

Architektury systemów komputerowych 2017

Lista zadań nr 11

Na zajęcia 24–25 maja 2017

Zadanie 1 (1 pkt.). Poniżej podano funkcję transponującą macierz kwadratową o rozmiarze n . Niestety jej kod charakteryzuje się niską lokalnością przestrzenną dla tablicy `dst`. Używając metody kafelkowania zoptymalizuj poniższą funkcję pod kątem lepszego wykorzystania pamięci podręcznej.

```
1 void transpose(int *dst, int *src, int n) {
2     for (int i = 0; i < n; i++)
3         for (int j = 0; j < n; j++)
4             dst[j * n + i] = src[i * n + j];
5 }
```

Należy uzupełnić ciało funkcji `transpose2` w pliku źródłowym `transpose.c`. Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
# ./transpose -n 15 -v 0
Time elapsed: 21.917466 seconds.
# ./transpose -n 15 -v 1
Time elapsed: 5.416954 seconds.
```

Zadanie 2 (1 pkt.). Wydawałoby się, że w poniższym kodzie źródłem problemów z wydajnością będą dostępy do pamięci. Zauważ, że instrukcje warunkowe w liniach 17, 20 i 23 zależą od losowych wartości. W związku z tym procesorowi będzie trudno przewidzieć czy dany skok się wykona czy nie. Kara za błędną decyzję predyktora wynosi we współczesnych procesorach x86-64 około 20 cykli.

```
1 int randwalk(uint8_t *arr, int n, int len) {
2     int sum = 0, k = 0;
3     uint64_t dir = 0;
4     int i = fast_random() % n;
5     int j = fast_random() % n;
6
7     do {
8         k -= 2;
9         if (k < 0) {
10             k = 62;
11             dir = fast_random();
12         }
13
14         int d = (dir >> k) & 3;
15
16         sum += arr[i * n + j];
17         if (d == 0) {
18             if (i > 0)
19                 i--;
20         } else if (d == 1) {
21             if (i < n - 1)
22                 i++;
23         } else if (d == 2) {
24             if (j > 0)
25                 j--;
26         } else {
27             if (j < n - 1)
28                 j++;
29         }
30     } while (--len);
31
32     return sum;
33 }
```

Podglądając kod wynikowy z kompilatora usuń wszystkie instrukcje skoków z ciała pętli w powyższym kodzie. Skorzystaj z faktu, że kompilator przy spełnieniu pewnych warunków tłumaczy wyrażenie $x = b ? p : q$ z użyciem instrukcji warunkowego kopiowania `cmov`.

Należy uzupełnić ciało funkcji `randwalk2` w pliku źródłowym `randwalk.c`. Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./randwalk -n 7 -s 16 -t 14 -v 0
Time elapsed: 6.480445 seconds.
$ ./randwalk -n 7 -s 16 -t 14 -v 1
Time elapsed: 3.801035 seconds.
```

Zadanie 3 (2 pkt.). Posortowaną dużą tablicę liczb całkowitych będziemy wielokrotnie przeszukiwać używając metody wyszukiwania binarnego. Niestety podany niżej algorytm wykazuje niską lokalność przestrzenną. Dzięki reorganizacji danych w tablicy do struktury kopca binarnego (tj. układamy w pamięci liniowo kolejne poziomy drzewa poszukiwań binarnych) można uzyskać znaczące przyspieszenie — w trakcie prezentacji zadania podaj uzasadnienie. Dla uproszczenia przyjmujemy, że w tablicy jest $2^n - 1$ elementów, tj. zajmujemy się tylko pełnymi drzewami binarnymi.

```
1 bool binary_search(int *arr, int size, int x) {
2     do {
3         size >>= 1;
4         int y = arr[size];
5         if (y == x)
6             return true;
7         if (y < x)
8             arr += size + 1;
9     } while (size > 0);
10    return false;
11 }
```

W pliku źródłowym `randwalk.c` należy uzupełnić ciało funkcji `heapify`, która zmienia ułożenie elementów tablicy na strukturę kopcową, a także `heap_search`, unikając instrukcji skoku warunkowego w ciele pętli. Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./bsearch -n 23 -t 24 -v 0
Time elapsed: 8.101319 seconds.
$ ./bsearch -n 23 -t 24 -v 1
Time elapsed: 3.290986 seconds.
```

Zadanie 4. Tworzymy tablicę 32-bitowych słów T o n elementach. Tablica zaczyna się pod adresem podzielonym przez rozmiar strony i ma długość wielokrotności rozmiaru strony. Mamy zbiór S wszystkich indeksów tej tablicy. Należy wygenerować pewne szczególne permutacje zbioru $U \subseteq S \setminus \{0\}$, tj. ciągi niepowtarzających się indeksów i_1, i_2, \dots, i_l , gdzie $l \leq n$. Ciągi te będziemy reprezentować w tablicy T następująco: $T[0] := i_1$, $T[i_k] := i_{k+1}$, $T[i_l] \in \{0, -1\}$. Procedura `array_walk` w pliku `cache.c` będzie przechodziła kolejno po elementach tablicy T mających indeksy zakodowanego ciągu. Działanie zakończy po osiągnięciu ostatniego elementu ciągu lub po wykonaniu k kroków.

Dobierając odpowiednie permutacje przeglądanie tablicy T będzie generować pewną ilość chybień, które będą wyraźnie zaburzać czas wykonania programu. Na tej podstawie chcemy ustalić następujące parametry podsystemu pamięci:

- **(1 pkt.)** długość linii pamięci podręcznej,
- **(1 pkt.)** rozmiar w bajtach pamięci podręcznej L1 dla danych, L2 i L3,
- **(1 pkt.)** rozmiar zbioru sekcyjno-skojarzeniowej pamięci podręcznej L1 dla danych, L2 i L3,
- **(bonus, 1pkt.)** ilość wpisów w TLB pierwszego poziomu dla danych i TLB drugiego poziomu.

Oczekujemy, że studenci w trakcie prezentacji rozwiązania będą w stanie sprawnie wytłumaczyć w jaki sposób zbadali organizację pamięci podręcznej i na jakiej podstawie wyznaczyli poszczególne parametry. Zebrane wyniki i tok rozumowania muszą wystarczyć do przekonania prowadzącego.

Pamiętaj, że właściwy sposób mierzenia czasu wykonania programu polega na wielokrotnym jego uruchomieniu, odrzuceniu skrajnych pomiarów i uśrednieniu reszty wyników.