Tehnike dinamičke analize za otkrivanje grešaka u strukturama podataka bez zaključavanja

Marinković, Gabriel

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 618

TEHNIKE DINAMIČKE ANALIZE ZA OTKRIVANJE GREŠAKA U STRUKTURAMA PODATAKA BEZ ZAKLJUČAVANJA

Gabriel Marinković

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 618

TEHNIKE DINAMIČKE ANALIZE ZA OTKRIVANJE GREŠAKA U STRUKTURAMA PODATAKA BEZ ZAKLJUČAVANJA

Gabriel Marinković

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 618

Pristupnik:	**Gabriel Marinković (0036522675)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	izv. prof. dr. sc. Ante Đerek

Zadatak: **Tehnike dinamičke analize za otkrivanje grešaka u strukturama podataka bez zaključavanja**

Opis zadatka:

Strukture podataka bez zaključavanja (lock-free data structure) su posebna vrsta konkurentnih podatkovnih struktura koje omogućavaju sigurno pristupanje s više dretvi bez potrebe za zaključavanjem. U sklopu diplomskog rada potrebno je istražiti postojeće metode i alate za otkrivanje grešaka nadmetanja (race conditions) u algoritmima i strukturama podataka. Na temelju istraživanja potrebno je osmisliti i ostvariti novu ili prilagoditi postojeću takvu metodu u svrhu analize implementacija struktura podataka bez zaključavanja. Razvijenu metodu je potrebno vrednovati ručno izrađenim testovima s poznatim greškama. Ako je moguće, potrebno je i provesti analizu javno dostupnih biblioteka. Radu je potrebno priložiti izvorni kod razvijenih i korištenih programa, citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

1. Uvod

Testiranje softvera izuzetno je važno kako bi se osigurala njegova ispravnost i pouzdanost. Testiranje je posebno bitno u kontekstu paralelnog i konkurentnog (eng. *concurrent*) programiranja. Moderni softver se sve više oslanja na višedretvenost kako bi postigao bolje performanse, zbog čega složenost upravljanja dijeljenim resursima između dretvi postaje sve složenija. Upravljanje resursima najčešće podrazumijeva korištenje sinkronizacijskih primitiva koje implementira operacijski sustav kao što su kritički odsječci (eng. *lock*, *mutex*), semafori i slično. Moderni jezici rijetko imaju razrađene mehanizme za sprječavanje pogreški koje proizlaze iz njihova korištenja, kao što su *deadlock*, inverzija prioriteta i slično [1].

Klasa *lock-free* algoritama se razvila kao potencijalno rješenje za ove probleme. To su algoritmi koji garantiraju napredak programa bez mogućnosti *deadlock*-a. Namijenjeni su da se koriste u kontekstima gdje su pogreške kao *deadlock*-ing nedopustive (*realtime* sustavi) ili gdje postoje strogi zahtjevi za visoke i predvidive performanse (*driver*-i komponenti, jezgra OS-a). *Lock-free* algoritmi su znatno složeniji za ispravno implementirati, verificirati i testirati od uobičajenog višedretvenog koda. Optimizacije tijekom prevođenja, vremenski rasporedi dretvi i stanje procesora svi mogu utjecati na pojavljivanje greški koje se mogu pojaviti vrlo rijetko i koje mogu biti vrlo teške za reproducirati i otkloniti. Standardne metode testiranja ne pomažu u pronalasku suptilnih konkurentnih problema zbog nedeterminističkih uzroka ovakvih pogreški.

Ovaj rad predstavlja novi sustav za testiranje *lock-free* algoritama i struktura podataka koji rješava neke od ovih problema. Naš sustav koristi dinamičku instrumentaciju strojnog koda kako bi dokučio koji sve rasporedi dretvi potencijalno mogu uzrokovati drugačije ponašanje programa, te zatim sistematski isproba sve moguće kombinacije vremenskih rasporeda dretvi kako bi pronašao *execution history* koji dovodi do greške. Korisnik

zatim može ponovno pokrenuti program s istim rasporedom dretvi kako bi reproducirao i lakše otklonio pogrešku. Zbog korištenja dinamičke instrumentacije umjesto standardnije statičke analize izvornog koda, sustav može pronaći greške koje su uzrokovane optimizacijama tijekom prevođenja. Uz to moguće je testirati kod bez pristupa njegovom izvornom kodu (ako je distribuiran samo kao biblioteka) i trivijalno prilagoditi sustav nekom jeziku osim C-a i C++-a koji se prevodi u strojni kod.

Ostatak rada će dati kratki uvod u teoriju *lock-free* algoritama, primjer jednostavne *lock-free* strukture i nedeterminističke greške u njoj. Slijedi pregled postojećih sustava za testiranje koji pokušavaju riješiti slične probleme i njihova usporedba s našim sustavom. Nakon toga slijedi detaljan opis implementacije sustava za testiranje i primjer testa koji testira strukturu predstavljenu u uvodu. Na kraju slijede rezultati testiranja raznih javno dostupnih implementacija *lock-free* struktura podataka i rasprava o nedostacima i budućim unaprjeđenjima sustava.

2. *Lock-free* algoritmi i strukture podataka

2.1. Klase algoritama

Lock-free ime ima dva široko korištena značenja. Službeno se odnosi na pod-klasu *block-free* algoritama, a neslužbeno se često koristi kao opći pojam za sve algoritme koji ne koriste *lock*-ove. Algoritmi koji ne koriste *lock*-ove dijele se na algoritme bez blokiranja (eng. *block-free*) i algoritme bez međusobnog nadmetanja (eng. *obstruction-free*). Sve navedene klase se međusobno razlikuju po teoretskim garancijama o mogućnosti *napredovanja* programa. Algoritmi koji koriste *lock*-ove nemaju nikakve garancije o napredovanju – ako jedna dretva uzme *lock* te ju zatim operacijski sustav suspendira, ne postoji garancija kada (ni ako) će ju operacijski sustav ponovno pokrenuti. Za to vrijeme ostale dretve moraju čekati da *lock* ponovno postane dostupan. U ostatku rada ćemo se koristiti neslužbenim značenjem riječi *lock-free* i pod tim pojmom ćemo podrazumijevati algoritme koji s više dretva pristupaju dijeljenoj memoriji a ne koriste sinkronizacijske primitive koje pruža OS. Najčešće će koristiti hardversku podršku za atomarne operacije.

U algoritme bez blokiranja ubrajamo algoritme bez zaključavanja (eng. *lock-free*) i bez čekanja (eng. *wait-free*) [2]. *Wait-free* algoritmi garantiraju da će svaka dretva uvijek moći napredovati s izvršavanjem algoritma u konačno mnogo koraka (tj. ako je potrebno sinkronizirati memoriju između dretvi, sinkronizacija je moguća u konačno mnogo koraka), bez obzira na stanja u kojima se nalaze druge dretve, i bez obzira na to jesu li suspendirane ili ne. *Lock-free* algoritmi imaju slabiji uvjet te garantiraju da će u svakom trenu barem jedna dretva moći napredovati s izvršavanjem algoritma u konačno mnogo koraka.

Isječak 2.1. na pojednostavljenom primjeru uspoređuje *lock-free* i *wait-free* algoritme. Sljedeće funkcije inkrementiraju varijablu. Više dretvi poziva funkcije istovremeno. Ako dvije dretve istodobno pokušaju pozvati `increment_lock_free()` postoji šansa da će jedna od njih morati čekati u petlji dok druga ne završi s vlastitim pozivom. Sve dretve će uvijek završiti s izvođenjem `increment_wait_free()` u jednom koraku, zbog hardverske podrške za atomarno zbrajanje.

Isječak 2.1.: Usporedba *lock-free* i *wait-free* algoritma.

```
#include <atomic>

std::atomic<int> counter(0);

void increment_lock_free() {
  int old_value;
  do {
    old_value = counter.load();
  } while (!counter.compare_exchange_weak(old_value, old_value + 1));
}

void increment_wait_free() {
  counter.fetch_add(1);
}
```

Algoritmi bez međusobnog ometanja, odnosno *obstruction-free* algoritmi imaju još slabiju garanciju o mogućnosti napredovanja dretve. *Obstruction-free* algoritmi garantiraju napredak dretve samo u slučaju da ne postoji nadmetanje (eng. *obstruction*) nad dijeljenim resursom u trenu kada ga dretva pokušava koristiti [3]. To se može desiti kada npr. druge dretve izvršavaju neki drugi dio algoritma ili ih je operacijski sustav suspendirao. U izvedbama ovih algoritama često postoje mehanizmi odmicanja (eng. *back-off*) kako bi se smanjilo nadmetanje za određeni resurs.

Unatoč značajno slabijim garancijama o teoretskoj mogućnosti napredovanja dretvi *obstruction-free* i *lock-free* algoritmi se često u praksi preferiraju nad *wait-free* algoritmima jer su značajno jednostavniji za ostvarivanje i često imaju bolje performanse, a i dalje imaju „dovoljno dobra” svojstva o mogućnostima napredovanja izvršavanja. Ova razlika u garancijama je još manja kod sustava koji ne moraju raditi u stvarnom vremenu, gdje će nadmetanje nad resursima biti ublaženo dok su neke dretve suspendirane.

2.2. Primjer *wait-free* strukture

2.2.1. SPSC FIFO red

Detaljnije ćemo analizirati jednu *wait-free* strukturu podataka: FIFO red s jednim proizvođačem i jednim potrošačem (eng. *single-producer single-consumer queue*, odnosno *SPSC queue*). Implementacija red nalazi se u isječku 2.2. Red je implementiran po uzoru na Lamportov kružni niz [4]. Samo jedna dretva može ubacivati podatke u red te samo jedna dretva može čitati podatke iz njega. Ostale dretve ne smiju pristupati redu, niti se dretva proizvođač smije ponašati kao potrošač (i obrnuto). Obje dretve uvijek mogu napredovati u izvođenju algoritma. Dretva proizvođač će uvijek uspjeti ubaciti vrijednost u red ako u njemu ima slobodnog mjesta, bez čekanja. Analogno vrijedi i za dretvu potrošač.

Red koristi dva indeksa za održavanje konzistencije strukture, jedan za čitanje (`r_`) i jedan za pisanje (`w_`). Red ne ovisi ni o kakvim posebnim atomarnim operacijama; indeksi su deklarirani kao atomarne vrijednosti kako bi bilo moguće precizno definirati redoslijed pristupa memoriji (eng. *memory ordering*) za vrijeme pisanja i čitanja. Red koristi jedan element polja kao oznaku za puni ili prazni red; ako su oba indeksa isti red je prazan i nije moguće pročitati vrijednost iz njega dok se indeks za pisanje ne pomakne ispred indeksa za čitanje.

Proizvođač dodaje elemente u red tako da pročita trenutne pozicije indeksa za pisanje i čitanje. Ako je sljedeći indeks za pisanje veći od veličine unutarnjeg polja, indeks se vraća na početak polja. Ako je sljedeći indeks za pisanje jednak trenutnom indeksu za čitanje, red je pun i nije moguće pisati u njega sve dok potrošač nešto ne pročita iz njega. Budući da samo jedna dretva mijenja vrijednosti `w_` indeksa, moguće ga je čitati koristeći `memory_order_relaxed`. Ažuriranje vrijednosti `w_` potrebno je izvršiti koristeći `memory_order_release` kako bi druga dretva mogla vidjeti prethodnu promjenu na `data_` polju, te kako prevoditelj ne bi zamijenio redoslijed pisanja u `data_` i `w_`. Kod za čitanje elemenata reda vrlo je sličan (metoda `Pop()`) te samo zamjenjuje sve operacije pisanja s operacijama čitanja.

Isječak 2.2.: *Wait-free* FIFO red.

```
#include <atomic>

template <typename T, size_t size>
class Queue {
 public:
  bool Push(const T &element) {
    const size_t w = w_.load(std::memory_order_relaxed);
    const size_t r = r_.load(std::memory_order_acquire);

    const size_t w_next = (w + 1) % size;
    if (w_next == r) return false;

    data_[w] = element;
    w_.store(w_next, std::memory_order_release);
    return true;
  }

  bool Pop(T *element) {
    const size_t r = r_.load(std::memory_order_relaxed);
    const size_t w = w_.load(std::memory_order_acquire);

    if (r == w) return false;

    *element = data_[r];

    const size_t r_next = (r + 1) % size;
    r_.store(r_next, std::memory_order_release);
    return true;
  }

 private:
  std::atomic<size_t> r_;
  std::atomic<size_t> w_;
  T data_[size];
};
```

2.2.2. Primjer greške u *lock-free* algoritmima

Pristup i sinkronizacija dijeljene memorije bez korištenja kritičkih odsječaka je složen problem koji komplicira implementaciju *lock-free* algoritama te jako otežava provjeru njene ispravnosti. Male promjene u kodu mogu izazvati greške koje će se pojaviti samo za određene vremenske rasporede dretvi. Osim toga neke greške se mogu pojaviti kada se koristi jedan prevoditelj kojih nema kada se koristi neki drugi prevoditelj (ili druga verzija istog). Prevoditelju je dozvoljeno da radi proizvoljne optimizacije, kao na primjer zamjena memorijskih operacija, sve dok te zamjene imaju isto semantičko značenje prema standardu jezika [5]. Moguće je da zbog određenih heuristika u koraku optimizacije jedan prevoditelj „slučajno” prevede neispravan izvorni kod u ispravan strojni kod, dok će drugi prevoditelj generirati očekivani neispravni strojni kod. Do sličnih grešaka može doći i kada se uvijek koristi isti prevoditelj, ali okolni kontekst izvornog koda oko greške uzrokuje različitu generaciju strojnog koda.

Isječak 2.3.: Primjer moguće greške u implementaciji FIFO reda.

```
1  bool Push(const T &element) {
2    const size_t w = w_.load(std::memory_order_relaxed);
3    const size_t r = r_.load(std::memory_order_acquire);
4
5    const size_t w_next = (w + 1) % size;
6    if (w_next == r) return false;
7
8    data_[w] = element;
9    // Change ‘std::memory_order_release‘ into ‘std::memory_order_relaxed‘.
10   w_.store(w_next, std::memory_order_relaxed);
11   return true;
12 }
```

Primjer 2.3. takve greške moguće je konstruirati u metodi `Push()` s promjenom redoslijeda pristupa memoriji pri zapisivanju novog *write* indeksa nakon upisivanja vrijednosti u polje. Promjena iz `memory_order_release` u `memory_order_relaxed` omogućava prevoditelju da zapisivanje u `w_` prebaci prije zapisivanja u `data_`, što omogućava potrošaču da pročita vrijednost s lokacije `w_next` prije nego što je u nju zapisana vrijednost. Važno je napomenuti da do ove greške može, ali i ne mora, doći ovisno o inačici prevoditelja i/ili okolnog konteksta u izvornom kodu. U slučaju da prevoditelj ipak generira „ispravnu" inačicu koda, procesor može zamijeniti redoslijed izvršavanja ove dvije operacije i opet izazvati grešku. To je moguće samo na arhitekturama s relaksiranim memorijskim modelom, poput `AArch64` (`Arm64`), što znači da greška može biti prisutna samo na nekim procesorskim arhitekturama. U svakom od ovih navedenih slučajeva grešku nećemo nužno zamijetiti izvođenjem jednostavnih *brute-force* testova jer njeno pojavljivanje ovisi o *contention*-u nad redom i vremenskom rasporedu dretvi kojeg kontrolira operacijski sustav, a koji se iz naše perspektive ponaša nasumično.

Na ovom jednostavnom primjeru moguće je primijetiti kako male greške u implementaciji mogu uzrokovati greške koje je teško identificirati i reproducirati. Greške se mogu pojavljivati ili nestajati ovisno o inačici prevoditelja, arhitekturi procesora, rasporedu dretvi kojeg kontrolira operacijski sustav, zauzeću ostatka sustava i slično. Budući da se *lock-free* algoritmi i strukture podataka koriste u okruženjima s najstrožim zahtjevima za performanse i predvidljivost, potrebno ih je testirati posebnim tehnikama.

3. Dinamička instrumentacija strojnog koda

Dinamička instrumentacija strojnog koda je proces ubacivanja koda u program za vrijeme njegovog izvršavanja. Ubačeni kod mora biti transparentan instrumentiranom programu. Program bi se trebao izvršavati isto, vršila se na njemu instrumentacija ili ne. To u praksi generalno nije moguće postić – na primjer, program može mjeriti vlastito vrijeme izvođenja, koje će trajati dulje zbog izvođenja instrumentacijskog koda. No istovremeno instrumentacijski kod ne bi smio modificirati unutarnje stanje programa, suspendirati njegove dretve, uzimati kontrolu nad objektima koje je operacijski sustav dodijelio programu (cjevovod za standardni ulaz i izlaz itd.) [6]. Instrumentacija se stoga najčešće koristi za prikupljanje podataka o programu dok se on izvodi. Neki od najpopularnijih alata za dinamičku instrumentaciju su *Valgrind* [7], *ASan* i *UBSan*, koji pokušavaju pronaći greške za vrijeme izvršavanja programa.

3.1. DynamoRIO

Ubacivanje koda u program koji se izvršava, bez da se naruši izvršavanje originalnog programa je vrlo složen zadatak za koji se često koriste radni okviri za instrumentaciju kao što su Intel PinTool i DynamoRIO [6]. Radni okviri omogućavaju jednostavno procesiranje događaja unutar programa, kao što su izvođenje određene instrukcije, stvaranje i uništavanje procesa, početak i kraj sistemskih poziva, stvaranje novih dretvi itd. Osim toga moguće je pregledati sve instrukcije koje će se neposredno izvršiti te ubaciti vlastite instrukcije između njih. Isječak 3.1. prikazuje primjer korištenja DynamoRIO radnog okvira. Isječak koda iterira po svim instrukcijama koje pristupaju memoriji te dodaje proizvoljnu instrumentaciju svakoj takvoj instrukciji.

Isječak 3.1.: Primjer koda za instrumentaciju u radnom okviru DynamoRIO.

```
instr_t* instr_fetch = drmgr_orig_app_instr_for_fetch(drcontext);
if (instr_fetch) {
  if (instr_reads_memory(instr_fetch)  instr_writes_memory(instr_fetch)) {
    InstrumentInstruction(drcontext, bb, where, instr_fetch);
  }
}

instr_t* instr_operands = drmgr_orig_app_instr_for_operands(drcontext);
if (instr_operands) {
  if (instr_reads_memory(instr_operands)  instr_writes_memory(instr_operands)) {
  for (int i = 0; i < instr_num_srcs(instr_operands); i++) {
    const opnd_t src = instr_get_src(instr_operands, i);
    if (opnd_is_memory_reference(src)) {
      InstrumentMemoryReference(drcontext, bb, where, src, false);
    }
  }

  for (int i = 0; i < instr_num_dsts(instr_operands); i++) {
    const opnd_t dst = instr_get_dst(instr_operands, i);
    if (opnd_is_memory_reference(dst)) {
      InstrumentMemoryReference(drcontext, bb, where, dst, true);
    }
  }
}
```

3.2. Usporedba sa statičkom analizom

U nekim slučajevima je umjesto instrumentacije moguće koristiti statičku analizu koda. Statičkom analizom moguće je detaljno analizirati strukturu i dokazati specifična svojstva o programu. Instrumentacijom je moguće provesti dinamičku analizu koda koji se trenutno izvršava i općenito pruža manje detalja o strukturi koda koji su potrebni za kvalitetnu analizu. S druge strane dinamička analiza omogućava znatno jednostavnije analiziranje slijeda operacija koje se izvršavaju i uglavnom ne zahtijeva pristup izvornom kodu, no podložnija je *false-negative* rezultatima. Instrumentira se samo kod koji se izvodi, što znači da neki dijelovi koda mogu biti preskočeni za analizu.

4. Povezan rad

Postoje brojni alati koji pokušavaju olakšati pisanje *lock-free* algoritama i ostalih konkurentnih i distribuiranih sustava. Alati postoje na svim razinama apstrakcije, od modeliranja specifikacije nekog sustava do testiranja dijelova njihove implementacije za određenu procesorsku arhitekturu.

4.1. TLA+

TLA+ (eng. *Temporal Logic of Actions*) je jezik za formalnu specifikaciju koji se uglavnom koristi za modeliranje i provjeravanje svojstava distribuiranih i konkurentnih sustava [8]. Sustav se modelira kao skup stanja i tranzicije između njih. Svako stanje predstavlja cijeli sustav u nekom trenutku, tranzicije predstavljaju događaje ili radnje. Nakon definicije sustava moguće je iskazati tvrdnje o ispravnosti sustava (možemo biti sigurni da su određene situacije nemoguće) i o *liveness*-u (možemo biti sigurni da će sustav u nekom konačnom vremenu doći do željenog stanja). TLA+ se koristi za modeliranje cijelog sustava prije njegove implementacije, kako bi se pronašle moguće greške u njegovom dizajnu.

TLA+ je izuzetno koristan alat sa širokom primjenom u industriji, no zahtijeva poznavanje formalnog modeliranja sustava i vještinu pretvaranja formalnog modela u stvarnu implementaciju. Budući da TLA+ verificira dizajn, no ne i implementaciju sustava, moguće je napraviti implementacijske greške zbog kojih će se sustav ponašati neispravno.

4.2. SPIN

SPIN je alat za formalnu verifikaciju višedretvenih programa i konkurentnih sustava [9]. SPIN koristi jezik za modeliranje sustava *Promela* (*Process Meta Language*) kojim se opisuju specifikacije i ponašanje sustava. Simulacijom je moguće efikasno provjeriti sva moguća stanja sustava te tako utvrditi odgovara li sustav specifikaciji ili u njemu postoji greška. Korisnik mora izraziti željena svojstva sustava koristeći linearnu temporalnu logiku.

Simulacijom modela provjerava se što veći mogući broj programskih stanja (idealno sva moguća stanja). Ovakvom provjerom može se potvrditi da ne postoje greške u modelu, ili ako postoje, korisniku se može prijaviti lista koraka koja je dovela da pogrešnog stanja. SPIN koristi nekoliko tehnika koje ubrzavaju pretraživanje prostora stanja. Jedna od takvih tehnika je *partial order reduction* koja se oslanja na komutativnost nezavisnih operacija u višedretvenim sustavima. Ako dvije dretve ne pristupaju istoj memorijskoj lokaciji tada njihove operacije ne mogu međusobno utjecati jedna na drugu. U velikom broju slučajeva ove nezavisnosti je moguće statički odrediti što može značajno smanjiti broj stanja koji je potrebno provjeriti.

SPIN također ima široku primjenu u industriji pri verifikaciji svojstava algoritama za rad u stvarnom vremenu. Jezik za modeliranje sustava bliži je proceduralnim jezicima kao C te podržava alate za polu-automatsku generaciju modela iz izvornog koda, što alat čini pristupačnijim. Budući da prevođenje algoritma iz izvornog koda u definiciju sustava nije jedan naprema jedan, i dalje je moguće napraviti greške kojih nema u simulaciji, ali se pojavljuju u stvarnoj implementaciji.

4.3. Relacy

Relacy je biblioteka i simulator za detektiranje *race condition*-a u višedretvenim C++ programima [10]. Ovaj alat najviše je inspirirao naš sustav za testiranje. Korisnik definira svoj program slično kao da piše standardni C++, osim što mora označiti pristupe dijeljenoj memoriji koji će simulator koristiti za sinkronizaciju dretvi. Simulator će serijalizirati izvođenje programa - uvijek će izvoditi samo jednu dretvu u svakom trenutku, dok će ostale dretve čekati. Na označenim sinkronizacijskim točkama može se desiti promjena aktivne dretve. Ako simulacija pronađe grešku ispisuje se povijest izvođenja sa redoslijedom dretvi koji je izazvao grešku.

Relacy-eva dinamička analiza tijekom izvođenja programa pamti sve označene sinkronizacijske točke kako bi simulator mogao isprobati zamjenu dretvi u budućim iteracijama. Budući da ne radi statičku analizu ima slabije garancije korektnosti, ali zbog svoje implementacije kao biblioteka za C++ vrlo je pristupačan velikom broju korisnika. Najveći nedostatak alata je potreba za ručnim označivanjem sinkronizacijskih točaka. Takvih točaka je uglavnom malo no potreba za označivanjem izvornog koda onemogućava testiranje dijeljenih biblioteka (kao što je npr. standardna C i C++ biblioteka), a testiranje drugih jezika zahtijeva reimplementaciju velikog dijela sustava. Ekspanzija koja se događa oko `$()` oznaka ubacuje velik dio koda koji može onemogućiti prevoditelju da napravi optimizacije koje bi inače napravio, čime je moguće da simulacija ne pronađe greške koje se javljaju samo u slučaju optimizacija.

Isječak 4.1.: Primjer Relacy testa sa greškom gdje dvije dretve inkrementiraju varijablu bez korištenja atomarnih operacija

```
1  struct increment_test : rl::test_suite<increment_test, 2> {
2    int x;
3
4    void before() {
5      x($) = 0;
6    }
7
8    void thread(unsigned thread_index) {
9      int value = x($);
10     x($) = value + 1;
11   }
12
13   void after() {
14     RL_ASSERT(x($) == 2);
15   }
16 };
```

4.4. C11Tester

C11Tester je alat za pronalaženje *race condition*-a u višedretvenim programima. Za razliku od ostalih navedenih alata, C11Tester podržava gotovo cijeli C++ memorijski model [11]. Moguće je isprobati sve moguće zamjene memorijskih operacija koje dopušta memorijski model kako bi se pronašle greške. Testovi za C11Tester izgledaju slično kao i Relacy testovi (potrebno je označiti operacije koje mogu utjecati na izvođenje druge dretve). Osim toga potrebno je prevoditi izvorni kod s ekstenzijom koja obavlja statičku analizu i zamjenjuje implementaciju standardnih atomarnih i sinkronizacijskih primitiva. Mogućnost pronalaženja greški koje proizlaze iz semantika memorijskog modela izrazito je bitan napredak u ovom području, no dodatno ograničava maksimalnu moguću veličinu testnog programa.

5. Sustav za testiranje

5.1. Opis i ciljevi

Glavni cilj našeg sustava je omogućiti pronalazak nedeterminističkih pogreški u *lock-free* algoritmima koje su uzrokovane vremenskim rasporedom dretvi. Veliku važnost stavljamo na korisničko iskustvo korištenja sustava. Testiranje algoritma ne bi nikako trebalo podrazumijevati mijenjanje njegovog izvornog koda. Osim što to pojednostavljuje korištenje sustava, također nam omogućuje da testiramo algoritme od kojih nemamo izvorni kod (npr. iz biblioteka) i najmanje moguće utječemo na optimizacije tijekom prevođenja. Testiranje algoritama sa svim mogućim prevoditeljskim optimizacijama nije samo moguće, već je i poželjno. Ove mogućnosti smatramo najvećim unaprjeđenjem naspram postojećih sustava za testiranje.

Naš sustav za testiranje se sastoji od tri glavna dijela: *tracing* koraka, analize prikupljenih podataka, i serijaliziranog izvođenja testa. U *tracing* koraku test se izvršava normalno te se bilježe informacije o izvršenim instrukcijama. Neke od prikupljenih informacija su vrsta i adresa instrukcije, identifikacijska oznaka dretve koja izvodi instrukciju, popis operanda i memorijskih lokacija iz kojih instrukcija piše ili čita. U sljedećem koraku jednostavnom analizom zaključujemo koje instrukcije predstavljaju serijalizacijske točke, te koliko ih puta izvodi svaka dretva. U konačnom koraku test se izvršava velik broj puta, gdje se jedan za drugim isprobavaju svi mogući vremenski rasporedi dretvi. To se postiže ubacivanjem koda ispred svake instrukcije određene prethodnom analizom. Dolaskom do instrukcije koja predstavlja serijalizacijsku točku dretva odlučuje hoće li nastavljati s izvršavanjem, ili će prepustiti izvršavanje drugoj dretvi.

5.2. Struktura testnog programa

Kako bi se testirala neka struktura podataka potrebno je napisati poseban testni program koji koristi funkcije definirane u `test_tools.h`. Svaki test je samostojeći program koji ne ovisi o našem sustavu za testiranje. Test je moguće pokrenuti kao normalan program. Funkcije iz `test_tools.h` ne rade gotovo ništa; pokretanjem programa kroz sustav za testiranje njihov kod se zamjenjuje pripadajućom implementacijom potrebnom za rad sustava. Njihova implementacija varira ovisno o tome izvršava li se test u `tracing` ili `testing` koraku. Isječak 5.1. služi kao predložak za strukturu testnog programa.

Isječak 5.1.: Struktura testnog programa

```
#include <thread>
#include "test_tools.h"

void test() {
  int thread_id = RegisterThread();

  while (Testing()) {
    // Slijedi inicijalizacijski kod, prije poziva ‘RunStart()‘.

    RunStart();
    // Slijedi testni kod, nakon poziva ‘RunStart()‘ i prije poziva ‘RunEnd()‘.
    // Primjer provjera uvjeta testa.
    AssertAlways(true);

    RunEnd();
  }
}

int main() {
  std::thread t1(test);
  std::thread t2(test);
  t1.join();
  t2.join();
  return 0;
}
```

Testni program može stvoriti neograničen broj pomoćnih dretvi, no naš sustav podržava instrumentaciju i testiranje samo dvije dretve koje je potrebno registrirati pozivom na `RegisterThread()` funkciju. Vraćena vrijednost je identifikacijska oznaka dretve i može se koristiti za npr. pomoćni ispis tijekom otklanjanja grešaka. Glavna petlja služi sustavu za testiranje kako bi bilo moguće uzastopno ponoviti izvršavanje testa, svaki put s drugačijim rasporedom dretvi, bez ponovnog stvaranja novog procesa.

Kod neposredno nakon ulaska u petlju služi za inicijalizaciju testnog stanja. Nakon izvršavanja inicijalizacijskog koda, odnosno tijekom poziva `RunStart()` funkcije, stanje programa mora biti isto bez obzira na specifičnu iteraciju testne petlje. Inicijalizacijski blok se može smatrati kritičnom sekcijom - svaka dretva će jedna za drugom izvršavati svoj inicijalizacijski kod, dok ostale dretve čekaju.

Svaka dretva čeka ostale u `RunStart()` pozivu, nakon čega izvršavanje testnog koda može započeti. Kod između poziva `RunStart()` i `RunEnd()` bit će instrumentiran i analiziran za međuovisnosti o dijeljenoj memoriji te će sustav ponavljati njegovo izvršavanje za sve moguće rasporede dretvi. Unutar ovog bloka koriste se funkcije `AssertAlways()` (uvjet treba biti istinit u svakom mogućem rasporedu dretvi) i `AssertAtleastOnce()` (postoji barem jedan raspored dretvi za koju je uvjet zadovoljen).

Isječak 5.2. sadrži potpuni testni program za FIFO red iz uvodne sekcije 2.2. Implementacija FIFO reda je ispravna, no sami test sadrži grešku. Testiramo red koji može pohraniti najviše dva elemenata, a u red pokušavamo ubaciti tri elementa. Tako možemo provjeriti da se red ponaša ispravno kad u njega dodajemo ili mičemo elemente bez obzira je li prazan, pun, ili sadrži nekoliko elemenata. Nakon analize izvedenih instrukcija sustav će umetnuti sinkronizacijske točke u dijelove koda koje pristupaju dijeljenoj memoriji unutar `Push()` i `Pop()` metoda.

Isječak 5.2.: Test SPSC FIFO reda iz isječka 2.2.

```
using QueueT = QueueT<int, 3>;

void producer(QueueT& q) {
  int thread_id = RegisterThread(0);
  while (Testing()) {
    // Reset the queue.
    q.~QueueT();
    new (&q) QueueT();

    RunStart();
    bool push1 = q.Push(PreventOpt(1));
    bool push2 = q.Push(PreventOpt(2));
    // `push3` might fail because the queue can contain at most 2 elements.
    bool push3 = q.Push(PreventOpt(3));

    AssertAlways(push1 && push2);
    // This is wrong! `push3` might not succeed every time.
    AssertAlways(push3);
    RunEnd();
  }
}
void consumer(QueueT& q) {
  int thread_id = RegisterThread(1);
  while (Testing()) {
    RunStart();
    uint32_t value;

    // Any pop might fail because a value might not be present.
    bool pop1 = q.Pop(value);
    AssertAlways(!pop1 || (value > 0 && value == 1));

    bool pop2 = q.Pop(value);
    AssertAlways(!pop2 || (value > 0 && value <= 2));

    bool pop3 = q.Pop(value);
    AssertAlways(!pop3 || (value > 0 && value <= 3));
    RunEnd();
  }
}
int main() {
  QueueT q;
  std::thread t1(producer, std::ref(q));
  std::thread t2(consumer, std::ref(q));
  t1.join();
  t2.join();
  return 0;
}
```

Nakon nekoliko iteracija test će prijaviti grešku sa točnim redoslijedom izvođenja instrukcija i rasporeda dretva (tzv. *execution history*) (isječak 5.3.). Korisnik može ponovno pokrenuti isti *execution history* s dodatnim ispisom za pronalaženje greške ili priključenim *debugger*-om.

Isječak 5.3.: Primjer ispisa *execution history*-a koji je doveo do greške.

```
< T1: going to sleep before executing 0x4014d0 (mov)
        > T2: will execute 0x401510 (mov)
        > T2: will execute 0x401514 (mov)
        > T2: will execute 0x401510 (mov)
        > T2: will execute 0x401514 (mov)
        > T2: will execute 0x401510 (mov)
        > T2: will execute 0x401514 (mov)
        < T2: completed the test!
> T1: will execute 0x4014d0 (mov)
> T1: will execute 0x4014d9 (mov)
> T1: will execute 0x4014f0 (mov)
> T1: will execute 0x4014f8 (mov)
> T1: will execute 0x4014d0 (mov)
> T1: will execute 0x4014d9 (mov)
> T1: will execute 0x4014f0 (mov)
> T1: will execute 0x4014f8 (mov)
> T1: will execute 0x4014d0 (mov)
> T1: will execute 0x4014d9 (mov)
< T1: ASSERTION FAILED!
```

5.3. Implementacija

Implementaciju sustava za testiranje moguće je podijeliti u četiri veće cjeline:

1. Instrumentacija za prikupljanje podataka o izvršavanju programa,
2. analiza prikupljenih podataka i određivanje sinkronizacijskih točaka,
3. program za pokretanje testa u serijaliziranom načinu rada i
4. pomoćne funkcije za korištenje u testnim programima.

5.3.1. Instrumentacija

Sustav za testiranje najprije pokreće test s uključenom instrumentacijom i prikuplja informacije o programu koji se izvodi. Instrumentacija se događa na razini strojnog koda, nije potreban pristup izvornom kodu testnog programa. Prikupljanje podataka je implementirano pomoću radnog okvira za dinamičku instrumentaciju DynamoRIO u datoteci `src/collector.cpp`. Bilježimo informacije samo o instrukcijama koje pristupaju memoriji. Od informacija bilježimo vrstu i adresu instrukcije, koja dretva izvršava instrukciju te memorijske adrese koje instrukcija čita i/ili piše. Prije svake relevantne instrukcije ubačeno je nekoliko instrukcija koje zapisuju te podatke u međuspremnik koji se periodično zapisuje u datoteke za daljnju analizu.

Veliki nedostatak našeg sustava je da prikupljamo ove informacije samo o izvršenim instrukcijama. Budući da za vrijeme instrumentacije ne kontroliramo vremenski raspored dretvi, moguće je da preskočimo izvršavanje nekih instrukcija koje bi služile kao sinkronizacijske točke. Ovaj problem umanjujemo pokretanjem instrumentacije veliki broj puta i dodavanjem nasumičnog čekanja (od $1e1$ do $1e9$ instrukcija) prije svakog pokretanja. U diskusiji (6.2.2.) predlažemo robusnija rješenja ovog problema.

Osim informacija o izvršenim instrukcijama također prikupljamo podatke o učitanim modulima. Modul može biti ili biblioteka (`.so` i `.dll` datoteke) ili izvršna datoteka koja se dinamično učita za vrijeme izvođenja (to je najčešće sistemski `libc` ali uključuje i sve ostale dinamičke biblioteke koje koristi testni program). Bilježenje podataka o modulima je nužno zbog ASR-a sigurnosne značajke koju pružaju operacijski sustavi. Zbog ASR-a

(eng. *Address Space Randomization*) tijekom svakog pokretanja programa moduli će biti inicijalizirani na različite nasumične virtualne adrese. Moramo bilježiti te nasumične bazne adrese modula kako bi tijekom faze testiranja mogli ubaciti sinkronizacijske točke na točna mjesta.

5.3.2. Analiza

Postoji prevelik broj kombinacija vremenskog rasporeda dretvi da bi ih isprobali sve. Ako dvije dretve izvode n i m instrukcija, ukupan broj mogućih rasporeda dretvi je $\binom{n+m}{n}$. Već za $n, m >= 24$ potrebno je nekoliko dana za isprobati sve moguće kombinacije na tipičnom računalu. Potrebno je utvrditi koji rasporedi dretvi mogu utjecati na rezultat izvođenja programa. To mogu biti samo rasporedi dretvi gdje jedna dretva predaje kontrolu drugoj prije instrukcije koja pristupa dijeljenoj memoriji. Ako neka instrukcija ne pristupa memoriji, ili pristupa lokalnoj memoriji neke dretve, nije moguće da prebacivanje kontrole drugoj dretvi utječe na rezultat izvođenja programa.

Dijeljena memorija je ona kojoj se pristupa iz više od jedne dretve, a cilj analize je pronaći instrukcije koje pristupaju dijeljenoj memoriji. Za svaku dretvu potrebno je pronaći skup svih memorijskih adresa s kojih dretva čita, i skup svih adresa na koje dretva piše. Adrese dijeljene memorije su presjek adresa s kojih čita ili piše jedna dretva, i adresa na koje piše druga. Ako dvije dretve samo čitaju s neke adrese (nikada ne pišu), ta adresa se ne smatra dijeljenom memorijom. Obje dretve mogu čitati s iste adrese u bilo kojem redoslijedu bez da promijene rezultat izvršavanja programa, stoga je potrebno da barem jedna dretva na nju upisuje podatke kako bi ju smatrali dijeljenom.

Kao što smo naveli u objašnjenju instrumentacije (sekcija 5.3.1.), moguće je da tijekom izvođenja testa u fazi instrumentacije dio instrukcija bude preskočen. Na primjer, zbog preskakanja neke kondicionalne grane moguće je preskočiti instrukcije koje bi inače pristupale dijeljenoj memoriji i te instrukcije tada ne bi bile uključene u analizu. Ovu limitaciju pokušavamo ublažiti ponavljanjem procesa instrumentacije veliki broj puta, kako bi korak analize pronašao što više relevantnih instrukcija. Pronalazimo set instrukcija koje pristupaju dijeljenoj memoriji za svako pokretanje instrumentacije i na kraju koristimo njihovu uniju kao skup sinkronizacijskih točaka.

Algorithm 1 Određivanje instrukcija koje pristupaju dijeljenoj memoriji

t: oznaka dretve
i: adresa instrukcije
r: set memorijskih adresa s kojih instrukcija čita
w: set memorijskih adresa u koje instrukcija piše

Ulaz: A ▷ Lista uređenih parova (t, i, r, w)
Izlaz: S ▷ Lista parova (t, i) instrukcija koje pristupaju dijeljenoj memoriji
$R \leftarrow \emptyset$
$W \leftarrow \emptyset$
for all $(t, i, r, w) \in A$ **do**
    $R \leftarrow R \cup \{(t, r)\}$
    $W \leftarrow W \cup \{(t, w)\}$
end for
$M \leftarrow \emptyset$ ▷ Dijeljene memorijske adrese
for all $(t_w, a_w) \in W$ **do**
    $M \leftarrow M \cup \{a_w \cap a_r \mid (t_r, a_r) \in R, t_r \neq t_w\}$
    $M \leftarrow M \cup \{a_w \cap a_{w2} \mid (t_{w2}, a_{w2}) \in W, t_{w2} \neq t_w\}$
end for
$S \leftarrow [(t, i) \mid (t, i, r, w) \in A, (w \cap M)$ **or** $(r \cap M)]$
return S

5.3.3. Pokretanje serijaliziranih testova

U serijaliziranom načinu izvođenja testova u svakom trenu samo jedna dretva izvodi program dok druga čeka svoj red. Dretve se mogu zamijeniti za kontrolu samo na sinkronizacijskim točkama. Sinkronizacijske točke su umetnute prije svake instrukcije koja pristupa dijeljenoj memoriji. Rezultat analize je set instrukcija koje pristupaju dijeljenoj memoriji i koliko puta svaka od dretvi izvršava te instrukcije. Ako prva i druga dretva izvode n i m instrukcija koje pristupaju dijeljenoj memoriji tada je ukupan broj mogućih rasporeda dretvi $\binom{n+m}{n}$. Iz praktičnih razloga ograničavamo maksimalni broj rasporeda dretvi na 2^{32}. Ako obje dretve izvršavaju isti kod ograničeni smo na 17 sinkronizacijskih točki po dretvi, što je u praksi dovoljno za testiranje jednostavnijih *lock-free* struktura.

Sustav će redom izvršiti sve moguće vremenske rasporede dretvi. Sustav ubacuje pozive posebnim sinkronizacijskim funkcijama prije svake relevantne instrukcije, gdje dretva može ili nastaviti s radom ili predati kontrolu drugoj dretvi i čekati svoj red. Unutar svakog pokretanja testnog programa svi `AssertAlways()` pozivi moraju biti istiniti, a u suprotnom sustav javlja grešku s identifikacijskom oznakom *execution history*-a. Korisnik može ponovno pokrenuti neuspješan testni primjer sa zadanim rasporedom dretvi što omogućava lakše reproduciranje i *debugging* grešaka, bez da ovisi o nasumičnom raspoređivanju dretvi.

Isječak 5.4. prikazuje kako radi izvođenje testa u serijaliziranom načinu rada. Prikazan je program koji zbraja i oduzima 1 nekoj varijabli. Program izvršavaju dvije dretve (`T1` i `T2`). Program (pogrešno) pretpostavlja da će rezultat na kraju uvijek biti 0, no sustav za testiranje pronalazi *execution history* za koji ta pretpostavka ne vrijedi.

Isječak 5.4.: Primjer *execution history*-a.

```
                    T1                                  T2
1       mov eax, [x]    (eax: 0)
2       add eax, 1      (eax: 1)
3       mov [x], eax    (x: 1)

1                                       mov eax, [x]    (eax: 1)
2                                       add eax, 1      (eax: 2)
3                                       mov [x], eax    (x: 2)
4                                       mov eax, [x]    (eax: 2)
5                                       sub eax, 1      (eax: 1)
6                                       mov [x], eax    (x: 1)
7                                       test [x], 0     (FAIL, x != 0)
```

5.3.4. Pomoćne funkcije za pisanje testova

Header datoteka `test_tools.h` sadržava pomoćne funkcije za pisanje testova. Sustav za testove prepoznaje pozive ovim funkcijama i zamjenjuje ih internim implementacijama za vrijeme izvršavanja testnog programa. Budući da stvarni sadržaj funkcija nije poznat za vrijeme prevođenja programa prevoditelj može generirati optimizacije koje će učiniti kod neispravnim. Na primjer, prevoditelj smije premjestiti poziv funkcije ispred ili iza nekog drugog koda ako može dokazati da premještaj ne utječe na rezultat programa (isječak 5.5.).

Isječak 5.5.: Primjer *reordering*-a koje prevoditelj smije napraviti budući da je funkcija prazna za vrijeme prevođenja.

```
void FunctionReplacedAtRuntime(int x) {
    // Function does nothing now, but will have its body replaced at runtime.
}
int main() {
    int x = 1;
    FunctionReplacedAtRuntime(x);
    // The compiler place move the call here...
    int y = x + 1;
    // ... or here. The program is still valid.
    return y;
}
```

Prevoditelj također može preskočiti spremanje konteksta na stog prije poziva funkcije. U isječku 5.5. moguće je da varijabla `x` bude *inlined* u poziv naše funkcije. Budući da funkcija više nema argumenata registar u kojem bi se inače nalazila vrijednost argumenta `x` neće biti očuvan na stogu. Nakon što sustav zamjeni funkciju za vrijeme testiranja ta optimizacija više nije ispravna i program se nalazi u nedefiniranom stanju. Svakoj funkciji je stoga potrebno dati eksternu praznu definiciju te onemogućiti premještanje prije ili poslije poziva (isječak 5.6.). Ovo je jedini dio sustava koji treba redefinirati za testiranje koda napisanog u nekom drugom programskom jeziku koji se prevodi u strojni kod.

Isječak 5.6.: Primjer interne definicije funkcije `Testing()`

```
// Declare the function to be replaced by the system
// and disable C++ name mangling.
extern "C" bool _Testing();

// Define the symbol to be a function that returns immediately.
// This prevents the optimizer from "looking into" a function which
// means the call site can't elide backing up caller-saved registers.
__asm__(
    ".global _Testing\n"
    "_Testing:\n"
    "  ret"
);

// The user will actually call this wrapper function.
bool Testing() {
    // Prevent instruction reordering before/after call.
    __asm__ __volatile__("" ::: "memory");
    bool result = _Testing();
    __asm__ __volatile__("" ::: "memory");
    return result;
}
```

6. Rezultati i rasprava

6.1. Rezultati

Sustav smo testirali na nekoliko vlastitih implementacija jednostavnih *lock-free* algoritama i struktura podataka. Dio njih su uključivale očite programske greške kako bi mogli provjeriti da ih sustav može pronaći.

- `inc_dec` je jednostavni testni program koji inkrementira i dekrementira varijablu te zatim provjerava da je konačna vrijednost nepromijenjena. Test ne koristi atomarne operacije i očekujemo da će sustav u njemu pronaći greške.
- `inc_dec_atomic` je sličan prethodnom testu no koristi atomarne operacije za inkrementiranje i dekrementiranje. Očekujemo da će ovaj test proći bez greške.
- `stack` je jednostavna implementacija stoga koja koristi polje i pokazivač na vrh stoga. Ne koristi atomarne operacije. Očekujemo pronaći grešku.
- `stack_lockfree` je SPSC implementacija stoga koja koristi ulančanu listu i CAS operacije kako bi postigla sinkronizaciju. Očekujemo da će ovaj test proći bez greške.
- `queue` je jednostavna implementacija reda koja koristi polje i pokazivače za pisanje i čitanje, no ne koristi atomarne operacije. Očekujemo pronaći grešku.
- `queue_lockfree` je SPSC implementacija reda vrlo slična `queue` implementaciji, no koristi atomarne operacije kako bi postigla sinkronizaciju. Očekujemo da će ovaj test proći bez greške.

Nakon verifikacije sustava testirali smo i provjerili ispravnost nekoliko implementacija jednostavnijih *lock-free* struktura iz često korištenih biblioteka (tablica 6.1.). Bili smo ograničeni na implementaciju samo jednostavnijih struktura kao što su SPSC red, stog, *ring buffer* i slično zbog predugog vremena testiranja složenijih struktura, što je moglo trajati i do nekoliko dana.

Tablica 6.1. Pregled eksperimenata

Biblioteka	**Podatkovna struktura**	**Rezultat**
Vlastita implementacija	`inc_dec`	**pronađene greške**
	`inc_dec_atomic`	nisu pronađene greške
	`stack`	**pronađene greške**
	`stack_lockfree`	nisu pronađene greške
	`queue`	**pronađene greške**
	`queue_lockfree`	nisu pronađene greške
Boost Lockfree [12] ver 1.81.0-8	`spsc_queue`	nisu pronađene greške
	`stack`	nisu pronađene greške
	`mpmc_queue` (veličina 1)	nisu pronađene greške
	`mpmc_queue` (veličina 3)	testiranje traje predugo
Concurrency Kit [13] ver 0.7.2	`ck_fifo`	nisu pronađene greške
	`ck_hp_fifo`	testiranje traje predugo
	`ck_queue`	nisu pronađene greške
	`ck_ring`	nisu pronađene greške
	`ck_stack`	nisu pronađene greške
`lockfree` [14] ver 2.0.8	`mpmc::Queue`	nisu pronađene greške
	`spsc::Queue`	nisu pronađene greške
	`spsc::RingBuf`	nisu pronađene greške

6.2. Rasprava i budući rad

U ovom radu predstavljen je sustav za testiranje *lock-free* algoritama osnovan na instrumentaciji strojnog koda i dinamičkoj analizi. Sustavom je moguće sistematski isprobati sve moguće vremenske rasporede dretvi koje bi mogle utjecati na rezultat testnog programa, bez promjene izvornog *lock-free* koda. Sustav smo isprobali na nekoliko javno dostupnih *lock-free* algoritama i struktura podataka te smo provjerili njihovu ispravnost s obzirom na vremenski raspored dretvi.

6.2.1. Prednosti

Glavna prednost našeg sustava naspram ostalih sličnih sustava za testiranje *lock-free* koda je mogućnost testiranje postojećih algoritama i struktura podataka bez njihovog mijenjanja. Potrebno je samo napisati testni program koji koristi gotov *lock-free* kod. Osim toga, sustavom je moguće testirati kod čijem izvornom kodu nemamo pristup, na primjer ako se kod nalazi u dijeljenoj biblioteci. To je slučaj za većinu koda zbog zavisnosti na `libc` biblioteci od sustava. Naš sustav može pratiti i pristupe memoriji koje čine funkcije poput `memmset`, `memmove`, `malloc` itd. Izravno testiranje prevedenog izvršnog programa nam također omogućuje testiranje koda pri različitim optimizacijskim razinama, što je značajan napredak naspram ostalih sustava za testiranje. Mnoge programske greške u *lock-free* kodu javljaju se tek pri visokim optimizacijskim razinama. Čak i ako drugi sustavi podržavaju kompilaciju sa optimizacijama, oni zahtijevaju mijenjanje izvornog koda algoritma što znači da prevoditelj može primijeniti bitno drugačije optimizacije za vrijeme testiranja i za vrijeme stvarnog prevođenja. Reimplementiranjem `test_tools.h` datoteke moguće je pisati testove i isprobavati algoritme pisane u drugim jezicima osim `C`-a i `C++`-a koji se prevode u strojni kod.

6.2.2. Mane i moguća poboljšanja

Unatoč nekim napredcima u usporedbi na ostale slične sustave za testiranje, naš sustav ima velik broj mana i mogućnosti za poboljšanje. Najveća mana našeg sustava je korak instrumentacije koji treba prikupiti informacije o instrukcijama koje pristupaju dijeljenoj memoriji. Ovisno o vremenskom rasporedu dretvi za vrijeme instrumentacije moguće je preskočiti neke instrukcije. Ako je preskočena čak i jedna takva instrukcija više ne možemo biti sigurni da nećemo dobivati *false-positive* rezultate. Jedno rješenje ovog problema je provesti jednostavnu statičku analizu na strojnom kodu testnog programa. DynamoRIO pruža mogućnost analize svog strojnog koda u programu (ne samo instrukcija koje su bile izvršene) što bi nam omogućilo da provedemo jednostavnu analizu međuzavisnosti instrukcija i dijeljene memorije. Analiza ne bi bila potpuna u svim mogućim slučajevima, no omogućila bi sustavu da prijavi korisniku dijelove strojnog koda koji nisu mogli biti analizirani, te bi se oni tada mogli ručno označiti. Drugo rješenje je cikličko pozivanje cijelog sustava. Nakon prvog pokušaja instrumentacije i analize izvodi se testni korak za čije vrijeme je također uključena instrumentacija. Ako za vrijeme

izvođenja testa s nekim zadanim redoslijedom dretvi pronađemo novu instrukciju koja pristupa dijeljenoj memoriji ponavljamo proces analize i ponovno pokrećemo testiranje. Ovaj proces bi kroz neki broj iteracija trebao pronaći sve instrukcije koje pristupaju dijeljenoj memoriji - ako je određenim redoslijedom izvođenja dretvi neka instrukcija bila preskočena, tada će izvođenjem svih mogućih redoslijeda ta instrukcija biti pronađena. Ova promjena može se jednostavno nadodati u postojeći sustav te bi joj jedini nedostatak bio povećano ukupno vrijeme testiranja koje je već sada dugo.

Naš sustav za testiranje nikako ne uzima u obzir memorijski model procesorske arhitekture i *memory-order* atomarnih operacija. Budući da je izvođenje testova uvijek potpuno serijalizirano (u nekom trenu izvršava se samo jedna dretva), sve atomarne operacije automatski dobivaju najstroži mogući poredak (`std::memory_order_seq_cst`). Ovaj nedostatak je izvan dosega ovog rada. Naš sustav izvršava testove na optimiziranom strojnom kodu, a prevoditelju je dozvoljeno zamijeniti redoslijed operacija ako to i dalje zadovoljava memorijski model. Tako „slučajno" možemo pronaći neke od ovih grešaka.

Preostale mane su povezane s ograničenjima kojih se testni programi moraju pridržavati i dugim vremenom potrebnim za testiranje. Naš sustav trenutno isprobava sve moguće vremenske poretke dretvi. Broj tih permutacija raste vrlo brzo i dovodi do kombinatorne eksplozije. To nas ograničava na testiranje malih programa koji ne izvode više od otprilike 32 instrukcije koje pristupaju dijeljenog memoriji po dretvi.

Jedan od načina smanjivanja broja isprobanih redoslijeda dretvi je bilježenje onih redoslijeda u kojima jedna od dretvi završi izvođenje ranije (odnosno, koja izvrši manje instrukcija nego što je određeno analizom). Svi ostali redoslijedi koji imaju isti prefiks, odnosno jedna od dretvi završava ranije istim slijedom instrukcija, mogu biti preskočeni.

Broj isprobanih permutacija također je moguće smanjiti primjenom statičke analize, kao što je redukcija simetrijom [15]. Ako obje dretve izvode isti kod (na primjer, obje izvršavaju istu testnu funkciju) redoslijed izvođenja dretvi $0 \rightarrow 1 \rightarrow 0$ imat će isti rezultat kao i $1 \rightarrow 0 \rightarrow 1$. U ovom slučaju možemo odstraniti pola svih kombinacija dretvi. Ako dretve izvode različiti kod i dalje je moguće pronaći neki podskup instrukcija koje obje dretve izvršavaju i tako smanjiti ukupan broj permutacija.

Druga metoda statičke analize koja može smanjiti broj isprobanih permutacija dretvi je redukcija parcijalnog poretka (eng. *partial order reduction*) [16]. Analizom podatkovnih međuzavisnosti (eng. *data dependencies*) moguće je podijeliti dijeljenu memoriju i instrukcije koje im pristupaju na više disjunktnih podskupova. Testovi koje smo koristili za validaciju sustava uglavnom nisu imali podatkovne nezavisnosti, no ova analiza bi omogućila pisanje duljih testova gdje si možemo priuštiti ubacivanje više nezavisnih dijelova (npr. korištenje više od jednog SPSC reda u testu).

Smanjivanje broja isprobanih permutacija bi omogućilo testiranje većih programa, što bi omogućilo praktično pisanje testova koji koriste više od dvije dretve. Sustav se jednostavno može proširiti s podrškom za više od dvije dretve.

Naš sustav ne podržava korištenje kritičkih odsječaka. Ulazak u kritički odsječak korištenjem API-eva poput `futex` stvara *deadlock* od kojeg se sustav ne može oporaviti. DynamoRIO omogućava presretanje sistemskih poziva što nam omogućava prijenos kontrole izvođenja s dretve koja bi ušla u čekanje ispred kritičkog odsječka na slobodnu dretvu. Ova nadogradnja bi omogućila testiranje raznih primitiva za sinkronizaciju kao što su razne varijante *lock*-ova i semafora, kondicijske varijable (monitori) i ostalo.

7. Zaključak

U ovom smo radu predstavili sustav za testiranje *lock-free* algoritama i struktura podataka koji ne zahtijeva modificiranje izvornog koda algoritma niti modeliranje svojstava algoritma u nekom drugom jeziku. Posljedično naš sustav može testirati algoritme bez pristupa izvornom kodu (npr. iz zapakiranih biblioteka) te može efektivno testirati algoritme koji su prevedeni s najvišim stupnjem optimizacija. Sustav pokreće testni program u serijaliziranom načinu rada – u svakom trenu izvodi se jedna točno određena dretva, dok druga dretva čeka svoj red. Sustav isprobava sve redoslijede dretvi koje potencijalno mogu promijeniti rezultat programa. Koristeći dinamičku instrumentaciju strojnog koda sustav određuje *sinkronizacijske točke* – instrukcije ispred kojih se može desiti zamjena aktivne dretve.

Smatramo da je naš sustav pristupačniji korisnicima od prijašnjih alata slične prirode. Korisnici samo trebaju napisati standardni *unit* test kojim definiraju željena svojstva algoritma – nije potrebno označavati kritične sekcije izvornog koda koje potencijalno mogu sadržavati greške. Do sada zanemarena mogućnost testiranja optimiziranog strojnog koda predstavlja značajan napredak u automatskom testiranju *lock-free* algoritama.

Unatoč spomenutim postignućima naš sustav ima i velik broj nedostataka. Kao i svi ostali alati slične prirode ograničeni smo na testiranje relativno kratkih *lock-free* algoritama. Naš sustav je ovdje ograničeniji od nekih drugih zbog nedostatka statičke analize koja bi mogla smanjiti ukupan broj stanja koje treba evaluirati.

Daljnji rad na sustavu bi najprije riješio implementacijski problem dinamičke instrumentacije. Za vrijeme prikupljanja podataka o izvršenim instrukcijama moguće je preskočiti instrukcije koje bi služile kao sinkronizacijske točke. Problem se može riješiti rekurzivnim pozivanjem instrumentacije i izvršavanja testa u serijaliziranom načinu rada (detaljno obrazloženje nalazi se u raspravi 6.2.2.).

Sustavom smo provjerili ispravnost nekoliko javno dostupnih *lock-free* algoritama i struktura podataka (npr. `boost` *lock-free* biblioteku). Testiranjem nismo otkrili greške u bibliotekama, no povećali smo povjerenje u njihov ispravan rad bez obzira na vremenski poredak dretvi.

Literatura

[1] R. H. Netzer i B. P. Miller, "What are race conditions? some issues and formalizations", *ACM Letters on Programming Languages and Systems (LOPLAS)*, sv. 1, br. 1, str. 74–88, 1992.

[2] R. Alur i G. Taubenfeld, "Contention—free complexity of shared memory algorithms", u *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 1994., str. 61–70.

[3] M. Herlihy, V. Luchangco, i M. Moir, "Obstruction-free synchronization: Double-ended queues as an example", u *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* IEEE, 2003., str. 522–529.

[4] R. Alur i G. Taubenfeld, "Contention—free complexity of shared memory algorithms", u *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 1994., str. 61–70.

[5] "ISO/IEC 14882:2023 Programming Languages - C++; Abstract Machine", Standard, International Organization for Standardization, str. 11, prosinac 2023., working Draft. [Mrežno]. Adresa: https://www9.open-std.org/JTC1/SC22/WG21/docs/papers/2023/n4950.pdf

[6] D. Bruening i S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation", 2004.

[7] N. Nethercote i J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", *ACM Sigplan notices*, sv. 42, br. 6, str. 89–100, 2007.

[8] L. Lamport, “Specifying systems: the tla+ language and tools for hardware and software engineers”, 2002.

[9] M. Ben-Ari, *Principles of the Spin model checker*. Springer Science & Business Media, 2008.

[10] D. Vyukov, “Relacy race detector”, 2024., accessed: 2024-06-20. [Mrežno]. Adresa: https://www.1024cores.net/home/relacy-race-detector

[11] W. Luo i B. Demsky, “C11tester: a race detector for c/c++ atomics”, u *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021., str. 630–646.

[12] B. Schäling, *The boost C++ libraries*. Boris Schäling, 2011.

[13] C. K. Contributors, “Concurrency kit: Concurrency primitives, safe memory reclamation mechanisms and non-blocking data structures”, https://github.com/concurrencykit/ck, 2024., bSD License.

[14] D. Nedic, “lockfree: A collection of lock-free data structures written in standard c++11”, https://github.com/DNedic/lockfree, 2024., mIT License.

[15] E. M. Clarke, E. A. Emerson, S. Jha, i A. P. Sistla, “Symmetry reductions in model checking”, u *Computer Aided Verification: 10th International Conference, CAV’98 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*. Springer, 1998., str. 147–158.

[16] C. Flanagan i P. Godefroid, “Dynamic partial-order reduction for model checking software”, *ACM Sigplan Notices*, sv. 40, br. 1, str. 110–121, 2005.

Sažetak

Tehnike dinamičke analize za otkrivanje grešaka u strukturama podataka bez zaključavanja

Gabriel Marinković

Testiranje višedretvenih *lock-free* algoritama predstavlja veliki problem zbog nedeterminističke prirode greški koje mogu ovisiti o vremenskom rasporedu dretvi. U ovom radu predstavljamo sustav kojim je moguće sistematski izvršiti sve moguće vremenske rasporede dretvi koje mogu dovesti do greške. U slučaju pronađene greške moguće je pouzdano reproducirati raspored dretvi koji ju izaziva. Sustav koristi dinamičku instrumentaciju strojnog koda kako bi drastično smanjio broj rasporeda dretvi koje treba testirati. Najveća postignuća našeg sustava naspram postojećih rješenja su mogućnost testiranja postojećih algoritama bez potrebe za izmjenom izvornog koda i mogućnost testiranja visoko-optimiziranih izvršnih datoteka. Koristeći naš sustav evaluirali smo i provjerili ispravan rad nekoliko popularnih javno dostupnih biblioteka sa *lock-free* strukturama podataka.

Ključne riječi: automatsko testiranje, konflikt istovremenog pristupa, dinamična instrumentacija, višedretvenost

Abstract

Dynamic analysis techniques for finding bugs in lock-free data structures

Gabriel Marinković

Automated testing of multithreaded lock-free algorithms poses a significant challenge due to the possibility of bugs whose appearance depends on the scheduling of threads. In this thesis we present a novel system for systematic testing of all possible thread scheduling orders which might impact the program's final result. Discovered bugs can be reliably reproduced with the same thread schedule. The most significant improvement of our system compared to previous methods is the ability to test existing code without any modifications to its source, and the ability to test their highly optimized builds. We used our system to evaluate and check the correctness of several popular publicly available libraries of lock-free data structures.

Keywords: automatic testing, race conditions, dynamic binary instrumentation, multithreading