Dodavanje podrške za JIT za arhitekturu MIPS64 u okviru prevodioca LuaJIT

Nemanja Stojanović, Đorđe Kovačević, Stefan Pejić, Petar Jovanović

***Apstrakt*—U ovom radu je opisana realizacija prilagođavanja programskog prevodioca LuaJIT arhitekturi MIPS64 sa i bez koprocesora (jedinice za operacije u aritmetici pokretnog zareza). Detaljno je opisan pristup, način i redosled prilagođavanja interpretera LuaJIT i prevodioca LuaJIT za arhitekturu MIPS64. Izvršen je paralelan prikaz realizacije rešenja u slučaju da arhitektura MIPS64 sadrži koprocesor i slučaj kada ga arhitektura MIPS64 ne poseduje.**

***Ključne reči*—programski prevodilac, interpreter, JIT, LuaJIT, MIPS64, Lua**

I. Uvod

Lua je interpretirani skriptni programski jezik. Trenutno je, kao jedan od najbržih interpretiranih jezika, sve zastupljeniji u upotrebi. Odlikuje ga, između ostalog, jednostavnost, prenosivost i laka ugradivost [1]. Koristi se u mašinskom učenju, programiranju igara, razvoju internet i mobilnih aplikacija i sl.

Poslednjih godina klasični interpreteri postaju sve manje popularni. Iako su interpreteri efikasniji od klasičnih programskih prevodilaca, postojale su neke situacije u kojima su i interpreteri bespotrebno gubili vreme. Naime, interpreter je prevodio liniju izvornog koda i odmah je izvršavao na ciljnog arhitekturi. Ovakav pristup je dovodio do mnogobrojnog prevođenja istog koda, najčešće unutar petlji, što je imalo značajan uticaj na performanse izvršavanja. Dalji razvoj na programskim prevodiocima i rešavanju ovog problema doveo je do realizacije novog pristupa prevođenja nazvanog pravovremeno prevođenje (*just-in-time compilation*), koje kombinuje prednosti klasičnog prevođenja i interpretacije u cilju poboljšavanja perfomansi [2].

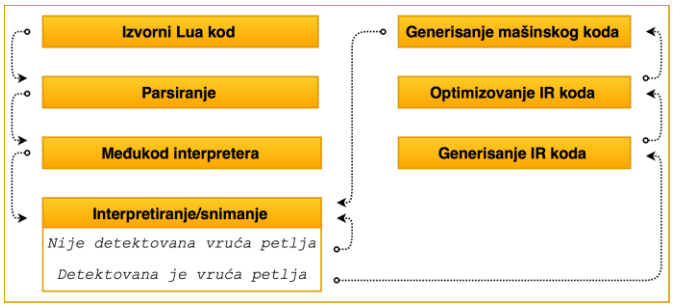
II. LUAJIT

LuaJIT je programski prevodilac tipa JIT (eng. *Just-in-time*) koji se koristi za prevođenje programa napisanih u programskom jeziku Lua. LuaJIT je autorski projekat objavljen pod licencom otvorenog koda MIT-a. Odlikuje ga fleksibilnost, visoke perfomanse i mala potrošnja memorije čime je stekao veliku popularnost u numeričkim i grafičkim simulacijama. [1]

LuaJIT podržava različite arhitekture, međutim, u poslednje vreme pojavila se potreba za proširenjem podrške i na arhitekturu MIPS64 sa i bez jedinice za operacije nad brojevima u pokretnom zarezu. Ovaj rad proističe kao rezultat upravo iz realizacije ove podrške.

LuaJIT je programski prevodilac zasnovan na tragovima izvršavanja. Ovaj tip prevodioca zasniva se na činjenici da se najveći deo vremena izvršavanja programa provede u petljama ponavljajući isti kod veliki broj puta. Osnovni koncept ovakvih prevodilaca je da se proizvede mašinski kod samo za često izvršavane sekvence, dok se ostatak koda interpretira.

Ukoliko prevodilac ustanovi da se nalazi u takvoj petlji (ukoliko brojač izvršavanja istog koda pređe određeni prag), on je detektuje kao vruću petlju (*hot loop*) i kreira trag (*trace*) [2][3][4]. Trag koji predstavlja instrukcije datog segmenta izvornog koda se snima, a od njega se kreira međukod prevodioca, odnosno IR kod (*Intermediate Representation*). Svaki put kada se u nekoj petlji dostigne prag, proverava se da li je taj trag već snimljen, ako jeste, izvršava se odgovarajući preveden kod, a ako nije vrši se snimanje. Algoritam po kojem LuaJIT prevodi izvorni kod prikazan je na slici 1.



Sl. 1. Proces prevođenja Lua koda u okviru LuaJIT-a

III. MIPS

MIPS je arhitektura procesorskog instrukcijskog skupa koja spada u familiju RISC (*Reduced Instruction Set Computing*). [5]. MIPS32 i MIPS64 su arhitekture koje poseduju 32 registra širine 32 bita, odnosno 64 bita. Sve instrukcije arhitekture MIPS su širine 32 bita i u velikoj meri su zajedničke za procesore MIPS32 i MIPS64.

Prilikom rada sa arhitekturom MIPS moraju se poštovati određene konvencije. Postoje par starih i par novih konvencija za pozive (*calling conventions*) za MIPS32 (o32, n32) i za MIPS64 (o64, n64).

Nemanja Stojanović, Istraživačko-razvojni Institut RT-RK, Narodnog fronta 23a, 21000 Novi Sad, Srbija; (e-mail: nemanja.stojanovic@rt-rk,com)
Đorđe Kovačević, Istraživačko-razvojni Institut RT-RK, Narodnog fronta 23a, 21000 Novi Sad, Srbija; (e-mail: djordje.lj.kovacevic@rt-rk.com)
Stefan Pejić, Istraživačko-razvojni Institut RT-RK, Narodnog fronta 23a, 21000 Novi Sad, Srbija; (e-mail: stefan.pejic@rt-rk.com)
Petar Jovanović, Istraživačko-razvojni Institut RT-RK, Narodnog fronta 23a, 21000 Novi Sad, Srbija; (e-mail: petar.jovanović@rt-rk.com)

Procesor MIPS može da sadrži koprocesor za arhitekturu sa pokretnim zarezom. Ovaj opcioni deo procesora MIPS32 i MIPS64 sadrži 32 registra za aritmetiku sa pokretnim zarezom koji su široki 32, odnosno 64 bita.

Prilikom realizacije podrške za JIT za arhitekturu MIPS64, ta podrška je već postojala za MIPS32. Pristup je bio takav da je zbog sličnosti arhitektura MIPS32 i MIPS64, prvenstveno zbog njihovih sličnosti u instrukcijskom skupu, trebalo realizovati rešenje u istim datotekama u okviru izvornog koda LuaJIT-a. Ovo je zahtevalo zadovoljavanje konvencija i posebno obraćanje pažnje na širinu registara.

IV. ABI

Jedna od najbitnijih stavki o kojima je trebalo voditi računa prilikom prilagođavanja prevodioca je prilagođavanje binarne sprege ABI (*Application Binary Interface*). ABI je skup konvencija koje se primenjuju prilikom izvršavanja programa zaključno sa programskim prevodiocem, asemblerom i povezivačem.

ABI čine pravila koja se moraju ispoštovati da bi dva nepovezana segmenta koda u binarnom obliku mogla zajedno da funkcionišu. U ta pravila spadaju pozivne konvencije koje definišu kako će se vršiti pozivanje funkcija, kako se pozvanoj funkciji prosleđuju argumenti, kako se vraćaju rezultati funkcije, kako će se pristupati spoljnoj biblioteci i slično.

Pozivne konvencije za o32 i n64 MIPS ABI za arhitekturu MIPS sa koprocesorom se razlikuju u definisanim registrima koji su namenjeni za argumente i povratne vrednosti. Prilikom realizacije rešenja za MIPS64 bez koprocesora korišćenje ovih registara koprocesora je trebalo onemogućiti, a umesto njih su korišćeni registri centralnog procesora.

(*Floating-Point Registers* – registri koprocesora) i instrukcija namenjenih za rad sa FPR, a sa druge strane u slučaju da MIPS ne sadrži koprocesor morale su se izvršiti pozivi ka spoljnim funkcijama napisanim u C-u.

Pomenute spoljne funkcije su definisane u biblioteci koju pruža GCC koji je korišćen za prevođenje LuaJIT-a za arhitekturu MIPS.

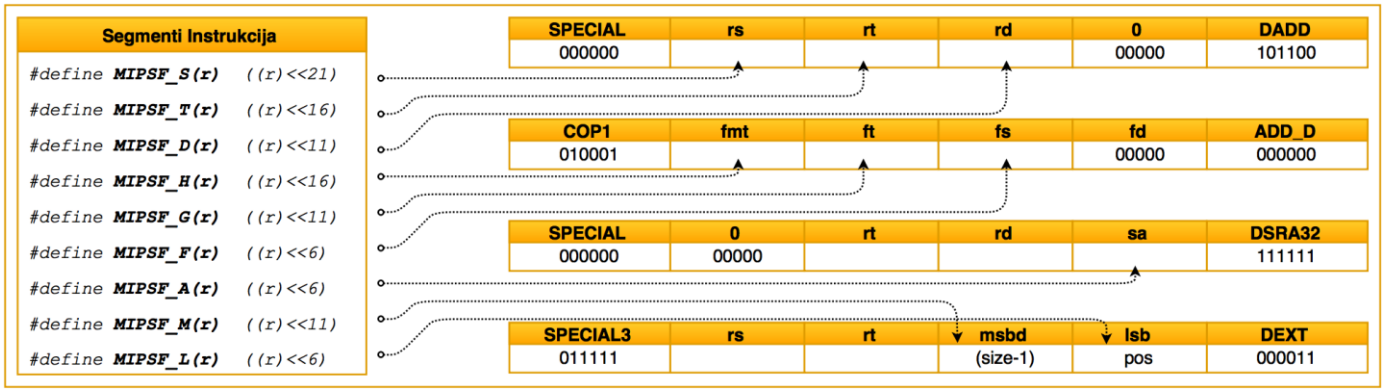
Dalji rad na interpreteru je zahtevao poštovanje pravila u vezi sa stekom, ABI-jem, registrima i ulaznim i izlaznim parametrima funkcija.

Inicijalna izmena je proistekla iz potrebe da se u određenim situacijama koriste instrukcije *sd*/*ld* umesto instrukcija *sw*/*lw*, jer su registri na arhitekturi MIPS64 64-bitni. Ta promena je izazvala grešku, jer instrukcija *ld* nije bila prepoznata od strane asemblera.

Da bi asembler prepoznao neku novu funkciju, ona treba da se doda u listu podržanih instrukcija, a definiše se imenom i heksadecimalnom predstavom koda instrukcije. Dodavanje instrukcija je proces na koji se moralo vratiti povremeno i dodati instrukcija koja je potrebna, a nije definisana, jer, u početku nije bilo potrebno dodati sve nove instrukcije koje će se koristiti.

VI. PROŠIRIVANJE LISTE PODRŽANIH INSTRUKCIJA

Na slici 2 imamo prikaz konstanti uz pomoć kojih se pristupa određenim poljima instrukcija. Moralo se voditi računa koje konstante se koriste prilikom pristupa različitim instrukcijama iz MIPS-ovog instrukcijskog skupa. Na slici 3 prikazano je nekoliko instrukcija u svom hexadecimalnom zapisu i funkcije uz pomoć kojih pristupamo tim instrukcijama. Svako dodavanje instrukcije zahtevalo je njeno



Sl. 2. Prikaz četiri instrukcije i konstante uz pomoć kojih pristupamo poljima tih instrukcija

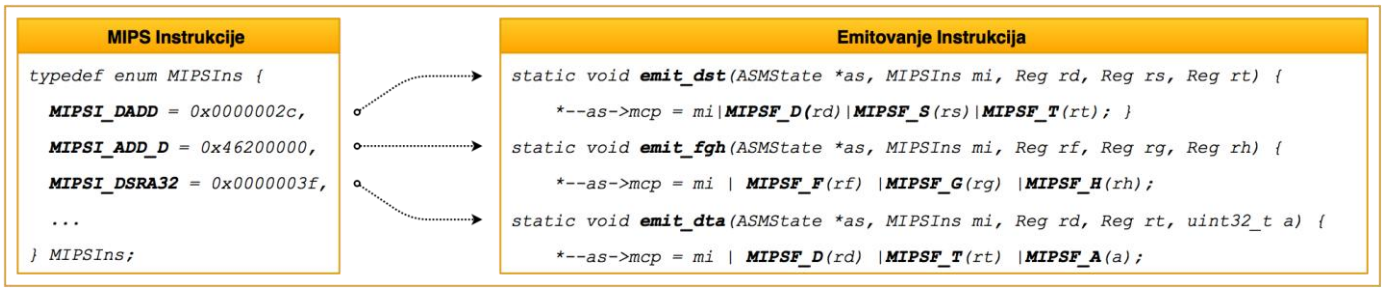
V. PRILAGOĐAVANJE INTERPRETERA

Iako interpreter predstavlja sastavni deo prevodioca LuaJIT, on može da se koristi kao samostalna celina za interpretiranje izvornog koda. Interpreter je realizovan u asemblerskom jeziku ciljne arhitekture, dok je prevođenja tipa JIT realizovano u programskom jeziku C.

Kod interpretera se nalazi u datoteci *vm_mips64.dasc*. Segmenti koda koji su pretrpeli najviše izmena su aritmetičke operacije i operacije poređenja. Ove operacije su za MIPS sa koprocesorom morale biti realizovane korišćenjem FPR

definisanje i u disasembleru. Disasembler se nalazi u okviru *jit* modula i napisan je u programskom jeziku Lua. Na slici 4 imamo prikaz izvornog koda koji predstavlja način na koji su neke od datih funkcija dodate. Kao što može da se vidi, nakon dodavanja instrukcije neophodno je i definisati karaktere koji jedinstveno označavaju koja polja date instrukcije sadrže.

Za realizaciju rešenja bilo je neophodno proširiti listu podržanih instrukcija arhitekture MIPS32 sa još 56 instrukcija.



Sl. 3. Prikaz tri instrukcije u hexadecimalnom zapisu i funkcija za njihovo pozivanje

VII. PREVOĐENJE IR KODA

U izvornom kodu LuaJIT-a trebalo je promeniti, odnosno dodati one funkcije koje prevode dati međukod u sekvencu mašinskih instrukcija. Prilikom realizacije rešenja bilo je neophodno iskoristiti što više postojećeg koda namenjenog podršci arhitekturi MIPS32, a delovi koji nisu kompatibilni napisati za arhitekturu MIPS64. Sav kod kojim je realizovano stvaranje mašinskih instrukcija se nalazi u datotekama za prevođenje međukoda prevodioca (*lj_asm_mips.h*) i emitovanje funkcija (*lj_emit_mips.h*).

Instrukcije

```
local map_special = {
    shift = 0, mask = 63, ..., [44] = "daddDST",...}
local map_special3 = {
    shift = 0, mask = 63, ..., [3] = "dextTSAK",...}
```

Polja Instrukcija

```
for p in gmatch(pat, ".") do
    if p == "D" then x = map_gpr[band(rshift(op, 11), 31)]
elseif p == "T" then x = map_gpr[band(rshift(op, 16), 31)]
elseif p == "S" then x = map_gpr[band(rshift(op, 21), 31)]
elseif p == "K" then x = band(rshift(op, 11), 31) + 1
elseif p == "A" then x = band(rshift(op, 6), 31)
```

Sl. 4. Prikaz izmena u disasembleru

Izmene neophodne kako bi se prilagodio ABI unete su u funkciju *asm_gencall*, koja sadrži kod koji će proizvesti mašinske instrukcije za pozivanje funkcija u C-u. Slične izmene unete su i u funkciju *asm_setupresult*. Ova funkcija pravi instrukcije koje će obaviti premeštanje povratnih vrednosti iz odredišnih registara u željene registre. U slučaju rada sa koprocesorom premeštanje se vršilo iz registra $f0 u željene registre, a u slučaju kada ne postoji koprocesor dodat je segment koji vrši premeštanje iz registara $v0 i $v1 u željene registre.

Značajan deo izmena izvršen je i nad funkcijama zaduženim za čitanje i upis sadržaja na memoriju. U zavisnosti od tipa sadržaja koji se čita, odnosno upisuje postoji instrukcija IR koda. Te instrukcije međukoda se prevode u mašinski kod u funkcijama *asm_ahuvload*, *asm_ahustore*, *asm_sload*. Sve ove funkcije su podlegle izmenama.

Na slici 5 su sa masnim slovima označene izmene u funkciji *asm_add*, jednoj od jednostavnijih funkcija u okviru datoteke *asm_mips.h*.

Funkcija asm_add

```
static void asm_add(ASMState *as, IRIns *ir)
{
  IRType t = ir->t;
#if !(LJ_SOFTFP && LJ_32)
  if (irt_isnum(t)) {
    LJ_SOFTFP ? asm_callid(as, ir, IRCALL_softfp_add) :
                asm_fparith(as, ir, MIPSI_ADD_D);
  } else
#endif
  {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg right, left = ra_hintalloc(as, ir->op1, dest,
                                   RSET_GPR);
    if (irref_isk(ir->op2)) {
      intptr_t k = get_kval(IR(ir->op2));
      if (checki16(k)) {
        emit_tsi(as, (LJ_64 && irt_is64(t)) ?
              MIPSI_DADDIU : MIPSI_ADDIU, dest, left, k);
        return; }}
right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR,
            left));
emit_dst(as, (LJ_64 && irt_is64(t)) ? MIPSI_DADDU :
         MIPSI_ADDU, dest, left, right); }}
```

Sl. 5. Izmene izvršene nad funkcijom *asm_add*

Na slici 6 i slici 7 su prikazane rutina *vm_tointg* i funkcija *asm_tointg* koje vrše konverziju vrednosti dvostruke preciznosti u celobrojnu vrednost na arhitekturi MIPS64 bez

Rutina vm_tointg

```
|//Returns converted value in CARG1 and check result in
|//CARG2,
|->vm_tointg:
|  dsll CARG2, CARG1, 1
|  li TMP1, 1076
|  dsrl TMP0, CARG2, 53
|  dsubu TMP1, TMP1, TMP0
|  sltiu TMP0, TMP1, 54
|  beqz TMP0, >1
|.   dextm CARG2, CARG2, 0, 20
|  dinsu CARG2, TMP0, 21, 21
|  slt TMP0, CARG1, r0
|  dsrlv CARG1, CARG2, TMP1
|  dsubu AT, r0, CARG1
|  movn CARG1, AT, TMP0
|  //Integer check.
|  li AT, 64
|  subu TMP1, AT, TMP1
|  dsllv CARG2, CARG2, TMP1
|  //Range check.
|  sextw TMP0, CARG1
|  xor TMP0, CARG1, TMP0
|  jr ra
|.   movz CARG2, TMP0, CARG2
|1:
|  jr ra
|.   li CARG2, 1
```

Sl. 6. Rutina vm_tointg

koprocesora. Ova konverzija je morala biti u celosti implementirana za arhitekturu MIPS64 bez koprocesora jer se na arhitekturi MIPS32 ona obavljala na potpuno drugačiji način, uz pomoć razdvajanja IR koda, odnosno splitovanja.

Funkcija asm_tointg

```
#if LJ_SOFTFP
static void asm_tointg(ASMState *as, IRIns *ir, Reg left)
 {
   /* The modified regs must match with the *.dasc
      implementation. */
   RegSet drop = RID2RSET(RID_R1)|RID2RSET(REGARG_FIRSTGPR)|
                 RID2RSET(RID_R5)|RID2RSET(RID_R12)|
                 RID2RSET(RID_R13);
   if (ra_hasreg(ir->r)) rset_clear(drop, ir->r);
   ra_evictset(as, drop);
   ra_destreg(as, ir, REGARG_FIRSTGPR);
   asm_guard(as, MIPSI_BNE, RID_R5, RID_ZERO);
   emit_call(as, (void *)lj_ir_callinfo[
             IRCALL_lj_vm_tointg].func, 0);
   if (left != REGARG_FIRSTGPR)
      emit_ds(as, MIPSI_MOVE, REGARG_FIRSTGPR, left);
 }
#endif
```

Sl. 7. Funkija *asm_tointg* za MIPS64 soft float

A. *Problem sa pokazivačima*

U daljem radu pojavio se problem sa pokazivačima. Pretpostavka je bila da su pokazivači 32-bitni tako da je dobavljanje i pohranjivanje podataka vršeno korišćenjem instrukcije *lw* i *sw* koje su morale biti zamenjene sa 64-bitnim instrukcijama za dobavljanje i pohranjivanje, odnosno *ld* i *sd*. Takođe, bilo je potrebno ispraviti sabiranje pokazivača sa nekim pomerajem. Problem se manifestovao i u funkcijama koje generišu sabiranje pokazivača sa pomerajem kao i u odvojenim primerima u radu sa stekom.

B. *Problem sa konstantama*

Jedan od problema koji se pojavio prilikom implementacije rešenja je alociranje registara za konstantu. Podrška za konstante je pretpostavljala da su konstante 32-bitne i nije bilo moguće alocirati jedan registar za 64-bitnu konstantu, iako MIPS64 ima 64-bitne registre. Ovo je ispravljeno korišćenjem tipa koji je veličine 32 ili 64 bita u zavisnosti od arhitekture.

VIII. DUALNA PREDSTAVA BROJEVA

Prilikom realizacije bez podrške za aritmetiku pokretnog zareza bilo je neophodno uvesti podršku za dualno predstavljanje brojeva. U nedostatku koprocesora sve aritmetičke operacije se izvršavaju programski, tj. pozivanjem odgovarajuće funkcije. Međutim, operacije nad celim brojevima se mogu izvršiti u centralnom procesoru mnogo brže, pa bi ovo pozivanje spoljnih funkcija predstavljalo nepotrebno gubljenje vremena. Stoga je bilo potrebno izbeći predstavljanje celih brojeva u formatu sa pokretnim zarezom. Upravo iz ovog razloga je uvedena podrška za dualno predstavljanje brojeva. Ova podrška je omogućena odgovarajućim izmenama u okviru datoteke *lj_arch.h*.

IX. PODRŠKA ZA ARITMETIKU POKRETNOG ZAREZA

Kao što smo već naveli koprocesor je opcioni deo arhitekture MIPS64. Ukoliko bi se LuaJIT preveden za rad na MIPS64 sa koprocesorom pokrenuo na MIPS64 koji nema koprocesor, tada bi koprocesorske instrukcije mogle da se izvrše jedino ako operativni sistem ima podršku za njihovim emuliranjem. Stoga je bilo neophodno programski realizovati ove operacije u slučaju ne postojanja koprocesora.

Za učitavanje vrednosti na arhitekturi MIPS64 sa koprocesorom iz memorije u registre i obrnuto koriste se posebne instrukcije: *lwc1/swc1* za vrednosti u jednostrukoj preciznosti i *ldc1/sdc1* za vrednosti u dvostrukoj preciznosti. Za premeštanje sadržaja iz registra koprocesora u registar opšte namene i obrnuto koriste se instrukcije *mtc1* i *mfc1*. Za pretvaranje brojnih tipova na arhitekturi sa koprocesorom koristi se instrukcija *cvt.x.y*. Uslovna grananja se baziraju na poređenju realnih brojeva. Na osnovu rezultata poređenja se izvršava ili ne izvršava grananje. Poređenja su se izvršavala sa instrukcijama *c.x.s* za poređenja u jednostrukoj preciznosti i sa *c.x.d* u dvostrukoj preciznosti.

Aritmetičke operacije u slučaju kada je onemogućeno korišćenje koprocesora je bilo potrebno izvršiti pozivanjem spoljne funkcije napisane u C-u. Samim tim bilo je neophodno smestiti operande u registre opšte namene. Odgovarajuće C funkcije su definisane u GCC-u (libgcc.a).

Za poređenja korišćena je funkcija *__leadf2*. Funkcija će vratiti rezultat 0 ako su operandi jednaki, -1 ako je prvi operand manji od drugog, 1 ako je prvi veći od drugog i 2 ako jedan od operanada nije numerički tip (NaN).

Na slici 8 imamo definisane funkcije i instrukcije koje su korišćene prilikom izrade rešenja na MIPS-sa procesorom i njihove zamene u slučaju da procesor nije sadržan.

Na slici 9 dat je lua test, njegov IR kod i generisani mašinski kod u oba slučaja, prilikom rada sa koprocesorom i bez njega. Na slici možemo da vidimo pozive spoljnih C funkcija i par instrukcija koje to isto omogućavaju vrlo brzo u

Instrukcija	Funkcija	Zamena
ldc1/sdc1	Čitanje/pisanje sadržaja u FPR	**ld/sd** u GPR
add.d/sub.d	Sabiranje/oduzimanje brojeva u dvostrukoj preciznosti	**__adddf3/__subdf3**
mul.d/div.d	Množenje/Deljenje brojeva u dvostrukoj preciznosti	**__muldf3/__divdf3**
cvt.w.d/cvt.d.w	Pretvaranje iz int u double i iz double u int	**__fixdfsi/__floatsidf**
c.olt.s/c.olt.d	Instrukcija za poređenje	**__ledf2**
movt/movf	Premeštanje sadržaja u zavisnosti od rezultata poređenja	**mov**
bc1f/bc1t	Uslovno grananje u zavisnosti od rezultata poređenja	**b/beq/bne/beqz/bnez**

Sl. 8. Prikaz instrukcija koje se koriste na arhitekturi MIPS64 sa koprocesorom i njihove zamene u slučaju da koprocesor ne postoji

slučaju rada sa koprocesorom. Funkcije *softfp_mul*, *softfp_i2d*, *softfp_div*, *softfp_add* predstavljaju nazive za funkije *__muldf3*, *__floatisdf*, *__divdf3*, *__adddf3* koje se koriste za konverzije iz celobrojnih vrednosti u brojeve dvostruke preciznosti, i za sabiranje, množenje i deljenje.

Test.lua

```
local a = {}
a[1] = 0.5
for i = 1, 58 do
  a[1] = a[1] * a[1] + i / i
end
print (a[1])
```

Part of Trace

```
---- TRACE 1 IR
...
0015 ------ LOOP ------------
0016 num MUL 0011 0011
0017 num CONV 0013 num.int
0018 num DIV 0017 0017
0019 +num ADD 0018 0016
0020 num ASTORE 0006 0019
0021 +int ADD 0013 +1
...
```

MIPS64 soft float

```
...
->LOOP:
...
move r5,r4
jal  0x12011c4a0->softfp_mul
nop
sd   r2,24(sp)
li   r25,1
dsll r25,r25,16
ori  r25,r25,8209
dsll r25,r25,16
ori  r25,r25,53008
move r4, r22
jal  0x12011cf10->softfp_i2d
nop
move r4,r2
li   r25,1
dsll r25,r25,16
ori  r25,r25,8209
dsll r25,r25,16
ori  r25,r25,48592
move r5,r4
jal 0x12011bddo->softfp_div
nop
move r4,r2
li   r25,1
dsll r25,r25,16
ori  r25,r25,8209
dsll r25,r25,16
ori  r25,r25,47136
ld   r5,24(sp)
jal  0x12011bddo->softfp_add
...
----TRACE 1 stop -> loop
```

MIPS64 hard float

```
...
->LOOP:
mul.d   f30,f31,f31
mtc1    r23,f31
cvt.d.w f31,f31
div.d   f31,f31,f31
add.d   f31,f31,f30
sdc1    f31,0(r1)
addiu   r23,r23,1
slti    ra,r23,59
bnez    ra,0x12abeffc8
li      ra,3
---- TRACE 1 stop -> loop
```

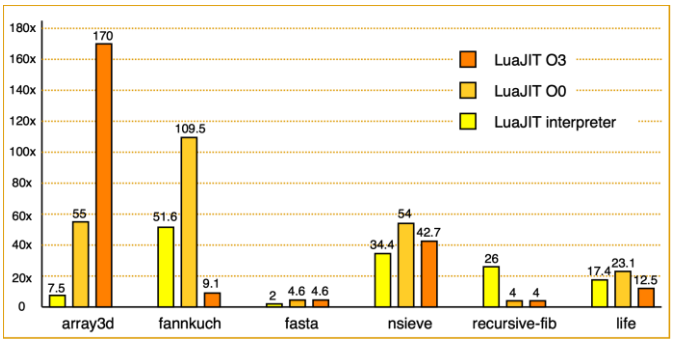
Sl. 9. Paralelni prikaz disasembliranog koda

X. ISPITIVANJE

Ova ispitivanja su vršena na mašini sa procesorom Cavium Octeon II V0.1. Za prevođenje i povezivanje Lue i LuaJITa korišćen je prevodilac GCC 4.8.4 i skup alata (*toolchain*) za MIPS (Sourcery CodeBench Lite 2014.05-27). Izvršne datoteke su bez simbola za ispravljanje grešaka, nivo optimizacije je onaj koji je definisan u datotekama *Makefile* projekata Lua i LuaJIT.

Na slici 10 imamo prikazan odnos izvršavanja šest benchmark testova između LuaJIT-a sa uključenim različitim opcijama i Lua interpretera. Lua interpreter na slici ima vrednost 1. Razliku između LuaJITa sa uključenim optimizacijama JIT koda O1, O2 i O3 možemo da vidimo na slici 11.[7][8][9]

Brzina izvršavanja varira među LuaJIT interpreterom i LuaJIT-om sa uključenim osnovnim i svim optimizacijama, međutim brzina u odnosu na klasičan interpreter je očigledna. Najveće ubrzanje možemo da primetimo u realizaciji *array3d.lua* testa, kada LuaJIT dostiže brzinu izvršavanja od stotinu i sedamdeset puta veću od Lua interpretera. Značajno je ubrzanje u radu sa 3D matricama i značajno je ubrzanje



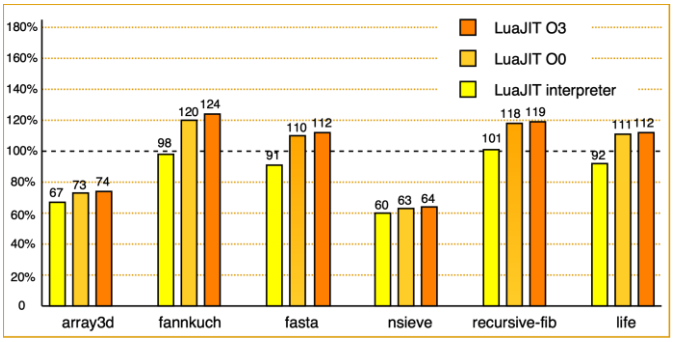
Sl. 10. Prikaz brzine izvršavanja aplikacija u odnosu na Lua interpreter

algoritama koji se često iterativno pokreću mnogo puta, kao što su algoritam života (*life.lua*) ili generisanje fraktala (*mandelbrot.lua*).[10][11]

Flag	O1	O2	O3	Optimizacija
fold	○	○	○	Constant Folding, Simplification
cse	○	○	○	Common-Subexpression Elimination
dce	○	○	○	Dead-Code Elimination
narrow		○	○	Narrowing of numbers to integers
loop		○	○	Loop Optimization (code hoisting)
fwd			○	Load Forwarding and Store Forwarding
dse			○	Dead-Store Elimination
abc			○	Array Bound Check Elimination
sink			○	Allocation/Store Sinking
fuse			○	Fusion of operands into instructions

Sl. 11. Optimizacione metode koje se izvršavaju

Na slici 12 imamo prikaz odnosa upotrebe memorije na MIPS64 arhitekturi sa koprocesorom koristeći klasičan Lua interpreter i LuaJIT. Lua interpreter na slici ima vrednosti 100%. Očekivana vrednost upotrebe memorije za izvršavanje lua aplikacije je niža od klasičnog interpretera. Očigledno je i manja upotreba memorije na skoro svim testovima korišćenjem LuaJIT interpretera u odnosu na Lua interpreter.



Sl. 12. Prikaz upotrebe memorije u odnosu na Lua interpreter

Iz rezultata se može zaključiti da je stabilnost LuaJIT interpretera veoma dobra, kao i brzina izvršavanja

XI. Zaključak

U ovom radu je opisano prilagođavanje programskog prevodioca LuaJIT arhitekturi MIPS64 sa i bez podrške za operacije u aritmetici pokretnog zareza. Rešenje je realizovano za MIPS64 sa koprocesorom, dok je prilagođavanje LuaJIT za MIPS64 bez koprocesora još uvek u procesu prilagođavanja.

Iz činjenice da se sve brojne vrednosti isto predstavljaju, što je predstavljalo nepotrebno gubljenje vremena, obezbedila se dualna predstava brojeva. Time je izvršavanje operacija nad celobrojnim vrednostima postalo efikasnije, a samim tim su poboljšane performanse LuaJIT-a prilagođenim za arhitekturu MIPS64 bez koprocesora.

Zahvalnica

Ovaj rad je delimično finansiran od strane Ministarstva za nauku I tehnologiju Republike Srbije, na projektu tehnološkog razvoja broj: III44009-1

Literatura

[1] Lerusalimschy, Roberto, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. "Lua-an extensible extension language." *Softw., Pract. Exper*. 26.6 (1996): 635-652.

[2] Schilling, Thomas. *Trace-based Just-in-time Compilation for Lazy Functional Programming Languages.* Diss. PhD thesis, University of Kent, 2013.

[3] Bolz, Carl Friedrich, et al. "Tracing the meta-level: PyPy's tracing JIT compiler." *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 2009.

[4] Aycock, John. "A brief history of just-in-time." *ACM Computing Surveys (CSUR)* 35.2 (2003): 97-113.

[5] Sweetman, Dominic. *See MIPS run*. Morgan Kaufmann, 2010.

[6] Matz, Michael, et al. "System V application binary interface." *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013).

[7] Pall, Mike. "The LuaJIT project." *Web site: http://luajit. org* (2008).

[8] Goss, Clinton F. "Machine Code Optimization-Improving Executable Object Code." *arXiv preprint arXiv:1308.4815* (2013).

[9] Padua, David A., and Michael J. Wolfe. "Advanced compiler optimizations for supercomputers." *Communications of the ACM* 29.12 (1986): 1184-1201.

[10] Conway, John. "The game of life." *Scientific American* 223.4 (1970): 4.

[11] Mandelbrot, Benoit B., and Roberto Pignoni. "The fractal geometry of nature." (1983).

Abstract

In this paper we have described porting of LuaJIT to MIPS64 architecture with or without the FPU. Porting of LuaJIT interpreter and LuaJIT compiler have been discussed in detail. Realization of solution for MIPS64 with or without the coprocessor has been done paralelly.

Adding support for MIPS64 in LuaJIT

Nemanja Stojanović, Đorđe Kovačević, Stefan Pejić, Petar Jovanović