Limbaje formale si tehnici de compilare

Laborator 9

Pentru functia de mai jos:

```
int sum()
{
    int  i,v[5],s;
    s=0;
    for(i=0;i<5;i=i+1){
        v[i]=i;
        s=s+v[i];
        }
    return s;
}
```

Generatorul de cod va genera urmatoarele instructiuni (pe 32 de biti):

```
    ENTER       28    // 4+20+4 bytes local vars
    PUSHFPADDR  24    // s=0
    PUSHCT_I    0
    INSERT      8,4
    STORE       4
    DROP        4
    PUSHFPADDR  0     // i=0
    PUSHCT_I    0
    INSERT      8,4
    STORE       4
    DROP        4
L2: PUSHFPADDR  0     // if(!(i<5))goto L1
    LOAD        4
    PUSHCT_I    5
    LESS_I
    JF_I        L1
    PUSHFPADDR  4     // v[i]=i
    PUSHFPADDR  0
    LOAD        4
    PUSHCT_I    4
    MUL_I
    OFFSET
    PUSHFPADDR  0
    LOAD        4
    INSERT      8,4
    STORE       4
    DROP        4
    PUSHFPADDR  24    // s=s+v[i]
    PUSHFPADDR  24
    LOAD        4
    PUSHFPADDR  4
    PUSHFPADDR  0
    LOAD        4
    PUSHCT_I    4
    MUL_I
    OFFSET
    LOAD        4
    ADD_I
    INSERT      8,4
    STORE       4
    DROP        4
    PUSHFPADDR  0     // i=i+1
    PUSHFPADDR  0
    LOAD        4
    PUSHCT_I    1
    ADD_I
    INSERT      8,4
    STORE       4
    DROP        4
    JMP         L2
L1: NOP
    PUSHFPADDR  24    // return s
    LOAD        4
    RET         0,4
```

La analiza acestui cod constatam ca exista mai multe posibilitati de optimizare. Daca avem posibilitatea sa modificam codul MV, am putea de exemplu sa inlocuim secventele repetitive de instructiuni cu o singura instructiune care sa realizeze acelasi lucru. In acest fel se reduce numarul de iteratii in bucla „while(1){switch(IP->opcode){…}}” din

functia „run” a MV si deci un mai mare procent din timp va fi alocat executiei unei instructiuni, reducandu-se timpul necesar decodificarii instructiunilor. In acelasi timp scad accesele la memorie si operatiile cu stiva, deci inca un castig.

De exemplu, secventa „PUSHCT_I 4; MUL_I; OFFSET” de fapt are rolul sa inmulteasca o valoare cu constanta „4” si apoi sa adauge rezultatul la o adresa. Astfel se calculeaza adresa lui „v[i]”, pornind de la adresa de baza a lui „v” si considerand un element de tipul „int” care in acest caz are 4 (sizeof(long int)) octeti: &v[i]=(char*)v+i*4. Tinand cont ca este posibil ca multi vectori sa aiba elemente de tip „int”, aceasta secventa este suficient de generala ca sa adaugam la MV o instructiune care sa faca acelasi lucru. Vom denumi aceasta instructiune „OFFSET_I” si ea va avea asupra stivei urmatoarea actiune: [addr, i] -> [addr+i*sizeof(long int)].

Ca sa putem inlocui cele trei instructiuni prin „OFFSET_I”, trebuie sa ne asiguram ca a doua si a treia instructiune din secventa nu sunt adresele de destinatie (tintele) unor instructiuni gen „JMP”, „JT”, „JF” sau ”CALL”. Intr-adevar, daca am avea urmatorul cod:

```
        JMP     L1
        ...
        PUSHCT_I 4
L1:     MUL_I
        OFFSET
```

Inseamna ca uneori se poate executa „MUL_I; OFFSET” fara sa se execute „PUSHCT_I 4”, adica nu in toate cazurile secventa celor trei instructiuni este indivizibila. In aceste situatii nu se poate inlocui secventa printr-o instructiune. In mod formal spunem ca aceasta secventa trebuie sa faca parte din acelasi **bloc de baza** (basic bloc, BB). Un BB este o secventa de instructiuni care poate incepe doar cu prima sa instructiune (nu exista salturi la instructiuni interioare dintr-un BB), fiecare instructiune continua cu cea de dupa ea din lista de instructiuni si BB se termina intotdeauna cu ultima sa instructiune (nu exista salturi din BB, dar pot exista „CALL” la functii care revin in BB dupa „CALL”).

Pentru a testa daca „MUL_I; OFFSET” fac parte din acelasi BB ca „PUSHCT_I 4”, acestea nu trebuie sa fie adresele de destinatie a unor instructiuni de salt/apel. Pentru acest test vom folosi urmatoarele functii:

```
int     needTargetInstr(Instr *crt)
{
    switch(crt->opcode){
        case O_CALL:
        case O_JF_A:case O_JF_C:case O_JF_D:case O_JF_I:
        case O_JMP:
        case O_JT_A:case O_JT_C:case O_JT_D:case O_JT_I:
            return 1;
        default:
            return 0;
        }
}

int     isTarget(Instr *crt)
{
    Instr  *i;
    for(i=instructions;i;i=i->next){
        if(needTargetInstr(i)&&i->args[0].addr==crt)return 1;
        }
    return 0;
}
```

Functia „isTarget” itereaza prin toate instructiunile si verifica daca o instructiune care necesita o adresa de destinatie are destinatia „crt”. La un cod mai mare, aceasta metoda de a cauta in toate instructiunile poate deveni prohibitiva, dar pentru AtomC ea este suficienta si are avantajul de a fi simplu de implementat.

Deoarece „PUSHCT_I 4” poate fi inceputul unui BB, daca o vom sterge vom avea nevoie de o functie care sa mute toate adresele de destinatie care o pointeaza la o alta instructiune:

```
void	moveTarget(Instr *src,Instr *dst)
{
    Instr	*i;
    for(i=instructions;i;i=i->next){
        if(needTargetInstr(i)&&i->args[0].addr==src){
            i->args[0].addr=dst;
            }
        }
}
```

In sfarsit, vom avea nevoie de o functie care sa stearga o instructiune din lista de instructiuni:

```
void	deleteInstr(Instr *i)
{
    Instr	*last=i->last,*next=i->next;
    if(last==NULL){
        instructions=next;
        }else{
        last->next=next;
        }
    if(next==NULL){
        lastInstruction=last;
        }else{
        next->last=last;
        }
    free(i);
}
```

Cu aceste functii auxiliare putem scrie functia de optimizare a secventei discutate:

```
// PUSHCT_I sizeof(long int); MUL_I; OFFSET -> OFFSET_I
void	passOffset()
{
    Instr	*i1,*i2,*i3;
    for(i1=instructions;;i1=i2){
        i2=i1->next;
        if(!i2)break;
        if(i1->opcode!=O_PUSHCT_I||i1->args[0].i!=sizeof(long int))continue;
        if(i2->opcode!=O_MUL_I)continue;
        if(isTarget(i2))continue;
        i3=i2->next;
        if(!i3||i3->opcode!=O_OFFSET)continue;
        if(isTarget(i3))continue;
        i3->opcode=O_OFFSET_I;
        moveTarget(i1,i3);
        deleteInstr(i1);
        deleteInstr(i2);
        i2=i3;
        optimized=1;
        }
}
```

In general o optimizare este realizata la o „trecere” (pass) prin cod, de aceea numele functiei incepe cu „pass”. Pentru fiecare instructiune, se testeaza daca ea este inceputul secventei cerute, se elimina cazurile care nu se pot optimiza si in final se face optimizarea prin stergerea primelor doua instructiuni din secventa si modificarea „opcode”-ului celei de a treia instructiuni din „OFFSET” in „OFFSET_I”. Evident in MV va trebui definita instructiunea

„OFFSET_I”. „optimized” este o variabila globala care se seteaza daca a avut loc o optimizare. De multe ori o optimizare deschide calea altor optimizari si atunci, daca au avut loc optimizari e bine sa mai rulam o data secventa de treceri prin cod. Functia care aplica toate optimizarile este de forma:

```
void	optimize()
{
	do{
		optimized=0;
		passOffset();
		//…other passes…
		}while(optimized);
}
```

O alta optimizare posibila, care nu necesita interventia in MV, este urmatoarea: „INSERT sizeof(void*)+n,n; STORE n; DROP n” -> „STORE n”. Instructiunea „INSERT” este generata in cazul instructiunilor de atribuire „a=val”, pentru a se duplica valoarea de pe stiva (val) in cazul de atribuire multipla „a=b=val”. „DROP” se genereaza la sfarsitul unei expresii non-void cand apare „;”, deoarece la generarea de cod orice expresie trebuie sa lase pe stiva valoarea ei, iar ultima valoare, care nu este folosita, trebuie stearsa de pe stiva. Rezulta ca este inutil sa duplicam o valoare daca dupa aceea oricum stergem duplicatul nou creat. Functia de optimizare a acestei secvente este:

```
// INSERT sizeof(void*)+n,n; STORE n; DROP n -> STORE n
void	passDelDuplications()
{
	Instr	*i1,*i2,*i3;
	int	n;
	for(i1=instructions;;i1=i2){
		i2=i1->next;
		if(!i2)break;
		if(i1->opcode!=O_INSERT)continue;
		n=i1->args[1].i;
		if(i1->args[0].i!=sizeof(void*)+n)continue;
		if(i2->opcode!=O_STORE||i2->args[0].i!=n)continue;
		if(isTarget(i2))continue;
		i3=i2->next;
		if(!i3||i3->opcode!=O_DROP||i3->args[0].i!=n)continue;
		if(isTarget(i3))continue;
		moveTarget(i1,i2);
		deleteInstr(i1);
		deleteInstr(i3);
		optimized=1;
		}
}
```

Aplicatia 9.1: *Sa se implementeze o optimizare care sterge toate instructiunile „NOP”.*

Aplicatia 9.2: *Sa se implementeze o optimizare care inlocuieste secventa „PUSHFPADDR i; LOAD sizeof(long int)” cu „PUSHFP_I i”, unde „PUSHFP_I i” este o noua instructiune a MV care depune pe stiva o valoare intreaga de la indexul (fata de FP) specificat.*

Dupa ce toate optimizarile au fost adaugate in functia „optimize”, optimizarile se pot testa atat ca timp de executie cat si ca numar de instructiuni executate. Pentru a masura timpul avem nevoie de o functie cu o precizie cat mai mare, care in C poate fi implementata folosind biblioteca standard astfel:

```
double		myTime()
{
    return (double)clock()/(double)CLOCKS_PER_SEC; // in seconds
}
```

Pentru a contoriza instructiunile, vom modifica functia „run” ca ea sa returneze numarul instructiunilor executate (int). Acestea se pot contoriza cu un contor local care se incrementeaza inainte de „switch(IP->opcode)”. La „HALT” se va returna acest contor. Deoarece functiile „printf” din „run” ar impiedica testarea vitezei, trebuie sa avem o metoda de a le anula la dorinta. Pentru aceasta le putem prefixa pe toate cu cate un „if” care sa permita afisarea sau nu: „if(RUN_PRINT)printf(...);”, unde „RUN_PRINT” este un „#define” cu valori 0 sau 1.

Functia „run” va fi apelata de forma:

```
t1=myTime();
executedInstructions=run(instructions);
t2=myTime();
printf("time=%g sec (%d executed instructions)\n",t2-t1,executedInstructions);
```

cu sau fara optimizari si se vor compara rezultatele. In cazul compilatoarelor de C este interesant de testat si cat optimizeaza compilatorul in sine functia „run”. Pentru aceasta se poate rula AtomC fara optimizari din partea compilatorului (ex: gcc -O0) si apoi AtomC optimizat de compilator (ex: gcc -O3).
Deoarece functia „sum” de la inceputul laboratorului se executa foarte repede, rularea ei va trebui repetata de mai multe ori in „main”-ul programului de testat, in asa fel incat o rulare totala sa fie de ordinul secundelor:

```
void main()
{
    int     i,s;
    for(i=0;i<1000000;i=i+1)
        s=sum();
    put_i(s);
}
```