AUTOMATSKO ISPITIVANJE C PREVODIOCA ZA JEDNU KLASU PROCESORA ZA DIGITALNU OBRADU SIGNALA

Snežana Crnogorac, Aleksandar Simeonov, *MicronasNIT, Novi Sad*; Zoran Jovanović, *Fakultet Tehničkih Nauka, Novi Sad, Katedra za računarsku tehniku*

Sadržaj – *U radu je prikazano automatsko ispitivanje C prevodioca za jednu klasu procesora za digitalnu obradu signala. Istaknut je značaj ispitivanja prevodioca i problemi koji se javljaju tom prilikom. Dat je postupak ispitivanja. Procesor za koji je prevodilac razvijen definiše tok instrukcija (pipeline) koji je u potpunosti otkriven za korisnika. Prevodilac je razvijen korišćenjem alata za razvoj prevodioca holandske grupe ACE nazvanog CoSy, odnosno ne koristi se neki od standardnih jezika višeg programskog nivoa.*

1. UVOD

Tipični prevodilac sadrži ulaznu sprežnu komponentu (*front-end component*) koja prevodi izvorne datoteke (pisane u C kodu) u međujezik (*IR-Intermediate Representation*), različite analizatore i optimizatore nad međujezikom i na kraju generator koda koji prevodi međujezik u asemblerski kod [1]. IR je niz stabala izraza. Generator koda obrađuje IR tako što obrađuje svako stablo ponaosob i na izlazu daje asemblerski kod [2].

Broj C programa kojima se ispituje prevodilac je konačan, a problem u prevodiocu se može javiti jedino pri specifičnoj kombinaciji C instrukcija unutar C programa pa je zbog toga ispitivanje C prevodioca od izuzetne važnosti. Takođe, za prevodioce može da se kaže da nikada nisu završeni. Usled stalne potrebe za boljim performansama, stalne pojave novih C programskih aplikacija i novih karakteristika ciljne arhitekture javlja se i stalna potreba za poboljšanjem prevodioca, a samim tim i za povećanjem broja C programa za njegovo ispitivanje.

Prevodilac čije ispitivanje je opisano u ovom radu je razvijan korišćenjem alata za razvoj prevodioca holandske grupe ACE nazvanog CoSy, odnosno ne koristi se neki od standardnih jezika višeg programskog nivoa.

U ovom radu je ispitivan prevodilac za procesor koji pripada klasi procesora za obradu digitalnih signala, i definiše tok instrukcija (*pipeline*) koji je u potpunosti otkriven za korisnika. Procesor za obradu digitalnih signala o kome je reč je nazvan *APX* (*Advanced Processor with eXpandable architecture*) koji razvija nemačka firma *Micronas*, *Freiburg*. Radi se o arhitekturi koja sadrži četiri magistale, od toga dve za memorijske podatke, jednu programsku i jednu posebnu magistralu za neposredni pristup memoriji (*DMA – Direct Memory Access*). Procesor u potpunosti podržava režime adresiranja (indeksni, preinkrement, postinkrement) i programskog toka, direktni pristup memoriji, kao i kontrolu prekida.

Aplikacije koje se prevode pomoću prevodioca su vrlo složene, a prethodno opisana arhitektura za koju je razvijan prevodilac nije realizovana na čipu nego se koristi simulator. Simulacije dugo traju što otežava pronalaženje grešaka u prevodiocu. To je dodatni razlog zbog čega je važno kvalitetno ispitivanje prevodioca, odnosno potrebno je da ispitni programi budu tako napisani da se njihovim prevođenjem vrlo brzo može zaključiti gde je došlo do greške u prevodiocu ukoliko do nje dođe.

2. OKRUŽENJE ZA ISPITIVANJE PREVODIOCA U OKVIRU ALATA ZA RAZVOJ PREVODIOCA

CoSy (*Compiler System*) je sistem za razvoj prevodioca, čiji je koncept zasnovan na relativno nezavisnim komponentama (modulima), od kojih svaka obavlja određeni zadatak i označena je kao '*engine*'. Modularnost ovakvog sistema omogućava jednostavno ponovno korišćenje algoritama prevodioca za razvoj novih, različitih prevodilaca.

Od posebne važnosti za ovaj rad su pravila. Svaka instrukcija ciljne arhitekture opisana je stablom međukoda. Pridruživanjem određenih klauzula (*CONDITION*, *COST*, *EMIT*) datom stablu međukoda definiše se pravilo. Pravila se nalaze u datotekama za opis generatora koda CGD (*code generator description*). Informacije se od jednog pravila ka drugom prenose pomoću *neterminala*. *Neterminal* odgovara nekoj osobini ciljne mašine koja je zajednička za određeni broj instrukcija. Pravila imaju sledeći oblik [4][2]:

```
RULE [<ime pravila>]
      ( <šablon za popločavanje> )  [ -> <rezultat pravila> ];
CONDTION
      { < uslov pod kojim je moguće primeniti pravilo> }
COST
      <celobrojna vrednost koja predstavlja cenu pravila>;
EMIT
      { <kod čije izvršavanje generiše asemblerski izlaz> }
```

RULE je ključna reč kojom započinje deklaracija pravila. *CONDITION* je klauzula kojom se definišu uslovi pod kojim dato pravilo može biti izabrano. *COST* definiše cenu primene pravila. *EMIT* klauzulom se definišu akcije koje se izvršavaju kada je pravilo primenjeno.

SuperTest je okruženje u okviru *CoSy* alata za ispitivanje prevodioca. Međutim, i pored ispitivanja prevodioca pomoću *SuperTest*-a prilikom prevođenja složenih aplikacija pojavljuje se veliki broj grešaka u prevodiocu. Otuda potreba za dodatnim ispitivanjem. S obzirom da *SuperTest* ima mogućnost dodavanja novih ispitnih programa [3], u njega su dodavani novi ispitni programi i na taj način prevodilac je automatski ispitivan.

3. ANALIZA POKRIVENOSTI PRAVILA ISPITNIM PROGRAMIMA I IZBOR PRAVILA ZA ISPITIVANJE

Ispitivanju se pristupilo tako što je pomoću *rulestat* komponente napravljena statistika pokrivenosti pravila ispitnim programima iz *SuperTest*-a. Rulestat komponenta je komponenta u okviru *CoSy* sistema koja daje statistiku o pokrivenosti pravila ispitnim programima.

Analizom je dobijeno da u sastavu prevodioca postoji 871 pravilo od kojih je samo 419 ispitano pomoću *SuperTest*-a, što znači da ukoliko se prevodilac ispituje samo pomoću *SuperTest*-a 452 pravila ostaje neispitano.

Zatim su prevedene 4 složene aplikacije (AAC, DTS, MP3 i DRM). Analizom je dobijeno da se 28 pravila od onih 452 koja nisu ispitana *SuperTest*-om pojavljuje u tim aplikacijama. To je razlog što su se prilikom razvoja prevodioca pojavljivale greške u prevodiocu prilikom prevođenja tih složenih aplikacija i pored toga što je S*uperTest* verifikovao nepostojanje grešaka u prevodiocu,

odnosno to je razlog što se javila potreba za dodatnim ispitivanjem prevodioca.

S obzirom da *emit* deo pravila može da sadrži nekoliko grana, veoma je važno ispitati svaku njegovu granu. Takođe, statistika koja je pokazala da je neko pravilo ispitano pomoću *SuperTest*-a ne potvrđuje da su sve grane tih pravila ispitane. To je dodatni razlog zbog čega su se pojavljivale greške čak i u pravilima koja su pokrivena *SuperTest*-om.

Nakon obavljene analize pokrivenosti pravila ispitnim programima iz *SuperTest*-a, pristupilo se pisanju dodatnih ispitnih programa koji će ući u sastav *SuperTest*-a. Prilikom pisanja ispitnih programa veoma se vodilo računa o tome da se ispitaju sve grane *emit* dela pravila.

Prvo su pisani ispitni programi za pravila koja se pojavljuju u 4 prethodno navedene aplikacije, a koja nisu pokrivena S*uperTest*-om. Zatim su pisani ispitni programi za pravila koja se ispituju mali broj puta u S*uperTest*-u (do 10 puta) pod pretpostavkom da je manja verovatnoća da su ispitane sve grane *emit* dela pravila ukoliko je pravilo ispitano mali broj puta pomoću S*uperTest*-a. Nakon toga se pristupilo pisanju pravila koja nisu pokrivena niti *SuperTest*-om niti aplikacijama, pod pretpostavkom da bi se ta pravila mogla pojaviti u nekim drugim složenim aplikacijama.

4. STRUKTURA ISPITNOG PROGRAMA I NAČIN PISANJA ISPITNIH PROGRAMA

Svaki ispitni program uključuje u sebe *def.h* datoteku u kome su definisani makroi koji se koriste u ispitnom programu. Neki od najčešće primenjivanih makroa [3] su:

- MAIN – određuje tip main funkcije ispitnog programa,
- HEADER – daje zaglavlje ispitnog programa na standardnom izlazu,
- VERIFY_TEST – proverava vrednost izraza u okviru svog argumenta, ta vrednost se prosleđuje kao dijagnostička poruka na standardni izlaz,
- END_SECTION – označava kraj segmenta ispitnog programa i daje dijagnostičku poruku o njegovoj prolaznosti (“PASSED”, “FAILED”).

Da bi se na najbolji način objasnila struktura ispitnog programa i način njegovog pisanja dat je primer pravila logičke operacije *ili* za opseg vrednosti od 0 do 128:

```
RULE [or_int_scnst] o:mirOr(rs1:reg, c:mirIntConst) -> rd:reg;
CONDITION { is_7b_const(UnivInt_to_int(c.Value)) }
COST 1;
CONSUMER CSTAGEX;
PRODUCER PSTAGEY;
CLASS alu_cmp;
CLASS alu_imm_mov;
EMIT {
   Condition dcond = cond_true;
   BOOL EmitConditional = FALSE;
   `lirCombine combine;

   combine = `gcg_this_psc->Combine;

   if (lirPscNode_get_AlterCondition(gcg_this_psc)) {
      dcond = lirPscNode_get_cond(gcg_this_psc);
      EmitConditional = TRUE;
   }
   if (!EmitConditional) {
      if (!is_nil(combine)) {
         switch (get_comb_id(`combine->combine)) {
            case risc_alu_int_cmp:
               emit_cond_aluop3_imm8(gcg_this_psc, state, dcond, rd,
                                     rs1, aluopext_or,
UnivInt_to_int(c.Value), TRUE);
               break;
            case risc_alu_imm_mov:
               `combine->Combine_alu_imm_mov_c.immval =
UnivInt_to_int(c.Value);
               `combine->Combine_alu_imm_mov_c.op = aluop_or;
               break;
            default:
               assert(0);
         }
      }
      else
         emit_cond_aluop3_imm8(gcg_this_psc, state, dcond, rd, rs1,
                               aluopext_or,
UnivInt_to_int(c.Value), FALSE);
   }
   else
      emit_cond_aluop3_imm8(gcg_this_psc, state, dcond, rd, rs1,
                            aluopext_or,
UnivInt_to_int(c.Value), FALSE);
}
```

Može se primetiti da dato pravilo ima četiri grane u *emit* delu. Jedna grana generiše asemblersku instrukciju u kojoj se u akumulator *aluregc* upisuje vrednost dobijena nakon primenjene logičke operacije između vrednosti koja je sadržana u akumulatoru *aluregb* i označene osmobitne vrednosti i ima sledeći oblik [5]:

YCaluregc#4 **= XC**aluregb#4 aluop index#-8 (1)

Druga grana generiše asemblersku instrukciju u kojoj se instrukcija (1) izvršava ukoliko je ispunjen određeni uslov, odnosno u zavisnosti od vrednosti koja se nalazi u registru statusa (*flags*) poslednje ALU operacija, a ima oblik[5]:

cond **YC**aluregc#4 **= XC**aluregb#4 aluop index#-8 (2)

Treća grana generiše asemblersku instrukciju koja predstavlja kombinaciju instrukcije (1) i instrukcije koja prebacuje sadržaj akumulatora u registar, odnosno rezultat logičke operacije koji se nalazi u *aluregc* se prebacuje u *riscreg* registar. Njen oblik je [5]:

YCaluregc#4 aluop index#-8, **ZR(**riscreg#8**)** = **YR(**aluregc#4**)** (3)

Četvrta grana generiše asemblersku instrukciju koja predstavlja kombinaciju instrukcije (1) i ujedno postavlja vrednost u registar statusa (*flags*) koja se dobija na osnovu rezultata logičke operacije. Instrukcija ima oblik [5]:

YCaluregc#4 **= XC**aluregb#4 aluop index#-8, **FLAGS** (4)

Pravilo je u potpunosti ispitano ukoliko je ispitana svaka njegova grana kao što je slučaj u narednom primeru ispitnog programa koji ispituje prethodno (*or_int_scnst*) pravilo:

```
/*
 * @(#) t_or_int_scnst.c          Dec/04
 */

#include "def.h"

int init (int cnt)
{
   switch (cnt) {
      case 0:  { return 0; break; }
      case 1:  { return 1; break; }
      case 2:  { return 2; break; }
      case 3:  { return 3; break; }
      default: { return 4; }
   }
}

MAIN
{
   int a, b, c, d;

   diagn = "Testing of or_int_scnst";
   HEADER();

   diagn = "No combine, AlterCondition is not set";
   c = init(1);
```

```
    c |= 124;
    VERIFY_TEST(c == 125);
    END_SECTION();

    diagn = "No combine, AlterCondition is set";
    a = init(1);
    if (a>0)
       a |= 127;
    VERIFY_TEST(a == 127);
    END_SECTION();

    diagn = "Combine: risc_alu_imm_move, AlterCondition is not set";
    b = init(3);
    b |= 125;
    d = b;
    b += d;
    d += b;
    VERIFY_TEST((d == 381) && (b == 254));
    END_SECTION();

    diagn = "Combine: risc_alu_int_cmp, AlterCondition is not set";
    b = init(1);
    b |= 126;
    if(b<0)
       b = 2;
    VERIFY_TEST(b == 127);
    END_SECTION();
}
```

Nakon prevođenja prethodnog ispitnog programa generisani asemblerski kod ima sledeći oblik:

```
_TEST__MAIN
   SETFRAME ( 0, 9, 0 )
    . . .
      YC0 = XC0 | 124                  ; or_int_scnst 15
      . . .
      IF GT   YC0 = XC0 | 127          ; or_int_scnst 0
      . . .
      YC0 |= 125, ZR(R1) = YR(R0)      ; mov_reg 14
      . . .
      YC0 = XC0 | 126, flags           ; or_int_scnst 11
      IF LT EXEC 1                     ; if_thenisnext 12
         YR(R0).clearhigh = 2          ; ldi_low 0
      . . .
   RETURN YPC=XRETREG, ARETREG=ZM[P7(+=1)], skip
```

Asemblerski kod pokazuje da su ispitane sve četiri grane pravila.

I pored toga što su izgenerisana sva četiri oblika asemblerskih instrukcija pravila *or_int_scnst*, pravilo mora da se potvrdi konkretnim vrednostima. Za to je zadužen gore opisani makro VERIFY_TEST. Verifikacijom ispitnog programa se na standardnom izlazu dobija:

```
TESTING: Paragraph: a.b.c Test: t_or_int_scnst

   Testing of or_int_scnst

ACE C validation on MACH: COMP DATE

TESTING: Paragraph: a.b.c Test: t_or_int_scnst

   Testing of or_int_scnst

   No combine, AlterCondition is not set                    PASSED

   No combine, AlterCondition is set                        PASSED

   Combine: risc_alu_imm_move, AlterCondition is not set    PASSED

   Combine: risc_alu_int_cmp, AlterCondition is not set     PASSED

RESULT: Paragraph: a.b.c Test: t_or_int_scnst               PASSED
```

Na kraju se može izvesti zaključak da je pravilo *or_int_scnst* uspešno i u potpunosti ispitano.

5. REZULTATI

U toku razvoja prevodioca napisana su 102 dodatna ispitna programa i na osnovu njih je pronađeno 18 grešaka u prevodiocu.

U toku testiranja primećen je određen broj situacija u kojima prevodilac ne generiše opimalan kod. U 49 takvih slučajeva je uklonjeno emitovanje nepotrebnih instrukcija iz *emit* dela pojedinih pravila. Time je značajno povećana efikasnost prevodioca.

Takodje, ispitivanjem je otkrivena i jedna greška u asembleru, kao i dve greške u prevodiocu koje su bile posledice grešaka koje su postojale u alatu za razvoj prevodioca.

Dodavanjem novih *neterminala* i novih pravila prilikom razvoja prevodioca neka pravila, odnosno neke grane *emit* dela pravila su zastarela. Ispitivanjem su pronađena ta nepotrebna pravila, odnosno nepotrebne grane i izostavljene čime je dobijeno na preglednosti koda. Izostavljeno je 44 pravila i 47 grana pravila.

6. ZAKLJUČAK

Na osnovu broja pronađenih grešaka prilikom prethodno opisanog ispitivanja prevodioca može se zaključiti da je ovakvim ispitivanjem značajno poboljšana pouzdanost prevodioca.

Takođe, na osnovu broja smanjenih instrukcija koji se javio kao posledica ispitivanja može se zaključiti da je značajno povećan kvalitet generisanog asemblerskog koda.

Preglednost i jednostavnost koda je takođe povećana ispitivanjem što se može zaključiti na osnovu broja pravila koja su izostavljena, odnosno na osnovu broja izostavljenih grana u *emit* delu pravila.

LITERATURA

[1] Vladimir Kovačević, Miroslav Popović. “Sistemska programska podrška u realnom vremenu,” *Univerzitet u Novom Sadu, Fakultet Tehničkih Nauka.*

[2] Emmelmann, Helmut, Friedrich-Wilhelm Schröer, Rudolf Landwehr. 1989 (July). “BEG --- a generator for efficient back ends,” *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, in SIGPLAN Notices,* 24(7):227--237.

[3] ACE Associated Compiler Experts bv. *SuperTest*. Ref. CoSy-8052-supertest, 2003.

[4] ACE Associated Compiler Experts bv. *BEG-CoSy Manual*. Ref. CoSy-8005-beg, 2003.

[5] Micronas. *APX Design Specification*. 2004.

Abstract – This paper presents automatic testing of C compiler for one class of digital signal processors. Significance of compiler testing is emphasized, as well as problems occured during its testing. The testing procedure is presented, too.

AUTOMATIC TESTING OF C COMPILER FOR ONE CLASS OF DIGITAL SIGNAL PROCESSORS

Snežana Crnogorac, Aleksandar Simeonov, Zoran Jovanović