Performance & Memory Impacts of a Hopefully Final Version of a Generational DOM GC

Kentaro Hara (haraken@chromium.org)
2012 Oct 19

Summary: This document demonstrates performance & memory impacts of a hopefully final version of a generational DOM GC. The impacts are promising:

- For micro benchmarks and Dromaeo, the generational DOM GC significantly reduced the maximum stop time and improved throughput.

- For real world applications, minor GC cycles of the generational DOM GC freed a substantial amount of memory (24 MB for Facebook, 235 MB for Google Calendar) with acceptable overhead (~10 ms per minor GC cycle). For Facebook, Gmail, Google Presentation and Google Calendar, minor GC cycles reclaimed more than 40% of all Node wrappers.

(Note: This document just explains the results of performance & memory investigations. Please read this document first.)

Experimental settings

Experimental settings are as follows:

- Chromium r161309
- WebKit r130715
- V8 r12691
- Xeon E5520 2.27GHz x 6 cores with hyper-threading disabled
- 24 GB of memory
- Linux 2.6.38.8

In this document, a *non-generational DOM GC* means the GC in the current WebKit + V8. A *generational DOM GC* means the GC we have been proposing in this document. (See the V8 side patch and the WebKit side patch for more details.)

Evaluation in the most pathological case

To observe the performance for the most pathological case, I measured the execution time of each iteration of the following code:

```
for (var iter = 0; iter < 300; iter++) { // each iteration
  for (var i = 0; i < 100000; i++)
    document.createElement("div");
}
```

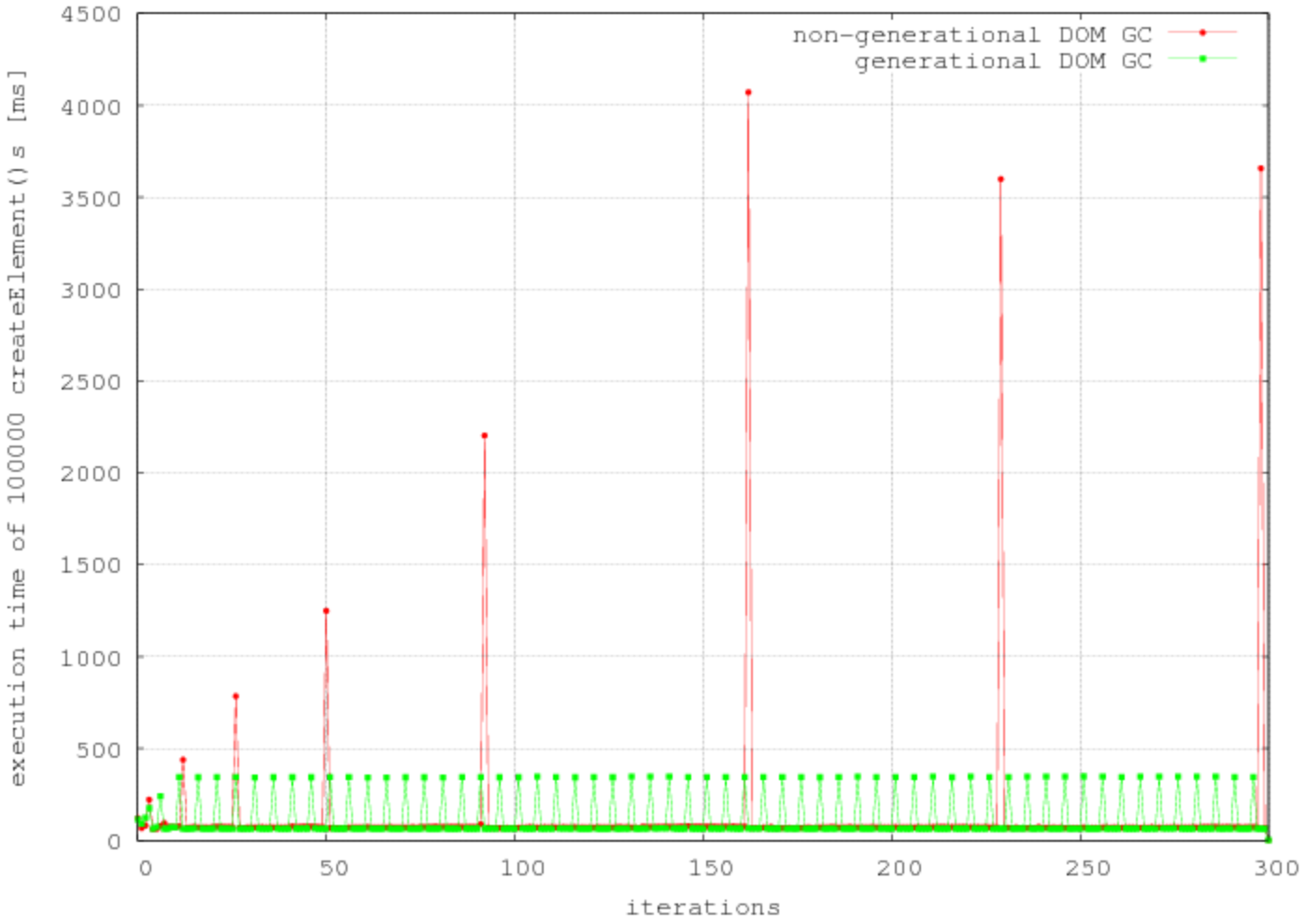

Fig.1 Execution time of 100000 createElement()s

Fig.1 shows the execution time of each iteration, which consists of 100000 createElement()s. From Fig.1, we can observe the following points:

- The maximum stop time is 4072 ms for the non-generational DOM GC and 348 ms for the generational DOM GC.

- The median time is 70 ms for the non-generational DOM GC and 63 ms for the generational DOM GC.

- The average time is 123 ms for the non-generational DOM GC and 118 ms for the generational DOM GC.

In summary, the generational DOM GC significantly reduces the maximum stop time and slightly improves throughput.

(Note: If the generational DOM GC could reclaim all Nodes in minor GC cycles, the execution times of the generational DOM GC would become constant. So you may wonder why the execution times are not constant. The reason is that not all Nodes are reclaimed by minor GC cycles due to the GC complexity. I will explain the details later.)

Evaluation in micro benchmarks

Micro benchmarks

I prepared the following 4 micro benchmarks, which will be most affected by GC performance:

```
// [createElement]
// Create a lot of Elements
for (var i = 0; i < 100000; i++)
  document.createElement("div");

// [appendChild-1]
// Construct a tree by appendChild(), and then make the tree
unreachable
for (var i = 0; i < 50000; i++) {
  var div = document.createElement("div");
  for (var j = 0; j < 100; j++)
    div.appendChild(document.createElement("div"));
}

// [appendChild-2]
// Construct a tree by appendChild(), and then keep the tree
reachable
var div = document.createElement("div");
for (var i = 0; i < 50000; i++) {
  for (var j = 0; j < 100; j++)
    div.appendChild(document.createElement("div"));
}

function binTree(depth) {
  return depth == 0 ? "" : ("<div>" + binTree(depth - 1) +
binTree(depth - 1) + "</div>");
}

// [innerHTML]
// Construct a tree by innerHTML, and then make the tree
unreachable
```

```
var html = binTree(4);
for (var i = 0; i < 100000; i++) {
  var div = document.createElement("div");
  div.innerHTML = html;
}
```

Overall performance

	average (non-gen GC) [ms]	median (non-gen GC) [ms]	max (non-gen GC) [ms]	average (gen GC) [ms]	median (gen GC) [ms]	max (gen GC) [ms]	perf diff
createElement	92.5	55.0	2803	87.9	46.0	264	+5.0%
appendChild-1	5633	6665	6747	5418	5418	5538	+3.8%
appendChild-2	6773	7353	7838	6755	6981	7665	+2.6%
innerHTML	3057	2164	6599	2475	2470	2516	+19.0%

Table.1 Performance comparison (micro benchmarks)

Table.1 shows the performance of the 4 micro benchmarks. The first 3 columns indicate the average/median/max of the non-generational DOM GC, and the following 3 columns indicate the average/median/max of the generational DOM GC. *perf diff* is calculated by 1 - *average (gen GC)* / *average (non-gen GC)*.

(Note: The numbers in Table.1 are the averages/medians/maxes of enough runs. I observed that standard deviations of these numbers are a bit large due to the GC complexity.)

From Table.1, we can observe the following points:

- The generational DOM GC is faster than the non-generational DOM GC for all micro benchmarks. This indicates that the throughput of the generational DOM GC is larger than that of the non-generational DOM GC.

- The maximum stop time of the generational DOM GC is smaller than that of the non-generational DOM GC, especially for createElement and innerHTML.

- The difference between *max* and *average* is smaller in the generational DOM GC than in the non-generational DOM GC. This indicates that the generational DOM GC is more stable than the non-generational DOM GC.

Break-down of the non-generational / generational DOM GC

To break down the overall performance, I measured how much time is consumed by minor GC cycles and how much time is consumed by major GC cycles.

Before looking at the results, please keep the following points in mind:

- In the non-generational DOM GC, minor GC cycles do nothing for Nodes. All the burden is on major GC cycles. Thus, minor GC cycles will finish very quickly and major GC cycles will take a long time.

- In the generational DOM GC, minor GC cycles can reclaim dead Nodes. However, we cannot expect that all dead Nodes are reclaimed by minor GC cycles, due to the GC complexity:

    - Normally, wrappers survive two minor GC cycles and then are promoted to the old space.

    - However, in a situation where wrappers are created at high speed (Note: In particular this situation is likely to happen in micro benchmarks. Even in real world applications, this is not a rare situation.), the new space becomes 25% full shortly. If the new space becomes 25% full, wrappers in the new space are forcibly promoted to the old space without surviving two minor GC cycles. In this way, wrappers can be promoted to the old space at the first minor GC cycle. In addition, when a major GC cycle is triggered, all wrappers in the new space are forcibly promoted to the old space. In summary, the condition under which a wrapper X is promoted to the old space is as follows:

        - The wrapper X survived two minor GC cycles; or

        - A major GC cycle is triggered; or

        - The new space is 25% full.

    - It requires two GC cycles to reclaim wrappers. The first GC cycle disposes the wrappers. The second GC cycle reclaims the wrappers. Therefore, the only case where minor GC cycles can reclaim dead wrappers is the case where the wrappers survive two minor GC cycles. Otherwise (i.e. in a case where wrappers are promoted to the old space at the first minor GC cycle), the wrappers have to be reclaimed by the next major GC cycle.

    - The size of the new space changes dynamically. The new space can expand or

shrink dynamically depending on its saturation.

- The minor GC algorithm is conservative. The algorithm might fail to reclaim Nodes that are actually dead. Specifically, the Nodes that the minor GC algorithm can reclaim are only Nodes in a DOM tree X that meets the following condition:

    - All wrappers in the DOM tree X have been created since the most recent minor/major GC cycle. All the wrappers are neither strongly-reachable nor weakly-reachable.

(Note: These GC heuristics are designed based on the assumption that minor GC cycles cannot reclaim Nodes. Now minor GC cycles can reclaim a lot of Nodes, we might want to optimize the heuristics so that the generational DOM GC works more smartly.)

	minor count	minor avg [ms]	minor max [ms]	minor total [ms]	major count	major avg [ms]	major max [ms]	major total [ms]	GC total [ms]
createElement	1500	1.1	16	1598	9	1687	3852	15187	16785
appendChild-1	1260	1.1	16	1339	8	1658	4534	13265	14604
appendChild-2	1240	1.1	16	1367	7	2274	5208	15923	17290
innerHTML	6	4.5	9	27	2	590	907	1180	1207

Table.2 Break-down of the non-generational DOM GC (micro benchmarks)

	minor count	minor avg [ms]	minor max [ms]	minor total [ms]	major count	major avg [ms]	major max [ms]	major total [ms]	GC total [ms]
createElement	64	203	223	13017	61	65	69	3977	16994
appendChild-1	40	134	157	5365	37	178	198	6581	11946
appendChild-2	992	2.1	26	2085	6	1721	4703	10324	12409
innerHTML	5	153	402	767	2	233	315	466	1233

Table.3 Break-down of the generational DOM GC (micro benchmarks)

Table.2 and Table.3 show the break-down of the non-generational / generational DOM GC, respectively. Each column indicates the following values:

minor count: How many times minor GC cycles are triggered.

minor avg: The average time of minor GC cycles.

minor max: The maximum time of minor GC cycles.

minor total: The total time consumed by minor GC cycles. *minor avg* = *minor total* / *minor count*.

major count: How many times major GC cycles are triggered.

major avg: The average time of major GC cycles.

major max: The maximum time of major GC cycles.

major total: The total time consumed by major GC cycles. *major avg* = *major total* / *major count*.

GC total: The total time consumed by minor and major GC cycles. *GC total* = *minor total* + *major total*.

(Note: I measured these GC statistics in a separate run from the run to measure the overall performance shown in Table.1 (because the overhead to measure the GC statistics is not ignorable). Thus, comparing the values in Table.1 with the values in Table.2 and Table.3 does not make sense.)

From Table.2 and Table.3, we can observe the following points:

- In the non-generational DOM GC, a minor GC cycle finishes very quickly. It just takes 1.1 ms or 4.5 ms in average. On the other hand, in the generational DOM GC, a minor GC cycle takes ~203 ms in average.

- The maximum stop time can be calculated as max(*minor max*, *major max*). This value is much smaller in the generational DOM GC than in the non-generational DOM GC. For example, for appendChild-1, while the maximum stop time of the non-generational DOM GC is 4534 ms, the maximum stop time of the generational DOM GC is 198 ms.

- *GC total* is smaller in the generational DOM GC than in the non-generational DOM GC.

- One interesting thing is that although *GC total* of innerHTML is almost the same between the non-generational DOM GC and the generational DOM GC, the overall performance of the generational DOM GC is significantly better than the non-generational DOM GC in Table.1. This would be due to cache efficiency (at the level

of hardware cache or at the level of the tcmalloc algorithm). Since the generational DOM GC can reclaim dead Nodes earlier, the amount of memory used by a program is reduced, which would lead to the better performance.

How much work minor GC cycles did

	Reclaimed Nodes	Disposed Node wrappers	Promoted Node wrappers to Old	Minor GC rate
createElement	29700000	29700000	9	100%
appendChild-1	25186774	25186774	5400	100%
appendChild-2	15000003	15000561	24894334	37%
innerHTML	6400000	400000	10	100%

Table.4 How much work minor GC cycles did in the generational DOM GC (micro benchmarks)

Next, I measured how much work minor GC cycles did in the generational DOM GC. Specifically, I measured the following values:

Reclaimed Nodes: The number of Nodes destructed by minor GC cycles. Specifically, I counted the number of Node destructors called inside weakNodeCallback() called back by minor GC cycles.

Disposed Node wrappers: The number of Node wrappers disposed by minor GC cycles. Some Node wrappers might be promoted to the old space and wait for the next major GC cycle that will reclaim the Node wrappers. Remember that it takes two GC cycles to reclaim wrappers: the first GC cycle disposes a wrapper, and the second GC cycle reclaims the wrapper.

Promoted Node wrappers to Old: The number of Node wrappers that are promoted to the old space. This value does not include the number of Node wrappers that are disposed just after the promotion (i.e. this value does not include the number of Node wrappers that are promoted to the old space and just waiting for the next major GC cycle that will reclaim the Node wrappers). Also this value does not include the number of non-Node wrappers, which we are not interested in now.

Minor GC rate: The rate of Node wrappers that are disposed by minor GC cycles to the whole Node wrappers. This value is calculated by *Disposed Node wrappers* / (*Disposed Node wrappers* + *Promoted Node wrappers to Old*). This value indicates “how helpful minor GC cycles are”.

From Table.4, we can observe the following points:

- For createElement, appendChild-1 and innerHTML, 100% of Node wrappers are disposed by minor GC cycles. This is an expected behavior. Minor GC cycles are super helpful.

Evaluation in Dromaeo

I conducted the same experiment for Dromaeo.

Overall performance

	average (non-gen GC) [runs/s]	average (gen GC) [runs/s]	perf diff
dom-attr	6309.95	6352.98	+0.68%
dom-modify	3866.63	4173.96	+7.95%
dom-query	786324.76	784739.41	-0.20%
dom-traverse	2592.00	2578.67	-0.51%

Table.5 Performance comparison (Dromaeo)

Table.5 shows the performance of Dromaeo. Note that dom-modify is the only benchmark that will be affected by GC performance. dom-attr, dom-query and dom-traverse do not create a lot of Nodes. From Table.5, we can observe 8% speed-up for dom-modify.

Break-down of the non-generational / generational DOM GC

	minor count	minor avg [ms]	minor max [ms]	minor total [ms]	major count	major avg [ms]	major max [ms]	major total [ms]	GC total [ms]
dom-attr	440	0.0	1	2	107	4.2	8	452	454

dom-modify	402	1.4	21	577	46	155	910	7134	7711
dom-query	540	0.0	1	3	1	5.0	5	5	8
dom-traverse	4	1.0	1	4	0	0.0	0	0	4

Table.6 Break-down of the non-generational DOM GC (Dromaeo)

	minor count	minor avg [ms]	minor max [ms]	minor total [ms]	major count	major avg [ms]	major max [ms]	major total [ms]	GC total [ms]
dom-attr	680	0.0	1	3	109	4.2	8	463	466
dom-modify	452	10	343	4763	52	21	152	1127	5890
dom-query	480	0.0	1	3	1	5.0	5	5	8
dom-traverse	4	1.0	1	4	0	0.0	0	0	4

Table.7 Break-down of the generational DOM GC (Dromaeo)

Table.6 and Table.7 show the break-down of the non-generational / generational DOM GC. Focusing on dom-modify, the generational DOM GC succeeds in significantly reducing the time consumed by major GC cycles (7134 ms → 1127 ms) instead of increasing the time consumed by minor GC cycles (577 ms → 4763 ms). As a result, the GC total time is significantly reduced (7711 ms → 5890 ms).

How much work minor GC cycles did

	Reclaimed Nodes	Disposed Node wrappers	Promoted Node wrappers to Old	Minor GC rate
dom-attr	247	222	204	52%
dom-modify	13003980	8207561	1376	100%
dom-query	27	149	593	20%
dom-traverse	202	175	1358	11%

Table.8 How much work minor GC cycles did in the generational DOM GC (Dromaeo)

Table.8 shows how much work minor GC cycles did in the generational DOM GC. Focusing on dom-modify, 100% of DOM nodes are disposed by minor GC cycles. Perfect.

Evaluation in real world applications

Real world applications

Finally, I measured the performance & memory impacts of the generational DOM GC on real world applications. Specifically, I manually crawled the following web applications for 2 mins and measured the GC behavior (a.k.a. network stalking):

- Facebook
- Twitter
- Gmail
- Google Docs
- Google SpreadSheet
- Google Presentation
- Google Calendar

I conducted this experiment for the generational DOM GC only. Because the manual crawling is not reproducible, it does not make sense to compare the results between the non-generational DOM GC and the generational DOM GC.

Break-down of the generational DOM GC

	minor count	minor avg [ms]	minor max [ms]	minor total [ms]	major count	major avg [ms]	major max [ms]	major total [ms]	GC total [ms]
Facebook	28	8.2	15	231	14	33	100	462	693
Twitter	17	8.5	19	144	10	28	75	285	429
Gmail	14	9.2	18	129	9	39	128	356	485
Docs	24	9.5	20	228	7	24	47	168	396
Spreadsheet	9	2.0	4	18	16	19	48	297	315
Presentation	52	5.6	17	292	7	24	70	171	463
Calendar	103	2.5	10	260	18	39	90	705	965

Table.9 Break-down of the generational DOM GC (real world applications)

Table.9 shows the break-down of the generational DOM GC. We can observe the following points:

- The average time of minor GC cycles is ~10 ms in average. The maximum time of minor GC cycles is ~20 ms. This would be acceptable for a stop time of minor GC cycles. (When I observed 300 ms~ stop time of minor GC cycles for micro benchmarks in Table.3, I was afraid that such a heavy minor GC would not be acceptable. However, such a pathological case seems not to happen in real world applications.)

- On the other hand, the average time of major GC cycles is ~40 ms. The maximum time of major GC cycles is ~705 ms.

- One interesting thing is that the GC total time is smaller than I expected. Although I manually crawled each application for 2~ mins, the GC total time is just ~965 ms.

How much work minor GC cycles did

	Reclaimed Nodes	Disposed Node wrappers	Promoted Node wrappers to Old	Minor GC rate	Freed memory
Facebook	59010	17156	19780	46%	24.7 MB
Twitter	97149	5450	10697	34%	8.07 MB
Gmail	7941	3858	5201	43%	1.90 MB
Docs	3060	2511	18678	12%	1.72 MB
Spreadsheet	1452	958	1917	33%	0.51 MB
Presentation	60265	36975	28525	56%	35.3 MB
Calendar	1452833	7308	11190	40%	235 MB

Table.10 How much work minor GC cycles did in the generational DOM GC (real world applications)

Table.10 shows how much work minor GC cycles did in the generational DOM GC. *Freed memory* indicates the amount of memory freed inside Node destructors triggered by minor GC cycles. I measured the amount by hooking TcMalloc. From Table.10, we can observe the following points:

- For Facebook, Gmail, Presentation and Calendar, the minor GC rate is 40% or more. This implies that minor GC cycles will reduce the work of major GC cycles by roughly 40% or more. This will reduce the maximum stop time of major GC cycles.

- Minor GC cycles freed 25 MB for Facebook, 35 MB for Presentation, and 235 MB for Calendar.

Overall, the fact that minor GC cycles can free a substantial amount of memory just with the overhead of ~10 ms per cycle demonstrates the effectiveness of the generational DOM GC.

Summary

This document demonstrated performance & memory impacts of a hopefully final version of a generational DOM GC. The impacts are promising:

- For micro benchmarks and Dromaeo, the generational DOM GC significantly reduced the maximum stop time and improved throughput.

- For real world applications, minor GC cycles of the generational DOM GC freed a substantial amount of memory (24 MB for Facebook, 235 MB for Google Calendar) with acceptable overhead (~10 ms per minor GC cycle). For Facebook, Gmail, Google Presentation and Google Calendar, minor GC cycles reclaimed more than 40% of all Node wrappers.