Optimizarea performanțelor unui program în limbajul C

1. Algoritmul este descris într-o formă adecvată arhitecturii microcontrolerului (se iau în considerare elementele de paralelism, artimetica utilizată de microcalculator, modurile de adresare, tipurile de instrucțiuni).
2. Codificarea algoritmului (scrierea programului) se realizează cât mai simplu din punct de vedere al codului şi a strucrurilor de date utilizate.
3. Secțiunile critice ale programului se pot rescrie în limbaj de asamblare (pentru minimizarea timpului de execuție şi a dimensiunii codului)
4. Se evită efectuarea calculelor ce vor fi emulate (operații aritmetice pentru care microcontrolerul nu are intrucțiune nativă – împărțire, calcule în virgulă mobilă)
5. Utilizarea analizei interprocedurale

    - definirea constantelor

De exemplu pentru urmatorul program:

```
#include <stdio.h>
int val; // initialized to zero
void init() {
val = 3; // re-assigned
}
void func() {
printf("val %d",val);
}
int main() {
init();
func();
}
```

prin analiza interprocedurala nu se poate determina faptul ca val este o constanta. Pentru acest lucru programul trebuie rescris astfel:

```
#include <stdio.h>
const int val = 3; // initialized once
void init() {
}
void func() {
printf("val %d",val);
}
int main() {
init();
func();
}
```

- evitarea alierii pointerilor

Fie urmatorul program:

```
void fn(char a[], char b[], int n) {
int i;
for (i = 0; i < n; ++i)
a[i] = b[i];
}
```

Variabilele de tip pointer a si b pot indica acelasi tablou (fenomenul se numeste alierea pointerilor). Prin analiza interprocedurala compilatorul va incerca sa determine situatiile de aliere si sa le rezolve.

Daca functia fn este apelata de doua ori intr-un program, analiza interprocedurala poate determina urmatoarele rezultate:

```
fn(glob1, glob2, N);
fn(glob1, glob2, N);
```

In acest caz pointerii a si b nu se intersecteaza (nu apare alierea).

```
fn(glob1, glob2, N);
fn(glob3, glob4, N);
```

In acest caz pointerii a si b nu se intersecteaza (nu apare alierea).

```
fn(glob1, glob2, N);
fn(glob3, glob1, N);
```

In acest caz pointerii a si b pot indica acelasi tablou (poate apare alierea).

Aeasta situatie apare deoarece analiza interprocedurala ia in considerare toate apelurile de functii in mod global si nu individual, atunci când determina riscul aparitiei alierii. Daca apelurile ar fi luate in considerare individual analiza interprocedurala trebuie sa analizeze fluxul de intructiuni sis a ia in considerare un numar mare de permutari posibile, ceea ce ar mari foarte mult timpul de compilare.

Evitarea alierii pointerilor se poate realize şi prin utilizarea calificatorilor restrict si const.

Secventa:

```
for (i=0; i<100; i++)
a[i] = b[i];
```

poate fi rescrisă pentru a elimina alierea pointerilor a si b astfel:

```
int * restrict p = a;
int * restrict q = b;
for (i=0; i<100; i++)
*p++ = *q++;
```

Calificatorul restrict precizeaza compilatorului că accesele la memorie nu se vor suprapune. Utilizarea calificatorului const este ilustrată de secvența:

```
void copy(short *a, const short *b) {
int i;
for (i=0; i<100; i++)
a[i] = b[i];
    }
```

6. Utilizarea tablourilor indexate sau a pointerilor, după caz

   Limbajul C permite ca accesarea datelor sa se efectueze fie prin indexarea unui tablou, fie incrementând un pointer la tablou. Cele două posibilități sunt prezentate în continuare:

   - cu tablouri indexate:

```
void va_ind(const short a[], const short b[], short out[], int n) {
int i;
for (i = 0; i < n; ++i)
out[i] = a[i] + b[i];
}
```

   - cu pointeri

```
 void va_ptr(const short a[], const short b[], short out[], int n) {
int i;
short *pout = out;
const short *pa = a, *pb = b;
for (i = 0; i < n; ++i)
*pout++ = *pa++ + *pb++;
}
```

In funcție de aplicație codul generat la compilare poate fi mai eficient folosind unul dintre cele două stiluri.
Cea mai buna strategie este de a incepe prin scrierea codului folosind indexarea; daca rezultatul compilarii nu este satisfacator se pot folosi pointeri.

7. Utilizarea funcţiilor *inline* sau a limbajului de asamblare (dacă pentru funcția dorită nu există varianta *inline*)

8. Utilizarea de secțiuni diferite de memorie pentru variabilele programului (memorie de program, memorie de date) pentru exploatarea paralelismului microcalculatorului.
9. Reguli pentru scrierea buclelor de program:

    - buclele de program trebuie sa fie cât mai scurte (pentru a putea fi optimizate automat de către compilator)
    - se va evita desfacerea (*unrolling*) manuală a buclelor – în scopul exploatării paralelismului unităților aritmetice ale microcalculatorului; acest lucru se va face automat de către compilator, dacă e posibil.

Desfacerea manuală a buclelor poate face programul mai greu de inteles si poate împiedica realizarea optimizării automate de către compilator.

```
void va1(const short a[], const short b[], short c[], int n)
{
int i;
for (i = 0; i < n; ++i) {
c[i] = b[i] + a[i];
}
}
```

Compilatorul va optimiza bucla.

```
void va2(const short a[], const short b[], short c[], int n)
{
short xa, xb, xc, ya, yb, yc;
int i;
for (i = 0; i < n; i+=2) {
xb = b[i]; yb = b[i+1];
xa = a[i]; ya = a[i+1];
xc = xa + xb; yc = ya + yb;
c[i] = xc; c[i+1] = yc;
}
}
```

Este dificil pentru compilator sa optimizeze bucla.

- evitarea rotirii manulale a buclelor pentru a încărca sau stoca în avans variabile ale programului

```
int ss(short *a, short *b, int n) {
int sum = 0;
int i;
for (i = 0; i < n; i++) {
sum += a[i] + b[i];
}
```

```
return sum;
}
```

Bucla va fi rotită de către compilator.

```
int ss(short *a, short *b, int n) {
short ta, tb;
int sum = 0;
int i = 0;
ta = a[i]; tb = b[i];
for (i = 1; i < n; i++) {
sum += ta + tb;
ta = a[i]; tb = b[i];
}
sum += ta + tb;
return sum;
}
```

Este dificil de optimizat datorită variabilelor auxiliare ta si tb.

- lungimea buclelor interioare trebuie să fie mai mare ca lungimea buclelor exterioare
- se va evita utilizarea instrucțiunilor condiționale în interiorul buclelor

Urmatoarea secventa:

```
 for (i=0; i<100; i++) {
if (mult_by_b)
sum1 += a[i] * b[i];
else
sum1 += a[i] * c[i];
}
```

nu poate fi optimizata; este mai bine ca secventa sa fie rescrisă astfel:

```
if (mult_by_b) {
for (i=0; i<100; i++)
sum1 += a[i] * b[i];
} else {
for (i=0; i<100; i++)
sum1 += a[i] * c[i];
}
```

In acest caz secventa contine două bucle simple ce pot fi optimizate.

- se vor evita apelurile de funcții în interiorul buclelor

- contorul buclei trebuie sa fie incrementat sau decrementat cu factorul 1, în caz contrar optimizarea devine dificilă
- contorul buclei trebuie sa fie variabilă locală pentru ca bucla sa poata fi controlată hardware

Contorul buclei treniie sa fie variabilă locală (care va fi stocată intr-un registru) sin u variabilă globală. Următorul cod:

```
for (i=0; i<globvar; i++)
a[i] = 10;
```

poate necesita reincarcarea variabilei `globvar` la fiecare iterație şi nu va transforma bucla într-o buclă controlată hardware (cu ajutorul unui hardware dedicate ce asigura incrementarea contorului si verificarea termibarii buclei).
Codul poate fi rescris astfel:

```
int upper_bound = globvar;
for (i=0; i<upper_bound; i++)
a[i] = 10;
```

ceea ce permite transformarea buclei într-o buclă hardware.

10. Utilizarea funcțiilor de lucru în aritmetică fracționară(1.15)

Dacă programul necesită efectuarea unor calcule în aritmetică fractionară, ca în exemplul următor:

```
long dot_product (short *a, short *b) {
int i;
long sum=0;
for (i=0; i<100; i++) {
/* this line is performance critical */
sum += (((long)a[i]*b[i]) << 1);
}
return sum;
}
```

este mai eficient să se utilizeze funcțiile aritmetice ale compilatorului pentru opeartii în format fracționar, în locul deplasărilor astfel:

```
#include <math.h>
fract32 dot_product(fract16 *a, fract16 *b) {
int i;
fract32 sum=0;
for (i=0; i<100; i++) {
/* this line is performance critical */
```

```
sum += __builtin_mult_fr1x32(a[i],b[i]);
}
return sum;
}
```

Această secventă utilizeaza în mod eficient arhitectura microcontrolerului.

11. Utilizarea calificativului *pragma* pentru optimizari locale pentru diferite secțiuni ale programului
12. Aplicațiile mici vor fi optimizate pentru dimensiunea codului