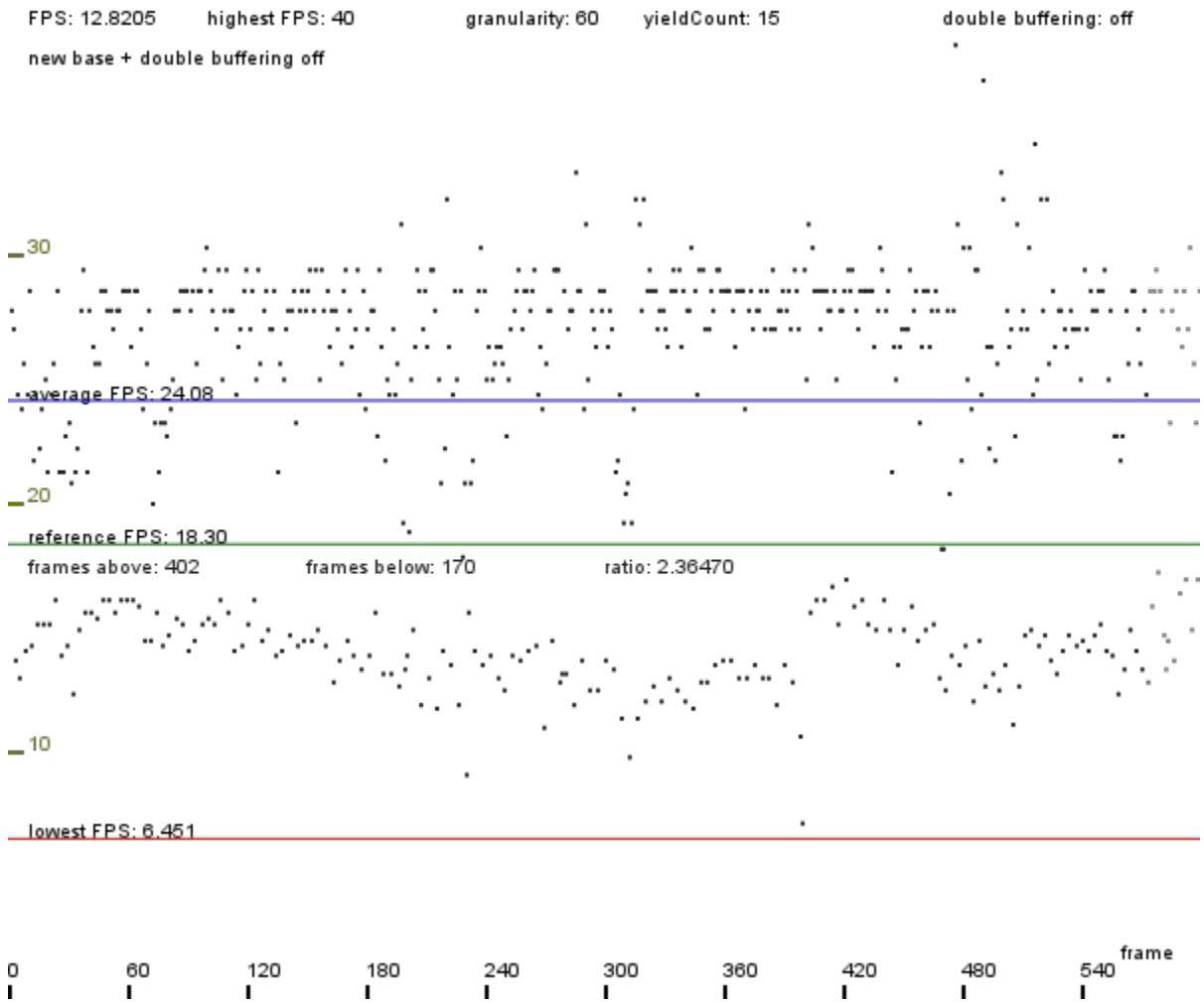


Chart 10: First two and the fourth optimization combined

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:

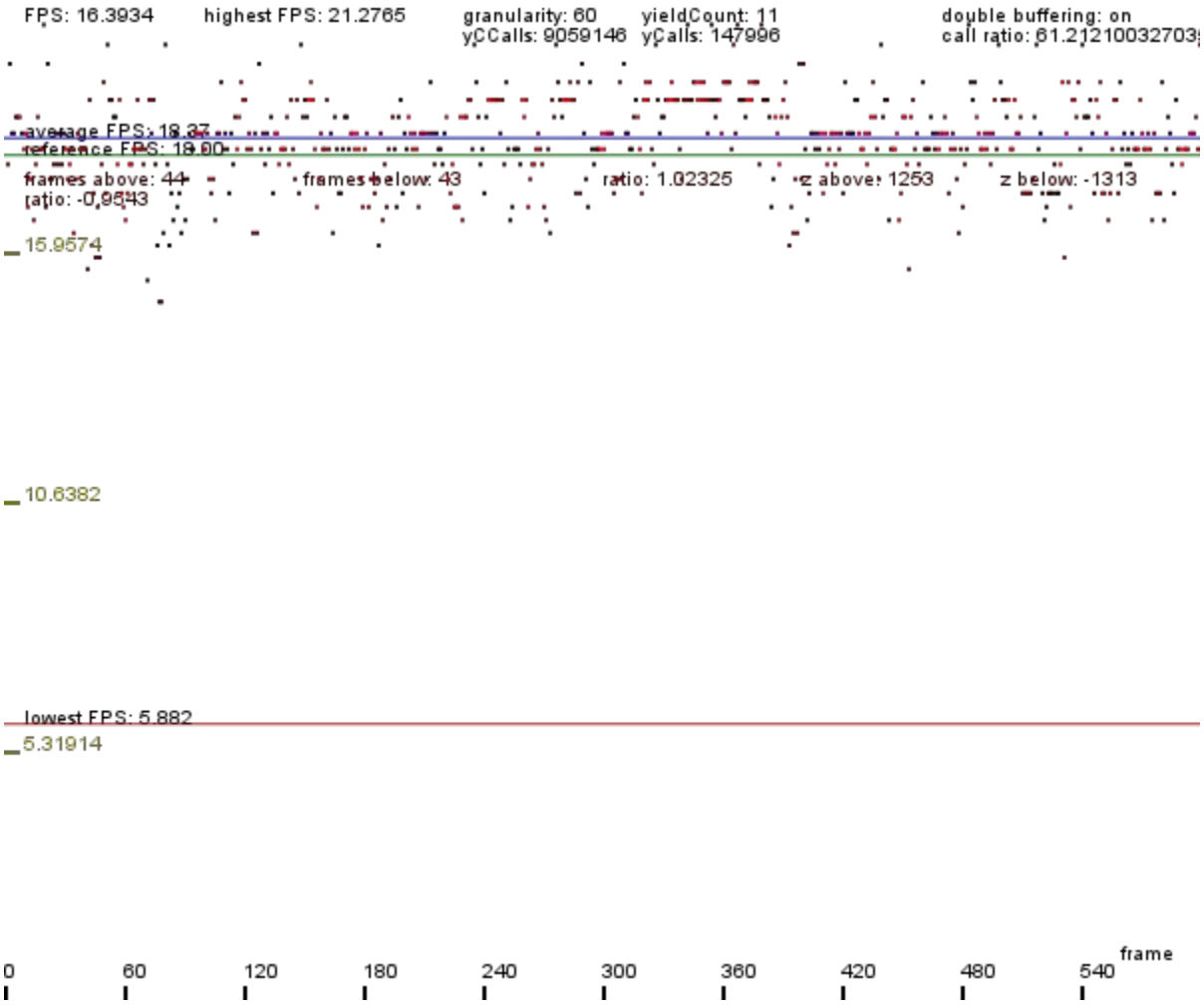


Chart 11: Invoking GC after mapping a function over a big list to clean up stabilizes the FPS; optimizing the implementation of map may significantly boost the performance

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.

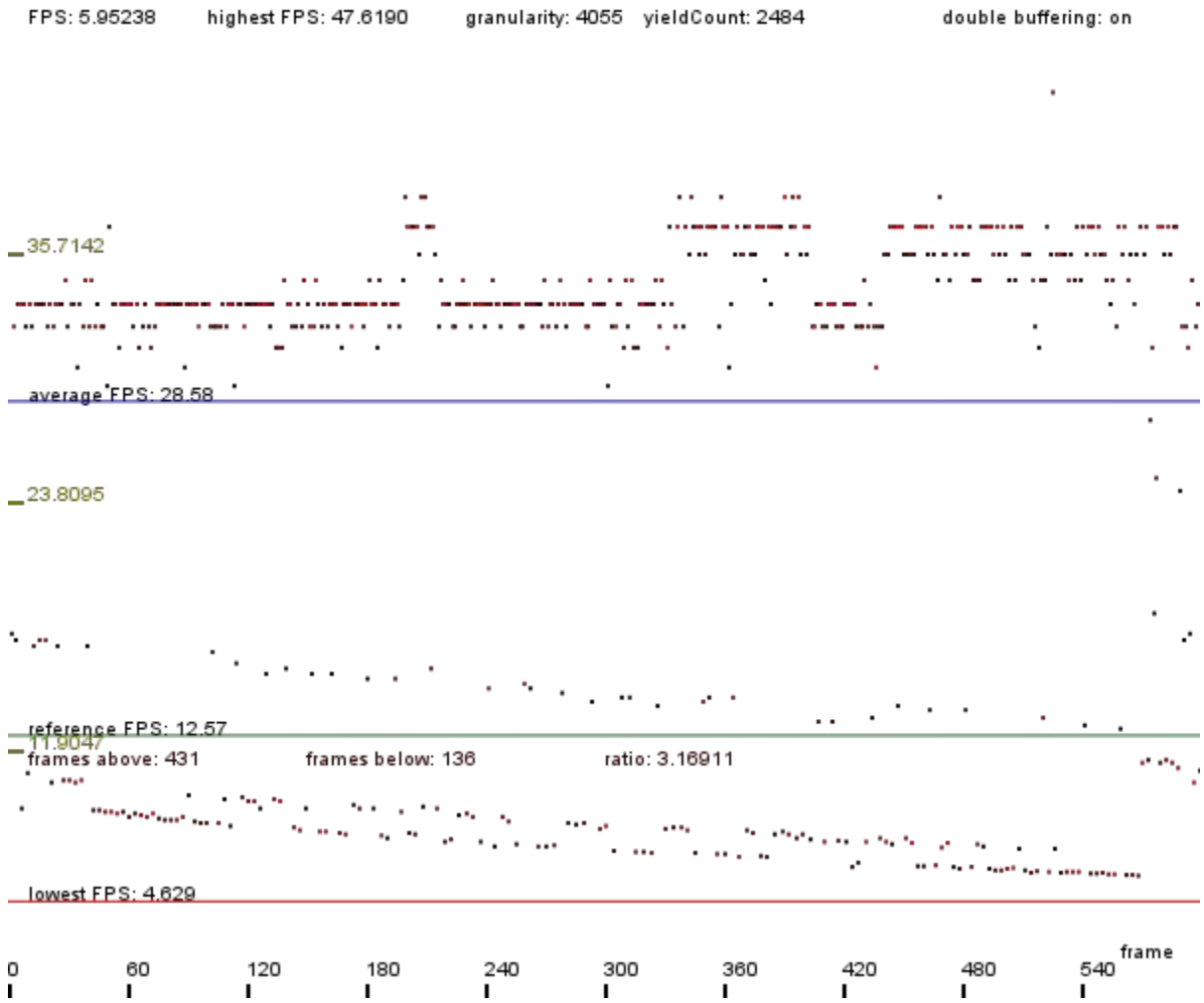


Chart 12: Too high `_yieldGranularity` results in more GC slowdowns

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[13]. 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:

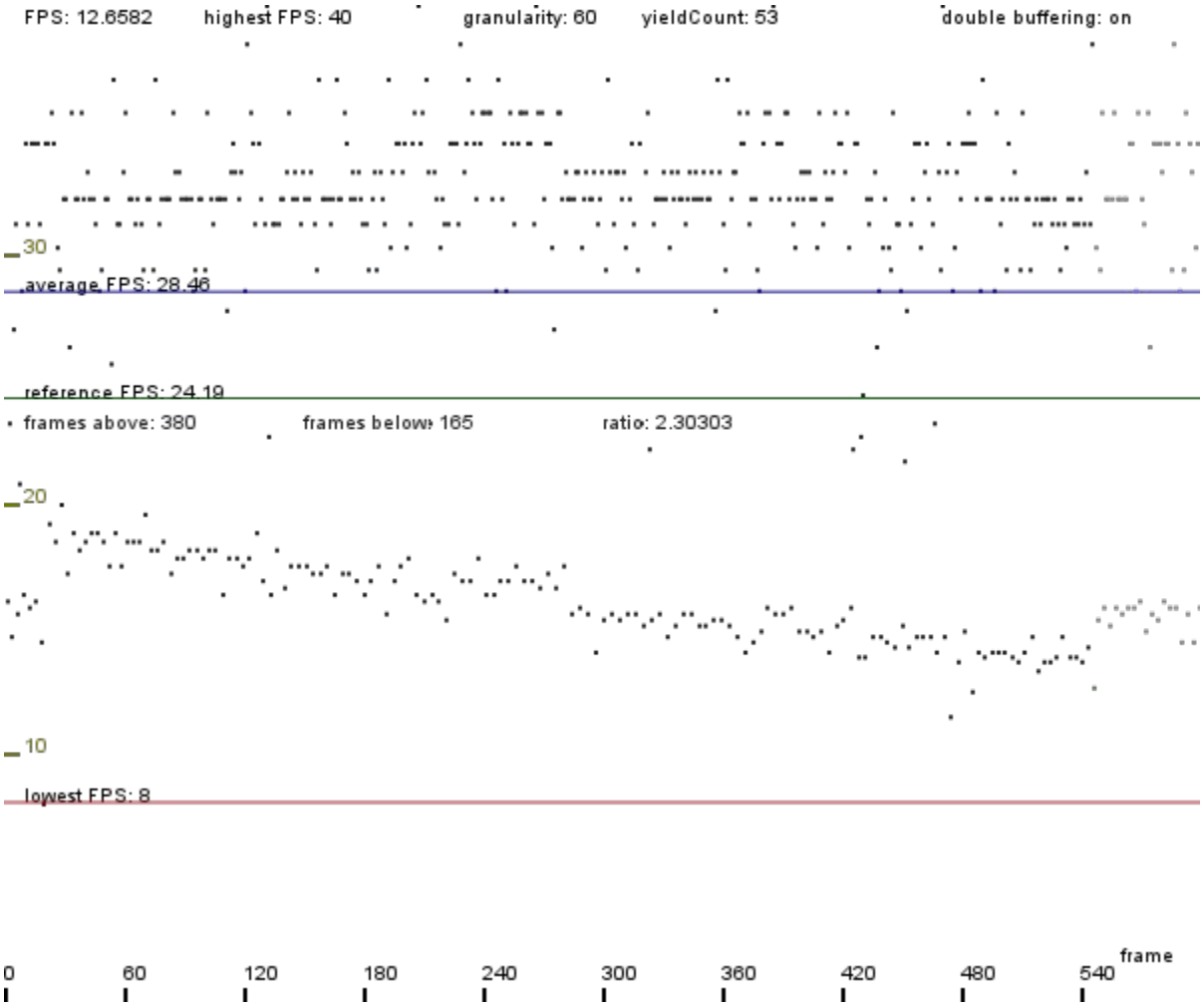


*Chart 13: Performance of the **unoptimized** code*

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

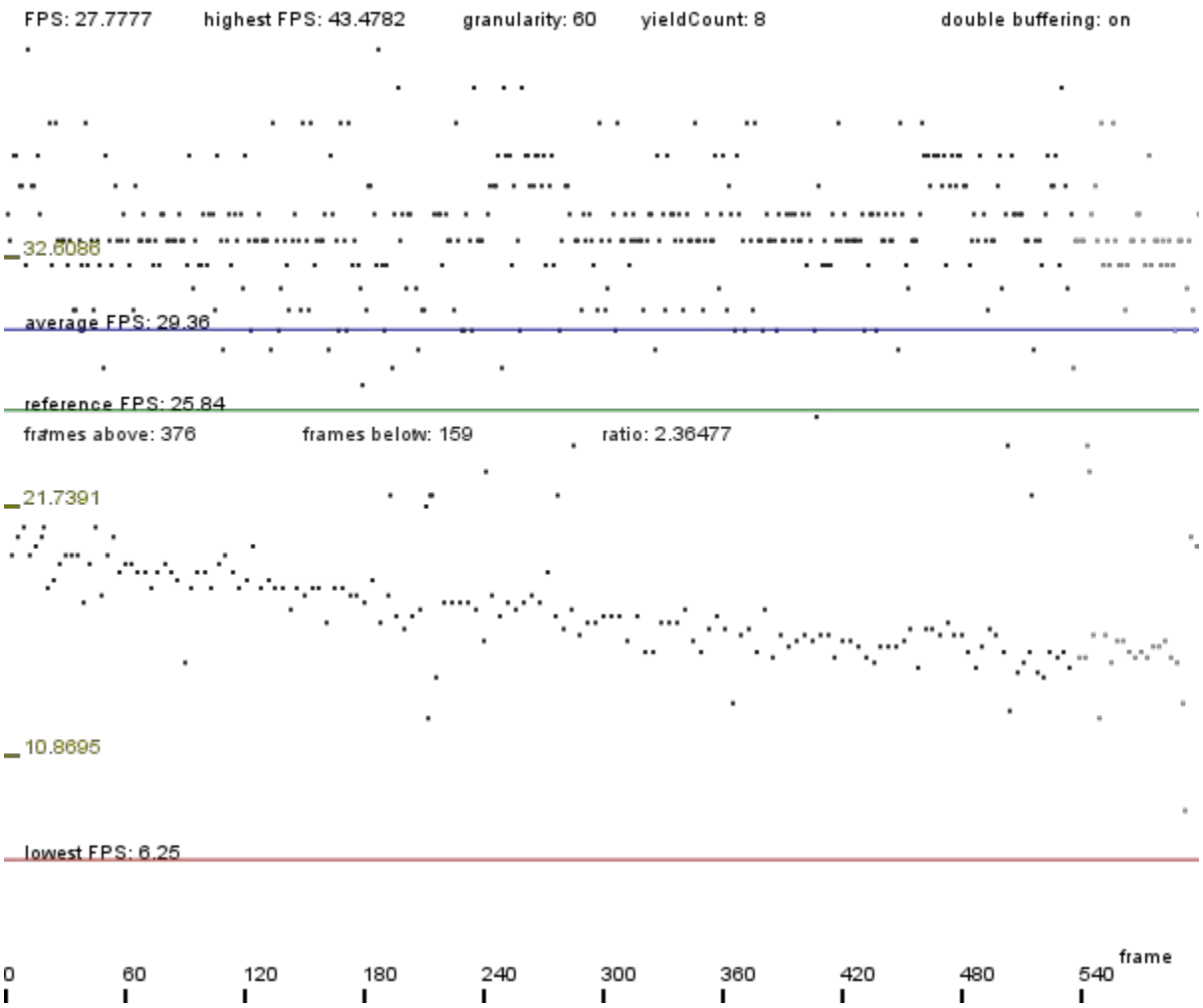


*Chart 14: Performace of the code **optimized** with the Closure Compiler*

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.

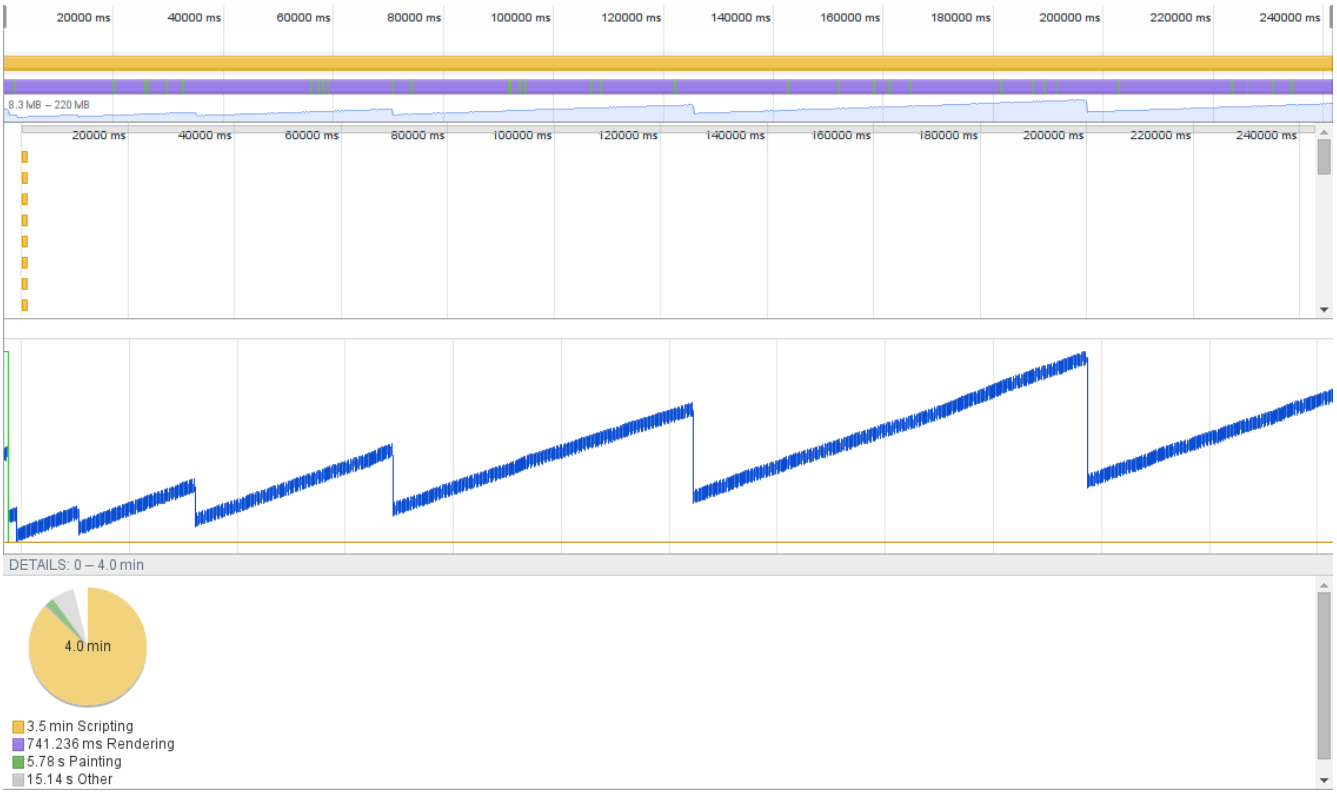


Chart 15: Chromium profiling timeline plot for ***unoptimized*** *code*

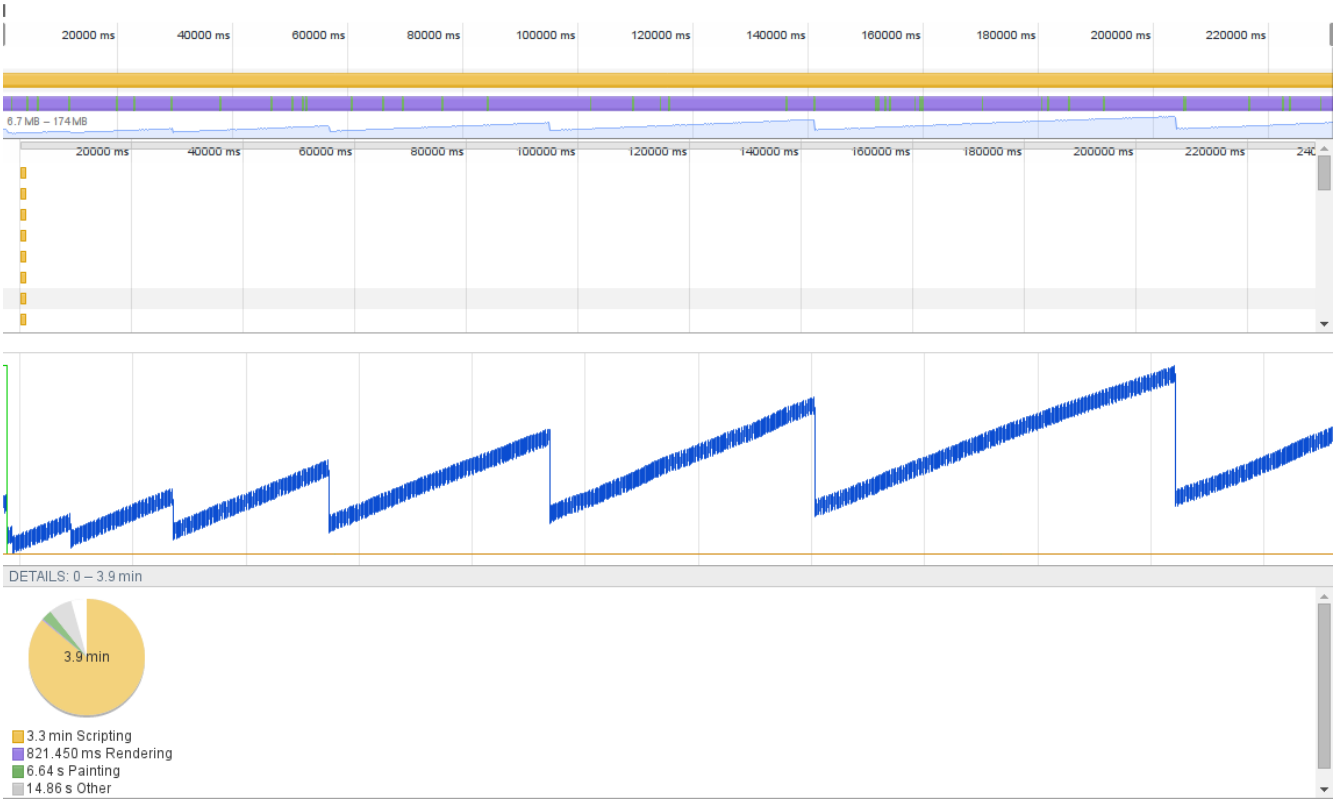


*Chart 16: Chromium profiling timeline plot for code **optimized** with the Closure Compiler*

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:

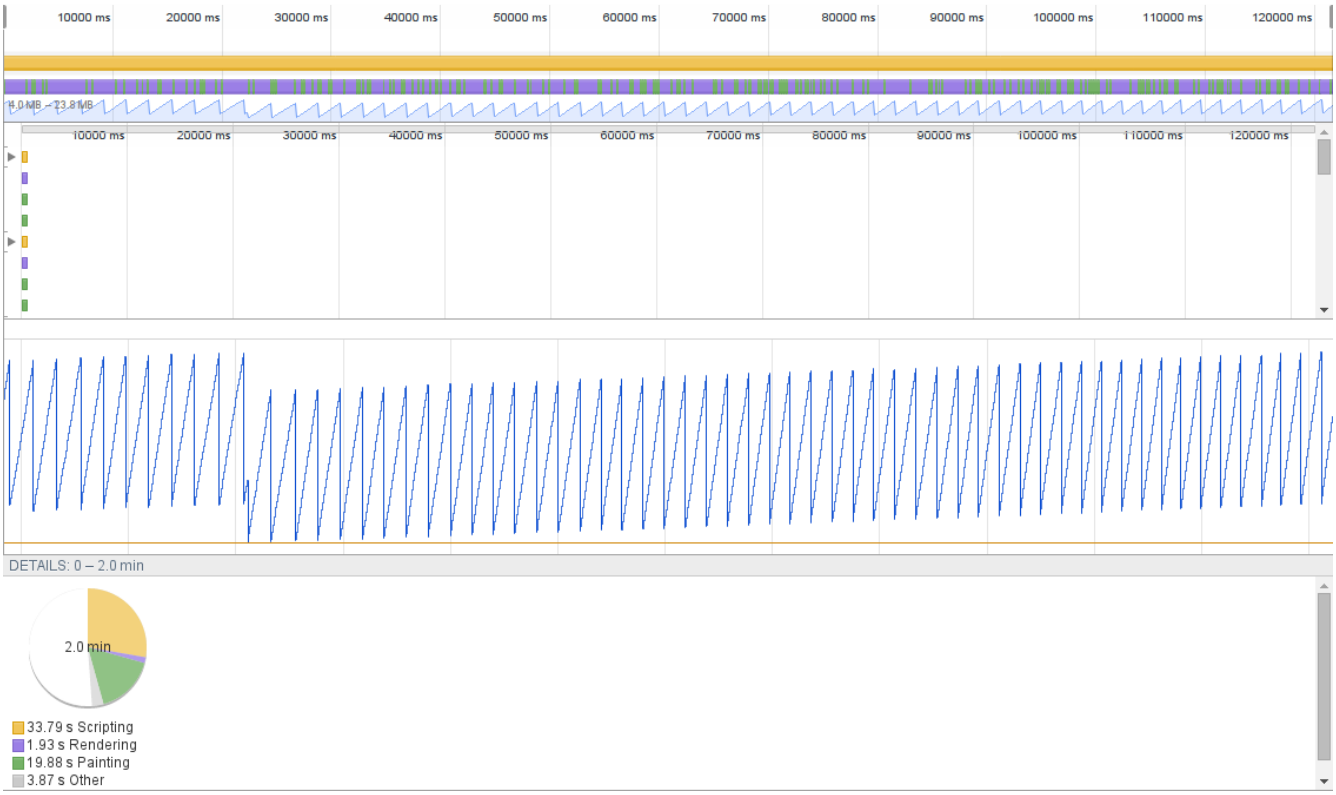
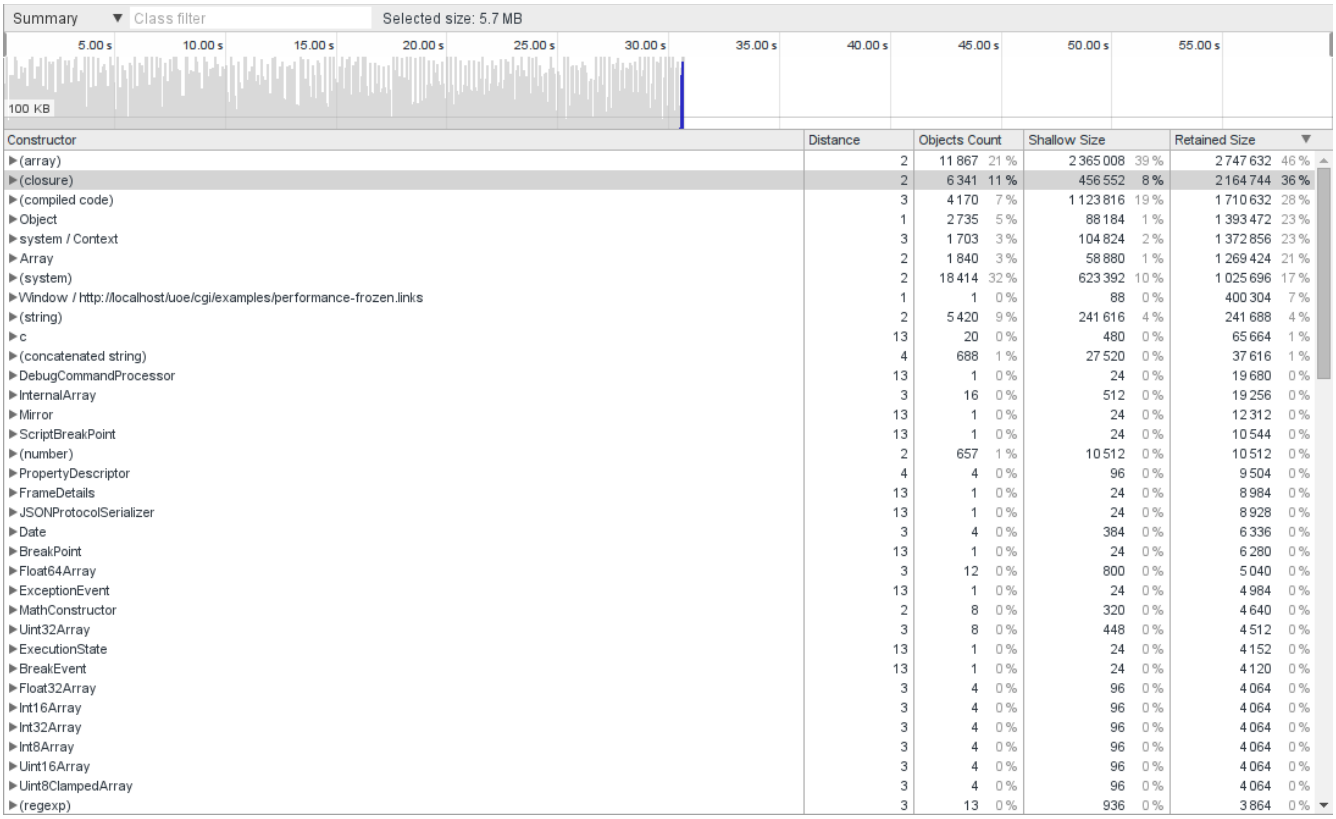


Chart 17: Timeline for the native JavaScript version

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.

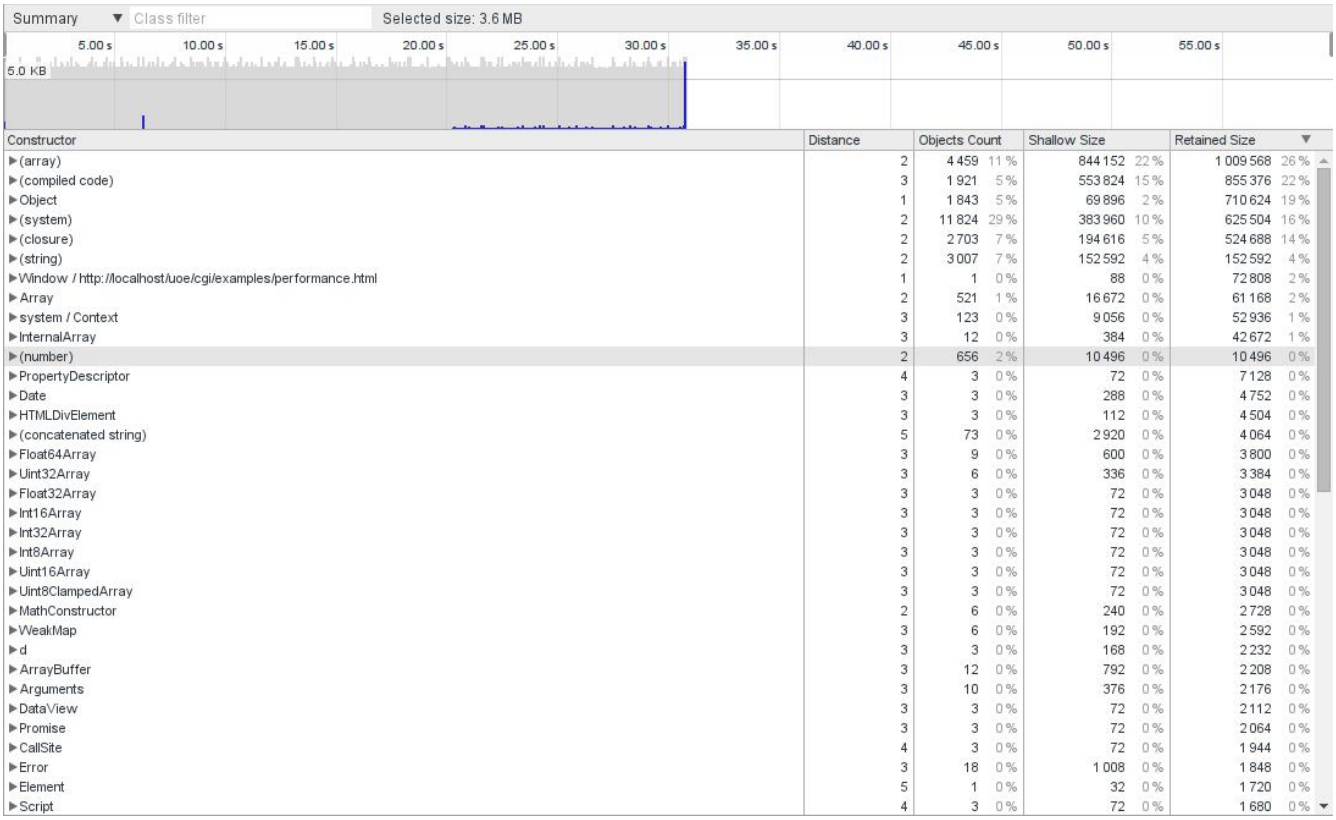
Compare also the heap allocation record:



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`[14]):

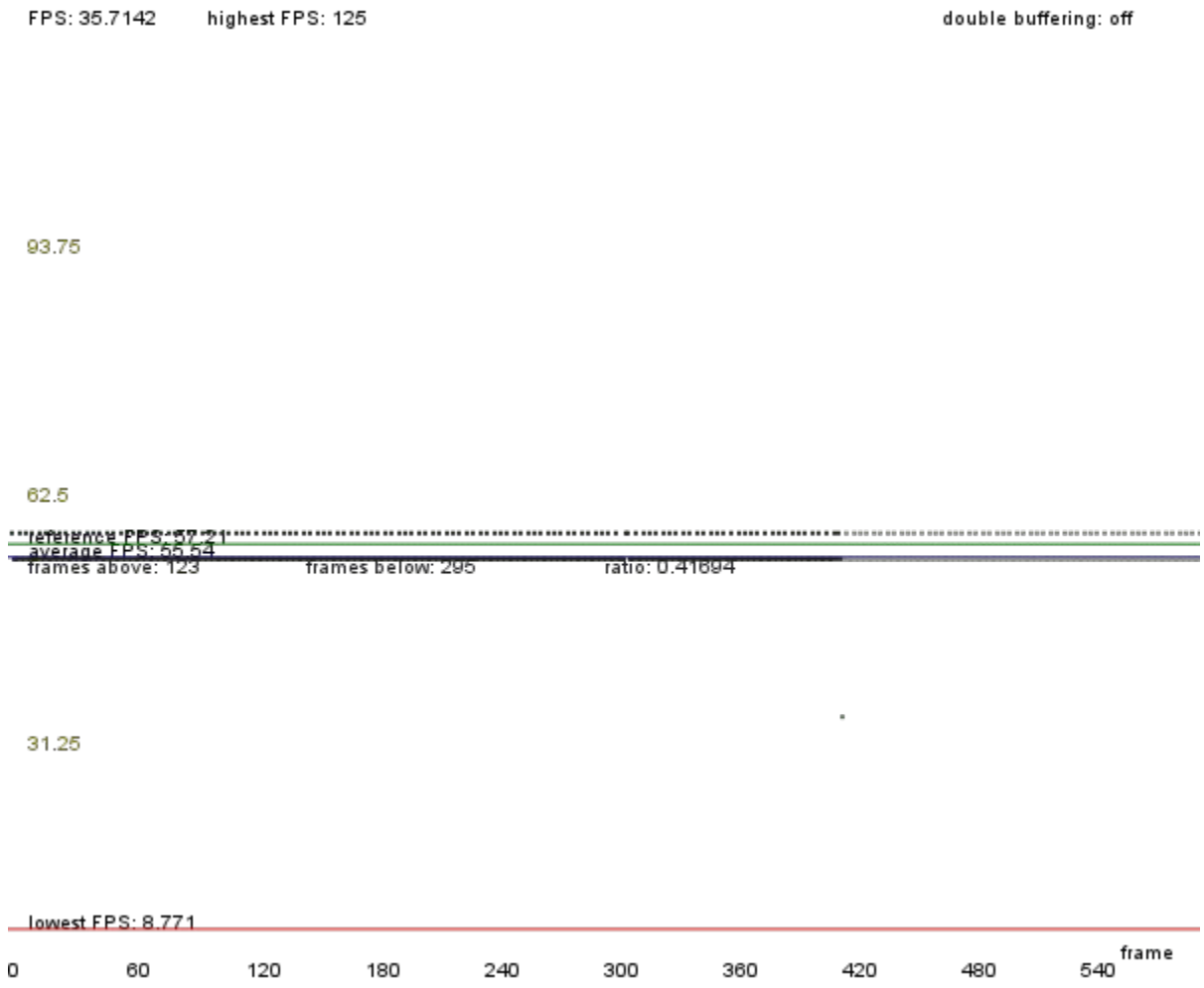


Chart 20: A chart generated by the JavaScript version of the benchmark

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

14 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[15] – 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[16] 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()`

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

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

Todo and remarks

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

Todo

- Test the performance of the current implementation of lists. A lot of the garbage is probably generated because of unneccessary copying of JavaScript arrays (which are used to represent lists)
- Analyze how various things are implemented in Elm[17] and Opa[18]
- Make a more efficient implementation of lists and list manipulating functions based on linked lists and test that
- Test some optimizations to LINKS.eq – make specialized equality functions and compare their performance
- Maybe try out various other small optimizations (like reducing the frequency of context switching) and tests
- Implement a clone of Tetris and a clone of Pacman and use these games as benchmarks
- Quantify some optimizations described in this document, analyze them in more detail and update the document accordingly
- Update my GitHub branch to the latest version of the *sessions* branch
- Create another document, based on this one, that will describe the optimizations and tests planned in this todo[19]

Random remarks

- 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)
- A good place to test and compare the performance of various JavaScript constructs is jsPerf[20]

---

17 http://elm-lang.org/
18 http://opalang.org/
19 That's a lot of self-reference. Not enough[19].
20 http://jsperf.com/