Architektury systemów komputerowych

Lista zadań nr 12

Na zajęcia 27 i 28 maja 2024

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.

Kod skompilowany z opcją «`-Os -mno-sse`» należy prezentować przy pomocy Compiler Explorer[1] używając najnowszej dostępnej wersji kompilatora `GCC`.

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? Dodaj atrybut[2] «`pure`» do funkcji «`my_strlen`». Czemu tym razem kompilator był w stanie lepiej zoptymalizować funkcję «`my_index`»? Czym charakteryzują się **czyste funkcje**? Następnie uzupełnij ciało funkcji «`my_strlen`» tak, by wykonywała to samo co «`strlen`». Następnie usuń atrybut «`pure`» i dodając słowo kluczowe «`static`» zawęź zakres widoczności funkcji do bieżącej jednostki translacji. Co się stało w wyniku przeprowadzenia **inliningu**? Czy kompilatorowi udało się samemu wywnioskować, że funkcja jest czysta?

```
1 __attribute__((leaf))
2   size_t my_strlen(const char *s);
3
4 const char *my_index(const char *s, char v) {
5   for (size_t i = 0; i < my_strlen(s); i++)
6     if (s[i] == v)
7       return &s[i];
8   return 0;
9 }
```

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 }
```

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

Zadanie 4. Na wykładzie została zaprezentowana funkcja «compute» i na jej przykładzie wyjaśniono optymalizacje transformacji pętli, tj. **zamiany pętli** (ang. *loop interchange*), **łączenia pętli** (ang. *loop fusion*) oraz **usuwania zmiennych indukcyjnych** (ang. *induction variable elimination*). Czy kompilator był w stanie przeprowadzić te optymalizacje na poniższej funkcji? Jeśli nie, to zoptymalizuj ją krok po kroku. Przeanalizuj dokładnie działanie funkcji, czy jesteś w stanie poczynić jakieś obserwacje, które pozwalają Ci wykonać dodatkowe optymalizacje, np. *ponowne wyliczanie wartości* (ang. *rematerialization*[3])?

```
1  void compute2(long *a, long *b, long k) {
2    long n = 1 << k;
3    for (long i = 0; i < n; i++)
4      a[i * n] = a[i] = 0;
5    for (long i = 1; i < n; i++)
6      for (long j = 1; j < n; j++)
7        a[j * n + i] = i * j;
8    for (long i = 1; i < n; i++)
9      for (long j = 1; j < n; j++)
10       b[i * n + j] = a[i * n + j]
11                    + a[(i - 1) * n + (j - 1)];
12 }
```

Zadanie 5. 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 }
```

Zadanie 6. Zapoznaj się z [1, §5.11.2] i na podstawie [1, §5.7.1] wyjaśnij jak procesor przetwarza skoki warunkowe. W jakim celu procesor używa **predyktora skoków**? Co musi zrobić, jeśli skok zostanie źle przewidziany? Które skoki warunkowe warto zoptymalizować do instrukcji «cmov»? Posługując się narzędziem *Compiler Explorer* z opcją «-O1» przepisz poniższą funkcję tak, żeby w wierszu 4 kompilator nie wygenerował skoku warunkowego.

```
1 void merge1(long src1[], long src2[], long dest[], long n) {
2   long i1 = 0, i2 = 0;
3   while (i1 < n && i2 < n)
4     *dest++ = src1[i1] < src2[i2] ? src1[i1++] : src2[i2++];
5 }
```

Wskazówka: W kodzie po optymalizacji mogą wystąpić tylko dwa skoki warunkowe.

Zadanie 7. 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].

```
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 * 11;
7     long f = g * h;
8     long b = a[f];
9     long c = e * 8;
10    *dp = m + c;
11    *jp = b + 4;
12  }
```

[3] https://en.wikipedia.org/wiki/Rematerialization

Zadanie 8. 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*). Jak technika **przezywania rejestrów** (ang. *register renaming*) pomaga procesorowi w usuwaniu fałszywych zależności między instrukcjami? Ile cykli zegarowych zajmie mu wykonanie instrukcji z poprzedniego zadania?

Zadanie 9. Posługując się programem `llvm-mca` wbudowanym w *Compiler Explorer* przedstaw symulację pojedynczego wywołania funkcji «`nonsense`». Do parametrów programu dodaj: «`-mcpu=haswell -timeline -iterations=1`». Na diagramie `Timeline view`[4] wskaż punkty **wysłania** (ang. *dispatch*), **wykonania** (ang. *execute*) i **zatwierdzenia** (ang. *retire*) instrukcji. Ile czasu zajmuje wykonanie funkcji według symulatora? Wyjaśnij z czego wynikają **przestoje** (ang. *stall*) w przetwarzaniu instrukcji?

Zadanie 10 (bonus). Na podstawie uproszczonego (sic!) diagramu[5] mikroarchitektury `Haswell` postaraj się wyjaśnić jaka droga czeka instrukcję od jej pobrania z pamięci do ukończenia jej wykonywania. Powróćmy do wydruku z narzędzia `llvm-mca` z poprzedniego zadania. Przyjrzyjmy się wydrukom z sekcji *Instruction Info* i *Resource pressure by instruction*. Która z wygenerowanych przez kompilator instrukcji zostanie rozłożona przez **dekoder instrukcji** na **mikro-operacje**? Które jednostki funkcyjne będą używane do wykonania poszczególnych instrukcji z naszej funkcji? Dlaczego kompilatory lubią umieszczać wejścia do pętli pod adresami podzielnymi przez 16?

Uwaga! Nie wszystkie instrukcje x86-64 można wykonać wyłącznie z użyciem jednej jednostki funkcyjnej.

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

[4] `https://llvm.org/docs/CommandGuide/llvm-mca.html#timeline-view`
[5] `https://en.wikichip.org/wiki/File:haswell_block_diagram.svg`