DLACZEGO TAK WOLNO?

sobota, 22 czerwca 2024 22:06

W sensie dlaczego kod jest tak wolny?
To spróbujemy dzisiaj określić.

Bardzo rzadko zaczyna się pisanie kodu od myślenia o tym jaka powinna być wersja najszybsza. Czasami jest tak, że jeśli wiemy gdzie chcemy trafić na wykresie to na początku projektuje się ogólne założenia tak, żeby mieć szansę uzyskać określoną wydajność, natomiast najpierw należy napisać program działający. Potem sprawdzić czy ten algorytm jest wolny z tego powodu, z którego przypuszczamy, że jest. Przeważnie nie, przeważnie powód jest zupełnie inny.

„Przedwczesna optymalizacja jest źródłem ***wszelkiego zła.”***

Premature optimization is the root of all evil.

Donald Ervin Knuth, Structured Programming with go to Statements, ACM Computing Surveys, Vol 6, No. 4, Dec. 1974

Charles Antony Richard Hoare ?

Edsger Wybe Dijkstra ?

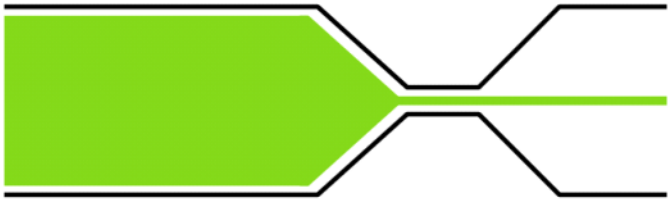
Jeśli weźmiemy odpowiednio dużą pulę kodów to statystycznie będzie tak, że większość czasu ten kod spędza w swojej niewielkiej części. Podczas profilowania czyli sprytnego mierzenia zachowania kodu w czasie, jego wydajności, celem jest znalezienie tych miejsc w kodzie, które zajmują najwięcej czasu oraz znalezienie powodów tego dlaczego to tyle zajmuje.

Zasada Pareta (80/20)

Statystycznie **(!)**,
80% zasobów jest związanych z 20% przypadków.

20% linii kodu zajmuje 80% czasu wykonania.

Cel profilowania – identyfikacja „gorących obszarów” w kodzie (*ang. hot spots*) i „wąskich gardeł” (*ang. bottleneck*).



41

Jeśli już mamy napisany kod i on działa, należy zmierzyć jego wydajność pod różnymi względami. A potem sprawdzić czy nasze poprawki rzeczywiście polepszyły parametry tego kodu, bo zazwyczaj się okazuje, że jednak pogorszyły.
Jak to określić?
Np. przez stosunek czasu.

Przyspieszenie

$$S = \frac{T_{ulepszone}}{T_{referencyjne}}$$

$$T = T(p) + T(s) = T(p) + T(1-p)$$

p - część programu, która **jest** przyspieszana

$s = 1-p$ - część programu, która **nie jest** przyspieszana

42

Uwaga! Tu jest pułapka! Jeżeli jako parametry te traktowalibyśmy czas wykonania to przyspieszenie wyjdzie nam jako wartość ułamkowa więc tu trzeba właściwie dobierać parametry. Jeżeli porównujemy czas wykonania programu to trzeba ten czas wolniejszy, przed przyspieszeniem podzielić przez czas szybszy czyli po przyspieszeniu.

T - czas wykonania programu, składa się z dwóch części:

a) p - części przyspieszanej
b) s - części nieprzyspieszanej, możemy też zapisać jako 1 - p

BARDZO WAŻNE, Z TEGO WYNIKAJĄ DWA FUNTAMENTALNE PRAWA!

Prawo Am(i)dhala

sobota, 22 czerwca 2024 22:36

PRAWO AMDAHLA:
"Jeżeli będziemy przyspieszali pewną część programu to maksymalne przyspieszenie jakie uzyskamy wynika nie z części przyspieszanej, ale nieprzyspieszanej. Nawet jeśli część przyspieszaną wyrzucimy kompletnie to pozostały czas będzie częścią nieprzyspieszaną "

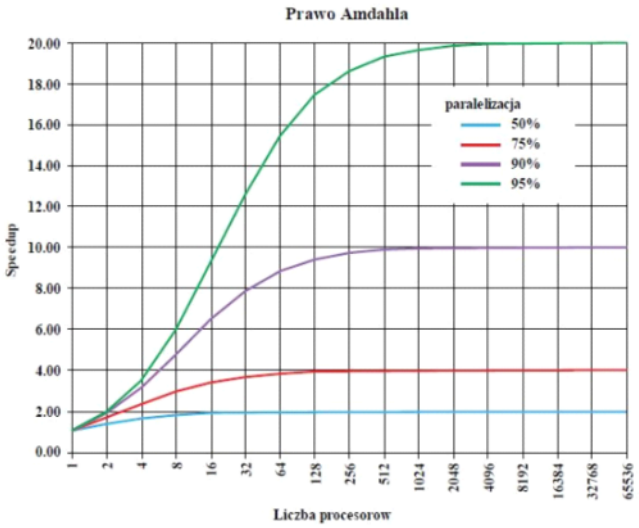
Dodatkowo:
"Jeśli kod przyspieszany wpływa na kod nieprzyspieszany, kod nieprzyspieszany staje się częścią przyspieszaną".

Ten wzór jest ważny:

Przyśpieszenie przetwarzania (ang. speed-up) jest ograniczone przez część algorytmu, której nie można wykonać równolegle z innymi jego częściami

$$S_{latency} = \frac{T(1-p)+Tp}{T(1-p)+Tpn^{-1}} = \frac{1}{1-p+\frac{p}{n}}$$

$1-p$ – udział części sekwencyjnej, n – liczba procesorów/zasobów zrównoleglonych



43

S - przyspieszenie całkowite kodu
P - część przyspieszanego kodu, np. jeśli 30% to p= 0,3
N - krotność przyspieszenia kodu przyspieszanego

Jeśli mamy kod, w którym przyspieszamy 95% całego programu to maksymalne przyspieszenie (po wyrzuceniu całkowicie części przyspieszanej) wynosi 20 razy (bo zostaje nam 5% z 100% czyli 1/20). Szybciej się nie da. To jest właśnie prawo Amdahla.

W ogólnym przypadku prawo Amdahla jest stosowane do określania przyspieszania programów równoległych:
Jest część, którą przyspieszamy na wiele procesorów.
Jest część, której nie damy rady przyspieszyć, bo ona z natury swojej jest szeregowa.
Bardzo rzadko zdarzają się algorytmy, w których możemy przyspieszyć całość. Ale wtedy skalowało by się bardzo ładnie.
Te algorytmy nazywamy "algorytmami trywialnie zrównoleglalnymi".

Zawsze będzie jakaś część szeregowa, żeby załadować z dysku obraz programu do pamięci i go uruchomić.
ZAWSZE BĘDZIE COŚ SZEREGOWEGO
ZAWSZE
ZAWSZE BĘDZIE COŚ SZEREGOWEGO!!!!

Choćbyśmy mieli dziesiątki tysięcy procesorów niż tam kilka razy czy kilkanaście razy (dla określonej klasy problemów).
To samo prawo można stosować do kodów przyspieszanych w dowolny sposób. Jeśli optymalizujemy fragment naszego kodu to maksymalna wydajność jaką uzyskamy wynika z fragmentu kodu, którego nie przyspieszamy.

Zastosowanie: nie tylko przetwarzanie równoległe

Przykład: **10-krotne przyspieszenie** (**bardzo trudne!**) fragmentu kodu zajmującego 5% czasu wykonania spowoduje przyspieszenie

$$S = \frac{1}{1 - 0.05 + \frac{0.05}{10}} \approx 1.047$$

czyli skrócenie czasu wykonania do 1 / 1.047 = 0.955 początkowej wartości (o **4.5%**). 44

Wniosek: Optymalizować należy to co da nam zyski z optymalizacji.

Z prawa Amdahla można określić jakąś część kodu trzeba przyspieszyć, żeby uzyskać jakieś przyspieszenie.

Jeśli jesteśmy w 5-10 minut w stanie przyspieszyć program o 1,5% warto to zrobić. Wszystko kwestia tego jak skuteczne będą nasze działania w danym czasie.

Jak przyspieszymy potem dłuższą część to ta krótka stanie się dłuższa i zyskamy jeszcze więcej.

Prawo Gus()tafsona

sobota, 22 czerwca 2024 23:37

Mamy już kod przyspieszony.
Rozpatrywana od razu wersja zrównoleglona.
Bierzemy algorytm, który pracuje na 1000 procesorów. On wykonuje się na nim jakiś czas. To czas odniesienia.
Porównujemy czas odniesienia na tym samym kodzie z tymi samymi danymi na jednym procesorze.

PRAWO GUSTAFSONA:
"Jeżeli będziemy brali coraz większy problem mamy szansę na liniowe przyspieszenie, ale tylko jeśli te problemy będą coraz większe.

Lub

"Jeśli będziemy brali coraz mocniejsze maszyny będziemy w stanie rozwiązywać coraz to większe problemy"
Uzupełnia się to z prawem Amdhala.

Na 1000 procesorów bierzemy problem tysiąc razy większy niż na jednym procesorze.
Na 10000 procesorów bierzemy problem dziesięć tysięcy razy większy niż na jednym procesorze.
I wtedy ta wydajność nam się skaluje liniowo z liczbą procesorów natomiast tylko jeśli rozmiar problemu rośnie
"
Rozpatrujemy więc przypadki gdy mamy coraz większe problemy na coraz większych maszynach.

BARDZO WAŻNE (WZORY)!!!!

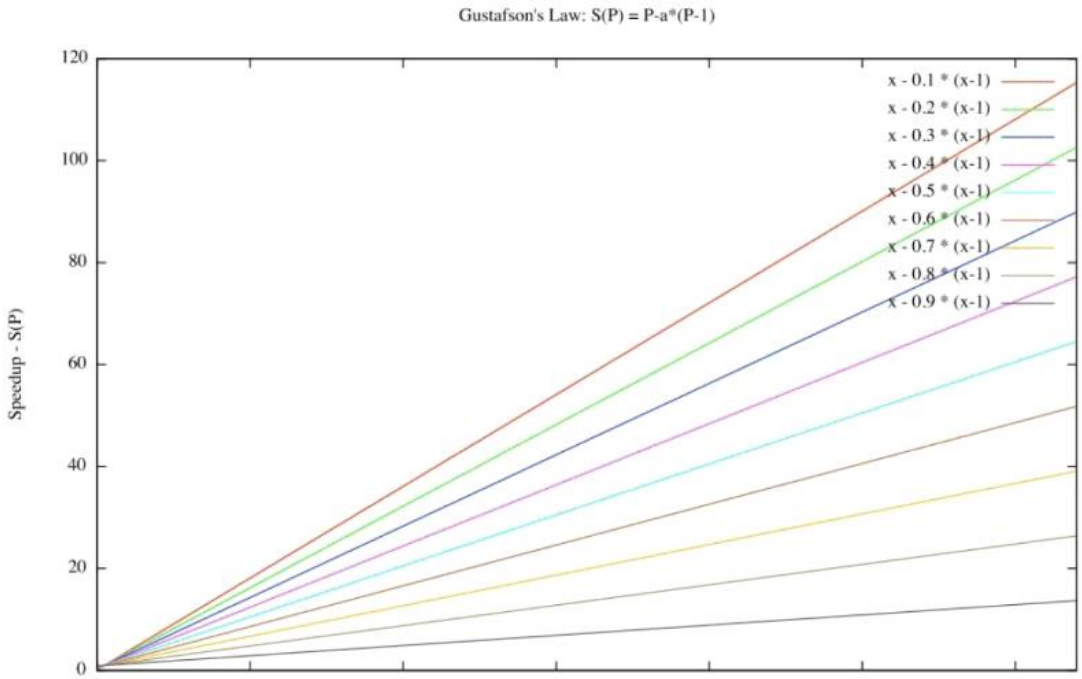
Czas wykonania procesu na jednym rdzeniu, na jednym procesorze to jest część nieprzyspieszona + n razy wykonana część zrównoleglona. W punkcie odniesienia założyliśmy, że część p była wykonana na n procesorach i trwała ileś czasu. Jeżeli wykonujemy na jednym procesorze trzeba ją wykonać n razy.

Obciążenie robocze (ang. *workload*) po ulepszeniu w skali n mamy $W(n)=(1-p)W+npW$

Przyśpieszenie wykonania (*skrócenie zwłoki w wykonaniu*) zadania w ustalonym czasie wykonania (*at fixed execution time*), spodziewane po ulepszeniu zasobów w skali n.

$$S(n) = TW(n)/TW = W(n)/W = 1 - p + pn$$

- n – przyśpieszenie (*speed-up in latency*) wykonania części zadania z użyciem ulepszonych zasobów;
- p – udział części ulepszanej w nakładzie czasu pracy całego zadania (przed ulepszeniem);



Jeżeli będziemy brać coraz większe n, a p czyli rozmiar problemu na dany procesor będzie stały to przyspieszenie będzie rosło liniowo. W stosunku do uruchomienia na jednym rdzeniu czyli słabej maszynie.

Przykład: jeśli jakiś fragment kodu zajmuje 90% czasu wykonania i jego przyspieszenie jest trudne, to **być może dla większego** (innego) zbioru danych wejściowych **inny fragment kodu** stanie się gorącym punktem.

Jeśli optymalizujemy kod to musi być on optymalizowany w tych warunkach, w których będzie uruchamiany. Czyli na tych rozmiarach danych, na których będzie rzeczywiście pracował. Częstym błędem jest przyspieszanie kodu dla mały danych podczas gdy dla dużych problem jest w zupełnie innym miejscu, którym się odpowiednio nie zajęliśmy.

Od czego zależy maksymalny zbiór danych, który możemy rozwiązać na maszynie?

a) Od ilości dostępnej pamięci (fizycznie)
b) Od czasu przetwarzania danych "Nikt nie będzie roku czekał na zakończenie jakiejś symulacji"
c) Sam problem, który nie może przyjąć za dużo danych

Jeśli przetwarzamy mały zbiór danych na każdego człowieka na Ziemi, jeżeli na każdego potrzeba kilobajt, nie będziemy przetwarzali petabajtów danych, bo nie ma tylu ludzi.

PORÓWNANIE

niedziela, 23 czerwca 2024 00:05

Porównanie

Prawo Amdahla: Samochód pokonujący dystans 60 km pokonał w ciągu 1h połowę dystansu jadąc z prędkością 30 km/h. Jest niemożliwe, by uzyskał średnią 90 km/h na całej trasie.

Prawo Gustafsona: Samochód jakiś czas jechał z prędkością < 90 km/h. Jeśli wystarczy czasu i drogi może uzyskać bliską jej średnią niezależnie od czasu i odległości już przebytej.

Prawo Amdahla – perspektywa przyśpieszenia jednego zadania

- wymagania obliczeniowe pozostają stałe mimo wzrostu mocy obliczeniowej (analiza tych samych danych zajmie mniej czasu przy wzroście mocy obliczeniowej).
- wpływ wielu rdzeni na szybkość postawienia systemu operacyjnego; jeśli zrównoleglone operacje są wewnętrznie sekwencyjne, to dalsze zrównoleglanie nie da efektu
- nie uwzględnia bariery przepustowości pamięci oraz we/wy,
- zrównoleglanie algorytmu o złożoności >O(N) jest nieefektywne (Snyder: zrównoleglenie części algorytmu o złożoności O(N^3) powoduje wzrost jego rozmiaru o ok. 26%)

Prawo Gustafsona – perspektywa lepszego użycia zasobów w ustalonym czasie.

- wzrost mocy obliczeniowej umożliwi dokładniejszą analizę lecz nie większą skalę obliczeń.
- wzrost mocy spowoduje oczekiwanie wzrostu możliwości systemu, nawet kosztem jego dłuższej inicjalizacji systemu
- zrównoleglenie zadań wykonywanych na ograniczonym zbiorze danych (np. populacja ludności Ziemi) jest mało skuteczne w kontekście użycia zwiększonych zasobów,
- ograniczenia wykonania części sekwencyjnej mogą być zrekompensowane

48

SKALOWANIE

niedziela, 23 czerwca 2024 00:05

SILNE:

Mamy problem o stałym rozmiarze i uruchamiamy go na coraz szybszych maszynach lub coraz większej liczbie procesorów (można to adaptować jednak na inne parametry).

- Na podstawie prawa Amdahla
- Mierzona wydajność w funkcji np. liczby procesorów dla problemów o ustalonym rozmiarze
- Wraz ze wzrostem liczby procesorów:
  - rozmiar problemu na procesor spada
  - coraz bardziej widoczne narzuty „organizacyjne", np. komunikacja pomiędzy procesorami, przygotowanie danych, itp. „dodatkowe" operacje

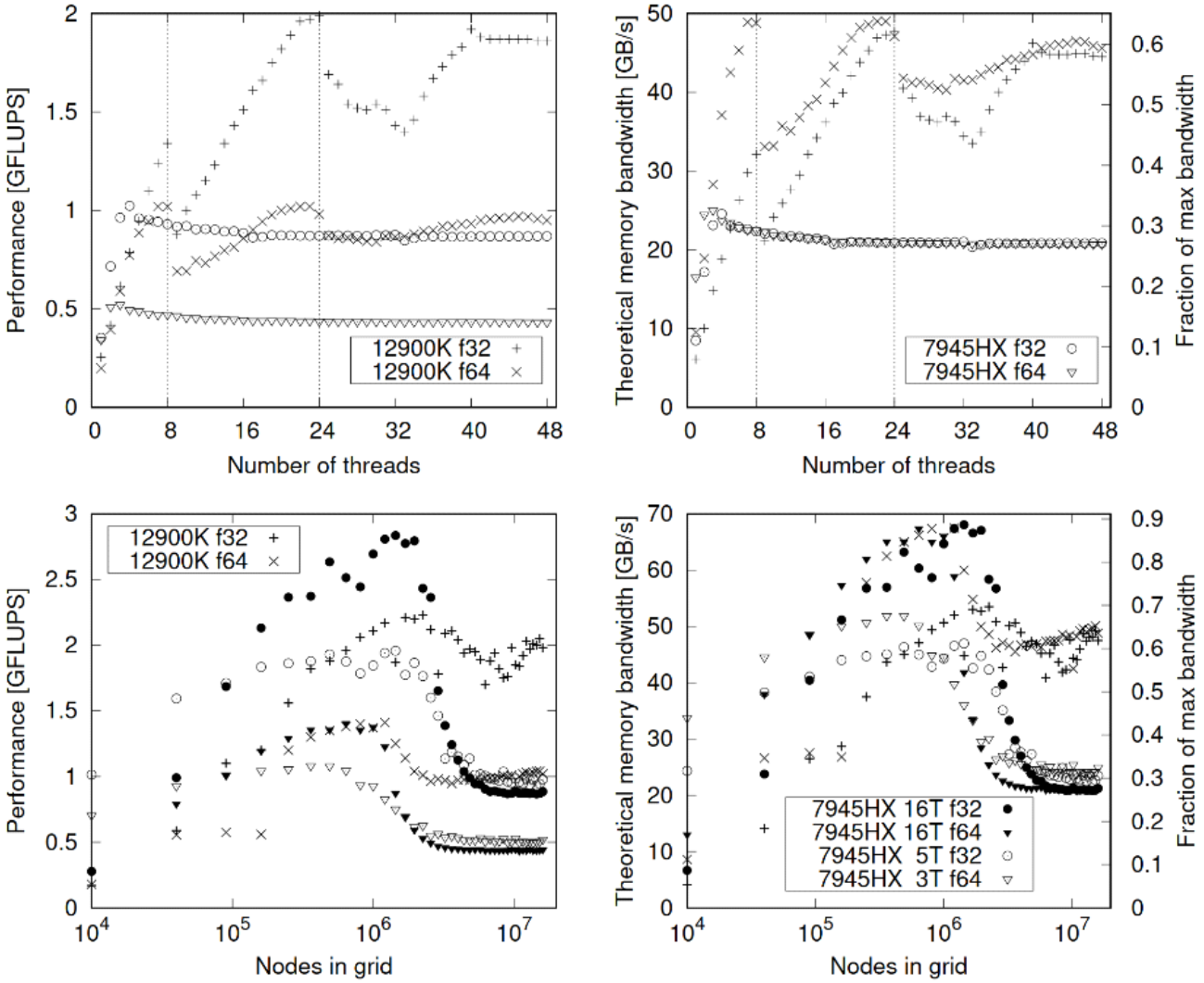
W miarę wzrostu wydajności maszyny przyspieszenie będzie się wypłaszczało i asymptotycznie dążyło do jakiejś wartości. Przyspieszamy część kodu i coraz większe znaczenie ma część, której nie przyspieszamy.
Im problem na procesor większy tym większe znaczenie ma komunikacja między procesorami.

SŁABE:

"Wraz ze wzrostem parametrów maszyny np.. Liczby procesorów zwiększamy liniowo rozmiar problemu tak, żeby w przeliczeniu na jeden umowny procesor maszyny rozmiar problemu był stały " 4 razy więcej procesorów ---- > 4 razy większy problem. Powinniśmy obserwować praktycznie liniowy wzrost wydajności w stosunku do czasu zużytego na rozwiązanie tego większego problemu na jednym procesorze - na tej słabszej maszynie." Natomiast różnie to bywa, bo może być tak, że ta część nieprzyspieszana jakoś wpływa na część przyspieszaną, szczególnie jeśli zwiększamy problem. Dla większego problemu ta część nieprzyspieszana może się inaczej zachowywać niż dla mniejszego.

- Na podstawie prawa Gustafsona
- Mierzona wydajność w funkcji np. liczby procesorów dla problemów o ustalonym rozmiarze **na pojedynczy procesor**
- Wraz ze wzrostem liczby procesorów:
  - rozmiar całego problemu rośnie
  - dodatkowe operacje mogą zajmować coraz więcej czasu!

*Praktyka:



Na obrazku widzimy pomiary rzeczywistej wydajności. To na górze to w zasadzie skalowanie silne, a to na dole jest trochę związane ze skalowaniem słabym, ale nie do końca.
Wyniki dla algorytmu napisanego przez Tomczaka i ekipę przetestowany na dwóch procesorach czyli intelu dwie generacje wstecz i najnowszym laptopowym amd. Krzyżyki to jest intel, a kółeczka i trójkąty to są AMD.
Na pierwszym wykresie widać, że:

a) Do tych pierwszych ośmiu rdzeni szybkich wydajność ładnie rosła wraz ze wzrostem liczby rdzeni.
b) Jak się zaczęły włączać wolne rdzenie wydajność zmalała, ale potem nadal rosła
c) Kiedy przekroczyliśmy liczbę rdzeni w procesorze wydajność zaczęła spadać, bo uruchomione wątki zaczęły konkurować o dostępność do fizycznej… (nierozpoznawalne: pamięci?)

W AMD wydajność bardzo szybko dochodzi do maksymalnej wartości, a potem stosunkowo maleje pomimo tego, że w tym procesorze jest trochę więcej rdzeni. Co więcej ten górny punkt jest znacznie niższy niż w intelu, z powodu nie dlatego, że AMD jest wolniejsze tylko miało wolniejszą pamięć. No bo był laptopowy. I tam pamięć jest dużo słabsza.

Pod spodem:
Wzrost rozmiaru problemu, ale przy ustalonej liczbie elementów liczących (rdzeni). Kiedy problem rośnie to wydajność też rośnie, ale potem gwałtownie spada i się stabilizuje.
Na początku rozmiar problemu na procesor był bardzo mały i narzuty były bardzo widoczne. Jak zwiększaliśmy rozmiar problemu narzuty znaczyły coraz mniej w stosunku do obliczeń.
W pewnym momencie skończyła się pamięć podręczna. I wydajność zaczęła spadać a potem się wypłaszczyła na poziomie ograniczonym przez przepustowość pamięci czyli znaleźliśmy

się wracając do modelu sufitowego znaleźliśmy się na prostej.

Po co optymalizować kod?

niedziela, 23 czerwca 2024 00:48

● Skrócenie czasu wykonania (prawo Amdahla?)
A BARDZO CZĘSTO NAWET BARDZIEJ:
● Zwiększenie przepustowości (prawo Gustafsona?)
Ilość danych przetwarzanych w jakiejś jednostce czasu, te dane nadchodzą stale czyli możemy rozmiar problemu zwiększać. Czy czas utrzymuje się stale, a my cały czas zwiększamy przepustowość, polepszamy kod.
Te dwa parametry są ze sobą powiązane w określonych warunkach.
W praktyce jak się jedno polepszy to się drugie pogorszy więc trzeba tak kombinować, żeby średnio wyszło lepiej niż było. TO JEST SZTUKA
Znany jest model sufitowy i pewne związane z nim granice, reszta zależy od programisty. Co więcej czasami można przebudować algorytm, żeby przesunąć te granice, zmniejszyć intensywność algorytmu lub stosować inny parametr maszyny.

Prawo (Do)Little'a

niedziela, 23 czerwca 2024 00:56

$$n = T \cdot l$$

W stanie ustalonym (**stacjonarnym**) **średnia** liczba aktualnie wykonywanych operacji n jest iloczynem tempa napływania operacji T (przepustowości) i czasu trwania operacji (opóźnienia) l.

Pierwsza sytuacja:
Pewna operacja trwa 16 cykli zegara. Nasz system pozwala na wykonywanie jednej operacji w danej chwili.
Z prawa Little-a wychodzi, że maksymalna przepustowość dla naszego systemu to 1/16 operacji na cykl.

Druga sytuacja:
Gdybyśmy na poziomie tych samych operacji chcieli uzyskać na poziomie jednej operacji na cykl to zgodnie z Prawem Little'a musimy mieć naraz w systemie non stop wykonywane 16 operacji np. szesnaście jednostek wykonawczych.

WAŻNE! Prawo Little'a działa w stanie ustalonym kiedy operacje stale przychodzą i stale mamy co robić.

Ślimak wśród trawy

niedziela, 23 czerwca 2024 11:10

Metody profilowania kodu (znajdowania gorących obszarów):

- Instrumentacja kodu (*ang. code instrumentation*)
  - Konieczna modyfikacja kodu źródłowego lub maszynowego, modyfikowane biblioteki
  - Przykład: gprof
- Symulatory
  - Wysokie narzuty (zależne od technologii)
  - Przykład: valgrind
- Próbkowanie, w tym statystyczne
  - Różne techniki: przerwania zegarowe, debuger (!), liczniki sprzętowe (*ang. hardware performance counters*), modyfikowane biblioteki (*ang. hooks*)
  - Mała dokładność dla rzadko wykonywanego lub **krótkiego** kodu 56
  - Przykłady: oprofile, perf, Intel VTune, AMD µProf

Sposób 1:
Instrumentacja kodu - modyfikowanie kodu w taki sposób, żeby podczas wykonywania informował ile dana część kodu zajmuje. Ręczne printfy albo narzędzia automatyczne. Mało wydajne i mało wiarygodne.

Sposób 2:
Symulatory, nazwijmy je roboczo "maszynami wirtualnymi", które symulują działanie procesora i dla tej maszyny, a ponieważ to jest symulowane to możemy sobie wszystko dokładnie obejrzeć. Wada: Strasznie wolne, trzeba symulować procesor i spisywać rzeczy które się dzieją w trakcie. Wiele rzędów wolniejsze niż normalne uruchomienie.

^
|
|
TE METODY SĄ ŚREDNIE.
TO CO NAS NAJBARDZIEJ INTERESUJE:

Sposób 3:
Próbkowanie statystyczne - uruchamiamy program i co jakiś czas go przerywamy i patrzymy gdzie trafiliśmy, zapisujemy gdzie trafiliśmy i gdzie byliśmy,

Przykład: Milion razy przerwiemy nasz program i pół miliona razy trafiło w okolicach jakiejś instrukcji, to wiadomo, że połowa czasu jest gdzieś w tym fragmencie kodu.

Mamy dostępne różne warunki przerywania programu:

a) Co ileś milisekund
b) Liczby zakończonych instrukcji
c) Liczby transakcji do pamięci
d) Cokolwiek co sobie wymarzymy!

Zalety: Całkiem sprawnie to działa i nie wymaga modyfikacji kodu. Narzuty na działanie kodu są stosunkowo niewielkie. Jak dobrze się poustawia to są prawie pomijalne, na poziomie pojedynczych procentów.
Wady:
1. To profilowanie daje wyniki niedokładne: Jeśli dostaniemy informację, że 13,5 % program spędza w danej instrukcji to nie znaczy, że spędza tam 13,5 %, bo to jest próbkowanie statystyczne - mniej niż jedna piąta, albo w okolicach jednej piątej całego czasu.
2. Jeśli jakieś instrukcje są wykonywane rzadko to pewnie w ogóle nigdy tam nie trafimy. Profilowanie powie nam, że one wcale nie wpływają na wydajność, a one jednak wpływają.

Natomiast to jest doskonała technika to wykrywania gorących obszarów w kodzie!!!

Metoda dla biednych: Przy uruchamianiu programu przerywamy go debuggerem dziesięć, dwadzieścia razy i jeśli trafimy w jakiś miejsce to znaczy, że tam spędza najwięcej czasu.

Tomczak korzysta z perfa, bo jest przenośny i za darmo w Linuxie. To narzędzie po prostu działa!

Kod do profilowania

niedziela, 23 czerwca 2024 11:51

```
1 int main()
2 {
3   long long sum = 0 ;
4   for (long long i=0 ; i < 1000000000 ; i++)
5   {
6     sum += i * i ;
7   }
8   return sum ;
9 }
```

Jedno dodawanie, jedno mnożenie i nic więcej. Nie ma nawet operacji na pamięci. Do tego:

```
$ g++ -O0 -ggdb -m32 main.cpp -o p0
$ perf stat taskset -c 1 ./p0
```

Jest skonfigurowany z niskim poziomem optymalizacji.
-O0 to wyłączenie wszystkich optymalizacji ,żeby kompilator z ty nic nie zrobił.
Perf stat - próbkowanie statystyczne

taskset - przywiązanie do konkretnego rdzenia, żeby się nie przemeiszczał
Uruchamiamy ten program pod kontrolą profilowania statycznego
Wyświetlane jest to:

```
Performance counter stats for 'taskset -c 1 ./p0':

          1 361,07 msec task-clock                #    0,999 CPUs utilized
                 5      context-switches          #    3,674 /sec
                 1      cpu-migrations            #    0,735 /sec
               105      page-faults               #   77,145 /sec
     6 757 368 899      cycles                    #    4,965 GHz
            51 393      stalled-cycles-frontend   #    0,00% frontend cycles idle
        15 943 852      stalled-cycles-backend    #    0,24% backend cycles idle
    19 005 096 889      instructions              #    2,81  insn per cycle
                                                  #    0,00  stalled cycles per insn
     1 001 087 460      branches                  #  735,517 M/sec
           133 597      branch-misses             #    0,01% of all branches

       1,362121726 seconds time elapsed

       1,361671000 seconds user
       0,000000000 seconds sys
```

Co tu widać:

a) Minęło tyle cykli
b) Czas:

Time elapsed - czas ścienny
Task-clock - czas procesora na wykonanie zadania
User - czas użytkownika
Sys - czas systemowy (zero nie, bo były uruchamiane funkcje systemowe)

c) Ile instrukcji się wykonało
d) Ile instrukcji na jeden cukl wychodzi (insn per cycle) = prawie 3
e) Iles skoków zostało wykonanych (branches)

Inny przykład:

```
$ perf record taskset -c 1 ./p0
…
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0,211 MB perf.data (5459 samples) ]
$ perf report
```

```
Samples: 5K of event 'cycles', Event count (approx.): 6758751013
Overhead  Command  Shared Object     Symbol
  99,97%  p0       p0                [.] main
   0,03%  p0       [unknown]         [k] 0xffffffff938d5670
   0,00%  taskset  [unknown]         [k] 0xffffffff93c618fa
   0,00%  taskset  [unknown]         [k] 0xffffffff94856909
   0,00%  perf-ex  [unknown]         [k] 0xffffffff93b3ecb6
   0,00%  taskset  [unknown]         [k] 0xffffffff9393c857
   0,00%  perf-ex  [unknown]         [k] 0xffffffff93967cd6
   0,00%  taskset  [unknown]         [k] 0xffffffff9385c710
   0,00%  perf-ex  [unknown]         [k] 0xffffffff938b9464
   0,00%  taskset  [unknown]         [k] 0xffffffff938b9464
```

Przy tym uruchomieniu jest nagrywany nasz proces i można zobaczyć, w której funkcji nasz proces spędził najwięcej czasu.

A jak najedziemy na to main i wciśniemy "chyba enter":

```
Percent
                for (long long i=0 ; i < 1000000000 ; i++)
                  movl  $0x0,0x8(%esp)
                  movl  $0x0,0xc(%esp)
                ↓ jmp   68
                {
                sum += i * i ;
  6,11    34:   mov   0xc(%esp),%eax
  5,27          imul  0x8(%esp),%eax
  5,39          mov   %eax,%edx
  4,67          mov   0xc(%esp),%eax
  4,89          imul  0x8(%esp),%eax
  4,53          lea   (%edx,%eax,1),%ecx
  4,10          mov   0x8(%esp),%eax
  8,52          mull  0x8(%esp)
  3,49          add   %edx,%ecx
  2,64          mov   %ecx,%edx
  6,04          add   %eax,(%esp)
  7,13          adc   %edx,0x4(%esp)
                for (long long i=0 ; i < 1000000000 ; i++)
  6,74          addl  $0x1,0x8(%esp)
  6,50          adcl  $0x0,0xc(%esp)
  3,00    68:   mov   $0x3b9ac9ff,%edx
  3,79          mov   $0x0,%eax
  5,29          cmp   0x8(%esp),%edx
  6,04          sbb   0xc(%esp),%eax
  5,86          jge   34
                }
                return sum ;
                  mov   (%esp),%eax
                }
                  leave
                ← ret
```

```
Samples
                for (long long i=0 ; i < 1000000000 ; i++)
                  movl  $0x0,0x8(%esp)
                  movl  $0x0,0xc(%esp)
                ↓ jmp   68
                {
                sum += i * i ;
  333     34:   mov   0xc(%esp),%eax
  287           imul  0x8(%esp),%eax
  294           mov   %eax,%edx
  254           mov   0xc(%esp),%eax
  267           imul  0x8(%esp),%eax
  246           lea   (%edx,%eax,1),%ecx
  224           mov   0x8(%esp),%eax
  464           mull  0x8(%esp)
  188           add   %edx,%ecx
  142           mov   %ecx,%edx
  329           add   %eax,(%esp)
  389           adc   %edx,0x4(%esp)
                for (long long i=0 ; i < 1000000000 ; i++)
  366           addl  $0x1,0x8(%esp)
  355           adcl  $0x0,0xc(%esp)
  164     68:   mov   $0x3b9ac9ff,%edx
  206           mov   $0x0,%eax
  288           cmp   0x8(%esp),%edx
  330           sbb   0xc(%esp),%eax
  320           jge   34
                }
                return sum ;
                  mov   (%esp),%eax
                }
                  leave
                ← ret
```

Możemy sobie wybrać wyniki na poziomie poszczególnych instrukcji, można wybrać czy przy każdej instrukcji ma wyświetlać procenty w stosunku do całości czy liczbę trafień!!! Jeśli liczba trafień jest bardzo mała wyniki są bardzo niewiarygodne. Jak trafimy 3 razy to nie wiadomo, jak 103 to pewnie już gdzieś w tej okolicy mogło być.

Gorętsze fragmenty na czerwono.

Tomczak dodał jeszcze funkcję gdb, która do pliku wykonywalnego dodała informację o powiązaniu instrukcji asemblerowych z liniami kodu źródłowego. Dzięki czemu oprócz funkjcji asemblerowych mamy przeplecione fragmenty kodu w c, którym te instrukcje "mniej więcej odpowiadają".

"Mniej więcej", bo nie da się tak jeden do jednego powiązać kodu w c z kodem asemblerowym. Kod asemblerowy będzie robił to co jest zapisane w c, natomiast na pewno w innej kolejności.

Odpowiedź na pytanie:

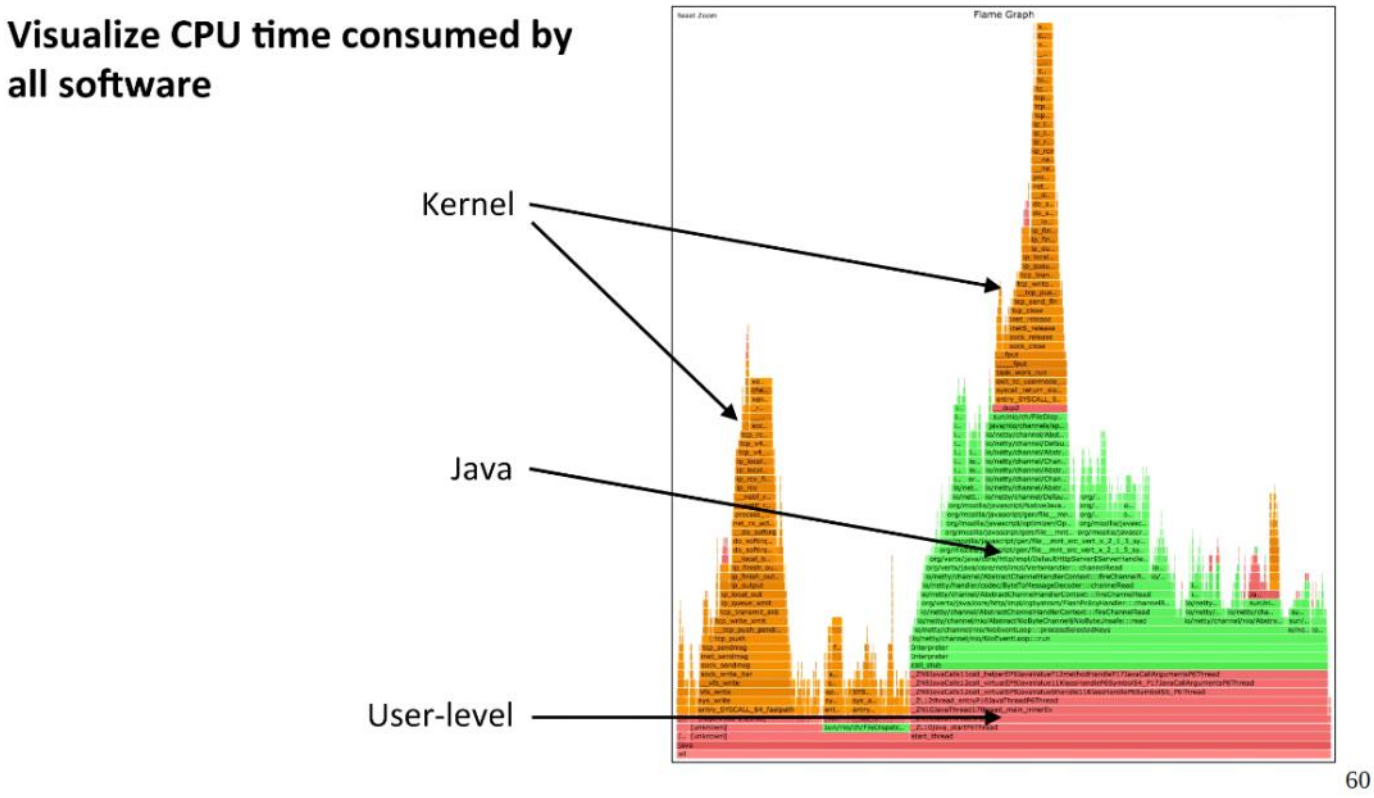
Jeżeli stosujemy normalne próbkowanie to ono przerywa co ileś cykli zegara i trafiamy częściej w instrukcje, które są częściej wykonywane. Natomiast jeśli instrukcja jest rzadko wykonywana to najpewniej w ogóle jej nie znajdziemy. Tyle, że ich może być dużo.

Natomiast dużo mówią nam te procenty, zsumowane określają długość instrukcji stosunkowo do całego czasu.
Natomiast nie należy tego brać dosłownie, że mnożenie wyżej zajmuje dokładnie 8,5 %. Wcale tak nie musi być. To jest tylko pogląd.

"Mapy ciepła"

niedziela, 23 czerwca 2024 12:17

Tak naprawdę "Flame Graph":



Z większych funkcji (od dołu) wywoływane są mniejsze funkcje (w górę), które zajmują coraz mniej czasu.
W przypadku profilowania bardzo ważnym parametrem jest to czy czas podawany jest podawany tylko dla funkcji z wyłączeniem innych funkcji, które są wewnątrz tej funkcji czy jest podawany dla funkcji i wszystkich funkcji wywoływanych z tej funkcji.
XDDDDDD
Czasy z poprzedniego obrazka z poprzedniej sekcji obejmowały też wszystkie wywoływane w danej funkcji.
W oryginale to się nazywa inclusive (wszystko razem z dziećmi) i exclusive (bez potomków).

Albo z psem:

Coś tam można z tych map wywnioskować, głównie to, że jakie funkcje ile potomków mają i coś tam wyciągnąć.

Optymalizacje kompilatora

niedziela, 23 czerwca 2024 12:23

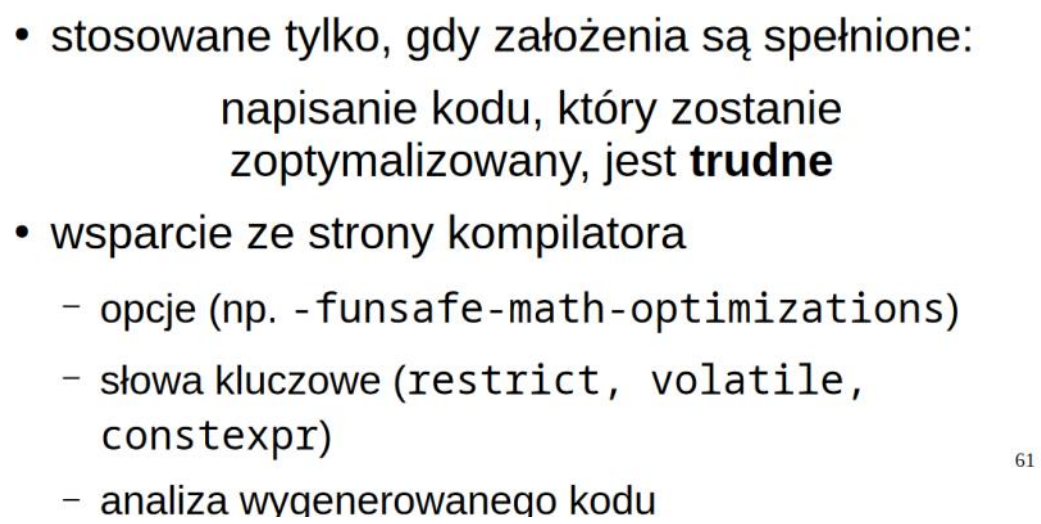
Optymalizacji kompialtora jest bardzo dużo. Są specyficzne, bardzo precyzyjnie opisane. Trzeba rozumieć które włączyć kiedy i co tam dalej zrobić. Czasami jest tak, że nie do końca wiadomo i trzeba próbować. Optymalizacje modyfikują działanie kodu. Bardzo często dość intensywnie i twórczo korzystają z rzeczy, które w standardzie języka są niezdefiniowane. Czyli jeśli coś w standardzie języka jest niezdefiniowane do kompilator wykorzysta to na naszą szkodę.
Kod będzie szybszy, ale może się zachowywać inaczej niż się spodziewamy.

Szczególnie przy wysokich poziomach optymalizacji trzeba bardzo dobrze rozumieć język, którego się używa, ze wszystkimi niuansami.

Żeby optymalizacje mogły być zastosowane przez kompilator to musi być udowodniona ściśle, fomlanie możliwość ich zastosowania. Kompilator buduje sobie drzewo w wyrazie i jeśli w tym drzewie wszystkich zależności jest w stanie coś wykryć, formalnie udowodnić na podstawie teorii grafów to wtedy może część grafów wyrzucić, albo zastąpić czymś innym.

Podstawowe poziomy optymalizacji - od O0 do O3, im wyższy numer tym większy stopień optymalizacji.

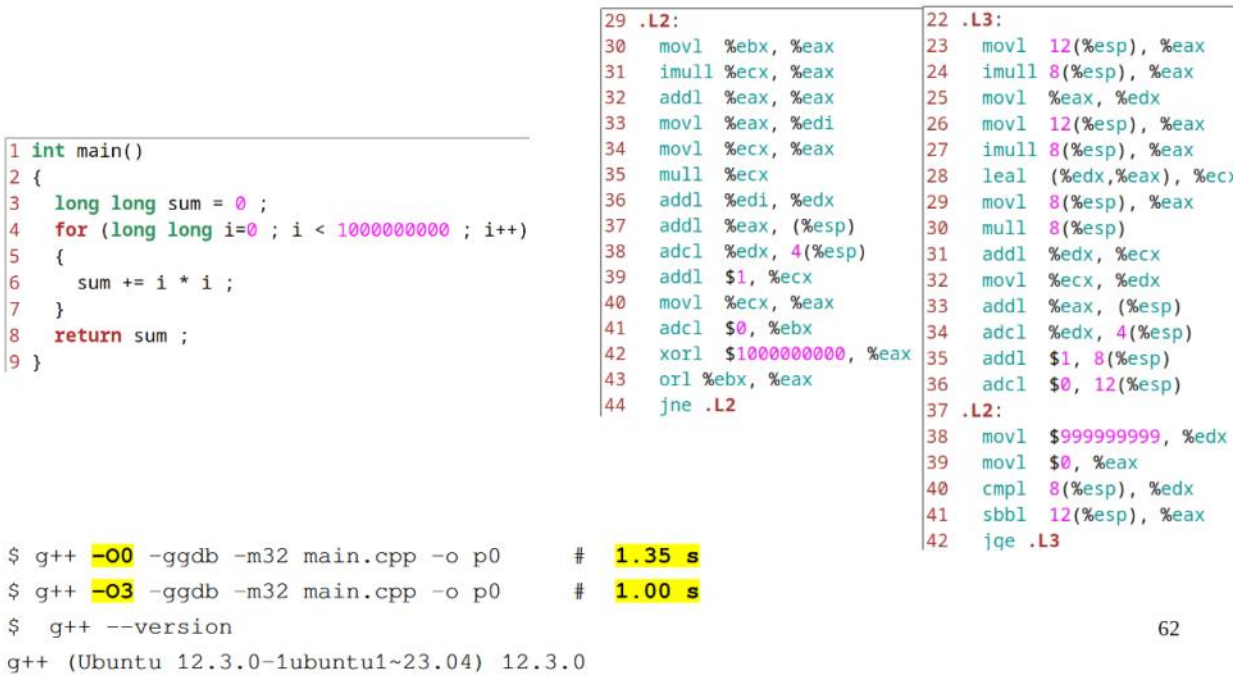
Tu zajrzeć po więcej:
http://www.compileroptimizations.com



restrict - wskaźniki przekazywane dotyczą tylko tego obszaru danych, które dotyczą tego wskaźnika i inny wskaźnik na ten sam obszar nie będzie wskazywał, nie trzeba tych danych drugi raz czytać

ZAWSZE NALEŻY OGLĄDAĆ TEN KOD, KTÓRY WYGENERUJE KOMPILATOR, ZAWSZE!

Przykład:



```
1 int main()
2 {
3   long long sum = 0 ;
4   for (long long i=0 ; i < 1000000000 ; i++)
5   {
6     sum += i * i ;
7   }
8   return sum ;
9 }
```

```
29 .L2:
30   movl  %ebx, %eax
31   imull %ecx, %eax
32   addl  %eax, %eax
33   movl  %eax, %edi
34   movl  %ecx, %eax
35   mull  %ecx
36   addl  %edi, %edx
37   addl  %eax, (%esp)
38   adcl  %edx, 4(%esp)
39   addl  $1, %ecx
40   movl  %ecx, %eax
41   adcl  $0, %ebx
42   xorl  $1000000000, %eax
43   orl %ebx, %eax
44   jne .L2
```

```
22 .L3:
23   movl  12(%esp), %eax
24   imull 8(%esp), %eax
25   movl  %eax, %edx
26   movl  12(%esp), %eax
27   imull 8(%esp), %eax
28   leal  (%edx,%eax), %ec
29   movl  8(%esp), %eax
30   mull  8(%esp)
31   addl  %edx, %ecx
32   movl  %ecx, %edx
33   addl  %eax, (%esp)
34   adcl  %edx, 4(%esp)
35   addl  $1, 8(%esp)
36   adcl  $0, 12(%esp)
37 .L2:
38   movl  $999999999, %edx
39   movl  $0, %eax
40   cmpl  8(%esp), %edx
41   sbbl  12(%esp), %eax
42   jge .L3
```

```
$ g++ -O0 -ggdb -m32 main.cpp -o p0      #  1.35 s
$ g++ -O3 -ggdb -m32 main.cpp -o p0      #  1.00 s
$  g++ --version
g++ (Ubuntu 12.3.0-1ubuntu1~23.04) 12.3.0
```

62

Ta sama funkcja co wcześniej, po lewej O0, po prawej O3. Po optymalizacji zarobiliśmy 35% czasu wykonania. Nic szczególnego, jest mniej instrukcji i jest mniej operacji na pamięci w odzie zoptymalizowanym i tyle.

Ciekawszy przykład:

```
1 int main()
2 {
3   long long sum = 0 ;
4   for (long long i=0 ; i < 1000000000 ; i++)
5   {
6     sum += i ;
7   }
8   return sum ;
9 }
```

```
1 int main()
2 {
3   long long sum = 0 ;
4   for (long long i=0 ; i < 1000000000 ; i++)
5   {
6     sum += i ;
7   }
8   return 0 ;
9 }
```

```
 7 main:
 8 .LFB0:
 9   .cfi_startproc
10   movl  $-1243309312, %eax
11   ret
```

```
 7 main:
 8 .LFB0:
 9   .cfi_startproc
10   xorl  %eax, %eax
11   ret
```

63

W pierwszej kolumnie kompilator od razu wyliczył wynik pętli.
W drugiej kolumnie widzimy, że wynik z pętli nie jest nigdzie dalej używany, kod został uznany za martwy (dead code) i przez kompilator wyeliminowany.

KOMPILATORY MOGĄ ZDZIAŁAĆ CUDA!

Z DRUGIEJ STRONY JEŚLI CHCIELIBYŚMY MIERZYĆ CZAS DLA KONKRETNEGO PRZYPADKU NA PEWNYM SPRZĘCIE MUSIMY UNIKAĆ TEGO TYPU OPTYMALIZACJI!

W gcc jest opcja pozwalająca wyświetlić decyzje podjęte przez kompilator w trakcie optymalizacji.

Równoległość drobnoziarnista

niedziela, 23 czerwca 2024 13:06

Będziemy się zastanawiali jak sprawić, żeby jeden proces na jednej maszynie wykorzystywał maksymalnie jej możliwości. Jest rząd lub dwa wielkości do ugrania. Za jeden procesor Xeonowy trzeba teraz chyba zapłacić 3-5 dolarów. Jeśli chcielibyśmy sto to się robi gruba suma.

Dlaczego równoległość? Na wykresach z FPGA (patrze. Wykład 5 "Ołtarzyk Samouwielbienia") podstawową techniką uzyskiwania wydajności jest to, że będziemy robili wiele rzeczy naraz.

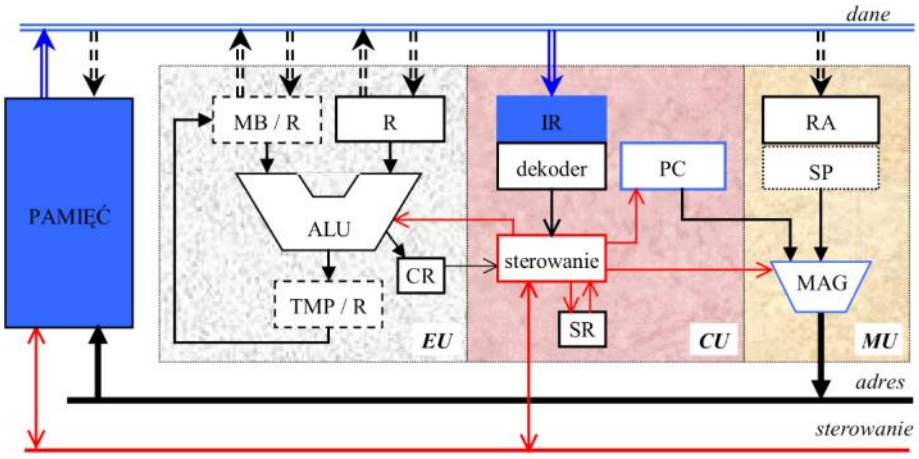
Druga rzecz: Jak spowodować, żeby te najdłuższe rzeczy trwały krócej? (W dalszej części kursu).4

Najczęściej będziemy się zajmowali równoległością na poziomie instrukcji czyli możliwością wykonywania w tym samym czasie sąsiednich instrukcji, czyli zamiast wykonywać jedną instrukcję a potem dugą bęidzemy dtarali się wykonywać wiele instrukcji naraz.

Na poprzednich wykładach mówiliśmy w zasadzie o liście rozkazów czyli o tym jakie mamy rozkazy, jakie mogą być argument itd.
Natomiast teraz będziemy się zajmowali wewnętrzną konstrukcją procesora na poziomie bramek logicznych, przerzutników, pamięci. Do tranzystorów nie dojdziemy wcale, bo to nam do niczego nie potrzebne.

Tomczak wspomina stare dobre obrazki prof. Biernata z Wykładu 1:

Organizacja procesora sekwencyjnego (F)



- *F* (pobranie kodu): adres (PC) → pamięć → rejestr rozkazów IR

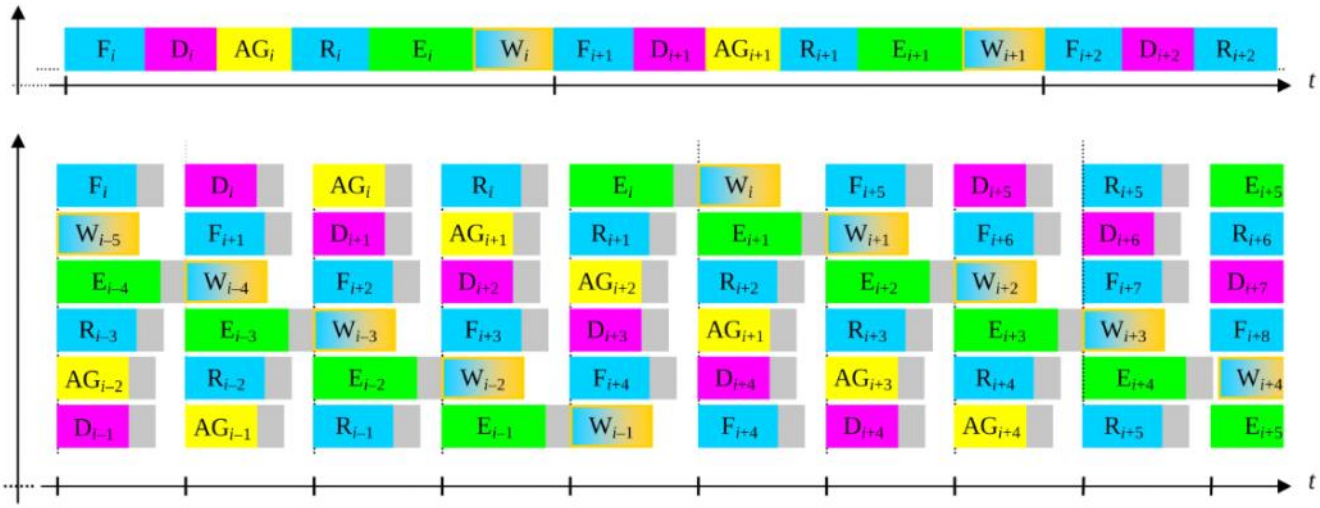
Wspomina o cyklu rozkazowym.
O tym, że w taki sposób są budowane jednak tylko najmniejsze, najprostsze procesory.

Odwołując się do obrazka wyżej, dlaczego reszta elementów nic nie robi? Mogłaby.
Przepływ zawsze jest w jedną stronę.
Dałoby się tak zrobić, żeby pracowały wszystkie elementy i wykonywały kilka rozkazów, każdy na innym etapie.
Jak na linii produkcyjnej.

Przetwarzanie potokowe

niedziela, 23 czerwca 2024 13:22

Potok CISC - architektura R/M



Każdy etap cyklu rozkazowego wykonuje specjalizowany układ funkcjonalny:

- pobranie rozkazu **z pamięci** (ang. *fetch*) – wskaźnik rozkazów i układ odczytu
  - rozmiar kodu rozkazu
  - czas dostępu do pamięci (odczyt słowa lub jego części)
- **dekodowanie** (ang. *decode*) – dekoder rozkazu
  - architektura listy rozkazów (złożoność i różnorodność działań)
  - struktura kodu rozkazu (niejednorodność)
- pobranie operandu **z pamięci** (ang. *data read*) – układ adresowania i odczytu
  - tryb adresowania i czas dostępu do pamięci (odczyt)
- **wykonanie** (ang. *execute*) – układ/układy wykonawcze (ALU/FPU/...)
  - złożoność wykonywanych działań
- zapis wyniku do **rejestru** (ang. *put away*) [lub **do pamięci** (ang. *data write*)]
  [czas dostępu do pamięci (zapis słowa lub jego części)]

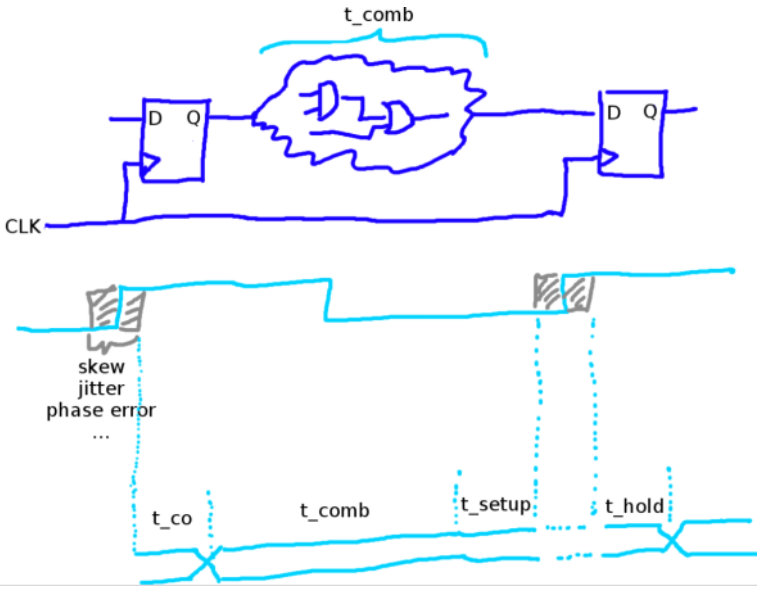
Przepływ danych między układami funkcjonalnymi jest **jednokierunkowy**

- możliwe *jednoczesne wykonanie różnych etapów*
- szybkość wykonania ogranicza *najdłuższy etap* i *narzut separacji etapów*

Dzielimy procesor na etapy, tworzymy "linię produkcyjną" możemy w danym cyklu zegara wykonywać 6 różnych rozkazów (6 etapów) każdy na swoim etapie. Wszystkie układy w potoku cały czas pracują.

Kilka problemów:

a) W przetwarzaniu potokowym wszystkie etapy muszą trwać tyle samo czasu. Muszą być tak długie jak najdłuższy z nich.
b) Żeby można było w różnych fragmentach procesora wykonywać różne etapy różnych rozkazów układy wykonawcze muszą być od siebie odseparowane w przerzutnikach, przerzutniki wprowadzają dodatkowe czasy, które wynikają z ich konstrukcji



Mamy naszą część kombinacyjną i dwa przerzutniki, które separują część kombinacyjną od innych części kombinacyjnych.

t_com - czas przetwarzania danych w części kombinacyjnej

Pierwszy problem, jak na wejściu przerzutnika ustabilizuje się wartość sygnału możemy podać zbocze sygnału zegarowego. Ale zbocza sygnału zegarowego przemieszczają się w czasie z różnych powodów:

a) Z powodu temperatury
b) Wahań napięcia zasilania
c) Doregulowują się, proces regulacji, która wymaga wahania częstotliwością
d) Sieć rozprowadzająca sygnał zegarowy się różnie zachowuje
e) Cała masa innych zjawisk…

Drugi problem: Jak już zbocze sygnału się ustabilizuje to potem jest czas potrzebny na propagację sygnału od wejścia przerzutnika do wyjścia

t_co - czas potrzebny na propagację sygnału od wejścia przerzutnika do wyjścia (clock to output)

Teraz przychodzi czas na część kombinacyjną, uwaga!!!
Żeby kolejny przerzutnik to jakiś czas przed sygnałem zegarowym sygnał na jego wejściu musi być ustalony.

t_setup - czas ustalenia sygnału przed drugim przerzutnikiem

Potem czas o niekreślonej nazwie, w którym nasze zbocze sygnału pływa.

Po zboczu sygnału zegarowego na wejściu przerzutnika ten czas też musi być jeszcze stabliny.

t_hold - czas żeby sygnał mógł jeszcze propagować z wejścia do wnętrza przerzutnika

Dopiero jak te wszystkie czasy zostaną spełnione można zmienić wartość sygnału.

I te wszystkie rzeczy tworzą te szare fragmenty, czas potrzebny na organizacje danych
Im krótsze będą części kombinacyjne tym będą one bardziej widoczne.
Dla danego przerzutnika one zawsze będą wynosiły ustalony czas np. 12 mikrosekund i tego nie przeskoczymy.

Trzeci problem:
Podczas wykonywania rozkazu należy pobrać kod z pamięci, pobrać argumenty i zapisać wynik do pamięci i jeżeli te 3 rzeczy dzieją się stale w każdym cyklu to musimy w każdym cyklu zegara zrobić trzy operacje na pamięci, a są one bardzo wolne.

Są potrzebne albo jakieś specjalne techniki ukrywania operacji na pamięci albo układ będzie działał wolno. Oba rozwiązania są drogie, skomplikowane i wolne. Tego też nie przeskoczymy.

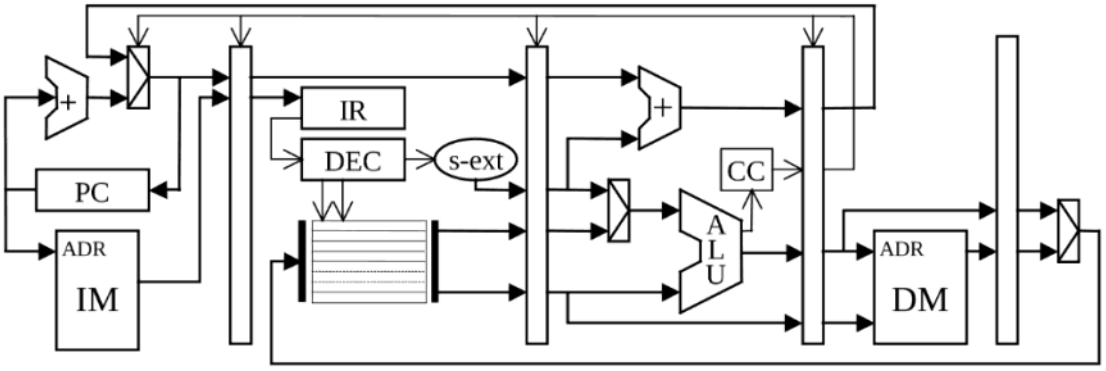
Jeżeli procesor ma rozkazy wykonujące działania na pamięci może się zdarzyć, że :

1. jeden rozkaz będzie odczytywany z pamięci jako kod maszynowy
2. Drugi rozkaz będzie odczytywał z pamięci swój argument
3. Trzeci rozkaz do pamięci zapisywał swój wynik w danym cyklu zegara.

Jeśli źle napiszemy kod!
Ale jeżeli się dobrze zrobi to wtedy działa szybko

Struktura potokowa

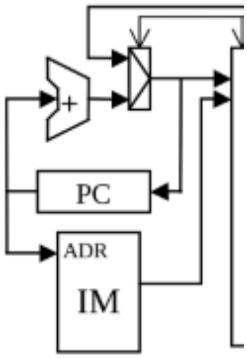
niedziela, 23 czerwca 2024 13:59



Przepływ danych w potoku statycznym

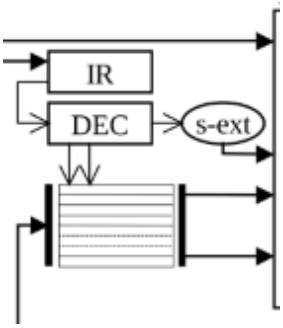
Etapy:

1. :



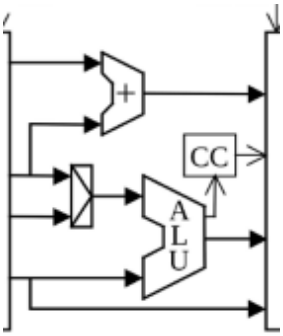
Zwiększany licznik rozkazów, Pobierany z pamięci programu kod rozkazu.

2. :



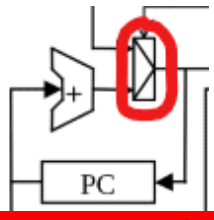
Jest rejestr instrukcji, dekoder. Od razu są wybierane są argumenty z rejestrów

3. :



Podawane na wejścia jednostki arytmetyczno logicznej. Jest jednostka generowania adresu AGU

……..



To jest pewnie jednostka generowania adresu AGU lub skoku, bo to wchodzi do licznika rozkazów
UWAGA! Mogłem się pomylić i może nie chodzić o to miejsce.

……..

4. W kolejnym cyklu zegara do pamięci jest zapisywany wynik.
   W takim układzie mamy 4-etapowy potok i możemy wykonywać 4 rozkazy naraz.

WAŻNE!
Jednak to, że możemy wykonywać 4 rozkazy naraz nie oznacza, że będziemy wykonywać 4 rozkazy naraz.

W potoku bardzo często mogą się pojawić przestoje zwane też "bąbelkami".
Skąd się biorą?
Np. stąd, że rozkaz następny potrzebuje wyniku obliczonego w rozkazie poprzednim. I nie możemy wykonać operacji w rozkazie następnym jeśli nie zakończymy tej operacji z rozkazu wcześniejszego.

Są 3 rodzaje konfliktów powodujących przestój w potoku:

a) Konflikt danych - potrzebujemy jako argumentu wyniku wytworzonego we wcześniejszych rozkazach, które się jeszcze nie zakończyły (adres, flagi, wartości itd.)
b) Konflikty strukturalne - w jakimś etapie wykonywania jakiegoś rozkazu potrzebujemy dostępu do jakiegoś układu, którego w procesorze jest za mało np. musimy wykonywać 3 operacje na pamięci, ale ktoś skonstruował procesor tak, że doszedł do wniosku, że taka sytuacja będzie rzadka więc dajemy tylko jeden układ dostępu do pamięci. Jeśli rozkazy będą chciały mieć naraz dostęp do pamięci drugi musi poczekać, aż ten pierwszy zwolni.

Trochę dziwny fragment:
"Układ dzielący z natury swojej jest bardzo wolny i w zasadzie nieprzyspieszalny to jeśli ktoś wykonuje dzielenie to musi być ono ileś wolniejsze od dodawania i mnożeni, więc to powoduje, że to co wcześniej musiało się wykonać musi się skończyć dziać"

c) Konflikty sterowania - w potoku musimy dostarczać rozkazy co cykl zegara, bo inaczej nie będzie w każdym cyklu zegara przetwarzał nowego rozkazu, a nie jesteśmy w stanie, a w przypadku skoku nie jesteśmy w stanie pobrać nowego rozkazu jeśli skoku nie zakończymy, a skoki warunkowe do decyzji o tym czy zostanie wykonany skok czy nie, do wyznaczenia adresu instrukcji następnej muszą się zakończyć, muszą być zbadane jakieś flagi, trzeba porównać coś z czymś itd. Jeszcze gorzej jeśli adres skoku jest w pamięci, nie możemy wykonać skoku jeśli z pamięci nie odczytamy adresu docelowego skoku.

Tier lista skoków:

- Skoki bezwarunkowe
- Skoki warunkowe niewykonane - na zapas zaczynamy wykonywanie rozkazu

następnego i jeśli skok nie zostanie wykonany i prawie nic nie stracimy

- Skoki warunkowe wykonane - adresem instrukcji powinna być inna niż ta, którą zaczęliśmy wykonywać, ale dowiadujemy się tego dopiero po zakończeniu rozkazu skoku

Slajd z wykładu:

konflikt danych (ang. *data hazard*)

- niedostępność operandu, będącego wynikiem instrukcji nieukończonej

konflikt strukturalny (*zasobu*) (ang. *resource hazard*)

- kilka instrukcji wymaga jednocześnie dostępu do unikatowego zasobu
- instrukcja wymaga kilku etapów EX (ang. *run-on effect*)

konflikt sterowania (ang. *control hazard*)

- wznowienie potoku po pobraniu instrukcji docelowej (ang. *branch target*)
- opóźnienie użycia warunku

→ rozgałęzienie warunkowe niewykonane (ang. *branch not taken*) – instrukcja następna w sekwencji (ang. *in-line instruction*) jest dekodowana dopiero w cyklu następującym po wytworzeniu warunku

→ rozgałęzienie warunkowe wykonane (ang. *branch taken*) – dekodowanie instrukcji docelowej jest możliwe po jej pobraniu i wytworzeniu warunku

→ rozgałęzienie bezwarunkowe (ang. *branch unconditional*) – dekodowanie instrukcji docelowej jest możliwe dopiero po jej pobraniu

CAŁA MASA RÓŻNCYH PROBLEMÓW Z TYM POTOKIEM ŻEBY UDAŁO SIĘ UZYSKAĆ PRZEPUSTOWOŚĆ JEDNEGO ROZKAZU NA CYKL!
TO WYNIKA Z PRAWA LITTLE-A. ROZKAZ TRWA 4 ETAPY, WYKONUJEMY 4 ROZKAZY NA RAZ. UZYSKAMY PRZEPUSTOWOŚĆ: 1 ROZKAZ NA CYKL.

"Kto nie RISC-uje ten nie je"

15:21

Problemy, problemy i jeszcze raz problemy:

1. **Dużo kodów rozkazu i nieregularność:**

W architekturze Intela problemem jest, że kod rozkazu zajmuje od 1 do "chyba" 20 bajtów. Jest nieregularny i nie wiadomo jak to dekodować to siłą rzeczy pobieranie takiego kodu rozkazu i jego dekodowanie musi trwać i musi być warunkowe : jak odczytamy pierwszy bajt i wynika z tego, że musimy doczytać dalej to to robimy
Regularne kody rozkazów:
W ładnych architekturach rozkazy są podzielone na kilka kategorii tylko 3,4,5 i tam są ustalenia typu:

a) Najstarsze 4 bity to jest kod operacji arytmetycznej, zapisane dodawanie, odejmowanie, mnożenie i coś
b) Na kolejnych 4 bitach jest adres rejestru
c) Jeszcze coś innego itd.

Dekoder to parę multiplekserów, które odpowiednie bity wypuszczają na odpowiednie magistrale.

Kolejny problem:
Skomplikowane tryby adresowania: Jeżeli mamy rozkazy operujące na argumentach w pamięci i tryby adresowania są dość skomplikowane (jak w intelu) to obliczenie adresu, który składa się z przemieszczenia, rejestru bazowego, rejestru indeksowego i skali, a potem jeszcze segmentu wymaga więcej czasu niż w sytuacji gdyby tryby adresowania były proste, najlepiej nie wymagające liczenia niczego.

Kolejny problem:
Niejednakowość rozmiaru argumentów:
Jeśli procesor obsługuje argumenty różnego rozmiaru: duże, małe, średnie, bardzo duże to czasy pobierania dużych rozkazów będą dłuższe niż krótkich argumentów i na dodatek będą to czasy nieregularne

Kolejny problem:
Niejednakowość działań:
Im prostsze działania będą tym szybciej możemy je skończyć i to wszystkie. Jak procesor nie będiz emiał dzielenia to nie będzie trzeba się przejmować że długo trwa.

A o de mnie, jeśli nie będziemy mieli rąk nie będziemy musieli się obawiać, że źle napiszemy egzamin.

Kolejny problem:
Zapisywanie wyniku do pamięci:
Jeśli wytworzymy wynik to lepiej by było tego wyniku do pamięci nie zapisywać, bo to trwa tylko gdzieś bliżej

Dawno dawno temu byli sobie ludzie, którzy popatrzyli na to i powiedzieli:
"Rozwiązaniem byłoby zrobienie wszystkiego tego o czym wspomniał Tomczak na wykładzie i skonstruowanie komputera o zredukowanej liczbie rozkazów RISC - Reduced Instruction Set Computer, ten potworek intelowski, o którym mówiliśmy wcześniej to CISC - Complex Instruction Set Computer "

RISC nie oznacza, że ma mało rozkazów, ma dużo! Tyle, że są prostsze do wczytania, zdekodowania i wykonania.

Dlaczego?
Bo:

1. **Wszystkie rozkazy zajmują tyle samo miejsca w pamięci, 32 lub 64 bitowe**
2. **Wszystkie rozkazy są ładnie podzielone**
3. **ALU wykonuje działania tylko na rejestrach (architektura rejestrowa)**
4. **Jeśli chcemy wykonywać operacje na pamięci to do tego są osobne rozkazy (load, store)**
5. ***Ograniczony zakres i krótkie kody stałych - tutaj jest pewien problem, bo jak maszyna jest 32-bitowa, rejestry w tym rozmiarze, kody rozkazów to w kodzie rozkazu natychmiastowego argumentu 32-bitowego nie zmieścimy**

TA ARCHITEKTURA MA NIEWIELKIE OGRANICZENIA, ALE ZALETY PRZEWAŻAJĄ NAD PROBLEMAMI!

W architekturze RISC musi być więcej rejestrów minimum 16, standardowo 32, dalej 64, *+
W takim procesorze jest cała masa pamięci podręcznej czyli buforów w procesorze, które pozwalają maskować oczekiwania na operacje z pamięci. Wtedy zapisujemy dane do bufora i zostawiamy je, niech je sobie bufor do pamięci przepisze, a my idziemy dalej.

Porównanie CISCO i RISC:

Architektura klasyczna CISC

CISC – Complex Instruction Set Computer

Lista rozkazów

- rozkazy realizujące działania proste i skomplikowane
- rozbudowane sposoby (tryby) adresowania
- argumenty umieszczone zwykle w pamięci
- stałe w dodatkowych słowach kodu
- niejednolita struktura – różna liczba słów kodu

Organizacja – rozwiązania intuicyjne

- akumulator lub niewiele rejestrów uniwersalnych
- większość argumentów w pamięci, rejestry specjalizowane
- trudne buforowanie i dekodowanie rozkazów (zmienny rozmiar)

Skutki

- zmienny czas wykonania tych samych etapów przetwarzania
- bariera przepustowości pamięci

Koncepcja RISC

RISC – Reduced Instruction Set Computer

!***redukcja*** *dotyczy* ***różnorodności*** *a nie liczby instrukcji* (*racjonalizacja*)!

Lista rozkazów

- rozkazy proste
- proste tryby adresowania
- specjalne rozkazy komunikacji z pamięcią
- ograniczony zakres i krótkie kody stałych
- jednolita struktura kodu

Organizacja (zasada lokalności, buforowanie informacji)

- dużo rejestrów uniwersalnych
- buforowanie informacji (kolejka rozkazów, cache)

Korzyści

- prawie stały czas wykonania tych samych etapów przetwarzania
- podobny czas wykonania różnych etapów przetwarzania
- łatwa implementacja potoku

Ewolucja CISC → RISC

oczekiwane efekty:

uproszczenie listy rozkazów oraz ujednolicenie struktury słowa kodu:

→ stały czas pobrania kodu i dekodowania rozkazu

→ prosty dekoder kombinacyjny

→ proste układy wykonawcze, krótki czas wykonania podstawowych działań

zwiększenie liczby rejestrów procesora – przechowanie wyników cząstkowych:

→ mniejsza intensywność komunikacji z pamięcią etapy R i W (load/store)

→ rzadkie konflikty dostępu podczas wykonania etapów F, R, W

ograniczenie repertuaru trybów adresowania danych –

→ prostszy układ obliczania adresu

problem: kodowanie argumentów stałych:

→ większość możliwych stałych jest bardzo rzadko używana

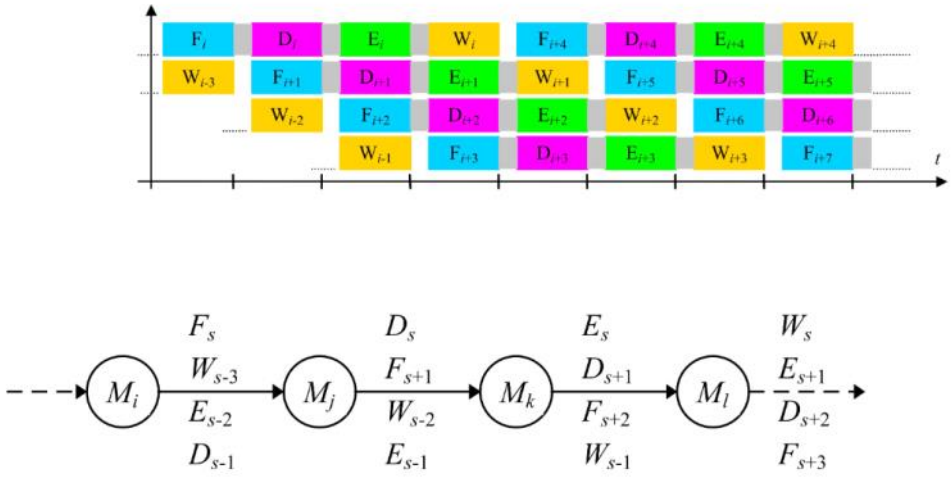
→ stałe często używane mają małe wartości

wniosek: skrócone kodowanie stałych często używanych,
rzadko używane można tworzyć proceduralnie

Jak wygląda potok w RISC?

niedziela, 23 czerwca 2024 16:16

Procesor potokowy RISC



Cykle procesora potokowego RISC

W RISC potok jest prostszy.
Dekodowanie jest krótsze.
Nie a osobnego etapu do pobierania danych, wartości argumentów z pamięci.
Argumenty są tylko w rejestrach, więc możemy adresy tych rejestrów wytworzyć już na etapie dekodowania rozkazu. Bo rejestry są blisko procesora.
Tak samo z zapisaniem wyniku.
Tylko do rejestrów!
Nie potrzebujemy osobnego etapu na operacje na pamięci zewnętrznej tylko wszystko się dzieje na rejestrach.
Tutaj w każdym cyklu jest tylko jedna operacja na pamięci, mianowicie: pobranie kodu rozkazu.
Rozkazy load store występują w kodzie średnio rzadziej więc możemy się zgodzić na drobne komplikacje w potoku:

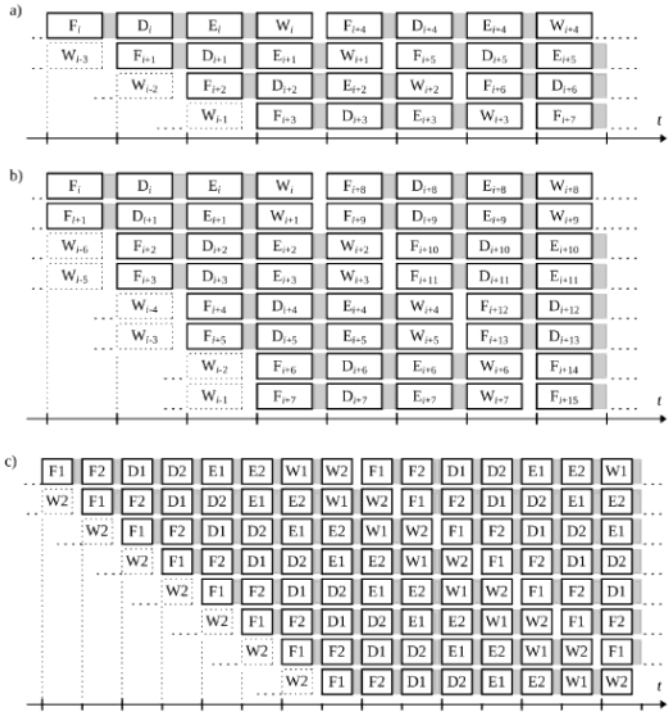
a) W przypadku rozkazu store nie wykonujemy pełnej operacji na pamięci i czekamy, aż zakończymy tylko zapisujemy do jakiegoś bufora, do jakiejś kolejki te dane, które chcemy zapisać do pamięci i zapominamy o nich i przechodzimy dalej, wysłaniem tych danych do pamięci zewnętrznej zajmuje się kolejka.
   Jeśli kolejka się przepełni ---> potok zostanie wstrzymany.
   Jeśli częstotliwość zapisu do pamięci będzie niewielka, żeby kolejka nadążała się opróżniać nie będzie problemu

BWK

niedziela, 23 czerwca 2024 16:27

BARDZO WAŻNA KLASYFIKACJA:

Do tej pory mówiliśmy o procesorach zwykłych potokowych.
Są jeszcze:
a) superpotokowe , w których każdy etap dzieli się na mniejszy, możemy zwiększyć częstotliwość taktowania, za bardzo nie możemy , bo narzuty na obsługę rejestrów staną się zbyt duże, ale
b) superskalarne, w których jest wiele potoków działających równocześnie w ramach jednego procesu.



Przetwarzanie: a) potokowe, b) superskalarne i c) superpotokowe

W tej chwili procesory superpotokowe zawierają mniej więcej tyle etapów:

x86 microarchitectures

Year	Micro-architecture	Pipeline stages	Max clock (MHz)	Process node
2023	Redwood Cove			Intel 4
2023	*Crestmont*			Intel 4
2022	Raptor Cove	12 unified	6200	Intel 7
2021	Cypress Cove	14 unified	5300	14 nm
2021	Golden Cove	12 unified	5500	Intel 7
2021	*Gracemont*	20 unified with misprediction penalty	4300	Intel 7
2020	*Tremont*	20 unified	3300	10 nm
2020	Willow Cove	14 unified	5300	10 nm
2019	Sunny Cove	14–20 (misprediction)	4100	10 nm
2018	Palm Cove	14 (16 with fetch/ retire)	3200	10 nm
2017	*Goldmont Plus*	20 unified with branch prediction (?)	2800	14 nm
2016	*Goldmont*	20 unified with branch prediction	2600	14 nm
2015	*Airmont (die shrink)*	14–17 (16–19 with fetch/retire)	2640	14 nm
2015	Skylake	14 (16 with fetch/ retire)	5200	14 nm
2014	Broadwell (die shrink)	14 (16 with fetch/ retire)	3700	14 nm

DZIĘKUJĘ.