Dekodiranje instrukcija u softverskim emulatorima mikroprocesora

Samir Ribić
Univerzitet u Sarajevu
Elektrotehnički fakultet
Sarajevo, Bosna i Hercegovina
Samir.ribic@etf.unsa.ba

Sadržaj—**Dekodiranje instrukcija je operacija koja se izvodi prilikom izvršavanja softverskih emulatora različitih arhitektura izuzetno često. S obzirom na vremensku kritičnost date operacije, toj operaciji treba posvetiti posebnu pažnju. Rad prikazuje primjere kako se dekodiraju instrukcije emuliranih procesora na različitim domaćinskim platformama uz diskusije prednosti i nedostataka pojedinih tehnika.**

Ključne riječi-emulacija; dekodiranje; procesori

I. UVOD

Emulacija je proces u kome se jedan računarski sistem oponaša drugim računarskim sistemom na način da se izvršavaju mašinske instrukcije oponašanog računara ekvivalentnim skupom instrukcija domaćinskog računara. Ima više načina za izvršavanje softvera namijenjenog drugim računarskim sistemima na domaćinskom računaru, kao što su virtualizacija, paravirtualizacija, prekompajliranje izvornog koda, emulacija, simulacija, itd. Od tih pristupa, softverska emulacija ima najmanje ograničenja u izboru koja se platforma može oponašati na kojoj, barem sa stanovišta instrukcijskog skupa oponašane platforme. Softverskom emulacijom je moguće izvršavati i programe razvijane za sasvim drugu mikroprocesorsku familiju, a kako emulatori izvršavaju kod operativnog sistema, moguće je u oponašanom sistemu instalirati i proizvoljni operativni sistem.

II. UTICAJ RUTINE ZA DEKODIRANJE NA BRZINU SOFTVERSKE EMULACIJE

Međutim, softverska emulacija tipično ima ograničenje sa stanovišta brzine. Cilj je postići da se na domaćinskom računaru postigne brzina što približnija brzini emuliranog računara, a za mnoge primjene i veća. Zavisno od načina pisanja emulatora i programskog jezika u kome je on pisan i razlike između arhitektura domaćinskog i oponašanog mikroprocesora, razlika u brzini domaćinskog i oponašanog sistema treba da bude najmanje faktor 3. Do ove procjene se dolazi jer je pored oponašanja same instrukcije potrebno oponašati i proces dohvatanja operacionog koda, određivanja pojedine rutine i uvećanja simuliranog programskog brojača. Zbog razlika u instrukcijskom skupu taj faktor je u praksi veći, jer je potrebno više instrukcija domaćinskog procesora da se emulira jedna. Na primjer, programi za 3.5 MHz Z80 sistem, ako je emulator pisan u maksimalno optimizovanom asemblerskom jeziku, izvršavaju u realnom vremenu na sistemima počev od 386 SX 20 MHz naviše, dakle približno 6 puta bržem sistemu gledajući samo frekvencije, odnosno 32 puta bržem gledajući VAX MIPS.

Mnogi emulatori su, međutim pisani u jezicima visokog nivoa, najčešće C, ali i jezicima čiji kompajleri generišu manje optimalan kod poput Visual BASIC-a, pa čak i jezicima koji se interpretiraju, kao što je JavaScript. Testiranja brzine raznih emulatora istog Z80 sistema pokazuje da neke implementacije (npr. spomenuta JavaScript implementacija) ne postižu brzinu originala ni na sistemima Pentium 4 na 2GHz. (razlika 570 puta u frekvenciji i 4000 puta u MIPS).

Pošto emulacija nije sama sebi cilj, nego služi za izvršavanje sistemskih i aplikativnih programa, a sama predstavlja čisti režijski rad, brzina ovdje treba biti ispred drugih ciljeva softverskog razvoja, poput prenosivosti, čitljivosti ili zauzeća memorije.

Ako se skoncentriše samo na emulaciju instrukcija, osnovni algoritam većine softverskih emulatora je sljedeći:

```
pc:=init;
repeat
  pc:=pc+1;
 instruction:=mem[pc];
 execute_simulated (instruction);
until not terminated;
```

U prethodnom algoritmu pc predstavlja simulirani programski brojač (obično rezervisani registar domaćinskog računara ili memorijska lokacija), a niz mem simulirani memorijski prostor. Algoritam pokazuje da za svaku simuliranu instrukciju treba uvećati simulirani programski brojač, naći lokaciju izvršne rutine, provjeriti izlazni kriterij i ponoviti petlju. Ako je implementacija same instrukcije relativno jednostavna, taj režijski dio koda može postati značajan procenat vremena izvršavanja emulatora.

Modifikacijom ovog algoritma, može se petlja odmotati, izbjeći poziv podprograma, a uslov testiranja izlaza premjestiti da bude rjeđi, na primjer da se uslov završetka emulacije testira

samo u interrapt rutinama. Sada bi tijelo svake oponašane instrukcije izgledalo ovako:

```
execute_simulated_code;
pc:=pc+1;
instruction:=mem[pc];
jump (routine_location[instruction]);
```

Zadnje tri instrukcije ovog pseudokoda (uvećanje simuliranog programskog brojača, čitanje operacionog koda i skok na odgovarajuću rutinu) predstavljaju rutinu za dekodiranje emulirane instrukcije. Ako je simulirani kod jednostavan (npr. jedna instrukcija koja premješta podatke između dva registra), u nekim oponašanim instrukcijama ta rutina može predstavljati i 80% izvršavanog koda. Veliki efekti optimizacije koda se mogu postići ubrzavanjem te rutine. U daljnjem toku će biti analizirane neke rutine korištene u raznim emulatorima ili predložene u ovom radu, na različitim domaćinskim platformama. Rutine će biti poređene prema broju mašinskih instrukcija potrebnom za ostvarivanje datog zadatka. Ne znači nužno da će rutina koja ima manje instrukcija uvijek biti brža, jer vremena izvršavanja pojedinačne instrukcije zavise od generacije mikroprocesora, redoslijeda instrukcija u cjevovodima, složenosti instrukcija, uključenosti keševa, memorijskih adresa na kojima se instrukcije nalaze, uparenosti instrukcija u višestrukim cjevovodima, predviđanja grananja itd. Ipak, u većini slučajeva kraća rutina je i brža.

III. DEKODIRANJE INSTRUKCIJA U JEZIKU C

A. Dekodiranje koristeći instrukciju switch

Ako je portabilnost emulatora bitna, dva najčešća izbora programskog jezika za realizaciju softverskog emulatora mikroprocesora su C i Java. Prvi od ta dva jezika je dosta prenosiv na nvou izvornog koda, a drugi na nivou izvršnog koda. Sintaksno su jezici dosta slični, pa se ideje u realizaciji rutine za dekodiranje mogu primijeniti na oba jezika.

Prva ideja za prepoznavanje instrukcija je velika switch naredba, principijelno opisana u [1]. Neka se, na primjer u jeziku C oponaša procesor MOS 6502, koji ima 256 operacionih kodova. Ako je svaka emulirana instrukcija izdvojena u posebnu funkciju, ili (zbog brzine bolje) makro, dio koda koji prepoznaje instrukcije i izvršava bi izgledao slično sljedećem:

```
int pc;
char mem[65536];
void main(void) {
  while (1) {
    switch (mem[pc++]) {
       case 0: brk(); break;
       case 1: oraizx(); break;
       case 2: kil(); break;
       case 3: sloizx(); break;
 /* isjeceno */
       case 255: iscabx(); break;
    }
  }
}
```

Velika prednost ovog pristupa je u njegovoj standardnosti,. Prepravke za emulaciju procesora sa operacionim kodovima druge dužine od osam bita su minimalne, a rutina se u neizmjenjenom obliku može koristiti u većini jezika razvijenih iz C, kao što su Java i C#. Ako je realizaciji switch naredbe pristupljeno sistematično, tako da su svi operacioni kodovi popunjeni, kompajleri kao što je gcc će generisati tabelu skokova za svaku instrukciju. Drugi načini realizacije switch naredbe, sekvencijalni testovi, tablice skokova, testovi opsega, balansirana stabla ili bit testovi, opisani u [2] zahtijevaju manje prostora, ali se sporije izvršavaju. Koji će od tri algoritma za realizaciju switch naredbe kompajler odabrati zavisi od distribucije uslova u switch naredbi. No, i ovaj najbrži pristup generisanju koda iz switch naredbe ne rezultuje najbržim mogućim kodom realizacije dekodiranja instrukcija, što se može pogledati iz generisanog koda za i386 arhitekturu, konvertovanog u MASM sintaksu.

```
_MAIN:
        PUSH    EBP
        MOV     EBP, ESP
        AND     ESP, -16
        CALL    ___MAIN
        JMP     L12
L14:
        NOP
L12:
        MOV     EDX, DWORD PTR _PC
        MOV     AL, BYTE PTR _MEM[EDX]
        MOVSX EAX, AL
        INC     EDX
        MOV     DWORD PTR _PC, EDX
        CMP     EAX, 255
        JA      L14
        MOV     EAX, DWORD PTR L11[0+EAX*4]
        JMP     EAX
        .SECTION .RDATA,"DR"
        .ALIGN 4
L11:
        .LONG  L3
        .LONG  L4
        .LONG  L5
        .LONG  L6
        ....
        .LONG  L258
        .TEXT
L3:
        CALL    _BRK
        JMP     L2
L4:
        CALL    _ORAIZX
        JMP     L2
...
L258:
        CALL    _ICSABX
        JMP     L12
L2:
        JMP     L12
```

U gornjem uzorku, proces dohvatanja instrukcije, dekodiranja, uvećanja programskog brojača i skoka u petlju zauzima 11 instrukcija, počev od labele L12 do instrukcije jmp eax uz instrukcije jmp L2 i jmp L12. Uočava se nepotrebni skok na lokaciju na kojoj se ponovo nalazi jmp instrukcija, što je posljedica instrukcija break i while. Pored ovoga, vrijeme se troši na testiranje da li je vrijednost operacionog koda instrukcije veće od 255, kao i na prebacivanje vrijednosti iz memorije u registre.

B. Dekodiranje koristeći niz pointera na funkcije

Broj nepotrebnih skokova se može smanjit koristeći niz pointera na funkcije. U navedenom primjeru pojednostavljene emulirane arhitekture koja ima tri instrukcije, sa kodovima 0, 1 i 2, glavna petlja sadrži više puta ponovljenu instrukciju poziva one funkcije iz niza funkcija s indeksom koja je vrijednost memorije na oponašanom programskom brojaču. Ovo ponavljanje smanjuje potrebu za skokom nakon svake instrukcije.

```
int pc;
unsigned char a;
unsigned char mem[65536];
static inline void add() {
  a=a+mem[pc++];
}
static inline void sub() {
  a=a-mem[pc++];
}
static inline void or() {
  a=a | mem[pc++];
}
void (*func_ptr[3])() = {add, sub, or};

main() {
 do {
       (*func_ptr[(int)mem[pc++]])();
       (*func_ptr[(int)mem[pc++]])();
       (*func_ptr[(int)mem[pc++]])();
       (*func_ptr[(int)mem[pc++]])();
       (*func_ptr[(int)mem[pc++]])();
       (*func_ptr[(int)mem[pc++]])();
       (*func_ptr[(int)mem[pc++]])();
     } while (1);
  return 0;
}
```

Generisani kod za pojedinačni poziv izgleda zahtijeva sedam instrukcija.

```
MOV     EAX, DWORD PTR PC
MOVZX   EDX, BYTE PTR MEM[EAX]
MOVZX   EDX, DL
MOV     EDX, DWORD PTR FUNC_PTR[EDX*4]
ADD     EAX, 1
MOV     DWORD PTR PC, EAX
CALL    EDX
```

Međutim, svaka pojedinačna instrukcija ima četiri obavezne instrukcije za uspostavljanje steka i povratak iz podprograma (push ebp; mov ebp,esp; pop ebp i ret), a kod ovog pristupa nije moguće prebaciti tijelo oponašane instrukcije u glavnu petlju, što se vidi iz generisanog koda za instrukciju add. Stoga i ovaj pristup zahtijeva 11 instrukcija za proces dekodiranja oponašane instrukcije.

```
PUSH    EBP
MOV     EBP, ESP
MOV     EAX, DWORD PTR PC
MOVZX           ECX, BYTE PTR MEM[EAX]
MOVZX           EDX, BYTE PTR A
ADD     EDX, ECX
MOV     BYTE PTR A, DL
ADD     EAX, 1
MOV     DWORD PTR PC, EAX
POP     EBP
RET
```

C. Dekodiranje koristeći sračunati goto

Kompajler gcc je uveo jedno malo poznato proširenje jezika C, nizove labela i sračunati goto. Kako je navedeno u [2], tehnika je korištena u emulatorima JVM. Na ovaj način se može izbjeći poziv podprograma za svaku instrukciju i upotreba petlje. Prefiksni operator && predstavlja adresu labele. Prethodni primjer emulatora sada predstavljen koristeći ovo proširenje izgleda ovako.

```
int pc;
unsigned char a;
unsigned char mem[65536];
main(){
  pc=0;
  static void *table[]={&&add, &&sub, &&or};
add:
  a=a+mem[pc++];
  goto *table[mem[pc++]];
sub:
  a=a-mem[pc++];
  goto *table[mem[pc++]];
or:
  a=a | mem[pc++];
  goto *table[mem[pc++]];
   return 0;
}
```

Generisani kod za instrukciju goto *table[mem[pc++]]; koja obavlja dekodiranje sljedeće instrukcije izgleda ovako.

```
MOV     EAX, DWORD PTR PC
MOVZX           EDX, BYTE PTR MEM[EAX]
MOVZX           EDX, DL
MOV     EDX, DWORD PTR TABLE[0+EDX*4]
ADD     EAX, 1
MOV     DWORD PTR PC, EAX
MOV     EAX, EDX
JMP     EAX
```

Iako je rutina za dohvatanje instrukcija skraćena na 8 instrukcija, ponovo se uočava nepotreban kod. Na primjer, instrukcija movzx edx,dl je potpuno nepotrebna, jer su viša tri bajta registra edx već popunjena nulama prethodnom instrukcijom, instrukcijski par add eax,1 i mov DWORD PTR

pc, eax se može zamijeniti s inc DWORD PTR pc, a zadnje dvije instrukcije se mogu zamijeniti jednom jmp edx.

Očekivati je da će se kvalitet optimizacije izlaznog koda vremenom poboljšavati, ali da će ručno pisane asemblerske rutine za dekodiranje instrukcija emuliranih procesora i dalje imati performansne prednosti. U daljnjem toku će biti opisane neke rutine za dekodiranje instrukcija pisane u asemblerskom jeziku popularnih arhitektura.

IV. DEKODIRANJE INSTRUKCIJA NA INTEL IA16 I IA32

A. Dekodiranje koristeći tablicu adresa

Kada se emulator piše u asemblerskom jeziku domaćinskog računara moguće je dalje optimizovati kako implementaciju samih instrukcija, tako i rutinu za njihovo dekodiranje. Veliki efekti ubrzanja se postižu čuvanjem sadržaja važnijih emuliranih registara i pointera na memoriju emuliranog sistema u procesorskim registrima domaćinskog računara. I u jeziku C je pristup koristeći tablicu adresa davao bolje rezultate, pa se on u optimizovanom obliku može upotrebiti i u emulatorima pisanim u asemblerskom jeziku.

Sljedeća rutina je pisana koristeći Intel 8086 instrukcijski skup i omogućava dekodiranje osmobitnih operacionih kodova. Korištena je npr. u emulatoru Warajevo, autora Ž. Jurić i S. Ribić. Ovdje registar SI predstavlja oponašani programski brojač, BX je pomoćni registar. Segment DS pokazuje na memoriju oponašanog Z80 procesora, a tabela instrukcija je smještena u stek segment.

```
INC SI
MOV   BL,[SI]
XOR   BH,BH
ADD   BX,BX
JMP   SS: TABLE1[BX]

TABLE1  DW   NOP_, LDBCNN, LDMBCA, INCBC, ...
```

U nekim slučajevima se instrukcija xor bh,bh može izostaviti, ako je prethodnom instrukcijom registar bh već postavljen na nulu, pa rutina uzima 4 ili 5 instrukcija.

Rutina je primjenjiva i ako operacioni kodovi imaju specijalne prefikse ili sufikse, što efektivno znači da neke instrukcije imaju šesnaest-bitne ili veće kodove, ali ne sve. To se postiže ponavljanjem rutine nad različitim tabelama za više operacionih kodova.

Zbog ograničenja veličine segmenta u 16 bitnim režimima rada, rutina je nepogodna za emulaciju procesora čiji su operacioni kodovi dominantno šesnaest-bitni (PowerPC, PDP 11, Motorola 68000, DEC Alpha) jer se 65536 adresa ne može smjestiti u jednu tabelu.

Uvođenjem instrukcija MOVZX i adresnih režima dostupnih od 386 procesora i jačih┬, rutina za dekodiranje instrukcija se može dalje znatno skratiti. Za osmobitne operacione kodove, ukoliko registar ESI ima ulogu emuliranog programskog brojača, rutina izgleda ovako.

```
INC SI ; AKO JE ADRESNI PROSTOR 16 BITNI
MOVZX   EBX,BYTE PTR [ESI]
JMP   TABLE[4*EBX]
….
TABLE  DD   NOP_, LDBCNN, LDMBCA, INCBC, ...
```

Ako se prihvati žrtva EAX registra (teška odluka, jer je on kandidat za emulaciju akumulatorskih registara), tako da njegova viša tri bajta uvijek imaju vrijednost 0, rutina se može smanjiti i na samo dvije instrukcije,

```
LODSB ; AKO JE ADRESNI PROSTOR 16 BITNI
JMP   DWORD TABLE[4*EAX]
….
TABLE  DD   NOP_, LDBCNN, LDMBCA, INCBC, ...
```

Za emulaciju procesora s šesnaest-bitnim operacionim kodovima, rutina je samo malo drugačija.

```
ADD ESI,2
MOVZX   EBX,WORD PTR [ESI]
JMP   DWORD TABLE[4*EBX]
….
TABLE  DD   INSTR1, INSTR2, INSTR3, ...
```

U varijanti s žrtvovanim EAX registrom rutina za dekodiranje šesnaest-bitnih instrukcija izgleda ovako.

```
LODSW
JMP   DWORD  TABLE[4*EAX]
….
TABLE  DD   INSTR1, INSTR2, INSTR3, ...
```

B. Dekodiranje koristeći poziciono ovisni kod

Ako se emulirane instrukcije smjeste unutar kodnog segmenta u koracima po 256, tako da se instrukcija sa operacionim kodom 00 nalazi na lokaciji 00, instrukcija s operacionim kodom 01 na lokaciji 256, instrukcija s operacionim kodom 02 na lokaciji 512 itd, može se rutina za dekodiranje instrukcija dalje skratiti. Ako je SI registar za oponašanje programskog brojača, uvećanje programskog brojača i skok na odgovarajuću rutinu se obavlja sa

```
INC SI
MOV BH,[SI]
MOV BL,0
JMP BX
```

Pri tome, instrukcija mov bl,0 često može biti izostavljena, ako emulirana instrukcija nije mijenjala sadržaj registra bl. Rutina je korištena u emulatoru Z80, autora G. Lunter. Ova rutina čini kod dosta razrijeđenim. Primjenjiva je za operacione kodove sa osmobitnim prefiksima, gdje se emulacija kodova lociranih iza prefiksa asemblira od adresa koje podijeljene s 256 daju isti ostatak pri dijeljenju. Rutina za dekodiranje nakon prepoznavanja prefiksa tada u registar bl treba smjestiti taj ostatak. Nije primjenjiva za operacione kodove druge dužine od osam bita.

Eksperimenti su pokazali da je ova rutina brža od rutine opisane u IV-A u većini slučajeva, kada nema instrukcijskog keša ili kada je instrukcijski keš veći. Jedino se nije pokazala

bržom kada je instrukcijski keš razmjerno mali, jer skokovi na međusobno dosta udaljene lokacije imaju više grešaka keša nego iščitavanje susjednih lokacija u tabelama.

C. *Dekodiranje koristeći preračunate adrese u emuliranoj memoriji*

Koristeći segmentne registre dodane u 386 ili jačim procesorima u 16 bitnom rutina za dekodiranje osmobitnih operacionih kodova čija memorija ne prelazi 64 K, se može svesti na svega dvije instrukcije. Koriste se tri segmenta. DS pokazuje na emuliranu memoriju, dok segmenti FS i GS sadrže pokazivače na izvršne rutine za svaki bajt u emuliranoj memoriji. To su, za instrukcije na parnim adresama:

```
INC SI
JMP FS:TAB1[SI]
```

Za instrukcije na neparnim adresama, rutina izgleda ovako.

```
INC SI
JMP GS:TAB2[SI-1]
```

Mana rutine je u potrebi za ažuriranjem tablica pri svakom upisu u memoriju, zbog samomodifikujućeg koda. Sa stanovišta emulatora, samomodifikujući kod se javlja veoma često, jer i izvršavanje kompajlera pod emulatorom takođe modifikuje prostor izvršnih instrukcija. No, za emulaciju sistema koji uglavnom izvršavaju osmobitni kod u ROM-u, poput konzola za igre, metoda je jako pogodna. Emulator koji je koristio ovakvu rutinu je JPP, autor A. Guldbransen.

U trideset-dvobitnim režimima rada dovoljna je jedna tablica adresa instrukcija emuliranih u memoriji. Ta tablica je četiri puta veća od veličine emulirane memorije, i treba je ažurirati svaki put kada se upisuje podatak u emuliranu memoriju, osim kada se emuliraju sistemi koji nemaju samomodifikujući kod. Rutina se sastoji od dvije instrukcije, i ista je za instrukcije na parnim i neparnim adresama.

```
INC ESI
JMP DWORD TABLE1[ESI*4]
```

D. *Dinamička rekompilacija*

Postoji način da se dužina rutine za dekodiranje instrukcija svede na nulu. Zove se dinamička rekompilacija. Najprije se instrukcije dekodiraju koristeći metodu, na primjer opisanu u V-B. Emulirane instrukcije pored oponašanog koda, upisuju u izlazni bafer skup ekvivalentnih instrukcija domaćinskog procesora, bez upisivanja same rutine za dekodiranje. Kada se dođe do emulacije instrukcija skokova, pokazivač na izvršenje prve instrukcije se prebacuje da pokazuje na izlazni bafer. Kako je opisano u [3], algoritam se može proširiti i drugom fazom koja transformiše i generisane instrukcije, dodatno povećavajući brzinu rada. Ovaj algoritam ne podržava samomodifikujući kod i zahtijeva povećanu memoriju. Prvi prolaz kroz instrukcije je sporiji nego što bi bio sa klasičnim načinom dekodiranja, pa se efekti ubrzanja osjećaju samo kod instrukcija unutar petlji. Primjenjiv je i na drugim arhitekturama, pa neće biti posebno ponavljan u daljnjem toku teksta.

V. DEKODIRANJE INSTRUKCIJA NA MOTOROLA 68000

A. *Dekodiranje koristeći tablicu adresa*

Adresni režimi procesora iz familije Motorola 68000 koji podržavaju postinkrementiranje adresnih registara, omogućavaju da na ovoj familiji procesora rutine dekodiranja budu kraće nego na Intel platformama. Tako se neki od adresnih registara A0-A7 može koristiti za emulirani programski brojač. S druge strane skokovi nisu indirektni, pa se sam skok treba izvesti s dvije instrukcije. Ako registar A5 oponaša programski brojač, registar A3 pokazuje na dio koda gdje su emulirane instrukcije, a registar A2 pokazuje na tablicu skokova. Ovakva rutina se realizuje s četiri instrukcije.

```
MOVE.B (A5)+,D0
ADD.W D0,D0
MOVE.W (A2,D0.W), D0
JMP (A3,D0.w)
```

No, na Motorola 68020 i jačim procesorima, koji posjeduju indirektne skokove, za ovu rutinu su potrebne svega dvije instrukcije.

```
MOVE.B (A5)+,D0
JMP ([0,A0],D0.W*2)
```

Prilikom upotrebe ovih rutina paziti na problem kada PC dostigne maksimalnu vrijednost emulirane arhitekture. Npr., ako se emulira arhitektura sa šesnaest-bitnim programskim brojačem treba ponoviti u emuliranoj memoriji instrukcije koje počinju od emulirane lokacije 0 i na lokacijama 65536 i više, sve dok se ne bude sigurno da će eventualni apsolutni skok vratiti programski brojač u uobičajeni opseg.

Rutine je potrebno mijenjati za emuliranje procesora s dominantno šesnaest-bitnim operacionim kodovima, jer se programski brojač ne može više ažurirati paralelno s dohvatanjem instrukcije. Tada prva rutina postaje

```
MOVE.W (A5)+,D0
LEA (A5)+,A5
ASL.L #2,D0
MOVE.W (A2,D0.L), D0
JMP (A3,D0.L)
```

Rutina za MC68020 procesore i jače postaje:

```
MOVE.W (A5)+,D0
LEA (A5)+,A5
JMP ([0,A0],D0.L*4)
```

B. *Dekodiranje koristeći poziciono ovisni kod*

I na procesorima familije MC 68000 se može realizovati ubrzano dekodiranje pišući emulirane instrukcije na lokacijama koje podijeljene s 256 daju isti ostatak. Neka registar A6 predstavlja oponašani programski brojač, a registar A5 pokazuje na dio koda gdje se nalazi implementacija instrukcija. Tada rutina koja radi na svim procesorima Motorola 68K familije izgleda ovako.

```
MOVE.B    (A6)+,D0        ;
ASL.W #8,D0
JMP (A5,D0.L)
```

No, primjenom samomodifikujućeg koda, na procesoru MC 68000 može se napraviti i rutina od svega dvije instrukcije. Ona neće raditi na MC 68020 zbog dubljeg cjevovoda instrukcija, niti u slučaju ako se emulator izvršava u ROM-u. Prva od ovih instrukcija dohvati operacioni kod i modifikuje narednu JMP instrukciju.

```
MOVE.B      (A6)+,DVA+2-BASE(A5)
DVA:        JMP     $FF00(A5)
```

U prethodnom primjeru, ako je operacioni kod instrukcije koja se emulira bio $3E, druga instrukcija se prije izvršenja pretvori u JMP $3E00(A5), što izaziva skok na pravu adresu. Ova rutina je korištena u emulatoru Tezxas, autor S. Ribić. Prilikom realizacije instrukcijskih skupova s prefiksnim operacionim kodovima, treba promijeniti relativnu poziciju JMP instrukcija.

C. Dekodiranje koristeći preračunate adrese u emuliranoj memoriji

Ako se ne koristi dinamička rekompilacija, najbrža rutina za MC 68000 familiju koristi dodatnu tablicu veličine dva puta veće od emulirane memorije. Da bi radio samomodifikujući kod, svaka izmjena memorije u emuliranom sistemu povlači i ažuriranje ove tablice. Korištena je u emulatoru Spectrum, za računar Amiga, autor P. McGavin. Svaki element te tablice sadrži relativnu adresu emulirane instrukcije. Registar A3 predstavlja emulirani programski brojač pomnožen s 2.

```
MOVE.W (A3)+,D6;
JMP TABLE(PC,D6.W)
```

VI. DEKODIRANJE INSTRUKCIJA NA JVM

JVM nema indirektno adresiranje, ali se instrukcijom tableswitch, lako implementira tablica skokova. Kako je navedeno u [4], ova instrukcija je prostorno manje efikasna od lookupswitch, ali je brzinski daleko efikasnija. Ako je emulirani programski brojač smješten u lokalnu varijablu 0, a pointer na niz emulirane memorije u lokalnu varijablu 1, tada je rutina za dekodiranje osmobitnog operacionog koda ovakva.

```
iinc 0 1     ;     PC=PC+1
aload_1      ; adresa memorije mem
iload_0      ; varijabla 0 predstavlja PC
baload       ; element niza
tableswitch
      0      : Instr0
      1      : Instr1
      ...
      default : Dlabel
```

Za emulaciju šesnaest-bitnim operacionim kodovima, instrukciju baload treba zamijeniti instrukcijom saload.

VII. DEKODIRANJE INSTRUKCIJA NA POWERPC

Ova arhitektura nema složenije adresne režime, pa se to reflektuje na rutine za dekodiranje, Ukoliko registar R7 pokazuje na tablicu emuliranih instrukcija, a registar R6 oponaša programski brojač, tada se dekodiranje instrukcija obavlja rutinom sličnom ovoj, razvijenoj iz preporuke za implementaciju switch naredbe u [5].

```
lbzu  R4,1(R6)   # Učitaj bajt i uvećaj R6 za 1
slwi R5,R4,2     # pomnoži bajt s četiri
lwzx R3,R7,R5 # R3 = TABLE[x]
mtctr R3         # napuni brojački registar
bctr             # skoči na brojački registar
```

Za emulaciju procesora s šesnaest-bitnim operacionim kodovima, prvu instrukciju treba promijeniti u lwzu R4,2(R6).

VIII. DEKODIRANJE INSTRUKCIJA NA ARM

Suprotno očekivanju za RISC procesore, i na ovoj arhitekturi se proces dekodiranja može realizovati u svega dvije instrukcije [6]. Neka R2 oponaša programski brojač, R3 pokazuje na tabelu skokova, a R0 se koristi kao pomoćni instrukcijski registar. Tada rutina izgleda ovako.

```
LDRB   R0, [R2],#1
LDR    PC, [R3,R0,LSL#2]
JUMPTABLE
      DCD    DOADD
      DCD    DOSUB ....
```

IX. ZAKLJUČAK

Pažljivim kodiranjem, izborom programskog jezika za razvoj i upotrebom trikova, može se na većini testiranih arhitektura skratiti rutina za dekodiranje instrukcija na 2 do 5 instrukcija, a primjenom dinamičke rekompilacije izbjeći. Nakon razrješenja ovog uskog grla, daljnje optimizacije emulatora se postižu optimizacijom instrukcija i ulaza/izlaza.

LITERATURA

[1] V. Moya del Barrio, "Study of the techniques for emulation programming", Universitat Politècnica de Catalunya, 2001

[2] R. Anthony Sayle. "A Superoptimizer Analysis of Multiway Branch Code Generation, Proceedings of the, GCC Developers' Summit", June 17th–19th, 2008, Ottawa, Ontario Canada

[3] M. Steil, "Dynamic Re-compilation of Binary RISC Code for CISC Architectures", Technische Universitat Munchen, 2004

[4] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, The Java ® Virtual Machine Specification", Oracle America, 2011, pp. 57

[5] S. Hoxey, F. Karim, B. Hay, H. Warren, "The PowerPC Compiler Writer's Guide", Warthman Associates, 1996

[6] W. Hohl "ARM Assembly Language: Fundamentals and Techniques", CRC Press, 2009, pp. 140

ABSTRACT

An instruction decoding is very frequently performed in software emulators. As this operation is time critical, we need togive special attention to it. The paper compares different decoding routines on variety of platforms.

INSTRUCTION DECODING IN CPU EMULATORS

Samir Ribić