Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Výukové prostredie na simuláciu SIMD algoritmov

Bakalárska práca

2022
Jozef Číž

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Výukové prostredie na simuláciu SIMD algoritmov

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	prof. RNDr. Rastislav Kráľovič, PhD.

Bratislava, 2022
Jozef Číž

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Jozef Číž
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Výukové prostredie na simuláciu SIMD algoritmov
Environment for teaching SIMD algorithms

Anotácia: Cieľom je vytvoriť výukové prostredie, v ktorom sa budú dať spúšťať paralelné SIMD algoritmy zapísané vo formalizme z učebnice JáJá: An introduction to parallel algorithms. Táto populárna učebnica požíva work-time formalizmus zavedený Vishkinom ako snahu vytvoriť technologicky nezávislý vysokoúrovňový
jazyk, v ktorom sa ľuďom pohodlne navrhujú a analyzujú paralelné algoritmy. Pri kurze postavenom na tejto učebnici je vhodné, aby študenti mali možnosť programovať paralelné algoritmy v tomto formalizme v prostredí, ktoré nie je náročné na naučenie; v ideálnom prípade by pseudokód z učebnice mal byť priamočiaro spustiteľný v simulátore. V súčasnosti existuje kompilátor a virtuálny stroj, ktorý tento jazyk podporuje, ale jeho použitie pri výuke je obmedzené neveľmi prívetivým používateľským rozhraním. Cieľom práce je vytvoriť
multiplatformové vývojové prostredie s editorom a debuggerom, ktorý umožní prehľadne trasovať beh programov.

Vedúci: prof. RNDr. Rastislav Kráľovič, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 04.11.2021

Dátum schválenia: 04.11.2021 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Chcem poďakovať môjmu školiteľovi, Rasťovi Královičovi, za výučbu masívne zaujímavého predmetu EPA a za pomoc s bakalárskou prácou, Samuelovi Čavojovi za kritickú pomoc s CMakom a LaTexom, Dávidovi Mišiakovi za občasnú motiváciu a Janke Uhrinovej za extrémnu pomoc s čiarkami.

Abstrakt

Implementovali sme funkčné grafické vývojové prostredie určené pre výučbu a tvorbu paralelných programov v jazyku WT* na magisterskom predmete Efektívne paralelné algoritmy. Súčasťou prostredia je textový editor s podporou pre programovací jazyk WT*, súborový strom a plnohodnotný ladič s podporou pre dynamické body prerušenia, krokovanie a zobrazovanie obsahu premenných. Body prerušenia sme spravili podmienené, pričom podmienky sú písané v natívnom jazyku a podporujú väčšinu jeho schopností vrátane cyklov a manipulácie s vonkajšími premennými.

Kľúčové slová: PRAM, IDE, kompilátor, ladič

Abstract

We have implemented a functioning integrated graphical development environment designed for teaching and creating parallel programs in the WT* language as a part of masters course Effective parallel algorithms. The environment includes an editor with support for WT* programming language, filetree and a fully featured debugger with support for dynamic breakpoints, stepping, and displaying content of variables. We made breakpoints conditional, with conditions written in native language and supporting most of its features, including cycles and manipulation of global variables.

Keywords: PRAM, IDE, compiler, debugger

Obsah

Zoznam obrázkov

Úvod

Vo svete pozorujeme stúpajúcu potrebou riešiť stále náročnejšie problémy a spracovávať masívnejšie množstvá dát. Dopyt po neustále väčšej výpočtovej sile tak neprestal ani po dekádach jej exponenciálneho rastu. Na uspokojenie výpočtových nárokov digitálneho sveta sa hľadajú všetky možné riešenia a jednou z odpovedí je paralelizmus. Využitie paralelnej architektúry však prináša veľmi odlišný spôsob výpočtu, a teda aj nové problémy. To v kombinácii s lacným zrýchľovaním jednovláknového výkonu malo za následok, že viacjadrové procesory a masívne paralelné grafické karty sa rozšírene začali používať iba relatívne nedávno.

Problémy paralelnej paradigmy je potrebné zohľadniť, a teda im prispôsobiť postup riešenia úloh a návrh algoritmov. Oblasť paralelných algoritmov je skúmaná už dlho. Jedným z najznámejších a najpoužívanejších modelov používaných na ich výskum je Parallel Random Access Machine (PRAM). Vďaka tomuto modelu abstraktného stroja môžeme skúmať podstatu problému bez nutnosti jednania s detailami konkrétnych reálnych architektúr, rovnako ako pri využívaní modelu Random Access Machine v prípade sekvenčných algoritmov. Vieme tak hľadať všeobecné výpočtové hranice paralelných algoritmov a nie konkrétne hranice niektorých architektúr. Keďže ide o abstraktný model, kvôli niektorým jeho aspektom ho nie je možné efektívne zrealizovať v reálnom svete, ale ukazuje sa, že modely podobného charakteru realizovať možné je [1], a teda určite má význam PRAM študovať.

Parallel Random Access Machine je model stroja pracujúceho s niekoľkými procesormi nad zdieľanou pamäťou s podporou pre paralelný prístup. Konflikty prístupu do pamäte môžu byť riešené viacerými spôsobmi od exkluzívneho prístupu pri čítaní a zapisovaní (EREW) až po súbežné čítanie a zapisovanie s určenými prioritami zapisovaných dát (Priority CRCW). Programy pre tento model sú písané formou Single Instruction Multiple Data (SIMD), kde v každom momente všetky vlákna vykonávajú rovnakú inštrukciu, pričom niektoré z vlákien môžu byť neaktívne a dáta, s ktorými inštrukcie pracujú, sa môžu líšiť. Toto je podobné viacvláknovému programovaniu na bežných počítačoch, avšak v našom prípade sú všetky vlákna implicitne synchronizované po každej inštrukcii, čo zjednodušuje premýšľanie nad priebehom vyhodnocovania.

Výuka s využitím modelu PRAM prebieha aj na našej Univerzite Komenského na magisterskom predmete Efektívne paralelné algoritmy (EPA). Študenti spoznávajú zá-

kutia paralelného programovania a zoznamujú sa s komplikovanými algoritmami na základe učebnice *An introduction to parallel algorithms* od autora Joseph JáJá [2]. V tejto učebnici JáJá opisuje paralelné algoritmy pomocou *práca-čas prezentačnej platformy* (anglicky *work-time presentation framework*), ktorú budeme volať WT. WT poskytuje vysokoúrovňový prístup ku tvorbe paralelných algoritmov. Algoritmy píšu v jednoduchom jazyku podobnom pseudokódu a ich efektivita sa meria z hľadiska asymptotickej práce a času, ktoré boli počas výpočtu spotrebované (paralelné algoritmy sú schopné vykonať viac jednotiek práce za jednu jednotku času). Súčasťou predmetu EPA sú domáce úlohy, pri ktorých študenti hľadajú optimálne riešenie problémov vo WT, čím získajú lepšie pochopenie a intuíciu pri práci s paralelnými algoritmami. Dobrým spôsobom, ako lepšie predviesť a pochopiť paralelný spôsob riešenia problémov je ich praktické riešenie, teda tvorba programov, ktoré ich riešia.

S týmto cieľom vznikla výučbová programátorská platforma WT* ([3]), založenej na platforme WT. Umožňuje tvorbu SIMD programov v emulovanom hardvérovonezávislom vysokoúrovňovom programovacom jazyku a prostredie pre tvorbu a spúšťanie paralelného kódu pre PRAM. Vykonávanie sa deje formou emulácie tohoto stroja a skompilovaný kód nie je nutne optimalizovaný ani nebeží paralelne. Keďže ide o učebný nástroj a nie je cieľom tvoriť produkčný kód, tak efektivitu nie je vôbec potrebné zohľadňovať. Pokým iné simulátory či jazyky sa snažili vytvoriť produkt, ktorý v praxi dokázal prejaviť zrýchlenie výpočtu vďaka paralelizácii a musel kvôli tomu zaviesť obmedzenia v niektorých oblastiach, WT* tieto požiadavky nemá, a tým pádom nemusí zavádzať žiadne obmedzenia. Programy vytvorené v jazyku WT* teda nebudú nikdy použité na tvorbu použiteľných aplikácii. Avšak tento jazyk vie študentov odbremeniť od ľubovoľných ťažkostí a poskytnúť im nástroj, ktorý umožňuje sústrediť sa na hlavné aspekty algoritmu a jeho jednoduché písanie na dostatočne vysokej úrovni. Programátorská platforma WT* obsahuje programovací jazyk, virtuálny PRAM stroj a súbor nástrojov potrebných pre tvorbu a vykonávanie paralelných programov zahŕňajúci kompilátor, emulátor, jednoduchý ladič a online vývojové prostredie. Daný programovací jazyk používa syntax podobnú jazyku C. Okrem iného pridáva príkaz `pardo(i:n)`, ktorý jemu prislúchajúci blok kódu vykoná paralelne na `n` vláknach s novými identifikátormi od `0` po `n-1` prístupnými pre každé vlákno prostredníctvom premennej `i`. Tento príkaz je možné vnárať rovnako, ako ktorýkoľvek iný riadiaci výraz, a teda pomocou neho a rekurzie vieme vytvoriť ľubovoľne rozvetvený paralelný výpočet.

Aktuálny stav síce umožňuje študentom tvoriť a simulovať paralelné programy, ale niektoré žiadané funkcie nie sú implementované a iné zase nie sú plne funkčné. Cieľom tejto práce je nadviazať na a rozšíriť výukové prostredie WT*. Hlavnými cieľmi je doplniť chýbajúcu alebo neúplnú funkcionalitu, vylepšiť aktuálne veľmi jednoduchý ladič, vytvoriť jednotné multiplatformové grafické vývojové prostredie a doplniť študijné

materiály vo forme programových implementácii relevantných algoritmov.

Kapitola 1

O probléme

V tejto kapitole predstavíme niekoľko prác vytvorených na našej univerzite, ktoré sa týkajú podobnej problematiky a ciele našej práce, ich motivácie a ozrejmíme spôsoby ich riešenia pre jednotlivé časti tejto práce.

1.1 Súčasný stav problematiky

S tvorbou nástrojov na prácu s paralelnými výpočtovými modelmi sa na našej univerzite zapodievalo už niekoľko ľudí. Jednou prácou, ktorá silno súvisí s tou našou je práca o simulovaní PRAM výpočtov [4], keďže išlo tiež o výučbový nástroj pre predmet EPA a na ktorý nadväzuje platforma WT*. Taktiež išlo o model vychádzajúci z WT, avšak išlo o trochu odlišný model PRAM, ktorý nebol striktne SIMD, čo sa nakoniec ukázalo menej vhodné na výučbu. Navyše, cieľom danej práce nebolo vytvoriť vývojové prostredie, ktoré by bolo pohodlné pre používateľa.

Druhou prácou pochádzajúcou z našej univerzity je práca, ktorá vytvorila vývojové prostredie pre experimentovanie s PRAM počítačmi [5]. Zahŕňala tvorbu interpretera a vývojového prostredia s editorom a ladičom (anglicky *debugger*) pozostávajúceho z krokovača a zobrazovača pamäte. Cieľ tejto práce je teda zhodný s naším cieľom. Kvôli veľkému rozsahu práce však ani jeden z jej aspektov nie je dostatočne vyvinutý na rozumné použitie. Výpočtový model je veľmi nízkoúrovňový a má niekoľko nepríjemných obmedzení, napríklad je potrebné vopred určiť počet procesorov a veľkosť výstupu. Programovací jazyk v tejto práci je založený na nízkoúrovňovom jazyku Assembler, čiže inštrukčná sada je obmedzená iba na tie najjednoduchšie operácie a aj základné konštrukty, ako cykly a funkcie musia byť riešené skokmi.

V našej práci budeme používať existujúci model, ktorý takéto obmedzenia nemá, takže môže používať dynamicky ľubovoľne veľa procesorov, veľkosť výstupu nebude musieť byť dopredu známa a programovací jazyk bude založený na C narozdiel od Assembleru. Naša práca k tejto existujúcej kostre pridáva užívateľsky prívetivé prostredie

s podporou viacerých operačných systémov a webového rozhrania. Na rozdiel od spomínanej práce bude editor podporovať základnú funkcionalitu potrebnú pre efektívnu tvorbu kódu, ako napríklad zvýrazňovanie syntaxe. Súčasťou spomínanej práce bol ladič, ktorý podporoval krokovanie a zobrazenie obsahu pamäte. Cieľom našej práce bude poskytnúť rozšírenú funkcionalitu s podmienenými bodmi prerušení (anglicky *conditional breakpoints*) a čo najlepšie zobrazenie pamäte a intuitívnou navigáciou stromu volaní.

1.2 Motivácia a návrh

Hlavnou formou využitia tohoto nástroja sa predpokladá byť výuka paralelných algoritmov na vysokej škole. Keďže tá prebieha iba niekoľko mesiacov, teda veľmi krátky čas, neočakáva sa, že by sa toto prostredie využívalo na tvorbu rozsiahlych projektov. Prioritou by mal byť krátky čas potrebný na počiatočnú inicializáciu prostredia a malá krivka učenia, ktoré nebudú mrhať obmedzený čas vyhradený na štúdium tohoto predmetu. Nie je dôležité implementovať funkcie, ktoré by síce spríjemnili život zopár používateľom, ale bez ktorých by sa väčšina z nich zaobišla.

V aktuálnom stave prostredia WT* sú študenti schopní písať programy, využívať veľmi základnú funkcionalitu ladenia a pozorovať výsledky a teoretické časové zrýchlenia svojich funkčných paralelných programov. Študentom sú síce poskytnuté spúšťateľné súbory kompilátora a emulátora, no neexistuje žiadne offline prostredie s podporou pre WT*, a teda jediným praktickým spôsobom, ako písať kód je využívať online vývojové prostredie. Online prostredie má však iné problémy, čiže aktuálne najlepším spôsobom využívania tejto platformy je nešikovné prehadzovanie zdrojového kódu medzi online a offline nástrojmi. Jedným z cieľov našej práce je vytvoriť vývojové prostredie s technológiami, ktoré umožňujú kompiláciu do *WebAssembly* pre použitie vo webovom prehliadači a offline prostredia použiteľného v rôznych hlavných operačných systémoch zahŕňajúcich Linux, Windows a MacOS.

Jedným z problémov WT* v jeho aktuálnom stave je chýbajúca podpora makra `#include` vo webovom prostredí. Aj keď sa vo všeobecnosti nejedná o kritickú funkcionalitu, v prípade tematiky paralelných algoritmov, ktoré sa vyučujú na Univerzite Komenského, ide o veľké znepríjemnenie vývoja. Výučba a algoritmy totiž silno nadväzujú na predchádzajúce algoritmy a procedúry. Táto funkcionalita taktiež veľmi dobre poukazuje na rozdiely vo vývoji natívnych a webových aplikácií. Pokým správa súborov v súborovom systéme počítača má ľahko použiteľne štandardizované rozhranie, v rámci vývoja webovej verzie si musíme rozmyslieť, ako poskytovať súbory štandardnej knižnice. Ďalším dôležitým aspektom práce s online prostredím je tiež udržiavanie prebiehajúceho pokroku tvoreného programu medzi reláciami (anglicky *session*) tak, aby

užívateľ mohol pokračovať v práci aj po opustení stránky.

Najmenej prepracovanou časťou WT* je proces ladenia programov. Jednoduchý ladič, ktorý je súčasťou tejto platformy, poskytuje iba veľmi nemotorný spôsob prerušovania a zobrazovania pamäte. Preto je veľká časť našej práce venovaná tejto oblasti. Proces ladenia sme zjednodušili grafickým vytváraním bodov prerušení a možnosťou následného krokovania behu programu rovnakým štýlom, aký je dostupný vo veľa iných známych prostrediach. Body prerušenia môžu byť nepodmienené alebo podmienené, čiže to, či nastane prerušenie programu môže byť voliteľne riadené zvoleným výrazom, ktorý môže využívať hodnoty premenných bežiaceho programu. V prípade krokovania podporujeme tri štandardné typy krokov: krok na rovnakej úrovni, krok o úroveň hlbšie a krok o úroveň vyššie.

Jadrom tejto práce je vytvorenie grafického vývojového prostredia podstatou rovnakého mnohým iným už existujúcim prostrediam, ako aj existujúcemu online vývojovému prostrediu WT*. Hlavnými prvkami sú textový editor so zvýrazňovaním syntaxe jazykov a podporou pre viacero súborov, správca súborov, oblasti pre vstup a výstup programu a hlavne priestor pre ladenie programu so vstavanou podporou pre ladič WT*. Keďže cieľom je vytvoriť produkt, ktorý už mnohokrát existuje (*VSCode*, *Sublime*, *Atom*, ...), je na mieste opýtať sa, prečo ako základ tejto práce nevyužiť niektorý z nich. Súčasťou požiadaviek je, aby dané vývojové prostredie bolo jednoduché, nenáročné a tiež, aby fungovalo natívne, ale aj vo webovom prehliadači. Tieto požiadavky ešte spĺňajú niektoré existujúce prostredia, napríklad *VSCode* [6]. V tomto prípade by bolo potrebné vytvoriť rozšírenie pre podporu jazyka a ladič WT*. Vytvoriť však celé vývojové prostredie s použitím existujúcich vhodných knižníc by nemalo byť omnoho náročnejšie a môže to znamenať jednoduchšiu údržbu. Pri budúcom vývoji našej práce môžeme zvážiť aj prístup formou rozšírení pre existujúce populárne vývojové prostredia. Aj napriek tomu, že sme sa rozhodli ísť cestou vlastného prostredia, táto práca nerozširuje existujúce WT* online vývojové prostredie. Keďže išlo iba o prototyp, ktorý by bolo nutné vo veľkej miere prepísať a jazyk, v ktorom bolo toto prostredie písané, nie je najlepšou voľbou pre naše požiadavky.

Ako už bolo spomínané, hlavným dôvodom, prečo je súčasťou tejto práce tvorba celého nového prostredia, je nutnosť pre priestor pre informácie ladiča. Štandardný ladič je totiž buď prispôsobený pre sledovanie a analýzu sekvenčného programu alebo paralelného programu len so zopár procesormi. Náš prípad použitia, teda SIMD program s bežným počtom procesorov rádovo v stovkách aj na malých vstupoch si pre prehľadné zobrazenie dát vyžaduje iný prístup. Tok výpočtu má kvôli vetveniu, volaniam funkcií a rekurzií tvar stromu, ktorého každý vrchol reprezentuje aktuálny stav lokálneho a globálneho kontextu. Takže kým pre programy sekvenčného charakteru by sme v niektorom okamihu skúmali stopu zásobníku, v paralelnom programe budeme navigovať strom. Konkrétnu funkcionalitu a spôsob, ako s ladičom vieme narábať bude popísaný

v neskoršej kapitole.

Je však možné, že niektorí používatelia nebudú využívať funkcionalitu ladiča integrovaného do nášho grafického prostredia alebo budú ochotní využívať ho iba dočasne počas procesu ladenia a primárne kód vyvíjať vo svojom preferovanom prostredí. V tom prípade je vhodné poskytnúť už spomínanú podporu niektorých vybraných prostredí tretích strán, aj keď iba limitovanú. V budúcnosti by sme chceli pridať podporu do aspoň zopár editorov.

Výučba paralelných algoritmov v každom bode silno nadväzuje na algoritmy a poznatky získané v predchádzajúcich prednáškach. Rozsah týchto algoritmov preto narastá do veľkých rozmerov už pri štandardne jednoduchých problémoch, ako napríklad triedenie. Náročnosť úloh, ktoré môžu študenti prakticky riešiť je tak obmedzená veľkosťou knižnice štandardných algoritmov, ktorú majú k dispozícii. Dostupnosť otestovaných, odladených, prehľadných a efektívnych algoritmov na riešenie štandardných problémov poskytne druhý pohľad na látku preberanú na hodinách a štartovný bod, ktorý vedia použiť pri riešení domácich úloh a cvičení. Môžu sa totiž zamerať na hľadanie a implementovanie riešení pre úlohy, ktoré sú im ešte neznáme na rozdiel od trápenia sa s implementáciou algoritmov z prednášok. Knižnica algoritmov preberaných na hodinách je žiaľ momentálne implementovaná do menej ako polovice rozsahu učiva.

Projekt WT* využíva rôzne technológie, no primárne je písaný v jazyku C. Pre našu prácu sme zvolili jazyk C++ ako kompromis medzi originálnym a modernejším jazykom s predpokladom dobrej kompatibility. Vývoj grafického prostredia taktiež silno využíva existujúce knižnice s voľne dostupným zdrojovým kódom na tvorbu softvéru užívateľského rozhrania. Ako základ využívame knižnicu *Dear ImGui* ([7]) a textový editor *ImGuiColorTextEdit* ([8]).

Kapitola 2

WT*

V tejto kapitole opíšeme projekt WT*, na ktorom stavia naša práca. Opíšeme v akom stave sa projekt nachádzal, čoho bol schopný a vysvetlíme interné fungovanie jeho dôležitých častí. Následne opíšeme zmeny, ktoré naša práca spravila v rámci tohoto projektu, s akými problémami sme sa museli vysporiadať a niekoľko problémov, ktoré ešte pretrvávajú.

2.1 Predstavenie

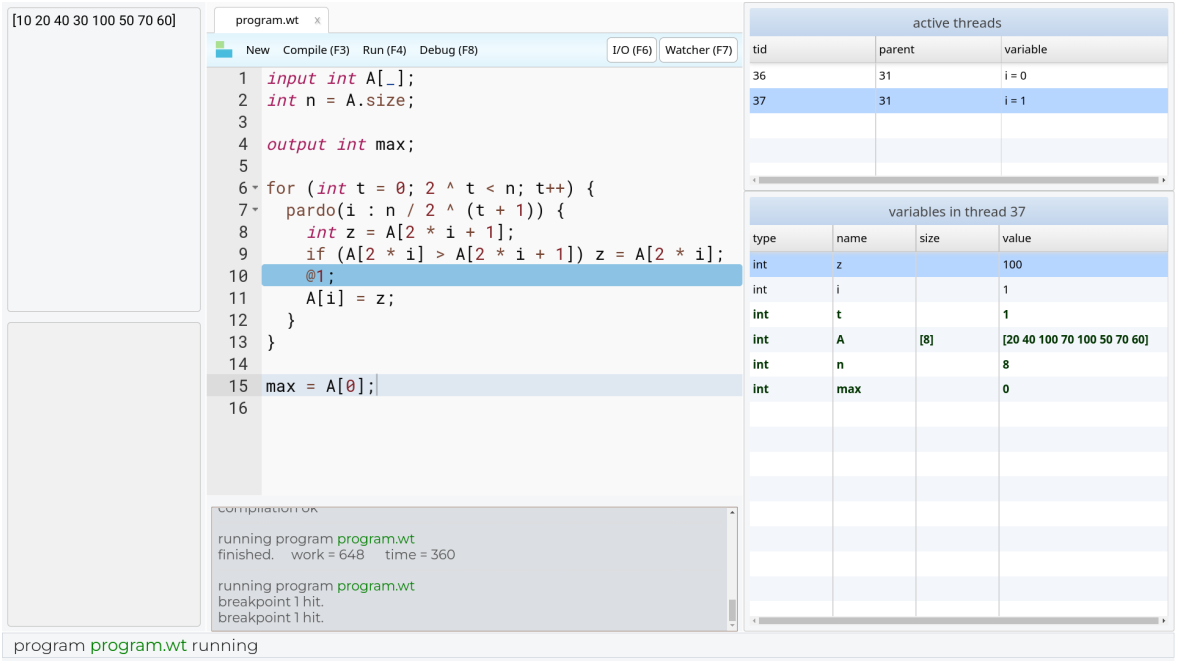
Projekt WT* je platforma slúžiaca na výučbu paralelných algoritmov. Programy sa píšu v programovacom jazyku podobnom zjednodušenému jazyku C rozšírenom o výraz `pardo` reprezentujúci paralelný cyklus a vykonávané sú na virtuálnom PRAM stroji. Výstupom projektu WT* bolo 5 programov – webové IDE a terminálové programy *wtc*, *wtrun*, *wtdb*, *wtdump*. Vývoj týchto nástrojov aj jazyka bol postupný a počas jeho priebehu sa častokrát zmenili požiadavky a prístup k riešeniu. Na veľa miestach v kóde projektu teda nájdeme časti, ktoré už sa nikde nepoužívajú alebo by boli riešené celkom iným spôsobom, keby boli písané nanovo.

Každý z terminálových programov bol vytvorený na jeden účel. Najdôležitejšími boli kompilátor *wtc* a emulátor skompilovaných programov *wtrun*. Ďalším terminálovým programom bol *wtdb* slúžiaci na ladenie programov. Zvláda spúšťať programy, zastavovať na bodoch prerušenia a vypisovať obsah premenných. Používa sa však ťažko a v prípade potreby ladiť kód je rýchlejšie a jednoduchšie použiť webové prostredie. Program *wtdump* slúžil na vypisovanie informácií použitých pri ladení a je celkom zastaralý a nahradený v prospech *wtdb*.

Webové vývojové prostredie kombinuje funkcionalitu všetkých doterajších programov, teda zvláda WT* programy kompilovať, spúšťať a ladiť z grafického prostredia, ktorého súčasťou je editor s podporou syntaxe WT* (obrázok 2.1).

Webové IDE a *wtdb* síce vedeli ladiť programy, ale iba vo veľmi základnej forme.

Funkcionalitu, čo majú v tejto práci prepíšeme upraveným spôsobom spolu s modernejšími taktikami v C++ a rozšírime do použiteľnejšej a rozsiahlejšej podoby.



Obr. 2.1: Webové prostredie WT* počas ladenia algoritmu na hľadanie maxima

2.2 Pôvodné fungovanie

Jazyk WT* vychádza z jazyka C a patrí do triedy jazykov *LALR(1)* [9]. Kompilácia kódu jazyka WT* vďaka tomu môže prebiehať v zopár štandardných krokoch a je možné použiť štandardné nástroje. Kód je najprv lexikálne analyzovaný programom vygenerovaným z popisu jazyka WT* nástrojom *Flex* [10] [11]. Tým z textového súboru získame postupnosť lexikálnych symbolov, teda reťazcov so známym významom ako kľúčové slová, operátory a čísla. Tie následne spracuje syntaktický analyzátor vygenerovaný nástrojom *Bison* [12], ktorého výsledkom bude abstraktný syntaktický strom (*AST*). Tento strom potom spracuje generátor kódu, ktorý na základe neho vypočíta relatívne adresy premenných a vygeneruje inštrukcie a pomocné dáta pre ladenie, ktoré sa môžu zapísať do výsledného binárneho súboru.

Pri spúšťaní programu sa daný binárny súbor načíta do pamäte a na základe údajov v ňom sa inicializuje virtuálny stroj. Virtuálny stroj naplní vstupné premenné na základe vstupného textu, ktorý mu užívateľ poskytne a následne emuluje inštrukcie získané z binárneho súboru.

Syntaktický strom slúži reprezentuje syntaktickú štruktú programu. Vrcholy reprezentujú prvky programu, napríklad príkazy a výrazy. Vnútornými vrcholmi môžu byť operátory, ktorých deti sú listové vrcholy im prislúchajúcim operandom, ako premenné,

literály, či iné výrazy. Vrcholy môžu reprezentovať aj zložitejšie štruktúry kódu, ako cykly, skôpy; [1] alebo funkcie, pričom koreň stromu reprezentuje celý program. V našej štruktúre syntaktického stromu má vrchol pre skôp niekoľko detí uložených formou spájaného zoznamu reprezentujúceho postupnosť príkazov, ktorý daný skôp obsahuje. Ďalej napríklad vrchol pre príkaz `for` má ako deti štyri vrcholy pre inicializačný príkaz, podmieňovací výraz, inkrementačný príkaz a blok jemu prislúchajúcemu kódu. Príklad kódu a jemu prislúchajúceho syntaktického stromu môžeme vidieť na obrázku 2.2.

Keďže model nášho stroja je SIMD, v prípade pozorovania ľubovoľného jedného vlákna sa dá naňho pozerať ako na sériový stroj. Virtuálny stroj je podobný štandardným zásobníkovým strojom [13]. Obsahuje dátové priestory ako priestor pre inštrukčný kód, operačný zásobník, akumulátorový zásobník a haldu pre dynamické dáta ako polia.

Inštrukcie berú operandy zo zásobníku a výsledok vracajú na zásobník. Aritmetické výpočty sa tým pádom vedia prirodzene počítať v postfixovej notácii. Premenné so statickou pamäťou, napríklad čísla, sú uložené na zásobníku. V prípade, že ich hodnotu chceme použiť vo výraze, vieme ju načítať inštrukciou `LDC`, ktorej parameter bude adresa. Ukladanie do premennej na zásobníku bude prebiehať obdobným spôsobom s použitím inštrukcie `STC`. Premenné s dynamickou veľkosťou pamäte ako polia majú na zásobníku uloženú adresu dát v halde. Pri vykonávaní inštrukcií sa kontroluje porušenie zvolenej politiky prístupu do pamäte, teda čítanie a zápis rovnakej adresy viacerými vláknami.

Každý blok inštrukcií tvoriaci jeden skôp je ohraničený inštrukciami `MEM_MARK` a `MEM_FREE`. Vykonanie inštrukcie `MEM_MARK` si vo virtuálnom stroji zapamätá pozíciu vrcholu zásobníka a haldy. Následné vykonanie inštrukcie `MEM_FREE` vyčistí zásobník a haldu po tieto značky. Lokálne premenné a zmeny, ktoré sa diali v danom skôpe sa teda po jeho skončení stratia a nasledujúci beh môže pokračovať s vrchom zásobníka v pôvodnom stave.

Pre podporu podprogramov si virtuálny stroj pamätá zásobník volacích rámcov (anglicky *call frame*). Pri volaní funkcie sa na tento zásobník pridá rámec. Tento rámec si mimo iného zapamätá aktuálnu veľkosť zásobníka ako svoju bázu. Lokálne premenné tela funkcie sa potom budú umiestňovať na vrch zásobníka, no adresy, s ktorými budú inštrukcie pracovať budú relatívne vzhľadom na bázu volacieho rámca, čím sa stratí viditeľnosť premenných z vonku funkcie. Po návrate z funkcie sa obnoví báza rámca, čím sa znova získa prístup k lokálnym premenným z predchádzajúceho rámca.

Náš virtuálny stroj sa od bežných sériových zásobníkových strojov líši tým, že má k dispozícii pomocný akumulátorový zásobník a haldu, čo však príliš neovplyvňuje

[1] Skôp, anglický preklad *scope*. Alternatívne slovenské preklady sú sféra viditeľnosti, rozsah platnosti alebo kontext. Ich použitie v texte je však ošemetné a v niektorých situáciach mätúce. Rozhodli sme sa teda vymyslieť nový slovenský preklad, a pevne dúfame, že sa ujme v informatickej vedeckej komunite

Algoritmus (2.1) WT* kód

```
1  input int x;
2  input int A[_];
3  output int sum;
4
5  pardo(i: A.size) {
6          A[i] *= x;
7  }
8
9  for(int i = 0; i < A.size; ++i) {
10         sum += A[i];
11 }
```

Algoritmus (2.2) čísla prisúchajúcich riadkov sú označené `r:od-do`; každý vrchol má identifikátor `id`; vrcholy v rámci skôpov sú uložené v spájanom zozname, a teda vrcholy ukazujú majú smerník na nasledovníka `dalsi`; skôpy majú smerník na rodičovksý skôp `rodic`

```
[ r:'0-12' AST_VRCHOL_SKOP id:'0' dalsi:nie rodic:'nie' ]
  [ r:'1-1' AST_VRCHOL_PREMENNA id:'13' nazov:'x' IO_TYP_VSTUPNA typ:'int' dalsi:'14' ]
  [ r:'2-2' AST_VRCHOL_PREMENNA id:'14' nazov:'A' IO_TYP_VSTUPNA typ:'int' pocet_dimenzii=1 dalsi:'15' ]
  [ r:'3-3' AST_VRCHOL_PREMENNA id:'15' nazov:'sum' IO_TYP_VYSTUPNA typ:'int' dalsi:'29' ]

  # pardo cyklus
  [ r:'5-8' AST_VRCHOL_PRIKAZ PRIKAZ_PARDO id:'29' dalsi:'45' ]
    [ r:'5-8' AST_VRCHOL_SKOP id:'16' dalsi:'nie' rodic:'0' ]
      [ r:'5-5' AST_VRCHOL_PREMENNA id:'17' nazov:'i' IO_TYP_ZIADNA typ:'int' dalsi:'20' ]
      [ r:'5-8' AST_VRCHOL_SKOP id:'20' dalsi:'nie' rodic:'16' ]
        [ r:'6-6' AST_VRCHOL_VYRAZ VYRAZ_BINARNY '*=' typ:'int' id:'24' dalsi:'nie' ]
            [ AST_VRCHOL_VYRAZ VYRAZ_PRISTUP_DO_POLA array:'A' typ:'int' id:'22' dalsi:'nie' ]
              [ AST_VRCHOL_VYRAZ VYRAZ_PREMENNA var:'i' typ:'int' id:'21' dalsi:'nie' ]
            [ AST_VRCHOL_VYRAZ VYRAZ_PREMENNA var:'x' typ:'int' id:'23' dalsi:'nie' ]
    [ AST_VRCHOL_VYRAZ VYRAZ_VELKOST variable:'A' typ:'int' id:'18' dalsi:'nie' ]
      [ AST_VRCHOL_VYRAZ VYRAZ_CISLO value:0 typ:'int' id:'19' dalsi:'nie' ]

  # for cyklus
  [ r:'10-12' AST_VRCHOL_PRIKAZ PRIKAZ_FOR id:'45' dalsi:'nie' ]
    [ r:'10-12' AST_VRCHOL_SKOP id:'30' dalsi:'nie' rodic:'0' ]
      # inicializacny prikaz
      [ AST_VRCHOL_PREMENNA id:'31' nazov:'i' IO_TYP_ZIADNA typ:'int' dalsi:'37' ]
        [ AST_VRCHOL_VYRAZ EXPR_INITIALIZER typ:'int' id:'33' dalsi:'nie' ]
            [ AST_VRCHOL_VYRAZ VYRAZ_CISLO value:0 typ:'int' id:'32' dalsi:'nie' ]
      # podmienovaci vyraz
      [ AST_VRCHOL_VYRAZ VYRAZ_BINARNY '<' typ:'int' id:'37' dalsi:'39' ]
        [ AST_VRCHOL_VYRAZ VYRAZ_PREMENNA var:'i' typ:'int' id:'34' dalsi:'nie' ]
        [ AST_VRCHOL_VYRAZ VYRAZ_VELKOST variable:'A' typ:'int' id:'35' dalsi:'nie' ]
            [ AST_VRCHOL_VYRAZ VYRAZ_CISLO value:0 typ:'int' id:'36' dalsi:'nie' ]
      # inkrementacny prikaz
      [ AST_VRCHOL_VYRAZ VYRAZ_PREFIX '++' typ:'int' id:'39' dalsi:'40' ]
        [ AST_VRCHOL_VYRAZ VYRAZ_PREMENNA var:'i' typ:'int' id:'38' dalsi:'nie' ]
      # blok prikazov
      [ AST_VRCHOL_SKOP id:'40' dalsi:'nie' rodic:'30' ]
        [ r:'11-11' AST_VRCHOL_VYRAZ VYRAZ_BINARNY '+=' typ:'int' id:'44' dalsi:'nie' ]
            [ AST_VRCHOL_VYRAZ VYRAZ_PREMENNA var:'sum' typ:'int' id:'41' dalsi:'nie' ]
            [ AST_VRCHOL_VYRAZ VYRAZ_PRISTUP_DO_POLA array:'A' typ:'int' id:'43' dalsi:'nie' ]
              [ AST_VRCHOL_VYRAZ VYRAZ_PREMENNA var:'i' typ:'int' id:'42' dalsi:'nie' ]
```

Obr. 2.2: Príklad kódu a jemu prislúchajúcemu syntaktickému stromu

podstatu jeho fungovania.

Tento model sa začne komplikovať, keď zavedieme viaceré vlákna. V prípade jedného vlákna s doterajším prístupom nebudeme mať žiaden problém. V našich programoch sa však môže nachádzať riadiaci príkaz `pardo`. Inštrukčný kód jeho tela sa začína inštrukciou `FORK`, ktorá tok programu zduplikuje do niekoľkých vlákien. Tento paralelný cyklus sa končí vykonaním inštrukcie `JOIN`, po ktorej očakávame, že sa počet vlákien vráti do stavu pred jeho spustením. Virtuálny stroj si teda pamätá zásobník skupín vlákien. V prípade vykonania `FORK` naň pridáme skupinu vlákien pozostávajúcu z rozmnožených vlákien a v prípade vykonania `JOIN` túto skupinu zo zásobníku odstránime. Aktuálnu skupinu vlákien tak vždy nájdeme na vrchu zásobníku vlákien.

Zásobník vlákien využijeme aj v iných situáciách. Pozrime sa na fungovanie riadiaceho výrazu `if-else`. Keďže všetky vlákna vykonávajú vždy tú istú inštrukciu, avšak vlákna môžu obsahovať rôzne dáta, je bežné, že časť vlákien spĺňa podmienku `if` a zvyšok nie. V inštrukčnom kóde budeme mať postupnosť inštrukcií, ktorú má vykonať prvá skupina nasledovanú postupnosťou inštrukcií určenú pre druhú skupinu. Vyhodnotenie `if(x) { A } else { B }` teda pre každé vlákno vyhodnotí výraz `x` a jeho výsledok umiestni na zásobník daného vlákna. Inštrukcia `SPLIT` na základe tejto hodnoty rozdelí vlákna do dvoch skupín. Do prvej budú patriť vlákna, pre ktoré bol výraz pravdivý, teda tie, ktoré budú vykonávať blok `A`, a do druhej vlákna, pre ktoré bol nepravdivý, a teda budú vykonávať kód patriaci do `B`. Obe tieto skupiny pridáme na zásobník skupín vlákien tak, že na vrchu nájdeme skupinu vlákien pre blok `B`. Na konci oboch postupností inštrukcií kódov blokov `A` a `B` pridáme inštrukciu `JOIN`. Obe skupiny teda vykonajú iba svoje prislúchajúce inštrukcie a následne sa odstránia zo zásobníku.

Vlákna môžu byť navyše aktívne alebo neaktívne. V prípade, že niektoré vlákno vo funkcii použije príkaz `return`, nemalo by vykonávať žiadne ďalšie inštrukcie. Dané vlákno je iba označené za neaktívne a v prípade, že funkcia mala návratovú hodnotu sa tento výsledok ponechá na zásobníku odkiaľ ho volajúci kód po návrate všetkých vlákien môže prečítať.

Pozrime sa na nasledujúci kód:

```
int k = 10;
int A[k];
pardo(i: k)
    A[i] *= k;
```

Všetky vlákna musia mať oddelený prístup ku svojej lokálnej premennej `i`, avšak spoločný prístup k premenným `k` na zásobníku a `A` na halde. Tieto premenné môžu byť globálne, ale aj lokálne a potrebujeme ich vedieť čítať aj zapisovať. Aby sme zachovali separáciu lokálnych premenných medzi vláknami, každé vlákno musí mať svoj vlastný

zásobník a haldu. Na to, aby sme umožnili vláknam prístup k rovnakým premenným z vyšších skôpov, pri ktorom sa zmeny vykonané v jednom vlákne prejavia aj v ostatných, vlákna musia zdieľať časť zásobníku pred aplikovaním inštrukcie `FORK`. Po vykonaní inštrukcie `JOIN` a odstránení skupiny vlákien chceme, aby tieto zmeny zostali zachované. Navyše pri rozdelení toku na stovky vlákien by bolo kopírovanie zásobníku veľmi neefektívne.

Virtuálny stroj preto využíva takzvaný kaktusový zásobník (obrázok 2.3). Každé vlákno si pamätá iba segment zásobníku, ktorý ma rozdielny od ostatných vlákien a referenciu na ich spoločný zásobník. Ten samozrejme taktiež môže byť iba segment, ktorý si pamätá referenciu na iný spoločný zásobník, čím vzniká štruktúra vizuálne podobná kaktusu. Postupnosť referencií na rodičovské segmenty tvorí jednosmerné spájaný zoznam, ktorého pomyselným spojením získame súvislý zásobník pre dané vlákno. V prípade, že chce vlákno pristupovať k premennej, postupne prezerá tento spájaný zoznam pokým nenájde segment obsahujúci požadovanú adresu.

Jazyk WT* podporuje dva typy bodov prerušenia dostupných ako príkazy v kóde. Prvý so syntaxou `@id(expr);` pre bod prerušenia s identifikačným číslom `id` a podmienený výrazom `expr`. Druhý nepodmienený bod prerušenia so syntaxou `@id;` ekvivalentný príkazu `@id(1);`. V inštrukčnom kóde sa vyhodnotí výraz `expr`, ktorý ponechá svoj výsledok ako hodnotu na zásobníku. Následne sa vykoná inštrukcia `BREAK`, ktorá prečíta danú hodnotu zo zásobníku. V prípade, že sa v aspoň jednom aktívnom vlákne nachádzala nenulová hodnota, vykoná sa prerušenie vyhodnocovania a vráti sa kontrola ladiacemu programu, ktorý ho spustil.

2.3 Problémy a riešenia

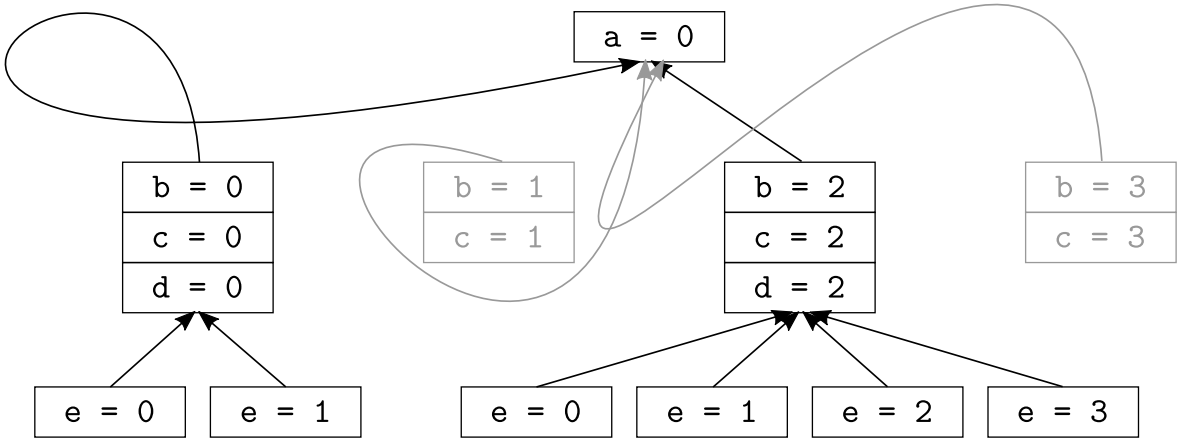
Pred tým, ako sme mohli začať s prácou na dynamických bodoch prerušenia a vylepšenom ladiči sme najprv museli vyriešiť niekoľko problémov v projekte WT*.

Mnoho dát využívaných počas krokov kompilácie bolo deklarovaných ako globálne premenné. Toto okrem toho, že ide o zlú prax spôsobovalo aj problémy pri spôsobe kompilácie, ktorý sme neskôr chceli využívať. Pri implementácii kompilácie podmienok bodov prerušení (ktorej detailný postup bude vysvetlený neskôr) sme potrebovali spúšťať kompiláciu zo zmutovaného stavu, čiže zo stavu s aplikovanými zmenami oproti počiatočnej konfigurácii. Pôvodný spôsob mal v tomto problém, keďže získavanie a nastavovanie všetkých globálnych stavových premenných bolo komplikované a náchylné na chyby. Kroky syntaktickej a lexikálnej analýzy a generovanie kódu boli prepísané, aby sa menej spoliehali na globálne premenné.

Pri syntaktickej analýze sa základné typy ako napríklad `int` a `void` taktiež pamätali ako globálne premenné. Počas analýzy sa premenným a výrazom určovali typy, ktoré

```
void f(int i) {
    int c = i;
    if(i % 2 == 0) {
        int d = i;
        pardo(j: i + 2) {
            int e = j;
            @1; // program zastavi v tomto bode
        }
    }
}

int a = 0;
pardo(i: 4) {
    int b = i;
    f(i);
}
```



Obr. 2.3: Príklad kódu a jemu prislúchajúceho kaktusového (resp. špagetového) zásobníka. Listy stromu reprezentujú vlákna, pričom v bode prerušenia je šesť vlákien aktívnych a dve zvýraznené šedou neaktívne.

boli použité na typovú kontrolu. Tieto typy boli uložené ako adresa smerníku na príslušnú globálnu premennú typu. Pri rekompilácií, ktorú sme spúšťali na zmutovanom stave, však nastávali problémy, keď sa tieto objekty v rôznych okolnostiach vygenerovali nanovo alebo vyčistili. Riešením bolo zbaviť sa týchto globálnych premenných a namiesto toho ich naviazať na im prislúchajúci syntaktický strom. Navyše sme porovnávanie typov prepísali z porovnávania adries na porovnávanie hodnôt typov, čím určovanie fungovalo správne aj pre premenné pochádzajúce z rôznych behov kompilácii.

Štruktúra syntaktického stromu neobsahovala koreňový vrchol, ale iba objekt koreňového skôpu, ktorý už obsahoval všetky zvyšné vrcholy. Tento prístup bol zvolený z dôvodu uľahčenia implementácie v rámci krokov kompilácie. Neskôr však táto nekonzistencia znateľne komplikovala traverzovanie stromu, keďže bolo potrebné všade ošetrovať špeciálny prípad koreňa. Štruktúru sme teda zmenili tak, aby bol koreňom stromu *vrchol* so skôpom.

Podobne aj vrcholy syntaktického stromu reprezentujúce riadiace výrazy, ako podmienky alebo cykly, mali a stále majú podivnú štruktúru. Každý vrchol v totiž tejto štruktúre obsahuje dve deti. Jednotlivých typy výrazov si však vyžadujú jeden až štyri deti, a tak je obmedzenie na práve dve deti veľmi nevýhodné. Pôvodne bola táto štruktúra taktiež myslená pre zjednodušenie, avšak nielenže nič neuľahčila, ale výslovne implementáciu na mnohých miestach skomplikovala. Keďže zmena štruktúry by si vyžadovala rozsiahlejší prepis syntaktického analyzátora, tento problém zatiaľ nebol vyriešený.

Okrem toho si syntaktický analyzátor vyžaduje ďalší prepis v oblasti riadiacich príkazov. Výrazy `if`, `while` a `do-while` totiž nesprávne generujú pomocné informácie o mapovaní týchto príkazov na rozsah kódu. To má za následok, že dynamické body prerušenia sa nedajú umiestniť na riadky týchto príkazov.

Ďalším rozhodnutím, ktoré veľmi mierne zjednodušilo implementáciu na zopár miestach bolo rozhodnutie pri inštrukcií `SPLIT` vykonávať najprv skupinu vlákien s nulovou hodnotou pred skupinou vlákien s nenulovou hodnotou. Keďže podmienené vykonávanie, a tým pádom aj inštrukcia `SPLIT`, sa používajú v cykloch, vďaka tomuto je postupnosť inštrukcií zabezpečujúca cyklenie o kúsok jednoduchšia. Okrem toho to však malo za následok, že v príkaze `if-else` sa vykonávala `else` vetva pred `if` vetvou. Toto bol problém, keďže pri krokovaní počas ladenia by sa mätúco vykonávali riadky kódu v inom poradí, než v ktorom sú napísané v zdrojovom kóde. Poradie vykonávania vetiev sme vymenili.

Pôvodná syntax bodov prerušenia `@id(expr);` bola neštandardná a zbytočne využívala číselný identifikátor, ktorý bol taktiež aj umiestnený do kódu. Tento identifikátor je tak trochu zbytočný a, ako neskôr vysvetlíme, dynamické body prerušenia preňho v inštrukčnom kóde nebudú mať miesto. Číselný identifikátor sme teda odstránili aj zo statických bodov prerušenia a syntax zmenili na `BRK(expr);`.

Projekt využíval kód manipulujúci so súborovými cestami. Ten bol však úplne nefunkčný a nahradili sme ho voľne dostupnou populárnou knižnicou. Okrem toho sme opravili viacero implementačných chýb, ako preklepy spôsobujúce chybné správanie, prístupy mimo poľa, neinicializované premenné, unikajúcu pamäť, ...

2.4 Implementácia dynamických bodov prerušenia

Či už pred alebo počas behu programu, užívatelia majú byť schopní umiestniť bod prerušenia. Na rozdiel od doterajších statických bodov prerušenia umiestnených priamo v zdrojovom kóde požadujeme, aby sa informácia nachádzala mimo zdrojového kódu. Pre jednoduchosť sa body prerušenia mapujú na riadky kódu. Teda v prípade, že sa na jednom riadku vyskytuje viacero príkazov je možné umiestniť bod prerušenia iba na prvý z nich. Naopak, ak sa jeden príkaz rozpína cez viacero riadkov, mohlo by byť možné umiestniť bod prerušenia na jeho jednotlivé časti. Aktuálna implementácia metadát mapovania riadkov kódu na inštrukcie toto neumožňuje, avšak je možné, že v budúcnosti bude táto funkcionalita prirobená.

Jedným spôsobom, ako pridať dynamické body prerušenia je pamätať si pre inštrukcie, či sa má na nich beh programu zastaviť. Keďže však existuje inštrukcia `BREAK` a tento spôsob bol jednoduchší na implementáciu, body prerušenia pridávame modifikáciou inštrukčného kódu virtuálneho stroja. Vkladanie inštrukcií doprostred kódu nie je prípustné, keďže by sa mohli pokaziť adresy skokov. Namiesto toho nahradíme inštrukciu na vhodnom mieste inštrukciou `BREAK`. Je dôležité, aby sme touto zmenou neovplyvnili fungovanie programu. Bežné ladiče volia prístup, kedy si danú nahradzovanú inštrukciu zapamätajú a po obnovení z prerušenia ju dodatočne vykonajú. Keďže my máme kontrolu nad kompiláciou a nie je pre nás dôležitá najvyššia efektivita, zvolili sme iný prístup. Nie je totiž problém nahradiť inštrukciu inou, ak pôvodná nič nerobila. Pri kompilácii si teda zaručíme, že každé miesto kam sme schopní umiestniť dynamický bod prerušenia obsahuje predpripravenú inštrukciu `NOOP`, ktorá nemá sama o sebe žiaden efekt na beh programu.

Podmienky dynamických bodov prerušenia by mohli fungovať celkom ľubovoľným spôsobom, avšak bolo by príjemné, keby boli konzistentné so zvyškom projektu, čiže fungovali rovnako ako ich statické náprotivky. Mali by obsahovať určitý kód, ktorého vyhodnotením vznikne hodnota na zásobníku. Rovnako by bolo vhodné, aby bol daný výraz písaný v jazyku WT*. Keďže využívame rovnaký jazyk, pokúsime sa taktiež čo najviac využiť už existujúce nástroje, konkrétne už existujúci kompilátor. Výrazy, ktoré budú podporované dynamickými bodmi prerušenia potom navyše nebudú musieť byť len jednoduché výrazy, ale takmer ľubovoľný kód jazyka WT* vrátane cyklov a deklarácii premenných. Keďže podmienka dynamického bodu prerušenia bude môcť

obsahovať viacero príkazov, na rozdiel od statických bodov prerušenia sa budú tieto príkazy vykonávať v novom skôpe umiestnenom vnútri skôpu obsahujúceho bod prerušenia. Vďaka tomuto mimo iného nenarazíme na konflikty pri deklaráciách premenných a zásobník sa po vyhodnotení podmienky vráti do pôvodného stavu.

Jedným dôležitým problémom je, kam umiestniť inštrukčný kód skompilovanej podmienky. Jednou možnosťou by bolo vložiť ho pred prislúchajúcu inštrukciu `BREAK`, lenže, ako sme už spomínali, vkladanie inštrukcií doprostred kódu nie je vhodný spôsob. Alternatívne si ho vieme pamätať v separátnej štruktúre a pri výskyte inštrukcie `BREAK` začať vyhodnocovať inštrukcie z tejto štruktúry. V prípade, že v podmienke využívame volania funkcii by sme tak museli skákať medzi hlavným inštrukčným kódom a danou štruktúrou, čo je zbytočne komplikované. Namiesto toho tento kód pridáme na koniec inštrukčného kódu. Aj pridávanie na koniec by mohlo byť potenciálne ťažké v prípade, že by sa kód nachádzal primárne na disku. Avšak implementácia virtuálneho stroja si celý kód drží v pamäti vo vnútri dynamického poľa, čiže jeho rozšírenie nie je náročné. V prípade, že sa rozhodneme bod prerušenia odstrániť alebo jeho podmienku upraviť, inštrukčný kód pôvodnej podmienky jednoducho necháme na konci, keďže jeho zmazanie neprináša žiaden rozumný benefit. V prípade, že pôvodný kód obsahoval skoky za koniec inštrukčného priestoru budú tieto skoky teraz potenciálne skákať na inštrukcie podmienky bodu prerušenia, čo bude mať celkom určite za následok pád emulovaného programu. Toto správanie je celkom vporiadku, keďže skoky za koniec inštrukčného priestoru nemajú definované správanie.

V momente, keď narazíme na inštrukciu `BREAK` a overíme, že ide o dynamický bod prerušenia s podmienkou, nastavíme inštrukčný ukazovateľ na začiatok inštrukčného kódu prislúchajúcej podmienky. Keďže takýchto podmienok bude na konci inštrukčného kódu potenciálne za sebou niekoľko, potrebujeme ich od seba oddeliť. To je možné spraviť viacerými spôsobmi. My sme zvolili implementačné najjednoduchší, ktorý tento problém rieši novopridanou inštrukciou `BREAKOUT`. Táto inštrukcia bude mať viacero účelov. Kód podmienky vždy vykonávame v kontexte nového skôpu. S týmto nám ale vzniká problém, keďže ukončením podmienky, a teda ukončením skôpu, a teda vyčistením zásobníku stratíme informáciu o výsledku podmienky. Preto pred poslednú inštrukciu podmienky, ktorou je vždy inštrukcia `MEM_FREE` umiestňujeme inštrukciu `BREAKOUT`. Keď narazíme na túto inštrukciu, prečítame hodnotu zo zásobníku, vykonáme poslednú inštrukciu, ktorou sa vyčistí zásobník, hodnotu vrátime na zásobník a inštrukčný ukazovateľ vrátime na miesto, na ktorom bol pred vyhodnocovaním podmienky. Vyriešili sme tým problém oddelenia inštrukcií podmienok a zanechanie výslednej hodnoty na zásobníku.

Zanechanie výslednej hodnoty na zásobníku má aj ďalší problém. Pôvodný návrh a implementácia uvažovala podmienku, ktorá obsahuje aspoň jeden výraz (napríklad `i % 2 == 0`), ktorého výsledok sa ponechá na zásobníku. Problém však bol, že štandardné

správanie kompilátora je pre výrazy, ktorých výsledok sa nepoužíva, daný výsledok následne zo zásobníku zmazať, keďže predpoklad bol, že je zbytočný. Toto sa samozrejme dá zmeniť, avšak nesie to so sebou niekoľko problémov. Keďže kompilátor nevie výsledok ktorého výrazu má a ktorého nemá nechať, jediný spôsob je nechať všetky výsledky. Tento prístup nie je optimálny, keďže zásobník sa v tomto prípade postupne plní nechcenými hodnotami. Na funkčnosť programu to však nemá veľký vplyv a tento problém sa dá zmierniť, ak kompilátor informujeme o tom, či generuje kód pre podmienku bodu prerušenia, pričom ho necháme ponechávať výsledky na zásobníku iba vtedy. Ďalší problém nastane, ak užívateľ zadá podmienku, ktorá nenechá na zásobníku hodnotu. Tento posledný problém sa nedá efektívne kontrolovať.

Podmienka bodu prerušenia sa však veľmi podobá na volanie funkcie s návratovou hodnotou `int`. Dôvod prečo nemôžeme priamo použiť volanie funkcie je, že podmienky bodu prerušenia majú mať prístup ku premenným v kontexte okolia v akom sa nachádzal ich príslušný bod prerušenia. Volaním funkcie by sa vytvoril volací rámec a tento prístup stratil. Vieme však využiť hybridný spôsob. Príkaz `return` totiž pre tento problém funguje veľmi výhodne. V aktívnom vlákne, ktoré vykoná príkaz `return val;` sa na zásobník uloží hodnota `val` a dané vlákno sa označí za neaktívne. Pri zahájení vyhodnocovania podmienky bodu prerušenia takto vieme vytvoriť na zásobníku skupín vlákien novú skupinu obsahujúcu iba aktuálne aktívne vlákna a zahájiť vyhodnocovanie inštrukcií podmienky. Keď narazíme na inštrukciu `BREAKOUT`, vieme, že všetky neaktívne vlákna museli vykonať príkaz `return`, a tým pádom obsahujú požadovanú hodnotu na zásobníku. O ostatných vláknach jednoducho predpokladáme, že sa vyhodnotili na `0`.

Ak chceme bod prerušenia dočasne vypnúť, stačí jemu prislúchajúcu inštrukciu `BREAK` nahradiť naspäť inštrukciou `NOOP`.

Kapitola 3

Ladič

V tejto kapitole si predstavíme schopnosti ladiča a detaily ich implementácie.

Úlohou ladiča bude sprostredkovávať rozhranie medzi užívateľom a virtuálnym strojom. Na základe dát a metód virtuálneho stroja bude vykonávať akcie užívateľa. Na spustenie programu je potrebné zvoliť súbor, ktorý obsahuje zdrojový kód programu, tento kód skompilovať do binárneho súboru, poskytnúť text, ktorý bude slúžiť ako vstup programu a následne výsledný binárny súbor spustiť. So spusteným programom je potom možné manipulovať. Manipulovať je možné ovplyvňovaním behu programu, konkrétne pridávaním a manipulovaním s bodmi zastavenia a pokračovaním, krokovaním, reštartovaním alebo ukončením behu. Počas celého behu programu je možné pozorovať hodnoty premenných v rámci všetkých vlákien a rámcov volaní. Na sprostredkovanie týchto akcií ladič poskytuje rozhranie, ktoré využívajú grafické moduly ladiča (sekcia 4.3.4).

3.1 Kompilácia podmienok dynamických bodov prerušenia

Na kompiláciu podmienok bodov prerušenia sa pokúsime v čo najväčšej miere využiť existujúci kód kompilátora WT*. Jednotlivé kroky budeme musieť mierne upraviť. Kompilácia sa skladá z krokov lexikálnej analýzy tvoriacej postupnosť symbolov, syntaktickej analýzy tvoriacej syntaktický strom a generovania inštrukčného kódu.

Krok lexikálnej analýzy nemusíme meniť.

Syntaktická analýza tvorí syntaktický strom a zároveň kontroluje správnosť syntaxe a počíta pomocné informácie ako adresy premenných. V tomto kroku je teda potrebné, aby bol kód podmienky analyzovaný v kontexte kódu, do ktorého bude vkladaný. Chceme docieliť, aby sme po zanalyzovaní kódu podmienky získali rovnaký alebo aspoň dostatočne podobný strom ako v prípade, že by sme analyzovali pôvodný kód programu s podmienkou bodu prerušenia vloženou na jej príslušné miesto. Tento strom

sa od syntaktického stromu pre pôvodný kód líši iba veľmi málo. Keďže podmienka bodu prerušenia bude vždy obalená novým skôpom, syntaktický strom s podmienkou sa od pôvodného bude líšiť iba pridaním jedného vrcholu reprezentujúceho daný naplnený skôp. Syntaktická analýza tvorí syntaktický strom postupne a pracuje iba s lokálnym kontextom stromu. V prípade, že analyzuje jednoduché príkazy, prislúchajúce vytvorené vrcholy syntaktického stromu pridáva ako deti do pracovného vrcholu reprezentujúceho aktuálny skôp. Pri zahájení analýzy podmienky bodu prerušenia analýzu nezačneme s čistým stromom v globálnom skôpe. Namiesto toho začneme s už existujúcim hotovým úplným stromom pre kód programu, ktorému na správne miesto pridáme prázdny vrchol typu skôp a ten aj zvolíme ako aktuálny pracovný skôp. Počas analýzy sa potom všetky nové vrcholy budú vytvárať v rámci tohoto jedného skôpu. Po skončení vieme tento vrchol zo stromu vytrhnúť a strom vrátiť do pôvodného stavu pre budúce použitie.

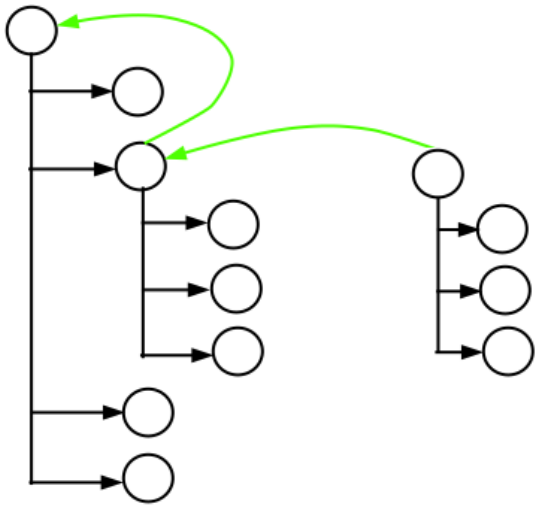
Generovanie kódu je implementované rekurzívne na syntaktickom strome, čiže existuje funkcia implementujúca generovanie kódu pre vrchol skôpu a na získanie kódu pre vrchol reprezentujúci podmienku bodu prerušenia ju stačí zavolať.

3.1.1 Vkladanie vrcholu podmienky do syntaktického stromu

Pre začatie syntaktickej analýzy podmienky bodu prerušenia je potrebné do syntaktického stromu pridať vrchol prázdného skôpu. Ten bude slúžiť ako kontajner pre vrcholy, ktoré počas analýzy vzniknú. Tento vrchol je potrebné vložiť na správne miesto stromu. Vrcholy stromu obsahujú pomocnú informáciu, ktorá ich mapuje na rozsahy kódu WT* a rozsahy vygenerovaného inštrukčného kódu. Vďaka týmto informáciám sme schopní nájsť vrchol prislúchajúci príkazu, na ktorý bol umiestnený bod prerušenia. Prázdny vrchol skôpu by bolo vhodné potom umiestniť pred tento vrchol na rovnakú úroveň. Spôsob, akým je kompilácia implementovaná však má za následok, že umiestnenie v rámci skôpu nemá na priebeh kompilácie žiaden vplyv. Nie je teda nutné prázdny vrchol vkladať doprostred spájaného zoznamu detí skôpu obsahujúceho bod zlomu. Navyše, keďže kompilujeme iba kód podmienky a nie kód programu, syntaktický strom nemusí dokonca obsahovať žiadne hrany smerom do vrcholu podmienky. Na napojenie prázdneho vrcholu teda postačuje nastaviť rodičovský skôp vrcholu (obrázok 3.1).

3.1.2 Počítanie bázovej adresy pre skôp

Každá premenná má pridelenú adresu na zásobníku. Globálne premenné používajú absolútne adresy, lokálne premenné adresy relatívne k bázovej adrese volacieho rámca. Tieto adresy sú premenným pridelené počas kompilácie a rovnako ich budeme musieť prideliť aj premenným deklarovaným v podmienke bodu prerušenia. Výpočet adries



Obr. 3.1: Jednosmerne zavesený vrchol skôpu podmienky. Vrchol podmienky je na syntaktický strom pripojený iba hranou na rodičovský skôp.

prebieha rekurzívne na syntaktickom strome. Každej premennej je pridelená najmenšia možná adresa, pričom globálne premenné majú adresy menšie ako ľubovoľná lokálna premenná. Rovnako aj pre premenné deklarované v podmienke bodu prerušenia existuje najmenšia možná použiteľná adresa. Keďže ale vrchol podmienky do stromu neumiestňujeme na presné miesto, nie je možné zistiť aká presne je. To však nie je problém, keďže stačí, aby mali všetky viditeľné premenné v každom momente rozličné adresy, a pritom nemusia byť nutne najmenšie možné. Každý skôp obsahuje iba konečne veľa premenných, je teda o každom skôpe možné povedať, aký rozsah adries je mu pridelený. Konflikt nemôže nastať, ak premenným z podmienky priradíme adresy väčšie ako tie použité v skôpe obsahujúcom daný bod prerušenia. Konflikt nenastane ani pri pridávaní ďalších bodov prerušenia do toho istého skôpu, keďže pri vyhodnocovaní ľubovoľnej podmienky bodu prerušenia budú už všetky predchádzajúce podmienky skončené, ich skôpy opustené a zásobník vyčistený. Pri kompilácii si pre každý vrchol reprezentujúci skôp zapamätáme rozsah v ňom použitých premenných a tieto informácie použijeme pri výpočte adries premenných podmienok bodov prerušení.

3.2 Krokovanie

Pri ladení je bežné sledovať beh programu krok po kroku. Toto sa štandardne vykonáva tromi akciami – *krok dnu* (*step in*), *krok cez* (*step over*) a *krok von* (*step out*). Najzákladnejšou funkciou krokovania je *krok dnu*, ktorá jednoducho vykonáva inštrukcie pokým nenarazíme na inštrukcie reprezentujúce nasledujúci príkaz v zdrojovom kóde. *Krok cez* a *krok von* fungujú na podobnom princípe v tom, že zastavia na inštrukcií nasledujúceho príkazu, avšak nie nutne zastavia na najbližšom príkaze. Kým *krok dnu*

pri volaní funkcie vstúpi dovnútra funkcie a zastaví na prvom príkaze jej tela, *krok cez* a *krok von* prejdú cez celé telo danej volanej funkcie bez zastavenia. Keď si volanie funkcie predstavíme ako krok o úroveň nižšie a návrat z funkcie ako krok o úroveň vyššie, potom *krok cez* zastaví na nasledujúcom príkaze, ktorý je na úrovni väčšej alebo rovnej úrovni aktuálneho príkazu a *krok von* na úrovni ostro väčšej. Prakticky to približne znamená, že *krok cez* sa v rámci zdrojového kódu presunie na ďalší riadok a *krok von* dokončí príkazy aktuálnej funkcie. Vieme teda povedať, že *krok cez* a *krok von* pozostávajú z niekoľko akcií *krok dnu*. Na implementáciu krokovania teda treba najprv vyriešiť ako vykonať *krok dnu*.

Správanie krokovania používajúce *krok dnu* je podobné, ako keby používateľ alebo ladič umiestnil na každý riadok programu bod prerušenia. Umiestniť bod prerušenia na každý riadok je však neefektívne rovnako ručne ako aj automatizovane, keďže väčšina takýchto bodov prerušenia by bola s veľkou pravdepodobnosťou nevyužitá, keď sa užívateľ rozhodne pre akciu *krok von* alebo pokračovanie behu programu bez krokovania. Lepším riešením je umiestňovať bod prerušenia vždy iba na najbližší nasledujúci riadok alebo alternatívne ani neumiestňovať žiaden bod prerušenia, ale v prípade, že užívateľ zvolil akciu krokovania pri vykonávaní inštrukcií kontrolovať, či je aktuálna inštrukcia, ktorú ideme vykonať prvou inštrukciou skupiny prislúchajúcej niektorému príkazu zdrojového kódu. Tu napríklad môžeme vidieť rozdiel medzi krokovaním a bodmi prerušenia, keďže v našej reprezentácii bodov prerušenia jednému riadku kódu prislúcha iba jeden bod prerušenia, zatiaľčo pri krokovani dokážeme rozlíšiť viacero príkazov na jednom riadku do viacerých krokov. Navyše je aktuálna reprezentácia a spôsob pridávania bodov prerušenia zbytočne komplikovaná pre porovnateľne jednoduchšiu aktivitu krokovania, ktorá nevyžaduje podporu pre podmienky. Keďže kompilátor WT* generuje pomocné dáta ako mapovanie rozsahov inštrukcií na riadky kódu, posledné navrhnuté riešenie kontroly pri vykonávaní je teda potenciálne uskutočniteľné z aktuálne dostupných informácií. Túto kontrolu by sme mohli vykonávať externe v programe ladiča alebo interne v rámci virtuálneho stroja. Druhá z týchto dvoch možnosti je lepšia, keďže virtuálny stroj už má podporu pre vykonávanie inštrukcií pokým nenarazí na bod zastavenia, čo je veľmi podobný úkon a znamenalo by to duplikáciu tejto funkcionality v programe ladiča.

Samozrejme, keďže túto kontrolu musíme vykonávať po každej jednej inštrukcií, je potrebné, aby prebiehala efektívne. Najrýchlejším riešením je pamätať si, na ktorých inštrukciách je potrebné zastať, čím by sa kontrola zredukovala na prečítanie hodnoty. Túto informáciu je možné predpočítať pri vytvorení virtuálneho stroja. My sme namiesto toho zvolili prístup, kedy túto informáciu nepredpočítavame pri vytváraní virtuálneho stroja, no využijeme, že táto informácia je už známa pri kompilácii. Zakomponovali sme ich do inštrukčného kódu binárneho súboru programu formou dvoch inštrukcií `STEP_IN` a `STEP_OUT` reprezentujúcich začiatok a koniec skupiny inštrukcií

pre príkaz. `STEP_IN` reprezentuje miesta, na ktorých treba zastaviť pri akciách *krok dnu* a *krok cez*, zatiaľčo `STEP_OUT` miesta, kde treba zastaviť pri akcii *krok von*. Dôvod prečo *krok von* zastavuje na inom mieste ako *krok dnu* alebo *krok cez* je, že tento spôsob umožňuje vynoriť sa na rovnakom príkaze, ktorý spôsobil volanie funkcie. To znamená, že *krok von* vždy vykoná vynorenie o práve jednu úroveň vyššie.

Pridanie týchto dvoch inštrukcií okolo každého bloku inštrukcií prislúchajúcich príkazu malo za následok nárast veľkosti inštrukčnej sady a binárnych súborov a potenciálne spomalenie vyhodnocovania programov. Vplyv na výkon však nebol meraný a ak existuje, tak je pravdepodobne zanedbateľný voči všetkým ostatným rádovo náročnejším operáciám. Výhodou bolo, že implementácia bola veľmi jednoduchá, keďže bola rovnaká ako pridávanie `NOOP` inštrukcií slúžiacich ako voľné miesta pre `BREAK` inštrukcie. Ďalšou výhodou bolo, že testovanie a ladenie tejto a ďalších funkcionalít ladiča a virtuálneho stroja bolo zjednodušené, keďže bolo z vykonávaných inštrukcií ľahko viditeľné, ktorý príkaz a ktorá jeho časť sú aktuálne vyhodnocované. Jednotlivé mechanizmy krokovania dnu, cez a von boli potom už jednoducho naprogramovateľné. Pri navštívení požadovanej inštrukcie `STEP_IN` alebo `STEP_OUT` sa porovnala veľkosť zásobníku volaní s jeho veľkosťou na začiatku vyhodnocovania a v prípade správneho vzťahu sa vyhodnocovanie prerušilo.

3.3 Pridávanie bodov prerušení mimo behu programu

Pridané body prerušenia s podmienkami môžu pristupovať k niektorým existujúcim premenným. Na overenie toho, či je daná podmienka valídny výraz, teda má správnu syntax a využíva iba viditeľné premenné, ju potrebujeme skompilovať v kontexte zvyšku kódu. V prípade, že budeme využívať premenné mimo rozsahu podmienky, potrebujeme vedieť adresy, na ktorých sa nachádzajú, teda potrebujeme syntaktický strom AST, ktorý už má pre všetky premenné pridelené adresy a rozsahy adries pre skôpy. Na to, aby sme ho získali je potrebné vykonať väčšinu procesu kompilácie a pre jednoduchosť a kompletnosť ho vykonáme celý.

Akcia kompilácie vytvára binárny súbor, čiže pracuje s diskom, a teda má trvalé vedľajšie následky. Chceli by sme zachovať predpoklad, že binárny súbor sa na disku vytvorí či prepíše iba ak užívateľ výslovne stlačí tlačidlo pre akciu kompilácie. Program tak vo všetkých ostatných prípadoch budeme kompilovať do pamäte. V prípade, že sa program nepodarí skompilovať, užívateľ sa nedozvie, či je jeho podmienka valídna, avšak takéto obmedzenie je z nášho pohľadu prijateľné. V prípade, že sa program podarí úspešne skompilovať, získame tým syntaktický strom, binárny súbor v pamäti a virtuálny stroj, ktorý sa ale pred spustením programu zmaže. Na pridávanie bodov prerušenia nám postačuje syntaktický strom, no body prerušenia pridávame aj do tohoto

dočasného virtuálneho stroja, pre prípad, že tento druhý krok zlyhá a mohli sme tak používateľovi poskytnúť presnejšiu spätnú odozvu.

3.4 Limitácie

Kvôli dizajnu virtuálneho stroja a dynamických bodov prerušenia sa schopnosť podmienok líši od schopnosti plnohodnotného jazyka WT*.

V rámci podmienok nie je možné deklarovať nové globálne a vstupno-výstupné premenné. Globálne premenné nie je možné vytvárať, keďže ich adresy a pozícia na zásobníku majú byť menšie ako adresy ľubovoľnej lokálnej premennej. Deklarovať nové vstupno-výstupné premenné by si voči aktuálnemu postupu vyžadovalo dodatočnú modifikáciu štruktúr virtuálneho stroja. Problém s touto zmenou je hlavne principiálny, a to, že zmena vstupno-výstupných premenných vlastne ovplyvňuje rozhranie programu. V prípade, že by používateľ vyžadoval dodať dáta do bežiaceho programu alebo zistiť hodnotu niektorých premenných, existujú na to iné spôsoby.

Funkcie sú vo virtuálnom stroji taktiež uložené v špeciálnych pomocných štruktúrach. Ich pridávanie počas behu programu by si vyžadovalo prehľadanie syntaktického stromu podmienky, nájdenie všetkých funkcií a doplnenie týchto štruktúr vo virtuálnom stroji o potrebné údaje. Definovanie nových funkcií počas behu programu by teda bolo možné, avšak užitočnosť je pochybná. Potreba pridávať funkcie za behu naznačuje, že užívateľ chce tvoriť príliš veľa funkcionality a usúdili sme, že v danom prípade je vhodné program vypnúť a túto funkcionalitu pridať do zdrojového kódu programu. Ten sa navyše ukladá na disk, čiže vytvorený kód nie je stratený pri vypnutí editora. Tieto nanovo pridané funkcie by boli využiteľné iba v rámci bodov prerušenia a najčastejšie iba v rámci daného jedného bodu prerušenia, v ktorom boli definované. Znamenalo by to, že body prerušenia sú závislé na iných bodoch prerušenia a situácia by sa skomplikovala v prípade, že by sa bod prerušenia s definíciou vymazal. Ďalšou otázkou by bolo, či by malo byť dovolené používať funkcie v bodoch prerušení, ktoré predchádzajú ich definíciu. Ak nie, vyžadovať od užívateľa umiestnenie bodu prerušenia s deklaráciou je celkom nepraktické. Ak to však dovolíme, porúšame pravidlá jazyka WT*. Prospešnejším riešením je pridať do jazyka WT* podporu pre anonymné funkcie.

Aktuálne nie je kompilačnou chybou podmienky bodu prerušenia použiť ešte nedeklarovanú premennú patriacu do niektorého zo skôpov, v ktorých sa nachádza podmienka. V tomto prípade sa prečíta alokovaná pamäť zásobníku, avšak s nesprávne inicializovanou hodnotou, či už náhodnou alebo nejakým pozostatkom z doterajších výpočtov.

Počas vykonávania podmienky bodu prerušenia sa vyhodnocovanie nezastaví na inom bode prerušenia. Podmienky týchto bodov prerušení budú vyhodnotené pre prí-

pad, že nimi užívateľ chcel docieliť nejaký vedľajší účinok. Avšak aj v prípade, že sa vyhodnotia s nenulovým výsledkom na nich vyhodnocovanie nebude zastavené. Dalo by sa argumentovať, že zastavenie počas vyhodnocovania podmienky je dokonca nežiadané. Okrem toho považujeme prerušenie počas *potenciálneho* prerušenia, keďže hlavná podmienka sa ešte nemusí vyhodnotiť na nenulovú hodnotu, za mätúcu situáciu.

Podobne aj krokovanie počas vyhodnocovania podmienky bodu prerušenia má iba pochybné využitie. V prípadoch, kedy by sa dalo využiť je veľmi pravdepodobné, že daný problém sa dá buď riešiť lepšie alebo ide o problém, ktorý by nemal vznikať. Účelom krokovania je hľadanie problémov v kóde, ak teda užívateľ chce krokovať kód podmienky bodu prerušenia, mal by sa zamyslieť, prečo píše podmienky, ktoré majú v sebe problémy alebo či informácia, ktorú sa snaží získať nie je dostupná iným prístupom.

Na prvý pohľad by sa zdalo, že podmienky písané v natívnom jazyku by si zaslúžili plnú funkcionalitu daného jazyka. Na druhý pohľad je daná chýbajúca funkcionalita nežiadaná. Je rozumné si položiť otázku, či je vôbec vhodné mať možnosť písať podmienky s aktuálne podporovanou silou, keďže už to, čo dokážu teraz je možno nežiadané. Odpoveď na túto otázku je pravdepodobne, že aktuálne možnosti písania podmienok sú naozaj zbytočne silné. Zároveň však nemá zmysel ich umelo obmedzovať. Je dobré, aby užívatelia nepísali príliš komplikované podmienky, ale neuškodí ak sa niekto rozhodne, že chce využiť ich silu naplno. Užívatelia tak majú možnosť do veľkej miery ovplyvňovať fungovanie programu za behu. Navyše bol tak tento problém ako aj jeho riešenie zaujímavé a implementovať slabšiu formu podmienok by bolo porovnateľne náročné.

3.5 Zobrazovanie obsahu premenných

Virtuálny stroj obsahuje pomocné informácie mapujúce skôpy na rozsahy inštrukčného kódu. Preto počas bodu prerušenia vieme na základe inštrukčného ukazovateľa zistiť, v akom skôpe sa program nachádza. Prejdením tohoto a všetkých jeho rodičovských skôpov získame zoznam všetkých viditeľných premenných spolu s ich dátami ako mená a adresy. S pomocou týchto informácií vieme prečítať zásobník ľubovoľného vlákna a zobraziť hodnoty všetkých viditeľných premenných.

Každý volací rámec musí mať zaznamenanú návratovú adresu funkcie. Túto adresu vieme použiť ako inštrukčný ukazovateľ a rovnakým spôsobom opísaným vyššie zobraziť informácie o viditeľných premenných platných v ľubovoľnom rámci na zásobníku volacích rámcov.

Kapitola 4

Vývojové prostredie

V tejto kapitole predstavíme funkcionalitu vývojového prostredia, ukážeme ako sú v ňom doterajšie komponenty pospájané dokopy a predvedieme ukážku použitia.

4.1 Hlavný cyklus udalostí

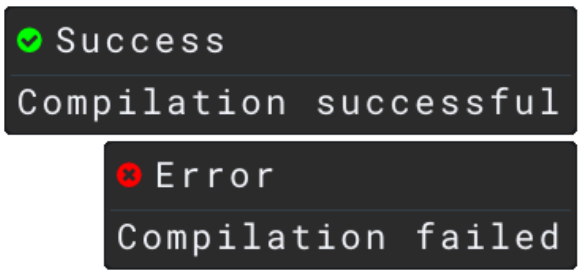
Vykonávanie aplikácie beží v hlavnom cykle udalostí (anglicky *main event loop*). Ten neblokujúco spracuje všetky existujúce nespracované udalosti, aktualizuje stav aplikácie a prekreslí grafické rozhranie. Týmto spôsobom by sa aplikácia obnovovala tak rýchlo, ako by jej to len výkon počítača dovolil. Obnovovacia frekvenciu teda zhora limitujeme na obnovovaciu frekvenciu obrazovky, na ktorej je aplikácia vykresľovaná, keďže častejšie prekresľovanie nie je na obrazovke pozorovateľné. Aj tento prístup však väčšinou mrhá prostriedkami procesoru, keďže aplikácia zbytočne obnovuje a prekresľuje rozhranie aj keď užívateľ nepredstavil žiadnu akciu či zmenu. V prípade, že užívateľ nevykonáva žiadnu akciu, teda program nezaznamenal žiadnu udalosť, aplikáciu nie je nutné obnoviť. Toto by sme vedeli dosiahnuť zmenou neblokujúceho získavania udalostí na blokujúce získavanie. V tomto prípade by avšak prestali fungovať niektoré vlastnosti grafického rozhrania, ktoré bežia aj bez užívateľského vstupu (ako blikanie kurzoru) alebo dlhší čas po skončení užívateľského vstupu (animácia alebo vypisovanie kompilačnej chyby). Na konci každej obnovovacej iterácie čakáme, kým nevznikne udalosť alebo pokým neubehlo 100 ms od začiatku iterácie, čím obnovovaciu frekvenciu zdola ohraničíme na 10 Hz. Keďže obnovovacia frekvencia bežných obrazoviek je 60 Hz, v praxi to znamená, že ak užívateľ nič nerobí, šetríme aspoň 83% procesorového času, Ak ale užívateľ niečo robí, aplikácia je tak responzívna, ako je len možné a zároveň míňame iba toľko prostriedkov koľko je nevyhnutné. Existujú spôsoby, ako ušetriť ešte viac procesorových prostriedkov, avšak aplikácia je aktuálne dostatočne úsporná.

4.2 ImGui

Na vykresľovanie používame knižnicu *Dear Imgui* [7]. Ide o populárnu nenáročnú ľahko použiteľnú implementáciu základného užívateľského rozhrania s *Immediate Mode GUI* paradigmou. Jej výhody sú rýchle nastavenie a tvorenie grafického rozhrania s minimálnym stavom a synchronizáciou dát na strane nášho projektu. Dear ImGui nie je primárne mierené na tvorbu užívateľského softvéru (anglicky *software*), avšak je veľmi vhodné na tvorbu ladiča a spolu so schopnosťou rýchlo tvoriť jednoduché použiteľné rozhranie je ideálnou voľbou pre tento projekt.

4.2.1 Notifikácie

Na upozornenie užívateľa ohľadom dôležitých udalosti, ako napríklad ukončenie kompilácie používame notifikácie sprostredkované knižnicou *imgui-notify* [14]. V prípade kompilácie slúžia na zdôraznenie výsledku, keďže bez nich si ho užívatelia často krát nevšimli keď bola kompilácia neúspešná. Príklad niektorých notifikácii môžeme vidieť na obrázku 4.1.



Obr. 4.1: Notifikácie ohľadom výsledkov kompilácie

4.3 Moduly

Aj keď sa rôzne populárne a nepopulárne prostredia líšia vo funkcionalite, zameraní a ich implementácii, vieme medzi nimi nájsť veľa spoločného. Naše vývojové prostredie sa bude snažiť obsahovať tieto spoločné, a teda základné prvky a pridá ďalšie prvky špecifické a prospešné pre vývoj programov vo WT*.

Základnými prvkami sú textový editor, súborový strom a dialógové okno pre výber súborov. S týmito prvkami bude užívateľ schopný vytvoriť si projektové súbory a napísať ich obsah. Ďalšími základnými prvkami potrebnými pre tvorbu programov sú ovládač behu programu a priestory pre manipuláciu s textovými informáciami. Vďaka týmto prvkom bude užívateľ schopný vytvorené zdrojové kódy skompilovať, dozvedieť sa o prípadných problémoch, ktoré počas kompilácie nastali, zadať vstupné textové

dáta a získať výstupné textové dáta. Nakoniec pre potreby ladenia potrebujeme prvok, ktorý bude ovládať beh programu a zobrazovať informácie o programe.

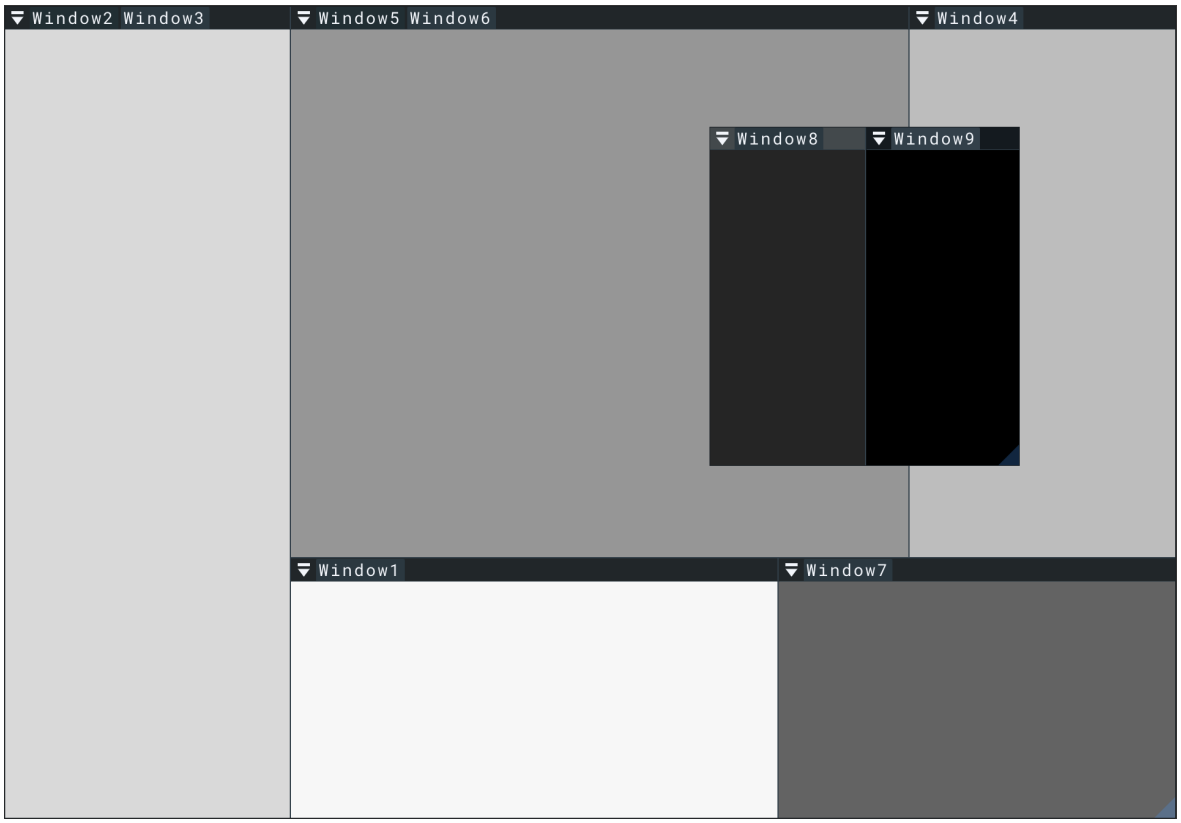
4.3.1 Implementácia

Tieto jednotlivé prvky sú vo veľkej miere od seba nezávislé, každý z nich má mať v grafickom rozhraní svoj vyhradený priestor pre fungovanie veľa z nich má podporovať viacero inštancií (napríklad editor) a chceme aby bol vývoj a pridávanie potenciálnych nových prvkov jednoduché a modulárne. Chceme teda, aby boli tieto prvky od seba odseparované graficky aj implementačne, čo docielime architektúrou modulov (anglicky *plugin*). Každý takýto samostatný prvok je interné implementovaný ako trieda dediaca od grafického modulu, čiže objekt podporujúci jednoduché rozhranie ktorého súčasťou je metóda vykresľovania. Jadro aplikácie si potom pamätá zoznam všetkých aktívnych modulov a v hlavnom cykle na každom z nich zavolá metódu vykreslenia. Jadro aplikácie si nemusí o grafických prvkoch pamätať žiadne informácie súvisiace s vykresľovaním, všetka logika a dáta sú obsiahnuté v module.

Táto separácia uľahčuje implementáciu, keďže pridanie nového prvku zahŕňa iba vytvorenie objektu implementujúceho malé rozhranie a inicializáciu modulu v jadre aplikácie. Keďže niektoré moduly ovplyvňujú stav hlavnej aplikácie (napríklad súborový strom od užívateľa dostal príkaz na otvorenie súboru) alebo vyžadujú komunikáciu medzi sebou, je potrebné umožniť modulom určitý spôsob komunikácie smerom von. Súčasťou rozhrania modulov je tak aj funkcionalita spätných volaní (anglicky *callback*). Jadro aplikácie môže kedykoľvek (ale väčšinou počas inicializácie) poskytnúť modulu funkciu, ktorú modul zavolá v prípade špecifickej udalosti. Táto funkcia môže patriť komukoľvek, či už jadru aplikácie, inému modulu alebo ísť o globálnu funkciu. Táto funkcia berie dáta ako parameter a vracia dáta ako návratovú hodnotu, vďaka čomu môžu moduly posielať a získavať informácie, umožňujúce jednoduchú všesmernú komunikáciu. Výhodou spätných volaní je jednoduchosť ich používania a separácia informácií. Modul nemusí vedieť čo daná návratová funkcia robí, stačí aby vedel kedy ju má zavolať, aké dáta jej má poskytnúť ako parameter a čo robiť s dátami, ktoré dostane späť. Oproti udalostiam majú návratové funkcie výhodu, že sú schopné byť synchrónne.

4.3.2 Grafické rozloženie

Pre praktické využitie prostredia je potrebné, aby boli grafické prvky umiestnené na rozumných miestach – textový editor v strede a ovládacie prvky po stranách s možnosťou meniť pomer veľkosti jednotlivých sekcii. Taktiež je praktické, ak je užívateľ schopný mať otvorených viacero súborov naraz usporiadaných v paneli kariet (anglicky *tab bar*). Funkcionalita rozdelenia obrazovky do viacerých sekcii a umiestňovania prvkov do panelu kariet sa volá dokovanie (anglicky *docking*). Dokovanie nie je podpo-



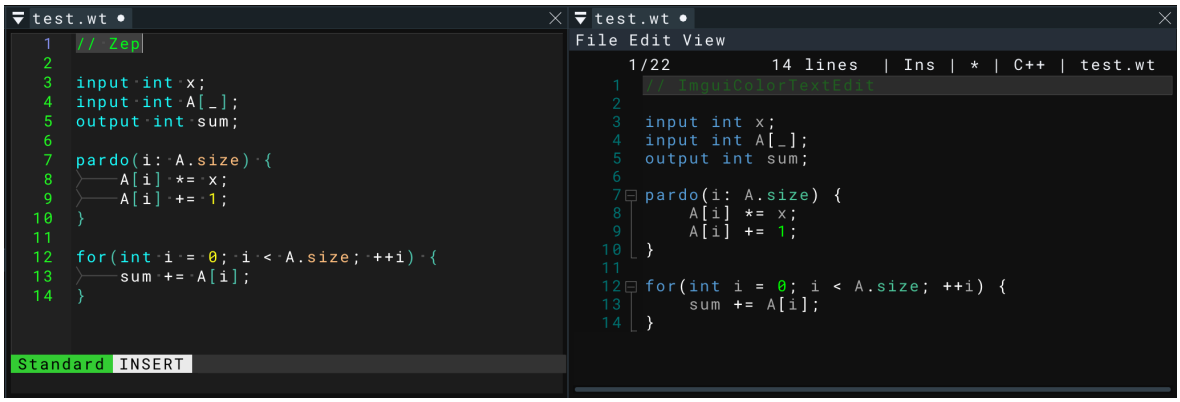
Obr. 4.2: Reprezentácia využitia dokovania

rované v základnej verzii Dear ImGui, avšak je témou vedľajšej masívnej vetvy [15], ktorej je venovaná väčšina vývoja daného projektu. Na implementáciu dokovania teda stačí, aby sme ako grafickú knižnicu používali dokovaciu verziu Dear ImGui a každý modul bol obsiahnutý v samostatnom okne. Tie potom môže užívateľ presúvať a dokovať podľa svojich požiadaviek a vytvoriť si tak vlastné preferované rozloženie. Príklad rozloženia môžeme vidieť na obrázku 4.2.

4.3.3 Modul Editoru

Editor je štandardný prvok grafických rozhraní a pre jeho implementáciu využijeme niektorú existujúcu knižnicu. Modul editora bude rozširovať rozhranie grafického modulu o zopár funkcií, ako otváranie a ukladanie súboru. Na pridanie knižnice editora formou modulu do našej aplikácie bude potom stačiť vytvoriť malý implementačný obal (anglicky *wrapper*) plniaci toto rozhranie. Tento prístup nám umožnil jednoducho pridať dve rôzne implementácie editorov, viditeľné na obrázku 4.3.

Jazyk WT* má syntax založenú na jazyku C. Navyše nepodporuje príliš zložité konštrukty, takže zvýrazňovanie syntaxe je celkom jednoduché. Keďže jazyk C je veľmi rozšírený, vo väčšine editorov ktoré podporujú zvýrazňovanie sa syntax pre WT* dá odvodiť zjednodušením a miernou úpravou syntaxe pre jazyk C alebo C++.



Obr. 4.3: Ukážka dvoch podporovaných modulov editorov (Zep vľavo, ImGuiColorTextEdit vpravo)

4.3.4 Moduly pre ladenie programu

Na ovládanie a sledovanie ladiča behu programu sme vytvorili viacero grafických prvkov.

Prvým z nich je ovládač ladiča, ktorý slúži na ovládanie behu programu. V tomto module ovládame kompiláciu, spúšťanie a zastavovanie programu, krokovanie, pridávanie a odstraňovanie bodov prerušení a zobrazujeme užívateľovi aktuálnu pozíciu vyhodnocovania v rámci zdrojového a inštrukčného kódu. Tento modul môžeme vidieť na obrázku 4.4a. Nejde však o finálny stav, keďže dizajn bude neskôr zjednodušený a o funkcionalitu manipulácie s bodmi prerušenia a zobrazovanie aktuálnej polohy sa bude starať hlavne editor.

Druhým prvkom je analyzátor programu. Tento slúži hlavne na vývojové účely, keďže spracúva a zobrazuje informácie o programe WT* dostupné z binárneho súboru. Ide o informácie ako zoznam globálnych premenných, pamäťový mód programu, inštrukčný kód a podobne. Bežný používateľ pre väčšinu z nich nemá veľké využitie. Okrem toho však má na starosti existujúce body prerušenia, ktoré môžeme upravovať, odstraňovať a dočasne povoliť či zakázať, čo môžeme vidieť na obrázku 4.4b. Nachádzajú sa tu aj body prerušenia, ktorých pridávanie bolo z akéhokoľvek dôvodu neúspešné a užívateľ sa môže tento problém pokúsiť opraviť.

Tretím prvkom je zobrazovač premenných, ktorý slúži na čítanie hodnôt všetkých dostupných premenných bežiaceho programu. Tento modul využíva rozhranie ladiča na získavanie všetkých premenných a ich hodnôt. Zobrazené sú tri sekcie pre globálne premenné, premenné zo zásobníku volacích rámcov a premenné vlákien. Viditeľné premenné z určitej oblasti sú zobrazené v tabuľke a utriedené podľa skôpov, v ktorých sa nachádzajú, pričom zobrazené sú aj zatienené (anglicky *shadowed*) premenné. Sekcia premenných zo zásobníku volacích rámcov zobrazuje takúto oblasť pre každý volací rámec aktuálne zvoleného vlákna. Sekcia premenných vlákien zasa zobrazuje premenné z

oblasti každého aktuálne aktívneho vlákna. Každé z týchto vlákien vie byť zvolené pre zobrazovanie jeho premenných v sekcii zásobníku volacích rámcov. Všetky tri sekcie môžeme vidieť na obrázku 4.4c.

4.4 Knižnice

4.4.1 Organizácia v Git

V projekte používame distribuovaný systém riadenia revízií *Git*. Knižnice použité v našom projekte boli pridané ako takzvané podmoduly (anglicky *git submodules*). Toto umožňuje jednoduchú organizáciu, malú veľkosť repozitára a ľahké aktualizovanie knižníc. Verejne dostupný kód, ktorý v našom projekte využívame je vo veľa prípadoch vytvorený iba jedným individuálom. Tieto projekty nie sú veľmi otestované, kompatibilné s inými systémami alebo jednoducho dobre napísané. Častými problémami boli implementácie nekompatibilné s Linuxom alebo problémy so zlou správou pamäte. Viacero z knižníc, ktoré v tomto projekte používame teda vyžadovalo dodatočné zmeny týkajúce sa opráv alebo rozšírení pre potreby nášho projektu. V prípade aktívnych projektov tieto vylepšenia plánujeme navrhnúť pôvodným projektom, nie každý projekt je však ešte aktívny. V tomto prípade vytvoríme vidlu (anglicky *fork*) s aplikovanými zmenami pre využitie zvyškom komunity.

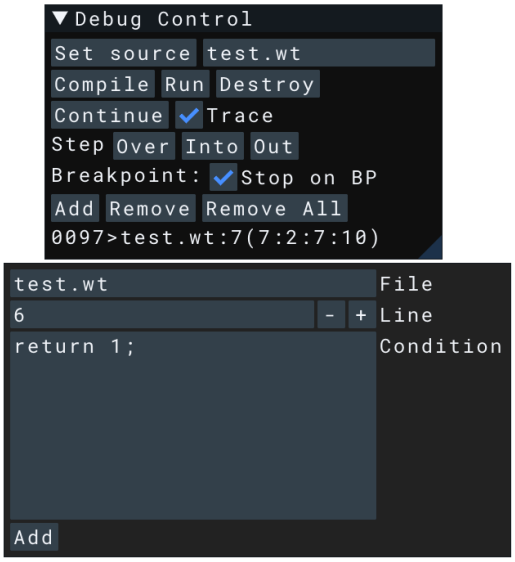
4.4.2 Porovnanie dialógových okien pre výber súborov

Dialógové okno pre výber súborov je úplne štandardný prvok grafických aplikácií, a tak použijeme už existujúcu knižnicu. Na výber sa nám ponúka viacero možností. Vyberať si môžeme medzi implementáciami využívajúce bekend (anglicky *backend*) systému [16] [17] [18] [19] alebo našej grafickej knižnice Dear ImGui [20].

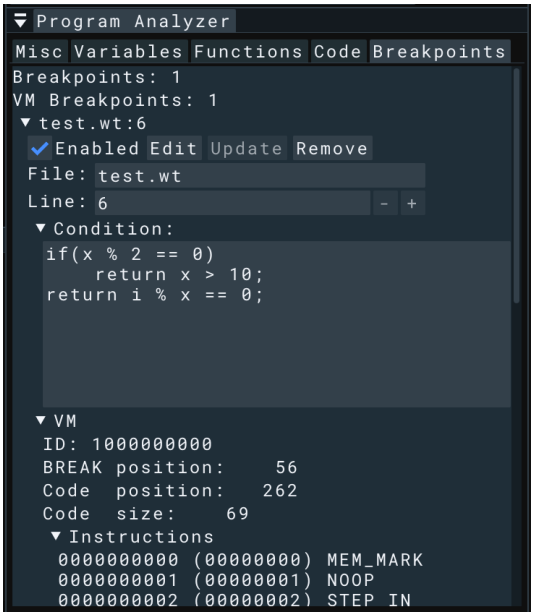
V rámci knižníc vytvorených v Dear ImGui existuje veľký výber, skoro všetky z nich však vizuálne nevyzerajú dobre. Jednou výnimkou je *ImFileDialog* [21], ktorý by sme mohli použiť.

Najlepšou možnosťou je však použiť systémovú implementáciu, keďže bude používať grafické rozhranie, na ktoré bude užívateľ najviac zvyknutý a zvyčajne majú aj najviac funkcionality. Problém s týmto výberom je, že nás program bude podporovať iba systémy, ktoré podporuje daná knižnica. Z týchto knižníc sa ako najlepšie alternatívy podporujúce najviac systémov javili *Native File Dialog Extended* [17] a *tiny file dialogs* [19], medzi ktorými sme použili tú prvú.

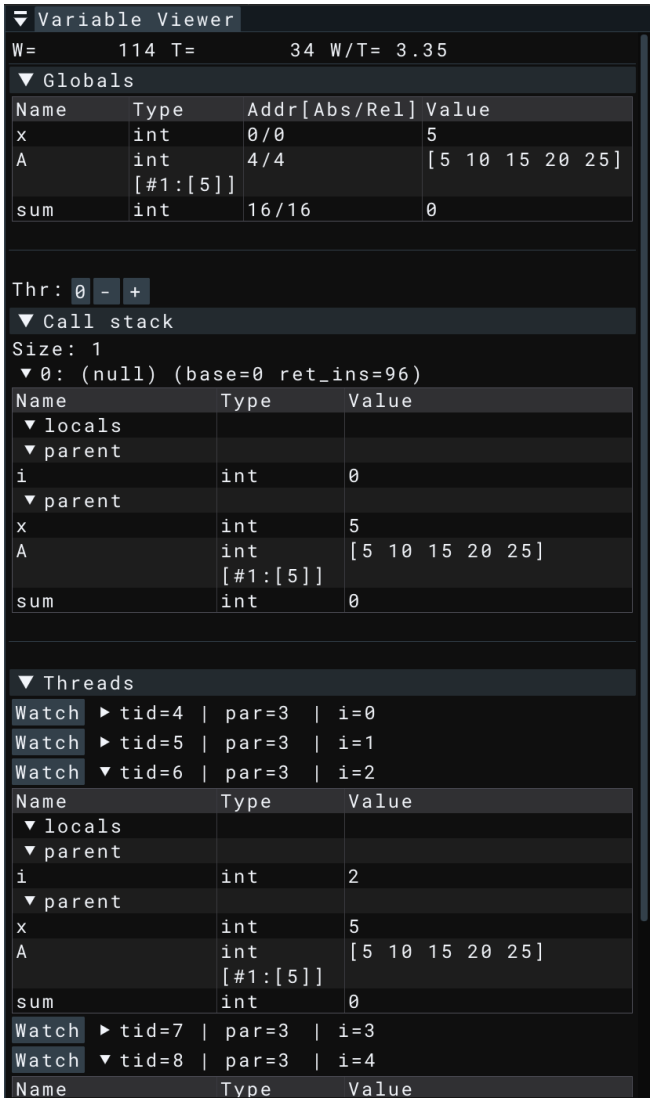
Je však možné, že pri krížovom preklade (anglicky *cross compilation*) na iné operačné systémy alebo web narazíme na nekompatibilitu. V tomto prípade môžeme pre nekompatibilné platformy pridať ďalšie knižnice alebo sa môžeme vrátiť ku Dear ImGui



(a) Ovládač ladiča a pridávanie podmieneného bodu prerušenia



(b) Manipulácia s bodom prerušenia v analyzátore programu



(c) Zobrazovač premenných

Obr. 4.4: Moduly pre ladenie programu

implementácii, ktorá je kompatibilná s každým systémom.

4.4.3 Editor Zep

Zep [22] sa zdá byť najlepším projektom pre textový editor slúžiaci na programovanie v Dear ImGui. Sľubuje podporu pre štandardný textový mód, ale aj ovládanie Vim štýlom. Významnou vlastnosťou je podpora pre zvýrazňovanie textu slúžiaceho na zobrazenie chýb v zdrojovom kóde. Navyše podporuje aj viaceré pokročilé funkcie ako dokovanie a takzvané *REPL* (z anglického *Read-eval-print loop*), čo je jednoduché interaktívne počítačové programovacie prostredie. Interne používa rovnako ako my systém spätných volaní a dizajn je rozdelený do niekoľkých vrstiev.

Toto bol dôvod prečo bol spočiatku pre náš projekt ako textový editor zvolený Zep. Pri vývoji a používaní Zepu sme však objavili veľké množstvo problémov, čo nakoniec viedlo k zmene textového editoru. Najväčším a najzjavnejším problémom je chýbajúca funkcionalita. Aj napriek všetkému čo Zep ponúka, masívne množstvo funkcionality stále chýba. Základné použitie myši, ako označovanie textu alebo dokonca rolovanie (anglicky *scroll*) nie je podporované. Zep síce obsahuje jednoduchý Vim mód, avšak veľkou výhodou Vimu je jeho rozsiahla škála funkcií, ktorú obsahuje, čo sa prejaví ako veľká nevýhoda v prípade Zepu, ktorému táto škála chýba. Štandardný štýl ovládania je implementovaný ako *Insert* mód Vimu. Toto je celkom dobrý nápad, keďže tieto dva spôsoby sú si veľmi podobne, avšak nie sú identické, čo spôsobuje množstvo bizarných problémov. Implementácia Zepu je rozdelená do niekoľkých vrstiev, čo však znamená, že kód je rozlezený a ťažko sa číta a mení ak sa v ňom človek perfektne nevyzná. Veľké časť funkcionality, ako napríklad REPL, dokovanie a iné nie sú v našom projekte nevyužiteľné a zbytočné.

Niektoré problémy sme opravili:

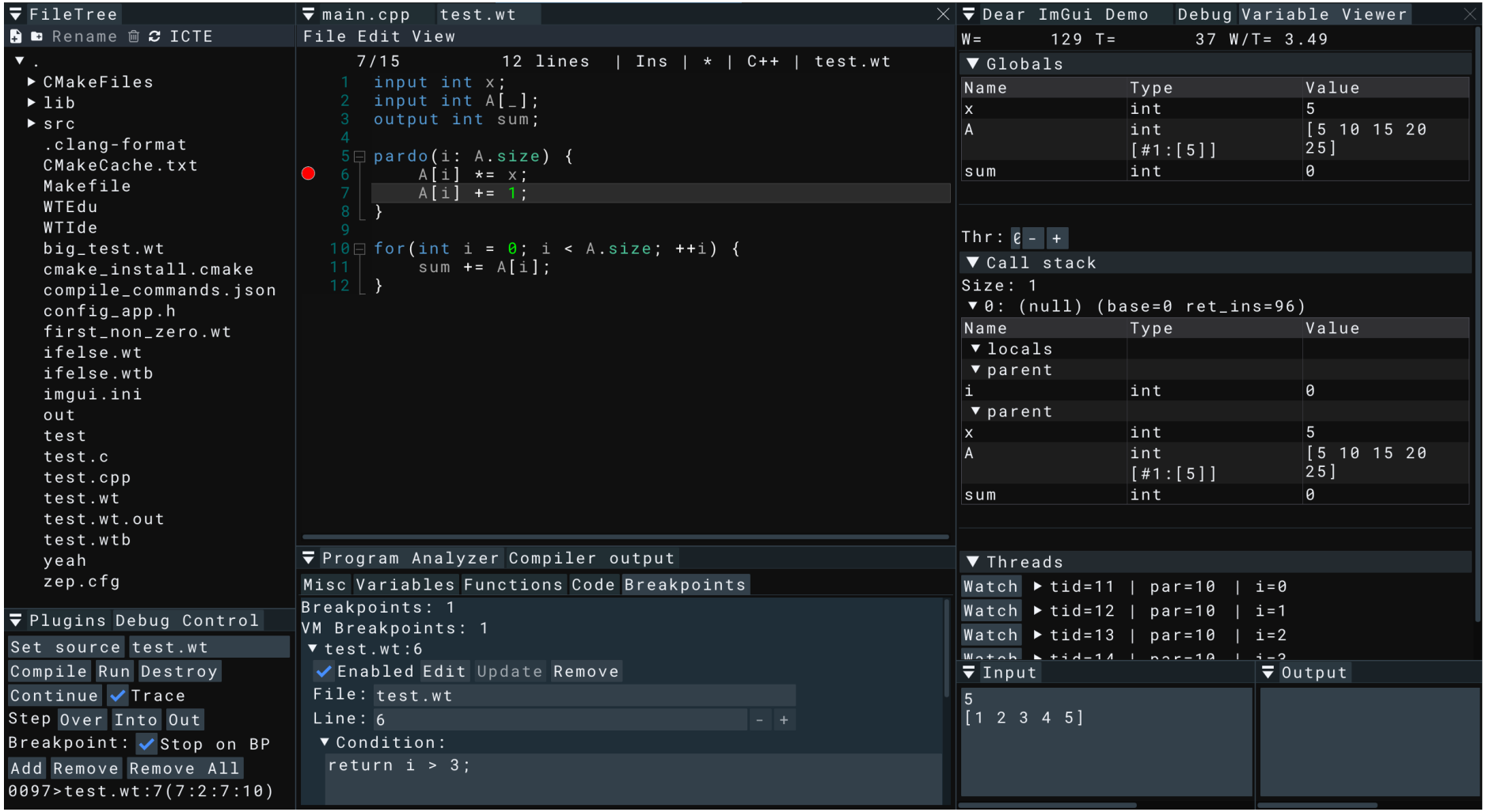
- Vyriešili sme hlavné rozdieli medzi štandardným štýlom a *Insert* módom
- Opravili sme problém so zvýrazňovaním textu obsahujúceho `\\`
- Pridali sme podporu pre rolovanie myšou
- Prepísali a zjednodušili sme kód týkajúci sa klávesových skratiek, vďaka čomu podporuje klávesové skratky obsahujúce čísla, špeciálne znaky a modifikátory `Alt` a `Shift`
- Pridali sme niektoré často používané klávesové skratky ako napríklad `Ctrl+S` pre uloženie dokumentu
- Zrušili sme otravné vypisovanie textových reprezentácii nespracovaných klávesových skratiek

- Opravili sme kopírovanie do systémovej schránky
- ...

Mnoho problémov však zotrvalo, čo malo za následok zmenu hlavného textového editoru. Keďže však už bolo integrácii Zepu venovaného masívne množstvo času, je stále k dispozícii na použitie pre užívateľov, ktorý by preferovali Vim štýl navigovania svojho editoru.

4.4.4 Editor ImGuiColorTextEdit

Hlavný repozitár projektu ImGuiColorTextEdit [23] nebol už viac ako 3 roky aktualizovaný. Toto bol jeden z hlavných dôvodov prečo nebol pre editor zvolený tento projekt. V neskoršom štádiu vývoja sme však objavili vidlu, vytvorenú taktiež užívateľom vyvíjajúcom grafické vývojové prostredie [8]. Tento repozitár mal nedávnu aktivitu a veľkým množstvom *commit*ov, ktorých pridaná funkcionalita bola veľmi výtaná aj v našom projekte. Okrem toho je tento editor výrazne jednoduchší oproti Zepu a zároveň poskytuje viac funkcionality v súlade s našimi požiadavkami. Pri integrácii sa vyskytlo zopár problémov, avšak boli rádovo menšie ako tie v Zepe. Tento editor má jednoduchší dizajn, lepšie rozhranie a zvýrazňovanie syntaxe a taktiež podporuje anotácie rovnako ako Zep. Narozdiel od Zepu plne podporuje ovládanie myšou, vrátane označovania textu, rolovania a dokonca pridávania grafických bodov zlomu. Keďže bol do nášho projektu pridaný neskoro, tieto vynikajúce vlastnosti však ešte nie sú prepojené a využité.



Obr. 4.5: Ukážka vývojového prostredia počas ladenia programu

Kapitola 5

Budúca práca

Tvorba vývojových prostredí je nekonečný boj. Populárne nástroje sa vyvíjajú roky tímom ľudí a prinášajú neuveriteľnú funkcionalitu a nastaviteľnosť. To ale nie je cieľom tohoto projektu. Napriek tomu ešte existuje niekoľko vlastností, ktoré by bolo užitočné implementovať.

5.1 WT*

Keďže jazyk WT* je zjednodušený oproti jazyku C, mohlo by byť vhodné ho rozšíriť. Tým nemyslíme, že by sme v jazyku WT* mali byť schopní pracovať so systémovými zariadeniami. V jazyku však chýbajú schopnosti ako práca s reťazcami, vypisovanie na štandardný výstup alebo skoré prerušenie cyklov príkazom `break;`.

Ďalšou oblasťou, na ktorej by sa dalo pracovať, je syntaktický analyzátor a s ním spojené problémy.

Aktuálne je WT* funkčný, avšak v prípade, že by sa táto či iná funkcionalita mala pridávať, bolo by pravdepodobne výhodné najprv prepísať aspoň niektoré časti kódu nanovo alebo dokonca v C++. Na mnohých miestach sa napodobňovala funkcionalita C++ ručne implementovanými riešeniami, ktoré často obsahovali chyby a používajú sa ťažšie ako riešenia štandardu C++. Nie je však isté, či je túto funkcionalitu potrebné pridávať, keďže jej užitočnosť závisí na jej využití počas výučby.

5.2 Ladič

Schopnosti ladiča sú hlavne závislé na schopnostiach WT*. Ladič by sa dal vylepšiť v niektorých oblastiach, ako napríklad pridávanie bodov prerušení na jednotlivé časti príkazu. Tieto schopnosti sú zatiaľ hlavne obmedzené na strane WT*. Rovnako bude pravdepodobne najväčšou náplňou práce na ladiči prispôsobovanie jeho funkcionality v prípade, že sa budú prepisovať časti WT*.

5.3 Únik pamäte

Aj náš program, podobne ako veľa iných programov písaných v C a C++, trpí únikom pamäte. Pamäť uniká vo veľkosti rádovo niekoľko megabajtov, čo nie je kritické, ale únik pamäte je tiež chyba, ktorú je treba opraviť. S použitím nástrojov *Valgrind* [24] a *sanitizer* [25] budeme hľadať a odstraňovať príčiny úniku. Z doterajšieho skúmania je zjavné, že väčšina uniknutej pamäte pochádza z časti prislúchajúcim podprojektom Zep a WT*.

5.4 Vývojové prostredie

5.4.1 Testovanie a chyby

Vývojové prostredie je hotové a funkčné, avšak iba v rozsahu, v akom bol kód otestovaný. Je teda potrebné program intenzívnejšie pretestovať a odstrániť nájdené chyby. Odstrániť všetky je však asi nemožné. Je taktiež potrebné pridať mechanizmus, ktorý umožní užívateľovi spamätať sa z pádu programu, čiže priebežne ukladať zálohy súborov a nastavení prostredia.

5.4.2 Kompilácia na iné platformy a web

Vývoj tohoto projektu bol doteraz zameraný primárne na operačný systém Linux s úmyslom vytvoriť funkčný a použiteľný program pre aspoň jednu platformu. Aby bol nakoniec využiteľný všetkými študentmi, je potrebné skompilovať ho a opraviť prípadne problémy s kompatibilitou na zvyšných platformách.

5.4.3 Prepojenie ladiča s editorom

Ovládanie ladiča je teraz izolované do modulu ovládača ladiča. To je úmyselne jednoduché a kostrbaté, keďže zámer je manipuláciu s bodmi prerušenia vykonávať skrz grafické rozhranie vstavaného editora. Informácie z tohoto modulu ako pozícia programu v rámci súboru počas krokovania bude taktiež vizuálne zobrazená v editori. Obsahy premenných je aktuálne možné čítať iba v module zobrazovača premenných, avšak neskôr bude možné zobraziť obsah premenných podržaním kurzora myši nad názvom premennej vo vnútri textového editora.

5.4.4 Modul grafu behu programu

V aktuálnom stave sa pri ladení dajú pozrieť hodnoty všetkých premenných zo všetkých vlákien a volacích rámcov. V tomto zobrazení však nie je vidieť, ako sa odohrával

beh programu, teda v ktorom bode behu sa program rozdelil na koľko vlákien. Jedným z hlavých cieľov bude vo vývojovom prostredí vytvoriť modul s prehľadnou reprezentáciou behu programu. Prirodzená reprezentácia priebehu programu je strom, ktorého vnútorné vrcholy reprezentujú bod behu programu, v ktorom sa buď beh rozdelil do niekoľkých vlákien alebo sa beh presunul na iné miesto z dôvodu zavolania funkcie. Programy používané na predmete EPA môžu používať rádovo stovky až tisíce vlákien a algoritmy sú väčšinou rekurzívneho charakteru s hĺbkou pár desiatok volaní. Takýto veľký strom nie je možné v čitateľnej podobne celý zmestiť na jednu obrazovku, takže je potrebné vytvoriť responzívne a interaktívne zobrazenie. To znamená, že užívateľ sa môže rozhodnúť pozorovať štruktúru celého stromu, pričom v tomto prípade bude možné vidieť všetky vrcholy, avšak žiaden z vrcholov nebude môcť zobrazovať žiadne detaily o stave programu. Navyše v prípade, že pôjde naozaj o program obsahujúci masívne množstvo vlákien, pravdepodobne nebude možné ani v tejto štrukturálnej reprezentácii zobraziť všetky vrcholy stromu. V tomto prípade budú niektoré vrcholy reprezentovať skupiny vlákien, napríklad veľkosti desať. Alternatívne sa užívateľ bude môcť pozrieť zblízka na obmedzenú časť tohoto stromu, či už výberom a zväčšením určitého výseku, alebo vyfiltrovaním iba niektorých vetiev či vrstiev stromu. V druhom prípade budú môcť vrcholy stromu zobrazovať niekoľko na prvý pohľad viditeľných užívateľom zvolených dôležitých informácií. Kliknutím na vrchol stromu sa v module zobrazovača premenných zobrazí sekcia reprezentujúca daných vrchol, čím si užívateľ bude môcť pozrieť detaily stavu programu v danom bode.

5.5 Rozšírenie knižnice algoritmov jazyku WT*

Tento projekt chceme využiť na naprogramovanie vzorových riešení domácich úloh a implementácii komplikovanejších algoritmov preberaných na predmete EPA. Tie budú slúžiť nie len ako referencia pre študentov, ale aj ako dodatočný test na otestovanie stability, funkčnosti a schopnosti používať toto vývojové prostredie na výučbu a riešenie náročnejších problémov v modeli WT*.

Záver

Podarilo sa nám implementovať funkčné grafické vývojové prostredie WTIde pre platformu WT* s plne schopným ladičom. V rámci ladiča podporujeme dynamické body prerušenia, krokovanie a zobrazovanie obsahu všetkých existujúcich premenných. Body prerušenia sme spravili podmienené, pričom podmienky sú písané v natívnom jazyku a podporujú väčšinu jeho schopností vrátane cyklov a manipulácie s vonkajšími premennými. Použiteľnosť aplikácie je na vysokej úrovni a študenti magisterského predmetu EPA sa budú môcť ľahšie zoznamovať s tvorbou paralelných algoritmov.

V priebehu vývoja sme pracovali s projektom WT* a niekoľkými knižnicami, v ktorých sme opravili viacero chýb a pridali novú funkcionalitu.

Vytvorili sme dobrý základ, ktorý môžeme v budúcnosti ďalej rozširovať. Nasledujúcim krokom tohoto projektu bude podpora zvyšných operačných systémov a webových prehliadačov. Zvyšnými veľkými cieľmi sú napravenie úniku pamäte, dokončenie prepojenia editora s ladičom a doplnenie grafického rozhrania o prvok vizualizujúci beh programu.

Aktuálne zdrojové kódy a najnovšie verzie stiahnuteľných skompilovaných programov je možné nájsť na `https://github.com/fezjo/WTIde`.

Literatúra

[1] Vishkin, U.: Using Simple Abstraction to Reinvent Computing for Parallelism. *Commun. ACM*, ročník 54, č. 1, jan 2011: str. 75–85, ISSN 0001-0782, doi:10.1145/1866739.1866757.
URL `https://doi.org/10.1145/1866739.1866757`

[2] JáJá, J.: *An introduction to parallel algorithms*. Addison-Wesley, 2001.

[3] Kráľovič, R.: WT*. Stránka `https://beda.dcs.fmph.uniba.sk/wtstar`.

[4] Tomcsányi, G.: Simulácia PRAM výpočtov. Univerzita Komenského v Bratislave, 2012, dostupné z `http://www.dcs.fmph.uniba.sk/diplomovky/obhajene/Detail.php?id=340`.

[5] Smažáková, D.: Prostredie pre experimentovanie s PRAMami. Univerzita Komenského v Bratislave, 2009, dostupné z `http://www.dcs.fmph.uniba.sk/bakalarky/obhajene/Detail.php?id=110`.

[6] Microsoft: VSCode - Visual Studio Code. Stránka `https://code.visualstudio.com`.

[7] Cornut, O.: Dear ImGui. Stránka `https://github.com/ocornut/imgui`.

[8] BalazsJako and dfranx: ImGuiColorTextEdit fork od dfranx. Stránka `https://github.com/dfranx/ImGuiColorTextEdit`.

[9] Wilhelm, R.; Seidl, H.; Hack, S.: *Compiler Design: Syntactic and Semantic Analysis*. Springer Berlin Heidelberg, 2013, ISBN 9783642175404.
URL `https://books.google.sk/books?id=NTIkJAuytiwC`

[10] Levine, J.: *flex and bison*. Sebastopol, CA: O'Reilly Media, aug 2009, ISBN 978-0-596-15597-1.

[11] Paxson, V.: Flex. Stránka `https://github.com/westes/flex`.

[12] Corbett, R.; The GNU Project: Bison. Stránka `https://www.gnu.org/software/bison/`.

[13] Aho, A.; Lam, M.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Pearson Addison-Wesley, 2014, ISBN 9789332518667.
URL https://books.google.sk/books?id=KV4KrgEACAAJ

[14] patrickcjk: imgui-notify. Stránka https://github.com/patrickcjk/imgui-notify.

[15] Cornut, O.: Dokovacia verzia Dear Imgui. Stránka https://github.com/ocornut/imgui/tree/docking.

[16] mlabbe: Native File Dialog. Stránka https://github.com/mlabbe/nativefiledialog.

[17] btzy: Native File Dialog Extended. Stránka https://github.com/btzy/nativefiledialog-extended.

[18] samhocevar: Portable File Dialogs. Stránka https://github.com/samhocevar/portable-file-dialogs.

[19] vareille: tiny file dialogs. Stránka https://sourceforge.net/projects/tinyfiledialogs/files/.

[20] Zoznam projektov na dialógové okná pre výber súborov. Stránka https://github.com/ocornut/imgui/wiki/Useful-Extensions#file-browsers--file-dialog.

[21] dfranx: ImFileDialog. Stránka https://github.com/dfranx/ImFileDialog.

[22] Rezonality: Zep - A Mini Editor. Stránka https://github.com/Rezonality/zep.

[23] BalazsJako: ImGuiColorTextEdit. Stránka https://github.com/BalazsJako/ImGuiColorTextEdit.

[24] Seward, J.; Valgrind Development Team: Valgrind. Stránka https://valgrind.org.

[25] LLVM Developer Group: LeakSanitizer. Stránka https://clang.llvm.org/docs/LeakSanitizer.html.