Tillämpning av WebAssembly på webben

Tom Yrjas, 41463
Kandidatavhandling i datavetenskap
Handledare: Jerker Björkqvist
Fakulteten för naturvetenskaper och teknik
Åbo Akademi
2020

Referat

Webben utvecklas konstant för att uppfylla de särskilda behov som företag och organisationer har för sina produkter och tjänster. Avsikten med WebAssembly är att utvidga vad som är rimligt att implementera i webben, och med syfte att tyda WebAssemblys roll i webben har detta arbete skapats. Detta utförs genom att granska olika befintliga tillämpningar av WebAssembly.

Arbetets resultat visar hur WebAssembly excellerar i prestandakonsekventa program jämfört med Javascript. Samt WebAssemblys övriga fördelar över Javascript, som multitrådning, obfuskering och sitt kompakta format.

Innehållsförteckning

1 Inledning ... 1
2 Händelseförlopp ... 1
3 WebAssembly ... 2
3.1 Emscripten och LLVM ... 3
3.2 Webbläsaren och WebAssembly ... 4
3.3 WebAssembly versus Javascript ... 5
4 Tillämpningar ... 8
4.1 Google Earth Web ... 8
4.2 AutoCAD ... 9
4.3 Blazor ... 11
4.3.1 Modeller ... 12
4.4 eBay ... 12
4.5 Unity ... 13
4.6 ViennaTS ... 14
4.7 Kryptovaluta-utvinning i webben ... 15
5 WebAssemblys företrädare ... 16
6 Framtida utveckling på WebAssembly ... 18
7 Slutsats ... 19
8 Referenser ... 21

1 Inledning

Detta arbete introducerar hur WebAssembly kan tillämpas, samt hur det fungerar. Webben är i konstant utveckling för att uppfylla de särskilda behov som företag och organisationer har. WebAssembly är en lösning på en delmängd av problem vilket går utanför Javascripts och olika baksystemslösningars förmågor och duglighet, därför kommer detta arbete redogöra hur olika företag har tillämpat WebAssembly som en lösning i deras område, vilket inte var rimligt före.

WebAssembly är ett säkert, portabelt binärformat gjord för effektiv exekvering, och är också plattform oberoende, men primärt avsett för webbläsare. (W3C, 2019)

Det begränsar sig själv inte till något språk, utan med olika verktygskedjor som Emscripten genereras WebAssembly filer (.wasm) från alla språk som kan kompileras till LLVM (Low Level Virtual Machine). WebAssembly-filerna laddas in till exempel på webbsidan, instansieras, och sen kan funktioner anropas till WebAssembly genom Javascript. (Haas, et al., 2018)

WebAssembly är senaste lösningen på det som andra företag redan försökt, det vill säga att möjliggöra resurskrävande program i webbläsaren genom andra programmeringsspråk. De har försökt göra detta med till exempel Microsoft Silverlight, Adobe Flash eller Google Native Client, och till skillnad från dessa har nu företagen istället kommit tillsammans för att förverkliga det. Detta bidrar till WebAssemblys intresse samt succé, eftersom de största webbläsare-leverantörerna stöder och aktivt deltar i utvecklingen av WebAssembly.

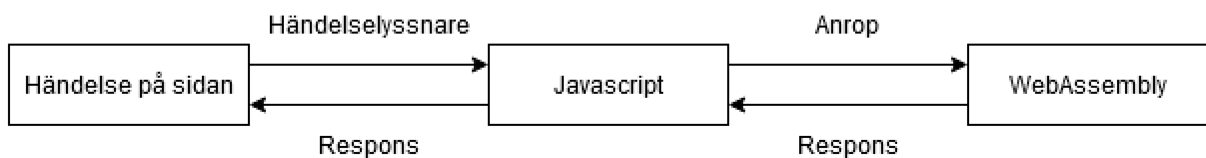
2 Händelseförlopp

I juni 2015 skrev Luke Wagner i sin blogg att Mozilla, tillsammans med Google, Microsoft och Apple har påbörjat en ny standard kallad WebAssembly, som ”definierar ett portabelt, storleks- och belastningseffektivt format och exekveringsmodell specifikt utformad för att tjäna som ett samlingsmål för webben” (Wagner, 2015). Från och med september 2017 har standarden börjat stödas av alla större webbläsare. (se figur 1, kvadraternas nummer representerar webbläsarnas versioner) (Deveria, 2020) I december 2019 blev WebAssembly en W3C rekommendation. (W3C, 2019)

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser
		2-46						
	12-14	47-51	4-50		10-37			
	15	52	51-56	3.1-10.1	38-43	3.2-10.3		
6-10	16-79	53-71	57-79	11-12.1	44-65	11-13.1		2.1-4.4.4
11	80	72	80	13	66	13.2	all	76
		73-74	81-83	TP		13.3		

Figur 1, https://caniuse.com/#feat=wasm

3 WebAssembly



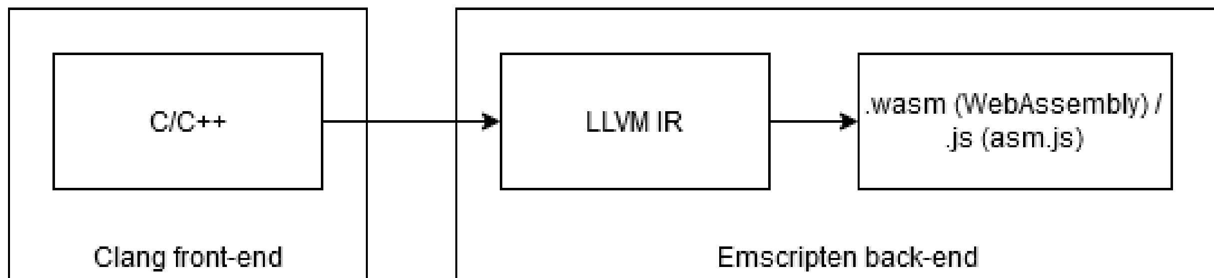
Figur 2

Avsikten med WebAssembly är inte att ersätta Javascript, snarare att komplementera det och utvidga vad som är rimligt att implementera i en webbläsare.

För närvarande krävs Javascript *limkod* (glue code) för att instansiera WebAssembly-filer i webbläsaren. Samt på grund av WebAssemblys natur är det inte möjligt att manipulera dokumentobjektsmodellen (DOM), det vill säga strukturen på sidan. Det är inte heller möjligt att lyssna för händelser. För att hantera dessa gränssnitts-relaterande operationer krävs Javascript vilket då anropar de funktioner som är exponerade i WebAssembly filerna. (se figur 2)

Med WebAssembly används klientens resurser istället för serverns, lika som Javascript. Dock, ifall man vill utnyttja serverns resurser istället är detta också möjligt med WebAssembly, detta beskrivs mer om i kapitel 4.1.

3.1 Emscripten och LLVM



Figur 3

Emscripten är en källa-till-källa kompilator som konverterar C eller C++ till Asm.js eller WebAssembly. Initialt skapat för Asm.js, där principen är att C eller C++ blir med Emscripten kompilerad till Javascript. (se figur 3) Lika som Asm.js, så används Emscripten också för att skapa WebAssemblys binärfiler (se figur 3), där principen är det samma. Från C eller C++ kompileras koden till LLVM (Low Level Virtual Machine) maskinkod (Intermediär representation), och sedan med Emscripten kompileras det till WebAssemblys binärformat eller till Asm.js:s format. (Zakai, 2011)

Orsaken att LLVM:s mellansteg finns är för att alla språk som kan kompileras till LLVM:s maskinkod kan då också kompileras till WebAssembly eller Asm.js. Ifall detta inte vore fallet skulle en kompilator per språk måste skapas för WebAssembly och Asm.js. (LLVM stöder kompilering från till exempel Ada, D, Delphi, Fortran, Haskell, Scala, Objective-C, Rust och Swift.)

Nu tack vare WebAssembly är Asm.js i princip irrelevant, Eftersom de utför samma sak, medan WebAssembly är snabbare och per kompakt (binärt format). Dessutom har Asm.js olika icke-optimala sätt för att möjliggöra olika kant-fall vilket påverkar prestandan negativt. (Zakai, 2017)

<table>
<tr><th>C</th><th>Binär</th><th>Text</th></tr>
<tr><td><pre>int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n - 1);
}</pre></td><td>20 00
42 00
51
04 7e
42 01
05
20 00
20 00
42 01
7d
10 00
7e
0b</td><td>get_local 0
i64.const 0
i64.eq
if i64
  i64.const 1
else
  get_local 0
  get_local 0
  i64.const 1
  i64.sub
  call 0
  i64.mul
end</td></tr>
</table>

Tabell 1, C samt dess binära och textuella representation (Golsch, 2019)

I tabell 1 visas C kod och dess binära samt textuella representation, efter det har kompilerats till WebAssembly med Emscripten. Eftersom binära formatet är mest kompakt används det då WebAssembly hämtas och instansieras i webbläsaren. (se kapitel 3.2) WebAssemblys textuella representation (.wat), finns för att främja felsökning, samt att möjliggöra manuell programmering. (Golsch, 2019) Textuella representationen kan konverteras till binära formatet, och binära formatet kan också konverteras till textuella representationen.

3.2 Webbläsaren och WebAssembly

WebAssembly kan inte exekveras självständigt, utan måste bli inbäddad på en plattform som kan instansiera det. (Golsch, 2019) Webbläsarnas gränssnitt till WebAssembly är implementerat i deras Javascriptmotor, till exempel Firefoxs SpiderMonkey eller Chromes V8. Detta betyder att WebAssembly exekveras inuti webbläsarnas minnessäkra Javascript sandlåda (sandbox), samt att WebAssemblys förmågor är begränsade till sandlådan. (Golsch, 2019)

I figur 4 visas ett exempel på hur WebAssembly kan laddas in och instansieras. *Fetch()* hämtar WebAssembly-filen, och *arrayBuffer()* kallas på det, vilket returnerar ett *ArrayBuffer*-objekt (används för att representera binär data). Detta passeras till *instantiate()* (*WebAssembly*-objektet har ingen konstruktör) i *WebAssembly*-objektet, vilket returnerar en instans som har tillgång till WebAssembly-filens exporterade funktioner. (W3C & Ehrenberg, 2019)

```
fetch("../out/main.wasm").then(response =>
   response.arrayBuffer()
).then(bytes =>
   WebAssembly.instantiate(bytes)
).then(results => {
   console.log(results.instance.exports.myFunction());
}).catch(console.error);
```

Figur 4, Javascript limkod. Instansiering av WebAssembly, samt ett anrop till exporterade funktionen myFunction()

I exemplet används *WebAssembly.instantiate()*, men *WebAssembly.instantiateStreaming()* kan också användas, vilket hämtar, kompilerar och instansierar filerna medan de laddas ned. (Mozilla, 2020)

Eftersom Javascript exekveras sekventiellt, beroende på storleken av WebAssembly-filerna, eller tyngden av uppgifterna som skall utföras, bör detta oftast utföras i asynkron, vilket kan uppnås med webbarbetare (Web Worker). Ifall det inte utförs med en webbarbetare kan interaktionen med sidan blockeras medan det exekveras.

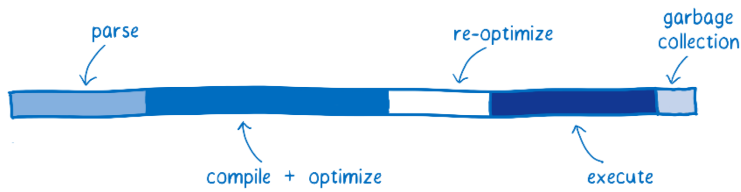
3.3 WebAssembly versus Javascript

WebAssemblys prestanda jämfört med Javascript är ett mångsidigt ämne. Till att börja med, primära skillnaden mellan hur WebAssembly och Javascript körs är att WebAssembly kompileras Ahead-of-time (AOT), medan Javascript kompileras och exekveras på samma gång, det vill säga Just-in-time (JIT).

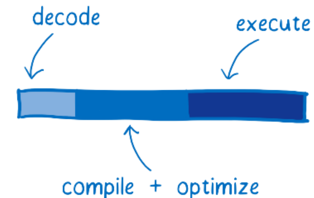
Prestandan för JIT är oförutsägbar på grund av hur det fungerar. Det användar olika metoder för att få bästa möjliga prestanda. Till exempel samlar det in information om koden medan det körs, och det som körs ofta markeras som ”het” kod. Då något har markerats som ”het” kompileras den och nästa gång den ska köras körs den ännu snabbare. Detta sparas tills den blir ”kall”, eller efter den blir bortknuffad av någon annan varmare kod. (Clark, 2017) Beroende på vilken kod som körs, vad annat som körs medan denna kod körs, samt hur det körs kan göra Javascript väldigt snabb eller långsam, på samma kod. Övriga faktorer som skräpsamlare (garbage collector) kan påverka, samt Javascripts automatiska förenkling av anrop. (Rossberg, 2018)

AOT kompileras dock komplett före det exekveras, och i kompileringsskedet kan olika optimeringar utföras (vilket i WebAssemblys fall utförs i Emscriptens verktygskedja). På grund av detta är AOT:s prestanda förutsägbar, vilket är viktigt i applikationer som är resursintensiva.

Följden av detta är att då prestanda mellan WebAssembly och Javascript mäts, är resultaten oftast varierande, och att mäta specifika kodsnuttar mellan dem kan skeva resultatet. (Rossberg, 2018) (Eberhardt, 2017) (Zakai, 2017) (Gurgone & Spless, 2018) (Chen, 2018)



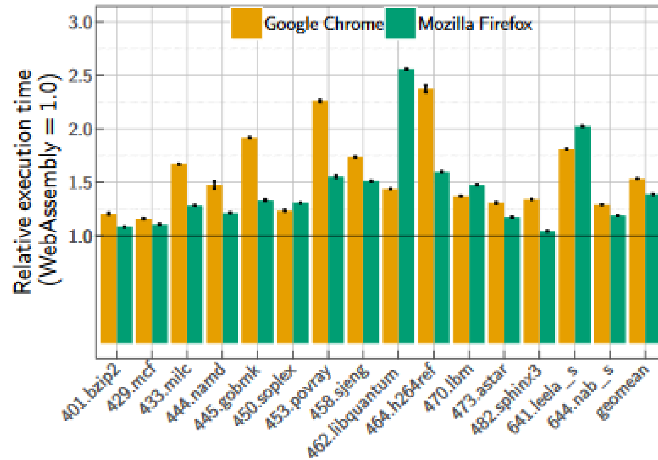
Figur 5 (Clark, 2017)



Figur 6 (Clark, 2017)

Förutom snabbheten har WebAssembly andra fördelar. Figur 5 representerar hur Javascript exekveras tillsammans med JIT, och figur 6 representerar hur WebAssembly exekveras som AOT. WebAssembly – som redan är kompilerat och i ett binärt format – behöver endast avkodas till ett abstrakt syntaxträd som översätts till bytekod som webbläsarens Javascript-motor förstår. Efter det kompileras och optimeras bytekoden, och efter det kan den exekveras. (Clark, 2017) (Haas, et al., 2018)

WebAssembly kommer också kunna utnyttja olika metodologier för att utföra saker effektivt, som LLVM-kompatibla språk redan använder. Till exempel multitrådning, SIMD (singulär instruktion, multipla data), MIMD (multipla instruktioner, multipla data) och andra parallelliseringssätt. Dock för tillfället är sådant under utveckling eller inaktiverat på olika webbläsare. (se kapitel 6) I kontrast, Javascript är enkeltrådat och stöder inte multitrådning – dock är webbarbetare, som beskrivs i kapitel 4.2, en typ av multitrådning. Det Javascript dock stöder är att utföra uppgifter i asynkron, det vill säga att då något som utförs är beroende på något annat defineras en tillbakaropning som exekveras då det som det var beroende på är klart. Dock exekveras detta fortfarande sekventiellt.



Figur 7, SPEC CPU prestandatest (Jangda, et al., 2019)

(Jangda, et al., 2019) jämförde WebAssemblys prestanda med Asm.js. De jämför det med Asm.js eftersom då kan samma kod användas, men endast kompileras till separata mål. De använde *SPEC*, som är en organisation som etabliserat och upprätthåller standardiserade tester och verktyg för att utvärdera prestanda. Figur 7 visualiserar deras resultat. Svarta linjen som går igenom hela diagrammet är hur WebAssembly presterade. De fann att WebAssembly är snabbare med en faktor på 1.39 i Firefox jämfört med Asm.js, och 1.54x snabbare i Chrome jämfört med Asm.js. Dessutom att i allmänhet är WebAssembly 1.3x snabbare än Asm.js. (Haas, et al., 2018) rapporterade liknande, där de använde en samling liknande tester som SPEC, som istället heter PolyBenchC. Med detta fann de att WebAssembly är 33,7% snabbare i medelvärde jämfört med Asm.js.

(Fitzgerald, 2018) utvärderade också WebAssemblys prestanda jämfört med Javascript (utan Asm.js) genom att partiellt implementera Javascripts *source-map* parser med WebAssembly kompilerad från Rust. (Fitzgerald, 2018) fann att ”WebAssembly är upp till 5.89 gånger snabbare än Javascripts implementation”, samt att ”prestandan är också mer konsekvent: relativa standardavvikelser förminskades.”

(Herrera, et al., 2018) jämförde WebAssembly och Javascript på personliga datorer, mobiltelefoner och läsplattor. De fann att ”alla webbläsare demonstrerar signifikanta prestandaförbättringar för WebAssembly, inom intervallet av 2x ökning över webbläsarens samma Javascript motor.”.

4 Tillämpningar

Företag och organisationer utnyttjar WebAssembly för vad deras webbsidor specifikt kräver. Till exempel, i kapital 5.1 beskrivs Google Earth Web, som fullständigt baserar sig på sin föregående klient-baserade applikationen, Google Earth. Men, i kapitel 4.2 beskrivs AutoCAD; som istället har valt att partiellt implementera sin renderingsmotor i WebAssembly, vilket ger användarna möjlighet att visualisera ritningarna de har skapat, men stöder inte samma nivå av manipulation av ritningen som deras applikation gör. Och till skillnad från dessa två – vars hela sida är beroende på WebAssembly – har eBay använt WebAssembly enskilt för en föga streckkodssläsare för att möjlighetsgöra större sannolikhet att den lyckas.

Avsikten med detta kapitel är att redogöra hur olika organisationer använder WebAssembly, och granska hur det uppfyller deras målsättningar för produkten. Även fast liknande produkter är möjliga utan WebAssembly, skulle det vara orimligt eller olönsamt att utföra samma sak i till exempel Javascript.

4.1 Google Earth Web

Google Earth är ett program som simulerar ett jordklot baserat på satellitbilder, där användare kan förflytta sig eller zooma in och ut för att utforska jorden. Det är liknande till Google Maps, men Google Maps fokuserar på navigation medan Google Earth är fokuserat på forskning, utbildning och kartläggning. Det ger också ut data på klimatet, vädret, och annan geospatial information.

Det lanserades initialt 2001 och efter system som iOS, Android blev populära anpassade Google kodbasen för att stöda dessa. Tillsammans med Google Native Client (NaCl), som är liknande teknik som WebAssembly, tog de Google Earth till webben i 2017.

NaCl möjliggör att köra LLVM-kompatibla programmeringsspråk i webbläsaren i en isolerad miljö, lika som WebAssembly. Detta skapades dock enskilt av Google med Chrome i tanke, därav valde övriga webbläsare att inte implementera det av flera orsaker. (Zbarsky, 2015) (se kapitel 5) På grund av det lanserades aldrig Google Earth i webben med NaCl, även fast de fick det att fungera. (Mears, 2019) I samma skede utfasades också NaCl för att växla fokus till WebAssembly. (Nelson, 2017)

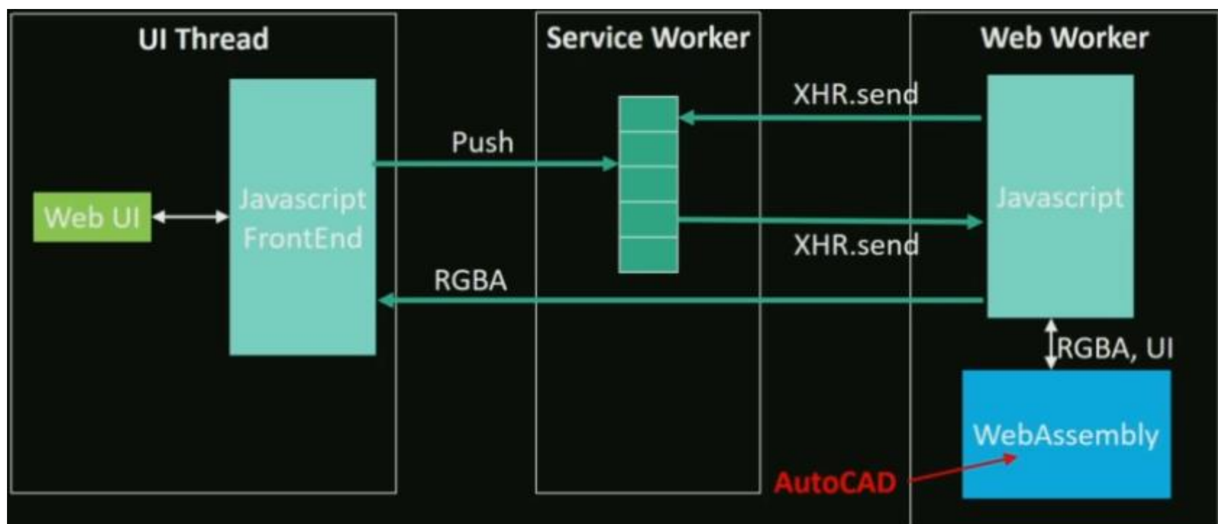
Google Earths kodbas är baserat på C++, och använder Qt – ett plattformsoberoende C++-baserat ramverk för grafiska gränssnitt – samt ett ytterligare projekt kallat Google Ion vilket är partiellt en abstraktion över OpenGL. (Mears, 2019) För att stöda alla plattformar som Google vill så används samma kodbas som är fullt abstraherad, och de behöver endast anpassa applikationens framända till plattformen. Dock, då de anpassade Google Earth Web – då baserat på NaCl – hade de redan framändan, och på grund av applikationens djupa abstraktion behövde de endast kompilera kodbasen till WebAssembly och ersätta NaCl:s kopplingar med WebAssembly. (Mears, 2019)

Google Earth är beroende på nätverksöverföringar. Medan användaren förflyttar sig på jordklotet hämtas bilder som dekomprimeras och sen renderas på skärmen. Att utföra detta är intensivt, därför är det viktigt för Google Earth Web att WebAssembly stöder olika metodologier som kan främja prestanda, som SIMD (singulär instruktion, multipla data, se kapitel 6), samt multitrådning genom *SharedArrayBuffer*, vilket är vilket är inaktiverat sedan 2018 på grund av gren-spekulations-säkerhetsproblemet Spectre. (Mozilla, 2020)

(Mears, 2019) förklarar att orsaken de lyfte fram Google Earth till webben är för att göra det mer tillgängligt för användare. Det är förnuftigt, att i jämförelse med tidigare, då användarna måste ha en separat klient nerladdad och installerad, är det nu simplare att använda Google Earth genom att endast gå till en webbsida. För användare är det bekvämare, och för vissa kan det kännas säkrare eftersom de inte behöver ladda ned något på sin dator, och det är dessutom mer intuitivt användbart på grund av denna tillgänglighet. Dessutom, ifall de skulle ha fortsatt med NaCl skulle de begränsat sina användare till Chrome.

4.2 AutoCAD

AutoCAD är ett designverktyg som släpptes i början av 1980-talet för att skapa ritningar i 2D eller 3D, och sedan 2018 har det funnits tillgängligt i webben. AutoCADs mjukvara har alltid primärt blivit utvecklad för Windows, därav har stöd för övriga operativsystem varit oregelbundet. Före WebAssembly satsade de på att skriva om kodbasen till Java för kors-plattformsstöd, men då Emscripten och Asm.js kom skrotades detta. Istället påbörjade de anpassa AutoCADs originella kodbas att kompileras till Asm.js, och nu WebAssembly. Tack vare detta slipper de upprätthålla två separata kodbaser, utan de kan istället bara kompilera sin enskilda kodbas till sina respektive mål. (Cheung, 2019)



Figur 8 (Cheung, 2018)

AutoCAD kräver ett komplext användargränssnitt anpassad till de förmågor programmet möjliggör. På figur 8 visualiseras hur AutoCAD är implementera tillsammans med UI (user interface, användargränssnitt), tjänstarbetare och webbarbetare.

WebAssembly erbjuder inga lösningar för att manipulera dokumentobjektsmodellen (DOM), utan det behövs fortfarande ett Javascript frontsystem vilket hanterar händelser och lyssnar för förändringar.

Eftersom Javascript exekveras sekventiellt behövs en eller flera webbarbetare (Web Worker) för att kunna utföra uppgifter med flera trådar.

Ifall WebAssemblys binärfiler instansieras utan webbarbetare blockeras renderingen av sidan medan detta utförs, det vill säga all möjlig interaktion som kan utföras är oanvändbar för den tid som binärfilerna instansieras. Detta kan pågå från någon millisekund till flera sekunder, beroende på filernas storlek. Därför instansierar AutoCAD WebAssembly i en webbarbetare. AutoCAD utnyttjar också tjänstarbetare (Service Worker). De används som mellanprogram för dataöverföring för att temporärt lagra regelbundet använd data (Cache), samt kan tjänstarbetare utnyttjas för att utföra nätverkskommunikation i bakgrunden eftersom det körs på en separat tråd.

Denna anläggning av olika tekniker är inte unik till AutoCAD, och andra tillämpningar av WebAssembly bör vara generellt liknande.

4.3 Blazor

Microsoft har integrerat WebAssembly i deras egna .NET ramverk. Att köra annan kod än Javascript är en idé som Microsoft har haft sen 2007 då de lanserade Silverlight. Men eftersom Silverlight är beroende på funktionalitet som måste installeras separat från webbläsaren blir kompatibilitet ett problem med andra operativsystem och webbläsare, därav dog det sakta ut. (se kapitel 5)

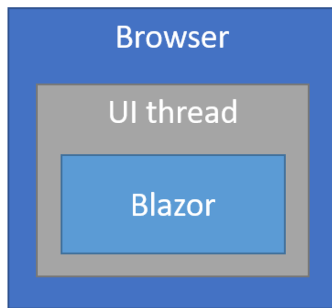
```
<h1>Counter</h1>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Figur 9. Blazor och Razor. (Microsoft, 2020)

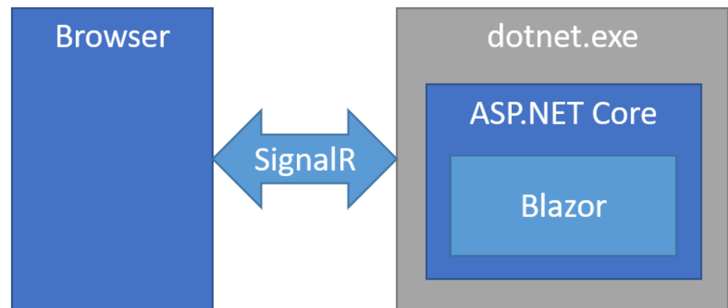
Nu med WebAssembly kan de uppfylla den poäng som Silverlight hade. Tillsammans med Blazor kommer Razor, ett komponent-baserat syntax-markeringsspråk med en vy-motor för att skapa dynamiska sidor. Det vill säga, HTML med logik som är processad på servern innan den serveras till klienten. (se figur 9) (Anderson & Nowak, 2020) Eftersom WebAssembly och Javascript är interoperabla är det möjligt med Blazor och Razor att definiera händelser och hantera dem utan att skriva Javascript kod. (Blazor tar hand om det)

I figur 9 visas ett exempel hur Blazor (WebAssembly) integreras i Razors templatfiler, samt hur händelsehantering utförs. På *<button>* definieras attributen *@onclick* att anropa funktionen *IncrementCount(),* som definieras i *@code* blocket. Utanför .NET:s ekosystem måste händelsen definieras i Javascript vilket då instansierar WebAssembly och anropar funktionen. (se figur 2) Medan i .NET utför Blazor och Razor detta automatiskt. (Nelson, et al., 2020)

4.3.1 Modeller



Figur 10 (Roth, 2020)



Figur 11 (Roth, 2020)

Eftersom WebAssembly är portabelt måste det inte köras inuti en webbläsare. Det kan instansieras på alla typer av plattformar så länge det finns ett gränssnitt för att hantera WebAssemblys filer. På grund av detta finns det en frihet hur system kan byggas upp tillsammans med WebAssembly.

På figur 10 representeras den allmänt använda modellen. Blazor (WebAssembly) körs i webbläsaren på klientens dator, och beräkningarna utförs också på klientens dator. (Roth, 2020) förklarar en alternativ modell, där Blazor körs på servern istället för i webbläsaren. (figur 11) Tillsammans med *SignalR*, ett bibliotek inom .NET ramverket vilket kan hantera all klient-serverkommunikation, är det möjligt att skapa ett gränssnitt där all interaktion på sidan belastar servern istället för klientens dator.

Webbsidor med denna modell är viktlös, som inte är begränsad till webbläsarens kapacitet eller datorns. Förstås är då webbsidan beroende på nätanslutning, samt belastas servern intensivt då den måste rendera webbsidan och hålla reda på alla klienters stat. (Roth, 2020)

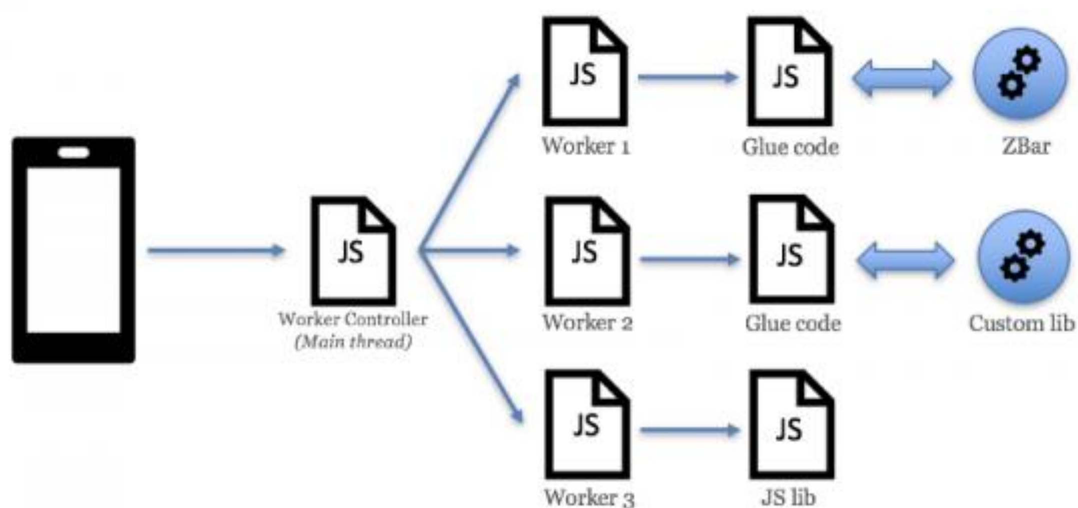
Bägge modellerna kan också användas på grund av WebAssemblys portabilitet. Ifall det som skall utföras ses som för intensivt eller tidskrävande är det möjligt att instansiera samma WebAssembly filer i baksystemet och exekvera dem där istället. I Blazor kallas detta delad logik.

4.4 eBay

Försäljare som lägger till sina produkter på eBay kan göra det genom att manuellt mata in uppgifterna, eller genom att använda en streckkodsläsare på sin telefon med kameran. Eftersom streckkodsläsaren är skriven i C++ kunde den implementeras direkt till eBays

mobilapplikation, men för försäljare som föredrar att använda webbläsaren på sin telefon var det inte möjligt att använda samma C++ kod för att skanna streckkoderna, före WebAssembly.

Före WebAssembly experimenterade eBay med ett Javascript bibliotek kallad BarcodeReader, men eftersom det fungerade inkonsekvent föredrog de att inaktivera hela funktionaliteten än att erbjuda opålitlig funktionalitet. Dock, då de implementerade sin egen C++ streckkodsläsare i WebAssembly var den också opålitlig eftersom mobilapplikationer utnyttjar inbyggda sätt att telefonkameran skall autofokusera, medan detta inte var möjligt i webbläsaren. Med detta i baktanke prövade de ett annat alternativ, en öppen källa streckkodsläsare kallad ZBar vilket fungerade, men var också inkonsekvent.



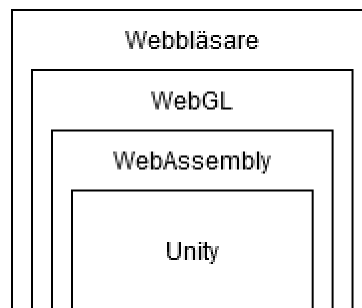
Figur 12 (Padmanabhan & Jha, 2019)

Eftersom de hade nu tre partiskt fungerade metoder som utför samma sak så skapade de en tävling mellan metoderna. De skapade en webbarbetare per metod (se figur 12), och då kameran tar bilder skickas denna data till varje webbarbetare. Den första metoden som returnerar en lyckad streckkodsläsning vinner. Även fast metoderna är opålitliga separat, så uppger de att utföra det på så här lyckas det nära till 100%. (Padmanabhan & Jha, 2019)

4.5 Unity

Unity är en kommersiell spelmotor som är plattformsoberoende, det inkluderar också en integrerad utvecklingsmiljö för att främja användarvänlighet. Unity riktar sig mot amatörer och

professionella spelutvecklare, detta speglar sig i deras prenumerationsbaserade affärsmodell – där användning är gratis för privatpersoner eller företag så länge spelets avkastning eller finansiering är under $200,000.



Figur 13

Unity är dock endast en spelmotor. För att kunna köra Unity i webbläsaren behövs WebGL – ett Javascript gränssnitt för att rendera grafik (plugin behövs inte) (Mozilla, 2020). Med Emscripten kompileras Unitys driftkod till WebAssembly, vilket sen utnyttjar HTML5:s *<canvas>* tillsammans med WebGL för att förverkliga grafiken. (se figur 13) (Unity, 2020)

Initialt använde Unity Asm.js, men då WebAssemblys MVP (minsta möjliga produkt) lanserades bytte de till det direkt. (Echterhoff, 2014) (Trivellato, 2018) Med denna förändring noterade de skillnaden på prestanda, filstorlek, samt hur det tillät flexibilitet i Unitys interna minnehantering. (Trivellato, 2018)

4.6 ViennaTS

ViennaTS är en ”öppen källkod C++-baserad processeringssimulator” för ”simuleringar av processer för tillverkning av halvledare”. (Klemenschits, et al., 2019)

ViennaTS är resursintensiv. På grund av det skulle någon typ av Javascript-baserad implementation vara orimlig. (Klemenschits, et al., 2019) mätte exekveringstiden och konstaterande att WebAssembly-implementationen jämfört med ursprungliga applikationen är långsammare med en faktor mellan 2.3 och 2.8. (Klemenschits, et al., 2019) förklarar dessutom att utföra ViennaTS:s kompilering till WebAssembly med Emscripten krävde att de endast ändrade på 570 linjer, utav totalt 28 986 linjer i kodbasen.

Hädanefter kan slutsatsen dras att fastän WebAssembly presterar relativt sämre, är det visserligen inte ett stort arbete att göra originella kodbasen förenlig med WebAssembly. Under

antagandet att kodbasen inte är rikligt beroende på sådant som inte är exponerat till WebAssemblys exekveringsmiljö. Frågan, som (Klemenschits, et al., 2019) dock inte tar ställning till, är huruvida WebAssemblys kompromiss mellan relativt sämre prestanda, gentemot bättre användbarhet samt tillgänglighet är relevant till användarens värde på ViennaTS i webbläsaren. De beaktar dock inte försämringen av prestanda, därav verkar den vara acceptabel för dem inom deras mål. (Klemenschits, et al., 2019)

4.7 Kryptovaluta-utvinning i webben

Sedan blockkedja-baserade kryptovalutor framkommit har det lagts fokus på olika metoder att utvinna dessa. Grafikkortsfarmer ger mest skörd, men webbsidor med mycket trafik sitter i en unik position att utnyttja sina besökares resurser för en ytterligare källa av inkomst. När detta insågs började olika tjänster dyka upp som förverkligade detta. Före WebAssembly var detta baserade på Javascript. På webbsidan läggs det till ett skript vilket utför utvinnandet. Sen kan den ansvarige begära att tjänsten betalar ut det ekvivalenta värdet som sidan har producerat. Tjänsterna behåller en del av intäkterna som deras kunders kunder har producerat.

WebAssembly är perfekt för denna användning. Eftersom WebAssembly är i binärt format påverkar den sidans laddningstider mindre, samt är den automatiskt obfuskerad och är därmed svårare att analysera och hejda. Dessutom är WebAssembly mångfaldigt snabbare än Javascript. Det är också möjligt att anropa och exekvera Javascript genom WebAssembly, tillsammans med *EM_JS({})*, vilket är försedd av *emscripten.h* (Emscripten, 2020). Det vill säga, fastän tjänsten inte vill skriva om sin Javascript-kod till ett LLVM-kompatibelt programmeringsspråk, kan de fortfarande utnyttja WebAssemblys naturliga obfuskering. (Musch, et al., 2019)

(Musch, et al., 2019) utförde en undersökning 2019 baserat på *Alexas Top 1 Million* [webbsidor] för att hitta WebAssemblys utbredning, samt hur det tillämpas. De fann att 1639 sidor använde WebAssembly på något sätt, varav 55,7% av webbsidorna använde det för att utvinna kryptovalutor (Musch, et al., 2019) . En liknande undersökning utfördes 2018 av (Rüth, et al., 2018). Denna fokuserar på effektiviteten av ett webbläsare-plugin kallat NoCoin, som är gjort för att hitta och blockera kryptovaluta-utvinning på webbsidor. Undersökningen baserar sig på samma data som (Musch, et al., 2019), men här klassificerar de dessutom var dessa kryptovaluta-utvinningsskript används.

NoCoin		Signatur	
Spel	19%	Pornografi	19%
Utbildning	9%	Teknologi	8%
Handel	8%	Fildelning	8%
Pornografi	7%	Utbildning	5%
Teknologi	6%	Underhållning	5%

Tabell 2, topp 5 kategorier av sidor vilket utnyttjar kryptovaluta-utvinningsskript. Klassificeringen utfördes med Symantec Rulespace. (Rüth, et al., 2018)

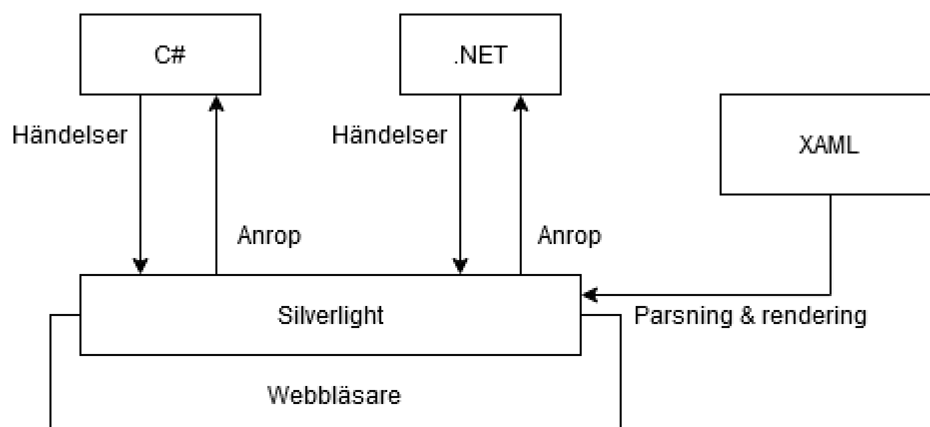
Tabell 2 visar deras resultat. NoCoins spalt innehåller det som det har detekterat som ett kryptovaluta-utvinningsskript, medan signatur-spalten innehåller det som (Rüth, et al., 2018):s egna kryptovaluta-utvinningsskripts identifieringsmetod detekterat.

Detta tas upp med syfte att etablera WebAssemblys utbredning och tillämpning inom detta område. Förståeligt tas det i bruk på webbsidor vars inkomst baseras på innehållet.

På grund av denna utveckling har säkerhets- och antireklam-program anpassat sig att blockera detta.

5 WebAssemblys företrädare

Före WebAssembly har det funnits olika liknande tekniker. Men på grund av vissa omfattande nackdelar har uppehåll av dessa upphört, och deras respektive ansvariga företag avvärjar användning.



Figur 14, (Ran & Li, 2010)

Microsoft Silverlight är ett plugin-baserat ramverk inom .NET:s ekosystem för att leverera rikt innehåll i webbläsare. Likt WebAssembly tillåter det exekvering av C# i webbläsaren, med skillnaden att medan WebAssembly är beroende på att bli instansierat genom ett Javascript objekt *WebAssembly* (Mozilla, 2020), så kräver Silverlight att klienten har installerat Silverlights mjukvara på operativsystemet, samt att Silverlights plugin är aktiverat på webbläsaren. Istället för att Silverlight körs i webbläsaren, så körs det på webbläsaren (se figur 14). På grund av detta har Silverlight större befogenhet över webbsidans innehåll, att manipulera DOM eller att hantera händelser är integrerat i Silverlight, gentemot WebAssembly vilket kräver att Javascript hanterar detta.

Eftersom Silverlight är beroende på att dess mjukvara stöds av operativsystemet, samt att webbläsaren stöder plugins så blev stödet lidande för olika plattformar. Speciellt på Apples läsplattor där de varken stöder Silverlight mjukvara, eller plugins för Safari, med avsikt att utfasa Adobe Flash på grund av dess inkompatibilitet med bärbara enheter. (Jobs, 2010) Konsekvensen av detta blev dock också en oavsiktlig utfasningen av Silverlight. (Lhotka, 2018) Microsoft Silverlight kommer att slutas stödas helt i oktober, 2021. (Microsoft, 2019)

Adobe Flash – ett ytterligare sätt att leverera rikt innehåll i webbsidor, liknande till Silverlight att det kräver ett plugin i webbläsaren för att fungera. Då Flash lanserades 1996 var avsikten att skapa applikationer – men adopterades av webbutvecklare eftersom det gav dem större flexibilitet på innehållet på webbsidor, utöver det som HTML och CSS tillät. Men, webbens natur förändrades då Apple samt andra företag pressade webben att vara användbar på telefonskärmar också. Flash anpassar sig inte till pekskärmar, har dålig prestanda, samt är en stängd standard. (Jobs, 2010) Flash är dessutom ökänd för omfattande säkerhetsbrister (Caseldon, et al., 2015). På grund av alla dessa skäl började en stegvis utfasning av Flash från företag. (Adobe, 2017)

Google Native Client (NaCl), senare Portable Native Client (PNaCl) – som dessutom beskrivs i kapitel 4.1 – är Googles lösning på att leverera högeffektiva program till webbläsaren. Lika som WebAssembly är NaCl beroende på LLVM, samt att i motsats till Silverlight och Adobe Flash kräver NaCl inte en separat installation. Dock till skillnad till WebAssembly, vilket också ledde till dess utfasning, är NaCl:s egocentriska utveckling. (Zbarsky, 2015), en ingenjör för Mozilla, beskriver att ”det finns ingen klar definition av Peppars gränssnitt, den definieras i princip av Googles implementering”, (Peppar är gränsnittet mellan NaCl och webbläsarens Javascript motor), samt ”Googles implementering är knuten till Blink, och är inte designad att

fungera med någon annan renderingsmotor.” (Blink är Chromes renderingsmotor) (Zbarsky, 2015) På grund av detta understöddes inte NaCl av andra webbläsare, och Google var mer än villig att arbeta på WebAssembly.

	NaCl	Flash	Silverlight	WebAssembly
Separat installation	Nej	Ja	Ja	Nej
Plattformsoberoende	Nej	Ja (på de plattformar som stöds)	Ja (på de plattformar som stöds)	Ja
Språkbegränsad	Nej	Ja	Ja	Nej
Kräver limkod (glue code)	Nej	Nej	Nej	Ja
Öppen standard	Ja	Nej	Nej	Ja
Utfasad	Ja	Ja	Ja	Nej

Tabell 3, summering av egenskaper hos WebAssemblys företrädare

Även fast WebAssemblys företrädare har principiellt uppfyllt samma uppgift som WebAssembly, har de varit bristfälliga i deras flexibilitet som krävs i webbläsare. Samt, till skillnad från NaCl, Flash och Silverlight blev WebAssembly utvecklat tillsammans, där alla deltagande företag är konkurrenter till varandra. Med kollektivt intresse i WebAssembly, fick företagen utforma sina korporativa intressen i WebAssembly, samt säkerställa att konkurrenterna stöder samma mål.

6 Framtida utveckling på WebAssembly

Den befintliga versionen av WebAssemby är vad dess utvecklare har kommit fram som minsta möjliga produkt (MVP) i konsensus. I denna MVP lämnade de ut önskvärd funktionalitet för att få ut en användbar version av WebAssembly, dock eftersom MVP:en är nu lanserat kan de påbörja utveckla på tidigare önskad funktionalitet, samt förbättra befintliga versionen.

Detta inkluderar SIMD (singulär instruktion, multipla data). SIMD är en ”instruktion av en special klass av instruktioner vilket utnyttjar data parallellism i applikationer genom att samtidigt utföra samma operation på multipla dataelement.” (V8, 2020).

Utvecklingsteamet för WebAssembly antyder också på olika sätt att möjliggöra manipulation av dokumentobjektsmodellen (DOM), det vill säga, istället för att WebAssembly kräver någon sorters limkod som befintliga MVP, skulle det vara möjligt med bindningar till värden för att manipulera det. Det tas inte ställning till ifall det skulle ha liknande egenskaper som till exempel komponent-baserade Shadow DOM, som olika Javascript-baserade ramverk använder sig av för att innesluta och manipulera delar av webbsidan.

Ifall ett sätt att manipulera DOM:en förverkligas, skulle också någon sorters skräpsamlare (garbage collector) också behövas. Speciellt ifall något programmeringsspråk som är beroende på skräpsamlare, till exempel Java, C# och Go, skulle anpassas att kompileras till WebAssembly. (Wagner, et al., 2018)

7 Slutsats

WebAssembly utvidgar webbläsarens förmågor och roll inom IT. Det som tidigare inte vore rimligt är nu tänkbart med WebAssembly. Även fast WebAssembly inte egentligen möjliggör något nytt, det vill säga allt möjligt med WebAssembly är också möjligt utan WebAssembly, men det erbjuder tillgänglighet till bägge användare samt utvecklarna då det utförs med WebAssembly.

Google Earth Web är resurskrävande, och det vore ovettigt att implementera det i webben med kunskapen att det inte skulle prestera till en viss standard, även fast användare skulle få snabbare tillgång att använda programmet i sådant fall. (Klemenschits, et al., 2019) visade att ViennaTS:s WebAssembly-implementation presterade noterbart sämre, vilket Google Earth Web också måste ha noterat i början. Dock eftersom bägge fortsatte med utvecklingen kan det anses att denna försämring av prestanda var acceptabel för dem.

Tillgänglighet samt hastighet är betydande för internets användare (Brutlag, 2009). Gentemot att ladda ned en applikation samt installera den, med WebAssembly främjas kundförvärvning eftersom kundens bemöda förändras inte i relation till programmets komplexitet eller resursanvändning. Dessutom osäkra kunder – som skulle ha motsatt sig att ladda ner en

applikation – är nu opartisk att använda programmet oberoende övriga faktorer, till exempel prestanda.

(Klemenschits, et al., 2019) antydde dessutom enkelheten att omvandla ViennaTS:s kodbas att kompilera till WebAssembly. Vilket (Cheung, 2019) också noterade då de partiellt implementerade AutoCAD i webben med WebAssembly, eftersom de skrotade sitt tidigare arbete att få AutoCAD till webben med Java. Förstås, svårigheten att omvandla en kodbas att kunna kompileras till WebAssembly beror på kodbasens beroenden, eftersom Javascripts sandlåda begränsar vissa förmågor på grund av säkerhetsrisker.

Eftersom WebAssembly utnyttjas av olika kryptovaluta-utvinningstjänster visar dessutom WebAssemblys flexibilitet, där prestanda samt kodens storlek spelar roll.

Sammanfattningsvis, att använda WebAssembly passar bäst ifall projektet förlitar sig på konsekvent prestanda – eller ifall det skulle kunna utnyttja paralliseringsmetoder – medan det håller saklig storlek, eller ifall koden bör obfuskeras. Ifall det finns en föregående kodbas och den är i ett LLVM-kompatibelt språk är det enkelt att kompilera det till WebAssembly. Befintligt existerande bibliotek som är skrivna i LLVM-kompatibla språk kan även användas utan besvär.

8 Referenser

Adobe, 2017. *Flash & The Future of Interactive Content.* [Online]
Available at: https://theblog.adobe.com/adobe-flash-update/
[Accessed 13 March 2020].

Anderson, R. & Nowak, R., 2020. *Introduction to Razor Pages in ASP.NET Core.* [Online]
Available at: https://docs.microsoft.com/en-us/aspnet/core/razor-pages
[Accessed 26 Februari 2020].

Brutlag, J., 2009. *Speed Matters for Google Web Search,* Mountain View: Google, Inc..

Caseldon, D., Souffrant, C. & Jiang, G., 2015. *Flash in 2015.* [Online]
Available at: https://www.fireeye.com/blog/threat-research/2015/03/flash_in_2015.html
[Accessed 13 March 2020].

Chen, W., 2018. *Performance Testing Web Assembly vs JavaScript.* [Online]
Available at: https://medium.com/samsung-internet-dev/performance-testing-web-assembly-vs-javascript-e07506fd5875
[Accessed 24 March 2020].

Cheung, K., 2018. *AutoCAD & WebAssembly.* New York, QCon.

Cheung, K., 2019. *AutoCAD & WebAssembly Moving a 30 Year Code Base to the Web.* [Online]
Available at: https://www.infoq.cn/article/V7mu3J4sJe-srZoNiiJf
[Accessed 18 Februari 2020].

Clark, L., 2017. *A crash course in just-in-time (JIT) compilers.* [Online]
Available at: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/
[Accessed 24 March 2020].

Clark, L., 2017. *What makes WebAssembly fast?.* [Online]
Available at: https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/
[Accessed 24 March 2020].

Deveria, A., 2020. *Can I Use.* [Online]
Available at: https://caniuse.com/#feat=wasm
[Accessed 9 Februari 2020].

Eberhardt, C., 2017. *Exploring different approaches to building WebAssembly modules.* [Online]
Available at: https://blog.scottlogic.com/2017/10/17/wasm-mandelbrot.html
[Accessed 24 March 2020].

Eberhardt, C., 2020. *WebAssembly and the Death of JavaScript.* London: JS Monthly London Meetup.

Echterhoff, J., 2014. *On the future of Web publishing in Unity.* [Online]
Available at: https://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/
[Accessed 8 Mars 2020].

Emscripten, 2020. *Emscripten: Calling JavaScript from C/C++.* [Online]
Available at: https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-call-javascript-from-native
[Accessed 22 March 2020].

Fitzgerald, N., 2018. *Oxidizing Source Maps with Rust and WebAssembly.* [Online]
Available at: https://hacks.mozilla.org/2018/01/oxidizing-source-maps-with-rust-and-webassembly/
[Accessed 24 March 2020].

Golsch, L., 2019. *WebAssembly: Basics,* Brunswick: s.n.

Gurgone, G. & Spless, P., 2018. *A Real-World WebAssembly Benchmark.* [Online]
Available at: https://pspdfkit.com/blog/2018/a-real-world-webassembly-benchmark/
[Accessed 24 March 2020].

Haas, A. o.a., 2018. Bringing the web up to speed with WebAssembly. *Communications of the ACM,* Volym 61, pp. 107-115.

Herrera, D., Chen, H. & Lavoie, E., 2018. *WebAssembly and JavaScript Challenge : Numerical program performance using modern browser technologies and devices,* Montreal: Sable McGill.

Jangda, A., Powers, B., Berger, E. & Guha, A., 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In: *2019 USENIX Annual Technical Conference.* Renton: USENIX Association, pp. 107-120.

Jobs, S., 2010. *Thoughts on Flash.* [Online]
Available at: https://www.apple.com/hotnews/thoughts-on-flash/
[Accessed 11 March 2020].

Klemenschits, X., Manstetten, P., Filipovic, L. & Selberherr, S., 2019. Process Simulation in the Browser: Porting ViennaTS using WebAssembly. In: *2019 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD).* Vienna: s.n., pp. 1-4.

Lhotka, R., 2018. *Why did Microsoft Silverlight fail? What were the problems with it?.* [Online]
Available at: https://www.quora.com/Why-did-Microsoft-Silverlight-fail-What-were-the-problems-with-it
[Accessed 11 March 2020].

Mears, J., 2019. *Google Earth WebAssembly with Jordon Mears* [Interview] (2 July 2019).

Mears, J., 2019. *Performance of WebAssembly: a thread on threading.* [Online]
Available at: https://medium.com/google-earth/performance-of-web-assembly-a-thread-on-threading-54f62fd50cf7
[Accessed 11 Februari 2020].

Mears, J., 2019. *web.dev.* [Online]
Available at: https://web.dev/earth-webassembly/
[Accessed 11 Feburari 2020].

Microsoft, 2019. *Silverlight End of Support.* [Online]
Available at: https://support.microsoft.com/en-us/help/4511036/silverlight-end-of-support
[Accessed 11 March 2020].

Microsoft, 2020. *Microsoft .NET.* [Online]
Available at: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor
[Accessed 26 Februari 2020].

Mozilla, 2020. *SharedArrayBuffer.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
[Accessed 11 Februari 2020].

Mozilla, 2020. *WebAssembly Javascript Object.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly
[Accessed 10 March 2020].

Mozilla, 2020. *WebGL: 2D and 3D graphics for the web.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
[Accessed 8 Mars 2020].

Musch, M., Wressnegger, C., Johns, M. & Rieck, K., 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In: R. Perdisci, C. Maurice, G. Giacinto & M. Almgren, eds. *Detection of Intrusions and Malware, and Vulnerability Assessment.* Gothenburg: Springer, pp. 23-42.

Nelson, B., 2017. *Chromium Blog.* [Online]
Available at: https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html
[Accessed 11 Februari 2020].

Nelson, J. C., Roth, D. & Latham, L., 2020. *Call .NET methods from JavaScript functions in ASP.NET Core Blazor.* [Online]
Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/call-dotnet-from-javascript
[Accessed 26 Februari 2020].

Padmanabhan, S. & Jha, P., 2019. *WebAssembly at eBay: A Real-World Use Case.* [Online]
Available at: WebAssembly at eBay: A Real-World Use Case
[Accessed 23 Februari 2020].

Ran, C.-s. & Li, N., 2010. Design and implementation of communication model based on Silverlight and WCF in EAM system. In: *2010 2nd IEEE International Conference on Information and Financial Engineering.* s.l.:s.n., pp. 590-593.

Rossberg, A., 2018. *Why is webAssembly function almost 300 time slower than same JS function.* [Online]
Available at: https://stackoverflow.com/a/48175569
[Accessed 24 March 2020].

Roth, D., 2020. *ASP.NET Core Blazor hosting models.* [Online]
Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models
[Accessed 26 Februari 2020].

Rüth, J., Zimmermann, T., Wolsing, K. & Hohlfeld, O., 2018. Digging into Browser-Based Crypto Mining. In: *Proceedings of the Internet Measurement Conference 2018.* New York: Association for Computing Machinery, pp. 70-76.

Trivellato, M., 2018. *WebAssembly is here!.* [Online]
Available at: https://blogs.unity3d.com/2018/08/15/webassembly-is-here/
[Accessed 8 Mars 2020].

Trivellato, M., 2018. *WebAssembly Load Times and Performance.* [Online]
Available at: https://blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance/
[Accessed 8 Mars 2020].

Unity, 2020. *Getting started with WebGL development.* [Online]
Available at: https://docs.unity3d.com/Manual/webgl-gettingstarted.html
[Accessed 8 Mars 2020].

V8, 2020. *Fast, parallel applications with WebAssembly SIMD.* [Online]
Available at: https://v8.dev/features/simd
[Accessed 21 March 2020].

W3C, 2019. *WebAssembly Core Specification.* [Online]
Available at: https://www.w3.org/TR/wasm-core-1/
[Accessed 9 Februari 2020].

W3C & Ehrenberg, D., 2019. *WebAssembly Javascript Interface.* [Online]
Available at: https://www.w3.org/TR/wasm-js-api-1/#webassembly-namespace
[Accessed 17 March 2020].

Wagner, L., 2015. *Mozilla.* [Online]
Available at: https://blog.mozilla.org/luke/2015/06/17/webassembly/
[Använd 9 Februari 2020].

Wagner, L., Clark, L. & Schneidereit, T., 2018. *WebAssembly's post-MVP future: A cartoon skill tree.* [Online]
Available at: https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/
[Accessed 24 March 2020].

Zakai, A., 2011. Emscripten: an LLVM-to-Javascript compiler. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion.* New York: Association for Computing Machinery, pp. 301-312.

Zakai, A., 2017. *Why WebAssembly is Faster Than asm.js.* [Online]
Available at: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/
[Accessed 17 Feburari 2020].

Zbarsky, B., 2015. *Bugzilla.* [Online]
Available at: https://bugzilla.mozilla.org/show_bug.cgi?id=729481#c83
[Accessed 11 Februari 2020].