Architektury systemów komputerowych

Lista zadań nr 12

Na zajęcia 28 i 29 maja 2018

Rozwiązanie zadań polega na zmodyfikowaniu plików źródłowych dostępnych na stronie przedmiotu oraz napisaniu sprawozdania. Raport ma być pojedynczym plikiem tekstowym w formacie markdown[1] lub LaTeX, do którego można dołączyć wykresy i diagramy. Do każdego zadania dołączono listę pytań, na które należy odpowiedzieć w sprawozdaniu. Zarówno odpowiedź pozytywną jak i negatywną należy właściwie uzasadnić demonstrując posiadaną wiedzę o strukturze pamięci podręcznych. Sprawozdanie musi zawierać również informacje o środowisku, w którym przeprowadzono eksperymenty – patrz przykładowy plik «`raport.md`».

Programy dostarczone przez prowadzącego należy kompilować przy użyciu systemu Linux dla architektury `x86-64`. Powinny również działać pod systemem macOS zbudowane kompilatorem «`clang`». Należy zadbać o niską wariancję wyników uruchomienia polecenia z danym zestawem parametrów. Najłatwiej osiągnąć to minimalizując obciążenie systemu, np. ograniczając liczbę współbieżnie działających procesów.

UWAGA! Uruchamianie eksperymentów pod systemem zainstalowanym w maszynie wirtualnej może poważnie zaburzać wyniki!

Do każdego z rozwiązanych zadań należy dostarczyć wszystkie pliki niezbędne do powtórzenia eksperymentu na komputerze osoby sprawdzającej zadanie. Wyniki umieszczone w raporcie muszą jednoznacznie wspierać prezentowaną tezę – sprawdzający nie ulegnie pokusie optymistycznego naginania rzeczywistości.

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

Wyniki swojej pracy należy wysłać w archiwum «`tgz`» o nazwie «`indeks_imie_nazwisko.tgz`» z użyciem systemu SKOS. Rozpakowanie plików poleceniem «`tar`» ma dać następującą strukturę katalogów:

```
999999_jan_nowak/
  Makefile
  raport.md
  bsearch.c
  cache.c
  common.c
  common.h
  matmult.c
  randwalk.c
  transpose.c
  ...
```

Oceniający zadania używa komputera z zainstalowanym systemem `Debian GNU/Linux 9` dla architektury `x86-64`. Ściąga z systemu SKOS archiwum dostarczone przez studenta, po czym:

- sprawdza poprawność struktury katalogów,
- wykołuje polecenie «`make`» by zbudować pliki wykonywalne (w tym dokument «`pdf`» z pliku «`tex`»),
- czyta raport i sprawdza dostępność plików niezbędnych do powtórzenia eksperymentów,
- czyta treść rozwiązań celem znalezienia usterek i plagiatów,
- powtarza wybrane eksperymenty zgodnie z instrukcjami w raporcie,
- wywołuje polecenie «`make clean`», by usunąć wszystkie pliki otrzymane w procesie budowania.

[1] `https://daringfireball.net/projects/markdown/syntax`

Zadanie 1. Na slajdach do wykładu pt. „Cache Memories” zaprezentowano różne podejścia do implementacji mnożenia dwóch macierzy. Na slajdzie 35 podano trzy rozwiązania o różnej kolejności przeglądania elementów tablicy. Na slajdzie 41 widnieje rozwiązanie wykorzystujące technikę kafelkowania.

Należy uzupełnić ciało procedur «multiply0» ... «multiply3» w pliku źródłowym «matmult.c». Po dodaniu ich implementacji na komputerze testowym uzyskano następujące wyniki:

```
$ ./matmult -n 1024 -v 0
Time elapsed: 3.052755 seconds.
$ ./matmult -n 1024 -v 1
Time elapsed: 0.746337 seconds.
$ ./matmult -n 1024 -v 2
Time elapsed: 9.882309 seconds.
$ ./matmult -n 1024 -v 3
Time elapsed: 0.698795 seconds.
```

Powtórz eksperyment ze slajdu 36 dla rosnących wartości n rozmiaru boku macierzy. Zbierz rezultaty uruchomień do pliku tekstowego i utwórz z nich wykres. Tworzenie wykresów z danych numerycznych przy pomocy narzędzia gnuplot[2] przystępnie wyjaśniono na stronie gnuplot: not so Frequently Asked Questions[3].

Sprawozdanie: Czy uzyskane wyniki różnią się od tych uzyskanych na slajdzie? Z czego wynika rozbieżność między wynikami dla poszczególnych wersji mnożenia macierzy? Jaki wpływ ma rozmiar kafelka na wydajność «multiply3»?

Zadanie 2 (bonus). W pliku źródłowym «matmult.c» do poprzedniego zadania zdefiniowano wartości «A_OFFSET», «B_OFFSET», «C_OFFSET». Dobrane wartości wymuszają, aby macierze nie zaczynały się pod takimi samymi adresami wirtualnymi modulo rozmiar strony. Jeśli po ustawieniu definicji tych wartości na 0 obserwujesz spadek wydajności w kafelkowanej wersji mnożenia macierzy postaraj się wyjaśnić ten fenomen.

Sprawozdanie: Dla jakich wartości n obserwujesz znaczny spadek wydajności? Czy rozmiar kafelka ma znaczenie? Czy inny wybór wartości domyślnych «OFFSET» daje poprawę wydajności?

Wskazówka: Obserwowany efekt najprawdopodobniej wynika z generowania konfliktów w obrębie zbiorów.

Zadanie 3. 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 procedury «transpose2» w pliku źródłowym «transpose.c». Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./transpose -n 4096 -v 0
Time elapsed: 21.528841 seconds.
$ ./transpose -n 4096 -v 1
Time elapsed: 5.251710 seconds.
```

Sprawozdanie: Jaki wpływ na wydajność «transpose2» ma rozmiar kafelka? Czy czas wykonania programu z różnymi rozmiarami macierzy identyfikuje rozmiary poszczególnych poziomów pamięci podręcznej?

[2] http://www.gnuplot.info/
[3] http://lowrank.net/gnuplot/datafile2-e.html

Zadanie 4. Poniższy kod realizuje losowe błądzenie po tablicy. Intuicyjnie źródłem problemów z wydajnością powinny być 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 (np. i7-6700[4]) 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 = n / 2;
5    int j = n / 2;
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 poleceniem «objdump» zamień instrukcje warunkowe z linii 17...29 na obliczenia bez użycia instrukcji skoków warunkowych. Skorzystaj z faktu, że kompilator tłumaczy wyrażenia obliczające wartość porównania dwóch liczb z użyciem instrukcji «SETcc».

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

```
$ ./randwalk -S 0xea3495cc76b34acc -n 7 -s 16 -t 14 -v 0
Time elapsed: 5.943448 seconds.
$ ./randwalk -S 0xea3495cc76b34acc -n 7 -s 16 -t 14 -v 1
Time elapsed: 3.066678 seconds.
```

Opcja «-S» służy do podawania ziarna generatora liczb pseudolosowych. Bez tej opcji każde uruchomienie programu będzie generowało inną tablicę, a zatem i inne wyniki.

Sprawozdanie: Ile instrukcji maszynowych ma ciało pętli przed i po optymalizacji? Ile spośród nich to instrukcje warunkowe? Czy rozmiar tablicy ma duży wpływ na działanie programu?

[4] https://www.7-cpu.com/cpu/Skylake.html

Zadanie 5 (2 pkt.). Posortowaną dużą tablicę liczb całkowitych będziemy wielokrotnie przeszukiwać używając metody wyszukiwania binarnego. Niestety podany niżej algorytm wykazuje niską lokalność przestrzenną. Dzięki zbudowaniu kopca binarnego z elementów tablicy (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 «bsearch.c» należy uzupełnić ciało procedury «heapify», która zmienia ułożenie elementów tablicy na strukturę kopcową, a także procedurę «heap_search». Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./bsearch -S 0x5bab3de5da7882ff -n 23 -t 24 -v 0
Time elapsed: 7.616777 seconds.
$ ./bsearch -S 0x5bab3de5da7882ff -n 23 -t 24 -v 1
Time elapsed: 2.884369 seconds.
```

Sprawozdanie: Czemu zmiana organizacji danych spowodowała przyspieszenie algorytmu wyszukiwania? Czy odpowiednie ułożenie instrukcji w ciele «heap_search» poprawia wydajność wyszukań?

Wskazówka: Rozważ prawdopodobieństwo ponownego użycia elementów tablicy sprowadzonych do pamięci podręcznej.

Zadanie 6. Tablica T przechowuje n elementów typu «int». Zaczyna się pod adresem podzielnym przez rozmiar strony i ma długość wielokrotności rozmiaru strony. S to zbiór wszystkich indeksów tej tablicy. Należy wygenerować pewne szczególne permutacje zbioru $U \subseteq S \setminus \{0\}$, tj. ciągi niepowtarzających się indeksów $i_1, i_2, \ldots, i_l$, gdzie $l \leq n$. Będziemy reprezentować je 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» przechodzi kolejno po elementach tablicy T. Działanie zakończy po osiągnięciu ostatniego elementu ciągu lub po wykonaniu k kroków.

Uważny wybór permutacji pozwala kontrolować liczbę chybień towarzyszących przeglądaniu T. Dodatkowo należy zminimalizować wariancję stosunku kosztu chybienia do kosztu trafienia. Przy tak ostrożnie zaprojektowanych eksperymentach chcemy ustalić następujące parametry podsystemu pamięci:

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

Oczekuje się, że student w trakcie prezentacji rozwiązania będzie w stanie sprawnie wytłumaczyć w jaki sposób zbadał organizację pamięci podręcznej i na jakiej podstawie wyznaczył poszczególne parametry. Zebrane wyniki i tok rozumowania muszą wystarczyć do przekonania prowadzącego.

Jeśli jest taka potrzeba można zmodyfikować listę parametrów linii poleceń przyjmowanych przez program.

Sprawozdanie: Co chcesz pokazać przeprowadzając swój eksperyment? Jak zaprojektowano eksperyment? Jak chcesz wykorzystać pozyskane dane? Jakie jest zadanie wygenerowanej permutacji? Czy koszt chybienia jest stały? Jak poradzono sobie z wyeliminowaniem czynników zakłócających pomiary?

Wskazówka: Mając na uwadze strukturę pamięci DRAM postaraj się zmaksymalizować koszt chybienia w pamięć podręczną.