VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZŠÍRENIE NÁSTROJA ANACONDA PRE DYNAMICKÚ ANALÝZU PARALELNÝCH PROGRAMOV

AN EXTENSION OF THE ANACONDA TOOL

FOR DYNAMIC ANALYSIS OF CONCURRENT PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MICHAL HORŇÁK**
AUTHOR

VEDOUCÍ PRÁCE **Prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Horňák Michal**
Obor: Informační technologie
Téma: **Rozšíření nástroje ANaConDa pro dynamickou analýzu paralelních programů**
An Extension of the ANaConDa Tool for Dynamic Analysis of Concurrent Programs
Kategorie: Analýza a testování softwaru

Pokyny:

1. Seznamte se s problematikou testování a dynamické analýzy paralelních programů s využitím vkládání šumu a extrapolující analýzy.
2. Seznamte s nástrojem ANaConDa pro dynamickou analýzu C/C++ programů na binární úrovni.
3. Navrhněte rozšíření možností nástroje ANaConDa o vhodně zvolené extrapolující analýzy a/nebo mechanismy vkládání šumu, které dosud nejsou v tomto nástroji podporovány.
4. Implementujte navržená rozšíření a otestujte je na testovacích případech nástroje ANaConDa, případně doplňte další vhodné testovací případy.
5. Shrňte a diskutujte dosažené výsledky a možnosti jejich dalšího rozšíření v budoucnu.

Literatura:

- Fiedor, J., Hrubá, V., Křena, B., Letko, Z., Ur, S., Vojnar, T.: Advances in Noise-based Testing of Concurrent Software, In: STVR, 25(3), Elsevier, 2015.
- Fiedor, J., Vojnar, T.: ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level, In: Proc. of RV'12, LNCS 7687, Springer, 2012.
- Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection, In: Proc. of PLDI'09, ACM, 2009.
- Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multi-threaded Programs, In: Proc. of SOSP'97, ACM, 1997.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání a alespoň začátek práce na bodě třetím.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vojnar Tomáš, prof. Ing., Ph.D.**, UITS FIT VUT
Konzultant: Fiedor Jan, Ing., UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Cieľom tejto práce bolo implementácia algoritmu FastTrack pre dynamickú analízu viac-vláknových programov v jazyku C/C++. Ide o algoritmus detekujúci chyby typu data race. Je založený na relácii happens-before zakódovanej do tzv. vektor-klokov. Tie umožňujú extrapolovať beh programu a odhaľovať tak potenciálne chyby, ktoré sa v aktuálnom behu nevyskytli, ale v iných exekúciách by sa mohli vyskytnúť. Algoritmus je implementovaný v prostredí ANaConDA. Jedná sa o nástroj slúžiaci pre jednoduchšie implementovanie dynamických analizátorov monitorujúcich paralelné programy na binárnej úrovni. ANaConDA poskytuje analyzátorom potrebné informácie o behu programu, ktoré detektory následne využívaju k odhaľovaniu chýb.

Abstract

The main goal of this thesis is to implement algorithm FastTrack for dynamic analysis of multi-threaded programs in C/C++. FastTrack is algorithm which detects data race errors. It is based on happens-before relation encoded into the vector-clocks. Vector-clocks allows extrapolation of the execution which improves detection of potential errors, which were not seen in the actual run of the program however in other executions they could cause problems. Algorithm is implemented into the framework ANaConDA. ANaConDA is a tool for implementation of dynamic analyzers of parallel programs on binary level. It provides neccessary run time program informations for detectors use to discover concurency errors.

Kľúčové slová

dynamická analýza, FastTrack, ANaConDA, C, C++, data race, viac-vláknové programovanie, paralelné programy, vektor-klok, Djit+

Keywords

dnamic analysis, FastTrack, ANaConDA, C, C++, data race, multithreading, parallel programs, vector-clock, Djit+

Citácia

HORŇÁK, Michal. *Rozšírenie nástroja ANaConDA pre dynamickú analýzu paralelných programov*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Prof. Ing. Tomáš Vojnar, Ph.D.

Rozšírenie nástroja ANaConDA pre dynamickú analýzu paralelných programov

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána prof. ing. Tomáša Vojnara, Ph.D. a odborného konzultanta Ing. Jana Fiedora. Uviedol som všetky literárne pramene, zdroje a publikácie, z ktorých som čerpal.

.....
Michal Horňák
18. mája 2017

Poďakovanie

Rád by som sa poďakoval prof. ing. Tomášovi Vojnarovi, Ph.D., za poskytnutie možnosti pracovať na vývoji nástroja ANaConDA a Ing. Janu Fiedorovi za poskytnutú pomoc, čas a trpezlivosť ktorú mi venoval pri konzultáciách.

Obsah

Kapitola 1

Úvod

V súčastnosti sú viac-jadrové procesory výpočetným centrom každedého osobného počítača. Rozvoj takýchto procesorov priniesol viacero techník pracujúcich na báze súbežných výpočtov ako viac-procesorové programovanie (multi-processing) alebo viac-vláknové programovanie (multi-threading), ktoré umožňujú využívanie novo-vzniknutých výpočetných zdrojov. Takéto paradigma zvyšuje rýchlosť, s akou sa jednotlivé operácie v počitači vykonávajú vďaka paralelizácii, ktorá umožňuje množstvu činností prebiehať súčastne. Pochopitelne, so zvyšujúcim sa počtom operácií prebiehajúcich súbežne, musia aj vlákna, ktoré sprostredkovavávajú tieto činnosti, zdielať určité dáta s ostatnými vláknami. S takýmto nárastom komplexnosti prichádza aj značná množina rizík vznikajúcich:

- pri komunikácii s ostatnými vláknami pri prístupe ku zdieľaným dátam (data race) [2]
- pri prerušení vykonávania operácie vláknom a zmene kontextu v nepraví čas (atomicity violations) [10]
- pri prístupe k dátam, kedy vznikne kruhová zavislosť medzi vláknami čakajúcimi na uvoľnenie zdrojov, a tak žiadne z nich nie je schopné sa k dátam dostať (deadlock) [2]

K odhaľovaniu vyššie zmienených chýb boli vyvinuté techniky popísané v ďalších častiach tejto práce.

1.1 Téma práce

Táto práca sa zaoberá problémom chýb vznikajúcich v paralelných programoch, prevažne ich detekciou pomocou dynamickej analýzy [3], ktorá monitoruje program počas jeho exekúcie.

Cieľom je implementácia algoritmu pre dynamickú analýzu C/C++ programov nazývaného FastTrack[5] v prostredí ANaConDA[4]. ANaConDA je nástroj slúžiaci pre jednoduchšie implementovanie a využívanie dynamických analyzátorov k analyzovaniu bežiacich programov na binárnej úrovni. Toto prostredie zabezpečuje komunikáciu a poskytuje zdroje a informácie o analyzovanom programe pre daný dynamický algoritmus.

1.2 Motivácia

Motiváciou tejto práce je nedostatok nástrojov pre dynamickú analýzu C/C++ programov na binárnej úrovni. RoadRunner[6] už obsahuje implementáciu algoritmu FastTrack, no slúži len na analýzu java programov. V tejto práci ide o implementáciu tohto algoritmu pre C/C++ programy, naprogramovanom rovnako v jazyku C/C++.

Jedným z vyššie spomínaných nástrojov je ANaConDA. Problémom dynanickej analýzy je závisloť na konkrétnej exekúcii, kedy sa chyba prejaví len v malom počte z veľkého množstva behov programu. FastTrack umožňuje rýchlu detekciu data race chýb pomocou extrapolácie tzn., že dochádza k odvodeniu z jednej exekúcie množstvo podobných a vo všetkých naraz hľadá chyby.

Implementačný jazyk

Hlavným motívom využívania jazykov ako sú C alebo C++ je dosiahnutie čo najvyššieho výkonu programu. Nástrojov na analýzu takýchto C/C++ programov je však málo a často krát nie sú určené pre analyzovanie skutočných programov z praxe. Nástroj ANaConDA predstavuje prostredie pre dynamickú analýzu práve programov implementovaných v jazykoch C a C++. Repertoár algoritmov, ktoré implementuje ANaConDA nezahŕňa niektoré významné analyzátori, čo bolo jednou z motivácií tejto práce.

Vlastnosti analyzátorov

Ďalšou motiváciou pre výber práve algoritmu FastTrack je jeho presnosť a rýchlosť. Existujú analyzátory ako napríklad Djit+[14] , ktorý sítce predstavuje presnú detekciu konkurentných chýb no vytvára veľkú pamäťovú a výpočtovú záťaž. Na druhej strane je algoritmus AtomRace[10], ktorý detekuje chyby rýchlo, avšak len tie, ktoré v exekúcii nastali.
FastTrack kombinuje odľahčenú verziu Djit+ algoritmu čím redukuje záťaž a presnosť detekcie konkurentných chýb. Jeho implementácia v rozhraní ANaConDA umožňuje rýchlu analýzu programu s garanciou odhalenia prvého konkurentného prístupu ku každej zdieľanej pamäťovej položke.

Kapitola 2

Verifikácia viacvláknových programov

Táto kapitola sa zaoberá princípmi komunikácie a synchronizácie v konkurentných programoch, typmi chýb, ktoré môžu v takýchto programoch nastať/vyskyrujú sa a základnými prístupmi pre detekciu konkurentných chýb.

2.1 Viacvláknové programy

Schopnosť spúšťať viac ako jeden proces v rovnaký čas sa nazýva multi-processing. Proces pozostáva z vykonávaného programu a vlastného vyhradeného pamäťového priestoru. Vlákna sú odľahčená verzaia procesov, predstavujúca najmenšiu jednotku inštrukcií manažovatelných plánovačom, kde v jednom procese existuje viacero vlákien ktoré delia záťaž do oddelených častí a umožňujú vykonávanie jednotlivých inštrukcií paralelne. Vlákna, ako odľahčená modifikácia procesov ponúkajú menšiu flexibilitu, no umožňujú rýchlejšiu inicializáciu a rýchlejšie prepínanie kontextu. V rovnakom procese zdieľajú spoločný pamäťový priestor (heap) ale každému vláknu je zároveň priradený aj vlastný vyhradený priestor (tvz. thread-local storage) resp. zásobník, ktorý je individuálny pre každé vlákno. Moderné objektovo-orientované programovacie jazyky dovoľujú programátorom vytvárať viacvláknové programy, ktoré však výrazne zvyšujú šancu vzniku chýb v kóde. Konkurentné chyby sú jednoduché na vytváranie, no na druhej strane náročné na odhaľovanie.

Príkladom môže byť triviálny program, ktorý inkrementuje hodnotu zdieľanej premennej x prostredníctvom vlákien $t1$ a $t2$. Obe vlákna vykonávajú príkaz $x = x + 1$. Táto operácia však môže pozostávať z viacerých, dielčích inštrukcií. Ak nie sú prístupy do pamäte synchronizované, môže nastať situácia, kedy si obe vlákna načítajú súčastne inicializačnú hodnotu premennej nezávisle na sebe a následne ju inkrementujú. Na spoločnom výstupe bude mať x hodnotu 1, z dôvodu súbežného prístupu k premennej bez dodatočnej synchronizácie. Dochádza teda k chybe nazývanej porušenie atomicity (2.2.2). Riešením takéhoto problému je aby jednotlivé vlákna pristupovali ku premenným jednotlivo alebo aby operácie čítanie a zápis boli atomickými.

2.1.1 Komunikácia a synchronizovanie v konkurentných programoch

Jenou z možností ako zabezpečiť program pred konkurentnými prístupmi v zdieľanej pamäti je požadovať, aby každé vlákno najskôr zabralo synchronizačný zámok chrániaci prístup ku premennej a následne ku nej pristupovalo. Takýto spôsob však vytvára zbytočnú

výpočtovú záťaž. Lepší spôsob je zohľadňovať ďalšie synchronizačné primitíva a špecifické situácie, pri ktorých nie je potrebné uskutočňovať synchronizáciu a vytvárať výlučný prístup
len v prípadoch kedy to je nutné.

Pri exkluzívnom prístupe vlákna k premennej sú ostatné vlákna blokované a nemôžu k zdieľanému zdroju pristupovať. Vlákno musí najskôr zabrať zdieľanú položku pre seba, a tým blokovať ostatné prístupy. Blokovanie zabezpečujú rôzne typy synchronizačných primitív vymenovaných nižšie. Ak je položka zamknutá, musí vlákno počkať kým sa zdroj odomkne a následne ho môže zabrať. Takáto situácia kedy viacero uchádzačov "bojujeö rovnaké zdroje negatívne vplíva na výkon viac-vláknových programov, lebo dochádza ku sekvenčnému prístupu, a tým zároveň ku serializácii výpočtu a degradácii výkonu. Sekcia, ktorá je vykonávané mezdi dvoma prvkami zabezpečujúcimi výlučný prístup sa nazýva kritická sekcia.

Na to, aby vlákna mohli komunikovať, potrebujú synchronizovať svoje operácie, a tým zdieľať informácie s okolím. Vlákna alebo procesy spolu môžu komunikovať cez správy odosielané cez spoločný kanál alebo si môžu vymieňať údaje cez zdieľanú pamäť. Komunikácia pomocou správ je zvyčajne rozšírená v synchronizácii medzi viacerými procesmi (zvyčajne v distribuovaných systémoch [9]). Zdieľaná pameť je preferovaná u vlákien, kde v jednom procese umožňuje vláknam komunikovať medzi sebou.

Operačný systém poskytuje nízko-úrovňové synchronizačné mechanizmi:

- spinlock
- semafór
- zámok
- barriéra
- podmienené premenné (condition variables) atď.

Programovacie jazyky ďalej vytvárajú pokročilejšiu synchronizáciu ako nadstavbu nad funkciami OS. V C,C++ jazykoch je sychnronizácia vykonávaná prostredníctvom volaní synchronizačných funkcií. Preto analyzátory, ktoré analyzujú C/C++ programy musia stopovať a monitorovať volania funkcií a hodnoty ich parametrov, ktoré reprezentujú synchronizačné primitíva, aby dokázali odhaliť kedy nastáva synchronizácia a nad akým synchronizačným objektom. Takéto stopovanie je náročné na implementáciu, preto sú analyzátory integrované do prostredí, ktoré takéto problémy riešia globálne pre všetky detektory, ako je tomu aj v nástroji ANaConDA [4].

Existuje viacero metód ako v programe vytvárať synchronizáciu:

- globálne zámky
- jemné zamykanie (fine-grained locking)
- transakčná pamäť [7]

Globálne zámky sú využívané globálne naprieč procesom. Jedno vlákno zamkne globálne prístup do zdieľanej pamäte a ostatné vlákna musia počkať na jej odblokovanie. Takýto prístup je jednoduchý na použitie no s ním aj degraduje výkon aplikácie kvôli serializácii prístupov.

Ďalším spôsobom je jemné zamykanie, predstavujúce snahu o zamykanie prístupov pre každú zdieľanú položku zvlášť a len na miestach kde je to nutné. Určiť nutnosť existencie zámku v danej situácii však nie je jednoduché, preto je tento prístup náchylný na množstvo chýb. Pri zlom poradí zamykania môže vzniknúť deadlock alebo pri zlom zámku nemusí vzniknúť žiadna synchronizácia z čoho vznikajú data race chyby (2.2.1). Oproti globálnym zámkom však pri správnej synchronizácii predstavuje výrazný nárast výkonu.

Tretím spôsobom je využitie transakčnej pamäte . Tá monitoruje sekvencie inštrukcií načítania (load) a uloženia (store) do pamäte, ktoré sú vykonané atomicky. Ak dôjde ku porušeniu atomicity, je výsledok všetkých operácií anulovaný a hodnoty v pamäti sú navrátené do stavu pred vykonaním celej sekvencie. Transakčná pamäť alternatívu ku synchronizácii pomocou zámkov. Je kombináciou jednoduchého používania a následného dobrého výkonu programu, no nevylučuje existenciu chýb.

2.2 Chyby vo viac-vláknových programoch

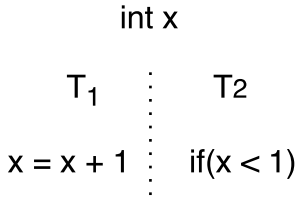
Viac-vláknové programy vytvorili možnosť pre príchod nepravidelných, konkurentných chýb[2], ktoré sú veľmi ťažko detekovatelné pomocou bežných testovacích metód. Hlavným problémom je že viac-vláknové programy môžu byť vykonané rozlišne pri násobnom spustení aplikácie s rovnakou inicializáciou ale nedeterministickým usporiadaním vlákien.

2.2.1 Data race

Časovo závislá chyba nad dátami (data race)[8], je jednou z najčastejšie sa vyskytujúcich chýb v konkurentných programoch. K identifikácii výskytu data race chyby v behu konkurentného programu, je nevyhnutné určiť (1) ktoré premenné sú zdieľané pre akúkoľvek dvojicu vlákien (2) či akákoľvek dvojica prístupov k danej premennej je synchronizovaná.

Definícia 1 *V programe sa nachádza data race ak dve predom nesynchronizované vlákna pristupujú ku zdieľanej premennej súčastne a minimálne jedna operácia je zápis.*

Nasledujúci obrázok 5.3 zobrazuje situáciu kedy za určitých okolností môže dôjsť ku chybe v súbežnom prístupe do pamäte. Jedna z možností ako by sa program mohol správať je, že by sa operácia porovnania vo vlákne T_2 vykonala ako prvá, ktorá predstavuje nedefinovanú premennú a zostáva na programovacom jazyku ako sa k tejto situácii zachová. Druhou možnosťou je priebeh v poradí priradenie nasledované pdmienkou. V tomto prípade by potenciálna chyba zostala nepovšimnutá. Poslednou možnosťou je problém atomických operácií vysvetlovaných už na príklade v sekcii 2.1. Ako sa detekujú takéto chyby pomocou dynamickej analýzy je podrobnejšie vysvetlené v sekcii 3.1.



Obr. 2.1: Obrázok zobrazuje pseudokód programu, v ktorom sa vyskytuje potenciálna data race chyba.

2.2.2 Porušenie atomicity

Atomická operácie predstavuje vykonanie jednotného príkazu a to tak, že systém rozpoznáva len 2 stavy: stav pred a stav po vykonaní atomickej operácie a nevidí žiaden medzistav. Porušenie atomicity je často spájané a nekorektne označované ako data race chyba. Navonok jednotné a atomicky pôsobiace operácie môžu pozostávať z viacerých inštrukcií, ktoré sa vykonávajú samostatne, sériovo. Ak sú takéto neatomické operácie vykonávané paralelne, môže dôjsť ku nesynchronným prístupom do pamäte počas vykonávania nezosynchronizovaných atomických operácií tvoriacich jednotný zdanlivo atomický príkaz.

Definícia 2 *Beh programu porušuje atomicitu, ak nie je možné nájsť ekvivalentný beh v ktorom všetky atomické sekcie sú vykonané sériovo.*

2.2.3 Ostatné chyby

Nasledujúce chyby[2] v konkurentných programoch predstavujú nemenej významné problémy, no z hľadiska tejto práce sú okrajové.

Uviaznutie (deadlock) - uviazdnutie predstavuje situáciu, v ktorej dva a viac procesov (v našom prípade vlákien) čaká na podmienky, ktoré nemôžu nastať. Dve vlákna sa nachádzajú v stave uviaznutia, ak prvé vlákno je blokované a čaká na udalosť, ktorá by ho odblokovala, ale táto udalosť je vykonaná druhým vláknom, ktoré rovnako čaká v blokovanom stave na udalosť sprostredkovanú prvým vláknom - cyklická závyslosť medzi prvým a druhým vláknom.

Nesprávne poradie - k takejto chybe dochádza v prípade, kedy nastáva porušenie určitých požiadaviek na poradie vykonávaných operácií (napr. súbor musí byť otvorený skôr ako používaný) a tieto akcie nenastanú v očakávanom poradí.

Zmeškaný signál - zmeškaný signál predstavuje správu, ktorá mala byť doručená určitému vláknu, no doručenie nenastalo. Buď bola správa odoslaná inému vláknu, alebo skôr, ako na ňu správne vlákno začalo čakať.

Blokované vlákno - blokované vlákno čaká na udalosť v programe, ktorá by ho odblokovala no v behu programu takáto udalosť nikdy nenastane.

Uviaznutie s aktívnym čakaním (livelock) - rozdiel medzi livelock a deadlock je, že pri livelock sa vykonáva naviac (napríklad cyklus), teda niečo robia ale vlákna nie sú označené ako blokované a aktívne čakajú na odblokovanie.

2.3 Metódy pre analýzu programov

2.3.1 Programové testovanie

Jedným z najbežnejších prístupov k vyhľadávaniu chýb v software je programové testovanie. Účelom testovanie je odhalenie chýb, ktoré vedú ku neočakávanému správaniu programu. Bežným postupom pri testovaní je systematické spúšťanie testovacích prípadov, ktoré tvoria jednotlivé testy a ich vstupy a výstupy. Výstupom testovacieho procesu je odpoveď áno-/nie, či program alebo programová časť funguje správne/nesprávne. Avšak nájsť testovacie prípady, ktoré overia dostatočnú časť správania sa programu, nie je jednoduché.

Problém odhaľovania chýb v súbežných programoch pomocou testovania je vo všeobecnosti obtiažny z dôvodu veľkého množstva preložení vlákien a prirodzenému nedeterminizmu v plánovaní poradia vykonávania operácií vláknami počas exekúcie programu. Teda viacero exekúcií totožného programu s rovnakými parametrami, môže viesť k rôznym výsledkom. Preto jediné spustenie a otestovanie programu nie je dostačujúce pre evaluovanie korektnosti aplikácie.

2.3.2 Dynamická analýza

Dynamickú analýzu[15][14] je možné využívať aj bez znalosí zdrojového kódu, ktorý je naopak potrebný pri staticke analýze, a tak nie sú potrebné pre analýzu implementačné detaily monitorovanej aplikácie. Ide o metódu založenú na stopovaní priebehu programu (program tracing), ktorá zbiera informácie o aktuálne spustenej instancii programu, a tie sú analyzované za účelom odhalenia chýb. Navzdory tomu, že analýza pracuje s informáciami pochádzádzajúcimi len z jediného behu, často krát dokáže odhaliť chyby, ktoré nie sú priamo vykonávané v spustenom programe vďaka extrapolácii. Nevýhodou dynamického monitorovania kódu je problém prehliadnutia kritických scenárov aplikácie, pretože analýza sleduje len aktuálny beh, ale nemonitoruje behy, ktoré by mohli nastať zmenou použitia programu a najmä zmenou preloženia vlákien a tým spôsobenou zmenou vykonávania operácií. Výhodou pozorovania len jednoho behu programu za jeho exekucie je dobrá škálovateľnosť čím zvládne pomerne rozsiahle programy a umožňuje prácu s presnými informáciami.

- – Analýza jednoho behu
- \+ Dobrá škálovateľnosť
- \+ Presné informácie o behu

2.3.3 Statická analýza

Statická analýza[1] je prístup k overovaniu programov z oblasti formálnej verifikácie. Je založená na princípe získavania informácií zo zdrojového kódu, bez toho aby bol program spustený. Oproti predchádzajúcim metódam verifikácie software, nie je statická analýza závislá na jednom spustení programu, čo jej umožňuje v teórii preskúmať aj cesty programu, ktoré by boli veľmi ťažko dostuné pri bežnom testovaní. Táto skutočnosť však znamená, že je potrebné preskúmať exponenciálny počet možných preložení a usporiadavaní vykonávania udalostí, čo však v praxi nie je príliš reálne. Statická analýza vychádza z kódu analyzovanej aplikácie, k čomu však nevyžaduje prostredie programu (knižnice, stupy, výstupy atď.), tým prispieva k vysokej miere možnej automatizácie. Keďže táto technika neprebieha počas behu programu (pracuje öffline"), znamená to tiež, že využíva na analýzu len zdroje poskytované z kódu programu, čo v praxi znamená stratu špecifických informácií o programe (ako napríklad konkurentné informácie o vstupoch a výstupoch s ktorými môže program pracovať) a následne možný vznik falošných alarmov. Falošné alarmy predstavujú chyby, ktoré sa v programe vôbec nevyskytujú, lebo statická analýza overuje behy so vstupmi a výstupmi, ktoré nikdy nenastanú.

+ Analýza všetkých možných exekúcií
− Slabá škálovateľnosť pri analýze
− Minimálne informácie o behu

2.3.4 Model checking

Model checking[13] je ďalšou, plne automatizovanou technikou formálnej verifikácie na odhalovanie chýb vo viac-vláknových programoch. Táto metóda je založená na princípe zjednodujšeného modelu zdrojového kódu a postupnom priechode všetkými jeho stavmi. Problémom tejto metódy overovania software je množstvo stavov, v ktorých sa systém môže nachádzať, a ktoré treba monitorovať, čo vytvára výraznú časovú a pamäťovú náročnosť.

Kapitola 3

Dynamická analýza

Dynamická analýza spomínaná už v 2.3.2, predstavuje populárnu metódu verifikácie konkurentného software, najmä z hľadiska relatívne jednoduchého nasadzovania (spustím a sledujem) a dobrej škálovatelnosti umožnujúcej analyzovať aj komplexné programy. Okrem sledovania výstupov behu programu je možné takpovediac vstúpiť do programu, preskúmať a zozbierať určité informácie na základe ktorých je možné určiť či v programe existuje potenciálne miesto vzniku chyby. Táto technika vkladania dodatočného kódu do programu na konkrétne miesta sa volá inštrumentácia kódu [4]. Odhalenie potenciálneho miesta chyby práve preto, lebo dynamická analýza dokáže rozšíriť princíp testovania takým spôsobom, že extrapoluje a vyhodnocuje nie len udalosti v konkrétnom behu programu, ale zároveň aj množstvo ďalších exekúcií odvodených od inicializačného behu. Avšak pri veľkej abstrakcii (odvodení ďalších behov) nie všetky hlásenia sú hlásenia o chybe v programe, ale typicky ide o chyby v abstrakcii, čo predstavuje falošné alarmy.

3.1 Detekovanie konkurentných chýb

Dynamická analýza sa využíva na detekovanie najbežnejších konkurentných chýb spomenutých v sekcii 2.2. Táto práca sa zaoberá odhaľovaním data race chýb pomocou FastTrack algoritmu, a preto sa aj aktuálna sekcia primárne sústreďuje na bližšie definovanie data race stavov a ich odhaľovanie v paralélnom software.

3.1.1 Data Race chyby

Jedným z najznámejších problémov konkurentných programov sú chyby v súbežnosti (chyby typu data race). Jedná sa o chybu simultánneho prístupu vlákien do zdieľanej pamäte. Pretože tieto chyby spôsobujú nepredvídatelné správanie programu, ich odhaľovanie je veľmi dôležité pri testovaní software. Techniky pre detekovanie data race chýb sú založené buď na princípe locksetov alebo happens-before relácie.

Lockset analyzátory

Lockset algoritmy sú založené na princípe synchronizácie pomocou zámkov. Podmienkou je, aby každá zdieľaná pamäťová položka mala definovaný výlučný prístup, takže nemôže dôjsť k súbežnému prístupu dvoch konkurentných vlákien, a teda nemôže nastať data race chyba. V základe Lockset algoritmus kontroluje zamykanie premenných aby dochádzalo len ku výlučným prístupom. Porušenie takejto logiky však nemusí znamenať data race

chybu, z čoho vzniká problém hlásenia veľkého množstva nevyžiadaných falošných chýb, ktoré skrívajú reálne data race chyby. Princíp algoritmu Lockset je využívaný v analyzátore Eraser [15] ktorý vykazuje veľmi dobré výsledky.

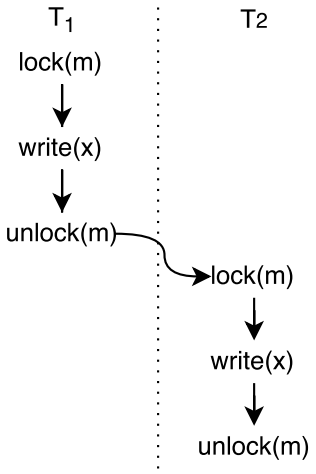
Happens-before (H-B) analyzátory

Happens-before relácia sa využíva pre určenie čiastočného zoradenia udalostí v paralelných programoch, bez využitia fyzických hodín. Happens-before relácia na princípe distribuovaných systémov je popísana v [9] nasledovne. Uvažujeme systém pozostávajúci z procesov obsahujúcich kolekciu sekvencie udalostí. Jeden proces je definovaný ako sada udalostí s úplnym usporiadaním, teda udalosti sa vykonávajú postupne v poradí za sebou (a *happens before* b). Predpokladáme že zaslanie alebo prijatie správy je jedna udalosť v procese, potom môžme definovať "happenes before"reláciu, zapísanú ako " $\rightarrow$ " takto:

Definícia 3 (H-B) *Relácia* $\rightarrow$ *medzi skupinou udalostí viacerých procesov je najmenšia relácia spĺňujúca tieto tri podmienky:*

- *ak* a *a* b *sú udalosti v rovnakom procese a* a *predchádzalo* b*, potom platí* $a \rightarrow b$ *(program order)*
- *ak* a *je odosielateľ a* b *je adresát rovnakej správy (udalosti) ale v inom procese, potom* $a \rightarrow b$
- *ak* $a \rightarrow b$ a $b \rightarrow c$ *potom* $a \rightarrow c$ *(tranzitivita)*

Dve nezhodujúce sa udalosti a a b *sú považované za konkurentné ak* $a \nrightarrow b$ a $b \nrightarrow a$



Obr. 3.1: Lamportova H-B relácia

V prostredí paralelných (viac-vláknových) programov sa jedná o podobný princíp. V každom vlákne sú udalosti usporiadané sekvenčne za sebou. Medzi vláknami sú udalosti zoradené na základe toho, ako dochádzalu ku synchronizovaniu medzi vláknami a aké synchronizačné primitíva boli využité. Ak jedno vlákno pristupuje ku synchronizačnému objektu a ďalší prístup prichádza z iného vlákna, potom je prvý aj druhý prístup synchronizovaný a definovaný ako H-B relácia medzi prvým a druhým vláknom. Ku príkladu na obrázku 5.2 je znázornený úsek synchronizácie dvoch vlákien. Treba zdôrazniť, že uzamknutie objektu

v poradí ako je na obrázku je len jedna z možných variant. V inom prípade by vlákno T_2 mohlo zabrať zámok m ako prvé. Všetky tri operácie vo vlákne T_1 sú zoradené podľa happens-before relácie, lebo sú spúšťané sekvenčne v rámci jednoho vlákna. Udalosť $lock(m)$ vlákna T_2 dodržiava happens-before reláciu s operáciou $unlock(m)$ vlákna T_1, lebo zámok m môže byť vlastnený maximálne jedným vláknom a nemôže byť zabratý pred tým, ako ho uvolní aktuálny vlastník. Nakoniec tri operácie odohrávajúce sa vo vlákne T_2 sú rovnako zoraden ako v prípade T_1.

Ak dve vlákna pristupujú k rovnakej premennej a prístupy nie sú zoradené podľa happens-before relácie, potom v ďalších behoch programu môže dôjsť k zmene rýchlosti vykonávania operácií vlákanmi a vzniká riziko simultánneho prístupu, čo pri určitých prípadoch kedy je aspoň jeden prístup zápis, môže nastať až data-race chyba. Algoritmy založené na takomto princípe využívajú zakódovanie happens-before relácie v podobe vektor-klokov.

3.2 Vkladanie Šumu

Vkladanie šumu [3] je technika využívaná v prostredí testovania a dynamickej analýzy pre zvyšovanie šance na odhalenie chýb ovplivňovaním plánovania vlákien programu. V princípe ide o to, vo vhodný okamžik pozastaviť alebo spomaliť vlákno pomocou šumu a umožniť ostatným vláknam postupovať ďalej vo vykonávaní kódu. Tým pádom dochádza ku zmene poradia vykonávaných operácií programu a zvyšuje sa tak šanca pozorovať scenáre, ktoré by mohli viesť ku chybám, ale nenastali by za bežných podmienok prostredia.

Šum sa vkladá do programu pomocou takzvanej inštrumentácie bližšie vysvetlenej v sekcii 4.1. Ide o vloženie kódu generujúceho šum do sledovaného programu a tým pádom táto metóda nevyžaduje žiadnu modifikáciu originálnych testov a prostredia, v ktorom sa program spúšťa.

Pri vkladaní je potrebné riešiť dva problémy: (1) Kedy vytvárať šum a (2) akým spôsobom ovplivniť program (aký typ šumu vložiť). Pri vyberaní miesta inštrumentácie šumu je dôležité zvážiť či na danom mieste bude mať efekt. Vkladať šum na miesta v kóde, ktoré neovplivňujú synchronizáciu alebo komunikáciu medzi vláknami je bezpredmetné. Z tohto dôvodu je vhodné vyberať miesta ovplivňujúce konkurentné správanie ako je spúšťanie udalostí pôsobiacich na synchronizáciu a prístupy do zdieľanej pamäte. Pri vložení monitorovacieho kódu pred každú inštrukciu môže vznikať problém vyrušenia šumu, kedy sa jednotlivé inštrumentované rutiny vyrušia a dochádza v súčte len ku veľkému spomaleniu. Existujú rôzne heuristické metódy pre určenie miest pre inštrumentovanie šumu. Jednoduchým a lacným riešením je využiť pseudo-náhodného generátoru a pomocou neho vkladať šum náhodne. Do programu sa vkladajú rôzne typy šumu:

- Zastavenie vlákna na určitú dobu alebo do určitej udalosti (sleep/wait)
- Okapované volanie inštrukcie pre uvolnenie procesoru (yield)
- Vykonávanie nadbytočného kódu (busy-wait)

Úspešnosť analýzy závisí na behu, ktorý algoritmus videl. V kombinácii šumu, ktorý umožňuje vidieť aj iné scenáre ako tie za bežného testovania s rovnakým prostredím a dynamickou analýzu, ktorá to obalí do abstrakcie umožňujúcej monitorovať aj okolité scenáre, rastie šanca na odhalenie chyby.

Kapitola 4

ANaConDA

ANaConDA - "adaptable native-code concurrency-focused dynamic analysis"[4], je platforma pre vytváranie dynamických analyzátorov, postavená nad nástrojom PIN [11]. Umožňuje monitorovanie paralelných C/C++ programov na binárnej úrovni za účelom odhalenia konkurentných chýb vo viac-vláknových aplikáciách(2.2). Cieľom nástroja je zjednodušenie vytvárania dynamických analyzátorov pre analyzovanie viac-vláknových C/C++ programov. Analyzátorom poskytuje monitorovaciu vrstvu, ktorá zbiera informácie o bežiacom programe a interpretuje oznámenia o dôležitých udalostiach ako sú synchronizácia alebo prístupy do pamäte, čím do seba zabaľuje laicky "bonzovaciu"vrstvu a dovoľuje ju využívať analyzátrom, a tým zjednodušuje vývoj ďalších analyzátrov. ANaConDA zároveň poskytuje injektovanie šumu 3.2 do programu vďaka čomu umožňuje analyzátorom zlepšiť pokrytie možných situácií medzi vláknami a tým udhaliť viacej chýb.

Sondy Sondy slúžia k získaniu konkrétneho typu informácií o behu testovaného objektu. ANaConDA má možnosť sledovať množstvo informácií:

- Prístupy do pamäte
- Synchronizovanie vlákien
- Spúštanie a ukončenie behu vlákien
- Postupnosti volaní
- Volané funkcie a ich parametre

Implementované analyzátory ANaConDA v súčastnosti podporuje radu analyzátorov:

- Atomrace [10] - analyzátor na odhaľovanie chýb typu data race
- Contract-validator - detektor porušenia kontraktov[1] pre súbežnosť
- Data-validator
- Goodlock - detektor uviaznutí

[1]Kontrakty umožňujú popísať ako volať funkcie v paralelnom programe, aby nedošlo ku chybám ako je porušenie atomicity, nesprávne proadie a zmeškaný signál

- Hldr-detector - detektor High-level data race (podmnožina chýb typu porušenie atomicity)
- Tx-monitor - analyzátor práce s transakčnou pamäťou [2]

4.1 Inštrumentácia kódu

Inštrumnetácia [4] je technika pre vkladanie dodatočného kódu do aplikácie ktorý umožňuje monitorovanie či modifikovanie behu programu.

Inštrumentačné nástroje ako Pin alebo Valgrind[12] sa vo veľkej miere používajú pre zbieranie informácií o exekúcii programu. Tieto dáta môžu byť efektívne využívané pri odchytávaní chýb programov, optimalizáciách alebo verifikácii bezpečnosti.

Existujú tri možnosti kam monitorovací kód umiestniť: do zdrojového kódu, medzikódu alebo do binárneho kódu. Inštrumentácia na binárnej úrovni poskytuje značné výhody oproti ostatným vrstvám. Pri analyzovaní programu, od ktorého užívateľ nemá zdrojové kódy, môže byť veľký problém najmä pri práci s knižnicami, tie však analyzátor, ktorý pracuje pomocou inštrumentácie na binárnej úrovni nepotrebuje. Ďalšou výhodou je lepšia presnosť vkladania kódu, keďže sa pohybujeme na strojovej úrovni inštrumentačný nástroj môže vkladať kód presne na potrebné miesto a nedochádza ani k nechceným optimalizáciám kódu prekladačom, ktorý by mohol monitorovací kód rôznym spôsobom modifikovať. Inštrumentácia zdrojových kódov je značne závislá na implementačnom jazyku, čo pri binárnej neplatí. Existujú dva prístupy k binárnej inštrumentácii.

Statická binárna inštrumentácia vkladá kód do programu pred jeho spustením a tým trvalo modifikuje obsah jeho binárnych súborov. Tento prístup môže byť výhodnejší, lebo pri modifikovaní testovaného programu vkladá kód pred spustením, čím odpadá záťaž, ktorú vytvára dynamická inštrumentácia. Statická inštrumentácia však nie je schopná inštruovať zdieľané knižnice pokial nie sú inštruované oddelene od originálnych, ktoré využívajú aj iné programy a pri ich modifikovaní by dochádzalo ku chybám. Taktiež umožňuje menšiu flexibilitu, keďže inštrumentačný kód zotrváva v programe počas celej doby exekúcie.

Dynamická binárna inštrumentácia na druhej strane vkladá inštrumentačný kód až počas exekúcie programu bez toho, aby došlo k permanentným modifikáciám akéhokoľvek kódu. Dokáže pracovať s dynamicky generovaným kódom, ktorý nie je známy pred spustením programu a pre statickú analýzu je prakticky nedosiahnuteľný. Tým, že dynamická inštrumentácia pracuje s kópiami binárnych zdrojov, dáva priestor pre analyzovanie knižníc využitých v testovanom programe a súbežne dovoľuje použitie rovnakých knižníc ostatným programom, čo v prípade statickej inštrumentácie nie je možné bez udržiavania dvoch separátnych verzií knižníc a ukazateľov (linkov) na ne. Nevýhodou je už spomínaná opakovaná réžia pri každom spostení analýzy čo môže predstavovať problém najmä pri časovo alebo priestorovo náročných programoch.

[2] Transakčná pamäť je nový spôsob synchronizácie vlákien kedy je kód vykonaný paralelne a pokial bola porušená jeho atomicita, je vlákno vykonávajúce tento kód vrátené do stavu perd jeho vykonaním (rollback)

4.2 Intel PIN

Pin[11] je instrumentačný nástroj, ktorý vykonáva dynamickú (run-time) binárnu inštrumentáciu C/C++ programov. Dovoľuje nástrojom vkladať dodatočný kód (v jazykoch C/C++) na ľubovolné miesta v spustiteľnom kóde. Pin ponúka bohaté aplikačné rozhranie, ktoré abstrahuje vrstvu inštrukčnej sady a dovoľuje tak kontextovým informáciám, ako sú obsahy registrov a pamäte, aby mohli byť predané do injektovaného kódu ako parametre. Pin automaticky ukladá a obnovuje registre, ktoré boli prepísané vkladaným kódom, takže aplikácia naďalej pokračuje v činnosti. Ako nástroj dynamickej binárnej inštrumentácie vykonáva Pin inštrumentáciu počas behu preložených binárnych súborov, teda nevyžaduje rekompilovanie zdrojových kódov a podporuje inštrumnetáciu programov, ktoré dynamicky vytvárajú svoj vlastný kód. Dokáže sa dynamicky pripojiť ku bežiacemu procesu, inštrumentovať ho a odpojiť, čím redukuje výpočtové náklady najmä u veľkých systémov. Pin pracuje ako "just in time"(JIT) prekladač. Vstupom prekladača nie je bytekód ale spustiteľný súbor. Pin pozdrží vykonanie prvej inštrukcie programu a vygeneruje novú sekvenciu kódu, takmer identickú ku originálu, ale zabezpečuje, aby po vykonaní novovygenerovaného kódu opäť prevzal kontrolu a generoval ďalši kód. V JIT móde je vykonávaný len novo-vygenerovaný kód, zdrojové dáta sú využívané len ako referencie. Práve pri generovaní nového kódu Pin umožňuje užívateľovi vložiť (inštrumentovať) dnu svoj vlastný kód. Nástroj ANaConDA využíva Pin z viacerých dôvodov:

- podporuje dynamickú binárnu inštrumentáciu
- neobmedzuje viac-vláknové programy (neserializuje ich exekúciu)
- dokáže sprostredkovať inštrumentáciu zdieľaných knižníc
- zvláda seba-modifikujúci kód
- umožňuje inštrumentáciu v prostredí Windows, Linux, MacOS

4.3 Vkladanie šumu v prostredí ANaConDA

AKo bolo už spomínané v sekcii 3.2, vkladanie šumu sa zameriava na zvýšenie počtu pozorovaných usporiadaní vlákien narúšaním plánovania vlákien programu. Toto je dosiahnuté pomocou vloženia špecifického kódu do programu na niektoré miesta a cieľom je prinútit testovaný program ku prepnutiu vykonávaného vlákna na iné.

Nástroj ANaConDa implementuje vkladanie viacerých typov šumu, primárne využívané sú hlavne yield a sleep, ktoré vykazujú úspešné výsledky aj v ďalších analyzátoroch pre dynamickú analýzu [4].

- **Yield** umožňuje vláknu vzdať sa procesoru, čo dovoľuje ostatným vláknam prebrať iniciatívu a pokračovať vo vykonávaní svojej činnosti.
- **Sleep** sa vzdá procesoru a na určitú dobu sa uspí, čím zamedzí spätnému prepnutie vlákien na určitú dobu.

Pri vkladaní treba odhadnúť rozumnú mieru sily šumu. Pri vkladaní yield, sa jedná o počet volaní tejto funkcie. Pri vkladaní sleep zas treba určiť na ako dlho sa má dané vlákno uspať.

4.4 Princíp analýzy programu

V princípe ide o komunikáciu medzi analyzátorom a ANaConD-ou nasledovne. Analyzátor by vyžadoval sledovanie určitých udalostí v programe. Zaujíma sa hlavne o to čo sa deje vo vnútri.

- interné dátové štruktúry
- volania pri danej operácii
- synchronizácia (zámky)
- volania podprogramov

ANaConDA pomocou vlastných sónd riadi zber týchto informácií a predávanie ich analyzátorom. Analyzátor musí byť vo forme zdieľaného objektu (pre Linux) alebo dynamickej knižnice (pre Windows), ktoré obsahujú špecificky pre každý algoritmus implementované funkcie, a ktoré sú následne volané pri požadovaných udalostiach (synchronizácia, prístup do pamäte atď.). Analyzátor musí zaregistrovať spätné volania funkcií pre udalosti o ktoré má záujem.

Pin využíva ANaConD-u ako zásuvný modul, pomocou ktorého registruje spätné volania kdekoľvek je potrebné vložiť nový kód. Inštrumentačný komponent predstavuje vkladanie týchto spätných volaní na funkcie, ktoré neskôr monitorujú stav systému v mieste vloženého volania a tým analyzujú bežiaci program. ANaConDA spolu s Pinom umožňuje vkladanie kódu pred a po operáciách zaujmavých pre analyzátory paralelných programov, ako sú synchronizácie a prístupy do pamäte alebo vytváranie a zanikanie vlákien počas exekúcie programu. Napríklad ak Pin narazí na pamäťové operácie *read* alebo *write*, ANaConDA umožňuje analyzátorom registrovanie spätných volaní napríklad $ACCESS\_beforeMemoryRead$ alebo $ACCESS\_beforeMemoryWrite$. Na tieto volania si už analyzátor naviaže vlastné rutiny, ktoré bude pri spätnom volaní vykonávať.

Kapitola 5

Odhaľovanie data race chýb

Data race chyby(viz. sekcia 2.2.1), sú veľmi nepríjemnou súčasťou vývoja konkurentných programov. Táto kapitola sa bližšie zaoberá vyhľadávaním chýb typu data race pomocou dynamickej analýzy, detekcia chýb pomocou techniky vektor-klokov a ich využitie v analyzátore Djit+ [14], z ktorého vychádza aj analyzátor FastTrack [5] , ktorý je predmetom tejto bakalárskej práce.

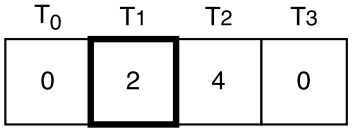
Dynamické data race detektory Vo všeobecnosti spadajú takéto analyzátory do dvoch skupín, tie ktoré ohlasujú len reálne chyby a analyzátory ktoré spolu s korektnými výstupmi hlásia aj falošné hlásenia. Vznik falošných hlásení časť kotví v pomerne veľkej abstrakcii analýzy. Presné race detektory zvyčajne využívajú myšlienku happens-before relácie na monitorovanie priebehu programu, zakódovanú pomocou tzv. vektor-klokov (5.1).

Lamportove hodiny Pri detekovaní data-race chýb v konkurentných programoch vzniká problém, ako monitorovať poradie pristupovania vlákien (procesov) ku zdielanému pamäťovému priestoru a následne určiť či môže nastať situácia kedy by pristupovalo viac vlákien ku zdielanej premennej zároveň. Každé vlákno má informácie len o svojich vlastných udalostiach a nemá žiadne poznatky o svojom okolí. Lamportove logické hodiny [9] sa zakladajú na veľmi jednoduchej myšlienke. Všetky vláknamajú svoje vlastné lokálne hodiny vytvárajúce ku každej udalosti, ktorá nastane v danom vlákne, časové razítko. Najbežnejším prevedením takýchto hodím je jednoduché počítadlo, ktoré sa inkrementuje pri každej udalosti daného vlákna a tak vytvára inkrementujúce sa časové známky unikátne pre postupne prichádajúce udalosti. Predchádzajúca udalosť má časové razítko menšie ako nasledujúca. Táto podmienka platí pre udalosti spracované v totožnom vlákne. V konkurentných programoch existujú aj prípady pri ktorých dochádza ku medzi vláknovej komunikácii cez synchronizačné objekty. Takéto správy predstavujú čiastočné zosynchronizovanie sa medzi participujúcimi vláknami a pomocou synchronizačnej operácie si spravia obraz o okolitých vláknach v akom stave (aké hodnoty počítadla) sa nachádzajú. Happens-before relácia 3.1.1 využíva práve lamportove logické hodiny pri vyjadrovaní informácií o synchronizácii vlákien.

5.1 Kódovanie H-B relácie pomocou Vektor-klokov

Vektor-kloky sú vektory logických (Lamportových) hodín, teda rozšírenie logických hodín pre paralelné programy. Pomocou H-B relácie, ktorú kódujú, dokážu bezpečne extrapolovať beh programu, a tak vedia overiť exekúcie podobné behu, ktorý sme videli, a nájsť v ňom chyby. Pretože ide o bezpečnú extrapoláciu, potom všetky ëxtrapolované behy"by mali byť reálne a tým aj chyby v nich.

Myšlienka za monitorovaním medzivláknovej komunikácie a synchronizácie v podobe vektor-klokov je nasledujúca. Každé vlákno je identifikované vlastným unikátnym identifikačným číslo t a vlastní svoj lokálny vektor hodín T_{vc}. Pozícia vo vektore o hodnote t predstavuje vlastné hodiny (logické lamportove hodiny) daného vlákna t. Čiže t znači identifikačné číslo vlákna a táto hodnota je zároveň pozícia hodín daného vlákna vo vektror-kloku. Ostatné hodnoty v T_{vc} predstavujú logické hodiny okolitých vlákien a značia poslednú udalosť, ktorá sa stala v inom vlákne a platí pre ňu happens-before (3.1.1) relácia vzhľadom ku aktuálnej operácii vlákna t.



Obr. 5.1: Vektor-klok vlákna T_1

Vektor-kloky sú čiastočne zoradené $\sqsubseteq$ v jendotnom smere, teda pri porovnaní musia byť rovnako usporiadané. Ku spojeniu dochádza pomocou operácie $\sqcup$, ktorá vyberie maximálne hodnoty z každého elementu vektoru, definujú minimálny element $\perp_V$
a k inkrementácii využívajú funkciu *inc*. Formálny zápis operácií popísaných vyššie kde V značí vektor-klok a t,u značia identifinkačné číslo vlákna:

$$V_1 \sqsubseteq V_2 \Leftrightarrow \forall t.\ V_1(t) \leq V_2(t) \tag{5.1}$$

$$V_1 \sqcup V_2 = \lambda t.\ max(V_1(t), V_2(t)) \tag{5.2}$$

$$\perp_v = \lambda t.\ 0 \tag{5.3}$$

$$inc_t(V) = \lambda u.\ if\ u = t\ then\ V(u) + 1\ else\ V(u) \tag{5.4}$$

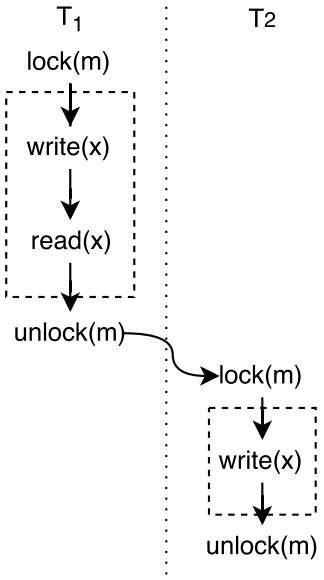
Inicializácia a inkrementácia hodín je riešená na úrovni implementácie, každý algoritmus môže myšlienku vektor-klokov reprezentovať trochu inak. Vo všeobecnosti sú si však operácie podobné. Na začiatku dochádza ku inicializácii kedy sú hodnoty hodín vo vektoroch nastavené rovnako a pri každej ne-synchronizačnej operácii dochádza ku inkrementácii logických hodín na indexe t vlákna ktoré danú operáciu vykonalo. Pri synchronizácii s okolitými vláknami nastáva kontrola happens-before relácie pomocou $\sqsubseteq$ porovnania a následnému spojeniu $\sqcup$ vektor-klokov medzi ktorými prebehla synchronizácia. Vektor-kloky bývajú implementované v rôznych podobách a dochádza ku rôznym optimalizáciám, no základný princíp rozšírených Lamportových hodín pre paralelné programy v podobe vektor-klokov zostáva rovnaký.

5.2 Djit+

Pred tým, ako si dopodrobna vysvetlíme algoritmus FastTrack 6, je podstatné zmieniť princíp analyzátora Djit+[14], na ktorého základe je postavený FastTrack. Djit+ využíva tzv. vektorové rámce, čo sú rámce založené na vektorových razítkach, ktoré slúžia na detekovanie data race chýb. Algoritmus pracuje s logovacím mechanizmom, ktorý je schopný dynamicky zaznamenávať prístupy k položkám v zdielanej pamäti. Cieľom algoritmu je sledovanie prístupov do zdielaných zdrojov a monitorovanie či všetky operácie splňujú happens-before reláciu. Djit+ je renomovaná verzia Djit algoritmu, u ktorého vylepšuje hlavne nedostatok detekcie viacerých chýb (Djit umožňuje detekovať len prvú data race chybu v behu).

5.2.1 VC v Djit+

Oproti klasickej implementácii vektor-klokov, Djit+ pracuje s upravenou verziou, kde miesto toho aby pri každej udalosti inkrementoval hodnotu lokálnych hodín, mení ich hodnotu len v určitej situácii. Priebeh exekúcie vlákna si je možné predstaviť ako sekvenciu časových rámcov. Ide o optimalizovanie monotónnej inkrementácie lokálnych hodín, ktorá sa deje len v čase synchronizačnej operácie, kedy dochádza ku zosynchronizovaniu vlákna s okolím. Vektorove rámce využívajúce vektorové razítka sú v princípe to isté ako pojmy vektor-kloky rozširujúce logické hodiny využívané napríklad u FastTrack-u.



Obr. 5.2: Vektorové rámce, čiarkovane sú vyznačené úseky počas ktorých nedochádza k inkrementácii lokálneho čítača(razítka), ale až po ukončení bloku (rámca)

5.2.2 implementácia VC v Djit+

Definícia operácií a konvencí zápisov

Program pozostáva z viacerých konkurentných vlákien s jedinečným identifikátorom $t \in Tid$, kde Tid je množina všetkých identifikátorov vlákien. Každé vlákno t si drží vektok logických hodín o veľkosti maximálneho počtu vlákien $\mid T \mid$ zapísaný ako C_t. Hodnota $C_t(t)$ predstavuje vlastné lokálne logické hodiny a $C_t(u)$,kde u je iné vlákno, značí logické hodiny

vlákna u . $C_t(u)$ teda predstavuje synchronizačný vzťah - ako vlákno t vidí vlákno u v daný okamih. Ďalej vlákna manipulujú so zdieľanými položkami $x \in Var$, kde Var predstavuje množinu všetkých zdieľaných premenných a ďalej využívajú zámky $m \in Lock$, kde $Lock$ je množina všetkých poskytovaných zámkov. Operácie, ktoré vlákno t môže vykonať zahŕňa:

- $rd(t, x)$, $wr(t, x)$ predstavujúce čitanie (read),zápis (write) z premennej x vláknom t
- $acq(t, m)$, $rel(t, m)$ značiace zabratie (acquire),uvolnenie (release) zámku m vláknom t
- $fork(t, u)$ je vytvorenie nového vlákna u vláknom t
- $join(t, u)$ je zpojenie vlákna u do vlákna t

5.2.3 Komunikačný protokol Djit+

K tomu aby mohli byť implementované pravidlá správania sa vektor-klokov pri synchronizačných operáciách musí byť priradený VC ku každému vláknu C_t, ku každému synchronizačnému objektu L_m a každá zdielaná premenná musí naviac ukladať svoju tzv. históriu prístupov. História predstavuje rámec posledného prístupu k danej položke zaznamenaný pre každé vlákno zvlášť ako W_x a R_x operácie zápis a čitanie z premennej x. Potom Djit+ definuje nasledujúce operácie:

Inicializácia
Každé vlákno t si inkrementuje svoju vlastné časové razítko:

$$\forall i : C_t(t) \leftarrow 1$$

História prístupov ku každej premennej x je vynulovaná:

$$\forall i : R_x(i) \leftarrow 0, W_x(i) \leftarrow 0$$

Synchronizačné zámky m sú taktiež vynulované:

$$\forall i : L_m(i) \leftarrow 0$$

Uvolnenie zámku (*release*)
Pri *release* vlákno t prechádza do nového rámca:

$$C_t(t) \leftarrow C_t(t) + 1$$

Každá položka zámku m je aktualizovaná na maximálnu hodnotu zjednotenia VC vlákna t a VC m:

$$\forall i : L_m(i) \sqcup max(C_t(i), L_m(i))$$

Získanie zámku (*acquire*)
Pri *acquire* je vlákno t aktualizovaná na maximálnu hodnotu zjednotenia VC t a VC zámku m:

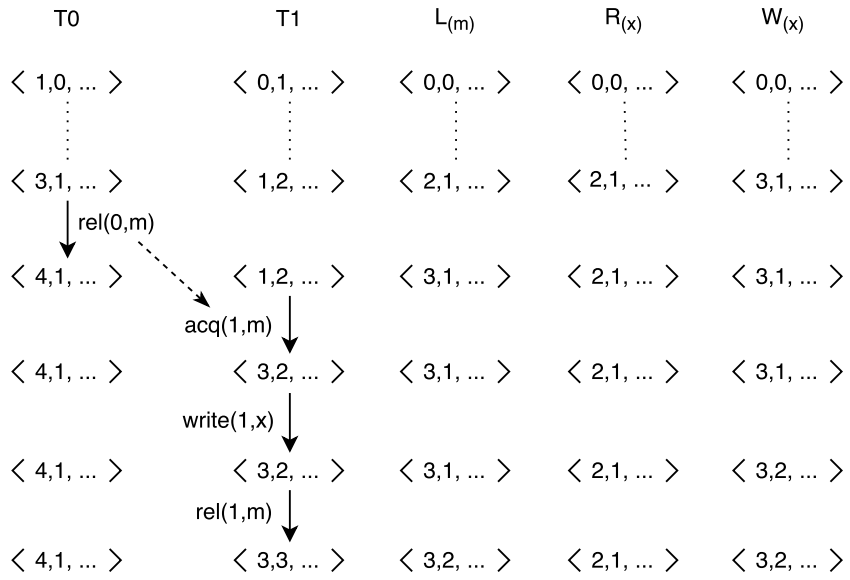
$$\forall i : C_t(i) \sqcup max(C_t(i), L_m(i))$$

R/W operácie
Počas prístupových operácií sa aktualizuje hodnota hodín VC na pozícii t vlákna t, ktoré pristupuje k položke x:

$$R_x(t) \leftarrow C_t(t) \text{ alebo}$$
$$W_x(t) \leftarrow C_t(t)$$

Pred tým, ako nastane priradenie je nutné vykonať overenie relácie happens-before pre odchytenie potenciálnych chýb typu data race

$$\text{pre read: } W_x(t) \sqsubseteq C_t(t)$$
$$\text{pre write: } W_x(t) \sqsubseteq C_t(t), R_x(t) \sqsubseteq C_t(t)$$



Obr. 5.3: Úsek programu monitorovaný Vektor-klokmi využívanými v Djit+.

5.2.4 Detekovanie data-race stavov

Na odhaľovanie chýb typu data race sa využíva happens-before relácia 3.1.1 zakódovaná do podoby vektor-klokov. H-B relácia platí ak napríklad operácie a, b vytvárajúce taký vzťah, že a sa udialo pred b a platí minimálne jedna podmienka:

- Operácie sú vykonané rovnakým vláknom
- Obe operácie využívajú totožný zámok
- Jedna z operácií je $fork(t, u)$ alebo $join(t, u)$ a druhá je vykonaná vláknom u

Ak dve operácie nie sú v relácii happens-before, potom sú považované za konkurentné.

Na detekovanie data race stavov, Djit+ uchováva históriu prístupov R_x a W_x pre každú zdieľanú premennú x. Každé vlákno t, ktoré sa pokúša pristupovať k danej premennej $R_x(t)$ a $W_x(t)$ uchováva hodiny posledného zápisu respektíve čítania. Operácia čítania z premennej

x vláknom u neobsahuje data race chybu ak platí, že sa udiala po zápise každého vlákna ako posledná operácia, $W_x \sqsubseteq C_t$, kde C_t je VC pri vykonávaní operácie čítanie. Pri zápise je potrebné určiť či k operácii vláknom u došlo po všetkých predchádzajúcich prístupoch k premennej, $W_x \sqsubseteq C_t, R_x \sqsubseteq C_t$ kde C_t je VC pri pro vykonávaní operácie zápis.
Pri operácii porovnania dvoch vektorových rámcov $\sqsubseteq$, dochádza k porovnaniu všetkých položiek oboch vektorov, čo vo výsledku predstavuje výpočtovú záťaž až $O(n)$. Týmto problémom sa neskôr zaoberá algoritmus FastTrack rozoberaný v nasledujúcej kapitole 6. Každé vektor zaberá pamäť o veľkosti 1 až maximálny počet vlákien $\mid T \mid$. V preklade ide opäť o náročnú pamäťovú záťaž $O(n)$, ktorú sa FastTrack opäť snaží optimalizovať.

Kapitola 6

FastTrack

Pri analyzovaní veľkých programov s množstvom vlákien alebo časovo limitovaných programov je rýchlosť analýzy značne dôležitá. Vysoká pamäťová náročnosť môže v niektorých systémoch znamenať nemalé problémy. Tieto často krát vysoké náklady je možné zredukovať. Detekovanie data race chýb v paralelných programoch je náročné a preto zaznamenanie čo i len prvého výskytu takejto chyby v reťazci naväzujúcich chýb prináša mnoho prínosu. FastTrack [5] je optimalizovná verzia Djit+ 5.2 algoritmu pracujúceho s myšlienkou, kde sa snaží zredukovať náklady na analýzu na minimum a pritom garantovať presné detekovanie apoň jednoho výskytu každej reálnej chyby.

6.1 Problémy využívania vektor-klokov

Extrapolácia

Počas monitorovania programu môžu vznikať hlásenia, ktoré nepredstavujú chyby v programe ale chyby v abstrakcii využitej techniky na monitorovanie synchronizácie a konkurentných prístupov do zdieľaného priestoru. Ako bolo vysvetlené v sekcii o vektor-klokoch (5.1), ide o bezpečnú extrapoláciu, teda extrapolujú sa reálne behy programu. Toto tvrdenie však nemusí vždy platiť. Jedná sa o bezpečnú extrapoláciu vzhľadom ku happens before relácii (3.1.1). To znamená, že ak H-B relácia bude správna, potom aj extrapolované behy budú správne (reálne). Ak by sa v behu programu objavila synchronizácia, ktorú sme nepremietli do H-B relácie, potom extrapolované behy nemusia byť reálne vykonatelné a analyzátor môžd detekovať falošné (false) alarmy. Jedná sa teda o prehliadnutie synchronizačných objektov pomocou vektor-klokov (H-B relácie). Ak by boli definované vlastné synchronizačné prvky v programe potom by ich analyzátor nedokázal detekovať a vytvoril by tak nepresné extrapolácie behu.

Záťaž na systém

Druhý problém spočíva v prevedení, akým vektor-kloky reprezentujú (ukladajú) dáta, pomocou ktorých je vykonávaná detekcia chýb. Vektor-kloky predstavujú pole počítadiel o veľkosti maximálneho počtu využitých vlákien v programe. Z toho vyplíva že vektor môže dynamicky narásť do veľkých rozmerov, čo je často-krát pravidlom pri viac-vláknových programoch. Takto vzniká značná záťaž na pamäť uchovávajúca tieto vektory, formálne sa jedná o pamäťovú náročnosť $O(n)$, kde n je maximálny počet vzniknutých vlákien za konkrétnej exekúcie.

Súčastne je problémom aj spôsob akým dochádza k overovaniu H-B relácie. V prípade VC operáciou $\sqsubseteq$. Jedná sa tak isto o náročnosť $O(n)$ ako pri pamäťových nárokoch, čo predstavuje opeť veľké výpočetné nároky na analýzu.

6.1.1 Epochy

Využívanie vektor-klokov (5.1) je veľmi výhodné z pohladu presnosti detekcie, no na druhej strane podstatne nákladné. Vo večšine monitorovaných prípadov však nie je nutné uchovávať históriu prístupov pre káždé vlákno, ale stačí využívať len klúčové informácie pre dosiahnutie plného pokrytia chybových stavov a presnosti detekcie. Za predpokladu, že nedošlo ku detekovaniu data race chyby pri prístupe ku zdieľanej premennej x, môžme považovať všetky zápisy do tejto premennej za zoradené podľa H-B relácie (3.1.1). Z tohto pozorovania plynie skutočnosť, že nie je nutné uchovávať históriu prístupov k premennej jednotlivých vlákien ale len poslednú operáciu zápisu a čítania. Spolu s identifikáciou vlákna t je možné bezpečne určiť, či pri pamäťových operáciách nedošlo k data race chybe. S takto redukovanými hodnotami vektor-klokov je možné garantovať len prvý výskyt chyby vzhľadom na spomínanú stratu údajov o predchádzajúcich pamäťových operáciách nad danou premennou ostanými vláknami. Dvojica logikcé hodiny c a identifikačné číslo vlákna t sa nazýva epocha, formálne zapísaná ako $c@t$.
Porovnanie na korektnosť H-B relácie medzi epochou $c@t$ a vektor-klokom V vyjadruje vzťah:

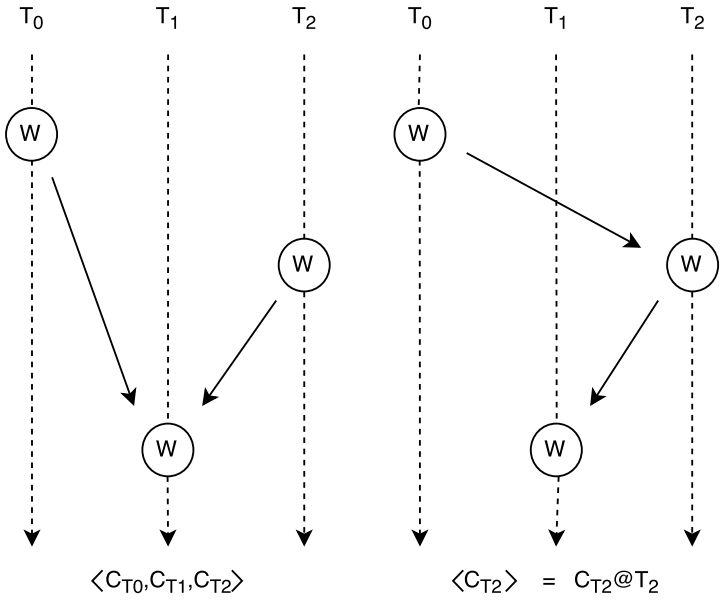
$$c@t \preceq V \;\Leftrightarrow c \leq V(t) \tag{6.1}$$

Ide teda o operáciu porovnania $\preceq$ s konštantnou zložitosťou $O(1)$, narozdiel od operácie $\sqsubseteq$, u porovnávaní dvoch vektor-klokov, ktorá vykazuje zložitosť $O(n)$. Epocha ako dvojica je nenáročná aj na pamäťový priestor a vykazuje tak len konštantnú pamäťovú náročnosť $O(1)$, naproti vektor-klokom pri ktorých už bolo viac krát spomínaná pamäťová záťaž $O(n)$.

Epochy teda predstavujú optimalizované riešenie vektor-klokov. K ich využívaniu dochádza pri zaznamenávaní histórie posledných prístupov k premennej pre zápis aj čitanie. V určitých situáciách však nie je možné pripustiť stratu informácií o histórii. Tu prichádza na radu dynamické prepínanie medzi epochami a vektor-klokmi popísané ďalej v tejto kapitole.

6.2 Detekovanie chýb typu data race

Optimalizácia vychádza z pozorovania, že značná väčšina operácií v konkurentných programoch sú zápis a čítanie zo zdieľanej pamäte, pri ktorých nie je nutné zaznamenávať celú vektorovú reprezentáciu ale len odľahčenú verziu v podobe epoch. Na druhej strane synchronizačné operácie (fork, join, lock, release atď.) na monitorovanie ktorých treba využívať celé vektor-kloky, tvoria len malú časť operácií vyskytujúcich sa v programe. Vo výsledku len pri malom počte operácií je potrebné využívať vektor-kloky. Keďže všetky operácie sú zoradené podľa H-B relácie (pokiaľ nenastala chyba), detekovanie chýb v súbežnosti predstavuje dvojicu operácií z ktorých je aspoň jedna operácia zápisu.



Obr. 6.1: Vektor-klok a epocha. Pri využívaní VC sa porovnáva H-B relácia so všetkými operáciami. Naproti epochy predpokladajú že sú všetky operácie zoradené, a tak porovnávajú len posledný prístup s tým aktuálny.

6.2.1 zápis-zápis

Detekovanie data race chyby dvoch konkurentných zápisov v sebe naplno uplatňuje optimalizácie, ktoré FastTrack prináša. Ak nedošlo doposiaľ k žiadnej detekcii data-race chyby algoritmus uplatňuje pozorovanie, že v tomto prípade sú všetky zápisy do zdielanej premennej úplne zoradené podľa H-B relácie. Preto jediné kritické informácie sú obsiahnuté v epochách.

6.2.2 zápis-čítanie

Pri detekcii kombinácie zápis a čitanie sa využíva rovnaký princíp ako pri predchádzajúcom prípade. Algoritmus sleduje či sa posledné čitanie z premennej x udialo po predchádzajúcom zápise pomocou $\preceq$ operácie v konštantnom čase $O(1)$.

6.2.3 čítanie-zápis

Kombinácia čítanie a zápis už predstavuje zložitejší prípad monitorovania ako v predchádzajúcich dvoch prípadoch, kde sme mali garantované zoradenie operácií na základe relácie happens-before v prípade že nedošlo k žiadnemu výskytu chyby. Operácie čítania nemusia byť podla tohto zvťahu zoradené ani v programoch ktoré sú tzv. *data race free*, teda neobsahujú žiadne konkurentné chyby. Posledný zápis môže potenciale kolidovať nielen s posledným čítaním ale aj s ktorým koľvek iným čítaním v ostatných vláknach. Z tohto dôvodu FastTrack prechádza na dočasné monitorovanie pomocou plnohodnotných vektor-klokov, avšak v množstve situácií je možné predísť využívaniu vektor-klokov. FastTrack rieši nasledujúce situácie:

- **Thread-local dáta** - dáta využívané len jedným vláknom a teda všetky prístupy sú zoradené a vykonávané sekvenčne preto sa naďalej využívajú epochy pre uloženie histórie.
- **Lock-protected dáta** - pamäťové položky chránené vlasnými výlučným zámkom pre každú položku osobytný. V týchto prípadoch sú operácie čítania zoradené a je možné využívanie epoch na monitorovanie histórie prístupov.
- **Read-shared dáta** - dáta, pri ktorých ide o situáciu, kedy je zdielaná položka najskôr inicializovaná jedným vláknom a následne nazdieľaná medzi ďalšie vlákna. Tu za použitia epoch môže dôjsť k prehliadnutiu data race chyby a preto je potrebný prechod na celé VC.

6.3 Adaptívne správanie FastTracku

FastTrack využíva adaptívnu reprezentáciu histórie zápisov čo mu umožňuje jednak značnú redukciu potrebných výpočtových a pamäťových prostriedkov v podobe epoch a zároveň mu poskytuje v prípade potreby plnú silu obsiahnutú vo vektor-klokoch. Pre *thread-local* a *lock-protected* dáta, ktoré zaručujú poradie operácií, V menejčastých prípadoch, kedy čítania nie sú usporiadané, si algoritmus uchováva celý vektor-klok. Pri situáciách kedy sa k dátam pristupuje len z jednoho vlákna (t*hread-local*) alebo kedy sú pristupované dáta zabezpečené pred súbežným prístupom pomocou synchronizačných objektov (*lock-protected*) FastTrack potrebuje zaznamenať len epochu. Pri veľmi raritných prípadoch, kedy sú dáta zdieľné medzi viacero vlákien (*read-shared*) algoritmus musí uchovávať calý vektor-klok. Avšak aj tu detekcia W-R a R-W data race chýb prebieha v konštantnom čase, pretože sa VC porovnávajú s epochami operácií zápisu v konštantnom čase pomocou $\preceq$ operácie.

6.4 Algoritmus

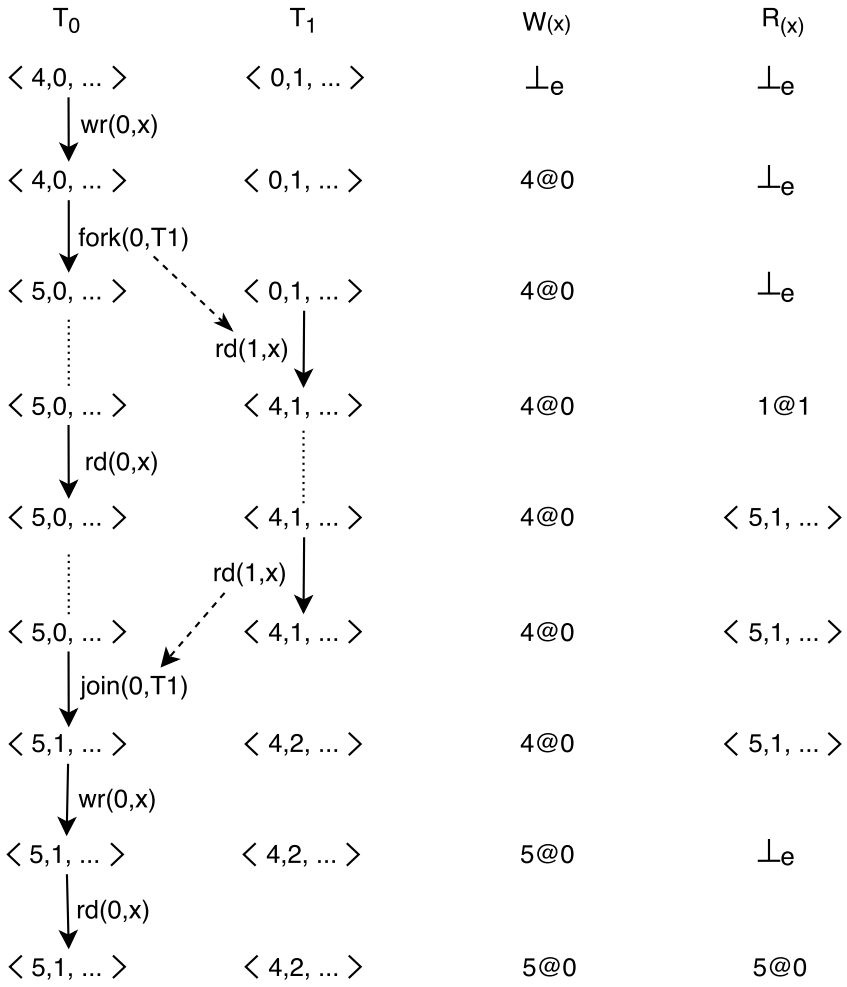
Algoritmus počas behu udržuje stav σ, v prípade ak analyzátor vykoná operáciu a, dôjde k aktualizovaniu stavu analýzy pomocou relácie $\sigma \Rightarrow^a \sigma'$. Inštrumentačný stav σ pozostáva zo štyroch komponent (C, L, R, W) :

- C_t identifikuje súčastný stav vektor-kloku vlákna t.
- L_m identifikuje súčastný stav vektor-kloku posledného uvoľnenia zámku m.
- R_x identifikuje vektor identifikuje súčastný stav buď epochy poslednej operácie čitania z premennej x, alebo ako plnohodnotný VC v prípade zdielania dát (read-share)
- W_x identifikuje epochu posledného zápisu do premennej x.

Počiatočným stavom analýzy je:

$$\sigma_0 = (\lambda t.inc_t(\bot_v),\ \lambda m\ .\ \bot_v,\ \lambda x\ .\ \bot_e,\ \lambda x\ .\ \bot_e)$$

V ďalších sekciách sú popísané klúčové detaily ako FastTrack spracúva a monitoruje čítanie (read), zápis (write) a synchronizačné operácie, v ktorych sú využívané konvencie zapísané (v 5.2.2).



Obr. 6.2: Úsek komunikácie dvoch vlákien, vlákno T_1 nazdieľalo vláknu T_2 položku x, pri zaznamenávaní histórie zápisov muselo dôjsť ku prechodu na VC. Po spojení vlákien algoritmus prechádza späť k využívaniu epoch

6.4.1 Operácia čítanie (read)

Analyzovanie $read(t, x)$ operácie môžu byť realizované v štyroch rôznych situáciách (*same epoch*, *read shared*, *exclusive*, *read share*) a sú zoradené podľa pravdepodobnosti s akou sa v programoch vyskytujú [5].

Pravidlo *Same epoch* predstavuje optimalizáciu situácie, kedy sa operácie čítania $rd(t, x)$ odohrávajú v tom istom vlákne, teda nie je potrebná kontrola happens-before, keďže operácie v rovnakom vlákne su vykonávané sekvenčne. Pravidlo predstavuje prevažnú časť všetkých situácií operácie čítania. $E(t)$ obsahuje aktuálnu epochu $c@t$, kde platí $c = C_t(t)$.

[Same epoch]

$$\frac{R_x \;=\; E(t)}{(C, L, R, W) \;\Rightarrow^{rd(t,x)}\; (C, L, R, W)}$$

V prípade *Read shared* (6.2.3) algoritmus už pracuje s plnohodnotnými VC v histórii čítaní z premennej. Overovanie H-B relácie so zápisom je však stále uskutočnené pomocou operácie $\preceq$. Ak je história R_x VC potom sa pri čítaní len aktualizuje potrebná položka vo VC R_x na pozícii t vlákna, ktoré k nej pristupuje.

[Read shared]

$$\frac{\begin{array}{c} R_x \;\in\; VC \\ W_x \preceq C_t \\ R' \;=\; R[x \;:=\; R_x[t \;:=\; C_t(t)]] \end{array}}{(C, L, R, W) \;\Rightarrow^{rd(t,x)}\; (C, L, R', W)}$$

Pravidlo *exclusive* predstavuje situáciu kedy vlákno pristupuje ku položke, ku ktorej predchádzajúce operácie pristupovali za využitia medzivláknovej synchronizácie. V tomto prípade sú všetky operácie optimalizované epochami, história R_x je aktualizované priradením epochy pristupujúceho vlákna t.

[Exclusive]

$$\frac{\begin{array}{c} R_x \;\in\; Epoch \\ R_x \preceq C_t \\ W_x \preceq C_t \\ R' \;=\; R[x \;:=\; E(t)] \end{array}}{(C, L, R, W) \;\Rightarrow^{rd(t,x)}\; (C, L, R', W)}$$

Pravidlo *read share* určuje situáciu, kedy dochádza ku nazdieľaniu premennej a tým musí dôjsť k prechodu zo systému využívajúceho epochy v histórii čítaní, na VC. Táto situácia nastáva len vo veľmi ojedinelých prípadoch, no je spomedzi všetkých najdrahšie, keďže dochádza ku vytváraniu nového VC.

[Read share]

$$\frac{\begin{array}{c} R_x \;=\; c@u \\ W_x \preceq C_t \\ V \;\; = \bot_v \; [t := C_t(t), u \;:= c] \\ R' \;=\; R[x \;:=\; V] \end{array}}{(C, L, R, W) \;\Rightarrow^{rd(t,x)}\; (C, L, R', W)}$$

6.4.2 operácia zápis (write)

Ďalšie tri situácie (*same epoch*, *write shared*, *exclusive*) predstavujú možné scenáre monitorovanie zápisov $wr(t, x)$.

Pravidlo *Same epoch* opäť optimalizuje situáciu, kedy už nastal zápis $wr(t, x)$ do rovnakej premennej x v rovnakom vlákne t.

[Same epoch]

$$\frac{W_x \;=\; E(t)}{(C, L, R, W) \;\Rightarrow^{wr(t,x)}\; (C, L, R, W)}$$

Pravidlo *Exclusive* kotroluje, že všetky predchádzajúce operáciu pristupujúce ku danej premennej x sa odohrali skôr ako aktuálny zápis $wr(t, x)$. V tomto prípade je história čítaní R_x v stave epochy.

[Exclusive]

$$\frac{\begin{array}{c} R_x \;\in\; Epoch \\ R_x \preceq C_t \\ W_x \preceq C_t \\ W' \;=\; W[x \;:=\; E(t)] \end{array}}{(C, L, R, W) \;\Rightarrow^{wr(t,x)}\; (C, L, R, W')}$$

Ak sa história R_x nachádza v stave VC nastáva situácia *write shared*, čo je zápis do takto definovanej pamäte a je potrebné využiť proovnanie medzi dvoma VC $\sqsubseteq$. K tejto situácii však dochádza len veľmi zriedka.

[Write Shared]

$$\frac{\begin{array}{c} R_x \;\in\; VC \\ R_x \sqsubseteq C_t \\ W_x \preceq C_t \\ W' \;=\; W[x := E(t)] \\ R' \;=\; R[x :=\bot_e] \end{array}}{(C, L, R, W) \;\Rightarrow^{wr(t,x)}\; (C, L, R', W')}$$

6.4.3 Synchronizačné operácie

Ostatné operácie $acquire(), release(), fork(), join()$ sú v porovnaní s $read()$ a $write()$ veľmi ojedinelé. Využitie úplnych vektor-klokov je v týchto prípadoch potrebná no v kombinácii s raritou akou sa vyskytujú pri monitorovaní, sa nejedná o veľkú záťaž na rýchlosť analýzy.

[Acquire]

$$\frac{C' \;=\; C[t := (C_t \sqcup L_m)]}{(C, L, R, W) \;\Rightarrow^{acq(t,m)}\; (C', L, R, W)}$$

[Release]

$$\frac{\begin{array}{c} L' \;=\; L[m := C_t] \\ C' \;=\; C[t := inc_t(C_t)] \end{array}}{(C, L, R, W) \;\Rightarrow^{rel(t,m)}\; (C', L', R, W)}$$

[Fork]

$$\frac{C' \;=\; C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L, R, W) \;\Rightarrow^{fork(t,u)}\; (C', L, R, W)}$$

[Join]

$$\frac{C' \;=\; C[t := C_t \sqcup C_u, u \;:= inc_u(C_u)]}{(C, L, R, W) \;\Rightarrow^{join(t,u)}\; (C', L, R, W)}$$

Kapitola 7

Implementácia

Táto kapitola sa zaoberá časťami implementácie, ktoré sú z mojho pohľadu zaujímavé a venuje sa odlišnostiam v implementácii od teórie.

7.1 Detaily implementácie

Implementácia FastTrack-u priraďuje ku každému vláknu jeden objekt `ThreadState`.Ten predstavuje lokálne hodnoty jednoho vlákna - lokálny vektor-klok (5.1) `currentClock`, identifikátor vlákna `ts_tid` a referenciu na zoznam ostatných `ThreadState` objektov `ThreadStateList` reprezentujúcich lokálne dáta ostatných vlákien. Ďalej priraďuje ku každej synchronizačnej entite reprezentujúcej zámok `LockState` objekt. `LockState` obsahuje vektor-klok `lockClock` zaznamenávajúci históriu prístupov ku zámku a hodnotu `lockValiability` reprezentujúcu stav zámku (`valid` alebo `invalidL`). Ku každej pamäťovej položke priraďuje objekt `VarState` obsahujúci dva vektor-kloky `R` a `W` inicializované ako epochy reprzentujúce históriu prístupov ku premennej a hodnotu `variableStatus` značiacu stav v akom sa premenná nachádza(`invalid`, `nonShared` alebo `readShared`).

7.2 Kódovanie H-B relácie

7.2.1 Implementácia epoch

Pri implementácii epoch (6.1.1) bolo potrebné rozšíriť vektor-kloky implementované ako trieda `VectorClock` tak, aby ich bolo možné použiť aj v podobe epoch. Cieľom bolo zjednotiť obe formy do jednotnej triedy tak, aby si epochy zachovali svoju nízku pamäťovú náročnosť a udržali si rýchlosť pri porovnávaní vektor-klokov pomocou operácie $\preceq$. Zároveň však zjednotenie oboch prevedení do jedného predstavovalo rýchlu a jednoduchú možnosť ako zaručiť adaptívne správanie vektor-klokov vo FastTrack-u (6), kde je potrebné transformovať epochy na vektor-kloky a opačne. Vektor-klok je implementovaný ako kontajner `vc` reprezentujúci vektor nezáporných celočíselných hodnôt. Ku každému vektoru je definovaná aj hodnota (flag) `epochID` nesúca dva druhy informácií. Záporná hodnota `epochID` indikuje, že objekt je aktuálne v stave plnohodnotného vektor-kloku. Ak je `epochID` nezáporné, jedná sa o epochu definovanú ako $c@t$. Potom `epochID` značí identifikačné číslo vlákna t a jeho hodnota je uložená ako prvá požka vektoru (položka na indexe 0), aby nedochádzalo ku plytvaniu pamäťovým priestorom. Preklopenie medzi epochou a VC je realizované pomocou metódy `switchToVC`, ktorá realizuje tento prevod inicializovaním všetkých poterbných položiek vektoru na hodnotu 0 pomocou metódy `initL`. Prednostne je však potrebné si uchovať

aktuálnu hodnotu epochy, a následne ju vložiť na požadované miesto `epochID` v novovziknutom vektore. Nakoniec dochádza ku prepnutiu signalizačnej premennej `epochID` na hodnotu -1. Podľa tejto hodnoty sa riadi aj v každej funkcii mechanizmus výberu teda aká funkcionalita nastane podla toho či ide o VC (`epochID` $== -1$)) alebo epochu (`epochID` $>= 0$).

7.2.2 Porovnávanie a spájanie VC

FastTrack predpokladá pevne daný počet vlákien programu, ktorý sa počas behu nezmení. V praxi však vlákna často zanikajú a vznikajú dynamicky behom exekúcie programu. Tým pádom sa veľkosť vektor-klokov mení v priebehu analýzy a implementácia sa s tým musí vysporiadať. Implementované sú ako dynamické vektor-kloky, ktoré dokážu meniť svoju veľkosť podľa potreby. Dynamicky sa meniaci počet vlákien nemá vpliv na využívanie epoch, no pri vektor-klokoch je to problém. Kôli premenlivosti počtu vlákien dochádza aj k porovnávaniu rôzne dlhých vektor-klokov operáciou $\sqsubseteq$. Toto porovnávanie je implementované vo FastTrack-u ako metóda `hb()`. Objekt volajúci túto metódu predstavuje v relácii happens-before (3.1.1) operáciu, ktorá sa udiala ako posledná a argument predstavuje históriu, teda operáciou ktorá sa udiala skôr (ak nedošlo ku data race chybe). Druhým argumentom je identifikačné číslo vlákna *tid* ktoré túto funkciu volá. Najskôr musí dôsť k overeniu či vektor-klok poslednej operácie obsahuje položku s vláknom *tid*. Ak je veľkosť VC <= ako hodnota *tid*, potom ešte nenastala synchronizácia, inak by informácia o vlákne s identifikátorom *tid* bola vo VC. Následne prebieha overovanie či je dodržana relácia H-B porovnávaním oboch vektor-klokov.

Pri spájaní dvoch vektor-klokov operáciou join $\sqcup$ implementovaná v rovnomennej metóde `join()` , môže dôjsť tak isto ku situácii, kedy sa spájajú dva VC s rozdielnými dĺžkami. Najskôr je nutné určiť, ktorý z dvojice vektor-klokov je dlhší a ktorý kratší. Následne prebehne spojenie $\sqcup$ preiterovaním cez spoločnú časť oboch VC a v prípade ak je volajúci vektor, do ktorého sa ukladá vysledok menším, prebehne priradenie zvyšných hodín druhého VC k prvému. Takýto postup je možný preto, lebo chýbajúce zložky VC reprezentujú hodnotu 0.

7.3 Úložisko dát

7.3.1 Lokálny priestor vlákien

FastTrack sítce uvádza, že každé vlákno obsahuje svoj *lokálny* VC, ale k týmto dátam potrebujú pristupovať aj okolité vlákna. Dôvodom je, že niektoré operácie algoritmu FastTrack (napr. join) operujú s lokálnymi VC nie len aktuálneho vlákna, ale aj okolitých vlákien. Implementácia rieši tento problém nasledovne. Každé vlákno má implementované svoje vlastné úložisko dát v objekte `ThreadLocal`, ktorá uchováva ukazateľ na objekt `ThreadState` v ktorej sú uložené dáta o konkrétnom vlákne. Okrem lokálnych dát však `ThreadState` disponuje referenciou na `ThreadStateList`, čo je vektor všetkých lokálnych úložísk pre každé vlákno. Z tohto plynie, že k lokáným údajom je možné pristupovať aj z ostatných vlákien. Týmto spôsobom porušujeme princíp lokálneho úložiska. Pri operácii spojenia vlákien *join* je nutné mať informácie o druhom, pripojovanom vlákne, keďže tu dochádza ku synchronizácii dvoch vlákien bez nejakého sprostredkovateľa ako napríklad zámku.

7.3.2 Globálne úložisko

Stav premennej `variableStatus` reprezentuje stav objektu `VarState` a stav zámku `lockValiability` reprezentujú stav objektu `LockState`. Pri implementácii globalnych úložísk pre zdieľanie histórie zámkov a premenných vznikol problém identifikovať novo pristupované objekty či už premenné alebo zámky.Pri implementovaní bolo treba najskôr vytvoriť `LockValiability` určuje či je instancia zámku používaná. Pri inicializácii je zámok nastavený do stavu `invalidL`. Po vložení zámku do globálnej mapy `globalLockState` sa jeho stav zmení na `valid` . U zdieľaných položiek je stav premennej rozšírený o hodnotu `readShared` , čo predstavuje určitý stav premennej kedy je zdieľaná naprieč vláknami a určená k čítaniu. História prístupov k premenným je teda globálne prístupná zo zdieľanej mapy `globalVarState`.

Kapitola 8

Dosiahnuté výsledky

Analyzátor FastTrack implementovaný v prostredí ANaConDA (4) je možné využívať na testovanie viac-vláknových programov. Jeho algoritmus prináša rýchlosť a presnosť v odhaľovaní data race chýb, čo nasledujúce experimenty porovnávajúce dva algoritmy FastTrack a AtomRace [10], potvrdzujú. Problémom pri získavaní relevantných údajov o výkonnosti analyzátora bolo nájsť také testovacie objekty, ktoré obsahujú data race chyby a ich existenciu je možné potvrdiť aj iným analyzátorom. Ďalším problémom bola komplexnosť programov nad ktorými bola analýza vykonávaná. Príliš jednoduché programy vykazovali odhalenie chyby pri každom behu analýzy a ich analýza prebehla vo veľmi krátkom čase, čo pre porovnanie s iným analyzátorom nepredstavovalo veľkú výpovednú hodnotu. Nakoniec boli zaznamenané len experimenty nad programami otestovanými v článku [4] pomocou dynamického analyzátora AtomRace. Porovnanie oboch analyzátorov je znázornené na obrazku 8.1.

8.1 Experiment

FastTrack v testovacích prípadoch odhaluje chyby takmer vždy, čo sa dalo očakávať. AtomRace nevykonáva žiadnu extrapolácie a bez nej musí chybu priamo vidieť, takže šanca, že ju uvidí bez pomoci inej techniky, ktorá by navýšila šancu pozorať chybu v programe (napr. vkladanie šumu) je malá aj pre takto triviálne programy, na ktorých boli oba algoritmy preverované. Naopak FastTrack pre takéto malé programy extrapoluje beh na väčšinu možných exekúcií a s veľkou pravdepodobnosťou sa medzi nimi bude vyskytovať aj niektorý s chybou.

8.1.1 Vyhodnotenie experimentu

Hodnoty namerané pomocou analyzátora FastTrack vychádzajú z nižšieho počtu analyzovaných behov programu ako pri AtomRace, no i tak znázorňujú priepastný rozdiel medzi oboma algoritmami. Pri testovaní pomocou FastTrack-u bol každý testovaný program spustený 50 krát. Oproti hodnotám získaných pri analýze pomocou AtomRace-u, ktorý analyzoval každý program 500 krát [4], je to len malá časť a hodnoty sú čiastočne skreslené, no aj napriek tomu znázorňuje vysoký nárast pri detekcii dat arace chýb pomocou vektor-klokov impelemntovaných vo FastTrack-u.

analyzátor/program	t01	t02	t03	t04	t05	t06	t07
FastTrack	100	100	100	100	100	100	100
AtomRace	2.4	11.8	0.2	1.2	0.0	1.0	1.6
analyzátor/program	t08	t09	t10	t11	t12	t13	
FastTrack	100	100	95	100	95	100	
AtomRace	2.2	0.4	0.0	0.0	0.0	32.2	

Obr. 8.1: Tabuľka porovnávajúca detektory FastTrack a AtomRace implementované v prostredí ANaConDA. Znázorňuje percento prípadov exekúcií kedy došlo k odhaleniu chyby v programe.

8.1.2 Ďalšie experimenty

Rýchlosť analýzy pri vyhľadávaní data race chýb v programoch, ktoré tieto chyby obsahujú nebola zaznamenávaná z dôvodov popísaných vyššie. Rýchlosť akou sa analýza vykonáva, by bolo najlepšie porovnať s ďalšími analyzátormi ako Djit+, využivajucimi vektor-kloky, ktorý však nie je implementovaný v prostredí ANaConDA.

Kapitola 9

Záver

Chyby typu data race sú vo viac-vláknovom programovaní veľkým problémom a ich odhalovanie môže byť veľmi problematické najmä z pohladu nedeterministického plánovania vlákien. Jednou z techník, ktorá sa snaží takéto chyby detekovať je dynamická analýza (3) založená na princípe monitorovania programu za jeho behu. Tým, že monitoruje len jeden beh umožňuje chybám, ktoré sa prejavia len v konkrétnych, často ojedinelých preloženia, zostať nepovšimnuté, a tak vystaviť užívatela do mylnej predstavy bezchybného programu. Z tohto dôvodu sa dynamická analýza rozširuje o ďalšie techniky extrapolácie, ktoré umožňujú analyzátorom overiť aj ďalšie exekúcie podobné primárnemu behu a nájsť v nich chyby. Extrapolácia môže byť prevedená pomocou happens-before relácie (3.1.1). Algoritmus pre odhaľovanie chýb typu data race (2.2.1) pomocou dynamickej analýzy FastTrack (6), implementuje túto reláciu v podobe vektor-klokov (5.1). V porovnaní s ďalšími analyzátormi sa snaží optimalizovať náklady na analýzu. FastTrack prichádza s myšlienkou optimalizovaných vektor-klokov v podobe epoch(6.1.1), ktoré tvoria dvojicu vlákno a hodnota hodín, na miestach kde je to bezpečné. FastTrack je implementovaný pre programy v jazyku Java. Cieľom tejto práce ho bolo implementovať pre C/C++ programy. Algoritmus je implementovaný v prostredí ANaConDA (4), ktoré tvorí monitorovaciu vrstvu medzi analyzátorom a analyzovaným programom, čím zabezpečuje vyťahovanie zaujímavých informácií z bežiaceho programu a tým poskytuje analyzátoru potrebné informácie ku monitorovaniu aplikácie.

9.1 Možnosti rozšírenia

Odovzdaná verzia algoritmu umožňuje detekovanie viac data race chýb v programe, no jeho rozšírenie je možné vo viacerých smeroch. V prvom rade analyzátor zatiaľ neposkytuje podrobný prehľad o vzniknutých chýbach. Oznámenia užívateľovi by bolo vhodné rozšíriť o konkrétne miesto v pamäti a konkrétne miesto v kóde na akom nastala chyba, o názov premennej ku ktorej sa pristupovalo, aké vlákno spôsobilo chybu a o akú operáciu sa jednalo. Ďalej by bolo možné optimalizovať epochy na maximum, implementovaním epoch v podobe jednej číselnej hodnoty, narozdiel od dvojice hodnota a identifikačné číslo vlákna. Jednalo by sa o číslo rozdelené na dve bitové časti reprezentujúce indentifikačné číslo vlákna a v druhej časti hodnotu hodín. Ďalšou z možností ako rozšíriť algoritmus by bolo rozšírenie podpory pre viac synchronizačných objektov ako napríklad `bariér` ale iných programátorom definovaných synchronizačných funkcií. Aktuálna verzia podporuje len základnú synchronizáciu v podobe zámkovej logiky `lockAcquire` a `lockRelease` a synchronizáciu pri vytváraní a zanikaní vlákien `fork` a `join`.

Ako ďalší postup by bolo možné otestovať rýchlosť s akou FastTrack analyzuje programy oproti iným analyzátorom ako napríklad už spomínaný AtomRace alebo Djit+. Do budúcna by bolo dobré otestovať FastTrack aj na komplexnejších programoch, kde by sa mala ukázať jeho prednosť v rýchlosti analýzy.

Literatúra

[1] Ferrara, P.: Static Analysis of the Determinism of Multithreaded Programs. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, Nov 2008, ISSN 1551-0255, s. 41–50, doi:10.1109/SEFM.2008.14.

[2] Fiedor, J.; Křena, B.; Letko, Z.; aj.: *A Uniform Classification of Common Concurrency Errors*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-642-27549-4, s. 519–526, doi:10.1007/978-3-642-27549-4_67.
URL http://dx.doi.org/10.1007/978-3-642-27549-4_67

[3] Fiedor, J.; Vojnar, T.: Noise-Based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In *PADTAD'12*, ACM, 2012, s. 36–46.

[4] Fiedor, J.; Vojnar, T.: ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of RV'13*, volume 7687 of LNCS, Springer-Verlag, 2013, s. 35–41.

[5] Flanagan, C.; Freund, S. N.: FastTrack: efficient and precise dynamic race detection. In *PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-392-1, s. 121–133, doi:http://doi.acm.org/10.1145/1542476.1542490.

[6] Flanagan, C.; Freund, S. N.: The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, New York, NY, USA: ACM, 2010, ISBN 978-1-4503-0082-7, s. 1–8, doi:10.1145/1806672.1806674.
URL http://doi.acm.org/10.1145/1806672.1806674

[7] Guerraoui, R.; Kapalka, M.: *Principles of Transactional Memory*. Morgan and Claypool Publishers, 2010.

[8] Kasikci, B.; Zamfir, C.; Candea, G.: Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-0759-8, s. 185–198, doi:10.1145/2150976.2150997.
URL http://doi.acm.org/10.1145/2150976.2150997

[9] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, ročník 21, č. 7, Červenec 1978: s. 558–565, ISSN 0001-0782, doi:10.1145/359545.359563.
URL http://doi.acm.org/10.1145/359545.359563

[10] Letko, Z.; Vojnar, T.; Křena, B.: AtomRace: data race and atomicity violation detector and healer. In *PADTAD'08*, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-052-4, s. 1–10.

[11] Luk, C.-K.; Cohn, R.; Muth, R.; aj.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, New York, NY, USA: ACM, 2005, ISBN 1-59593-056-6, s. 190–200, doi:10.1145/1065010.1065034.
URL http://doi.acm.org/10.1145/1065010.1065034

[12] Nethercote, N.; Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, ročník 42, č. 6, Červen 2007: s. 89–100, ISSN 0362-1340, doi:10.1145/1273442.1250746.
URL http://doi.acm.org/10.1145/1273442.1250746

[13] Parízek, P.; Lhoták, O.: Model Checking of Concurrent Programs with Static Analysis of Field Accesses. *Sci. Comput. Program.*, ročník 98, č. P4, Únor 2015: s. 735–763, ISSN 0167-6423, doi:10.1016/j.scico.2014.10.008.
URL http://dx.doi.org/10.1016/j.scico.2014.10.008

[14] Pozniansky, E.; Schuster, A.: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of PPoPP'03*, New York, NY, USA: ACM, 2003, ISBN 1-58113-588-2, s. 179–190, doi:http://doi.acm.org/10.1145/781498.781529.

[15] Savage, S.; Burrows, M.; Nelson, G.; aj.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, ročník 15, č. 4, Listopad 1997: s. 391–411, ISSN 0734-2071, doi:10.1145/265924.265927.
URL http://doi.acm.org/10.1145/265924.265927

Príloha A

Obsah CD

Priložené CD obsahuje nasledujúce súbory:

- BP_ elektronickaForma.pdf - elektronická forma bakalárskej práce
- BP_ listForma.pdf - forma pre tlač bakalárskej práce
- Manuál - manuál ku spusteniu analyzátora
- Anaconda - súbor obsahujúci framefork ANaConDA