Architektury systemów komputerowych

Lista zadań nr 12

Na zajęcia 27 maja 2021

Przed przystąpieniem do rozwiązywania zadań należy zapoznać się z [1, §5.1] – [1, §5.9].

UWAGA! W trakcie prezentacji należy być gotowym do zdefiniowania pojęć oznaczonych **wytłuszczoną** czcionką.

W poniższych zadaniach należy kod skompilowany z opcją «-O2» należy prezentować przy pomocy Compiler Explorer[1].

Zadanie 1. Intencją procedury «swap» jest zamiana wartości przechowywanych w komórkach pamięci o adresie «xp» i «yp». Odwołując się do pojęcia **aliasingu pamięci** (ang. *memory aliasing*) wytłumacz czemu kompilator nie może zoptymalizować poniższej procedury do procedury «swap2»? Pomóż mu zoptymalizować «swap» posługując się słowem kluczowym «restrict» i wyjaśnij jego znaczenie.

```
1 void swap(long *xp, long *yp) {
2   *xp = *xp + *yp; /* x+y */
3   *yp = *xp - *yp; /* x+y-y = x */
4   *xp = *xp - *yp; /* x+y-x = y */
5 }
```

```
1 void swap2(long *xp, long *yp) {
2   long x = *xp, y = *yp;
3   x = x + y, y = x - y, x = x - y;
4   *xp = x, *yp = y;
5 }
```

Zadanie 2. Ile razy zostanie zawołana funkcja «my_strlen» w funkcji «my_index» i dlaczego? Usuń atrybut[2] «noinline» funkcji «my_strlen». Czy zezwolenie kompilatorowi na przeprowadzenie **inliningu** pomogło? Dodaj atrybut «pure» do funkcji «my_strlen». Czemu tym razem kompilator był w stanie lepiej zoptymalizować funkcję «my_index»? Czym charakteryzują się **czyste funkcje**?

```
1 __attribute__((noinline))
2 size_t my_strlen(const char *s) {
3   size_t i = 0;
4   while (*s++)
5     i++;
6   return i;
7 }
```

```
8 const char *my_index(const char *s, char v) {
9   for (size_t i = 0; i < my_strlen(s); i++)
10    if (s[i] == v)
11      return &s[i];
12  return 0;
13 }
```

Zadanie 3. Na podstawie kodu wynikowego z kompilatora odtwórz zoptymalizowaną wersję funkcji «foobar» w języku C. Wskaż w poniższym kodzie **niezmienniki pętli** (ang. *loop invariants*), **zmienne indukcyjne** (ang. *induction variable*). Które wyrażenia zostały wyniesione przed pętlę i dlaczego? Które wyrażenia uległy **osłabieniu** (ang. *strength reduction*)?

```
1 void foobar(long a[], size_t n, long y, long z) {
2   for (int i = 0; i < n; i++) {
3     long x = y - z;
4     long j = 7 * i;
5     a[i] = j + x * x;
6   }
7 }
```

Zadanie 4. Na podstawie kodu wynikowego z kompilatora odtwórz zoptymalizowaną wersję funkcji «neigh» w języku C. Kompilator zastosował optymalizację **eliminacji wspólnych podwyrażeń** (ang. *common subexpression elimination*). Wskaż w poniższym kodzie, które podwyrażenia policzył tylko raz. Pokaż, że jesteś w stanie zoptymalizować funkcję lepiej niż kompilator – przepisz jej kod tak, by generował mniej instrukcji.

```
1 long neigh(long a[], long n, long i, long j) {
2   long ul = a[(i-1)*n + (j-1)];
3   long ur = a[(i-1)*n + (j+1)];
4   long dl = a[(i+1)*n - (j-1)];
5   long dr = a[(i+1)*n - (j+1)];
6   return ul + ur + dl + dr;
7 }
```

[2] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html

Zadanie 5. Na podstawie [1, §5.14.1] odpowiedz na następujące pytania. Do czego służą **programy profilujące**? Jakie informacje niesie ze sobą **profil płaski** i **profil grafu wywołań**? Czemu profilowanie programu wymaga zbudowania go ze specjalną opcją kompilatora `-pg`? Na czym polega **zliczanie interwałowe**? Jak przy pomocy programu profilującego zidentyfikowano w [1, §5.14.2] procedury, które należało zoptymalizować? Które optymalizacje przyniosły największy efekt?

Zadanie 6. Skompiluj poniższą funkcję i na podstawie wynikowego kodu maszynowego skonstruuj **graf przepływu danych** jak w [1, §5.7.3]. Wskaż w nim **ścieżkę krytyczną** na podstawie **czasu opóźnienia** przetwarzania instrukcji z tabeli [1, 5.12]. Która z użytych instrukcji zostanie rozłożona na **mikro-operacje**?

```
1  void nonsense(long a[], long k,
2                long *dp, long *jp) {
3    long e = a[2];
4    long g = a[3];
5    long m = a[4];
6    long h = k - 1;
7    long f = g * h;
8    long b = a[f];
9    long c = e * 8;
10   *dp = m + c;
11   *jp = b + 4;
12 }
```

Zadanie 7. Załóżmy, że kod z poprzedniego zadania wykonuje się na procesorze **out-of-order** `Core i7` z **mikroarchitekturą** `Haswell` opisaną w [1, §5.7.1]. Procesor potrafi w jednym cyklu zegarowym zdekodować i zlecić do wykonania maksymalnie 4 instrukcje i ma osiem **jednostek funkcyjnych** (ang. *functional units*). Ile cykli zegarowych zajmie mu wykonanie instrukcji z poprzedniego zadania?

Zadanie 8. Posługując się programem `llvm-mca` (ang. *machine code analyzer*) wbudowanym w Compiler Explorer przedstaw symulację pojedynczego wywołania funkcji «`nonsense`». Do parametrów programu dodaj: «`-mcpu=haswell -timeline -iterations=1`». Na diagramie Timeline view[3] wskaż punkty **wysłania** (ang. *dispatch*), **wykonania** i **zatwierdzenia** (ang. *retire*) instrukcji. Ile czasu zajmuje wykonanie funkcji według symulatora? Wyjaśnij dokładnie z czego wynikają przestoje w przetwarzaniu instrukcji? Na którym z rejestrów procesor użył techniki **przezywania rejestrów** (ang. *register renaming*)?

Literatura

[1] „Computer Systems: A Programmer's Perspective"
Randal E. Bryant, David R. O'Hallaron; Pearson; 3rd edition, 2016

[2] „Modern Processor Design: Fundamentals of Superscalar Processors"
John Paul Shen, Mikko H. Lipasti; McGraw-Hill; 1st edition, 2005

[3] `https://llvm.org/docs/CommandGuide/llvm-mca.html#timeline-view`