УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Стефана Церовина

ВИРТУЕЛНА МАШИНА ДАРТИНО-ИМПЛЕМЕНТАЦИЈА ИНТЕРПРЕТАТОРА ЗА ПЛАТФОРМУ МИПС

мастер рад

Београд, 2016.

Ментор:

др Милена Вујошевић Јаничић
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков
Универзитет у Београду, Математички факултет

др Филип Марић
Универзитет у Београду, Математички факултет

Датум одбране: _____

Брату, мами и тати

Наслов мастер рада: Виртуелна машина Дартино- имплементација интерпретатора за платформу МИПС

Резиме:

Кључне речи: МИПС, систем са уграђеним рачунаром, интерпретатор,

Садржај

Глава 1

Увод

Појам „Интернет ствари” (енг. *Internet of things*), скраћено ИоТ (енг. *IoT*), се све чешће среће, и он представља модел повезаности објеката који чине огроман систем у оквиру којег објекти међусобно комуницирају. Дефиниција „ствари” у оквиру појма „Интернет ствари” варира, али се може рећи да је то систем са уграђеним рачунаром (енг. *embedded system*) који преноси и прима информације путем мреже. Системи са уграђеним рачунаром су системи специјалне намене, који обављају једну или више функција које тој намени одговарају. Модерни системи са уграђеним рачунаром су углавном базирани на микроконтролерима, а ређе на микропроцесорима, зато што микроконтролере карактерише ефикасно управљање процесима у реалном времену, масовна производња, ниска цена и мала потрошња електричне енергије. Сматра се да је потенцијал за развој индустрије система са уграђеним рачунаром велики, и да је будућност у изградњи велики ИоТ система.

Развијање апликација за системе са уграђеним рачунаром се обично своди на програмирање у асемблеру или у програмском језику Ц (енг. *C*). Најчешће се користе компилатори ГЦЦ (енг. *GCC*) или ЛЛВМ (енг. *LLVM*), а од алата за дебаговање, обично алат ГДБ (енг. *GDB*). Развојно окружење се најчешће налази на другом рачунару, јер микроконтролер садржи малу количину РАМ и флеш меморије, па се на њему обично не може покренути ниједан регуларни оперативни систем, а и микролинукс често захтева више РАМ меморије од оне којом микроконтролер располаже. Због наведених услова, процес развоја апликација је доста захтеван, подложан грешкама па због тога и спор.

Пошто се Веб апликације много брже развијају од апликација за системе са уграђеним рачунаром, компанија Гугл (енг. *Google*) је дошла на идеју да

омогући програмирање система са уграђеним рачунаром у вишем програмском језику Дарт, за који постоји подршка за развој Веб, серверских и мобилних апликација. Више о програмском језику Дарт биће речено у поглављу 3. Тако је настала виртуелна машина Дартино, са намером да се програмирање система са уграђеним рачунаром олакша и приближи што већем броју програмера. Дартино омогућава писање апликација за мале микроконтролере, на језику који доста личи на Ц, али је објектно-оријентисан и садржи разне погодности које процес имплементације доста олакшавају, те се апликације могу развијати брже и ефикасније. Ова виртуелна машина је детаљније описана у поглављу 4.

У оквиру Дартино виртуелне машине биле су подржане само Интел и Арм архитектуре процесора. Због широке распрострањености МИПС процесора у системима са уграђеним рачунаром, у оквиру овог рада развијен је интерпретатор за платформу МИПС који је и званично интегрисан у Дартино пројекат. Имплементација интерпретатора описана је у поглављу 5, док се о платформи МИПС више може прочитати у поглављу 2.

Глава 2

МИПС

2.1 ЦИСК и РИСК

Глава 3

Програмски језик Дарт

3.1 Уопштено

3.2 Карактеристике

Приватност

Конкурентност

Грешке и упозорења

Промењљиве

Функције

Оператори

Класе

Интерфејси

Енумерације

Генерици

Уграђени типови

Библиотеке

3.3 Специфичности у односу на јаваскрипт

3.4 Платформе

Глава 4

Дартино

У овом поглављу биће описана Дартино виртуелна машина. На почетку је описано чему служи Дартино и које су компоненте виртуелне. Након тога је представљен модел програмирања. У трећем одељку су представљене значајне карактеристике и новине ове виртуелне машине: паралелизам, корутине, изолате, влакна итд. Након тога описане су најзначајније библиотеке у оквиру Дартина и чему оне служе. На крају је описана употреба Дартина из командне линије и улога Дарта и Дартина при покретању програма.

4.1 Опис виртуелне машине

Виртуелна машина представља софтверску симулацију физичког рачунара. У зависности од употребе и степена сличности са физичким рачунаром, виртуелне машине се деле на виртуелне машине за симулирање система и виртуелне машине за симулирање процеса. Дартино виртуелна машина представља виртуелну машину за симулирање процеса. Виртуелне машине за симулирање процеса додају слој изнад оперативног система, који служи за симулирање програмског окружења, које је независно од платформе, и у оквиру ког се може извршавати један процес. Извршавање више процеса може се постићи креирањем више инстанци виртуелне машине. Овај тип виртуелних машина обезбеђује висок ниво апстракције. Програми се пишу у програмском језику вишег нивоа, и могу се извршавати на различитим платформама. Имплементација виртуелне машине се заснива интерпретатору.

Као што је речено у уводу, Дартино виртуелна машина је настала са циљем да се писање апликација за микроконтролере олакша. Како би се направило

боље окружење за рад, потребна је отворена и приступачна платформа, која ће већини програмера бити позната. Требало би омогућити статичку анализу кода, допуњавање кода, библиотеке са разним функционалностима, и тиме развијање апликација учинити ефикаснијим.

Пре Дартина је било могуће програмирати микропроцесоре у Дарту, али не и микроконтролере. Особине микроконтролера постављају велика додатна ограничења при развоју виртуелне машине која ће на њима радити. Обично се под виртуелном машином подразумева нешто што преводи изворни код у некакав међукод, и извршава га. Према томе, виртуелна машина садржи некакву компоненту за извршавање, сакупљач отпадака, омогућује препознавање класа и објеката, и има подршку за дебаговање.

Компоненте Дартино виртуелне машине:

1. Компонента за извршавање
2. Објектни модел
3. Сакупљач отпадака

Ове три компоненте су неопходне када је у питању виши програмски језик као што је Дарт. ЈИТ (енг. *JIT - Just in time*) компилатор изворног кода је избачен из виртуелне машине, и направљен је систем у коме су компилатор и окружење за извршавање раздвојени, као што је случај код ГЦЦ-а или ЛЛВМ-а. Систем за дебаговање је поједностављен и смањен.

Компилатор може да се налази било где, обично на рачунару где се развија апликација, и он комуницира са окружењем за извршавање које се потенцијално налази на другом систему. Интерфејс за комуникацију је једноставан, јер окружење за извршавање мора бити веома једноставно, како би одговарало карактеристикама миктроконтролера. Окружење за извршавање садржи командни АПИ, преко ког му компилатор може затражити да уради одређене ствари. Испод тога се заправо налази стек машина, тако да се може поставити ново стање на стек окружења за извршавање, могу се правити класе, дефинисати објекти и методе путем овог АПИ-ја. Компилатор класама додељује индентификаторе, помоћу којих при примању објекта назад, може утврдити којој класи објекат припада. Компилатор је Дарт програм, а окружење за извршавање је Дартино виртуелна машина. Више о извршним фајловима **dart**, **dartino-vm** и **dartino** написано је у одељку 4.5.

Слика 4.1: Комуникација ЈИТ компилатора и окружења за извршавање

Сва комуникација се врши преко жичног протокола, тако да је омогућено удаљено извршавање, и обично је базирано на TCP/IP протоколу. На овај начин велики део посла се пребацује на компилатор, а окружење за извршавање се максимално упрошћава.

Помоћу овог протокола се може постављати структура програма, што обухвата:

1. Додавање нових класа на стек
2. Додавање нових метода на стек

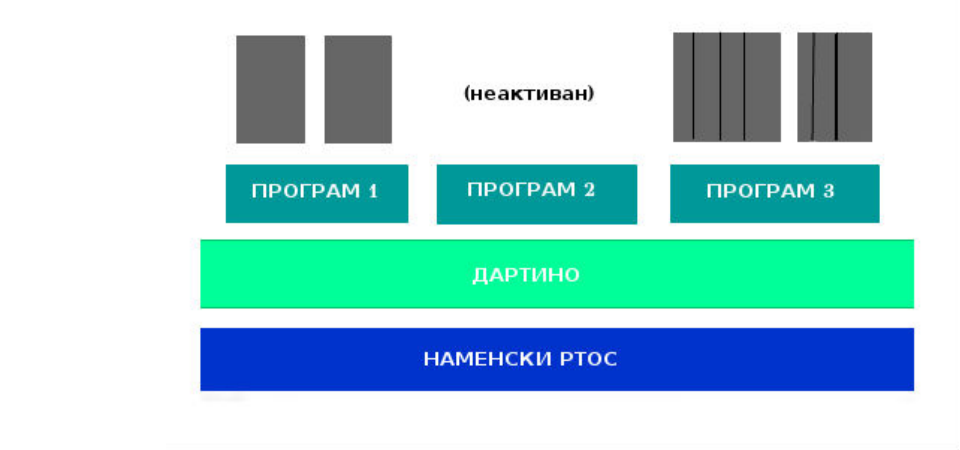
Може се мењати структура програма, што обухвата:

1. Мењање табела метода
   Може се рећи „од овог тренутка желим да ова класа поред метода А има и метод Б”. То значи да је омогућена висока интерактивност са системом.
2. Мењање шема
   Мењање шема се односи на мењање класе, нпр.додавањем нових поља. Оно што се дешава у тој ситуацији је да се врши пролаз кроз све инстанце класе, и оне се ажурирају и прилагођавају новом формату класе.
3. Дебаговање У оквиру дебаговања, могуће је:
   a) Покретање дебагера
   b) Постављање зауставних тачака
   c) Рестартовање дебагера

Систем за дебаговање је једноставан и ограничен, јер не зна ништа о изворном коду.

4.2 Модел програмирања

Доњи слој чини традиционални оперативни систем са уграђеним рачунаром у реалном времену(нпр. Фри РТОС). Изнад тога се налази Дартино окружење за извршавање, које може извршавати више програма, независно, и они не морају знати ништа једни о другима. Ово се разликује од уобичајених система који раде на оперативном систему са уграђеним рачунаром, и који обично не могу да извршавају више независних компоненти, на независан начин. За сваки програм можемо имати више процеса, више инстанци програма, или делова програма, који се извршавају конкурентно.



Слика 4.2: Модел програмирања

На слици 4.2 је приказана ситуација када имамо 3 програма. Програм2 се тренутно не извршава, за Програм1 су покренуте две инстанце, док се Програм3 извршава у 7 нити.

4.3 Значајне карактеристике

1. Лаки процеси

   За сваки процес нам је потребно мало меморије, неколико стотина бајтова. Сав код и подаци у меморији се деле, па меморију заузимају само

подаци који представљају процес. На овај начин ради већина оперативних система, али је то новина за оперативне системе са уграђеним рачунаром.

2. Блокирање процеса

   Процес се може блокирати, при чему се не заузимају ресурси, већ се главна нит враћа систему. Разлог за то је што у системима са уграђеним рачунаром обично имамо мали фиксирани број нити, па не желимо да их блокирамо.

3. Паралелизам

   Уколико процесор има више језгара, процеси се могу извршавати паралелно.

```
main() {
  for (int i = 0; i < 40000; i++) {
    Process.spawn(factoriel, 12);
  }
}

int factoriel(n) {
  if (n == 1) return 1;
  return n * factoriel(n-1);
}
```

Код **??** приказује покретање 40000 независних процеса, који ће бити распоређени на одређени број језгара и извршавати се паралелно. *Process.spawn* Креира нови процес и покреће га.

```
var server = new ServerSocket("127.0.0.1", 0);
while(true) {
  server.spawnAccept((Socket socket){
        //izvrsava se u novom procesu
        socket.send(UTF.encode("Hello"));
        socket.close();
  })
}
```

У коду **??** се креира ServerSocket који представља сервер, затим се помоћу метода *spawnAccept* креира нови процес. У оквиру тог процеса се чека

да клијент успостави конекцију са сервером, и самим тим блокира даље извршавање док се конекција не успостави. При успостављању конекције, клијент(*socket*) шаље серверу(*server*) поздравну поруку.

4. Комуникација

   Комуникација између процеса се обавља слањем порука, помоћу канала и портова. Канал представља један ред порука. Порт представља могућност слања поруке каналу, и без приступа одређеном порту, не може се слати порука каналу који чека на другој страни порта.

```
final channel = new Channel();
final port = new Port(channel);
Process.spawn(() {
        int i=0;
        while(i < 50) {
                port.send(i++);
        }
});
while(true) {
        print(channel.receive());
}
```

   Код ?? приказује један процес који шаље поруке, и функцију *print* која блокира извршавање све док не прими следећу поруку.

5. Дељено непроменљиво стање између процеса

   a) Сваки непроменљиви објекат се може слати као порука, без копирања

   b) Више процеса могу користити исти објекат истовремено

   c) Нема потребе за експлицитним примитивама за синхронизацију

6. Изолате

   Изолате (енг. *Isolates*) су независни воркери (енг. *Worker* [1]), свака изолата има своју хип меморију, и за разлику од нити нема дељења меморије између изолата. Изолате комуницирају само преко порука. Могу се паузирати, наставити и убити.

[1] Појам који означава нешто између процеса и нити. Извршава се у позадини и тиме не утиче на одзивност система.

```
main() {
  Expect.equals(1597, fib(16));
  Expect.equals(4181, fib(18));
}

fib(n) {
  if (n <= 1) return 1;
  var n1 = Isolate.spawn(() => fib(n - 1));
  var n2 = Isolate.spawn(() => fib(n - 2));
  return n1.join() + n2.join();
}
```

Код ?? приказује пример употребе изолата за рачунање елемената Фибоначијевог низа, рекурзивно, покретањем две независне изолате.

7. Влакна

Влакна (енг. *Fibers*) представљају нити извршавања у оквиру једног процеса, које деле спољну меморију, али не деле непроменљиву меморију. Омогућују да више независних нити једног процеса могу чекати на исту ствар(блокирати) независно једна од друге. На овај начин се имплементира вишеструка одвојена контрола тока.

```
void publishOnChange(Socket socket, String property, Channel input){
            int last = 0;
            while(true){
              int current = input.receive();
              if(current != last)
                socket.send(UTF.encode('("\$property": \$current)'));
              last = current;
            }
}

Fiber.fork(() => publishOnChange(server, "temperature", tempSensor));
Fiber.fork(() => publishOnChange(server,"humidity", humSensor));
```

Код ?? приказује употребу два влакна. У оквиру једног се чека на промену вредности у сензору за температуру, док се у оквиру другог чека на

промену вредности у сензору за влажност ваздуха. Функција *publishOnChange* блокира извршавање док се у сензору за који је позвана не деси промена вредности. На овај начин је омогућено да се у оквиру једног процеса чека на више ствари, и нема паралелизма, већ се уколико дође до блокирања у једном влакну, прелази на извршавање следећег који је спреман за извршавање.

8. Корутине

   Дартино има уграђену подршку за корутине. Корутине су објекти налик на функције, које могу давати више вредности, па су веома сличне генераторима. Када корутина врати вредност, зауставља се на тренутној позицији, а њен стек извршавања се чува за касније покретање. Када врати последњу вредност, сматра се завршеном, и више се не може покретати.

```
var co = new Coroutine((x) {
    Expect.isTrue(co.isRunning);
    Expect.equals(1, x);
    Expect.equals(2, Coroutine.yield(4));
    Expect.isTrue(co.isRunning);
});

Expect.isTrue(co.isSuspended);
Expect.equals(4, co(1));
Expect.isTrue(co.isSuspended);
Expect.isNull(co(2));
Expect.isTrue(co.isDone);
```

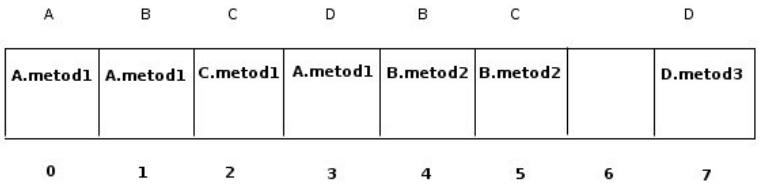
9. Динамичко отпремање метода

   Динамичко отпремање метода се односи на позивање преклопљених метода класа, при чему се проблем преклапања разрешава у фази превођења, а не у фази извршавања. Табела отпремања се имплементира као низ, у који се сваки метод, сваке класе, смешта на своју позицију. Позиција метода се рачуна помоћу редног броја класе и позиције метода. Оно што је неопходно при добијању одређеног метода из табеле је провера да ли је резултат заправо оно што нам треба. Уколико није, значи да тражении метод одређене класе не постоји. Загарантовано је константно

време отпремања, и табела отпремања се израчунава пре извршавања, па представља део апликације.

```
class A {
  metod1();
}
class B extends A {
  metod2();
}
class C extends B {
  metod1();
}
class D extends A {
  metod3();
}
```

Класама редом придружимо бројеве: 0, 1, 2 и 3. Методима придружимо позиције на мало компликованији начин. Метод *metod1*, који је дефинисан у највећем броју класа, придружимо позицију 0, и он се смешта на почетак табеле. Методу *metod2* придружимо позицију 3, и методу *metod3* придружимо 4. Позиције морају бити јединствени бројеви, како не би дошло до преклапања метода. Приметимо да се могу појавити празне позиције у табели.



Слика 4.3: Табела отпремања за дати пример

4.4 Најзначајније библиотеке

1. **file** - Интерфејс за рад са датотекама. Подржано само када се Дартино извршава на ПОСИКС платформи.

2. **ffi** - „foreign function interface" библиотека која омогућава да се из Дарт кода позивају функције дефинисане у неком другом програмском језику, нпр. функције програмског језика Ц.

3. **http** - Имплементација ХТТП (енг. *HTTP - HyperText Transfer Protocol*) клијента.

4. **mbedtls** - ТЛС (енг. *TLS - Transport Layer Security*) подршка, базирана на mbedtls. Ово се може користити на исти начин као нормални сокет (и да се прослеђује ХТТП пакету).

5. **mqtt** - MQTT клијентска библиотека за MQTT протокол, ИоТ протокол за размену порука заснован на објављивању/претплати.

6. **os** - Приступ оперативном систему. Подржано када се Дартино извршава на ПОСИКС платформи.

7. **socket** - Дартино имплементација ТЦП (енг. *TCP - Transmission Control Protocol*) и УДП (енг. *UDP - User Datagram Protocol*) сокета

8. **stm32** - Подршка за STM32 плоче.

4.5 Покретање Дарт програма унутар виртуелне мапи

Примарни начин за комуникацију са Дартино виртуелном машином је преко команде **dartino**. Ова команда комуницира са процесом који се извршава а не захтева интеракцију са корисником (енг. *persistent process*), који ради сав тежи посао. Помоћу њега **dartino** компајлира програм, а помоћу Дартино виртуелне машине (**dartino-vm**) извршава и дебагује програме .

Нешто више о извршним фајловима које се користе при покретању програма:

1. **dartino**
   Ова команда је C++ програм. Намера је била да буде што једноставнији програм који једноставно прослеђује стандардне улазно/излазне сигнале процесу који се извршава а не захтева интеракцију, преко сокета. Ово је

мали извршни фајл за покретање компилатора. Компилатор је написан у Дарту и извршава се у оквиру Дарт виртуелне машине. То је ЈИТ компилатор.

2. **dartino-vm**
   Самостална виртуелна машина која подржава превођење програма и покретање програма из бајткода.

3. **dart**
   Процес који се извршава а не захтева интеракцију са корисником. То је Дарт програм: Класична Дарт виртуелна машина за покретање компилатора. Овај извршни фајл се не генерише у оквиру Дартина, већ је део Дарта. Процес се састоји из главне нити, и неколико воркера. Главна нит ослушкује надолазеће конекције из **dartino** команде, и може да користи воркере да обави тражени задатак.

Шта се заправо дешава када се покрене команда **dartino run hello.dart**? Дартино је подразумевано повезан на локалну сесију, која је повезана на локалну виртуелну машину која ради на нашем рачунару. **dartino** преводи код до бајткода, помоћу Дарт компилатора, а онда га прослеђује **dartino-vm** која га извршава. Дартино виртуелна машина враћа резултат назад Дартину, и он га приказује.

Дартино подржава више сесија. Свака сесија може бити придружена различитој Дартино виртуелној машини, што омогућује кориснику да тестира више различитих уређаја истовремено. Тренутно је руковање сесијама експлицитно.

Покретање програма у оквиру одређене сесије је омогућено на следећи начин:

./out/DebugXMIPS/dartino create session my_session

Креирање нове сесије „my_session”.

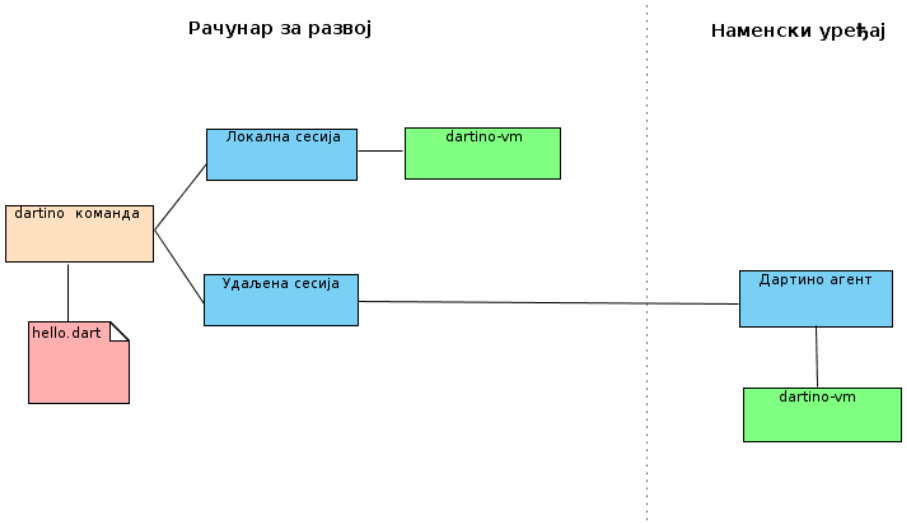
Након тога морамо да покренемо дартино виртуелну машину:

./out/DebugXMIPS/dartino-vm

Након чега се исписује порука типа „Waiting for compiler on 127.0.0.1:61745”. 61745 представља рандом генерисани порт.

У новом терминалу се прикачимо на дати сокет:

./out/DebugXMIPS/dartino attach tcp_socket 127.0.0.1:61745 in session my_session



Слика 4.4: Процес извршавања програма

А затим покренемо жељени програм у оквиру те сесије:

./out/DebugXMIPS/dartino run hello.dart in session my_session

Сесија се прекида помоћу следеће команде:

./out/DebugXMIPS/dartino x-end session my_session

Глава 5

Имплементација интерпретатора за платформу МИПС

У овом поглављу биће описан поступак имплементације интерпретатора за платформу МИПС, кроз делове кода и тест примере за које су одређене целине имплементиране. Након тога биће описан стек Дартино виртуелне машине, а затим и имплементирани систем за дебаговање. На крају ће бити описани још неки значајни тест примери, резултати тестирања на постојећем скупу тестова и апликација у језику Дарт имплементирана за потребе упоређивања перформанси интерпретатора.

5.1 Имплементација у програмском језику Ц++

Имплементација је рађена на рачунару са Интел x86 процесором, кроскомпилацијом, при чему је циљна архитектура МИПС32Р2. У почетку развоја, постојао је мултинаменски интерпретатор, који не зависи од архитектуре. Требало је само конфигурисати Дартино за платформу МИПС, и тиме би превођење за МИПС било омогућено. Конфигурисање подразумева додавање опције „mips" у списку подржаних архитектура и додавање опција које се користе приликом превођења. Да би се при превођењу добио одговарајући извршни фајл, било је непходно гцц-у проследити аргументе који указују на МИПС архитектуру: *-mips32r2*, *-EL*, и линкеру проследити *-EL*, што указује на литл ендиан (енг. *little endian*[1]). Да би се у фази линковања користиле МИПС библиотеке,

[1]Начин записа података у меморији тако да је на нижој адреси нижи бит меморијске речи.

постављене су путање до њих. МИПС извршне датотеке се покрећу помоћу КЕМУ (енг. *QEMU - Quick Emulator*) емулатора у корисничком режиму[2]. Да би се то вршило аутоматски, непходно је команди **update-binfmts** проследити вредности које одговарају мипс ЕЛФ (енг. *ELF - format of Executable and Linking Format* [3]) датотекама, и путању до скрипте која покреће КЕМУ емулатор. На овај начин ће се при покретању извршних датотека чије се меџик (енг. *magic*) и маск (енг. *mask*) вредности поклапају са овим, покретање извршити помоћу КЕМУ емулатора за МИПС.

```
sudo update-binfmts --install qemu-mipsel /<skripta>
--magic '\x7fELF\x01\x01\x01\x00\x00\x00\x00
        \x00\x00\x00\x00\x00\x02\x00\x08\x00'
--mask '\xff\xff\xff\xff\xff\xff\xff\x00\xfe\xff
        \xff\xff\xff\xff\xff\xfe\xff\xff\xff'
--credentials yes --package qemu-user-static
```

Где <skripta> представља путању до скрипте:

```
#!/bin/bash
qemu=/<qemu>/mipsel-linux-user/qemu-mipsel
exec $qemu $*
```

Где <qemu> представља путању до директоријума у ком се налази qemu емулатор.

Интерпретатор за платформу МИПС је урађен по угледу на постојећи интерпретатор за АРМ процесоре.

За почетак су направљене датотеке *assembler_ mips.h* и *assembler_ mips.cc* које описују МИПС асемблерски језик. Најпре је најпре дефинисана енумерација регистар која обухвата МИПС регистре (референца на део о мипс регистрима). Направљене су класе *Label*, *Constant* и *Address*.

[2]У овом режиму КЕМУ може покретати процесе преведене за једну врсту процесора на другој врсти процесора

[3]Бинарни формат извршних датотека на Линукс оперативном систему

```
class Label {
 public:
  Label() : position_(-1) {}

  int position() {
    if (position_ == -1) position_ = position_counter_++;
    return position_;
  }

 private:
  int position_;
  static int position_counter_;
};
```

Лабела представља линију у оквиру асемблерског фајла на коју се може скочити. Класа *Label* садржи два поља: *position* - представља редни број лабеле и *position_ counter* - заједничка променљива за све инстанце класе *Label*, на основу које се додељује позиција лабеле.

```
class Immediate {
 public:
  explicit Immediate(int32_t value) : value_(value) {}
  int32_t value() const { return value_; }

 private:
  const int32_t value_;
};
```

Инстанца класе *Immediate* представља константу вредност целобројног типа.

```
class Address {
 public:
  Address(Register base, int32_t offset)
      : base_(base), offset_(offset) {}

  Register base() const { return base_; }
  int32_t offset() const { return offset_; }

 private:
  const Register base_;
  const int32_t offset_;
};
```

Инстанца класе *Address* представља меморијску адресу, која је на платформи МИПС32Р2 32-битна. Адреса се рачуна као збир адресе у базном регистру и помераја. Базни регистар је регистар опште намене који садржи 32-битну адресу.

Дефинисани су макрои за МИПС инструкције са одговарајућим бројем аргумената, помоћу којих ће се по потреби увести одговарајуће инструкције које се користе при интерпретацији.

```
#define INSTRUCTION_3(name, format, t0, t1, t2)  \
  void name(t0 a0, t1 a1, t2 a2) {               \
    Print(format, Wrap(a0), Wrap(a1), Wrap(a2)); \
  }
```

Пример макроа за инструкцију која има три аргумента

Након тога је имплементирана функција *Print* за уписивање асемблерских инструкција у фајл. Дефинисан је начин записивања регистара, лабела и адреса у МИПС асемблеру.

```
case 'l': {
  Label* label = va_arg(arguments, Label*);
  printf(".L%d", label->position());
  break;
}
```

Пример записивања лабеле

```
case 'r': {
  Register reg = static_cast<Register>(va_arg(arguments, int));
  printf("\$%s", ToString(reg));
  break;
 }
```

Пример записивања регистра

```
void Assembler::PrintAddress(const Address* address) {
  printf("\%d(\$%s)", address->offset(), ToString(address->base()));
}
```

Пример записивања адресе.

На овај начин је омогућено генерисање МИПС асемблерског кода. Следећи корак је генерисање интерпретатора. Интерпретатор генерише асемблерску датотеку чијим превођењем добијамо програм који извршава бајткод на МИПС архитектури. Већина функција у интерпретатору представља имплементацију одговарајућих Дартино процедура.

Најпре су имплементиране специфичности у вези са МИПС позивном конвенцијом(референца). Функција *GeneratePrologue* служи за генерисање асемблерског кода који ће се извршити приликом позива сваке функције. Мора се сачувати садржај регистара S0-S7 на стеку, и такође садржај регистра RA, у ком се налази адреса повратка функције. По повратку из неке функције, позива се функција *GenerateEpilogue* у којој се скида са стека садржај регистара S0-S7 и регистра RA.

При скоку на функцију која се налази ван интерпретатора, мора се позвати функција за поравнање стека. На основу позивне конвенције, неопходно је резервисати 4 места на стеку за аргументе функције (уколико функција коју позивамо позива неку другу функцију). Осим тога, неопходно је сачувати садржај регистра GP. Стек увек мора бити поравнат на 8 (адреса стека мора бити дељива са 8), па се због тога сачува 6 речи уместо 5.

```
void InterpreterGeneratorMIPS::PrepareStack() {
  __ addiu(SP, SP, Immediate(-6 * kWordSize));
  __ sw(GP, Address(SP, 5 * kWordSize));
}
```

По повратку из неке спољне функције, позива се функција *RestoreStack*, која ради инверзну операцију функције *PrepareStack*.

```
void InterpreterGeneratorMIPS::RestoreStack() {
  __ lw(GP, Address(SP, 5 * kWordSize));
  __ addiu(SP, SP, Immediate(6 * kWordSize));
}
```

Након тога настављена је постепена имплементација интерпретатора. Први циљ је био превођење програма који декларише променљиву и иницијализује је на вредност 42, а затим је исписује на стандардни излаз.

```
main() {
  var number = 42;
  print(\${number});
}
```

Направљени су механизми за дебаговање у МИПС и ARM асемблеру који садрже две опције: могућност исписивања редног броја функције интерпретатора која се тренутно извршава и могућност исписивања садржаја задатог регистра. Ти механизми су у процесу развоја били део интерпретатора. Коришћени су како би се извршавањем програма на ARM архитектури добила информација о функцијама које је неопходно имплементирати да би се тај програм извршио на МИПС архитектури. Такође, када дође до грешке и прекида извршавања програма, овај механизам нам омогућава да знамо у којој функцији је дошло до грешке. Више о овом механизму може се прочитати у одељку 5.3.

Након програма који исписује број 42, имплементиран је остатак функција који је неопходан за исписивање поруке „Здраво свете”.

```
main() {
  print("Hello world");
}
```

Затим су имплементиране функције неопходне за превођење програма са основним аритметичким операцијама, рад са низовима, битовски оператори и слично. У наставку следи пример имплементације једне овакве функције.

```
void InterpreterGeneratorMIPS::InvokeAdd(const char* fallback) {
  Label no_overflow;
  LoadLocal(A0, 1);
  __ andi(T0, A0, Immediate(Smi::kTagMask));
  __ B(NEQ, T0, ZR, fallback);
  LoadLocal(A1, 0);
  __ andi(T1, A1, Immediate(Smi::kTagMask));
  __ B(NEQ, T1, ZR, fallback);

  __ xor_(T1, A0, A1);
  __ b(LT, T1, ZR, &no_overflow);
  __ addu(T0, A0, A1);  // Delay-slot.
  __ xor_(T1, T0, A0);
  __ b(LT, T1, ZR, fallback);
  __ Bind(&no_overflow);
  __ move(A0, T0);  // Delay-slot.
  DropNAndSetTop(1, A0);
  Dispatch(kInvokeAddLength);
}
```

Функција која сабира две целобројне вредности. У овој функцији се сабирају два броја, при чему се проверава да ли је дошло до прекорачења. Уколико је било прекорачења, непходно је скочити на део кода који врши опоравак.

Након тога имплементиране су функције које су неопходне да би се исправно извршио програм који исписује поздравну поруку и информације о машини на којој се програм извршава.

```
main() {
  SystemInformation si = sys.info();
  String nodeInformation =
      si.nodeName.isEmpty ? '' : ' running on \${si.nodeName}';
  print('Hello from ${si.operatingSystemName}$nodeInformation.');
}
```

На крају су имплементиране преостале функције, како би се могао покренути постојећи скуп тестова.

5.2 Дартино стек

У оквиру Дартино виртуелне машине се ради са локалним стеком, и за те потребе је искоришћен регистар S2. Све операције са стеком у оквиру интерпретатора се раде над тим стеком, при чему стек процесора користи Дарт виртуелна машина, односно компајлер. Функције за рад са локалним стеком:

```
void InterpreterGeneratorMIPS::Pop(Register reg) {
  __ lw(reg, Address(S2, 0));
  __ addiu(S2, S2, Immediate(1 * kWordSize));
}
```

Скидање садржаја регистра са стека.

```
void InterpreterGeneratorMIPS::Push(Register reg) {
  __ addiu(S2, S2, Immediate(-1 * kWordSize));
  __ sw(reg, Address(S2, 0));
}
```

Чување садржаја регистра на стеку.

5.3 Систем за дебаговање

С обзиром да се функције у интерпретатору извршавају у току превођења кода, а не у току извршавања, било је неопходно да се систем за дебаговање угради у асемблерски код.

```
#define push_asm(reg)                                    \
  assembler()->subi(SP, SP, Immediate(1*kWordSize));\
  assembler()->sw(reg, Address(SP, 0));
```

Макро за чување садржаја регистра на стеку.

```
#define pop_asm(reg)                                     \
  assembler()->lw(reg, Address(SP, 0));                  \
  assembler()->addi(SP, SP, Immediate(1*kWordSize));
```

Макро за скидање садржаја регистра са стека.

```
#define PrintRegister(reg)                          \
  push_asm(GP);                                     \
  push_asm(V0);                                     \
  push_asm(A0);                                     \
  push_asm(A1);                                     \
  push_asm(T9);                                     \
  push_asm(RA);                                     \
  assembler()->la(A0, "print_reg");                 \
  assembler()->move(A1, reg);                       \
  assembler()->lw(T9, "\%call16(printf)(\$gp)");\
  assembler()->jalr(T9);                            \
  assembler()->nop();                               \
  pop_asm(RA);                                      \
  pop_asm(T9);                                      \
  pop_asm(A1);                                      \
  pop_asm(A0);                                      \
  pop_asm(V0);                                      \
  pop_asm(GP);
```

Макро за исписивање садржаја регистра.

Неопходно је сачувати садржај регистара који се користе у макроу, како систем за дебаговање не би утицао на извршавање остатка програма. У A0 се смешта адреса ниске „print_reg”, коју генерише функција *GenerateDebugStrings* док се у A1 сачува садржај регистра који је аргумент макроа. Након тога се скочи на функцију *printf*, при чему се у A0 и A1 налазе аргументи функције.

```
void InterpreterGeneratorMIPS::GenerateDebugStrings() {
  int i;
  char *str = (char *)malloc(10);
  printf("\n\t.data\n");
  for(i=1;i<=255;i++) {
    sprintf(str, "string_\%d", i);
    printf("\%s: .asciiz \"\%d   \"\n", str, i);
  }
  printf("print_reg: .asciiz \"register_value: \%\%x\\n\"\n");
  free(str);
}
```

Функција која генерише ниске које се користе при дебаговању, у сектору података у асемблерској датотеци.

5.4 Тестирање и резултати

Упоредо са имплементацијом интерпретатора, писани су мали тест примери. Примери неких тестова дати су у опису имплементације. Овде ће бити наведени још неки од тестова који су помогли у решавању највећих багова.

Један од проблема који се јавио у току имплементације је функција *signalfd*, која у QEMU емулатору за МИПС није била имплементирана. То је установљено тест примером у коме се налазио следећи позив:

```
int fd = signalfd(-1, &signal_mask, SFD_CLOEXEC);
  if (fd == -1) {
    FATAL1("signalfd failed: %s", strerror(errno));
  }
```

Након тога је направљен тест у програмском језику Ц, у ком се такође користи функција *signalfd*, и преведен за МИПС, како би се утврдило да ли разлог због ког не пролази тест има везе са интерпретатором. При извршавању тог теста јављала се грешка неподржане функције. Када је подршка за функцију додата у оквиру КЕМУ емулатора, тест је прошао.

Приликом тестирања рада са сокетима јавила се грешка при позиву функције *Socket.connect*, односно функције *sys.socket* која се налази у оквиру ње.

```
fd = sys.socket(sys.AF_INET, sys.SOCK_STREAM, 0);
```

Утврђено је да унутар Дартина вредност макроа SOCK_STREAM није прилагођена МИПС архитектури, која користи различите вредности неких системских макроа у односу на остале архитектуре.

Направљено је и неколико тест примера у којима је уочен проблем при позиву функције *setsockopt*, којој су прослеђиване погрешне вредности макроа SOL_SOCKET и SO_REUSEADDR.

```
int _setReuseaddr(int fd) {
    int result =
      sys.setsockopt(fd, sys.SOL_SOCKET, sys.SO_REUSEADDR, FOREIGN_ONE);
    return result;
  }
```

Сличан проблем уочен је при прављењу теста у ком се генерише нова датотека. Проблем је био у макроима који се користе при системском позиву open: O_CREAT, O_APPEND и O_NONBLOCK.

У наставку су делови Дартино датотека (које чине библиотеку за приступ оперативном систему) у којима су исправљене вредности одговарајућих макроа, тако да зависе од архитектуре:

```
  static final bool isMips = sys.info().machine == 'mips';
  int get AF_INET6 => 10;

  int get O_CREAT => isMips ? 256 : 64;
  int get O_TRUNC => 512;
  int get O_APPEND => isMips ? 8 : 1024;
  int get O_NONBLOCK => isMips ? 128 : 2048;
  int get O_CLOEXEC => 524288;

  int get FIONREAD => isMips ? 0x467f : 0x541b;
  int get SOL_SOCKET => isMips ? 65535 : 1;
  int get SO_REUSEADDR => isMips ? 4 : 2;
```

Даотека system_linux.dart.

```
  int get SOCK_STREAM => isMips ? 2 : 1;
  int get SOCK_DGRAM => isMips ? 1 : 2;
```

Датотека system_posix.dart

Исправне вредности за МИПС платформу добијене су читањем одговарајућих датотека у оквиру МИПС тулчејна (енг. *toolchain*).

На крају имплементације, покренут је скуп тестова који је део Дартино пројекта. Тестови се покрећу помоћу пајтон (енг. *python*) скрипте *tools/test.py*.

Неопходно је навести на којој се платформи покрећу тестови. Пошто је за

неке тестове потребно мало више времена од подразумеваног, потребно је навести временско ограничење (енг. *timeout*) од 1200 секунди. Разлог зашто се неки тестови извршавају дуже него на ARM платформи је што на МИПС32P2 не постоје неке инструкције које постоје на ARM архитектури, па је било неопходно имплементирати их помоћу већег броја инструкција. Убрзање би се постигло уколико би нам референтна платформа била МИПС64P6, јер је уведен нови, богатији, скуп инструкција.

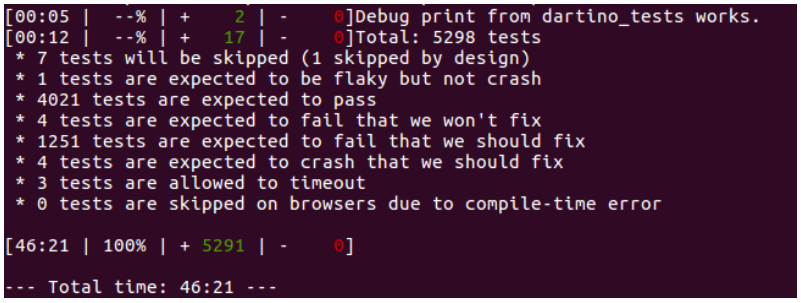
```
tools/test.py -axmips -t1200
```

Команда за покретање тестова на платформи МИПС.

Постојећим скуповима тестова тестиране су следеће карактеристике језика:

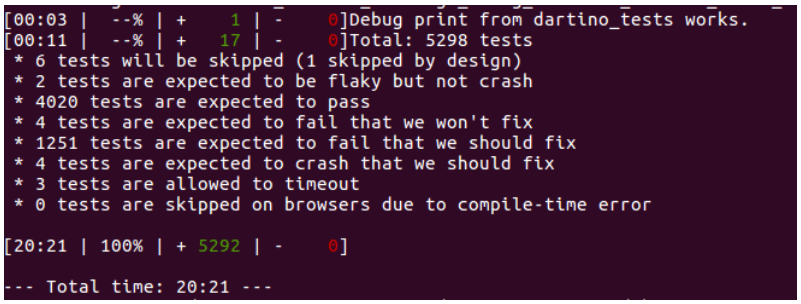
1. Корутине
2. Влакна
3. Изолате
4. Коришћење функција неких других програмских језика
5. Конекција на ХТТПС сервер коришћењем ТЛС протокола
6. Рад са уграђеним библиотекама

У наставку следе резултати тестирања на различитим платформама.

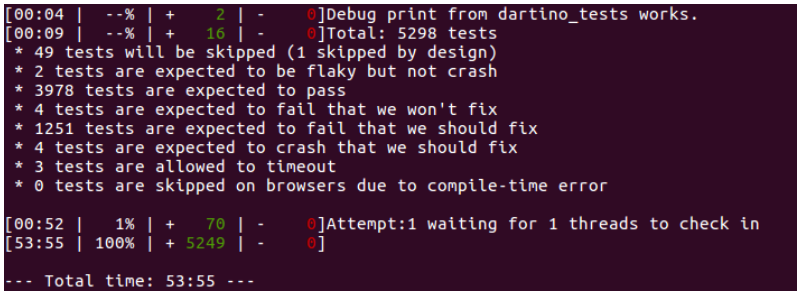


Слика 5.1: Резултати тестирања на платформи МИПС

5.5 Пример апликације у програмском језику Дарт



Слика 5.2: Резултати тестирања на платформи x86-64



Слика 5.3: Резултати тестирања на платформи ARM

Глава 6

Закључак

Библиографија

[1] Yuri Gurevich and Saharon Shelah. Expected computation time for Hamiltonian path problem. *SIAM Journal on Computing*, 16:486–502, 1987.

[2] Petar Petrović and Mika Mikić. Naučni rad. In Miloje Milojević, editor, *Konferencija iz matematike i računarstva*, 2015.