Filip Miletić

Analiza zavisnosti iskaza u programskom kodu

Precizna analiza zavisnosti iskaza u izvornom kodu je suština efikasnosti paralelizatorskog prevodioca [1]. *Ideja je da se u izvornom kodu programskog jezika koji je po definiciji sekvencijalan uoče celine koje se mogu izvršavati nezavisno i da se te celine prevedu u odgovarajucse paralelne konstrukcije. Pokazano je u* [1] *da je u opštem slučaju problem nerešiv, ali da su posebni i rešivi slučajevi u praksi razmerno česti. U ovom radu predstavljen je jedan skup tehnika koji se koriste za analizu tih posebnih slučajeva. One su ugrađene u Simple-analizator zavisnosti iskaza programskog koda.*

Realizacija

Načinjen je Simple – jednostavni analizator zavisnosti iskaza u jednostrukoj petlji. Simple je dvorprolazni parser izveden uglavnom kao konačni automat sa stanjima koja se mogu opisati u proširenoj Bakus-Naurovoj formi (EBNF). U njega su ugrađene ovde opisane osnovne tehnike koje se koriste kod optimizacije programa za paralelno izvršavanje, a koje se po pravilu oslanjaju na izračunavanje mogućih zavisnosti među obraćanjima elementima nizova u petlji. Simple izvodi tačno testove na jednodimenzionim nizovima i na visedimenzionim nizovima ukoliko njihovi indeksi imaju osobinu razdvojivosti [1].

Urađena je i elementarna analiza simboličkih izraza. Ona nije uvek moguća jer se ne može uvek predvideti vrednost konstanti sa kojima se ulazi u ciklus. U tom slučaju, analizator samo ispisuje odgovarajuću poruku i ne nastavlja sa analizom iskaza.

Ne uzimaju se u obzir indeksni izrazi koji nisu linearna funkcija brojača u petlji kao ni izrazi koji zavise od promenljive koja menja svoju vrednost u telu petlje. U teoriji nije poznat način kojim bi se u opštem slučaju mogli analizirati takvi iskazi [2].

Unapređenja programa Simple treba tražiti u jos jednoj metodi za utvrđivanje zavisnosti. Ona se sastoji u konstrukciji usmerenog grafa koji

Filip Miletić (1978), Kruševac, Ratka Šakotića 39, učenik 3. razreda Gimnazije u Kruševcu

ih opisuje. Pretragom ovog grafa može se ustanoviti koji su delovi koda međusobno nezavisni. Oni bi bili predstavljeni granama grafa koje su disjunktni skupovi. Ustrojstvo grafa omogućava da se ovim granama lako manipuliše i njihov raspored u kodu menja što može uticati da se optimizacijom dođe do efikasnijeg koda. Očigledan nedostatak je što je na taj način nemoguće analizirati reference na nizove. Zbog toga ovaj metod treba shvatiti kao dopunu ovde izloženih postupaka.

Jezik Simple

Simple nije programski jezik u punom smislu, već je forma koja se koristi za opisivanje ulaznih podataka u obliku bliskom programerima, jer je njegova gramatika podskup one u jeziku Paskal. Sintaksni analizator je izveden kao konačni automat. Prikupljanje ključnih reči i identifikatora izvodi se za to namenjenim primitivama. Ustanovljene su vrste leksema koje se razmatraju. Gramatika jezika Simple je sledeća:

```
$ SimpleProgram = "Simple" "Program" Identifier ";"
  ProgramBody "." .
$ ProgramBody = [ VarDeclaration ";"] StatementBlock .
$ VarDeclaration = "Var" VarDeclarationSequence .
$ VarDeclarationSequence = VarAssignment
  { ";" VarAssignment } .
$ VariableType = ArrayType | SimpleType .
$ ArrayType = "Array" "[" Range { "," Range} "]"
  "of" SimpleType .
$ Range = IntConstant ".." IntConstant.
$ SimpleType = integer .
$ StatementBlock = "begin" StatementSequence "end" .
$ StatementSequence = Statement { "," Statement } .
$ Statement = AssignmentStatement | StatementBlock |
  ForStatement .
$ AssignmentStatement = VariableReference
  ":=" Expression .
$ VariableReference = Identifier { VariableField } .
$ VariableField = "[" Expression {"," Expression } "]" .
$ Expression = [ "+" | "-" ] Term
  { AditiveOperator Term } .
$ AditiveOperator = "+" | "-" | "OR" .
$ Term = Factor { MultiplicativeOperator Factor } .
$ MultiplicativeOperator = "*" | "MOD" | "DIV" .
$ Factor = UnsignedConstant | VariableReference |
  "(" Expression ")".
$ ForStatement = "For" Identifier ":=" Expression
  ( "to" | "downto" ) "do" Statement .
```

Metode analize iskaza

Razmatraju se čitanja i pisanja u nizove koja se nalaze u telu ciklusa. Simple za sada sprovodi analizu odgovarajućih indeksa u izrazima. Tako, ako se analizira višedimenzioni niz, razmatraće se samo odgovarajući indeksi. Ukoliko ma koji od testova javi nezavisnost to znači da zavisnosti nema ni u ostalim indeksima.

Da bi dva iskaza bila nezavisna, mora biti zadovoljena jednakost:

(Out(i) In(j)) (In(i) Out (j)) Out(i) Out(j)) =

gde je In(i) skup svih ulaznih promenljivih za neku iteraciju i, a Out(i) skup svih izlaznih promenljivih. Rezultat testova je bilo vrednost brojača za koju se ispoljava zavisnost dveju referenci na niz a koje ce u ovom tekstu biti označavane kao i, bilo kao maksimalno rastojanje između dveju vrednosti brojača za koje te zavisnosti nema, sto ovde označavamo kao d.

Cilj ovakve analize je da u programskom kodu otkrije ono što se naziva paralelizmom na nivou instrukcije. Njom se ne može nadomestiti efikasnost programa koja se moze dobiti njegovim pažljivim projektovanjem uz primenu odgovarajućih algoritama.

ZIV test

Najjednostavniji metod analize je označen u [1] kao zero index variable test (ZIV). Indeksi koji zadovoljavaju ZIV test sadrže samo simbole invarijantne u petlji koja se analizira. To uključuje konstante i promenljive cija se vrednost ne menja u datoj petlji. Tipični ZIV test proverava jedino da li postoje slučajevi u kojima se podudaraju odgovarajući indeksi. Ukoliko takvih slučajeva nema, dva se iskaza smatraju nezavisnima. Podudarnost se utvrđuje oduzimanjem izraza koji čine indeks. Ukoliko je ova razlika neka ne-nulta vrednost onda su iskazi nezavisni.

Grupa single index variable (SIV) testova

Strong SIV

Kao Strong SIV [1] indeksi označavaju se elementi uređenog para poput $ai + c_1$; $ai + c_2$. Rastojanje se definiše kao:

$$d = i - i' = \frac{c_2 - c_1}{a_1} ,$$

uz uslov:

$$\left| d \right| \quad U \quad L ,$$

gde su U i L gornja i donja granica petlje respektivno. Ono sto je značajno kod ovog testa je sto se on moze lako proširiti tako da uključi i simboličku proveru iskaza. Program sa slike 1 će prema zadatom obrascu utvrditi nezavisnost izraza a[i+2*N] i a[i+N], utvrđujući za parametar *d* nejednakost:

N >>čNč-č1

Pošto je uslov za svaku vrednost *N* ispunjen, test utvrđuje nezavisnost iskaza. Ovako utvrđena nezavisnost garantuje da se svaka iteracija u programu sa slike moze izvršavati paralelno sa ostalima tako da je polazna jednakost zadovoljena.

```
Simple program Strong_SIV_Test;

var i, N : Integer;
    A: array [1..1000] of integer

begin
    N:=50;
    for i := 1 to N do
        a[i+2*N] := a[i+N] + 3
end.
```

Weak-Zero SIV

Ovaj test se izvodi nad uređenim parom indeksa tipa , gde samo jedan element uređenog para zavisi od brojača u petlji. Tada je moguće pronaći samo jednu iteraciju *i* u kojoj se javlja zavisnost i ona se dobija kao resenje jednačine:

$$ai + c_1 = c_2 ,$$

$$i = \frac{c_2 - c_1}{a} ,$$

pri čemu vrednost i mora biti ceo broj u okviru granica petlje. Tako će za kod:

```
Simple program Weak_Zero_SIV_Test;

var i, N : Integer;
    A: array [1..10] of integer
```

```
begin
    N:=10;
    for i := 1 to N do
        a[i] := a[1] + a[N]
end.
```

Simple dati da su zavisne iteracije broj 1 i N. Pronađena zavisnost uslovljava da kod prevođenja programa moramo obezbediti da se povezane iteracije izvrše onim redom kojim bi se izvršile u sekvencijalnom programu. Zato Simple predlaze da se kod sa slike prepiše na sledeći način, tako da se zavisne iteracije izdvoje iz ciklusa:

```
Simple program Weak_Zero_SIV_Test;

var i, N : Integer;
    A: array [1..10] of integer

begin
    N:=10;
    a[1]:=2*a[1];
    for i := 2 to N-1 do
        a[i] := a[1] + a[N];
    a[N]:=a[1]+a[N]
end.
```

Sada se u ciklusu nalaze sve same nezavisne iteracije što se može iskoristiti za paralelizaciju koda.

Weak crossing SIV

Odnosi se na test kod koga su uređeni parovi podneti na testiranje oblika ($ai + c_2; -ai + c_2$). Zavisna iteracija se nalazi u [1] kao:

$$i = \frac{c_2 - c_1}{2\,a} ,$$

iz uslova da zavisnost postoji kada je tačna jednakost

$$ai + c_1 = -ai + c_2 ,$$

gde i mora biti bilo ceo broj u okviru granica petlje bilo broj sa razlomljenim delom jednakim 0.5. Ovde Simple predlaze razbijanje ciklusa na dva, koji onda svaki za sebe sadrzi nezavisne iteracije. Uslov je jedino da se nikoje dve iteracije iz ova dva ciklusa ne mogu izvršavati paralelno jedna sa drugom. Ukoliko vrednost i ne ispunjava gore navedeni uslov, utvrđuje se nezavisnost iskaza.

SIV test u opštem slučaju

Problem SIV indeksa u opštem slučaju, kako je pokazano u [2] svodi se na rešavanje diofantske jednačine sa dve nepoznate. Neka ispitujemo sledeća dva obraćanja nizu u jednoj iteraciji, a[m*i + n] i a[p*i + q] gde je i brojač. Neka je prva referenca na element niza sa leve strane operatora dodele, a druga sa njegove desne strane. Ako označimo sa I i K vrednosti brojača u ciklusu za koje se reference obraćaju istom elementu niza, imamo:

$$mI + n = pK + q \ .$$

Rešavanjem ove jednačine po I dobijamo:

$$I = rK + s \ ,$$

gde je, zbog preglednosti smenjeno $r = \frac{p}{m}$ i $s = \frac{q - n}{m}$. Pretpostavimo za potrebe primera da je $r \neq 0$ i s >>č0. Gornji izraz predstavlja jednačinu prave u ravni sa koordinatnim osama I i K. Bilo koja tačka koja leži na toj pravoj i ima celobrojne kordinate je rešenje koje trazimo. Parove (I, K), $I, K \in Z$ nalazimo kao rešenja ove diofantske jednačine sa dve nepoznate. Metod po kome se ovo izvodi je poznat u matematici. Implementacija postupka za nalaženje rešenja jednačine dat je u prilogu. Maksimalno rastojanje među nezavisnim iteracijama se zatim lako nalazi kao:

$$d = \min\left\{ \left| K_n - I_n \right| \right\} \ .$$

Ukoliko presek pravih $I(K)$ i K izlazi van okvira koga zadaju gornja (U) i donja granica (L) iteracije, može se izvršiti dodatna provera kako bi se ustanovilo postoji li neko drugo d^* >>čd za koje je

$$L \leq I \leq U \text{ i } L \leq K \leq U \ .$$

Zaključak

Na primerima smo pokazali kako se izvodi analiza zavisnosti iskaza u programskom kodu. Svi primeri i tehnike koje smo opisali odnose se ne bez razloga upravo na ispitivanje zavisnosti kod referisanja elemenata nizova. Naime, slične operacije sa podacima često se sreću kod programa koji izvode naučna izračunavanja i operacije sa vektorima i matricama. Mogućnost da se paralelizuje ovakav zahtevan račun znači skraćenje praznog hoda koji nastaje u istraživanju kada se čekaju rezultati i korisnije

utrošeno tako uštedeno vreme. U isto vreme, dobija se mogućnost da se resursi paralelnih računara koriste a da se ne poseže eksplicitno za često zamršenim paralelnim algoritmima. Time se, naravno, ne dobija puna efikasnost jer je ručno optimizovan program bolji od ma kakve mašinske optimizacije. Ipak, postignuti kompromis je značajan za prirodnjaka koji stare algoritme može primeniti na novom računaru a da ne mora da poznaje detalje njegove građe.

Literatura

[1] Maydan, D. E., Hennessy, J. L., Lam, M. S. 1991. Efficient and Exact Dependence Analysis. *Preceedings of the ACM SIGPLAN 91 Conference on Programming languanges design and implementation*. Toronto

[2] Goff, G., Kennedy, K., Tseng, C. W. 1991. Practical Dependence Testing. *Preceedings of the ACM SIGPLAN 91 Conference on Programming languages design and implementation*. Toronto

Filip Miletić

Source Code Dependence Analysis

A fast and exact dependence test of source code expressions is of a vital importance for building an effective parallelizing compiler. The idea is to isolate a block of expressions from the sequential source code which can be executed independently and translate those expressions to their parallel equivalents. It is shown in [1] that the general case of dependence testing cannot be solved efficiently. However, it is shown that special cases that we are capable of solving exactly are likely to appear in scientific programs. This paper presents a set of some exact but yet efficient techniques regarding dependence tests within array references that are used in parallel compilers. The techniques are implemented in Simple, source code dependence analyzer. Simple uses a subset of Pascal grammar to describe input data.

Prilog: Rešavanje diofantske jednačine sa dve nepoznate; implementacija u jeziku C

```
#include <stdio.h>
#include <stdlib.h>

#define ABS(x) ( (x>=0) ? x : -x)

typedef struct {
  long a, b, c;
} DiophEquation;
typedef struct {
  long coef1, const1, coef2, const2;
} DiophSolution;

long DiophLevel = 0;

void Swap(long *a, long *b) {
  long t;
  t = *a;
  *a = *b;
  *b = t;
}
long NZD(long a, long b) {
 while (a != b)
  if (a > b)
   a -= b;
  else
   b -= a;
 return a;
}
void SolveDiophantine(DiophEquation *e, DiophSolu-
tion *s,
long *Solvable) {
  long CanSolve;
  long nzd, ValuesSwapped = 0;
  long FreeCoef;

  DiophEquation *SubEquation = NULL;
  DiophSolution *SubSolution = NULL;

  GlobalDiophLevel++;
  if (ABS(e->a) > ABS(e->b)){
    Swap(&(e->a), &(e->b));
    ValuesSwapped = 1;
  }
  CanSolve = 1;
  if ((nzd = NZD(ABS(e->a), ABS(e->b))) > 1){
    if (e->c % nzd != 0){
      CanSolve = 0;
    }else{
```

```
      e->c /= nzd; e->b /= nzd; e->a /= nzd;
    }
  }
  if (CanSolve == 1)
    if ((ABS(e->a) != 1) && (ABS(e->b) != 1)){
      SubEquation = malloc(sizeof (DiophEquation));
      SubSolution = malloc(sizeof (DiophSolution));
      FreeCoef       = -e->b / e->a;
      SubEquation->b =  e->b % e->a;
      SubEquation->a =  e->a;
      SubEquation->c =  e->c;
       SolveDiophantine(SubEquation, SubSolution,
&CanSolve);
      if (CanSolve != 0){

         s->coef1  = FreeCoef *
                     (SubSolution->coef2) -
                       ((e->b % e->a) * SubSolution-
>coef2) /
                     (e->a);
         s->const1 = FreeCoef *
                     (SubSolution->const2) +
                      (-(e->b % e->a) * SubSolution-
>const2 + (e->c))
                     / e->a;
         s->coef2 = SubSolution->coef2;
         s->const2 = SubSolution->const2;
      }
      free(SubEquation);
      free(SubSolution);
   }else{
     if (ABS(e->a) == 1){
       if (ABS(e->b) == 1){
         s->coef1 = 1;
         s->const1 = 0;
         s->coef2 = -e->a;
         s->const2 = (e->c) * (e->a);
       } else {
         s->coef2 = 1;
         s->const2 = 0;
         s->coef1 = -(e->b) * (e->a);
         s->const1 = (e->c) * (e->a);
       }
   } else {
     s->coef1  = 1;
     s->const1 = 0;
     s->coef2  = e->a * e->b;
     s->const2 = e->c * e->b;
   }
 }
 *Solvable = CanSolve;
```

```
 if (ValuesSwapped == 1){
  Swap(&(s->coef1), &(s->coef2));
  Swap(&(s->const1), &(s->const2));
 }
 DiophLevel--;
}
int DiophSolve(void) {
 DiophEquation e1; DiophSolution s1; long IsSolved;
 printf("Solving Diophantine equation: ");
 printf("Equation is: A*x + B*y = C\n\n");
 printf(" A = "); scanf("%d", &e1.a);
 printf(" B = "); scanf("%d", &e1.b);
 printf(" C = "); scanf("%d", &e1.c);
 SolveDiophantine(&e1, &s1, &IsSolved);
 if (IsSolved != 0)
  printf("Solved for (x, y) = ( %d*t%+d, %d*t%+d), t
is in Z\n\n",
       s1.coef1, s1.const1, s1.coef2, s1.const2);
 else
  puts("System has no integer solutions.");
 return 0;
}
```