VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

POKRYTÍM ŘÍZENÉ TESTOVÁNÍ VÍCEVLÁKNOVÝCH PROGRAMŮ

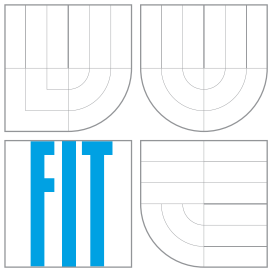
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ZUZANA LIETAVCOVÁ

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

POKRYTÍM ŘÍZENÉ TESTOVÁNÍ VÍCEVLÁKNOVÝCH PROGRAMŮ

COVERAGE-DRIVEN TESTING FOR MULTITHREADED PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ZUZANA LIETAVCOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZDENĚK LETKO, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá problematikou hledání chyb ve vícevláknových programech pomocí techniky pokrytím řízeného testování, jak je chápáno ve studovaném nástroji Maple. Testování se se skládá ze dvou fází. V první fázi se buduje množina pokrytelných chování testovaného programu. Následně se algoritmus snaží dosáhnout těchto chování za pomoci deterministického vykonání testu. Hlavní přínos práce spočívá v uceleném popisu nástroje Maple, včetně technických detailů. Na základě studia jsou identifikovány slabá místa. Některé z nich, konkrétně využívání náhodného rozhodování a prioritizace vynucovaných chování, jsou blíže studovány. Výsledkem je několik úprav nástroje Maple, ze kterých některé vedou k většímu počtu úspěšných dosažení chování a v určitých případech k vyššímu počtu vyvolání chyb, což je experimentálně demonstrováno na sadě vícevláknových programů.

Abstract

This work deals with a problem of searching errors in multithreaded programs using a coverage-driven testing technique as perceived in program Maple. The testing consists of two phases. In the first phase of testing a set of coverable behaviours of the tested program is being built. Consequently, the algorithm tries to achieve these behaviours with a help of deterministic test execution. The main acquisition of the work lays in a compact description of Maple including all the technical details. Based on the study of the tool there were weak places identified. Some of them are studied in detail, especially those which use random decision making and prioritizing of the forced behaviours. The result are several modifications of Maple, from which some lead to a higher number of exposed behaviours and higher error exposition in some cases. This is demonstrated on a test suite of parallel programs.

Klíčová slova

paralelní programování, vícevláknové programy, testování, dynamická analýza, instrumentace, virtualizace, Maple, Pin

Keywords

parallel programming, multithreaded programs, testing, dynamic analysis, instrumentation, virtualization, Maple, Pin

Citace

Zuzana Lietavcová: Pokrytím řízené testování
vícevláknových programů, bakalářská práce, Brno, FIT VUT v Brně, 2015

Pokrytím řízené testování vícevláknových programů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Letka, Ph.D.

.....
Zuzana Lietavcová
20. května 2015

Poděkování

Chcela by som poďakovať Ing. Zdeňkovi Letkovi, Ph.D. za odborné vedenie, prínosné rady a postrehy pri tvorbe práce a všetok čas ktorý mi venoval. Za odbornú pomoc pri riešení práce chcem poďakovať aj Ing. Janu Fiedorovi.

Obsah

Kapitola 1

Úvod

Verifikácia softvéru získava v súčasnosti stále väčší význam. Súvisí to s tým, že počítače prenikajú do takmer všetkých oblastí nášho života a preto sa od nich vyžaduje stále väčšia spoľahlivosť. Najrozšírenejou metódou verifikácie programov je testovanie, ktoré je v kombinácii s dynamickou analýzou predmetom tejto práce. Pojmy a techniky nevyhnutné pre pochopenie tejto práce sú predstavené v úvodnej kapitole 2, ktorá uvádza čitateľa do oblasti paralelného programovania a v kapitole 3, ktorá ozrejmuje použité verifikačné techniky.

Prvým z cieľov práce je poskytnúť ucelený popis programu pre testovanie viacvláknových programov, programu Maple. V kapitole 4 sú priblížené základy programu Pin, nástroja pre dynamickú inštrumentáciu programov, ktorý poskytol základ pre vytvorenie samotného programu Maple.

Testovanie v réžii programu Maple prebieha v dvoch fázach. V prvej fáze sa vytvára množina pokryteľných správaní testovaného programu, v druhej fáze sa algoritmus snaží dosiahnuť tieto správania s pomocou deterministického vykonania testu. Programu Maple je venovaná kapitola 5, ktorá vychádza z dostupných teoretických základov, na ktorých je program Maple postavený, avšak dopĺňa ich o podrobnejší popis skutočnej implementácie.

Ďalším cieľom práce je identifikovať slabé miesta programu Maple, diskutovať možnosti ich eliminácie 6 a realizovať úpravu programu pre odstránenie niektorých z nich.

Bolo realizovaných niekoľko úprav zameraných na proces náhodného rozhodovania a prioritizáciu niektorých správaní. Vplyv týchto úprav bude ukázaný a diskutovaný v kapitole 7 z viacerých hľadísk. Niektorými úpravami sa podarilo dosiahnuť vyššie pokrytie správaní programu, v niektorých prípadoch zvýšiť počet odhalení chyby v testovanom programe a odhaliť tieto chyby skôr.

Pre účely otestovania programu Maple a jeho úprav bola v rámci práce vytvorená sada testových programov tzv. benchmark. Každý z programov obsahuje chybu týkajúcu sa viacvláknových programov, ktorej najefektívnejšie odhalenie je cieľom testovania.

Záver práce hodnotí dosiahnuté výsledky a diskutuje možnosti ďalšieho rozvíjania práce.

Kapitola 2

Paralelné programovanie

Táto kapitola vysvetľuje koncept paralelného programovania a dôvody rozšírenia tohto konceptu a uvádza tak čitateľa do problematiky, ktorou sa táto práca zaoberá. Podkapitola 2.2 definuje základné pojmy týkajúce sa paraleného programovania – proces a vlákno. Ďalej je v kapitole načrtnutý spôsob behu paralelných programov, konkrétne prideľovanie výpočtového času procesoru a problémy z toho vyplývajúce.

2.1 Rozšírenie paralelného programovania

S rozšírením viacjadrových procesorov narástla popularita paralelných programov, či už vo forme viacprocesových alebo viacvláknových aplikácií. Cieľom bolo, aby program mohol bežať na viacerých jadrách súčasne a tým efektívnejšie využil výpočtové možnosti hardvéru. Tento nový prístup však viedol k novým typom chýb v programoch, ktoré sa u sekvenčných programov nevyskytujú. Vyhýbanie sa týmto chybám vyžaduje značnú programátorskú zručnosť a rovnako aj odhaľovanie týchto chýb je netriviálna úloha. Dôvod komplikácií leží v nedeterministickej povahe behu viacvláknových programov. Klasifikáciou a odhaľovaním týchto chýb sa zaoberá kapitola 3.3.

2.2 Vlákna vs. procesy

Proces [2] je samostatná jednotka, ktorá má vlastný pamäťový priestor a s inými procesmi komunikuje zasielaním a prijímaním správ. Proces vytvorí operačný systém pre každý vykonávaný program. Záznam o procese obsahuje informácie o vykonávaní – stav procesu, programové počítadlo, uložené hodnoty registrov, zásobník aktivačných záznamov a pod. Veľký objem informácií potrebných pre riadenie procesov spôsobuje nárast režijných výpočtov pri prepínaní medzi procesmi.

Riešenie tohto problému ponúkajú *vlákna* [2], ktoré možno charakterizovať ako odľahčené procesy. Proces môže byť tvorený viacerými vláknami a každé vlákno vždy patrí do určitého procesu. Na rozdiel od procesov, vlákna v rámci jedného procesu zdieľajú spoločné zdroje ako napríklad informácie o správe pamäti, súborové deskriptory a pod., a preto vyžadujú menšiu réžiu pri behu. Každé vlákno má však vlastný stav, programové počítadlo, zásobník aktivačných záznamov a stav registrov CPU. Na jednej strane zdieľanie zdrojov zjednodušuje komunikáciu vlákien medzi sebou, ale na strane druhej, zdieľanie zdrojov vo

viacvláknových programoch vedie k ťažšie odhaliteľným chybám vznikajúcim pri prístupu k zdieľaným prostriedkom akými sú napríklad pamäť a synchronizačné primitíva. Pojem *preloženie vlákien* možno chápať ako postupnosť vykonania inštrukcií rôznych vlákien počas behu. Keďže vlákna nie sú vykonávané sekvenčne, jedno za druhým, ale počas behu dodchádza k prepínaniu kontextu, existuje síce konečný, ale veľmi veľký počet možných preložení vlákien programu.

2.3 Prideľovanie výpočtového času

V kapitole 2.1 bolo zmienené, že koncept paralelného programovania bol motivovaný efektívnejším využitím viacjadrových procesorov. Súčasne však platí, že počet bežiacich vlákien v drvivej väčšine prípadov prevyšuje počet jadier vo výpočtovom systéme. Rovnako ako u jednoprocesorových architektúr teda ostáva nutnosť riešiť prerozdeľovanie výpočtového času medzi viacero jednotiek.

Stratégia, na základe ktorej sa bude prideľovať výpočtový čas jednotlivým vláknam sa nazýva *plánovacia stratégia* [10]. O prerozdedeľovaní procesorového času rozhoduje operačný systém, konkrétne jeho časť nazývaná *plánovač* [10]. Plánovač na základe zvolenej plánovacej stratégie určuje ktorému vláknu a na akú dlhú dobu bude pridelený procesor. Prepnutie medzi vláknami sa nazýva zmena kontextu. Prepínanie kontextu je zdrojom nedeterminizmu v paralelných programoch, ktorý absentuje v sekvenčných programov. Nedeterminizmus v programe komplikuje pátranie po zdroji chyby a tiež zopakovanie spustenia programu, ktoré viedlo k chybe, pretože spustenie programu s rovnakým vstupom neimplikuje rovnaký výstup. Jedným z prístupov, ktorý sa pri testovaní viacvláknových programov používa je virtualizácia behového prostredia, bližšie popísaná v kapitole 5, ktorá umožňuje získať kontrolu nad vykonávaním programu.

Kapitola 3

Verifikácia viacvláknových programov

Verifikácia viacvláknových programov [6] je oblasť informatiky zahŕňajúca niekoľko rôznych techník ako testovanie programov, dynamickú analýzu, statickú analýzu, abstraktnú interpretáciu, dokazovanie teorému a preľadávanie stavového priestoru (model checking). Tieto techniky je možné porovnávať z hľadiska ich schopnosti odhlaliť skutočné chyby. Technika by nemala prehliadnuť chybu, resp. vyhlásiť chybu za správny stav. Jedným z hlavných nedostatkov verifikačných techník je hlásenie tzv. falošných chýb, ktoré v skutočnosti nie sú chybami.

Táto práca sa zaoberá dvoma z týchto techník. Podkapitola testovanie 3.1 sa venuje technike testovania, ktorá je v súčasnosti najpoužívanejšiu metódou. V tejto kapitole budú vysvetlené základné pojmy, ktoré sa v súvislosti s testovaním používajú a pracuje s nimi aj táto práca. V podkapitole 3.2 bude predstavená technika dynamickej analýzy, na ktorej je založený aj nástroj Maple, skúmaný v tejto práci. V podkapitole 3.3 sa nachádza zaradenie chyby, na ktorej hľadanie je zameraná táto práca, do kontextu chýb typických pre viacvláknové programy. Poznať typy chýb, ktoré sa môžu vo viacvláknových programoch vyskytovať je základným predpokladom pre ich úspešné odhalenie a odstránenie, ideálne však samozrejme sa je sa im vyhýbať už pri programovaní.

3.1 Testovanie viacvláknových programov

Testovanie je najpoužívanejšia technika pre odhaľovanie chýb v programoch hlavne pre svoju relatívnu jednoduchosť. Testovanie obnáša (opakované) spúšťanie programu s cieľom vyvolať chybu, ktorá sa v programe potenciálne nachádza. Praktickým zmyslom testovania je zvýšenie kvality softvéru.

Všeobecný pojem chyba programu môže mať viacero významov:

1. *Vada* (fault) je statický defekt softvéru.
2. *Chyba* (error) označuje nekorektný stav systému, ktorý nastal prejavom defektu.
3. *Zlyhanie* (failure) je vonkajší prejav nesprávneho správania systému.

Pri testovaní sa často pracuje s pojmom *špecifikácia požiadaviek*, čo je popis korektného správania systému, vrátane popisu vstupných a výstupných dát. *Testovací prípad* je

definovaný vstupmi a zodpovedajúcimi výstupmi programu. Jednotlivé testovacie prípady tvoria *testovú sadu*. Pri testovaní sa vytvárajú testovacie prípady, ktoré sú následne vykonané. Ak pri vykonávaní nastala chyba, znamená to prítomnosť chyby v programe alebo v testovacom prípade. Väčšinou sa dozvieme o výskyte chyby, ale nie je známy jej pôvod, ktorý je potrebné dodatočne odhaliť. Pre detekciu chyby sa často používa tzv. výraz typu *assert* vpísaný do kódu programu. Tento výraz obsahuje výraz s pravdivostnou hodnotou, ktorý v danom mieste programu musí byť vždy pravdivý. Vyhodnotenie tohto výrazu ako nepravdivého spôsobí prerušenie vykonávania programu a chybové hlásenie.

Testovanie je technika, ktorá neprodukuje falošné chyby, ale zároveň neodhalí všetky chyby v programe. Neposkytuje teda garanciu o bezchybnosti programu, ako napríklad statická analýza, ale s ohľadom na svoju jednoduchosť je ideálna pre použitie pri nekritických programoch.

K testovaniu existuje viacero prístupov, ktoré sa v zásade delia na náhodný a systematický prístup. Náhodné testovanie obnáša spúšťanie programu s náhodnými vstupmi. Nevyžaduje žiadne znalosti testera – osoby zoberajúcej sa testovaním programov, na druhej strane ale nie je príliš efektívne v odhaľovaní chýb. Oproti tomu systematické testovanie je veľmi závislé na schopnostiach a skúsenostiach testera a môže byť veľmi efektívne. Systematické testovanie vyžaduje vhodne zvolenú množinu testovacích prípadov.

Pri testovaní je dobrým zvykom použitie analýzy pokrytia ako metriky pre vyhodnocovanie testovania. *Pokrytie* [3] vyjadruje percentuálny podiel jednotiek, napr. riadkov, výrazov alebo ciest, v programe, ktoré sa podarilo pri testovaní dosiahnuť.

Testovanie viacvláknových programov sa od testovania sekvenčných programov zásadne líši tým, že pri rovnakých vstupoch nedostaneme vždy rovnaké výstupy. To je spôsobené rôznym preložením vlákien pri rôznych behoch. Z toho vyplýva, že pri nájdení chyby počas behu programu nie je jednoduché tento beh programu zopakovať. Tiež z toho vyplýva, že ak nejaký beh programu nevyvolal chybu, neznamená to, že v inom behu s rovnakými vstupmi chyba tiež nenastane. Tento nedeterminizmus komplikuje používanie rovnakých metrík pokrytia ako u sekvenčných programov.

3.2 Dynamická analýza

Dynamická analýza je technika verifikácie viacvláknových programov, ktorá je založená na zbieraní a analýze informácií získaných počas behu programu. Na rozdiel od statickej analýzy vyžaduje (opakované) spúťanie programu, preto sa nazýva aj behová verifikácia. Analýza informácií môže prebiehať priebežne za behu alebo až po skončení vykonávania programu.

Hlavnou výhodou dynamickej analýzy je, že na rozdiel od techník ako statická analýza a model checking neskúma všetky možné preloženia vlákien, ale len preloženia, ktoré skutočne nastali pri spúšťaní programu. Tento fakt na jednej strane výrazne zmenšuje skúmaný priestor, na druhej strane ale neposkytuje záruku, že boli preskúmané všetky preloženia vlákien, ktoré môžu pri vykonávaní programu nastať. Avšak na rozdiel od testovania, dynamická analýza môže byť schopná odhaliť aj chyby, ktoré sa nenachádzajú priamo na vykonanej ceste programu.

Problémom dynamickej analýzy je tzv. *sondový efekt* (probe effect) [6]. Označuje skutočnosť, že trasovanie programu ovplyvňuje samotný beh verifikovaného programu. Tento fakt nie je možné príliš ovplyvniť, je potrebné naň však brať ohľad pri diskusii o výsledkoch analýzy.

3.3 Klasifikácia chyby

Táto podkapitola je zameraná na popis typu chyby, ktorej odhalenie je predmetom tejto práce. Názvy a dokonca aj definície chýb sa často v rôznych zdrojoch líšia. Táto práca vychádza z rozdelenia [4], ktoré delí chyby na dve základné skupiny a to *chyby bezpečnosti* a *chyby živosti*. Zameriame sa na chyby bezpečnosti, medzi ktoré patria chyby typu porušenie atomicity, porušenie poradia, uviaznutie a zmeškaný signál a časovo závislé chyby nad dátami.

Chyby bezpečnosti porušujú vlastnosti programu, ktoré musia byť vždy splnené. Tieto chyby vždy vedú k nejakému nežiaducemu (chybovému) stavu. Ďalej bude predstavený typ časovo závislej chyby nad dátami.

Časovo závislé chyby nad dátami Program obsahuje časovo závislú chybu nad dátami (data race) [4] pokiaľ obsahuje dva nesynchronizované prístupy k zdieľanej premennej a aspoň jeden z nich je zápis. Na identifikáciu tejto chyby je potrebné mať znalosť o tom, ktoré premenné sú zdieľané viacerými vláknami a ako sú prístupy k týmto premenným synchronizované. Detekcia časovo závislých chýb je dobre študovaným problémom a na jej riešenie existuje viacero techník, jednou z nich je práve dynamická analýza.

Príkladom takejto chyby je napríklad inkrementácia zdieľanej premennej dvoma vláknami. Tento príklad je znázornený na obrázku3.1, kde dva stĺpce predstavujú dve vlákna T1 a T2 a riadky predstavujú vykonanú postupnosť inštrukcií. Stĺpce T1.x a T2.x predstavujú hodnotu premennej s ktorou pracuje daná inštrukcia. Chyba sa prejaví tým, že hodnota zdieľanej premennej sa inkrementuje len jedenkrát, pretože každé vlákno inkrementuje svoju lokálniu kópiu premennej.

T1	**T2**	**T1.x**	**T2.x**
load		0	
	load		0
inc		1	
	inc		1
store		1	
	store		1

Tabulka 3.1: Príklad časovo závislej chyby nad datami pri inkrementácii zdieľanej premennej

Kapitola 4

Program Pin

Táto kapitola predstavuje ďalšie dôležité techniky na ktorých je založená táto práca a to techniky virtualizácie a inštrumentácie. Na nich je postavený inštrumentačný nástroj Pin, ktorý je pužitý v praktickej časti tejto práce. Popis samotného nástroja Pin tvorí jadro tejto kapitoly, pričom sú spomenuté len nutné základy, kompletný popis sa nachádza v manuále[1].

4.1 Virtualizácia

Virtualizácia je technika, ktorá zahŕňa vytváranie virtuálnej verzie hardvérového alebo časti softvérového systému. Výsledkom virtualizácie je *virtuálny stroj*, ktorý je emuláciou počítačového systému. Virtualizácia je v súčasnosti veľmi populárna technika, čo dokazuje jej hardvérová podpora v CPU a aj to, že sa jej venujú viaceré popredné počítačové firmy ako napríklad VMware, Oracle a Microsoft.

Motivácia pre použitie virtuálneho stroja môže byť rôzna, môže slúžiť ako zapuzdrenie nižšej vrstvy systému, zaistenie prenositeľnosti alebo ako alternatíva vytvorenia určitého prostredia pre beh programov. Cieľom virtualizácie je v kontexte tejto práce získanie kontroly nad vykonávaným programom. Nástroj virtualizácie môže vykonávanie programu priamo riadiť alebo môže slúžiť len na zber informácií o behu programu. Základom tejto techniky je čo najvernejšie napodobniť reálne behové prostredie, aby takto získané poznatky bolo možné aplikovať na reálne situácie.

4.2 Inštrumentačný nástroj Pin

Inštrumentácia [8] programu je technika, ktorá zahŕňa vkladanie kódu do inštrumentovaného programu za účelom monitorovania správania programu. Pri inštrumentácií často dochádza k zberu informácii o behu programu. Inštrumentácia môže byť realizovaná staticky – a to na úrovni zdrojového kódu alebo na úrovni binárnej reprezentácie programu, alebo dynamicky – za behu programu. V tejto práci sa používa varianta dynamickej inštrumentácie programu v binárnej forme za behu, ktorá bude vysletlená v nasledujúcim odstavci o samotnom nástroji Pin.

Pin je dynamický binárny inštrumentačný nástroj vyvinutý firmou Intel, ktorý poskytuje platformu pre vytváranie dynamických analyzačných programov. Je voľne dostupný pre nekomerčné účely. Pin umožňuje abstrahovať od systémových volaní tým, že poskytuje API (Application Programming Interface) vrstvu pre nástroje na ňom postavené. Pojem API sa prekladá ako rozhranie pre programovanie aplikácií a označuje súbor programových celkov, ktoré programátor volá namiesto písania vlastného kódu. Vstupom Pinu je priamo spustiteľný program. Tento fakt vedie k dojmu, že Pin je platformovo závislý, ale nie je to pravda. Väčšina jeho API je nezávislá na architekrúre a prenositeľná medzi operačnými systémami. Skompilované programy v binárnej forme inštrumentuje za behu, a teda inštrumentované programy nie je potrebné celé znova kompilovať.

Pin má dva módy činnosti – tzv. JIT (Just In Time) mód, čo sa dá preložiť ako okamžitý mód a sondový mód (Probe mode). Ďalej v práci sa bude predpokladať prvý zmienený mód, pretože ten využíva k svojej činnosti program použitý v tejto práci. Z podľadu tohto módu Pin funguje ako Just in time (JIT) kompilátor, teda kompilátor ktorý kompiluje daný kus kódu až keď je to potrebné t.j. keď sa kód má práve vykonať. Vďaka tomu, aj keď ide o viacvláknové programy, kompilátor produkuje sekvenčný kód. Iným názvom sa táto činnosť označuje ako *trasová inštrumentácia* (trace instrumentation). Princípiálne funguje tak, že Pin vždy na základe spustiteľného súboru vygeneruje sekvenciu kódu a odovzdá jej riadenie. Táto sekvencia je takmer identická s originálnym kódom testovaného programu, s tým rozdielom, že Pin si zaručí prevzatie kontroly po skončení tejto sekvencie. Pri generovaní kódu, ktorý sa vykoná, Pin umožňuje nástroju nad ním vkladať do vykonávaného programu vlastný kód. Vďaka tomu je teda možné napríklad pri inštrukcii zápisu do pamäti túto udalosť zaznamenať alebo dokonca pozmeniť.

4.3 Zásuvné moduly programu Pin

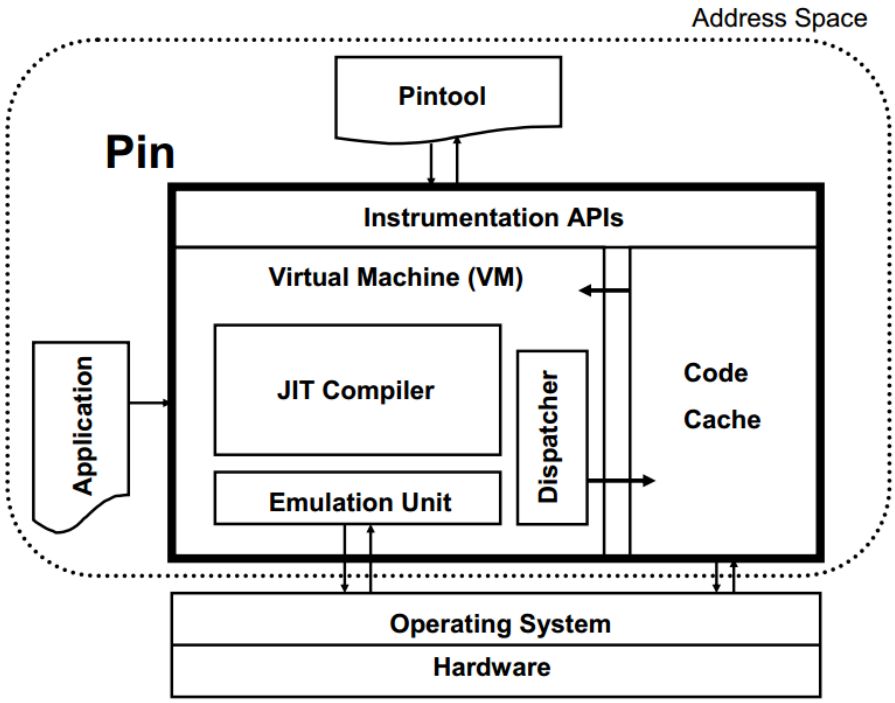
V predchádzajúcej kapitole boli spomenuté možnosti programu Pin, ktoré poskytuje prostredníctvom svojho rozhrania. Programy, ktoré toto rozhranie využívajú a sú naprogramované na konkrétne použitie sa nazývajú *Pintools*. Práve jedným z takýchto nástrojov sa zaoberá táto práca.

Pintool je v podstate zásuvný modul (plugin), ktorý vie upravovať proces generovania kódu vnútri Pinu. Pintool sa skladá z dvoch komponent:

1. Mechanizmu, ktorý rozhodne kde a aký kód bude vložený.
2. Kódu na vykonanie v bodoch vloženia.

Tieto dve komponenty fungujú v jednom spustiteľnom súbore. Keďže Pintool zdieľa adresový priestor s Pinom a inštrumentovaným programom, má prístup ku všetkým datám inštrumentovaného programu a ďalším informáciám o procese.

Na obrázku 4.1 sa nachádza schéma popisujúca vzájomné fungovanie nástroja Pin, programu typu Pintool a testovaného programu. V spoločnom adresovom priestore sa nachádza testovaný program (Application) a Pintool. Program Pintool komunikuje s nástrojom Pin prostredníctvom jeho rozhrania (Instrumentation APIs). Pin vyvtára virtuálny stroj (Virtual Machine VM), v ktorom sa nachádza spomínaný kompilátor (JIT



Obrázek 4.1: Schéma fungovania Pintools [7]

Compiler), emulačná jednotka (Emulation Unit) a plánovač (Dispatcher). Ďalej je v schéme znázornená vyrovnávacia pamäť obsahujúcu kompilovaný kód (Code Cache). Nástroj Pin komunikuje ďalej so samotným operačným systémom.

Jedným z Pintool nástrojov je aj *Pinplay* [9], ktorý umožňuje opakované prehrávanie spustenia programu, ktoré viedlo k chybe. Pinplay verzie 2.1 je kompatibilný s hlavným nástrojom tejto práce, tiež typu Pintool – programom Maple.

Kapitola 5

Program Maple

Táto kapitola je venovaná programu Maple, ktorý tvorí jadro práce. Prvá kapitola 5.1 predstavuje program Maple, na čo slúži a aké je jeho uplatnenie. Ďalšia kapitola 5.2 predstavuje metriku pokrytia vytvorenú autorom Maple, na ktorej je založené testovanie. V kapitole 5.3 je objasnený priebeh testovania a rozdelenie na fázy a iterácie. Kapitola 5.4 objasňuje princíp predpovedania preložení vlákien pre testovanie, ktorý sa odohráva v prvej fáze činnosti programu – profilovacej fáze. Na túto kapitolu nadviaže nasledujúca kapitola 5.5, ktorá bližšie popíše scenár aktívneho testovania preložení, predpovedaných v profilovacej fáze. V záverečnej kapitole je uvedený zoznam a verzie programov potrebné k používaniu programu Maple.

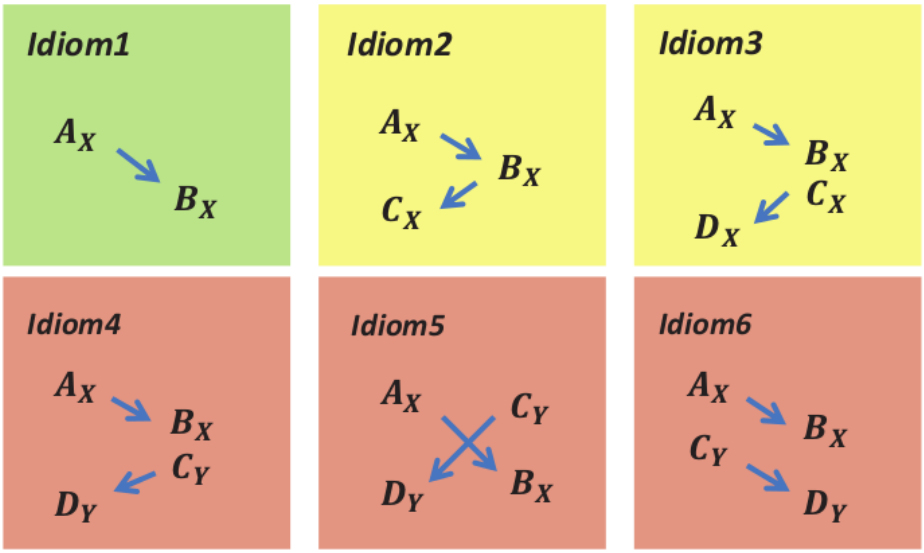
5.1 Charakteristika programu Maple

Maple je voľne šíriteľný program pod licenciou Apache. Bol vyvinutý na Michiganskej univerzite doktorom Jie Yu ako súčasť dizertačnej práce na tému Hľadanie a tolerovanie chýb v paralelných programoch [12]. Program Maple slúži na testovanie viacvláknových programov napísaných v jazyku `C++` s využitím knižnice pthread.h. Samotný Maple je napísaný v jazyku `C++` v kombinácii s jazykom Python. Pozostáva zo zdieľaných knižníc (*.so) a sady skriptov a funguje v spolupráci s programom Pin popísaným v kapitole 4. Prakticky to znamená, že exekúcia je v réžii Pinu, s tým, že program Maple zadá prostredníctvom rozhrania Pinu, kedy chce prevziať riadenie. V našom prípade ide o udalosti prístupu k pamäti, vytvárania a zanikania vlákien a podobne.

5.2 Metrika pokrytia

V tejto kapitole je predstavená metrika pokrytia, ktorá určuje priebeh testovania. Na rozdiel od sekvenčných programov, kde sa často používajú metriky pokrytia riadkov, výrazov a podobne, pre viacvláknové programy je situácia odlišná. Dôvody tohto problému boli diskutované v kapitole 3. Autor programu Maple preto pre účely vyhodnocovania pokrytia programu identifikoval vzory medzivláknových závislostí tzv. *idiomy preloženia* (interleaving idioms) [13]. Pred ich definícou je potrebné vysvetliť jeden dôležitý pojem.

Medzivláknová pamäťová závislosť je okamžitá závislosť medzi prístupmi k pamäti uskutočnených z dvoch vlákien. Za prístup k pamäti sa považuje buď prístup k dátam,



Obrázek 5.1: Kanonické idiomy [13]

alebo k synchronizačným primitívam.

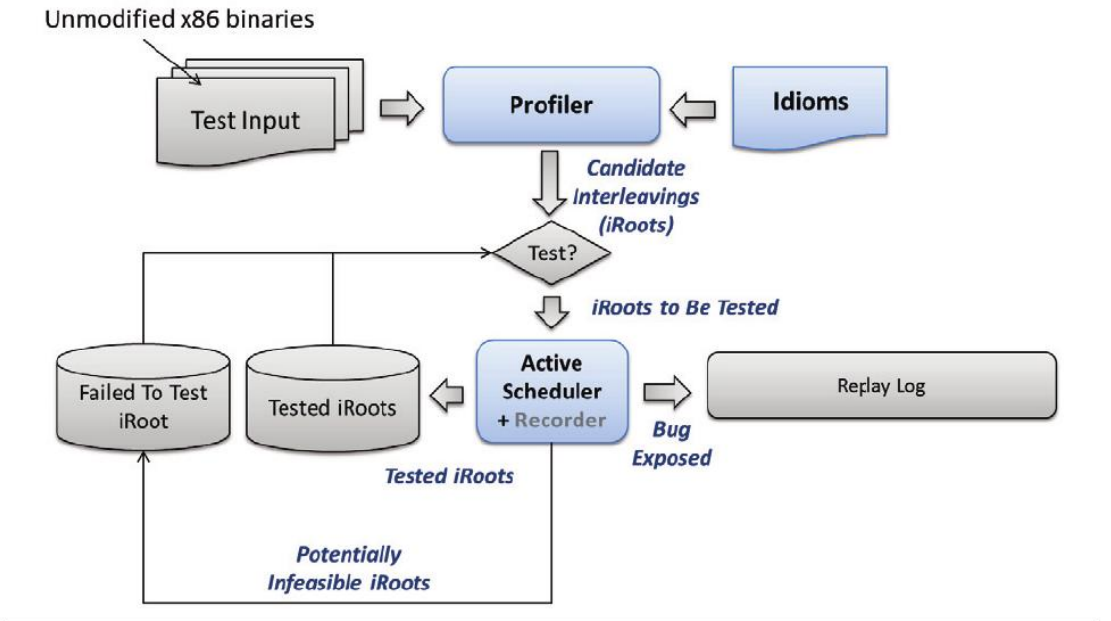
Idiom je vzor medzivláknových závislostí a súvisiacich pamäťových operácií. Idiomov je celkom šesť typov, pričom program Maple pracuje len s prvými piatimi typmi.

Na obrázku 5.1 sú znázornené spomínané základné typy idiomov. Prístupy k pamäti sú v každom idiome znázornené v dvoch stĺpcoch, pričom každý stĺpec znázorňuje jedno vlákno. Každý prístup je identifikovaný veľkým písmenom s indexom, ktorý identifikuje miesto v pamäti, ku ktorému sa aktuálne pristupuje. Idiom1 je jednoduchý idiom a znázorňuje, že medzi prístupom A_x a B_x sa nemôže vyskytnúť iný prístup k miestu x z akéhokoľvek vlákna. Ostatné idiomy sa nazývajú zložené. Napríklad schéma idiomu3 je interpretovaná tak, že k prístupu k miestu x nesmie dôjsť medzi A_x a B_x ani C_x a D_x, ale môže k nemu dôjsť medzi B_x a C_x.

Inštancia idiomu sa nazýva *iRoot* (interleaving root), čo je možné preložiť ako koreň preloženia. Prístup k pamäti v iRoote je identifikovaný statickou adresou pristupujúcej intštrukcie.
Pokrytie sa v tomto kontexte vyjadruje ako počet iRootov, ktoré boli splnené v niektorom testovom behu programu.

Princíp činnosti programu Maple vychádza z dvoch hypotéz, interpretovaných v kontexte viacvláknových programov:

1. Small scope hypothesis
   Väčšina chýb vo viacvláknových programoch môže byť odhalená pri použití malého počtu prepnutí vlákien.



Obrázek 5.2: Architektúra programu Maple [13]

2. Value – independence hypothesis
   Väčšina chýb vo viacvláknových programoch je odhalená, ak sú dosiahnuté medzivláknové pamäťové závislosti vyvolávajúce chybu, nezávisle na hodnote dát v pamäti.

5.3 Priebeh testovania

Program Maple pracuje v dvoch fázach činnosti. Každá fáza sa skladá z niekoľkých iterácií, v každej iterácii sa jedenkrát spustí testovaný program. Prvá fáza sa nazýva profilovacia a druhá aktívna. Tieto fázy sa môžu navzájom striedať, ale pravidlom je, že aktívnej fáze musí vždy predchádzať aspoň jedna iterácia profilovacej fázy.

Nasledujú dve kapitoly, ktoré sa podrobnejšie venujú každej fáze, približujú algoritmus identifikácie iRootov a stratégiu ich pokrývania. Vychádzajú z teoretických základov programu Maple zo zdrojov, ale sú doplnené popis o konkrétnych riešení.

Obrázok 5.2 znázorňuje činnosť programu. Pred začiatkom testovania je potrebné mať k dispozícii program v binárnej forme, ktorý chceme testovať na výskyt chyby (Test Input) a definíciu idiomov (Idioms), ktoré sú už súčasťou programu Maple. Testovanie sa začína profilovacou fázou (Profiler), ktorej výstupom sú kandidátne preloženia vlákien (iRoots). Tieto preloženia sú ukladané do databázy iRootov, ktorá sa používa pri aktívnej fáze. Aktívnu fázu má v režii aktívny plánovač programu Maple (Active Scheduler). Pred každou aktívnou iteráciou prebehne rozhodnovanie (Test), ktoré určí, ktorý iRoot z databázy sa bude plánovač v danom behu pokúšať splniť. Maple v aktívnej fáze teda dostane na vstupe iRoot a na výstupe ho podľa výsledku zaradí medzi dosiahnuté iRooty (Tested iRoots) alebo ho ponechá v databáze, s počítadlom neúspešných pokusov o jeho splnenie (Failed To Test iRoot). Po dosiahnutí určitého počtu neúspešných pokusov sa iRoot vyradí z databázy kandidátov a označí sa ako potenciálne nedosiahnuteľný. V prípade, že sa počas behu preja-

vila chyba programu, program Maple vytvorí záznam v zázname pre prehrávanie (Replay Log) pre prípadné zopakovanie chybného behu na želanie testera.

5.4 Profilovacia fáza

Táto kapitola vysvetľuje princíp úvodnej, profilovacej fázy, v ktorej sa predpovedajú iRooty, s ktorými sa bude pracovať neskôr v aktívnej fáze. Prvá podkapitola napovie ako sa tieto iRooty skladujú, druhá objasní samotný profilovací algoritmus pre predpovedanie iRootov aplikovaný v programe Maple.

Prístup k pamäti bude ďalej označovaný ako A_x, kde veľké písmeno A označuje statickú adresu inštrukcie a dolný index x bude označovať miesto v pamäti.

5.4.1 Databáza iRootov

IRooty sú ukladané do *databázy iRootov* vo formáte definovanom mechanizmom pre ukladanie serializovaných dát *protobuf* [11]. Pre zobrazenie tohto formátu dát v textovej podobe obsahuje program Maple skripty, ktoré sa spúšťajú parametrami z príkazového riadku. Okrem databázy iRootov je možné napríklad zobraziť zoznam iRootov, ktoré sa Maple snažil úspešne alebo neúspešne splniť v každej iterácii. Prístup k obsahu týchto súborov je dôležitý k pochopeniu vnútorého princípu fungovania programu Maple.

5.4.2 Online profilovací algoritmus

V nasledujúcich riadkoch budú predstavené možné riešenia problému predpovedania iRootov a skutočné riešenie – online profilovací algoritmus, ktoré Maple používa. Konečné riešenie vzniklo kombináciou predchádzajúcich prístupov. Detailnejší popis algoritmu a optimalizácií sa nachádzajú v článku o programe Maple [13].

Naivný prístup Naivný prístup predstavuje vytvorenie všetkých možných variácií z množiny prístupov k pamäti.

Non-mutex happens before analýza Táto technika predpovedá iRooty na základe pozorovania niekoľkých behov programu pre daný testovací vstup. Vyberá iRooty, ktoré sú označené ako realizovateľné v niektorom z behov programu. Sledujeme, že niektoré časové závislosti ostávajú rovnaké v každom behu programu a tie sa použijú na vylúčenie potenciálne nedosiahnuteľných iRootov. Pre predstavu, takéto nemenné závislosti môže vytvárať napríklad operácia *create*, *join*, použitie *bariéry* v programe a podobne.

Tento prístup je síce efektívny pri zmenšovaní množiny iRootov, avšak môže spôsobiť vynechanie dosiahnuteľného iRootu. Je to preto, že z toho, že iRoot sa javí nesplniteľný v niekoľkých behoch programu, nie je možné odvodiť záver, že je nie je splniteľný v žiadnom behu.

Analýza vzájomného vylúčenia Vzájomným vylúčením sa myslí zaistenie výlučného prístupu k zdroju použitím zamykacích mechanizmov. Analýzu ilustruje nasledujúci príklad. Majme tri prístupy k miestu x v pamäti – prístup A_x a následujúci prístup B_x sa nachádzajú v jednej kritickej sekcii a prístup C_x v inej kritickej sekcii, avšak obe kritické

sekcie sú chránené rovnakým zámkom m. Predpovedať postupnosť $A \Rightarrow C$ a $C \Rightarrow B$ je zbytočné, pretože nie je dosiahnuteľná, kvôli zamykaciemu obmedzeniu. Pri analýze, ktorá túto skutočnosť odhalí, používa Maple dve informácie a to informáciu o množine zámok, ktoré drží vlákno pri prístupe A_x a informáciu o tom, či ide o prvý alebo posledný prístupom k x v danej kritickej sekcii.

Online profilovací algoritmus Algoritmus skutočne použitý v profilovacej fáze vznikol ako kombinácia dvoch predchádzajúcich prítupov. Pre každý objekt (miesto v pamäti alebo zámok) je uchovávaná história prístupov pre každé vlákno. Prístupy v histórii sú zoradené podľa poradia vykonania daného vlákna a sú asociované s *vector clock* algoritmom a *anotovanou množinou zámok* (annotated lockset). Vector clock algoritmus je použitý na realizáciu vyšie spomenutej non-mutex happens-before analýzy a anotovaná množina zámok je použitá pre analýzu vzájomného vylúčenia.

5.5 Aktívna fáza

Fáza, ktorá nasleduje po profilovacej fáze má za cieľ dosiahnuť čo najviac iRootov, ktoré do databázy umiestnila profilovacia fáza. IRooty sa snaží dosiahnuť aktívnym testovaním prostredníctvom plánovača programu Maple, preto sa nazýva aktívna fáza. Aktívna fáza sa skladá z iterácií, pričom v každej iterácii sa Maple pokúša splniť jeden iRoot z databázy iRootov. Podľa parametrov pri spustení sa iteruje buď do nájdenia chyby alebo vyprázdenia databázy. Je možné aj spustenie len jednej iterácie. Pochopenie fungovania aktívnej fázy bolo z dôvodu absencie dokumentácie nutné skúmať zo zdrojových kódov. Jadro aktívnej fázy sa nachádza v súbore *src/idiom/scheduler_common.cpp*.

5.5.1 Princíp priorít

V úvode tejto kapitoly bolo spomenuté, že o dosahovanie iRootov sa snaží aktívny plánovač programu Maple. Robí tak prostredníctvom nastavovania priorít jednotlivým vláknam.

Prepokladajme, že máme kandidátny iRoot typu prvého idiomu $A \Rightarrow B$, ktorý sa Maple aktuálne snaží splniť. A a B sa nazývajú *kandidátne inštrukcie*. Počas behu programu môžu byť s jednou kandidátnou intrukciou asociované viaceré dynamické prístupy k pamäti. Plánovač po tom, ako nastane vo vlákne T_1 udalosť A, oddiali beh tohto vlákna a čaká na výskyt udalosti B vo vlákne T_2. Podľa tzv. *naivného prístupu* by mohlo dôjsť k uviaznutiu, v prípade, že by vlákno T_2 muselo čakať napríklad na bariére. Ako riešenie tohto problému je možné použiť *časovač*, po vypršaní ktorého by bolo umožnené vláknu T_1 pokračovať. Použitie časovača však vedie k problému zvolenia adekvátneho časového okna. Jeho optimálna hodnota sa líši pre rôzne výkonné systémy a môže spôsobiť prílišné spomaľovanie behu alebo naopak predčasné vzdávanie sa pokusu o splnenie. Ďalším možným riešením by bolo detekovať uviaznutie, čo by vyžadovalo sledovať stav každého vlákna.

Maple uplatňuje iný prístup, a to spoliehanie sa na vrstvu operačného systému, ktorá sama pozná stav každého vlákna. Je to možné vďaka používaniu *nepreemptívnych striktných priorít* a tomu, že vlákna nútene bežia na jednom procesore. Výsledkom je to, že vlákno s nižšou prioritou nikdy nebude uprednostnené pred vláknom s vyššou prioritou, pokiaľ toto vlákno nie je blokované. Z dôvodu prenositeľnosti využíva Maple *plánovač reálneho času* (real time scheduler) poskytovaný Linuxovým jadrom, ktorý je však len aproximáciou

zmieneného ideálneho plánovača. Pri experimentoch však tento princíp nefungoval, práve kvôli neprítomnosti ideálneho plánovača a k uviaznutiam dochádzalo bez úspešnej detekcie.

Ďalším problémom je absencia kontroly nad poradím vykonávania vlákien pred udalosťou, čo môže viesť k nedosiahnuteľnosti niektorého stavu napríklad v súvisloti so zámkami. Princíp riešenia spočíva v priradení rôznych počiatočných priorít vláknam v poradí ich vytvorenia, a to najprv od najnižšej po najvyššiu prioritu a potom naopak. Tento princíp sa nazýva *komplementárne plánovanie*.

Poslednou charakteristikou aktívneho plánovača, ktorá bude spomenutá je *optimalizácia sledovacieho módu*. Ak sa po udalosti A a následnom odložení vlákna T_1 nepodarí vo vlákne T_2 uskutočniť udalosť B pred tým, ako je z nejakého dôvodu zablokované, prechádza program do sledovacieho módu. V sledovacom móde sa zaznamenáva každý prístup k pamäti až pokým nenastane očakávaná udalosť B alebo dôjde k prístupu, ktorý porušuje podmienku splnenia cieľového iRootu, t.j. nenastane prístup k danému pamäťovému miestu z iného vlákna. Sledovací mód je implementovaný využitím selektívneho inštrumentovania nástrojom Pin.

5.5.2 Výber iRootu

V predchádzajúcej kapitole bolo uvedené, že, jednej kandidátnej inštrukcii môže zodpovedať viac dynamických prístupov počas behu programu. Úlohou plánovača je rozhodnúť, ktorý z týchto prístupov vezme do úvahy. Pre ilustráciu majme opäť iRoot $A \Rightarrow B$ vzťahujúci sa k miestu x v pamäti a predokladajme, že program sa nachádza v stave, kedy už došlo k dosiahnutiu udalosti A.

Prvou situáciou je, že v tomto stave nastane prístup k danému miestu x v inom vlákne, ktorý tiež zodpovedá kandidátnej inštrukcii A. Plánovač má na výber, či bude brať do úvahy skorší alebo neskorší prístup k pamäti.

Iným možným scenárom je, že iné vlákno dosiahne kandidátnu intrukciu B, ale pre iné miesto v pamäti y. Opäť sa vytvára priestor pre voľbu, ktorý prístup sa zaznamená.

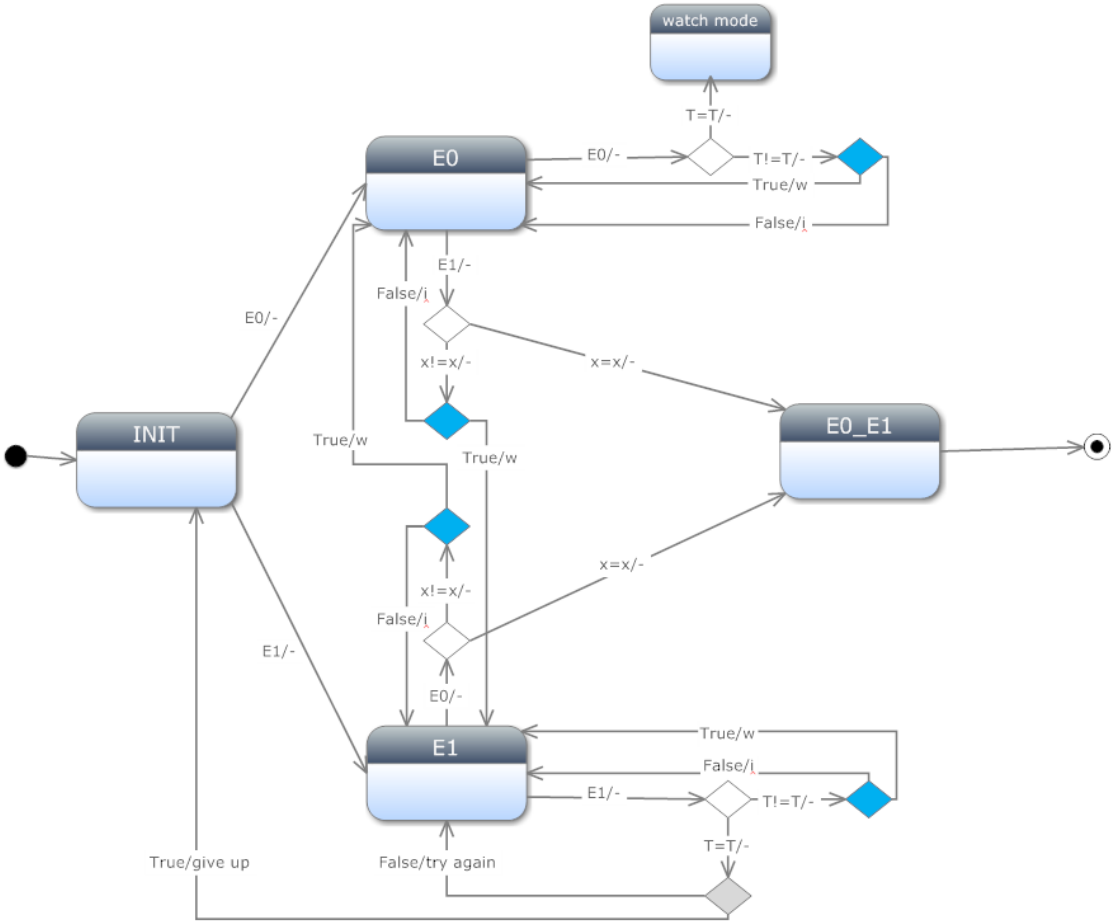
Tieto situácie sa rozhodujú náhodne s 50%-nou pravdepodobnosťou vo funkcii s názvon *RandomChoice*.

5.5.3 Implementácia algoritmu výberu iRootu

Na začiatku testovacieho behu sa vyberie iRoot z databázy iRootov, ktorý sa bude plánovač Maple snažiť dosiahnuť.IRoot sa vyberá podľa kľúčov v nasledovnom poradí: idiom (1 až 5), počet predošlých pokusov a identifikátor ID iRootu. Najprv sa vyberie iRoot najnižšieho idiomu s najmenším počtom spustení a najnižším ID.

Plánovač potom prechádza medzi stavmi, ktoré odrážajú fázu dosahovania iRootu. V nasledujúcom texte sa budú používať identifikátory stavov použité v zdrojvých kódoch a bude sa používať slovné spojenie "nastaviť cieľ", čo znamená určenie vlákna, ktoré má aktuálne najvyššiu prioritu behu. Vlákna budú označované dolným indexom, ktorý označuje ich usporiadanie v poli vlákien, ktoré je dané poradím ich vytvorenia. Implementácia je demonštrovaná na postupe dosiahnutia iRootu prvého idiomu.

Dosiahnutie iRootu typu Idiom1 5.1 znamená zaznamenanie dvoch udalostí, v zdrojovom kóde sú označované ako E0 a E1. Postupnosť stavov, ktorými plánovač prejde je v



Obrázek 5.3: Schéma zjednodušeného stavového automatu

ideálnom prípade:

1. IDIOM1_STATE_INIT
2. IDIOM1_STATE_E0
3. IDIOM1_STATE_E0_E1
4. IDIOM1_STATE_DONE

alebo postupnosť:

1. IDIOM1_STATE_INIT
2. IDIOM1_STATE_E1
3. IDIOM1_STATE_E0_E1
4. IDIOM1_STATE_DONE

Obrázok 5.3 znázorňuje časť stavového automatu charakterizujúceho plánovač programu Maple. Popis vnútorného fungovania plánovača tvorí základ pre neskoršiu identifikáciu potenciálnych miest pre zlepšenie, ktoré budú riešené neskôr v tejto práci. V stavovom diagrame sa nachádzajú štyri hlavné stavy $INIT$, $E0$, $E1$ a $E0\_E1$ a stav *watch mode*, ktorý predstavuje optimalizáciu programu Maple spomínanú v kapitole 5.5.1 V súlade s jazykom UML znázorňuje plné koliečko počiatočný stav a kruh s plným koliečkom vnútri koncový stav. Hrany medzi medzi stavmi majú popis v tvare `udalosť/akcia`. Pod udalosťou sa myslí aj platnosť uvedenej podmienky. Akcia w znamená prepis pôvodnych údajov o udalosti novými, akcia i znamená ignoráciu aktuálnej udalosti a znak pomlčky značí žiadnu akciu. Kosoštvorce označujú rozhodovanie. V diagrame sú odlíšené modrou farbou kosoštvorce, ktoré znázorňujú rozhodovanie funkciou *RandomChoice* a sivou farbou funkciou označenou ako *GiveUp*.

Majme situáciu, kedy sa program nachádza v stave $E0$ a vyskytne sa ďalšia udalosť $E0$. V prípade, že udalosť nastala v rovnakom vlákne ako predchádzajúca udalosť, program prejde do sledovacieho módu a aktuálny stav sa nastaví na *watch mode*. V prípade, že udalosť E0 nastala v inom vlákne ako predchádzajúca, plánovač musí vykonať rozhodnutie. Jedna možnosť je zabudnúť predošlú udalosť $E0$ a zaznamenať práve vyskytnutú udalosť. To konkrétne znamená prepísať údaje uchovávajúce stav, a to identifikačné číslo vlákna a adresu a veľkosť miesta v pamäti, ku ktorému sa pristupuje. Zároveň je potrebné nastaviť ako cieľ aktuálne vlákno T_0. Toto rozhodnutie robí plánovač na základe náhody, volaním funkcie *RandomChoice* s parametrom 0.5, ktorý značí 50%-nú pravdepodobnosť. Ak sa týmto spôsobom náhodne vyberie druhá možnosť, aktuálna udalosť $E0$ sa ignoruje a nemá žiaden vplyv na postup dosahovania iRootu.

Druhou možnosťou je, že program sa nachádza v stave $E1$ a vyskyne sa udalosť $E0$. Najprv sa vykoná kontrola, či sa udalosť $E0$ týka rovnakého pamäťového miesta ako predošlá $E1$. V kladnom prípade zaznamenajú údaje (ID vlákna, adresa a veľkosť pamäťového miesta), za cieľ sa nastaví vlákno T_1 a ako aktuálny stav sa zapíše $E0\_E1$. Tento stav znamená, že iRoot sa podarilo splniť.

V prípade, že miesto v pamäti sa líši, prichádza na rad rozhodnutie volaním funkcie `RandomChoice`, kde sa v prípade pravdivého výsledku prepíšu údaje údávajúce stav, za cieľ sa nastaví druhé vlákno T_1 a prejde sa do stavu $E0$. V prípade, že funkcia vráti nepravdivú hodnotu, udalosť $E0$ sa ignoruje a udalosť nemá žiaden vplyv.

5.6 Závislosti

Maple je podporovaný len na Linuxových platformách a jeho používanie je odporúčané na 64 – bitových architektúrach. Maple závisí na nasledujúcich programoch:

- GNU make, verzia 3.81 alebo vyššia
- Python, verzia 2.4.3 alebo vyššia
- Google protobuf, verzia 2.4.1
- Pin, revízia 62732 alebo vyššia

- (voliteľne) PinPlay, verzia 1.2 alebo vyššia

Postup inštalácie Maple sa nachádza v súbore README.

Kapitola 6

Modifikácia programu Maple

Táto kapitola sa zaoberá analýzou a úpravou programu Maple. Prvá kapitola identifikuje slabé miesta programu Maple a predkladá návrhy na ich odstránenie. Na túto kapitolu nadviaže nasledujúca kapitola, ktorá už popisuje realizované úpravy programu.

6.1 Slabé miesta programu Maple

Táto kapitola diskutuje o slabých miestach programu Maple, ktoré boli nájdené pri jeho štúdiu a experimentovaní. S niektorými z nich sú ďalej vykonané experimenty a niektoré sú ponechané ako námety na ďalšiu prácu. Kapitola stavia na predošlých kapitolách, v ktorých bol kladený dôraz na časti programu Maple, s ktorými sa bude v dokumente ďalej pracovať. Diskutované miesta sú podľa ich povahy rozdelené do podkapitol.

6.1.1 Prahové hodnoty

Na viacerých miestach programu sa vyskytujú prahové hodnoty. Ich hodnoty boli určené na základe autorovho experimentovania, je možné s nimi ďalej experimentovať alebo ich prispôsobovať na základe dodatočne získaných informácií. Ide hlavne o nasledovné hodnoty:

1. Konštanta udávajúcu počet pokusov o splnenie iRootu, po ktorých sa iRoot vyradí z databázy ako nesplniteľný (*DEFAULT_FAILED_LIMIT*).
2. Hodnota udávajúca maximálny počet inštrukcií medzi dvoma prístupmi k pamäti v jednom vlákne (vulnerability window *vw*), súvisiaca s identifikáciou *lokálnych párov* [13] vo vlákne, používaná v profilovacej fáze.
3. Počet dynamických inštrukcií vo vlákne v sledovacom móde, po ktorom sa plánovač vzdá snahy o pokrytie daného iRootu.

Body 2. a 3. v zozname sa týkajú zložených idiomov t.j. idiomov 1 až 5. V testovacej sade programov, ktorá bude predstavená neskôr, boli však identifikované len iRooty typu prvého idiomu. Z toho dôvodu sa týmito konštantami ďalej nebudeme v tejto práci zaoberať. IRooty týchto typov neboli identifikované ani v žiadnom z príkladov priložených autorom programu Maple, čo naznačuje, že zložené idiomy sa týkajú až výraznejšie komplikovaných programov.

6.1.2 Rozhodovanie plánovača v aktívnej fáze

V kapitole o aktívnej fáze je znázornená a popísaná zjednodušená schéma rozhodovania plánovača. Na niektorých miestach sa rozhoduje na základe návratovej hodnoty funkcie *RandomChoice* a *GiveUp*. Funkcie typu *GiveUp* sa rozhodujú na základe informácií, ktoré zbierajú pri svojom volaní. Na rozdiel od nich, funkcia RandomChoice sa rozhoduje náhodne na základe generátoru náhodných čísel. Preto je väčšina experimentov zameraná na ňu. Sľubné je aj to, že sa na základe nej rozhoduje na viacerých miestach programu, čo zvyšuje potenciálny vplyv jej úpravy.

6.1.3 Prerekvizity

Tento problém identifikoval už autor programu Maple a jedná sa o to, že niekedy v programe existujú tzv. *prerekvizity* (pre-conditions), bez ktorých splnenia sa vybraný iRoot nepodarí splniť. Plánovač však o nich v súčasnej verzii programu nevie a preto ich nemôže vynútiť. Jedným návrhom je, že by sa tieto prerekvizity mohli buď generovať automaticky z iRootu. Druhým návrhom je ich možné explicitné zadanie. Jedná sa o pomerne zložité riešenia, preto tento problém prenecháme na možnú neskoršiu prácu.

6.1.4 Čiastočne paralelný beh vlákien pri teste

Vlákna sú nútené bežať na jednoprocesore, aby bolo možné monitorovať a riadiť ich beh. Z dôvodu úspory času vznikol návrh, spúšťať týmto spôsobom len určité časti programov a naopak zvyšné časti nechať bežať paralelne. Riešenie by sa ale muselo realizovať na nízkej úrovni, ktorá už nespadá do kompetencie programu Maple, ale na úroveň programu Pin.

6.1.5 Výber iRootu pre test

Spôsob akým sa vyberá iRoot, ktorého splnenie bude cieľom daného testového behu je čiastočne deterministický – podľa idiomu, počtu spustení a ID. Nedeterminizmus spôsobuje výber iRootu podľa jeho identifikátoru, ktorý závisí od poradia jeho určenia v profilovacej fáze, čo závisí od konkrétnych behov programov, ktoré sa rôznia. Prípadný prepracovanejší výber iRootu by mohol viesť k odhaleniu chýb pri skorších iteráciách, čo by hlavne pri zložitejších programoch znamenalo časovú úsporu.

Ďalšie návrhy Ďalším možným rozšírením programu by bolo pripustiť závislosť výskytu chyby v programe na hodnotách, ktorá sa aktuálne, vychádzajúc z úvodných hypotéz 5.2, neuvažuje.

6.2 Úpravy programu Maple

V predchádzajúcej kapitole boli identifikované potenciálne miesta programu Maple, s ktorými by sa dalo pracovať pri snahe o jeho vylepšenie. Pre účely samotných úprav a zároveň vyhodnocovania efektu týchto úprav bolo základným rozšírením programu Maple zbieranie informácií o behu. Ide o počty volaní konkrétnych funkcií, počty nastavovania priorít vlákien, počty vstupov do sledovacieho módu a detekciu nájdenia chyby. Význam

hodnôt týchto ukazateľov v súvislosti s hodnotením úprav bude rozobraný v ďalšej kapitole o experimentoch.

Z týchto slabých miest bolo najprv vybrané rozhodovanie funkciou *RandomChoice*. Zo stavového diagramu 5.3 vidieť, že táto funkcia sa volá na viacerých miestach programu Maple. Z toho môžme usúdiť, že jej modifikácia by mohla priniesť viditeľný efekt. Táto funkcia jednoducho vygerenuje náhodné číslo a podľa jeho hodnoty vráti hodnotu typu *bool*, t.j. $True$ alebo $False$.

Prvou úpravou bolo uviesť rozhodovanie do extrémov. Namiesto pôvodnej polovičnej pravdepodobnosti, sme zaviedli pravdepodobnosť 0% a 100%. Prípad kedy funkcia vždy vracala hodnotu True, viedol vo všetkých spusteniach v rámci testov k zacykleniu programu. Opačný prípad bude prezentovaný v rámci experimentov v nasledujúcej kapitole.

Ďalšou úpravou bolo zaviesť hodnoty blízke extrémom. Namiesto pôvodnej 50% pravdepodobnosti sme určili pravdepodobnosť pravdivej návratovej hodnoty najprv na 80% a potom na 20%.

Poslednou skupinou úprav týkajúcich sa funkcie *RandomChoice* bola modifikácia správania funkcie za behu programu, a to na základe dodatočne zozbieraných údajov o aktuálnom testovacom behu. Rozhodovanie prebieha na základe počtu volaní tejto funkcie. Opäť boli zvolené dve stratégie a to začínať s nižšou pravdepodobnosťou a po prekročení prahu ju zvýšiť a naopak, začať s vyšou pravdepodnosťou a potom ju znížiť. Hodnota prahu, kedy prebehne zmena správania bola zvolená s ohľadom na počty volaní funkcie počas behov programov z testovej sady.

Ďalej bola realizovaná úprava týkajúca sa procesu výberu iRootu na začiatku iterácie aktívnej fázy. Cieľom bolo zistiť, či sa chyba neprejavuje častejšie pri niektorom type iRootu. Ako tento typ bol zvolený iRoot (idiomu1), ktorý obsahuje dve udalosti typu zápis – zápis a toto bolo zohľadnené pri výbere.

Kapitola 7

Experimenty

Táto kapitola popisuje experimenty uskutočnené s programom Maple na testovej sade programov, popísanej v podkapitole 7.1. Experimenty boli vytvorené na základe analýzy slabých miest a návrhu na ich odstránenie 6.2.

Experimenty číslo 1 (`RC0`) až 5 (`rc90-10`) sú zamerané na rozhodovanie funkcie RandomChoice. Experiment `ww` sa týka výberu procesu iRootu na začiatku aktívnej fázy.

7.1 Testová sada programov

Táto kapitola v krátkosti popisuje sadu testových programoch, na ktorých boli vykonané experimenty. Testová sada sa skladá z programov, ktoré boli súčasťou nástroja Maple a programov prevzatých z práce [5] (t01, t04, t06, t12), do ktorých boli doplnené chýbajúce assert výrazy pre detekciu chyby. Programy v testovej sade obsahujú časovo závislé chyby nad dátami, ktoré budú v tejto kapitole konkrétne popísané.

Vo výsledkových tabuľkách sa budú nachádzať skrátené názvy programov, z dôvodu úspory priestoru.

Program `bank_account` spúšťa dve vlákna, z ktorých jedno vyberie čiastku z bankového účtu a druhé vloží rovnakú čiastku na bankový účet. Chyba je spôsobená tým, že vlákno najprv uloží stav na účte zo zdieľanej premennej do lokálnej premennej, tam ho inkrementuje resp. dekrementuje a potom zapíše do zdieľanej premennej.

V programe `circular_list` sa vytvárajú dve vlákna, ktoré prechádzajú lineárny zoznam a inkrementujú dáta v každej položke. Problém je opäť v tom, že inkrementáciu vykonávajú v lokálnej premennej.

V programe `log_proc_sweep` bežia dve vlákna, z ktorých jedno vykoná logovanie a druhé vykoná reset logovacieho záznamu. Chyba nastane, keď vlákno pristúpi k odkazu na log, ktorý zatiaľ druhé vlákno zmazalo. Výraz assert pre detekciu chyby nie je v tomto prípade potrebný.

Program `mysql_169_extract` vytvorí dve vlákna, z ktorých jedno vykoná vloženie a druhé odobratie položky z tabuľky a akcie zapíšu do logovaciego súboru. Program testuje, či posledný záznam v logovacom súbore zopovedá stavu tabuľky.

Jednoduchý program `shared_counter` obsahuje podobnú chybu ako bank_account, ale v jednoduchej forme bez akýchkoľvek zámkov.

V programe `string_buffer` a vytvorí jedno nové vlákno. Hlavný program vkladá do bufferu dáta a vytvorené vlákno dáta z bufferu odstraňuje. Chyba je detekovaná nesprávnymi označeniami začiatku a konca bufferu.

V programe `t01` je zdieľaná premenná síce nastavená v zamknutej kritickej sekcii, ale je čítaná mimo nej.

V programe `t04` sa ukladajú údaje pre vlákno ako napr. jeho ID do štruktúry, ktorá sa predá vláknu ako parameter. Vlákno si potom uloží údaje do svojich lokálnych premenných, avšak medzitým mohli byť údaje už prepísané v hlavnom programe pri vytváraní ďalšieho vlákna.

Autor programu `t06` spôsobil chybu typu data race pri zápise dát do zdieľanej štruktúry typu *timespec*, ktorá nie je chránená zámkom.

Autor programu `t12` spôsobil chybu tým, že dva zámky, ktoré používa, inicializuje v každom vytvorenom vlákne. To znamená, že počas behu programu sa n-krát zresetujú informácie ako vlastníctvo zámku, stav a ďalšie, čo teoreticky môže viesť k tomu, že viac vlákien môže vstúpiť do kritickej sekcie. Tomuto však často zabráni porušenie výrazu assert v knižnici *pthread*. Porušenie výrazu assert na úrovni knižných volaní rieši Pin, ktorý v tejto situácii vytvorí súbor pin.log s informáciou o chybe.

7.2 Popis experimentov

Každé testovanie sa skladalo z troch profilovacích iterácií a následne n aktívnych iterácí t.j. aktívne sa iterovalo dovtedy, pokým nedošlo k vyprázdeniu databázy iRootov.

Originálnemu programu Maple bez úprav zodpovedá experiment č.0 (`default`). Experiment č.1 (`rc0`) je modifikácia funkcie RandomChoice tak, aby v 0% prípadov vrátila pravdivostnú hodnotu `true`, v experimente č.2 (`rc20`) je vždy pravdepodobnosť hodnoty `true` 20% a v experimente č.3 (`rc80`) 80%.

Experimenty č.4 a č.5 spočívajú v zmene správania funkcie za behu a to tak, že po prekročení prahovej hodnoty počtu volaní, na základe pozorovaní stanovenej na hodnotu 2, sa pravdepodobnosť zmení z 10% na 90% (`rc10-90`) a naopak z 90% na 10% (`rc90-10`).

V prílohe A sa nachádzajú tabuľky priemerných hodnôt údajov pre jednotlivé experimenty pre každý program. Význam stĺpcov v tabuľkách je:

1. identifikátor experimentu
2. trvanie jednej aktívnej iterácie
3. smerodatná odchýlku trvania
4. percentuálny podiel behov kedy sa úspešne podarilo dosiahnuť daný iRoot

5. RC = počet volaní funkcie RandomChoice
6. GU = počet volaní funkcie GiveUp
7. počet prechodov do sledovacieho módu
8. počet nastavovaní priorít vlákien

Z údajov o trvaní iterácie, pomere úspešných iterácií v zmysle pokrytia iRootu a v zmysle vyvolania chyby sú vytvorené súhrnné tabuľky, uvedené a popísané v nasledujúcej v kapitole 7.3.

7.3 Výsledky experimentov

Táto kapitola popisuje výsledky experimentov s úpravami programu Maple. Obsahuje tri tabuľky, porovnávajúce jednotlivé úpravy. Riadky tabuliek predstavujú programy z testovej sady a stĺpce označujú experimenty. Stĺpec s označením `default` označuje počiatočnú verziu programu, ktorá predstavuje základ pre výpočet ostatných stĺpcov. Posledný riadok tabuliek obsahuje priemery jednotlivých stĺpcov, vyjadruje teda priemerný efekt jednotlivých experimentov na programy.

Ďalšiu časť kapitoly tvorí porovnanie pokrytia iRootov a rýchlosti nájdenia chyby, ktorým sa zaoberala posledná úprava demonštrovaná experimentom `ww`.

Trvanie iterácie Tabuľka 7.1 obsahuje priemerné trvanie jednej aktívnej iterácie v sekundách. Hodnoty stĺpcov sú vypočítané ako percentuálna odchýlka danej hodnoty od hodnoty v referenčnom stĺpci (`default`). Čím kratší čas testového behu programu, tým je to pre nás výhodnejšie.

Z posledného riadku tabuľky vidieť, že trvanie iterácie sa najviac znížilo v prípade `rc0`. Tento výsledok spĺňa očakávania, pretože práve v tomto experimente bolo rozhodovanie zúžené len na jednu voľbu, a to striktne sa držať prvého nájdeného prístupu do pamäti zodpovedajúcemu udalosti danému iRootu. V programe *shared_counter* došlo k najväčším percentuálnym výkyvom, čo môže byť spôsobené tým, že tento program bol najmenej zložitým programom v testovej sade, s jedným z najkratších trvaní behu. Preto aj malé, niekedy náhodné zmeny počas behu vyvolali väčší efekt.

Vyvolanie chyby Druhá tabuľka 7.2 obsahuje údaje o percentách iterácií, ktoré spôsobili prejavenie chyby v programe. Vyvolanie chyby je cieľom testovania, preto má táto tabuľka najväčí význam pri posudzovaní úspešnosti testovania z vonkajšieho pohľadu.

Hodnoty stĺpcov sú vyjadrené ako percentuálny nárast oproti hodnote v percentách v referenčnom experimente (`default`).

Z hľadiska odhalenia chyby dopadol najlepšie experiment (`rc80`), ktorý stanovil pravdepodobnosť uprednostnenia novej udalosti prístupu k pamäti pred predošlou na 80%. Úspešným bol aj experiment `rc90-10`, ktorý začínal s vysokou pravdepodobnosťou a po prahovej hodnote ju znížil. Mierne zlepšenie môžme pozorovať aj u experimentu `ww`.

Najhoršie z tohto pohľadu dopadol experiment `rc10-90`, ktorý v začiatkoch výrazne uprednostňoval predošlé udalosti. Zhoršenie nastalo aj v experimente `rc20` a `rc0`.

Z výsledkov môžme usúdiť, že pokiaľ ide o vyvolanie chyby, je výhodné voliť vysoké pravdepodobnosti uprednostnenia novších udalostí. Pri porovnaní údajov o trvaní iterácie z tabuľky 7.1 vidíme, že úpravy ktoré mali lepšie výsledky pri nachádzaní chýb, mali dlhšie trvania behu. Experiment `rc20` vychádza nevýhodne z oboch pohľadov.

V tabuľke 7.2 vidieť veľké výkyvy spôsobené malým podielom výskytov chýb v neupravenej verzii. Rozdiel jedného výskytu, potom spôsobil vysoké percentuálne rozdiely najmä u jednoduchších programov s malým počtom iterácií (spôsobených malou množinou iRootov pre testovanie).

Úspešnosť dosiahnutia iRootu Tabuľka 7.3 obsahuje hodnoty priemernej úspešnosti snahy o dosiahnutie iRootu v jednotlivých behoch vyjadrené v percentách. Tento údaj je dôležitý z hľadiska vnútorného fungovania programu, pretože pokrytie iRootu neznamená odhalenie chyby a naopak. Hodnoty stĺpcov sú vyjadrené ako percentuálny nárast oproti percentuálnej hodnote v referenčnom stĺpci `default`.

Vidíme, že podiel iterácií s úspešným pokrytím sa zvýšil vo všetkých prípadoch, okrem posledného experimentu. Úspešnosť dosahovania iRootu sa najvýraznejšie zlepšila v experimente `rc80`, `rc90-10` a `rc20`.

Okrem pohľadu z troch hľadísk boli pri experimentoch sledované ešte ďalšie skutočnosti vyskytujúce sa v tabuľkách v prílohe A.

Keď sa zameriame na stĺpec zmien priorít, v 8 z 10 programov sa vykonalo najviac zmien prorít vlákien v experimente `rc80`. To je logické, keďže táto úprava podporuje prepisovanie predošlých udalostí novými a pri každom prepise sa zmení cieľ, o ktorý sa Maple v aktívnej fáze snaží. Zmena cieľa znamená zmeny priorít príslušných vlákien. Úprava reprezentovaná `rc80` má v 7 z 10 programov najvyššiu hodnotu počtu volaní funkcie `RandomChoice`, ktorá bola predmetom tejto úpravy.

Zhodnotenie experimentov V dvoch z troch skúmaných hľadísk vyšla najlepšie úprava reprezentovaná experimentom `rc80`. Dobré výsledky boli dosiahnuté aj úpravou `rc90-10` a v dvoch z troch pohľadov vyšli ako výhodnejšie ešte úpravy `rc0`a `rc10-90`. Žiadna z úprav nespôsobila zhoršenie vo všetkých aspektoch.

Rýchlosť pokrytia a nájdenia chyby Úprava `ww` používala nemodifikovaný rozhodovací mechanizmus funkcie `RandomChoice`, jej úprava spočívala v modifikácii výberu iRootu z databázy na začiatku iterácie v aktívnej fáze. Cieľom bolo skúmať, aký efekt dosiahneme uprednostnením iRootu, ktorého obe (v prípade vyšších idiomov by to boli aspoň dve) udalosti sú prístupom k pamäti typu zápis. Hypotézou bolo, že by mohli spôsobovať viac chýb ako iRooty z ktorých jeden prístup k pamäti je typu čítanie.

Výsledky sú demonštrované na grafoch v prílohe B. Pre každý program sú zostrojené dva grafy, z ktorých prvý reprezentuje počet výskytu chýb od začiatku testovania a druhý počet úspešne dosiahnutých iRootov od začiatku testovania, teda vyjadruje pokrytie. V oboch grafoch sú dve krivky, jedna reprezentujúca pôvodný stav bez úprav `default` a druhá úpravu `ww`. Dôvodom prečo grafy znázorňujúce pokrytie začínajú na číslach vyšších než jedna je, že do pokrytia sa počítajú aj iRooty, ktoré boli dosiahnuté v profilovacích fázach. V profilovacích fázach, na rozdiel od aktívnych je možné dosiahnuť viac ako jeden iRoot za

jednu iteráciu, pretože iRooty sa nedosahujú cielene, ale ešte len sa identifikujú sledovaním behu programu. Experiment sa od neupravenej verzie líší aj v celkovom počte nájdenia chyby, preto v niektorých prípadoch nemožno jednoznačne rozhodnúť o zlepšení, či zhoršení.

Z pohľadu počtov odhalení chyby boli výsledky v štyroch programoch totožné, v dvoch prípadoch došlo k zhoršeniu. V dvoch prípadoch však došlo v upravenej verzii k odhaleniu všetkých chýb skôr (circular_list a mysql_169_extract) a dva programy nemajú jednoznačný výsledok (t04 a t06).

Z pohľadu rýchlosti pokrytia ostali dva programy bez zmeny. Vidíme rozdiely hlavne v celkovom počte pokrytých iRootov, až 6 z 10 programov pokrylo väčší počet iRootov v upravenej verzii, jeden z programov (t12) pokryl síce menší počet, ale rýchlejšie.

Záver experimentov Z výsledkov experimentov možno získať predstavu o úspešnosti úprav programu Maple vzhľadom na sledovaný aspekt. Avšak z tabuliek vidieť, že experimenty na niektoré programy mali opačné vplyvy. To svedčí o tom, že dosiahnuť univerzálne platné zlepšenie nie je jednoduché a treba pri ňom zohľadniť viac skutočností. Realizované úpravy však poskytli vodítko k tomu, akým smerom je výhodné sa pri úpravách uberať.

Prog/Exp	**default**	**rc0**	**rc20**	**rc80**	**rc10-90**	**rc90-10**	**ww**
bank	0.795	-5.88%	8.65%	3.92%	2.24%	3.14%	5.70%
circ	0.882	-6.01%	3.03%	8.23%	-4.38%	5.92%	6.02%
log	0.772	2.05%	-3.57%	6.82%	5.39%	1.41%	0.35%
mysql	0.942	-3.11%	7.92%	7.93%	0.59%	9.85%	3.15%
shared	0.753	-19.96%	0%	-9.35%	-16.72%	-13.32%	-19.98%
string	0.715	-0.82%	6.49%	1.21%	0.56%	2.55%	3.21%
t01	0.987	-1.06%	11.40%	9.54%	-1.34%	2.00%	0.10%
t04	1.646	2.65%	-1.13%	4.30%	14.01%	3.89%	10.66%
t06	2.082	-0.70%	0.73%	3.32%	0.65%	0.24%	0.49%
t12	1.189	5.10%	17.72%	-7.70%	-3.32%	5.11%	0.90%
priemer	**-**	**-2.77%**	**5.12%**	**2.82%**	**-0.23%**	**2.07%**	**1.06%**

Tabulka 7.1: Percentuálne zrýchlenie iterácie daného stĺpca oproti stĺpcu `default`. Záporná hodnota značí zrýchlenie, čím vyššia kladná hodnota, tým väčšie spomalenie.

Prog/Exp	default	rc0	rc20	rc80	rc10-90	rc90-10	ww
bank	2.94%	20.06%	2.04%	22.44%	17.00%	13.26%	17.00%
circ	7.14%	16.66%	3.64%	3.64%	-28.86%	-50.99%	-10.65%
log	2.00%	1.00%	100%	88.50%	9.50%	-5.00%	-1.00%
mysql	0.94%	-100%	-100%	-100%	-100%	0%	-5.32%
shared	50.00%	0%	0%	0%	0%	0%	0%
string	4.34%	2.30%	7.14%	4.60%	-11.53%	2.30%	-2.08%
t01	11.76%	59.43%	-50.00%	0%	-50.00%	6.29%	0%
t04	52.94%	2.30%	1.70%	0%	4.93%	0%	1.70%
t06	7.46%	-16.22%	-1.48%	21.84%	-11.94%	3.08%	4.69%
t12	75.00%	0%	0%	0%	0%	0%	0%
priemer	**-**	**-1.44%**	**-3.69%**	**4.10%**	**-17.09%**	**3.10%**	**0.43%**

Tabulka 7.2: Percentuálny nárast iterácii, ktoré vyvolali chybu vzhľadom na referenčný stĺpec `default`. Čím vyššie percento tým lepšie, záporné hodnoty značia pokles pomeru odhalení chyby.

Prog/Exp	default	rc0	rc20	rc80	rc10-90	rc90-10	ww
bank	33.82%	-13.67%	0.02%	-6.78%	-8.25%	-1.45%	-3.17%
circ	26.78%	-6.65%	3.69%	-10.12%	-17.74%	4.81%	-20.58%
log	22.00%	14.77%	27.27%	28.63%	14.86%	-9.10%	-5.50%
mysql	11.32%	50.00%	-0.89%	26.14%	-5.75%	58.30%	2.47%
shared	100%	0%	0%	0%	0%	0%	0%
string	17.39%	15.00%	20.35%	30.64%	21.62%	15.00%	10.06%
t01	29.41%	27.50%	0%	0%	0%	27.50%	0%
t04	35.29%	-29.16%	30.77%	0%	57.41%	0%	-23.72%
t06	16.41%	-8.60%	16.45%	38.45%	-11.83%	3.10%	14.25%
t12	50.00%	0%	0%	0%	0%	0%	0%
priemer	**-**	**4.91%**	**9.76%**	**10.69%**	**5.03%**	**9.81%**	**-2.61%**

Tabulka 7.3: Percentuálny nárast iterácii, v ktorých sa podarilo dosiahnuť iRoot vzhľadom na referenčný stĺpec `default`. Čím vyššie percento tým lepšie, záporné hodnoty značia pokles pomeru úspešných iterácií.

Kapitola 8

Záver

Prínosom celej práce je vytvorenie uceleného popisu programu Maple, ktorý dosiaľ nebol k dispozícii, analýza slabých miest a následnú úpravy s cieľom ich eliminácie.

Štúdium programu nebolo z dôvodu absencie dokumentácie a minumu komentárov v zdrojovom kóde jednoduché. Boli pri ňom použité postupy reverzného inžinierstva a informácie boli získavané jednak zo statickej analýzy zdrojových kódov a jednak z debuggingu, za ktorého účelom bol program prenesený do prostredia Eclipse. Pri debugginku bolo však možné len trasovanie častí behu programu, keďže testovanie je v réžii programu Pin. Ten pri vzniku istých udalosti odovzdá riadeniu programu Maple. Komplikáciou bolo aj to, že v samotnom programe Maple dochádza k prelínaniu kódu v dvoch rôznych jazykoch. Niektoré nejasnosti pri vypracovávaní boli emailom diskutované priamo s autorom programu Maple.

Analýza programu Maple odkryla viaceré slabé miesta programu, ktoré vytvorili možnosti pre zlepšenie. Na základe nich boli navrhnuté možné prístupy k ich odstráneniu. Zadanie práce vyžadovalo pokus o elimináciu jedného z týchto miest, v práci bol realizovaný pokus o elimináciu niekoľkých z nich viacerými spôsobmi. V rámci práce bola vytvorená sada testových programov, na ktorých boli výsledky úprav porovnané.

Ako kľúčový prvkok pri odstraňovaní slabých miest vidíme získanie väčšieho množstva informácii z priebehu testovania. Tento prístup bol použitý aj v tejto práci pri experimentovaní s procesmi rozhodovania počas testovania.

Výsledky boli porovnané z viacerých hľadísk, z niektorých z nich prinášali úpravy skutočne lepšie výsledky. Viedli k vyššej úspešnosti pri dosahovaní identifikovaných správaní testovaného programu, v niektorých prípadoch k vyššiemu pokrytiu a k početnejším alebo rýchlejším nájdeniám chyby. Úpravy sa líšili svojím efektom aj v závislosti na sledovanom programe, čo naznačuje, že pre rôzne typy programov môžu byť vhodné rôzne prístupy.

Rozlíšenie použitia týchto prístupov je jedným z námetov pre budúcu prácu, rovnako aj zapojenie ďalších získaných informácií do rozhodovacích procesov v programe.

Literatura

[1] Pin 2.14 User Guide [online]. 2012-04-04.
URL https://software.intel.com/sites/landingpage/pintool/docs/67254/Pin/html/

[2] Carver, R. H.; Tai, K.-C.: *Introduction to Concurrent Programming*. John Wiley and Sons, Inc., 2005, ISBN 9780471744177, s. 1–45, doi:10.1002/0471744174.ch1.
URL http://dx.doi.org/10.1002/0471744174.ch1

[3] Fiedor, J.; Hrubá, V.; Křena, B.; aj.: Advances in Noise-based Testing of Concurrent Programs. *Software Testing, Verification and Reliability*, ročník 25, č. 3, 2015: s. 272–309, ISSN 1099-1689.
URL http://www.fit.vutbr.cz/research/view_pub.php.cz?id=10275

[4] Fiedor, J.; Křena, B.; Letko, Z.; aj.: A Uniform Classification of Common Concurrency Errors. Technická zpráva, 2010.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9426

[5] Fiedor, J.; Vojnar, T.: Noise-Based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In *PADTAD '12*, Proceedings of the 10th Workshop on Parallel and Distributed Systems, Association for Computing Machinery, 2012, ISBN 978-1-4503-1456-5, s. 36–46.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10068

[6] Letko, Z.: Analysis and Testing of Concurrent Programs. *Information Sciences and Technologies Bulletin of the ACM Slovakia*, ročník 5, č. 3, 2013: s. 1–8, ISSN 1338-1237.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=10494

[7] Luk, C.-K.; Cohn, R.; Muth, R.; aj.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, ročník 40, č. 6, Červen 2005: s. 190–200, ISSN 0362-1340, doi:10.1145/1064978.1065034.
URL http://doi.acm.org/10.1145/1064978.1065034

[8] Meyer, B.; Woodcock, J. (editoři): *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, *Lecture Notes in Computer Science*, ročník 4171, Springer, 2008, ISBN 978-3-540-69147-1, doi:10.1007/978-3-540-69149-5.
URL http://dx.doi.org/10.1007/978-3-540-69149-5

[9] Patil, H.; Pereira, C.; Stallcup, M.; aj.: PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, New York, NY, USA: ACM, 2010, ISBN 978-1-60558-635-9, s. 2–11, doi:10.1145/1772954.1772958.
URL http://doi.acm.org/10.1145/1772954.1772958

[10] Silberschatz, A.; Galvin, P. B.; Gagne, G.: *Operating System Concepts*. 2008, ISBN 0470128720.
URL http://www.amazon.com/Operating-System-Concepts-Abraham-Silberschatz/dp/0470128720

[11] Varda, K.: Protocol Buffers. http://code.google.com/apis/protocolbuffers/.

[12] Yu, J.: *Finding and Tolerating Concurrency Bugs*. Dizertační práce, The University of Michigan, 2013.

[13] Yu, J.; Narayanasamy, S.; Pereira, C.; aj.: Maple: A Coverage-driven Testing Tool for Multithreaded Programs. *SIGPLAN Not.*, ročník 47, č. 10, Říjen 2012: s. 485–502, ISSN 0362-1340, doi:10.1145/2398857.2384651.
URL http://doi.acm.org/10.1145/2398857.2384651

Příloha A

Tabuľky

V tejto prílohe sa nachádzajú tabuľky priemerných hodnôt údajov pre jednotlivé experimenty pre každý program. Význam stĺpcov v tabuľkách je:

1. identifikátor experimentu
2. trvanie jednej aktívnej iterácie
3. smerodatná odchýlku trvania
4. percentuálny podiel behov kedy sa úspešne podarilo dosiahnuť daný iRoot
5. RC = počet volaní funkcie RandomChoice
6. GU = počet volaní funkcie GiveUp
7. počet prechodov do sledovacieho módu
8. počet nastavovaní priorít vlákien

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.7954	0.3415	33.82%	2.94%	0.963	2.368	2.162	13.85
rc0	0.8421	0.344	29.20%	3.53%	0.531	2.584	2.354	13.04
rc20	0.7266	0.3398	33.83%	3.00%	0.624	2.436	2.180	13.00
rc80	0.7641	0.3399	31.53%	3.60%	2.640	2.460	2.216	18.73
rc10-90	0.7775	0.3406	31.03%	3.44%	0.603	2.448	2.172	12.57
rc90-10	0.7704	0.3410	33.33%	3.33%	1.750	2.35	2.117	15.88
ww	0.7500	0.3424	32.75%	3.44%	0.966	2.095	1.828	12.91

Tabulka A.1: Bank_account

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.8820	0.4125	26.78%	7.14%	1.339	7.232	5.214	27.00
rc0	0.9351	0.4125	25.00%	8.33%	1.133	6.467	4.700	23.33
rc20	0.8552	0.4086	27.77%	7.40%	1.370	7.185	5.370	26.20
rc80	0.8094	0.4070	24.07%	7.40%	3.056	7.148	5.556	32.48
rc10-90	0.9207	0.4074	22.03%	5.08%	1.915	7.254	5.627	30.22
rc90-10	0.8298	0.4073	28.07%	3.50%	1.561	7	4.772	26.05
ww	0.8289	0.4095	21.27%	6.38%	1.957	5.362	3.70	22.51

Tabulka A.2: Circular_list

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.7715	0.3664	22.00%	2.00%	1.26	2.46	2.42	14.85
rc0	0.7557	0.3458	25.25%	2.02%	0.56	2.36	2.32	12.15
rc20	0.7991	0.3574	28.00%	4.00%	0.46	3.28	3.16	15.74
rc80	0.7189	0.3460	28.30%	3.77%	3.00	3.057	2.94	22.17
rc10-90	0.7299	0.3437	25.27%	2.19%	0.46	2.56	2.51	13.12
rc90-10	0.7606	0.3409	20.00%	1.90%	1.50	2.37	2.27	14.83
ww	0.7688	0.3423	20.79%	1.98%	0.81	2.39	2.38	13.29

Tabulka A.3: Log_proc_sweep

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.9421	0.400	11.32%	0.94%	1.37	2.5	2.42	14.52
rc0	0.9714	0.381	16.98%	0%	0.53	3.17	3.06	14.45
rc20	0.8675	0.372	11.22%	0%	0.70	2.70	2.69	13.45
rc80	0.8674	0.374	14.28%	0%	1.38	3.43	3.10	18.24
rc10-90	0.9365	0.381	10.67%	0%	1.49	2.61	2.64	15.58
rc90-10	0.8493	0.374	17.92%	0.94%	1.30	2.11	1.98	13.34
ww	0.9124	0.377	11.60%	0.89%	0.88	2.29	2.13	12.59

Tabulka A.4: mysql_169_extract

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.7531	0.274	100.00%	50.00%	2	7.5	8	34.25
rc0	0.9034	0.330	100.00%	50.00%	0.5	7.5	8	29.75
rc20	0.7530	0.317	100.00%	50.00%	0.75	7.5	8	30.50
rc80	0.8235	0.326	100.00%	50.00%	4.75	7.5	8	42.50
rc10-90	0.8790	0.332	100.00%	50.00%	0.5	7.5	8	29.75
rc90-10	0.8534	0.342	100.00%	50.00%	1.5	7.5	8	32.75
ww	0.9035	0.334	100.00%	50.00%	1	7.5	8	31.25

Tabulka A.5: shared_counter

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.7156	0.316	17.39%	4.34%	0.83	2.63	2.39	12.46
rc0	0.7215	0.289	20.00%	4.44%	0.36	2.69	2.49	11.78
rc20	0.6691	0.281	20.93%	4.65%	0.86	3.00	2.74	13.33
rc80	0.7069	0.284	22.72%	4.54%	3.09	2.73	2.32	19.34
rc10-90	0.7115	0.286	21.15%	3.84%	1.12	2.65	2.54	14.04
rc90-10	0.6973	0.286	20.00%	4.44%	0.96	2.89	2.67	14.20
ww	0.6926	0.286	19.14%	4.25%	0.87	2.87	2.68	14.11

Tabulka A.6: string_buffer

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.9869	0.406	29.41%	11.76%	0	8.53	4.24	26.53
rc0	0.9973	0.369	37.50%	18.75%	0	7.00	4.25	23.81
rc20	0.8743	0.348	29.41%	5.88%	0	8.53	4.24	26.53
rc80	0.8927	0.356	29.41%	11.76%	0	8.53	4.24	26.53
rc10-90	1.0001	0.368	29.41%	5.88%	0	8.53	4.24	26.53
rc90-10	0.9672	0.363	37.50%	12.50%	0.44	6.94	4.25	24.44
ww	0.9858	0.365	29.41%	11.76%	0	8.53	4.24	26.53

Tabulka A.7: t01

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	1.6459	0.969	35.29%	52.94%	0.53	4.82	5.76	23.82
rc0	1.6022	0.909	25.00%	54.16%	1.17	3.42	3.17	14.67
rc20	1.6646	0.913	46.15%	53.84%	0.23	5.38	6.46	24.62
rc80	1.5750	0.900	35.29%	52.94%	0.65	4.82	5.76	23.76
rc10-90	1.4151	0.891	55.55%	55.55%	0.44	6.00	6.44	25.22
rc90-10	1.5818	0.888	35.29%	52.94%	0.65	4.82	5.76	23.41
ww	1.4703	0.872	26.92%	53.84%	1.15	3.31	4.15	16.73

Tabulka A.8: t04

Experiment	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	2.0817	0.461	16.41%	7.46%	0.76	7.75	6.72	31.39
rc0	2.0964	0.466	15.00%	6.25%	0.38	7.93	7.10	31.09
rc20	2.0665	0.467	19.11%	7.35%	0.49	8.25	7.24	32.09
rc80	2.0126	0.469	22.72%	9.09%	1.70	7.95	7.03	35.06
rc10-90	2.0680	0.466	14.47%	6.57%	0.53	7.91	6.97	31.09
rc90-10	2.0766	0.464	16.92%	7.69%	0.98	8.40	7.26	33.55
ww	2.0714	0.465	18.75%	7.81%	0.58	8.38	7.31	32.97

Tabulka A.9: t06

Exp	Čas [s]	σ času	Úspech	Chyba	RC	GU	Watch	Priority
default	0.8533	0.224	50.00%	75.00%	0	1.5	2	9
rc0	1.1287	0.462	50.00%	75.00%	0	8	9	30
rc20	0.9786	0.469	50.00%	75.00%	0	8.25	9	30.5
rc80	1.2810	0.486	50.00%	75.00%	0	8	9	30
rc10-90	1.2289	0.475	50.00%	75.00%	0	8.5	9	31
rc90-10	1.1286	0.465	50.00%	75.00%	0	8	9	30
ww	1.1786	0.463	50.00%	75.00%	0	8.25	9	30.5

Tabulka A.10: t12

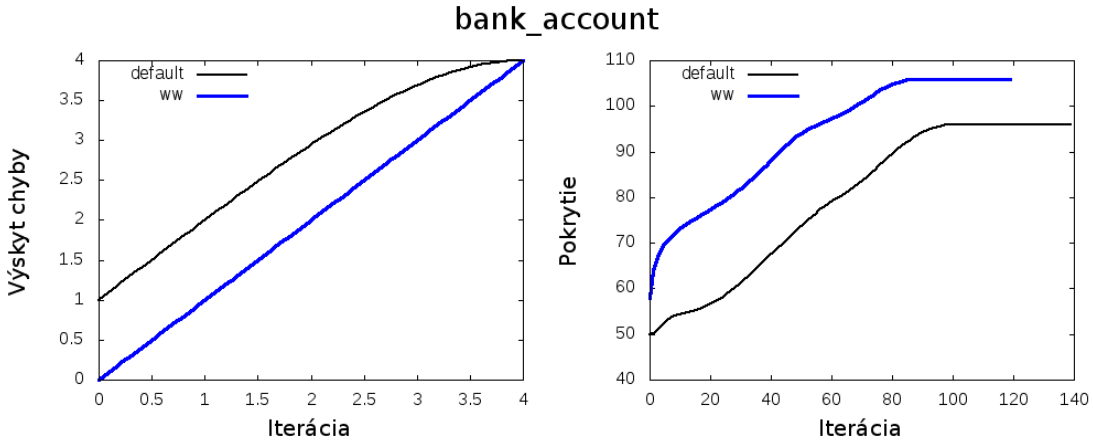
Příloha B

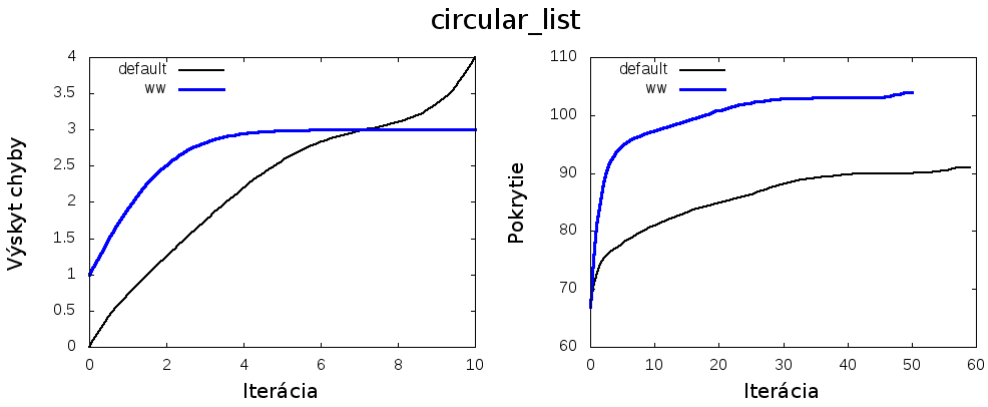
Grafy

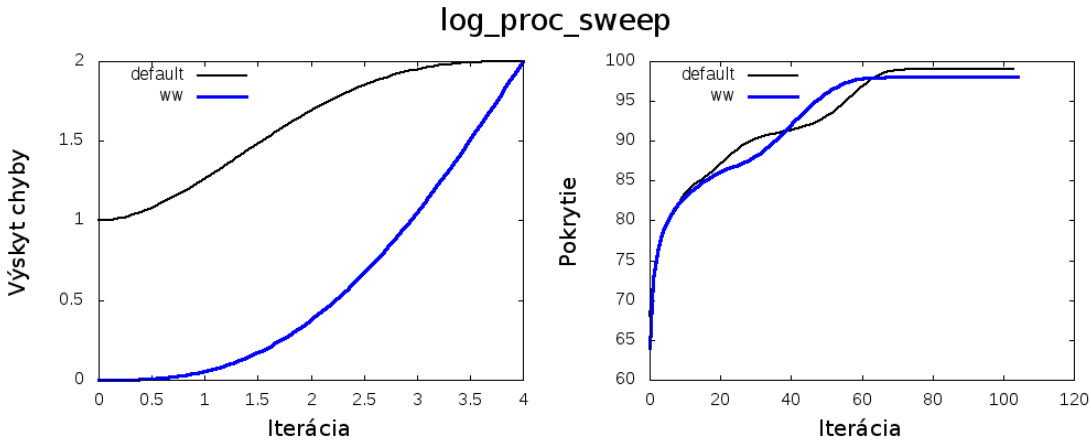
Grafy v tejto kapitole ukazujú vzťah pôvodnej verzie programu Maple a jeho úpravy s označením `ww` t.j. uprednostnenie iRootov typu zápis – zápis.

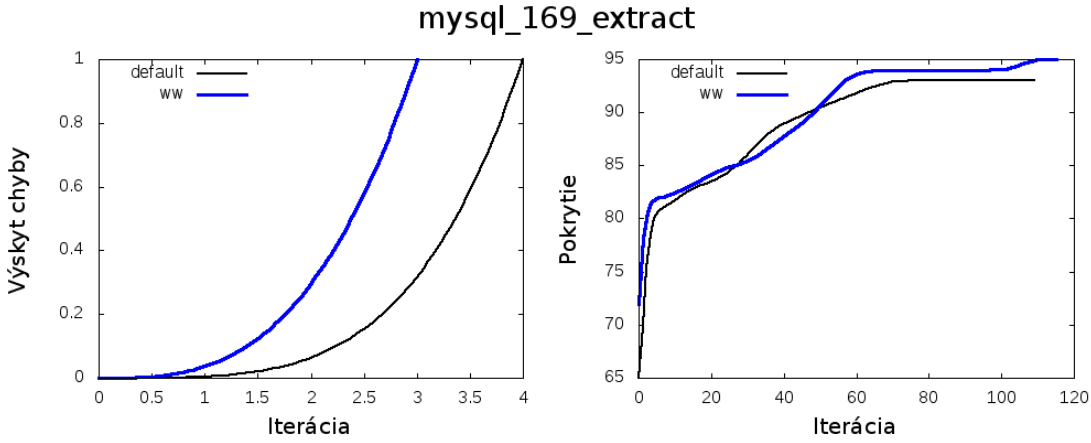
Grafy vľavo zobrazujú počet odhalení chyby od začiatku testovania. Na ose `x` sú poradové čísla iterácií a na ose `y` je počet odhalených chýb.

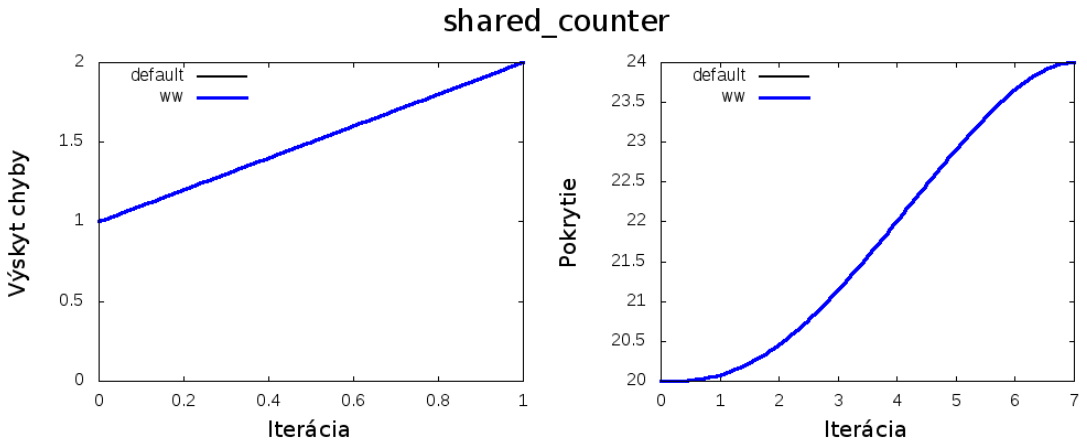
Grafy vpravi zobrazujú počet pokrytých iRootov od začiatku testovania. Na ose `x` sú poradové čísla iterácií a na ose `y` je počet pokrytých iRootov.

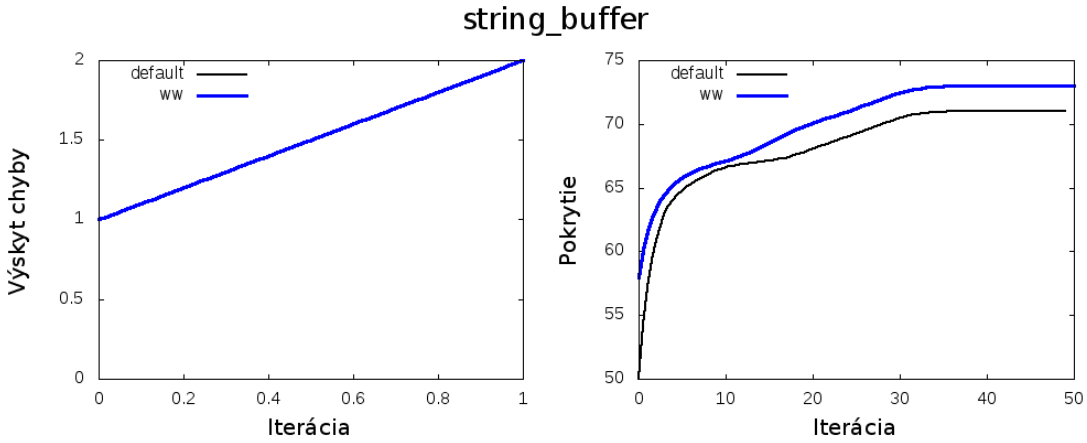


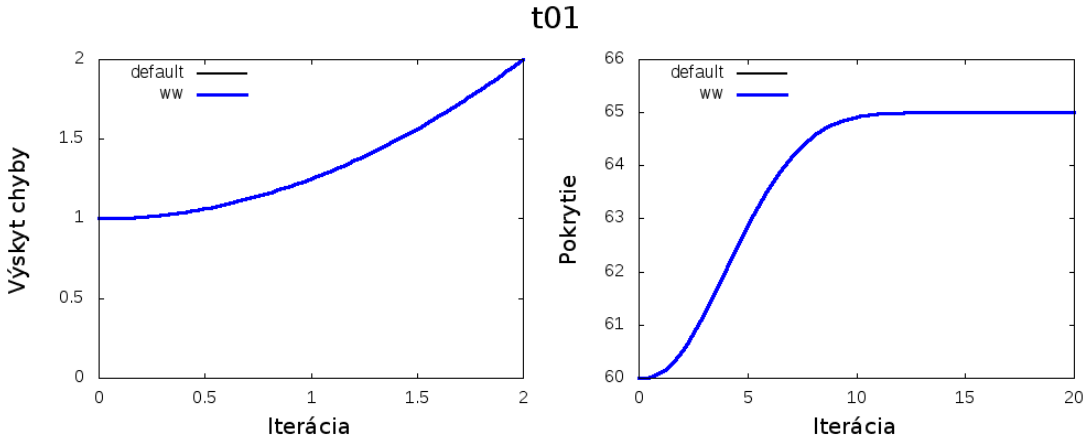

circular_list
default
ww
Výskyt chyby
Iterácia
Pokrytie
Iterácia


log_proc_sweep
default
ww
Výskyt chyby
Iterácia
Pokrytie
Iterácia

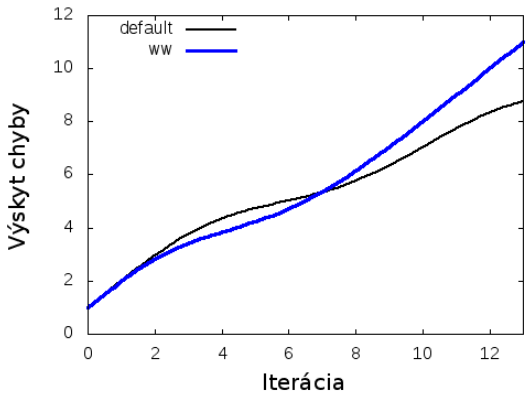

mysql_169_extract
default
ww
Výskyt chyby
Iterácia
Pokrytie
Iterácia

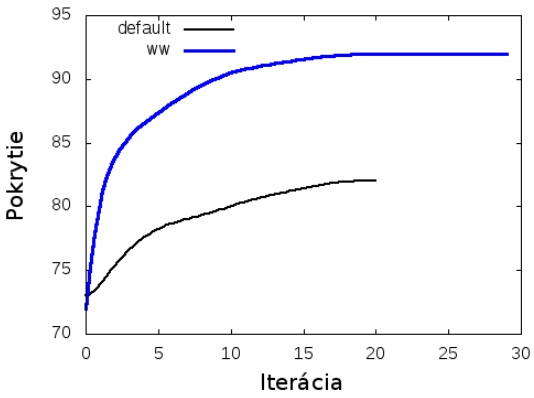

shared_counter
default
ww
Výskyt chyby
Iterácia
Pokrytie
Iterácia


string_buffer
default
ww
Výskyt chyby
Iterácia
Pokrytie
Iterácia

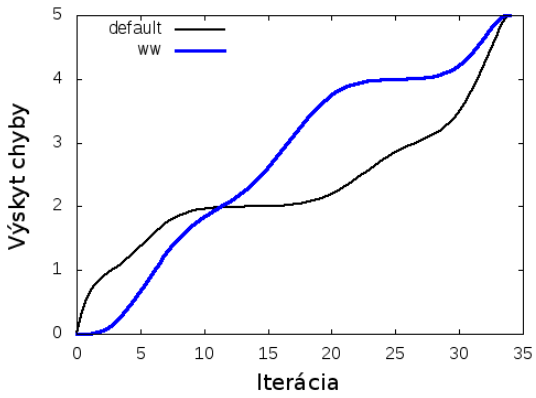

t01
default
ww
Výskyt chyby
Iterácia
Pokrytie
Iterácia

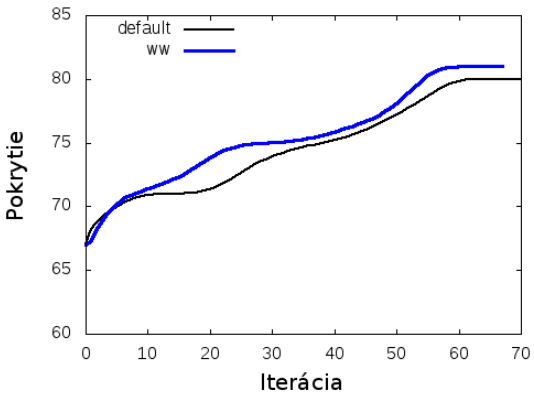
t04


default
ww
Výskyt chyby
Iterácia
0
2
4
6
8
10
12

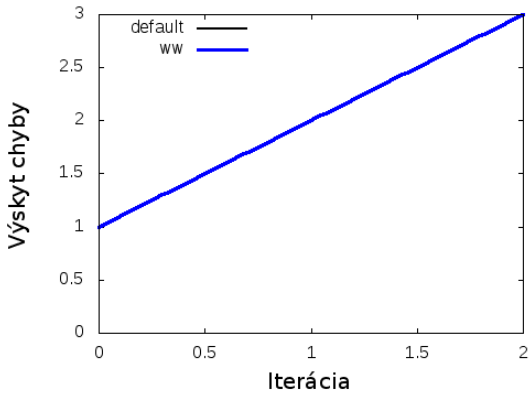

default
ww
Pokrytie
Iterácia
70
75
80
85
90
95
0
5
10
15
20
25
30

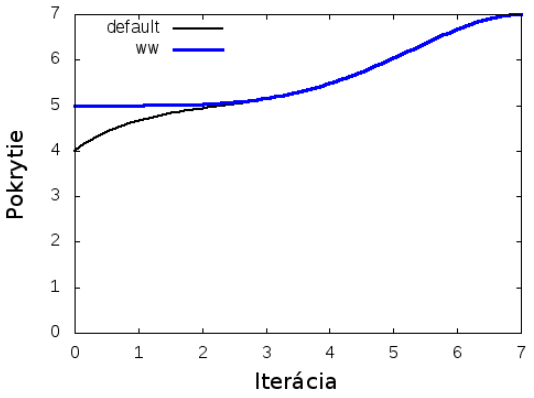
t06


default
ww
Výskyt chyby
Iterácia
0
1
2
3
4
5
0
5
10
15
20
25
30
35


default
ww
Pokrytie
Iterácia
60
65
70
75
80
85
0
10
20
30
40
50
60
70

t12


default
ww
Výskyt chyby
Iterácia
0
0.5
1
1.5
2
2.5
3
0
0.5
1
1.5
2


default
ww
Pokrytie
Iterácia
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7

Příloha C

Obsah CD

CD priložené k práci obsahuje:

- návod k stiahnutiu a inštalácii potrebných programov
- súbory obsahujúce modifikácie a skripty pre spracovanie dát
- návod k zopakovaniu a spracovaniu experimentov
- plagát reprezentujúci výsledky práce