UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

SYSTÉM NA DISTRIBUOVANIE VÝPOČTOV POMOCOU INTERNETOVÉHO PREHLIADAČA

BAKALÁRSKA PRÁCA

2013

Bruno Cuc

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

SYSTÉM NA DISTRIBUOVANIE VÝPOČTOV POMOCOU INTERNETOVÉHO PREHLIADAČA

BAKALÁRSKA PRÁCA

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra Informatiky
Vedúci bakalárskej práce:	RNDr. Michal Rjaško, PhD.

Bratislava, 2013
Bruno Cuc

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bruno Cuc
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Systém na distribuovanie výpočtov pomocou internetového prehliadača

Cieľ: Navrhnúť systém, ktorý bude slúžiť na distribuovanie výpočtovov k veľkému množstvu klientov, za účelom rátania nejakého ťažkého problému (hľadanie kolízii v hašovacej funkcii hrubou silou, a pod.). Program, ktorý bude vykonávať výpočet u klienta bude bežať v internetovom prehliadači a bude napísaný v JavaScripte. Pomocou tohto systému budu môcť prevádzkovatelia internetových stránok distribuovať namiesto reklamy program, ktorý využije výpočtovú silu klientského počítača.

Vedúci: RNDr. Michal Rjaško, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 25.10.2012

Dátum schválenia: 26.10.2012

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poď akovanie

Chcel by som sa poďakovať môjmu vedúcemu RNDr. Michalovi Rjaškovi, PhD. za cenné rady a konzultácie. Zároveň by som sa chcel poďakovať mojim rodičom za podporu počas štúdia.

Abstrakt

V práci sme navrhli systém, ktorý slúži na distribuovanie výpočtov veľkému množstvu klientov, za účelom výpočtu nejakého ťažkého problému (hľadanie kolízii v hašhovacej funkcií hrubou silou a pod.). Program, ktorý bude vykonávať výpočet u klienta beží v internetovom prehliadači a je napísaný v JavaScripte. Pomocou tohto systému budú môcť prevádzkovatelia internetových stránok distribuovať namiesto reklamy program, ktorý využíva výpočtovú silu klientskeho počítača.

KĽÚČOVÉ SLOVÁ: javascript, volunteer computing

Abstract

We designed system enabling distribution of computations towards large amount of clients to solve hard problems (hash function collision, etc.). Computation on client side is executed in web browser environment using JavaScript. This enables third-party website providers to distribute small application utilizing client resources insted of advertisement.

KEYWORDS: javascript, volunteer computing

OBSAH

Úvod

Tempo rastu výpočtovej kapacity počítačov aj v poslednej dekáde kopírovalo Moorov zákon. Vývoj v tejto oblasti pochopiteľne ovplyvnil aj osobné počítače. Kým v minulosti častokrát predstavoval procesor najpomalší článok systému, v súčasnosti sa do cenovo dostupných osobných počítačov integrujú procesory prevyšujúce nároky bežného užívateľa, ktoré rastú pomalšie ako výpočtový výkon. Tento rozdiel medzi maximálnym a využívaným výkonom pri súčasnom množstve počítačov predstavuje zaujímavý prostriedok pre koordinované využitie.

Na obdobnej myšlienke sú založené viaceré volunteer computing výpočty, ktoré využívajú nepotrebný výkon osobných počítačov na riešenie rôznych ťažkých úloh. Nevýhodou týchto systémov je potreba vynaloženia pomerne veľkého úsilia na zapojenie sa a nutnosť získať dôveru užívateľa, keďže pri konvenčnej architektúre sú schopné spúštania kódu na strojovej úrovni. Väčšina riešení problémov ale nutne nepotrebuje túto voľnosť a algoritmus na ich riešenie sa dá ekvivalentne prepísať do jazyka, ktorý má obmedzenú nízkoúrovňovú funkcionalitu a beží v obmedzenom bezpečnostnom kontexte, ako napríklad javascript.

Práve javascript má veľkú výhodu v rozšírenosti - program v ňom napísaný je možné, v porovnaní so štandardnými volunteer computing systémami spustiť oveľa jednoduchšie, napríklad nasmerovaním internetového prehliadača na vhodnú adresu alebo len načítaním kódu z už bežne navštevovanej stránky. Keďže náročnosť spustenia je nepriamo úmerná s rozšírenosťou, vhodné skombinovanie využívania nepotrebného výkonu a jednoduchosti spustenia by mohlo rozšíriť výpočet na oveľa väčšie množstvo počítačov.

Vývoj jazyka javascript a jeho populárnych interpreterov počas posledných rokov pridal funkcionalitu kritickú pre takúto konštrukciu a zároveň rôznymi optimalizačnými metódami ako predkompilovanie znížil mieru neefektívnosti javascriptu v porovnaní s kompilovanými jazykmi, vyplývajúcu z jeho interpretovanosti.

Štruktúra práce

V prvej kapitole odhadujeme výpočtovú silu v súčasnosti aktívnych počítačov. Ďalej uvádzame stručný prehľad súčasných riešení disponujúcich vysokým výkonom a ich nedostatkov, ktoré sú motiváciou pre túto prácu.

V druhej kapitole uvádzame špecifikáciu plynúcu z odôvodnených požiadaviek na konštrukciu nového vysokovýkonného systému.

Náš prínos

V práci sme v tretej kapitole navrhli originálnu architektúru, ktorá umožňuje distribúciu výpočtovo náročných programov výpočtovým klientom pomocou javascriptu. Výhodou našej konštrukcie je jej potenciálne široký dosah, keďže užívateľ nemusí vykonať žiadnu novú explicitnú akciu na to, aby sa do výpočtu zapojil (stačí, ak nasmeruje jeho prehliadač na nejakú doménu odkazujúcu náš distribujúci systém - môže to byť napríklad webové rozhranie elektronickej pošty, blog, či ľubovoľný iný portál, ktorý užívateľ bežne navštevuje).

Javascript, ako interpretovaný webový jazyk nie je dimenzovaný na naše špecifické využitie a preto vytvárame niekoľko netriviálnych konštrukcií umožňujúcich a optimalizujúcich vykonávanie výpočtov ako napríklad vyťaženie všetkých jadier či ukladanie zmrazených výpočtov.

Ďalej ponúkame riešenie problematiky bezpečnosti a transparentnosti výpočtov. Navrhli sme systém odhadovania aktuálnej záťaže klientskeho počítača a na základe toho schopnosť prispôsobovať náš podiel výpočtovej kapacity - pre užívateľa transparentný výpočet má väčšiu pravdepodobnosť byť dlhodobo tolerovaný. Z hľadiska bezpečnosti sme zabezpečili nemožnosť programu dodaného nevieryhodným zadávateľom prevziať kontrolu nad tokom vykonávaného skriptu či neoprávnený prístup k dátam.

Zostrojili sme demonštračné programy v javascripte pre niektoré kritické úseky s ktorými v architektúre konštrukcie predpokladáme.

Keďže klientov (užívateľov pripojených na internet) nepovažujeme za dôveryhodných, v štvrtej kapitole sme spomenuli niekoľko a navrhli jeden systém verifikácie prijatých výsledkov. Ďalšie spôsoby riešenia tohoto problému môžu byť inšpirované súčasnými konvenčnými riešeniami.

V poslednej kapitole spomíname etickú stránku charakteru takýchto výpočtov.

Kapitola 1

Úvod do problematiky

1.1 Prehľad súčasných riešení

Výpočtovo náročné problémy sa v súčasnosti riešia prevažne superpočítačmi alebo volunteer computing sieťami. Superpočítače majú oproti volunteer computing výhodu v spoľahlivosti a súkromia (ani čiastočné dáta nie sú zdieľané) a výkonu (elastickejšia manipulácia s dátami). Cena za jednotku výkonu (napríklad TFLOPS*rok) je oveľa nižšia pri volunteer computing (približne $\frac{1}{100}$) keďže náklady nezahŕňajú skutočnú cenu hardvéru. Ďalšou výhodou volunteer computing je postupne zlepšujúci sa hardvér - súčasne s tým, ako si bežní užívatelia zaobstarávajú výkonnejšie počítače, rastie s nimi aj výkon volunteer computing sietí, v ktorých sú zapojené. Problémom volunteer computing je malá popularita vyplývajúca z náročnosti zapojenia sa. V súčasnosti má len 12 projektov viac ako 10000 klientov a len 3 projekty viac ako 100000 klientov. Toto číslo je veľmi malé oproti množstvu všetkých počítačov pripojených na internet.

1.1.1 Riešenia mieriace na vysoké množstvo klientov

Plura processing

Jedná sa o komerčné-orientované riešenie, ktoré poskytuje zadávateľovi istú výpočtovú silu, ktorú Plura získava tak, že stránky zapojené do Plura umiestnia do svojho kódu java applet, ktorého účel je obdobný ako nášho javascriptového klienta, teda stiahnuť dáta a previesť na nich nejakú výpočtovo náročnú operáciu. Podľa množstva výpočtov, ktoré návštevníci ktorej-tej stránky odvedú získa stránka finančnú odmenu. Keďže sa jedná o java applet, samotný prehliadač nestačí na intepretovanie java bajtkódu a počítač potrebuje java virtual machine. Okrem toho, java applety si takmer vždy vyžadujú manuálnu konfirmáciu spustenia, čo znižuje transparentnosť a zvyšuje potrebu na interakciu užívateľa (ktorú je esenciálne minimalizovať za účelom rozšírenia). Java applet má taktiež väčšie právomoci

(prístup k súborovému systému a pod.). Využitie hardvérovej akcelerácie v java appletoch je podmienené podpísaným appletom. V našom systéme spúšťame cudzie kódy - napriek tomu, že aj pri našom riešení kód pred spustením algoritmicky preverujeme (čo by sme mohli robiť aj v prípade javy), v prípade zlyhania preverovania by distribúcia škodlivého kódu so zvýšenými právomocami mala oveľa horšie následky, keďže útočník by prakticky mohol spustiť ľubovoľný kód s právomocami bežného užívateľa u príslušného klienta. Kým popularita javascriptu rastie, popularita javy pre použitie na webstránkach klesá. V súčasnosti napríklad OS X v štandardnej konfigurácií pri nainštalovanej jave internetové applety nespúšťa bez explicitnej zmeny nastavení. Okrem zvýšenia výkonnosti interpreterov, vývoj javascriptu v posledných rokoch pridal niekoľko konštrukcií, ktoré výrazne zvýšili množstvo programov, ktoré sa dajú napísať v javascripte. Priamy dôsledok je možnosť nahradiť časť aplikácií, na ktoré boli potrebné java applety ekvivalentnou konštrukciou v javascripte (canvas, webworkers).

Existujúce riešenia v JavaScripte

Autorom tejto práce nie je známe žiadne riešenie v javascripte, ktoré by komplexne riešilo uvedený problém.

Po vytvorení konštrukcie a napísaní základu práce bol v januári 2013 publikovaný článok *Distributed Computing on an Ensemble of Browsers* [3] . Konštrukcia z článku je pomerne všeobecná a z časti sa teda podobá na tú z našej práce. Naša konštrukcia však okrem návrhu a analýzy špecifickej distribuovanej siete poukazuje a navrhuje riešenia niekoľkých ďalších problémov zvyšujúcich rozšíriteľnosť, výkonnosť, transparentnosť a bezpečnosť spúštania nevieryhodného kódu, s ktorými sa konštrukcia z článku nezaoberá.

1.2 Odhad výpočtového výkonu

V tejto časti sa pokúsime odhadnúť množstvo výpočtového výkonu, ktorým bude náš systém schopný disponovať.

Napriek tomu, že nie všetky programy ktoré budú na našej konštrukcií spúšťané budú nutne používať veľké množstvo operácií s desatinnou čiarkou (FLOP), tento spôsob merania výkonu použijeme, pretože:

- Vo všeobecnosti pri procesoroch (na rozdiel od grafických kariet) platí, že množstvo FLOPS koreluje so všeobecným výkonom a teda nameraná hodnota nám dokáže dostatočne relevantne charakterizovať výpočtovú silu procesora.
- Je často používaná jednotka - výkon superpočítačov ako aj volunteer computing sietí je najčastejšie udávaný v množstve FLOPS, ktorými disponujú.

- Aby náš odhad bol smerodajný, mal by byť porovnateľný so súčasne dostupnými prostriedkami, preto sa pokúsime odhadovať jeho silu v tejto miere.

1.2.1 Štruktúra odhadu

Keďže výkon, ktorý je k dispozícií záleží na veľmi veľkom množstve faktorov, budeme nútení niektoré faktory zovšeobecniť a pokúsime sa o konzervatívny (”najhorší”) odhad. Ak neskôr zistíme, že aj hodnota konzervatívneho odhadu je zaujímavá a reálny výkon je pravdepodobne väčší, má zmysel zostrojiť takýto systém. Odhadneme počet v súčasnosti *aktívnych* počítačov, priemernú dĺžku života počítača a súčasný výkon *nových* počítačov. Na základe tohoto vyjadríme celkový inštalovaný výkon. Ďalej skúsime odhadnúť využiteľnú časť inštalovanej sily.

1.2.2 Množstvo počítačov

Priemerná životnosť počítača

Priemerná životnosť počítača nás bude zaujímať z nasledujúceho dôvodu: Nech máme informáciu o počte počítačov N a priemernej životnosti L rokov a výkone nových počítačov V. Z tohoto vieme aproximovať, že približne $\frac{N}{L}$ počítačov je zo súčasného roku, rovnaké množstvo je staré $k = 0,1,...,L-1$ rokov. Z Moorovho zákona potom vieme určiť výkon všetkých počítačov V_{all} ako súčet geometrického radu:

$$V_{all} = \sum_{i=0}^{L-1} \left(\frac{1}{2}\right)^i \frac{N}{L} V \tag{1.2.1}$$

Množstvo osobných počítačov (stolné, prenosné) presiahlo miliardu v roku 2008, podľa predpovedí dosiahne dve miliardy v roku 2014 [1]. Na základe množstva vyrobených počítačov - za posledných niekoľko rokov približne 350 miliónov za rok[2] je priemerný ročný rast 160 miliónov počítačov, a teda približne 190 miliónov počítačov sa prestane používať. Náš odhad sa približne zhoduje s odhadom (Gartner), ktorý odhadol číslo 180 miliónov (2008). Vzhľadom na počet všetkých (miliarda v 2008) je to približne $1/5$ a teda životnosť asi 5 rokov.

Množstvo počítačov

Pri priemernom prírastku 160 miliónov nových počítačov ročne (interval 2008 - 2014) odhadneme množstvo aktívnych počítačov v 2013 na 1.85 miliardy.

[1] Reuters, http://www.reuters.com/article/2008/06/23/us-computers-statistics-idUSL2324525420080623

[2] http://www.worldometers.info/computers/

Priemerný výkon nových počítačov

Vo väčšine novodobých počítačov je výpočtový výkon charakterizovaný parametrami CPU (frekvencia, počet jadier, implementované inštrukcie, pamäť, a podobne). Keďže na základe týchto informácií nevieme dostatočne presne odhadnúť výkon vo FLOPS, množstvo FLOPS určíme pomocou passmark skóre. Ukážeme na koreláciu medzi passmark skóre a FLOPS, na základe nej potom odhadneme množstvo FLOPS pre bežné CPU.

Passmark skóre pre CPU

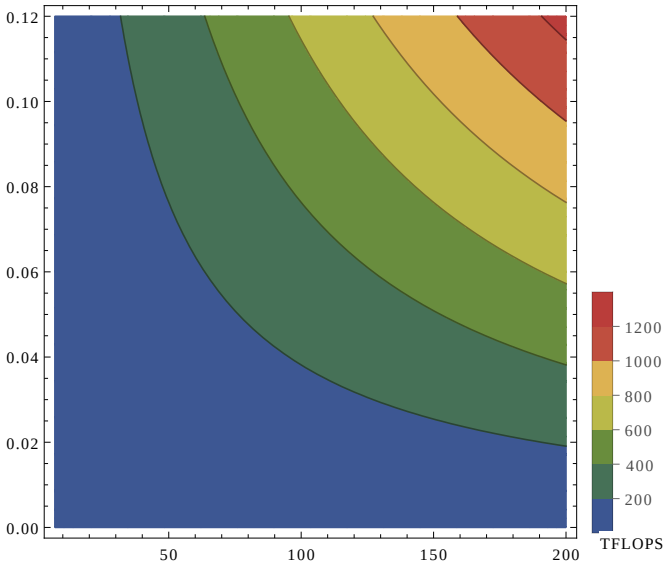
Passmark skóre je výsledok nejakej ohodnocovacej funkcie (benchmark) výkonu jednotlivých častí počítača. Nás bude zaujímať výsledná passmark hodnota pre CPU. Skóre je počítané programom, ktorý si užívatelia spustia. Program vyťaží systém na maximum a na základe množstva operácií, ktoré je schopný daný CPU za jednotku času vykonať, program priradí k CPU jeho skóre. Skóre pre model procesora je potom priemer všetkých skór vypočítaných na počítačoch obsahujúcich daný model CPU. Vzhľadom k vysokej korelácií medzi množstvom FLOPS a passmark skóre sme určili závislosť lineárnou regresiou. Získali sme tak približný pomer $150\frac{passmark}{GFLOP}$.

Charakteristika súčasných procesorov

Pokúsime sa konzervatívne určiť model rozšírených procesorov súčasnosti. Keďže v odhade množstva počítačov sú zaradené rôzne nízkovýkonné počítače určené pre rozvíjajúce sa trhy a edukačné projekty, radšej ako znížiť odhad priemerného CPU distribuovaného s novým počítačom, znížime množstvo počítačov o nejakú hodnotu a výsledné číslo budeme považovať za štandardné komerčné počítače. Veľkú časť predaných procesorov tvoria notebooky, pri ktorých je špecifické, že častokrát sú distribuované s CPU z podobnej triedy ako stolné počítače a označením mobile. Pochopiteľne, majú nižší výkon - keďže radšej vymeníme tesný odhad za konzervatívny, budeme uvažovať za charakteristický model CPU pre súčasné počítače (stolové aj prenosné) najnižší model moderného mobile procesora. Zvolíme teda i3-mobile so skóre 2730, čo predstavuje približne 18 GFLOPS.

Túto hodnotu dosadíme do 1.2.1. Ďalej zohľadníme:

- môžeme využiť len časť prostriedkov (náš systém nemôže plne vyťažovať CPU) ($k_1 = 0.85$)
- javascript je ako interpretovaný jazyk pomalší ako kompilované jazyky, ktoré by počítanie toho istého problému zvládli rýchlejšie (predpokladáme spomalenie 10-krát) ($k_2 = 0.1$)
- počítač nebeží stále (uvažujeme 8 hodín za deň) ($k_3 = 1/3$)



Obrázok 1.1: Výkon (V) systému v závislosti od priemerného výkonu v GFLOP (horizontálna os) a k_4 reprezentujúci podiel počítačov, ktoré systém využije (vertikálna os)

.

- množstvo počítačov, ktoré systém využije je 0.5% všetkých osobných počítačov (ekvivalentne - množstvo užívateľov, ktorí navštívia ľubovoľnú zo stránok, ktoré distribuujú náš systém) ($k_4 = 0.005$)

Potom priemerný výkon takéhoto systému bude $V_{sys} = V_{all} \prod_{i=1}^{4} k_i \doteq 1.7 PFLOPS$.

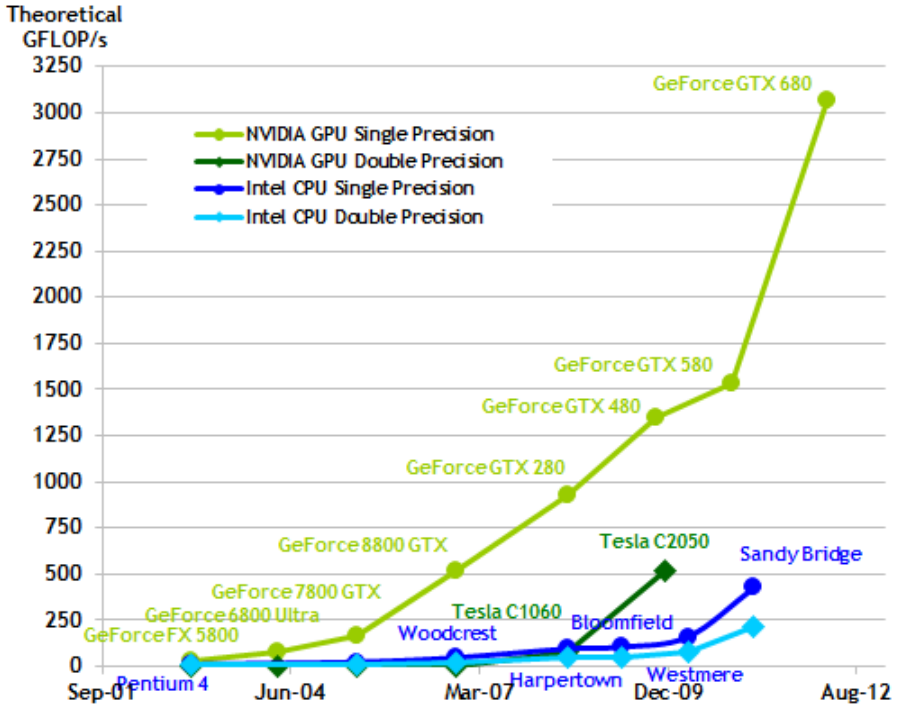
Keďže odhad je hrubý, väčšiu výpovednú hodnotu má rád - 10^{15}. Uvedená hodnota je zrejme citlivá prevažne od dvoch parametrov - výkon jedného počítača a množstvo počítačov. V obrázku 1.1 uvádzame výpočtovú silu konštrukcie vzhľadom na zmenené parametre.

Pre demonštráciu sme použili zámerne veľké hodnoty V_{all}, ktorých tempo rastu určuje rastúci výkon dostupných paralelných grafických kariet s rádovo menšou cenou za GFLOP ako pri štandardných x86 resp. x86_64 procesoroch, ktorých natívna podpora pre webové aplikácie existuje v experimentálnom štádiu vývoja s funkčnými demonštráciami[3]. Čísla v obr. 1.1 sú odvodené pre $k_2 = 0.1$, čo je malá hodnota v prípade použitia existujúceho API využívajúceho potenciál paralelných GPU.

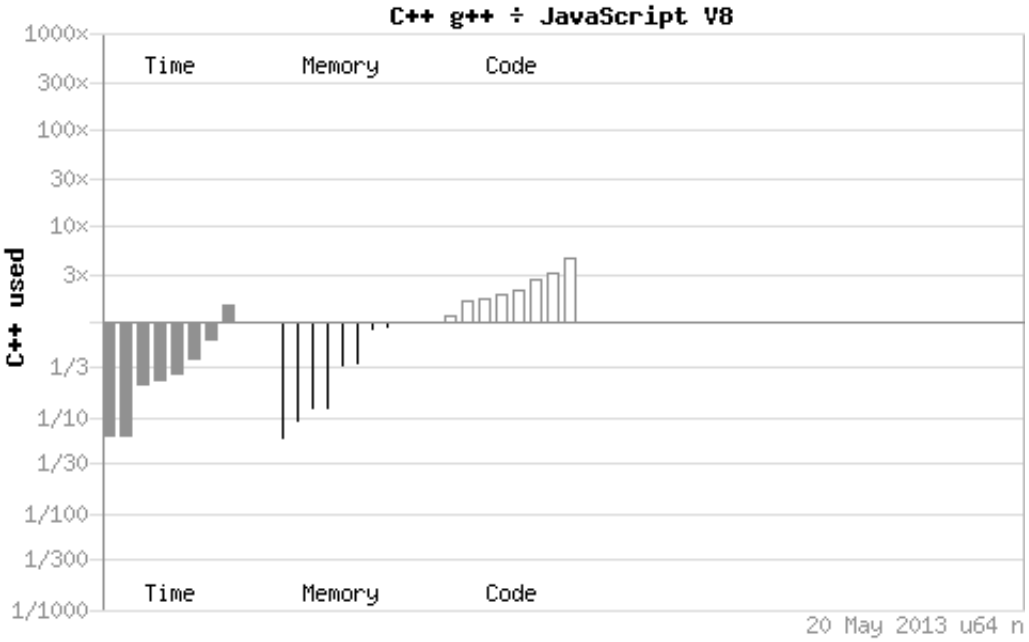
1.2.3 Porovnanie s dostupnými prostriedkami

Porovnáme výsledok so súčasnými systémami, ktoré sú v prevádzke:

[3] http://webcl.nokiaresearch.com/



Obrázok 1.2: Porovnanie vývoja výkonu CPU a GPU (dual.sphysics.org)



Obrázok 1.3: Porovnanie rýchlosti rôznych ekvivalentných programov napísaných v c++ (g++) a javascripte (V8). Každý program je reprezentovaný jedným stĺpcom v každej z troch kategórií. Z obrázka je vidno, že väčšina z testovaných programov je približne 3-krát taká pomalá v javascripte oproti ich c++ ekvivalentu. Detaily testovaných programov sa nachádzajú v [1]

Superpočítače

TOP500[4] udržiava zoznam najvýkonnejších superpočítačov. V súčasnosti 5 najvýkonnejších je uvedených v tabuľke 1.1.

Z tabuľky vidno, že výkon najrýchlejších superpočítačov má podobný rád ako náš výsledok. Ak by sme sa pozreli na najrýchlejší počítač postavený na x86 / x86_64 architektúre, vynechali by sme prvých 5 superpočítačov a najvýkonnejším by sa stal SuperMUC s výkonom 2.8 PFLOPS (6. miesto v TOP500)).

Názov	Inštitúcia	Výkon (PFLOPS)
Titan	Oak Ridge National Laboratory (USA)	17.5
Sequoia	Lawrence Livermore National Laboratory (USA)	16.3
K coputer	RIKEN Institute (Japonsko)	10.5
Mira	Argonne National Laboratory (USA)	8.1
JUQUEEN	Forschungszentrum Juelich (Nemecko)	4.1
SuperMUC	Leibniz Rechenzentrum (Nemecko)	2.8

Tabuľka 1.1: Najvýkonnejšie superpočítače (November 2012)

Volunteer computing

Najvýkonnejšou volunteer computing sieťou je v súčasnosti Folding@Home s výkonom 2.6 PFLOPS (november 2012). Súčet výkonov všetkých sietí pod volunteer computing platformou BOINC je 7.3 PFLOPS. Žiadna ďalšia verejná sieť nepresahuje hranicu 1 PFLOPS. Hranicu 100 TFLOPS stabilne prekračuje menej ako 10 projektov.

Výhodou nami popisovaného systému je podobne ako pri volunteer computing sieťach - automaticky zvyšujúci sa výkon konštrukcie tempom približným, ako výkon osobných počítačov (približne zdvojnásobenie výkonu každý rok). Taktiež, náklady na opravu poškodeného hardvéru sú nulové.

[4] http://www.top500.com

Kapitola 2

Špecifikácia

V tejto kapitole sa budeme zaoberať nárokmi a obmedzeniami systému. Keďže konštrukcia nášho systému je pomerne netradičná a použité technológie neboli primárne dizajnované na nami navrhnuté použitie, budeme nútení použiť niekoľko nekonvenčných konštrukcií.

2.1 Požiadavky na strane prevádzkovateľa

Prevádzkovateľ je subjekt, ktorý dostane zadanie od zadávateľa a rozdistribuuje ho klientom.

- **Minimalizácia podpornej infraštruktúry** - od systému žiadame, aby prevádzkovateľ nemusel vykonávať náročné výpočty
- **Maximalizácia výkonu** - systém by mal byť navrhnutý tak, aby využil čo najväčšiu možnú časť prostriedkov čo najväčšieho množstva osobných počítačov.
- **Verifikácia** - znemožniť, aby klient dokázal úspešne podhodiť ako odpoveď iný výsledok, ako ten, ktorý pre daný program a vstup očakávame. Alternatívne ohraničiť a vyjadriť pravdepodobnosť takejto udalosti.
- **Algoritmické zabezpečenie bezpečnosti** - vyriešiť nižšie uvedenú otázku bezpečnosti algoritmicky, aby nebolo potrebné programy overovať manuálne.

2.2 Požiadavky na strane klienta

Klient je užívateľ navštevujúci nejakú stránku, ktorá distribuuje daný výpočet. Vo všeobecnosti disponuje počítačom s priemerným výkonom.

- **Transparentné spustenie** - spustenie nebude potrebovať žiadnu interakciu na strane klienta.

- **Neviditeľnosť** - výpočet by mal využívať zdroje (výpočtové, pamäťové, sieťové) ”rozumne” - teda spôsobom, ktorý bude čo najmenej negatívne ovplyvňovať užívateľovu prácu s počítačom (súčasne s maximalizáciou výkonu)

- **Bezpečnosť** - pre zadaný program zabezpečiť, aby jeho spustenie nenarušilo súkromie klienta [1], aby bolo vykonané v súlade s hore uvedenou požiadavkou neviditeľnosti (systém je navrhnutý na distribúciu výpočtovo náročných programov, ale program od zadávateľa nebudeme považovať za dôveryhodný).

2.3 Požiadavky na strane zadávateľa

Zadávateľ je niekto, kto potrebuje využiť veľký výpočtový výkon.

- **Jednoduchá konštrukcia programu** - chceme dosiahnuť, aby modifikácia originálneho pseudokódu zadávateľom potrebná na to aby bol spustiteľný v našom systéme bola čo najmenšia

[1] Zadávateľ by mohol napríklad podhodiť program, ktorý by pristupoval k DOM štruktúre dokumentu, takto by mohol pristupovať na obsah stránky distribujúcej náš systém - napr. prístup k e-mailom alebo XSS útok

Kapitola 3

Konštrukcia

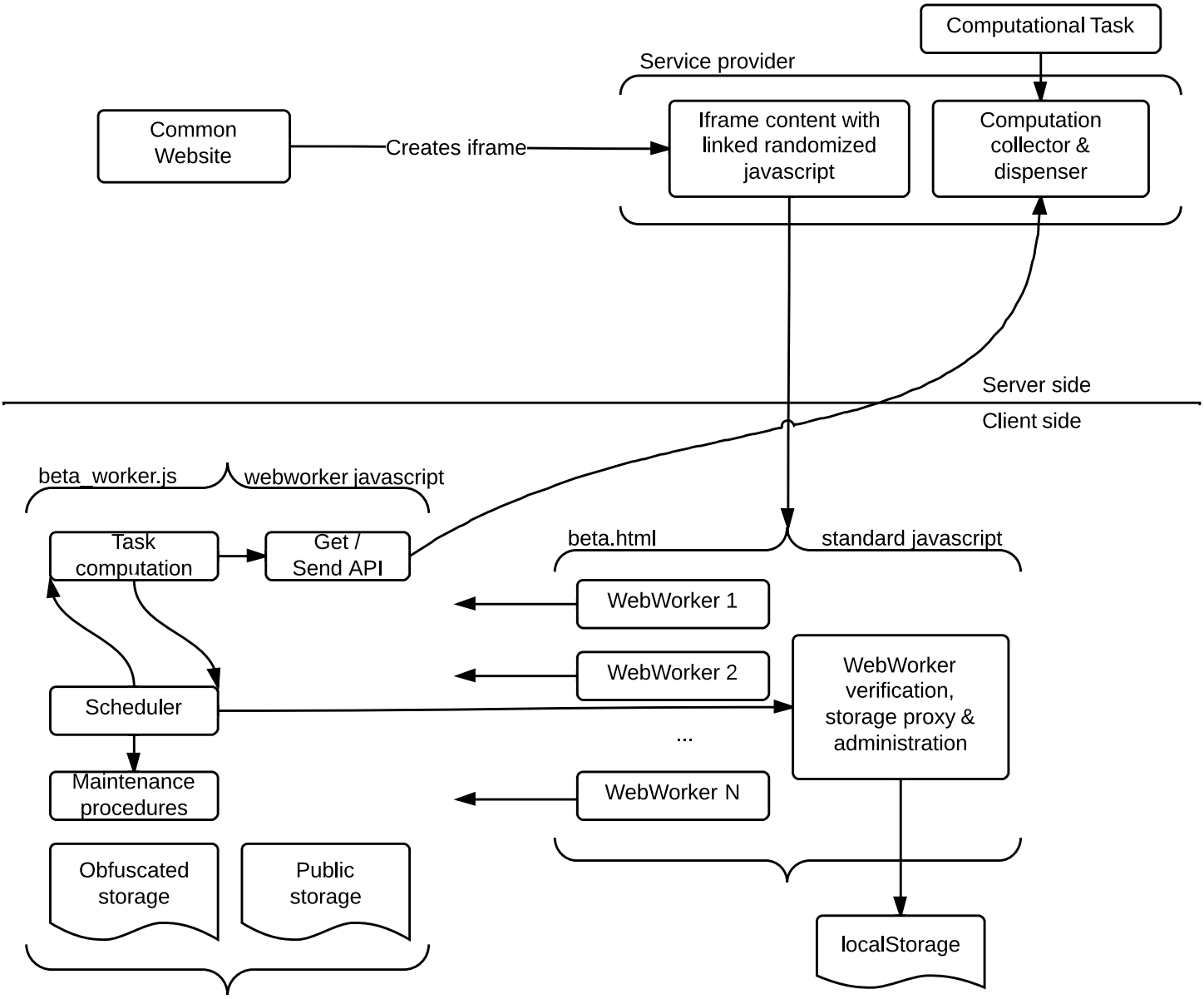
3.1 Základný popis

V tejto časti uvedieme základný popis nášho systému, ktorý je znázornený v obr. 3.1.

Po tom ako rozhodneme, že zadávateľov program spĺňa naše bezpečnostné požiadavky, začneme ho distribuovať. To znamená, že bežné stránky (postavené na ľubovoľnej architektúre s ľubovoľným účelom) odkazujúce náš javascriptový kód budú vytvárať `iframe`, ktorého cieľ bude stránka spúšťajúca výpočet. Tá po načítaní skontroluje, či na užívateľovom počítači bola niekedy otvorená. Ak nie, vykoná inicializačné operácie (odhad množstva využiteľných vlákien, vytvorenie záznamu v javascriptovom `localStorage`, prípadne generovanie unikátnej identity). Následne spustí príslušné množstvo inštancií webworkra, ktorý poskytuje základnú štruktúru pre výpočty a prepne sa do udržiavaceho módu verifikácie, javascriptového `localStorage` proxy (keďže webworker nemá prístup k tomuto objektu) a zmení procedúru zatvorenia stránky (toto potrebujeme, keďže si chceme výpočet pred opustením uložiť, čo trvá nejaký čas).

Verifikačný mód spočíva v tom, že náš systém riadenia toku vykonávania vo webworkri môže stratiť kontrolu nad tokom programu - napríklad ak do programovej funkcie zadávateľ vloží nekonečný cyklus. Toto nevieme odhaliť a zároveň takúto funkciu nevieme ukončiť zvnútra webworkra. Preto pomocou zabezpečenej funkcie každý webworker ohlasuje, že náš systém riadenia toku programu vrámci daného webworkra je stále aktívny. Toto je vykonávané pomocou javascriptového API `postMessage`, čo je spôsob prenášania informácií medzi rôznymi vláknami a hlavným (iframe) vláknom javascriptu. Na základe tohoto ohlasovania, ktoré je nastavené na pevný interval potom vieme, či je potrebné nejakého webworkra predčasne ukončiť.

V prípade že kontrolný cyklus vnútri webworkra kontrolu nestratí, sleduje čas spotrebovaný vykonávaním zadávateľovho programu a adekvátne uspáva vykonávanie (toto je potrebné z dôvodu transparentnosti), odhaduje zaťaženia daného vlákna (inými programami,



Obrázok 3.1: Základný popis systému

ktoré klient spúšťa) a vykonáva ostatné servisné procedúry.

Zadávateľov program potrebuje získať vstupné a odoslať výstupné dáta - *XMLHttpRequest* je dostupný vo webworkroch - okrem toho, jedná sa o načítanie v asynchrónnom móde a teda z dôvodu (neblokujúceho) načítania dát nie je potrebné spustiť ďalšie vlákno - ak zadávateľov program zistí, že mu "dochádzajú" dáta, jednoducho zavolá API funkciu, ktorá neblokujúco doplní ďalšie, čo zabezpečí neprerušené využitie dostupného výpočtového výkonu.

Pri opustení stránky, ktorá odkazuje náš systém a následnom uložení výpočtu je vo výpočte možné pokračovať akonáhle návštevník navštívi nejakú (aj inú) stránku odkazujúcu náš systém.

3.2 Odhad množstva jadier

Ak spustíme výpočet v jednom vlákne (jeden webworker), nie je možné plne využiť potenciál súčasných počítačov, ktoré sú bežne vybavené 2 a viac jadrami. Preto potrebujeme nájsť spôsob, ako efektívne využiť všetky jadrá.

Riešenie používajúce fixné množstvo vlákien

Pomerne neelegantný prístup riešenia problému je zobrať si dostatočné arbitrárne množstvo j webworkrov (napríklad $j = 8$) a aplikáciu na každom z nich spustiť. Pokiaľ bude aplikácia bežať na viacjadrovom počítači, určite využijeme všetky jadrá (až do j).

Nevýhoda tohoto prístupu je pri počítačoch s menším počtom jadier. Pri väčšom množstve webworkrov ako jadier budeme zdieľať niekoľkých webworkrov na jednom jadre. Vo všeobecnosti by mohol byť takýto rozdiel veľký čo by znamenalo, že na samotnú réžiu, ktorá nám neprináša žiaden úžitok by sme vyčlenili zbytočne väčšie systémové prostriedky. Keďže rôznorodosť množstva jadier v počítačoch v budúcnosti klesať pravdepodobne nebude, navrhneme iné riešenie, ako interaktívne odhadnuť počet jadier.

Riešenie prispôsobujúce množstvo vlákien

Napriek tomu, že internetový prehliadač prezrádza časť konfigurácie systému počítača (napríklad rozlíšenie displeja, operačný systém a podobne), neuvádza počet jadier. Keďže moderné interpretery javascriptu rozdeľujú viacero paralelne spustených webworkrov do viacerých jadier (ak je ich toľko k dispozícií), počet jadier vieme odhadnuť nasledovným spôsobom

V nekonečnom cykle budeme vykonávať funkciu zloženú z niekoľkých príkazov, ktoré budú vyťažovať výpočtové zdroje počítača čo najviac (napríklad aritmetické inštrukcie, log-

ické inštrukcie, práca s reťazcami a podobne). Po uplynutí časového intervalu `time_slice` vykonávanie zastavíme a pozrieme sa na počet zavolaní funkcie (označíme P_1) počas daného intervalu.

P_1 teda reprezentuje približný výkon jedného jadra. Následne budeme spúšťať paralelne k inštancií funkcie (pre vhodné k). Pri každom takomto paralelnom spustení sa pozrieme na súčet množstva zavolaní jednotlivých funkcií. Pokiaľ toto číslo bude približne kP_1, znamená to, že vieme efektívne využiť k paralelných inštancií

Počet efektívne využiteľných jadier procesora teda bude najväčšie k, pri ktorom ešte bude platiť horeuvedená podmienka.

Menením hodnoty `time_slice` môžeme prispôsobiť rýchlosť fázy odhadovania množstva efektívne využiteľných vlákien. Z abstraktného hľadiska by mohla byť hodnota takmer ľubovoľne malá (zmenou by sa zmenšilo množstvo operácií lineárne), v reálnych systémoch s množstvom ďalších bežiacich procesov a rôznymi fluktuáciami by však malá hodnota `time_slice` mohla skresliť reálny výkon - čím by bola menšia, tým viac by aj menej signifikantná fluktuácia alebo nekonzistentnosť systému ovplyvnila presnosť výpočtu.

Keďže množstvo vlákien v súčasných procesoroch je takmer vždy násobok 2, prechádzame len tieto hodnoty. Napriek tomu, že z prvého prezretia algoritmus čiastočne pripomína myšlienku z binárneho vyhľadávania, vzhľadom k tomu, že prehľadávame na množine s prvkami výlučne z $\{2^i; i \in \mathbb{N}\}$ je toto štandardné prehľadávanie. Vzhľadom k tomu, že v súčasnosti sa najbežnejšie stretneme s procesormi schopnými konkurentne spracuvávať 2, 4 alebo 8 vlákien, do finálnej implementácie zakomponujeme toto riešenie.

Riešenie pre veľajadrové počítače

Univerzálne riešenie efektívne pre ľubovoľné množstvo jadier: Nech $M = 2^m$ je maximálne možné množstvo jadier, $r \leftarrow m$ a $l \leftarrow 1$. Zmeriame pre $k = 2^{r-l/2}$. V prípade, že horeuvedená podmienka bude platiť, potom $l \leftarrow {}^{r-l}/_2$. V opačnom prípade $r \leftarrow {}^{r-l}/_2$. Zopakujeme pre nové hodnoty r, l. Pokiaľ $r - l < 2$, skončíme. Ekvivalentne formulované, binárne prehľadávame exponent dvojky.

3.3 Schéma

V nasledujúcej časti popíšeme schému, na základe ktorej funguje naša konštrukcia.

Pokračovanie výpočtu pri opustení stránky

Napriek tomu, že naša konštrukcia je vhodná pre stránky, ktoré zvyknú bežať dlhodobo v pozadí (napríklad webové rozhranie elektronickej pošty, sociálne siete a pod.), bolo by výhodné, keby sa riešenie dalo implementovať aj na stránkach, ktoré nezvyknú bežať v pozadí a teda ich URL adresa sa mení častejšie. Triviálne riešenie tohoto problému by bolo zníženie veľkosti atomickej úlohy tak, aby sa stihla vykonať medzi dvoma zmenami URL adresy. Takýto prístup však má viacero nedostatkov:

- Potenciálne signifikantné množstvo strateného procesorového času, keďže výpočet pre nejaký vstup môže byť takmer ukončený a stránka je opustená pred ukončením výpočtu
- Veľká diverzita časov medzi dvoma zmenami URL - na to, aby sa efektívne využili aj krátke návštevy by boli potrebné dalšie obmedzenia vzťahujúce sa na zadaný program. V prípade, že by sa krátke návštevy nevyužívali vôbec (program by skončil až pri dlhších návštevách), pridala by sa ďalšia strata procesorového času

Uvedieme dve možné riešenia, z ktorých jedno je implementovateľné priamo našou konštrukciou a toto aj naprogramujeme

Neopustenie URL adresy - spočíva v tom, že prevádzkovatelia stránok budú vo väčšom množstve využívať *AJAX*, čo bude mať za následok aj kontinuálnejšie vykonávanie nášho javascript kódu. Ďalšia výhoda, ktorú toto riešenie prináša je zníženie overheadu, ktoré v súčasnosti majú stránky, ktorých jednotlivé podstránky majú rovnakú štruktúru, ale iný obsah (napríklad blogy, stránky na wiki jadre a podobne)

Ukladanie stavu výpočtu pred opustením stránky - v štandarde HTML 5 boli pridané štruktúry *sessionStorage* a *localStorage*, ktoré javascript môže použiť ako asociatívne pole. *sessionStorage* expiruje po ukončení relácie, v prípade *localStorage* neexpiruje vôbec, ale jeho veľkosť je obmedzená, väčšinou na 5MB.

Vzhľadom na mechanizmus, ktorým vykonávame užívateľom zadaný program, stav výpočtu môže byť charakterizovaný dvojicou *(vrátená procedúra, programové dáta)*. Pri opustení stránky si nebudeme pamätať rôzne režijné meta-data. Výnimkou je množstvo efektívne využiteľných vlákien, čo nám umožní pri načítaní ďalšej stránky hneď pokračovať vo výpočte.

3.4 Transparentnosť a bezpečnosť systému

Potrebujeme zabezpečiť, aby vykonávaný program nezahltil celý výkon procesora. Keďže javascriptový webworker nemá implementovanú afinitu, naše riešenie bude pripomínať

round-robin plánovaciu schému. Z nedôvery k zadávateľovi a jednoduchosti implementácie zabezpečíme plánovanie výkonu v našom systéme.

Implementácia bude založená na funkcií *setTimeout*, keďže javascript nedisponuje pozastavením vykonávania webworkra a zároveň nemá štandardno-paradigmový sleep, ktorý oddiali vykonávanie príkazu za príkazom sleep o nejaký čas špecifikovaný ako parameter pre sleep bez aktívneho vyťaženia procesora

Z bezpečnostných dôvodov zakážeme volať *setTimeout* programovým procedúram, keďže na implementáciu round-robina potrebujeme mať absolútnu kontrolu nad tokom programu. Zakazovanie funkcií a ďalšie bezpečnostné mechanizmy vysvetlíme nižšie.

Výpis 3.1: Zjednodušený pseudokód réžie vykonávania zadávateľovho programu

```
proc = default
worked = 0
last_meta = now()

while True:
       proc, time_elapsed = execute (proc) # proc vracia proceduru pre dalsie
           spustenie
       worked += time_elapsed

       if (time_elapsed > CONST0):
              terminate

       if worked > CONST2:
       # na dalsie vykonavanie pockame, cim znizime priemerne zatazenie CPU
              delay(coef*worked)
              worked = 0

       if now()-last_meta > CONST1:
              report_alive ()
              estimate_round_robin()
```

3.4.1 Overenie zadaného programu

Potrebujeme spôsob, ako rozhodnúť o zadávateľovom programe, či spĺňa podmienky (bezpečnosť, rozdelenie programu do funkcií) . Jeden z prístupov riešenia tohoto problému je navrhnúť špeciálnu syntax (ako podmnožinu javascriptovej), ktorá bude príjmať len javascriptové programy, ktoré zároveň vyhovujú naším podmienkam z hľadiska bezpečnosti (neprijmať tie, ktoré pristupujú k nepovoleným premenným alebo funkciám) a riadenia toku programu.

Keďže toto riešenie by si vyžadovalo zložitejšiu syntaktickú analýzu, riešenie ktoré sme

sa rozhodli použiť bude založené na odlišnom princípe.

Súkromné premenné v javascripte

Na riadenie toku programu potrebujeme, aby riadiaci cyklus (funkcie, premenné) bol chránený proti zásahu programom zadávateľa. Toto môže byť problém, keďže program zadávateľa ”prilepíme” k riadiacemu programu (a teda nachádzajú sa v tom istom mennom priestore (namespace)) - toto požadujeme, keďže potrebujeme byť schopní kontrolovať tok programu. V javascripte neexistujú súkromné premenné tried v štandardnom zmysle ako v iných objektovo orientovaných jazykoch (napr. Java alebo C++). Preto navrhneme vlastné riešenie.

Javascript, podobne ako ďalšie interpretované jazyky umožňuje enumerovať všetky premenné. Ak by sme si uchovali riadiace informácie v premennej s tajným názvom, musíme zohľadniť nasledujúce:

- Názov tajnej premennej sa musí meniť (v opačnom prípade je možný triviálny útok: útočník pošle program, zapojí sa do výpočtu, zo zdrojového kódu vygenerovaného javascriptu zistí názvy riadiacich premenných a v ďalšom programe ich už vie modifikovať)
- Znemožniť enumeráciu všetkých premenných

V javascripte je možné enumerovať všetky premenné v danom mennom priestore napríklad pomocou objektu `window`. Ten obsahuje aj množstvo enviromentálnych premenných, prípadne vlastností okna. Ak ich označíme ako `env`, potom práve užitočné informácie vieme získať:

```
for (index in window){
       if index not in env:
              console.log (index,typeof window[index],window[index])
}
```

Podobne ako ďalších niekoľko objektov, `window` nie je k dispozícií v mennom priestore webworkrov) a tak pomocou neho nie je možné enumerovať všetky premenné. Zároveň nemusíme zakazovať niektoré funkcie, ktoré nechceme aby mohol užívateľov program volať a sú zároveň metódou `window`, ako napríklad `alert()`, ktorého zavolanie v skutočnosti volá `window.alert()`.

Keďže `window` nutne nemusí byť jediný spôsob, ako sa dostať k názvu všetkých premenným, navrhneme aj zložitejšie riešenie, ktoré nepredpokladá nemožnosť enumerácie všetkých premenných pre zachovanie si bezpečnosti.

Drahšie zaručené riešenie

V tejto časti navrhneme z pohľadu ceny prístupu drahšie, ale zaručené riešenie. Všetky riadiace premenné zapuzdríme dovnútra funkcie, čím vytvoríme ekvivalent súkromných premenných. Ďalej vytvoríme metódu, ktorá bude schopná čítať a zapisovať do chránených premenných a zároveň bude môcť byť volaná mimo funkcie.

Výpis 3.2: Ukladanie súkromných premenných

```
function secure_storage () {
  var private_variable
  this.create_variable = function (variable_name){
    eval ("var␣"+variable_name+";")
  };

  this.set_value = function (set_key,set_value) {
      eval (set_key+"=set_value;")
  };

  this.get_value = function(get_key){
    eval ("_buff␣=␣"+get_key)
    return _buff;
  };

}
```

Eval môžeme nahradiť aj vyššou štruktúrou (dictionary). V tomto prípade sa ku každému set/get pridá zložitosť výpočtu hešovacej funkcie pre kľúč. Zároveň je možné použiť jediný slovník - napriek tomu, že jeho názov bude pevne zvolený (a zistiteľný pomocou `secure_storage.toString()`) - nebude k nemu možné natívne pristupovať, ale len pomocou proxy metódy, čo znemožní enumeráciu. Posledná alternatíva je pole, čo môže byť použité v kombinácií s očíslovaním premenných počas obfuskácie.

Súkromné premenné ako `private_variable` sa nedajú žiadnym spôsobom enumerovať - útočník stále môže skúšať rôzne reťazce (ak by vedel názov inštancie `secure_storage` - čo predpokladáme, že vie). Keďže metódy `create_varialbe`, `get_value` a `set_value` sú verejné, je ich možné enumerovať. Preto aj v prípade použitia tejto schémy budeme obfuskovať názvy súkromných premenných.

Útočník by nám ale mohol inštanciu prepísať (podhodiť jeho vlastnú) alebo modifikovať - toto vieme vyriešiť nasledovne: pri každej čítacej operácií si vyžiadame aj jednu ďalšiu - pri obfuskácií ustanovenú náhodnú kontrolnú premennú. Počas obfuskácie riadiacich funkcií prepíšeme kód čítania každej premennej zo `secure_storage` za jednoduchú podmienku, v ktorej najskôr skontrolujeme obsah tajnej premennej. Podmienka zastaví vykonávanie v prí-

pade, že obsah tajnej premennej nebude zhodný s tým, ktorý sme nastavili počas inicializačnej procedúry, keďže by to znamenalo podhodený `secure_storage` objekt. Spôsobený overhead nebude veľmi veľký, keďže túto schému použijeme len pre globálne premenné (používanie lokálnych funkčných premenných je nezmenené).

3.4.2 Obfuskovanie názvov premenných

V ľubovoľnom prípade potrebujeme technicky zabezpečiť nejakú formu obfuskácie názvov premenných. Uvedená metóda je navrhnutá pre prvú metódu (bez použitia `secure_storage` schémy). V prípade použitia `secure_storage` by sa spôsob obfuskácie systematicky nezmenil. Obfuskáciu sme implementovali v jazyku python: zoznam premenných určených na obfuskáciu jednoducho nahradíme ich prekladom (všetky výskyty daného reťazca novým). Tento spôsob je vhodný pre svoju jednoduchosť implementácie. Aby sme dosiahli očakávaný výsledok, žiadna premenná nemôže byť podreťazcom inej a zároveň sa názvy premenných nesmú zmieniť v kontexte inom ako priradenie, čítanie, mazanie (nemôže ju obsahovať nejaký reťazec vrátane názvov natívnych javascriptových funkcií či riadiacich príkazov). Druhú podmienku máme zabezpečenú, tú prvú vieme overiť vrámci nášho obfuskátora triviálne (a vrátiť chybu v prípade nevhodných názvov premenných)

Výpis 3.3: Ukážka pôvodného a zobfuskovaného kódu (bez použitia `secure_storage`)

```
worked_wo_break = 0;
Qrun = user_main

function Qverify (){
while (true){
       d("New round")
       computation_start = (new Date()).getTime()
       if (verify_in_eval_mode){
              d("In eval mode")
              verify_in_eval_mode = false;
```

```
QB6DAFD48D183220821DC482A3080C399 = 0;
QC9C32FD235DC7EEB60AC2C9B7DD6914A = user_main

function Q513FBA994BD35E068AD1D209CEE197E4 (){
while (true){
       d("New round")
       computation_start = (new Date()).getTime()
       if (Q6DC6110A5A060DD4586F48C3891E8824){
              d("In eval mode")
              Q6DC6110A5A060DD4586F48C3891E8824 = false;
```

3.4.3 Zabránenie vykonania kódu mimo funkcie

Jednoduchý program obsahujúci výpočtovo náročný blok kódu mimo funkcie:

Výpis 3.4: Jednoduchý útok modifikujúci tok programu

```
function userprocedue1(){}
while (True){
       steal_cycles()
}
function userprocedue2(){}
```

dokáže prevziať kontrolu nad tokom programu. Podobne ako jeden z prístupov k zakazovaniu funkcií, aj v tomto prípade môžeme navrhnúť špeciálnu syntax, ktorá by povoľovala len kód v tele funkcií. Napriek tomu, že tento problém naše doterajšie riešenie ošetruje - užívateľ nevie korektne zavolať funkciu ohlasujúcu, že kontrolu nad programom má náš riadiaci cyklus a teda daný webworker so stratenou kontrolou vykonávania kódu bude automaticky ukončený. Keby sme tento problém chceli riešiť osobitne, naskytne sa nám zaujímavý podproblém, ktorého riešenie odhaľuje ďalšiu všeobecnú zraniteľnosť, ktorú ošetríme.

Odhalenie mimofunkčného kódu

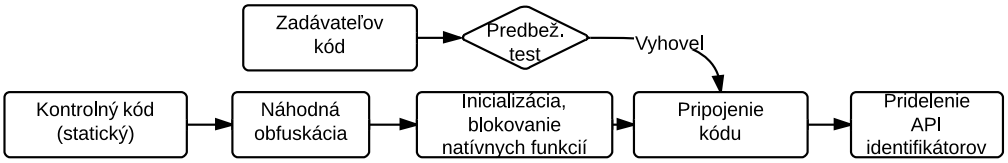
Kód, ktorý sa nachádza mimo tela nejakej funkcie a teda je zavolaný okamžite po inicializácií interpretera môžeme odhaliť nasledovne - užívateľov program spustíme a zapamätáme si čas behu. Pokiaľ táto hodnota bude nižšia, ako nejaká prahová hodnota, budeme považovať, že interpreter použil všetok čas na inicializáciu . Ak časť kódu záležala na nejakej premennej, po jej zmenení nejakou funkciou sa už do daného bloku kódu nemôžeme dostať. Oneskorené spustenie (`setTimeout,setInterval`) nie je možné, keďže tieto funkcie v danom mennom priestore neexistujú (ľubovoľný zadávateľov kód je spustený až po našej inicializačnej funkcií, ktorá znemožní volať zablokované funkcie na základe ich originálnych mien).

Tento prístup však vytvára možnosti na ďalšiu zraniteľnosť. Uvažujme nasledovné dva programy:

```
if (random(0,1)>0.5){
       steal_cycles()
}
```

```
if (Date() > const_future_date){
       steal_cycles()
}
```

Funkcie `Date` , `random` potrebujeme - pomocou `Date` napríklad meriame čas v programovej funkcií. Napriek tomu, že `Date` nie je vieryhodná, keďže vracia čas ako je nastavený na klientskom počítači, dá sa spoľahlivo využiť na meranie časového rozdielu. Užívateľ ju teda nutne nepotrebuje, ale môže potrebovať `random` (pravdepodobnostné algoritmy, simulácie, genetické algoritmy a podobne). Pokiaľ nejakú funkciu necháme voľne



Obrázok 3.2: Schéma generovania kódu pre webworker vlákna

dostupnú a zároveň ju budeme chcieť použiť aj my, vzniká riziko - zadávateľov program ju môže ľubovoľne predefinovať, prípadne môže podhodiť falošnú funkciu. Toto vyriešime viacerými identifikátormi:

Výpis 3.5: Dva identifikátory funkcií

```
public_api_random = random;
private_random = random;
random = null;
```

a `private_random` zobfuskujeme (respektíve zobfuskujeme a uložíme do `secure_storage`). Týmto spôsobom bude možné volať funkciu bezpečne a zároveň je jej volanie dostupné pre zadávateľov program. Priradenie funkcií do ich verejných identifikátorov vykonáme až na konci skriptu, aby nemohli byť volané mimofunkčným kódom.

Blokovanie funkcií

Napriek tomu, že funkcionalita javascriptu je veľmi obmedzená (v porovnaní s niektorými inými skriptovacími jazykmi alebo štandardnými kompilovanými jazykmi), ponúka funkcie, ktoré povolené výpočty nepotrebujú (väčšine vedecko-technických výpočtov stačia funkcie na manipuláciu s poľom, asociatívnym poľom, operácie s reťazcami, príkazy riadiace tok programu a matematická knižnica). Dalšie funkcie ako napríklad `setTimeout()` alebo `eval()` by mohli narušiť nami plánovaný tok vykonávania programu a preto ich používaniu potrebujeme zabrániť. Použijeme predošlý listing s tou zmenou, že identifikátor do verejnej premennej vôbec nepriradíme.

Zabránenie zatvorenia okna

Na to, aby sme si vedeli výpočet pred opustením stránky uložiť (zmraziť), potrebujeme aby existoval časový interval medzi zistením, že sa okno chystá zavrieť a reálnym zatvorením

okna.

HTML štandard ponúka vlastnosť `window.onbeforeunload`. Táto hodnota sa použije ako názov funkcie, ktorá sa zavolá tesne pred opustením stránky (napr. kliknutie na hypertextový odkaz, zavretie okna alebo tab-u prehliadača). Návratová hodnota podobných funkcií (ako napríklad `form.onSubmit()`) má význam - pri *false* sa akcia nevykoná (pri `onSubmit` sa formulár neodošle a pod). V prípade `window.onbeforeunload` sa však kôli potenciálnej možnosti zneužitia táto hodnota ignoruje (čo znemožní znemožnenie zatvorenia okna vrátením *false*) a namiesto toho zobrazí štandardné dialógové okno pýtajúce sa užívateľa, či chce stránku opustiť.

Túto vlastnosť sa nám podarilo obísť nasledovným spôsobom: Otvoríme AJAX request v synchrónnom móde. Na rozdiel od asynchrónneho módu, AJAX má v synchrónnom móde dva základné rozdiely: pred vykonaním ďalšieho kódu čaká na odpoveď a synchrónnu požiadavku môže poslať len na doménu z ktorej bol načítaný (neumožňuje takzvaný cross-domain request). V prípade, že kód na serveri zdrží odoslanie (napríklad. `<? sleep(2); ?>`) budeme môcť zatvorenie stránky ľubovoľne dlho zdržať.

Tento poznatok využijeme pri zmrazovaní výpočtu. Keďže našich webworkrov nemôžeme okamžite zmraziť, musíme počkať kým sa dostaneme do najbližšej iterácie hlavného riadiaceho cyklu - nastavením príznaku zatvorenia okna mu signalizujeme zastavenie a následné zmrazenie výpočtu. Vzhľadom na maximálny povolený časový úsek vykonávania užívateľskej funkcie vrátenie do hlavného riadiaceho cyklu trvá rádovo najviac desiatky milisekúnd (v prípade, že vykonávame odhad aktuálneho výpočtového výkonu, budeme overovať počas výpočtu daný príznak - ak bude signalizovať ukončenie, jednoducho zastavíme proces odhadovania aktuálneho výpočtového výkonu a budeme pokračovať v toku ako v prípade ukončenia jednej programovej funkcie). Keďže určitý čas trvá cesta od požiadavky na server a naspäť, čakanie na strane nemusí byť žiadne, alternatívne v desiatkach milisekúnd. Maximálne celkové predĺženie času zatvorenia bude v rádovo menších stovkách milisekúnd čo si užívateľ pravdepodobne vôbec nevšimne.

Toto správanie funguje aj pre iframe (ak je stránka načítaná pomocou iframe v inej, ktorá nemá túto metódu nastavenú)

Zmrazenie výpočtu

Výpočet zmrazujeme počas verifikačného cyklu - stav zadávateľovho programu v tomto momente charakterizuje obsah jeho globálnych verejných premenných a identifikátor ďalšej funkcie, ktorá sa má zavolať (funkcí je viacero, keďže limitujeme čas strávený v jednom zavolaní a ekvivalenciu `sleep` vieme emulovať pomocou `setTimeout`, pomocou ktorej tak môžeme skočiť na ďalší blok vykonávaného kódu - zabalenom a identifikovanom pomocou nejakej funkcie). Toto si pomocou `JSON.stringify` uložíme a požijeme hlavný skript

ako `localStorage` proxy, keďže tento objekt nie je dostupný v mennom priestore webworkrov. Ten podobne pri každom načítaní overí stav uložených výpočtov a v prípade, že nejaké nájde, pred štartom ich načíta webworkrom.

3.5 Dynamická alokácia výkonu vzhľadom na zaťaženie CPU inými procesmi

Vzhľadom na požiadavku, aby náš výpočet čo najmenej obťažoval užívateľa (zvýšením latencie ostatných procesov zahltením procesora) budeme vrámci round robin schémy implementovať dynamické menenie dĺžky prestávky. Pri statickom:

```
start = now()
work_slice() // 100% cpu
stop = now()
wait ((start-stop)*K)
```

By sme dosiali priemerné vytaženie (nášho podielu) procesora :

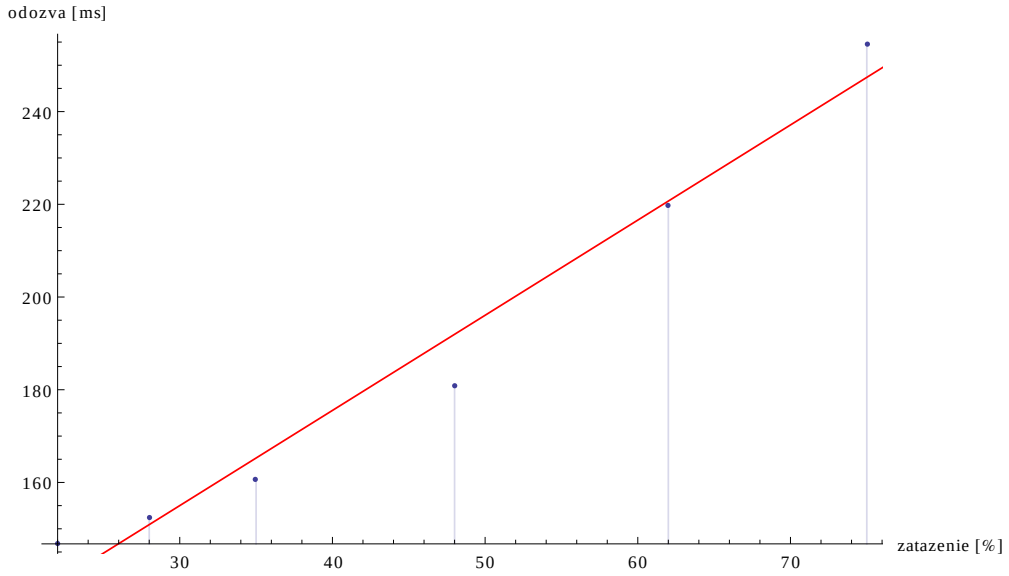
$$(\frac{1}{1+K}) * 100\% \tag{3.5.1}$$

V prípade, že užívateľ inicializuje spustenie ďalšieho procesu, ktorý je výpočtovo náročný, plánovač operačného systému zmenší náš podiel procesora. Napriek tomu však proces, ktorý spustil užívateľ bude bežať s menším podielom procesora ako v prípade nevykonávania nášho systému. Keďže zabezpečenie responzívnosti užívateľovho počítača má vyššiu prioritu ako rýchle vykonávanie nášho výpočtu (v opačnom prípade by mohlo jednoducho prísť k nedôvere zo strany používateľa a permanentnému zakázaniu vykonávania našich výpočtov), v prípade zaznamenania, že v systéme beží ďalší výpočtovo náročný proces, jednoducho znížime podiel nám prideleného výkonu procesora, ktorý využijeme.

Detekcia ďalšieho na výkon náročného procesu

Aby sme odskúšali, ako funguje prideľovanie výkonu, spustili sme v našom systéme jeden proces, ktorý stabilne vyťažoval jednu jadro CPU pre rôzne hodnoty vyťaženia. Ďalej sme na tom istom jadre spustili internetový prehliadač s benchmark funkciou a zmerali sme čas potrebný na jej vykonanie. Na danom jadre nebežali iné aplikácie (jednotlivé procesy sme pridelili jadrám pomocou unixového programu `taskset`). Oba procesy mali nastavenú rovnakú prioritu. Uvedenú závislosť zobrazuje graf 3.3

Z grafu sa dá usúdiť, že plánovač funguje približne lineárne. Tento poznatok použijeme pri výpočte koeficientu spomalenia.



Obrázok 3.3: Závislosť vyťaženia jedného CPU jadra a dĺžky vykonávania javascriptového benchmarku

Výpočet koeficientu spomalenia

Koeficient budeme pre každé vlákno rátať osobitne. Na jednej strane sa nám zvýši overhead, na strane druhej nás však jedno vyťažené jadro neprinúti obmedziť výpočet aj na užívateľom nevyťažených jadrách. Tento kód teda bude bežať na úrovni každého pracujúceho javascriptového webworkra.

Riadiaci cyklus vypočíta koeficient a pokračuje vo výpočte programu. Počas jednotlivých iterácií riadiaceho cyklu (návrat z funkcie) skontrolujeme, či nám platnosť koeficientu nevypršala - v ideálnom prípade by sme mali prehľad o vyťažení neustále (menil by sa spojito) - toto však nie je možné bez veľkého overheadu [1]. Samotné vypočítanie koeficientu je výpočtovo náročné (zahltí na chvíľu všetky prostriedky) a preto treba nájsť pomer medzi menej často počítaným koeficientom (čo môže mať za následok neadekvátne vyťaženie) alebo častým prepočítavaním (čo zvýši overhead).

Zároveň je potrebné odhadnúť aj približnú dĺžku výpočtu koeficientu - v prípade kratšej nám signifikantnejšie ovplyvní hodnotu aj menšia fluktuácia, dlhšie rátanie zvýši overhead. Keďže formálna optimalizácia týchto hodnôt závisí aj od ”škody” spôsobenej nepresným koeficientom, pravdepodobnosti a intenzity rôznych fluktuácií, ktoré je potrebné odhadnúť, vystačíme si s arbitrárne odhadnutými konštantami pre náročnosť výpočtu koeficientu a frekvenciu jeho obnovovania.

Na základe linearity plánovača teda vieme približne odhadnúť ako veľmi sa zmenilo za-

[1] Alternatívne by bolo možné rátať výkonnosť ako množstvo iterácií užívateľskej funkcie za jednotku času. Tento prístup ale potrebuje predpoklad, že kód vo všetkých funkciách beží homogénne (časom sa nedostáva na komplikovanejšie vstupy a zároveň sa výrazne nemení tok - napríklad pre rátanie kolízií a pod.)

ťaženie od posledného prepočítania. Keďže pri zvýšenej zátaži chceme menej ako lineárne veľa výpočtu, bude to nejaká klesajúca funkcia v závislosti od súčasného vyťaženia (označíme R).

Relativita koeficientu

Prvýkrát vypočítame koeficient pri inicializácií, ktorý považujeme za základný (pri žiadnom vedľajšom zaťažení). Môže nastať nasledovná situácia. Pre ilustráciu problému stanovíme

$$K = \frac{(\frac{R}{100})^{1.5}}{\frac{1}{0.167}} \tag{3.5.2}$$

# výpočtu	R ([ms] behu)	K	Približné zaťaženie CPU [%]
1	100	0.167	85.6%
2	200	0.472	67.9%
3	400	1.336	42.8%
4	50	0.059	94.4%

Menovateľ zlomka vzťahu 3.5.2 je prispôsobený, aby podľa 3.5.1 bol maximálny výkon normalizovaný na 85%.

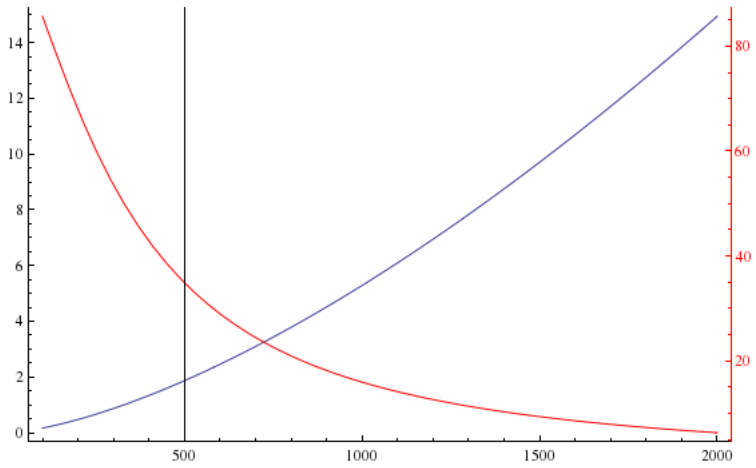
Štvrtý výpočet zrejme vracia nevhodnú hodnotu. Popísaný algoritmus má nedostatok v tom, že za maximálny výkon automaticky považuje výkon počas inicializácie a na základe toho prepočítava koeficient pre jednotlivé záťaže. Keďže na základe informácií, ktoré nám internetový prehliadač klienta posiela v hlavičke nevieme určiť výkon, použijeme túto metódu so samoopravnou vlastnosťou: v prípade, že výkon v nejakom okamihu stúpne nad 100% predpokladanej kapacity, jednoducho nahradíme maximálnu hodnotu výkonu novou a následne budeme prerátavať výkon analogicky podľa nej.
Nasledujúca tabuľka ukazuje možnosť pokračovania tauľky hore s použitím samoopravnej vlastnosti.

# výpočtu	R ([ms] behu)	K	Približné zaťaženie CPU [%]
5	50	0.167	85.6%
6	10	0.472	67.9%
7	50	1.336	42.8%

V prípade, že vyťaženie bolo krátkodobé, na (približne) správnu hodnotu sa dostaneme rýchlo.

Návrh závislosti K od R

Pre reálne nasadenie by krivka mala po nejakú hodnotu klesať pomalšie a od nejakej hodnoty klesať rýchlo, čo zabezpečí maximálny výkon v prípade, že ho užívateľ potrebuje.



Obrázok 3.4: Vzťah medzi R a K v ilustračnom príklade, horizontálna os - *R* [ms], čierna vertikálna - *K*, červená vertikálna - predpokladané vyťaženie CPU [%]

Keďže počítanie koeficientu je dlhšie trvajúca operácia (môže trvať viac ako maximálny povolený čas vrátenia hodnoty programovej funkcie), responzívnosť vlákna zabezpečíme tak, že každých niekoľko iterácií (resp. časových jednotiek) budeme hlásiť našu aktivitu kontrolujúcemu vláknu.

Zabránenie paralelného behu viacerých inštancií

Nech náš kód distribujú dve rôzne stránky, ktoré má užívateľ otvorené (respektíve viac ako jedno okno jednej distribujúcej stránky). Potrebujeme zabrániť, aby došlo k spusteniu viac ako jedenej inštancie.

Použijeme zamykanie cez `localStorage`. Po ukončení výpočtu konkrétna inštancia našej služby odomkne zámok zabraňujúci výpočtom ďalším inštanciám.

V prípade neočakávaného zrušenia výpočtu ostane hodnota zámku *locked* ako *true* ale prestane sa obnovovať `locked_last`, ktorú pravidelne obnovujeme kým výpočet beží. Po prekročení prahovej hodnoty rozdielu súčasného času a `locked_last` sa výpočtu uchopí ďalšia voľná inštancia (každá inštancia v pravidelných intervaloch kontroluje hodnotu zámku). Každá z nich má prístup k danej premennej, keďže aj pri rôznych stránkach šíriacich náš javascript - doména (ktorá určuje prístup k `localStorage`) distribujúca náš riadiaci skript a výpočet je stále tá istá.

3.5.1 Návrh API funkcií

Príjem a odosielanie dát

Keďže AJAX požiadavky, ktoré použijeme na načítanie vstupov a odosielanie výsledkov bežia nezávisle na ostatnom toku programu, v prípade, že zadávateľov program zistí, že sa

blíži ku koncu vstupu, jednoducho zavolá príslušnú funkciu, ktorá na pozadí získa vstup a pridá ho do nejakej verejnej premennej, obdobne pre odosielanie výsledkov. Takto nebude potrebné čakať na sieť a tým bude možné využiť väčšiu časť výpočtových zdrojov.

Referenčná hodnota aktuálneho výkonu

Na to, aby vykonávaný program vedel bežať efektívne a zároveň vracať sa z funkcie v určitých intervaloch, potrebujeme mu nejako dať vedieť aký výkonný je počítač (a interpreter), na ktorom momentálne beží kód a ako veľmi je aktuálne zaťažený. Túto informáciu my už máme - vyjadrili sme ju pomocou výpočtu koeficientu spomalenia. Pre zadávateľov program je táto hodnota len orientačná.

Kapitola 4

Verifikácia výsledkov

Podobne ako pri volunteer computingu, aj náš systém uskutočňuje výpočty u klientov, ktorí nie sú dôveryhodní. Výpočet nejakého programu teda môže byť jednoducho zmanipulovaný. Vzhľadom na konštrukciu nášho systému je pomerne jednoduché získať formát vstupných a výstupných dát. Útočník by teda mohol jednoducho sabotovať výpočet - získa dáta a následne odošle nejaké (ľubovoľné) pod hlavičkou výsledku.

Heuristika

Existuje viacero prístupov riešiacich tento problém heuristickými metódami - jemnou modifikáciou programových dát (napríklad pridaním určitého kontrolného súčtu) a následnou kontrolou integrity na serveri, blacklistovaním IP adries generujúcich zlé výsledky, analýza frekvencie a rýchlosti spracovávania a podobne. Nevýhodou týchto metód je ich nekompatibilita s Kerckhoffovým princípom - sú to metódy, ktorých cieľom je zmiasť útočníka. Dostatočne sofistikovaný útočník si však môže zmeniť IP adresu či odpozorovať princíp generovania kontrolného súčtu.

Sabotujúci klienty

Sabotujúcich klientov budeme uvažovať ako takých, ktorí na časť výsledkov odpovedajú nesprávne. Budeme predpokladať, že sa medzi sebou nedohadujú (teda ak dva sabotujúce klienty pre daný program a vstup odošlú nesprávne riešenia, nebudú rovnaké (nebudú sa medzi sebou dohadovať)).

V prípade nespolupracujúcich klientov je prakticky nemožné náhodne zvoliť tú istú zlú odpoveď, budeme teda predpokladať, že dve rovnaké odpovede pre daný vstup od rôznych klientov sú dostatočné na overenie správnosti.

V tejto časti je vhodné spomenúť algoritmy, ktoré sú odolné voči určitému podielu chybných výpočtov (napríklad niektoré evolučné algoritmy), prípadne také, pri ktorých nejaká

časť výpočtov, ktorá je chybná nepokazí riešenie alebo len zníži jeho kvalitu podľa nejakej ohodnocovacej funkcie.

Overhead

Overhead H zadefinujeme ako pomer vykonaných výpočtov voči množstvu výpočtov pri vlastnej architektúre ($H = 2$ znamená, že počítame dvojnásobok oproti tomu, čo potrebujeme a teda náš systém má v tomto prípade len polovičnú kapacitu (vo všeobecnosti $\frac{1}{H}$)

Homomorfné šifrovanie

Pri homomorfnom šifrovaní klient manipuluje s dátami a programom, ktoré boli pred načítaním transformované tak, aby bolo znemožnené podvádzať. Tento prístup má dve výhody - verifikovateľnosť výsledkov a znemožnenie prečítania originálneho programu a vstupných aj výstupných dát. Kým verifikovateľnosť môžeme zabezpečiť aj inými metódami (za rôznych predpokladov - sabotujúci klienti medzi sebou v dostatočnej miere nespolupracujú a ďalšie), utajenie programu a vstupu sú špecifické pre tento prístup. To môže mať výhody v rôznych aplikáciach (napríklad výpočtovo náročný proprietárny výskum a podobne). Samotné výpočty nad homomorfne zašifrovanými dátami sú však drahé, čo je hlavná nevýhoda tohto prístupu. Návrh takýchto výpočtov sa nachádza v [2]

4.0.2 Replikácia

Pri replikácií každú úlohu posielame vypočítať viackrát (zakaždým rôznemu klientovi) a o výsledku rozhodujeme hlasovaním. V prípade, že všetky výpočty zahlasujú za rovnaký výsledok (alternatívne väčšina) daný výsledok považujeme za správny. Problémom replikácie je vysoký overhead. Pri nespolupracujúcich klientoch stačí $H = 2$ v prípade spolupracujúcich je potrebné voliť H na základe odhadu podielu sabotujúcich klientov.

4.0.3 Kontrolné úlohy

V prípade, že nejaký klient vracia zlé výsledky, stačí nám nájsť jeden zlý výsledok, aby sme sabotujúceho klienta odhalili. Keďže kontrolné úlohy nemôžu byť rozpoznateľné od pravých, na overovanie budeme používať reálne vstupy pre program, ktorý počítame. Majme:

1. Pošleme niekoľko atomických úloh
2. Zozbierame výsledky
3. Náhodne vyberieme niekoľko výsledkov, ktoré overíme u iného klienta

4. V prípade nájdenia úlohy, na ktorú odpovedal zle, označíme klienta za sabotujúceho (v opačnom prípade pokračujeme ďalej od 1.)

Keďže zatiaľ nevieme, koľko úloh budeme musieť prepočítať na to, aby sme dostali vysokú pravdepodobnosť toho, že výsledok je korektný, budeme náhodne vybrané výsledky kontrolovať u klientov (alternatívne by sme ich mohli počítať mimo siete, čo by ale výrazne zvýšilo nároky na výpočty na našej strane).

Vyjadríme pravdepodobnosť odhalenia zlého výpočtu pri kontrole: p_{ns} označuje pravdepodobnosť neodhalenia chybného výpočtu za predpokladu, že kontrolujeme výsledky sabotujúceho klienta. Nech overujeme N úloh (výsledkov), z toho Z chybných výpočtov a reálne prerátavame K úloh. Pravdepodobnosť odhalenia chybného výpočtu za predpokladu sabotujúceho klienta označíme p_{os}. Potom máme:

$$p_{ns} = \frac{\binom{N-Z}{K}}{\binom{N}{K}} \tag{4.0.1}$$

Kde v čitateli je množstvo výberov úloh, ktoré sú vyrátané korektne a v menovateli je množstvo všetkých možných výberov. Predpokladáme $N \geq Z + K$ (v opačnom prípade $p_{ns} = 0$)

Vzťah nápadne pripomína hypergeometrické rozdelenie ($X \sim Hyp(N,Z,K)$) a

$$p_{ns} = P[X = 0] \tag{4.0.2}$$

Ďalej,

$$p_{os} = 1 - p_{ns} \tag{4.0.3}$$

Nech $k = \frac{K}{N}$ a $z = \frac{Z}{N}$, potom malé písmená vyjadrujú daný pomer k celkovému počtu úloh. Pre overhead platí

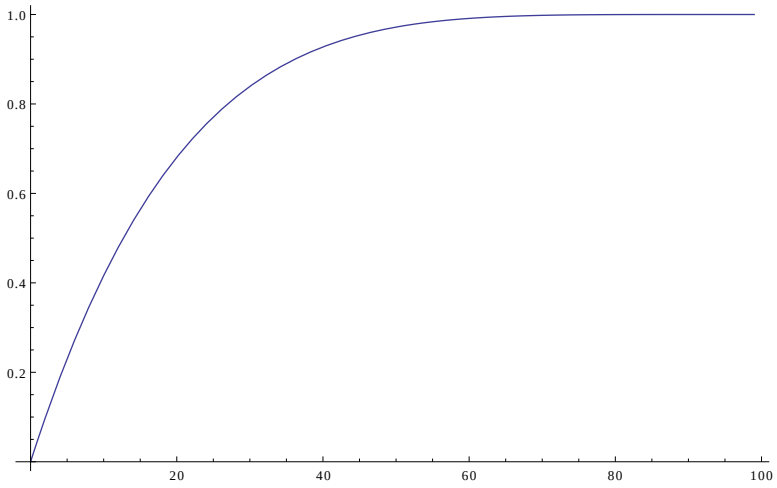
$$H = 1 + k$$

Zrejme menšie z bude znižovať pravdepodobnosť odhalenia chýb a väčšie k bude túto pravdepodobnosť zvyšovať. Sabotujúce klienty by preto teoreticky mohli zvoliť za z veľmi malé číslo - zvýšili by tým pravdepodobnosť akceptovania zlého výsledku. Na druhej strane túto voľnosť nemusia nutne mať, keďže malé množstvo chybných výpočtov nemusí postačovať na "pokazenie" výsledku. Zvolíme $z = 0.05$. Graf 4.1 ukazuje pravdepodobnosť odhalenia zlého výpočtu u sabotujúceho klienta od K pre $N = 100$

Pravdepodobnosť odhalenia chyby je väčšia ako 99% až pre $k = 0.59$, čo je pomerne veľký overhead. Navrhli sme preto alternatívny spôsob:

Zoskupovanie výsledkov

Pri zoskupovaní výsledkov zvýšime N. Pri nezmenenom z a k potom dostaneme oveľa vyššie hodnoty p_{os}. Ak by sme v jednej sérií klientovi poslali $N = 100$ atomických úloh, jednoducho



Obrázok 4.1:

počkáme kým vyrieši ďalšie série a kontrolu necháme prebehnúť neskôr.

Z definície p_{ns} máme

$$\frac{\binom{N-Z}{K}}{\binom{N}{K}} = \frac{(N-Z)!(N-K)!}{N!(N-Z-K)!} = \frac{(N-Z)^{\underline{K}}}{N^{\underline{K}}}$$

Pre zoskupených g vstupov

$$p_{ns} = \frac{(g(N-Z))^{\underline{gK}}}{gN^{\underline{gK}}}$$

Pre $Z > 0$ máme:

$$\lim_{g\to\infty} \frac{(g(N-Z))^{\underline{gK}}}{(gN)^{\underline{gK}}} = 0$$

Na základe tohoto vieme teoreticky dosiahnúť *ľubovoľnú* presnosť pri ľubovoľne malých hodnotách k,z.

Týmto spôsobom teda môžeme získať pre nezmenené k,z väčšiu presnosť, prípadne znížiť overhead pre nejakú arbitrárne zadanú presnosť. Z praktického hľadiska však g nemôže mať veľmi veľké hodnoty, keďže dané výpočty by nemusel jeden klient vykonať (prípadne by sa výrazne zvýšil tzv. turnaround time).

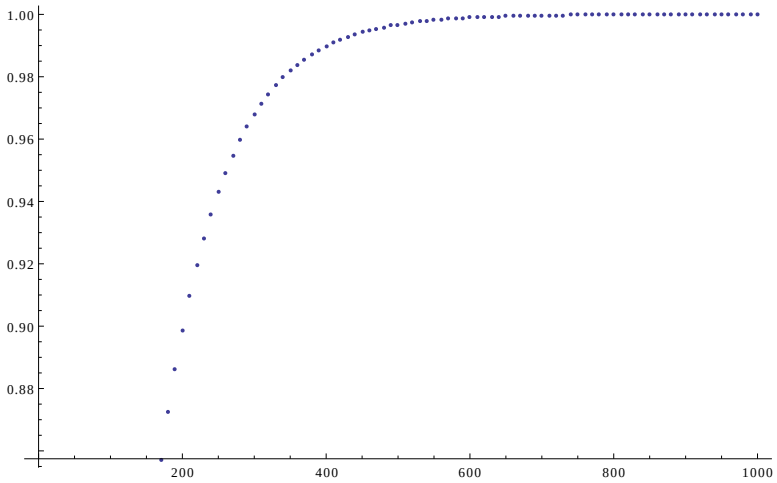
Celková chyba

Za predpokladu homogénneho rozdelenia úloh sabotujúcim aj nesabotujúcim klientom, pravdepodobnosť zle vypočítanej série úloh je menšia ako p_{ns}, keďže p_{ns} vyjadruje podmienenú pravdepodobnosť. Ak označíme s podiel sabotujúcich klientov, máme:

$$p_{ns} = P(neodhalchybu|sabotujci) = \frac{P(neodhalichybu \cap sabotujuci)}{P(sabotujuci)}$$

Keďže pri nesabotujúcom klientovi nemôžeme odhaliť chybu,

$$P(neodhalichybu \cap sabotujuci) = P(neodhalichybu)$$



Obrázok 4.2: Graf ukazuje závislosť p_{os} od N pre fixné k a z

a teda

$$P(neodhalichybu) = p_{ns}s$$

kde s je podiel sabotujúcich klientov.

True/False problém

V prípade, že je odpoveď na problém true alebo false (problém príslušnosti do jazyka). V tomto prípade existuje len jediná zlá odpoveď. Jeden z takýchto problémov môže byť napríklad

Je <string> obrazom nejakého vzoru pri danej hašovacej funkcií ?

V tomto prípade môžeme problém bez výrazného zvýšenia zložitosti modifikovať tak, že zmeníme formát výsledku, aby obsahoval nejakú informáciu z výpočtu, ktorú bude ťažké náhodne trafiť.

4.0.4 Granularita

Ďalším dôležitým aspektom je granularita úloh, ktoré počítame. Vysoká granularita znamená, že veľkosť atomickej úlohy je malá a vice versa. Optimálne nastavenie granularity je netriviálny problém, v ktorom je potrebné zohľadniť:

- Schopnosť klienta dokončiť výpočet (preferuje vyššiu granularitu)
- Overhead - väčšia administratíva na jednu atomickú úlohu - na strane klienta aj servera (preferuje nižšiu granularitu)
- Verifikácia - v prípade implementácie verifikácie náhodným výberom výsledkov na kontrolu je lepšia vyššia granularita

- Pamäťové nároky - všetky dáta sú uložené v RAM, ďalší limit je `localStorage` (preferuje vyššiu granularitu)

Overhead pri vyššej granularite

Z hľadiska vyťaženia procesora môžeme overhead pri zvýšenej granularite zanedbať (oproti výpočtovo náročným operáciám na dátach)

Z hľadiska zaťaženia siete - jedno AJAX volanie je HTTP požiadavka. Keďže vstupné dáta musíme prijať v každom prípade, pri rozdelení tých istých dát do viacerých sérií zvýšime množstvo balíkov, v ktorých ich dostaneme. Z tohto dôvodu budeme zbytočne prenášať HTTP hlavičky. Veľkosť hlavičky záleží od viacerých faktorov sa pohybuje v ráde niekoľkých stovkách bajtov. Vysoká granularita dát je zaťažená hneď 4 hlavičkami - pri načítaní vstupných dát (hlavička od prehliadača a servera) a pri odosielaní výsledku. V prípade 500B hlavičky sú to 2KB. Overhead zrejme závisí od pomeru množstva dát prenesených pod jednou hlavičkou a jej veľkosti (v prípade 20KB dát je overhead 10%). Ďalej je možné posielať viacero atomických úloh v jednej HTTP požiadavke (napríklad za účelom efektívnejšej verifikácie pri nezvýšených požiadavkách na sieťové prostriedky).

Problém optimálnej granularity je pomerne všeobecný a granularitu je teda potrebné odhadovať pre každý problém osobitne.

Kapitola 5

Etická stránka

Počas prezentácie témy mojej bakalárskej práce spolužiakom som sa prvý krát stretol s problémami týkajúcich sa etického charakteru.

Využívanie výpočtovej sily užívateľa bez jeho explicitného súhlasu na účel, ktorý nebol ním inicializovaný môže vyvolávať rôzne obavy. Z tohoto dôvodu je súčasťou práce robustný mechanizmus zabezpečujúci bezpečné vykonávanie programu, ktorý bol dodaný nedôveryhodnou treťou stranou. Architektúra zároveň znemožňuje pristupovanie k dátam DOM domény distribujúcej náš systém, čo zabezpečuje súkromie užívateľa. Celá štruktúra je navyše vykonávaná v javascriptovom interpreteri daného prehliadača, ktorý pridáva ďalšiu vrstvu ochrany a obmedzení predchádzajúcich vykonávaniu natívneho kódu.

5.0.5 Strata výkonu

Ďalšou etickou stránkou je strata výkonu - náš systém dynamickej alokácie vzhľadom na aktuálne zaťaženie znižuje požadovaný výkon, čo má za následok maximalizáciu dostupného výkonu v časoch, kedy ho užívateľ potrebuje najviac. Užívateľ taktiež môže regulovať množstvo alokovaného výkonu.

5.0.6 Zvýšenie spotreby fyzických prostriedkov

Náklady na elektrickú energiu

Spustením nášho systému bude daný osobný počítač zaťažený viac ako v prípade štandardného používania. To zabráni možnému podtaktovaniu, prípadne podobných schém šetrenia spotreby elektrickej energie. Účasťou vo výpočte bude implikovať zvýšené náklady na energiu. Vzhľadom na trend znižovania spotreby procesorov a spotreby súčasných procesorov by však nárast nákladov nemal byť výrazný.

Náklady na dáta

Pri určitých typoch pripojenia obmedzujúcich množstvo prenesených nespoplatnených dát, môže náš systém pre užívateľa vygenerovať ďalšie náklady vo forme zvýšeného objemu prenesených dát. Jedno z riešení by bolo vytvorenie zoznamu poskytovateľov pripojenia, ktorí výrazne spoplatňujú dáta (napr. mobilné siete, dial-up a podobne) - respektíve rozsahy IP adries, pre ktoré daní poskytovatelia limitujú množstvo nespoplatnených dát (keďže jeden operátor môže poskytovať rôznu škálu internetových pripojení) a distribuovať výpočty klientom, ktorých IP adresa nepatrí do nejakého z týchto rozsahov. Tiež je možné pamätať si množstvo prenesených dát a použiť ho na obmedzenie ďalších výpočtov.

5.0.7 Iné etické zábrany

Ďalšia, z etických zábran môže smerovať k povahe dát a programov, ktoré sú počítané u daného klienta. Môže existovať množstvo výpočtových projektov, ktorých sa užívateľ nechce zúčastniť z charakteru výpočtu (náboženské,filozofické a iné dôvody). Možné riešenie tohoto je pripínať ku každému distribuovanému programu aj text obsahujúci popis daného programu a možnosť nezúčastňovať a na výpočte daného projektu. Keďže odvodenie popisu nie je algoritmický možné, popis by musel dodávať spolu s programom zadávateľ. Zadávateľa ale nepovažujeme za dôveryhodného, preto je tento postup otázny.

5.0.8 Priamy benefit pre používateľa

V súčasnosti potrebujú vysoký výkon na riešenie zložitých problémov aj odvetvia ako biológia, chémia alebo medicína (skladanie bielkovín, návrh liekov, výpočty v genetike a podobne). Toto je jeden z príkladov, ako môže z výpočtu priamo benefitovať klient.

5.0.9 Zhrnutie

Užívateľ je taktiež schopný pozastaviť, prípadne úplne zakázať prebiehajúci, alebo aj všetky budúce programy. Vzhľadom k ostatným zmieneným pravidlám a opatreniam sa nám teda nepodarilo nájsť argument etického charakteru, ktorý by predstavoval relevantný protiargument k rozsiahlemu spúšťaniu našej konštrukcie.

Záver

Úspešne sa nám podarilo navrhnúť systém distribúcie výpočtov a naprogramovať ”proof-of-concept” schému. Poukázali a zanalyzovali sme viaceré aspekty ako výkon, konštrukcia, bezpečnosť, transparentnosť, jednoduchosť implementácie či verifikáciu.

Do budúcnosti môžeme predpokladať s nárastom podielu výkonu grafických kariet k celkovému inštalovanému výkonu a potrebou zohľadniť tento fakt v návrhu nášho systému. Predpokladáme, že efektívne využitie výkonu grafických kariet bude možné pomocou internetového prehliadača aj mimo štandardného grafického aspektu (WebGL), keďže viaceré výpočtovo náročné aplikácie prechádzajú do tohoto kontextu. V súčasnosti táto podpora existuje v počiatočných fázach (ako napríklad WebCL[1]) s funkčnými demonštráciami.

Ďalší zaujímavý projekt je *Native Client*[2], ktorý mieri na možnosť spúštania natívneho kódu v prehliadači, čo môže znížiť, prípadne eliminovať neefektivitu spôsobenú súčasným vykonávaním vrámci interpretera.

[1] http://webcl.nokiaresearch.com

[2] http://developers.google.com/native-client

Literatúra

[1] Computer language benchmark game http://benchmarksgame.alioth.debian.org/.

[2] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. 2009. http://eprint.iacr.org/.

[3] C. Reginald, G. Putra, A. Belloum, S. Koulouzis, M. Bubak, and C. de Laat. Distributed computing on an ensemble of browsers. *Internet Computing, IEEE*, PP(99):1–1, 2013.

[4] TOP500. Top500 supecomputer list http://www.top500.org/lists/2012/11/, 2012.

[5] W3C. Web workers candidate recommendation http://www.w3.org/TR/workers/, 2012.

[6] W3C. Web storage proposed recommendation http://www.w3.org/TR/webstorage/, 2013.