GRAPHICS ACCELERATION FOR HTML5 AND JAVASCRIPT ENGINE JIT OPTIMIZATION FOR MOBILE DEVICES

Contributors

Jonathan Ding
Software and Services Group,
Intel Corporation

Yuqiang Xian
Software and Services Group,
Intel Corporation

Yongnian Le
Software and Services Group,
Intel Corporation

Kangyuan Shu
Software and Services Group,
Intel Corporation

Haili Zhang
Software and Services Group,
Intel Corporation

Jason Zhu
Software and Services Group,
Intel Corporation

HTML5[1] is considered to be the future of the Web and is expected to deliver a user experience that is comparable to or better than that found only in native applications in the past. This does raise significantly higher demand of performance to sustain the much heavier web applications that manipulate more and richer contents than ever before. Consequently, the optimization to the web runtime is extremely important to achieve the success of the platforms, and in particular, the mobile devices, because their hardware capabilities are less powerful than PCs.

This article first introduces the challenges caused by HTML5 in terms of performance. As the rendering engine and JavaScript engine are two key fundamental building blocks of web runtime, it then discusses our solution of graphics acceleration and just-in-time (JIT) optimization applied to the Intel® Atom™[2] platform to dramatically improve the performance of these two components respectively. The article evaluates the impact of our solutions on typical HTML5 and JavaScript benchmarks, and the vision of future work is included as well.

Introduction

HTML5, the open and compelling web technology, is considered to be the future of the Web, and consequently, more and more client applications are created via HTML5, connected with the cloud, and deployed throughout the Web. Gartner[3] estimated that half of the mobile applications will be web applications by 2015 and HTML5 is one of the key driving forces. A recent and typical example is the famous Angry Birds game, which also has an HTML5 version[4] that delivers a user experience consistent with previous native versions of the software.

However, the popularization of HTML5 raises significant demand of performance on client devices to sustain the smooth user experience.

Web applications today have much greater computation requirements than traditional Web surfing and navigating pages. They usually have heavier and more complex logic. The number of lines of JavaScript code in the Gmail client reached 443,000 in 2010, which was almost 50 times the size of 6 years ago.[5] Furthermore, thanks to the proliferation of mobile devices with cameras and sensors attached, HTML5 web applications often produce and consume dramatically richer content, such as images and high definition video clips. Finally, a compelling user experience requires immediate responses to user

actions like touch gestures. It means that the web runtime should be capable to process such high priority interactive tasks in a timely manner.

In addition, there are more challenges of performance for web applications, compared with native or traditional managed binaries, which are composed with languages like C/C++, Java, or .NET. HTML5 promises cross-platform implementation; however, this is approached through an abstraction layer of JavaScript, HTML, and CSS. Such abstraction does bring more overhead than native solution. For example, even though JavaScript performance has improved by a factor of 100 in the past decade[6], it is slower than C/C++ in most cases by at least an order of magnitude. Meanwhile, HTML5 is still an emerging technology. Therefore, unlike the mature Java or .NET runtime, which have already been heavily optimized over decades, the software stack of HTML5 is still young and has much room for improvement, in both functionality and performance.

With complex web runtime applications, the rendering engine and JavaScript engine usually play the key roles and dominate the performance of most of the HTML5 applications. The rendering engine is responsible for drawing the contents and presenting them to the underlying window framework so that operation system can show them on the display. The JavaScript engine executes the application's logic, which is composed with JavaScript. Our analysis shows that on Android*[7] phones, the sum of both components usually occupies more than 60 percent of the CPU execution cycles for typical HTML5 games or general rich page navigations.

Therefore, a lot of existing optimization effort targets these two components. For example, modern desktop browsers such as Chrome[8] use graphics acceleration to improve the efficiency of the rendering engine, in particular, the HTML5 canvas 2D. JIT technology and other advanced compiler technologies are applied to JavaScript engine. Crankshaft[9] of the V8[10] JavaScript engine improves the score of v8bench[11] by more than 50 percent according to Google measurements.

Nevertheless, on mobile devices, such as Android- or Linux-based phones and tablets, the progress is much slower. The Android browser still uses pure CPU for the rendering of HTML5 canvas 2D, even on the latest Android 4.0. WebKit[12] is one of the most popular web engines. Advanced JIT technology, so-called DFG JIT, is enabled inside JSC (JavaScriptCore), the default JavaScript engine that comes with WebKit, whereas it is available on X64 architecture but doesn't work on IA32.

Our work in the article changes this situation on mobile devices. The rest of the article is organized as follows: "Graphics Acceleration for HTML5 Canvas 2D in Android 4.0" elaborates our solution of hardware-accelerated HTML5 Canvas 2D in Android 4.0 Intel Atom based devices. "DFG JIT on IA32" introduces the details of enabling of DFG JIT on 32-bit Intel architecture platforms. "Conclusion and Future Work" shares our view of future work on both areas.

Graphics Acceleration for HTML5 Canvas 2D in Android 4.0

HTML5 Canvas 2D [13] is one of the most important and mature features in HTML5 and is widely supported by modern mobile devices. HTML5 Canvas 2D allows application developers to invoke a broad range of 2D operations through APIs[14] defined in JavaScript on a canvas defined by the <canvas> tag. Such operations include manipulating images, drawing vector graphics like lines, rendering text, and so on. All of them are essential to a considerable number of typical usage scenarios such as games, photo albums, and visual editing tools.

HTML5 Canvas 2D and Benchmarks

For most games, taking the HTML5 version of the famous Angry Birds as an example, image-related HTML5 Canvas 2D APIs are particularly important, because image operations are typically heavily used in such scenarios and consequently dominate the user experience.

As a result, a few public benchmarks have been created to simulate these use cases and quantitatively measure their performance. FishIETank[15] from Microsoft is one of the most popular of benchmarks. It is widely referenced as a key performance indicator for smart phones and tablets by many third-party publications and sites. The original FishIETank is sensitive to various canvas sizes and also has some run-to-run variance because a random number is used in implementation. We made some slight corresponding modifications to ensure consistent results between multiple runs, with a fixed canvas size at 700x480 for better "apple to apple" comparisons. Hereafter, without specific declaration, the FishIETank discussed in rest of this article refers to the version we modified rather than the original.

GUIMark3[16] is another widely used benchmark in the industry. It contains two image-operation-focused test cases that are similar as FishIETank. In addition, there is also another test case that stresses the vector operations without touching images, like drawing circles.

Somewhat different from these two benchmarks is the Canvas_Perf[17] benchmark, which consists of a few API-level small test cases. It has a fairly broad coverage of HTML5 Canvas APIs. It evaluates the performance of this set of APIs, rather than one or two typical APIs invoked by FishIETank or GUIMark3.

HTML5 Canvas 2D Implementation in Android 4.0

Although the concept of HTML5 Canvas 2D is straightforward to understand, due to the complexity of web runtime, the underlying implementation actually involves a lot of building blocks and the execution flow usually crosses multiple processes and threads. As a good example, Figure 1 illustrates the high level implementation of the HTML5 Canvas 2D in the stock browser of Android 4.0.

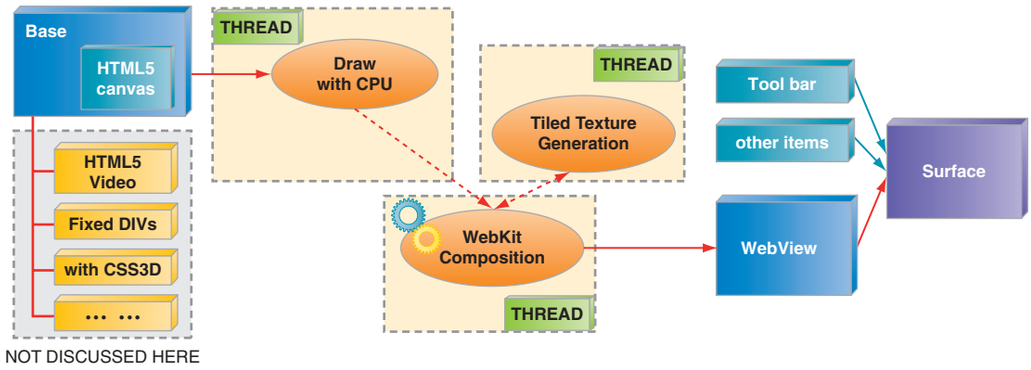

Figure 1: Default implementation of HTML5 Canvas 2D in the browser of Android 4.0 (Source: Intel Corporation, 2012)

In the stock Android browser, there are three different worker threads: the WebViewCore thread, the WebViewMain thread, and the TextureGenerator thread. Each thread serves different purposes.

- The WebViewCore thread generates the contents by utilizing the 2D operations supplied by Skia[18], the 2D graphics engine of Android. Currently, Skia in Android uses the CPU backend for such operations, which means that related calculations of HTML5 Canvas 2D are conducted in the CPU.
- The WebViewMain thread dispatches UI events and merges the generated contents from multiple layers into one single image for the system to display. The latter process is also known as *composition*, and mostly offloaded to GPU for better UI response since Android 4.0.
- As the CPU-generated contents have to be composited by the GPU, the TextureGenerator thread is introduced to convert the image from the bitmap located in CPU memory to the texture that is available for the GPU. As such conversion is time-consuming, the image is split into pieces called *tiles* to reduce the overhead, and only the tile with updates would be converted as necessary.

At least two drawbacks come with this default implementation based on our analysis. Firstly, the CPU is much less efficient than the GPU at generating the contents with typical image operations such as scaling and rotation. Secondly, the overhead of data exchanging between CPU and GPU in texture generation is too significant to ignore. In our tests on an Intel Atom based mobile platform, more than 20 percent CPU utilization is consumed by memory copies associated with this.

Graphics Acceleration of HTML5 Canvas 2D

Graphics acceleration of HTML5 Canvas 2D is a sound approach to address the above issues. By drawing contents with the GPU instead, it would boost the performance of content generation and meanwhile eliminate a large

portion of memory copies because data are already in the GPU. The improved implementation is illustrated in Figure 2. Inspired by Chromium and the implementation of HTML5 Video, we separate out the HTML5 Canvas 2D from other basic web contents and treat it as a standalone layer like HTML5 Video. With this change we are able to substitute the Skia CPU backend with a GPU backend specifically for this canvas layer, thus the GPU can draw the contents in a more efficient and direct way without any further need of texture generation.

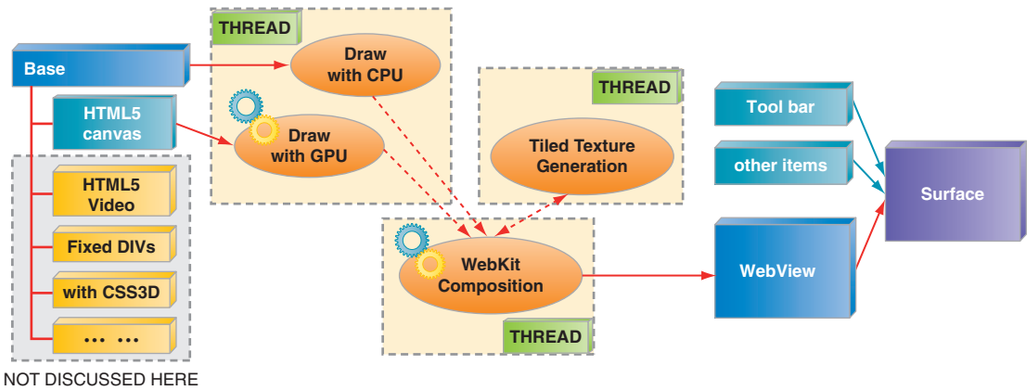

Figure 2: Optimization by using hardware to accelerate the HTML5 Canvas 2D (Source: Intel Corporation, 2012)

As expected, this implementation brings significant performance improvement in image-heavy HTML5 Canvas 2D benchmarks. On the Intel Atom platform, the FPS (frames per second) of the modified FishIETank is increased to as much as three times that of the original solution.

However, it is noticeable that at most 70 percent performance regression is observed on some APIs in the Canvas_Perf benchmark. The analysis shows that GPU acceleration is not always the fastest approach in every HTML5 Canvas 2D API. The Skia CPU backend is better than the GPU backend in certain cases:

- Non-image operations. Vector operations involving multiple vertices can be quite time consuming for the GPU if there is no well-designed 2D graphics unit to support it.
- Certain APIs like GetImageData(), which need access image data by CPU. This would cause remarkable performance regression if it were GPU accelerated, because in most cases, the GPU has to first be synchronized with the CPU and then must copy data to CPU memory.

This is actually one of the reasons why we need to separate out the HTML5 Canvas 2D rather than apply GPU acceleration to entire web contents.

In order to enjoy the benefit of graphics acceleration for image operations without paying the penalty for some inefficient scenarios, we designed and

implemented a mechanism to dynamically switch between the CPU and GPU path with certain heuristics.

As illustrated in Figure 3, execution of each frame would generate some hints such as the performance indicators and a list of HTML5 Canvas 2D APIs touched. The next frame would choose the suitable path through either GPU or CPU, based on these hints and predefined rules. The first frame always goes through the CPU path because there are no hints available at the very beginning.

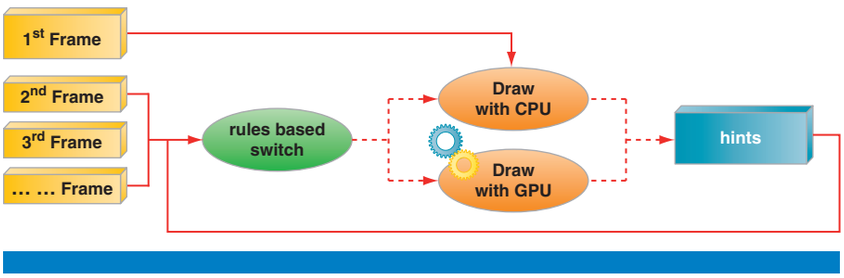

Figure 3: Dynamic GPU and CPU switch for HTML5 Canvas 2D (Source: Intel Corporation, 2012)

Current rules are quite simple and conservative, and aimed to provide an easy solution without regression compared with the default implementation of Android 4.0. The frame would use the GPU path if all of the following requirements were satisfied:

- System does use GPU to composite;
- Graphics context has been initialized;
- At least one image operation is invoked;
- No very GPU-inefficient API is invoked, for example: no GetImageData() or some (not all) vector APIs;
- After switches to GPU from CPU path, it never falls back to CPU path any more.

We regard this implementation based on current rules as fallback solution, because the last rule determines that the GPU path could only be switched at most once. Actually, this is very conservative and leaves much room to improve in the future. Even if the execution falls back to CPU for some reason, it could again be suitable to resume GPU acceleration for later frames. A set of smarter and more aggressive rules are under design and are part of our future work.

Impact of Graphics Acceleration and CPU Fallback

Figure 4 and Figure 5 illustrate the performance impact on an Intel Atom based device with the Canvas_Perf benchmark. For the pure GPU solution, Figure 4 indicates up to 70 percent regression in API hline and vline although it is 5 times faster on image-related operations. With the fallback mechanism applied, all of the regressions are resolved while outstanding boost remains for image operation, as illustrated in Figure 5.

It's worth noting that in Figure 5, the CPU fallback path is still faster than the original solution in the stock Android browser, which is also implemented by using a Skia CPU backend. This is because we always separated out the HTML5 Canvas 2D as a standalone layer, which would eliminate several memory copies and some other overhead, even after fallback to CPU.

Figure 6 and Figure 7 show the performance impact over FishIETank and GUIMark. Up to 3 times higher FPS and 50 percent reduction of CPU utilization can be observed at the same time.

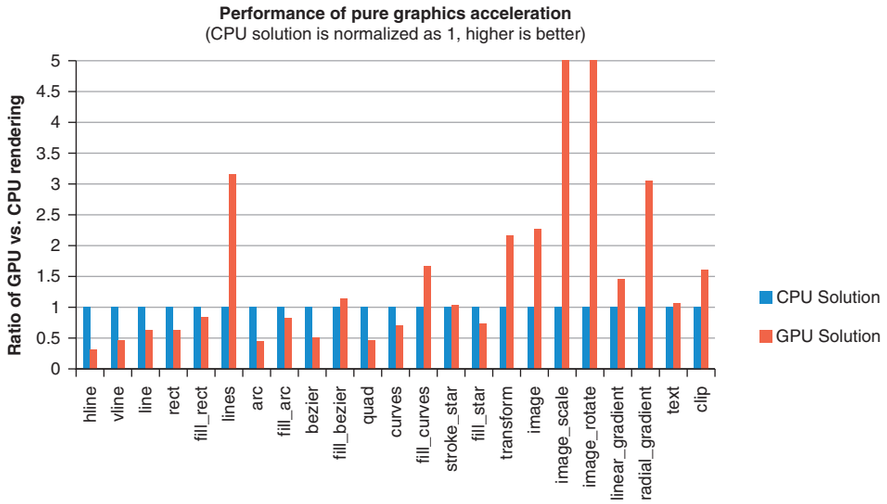

Figure 4: Performance of pure graphics acceleration
(Source: Intel Corporation, 2012)

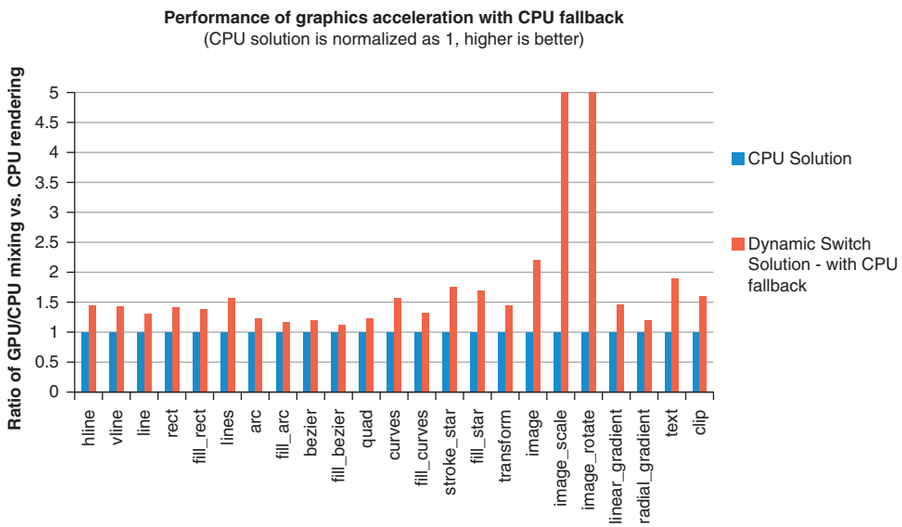

Figure 5: Performance of graphics acceleration with fallback
(Source: Intel Corporation, 2012)

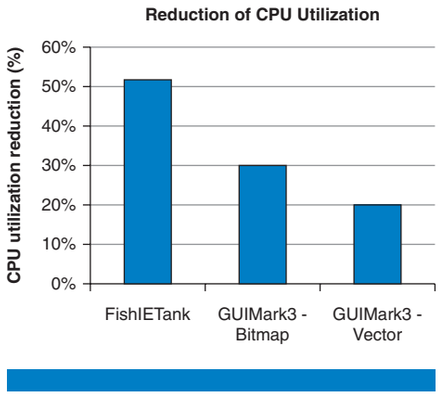

Figure 6: Reduction of CPU utilization (Source: Intel Corporation, 2012)

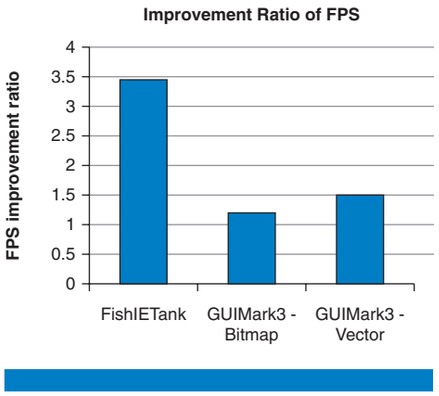

Figure 7: FPS improvement (Source: Intel Corporation, 2012)

DFG JIT on IA32

JIT is a well-proven technology to dramatically improve the runtime performance of language engines. JavaScript, as a dynamic language, however, has more concerns when applying advanced JIT optimizations. In particular, its types can be changed during the execution and thus are not statically determined. Consequently, most of the advanced optimizations based on accurate type information cannot be applied here. As a result, a lot of special handling becomes necessary, including but not limited to efficient object property accessing, type profiling and speculations, and on-stack-replacement (OSR) mechanism to support hotspot optimizations and deoptimizations due to speculation failures, and so on.

JIT technology in JavaScriptCore

There are two types of JITs inside JSC (JavaScriptCore), the default JavaScript engine of WebKit.

- Baseline JIT: The baseline JIT is optimized for compilation speed, which generates native code from the JSC bytecode without any heavy optimizations except for the classic inline caching optimization for dynamic languages. As many existing Web sites don't have much frequently executed JavaScript code, this lightweight JIT is a good fit for fast page loading.
- DFG JIT: The DFG JIT is a highly optimized JIT for better code generation with the tradeoff of compilation speed. It does type speculations based on the type profiling feedback from the baseline JIT, generates SSA-like DFG IR (Intermediate Representation) from the JSC bytecode, performs optimizations including type inference, local CSE (Common Sub-expression Elimination), local register allocation, and so on, and generates optimized native code from the IR.

JSC uses a tiered model to dynamically switch between these two JITs. The JavaScript code is first compiled to generic native code with the baseline JIT before execution. During the generic code execution, a profiler collects the

hotspot information as well as the type information. When a method or a loop is considered to be hot enough, the DFG JIT is triggered to recompile the method and generate optimized code based on type speculations. Possible speculation failures are possible. In such cases, deoptimization happens and the engine falls back to the generic execution. The JIT infrastructure in JSC is described in Figure 8.

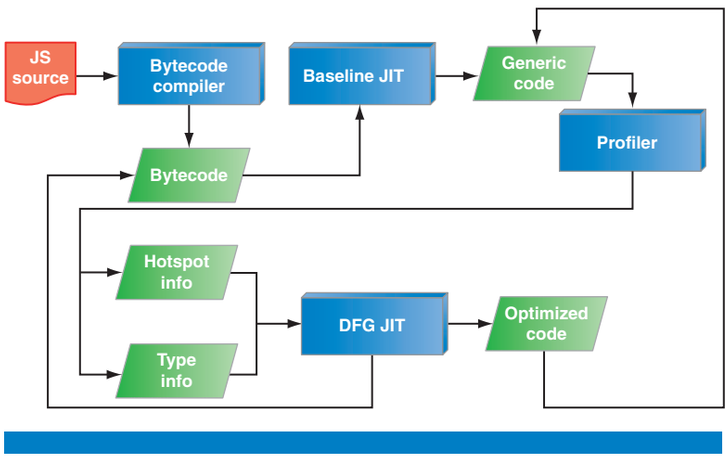

Figure 8: JSC JIT infrastructure
(Source: Intel Corporation, 2012)

It is worth noting that since March 2012, JSC has become a triple-tier VM (virtual machine) on Mac OS X and iOS, which introduces another tier called LLInt (Low Level Interpreter) below the baseline JIT.

JavaScriptCore on Intel® Atom™ with 32-Bit Linux

DFG JIT is really sound, especially for JavaScript heavy web applications, yet when it first came out it only provided the 64-bit version, so the 32-bit mobile systems could not benefit from it. For instance, on the Kraken JavaScript benchmark suite[20], which is constituted of some complex test cases for audio and image processing extracted from rich web applications, JSC without DFG JIT performs 2.5X slower than Google V8 on the Intel Atom platform with Linux. Therefore, we need to do more optimizations to process those programs more efficiently and the DFG JIT is a good fit for this purpose. Furthermore, with the tiered compilation model, the baseline JIT could remain simple enough for minimal compilation overhead. All of these facts tell us we need to bring the DFG JIT technology to more platforms for a broad usage, and our first target is IA32 Linux.

Applying DFG JIT to IA32

Usually in JavaScript engines, some specific data format is defined to represent different JavaScript values including pointers to the objects, integers, doubles, Booleans, undefined, and null values. We call the original values *unboxed* while

the specially formatted ones are referred to as *boxed* JSValues. JSC has two different data formats for 64-bit and 32-bit platforms respectively. Although both formats use a 64-bit length data to represent a JSValue, they have different notations for each bit.

As illustrated in Figure 9, on 32-bit platforms, the higher 32 bits of a JSValue is used as a tag to denote the type, and the lower 32 bits are the payload—the actual value, except for the doubles, which are represented with the total 64 bits. This makes use of unused NaN space for values other than doubles.

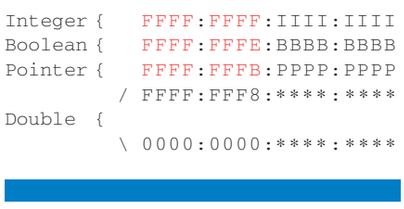

Figure 9: JSC data format for 32-bit platforms
(Source: Intel Corporation, 2012)

The data format for 64-bit platforms also utilizes unused NaN space, while it takes the higher 16 bits to denote the type, as illustrated in Figure 10.

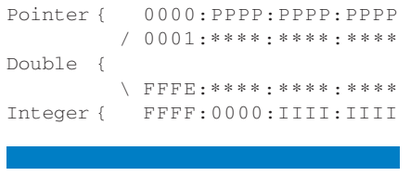

Figure 10: JSC data format for 64bit platforms
(Source: Intel Corporation, 2012)

AU: OK or 64-bit

Considering this background, the major challenge of enabling DFG JIT on 32-bit platforms is to ensure that DFG JIT is able to recognize a totally different data format. It's unfortunate that we need to write the same logic twice for those operations on boxed JSValues, one for 64-bit, which is already there, and the other for 32-bit to be added by us. On the other hand, it's fortunate that the DFG JIT is a type-speculative JIT and many operations can be performed on unboxed values, which can be shared between 64-bit and 32-bit.

Compared with x64, one major problem of x86 is the shortage of registers. There are only eight GPRs (general purpose registers) available on x86. In JSC, three of them are already reserved for special purposes so only five are left. In order to address the register pressure problem on x86, we need to have more values to be speculated and represented with unboxed 32-bit values. For example, the Boolean speculation for 32-bit is different from that for 64-bit, which uses a single 32-bit GPR to hold the unboxed Boolean value instead of two GPRs for a boxed JSBoolean. This design choice is made not only for performance, but also to save a register.

We have JS values and sometimes we need to speculate that a JS value is a specific type so that we can generate more efficient code. This inevitably involves value boxing and unboxing. Now considering the simple case where the value is in registers, since the JS values are 64-bit long values, we need to use two general purpose registers to represent the JS value on 32-bit platforms—one for the payload and the other for the tag. Unboxing a JS integer, Boolean, or pointer is cheap by simply adopting the register holding the payload. Similarly, boxing an integer, Boolean, or pointer just needs to fill the tag register with the correct data. What makes things more complex, however, is how we do conversion between JS doubles and unboxed doubles. Bear in mind that the JS double is in two general purpose registers and the unboxed double is in one floating point register. To do the conversion, one straightforward approach is to exchange the data through memory, while it results in very bad performance. On the other hand, we can exploit SSE2 packed data support to perform efficient conversion if the unboxed double is

actually in a XMM register. For example, the double boxing and unboxing can be performed using the below sequence respectively.

```
movd xmm0, eax
psrlq 32, xmm0

movd xmm0, edx
```

Code 1: Boxing a double value
Source: Intel Corporation, 2012

```
movd eax, xmm0
movd edx, xmm1
psllq 32, xmm1
por xmm1, xmm0
```

Code 2: Unboxing a JS double
Source: Intel Corporation, 2012

In this example, we suppose *eax* holds the payload of the JS double and *edx* holds the tag of it. Furthermore, *xmm0* holds the unboxed double value and *xmm1* is for temporary usage purpose. The example shows that the conversions between JS double and unboxed double no longer need to involve memory access, and as a proof, we get a 77-percent performance boost for the Kraken benchmark on IA32 by this approach. Over the long term, we may further improve double conversions if we know a JS value is a JS double at compile time and directly represent it with a FPR (floating pointer register) instead of two GPRs, though it may introduce additional complexities to the code generation logic in DFG.

Besides normal DFG JIT code generation, the different data format also impacts the deoptimization caused by speculation failures in DFG JIT. The generic code to be switched to always assumes that boxed JS values are in memory. However in fact, the DFG JIT code can produce unboxed values in both memory and registers. Consequently, when falling back to the generic code, we have to perform necessary data boxing and data transfers between memory and registers.

The calling conventions for x86 differ from those of x64, and in fact there are different conventions for different operating systems on x86. So another thing we need to do is to teach the DFG JIT the different calling conventions, which is necessary as we have some runtime helper functions invoked by the JIT code. We now support the cdecl calling convention for x86 in DFG, different from the fastcall support in the baseline JIT. The helper function call interfaces are also redesigned with the help from the community to support more different calling conventions easily.

Impact of DFG JIT on IA32

Figure 11 clearly reveals the performance improvement due to the enabling of DFG JIT. Nearly 2X improvement on IA32 for Kraken benchmark is observed on the November 2011 code base when we eventually finished our

enabling. Furthermore, our further collaboration with the community on the optimizations over the DFG JIT results in more performance improvement from November 2011 through March 2012.

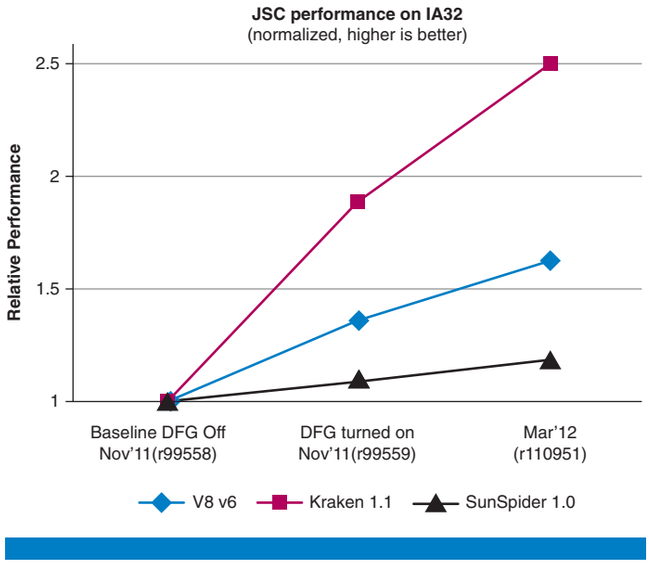

Figure 11: JSC performance on IA32
(Source: Intel Corporation, 2012)

Conclusion and Future Work

HTML5 is an inevitable trend of future web applications. It brings massive opportunities while it results in novel challenges in performance as well. Therefore, optimizations of web runtime become essential in the journey towards embracing HTML5. This article shares two important technologies, graphics acceleration and JIT, on the rendering engine and JavaScript engine respectively, with prominent performance gain achieved on Intel Atom based platforms.

Looking forward, several further improvements are pipelined as our future work:

- As mentioned in the section "Graphics Acceleration of HTML5 Canvas 2D," a more aggressive set of rules to switch between CPU and GPU path is under design. We believe that the new design would maximize the opportunity to utilize graphics acceleration and bring the performance of HTML5 Canvas 2D to next level.
- We also have a few ideas to improve the Skia GPU backend. This would mitigate some inefficient implementation of certain vector-related APIs on the GPU.
- Regarding JSC JavaScript engine, it's IR of DFG JIT is still quite simple and lacks of some advanced features. We are working together

with community engineers to improve this. With a more powerful implementation, adding a lot of typical JIT optimizations over JSC is expected to be much easier.

As graphics acceleration and advanced JIT are compelling technologies with high potential; we are exploring the feasiblity of applying them to the implementation of other emerging web technologies such as CSS3 and WebGL, too.

Complete References

[1] HTML5 specification (draft 25), World Wide Web Consortium (W3C), May 2011. *http://www.w3.org/TR/2011/WD-html5-20110525/*

[2] Processor family for mobile segments, Intel. *http://www.intel.com/content/www/us/en/processors/atom/atom-processor.html*

[3] Gartner, 2011. *http://www.gartner.com/it/page.jsp?id=1826214*

[4] HTML5 Angry Birds, Rovio Entertainments Ltd. *http://chrome.angrybirds.com/*

[5] Google to use HTML5 in Gmail. *http://www.computerworld.com/s/article/9178558/Google_to_use_HTML5_in_Gmail?taxonomyId=11&pageNumber=2*

[6] Brad Neuberg, Google. Introduction to HTML5. *http://googlecode.blogspot.com/2009/09/video-introduction-to-html-5.html*

[7] Mobile operating system, Google. *http://www.android.com/*

[8] Open source project for browser and mobile OS, Google. *http://www.chromium.org/*

[9] A New Crankshaft for V8, Google. *http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html*

[10] V8 JavaScript Engine. *http://code.google.com/p/v8/*

[11] V8 Benchmark Suite Version 6. *http://v8.googlecode.com/svn/data/benchmarks/v6/run.html*

[12] The WebKit Open Source Project. *http://www.webkit.org/*

[13] HTML Canvas 2D Context, World Wide Web Consortium (W3C). *http://www.w3.org/TR/2dcontext/*

[14] JavaScript Web API standards, World Wide Web Consortium (W3C). *http://www.w3.org/standards/webdesign/script*

[15] FishIETank workload, Microsoft. *http://ie.microsoft.com/testdrive/Performance/FishIETank/Default.html*

[16] Browser graphics benchmark, Sean Christmann. ***http://www.craftymind.com/guimark3/***

[17] Browser graphics benchmark, Hatena. ***http://flashcanvas.net/examples/dl.dropbox.com/u/1865210/mindcat/canvas_perf.html***

[18] Open project as 2D graphics engine, Google. ***http://code.google.com/p/skia/***

[19] SunSpider JavaScript Benchmark. ***http://www.webkit.org/perf/sunspider/sunspider.html***

[20] Kraken JavaScript Benchmark. ***http://kraken-mirror.googlecode.com/svn/trunk/kraken/hosted/index.html***

Author Biographies

Jonathan Ding (jonathan.ding@intel.com) is a software engineer in the Software and Services Group of Intel. His expertise covers browser, web runtime, HTML5, and related service frameworks.

Yuqiang Xian (yuqiang.xian@intel.com) is a software engineer in the Software and Services Group of Intel. His major interests include compilers and virtual machines.

Yongnian Le (yongnian.le@intel.com) is a software engineer in the Software and Services Group of Intel. He currently focuses on browser-related analysis and optimizations, in particular on Android mobile platforms.

Kangyuan Shu (kangyuan.shu@intel.com) is a software engineer in the Software and Services Group of Intel. His interests include Android and graphic subsystems.

Haili Zhang (haili.zhang@intel.com) is a software engineer in the Software and Services Group of Intel. His expertise covers HTML5 and open web platform related application frameworks and tools.

Jason Zhu (jason.zhu@intel.com) is a software engineer in the Software and Services Group of Intel. His interests include advanced and emerging web technologies and innovations.