Dodavanje podrške za FPXX konvenciju arhitekture MIPS32 unutar alata za dinamičku analizu binarnog koda Velgrind

Aleksandra Karadžić, Aleksandar Rikalo, Tamara Vlahović, Petar Jovanović *RT-RK*

Computer Based Systems, Novi Sad, Srbija

***Apstrakt* —Permanentan razvoj MIPS arhitekture donosi pored novih instrukcija nove standarde koje treba da zadovolje programski prevodioci. Jedna od inovacija jeste FPXX konvencija, koja pomaže u migraciji sa sistema sa 32-bitnim jedinicama za rad sa brojevima u pokrentom zarezu na one sa 64-bitnim jedinicama. Ovaj rad opisuje izmene načinjene u alatu za dinamičku analizu binarnog koda Velgrind, s ciljem dodavanja podrške za konvenciju FPXX.**

Ključne reči—Velgrind, MIPS, FPXX, FPU

I. Uvod

Tridesetdvobitne MIPS platforme podržavaju postojanje kako 32-bitnih, tako i 64-bitnih jedinica za rad u pokretnom zarezu. Ova dva tipa koprocesora nisu potpuno kompatibilna. Pored širine registara i dopunjenog instrukcijskog seta, kod 64-bitnih jedinica neke instrukcije promenile su ponašanje. Ovo znači da se u trenutku prevođenja programa, ukoliko se želi optimalan mašinski kod, mora opredeliti za jednu od dve opcije, tj. definisati ciljnu platformu.

Savremeni trendovi teže ka tome da nove platforme imaju 64-bitne jedinice za rad u pokretnom zarezu koje mogu da rade i u režimu kompatibilnom sa 32-bitnim jedinicama, dok su starije uglavnom opremljene 32-bitnom verzijom. Problemi sa kompatibilnošću se, međutim, ovde ne završavaju. Većina korisničkih programa koristi dinamički učitavač i deljene bibilioteke koje, opet, imaju definisan sopstveni režim rada u pokretnom zarezu.

FPXX konvencija je dodatak MIPS O32 ABI skupu pravila i definiše uslove koje treba da zadovoljava mašinski program kako bi se korektno izvršavao nezavisno od režima u kome radi jedinica za operacije sa pokretnim zarezom. Programski kod koji poštuje ovu konvenciju praktično koristi podskup instrukcija koje su zajedničke za oba režima i kao takav on je suboptimalan, ali je pogodan za kombinovanje sa kodom prevedenim za bilo koji režim rada u pokretnom zarezu. Ideja je da deljene biblioteke i korisnički programi koji treba da budu portabilni, budu prevedeni u skladu sa FPXX konvencijom.

Velgrind je skup alata za dinamičku binarnu analizu koda. Postoje Velgrind alati koji mogu automatski da detektuju probleme sa memorijom, procesima, kao i da izvrše optimizaciju samog koda. Pored toga, Velgrind se može koristiti i kao alat za pravljenje novih alata. Velgrind distribucija trenutno broji sledeće alate: detektor memorijskih grešaka, detektor grešaka niti, optimizator skrivene memorije i skokova, generator grafa skrivene memorije i predikcije skoka i optimizator korišćenja memorije. Podržane su sledeće arhitekture: X86/Linux, AMD64/Linux, ARM/Linux, MIPS32/Linux, MIPS64/Linux, TILEGX/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (od 2.3.x), ARM64/Android, X86/Android (od 4.0), MIPS32/Android, X86/Darwin, AMD64/Darwin (Mac OS X 10.10, sa inicijalnom podrškom za 10.11).

Nameće se pitanje da li Velgrind može da radi ako se prevodi u skladu sa FPXX konvencijom, kao i da vrši analizu programa koji su prevedeni u skladu sa istom.

Iz tog razloga, u ovom radu će biti opisano dodavanje podrške za FPXX u alatu Velgrind.

II. Analiza rada Velgrinda

Osnovni princip rada Velgrinda je zasnovan na prevođenje mašinskog izvršnog koda, blok po blok, u tkz. IR kod, i suprotno.

Velgrind prevodi blok koda na zahtev. Da bi preveo blok koda Velgrind prati instrukcije dok se ne zadovolji jedan od sledećih uslova:

1. Stiglo se do ograničenja u broju instrukcija (oko 50 zavisno od arhitekture);
2. Stiglo se do uslovnog skoka;
3. Stiglo se do skoka sa nepoznatom destinacijom;

Postoji osam translacionih faza. Sve sem faze instrumentacije koju obavlja alat, obavlja jezgro Velgrinda. Određene faze su zavisne od arhitekture.

Faza 1. **Disasembliranje** – Ova faza je zavisna od arhitekture. Mašinski kod se konvertuje u stablo IR reprezantacije. Svaka instrukcija se nezavisno disasemblira u jedan ili više iskaza. Ovi iskazi u potpunosti ažuriraju odgovarajuće simulirane registre.

Faza 2. **Optimizacija 1** – Prva faza optimizacije linearizuje IR reprezantaciju. Primenjuju se određene standardne optimizacije programskih prevodilaca kao što su uklanjanje redudantnog koda, eliminacija podizraza, jednostavno odmotavanje petlji i sl.

Faza 3. **Instrumentacija** – Blok kod u IR reprezentaciji se prosleđuje alatu, koji može proizvoljno da ga transformiše. Prilikom instrumentacije alat u zadati blok dodaje dodatne IR operacije, kojima proverava ispravnost rada programa.

Faza 4. **Optimizacija 2** – Ovo je druga faza optimizacije. Ona je jednostavnija od prve i uključuje množenje konstatni i uklanjanje mrtvog koda.

Faza 5. **Gradnja stabla** – Linearizovana IR reprezentacija se konvertuje natrag u stablo radi lakšeg izbora instrukcija.

Faza 6. **Odabir instrukcija** – Ova faza je zavisna od arhitekture. Stablo IR reprezentacija se konvertuje u listu instrukcija koje koriste virtualne registre.

Faza 7. **Alokacija registara** – Virtualni registri se zamenjuju stvarnim. Po potrebi se uvode prebacivanje u memoriju. Nezavisna je za platformu, koristi poziv funkcija koje nalaze iz kojih se registara vrši čitanje i u koje se vrši upis.

Faza 8. **Asembliranje** – Ova faza je zavisna od arhitekture. Izabrane instrukcije se enkoduju na odgovarajući način i smeštaju u blok memorije[3].

Da bi Velgrindovi alati mogli da vrše svoju funkciju potrebno je prvo izvršni kod tranformisati u opštiju IR reprezentaciju. IR reprezentaciju čine iskazi sastavljeni od stabala izraza. Da bi se jedna instrukcija mogla tranformisati potrebno je prepoznati koje operacije izvršava, koje registre koristi, njene neposredne vrednsoti i memoriju koju koristi.

III. MIPS ABI i režimi jedinica za rad sa pokrentim zarezom

MIPS ABI je menjan tokom vremena kako se menjala arhitektura. Promene koje su nastale u arhitekturi zahtevale su da se preispita stanje O32 ABI i proceni da li postoji mogućnost da se napravi ABI koji bi bio kompaktibilniji sa tadašnjim i svim budućim unapređivanjima arhitekture. Tri glavna razloga za proširivanje O32 ABI-ja su bila uvođenje MSA ASE, želja da se iskoristi FR=1 režim FPU-a i MIPS32r6 arhitektura koja podržava samo FR=1 režim.

- FR = 0 je režim rada FPU sa 32 32-bitna registra. Registri su numerisani od $f0 do $f31. Parovi parnih i neparnih registara se koriste za formiranje 64-bitnih podataka. Operacije sa duplom preciznošću ne mogu da se izvršavaju ukoliko se koriste neparni registri.
- FR = 1 je režim rada FPU sa 32 64-bitna registra. Registri su numerisani od $f0 do $f31 i svaki registar se može koristiti i za operacije sa preciznošću od 32 bita i za operacije sa preznošću od 64-bita.
- FRE = 1 je režim hardvera koji je uveden od MIPS32r5. Ovaj režim se koristi u konjukciji sa FR = 1 režimom, FPU ima 32 64-bitna registara ali je ponašanje pojedinih instrukcija kao u FR = 0 režimu. Operacije sa 64-bitnim ili širim formatima se izvršavaju na isti način kao da se izvršavaju u FR = 1 režimu, ali 32-bitni formati imaju ponašanje kao u FR = 0 režimu. Posebna karakteristika FRE režima se vrti oko rukovanja registara sa preciznošću od 32 bita i ponašanje posebno sa neparnim registrima. Da bi se FP32 režim izvršavao korektno kada se koriste neparni registri mora da se ažurira viši 32-bitni deo parnog 64-bitnog registra, takođe ažuriranje parnih 64-bitnih registra ima za posledicu ažuriranje neparnih 32-bitnih registara. FRE režim ovo dostiže tako što preusmerava čitanje i pisanje sa neparnih registara na gornjih 32-bita parnih registara.

IV. Velgrind i implementacija podrške za režime MIPS jedinica za rad sa pokretnim zarezom

Sam korak disasembliranja u Velgrindu prati implementacija obrade režima MIPS jedinice za rad u pokretnom zarezu u jezgru Linuksa, tj. simulira njegov rad. Prilikom imlpentriranja FPXX režima istraživan je rad jezgra i implementacija u jezgru, pa je na osnovu toga implementirana odgovarajuća podrška u Velgrindu. Deo čitanja ELF zaglavlja na osnovu koga se utvrđuje u kom režimu radi program, implementiran je jako slično kao u jezgru Linuksa.

Ispitivanje zaglavlja programa se vrši radi utvrđivanja ispravnost i/ili podobnost za sistem. Funkcija koja ovo utvrđuje poziva se po jednom za učitan ELF program i njegov interpreter[9].

Tabela 1
Opcije za prevođenje programa i vrednosti promeneljive FP_ABI

Options	FP_ABI
-mabi=32 -mfp32	1
-mabi=64 -mfp64	1
-msingle-float	2
-msoft-float	3
-mabi=32 -mfpxx	5
-mabi=32 -mfp64 -modd-spreg	6
-mabi=32 -mfp64 -mno-odd-spreg	7

U Tabeli 1 u koloni Options su predstavljene opcije sa kojima se prevodi program, dok u koloni FP_ABI su vrednosti koje se tom prilikom upisuju u odgovarajuće polje .MIPS.abiflags sekcije objektnog fajla. Prilikom odlučivanja u kom režimu će raditi program, jezgro čita vrednost FP_ABI iz samog programa kao i iz interpretera ukoliko se radi o dinamički provedenom programu. Na osnovu te dve vrednosti se u jezgru odlučuje u kom režimu će program raditi.

Na primer ako je program preveden sa opcijama -mabi=32 -mfp32 (FP_ABI = 1), a interpreter sa opcijama -mabi=32 -mfpxx (FP_ABI = 5) proces će započeti rad u režimu FR = 0. Ako je program preveden sa opcijama -mabi=32 -mfp64 -modd-spreg (FP_ABI = 6), a interpreter sa opcijama -mabi=32 -mfpxx (FP_ABI = 5) proces će započeti rad u režimu FR = 1.

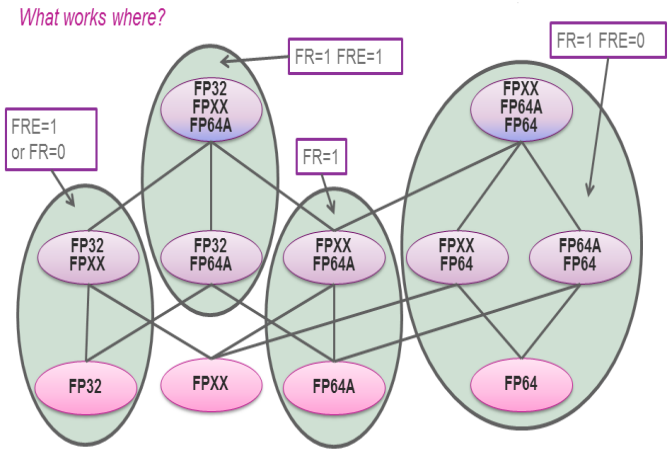
Na Sl 1. je prikazan graf koji oslikava sve podržane kombinacije ABI-ja i hardverskog režima koju svaka kombinacija zahteva. Rubovi grafa predstavljaju dodavanje ili uklanjanje varijantu ABI-ja iz ili u proces. Zadatak interpretera je da:

1. Dozvoli ili zabrani učitavanje novog objekta koji zavisi od trenutno učitanih objekata i podrške hardvera.
2. Menja režim procesora, tako da se sav učitan kod izvršava ispravno.

Velgrind kao virtuelna mašina radi u jednom FPU režimu, ako je preveden sa opcijom -mfpxx onda će raditi u

režimu koji jezgro odredi. S druge strane on emulira jedinicu za operacije sa pokrentim zarezom, s toga može da emulira drugačiji režim rada. Zbog toga je bilo potrebno implementirati algoritam za izbor FPU režima kao što je to urađeno u jezgru.

Trenutno Velgrind ne podržava FRE režim i nema podršku za FR1 režim na sistemima koji rade u FR0 režimu, tako da u ovim slučajevima smo primorani da odbacimo ELF zaglavnje.



Sl 1. Graf kompatiblinosti

V. Presretanje sistemskog poziva

Aplikacioni programi komuniciraju sa operativnim sistemom pomoću sistemskih poziva.

Sistemski pozivi se realiziju pomoću sistema prekida: korisnički program postavlja parametre sistemskog poziva na određene memorijske lokacije ili registre procesora, inicira prekid, zatim operativni sistem preuzima kontrolu, preuzima parametre, izvršava tražene radnje, rezultat stavlja na određene memorijske lokacije ili u registre i naposletku vraća kontrolu korisničkom programu.

U određenim situacijama Velgrind mora da presretne sistemske pozive i sam ih obrađuje. Potreba za takvim slučajem se javila i prilikom implementacije ovog rešenja. Presretanje sistemskog poziva prctl() u Velgrindu bilo je neophodno da bi podrška za FPXX režim bila potpuna.

Sistemski poziv prctl() se poziva sa prvim argumentom koji govori šta treba da se radi, dok ostali zavise umnogome od prvog argumenta. U implementaciji koja je odrađena veliki značaj su imali argumenti PR_SET_FP_MODE i PR_GET_FP_MODE. Prctl() sistemski poziv može da kontroliše trenutno stanje FPU registarski režim - trenutni režim se može pregledati i novi režim može biti postavljen. Prctl() sistemski poziv može da promeni režim rada svih niti koje su u tom trenutku aktivne. Da bi se sva nabrojana funkcionalnost obezbedila, u jezgru Linuksa su definisane nove moguće vrednosti za prvi argument ovog sistemskog poziva. Pregled tih novih vrednosti se vidi u sledećem segmentu koda:

```
#define PR_SET_FP_MODE 45
#define PR_GET_FP_MODE 46
#define PR_FP_MODE_FR  (1 << 0)
#define PR_FP_MODE_FRE (1 << 1)
```

prctl(PR_SET_FP_MODE, mode) će promeniti FP režim specificiran drugim argumentom *mode*, koji je ustvari kombinacija PR_FP_MODE_* bitova. Sve niti koje su aktivne u tom trenutku će promeniti svoj režim rada.

mode = prclt(PR_GET_FP_MODE) vraća trenutni FP režim izvršavanja.

Opcije PR_SET_FP_MODE i PR_GET_FP_MODE su implementirane u jezgru počevši od verzije 4.0, a sva jezgra u kojima ove opcije nisi implementirane vratiće -1 u slučaju njihovog pozivanja[7].

VI. Testiranje rešenja

U početnim fazama razvoja testirano je samo pravilno prepoznavanje FP režima u kom radi program. Ovo je rađeno pomoću trivijalnog programa koji je prevođen na različite načine.

Kasnije, kako je razvoj privođen kraju, testiranje je vršeno pomoću testa koji tokom svog izvršavanja menja režim rada. Taj test je sada jedan od testova, kojim se svakodnevo proverava ispravnost rada Velgrinda.

Detekcija režima u kom radi program se obavlja na sledeći način:

```
static int get_fp_mode(void) {

  unsigned long long result = 0;

  __asm__ volatile(

    ".set push\n\t"

    ".set noreorder\n\t"

    ".set oddspreg\n\t"

    "lui $t0, 0x3FF0\n\t"

    "ldc1 $f0, %0\n\t"

    "mtc1 $t0, $f1\n\t"

    "sdc1 $f0, %0\n\t"

    ".set pop\n\t"

    : "+m"(result)

    :

    : "t0", "$f0", "$f1", "memory");


  return (result != 0x3FF0000000000000ull);

}
```

U slučaju da test radi u režimu FP64, menjamo na režim FP32, proverava se da li je režim uspešno promenje i vraćamo ga na staro.

```
/* FP64 */

  if (fr_prctl == 1) {



    /* Change mode to FP32 */

    if (prctl(PR_SET_FP_MODE, 0) != 0) {

      fatal_error("prctl(PR_SET_FP_MODE, 0) fails.");

    }



    test(&fr_prctl, &fr_detected);



    /* Change back FP mode */

    if (prctl(PR_SET_FP_MODE, 1) != 0) {

      fatal_error("prctl(PR_SET_FP_MODE, 1) fails.");

    }
```

VII. Tekući status i buduće aktivnosti

Izvorni kod rešenja opisan u ovom radu je integrisan u relevatne repozitorujume izvornog koda Velgrinda. Samim tim, potvrđena je validnost, tehnička korektnost i relevantnost rešenja – ono je prošlo rigorozne provere od strane osoba zaduženih za arhitekturu, integraciju i održavanje ovog projekta.

Buduće aktivnosti obuhvataju potpunu implementaciju hibridnog režima FRE. Ovo će biti implementirano nakon dodavanja podrške za mips32/64r6.

VIII. Zaključak

U ovom radu opisan je ukratko način rada alata Velgrind, kao i režim FPXX. Prikazan je način implementacije FPXX režima u Velgrindu, koja predstavlja značajno proširenje podrške Velgrinda za MIPS procesore. Takođe su opisane metode tesitranja i planovi za proširenje postojećeg rešenja.

IX. Zahvalnica

Ovaj rad je delimično finansiran od strane Ministarstva za prosvetu, nauku i tehnološki razvoj Republike Srbije, na projektu broj: III_044009_1.

X. Literatura

[1] See MIPS Run, Sweetman, Dominic. 2nd ed. San Francisco, Calif.: Morgan Kaufmann Publishers/Elsevier, 2007

[2] Valgrind's Tool Suite dostupno na http://valgrind.org/info/tools.html

[3] Nicholas Nethercote , Julian Seward, „Valgrind: a framework for heavyweight dynamic binary instrumentation“, Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, June 10 - 13, 2007, San Diego, California, USA

[4] Dejan Jevtić, „Prilagođavanje alata za dinamičku analizu programskog koda Velgrind za arhitekturu MIPS“, master rad, Elektrotehnika i računarstvo, Fakultet tehničkih nauka, Novi Sad, Srbija, 2011

[5] Ashley W Brown, Paul H J Kelly, Wayne Luk, Profiling floating point value ranges for reconfigurable implemenation dostupno na http://valgrind.org/docs/floatwatch2007.pdf

[6] Tamara Vlahović „Proširavanje alata za dinamičku analizu koda Velgrind na instukcijski skup MIPS MSA”, diplomski rad, Elektrotehnički fakultet, Begorad, Srbija 2016

[7] GIT repozitorijum koji sadrži kod Linuksovog jezgra https://github.com/torvalds/linux

[8] SVN repozitorijum koji sadrži kod Velgrinda svn://svn.valgrind.org/valgrind/trunk

[9] MIPS O32 ABI – FR0 and FR1 interliniking dostupno na https://dmz-portal.mips.com/wiki/MIPS_O32_ABI_-_FR0_and_FR1_Interlinking#Appendix_C._FPU_hardware_modes