Systemy komputerowe

Lista zadań nr 7

Karolina Szlęk 300411

Przydatne:

* $imm ---> imm

* Ra ---> Reg[Ra]

* imm ---> Mem[imm]

* D(Rb, Ri, s) --->Mem[D + Reg[Rb] + (Reg[Ri] * s)]

* (Rb, Ri) ---> Mem[Reg[Rb] + Reg[Ri]]

* D(Rb, Ri) ---> Mem[Reg[Rb] + Reg[Ri] + D]

* (Rb, Ri, s) ---> Mem[Reg[Rb] + (s * Reg[Ri])]

*(R) ---> Mem[Reg[R]]

*D(R) ---> Mem[Reg[R] + D]

Zadanie 1

Zadanie 2

Zadanie 2. Rozważmy program składający się z dwóch plików źródłowych:

```
1   /* str-a.c */
2   #include <stdio.h>
3
4   char *somestr(void);
5
6   int main(void) {
7     char *s = somestr();
8     s[5] = '\0';
9     puts(s);
10    return 0;
11  }
```

```
1   /* str-b.c */
2   char *somestr(void) {
3     return "Hello, world!";
4   }
```

Próbujemy modyfikować zwrócony tekst. Nie możemy tego jednak zrobić ponieważ trafił on do sekcji .roddata (bo był stała). Wiemy zatem, że “Hello world!” jest w nieedytowalnej części pamięci (w sekcji .roddata). Żeby pozbyć się tego problemu musimy sprawić by znalazł się w sekcji .data (ta sekcja jest modyfikowalna). Wystarczy użyć alokacji pamięci i tam zapisać ten napis.

```
char *somestr(void) {
        char *str = malloc(sizeof(char) * 20);
        strcpy(str, "Hello, world!");
        return str;
}
```

Zadanie 3

Zadanie 4

Zadanie 4. Jakie konstrukcje językowe w C są blokadami optymalizacji (ang. *optimization blockers*)? Porównaj funkcje `combine1` i `combine4` ze slajdów i wyjaśnij, dlaczego wydajniejsza wersja musiała zostać stworzona ręcznie.

W języku C wyróżniamy takie blokady optymalizacji:

- Procedure calls (wywołanie procedur) kompilator nie może sam zastąpić wiele wywołań procedur jednym wywołaniem, bo nie wie jaki jest efekt uboczny wywołania procedur.
- Memory aliasing (Aliasowanie pamięci). W każdym obrocie pętli uzyskujemy dostęp do b[i] i zapisujemy tam jakąś wartość pośrednią. Jeśli zapisywalibyśmy wynik pośredni w pewnej zmiennej i po wykonaniu obliczeń zapisywali w tablicy, kosztowałoby nas to mniej czasu.

Blokada optymalizacji - aspekt programu, który może poważnie ograniczyć możliwości generowania zoptymalizowanego kodu przez kompilator. Kompilator może podjąć się optymalizacji tylko jeśli ma pewność, że nie będzie miała żadnych skutków na wyniku programu.

Wydajniejsza wersja musiała zostać stworzona ręcznie, ponieważ pojawiły się Blokady optymalizacji.
Optymalizacje jakie zostały wprowadzone ręcznie w funkcji combine4 :

- długość vectora v liczymy tylko raz a nie przy każdym obrocie pętli. (Rozmiar wektora nie zmieni się w procesie wywołania tej funkcji więc wystarczy raz obliczyć ją raz kompilator tego nie zrobi sam, bo jest to Blokada optymalizacji --> Procedure calls)
- rezygnujemy z funkcji get_vec_element która sprawdza czy index i jest poprawny. Nie musimy tego robić, bo mamy to zagwarantowanie przez warunek w pętli.
- zamiast cały czas używać wskaźnika dest i robi jego deferencje w dwóch miejscach i dopiero wykonywać OP na nim tworzymy zmienna lokalną trzymana w rejestrze i do tej zmiennej zapisujemy wynik i po obliczeniu ostatecznego wyniku robimy jedną dereferencje wskaźnika i zapisujemy tam wynik.

Kompilator tego nie robi sam z powodu problemów z aliasingiem. Kompilator nie potrafi wykonać sam takich optymalizacji, które mogłyby zmienić semantykę programu (zmieni potencjalnie to jak program zadziała). On podejmuje się optymalizacji tylko wtedy, kiedy ma pewność, że ta optymalizacja nie zmieni działania programu (nie powoduje negatywnych skutków). Dlatego warto pozbywa się blokerów optymalizacji, ponieważ możemy sami zoptymalizować to czego kompilator się nie podejmie.

Zadanie 5

Zadanie 5. Na podstawie rozdziału §5.7 podręcznika „Computer Systems: A Programmer's Perspective" i rysunku 5.11 wyjaśnij zasadę działania procesora o architekturze **superskalarnej**. Jak działa **spekulatywne** wykonywanie instrukcji? Co to znaczy, że poszczególne jednostki funkcjonalne procesora działają w sposób **potokowy**?

Zwykły procesor ma ustalona listę instrukcji, która ma po kolei wykonywać po jednej w cyklu zegara. Natomiast Procesor o architekturze superskalarnej może wykonywać wiele instrukcji w jednym cyklu zegara. Nie może tego jednak robić w sposób bezmyślny np. wykonanie na raz operacji dostępu do komórki pamięci a następnie operacja zwiększenie wartości w tej komórce o jeden może powodować niechciane zachowanie. Dlatego taki procesor posiada dwie główne części:

- instruction control unit (ICU)
- execution unit (EU)

ICU odpowiada za odczytanie sekwencji instrukcji z pamięci i planuje z nich zestawu instrukcji do wykonania, który stopniowo przekazuje do EU. Nie zawsze to planowanie jest możliwe np. w przypadku instrukcji warunkowej. W takim przypadku potrzebujemy znać wartość logiczna warunku, która będzie obliczona później przez EU. Działanie spekulatywne poleca na tym że icu wybierze jedna z gałęzi i zacznie planować kolejne instrukcji, jednakże może się okazać, że icu wybrał złą gałąź. W takim przypadku ICU na nowo musi zaplanować zestaw, ale to spowolni działanie procesora, ponieważ EU musi czekać na kolejne instrukcje.

Jeśli jednostka funkcjonalna procesora działa w sposób potokowy to oznacza, że jeśli instrukcje składają się z kilku etapów a wyniki tych instrukcji nie zależą od siebie to kolejna instrukcja nie musi czekać, aż cala poprzednia instrukcja się wykona. Oznacza to, że w momencie, gdy np. pierwsza instrukcja jest w “Fazie II” (jest wykonywany 2 etap) to jednocześnie druga instrukcja może zacząć się obliczać, czyli przejść do “Fazy I”.

Zadanie 6

Zadanie 6. Rozpatrujemy procesor o charakterystyce podanej w tabeli 5.12 podręcznika „Computer Systems: A Programmer's Perspective”. Zdefiniuj miarę wydajności CPE (ang. *cycles per element/operation*), następnie wyjaśnij pojęcie granicy opóźnienia (ang. *latency bound*) oraz granicy przepustowości (ang. *throughput bound*) procesora. Skąd biorą się wartości w tabelce na stronie 560?

	Integer			Floating point		
Operation	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3–30	3–30	1	3–15	3–15	1

Figure 5.12 **Latency and issue time characteristics of Intel Core i7 arithmetic operations.** Latency indicates the total number of clock cycles required to perform the actual operations, while issue time indicates the minimum number of cycles between two operations. The times for division depend on the data values.

Do rozwiązania zadania potrzebne nam będzie zdefiniowanie miary wydajności CPE (ang. cycles per element or operation).

CPE to miara wydajności dla programów operujących na wektorach albo listach. Informuje, jak dużo cykli było potrzebne na wykonanie programu operującego na liście o danej długości.

Cycles Per Element (CPE)

- **Convenient way to express performance of program that operators on vectors or lists**
- **Length = n**
- **In our case: CPE = cycles per OP (gives hard lower bound)**
- **T = CPE*n + Overhead**

Teraz wyjaśnijmy jeszcze pojęcie granicy opóźnienia (ang. latency bound) oraz granicy przepustowości (ang. throughput bound) procesora.

Granica opóźnienia (ang. latency bound), czyli minimalna wartość cykli procesora dla funkcji, która wykonuje swoje operacje w ściśle określonej kolejności. (Wynik jednej jest potrzebny do wykonania drugiej).

Granica przepustowości (ang. throughput bound) - to surowa moc obliczeniowa jednostek funkcjonalnych procesora, a zarazem ostateczna granica wydajności dla programu. Daje nam minimalne ograniczenie dla CPE w oparciu o maksymalne tempo, z jakim jednostki funkcjonalne mogą wytwarzać wyniki.

	Integer		Floating point	
Bound	+	*	+	*
Latency	1.00	3.00	3.00	5.00
Throughput	0.50	1.00	1.00	0.50

Zadanie 7

Zadanie 7. Z czego wynika wysoka wydajność funkcji `combine6` (tabelka 5.21, str. 573)? Jak zoptymalizować ten kod by dojść do granicy przepustowości procesora?

```
/ * 2 x 2 loop unrolling */

void combine6(vec_ptr v, data_t *dest)

{

        long i;
```

```
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;
/* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i+1];
    }
/* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

Pomysły na zoptymalizowanie:

- Można wykorzystać dwie niezależne zmienne lokalne. Pozwala nam to zredukować liczbę odczytów i zapisów do pamieci względem combine1.
- Można również wyodrębnić warunek pętli for (wywołanie vec_length(v) - poza nią pozwala wykonać tą procedurę tylko raz).
- Wyodrębnienie z ciała pętli wywołania get_vec_element([...]) oraz zmiana na get_vec_start(v) pozwalą na wykonanie tej procedury tylko raz otrzymując wskaźnik na początek wektora w pamięci.
- Dobrym pomysłem może być też wykorzystanie w pętli dwóch niezależnych od siebie zmiennych lokalnych, co pozwoli obliczać je równolegle maksymalizując zyski z wykorzystania potokowania jednostek funkcjonalnych procesora.

Dodatkowe obserwacje:

Rejestry są akumulatorami dla danej grupy elementów.
Operacje na danym akumulatorze są niezależne od pozostałych. Co za tym idzie instrukcje w danej iteracji są wykonywane w sposób potokowy (są pipelined) i wykonują się równolegle.

Korzystając z pomysłów na optymalizacja jak i obserwacji możemy wymyślić, jak dojść do granicy przepustowości.

Wiemy, że niezależne operacje mogą być wykonywane potokowo i równolegle. Skorzystajmy z tego. Naszym rozwiązaniem będzie zwiększenie liczby operacji na jedną iterację. Musimy jednak pamiętać, że mamy ograniczoną liczbę akumulatorów (rejestrów). Z książki wiemy, że unrolling rzędu 10x10 daje nam wydajność bardzo bliską tej granicy.
Dla operacji mającej latency L i capacity C, chcemy unrolling rzędu k >= L*C

Zadanie 8

Zadanie 8. Rozważmy dysk o następujących parametrach: jeden talerz; jedna głowica; 400 tysięcy ścieżek na powierzchnię; 2500 sektorów na ścieżkę; 7200 obrotów na minutę; czas wyszukiwania: 1ms na przeskoczenie o 50 tysięcy ścieżek.

1. Jaki jest średni czas wyszukiwania?
2. Jaki jest średni czas opóźnienia obrotowego?
3. Jaki jest czas transferu sektora?
4. Jaki jest całkowity średni czas obsługi żądania?

1. Jaki jest średni czas wyszukiwania?

Średni czas przesunięcia raczki na właściwą ścieżkę.

Oczekiwana odległość dwóch losowo wybranych punktów na odcinku jednostkowym to 1/3. Ogólny fakt z Rachunku prawdopodobieństwa.

40000 / 3 --> tyle ścieżek musimy przejść podzielić przez 5000, bo tyle przechodzimy na 1ms

Zatem średni czas wyszukiwania to w przybliżeniu

(40000/50000) * (1/3) ≈ 2, 67ms

2. Jaki jest średni czas opóźnienia obrotowego?

Korzystamy z odpowiedniego wzoru na opóźnienie obrotowe, do którego podstawiamy wartości.

(½)*(1m/7200) * (60s/1m) * (100ms/1s) ≈ 4, 16ms

3. Jaki jest czas transferu sektora?

Tu korzystamy ze wzoru na czas transferu sektora i podstawiamy odpowiednie wartości.

(1/2500)*(1m/7200)*(60s/1m)*(1000ms/1s) ≈ 0, 0033ms

4. Jaki jest całkowity średni czas obsługi żądania?

2, 67ms + 4, 16ms + 0, 0033ms ≈ 6, 8333ms

Zadanie 9

Zadanie 9. Rozważmy dysk o następujących parametrach: 360 obrotów na minutę, 512 bajtów na sektor, 96 sektorów na ścieżkę, 110 ścieżek na powierzchnię. Procesor czyta z dysku całe sektory. Dysk sygnalizuje dostępność danych zgłaszając przerwanie na każdy przeczytany bajt. Jaki procent czasu procesora będzie zużywała obsługa wejścia-wyjścia, jeśli wykonanie procedury przerwania zajmuje 2.5μs? Należy zignorować czas wyszukiwania ścieżki i sektora.

Do systemu dodajemy kontroler DMA. Przerwanie będzie generowane tylko raz po wczytaniu sektora do pamięci. Jak zmieniła się zajętość procesora?

Wskazówka W tym zadaniu procedura obsługi przerwania zajmuje się przetworzeniem przeczytanych danych i będzie wywołana za każdym razem, gdy zgłoszono przerwanie.

Czas na przeczytanie sektora:

(1/96) * (1m/360) * (60s/1m) * (1000ms/1s) ≈ 1, 736ms

Czas przerwań:

512 bajt/sektor * 2.5µs = 1.28 ms

Jaki procent czasu będzie zajmowała obsługa wejścia/wyjścia?

czas przerwań / czas transferu * 100 = (1.28ms / 1.736ms) * 100 =73.7%

Do systemu dodajemy kontroler DMA. Przerwanie będzie generowane tylko raz po wczytaniu sektora do pamięci. Jak zmieniła się zajętość procesora?

2,5µs / 1.736 ms * 100 = (0.0025 ms / 1.736ms) * 100 = 0.144%

Zadanie 10

Zadanie 10. W przeważającej większości systemów implementujących moduły DMA, procesor ma niższy priorytet dostępu do pamięci głównej niż moduły DMA. Dlaczego?

Wskazówka: Co się może stać, jeśli urządzenia nie mają gwarancji wykonywania transferów w regularnych odstępach czasu?

DMA - direct memory access; Bezpośredni dostęp do pamięci.

Procesor ma niższy priorytet dostępu do pamięci głównej, ponieważ on w przeciwieństwie do modułów DMA posiada pamięć podręczną.

Przykładem, który to zobrazuje jest karta sieciowa, która jest modułem DMA.
Jeżeli procesor zajmie dostęp do pamięci głównej a karta sieciowa dostanie kolejny pakiet danych, to ten który ma obecnie będzie musiała zapomnieć/ zniszczyć, aby przyjąć kolejny, bo nie posiada pamięci podręcznej w której mogłaby zapisać poprzedni pakiet.

Wyższy priorytet modułów DMA ma zapobiec potencjalnym błędom, czyli wystąpieniu overrun i underrun.

overrun - jeżeli urządzenie bez bufora chce zapisać dane w pamięci. Nie zdoła zrobić tego przed otrzymaniem nowych danych, oryginalne dane zostaną nadpisane i nie trafią do pamięci.

underrun –gdy urządzenie zażąda danych z pamięci, a te nie będą dostarczone w czasie. Może to spowodować podanie niezamierzonych danych jako output .