

VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

WebAssembly atminties valdymo modelis

WebAssembly Memory Management Model

Bakalauro baigiamasis darbas

Atliko: Aidas Vaikšnoras (parašas)

Darbo vadovas: Partn. Doc. Vaidas Jusevičius (parašas)

Darbo recenzentas: Asist. Dr. Audrius Šaikūnas (parašas)

Vilnius – 2020

Turinys

Santrumpa	4
Abstract	5
Įvadas	6
1. „WebAssembly“ apžvalga	8
1.1. „WebAssembly“ minimalus produktas	8
1.2. „WebAssembly“ ateitis	8
1.3. „WebAssembly“ šiukšlių surinkėjo pasiūlymas.	9
2. Atminties valdymo sprendimai egzistuojančiose aplinkose	10
2.1. Atmintis	10
2.2. Nuorodų skaičiavimas (angl. <i>reference count</i>)	10
2.3. Šiukšlių surinkėjo konceptas	11
2.4. Šiukšlių surinkėjo algoritmai	12
2.4.1. „Žymėk - valyk“ (angl. <i>Mark And Sweep</i>)	12
2.4.2. Trijų spalvų „Žymėk - valyk“ (angl. <i>Tri-color</i>)	13
2.4.3. Atminties fragmentacija	14
2.4.4. Judantis šiukšlių surinkėjas (angl. <i>moving garbage collector</i>)	14
2.4.5. „Žymėk-perkelk“ (angl. <i>compacting mark and sweep</i>)	15
2.4.6. Kelių kartų šiukšlių surinkėjas (angl. <i>generational garbage collector</i>)	16
2.4.7. Algoritmų apibendrinimas	17
3. Egzistuojančių „WebAssembly“ atminties valdymo sprendimų apžvalga	19
3.1. „Blazor WebAssembly“ sprendimas „.Net“ vykdomojoje aplinkoje.	19
3.2. „Go“ sprendimas	20
3.3. „WebAssembly“ modulių veikimas įterpiančiose aplinkose	21
4. Bendrinis šiukšlių surinkėjo pasiūlymas	22
4.1. Nuorodų tipų pasiūlymas	22
4.2. Bendrinio šiukšlių surinkėjo pasiūlymas	23
4.3. Bendrinio šiukšlių surinkėjo pasiūlymo greitaveikos eksperimentinis tyrimas	24

4.4.	Bendrinio „WebAssembly“ šiukšlių surinkėjo realizacija	24
4.4.1.	Atminties išskyrimo realizacija	25
4.4.2.	Atminties išvalymo realizacija	25
4.4.3.	Greitaveikos įvertinimo eksperimentas	26
4.4.4.	Greitaveikos įvertinimo eksperimento rezultatai	26
4.5.	Bendrinio „WebAssembly“ šiukšlių surinkėjo palyginimas su egzistuojančiais sprendimais	27
4.5.1.	Greitaveikos įvertinimo eksperimento rezultatai	27
	Rezultatai ir išvados	30
	Šaltiniai	31

Santrumpa

Po sėkmingo „WebAssembly“ minimalaus produkto pristatymo ir pirmųjų projektų išaugo susidomėjimas technologija, leidžiančia leisti vietinėmis (angl. *native*) programavimo kalbomis parašytą kodą naršyklėje. Tačiau „WebAssembly“ minimalaus produkto stadijoje palaikomos programavimo kalbos, kurios atmintį valdo rankiniu būdu. Automatiškai atmintį valdančioms programavimo kalboms reikalingas šiukšlių surinkėjas, kurio specifikacija pradėta ruošti kitai „WebAssembly“ versijai. Bendrinio „WebAssembly“ šiukšlių realizacija užtruks, nes reikia parengti nuodugnią specifikaciją, bei visoms naršyklėms ją realizuoti. Todėl dalis programavimo kalbų kūrėjų realizuoja savo šiukšlių surinkėjus, kurie galėtų veikti WebAssembly aplinkoje. Išanalizavus šiukšlių surinkėjų veikimo principus, „WebAssembly“ architektūrą bei jau sukurtus automatinio atminties valdymo sprendimus „WebAssembly“ aplinkoje, siūloma sukurti bendrinį šiukšlių surinkėją „JavaScript“ kalboje, kuriuo galėtų naudotis skirtingų programavimo kalbų kūrėjai, nelaukdami naršyklių bendrinio šiukšlių surinkėjo realizacijos. Tokia šiukšlių surinkėjo realizacija, palyginus su „Blazor WebAssembly“ realizacija, valdant iki 1 MB dydžio atmintį yra lėtesnė nuo 11% iki 45%.

Raktiniai žodžiai

WebAssembly, šiukšlių surinkėjas, automatinis atminties valdymas, WebAssembly pasiūlymas

Abstract

After successful WebAssembly minimal valuable product, which enables to run native programming language written code on the browser, attention to the technology has risen. Even though, WebAssembly minimal valuable product supports only languages which manage memory manually. Programming languages which manage memory automatically need a garbage collector to be specified and implemented. Specification and implementation will take time, because all browser vendors will have to implement the specification on its own. Because of uncertain feature schedule, some programming language vendors start implementing its own garbage collectors for linear memory in order to make it work with WebAssembly. After analyzing different garbage collector algorithms, WebAssembly architecture and already existing automatic memory management solutions for WebAssembly, the author suggests implementing common garbage collector in JavaScript, which could be used by different programming languages vendors in short term. Compared to Blazor, such garbage collector implementation is slower from 11% to 45% in case of managing up to 1MB size memory.

Keywords

WebAssembly, garbage collector, automatic memory management, WebAssembly proposal

Ivadas

Temos aktualumas

Šiandien, mobiliųjų platformų ir interneto technologijų amžiuje, interneto aplikacijų (angl. *web application*) rinka nesustodama auga. Tai galima aiškinti programinės įrangos pasiekiamumo problema. Anksčiau, kuomet nebuvo išmaniųjų telefonų, paprastai programinė įranga buvo pasiekama darbinuose stacionariuose kompiuteriuose. Tačiau atsiradus dideliame skirtingų prietaisų skaičiui, atsirado poreikis tą pačią programą pasiekti iš skirtingų įrenginių. Geriausia priemonė tam pasiekti tapo interneto aplikacijos. Greitai besivystančios interneto technologijos leido kurti vis sudėtingesnes interneto aplikacijas. Tačiau neretai atsiduriama greitaveikos (angl. *performance*) aklavietėse, dėl interneto technologijų ribotų galimybių. Viena tokių problemų kyla iš populiariausios interneto technologijų programavimo kalbos „JavaScript“. Ši kalba yra dinamiška ir veikimo metu (angl. *runtime*) yra naršyklės interpretuojama. „JavaScript“ puikiai tinka interneto puslapiams praturtinti, tokiomis funkcijomis kaip dinamiškas duomenų apdorojimas, bendravimas su serveriais ir t.t. Tačiau ši kalba netinka sudėtingiems ir ilgiems skaičiavimams atlikti. Todėl ilgus metus buvo ieškoma būdų, kaip būtų galima optimizuoti sudėtingus algoritmus interneto aplikacijose. 2017 buvo pristatytas iki šiol perspektyviausias sprendimas – kritinės greitaveikos požiūriu aplikacijos vietas pakeisti į statiskai tipizuotomis kalbomis parašytus kodo gabalus, kurie sukompiliuojami į naršyklės suprantamą instrukcijų formatą, kitaip tariant į „WebAssembly“ kodą. Ši nauja technologija leidžia išspręsti kai kurias greitaveikos problemas interneto aplikacijose.

Problema

Norint efektyviai vykdyti sudėtingas ir daug atminties reikalaujančias programas labai svarbus efektyvus atminties valdymas. Didžioji dalis bendrosios paskirties programavimo kalbų pateikia neišreikštinį atminties valdymą panaudojant šiukšlių surinkėjus (angl. *garbage collector*). Pastarieji valdo programos paskirtą atmintį ir užtikrina efektyvų jos atlaisvinimą. Šiuo metu „WebAssembly“ šiukšlių surinkėjas dar tik kuriamas, todėl lieka neaišku ar „WebAssembly“ pasirinkti atminties valdymo sprendimą bus tinkami sudėtingoms programoms vykdyti.

Darbo tikslas ir uždaviniai

Šio darbo tikslas – išanalizuoti bei įvertinti „WebAssembly“ atminties valdymo modelio pasiūlymą ir pateikti pasiūlymus jos vystymui. Siekiant šio tikslo iškelti tokie uždaviniai:

- išanalizuoti „WebAssembly“ architektūrą ir veikimo principus;
- išanalizuoti egzistuojančius atminties valdymo modelius ir juos palyginti;
- palyginti egzistuojančius „WebAssembly“ atminties valdymo sprendimus;
- sukurti prototipą realizuojantį pasiūlytą automatinio atminties valdymo metodą;

- palyginti pasiūlyto metodo greitaveikos parametrus su egzistuojančių sprendimų atitinkamais parametrais;
- pateikti rekomendacijas „WebAssembly“ atminties valdymo modelio vystymui.

1. „WebAssembly“ apžvalga

„WebAssembly“ kaip ir „Assembly“ yra žmogui suprantama ir skaitoma programavimo kalba, tačiau labai žemo lygio, todėl ja programuoti yra labai sudėtinga. „WebAssembly“ dažniau naudojamas kaip formatas, skirtas vykdomųjų programų atvaizdavimui dvejetainių instrukcijų pavidalu, kuris gali būti leidžiamas skirtingose aplinkose labai efektyviai. Norint paleisti aukšto lygio programavimo kalba parašytą programą kompiuteryje, ją reikia sukompiliuoti į mašininį kodą. Apibendrinus, kompiliavimas vyksta dviem etapais: pirmiausia, aukšto lygio programa sukompiliuojama į abstrakčią tarpinę kalbą (angl. *intermediate representation*), kuri antru kompiliavimo etapu sukompiliuojama į specifinę architektūrai mašininį kodą (x64, x86 ARM) [Cla17]. „WebAssembly“ nėra specifinis architektūrinis tipas, o labiau konceptualus abstraktus instrukcijų formatas, tačiau taip pat yra kompiliuojamas iš tarpinės kalbos. Kuomet sukompiliuotas „WebAssembly“ modulis (sutrumpintai *wasm*) užkraunamas naršyklėje, jis dar kartą yra kompiliuojamas į jau konkrečios architektūros mašininį kodą [And17].

1.1. „WebAssembly“ minimalus produktas

Naujasis vykdomojo kodo formatas buvo pradėtas kurti kaip įrankis, leidžiantis vykdyti savosiomis (angl. *native*) kalbomis („C/C++“, „Rust“) parašytas programas kliento naršyklėje kartu su „JavaScript“ vykdomąja aplinka. Vis dėlto, „WebAssembly“ buvo ir toliau yra projektuojamas taip, kad būtų nepriklausomas nuo vykdomosios aplinkos. Planuojama, kad „WebAssembly“ potencialą bus galima išnaudoti tiek naršyklėse, tiek serverio aplikacijose („NodeJs“ aplinkoje). 2017 metais sukurtas minimalus panaudojamas produktas (angl. *minimal valuable product*) parodė technologijos potencialą ir sulaukė daug susidomėjimo iš programuotojų bendruomenės. Kolkas prototipas gali veikti tik vienoje gijoje ir leisti kodą, kuris buvo sukompiliuotas iš programavimo kalbų, kurios atmintį valdo rankiniu būdu („C/C++“, „Rust“) [Mil17].

1.2. „WebAssembly“ ateitis

Sėkmingas minimalus produktas leidžia svajoti apie ateitį, kurioje bet kokia kalba parašytas kodas galėtų būti leidžiamas naršyklėje. Todėl 2018 metais buvo pradėta ruošti specifikacija tolimesniam „WebAssembly“ vystymui. Specifikacija ruošiamą taip, kad „WebAssembly“ palaikytų šiuolaikines aukšto lygio programavimo kalbas, tokias kaip „Java“ ir „C#“. Šiomis kalbomis parašytos programos veikia sudėtingose vykdomosiose aplinkose („JVM“, „.Net“), kurios palaiko daugiagijįškumą (angl. *multithreading*) ir automatinį atminties valdymą, naudojant šiukšlių surinkėją (angl. *garbage collector*), todėl naujoje specifikacijoje šioms funkcijoms skiriamas bene didžiausias dėmesys.

1.3. „WebAssembly” šiukšlių surinkėjo pasiūlymas.

„WebAssembly“ šiukšlių surinkėjo specifikacija šiuo metu yra pirmojoje fazėje, kuomet teoriškai diskutuojama, kaip sistema turėtų veikti, ko tikisi naudotojai, kokie potencialūs panaudojimo atvejai. Diskusijos vyksta jau metus, nes užduotis yra tikrai sudėtinga ir sprendimui priimti reikia atsižvelgti į daugybę faktorių. Vis dėlto, šiuo metu jau yra apibrėžta, kad šiukšlių surinkėjas turėtų būti:

- greitas, bet tipizuotas;
- universalus;
- nepriklausomas nuo programavimo kalbos;
- galintis dirbti keliose gijose.

Tam kad būtų galima įgyvendinti šiukšlių surinkėją, atitinkantį tokius reikalavimus, reikia paruošti nemažai infrastruktūros. Daugumoje šiukšlių surinkėjus naudojamų aplinkų (angl. *managed environment*) yra naudojami dviejų rūšių duomenų tipai: reikšmės (angl. *value type*) ir nuorodos (angl. *reference type*). Dažniausiai reikšmių tipo objektai saugomi dėklo (angl. *stack*) struktūroje, o nuorodos tipo objektai saugomi krūvos (angl. *heap*) atminties struktūroje, tačiau nuorodos į tą atminties vietą saugojamos dėklo struktūroje. Kol kas „WebAssembly“ leidžia naudoti tik reikšmės tipo kintamuosius, nes jų naikinimas vyksta iškart, kai objektas išimamas iš dėklo naudojant išmetimo (angl. *pop*) operacija. Ne atsitiktinumas, kad aplinkos, kurios naudoja šiukšlių surinkėją, atminties valdymui turi nuorodų tipus. Šiukšlių surinkėjai seka objektus krūvos (angl. *heap*) atminties struktūroje būtent nuorodų pagalba. Todėl, bene pirmas darbas, kurį reikia atlikti norint perleisti atminties valymą šiukšlių surinkėjui, tai nuorodų tipų (angl. *reference type*) įvedimas į „WebAssembly“ instrukcijų formatą kaip tai yra kitose egzistuojančiose aplinkose („JVM“, „.Net“). Nuorodų tipų įvedimui į „WebAssembly“ yra sukurtas atskiras pasiūlymas, kuris jau yra įgyvendinimo stadijoje (angl. *implementation phase*). Siūloma, kad nuorodų tipai apimtų įrašus (angl. *record/struct*), masyvus su dinaminiu indeksavimu (angl. *array*) bei šių tipų kompoziciją. To pakanka daugumai atvejų programuojant kalbomis, naudojančiomis šiukšlių surinkėjus.

Taip pat šiukšlių surinkėjo projektavimą šiek tiek apsunkina „WebAssembly“ aplinkos apribojimai. Kiekvienam „WebAssembly“ moduliui veikimo aplinkoje yra išskiriamas linijinis atminties blokas, kuris yra 64 KB arba šio skaičiaus kartotinio bitų masyvas. Žinant tai, kad pagrindinės sudėtingesnės aplinkos kaip „JVM“ ar „.Net“ atmintį valdo ir skirsto krūvos (angl. *heap*) duomenų struktūroje, linijinis modelis tampa dar vienu iššūkiu įgyvendinant šiukšlių surinkėją „WebAssembly“ vykdomojoje aplinkoje [Ros17].

2. Atminties valdymo sprendimai egzistuojančiose aplinkose

Norint suprasti „WebAssembly“ atminties valdymo problemą bus atsižvelgiama į esamus atminties valdymo sprendimus. Šiandien egzistuoja du atminties valdymo konceptai: objektų nuorodų skaičiavimas (angl. *reference counting*) ir šiukšlių surinkimas. Šios dvi idėjos yra fundamentaliai skirtingi požiūriai į atminties valdymą. Esminis skirtumas yra tai, kad objektų nuorodų skaičiavimas yra kompiliavimo metu apibrėžtas algoritmas, kada galima atlaisvinti atmintį, kuomet šiukšlių surinkėjas sprendžia, kurią atminties dalį galima atlaisvinti programos vykdymo metu [Sri18].

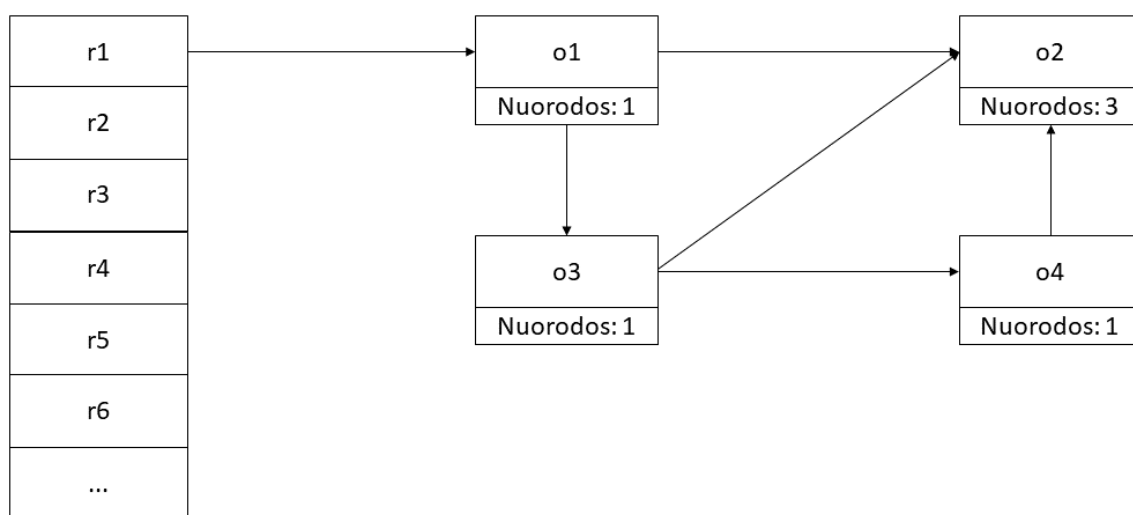
2.1. Atmintis

Kai kompiuterių moksle kalbama apie atmintį ir jos valdymą, dažniausiai kalbama apie objektų grafus, kurie prasideda šakniniuose objektuose (angl. *root*). Šakniniais objektais laikomi lokalūs kintamieji ar parametrai dėklo (angl. *stack*) atminties struktūroje, procesoriaus registrai arba statiniai kintamieji. Objektai yra programoje naudojami, jei yra pasiekiami grafe, keliaujant nuo šakninio objekto.

2.2. Nuorodų skaičiavimas (angl. *reference count*)

Nuorodų skaičiavimas yra gana paprastas būdas valdyti atmintį. Laikoma, kad objektas yra naudojamas, jei bent vienas kitas objektas turi nuorodą į jį. Tai įgyvendinama pridedant prie objekto papildomą nuorodų skaitiklio lauką, kuris nurodo, kiek kitų objektų jį naudoja (žr. 1 pav.).

Šakninių objektų sąrašas

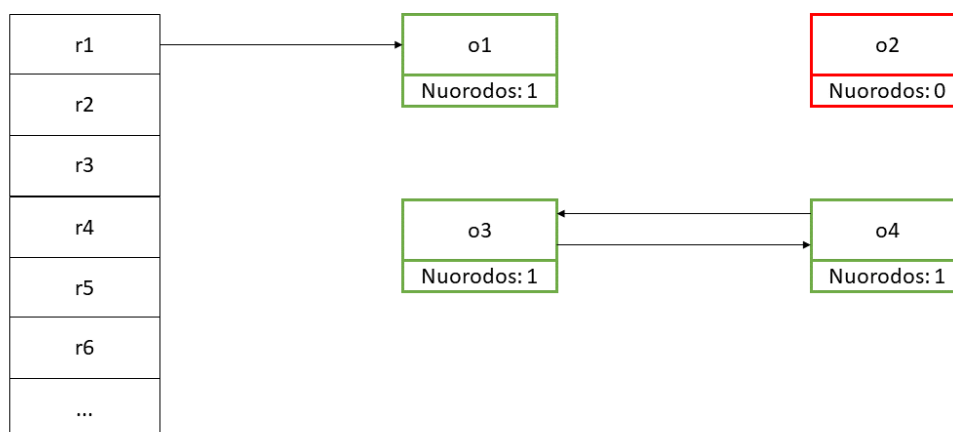


1 pav. Objektų nuorodų skaičiavimas

Kompiliavimo metu, prie tam tikrų operacijų pridedamos papildomos instrukcijos, kurios padidina arba sumažina nuorodų skaitiklio lauko reikšmę. Pavyzdžiui: priskyrimo operatorius nuorodų skaitiklį padidina, o kuomet objektas išeina iš savo galiojimo srities (angl. *scope*),

nuorodų skaitiklio reikšmė sumažinama. Kai nuorodų skaitiklio reikšmė pasiekia 0, objektas, priklausomai nuo sistemos realizacijos, gali būti išvalytas iš karto arba gali būti padėtas į nebenaudojamų objektų eilę išvalymui. Toks paprastas algoritmas nestabdo programos veikimo – atminties valdymas vyksta sklandžiai ir greitai. Tačiau nuorodų skaičiavimo algoritmas neišsprendžia problemos, kai keletas objektų laiko vienas kito nuorodas, taip sudarydami nuorodų ciklą (žr. 2 pav.). Tokiu atveju nuorodų skaitiklių reikšmės nėra 0 ir šie objektai negali būti išvalyti.

Šakninių objektų sąrašas



2 pav. Nuorodų ciklo problema

Programavimo kalbos, naudojančios nuorodų skaičiavimą atminties valdymui, šią problemą sprendžia silpnosiomis nuorodomis (angl. *weak reference*), kurios nepadidina nuorodų skaitiklio reikšmės. Tačiau šis sprendimas užkrauna atsakomybę programuotojui išreikštinai pažymėti silpnąsias nuorodas išeities kode (angl. *source code*).

2.3. Šiukšlių surinkėjo konceptas

Kompiuterių moksle šiukšlių surinkėjas yra alternatyva rankiniam nebenaudojamų atminties blokų atlaisvinimui. Šiukšlių surinkėjas pats identifikuoja, kurie objektai programos vykdymo metu yra nebenaudojami ir gali būti išvalyti. Paprastai šiukšlių surinkimas reikalauja papildomų laiko, procesoriaus ir atminties resursų, tačiau kartais šiukšlių surinkėjo valdomas atminties valymas gali veikti efektyviau nei rankinis. Be to, programuoti kalbomis, kurios palaiko automatinį šiukšlių surinkimą, yra žymiai paprasčiau, nes nereikia galvoti ir rūpintis, kada objektas bus nebenaudojamas ir atminties vieta galės būti išvalyta. Taip pat verta paminėti, kad automatinis šiukšlių surinkėjas dažnai leidžia išvengti atminties nutekėjimo problemų, kuomet atminties blokas nebuvo išreikštinai išvalytas dėl programuotojo klaidos. Šiukšlių surinkėjo idėja yra gana sena – John McCarthy ją pristatė 1959 metais supaprastindamas „Lisp“ kalbos atminties valdymą, paremtą „Žymėk – valyk“ (angl. *mark and sweep*) algoritmu. Tačiau šiukšlių surinkėjo atminties valdymo modelis nebuvo populiarus iki 1996 m., kai buvo išleista „Java“ 1.0 versija, kurios atminties valdymas buvo visiškai perleistas šiukšlių surinkėjui. Kaip bebūtų, per dešimtmečius

nuo John McCarthy idėjos ir pradinės realizacijos buvo sugalvota naujų algoritmų ir jų patobulinimų, kas galiausiai leido automatizuoti atminties valdymą pirmojoje „Java“ versijoje.

2.4. Šiukšlių surinkėjo algoritmai

Šiuo metu labiausiai yra paplitę atsekimo (angl. *trace*) principu veikiančys šiukšlių surinkėjai – algoritmai tikrinantys, ar visi objektai yra pasiekiami atminties struktūroje. Nebenaudojami objektai yra atpažįstami gana paprastai, nes jų negalima pasiekti nuorodomis iš šakninio objekto. Šie programoje nebenaudojami objektai gali būti ištrinti. Toks naudojamų objektų atpažinimas iš karto išsprendžia ciklinių priklausomybių problemą, kurios nuorodų skaičiavimas negali išspręsti neapkraudamas programuotojo. Tačiau dažniausiai atsekti, ar objektas yra pasiekiamas, reikia pereiti visą atmintį, kas reikalauja pristabdyti sistemos darbą ir atiduoti turimus įrangos resursus šalutiniam procesui. Šiukšlių surinkėjų algoritmai šią problemą sprendžia skirtingai paveikdami sistemos darbą. Vertinant šiukšlių surinkėjų algoritmų darbą yra svarbų pamatuoti, kokią įtaką šiukšlių surinkimas daro programos vykdymui. Tam pasitelkiama labai daug skirtingų metrikų, tačiau lygindami šiukšlių surinkimo strategijas apsiribosime šiomis trejomis:

1. Pauzės trukmė – laiko tarpas, per kurį sistema turi būti sustabdyta atliekant atminties valymą ar valymus. Ši metrika yra svarbi interaktyviose sistemose, kuriose svarbus sklandus, nepertraukiamas programos darbas.
2. Laikas, per kurį yra išvalomos visos šiukšlės iš atminties. Ši metrika yra svarbi sistemoms, kurios retai bendrauja su kitomis sistemomis. Pavyzdžiui programos, vykdančios ilgus skaičiavimus, kurioms svarbu kuo trumpesnis viso skaičiavimo laikas.
3. Atminties apribojimai įgyvendinant algoritmą. Dažniausiai algoritmai savo vykdymui turi rezervuoti dalį atminties arba pridėti papildomų duomenų prie objektų, kad būtų galima juos sekti.

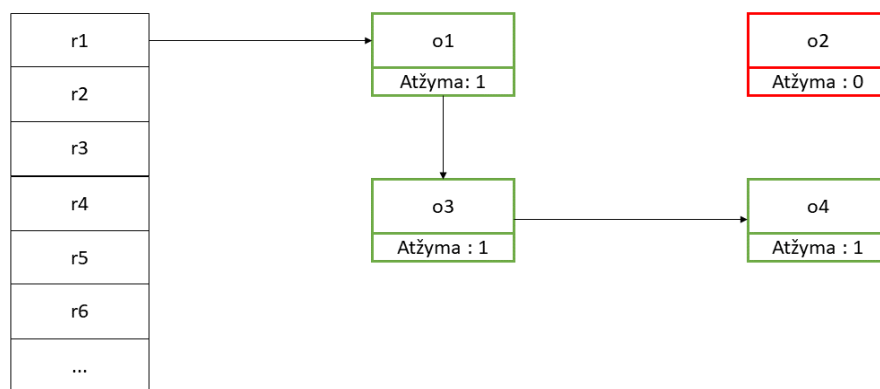
Pirmos dvi metrikos yra priešingos viena kitai. Kuo trumpesnis bendras valymo laikas, tuo mažiau valymų atliekama. Trumpiausias bendras valymo laikas gaunamas, kai visos šiukšlės yra išvalomos vienu sustojimu. Norint, kad sistema veiktų sklandžiau be didelių sustojimų, galima atmintį valyti po truputį, sistemos nesustabdant ilgam tarpui, tačiau prailginant bendrą šiukšlių surinkimo laiką. Kadangi visų metrikų gerų rezultatų vienu metu pasiekti neįmanoma, skirtingi šiukšlių surinkėjų algoritmai metrikas veikia skirtingai. Primityviausi algoritmai sustabdo visą programą vieną kartą ir išvalo visą atmintį iš karto. Toks buvo pirmasis „Žymėk-valyk“ (mark and sweep) algoritmas.

2.4.1. „Žymėk - valyk“ (angl. *Mark And Sweep*)

Pirmasis šiukšlių surinkėjo algoritmas „Žymėk - valyk“ veikia dviejų žingsnių principu: pirmiausia nenaudojamas objektas pažymimas, o vėliau objekto užimta atmintis atlaisvinama. Pažymėjimo žingsnis atliekamas prie objekto pridėdant papildomą bitą, kurį pavadinkime atžymos

bitu. Sukūrus objektą atžymos bitui priskiriama 0 reikšmė. Norint išvalyti nebenaudojamą sistemos atmintį reikia pereiti visus pasiekiamus objektus ir jų atžymos bitų reikšmes pakeisti į 1. Atlikus šį žingsnį galime matyti, kurie objektai su atžymos bitu lygiu 1 yra naudojami ir sistemai pasiekiami, o kurie objektai su atžymos bitu lygiu 0 yra nepasiekiami ir gali būti išvalyti. Sužymėjus objektus, galima atlikti valymo žingsnį, kuris atlaisvina nepasiekiamų objektų atmintį, o pasiekiamiems objektams pakeičia atžymos bitą į 0. Atlikus abu žingsnius turime tokią pat būseną kaip ir algoritmo pradžioje: visų objektų atžymos bitai lygūs 0 ir visi objektai yra sistemoje pasiekiami (žr. 3 pav.). [McC60]

Šakninių objektų sąrašas



3 pav. Atmintis po žymėjimo žingsnio

Šis paprastas šiukšlių surinkimo algoritmas nereikalauja daug papildomos atminties ir valymus atlieka tik tada, kai nepavyksta išskirti atminties naujam objektui. Tačiau visa programa turi sustoti, kol vyksta žymėjimas ir valymas. Be to, visas atminties grafas žymint turi būti pereitas bent vieną kartą, o dalis atminties – ir antrą kartą valant. Atminties valymo laikas – pilnas sustabdymas – reikšmingai didėja programai naudojant didesnės apimties atminties struktūras, nes vis daugiau objektų reikia pereiti [Aga17].

2.4.2. Trijų spalvų „Žymėk - valyk“ (angl. *Tri-color*)

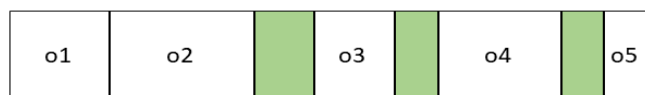
Primityvus „Žymėk-valyk“ algoritmas buvo patobulintas taip, kad nereikėtų stabdyti visos programos, įvedant sudėtingesnį, vadinamą trijų spalvų – baltos, pilkos ir juodos, objektų žymėjimą. Algoritmui įgyvendinti pasitelkiama trys sąrašai – po vieną sąrašą kiekvienai spalvai. Baltajame sąraše laikomos potencialių šiukšlių nuorodos. Juodajame – sistemoje naudojami objektai, kurie yra pasiekiami iš šakninio objekto ir neturi nuorodų į baltajame sąraše esančius objektus. Pilkajame – objektai pasiekiami iš šakninio objekto, tačiau dar pilnai neperskanuoti ar neturintys nuorodų į baltos spalvos objektus [Fis16]. Iš pradžių, visi naujai sukurti objektai yra pažymimi baltai. Tolimesnis objektų žymėjimas vyksta palaipsniui, vykdam natūralias objektų mutacijas. Nuo šakninio objekto pasiekiami objektai pažymimi pilkai. Šis žingsnis kartojamas, kol

pereinami visi pasiekiami objektai ir pažymimi pilkai. Kai visi pilko objekto pasiekiami objektai yra pilki, šio objekto spalva pakeičiama į juodą. Vykdam valymo žingsnį balti objektai yra ištrinami.

Šiam algoritmui nereikia sustabdyti visos programos vykdymo ilgoms pauzėms, nes žymėjimas vyksta tuomet, kai objektai yra naudojami – modifikuojami. Tačiau, verta paminėti, kad toks žymėjimas galiausiai pareikalauja daugiau: tiek laiko, tiek procesoriaus resursų nei vienas ilgesnis sužymėjimas ir išvalymas, kai yra sustabdoma visa programa. Kaip bebūtų, naudojant trijų spalvų žymėjimo algoritmą, sistemą išlieka interaktyvi, o tai yra labai svarbu programoms, kurios tiesiogiai komunikuoja su vartotoju, per grafinę vartotojo sąsają [Que17].

2.4.3. Atminties fragmentacija

Trijų spalvų žymėjimo algoritmas išsprendžia sistemos stabdymo ilgais intervalais valymo metu problemą. Tačiau, kaip ir primityvusis „Žymėk-valyk“ algoritmas, taip ir trijų spalvų algoritmas, po valymo atmintį palieka fragmentuotą. Programos pradžioje objektai atmintyje išsaugomi iš eilės, vienas paskui kitą, tačiau objektų gyvavimo laikas skiriasi – dalis objektų yra išvalomi anksčiau, kiti – vėliau. Išvalytų objektų vietoje gali būti išsaugoti kiti objektai, tačiau tik tokio pačio dydžio ar mažesni. Po kurio laiko gali nebeatsirasti vietos didesniems objektams, nors susumavus visą laisvą vietą, jos būtų dar pakankamai. Šis reiškinys vadinamas atminties fragmentacija (žr. 4 pav.). [Zor90]

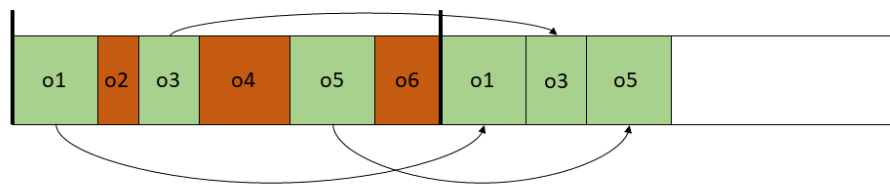


4 pav. Atminties fragmentacija

Šia problemą galima išspręsti iš naujo atmintyje surikiuojant objektus iš eilės. Tačiau tam vis tiek reikia sustabdyti programos veikimą, nes visų objektų nuorodos turi būti pakeistos. Tačiau šis žingsnis yra neišvengiamas, todėl dažniausiai naudojami kompaktiški „Žymėk-valyk“ algoritmai, kurie kartas nuo karto objektus perrikiuoja.

2.4.4. Judantis šiukšlių surinkėjas (angl. *moving garbage collector*)

Fragmentacijos problemą visiškai išsprendžia judantis (angl. *moving*) šiukšlių surinkimo algoritmai. Pirmasis judantis algoritmas buvo pristatytas dar 1969 m., kai buvo sugalvota visą aplikacijai skirtą atmintį padalyti į dvi lygias dalis ir perkeldinti objektus iš vienos pusės į kitą. Objektai yra išsaugojami vienoje pusėje tol, kol joje pakanka atminties naujam objektui. Kai vietos naujam objektui pristinga, visi pasiekiami objektai yra nukopijuojami į kitą atminties dalį. Taip pirmojoje dalyje lieka sistemai nebepasiekiami objektai, kurie yra iš karto išvalomi (žr. 5 pav.) [SR78].



5 pav. Judantis šiukšlių surinkėjas

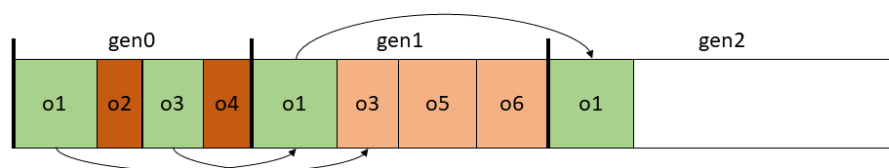
Šio algoritmo pirmasis privalumas yra tai, kad norint išskirti atmintį naujam objektui, nereikia ieškoti jam vietos atmintyje, nes objekto išsaugojamas tēra krūvos (angl. *heap*) atminties struktūros viršūnės nuorodos atnaujinimas. Toks objektų kūrimas yra efektyvesnis net už rankinį atminties valdymą, kuriam reikia, kiekvieno objekto kūrimo metu, pereiti dalį atminties, kad būtų rastas tinkamas laisvos atminties gabalas. Šiam algoritmui, galima sakyti, šiukšlių valymo procesas nieko nekainuoja, nes visos šiukšlės išvalomos viena operacija. Kitas, jau paminėtas, privalumas – išspęsta fragmentacijos problema, nes kaskart, nukopijuojant objektus į kitą atminties pusę, jie yra surikiuojami iš eilės. Be to, šis kopijuojantis algoritmas dirba tik su pasiekiamais objektais ir per šiukšlės kopijavimo metu nekeliauja, todėl veikia greičiausiai, kai yra mažai naudojamų objektų, kuriuos reikia perkelti, ir daug šiukšlių, kurias reikia išvalyti. Kaip bebūtų, algoritmas kopijavimo metu turi visiškai sustabdyti sistemą. Sustojimų trukmė priklauso, nuo sistemai rezervuotos atminties dydžio. Kaip ir „Žymėk-valyk“ algoritmas, taip ir šis, veikiant programose su didelėmis rezervuotomis atmintimis, stabdo sistemą vis ilgesniems tarpams. Šis algoritmas taip pat reikalauja dvigubai daugiau vietos arba turi dvigubai dažniau atlikinėti valymus, nei „Žymėk-valyk“ algoritmas, nes pusė rezervuotos atminties didžiąją dalį laiko yra tuščia ir nenaudojama. Be to, ilgai naudojami objektai yra kaskart pirmyn atgal kopijuojami [Bye07].

2.4.5. „Žymėk-perkelk“ (angl. *compacting mark and sweep*)

Pasiteisinus kopijuojančio algoritmo strategijai, buvo sugalvota kaip galima išspręsti didžiausią šio algoritmo trūkumą – dvigubą reikalingos atminties kiekį. Pasitelkus paprasto „Žymėk-valyk“ algoritmo pirmąjį žymėjimo žingsnį programoje pasiekiami objektai yra sužymimi. Antru žingsniu visi pasiekiami objektai nukopijuojami tame pačiame atminties bloke kompaktiškai nuo bloko pradžios. Kadangi visi pasiekiami objektai kompaktiškai surikiuoti, todėl lengva pastebėti ribą, tarp šiukšlių ir gyvų objektų, todėl valymo žingsnis yra labai pigus. Be to ilgai gyvenantys objektai yra linkę laikytis atminties struktūros pradžioje, todėl ilgainiui jų kopijuoti kiekvieno valymo metu nebereikia [Goe03].

2.4.6. Kelių kartų šiukšlių surinkėjas (angl. *generational garbage collector*)

Tobulinant šiukšlių surinkėjus buvo pradėta skirtingus algoritmus taikyti kartu, taip, esant tam tikroms aplinkybėms, pagerinant šiukšlių surinkimo efektyvumą. Pavyzdžiui: atlikus „Žymėk-valyk“ algoritmo žingsnius, likusius objektus verta perrašyti kompaktiškai atmintyje, tam, kad išsaugant naujus objektus nereikėtų ieškoti laisvos tinkamos vietos, o būtų galima juos iškart įterpti po paskutinio išsaugoto objekto. Tačiau bene didžiausias šiukšlių surinkėjo patobulinimas buvo ne naujas revoliucinis algoritmas, o empirinis pastebėjimas, kad daugiausia šiukšlėmis patampa labai trumpai gyvavę objektai. Nustatyta, kad 80-98% objektų yra nebenaudojami kas 1MB naujų objektų sukūrimų. Toks pastebėjimas buvo išvystytas į keleto kartų (angl. *generations*) atminties padalijimą. Priklausomai nuo aplinkybių kartų skaičius skiriasi, tačiau tiek „.Net“, tiek „JVM“ skirtingų kartų šiukšlių surinkėjai yra dalinami į tris skirtingo dydžio kartas, kur mažiausia, 1-16 MB, yra jauniausia karta (angl. *gen0*), kiek didesnė – vyresnė karta (angl. *gen1*) ir didžiausia – seniausia (angl. *gen2*). Naujai sukurti objektai talpinami jauniausios kartos atminties struktūroje. Kuomet daugiau objektų nebepavyksta išsaugoti, gyvi objektai nukopijuojami į vyresnės kartos struktūrą ir jauniausios kartos atmintis yra išvaloma vienu kartu. Šis valymas vadinamas mažuoju valymu (angl. *minor sweep*), nes valoma mažiausios kartos atminties blokas, kuris talpina nedaug objektų, todėl šis valymas trunka gana neilgai. Šis procesas kartojamas, kol užpildoma antroji karta. Valant antrosios kartos atminties dalį objektai analogiškai perkopijuojami į trečios kartos atmintį. Galiausiai, kai užpildoma ir trečios kartos atminties dalis, pradedamas įvykdyti didysis valymas (angl. *major sweep*), kuomet yra valoma visa atmintis, todėl programa turi būti sustabdyta ilgesniam laikui nei atliekant mažuosius valymus (žr. 6 pav.). [LH83]



6 pav. Kelių kartų šiukšlių surinkėjas

Skirtingų kartų algoritmas paveldi geriausias kopijuojančio algoritmo savybes – pigų naujų objektų kūrimą ir pigų šiukšlių valymą, tačiau papildo kopijuojančio algoritmo šiukšlių valymo procesą, tuo kad jauni objektai, kurie, tikėtina, greitai tampa nebenaudojamais, yra valomi dažnai sustabdant sistemą trumpam laikotarpiui, o ilgai gyvenantys objektai – valomi ir kopijuojami retai, taip neatliekant nereikalingo darbo dažnų valymų metu.

2.4.7. Algoritmų apibendrinimas

Apžvelgus egzistuojančius atminties valdymo sprendimus, matome, kad negalima vienareikšmiškai nustatyti, kuris algoritmas veikia geriausiai visais atvejais. Todėl apibendrinamas šiukšlių surinkėjų algoritmus (žr. 1 lentelė) autorius remiasi darbo pradžioje minėtomis trejomis charakteristikomis: pauzės trukmė, laiku, per kurį yra išvalomos visos šiukšlės iš atminties, atminties apribojimais ar papildomais reikalavimais bei pastebėjimais, apie algoritmo privalumus ir trūkumus.

1 lentelė. Šiukšlių surinkėjų algoritmų apibendrinimas

Algoritmas	Pauzės trukmė	Visas valymo laikas	Atminties apribojimai, reikalavimai	Pastabos
Nuorodų skaičiavimas	Pauzės nėra.	Nedidelis.	Papildomas nuorodų skaitiklio laukas prie kiekvieno objekto.	Neišsprendžia ciklinių objektų priklausomybių
„Žymėk-valyk“	Viena ilga pauzė sutampanti su visu valymo laiku.	Viena ilga pauzė, kol išvaloma visa atmintis.	Papildomas atžymos laukas prie kiekvieno objekto.	Valymo laikas didėja priklausomai nuo rezervuotos atminties kiekio
Trijų spalvų „Žymėk-valyk“	Pauzių nėra arba labai trumpos.	Visos atminties valymo laikas išsitiesia ilgam laikui.	Papildomas spalvos laukas prie kiekvieno objekto.	Nestabdo programos vienam ilgam valymui. Valymus atlieka palaipsniui.
Judantis	Neilgos pauzės, nes valoma pusė atminties	Valymai vyksta gana dažnai todėl, visas valymo laikas gana didelis.	Reikalauja dvigubai daugiau atminties.	Išsprendžia fragmentacijos problemą. Tačiau veikimui reikalinga dvigubai daugiau atminties.
„Žymėk-perkelk“	Neilgos pauzės	Visas valymo laikas gana didelis, nes atliekama daug nedidelių valymų	Papildomas laukas atžymos laukui.	Išsprendžia fragmentacijos problemą.
Kelių kartų	Trumpos pauzės	Visas valymo laikas gana didelis, nes atliekama daug nedidelių valymų	Efektyviai veikti reikalinga daugiau atminties.	Išsprendžia fragmentacijos problemą, ne visi objektai yra tikrinami kiekvieno valymo metu.

Apžvelgus algoritmus, galime pastebėti visą šiukšlių surinkėjų evoliuciją: nuo paties primityviausio – „Žymėk-valyk“ iki paties sudėtingiausio – kelių kartų šiukšlių surinkėjo. Algoritmų evoliucijos metu buvo išspręstos prieš tai egzistavusių algoritmų problemos, todėl galima teigti, kad optimalus šiuo metu yra kelių kartų šiukšlių surinkėjas, kuris yra naudojamas tiek „JVM“, tiek „.Net“ vykdomosiose aplinkose.

3. Egzistuojančių „WebAssembly“ atminties valdymo sprendimų apžvalga

Diskutuojant apie „WebAssembly“ šiukšlių surinkėją verta prisiminti diskusijos šaknis ir motyvus. Automatinio atminties valdymo problema sprendžiama tam, kad būtų techniškai įmanoma leisti jau sukurtas ar naujai kuriamas programas, kurios yra parašytos automatiškai atmintį valdomomis kalbomis, dažniausiai tai yra „Java“ ir „C#“. Šiomis kalbomis parašytos programos paprastai leidžiamos specialiose vykdomosiose aplinkose („JVM“, „.Net“), kurios yra atsakingos už objektų kūrimą ir naikinimą pasitelkiant šiukšlių surinkėjus. Šie šiukšlių surinkėjai veikia labai efektyviai, nes yra pritaikyti ir sukalibruoti veikti tose vykdomosiose aplinkose. Norint atkartoti tokį pat ar maksimaliai panašų programos veikimą internetinėje platformoje naudojant „WebAssembly“ reiktų kiekvienai aplinkai suprogramuoti šiukšlių surinkėją, kuris veiktų linijinėje atmintyje. Taip būtų išsaugota programos vykdymo greitis (angl. *runtime performance*). Tačiau išsaugant maksimaliai panašų veikimą, būtų prarandama labai daug užkrovimo laiko (angl. *load time*), nes prie wasm modulio, kuris pasižymi labai mažu užimamu atminties kiekiu, būtų siunčiamas sudėtingas šiukšlių surinkėjas, kuris galėtų užimti kelis ar net keliasdešimt kartų daugiau vietos, nei pats wasm modulis. Kadangi tiek wasm modulis, tiek šiukšlių surinkėjas būtų klientui siunčiami tinklu, internetinės aplikacijos užkrovimo laikas išaugtų proporcingai šiukšlių surinkėjo dydžiui. Šis požiūris netenkina nei „WebAssembly“ bendruomenės, nei potencialių vykdomų aplinkų kūrėjų.

Spręsti aukščiau apibrėžtą problemą naujoje „WebAssembly“ versijoje siūloma sukuriant šiukšlių surinkėją, kuris keliautų kartu su visa „WebAssembly“ vykdomąja aplinka. Šis šiukšlių surinkėjas būtų kompromisas tarp skirtingų vykdomųjų aplinkų poreikių ir, akivaizdu, kad negalėtų būti taip gerai subalansuotas kaip kad būtų originalus konkrečios vykdomosios aplinkos šiukšlių surinkėjas. Tačiau sutariama, kad programos vykdymo greitis skirtumas, gali būti paaukota vardan mažesnio duomenų kiekio siuntimo tinklu, taip reikšmingai nepadidinant užkrovimo laiko. Kadangi šiandien technologijos vystosi labai sparčiai ir kompiuteriai tampa vis galingesni ir galingesni, programos vykdymo greitis sumažėtų mažai arba vartotojui visai nepastebimai. Toks kompromisas tenkina tiek „WebAssembly“ bendruomenę, tiek vykdomųjų aplinkų kūrėjus, kadangi nei vienam iš jų nereiktų kurti šiukšlių surinkėjų skirtų linijinei atminčiai. Kaip bebūtų, vykdomųjų aplinkų kūrėjai nepraranda galimybės kurti savo šiukšlių surinkėjus ir juos siųsti kartu su wasm moduliu, jei siūlomas bendras „WebAssembly“ šiukšlių surinkėjas nepasiteisintų.

3.1. „Blazor WebAssembly“ sprendimas „.Net“ vykdomojoje aplinkoje.

„Microsoft“ nusprendė nelaukti bendrinio šiukšlių surinkėjo pabandyti jau dabar rasti būdą kaip paleisti „.Net“ programas naršyklėje. Tam buvo pradėtas kurti „Blazor WebAssembly“ karkasas leidžiantis kurti interaktyvias klientines interneto aplikacijas (angl. *client-side web*

application), kuris jungia „Razor“ interneto aplikacijų kūrimo karkasą bei „WebAssembly“ technologiją.

„Blazor WebAssembly“ karkasas veikia tokiu būdu:

- „C#“ kodas ir „Razor“ dokumentai sukompiliuojami į „.Net“ modulius (angl. *assembly*);
- šie moduliai kartu su visa „.Net“ vykdomąja aplinka parsiumčiami į naršyklę;
- „Blazor WebAssembly“ sukonfigūruoja vykdomąją aplinką ir užkrauna „.Net“ modulius.

Reikia paminėti, kad „Blazor“ naudoja ne standartinę „.Net“ vykdomąją aplinką, o atviro kodo realizaciją – „Mono“, kuri naudojama kuriant mobiliąsias programėles „Android“ ir „iOS“ operacinėms sistemos „Xamarin“ karkaso pagalba. „Mono“ vykdomoji aplinka orientuota į klientines aplikacijas (mobiliosios programėlės), todėl „WebAssembly“ tai yra geriausiai tinkanti „.Net“ vykdomosios aplinkos realizacija, kuri yra kompaktiškesnė ir pritaikyta mobiliems įrenginiams.

Šiuo metu „Mono“ „.Net“ vykdomoji aplinka gali leisti „WebAssembly“ dvejais režimais: interpretuojant (angl. *interpreted mode*) ir prieš vykdymą sukompiliuojant (angl. *ahead-of-time compiled mode*).

Interpretuotam režime „Mono“ vykdomoji aplinka yra sukompiliuojama į „WebAssembly“ mašininį kodą, bet „.Net“ moduliai lieka standartiniais „.Net“ moduliais (.dll). Naršyklė užkrauna ir paleidžia vykdomąją aplinką, kuri savo ruožtu užkrauna ir paleidžia „.Net“ modulius. Šie moduliai yra interpretuojami tuo metu, kai juos reikia vykdyti, todėl veikia šiek tiek lėčiau.

Prieš vykdymą sukompiliuotam režime tiek „.Net“ vykdomoji aplinka, tiek „.Net“ moduliai sukompiliuojami į „WebAssembly“, todėl kodas gali būti vykdomas naršyklės iš karto. „Blazor WebAssembly“ jau galima išbandyti, tačiau karkasas vis dar yra aktyviai vystomas ir dar nėra galimas bendros prieigos aplikacijoms. [RL20]

3.2. „Go“ sprendimas

„Go“ programavimo kalbos kūrėjai tai pat nusprendė pabandyti paleisti kodą naršyklėje pasitelkiant „WebAssembly“ technologiją. Žinant tai, kad „WebAssembly“ moduliai pasižymi kompaktiškumu ir mažu operuojamos atminties dydžiu, leidžia juos lyginti su programomis ir vykdymo sprendimais, kurie yra naudojami mikrovaldikliuose. Mikrovaldikliai dėl mažo fizinio dydžio, bei skaičiavimo galios dažniausiai gali paleisti nedideles programas. „Go“ programavimo kalbos kūrėjai šių išvalgų pagrindu perpanaudojo mikrovaldikliuose naudojamą „Go“ sprendimą – „TinyGo“. „TinyGo“ tai „Go“ kalbos kompiliatorius ir vykdomosios aplinkos realizacija skirta mikrovaldikliams.

Šis kompiliatorius gali daugumą „Go“ programų sukompiliuoti naudodamas „LLVM“ į mašininį kodą skirtingoms architektūroms. Šis sprendimas palaiko per 27 mikrovaldiklių tipų. Naujas kompiliatorius reikalingas būtent dėl mikrovaldiklių specifikos ir naudojamų specialių instrukcijų. Be to standartinė vykdomoji aplinka užima apie 1MB, kas mikrovaldikliams ir „WebAssembly“ yra gerokai per daug ir sukeltų greیتaveikos problemų.

Vykdomoji aplinka taip pat adaptuota ir kompaktiška palaiko tik būtiniausius paketus ir funkcija tokias kaip atminties išskyrimo paketas, operacijos su baziniais tipais ir kita. Vykdomoji aplinka, su būtiniausiais moduliais, skirta „WebAssembly“ gali būti sumažinta iki 16 KB [Kum19]

3.3. „WebAssembly“ modulių veikimas įterpiančiose aplinkose

Prieš pradėdant nagrinėti galimus atminties valdymo sprendimus „WebAssembly“, verta įsigilinti kaip atmintis valdoma šiandien veikiančiuose wasm moduluose.

Pirmas žingsnis, norint paleisti wasm modulį reikia jį sukompiliuoti. Skirtingoms kalboms naudojami skirtingi įrankiai, tačiau rezultatas gaunamas tas pats – dvejetainis wasm modulis, kurį įterpiančios aplinkos (angl. *embedder*) sugeba paleisti.

Antras žingsnis – tai „WebAssembly“ modulio inicializavimas pasitelkiant aplinkos programavimo sąsają (angl. *application programming interface*), kurioje paleidžiamas wasm modulis. Pvz.: „JavaScript“ programavimo sąsajos specifikacija įpareigoja naršyklių kūrėjus realizuoti „WebAssembly“ statinį objektą, kurio pagalba būtų galima užkrauti parsųstą wasm modulį. Kartais wasm modulio inicializavimas gali užtrukti, todėl „JavaScript“ „WebAssembly“ programavimo sąsaja apibrėžia ir asinchroninę inicializavimo galimybę, pasitelkiant pažadų (angl. *promise*) mechanizmą.

Trečias žingsnis – naudoti eksportuotas wasm modulio funkcijas įterpiančios aplinkos išeities kode.

Paprastai linijinė atmintis valdoma tiesiogiai „WebAssembly“ dvejetainio formato instrukcijų. Tačiau linijinė atmintis gali būti pasiekama ir inicializuojama įterpiančios aplinkos kodo lygyje pvz.: prieš inicializuojant patį modulį. Taip pat, „WebAssembly“ modulis gali importuoti įterpiančios aplinkos pateikiamas funkcijas, kurios dažnai yra būtinos vykdomajai aplinkai. Pvz.: bazinė rašymo į konsolę funkcija. [HRS17]

4. Bendrinis šiukšlių surinkėjo pasiūlymas

„WebAssembly“ šiukšlių surinkėjo pasiūlymas turėtų apibrėžti bendrinę specifikaciją, kurią visos naršyklės turės realizuoti. Šis pasiūlymas turėtų atitikti daugumos kalbų kūrėjų poreikius bei greitaveikos reikalavimus. Specifikacija, be kitų naujų funkcijų, bus išplėsta papildomais nuorodų tipais bei šiukšlių surinkėjo pasiūlymu. [Cla18]

4.1. Nuorodų tipų pasiūlymas

„WebAssembly“ duomenų tipai yra naudojami apibrėžti duomenų išsidėstymą atmintyje, skirtingai nei aukštesnio lygio programavimo kalbose, kuriose tipai dažniausia užtikrina operacijų ir duomenų sąsajas kompiliavimo metu. „WebAssembly“ tipai, dar kitaip vadinami statiniai tipai šiukšlių surinkėjui suteikia informacijos apie saugojamus duomenis, todėl juos galima efektyviau valdyti [IK00].

Ankstyvojoje realizacijos stadijoje „WebAssembly“ palaiko vos keletą reikšmės tipų (angl. *value type*). Tai yra 32 ir 64 bitų sveikųjų skaičių tipai (*i32*, *i64*) bei 32 ir 64 slankaus kablelio (angl. *floating point*) tipai (*f32*, *f64*). Taip pat yra funkcijos tipas skirtas saugoti nuorodai į funkciją (*functype*). Šių kelių bazinių tipų pakanka duomenimis operuoti rankiniu būdu linijinėje atmintyje, kaip tai daro „C/C++“ bei „Rust“ programavimo kalbos. Tačiau dauguma automatiškai atmintį valdančių programavimo kalbų operuoja objektais ir nuorodomis į juos. Dažnai nuoroda yra tiesiog paprastas sveikasis skaičius atmintyje. Todėl norint atskirti ar reikšmė atmintyje yra tiesiog skaičius ar nuoroda, reikia įvesti tipą nuorodomis. Šie nuorodų tipai leidžia objektus sieti vienus su kitais, todėl šiukšlių surinkėjai žinodami šakninio objekto nuorodą gali pasiekti visus susietus objektus ir pavyzdžiui sužymėti kaip pasiekiamus, kitaip tariant, atlikti žymėjimo fazę šiukšlių surinkimo procese.

Nuorodų tipų pasiūlymas apibrėžia tokius „WebAssembly“ tipų pakeitimus:

1. Įvedami nuorodų tipai *anyref*, *funcref*, *nullref*, kuriuos apibendrina tipas *reftype*:
`reftype ::= anyref | funcref | nullref`
2. Reikšmės tipai galės būti skaičiai arba nuorodos. Įvedami skaičių tipas *numtype* apibendrinantis sveikųjų skaičių ir slankiojo kablelio formatus, bei reikšmės tipas *valuetype* apibendrins skaičiaus ir nuorodos tipus:

```
numtype ::= i32 | i64 | f32 | f64  
valuetype ::= <numtype> | <reftype>
```

Be to nuorodos tipo reikšmių numatytoji reikšmė bus *null*.

3. Įvedamos naujos instrukcijos padėsiančios operuoti nuorodomis:

```
ref.null nustatys nuorodos reikšmę į null  
ref.is_null patikrins ar nuorodos reikšmė yra null
```

Šie nuorodų tipų pakeitimai yra būtinas žingsnis šiukšlių surinkėjo realizacijos kelyje. Jie leis atskirti skaičių reikšmes nuo nuorodų atmintyje. Šis nuorodų tipų pakeitimų pasiūlymas yra realizacijos fazėje. [Ros18]

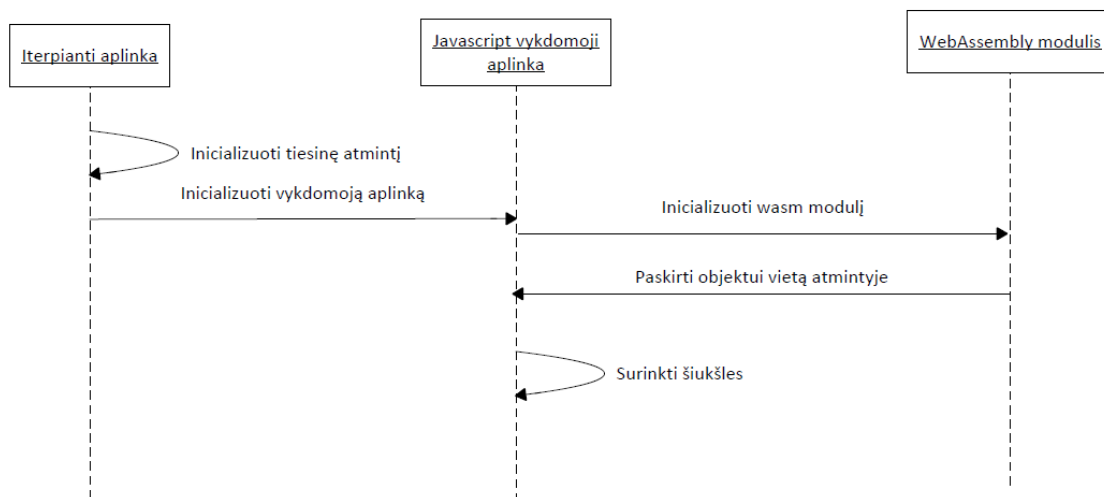
Be nuorodų tipų pakeitimų šiukšlių surinkėjo pasiūlymas siūlo įvesti dar keletą naujų tipų, kurie leis optimaliai išsaugoti agreguotas duomenų struktūras atmintyje:

1. Statiškai indeksuotų reikšmių struktūros tipas *struct* pvz.: taško objekto struktūra trimatėje erdvėje „WebAssembly“ atrodytų taip: `(type $point (struct (field $x f64) (field $y f64) (field $z f64)))` Šiam tipui inicializuoti turėtų būti įvestas *new* operatorius, kuris leistų inicializuoti visus struktūros laukus.
2. Dinamiškai indeksuotų vienodo tipo reikšmių masyvo tipas *array*. Vektoriaus struktūra „WebAssembly“ atrodytų taip: `(type $vector (array (mut f64)))`.
3. Masyvo inicializavimui planuojama įvesti instrukciją *array.new*, kuri parametrai priimtų pradinę inicializavimo reikšmę bei masyvo ilgį. Be to masyvui būtų įvesta *array.len* instrukcija, kuri gražintų elementų skaičių masyve.
4. Nuorodų palyginimo instrukcija *req.eq*, kuri palygintų nuorodų reikšmes ir gražintų tiesą (angl. *true*) jei nuorodos rodo į tą pačią atminties vietą. Pvz.: `(ref.eq (struct.new $point ...) (struct.new $point ...)) ;; false`.

Šios struktūros ir masyvų duomenų tipai turėtų būti nuorodos tipo potipiai, kurie galėtų būti valdomi šiukšlių surinkėjo. Svarbu paminėti, kad šių tipų palaikymą, bet kokių atveju, turėtų realizuoti kiekvienos vykdomosios aplinkos kūrėjai norintys palaikyti automatinį atminties valdymą šiukšlių surinkėjo pagalba. Todėl tai dar kartą parodo bendro šiukšlių surinkėjo realizacijos svarbą. [Ros19]

4.2. Bendrinio šiukšlių surinkėjo pasiūlymas

Šio darbo autorius siūlo į problemą pažvelgti iš kitos pusės ir pasvarstyti apie bendrinį automatinį atminties valdymo sprendimą, be visų naršyklių realizacijų, o realizavus vieną šiukšlių surinkėją „JavaScript“ lygyje tarp „WebAssembly“ modulio ir kodo, kuris naudoja eksportuotas *wasm* modulio funkcijas. Žinant tai, kad iš „wasm“ modulio galima kviesti „JavaScript“ funkcijas, leidžia pakankamai lengvai sukurti tarpinę „JavaScript“ vykdomąją aplinką, kuri automatiškai išskirtų ir valytų atmintį linijinėje struktūroje (žr. 7 pav.).



7 pav. Javascript vykdomosios aplinkos vieta „WebAssembly“ architektūroje

Panašias „JavaScript“ tarpines aplinkas naudoja dauguma egzistuojančių programavimo kalbų vystytojų, tačiau jie tai daro kitais tikslais pvz.: klaidų ar papildomos informacijos išvedimui į naršyklės konsolę.

4.3. Bendrinio šiukšlių surinkėjo pasiūlymo greیتaveikos eksperimentinis tyrimas

Prieš realizuojant pilnai veikiančią bendrinį šiukšlių surinkėją verta iš pradžių realizuoti dalį šiukšlių surinkėjo ir eksperimentiškai ištirti tokios automatinio atminties valdymo svarbiausią greیتaveikos parametą – visos atminties išvalymo laiką. Bendras atminties išvalymo laikas „WebAssembly“ yra svarbu todėl, kad dauguma wasm modulių yra pakankamai kompaktiški ir naudoja nedaug atminties. Tokiais atvejais, kai programai vykdyti užtenka nedaug atminties ir modulio atsakymo greitis nėra kritinė savybė, išvalant visas atmintyje esančias šiukšles vienu kartu, sumažinamas bendras atminties valymo laikas.

„JavaScript“ vykdomosios aplinkos dalinėje realizacijoje buvo sukurtos dvi funkcijos. Viena atminties išskyrimui ir objekto paskyrimui (angl. *allocate*) linijinėje atmintyje. Antroji skirta atminties išvalymui, kai atminties nepavyksta išskirti. Atminties valymui buvo pasirinkta „žymėk-valyk“ strategija dėl algoritmo paprastumo bei trumpiausio viso atminties išvalymo laiko savybės. Be to, svarbu priminti, kad dalinės realizacijos tikslas nėra rasti geriausiai veikiančią algoritmą, o įvertinti šiukšlių surinkėjo greیتaveiką naudojant „JavaScript“ vykdomąją aplinką.

4.4. Bendrinio „WebAssembly“ šiukšlių surinkėjo realizacija

Dalinėje „JavaScript“ vykdomosios aplinkos realizacijoje, linijinei atminčiai valdyti buvo naudojama „JavaScript“ „DataView“ struktūra, kuri leidžia valdyti atminties bloką nesirūpinant platformos bitų tvarka atmintyje (angl. *endianness*). „DataView“ struktūros programuotojo sąsaja

leidžia naudotis keletu funkcijų, kurios skirtos nustatyti tam tikru adresu reikšmę atmintyje (angl. *set*) bei gauti reikšmę žinant jos adresą (angl. *get*). Šios funkcijos apibrėžtos skirtingiems duomenų tipams pvz.: 32 ir 64 bitų slankaus kablelio skaičiui, sveikajam skaičiui, sveikajam neneigiamam skaičiui ir taip toliau.

4.4.1. Atminties išskyrimo realizacija

Realizuojant dalį „JavaScript“ vykdomosios aplinkos buvo pasirinkta dirbti su vieno tipo duomenimis, kuris tinka saugoti atmintyje tiek nuorodas tiek skaičių reikšmes – 32 bitų neneigiamo skaičiaus tipas (angl. *unsigned integer*). Objektų valdymui ir sekimui pasitelkta pagalbinė asociatyvaus masyvo (angl. *map*) duomenų struktūra nuorodoms ir papildomai informacijai – objekto dydžiui bei objekto atžymos reikšmei - šiukšlių surinkimo metu saugoti. Atminties išskyrimo metu svarbu žinoti, kokių adresu atmintis yra laisva tam, kad būtų galima lengvai įterpti naują reikšmę. Šiam mechanizmui pasitelktas statinis kintamasis, kuris visuomet rodo į laisvą atminties vietą. Kadangi visi objektai talpinami atmintyje iš eilės, tai šio statinio kintamojo reikšmė visuomet rodo į atminties vietą po paskutinio objekto. Atminties išskyrimo algoritmą galima apibūdinti šiais rekursyviais žingsniais:

1. Einama per paskiriamo objekto laukus.
2. Jei laukas yra skaičius, atmintyje paskutiniame laisvame adrese išsaugoma reikšmė bei paskutinis adresas padidinamas 4 baitais (nes pradinė „WebAssembly“ versija operuoja tik 32 bitų duomenų tipais).
3. Jei laukas yra objektas, tai rekursyviai kviečiama atminties išskyrimo funkcija, kuri kartoją algoritmą naujam objektui nuo pirmo žingsnio.
4. Išskirto objekto nuoroda išsaugojama pagalbiniame asociatyviame masyve kartu su paskiriamo objekto dydžiu baitais.

Be to, šakninių objektų nuorodos išsaugojamos papildomai šakninių nuorodų sąrašė, kad neužtekus atmintyje vietos, būtų galima nuo jų pradėti šiukšlių valymo žymėjimo žingsnį.

4.4.2. Atminties išvalymo realizacija

Atminties išvalymas vyksta dviem žingsniais - žymėjimu ir nepasiekiamų objektų valymu. Žymėjimo žingsnis vyksta taip:

1. Einama per šakninių objektų nuorodų sąrašą.
2. Tikrinama ar to objekto laukų reikšmė yra nuoroda, kuri buvo išsaugota nuorodų asociatyviame masyve.
3. Jei objekto laukas yra nuoroda, tai ši nuoroda pažymima kaip pasiekiamas ir algoritmas rekursyviai kartojamas nuo antro žingsnio.

Žymėjimo pabaigoje turime asociatyviame masyve sužymėtus pasiekiamus objektus, todėl galima atlikti antrąjį žingsnį – valymą.

1. Einama per visas nuorodas esančias asociatyviame masyve.
2. Jei atžymos reikšmė yra neigiama, atlaisvinama atmintis, kurią objektas buvo užėmęs.
3. Jei atžymos reikšmė yra teigiama, ji gražinama į pradinę reikšmę taip paruošiant atmintį kitam valymo ciklui.

4.4.3. Greitaveikos įvertinimo eksperimentas

„WebAssembly“ modulis gali operuoti skirtingo dydžio atmintimis pradedant nuo 64 KB iki 4GB. Šis apribojimas egzistuoja dėl maksimalios 32 bitų nuorodos reikšmės. Tačiau šis 4GB apribojimas yra labiau teorinis. Praktiškai naršyklės turi griežtesnį „JavaScript“ objekto dydžio apribojimą, kuris leidžia kurti objektus iki 2GB. Kadangi „WebAssembly“ atmintis yra „DataView“ tipo „JavaScript“ objektas, tai ir jam galioja naršyklės nustatytas maksimalus dydis.

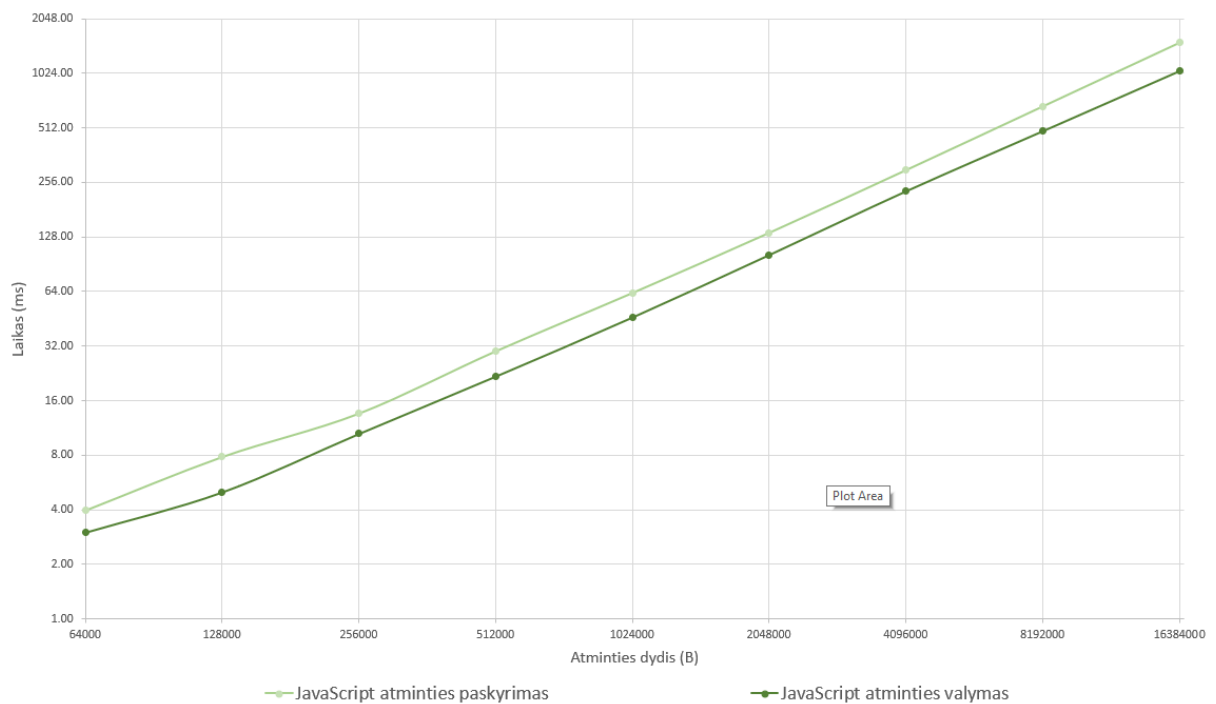
Norint išanalizuoti sprendimo veikimo greitaveiką bus atliktas eksperimentas su skirtingo dydžio atminties blokais pradedant nuo minimalaus 64KB dydžio. Testavimas tęsiamas kol šiukšlių surinkimo laikas neviršys 1 sekundės. Laikysime, kad didesnės pauzės šiukšlių surinkėjui nėra priimtinos. Eksperimento metu bus paskirstomi 16 baitų dydžio objektai, kurių struktūra yra: 2 skaičių laukai, 1 nuorodos laukas į objektą su vienu skaičiaus lauku. Eksperimentas vyks dviem žingsniais:

1. Objektai užpildoma visa atmintis.
2. Atmintis išvaloma „Žymėk - valyk“ šiukšlių surinkėjo.

Eksperimento metu šie žingsniai kiekvienam atminties dydžiui kartojami 50 kartų ir vertinami objektų paskirstymo ir šiukšlių valymo žingsnių laikų vidurkiai.

4.4.4. Greitaveikos įvertinimo eksperimento rezultatai

Atlikus eksperimentą matome, kad šios šiukšlių surinkėjo realizacijos greitaveika tiesiškai priklauso nuo operuojamo atminties dydžio (žr. 8 pav.). Taip pat galima pastebėti, kad objekto paskyrimas užtrunka ilgiau nei šiukšlių išvalymas. Kadangi realaus pasaulio programose dažniausiai objektai kuriami palaipsniui, todėl tokios pauzės nesusidarytų, bet paskyrimo operacija turėtų įtakos bendrai programos greitaveikai. Tačiau šiukšlių surinkimas pasiekė 1 sekundės pauzės ribą dirbant su 16384000 baitų (~16 MB) dydžio atmintimi. Todėl galima teigti, kad šis algoritmas galėtų būti naudojamas „WebAssembly“ moduluose, kurie operuoja iki 16 MB dydžio atmintimi.



8 pav. „JavaScript“ vykdomosios aplinkos greیتaveika

4.5. Bendrinio „WebAssembly“ šiukšlių surinkėjo palyginimas su egzistuojančiais sprendimais

Bendrinio šiukšlių surinkėjo greیتaveika gali atrodyti pakankama, tačiau gautus rezultatus verta palyginti su jau anksčiau aptartais sprendimais. Nors „TinyGo“ vykdomosios aplinkos kūrėjai skelbia palaikantys automatinę atminties valdymą „WebAssembly“ aplinkoje, tačiau vėliausioje „TinyGo“ versijoje (0.12.0) išreikštinis šiukšlių surinkimas bei atminties stebėjimo įrankiai neveikia, todėl eksperimentas bus atliktas naudojant „Blazor“ karkasą. Eksperimento metu bus paskirstomi panašios struktūros objektai ir stebimas laikas, kaip greitai objektai paskiriami ir galiausiai išvalomi. Šis eksperimentas paremtas „Net“ System bibliotekos klase „GC“, kuri leidžia išreikštinai paleisti kelių kartų (angl. *generational garbage collector*) „Net“ šiukšlių surinkėją ir išmatuoti, kiek šiukšlių valymas užtruko. Eksperimentas vykdomas analogiškai pradedant nuo 64 KB objektų paskyrimo ir jų išvalymo. Eksperimentas tęsiamas dvigubinant objektų dydį iki 16 MB, kad būtų galima lygiagrečiai palyginti gautus rezultatus su „JavaScript“ vykdomosios aplinkos rezultatais.

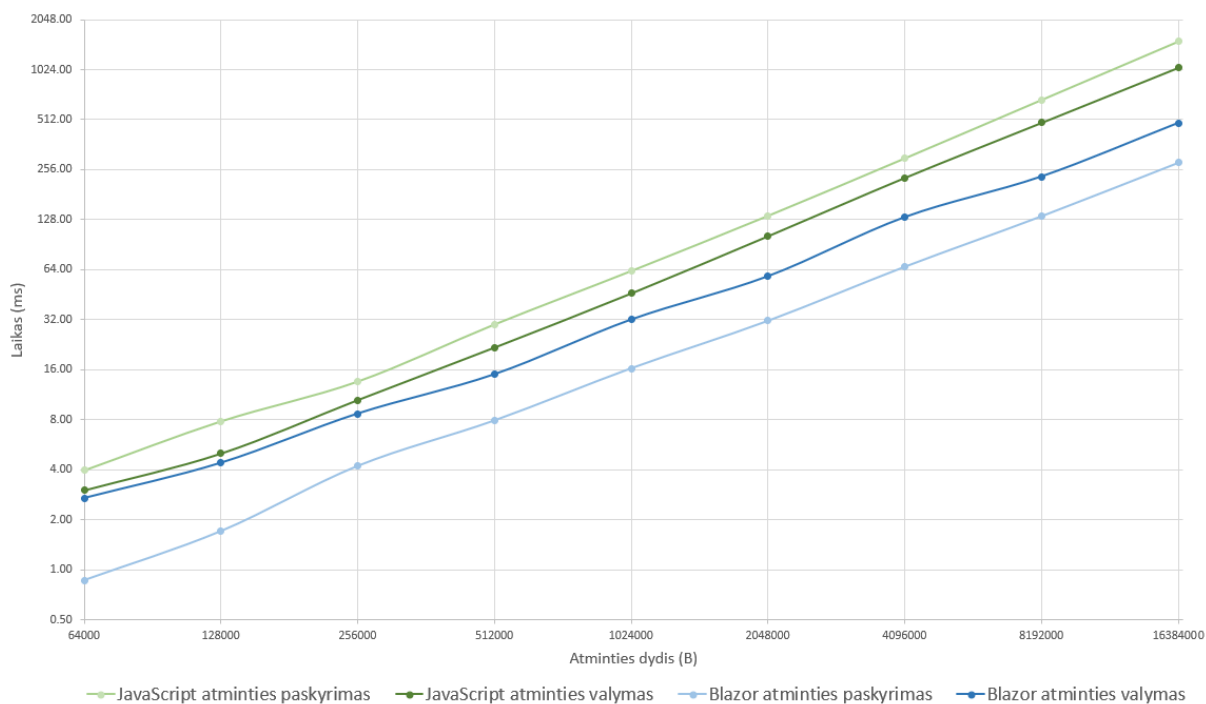
4.5.1. Greitaveikos įvertinimo eksperimento rezultatai

Atlikus eksperimentą matome, kad „Blazor“ atminties valdymo greیتaveika geresnė, nei „JavaScript“ vykdomosios aplinkos (žr. 2 lentelė).

lentelė 2 „JavaScript“ ir „Blazor“ vykdomųjų aplinkų atminties valdymo palyginimas

Atminties dydis baitais	„JavaScript“ objektų paskyrimas (ms)	„JavaScript“ šiukšlių surinkimas (ms)	„Blazor“ objektų paskyrimas (ms)	„Blazor“ šiukšlių surinkimas (ms)	Paskirstymo laiko palyginimas kartais	Šiukšlių surinkimo laiko palyginimas kartais
64000	3,96	3,00	0,86	2,70	4,60	1,11
128000	7,82	5,00	1,70	4,40	4,60	1,14
256000	13,58	10,50	4,20	8,70	3,23	1,21
512000	29,96	21,75	7,90	15,00	3,79	1,45
1024000	62,88	46,06	16,30	32,00	3,86	1,44
2048000	135,02	101,78	31,50	58,30	4,29	1,75
4096000	299,40	228,52	66,70	132,20	4,49	1,73
8192000	672,66	490,10	134,50	232,30	5,00	2,11
16384000	1511,26	1051,10	282,45	487,83	5,35	2,15

Galima pastebėti, kad „Blazor“ su objektų paskirstymo užduotimi susitvarko ženkliai greičiau – 4,6 karto greičiau skirstant 64000 baitų ir 5,35 kartus greičiau skirstant 1638400 baitų. „Blazor“ taip pat ženkliai greičiau susitvarko ir su šiukšlių surinkimu – 1.1 karto greičiau išvalant 64000 baitų ir 2,15 karto greičiau išvalant 1638400 baitų.



9 pav. „JavaScript“ ir „Blazor“ vykdomųjų aplinkų atminties valdymo greitaveikos palyginimas

„JavaScript“ vykdomoji aplinka veikia lėčiau dėl „JavaScript“ dinaminės kalbos prigimties, kuomet instrukcijos yra interpretuojamos vykdymo metu. Taip vykdomas kodas negali būti

maksimaliai optimizuotas. Todėl prieš laiką sukompiliuojamos programavimo kalbos (pvz. „Blazor“ naudojama „C#“) yra pranašesnės nei interpretuojamos programavimo kalbos, greitaveikos atžvilgiu, nes instrukcijos gali būti įvykdytos iš karto. Eksperimento metu pastebimas greitaveikos skirtumas gali būti paaiškintas fundamentaliais „JavaScript“ ir „C#“ programavimo kalbų skirtumais.

Nors „Blazor“ yra pranašesnis paskirstant objektus ir valant atmintį, reikėtų nepamiršti „.Net“ vykdomosios aplinkos dydžio ir jos atsiuntimo laiko užkraunant aplikaciją. Nors tai „.Net“ vykdomoji aplinka gali būti atsiųsta vieną kartą ir išsaugota naršyklės atmintyje, bet vertėtų šį parametą taip pat įtraukti į palyginimą (žr. 3 lentelė).

3 lentelė Vykdomųjų aplinkų dydžio palyginimas

Vykdomoji aplinka	Dydis	Laikas
„JavaScript“ vykdomoji aplinka	4200 B	53 ms
„Blazor“ vykdomoji aplinka	4 600 000 B	1550 ms

Apibendrinant eksperimentų rezultatus, galima teigti, jog „Blazor“ sprendimas veikia greičiau, tačiau nedidelės atminties eksperimentuose (iki 1 MB) pranašumas yra nuo 11% iki 45%. Nedaug atminties reikalaujančiose programose šį pranašumą nustelbia „.Net“ vykdomosios aplinkos atsiuntimo ir inicializavimo laikas (pirmą kartą 1550 ms). Todėl mažos atminties wasm moduliams gerai tinka paprastas ir nedidelis „JavaScript“ kalba realizuotas šiukšlių surinkėjas. Tačiau, daugiau atminties reikalaujančiose programose didelį pranašumą turėtų „.Net“ vykdomoji aplinka, naudojanti kelių kartų šiukšlių surinkėją (angl. *generational garbage collector*), kuris nestabdo programos ilgą kiekvieno valymo metu, atlieka dalinius nedidelius atminties valymus. Todėl didesnės apimties programoms geriau tiktų „.Net“ siūlomas sprendimas.

Kadangi iš anksto sunku nustatyti, koks algoritmas veiktų geriausiai visais atvejais, „WebAssembly“ kūrėjai ir projektuotojai turėtų į tai atsižvelgti ir leisti programuotojui pasirinkti, kuri atminties valdymo strategija veiktų optimaliausiai konkrečiu atveju. Juk „WebAssembly“ modulis galės būti naudojamas atliekant ilgus skaičiavimus, kuriems yra svarbu sugeneruoti rezultatą per trumpiausią laiką. Tokiam atvejui optimaliausias būtų paprastas giją stabdantis „Žymėk-perkelk“ algoritmas, pasižymintis ilgomis pauzėmis, tačiau trumpiausiu bendru visos atminties išvalymo laiku. Taip pat, toks algoritmas veiktų gerai wasm moduluose, kuriuose nebūtu išskiriama daug atminties. Kitais atvejais, kai „WebAssembly“ modulis generuos interaktyvius atsakymus, bus svarbus sklandus nepertraukiamas sistemos darbas, kuriam užtikrinti geriausiai tiktų kelių kartų šiukšlių surinkimo algoritmas, panašus į naudojamą originalioje vykdomojoje aplinkoje. Tačiau šis šiukšlių surinkėjas būtų gana sudėtingas ir reikalautų daugiau atminties efektyviam programos veikimui.

Rezultatai ir išvados

Darbo rezultatai:

1. Atlikta „WebAssembly“ architektūros, veikimo principų bei technologinio aktualumo apžvalga.
2. Išanalizuoti, aprašyti ir palyginti egzistuojantys atminties valdymo modeliai skirtingose programavimo kalbų aplinkose.
3. Išanalizuotos, aprašytos ir palygintos egzistuojančios automatinio atminties valdymo strategijos, šiukšlių surinkėjų veikimo principai.
4. Išanalizavus „WebAssembly“ atminties struktūrą ir specifiką, pateiktas atminties valdymo bendrinis sprendimas „WebAssembly“ aplinkai.
5. Bendrinis pasiūlymas palygintas su specifiniais programavimo kalbų atminties valdymo „WebAssembly“ aplinkoje sprendimais.

Darbo išvados:

1. „WebAssembly“ šiukšlių surinkėjas turėtų būti vienas sprendimas visoms programavimo kalboms, pateikiamas kartu su „WebAssembly“ platforma.
2. Turėtų būti leidžiama pasirinkti „WebAssembly“ šiukšlių surinkėjo algoritmą, taip parenkant optimalias atminties valdymo charakteristikas.
3. Ilgiems skaičiavimams aktualus bendras skaičiavimo laikas, todėl turėtų būti naudojamas greičiausiai visą atmintį išvalantis „Žymėk - valyk“ algoritmas Interaktyviems moduliams, svarbus nepertraukiamas sistemos darbas todėl, turėtų būti naudojamas judantis arba kelių kartų šiukšlių surinkėjas.
4. „WebAssembly“ šiukšlių surinkėjas turėtų būti kiek galima panašus į originalių vykdomųjų aplinkų šiukšlių surinkėjus.
5. „WebAssembly“ moduliams naudojantiems iki 16 MB atminties galėtų būti naudojamas „JavaScript“ aplinkoje realizuotas šiukšlių surinkėjas.

Šaltiniai

- [Aga17] Chirag Agarwal. *Mark-and-Sweep: Garbage Collection Algorithm*. Prieiga per internetą: <https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm>. Žiūrėta 2020-04-21
- [And17] Rasmus Andersson. *Introduction to WebAssembly*, 2017. Prieiga per internetą: <https://rsms.me/wasm-intro>. Žiūrėta 2020-04-20
- [Bye07] Rick Byers. *Garbage Collection Algorithms*. Prieiga per internetą: <https://courses.cs.washington.edu/courses/csep521/07wi/prj/rick.pdf>. Žiūrėta 2020-04-19
- [Cla17] Lin Clark. *A crash course in assembly*, 2017. Prieiga per internetą: <https://hacks.mozilla.org/2017/02/a-crash-course-in-assembly/>. Žiūrėta 2020-03-08
- [Cla18] Lin Clark. *WebAssembly's post-MVP future: A cartoon skill tree*, 2018. Prieiga per internetą: <https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/>. Žiūrėta 2020-03-07
- [Fis16] Jim Fisher. *How does tricolor garbage collection work?* Prieiga per internetą: <https://jamesfisher.com/2016/11/11/tricolor-gc>. Žiūrėta 2019-06-02
- [Goe03] Brian Goetz. *Garbage Collection Algorithms* Prieiga per internetą: <https://www.ibm.com/developerworks/library/j-jtp10283/index.html>. Žiūrėta 2020-04-26
- [HRS17] HAAS, Andreas, et al. *Bringing the web up to speed with WebAssembly*. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017. p. 185-200. Žiūrėta 2020-04-20
- [IK00] IGARASHI, Atsushi; KOBAYASHI, Naoki. *Garbage collection based on a linear type system*. In: *Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC'00)*. 2000. Žiūrėta 2020-04-20
- [Kum19] Sendil Kumar. *(Tiny)Go to WebAssembly* Prieiga per internetą: <https://dev.to/sendilkumarn/tiny-go-to-webassembly-5168>. Žiūrėta 2020-05-01
- [LH83] Henry Lieberman and Carl Hewitt. *A real-time garbage collector based on the lifetimes of objects*, 1983, 420–425. Žiūrėta 2020-04-26
- [McC60] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, 1960, 184–195. Žiūrėta 2020-04-26
- [Mil17] Chris David Mills. *WebAssembly Concepts*, 2017. Prieiga per internetą: <https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts>. Žiūrėta 2020-04-26

- [RL20] Daniel Roth, Luke Latham . *Introduction to ASP.NET Core Blazor*, 2020. Prieiga per internetą: <https://docs.microsoft.com/en-/aspnet/core/blazor/?view=aspnetcore-3.1&viewFallbackFrom=aspnetcore-3>. Žiūrėta 2020-04-21
- [Ros17] Andreas Rossberg . *GC extension*, 2017. Prieiga per internetą: <https://github.com/WebAssembly/gc/blob/master/proposals/gc/Overview.md>. Žiūrėta 2020-04-20
- [Ros18] Andreas Rossberg . *Reference Types for WebAssembly* , 2018. Prieiga per internetą: <https://github.com/WebAssembly/reference-types/commits/master/proposals/reference-types/Overview.md>. Žiūrėta 2020-04-20
- [Ros19] Andreas Rossberg . *GC Extension*, 2019. Prieiga per internetą: <https://github.com/WebAssembly/gc/blob/master/proposals/gc/Overview.md#structs-and-arrays>. Žiūrėta 2020-04-20
- [SR78] Graham, Susan L. and Ronald L. Rivest. “List Processing in Real Time on a Serial Computer.” (1978). 282 - 285 Žiūrėta 2020-04-26
- [Sri18] Mohanesh Sridharan. *Garbage Collection vs Automatic Reference Counting*, 2018. Prieiga per internetą: <https://medium.com/computed-comparisons/garbage-collection-vs-automatic-reference-counting-a420bd4c7c81>. Žiūrėta 2019-04-26
- [Que17] Andrew Quesenberry. *Tri-Color Incremental Mark & Sweep*. Prieiga per internetą http://wiki.luajit.org/New-Garbage-Collector#gc-algorithms_tri-color-incremental-mark-sweep. Žiūrėta 2020-04-26
- [Zor90] Benjamin Zorn. 1990. *Comparing mark-and sweep and stop-and-copy garbage collection*.. p 89 – 90. Žiūrėta 2020-04-25