elektrorevue
ISSN 1213 - 1539

Rok / Year:	Svazek / Volume:	Číslo / Number:	Jazyk / Language
2020	22	1	SK

WebAssembly – vývoj a budúcnosť

WebAssembly – development and future

Patrik Švikruha

patrik.svikruha@vut.cz

Fakulta elektrotechniky a komunikačních technologií VUT v Brně

DOI: -

Abstract: In nowadays the web platform is specific with a low diversity of technology for the development of client applications. A synonym for the target platform of client applications has become JavaScript, which is standardized under the name ECMAScript and is a W3C (World Wide Web Consortium) standard. However, JavaScript was not intended to be the target platform, and despite advanced optimizations of runtime environments, it does not achieve native performance. These issues will help create the WebAssembly specification, which represents a joint effort by the web community to create a new, secure, platform-neutral code representation that will achieve native performance. This review article describes the WebAssembly standard and the Portable Native Client and asm.js technologies on which WebAssembly is based. WebAssembly is often mistaken for a JavaScript replacement. Comparing WebAssembly and JavaScript will allow the reader to distinguish between different uses for the two technologies. Finally, the article evaluates the state of the W3C standard and the functionality, which will be added to the standard. This developed functionality aims to extend WebAssembly to a non-web platform.

elektrorevue
ISSN 1213 - 1539

WebAssembly – vývoj a budúcnosť

Patrik Švikruha

Fakulta elektrotechniky a komunikačních technologií VUT v Brně
Email: patrik.svikruha@vut.cz

Abstrakt – *V dnešnej dobe je webová platforma špecifická malou diverzitou technológii pre vývoj klientskych aplikácií. Synonymom pre cieľovú platformu klientských aplikácií sa stal JavaScript, ktorý je štandardizovaný pod názvom ECMAScript a je štandardom W3C (World Wide Web Consortium). JavaScript však nebol určený ako cieľová platforma, a aj napriek pokročilým optimalizáciam behových prostredí nedosahuje natívny výkon. Tieto problémy pomohli vzniku špecifikácie WebAssembly, ktorá predstavuje spoločné úsilie webovej komunity na vznik novej, bezpečnej a platformovo neutrálnej reprezentácie kódu, ktorá bude dosahovať natívny výkon. Tento prehľadový článok popisuje štandard WebAssembly, technológiu Portable Native Client a asm.js z ktorých WebAssembly vychádza. WebAssembly sa často mylne označuje ako náhrada JavaScriptu. Porovnanie WebAssembly a JavaScriptu umožní čitateľovi rozlíšiť rôzne použitia týchto dvoch technológií. Na záver článok hodnotí stav W3C štandardu a rozpracovanú funkcionalitu, ktorá bude do štandardu pridaná. Táto rozpracovaná funkcionalita má za cieľ rozšíriť WebAssembly aj mimo webovú platformu.*

1 Úvod

Malú diverzitu technológií a slabú podporu na poli vývoja klientskych webových aplikácií sa snažila vyriešiť webová komunita rôznymi spôsobmi. Okrem štandardizovaného JavaScriptu sa pre klientské aplikácie používali rôzne proprietárne riešenia, ktoré riešili výkonnostné problémy JavaScriptu. Medzi najznámenjšie technológie patria Java Applet, Silverlight a FlashPlayer. Tieto technológie boli zavrhnuté na jednej strane kvôli bezpečnostným problémom a na strane druhej kvôli tomu, že neboli štandardizované konzorciom W3C (World Wide Web Consortium). V roku 2015 W3C predstavila novú pracovnú skupinu s názvom WebAssembly Working Group, ktorá začala pracovať na vytvorení univerzálnej, jazykovo a platformovo neutrálnej reprezentácie kódu. Táto reprezentácia dostala názov WebAssembly a jej 1. verzia vyšla ako W3C štandard 5. decembra 2019. V tomto článku budú predstavené základné koncepty štandardu WebAseembly a dôvody jeho vzniku. Nasledujúce kapitoly budú popisovať technológie Portable Native Client a asm.js, z ktorých WebAssembly vychádza. Dôležitú úlohu pri implementácii tohoto štandardu zohrala aj sada kompilátorov LLVM, ktorá je popísaná v kapitole 3.1. Popis rozdielov medzi WebAseembly a JavaScriptom má za cieľ ukázať výkonnostné a koncepčné rozdiely medzi týmito dvoma technológiami. Popis aktuálneho stavu W3C štandardu priblíži čitateľovi možnosti využitia WebAssembly. Posledná kapitola sa zaoberá budúcnosťou a rozpracovanou funkcionalitou, ktorá má za cieľ rozšíriť tento WebAssembly ako univerzálny, platformovo nezávislý formát.

2 Predstavenie WebAssembly

WebAssembly je binárny inštrukčný formát pre zásobníkový virtuálny počítač. WebAssembly je dizajnovaný ako portable compilation target pre vysokoúrovňové jazyky ako C, C++, Rust a umožňuje vývoj pre webové a serverové aplikácie. Portable compilation target znamená, že skompilovaný kód nie je určený pre fyzickú platformu, ale pre virtuálnu, ktorá vytvára abstrakciu nad fyzickou platformou.

- Oficiálna stránka [31].
- Oficiálny repozitár [32].
- W3C pracovná skupina [25].

Táto technológia reprezentuje dôležitý vývojový stupeň evolúcie vývoja webových aplikácii. Umožňuje spúšťať skompilovaný a optimalizovaný kód v prostredí webového prehliadača bez nutnosti inštalácie externého pluginu.

WebAssembly teda umožňuje zmeniť prístup k vývoju webových aplikácií. Do uvedenia WebAssembly bol jediný programovací jazyk schválený konzorciom W3C ako štandard ECMAScript [1].

Nevýhody JavaScriptu ako assembly jazyka je jeho textová forma a nedeterministická optimalizácia kódu na strane JavaScriptových enginov. Táto nedeterministickosť je spôsobená rôznymi implementáciami JavaScriptových enginov a spôsobom distribúcie JavaScritpového kódu. Pre spustenie JavaScriptového kódu v prehliadači je nutné stiahnuť celý textový súbor s kódom, sparsovať kód a vytvoriť z kódu syntaktický strom (AST - Abstract syntax tree). Na základe syntaktického stromu sa vytvorí bytecode, ktorý následne JIT (Just in Time) infraštruktúra dokáže optimalizovať a reoptimalizovať.

Spôsob akým sa dnes spúšťa JavaScript má nevýhody, ktoré pochádzajú z jeho vlastnosti – textová reprezentácia

[1] V praxi sa stretávame s názvom JavaScript.

elektrorevue
ISSN 1213 - 1539

a hlavne dynamická typovosť. JavaScript bol pôvodne navrhnutý ako doplnkový jazyk k Jave [21] a jeho úlohou nebolo poskytnutie vysokého výkonu. WebAssembly je oproti JavaScriptu navrhnutá od začiatku s dôrazom na výkon a kompaktnosť kódu. S JavaScriptom zdieľa rovnaký princíp zabezpečenia sandboxom a pomocou W3C doporučení aj kompatibilitu s aktuálnymi prehliadačmi.

WebAssembly sa dá v jednoduchosti zhrnúť ako nízkoúrovňový jazyk podobný assembleru, založený na stack machine, ktorý je určený pre spustenie vo WebAssembly engine[2]. Špecifikácia jazyka určuje silnú typovosť a binárnu reprezentáciu [36]. Tieto vlastnosti, spolu s jednoduchosťou, umožňujú WebAssembly dosahovať výkon, ktorý je zrovnateľný s natívnym kódom. Aktuálna verzia špecifikácie určuje iba číselné typy (i32, i64, f32 a f64) [33].

Okrem binárnej reprezentácie kódu, špecifikácia určuje aj textovú reprezentáciu. Binárna reprezentácia sa označuje ako WASM a textová ako WAT. Textová a binárna forma sú navzájom zameniteľné a sú určené pre rôzne použitia. Textová forma je určená pre debugging a rôzne nástroje na analýzu kódu. Binárna forma je určená pre produkčné použitie a beh vo WebAssembly engine[3].

WebAssembly sa označuje aj ako compilation target. Toto označenie znamená že WebAssembly kód je platformovo a hardvérovo nezávislý. Pri vzniku bola motivácia vyvinúť doplnok k JavaScriptu, ktorý je s ním plne kompatibilný, bude bežať v JavaScriptovom engine ale poskytne predikovateľný výkon, ktorý bude porovnateľný s natívnym kódom. WebAssembly dokáže s JavaScriptom spolupracovať na základe *WebAssembly JavaScript API*.

WebAssembly je od počiatku vyvíjaná ako webový štandard pod vedením W3C, s podporou majoritných prehliadačov (Mozilla, Chrome, Safari, Edge) [17].

Táto technológia bola po prvýkrát predstavená v roku 2015 na blogu Luke Wagnera [30]. Hlavnou úlohou WASM je podporiť 3D rendering, editáciu videa, hry, kompresiu, a iné výpočtovo náročne použitia vo webovom prehliadači.

3 História a dôvody vzniku

Snaha o klientskú interakciu v prehliadači má dlhú históriu. Okrem JavaScriptu existujú technológie Java Applet, Silverlight, alebo Adobe Flash. Tieto technológie neboli úspešné hlavne kvôli tomu, že fungovali ako pluginy do prehliadačov. Tiež neboli štandardizované a ich vývoj určovala jedna firma (Sun Microsystems, Microsoft, Adobe).

S rozvojom internetu majú webové aplikácie stále väčšiu JavaScriptovú codebase, čo znamená nárast veľkosti a počtu súborov. Nepriamo sa túto nevýhodu snažilo konzorcium W3C vyriešiť vylepšovaním štandardu HTTP (Hypertext Transfer Protocol). Prvé vylepšenie bolo možnosť reťazení HTTP odpovedí v HTTP/1.1 pomocou jednoho HTTP spojenia. Druhé vylepšenie vo verzii HTTP/2 je multiplex viacerých požiadaviek skrz jedno HTTP spojenie. Ani tieto vylepšenia však neriešia najväčší problém JavaScriptového kódu, a to jeho veľkosť a textovú reprezentáciu.

Zväčšujúcou sa užívateľskou základňou chytrých telefónov sa zvyšuje počet užívateľov, ktorý prehliadajú stránky pomocou nízkorýchlostných mobilných dát. Tento fakt spolu so zvyšujúcimi sa nárokmi na interaktivitu stránok má za následok zväčšovanie codebase. Nárast veľkosti má za následok to, že medián interaktivity stránok pre mobilné zariadenia sa pohybuje okolo 7,5 s[4]. Tieto fakty nútia webovú komunitu hľadať iné spôsoby ako optimalizovať technológie, ktoré sa používajú v prehliadačoch.

Pluginy Java Applet a Silverlight mali výhodu vlastného runtime (JVM alebo CLR) v hosťovskom OS prehliadača a tak dokázali vykonávať kód takmer natívnou rýchlosťou.

Prekurzormi vzniku WebAssembly by sa dali nazvať pokusy skompilovať rôzne jazyky do JavasScriptu, čo malo svoje výhody a nevýhody medzi ktoré patrí:

- Statická a silná typovosť (niektorých) kompilovaných jazykov a ich sadu nástrojov pri vývoji.
- Compile-time kontrola chýb.
- Statická analýza kódu (pri dynamických jazykoch je to náročné).
- JavaScriptové enginy dokážu pomerne efektívne spúšťať aj veľké skompilované JavaScriptové codebase.
- Pri debuggovaní aplikácie a hľadaní chyby bolo nutné, aby vývojár poznal JavaScript a jeho sadu nástrojov.

V roku 2006 Google uvoľnil GWT (Google Web Toolkit – `http://www.gwtproject.org/`), čo je v skratke súbor nástrojov pre kompiláciu aplikácie napísanej v Jave do JavaScriptu. Pomocou nástroja GWT sú napísané Google Maps, Gmail alebo populárna platforma Blogger.

Napriek tomu, že existuje množstvo jazykov, ktoré je možné skompilovať do JavaScriptu, tento prístup má nasledovné **nevýhody**:

- Ignorácia sémantických rozdielov skompilovaného jazyka a JavaScriptu.
- Generovaný JavaScript nie je možné optimalizovať tak, ako kód, z ktorého je vygenerovaný (JavaScript je totiž optimalizovaný až v JavaScriptovom engine – JIT optimalizácia vs AOT optimalizácia).
- Debugovanie zdrojového kódu skompilovaného do JavaScriptu je obtiažne.

Spôsob distribúcie funkcionality pomocou transkompilácie do JavaScriptu ale nerieši problém výkonu JavaScriptu pre špecializované použitia, ako sú výkonnostne náročné

[2] Väčšina existujúcich JavaScripových enginov podporuje WebAssembly.

[3] Existujú samostatné WebAssembly runtime ako napríklad `https://wasmer.io/`.

[4] Údaje sú aktuálne k 1.1.2020 (zdroj: `https://httparchive.org/reports/loading-speed`).

elektrorevue
ISSN 1213 - 1539

výpočty. Medzi tieto aplikácie patrí spracovanie videa, grafiky, kryptografia, virtuálna realita, atď.

Tradične bol JavaScript iba interpretovaný, s rastúcim dopytom po interaktívnych webových stránkach však všetky JavaScriptové enginy využívajú JIT interpreter. JavaScriptové engine ako V8 (Google), Spidermonkey (Mozilla) alebo Chakra (Microsoft)[5] dokážu JavaScripotvý kód skompilovať do určitej formy bytecode a tú optimalizovať. Takto optimalizovaný kód je možné rýchlejšie „interpretovať“, čím sa zvyšuje efektívna rýchlosť JavaScriptu [37].

Jedna z menej známych optimalizácií JavaScriptu, je napríklad pridanie typu premennej (ukážka kódu 1). Pridanie operátora „`|`" informuje JavaScriptový engine, že nemusí zisťovať typ založený na výsledku výrazu, ale zistí ho vo fáze vytvárania AST, čím sa zefektívni jeho činnosť [2].

```
function compiledCalculation() {
    var x = f()|0;  // x je 32-bit int hodnota
    var y = g()|0;  // y je 32-bit int hodnota
    return (x+y)|0; // 32-bit sčítanie, nevykonáva sa
    ↪  žiadna kontrola typu alebo pretečenia
}
```

Výpis kódu 1: *Ukážka JavaScriptového kódu s explicitne definovanými typmi premenných*[6].

Google a Mozilla, tvorcovia prehliadačov Chrome a Firefox, sa rozhodli vyvinúť technológie, ktoré by riešili tieto problémy. Spoločnosti sa rozhodli pre odlišné prístupy k riešeniu vyššie uvedeného problému a vznikli dve technológie **Native Client** a **asm.js**.

3.1 Projekt LLVM

Názov LLVM pravdepodobne vznikol ako akronym z **L**ow **L**evel **V**irtual **M**achine, no podľa oficiálnej dokumentácie je LLVM celý názov projektu [14]. LLVM je modulárna sada kompilátorov, ktoré dokážu deterministicky a efektívne optimalizovať kód. Kompilácia zdrojového kódu pomocou LLVM sa skladá z dvoch po sebe idúcich kompilácií. O prvú kompiláciu sa stará kompilátor programovacieho jazyka (frontend kompilátor). Tento kompilátor skompiluje zdrojový kód do LLVM Intermedia Representation (LLVM IR) reprezentácie. Táto reprezentácia je následne optimalizovaná pomocou modulárnych optimalizátorov. Nato je zoptimalizaovaná LLVM IR reprezentácia skompilovaná pomocou druhého, systémového kompilátora (backend kompilára) pre cieľovú platformu (x86, AMD64 a WebAssembly). Dvojúrovňová kompilácia, spolu s modulárnosťou umožňujú aby sa LLVM používal pre veľké množstvo jazykov a platforiem.

Na schéme 1 je princíp fungovania LLVM toolchainu. Na vstupe je kód, ktorý je napísaný v rôznych vysokoúrovňových jazykoch (C++, C). Tento kód je pomocou frontend kompilátoru skompilovaný do Intermedia Representation (IR), ktorý je následne optimalizovaný. Táto optimalizovaná forma je následne skompilovaná pomocou backend kompilátora pre cieľovú platformu.

Obrázek 1: *Princíp fungovania LLVM toolchainu.*

Toto rozdelenie a modulárnosť LLVM má následovné výhody:

- Pre pridanie podpory nového vysokoúrovňového jazyka je nutné vytvoriť iba frontend kompilátor.
- Pre pridanie podpory nového compilation target je nutné vytvoriť iba nový backend kompilátor.
- Pre podporu rôznych optimalizácií je možné pracovať iba s IR kódom, čím sa tieto optimalizácie unifikujú skrz rôzne jazyky.

Jeden z compilation targets je aj WebAssembly. Toto umožňuje rôznym jazykom, pre ktoré existuje frontend kompilátor, aby mohli LLVM toolchain použiť pre kompiláciu do WebAssembly a bežať v prehliadači [13].

3.2 Technológie Native Client a Portable Native Client

Netscape vytvoril v 90. rokoch NPAPI (Native Pepper API), ktoré sprístupnilo pluginom prehliadača systémove volania. Pomocou NPAPI mohol teda kód pluginu volať systémové volania v kontexte užívateľa, pod ktorým bol spustený prehliadač. Toto spôsobovalo bezpečnostné riziko a bolo jedným z dôvodov zavrhnutia tejto technológie. Na NPAPI boli založené aj vyššie spomínané technológie Silverlight, Flash Player a Java Applet.

Mozilla v spolupráci s Google vytvorila PPAPI (Portable Pepper API) [10], v rámci projektu Mortar [12], ktoré pôvodne malo priniesť PDF čítač PDFium a Flash pluginy do prehliadača Firefox. Rozdiel oproti NPAPI bol ten, že cieľ kompilácie nebola architektúra hosťovského zariadenia prehliadača, ale LLVM IR reprezentácia. Mozilla, ktorá spolupracovala na projekte s Google však projekt zavrhla.

Google v tomto projekte pokračoval a ďalej ho rozvíjal a vytvoril technológiu Native Client (NaCl). NaCl bola možnosť ako spustiť natívny kód v prehliadači. Relatívne zabezpečenie tejto technológie sa riešilo výberom systémových volaní, ktoré boli označené ako bezpečné. Preto, aby sa kód nemusel pre každú architektúru kompilovať zvlášť, použil Google PPAPI a tím vznikol Portable Native Client (PNaCl) [20]. Tieto technológie sa skladajú z 2 elementov.

1. Toolchain[7] ktorá kompiluje C/C++kód do tzv. NaCl/PNaCl modulov.

[5] Microsoft Edge bude používať JavaScripový engine V8 a renderovací engine Blink [1], kvôli tomu, že prechádza na Chromium.

[7] Použitý LLVM s CLang kompilátorom.

elektrorevue
ISSN 1213 - 1539

2. Runtime komponenty, ktoré sú integrované v prehliadači a umožňujú spustenie NaCl/PNaCl modulov (pomocou PPAPI a NPAPI).

Po vytvorení modulov **nexe** (NaCl) a **pexe** (PNaCl) bolo možné tieto moduly spustiť v prehliadači. Interakcia modulu s prehliadačom bola možná vďaka vyššie spomínanému Pepper API (NPAPI a PPAPI). Štandard tohoto API bol však nekompletný a chýbala dokumentácia, takže jediný prehliadač, ktorý plne podporoval túto technológiu, bol Google Chrome (a Chromium porty) [8].

Výhoda tohoto prístupu spočíva v tom, že z C/C++sa pomocou LLVM toolchainu dokázal vytvoriť veľmi výkonný natívny kód (pretože používal inštrukčnú sadu danej architektúry – AMD64, x32, ARM).

Vo firme Google sa začala debata či pokračovať v NaCl, pretože sa to veľmi podobalo ére pluginov[8] a po debatách medzi V8 tímom a NaCl tímom sa Google rozhodol v máji 2017 oficiálne ukončiť tento projekt a vývojárov presunúť do novovzniknutého WebAssembly tímu [4].

3.3 Technológia asm.js

Mozilla sa rozhodla vydať po ukončení spolupráce na PPAPI cestou optimalizácie JavaScriptu a vznikol asm.js, čo je **subset** JavaScriptu, v ktorom sa používajú iba **číselné dátové typy**.

Obmedzenie na číselné dátové typy má za následok, že JavaScriptový engine nemusí pri JIT kompilácii JavaScriptového kódu zisťovať typy premenných, pretože tieto typy sú známe už z vytvorenia AST. Táto optimalizácia sa vykonáva pri kompilácii JavaScript/C/C++kódu do asm.js, t.j. Ahead of Time (AOT).

Asm.js je teda silne **typový** subset JavaScriptu [22], ktorý je optimalizovaný tak, aby JavaScript engine po vytvorení syntaktického stromu (AST) nemusel zisťovať dátový typ premenených [38].

Formát asm.js sa môže použiť ako compilation target, pretože umožňuje optimalizácie, aké samotný JavaScriptový kód neumožňuje [29]. Myšlienka pridať silnú typovosť do JavaScriptového kódu je v ostrom kontraste s vlastnosťou JavaScriptu, ktorou je jeho dynamická typovosť. Podľa benchmarkov bola rýchlosť asm.js voči natívnej rýchlosti[9] menšia iba o 50 % [23].

```
function f(i) {
    return i + 1;
}
```

Výpis kódu 2: *Kód pred optimalizáciou pomocou Emscripten.*

Výpis kódu 3 je vo formáte asm.js. Direktíva na prvom riadku „use asm" hovorí JavaScriptovému engine, že nasledujúci kód je v asm.js formáte. Ak engine podporuje

```
"use asm";
function f(i) {
    i = i|0;
    return (i + 1)|0;
}
```

Výpis kódu 3: *Kód po skompilovaní do asm.js formátu pomcou nástroja Emscripten.*

asm.js, tak pri vytváraní AST priradí premenným typy, ktoré sú definované pomocou znaku pre bitovú operáciu OR – „|" a typu „0", ktorý za ním nasleduje (0 znamená 32 bitový integer). Ak engine nepodporoval asm.js, tak sa tento kód vykonával ako štandardný JavaScript.

Subset asm.js je teda optimalizovaný subset JavaScriptu. Polia a premenné sú implicitne typové a používajú sa v ňom iba numerické dátové typy. Výkon asm.js sa mierne približuje výkonu natívnemu kódu. Asm.js má síce výhody skompilovaného jazyka, ale stále sa jedná o JavaScript, čiže nie je možné plne využiť všetky optimalizácie, ktoré ponúka LLVM kompilácia.

Vlastnosti asm.js možno zhrnúť do nasledujúcich bodov:

- asm.js kód predchádza „spomaľovaniu" kódu pomocou implicitného typu premenných. Toto umožňuje AOT (Ahead of Time) kompiláciu a optimalizáciu kódu do JavaScriptu. Výsledný kód má za následok zrýchlenie JIT kompilácie JavaScripového engine.

- JavaScriptový engine má garanciu, že typ danej premennej sa nebude v runtime meniť, takže môže vygenerovať efektívny medzikód.

3.4 Zhrnutie Native Client a asm.js

Native Client priniesol výkon natívneho kódu do prehliadača a asm.js zase prenositeľnosť kódu, ktorý je možné spustiť bez pluginu v každom prehliadači. Fúziou týchto myšlienok vznikla WebAssembly, ktorá je platformovo agnostická ako asm.js, ale zároveň umožňuje kódu výkon podobný Native Client. Native Client a aj asm.js sú popri Flash Player, Java Applet a Silverlight míľniky, ktoré boli prekurzormi technológie WebAssembly. Obrovskú zásluhu na rozvoji WebAssembly má Mozilla, ktorej prehliadač je lídrom v implementácii štandardu W3C WebAssembly.

Nečakaný výkon asm.js mal za následok, že sa vytvorila pracovná skupina z Mozilla, Microsoft, Google a Apple, a vytvorili pre tento subset JavaScriptu špecifikáciu.

4 Princíp fungovania WebAssembly

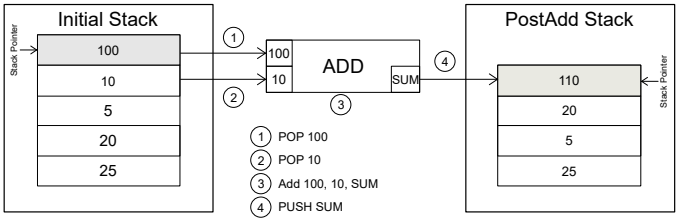
WebAssembly je vytvorená ako virtuálny zásobníkový počítač. Reprezentácia kódu zasobníkového počítača umožňuje kompaktnejší kód pre enkódovanie, dekódovanie, kompiláciu a interpretáciu [35].

O WebAssembly môžeme povedať, že jej inštrukčný set architektúry (ISA) je RISC (Reduced Instruction Set Computer).

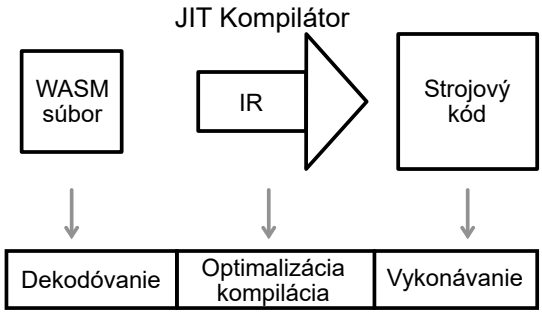
[8] Bolo treba do prehliadača pridať NaCl/PaCl plugin, a ten nebol štandardizovaný.

[9] Rýchlosť, ktorá by bola ekvivalentná rýchlosti ak by bol kód napísaný v strojovom kóde danej platformy.

elektrorevue
ISSN 1213 - 1539



Obrázek 2: *Zjednodušená schéma zásobníkového počítača - Stack machine.*



Obrázek 3: *Principiálna schéma spúšťania WebAssembly v JavaScriptovom engine spolu s časovým pomerom jednotlivých krokov.*

Na schéme 3 sa nachádza principiálna schéma spúšťania WebAssembly vo WebAssembly engine[10]. Engine pri načítaní súboru tento súbor iba dekóduje a vytvorí z neho IR kód. **Tento kód nie je LLVM IR kód**. Tento IR slúži JIT kompilátoru na optimalizácie a následnú kompiláciu na inštrukčnú sadu danej platformy.

WebAssembly je síce binárny formát, ale špecifikácia hovorí aj o textovej WAT (WebAssembly Text) reprezentácii, ktorá slúži pre sadu nástrojov a ľudí [15]. V praxi sa nepredpokladá, že by sa písala WebAssembly ručne pomocou WAT formátu. WAT zápis používa S-expression [15], ako je vidieť na zápise 4.

Tento kód je reprezentáciou C kódu z výpisu kódu 5.

4.1 Použitie v prehliadači

WebAssembly momentálne nie je možné v prehliadači spustiť priamo. Jediná možnosť je načítanie WebAssembly modulu cez JavaScript a následné používanie tohoto modulu pomocou JavaScript interoop. Výpis kódu 6 ukazuje načítanie WebAssembly modulu zo súboru **main.wasm**, ktorého WAT reprezentácia je na výpise 4.

Ak by prehliadač nepodporoval WebAssembly, je možné použiť fallback na asm.js. Tento fallback je ale veľmi nepraktický, pretože výsledný súbor vo formáte asm.js môže mať desiatky MB, čo je pre prostredie webu veľmi nepraktické. Keďže je WebAssembly štandard W3C a podporuje

[10] Môže sa jednať o JavaScripový engine s podporou WebAssembly.

```
(module
  (type $t0 (func))
  (type $t1 (func (result i32)))
  (func $__wasm_call_ctors (type $t0))
  (func $main (export "main") (type $t1) (result i32)
    i32.const 42)
  (table $T0 1 1 anyfunc)
  (memory $memory (export "memory") 2)
  (global $g0 (mut i32) (i32.const 66560))
  (global $__heap_base (export "__heap_base") i32
  ↪  (i32.const 66560))
  (global $__data_end (export "__data_end") i32
  ↪  (i32.const 1024))
)
```

Výpis kódu 4: *WAT zápis binárneho WASM súboru vygenerovaného z kódu 5.*

```
#define WASM_EXPORT __attribute__((visibility("default")))
WASM_EXPORT
int main() {
  return 42;
}
```

Výpis kódu 5: *Program v jazyku C pred kompiláciou.*

```
let wasmPath = "../out/main.wasm";
fetch(wasmPath).then(response =>
    response.arrayBuffer()
).then(bytes =>
↪  WebAssembly.instaOntiate(bytes)).then(results => {
    instance = results.instance;
    document.getElementById("container").textContent =
    ↪  instance.exports.main();
}).catch(console.error);
```

Výpis kódu 6: *Zavolanie funkcie main() pomocou JavaScript interoop.*

ho už takmer 90 % prehliadačov [7], nie je tento spôsob fallbacku praktický.

Ak by sme chceli pri kompilácii WASM súborov vytvoriť aj asm.js súbory, je to možné pomocou nástroja Emscripten, ktorý používa na kompiláciu vyššie spomínaný LLVM toolchain. WebAssembly je v porovnaní s asm.js o 30 % rýchlejšia a na mobilnej platforme Android až o 60 % [24].

5 Porovnanie WebAssembly a JavaScriptu

Nasledujúci zoznam obsahuje výhody, ktoré má WebAssembly oproti JavaScriptu.

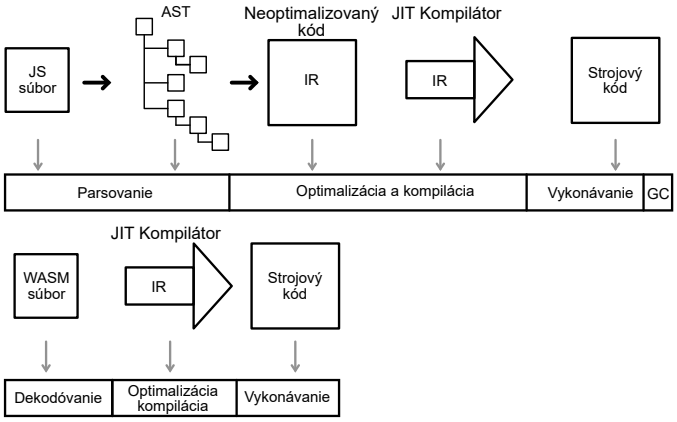
- Rýchlejšie načítanie binárneho súboru, ako textového súboru s JavaScriptom (aj keď je použitá komprimačná technológia gzip).
- Dekódovanie binárneho WASM súboru do AST je rýchlejšie a jednoduchšie ako parsovanie JavaScriptových súborov.
- Kompilácia a optimalizácia kódu je rýchlejšia, pretože WebAssembly je typová a je bližšie medzikódu,

elektrorevue
ISSN 1213 - 1539

který používajú JavaScriptové runtime. Optimalizácie pre WebAssembly je možné urobiť už pri kompilácii do WASM.

- Reoptimalizácia – na strane klienta nie je potrebná, pretože WebAssembly je typová. JavaScript engine nemusí optimalizovať WebAssembly kód a tým pádom nie je potrebná réžia na optimalizáciu a reoptimalizáciu, ako v prípade JavaScriptu
- Vykonávanie kódu zaberá menej času (aj kvôli vyššie spomenutým výhodám).

Na schéme 4 je vidieť pomer medzi vykonávaním JavaScriptového a WebAssembly kódu. Zo schémy je pomerovo vidieť jednoduchšie spracovanie a následné vykonanie WebAssembly kódu. Dekódovanie súboru je rýchlejší proces ako parsovanie, pretože engine nemusí pripradovať typy premenných na základe komplexného AST. Jednoduchší AST a implicitné typy premenných znamenajú rýchlejšiu JIT optimalizáciu a vykonávanie kódu [5].



Obrázek 4: *Porovnanie vykonávania JavaScriptového súboru a WASM súboru.*

6 Štandardy W3C a podpora prehliadačov

Štandard konzorcia W3C pre WebAssembly je aktuálne vo verzii 1 (z 5.12. 2019) a skladá sa z troch hlavných častí:

- WebAssembly Core Specification [26].
- WebAssembly JavaScript API [27].
- WebAssembly Web API [28].

Prvá špecifikácia s názvom *WebAssembly Core Specification*, hovorí o základných vlastnostiach WebAssembly ako sú formát kódu, jeho efektívne vykonávanie a jeho kompaktnosť. Druhý dokument s názvom *WebAssembly JavaScript API*, hovorí o spolupráci medzi WebAssembly a JavaScriptom. Posledný dokument s názvom *WebAssembly Web API* popisuje širšiu integráciu WebAssembly s webovou platformou.

Firefox vo verzii 52 bol prvý z prehliadačov, ktorý začal podporovať WebAssembly [3]. Momentálne je podpora WebAssembly v prehliadačoch viac ako 90 % [7]. Prehliadače implementujú WebAssembly pomocou globálneho objektu `WebAssembly.Global`, ktorý slúži na dynamické linkovanie viacerých modulov skrz import a export syntax [16]. Tento globálny objekt je dostupný iba cez JavaScriptové API, čo znamená, že WebAssembly zatiaľ nie je možné použiť v prehliadači bez inicializácie z JavaScriptového kódu [34].

7 Aplikácie WebAssembly

WebAssembly využívajú primárne výkonovo náročnejšie projekty:

- Oryol (3D framework) [9].
- PyBullet (Realtime Physics Simulation engine) [11].
- D3-force (Graph Simulation library) [6].
- ONNX.js (Open Neural Network Exchange) [18].
- Blazor WebAssembly (klientský framework pre vytváranie užívateľského rozhrania) [19].

Ako je vidieť zo zoznamu, jedná sa zatiaľ o veľmi špecifické použitie WebAssembly. Do budúcna sa však ráta s pridaním Garbage Collectoru a ostatných rozšírení pre lepšiu integráciu s „vyššími“ jazykmi.

8 Budúcnosť a rozpracovaná funkcionalita

Ako som spomínal v kapitolách vyššie, WebAssembly je vo vývoji a štandardizovaná je iba verzia 1. Špecifikácia WebAssembly 1 dosiahla nasledovné ciele:

- Kompiláciu do neutrálneho jazyka podobného assembleru, ktorý nie je určený pre špecifickú fyzickú architektúru.
- Rýchle vykonávanie WebAseembly kódu JavaScriptovým enginom.
- Kompaktnosť kódu, čo je dosiahnuté binárnou formou a reprezentáciou kódu pre zásobnikový počítač.
- Lineárna pamäť - stack machine. Toto je zabezpečenie voči tomu, aby WebAssembly kód mohol pristupovať iba k pamäti, ktorú mu určí JavaScriptový engine (pomocou TypedArray, čo je pole, ktoré obsahuje bajty pamäte).

Ďalší cieľ špecifikácie WebAssembly je pridať podporu pre funkcionality, ktoré umožnia vytvárať tzv. ťažké desktopové aplikácie, ako napríklad Photoshop alebo AutoCAD. Podpora takýchto aplikácii vyžaduje funkcionalitu, ktorá je popísaná v zoznamoch *Využitie vlastností moderného hardware* a *Vylepšenie loadtime WebAssembly v prehliadači*. WebAssembly je určená aj pre štandardný vývoj webových aplikácii. Požiadavky na funkcionalitu

elektrorevue
ISSN 1213 - 1539

sú popísané v zozname *Vylepšenia pre štandardný vývoj webových aplikácií.*

Pre lepšiu predstavu o aktuálnom stave funkcionality som pridal hodnotenie 0 až 4. Toto hodnotenie je založené na špecifikácii *WebAssembly Specification Release 1.0 (Draft, Apr 09, 2020)*[11].

Význam hodnotenia je následovný:

0. – Funkcionalita nemá draft a prebieha o nej diskusia.
1. – Pripravuje sa draft funkcionality.
2. – Funkcionalita má draft a prehliadače ju začínajú implementovať.
3. – Funkcionalita má draft a niektoré prehliadače ju experimentálne implementujú.
4. – Funkcionalita je schválená štandardom W3C.

8.1 Využitie vlastností moderného hardware

- Threading - pre prácu s modernými typmi procesorov je potrebné pridať podporu pre vlákna, čím sa otvára cesta k viac-jadrovým procesorom. **(3)**
- SIMD - je to ďalšia možnosť paralelného spracovania dát, pretože 1 inštrukciou dokážeme spracovať viacero dát. **(3)**
- Podpora AMD64 architektúry - s architektúrou x86 (32bit) je k dispozícii 4GB adresovateľnej pamäte. 64-bitová adresa pamäte umožňuje adresovať 16 exabajtov pamäte. **(0)**

8.2 Vylepšenie času načítania WebAssembly v prehliadači

- Streaming compilation - WebAssembly (JavaScript) engine dokáže spúšťať WASM súbor postupne, tak ako prichádzajú pakety. **(4)**
- Tiered compilation - odstupňovaná kompilácia, ktorá umožňuje pomocou viacnásobnej kompilácie dooptimalizovať výsledný kód. Táto technika sa často používa pri jazykoch, ktoré pre svoj beh používajú runtime (.NET, Java). **(3)**

8.3 Vylepšenia pre vysokoúrovňové jazyky

- Automatická správa pamäti - Garbage Collector (GC). **(2)**
- Exception Handling **(2)**
- Tail call - optimalizácia pre call stack volania. **(2)**
- WebAssembly system interface (WASI) - jedná sa o POSIX alternativu pre WebAssembly. **(2)**

[11] `https://webassembly.github.io/spec/core/`.

8.4 Vylepšenia pre štandardný vývoj webových aplikácií

- Integrácia ES modulov (používanie import a export direktív). **(3)**

8.5 Ostatné vylepšenia

Tieto vylepšenia nie sú súčasťou W3C WebAssembly špecifikácie. Komunitná skupina W3C ale diskutuje aj o nich.

- Jednoduchá a rýchla výmena dát medzi JavaScripotom a WebAssembly.
- Integrácia toolchain (balíčkovací systém, webpack, parcel, wasmpack)
- Implicitná HTTP cache - z cache si prehliadač vyberie správny WASM súbor, z ktorého je už vytvorený strojový kód (o ktorý sa postaral JavaScript engine).

S vylepšeniami popísanými vo vyšších zoznamoch, bude možné prepísať hlavné časti frameworkov ako Angular, Ember, React alebo Vue.js. Tento prepis bude mať za následok zrýchlenie behu týchto frameworkov, pretože vo WebAssembly bude možné paralelizovať výpočty, ktoré vykonávajú tieto frameworky (výpočty pre Virtual DOM a pod.). Pre React by to napríklad znamenalo prepis DPM diffing algoritmu, čo je algoritmus, ktorý porovnáva starý, reálny DOM a nový, aktualizovaný, virtuálny DOM.

WebAssembly znamená zmenu nielen pre vývoj webových aplikácií. So špecifikáciou WASI vznikne platforma, ktorá umožní zdielať a bezpečne spúšťať zdrojový kód skrz rôzne platformy a operačné systémy.

WebAssembly môžeme ale zhodnotiť ako štandard budúcnosti vývoja aplikácií, ktorý sa stále rozvíja. WebAssembly má obrovskú výhodu v tom, že od začiatku vzniká ako formálna špecifikácia pod vedením konzorcia W3C.

9 Záver

WebAssembly sa môže stať platformou ktorou sa kedysi chcela stať Java s mottom "Write once, run anywhere". Pôvodne bola určená pre prostredie webu no jej univerzálnosť a štandardizácia ju predurčuje k tomu, aby sa používala aj mimo webové prostredie. Hoci je WebAssembly často označovaná za nasledovníka JavaScriptu, nie je to pravda. WebAssemly treba chápať ako nízkoúrovňový, platoformovo nezávislý formát kódu. Tento formát je vytvorený ako compilation target, je teda od začiatku dizajnovaný s týmto cieľom, čo sa prejavilo na jej vlastnostiach. Vzťah WebAssembly a JavaScriptu by sa dal analogicky prirovnať ku vzťahu Javy a JavaScriptu v dobe vytvorenia JavaScriptu. Java mala byť zameraná na výkon a JavaScript na prácu s užívateľským rozhraním a bezpečné spúšťanie cudzieho kódu na danej platforme. Táto pôvodná myšlienka je presnou analógiu vzťahu WebAssembly a JavaScriptu. WebAssembly má slúžiť na výkonovo náročné aplikácie, medzi ktoré dnes patrí audio, video, kryptografia, umelá inteligencia a rozšírená realita. Medzi priority

elektrorevue
ISSN 1213 - 1539

na ktorých momentálne pracuje WebAssembly Working Group patrí integrácia s modernými viacjadrovými procesormi a rozšírenia, ktoré umožnia integráciu so správcami pamäte. Tieto rozšírenia umožnia expanziu WebAssembly a jej použite ako štandardizovaného formátu pre rôzne knižnice. Pre príklad, s WebAssembly by bolo možné distribuovať npm balíček node-sass ako jeden WASM balíček namiesto **55** verzií npm balíčku (operačný systém x inštrukčné sady).

Literatúra

[1] Belfiore J. *Microsoft Edge: Making the web better through more open source collaboration* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/#GmSJg4uFjBM5y8Hz.97>.

[2] Bright P. *The Web is getting its bytecode: WebAssembly* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://arstechnica.com/information-technology/2015/06/the-web-is-getting-its-bytecode-webassembly/>.

[3] Callahan D. *Firefox 52: Introducing WebAssembly, CSS Grid and the Grid Inspector* [online].[cit. 19. 04. 2020] Dostupné z URL: <https://kripken.github.io/mloc_emscripten_talk/cppcon.html>.

[4] Claburn T. *WebAssembly fandom kills Google's Portable Native Client* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://www.theregister.co.uk/2017/05/31/googles_portable_native_client_to_be_killed/>.

[5] Clark L. *What makes WebAssembly fast?* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/>.

[6] ColinEberhardt *ColinEberhardt/d3-wasm-force on github* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://github.com/ColinEberhardt/d3-wasm-force>.

[7] Deveria A. *Can I use WebAssembly* [online].[cit. 19. 04. 2020] Dostupné z URL: <https://caniuse.com/#search=WebAssembly>.

[8] Dr. Rauschmayer A. *Running code fast in web browsers: PNaCl versus asm.js* [online].[cit. 19. 4. 2020] Dostupné z URL: <http://2ality.com/2013/06/pnacl-vs-asmjs.html>.

[9] floooh *floooh/oryol on github* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://floooh.github.io/oryol/>.

[10] Greggman G. *Thoughts on asm.js vs PNaCl* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://games.greggman.com/game/thoughts-on-asm-js-vs-pnacl/>.

[11] kripken *kripken/ammo.js on github* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://github.com/kripken/ammo.js>.

[12] Larabel M. *Mozilla's Project Mortar Wants Pepper API Flash & PDFium In Firefox* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://www.phoronix.com/scan.php?page=news_item&px=Mozilla-Project-Mortar>.

[13] Lattner Ch. *Introduction to the LLVM - Compiler System* [online].[cit. 19. 4. 2020] Dostupné z URL: <http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf>.

[14] llvm.org *LLVM Overview* [online].[cit. 19. 04. 2020] Dostupné z URL: <https://llvm.org/>.

[15] MDN Web docs *Understanding WebAssembly text format* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format>.

[16] MDN Web docs *WebAssembly.Global* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Global>.

[17] MDN Web docs *WebAssembly* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/WebAssembly>.

[18] Microsoft *Microsoft/onnxjs on github* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://github.com/Microsoft/onnxjs>.

[19] Microsoft docs *Blazor documentation* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>.

[20] Nelson B. *LLVM on the Web - Using Portable Native Client to run Clang/LLVM in the Browser* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://llvm.org/devmtg/2015-02/slides/brad-pnacl.pdf>.

[21] Netscape Communications *NETSCAPE AND SUN ANNOUNCE JAVASCRIPT, THE OPEN, CROSS-PLATFORM OBJECT SCRIPTING LANGUAGE FOR ENTERPRISE NETWORKS AND THE INTERNET* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://web.archive.org/web/20080213171613/http://wp.netscape.com:80/newsref/pr/newsrelease67.html>.

elektrorevue
ISSN 1213 - 1539

[22] Rossberg, A., Titzer, B.L., Haas, A., Schuff, D.L., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F., Holman, M. . *Bringing the web up to speed with webassembly* Communications of the ACM 12 (Dec. 2018), 107-115. DOI: https://doi-org.ezproxy.lib.vutbr.cz/10.1145/3282510

[23] TECH.SAIGONIST.COM. *Assembly language vs Bytecode vs WebAssembly vs Asm.js* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://tech.saigonist.com/b/code/assembly-language-vs-bytecode-vs-webassembly-vs-asmjs.html>.

[24] Turner A. *WebAssembly Is Fast: A Real-World Benchmark of WebAssembly vs. ES6* [online].[cit. 19. 04. 2020] Dostupné z URL: <https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193>.

[25] W3C *W3C WebAssembly working group* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://www.w3.org/wasm/>.

[26] W3C *WebAssembly Core specification* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://www.w3.org/TR/wasm-core-1/>.

[27] W3C *WebAssembly JavaScript Interface* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://www.w3.org/TR/wasm-js-api-1/>.

[28] W3C *WebAssembly Web API* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://www.w3.org/TR/wasm-web-api-1/>.

[29] Wagner L. *asm.js AOT compilation and startup performance* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://blog.mozilla.org/luke/2014/01/14/asm-js-aot-compilation-and-startup-performance/>.

[30] Wagner L. *WebAssembly* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://blog.mozilla.org/luke/2015/06/17/webassembly/>.

[31] WebAssembly *WebAssembly documentation* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://webassembly.org>.

[32] WebAssembly *WebAssembly on github* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://github.com/webassembly>.

[33] WebAssembly Community Group *Types* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://webassembly.github.io/spec/core/syntax/types.html>.

[34] webassembly.org *JavaScript API* [online].[cit. 19. 04. 2020] Dostupné z URL: <https://webassembly.org/docs/js/>.

[35] webassembly.org *WebAssembly Design Rationale* [online].[cit. 19. 04. 2020] Dostupné z URL: <https://webassembly.org/docs/rationale/#why-a-binary-encoding>.

[36] webassembly.org. *FAQ* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://webassembly.org/docs/faq/>.

[37] Zakai A. *BIG WEB APP? COMPILE IT!* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://kripken.github.io/mloc_emscripten_talk/#/>.

[38] Zakai A. *EMSCRIPTEN & ASM.JS: C++'S ROLE IN THE MODERN WEB* [online].[cit. 19. 4. 2020] Dostupné z URL: <https://kripken.github.io/mloc_emscripten_talk/cppcon.html>.