Analiza i modelowanie wydajności obliczeniowej

Krzysztof Banaś

Spis treści

Wstęp

Wydajność obliczeniowa jest jedną z najważniejszych własności oprogramowania (obok takich cech jak np. poprawność, niezawodność, bezpieczeństwo). Optymalizacja oprogramowania pod kątem wydajności (zwana w dalszej części książki po prostu optymalizacją) oznacza dążenie do skrócenia czasu osiągnięcia pożądanego wyniku (*time-to-solution*). Czas uzyskania wyniku, zwany dalej czasem wykonania, jest rozumiany jako czas przekształcenia danych wejściowych dostarczanych programowi w dane wyjściowe, żądane przez użytkownika.

Czas wykonania jest podstawowym parametrem analizowanym w badaniach nad wydajnością obliczeniową. Ostatecznym celem modelowania wydajności jest uzyskanie wzorów pozwalających oszacować czas wykonania konkretnego programu dla konkretnych danych wejściowych. Istotną cechą uzyskiwanych oszacowań ma być uwzględnienie właściwości implementacji, dokonanej w konkretnym środowisku programowania oraz charakterystyk sprzętu, na którym dokonywane są obliczenia. W tym miejscu modelowanie wydajności (*performance modelling*) różni się istotnie od klasycznej analizy złożoności obliczeniowej. W tej ostatniej chodzi o uzyskanie oszacowania czasu wykonania, w którym ważny jest rząd zależności od rozmiaru danych wejściowych (ten ostatni rozumiany jest często w specyficzny sposób, zależny od analizowanych algorytmów). W takim ujęciu zależność od implementacji i sprzętu mieści się w stałych, najczęściej niepodlegających szacowaniu, pojawiających się przy wyrażeniach zawierających parametr określający rozmiar danych.

W analizie wydajności dąży się do uzyskania wzorów wyrażających czas wykonania programów bez nieokreślonych stałych. Oznacza to próby ilościowego ujęcia wpływu własności sprzętu komputerowego wraz z jego oprogramowaniem, systemowym i narzędziowym, na wykonanie programów. Dokonywane jest to poprzez wybór specjalnych parametrów, uzyskiwanych teoretycznie lub eksperymentalnie, umieszczanych we wzorach na czas wykonania.

Wydajność przetwarzania, w najbardziej ogólnym ujęciu, można określić jako odwrotność czasu wykonania. Im krótszy czas wykonania programu, tym wyższa osiągnięta wydajność. Problemem w badaniu wydajności jest fakt, że czas wykonania jest wielkością łatwą do zdefiniowania i zmierzenia, podczas gdy wydajność zazwyczaj ujmuje już specyfikę dziedziny zastosowań i konkretnego badanego programu. W hipotetycznym przypadku programu realizującego sekwencję takich samych operacji, wydajność przetwarzania może być definiowana jako liczba operacji wykonywanych w jednostce czasu. Czas wykonania jest wtedy ilorazem liczby operacji przez wydajność przetwarzania. Najczęściej w praktyce, uzyskanie oszacowań czasu wykonania wymaga zastosowania znacznie bardziej złożonych wzorów i miar wydajności.

Dokonywane w książce analizy wydajności dotyczą programów, a więc implementacji algorytmów w środowiskach programowania. Algorytm jest tutaj rozumiany jako ogólny przepis rozwiązania problemu obliczeniowego. Podstawowym, stosowanym w książce, sposobem zapisu algorytmów jest pseudokod, mający być intuicyjnie zrozumiały (do nadania struktury zapisowi algorytmów stosowane są standardowe konstrukcje wykorzystywane w najpopularniejszych językach programowania). Drobne modyfikacje algorytmu dokonywane w różnych implementacjach, jak np. często stosowana zmiana kolejności wykonywania pętli, są traktowane jako nie zmieniające istoty algorytmu. W tym ujęciu złożoność

obliczeniowa algorytmu pozostaje bez zmian, różnią się natomiast oszacowania czasu wykonania dla konkretnych implementacji (konkretnych programów).

Zapis implementacji algorytmów jest dokonywany w książce w języku C. Przyczyną jest z jednej strony popularność języka C, z drugiej jego podobieństwo w zapisie podstawowych konstrukcji programistycznych do innych języków programowania. Ważną cechą języka C jest także to, że wśród stosowanych obecnie języków, jest on często traktowany jako język relatywnie niskiego poziomu (bez zaawansowanych abstrakcyjnych konstrukcji), będący "blisko" sprzętu, a więc dobrze nadający się do analizy wydajności.

Programy zawarte w książce powinny pozwalać na badanie wydajności niezależnie od systemu operacyjnego i kompilatora. Na potrzeby tekstu wszystkie zostały przetestowane w środowisku systemu operacyjnego Linux, przy zastosowaniu popularnych kompilatorów *gcc* i *icc* (ten ostatni, udostępniany darmowo, po spełnieniu odpowiednich wymagań, lub płatnie przez firmę Intel, szczególnie dobrze nadaje się do badania wydajności wykonania na procesorach tej firmy). Do analizy wydajności omawianych programów wykorzystane zostały także wspomagające narzędzia, dostępne powszechnie w darmowych dystrybucjach Linuxa.

W klasycznej analizie złożoności obliczeniowej oszacowania wyrażane są najczęściej liczbą tzw. operacji dominujących. Operacje dominujące są określane jako te spośród realizowanych przy wykonaniu algorytmu, które będą najbardziej znaczące dla rzeczywistego czasu wykonania. Operacje dominujące mogą być różne dla różnych algorytmów w różnych dziedzinach zastosowań. W niniejszej książce obszar zainteresowań ograniczony jest do wybranej grupy algorytmów, w których operacjami dominującymi są podstawowe operacje wykonywane przez sprzęt komputerowy, takie jak operacje arytmetyczne, dostępy do pamięci, przesłania danych przez magistrale i sieci komputerowe.

Wynika to częściowo z wyboru obszaru informatyki, na którym głównie koncentruje się książka. Podstawowe opisywane algorytmy należą do numerycznej algebry liniowej i obejmują podstawowe operacje na wektorach (tablicach liczbowych) i macierzach. Operacje takie występują w wielu dziedzinach zastosowań, są między innymi podstawą bardziej rozbudowanych metod w obszarze nauk obliczeniowych (*computational science*) i obliczeń naukowo-technicznych (*scientific and technical computing*). Analiza i optymalizacja tego typu algorytmów jest związana od lat z dziedziną obliczeń wysokiej wydajności (*high performance computing*).

Zaprezentowane w książce sposoby analizy wydajności mają jednak charakter ogólny i mogą być wykorzystane w szerszym kontekście, dla algorytmów innych rodzajów, operujących na innych typach danych, wykorzystywanych w rozmaitych gałęziach informatyki. Takimi przykładowymi obszarami zastosowań, w których istotna jest wydajność przetwarzania i w których stosować można techniki omawiane w książce, są grafika komputerowa, analiza danych czy uczenie maszynowe.

Książka skupia się na interakcji sprzętu i oprogramowania, na analizie kodu źródłowego programów pod kątem wydajności wykonania oraz modelowaniu wydajności. Ze względu na ograniczenia objętości przedstawione są tylko podstawowe aspekty optymalizacji, interakcji z systemem operacyjnym, wykorzystania bibliotek (w tym standardowych bibliotek języków programowania). Omawiane programy są najczęściej krótkimi, prostymi procedurami, które mimo to pozwalają na pokazanie wielu aspektów złożonej problematyki wydajności obliczeń.

Organizacja książki, wymagania wstępne, czytelnicy

Książka jest połączeniem podręcznika i monografii. Jako podręcznik przedstawia całościowy obraz zagadnienia, począwszy od podstaw, definiując i odpowiednio ilustrując wykorzystywane pojęcia. Jako monografia zawiera elementy zaawansowane, związane z badaniami. Dla jasności i prostoty wywodu

książka często abstrahuje od mniej istotnych dla podstawowego tematu szczegółów, zbiera w całość, stosując pewne uproszczenia, informacje zawarte w wielu źródłach. Nie dokumentuje też wszystkich źródeł, wskazując tylko najważniejsze, najczęściej będące podręcznikami zawierającymi bardziej szczegółowe rozwinięcie zagadnień pokrewnych do omawianych w książce.

Zdecydowaną większość materiału zawartego w książce można odszukać w internecie (dotyczy to np. wielu zamieszczonych rysunków), w postaci wiedzy rozproszonej w rozmaitych artykułach i innych tekstach. Sensem powstania książki jest z jednej strony przedstawienie spójnego obrazu zagadnienia, traktującego temat szerzej niż pojedyncze, krótkie opracowania, a z drugiej przedstawienie go w języku polskim, co może mieć znaczenie dydaktyczne, ale także przyczyniać się do rozwijania polskiego słownictwa w dziedzinie informatyki.

Książka zakłada u Czytelnika pewien poziom wiedzy dotyczącej podstaw informatyki, architektur systemów komputerowych, systemów operacyjnych, programowania w klasycznych językach proceduralnych i obiektowych oraz obliczeń równoległych. Zakres omawiany na kursach studiów informatycznych pierwszego stopnia powinien być w zupełności wystarczający. Osoby nie studiujące informatyki, a mające pewne doświadczenie programistyczne, w tym doświadczenie w programowaniu równoległym, także nie powinny mieć kłopotu z korzystaniem z książki.

Integralną częścią książki jest zestaw przykładowych zadań i problemów, omawianych i rozwiązywanych w tekście, zebrany w formie praktycznych ćwiczeń. W obecnej postaci wybrane ćwiczenia dostępne są także na stronie `http://www.metal.agh.edu.pl/~banas/WO/WO.html` związanej z kursem "Wydajność oprogramowania" prowadzonym przez autora. Poza omówieniem problemów oraz wykorzystywanych przy ich rozwiązywaniu sposobów analizy oraz optymalizacji wydajności, tematy ćwiczeń zawierają także szereg, pominiętych w książce, dokładnych instrukcji uruchamiania programów czy obsługi stosowanych aplikacji i narzędzi.

Książka jest na bieżąco redagowana i aktualizowana, wszelkie uwagi na jej temat można kierować na adres internetowy pobanas@cyf-kr.edu.pl .

Rozdział 1

Model wykonania i wydajność obliczeń na pojedynczym rdzeniu mikroprocesora

1.1 Budowa systemów i mikrobenchmarki

Obiektem badań w książce są algorytmy i programy wykonywane na współczesnych systemach komputerowych, złożonych z mikroprocesorów, układów pamięci, rozmaitych sieci połączeniowych. Ze względu na ogromny stopień złożoności systemów, spojrzenie na ich architekturę jest w książce z konieczności uproszczone i skupiające się na wybranych elementach, których wpływ na wydajność obliczeń daje się prześledzić na poziomie analizy kodu źródłowego programów oraz sterowania parametrami wykonania za pomocą odpowiednich narzędzi.

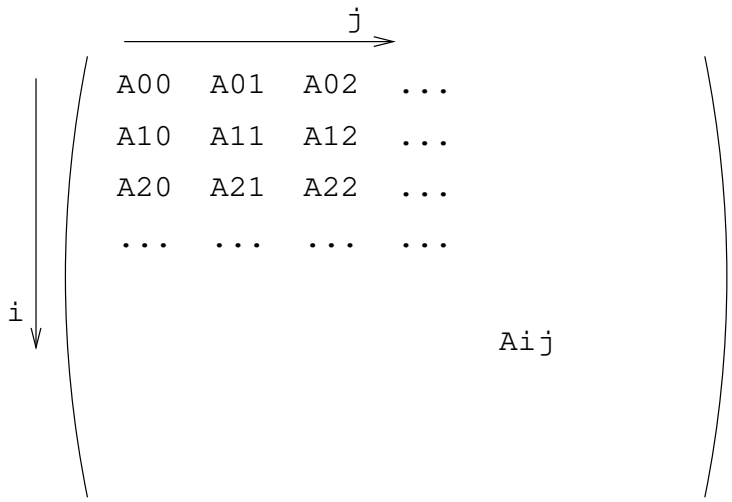
Jedną z konsekwencji złożoności systemów obliczeniowych, mającą wpływ na dziedzinę badania wydajności, jest praktyczny brak możliwości, dla zdecydowanej większości programów, uzyskania ilościowych charakterystyk wydajności i oszacowań czasu wykonania, wyłącznie na podstawie teoretycznej analizy sprzętu. Liczba czynników mających wpływ na wydajność oraz ich różnorodne interakcje, zależne często od indywidualnej specyfiki programów, utrudniają lub uniemożliwiają uzyskanie ogólnych wzorów matematycznych.

Z tego względu, często konieczne staje się wsparcie teoretycznych analiz technikami eksperymentalnymi. Techniki te mogą pozwolić na wskazanie fragmentów kodu źródłowego oraz wyróżnienie elementów sprzętowych mających decydujący wpływ na czas wykonania programu. Mogą także służyć określeniu wydajności całych systemów lub ich konkretnych komponentów dla wybranych algorytmów i ich specyficznych fragmentów.

Powszechnie stosowanymi narzędziami w technikach eksperymentalnych są benchmarki komputerowe, specjalnie zaprojektowane programy lub ściśle określone zadania i warunki wykonania dla istniejących aplikacji, których realizacja pozwala, po odpowiednich pomiarach, na uzyskanie odpowiednio zdefiniowanych miar wydajności.

Benchmarki komputerowe są stosowane do określania wydajności dla grup programów użytkowych o zbliżonej charakterystyce, wybranych elementów składowych systemów informatycznych, powszechnie stosowanych funkcji bibliotecznych, czy zestawów operacji na danych. Ważnym w rozważaniach o architekturze sprzętu rodzajem benchmarków są mikrobenchmarki, służące do badania wydajności konkretnych komponentów systemów komputerowych. Zgodnie ze swą nazwą mikrobenchmarki zawierają krótkie, często kilkulinijkowe, fragmenty kodu poddawanego badaniu.

Poza testowaniem wydajności dobrze rozpoznanych elementów, mikrobenchmarki mogą pełnić także inną, specyficzną rolę. Często budowa i szczegóły funkcjonowania konkretnego układu nie są publicznie udostępniane przez producentów lub też wzajemne interakcje kilku układów nie są łatwe do przewidze-



Rysunek 1.1: Macierz i oznaczenia jej elementów

nia dla konkretnego kodu. W takich przypadkach mikrobenchmarki można wykorzystać do wysnucia wniosków, lub postawienia hipotez, dotyczących budowy i funkcjonowania komponentów systemu oraz ich interakcji w trakcie wykonywania programów.

1.1.1 Kody mikrobenchmarków - pętle i dostępy do tablic

Kody mikrobenchmarków składają się najczęściej z prostych pętli, pojedynczych i podwójnych, zazwyczaj wykonywanych wielokrotnie. W pętlach tych realizowane są operacje arytmetyczne, z których dla celów badań wydajności przeprowadzanych w książce najważniejszymi będą operacje zmiennoprzecinkowe, a także dokonywane są dostępy do danych w pamięci. Dane zazwyczaj przechowywane są w postaci tablic, a czas pojedynczego dostępu w trakcie wykonania, co będzie dokładnie analizowane w dalszej części książki, zależy od wzorca dostępu do pamięci w pętli. Wzorzec może zakładać dostęp w kolejnych iteracjach pętli do kolejnych elementów tablicy, do losowych elementów tablicy, a także do elementów tablicy oddalonych w pamięci o stały odstęp, który można mierzyć liczbą bajtów, ewentualnie liczbą elementów tablicy.

Dla prostej pętli

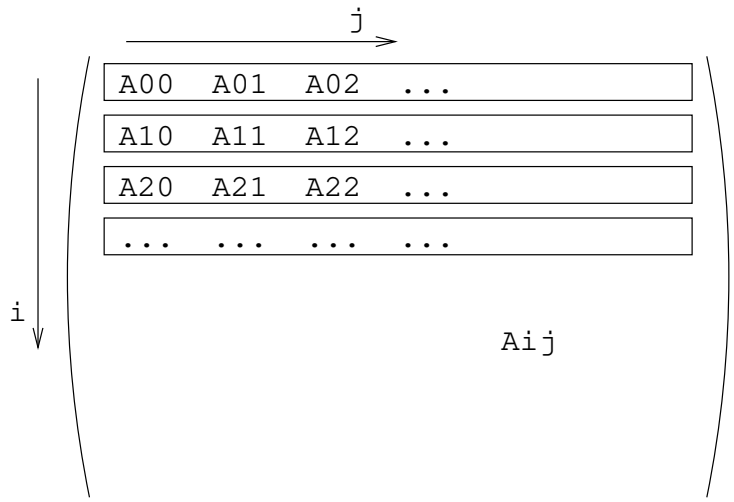
```
for (j=0; j<rozmiar_tab; j+=skok) tab[j]++;
```

dostępy do tablicy danych `tab` odbywają się kolejno do lokalizacji oddalonych o liczbę elementów równą wartości zmiennej `skok` (dla wyrażenia odstępu w bajtach, `skok` należy pomnożyć przez rozmiar pojedynczego elementu tablicy, np. 8 bajtów dla liczb podwójnej precyzji).

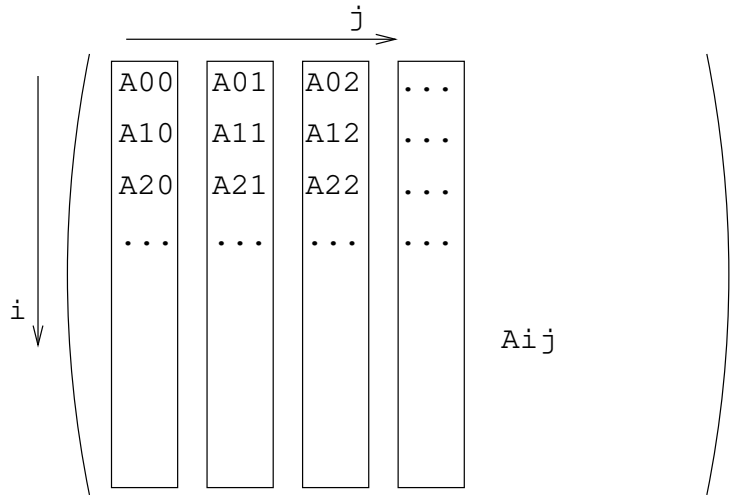
Specjalną rolę w benchmarkach, zwłaszcza z numerycznej algebry liniowej, odgrywają dostępy do tablic, które przechowują dane związane w algorytmach z macierzami. Macierze są zbiorami elementów, które dla przykładowej macierzy $\boldsymbol{A}$ zapisywane będą jako A_{ij}, gdzie i jest indeksem wiersza, a j indeksem kolumny macierzy (rys. 1.1). W przedstawionej na rysunku konwencji, stosowanej w całej książce, indeksowanie tak wierszy, jak i kolumn, rozpoczyna się, zgodnie z tradycją C, od 0.

Najprostszym sposobem przechowywania danych macierzowych jest wykorzystanie tablic dwuwymiarowych jako struktur danych w językach programowania. Przy standardowej alokacji tablicy dwuwymiarowej w C

```
double A[M][N];
```



Rysunek 1.2: Ilustracja przechowywania macierzy wierszami



Rysunek 1.3: Ilustracja przechowywania macierzy kolumnami

przechowywanie elementów odbywa się wierszami, co oznacza, że w kolejnych komórkach pamięci znajdują się kolejne wyrazy wiersza, odpowiadające kolejnym kolumnom, np. dla *i*-tego wiersza jego pierwsze wyrazy umieszczone są w pamięci w kolejności: $A_{i0}, A_{i1}, A_{i2}, ...$ Po umieszczeniu wiersza w pamięci, bezpośrednio po nim znajdą się wyrazy następnego wiersza (rys. 1.2). Oznacza to także, że kolejne wyrazy w dowolnej kolumnie, np. $A_{0j}, A_{1j}, A_{2j}, ...$, oddzielone będą w pamięci liczbą wyrazów równą długości wiersza *N* (będącej jednocześnie liczbą kolumn).

Przechowywanie macierzy kolumnami, jak na rys. 1.3, jest typowe dla Fortranu i ze względu na jego popularność w obliczeniach technicznych, wciąż często stosowane w rozmaitych bibliotekach.

Przechowywanie macierzy w tablicach jednowymiarowych

Dla większości implementacji w książce do przechowywania elementów macierzy stosowane będą tablice jednowymiarowe. Daje to większą elastyczność w programowaniu, wprowadzając niewielkie tylko komplikacje. Przechowanie macierzy w tablicy jednowymiarowej polega na zdefiniowaniu po-

jedynczej tablicy o rozmiarze pozwalającym na przechowanie wszystkich wyrazów:

```
double a[M*N];
```

W technice tej możliwy jest wybór sposobu przechowywania macierzy: wierszami (*row major*) lub kolumnami (*column major*). Sposób przechowywania wierszami, odpowiadający standardowej praktyce języka C, oznacza, że wyraz A_{ij} tablicy $\boldsymbol{A}$ znajduje się w elemencie o indeksie `a[i*N+j]` (*i*-ty wiersz, *j*-ty element w wierszu $=$ *j*-ta kolumna). Przechowywanie kolumnami będzie polegało na przypisaniu wyrazowi A_{ij} miejsca w pamięci `a[i+j*N]`.

Przeglądanie macierzy zapisanych w postaci tablic jednowymiarowych nadal najczęściej realizowane jest w podwójnej pętli, po wierszach i kolumnach. Dla tablic przechowywanych wierszami, przeglądanie z pętlą po wierszach jako zewnętrzną:

```
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
    ... a[i*N+j] ... // A[i][j]
  }
}
```

polega na odwiedzaniu kolejnych komórek w pamięci. Przeglądanie z pętlą po kolumnach jako zewnętrzną:

```
for(j=0;j<N;j++){
  for(i=0;i<M;i++){
    ... a[i*N+j] ... // A[i][j]
  }
}
```

oznacza odwiedzanie w kolejnych iteracjach wyrazów oddalonych o *N* elementów, a więc dostępy do pamięci ze skokiem *8*N* bajtów (dla zmiennych podwójnej precyzji).

Array padding czyli rozciąganie tablic

Dostęp do komórek pamięci w kolejnych iteracjach pętli ze skokiem o pewną liczbę bajtów może powodować opóźnienie działania programu, szczególnie widoczne dla specyficznych rozmiarów skoku. Może zdarzyć się tak, że skok przy dostępie do tablicy w algorytmie operującym na macierzach zależy od rozmiaru macierzy (np. w opisanym powyżej dostępie do kolejnych elementów macierzy w pojedynczej kolumnie, w przypadku macierzy przechowywanej wierszami).

Popularną techniką zapobiegania ewentualnym przyrostom czasu wykonania dla specyficznych rozmiarów macierzy jest technika *array padding*, co będziemy określać jako rozpychanie lub rozciąganie tablic. Polega ona na zaalokowaniu tablic większych niż wymagane w algorytmie, co pozwala między innymi na uniknięcie niekorzystnego wzorca dostępów do pamięci (*array padding* można stosować także w innych optymalizacjach, niekoniecznie związanych z korzystaniem z pamięci)). Dla tablic dwuwymiarowych zamiast alokować

```
double A[M][N];
```

definiuje się tablicę o wydłużonym wierszu:

```
double A[M][N+O];
```

gdzie parametr `O` dobierany jest odpowiednio do wymagań optymalizacji. Algorytmy modyfikuje się tak, żeby niezależnie od rozmiaru zaalokowanej pamięci dotyczyły tylko wyrazów odpowiadających oryginalnym tablicom, ewentualnie wypełnia się powiększone tablice, tak aby operując na dodanych wyrazach nie powodować zmiany wyników algorytmu.

W przypadku stosowania tablic jednowymiarowych, alokacja w technice *array padding* będzie dotyczyć `M*(N+O)` wyrazów, a dostęp do wyrazu A_{ij} macierzy przechowywanej wierszami w tablicy `a` (w *i*-tym wierszu i *j*-tej kolumnie), uzyskuje się za pomocą notacji `a[i*(N+O)+j]`.

1.1.2 Narzędzia wspomagające analizę wydajności

W książce wykorzystywany jest szereg programów narzędziowych typowych dla dziedziny badania wydajności. Podstawowymi narzędziami są tzw. *profilery*, służące do uzyskiwania czasów wykonania dla całych programów oraz ich poszczególnych fragmentów. Często pozwalają one także na uzyskanie dodatkowych informacji o zdarzeniach związanych z wykonaniem programu, mających istotny wpływ na wydajność (np. liczby dostępów do różnych rodzajów pamięci).

Profilery mogą działać w oparciu o kilka różnych zasad. Jedną z nich jest tzw. *instrumentacja* kodu, polegająca na dodaniu do programu, najczęściej w trakcie kompilacji, fragmentów przekazujących informację o zdarzeniach (*events*) związanych z wykonaniem kodu (np. wywołanie funkcji, przejście w tryb jądra systemu operacyjnego, itp.).

Inną z zasad działania profilerów jest statystyczne próbkowanie (*statistical sampling*). W trakcie wykonania programu (z kodem wykonywalnym poddanym wcześniejszej instrumentacji lub w przypadku realizacji programu pod kontrolą odpowiedniego środowiska nadzorującego), w określonych odstępach czasu, przetwarzanie jest przerywane i odczytywane są dane mające znaczenie dla analizy wydajności. Pobierane mogą być dane różnego rodzaju, np. stan stosu wywołań (*execution stack, call stack*) lub zawartość specjalnych rejestrów procesora (tzw. liczników sprzętowych, *hardware counters*), zliczających wskazane zdarzenia dotyczące użycia sprzętu (*hardware events*), takie jak np. dostępy do pamięci, takty zegara, wykonane rozkazy (bardziej szczegółowy opis wykorzystania liczników sprzętowych znajduje się w p. 1.2.7). Ze względu na statystyczną naturę zbierania danych, gdzie wnioskuje się o całości wykonania programu na podstawie próbkowania w wybranych chwilach czasu, wyniki zwracane przez tego typu profilery bywają obarczone błędami. Raportowana liczba zdarzeń może być różna od rzeczywistej (zazwyczaj w granicach kilku, kilkunastu procent), pewne rzadziej występujące zdarzenia mogą zostać pominięte.

Nadzorowanie wykonania programu może przybierać różne formy. Od prostego zbierania danych dotyczących zdarzeń lub uzyskanych z próbkowania, aż do pełnienia funkcji maszyny wirtualnej, która staje się środowiskiem wykonania programu. W takim przypadku, program nadzorujący, wykonujący pojedyncze instrukcje kodu, uzyskuje możliwość precyzyjnego badania wszystkich zdarzeń podczas wykonania. Wadą takiego podejścia jest występujący najczęściej duży narzut czasowy związany z realizacją obliczeń w ramach takiej maszyny wirtualnej. Narzut ten może być relatywnie niewielki w przypadku kiedy program (np. w postaci odpowiedniego kodu pośredniego, często poddawanego uprzednio instrumentacji) standardowo wykonywany jest przy pomocy maszyny wirtualnej, np. dla maszyn wirtualnych Javy lub .NET.

Informacje zbierane w trakcie wykonania mogą być na bieżąco udostępniane przez program nadzorujący, np. w postaci graficznej, lub zapisywane w odpowiednich plikach. Dane z plików mogą być odczytywane, interpretowane i wizualizowane przez dowolne programy rozpoznające format zapisu.

Istnieją dwie podstawowe formy prezentacji danych związanych z wykonaniem kodu, w szczególności dotyczących wydajności, jedną jest tzw. profil wykonania (*execution profile*), a drugą ślad wykonania (*execution trace*). Profil wykonania to zbiorcze zestawienie wybranych danych zebranych w trakcie wykonania, w szczególności czasów realizacji poszczególnych fragmentów kodu: funkcji, bloków kodu,

pojedynczych instrukcji. Dane mogą być wzbogacone o informacje związane z grafem wywołań (np. ile razy dana funkcja wywoływana była przez inną wybraną funkcję).

Ślad wykonania to zapis zdarzeń w kolejności chronologicznej (rozmiar pliku śladu, w przeciwieństwie do rozmiaru pliku profilu, rośnie proporcjonalnie do czasu działania programu). W badaniach wydajności, ślady wykonania są szczególnie popularne przy wykonaniu w środowiskach przesyłania komunikatów, gdzie pozwalają na optymalizację sposobu i czasu komunikacji między procesami.

W książce stosowane są narzędzia zbierania danych o wykonaniu powszechnie dostępne dla darmowych dystrybucji Linuxa. Najpopularniejszym z narzędzi jest program *gprof*. Wykorzystuje on hybrydowy mechanizm, łącząc próbkowanie statystyczne z instrumentacją kodu (*gprof* wymaga kompilacji ze specjalnymi opcjami, standardowo *-p* lub *-pg*). *gprof* tworzy tzw. płaski profil (*flat profile*) oraz profil z drzewem wywołań (*call graph*). Płaski profil zawiera informacje o czasie wykonania poszczególnych funkcji kodu (bezwzględnym oraz procentowym w stosunku do czasu wykonania całego programu), w tym także ile czasu zajmowało wykonanie samej funkcji, bez innych wywoływanych przez nią procedur, a ile łącznie z nimi. Profil z drzewem wywołań rozróżnia dla każdej funkcji czasy jej wykonania zależnie od procedury wywołującej tę funkcję. Użycie narzędzia *gprof* jest często pierwszym krokiem optymalizacji kodu – służy do wykrycia tych procedur, w których program spędza najwięcej czasu, a więc optymalizacja których może przynieść największe zyski czasowe.

Inne z wykorzystywanych, bardziej specjalistycznych, narzędzi analizy wydajności programów omówione są w późniejszych rozdziałach książki, w miejscach ich bezpośredniego zastosowania.

Wydajność efektywna

Skutkiem ubocznym zastosowania pewnych technik optymalizacji, w tym także np. *array padding*, może być sytuacja, w której w implementacji pewnego algorytmu wykonuje się dodatkowe operacje lub dostępy do pamięci, nie występujące w algorytmie i nie wpływające na jego efekty (w przypadku rozciągania tablic może to dotyczyć dostępów do i operacji na dodanych elementach wiersza o indeksach poza zakresem jego oryginalnej długości).

Jeśli wydajność programu wyrażana będzie w liczbie operacji na sekundę lub liczbie dostępów na sekundę (co może być także tłumaczone na szybkość transferu danych z pamięci), w konwencji przyjętej w książce liczyć się będą tylko te, które występują w oryginalnym algorytmie, tzn. takie które wykonują użyteczną pracę, ze względu na wynik obliczeń.

Taki sposób liczenia wydajności przyjmowany będzie we wszystkich badanych w książce programach. Ważna będzie użyteczna praca wykonana na potrzeby aplikacji, niezależnie od tego ile i jakie operacji zrealizuje sprzęt (wyjątkiem będą tylko pewne mikrobenchmarki, ukierunkowane na badanie funkcjonowania sprzętu).

Jest to naturalne podejście, odpowiadające nie tylko zastosowanym technikom optymalizacji (które mogą wprowadzać nieużyteczne, dodatkowe operacje, w celu skrócenia czasu wykonania programu), ale wynikające także np. z analizy pracy procesorów (rdzeni), które wykonują wiele operacji nie odpowiadających instrukcjom kodu (i rozkazom asemblera), np. przy stosowaniu omawianych w dalszych częściach książki technikach, takich jak pobieranie z wyprzedzeniem, przewidywanie skoków i inne rodzaje wykonania spekulatywnego. Z przyjęcia założenia, że wydajność dotyczy tylko operacji efektywnie wykonanych na potrzeby aplikacji, wynika także względna przydatność zliczania zdarzeń sprzętowych – istotne jest nie tyle ile operacji wykonał procesor (rdzeń), ale ile z tych operacji przełożyło się na efektywną pracę programu.

1.2 Mikroarchitektury – architektury rdzeni mikroprocesorów

Zawarty w niniejszym punkcie opis dotyczy podstawowych cech architektury pojedynczego rdzenia obliczeniowego, odpowiadającego klasycznej jednostce przetwarzania pojedynczego wątku obliczeń. Jednostka taka była określana tradycyjnie jako procesor (*CPU, central processing unit*), stąd też nazwy rdzeń i procesor często w książce używane są zamiennie (czasem stosowana jest także nazwa procesor logiczny). Współczesne układy scalone realizujące obliczenia, mające postać mikroprocesorów wielordzeniowych, będą czasami także skrótowo nazywane procesorami, w sytuacjach kiedy kontekst jawnie wskazuje, które znaczenie określenia procesor jest właściwe.

Poniższa prezentacja architektury pojedynczego rdzenia mikroprocesora, podobnie jak opisy innych elementów sprzętowych, jest gdzieniegdzie dokonywana na poziomie elementarnym. Ma to między innymi za zadanie wprowadzenie szeregu podstawowych pojęć stosowanych w dalszej części książki przy analizie i optymalizacji wydajności wykonania programów.

1.2.1 Architektura von Neumanna

Podstawowym modelem mikroarchitektury pojedynczego rdzenia, będącym punktem wyjścia analiz i realizowanym (z ewentualnymi modyfikacjami) we wszystkich współczesnych systemach komputerowych, jest model architektury von Neumanna. Sposób modyfikacji i rozbudowy jego podstawowych elementów definiuje szereg istotnych różnic między współczesnymi architekturami.

Maszyna von Neumanna, jako podstawowy model obliczeniowy, składa się z jednostki centralnej (CPU, *Central Processing Unit*), połączonej za pomocą odpowiednich kanałów komunikacyjnych z układem adresowalnej binarnej pamięci głównej (*main memory*). Procesor realizuje program zapisany w pamięci głównej, korzystając z danych wejściowych, przechowywanych w tej samej pamięci, i zapisując wyniki obliczeń, także w pamięci głównej.

Adresowalność pamięci oznacza, że można podzielić pamięć główną na podstawowe elementy, zawierające bity informacji, nazywane dalej komórkami pamięci (*memory cell, memory location*). Każda z takich komórek posiada adres, dzięki któremu możliwe jest pobieranie i zapisywanie danych w dowolnej komórce. Rozmiar (w bitach) pojedynczej (czyli posiadającej jeden adres) komórki pamięci może być różny, we wszystkich przykładach wykorzystywanych w książce wynosi jeden bajt (będący standardową jednostką dla najważniejszych współczesnych mikroprocesorów).

W pamięci przechowywane są dane o różnym typie, takim jak znak, liczba całkowita, liczba zmiennoprzecinkowa, z których każdy może być reprezentowany za pomocą różnej liczby bitów. Stąd, w niniejszej książce, pojęcia adresu i komórki pamięci są często używane w znaczeniu mniej ścisłym, a bardziej ogólnym: często stosowane będą określenia: ”komórka przechowująca liczbę” (choć w rzeczywistości liczba może być przechowywana w kilku komórkach), ”adres zmiennej” (co oznaczać będzie adres pierwszej komórki, w której znajdują się bity danej liczby).

Program w pamięci głównej składa się z ciągu przechowywanych rozkazów (*instructions*). Wykonanie programu przez procesor polega na pobieraniu kolejnych wykonywanych rozkazów (kolejny wykonywany niekoniecznie oznacza kolejny przechowywany w pamięci, np. przy realizacji rozkazu skoku), a następnie ich przetwarzaniu. Perspektywa przyjęta w niniejszej książce, związana z analizą wydajności wykonania, preferuje zawsze spojrzenie na wykonywane rozkazy, a nie rozkazy zapisane w kodzie binarnym. Wydajność staje się istotna w przypadku wykonywania wielkiej liczby rozkazów (rzędu co najmniej miliardów), co zawsze przekracza liczbę rozkazów w kodzie binarnym. W skrajnych przypadkach, przy użyciu pętli o bardzo dużej liczbie iteracji, analizowany kod binarny może zawierać tylko kilka lub kilkanaście rozkazów. W praktyce oznacza to, że często przy analizie wydajności obiektem zainteresowania sa tylko wybrane, często krótkie fragmenty kodu, które jednak prowadzą do dużej liczby realizowanych rozkazów i znacznego czasu wykonania (tzw. punkty zapalne, *hotspots*).

Podstawowymi rozkazami, uwzględnianymi w niniejszej książce, wykonywanymi przez procesory są:

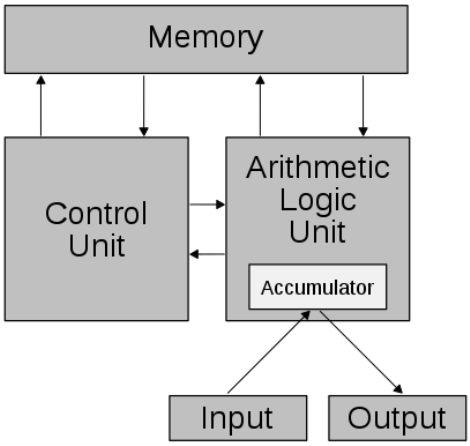
- pobieranie z pamięci i zapis do pamięci
- operacje arytmetyczne
- operacje logiczne
- transfer sterowania, skoki
- operacje wejścia/wyjścia

Rozkazy, jak widać z powyższego zestawienia, zawsze oznaczają wykonanie pewnej operacji, która może posiadać od zera do kilku argumentów (zazwyczaj maksymalnie trzech). Argumentami rozkazów mogą być liczby, zawartość rejestrów, traktowanych jako komórki wewnętrznej pamięci procesora o określonym rozmiarze i indywidualnych nazwach, oraz zawartość komórek pamięci, dostępna dzięki adresowi przechowywanemu w rejestrach.

W książce dla ilustracji zagadnień wydajności obliczeń, pojawiać się będą przykłady kodu, realizowanego przez procesory, zapisane w wersji języka asemblera typowej dla systemów operacyjnych z rodziny Unix. Oznaczeniami typowych, przykładowych rozkazów (występujących np. w procesorach rodziny x86) są:

- `mov`: przesunięcie danych pomiędzy rejestrami i komórkami pamięci (bez operacji pamięć-pamięć)
- `push, pop`: operacje na stosie, np. związane z obsługą wywołania procedur
- `ld, st`: dostępy do pamięci – pobranie i zapis
- `add, sub, mul`: dodawanie, odejmowanie, mnożenie argumentów
- `inc, dec`: zwiększenie, zmniejszenie o 1
- `neg`: zmiana znaku
- `lea`: obliczenie adresu (bez transferu danych)
- `xor, and, or`: działania logiczne (na bitach argumentów)
- `cmp`: obliczenie wartości logicznej porównania dwóch argumentów i zapisanie wyniku w odpowiednich rejestrach procesora (rejestrach stanu)
- `jmp`: skok bezwarunkowy
- `jge, je, jl`: skoki warunkowe – przeniesienie sterowania do określonego miejsca kodu, zależnie od wyniku poprzedzajacej operacji porównania, zapisanego w rejestrze stanu
- `call, ret`: obsługa wywołań procedur

W ramach przyjętej notacji, zawartości rejestrów (najczęściej stosowane będą rejestry 32 i 64-bitowe oraz wektorowe) zapisywane są ze znakiem `%` (np. `%eax` jako zawartość 32-bitowego rejestru `eax`), a zawartości komórek pamięci głównej z użyciem nawiasów (np. `(%eax)` jako zawartość komórek pamięci o adresie początkowym zapisanym w rejestrze `eax`). Liczba komórek pamięci, których dotyczy rozkaz, jest zależna od konkretnego rozkazu, czasem nazwa rozkazu jest modyfikowana w celu określenia rozmiaru argumentu.



Rysunek 1.4: Architektura von Neumanna [źródło: Wikipedia]

1.2.2 Przetwarzanie rozkazów

Przetwarzanie pojedynczego rozkazu składa się z szeregu etapów, o liczbie i charakterze zależnym od konkretnych rozwiązań technicznych procesora, wśród których zawsze można jednak wyróżnić podstawowe fazy:

- pobrania rozkazu z pamięci do procesora
- dekodowania rozkazu (w uproszczonym ujęciu można to rozumieć jako przekształcenie zapisanego w pamięci rozkazu na sekwencję wewnętrznych operacji procesora, czasem dekodowanie rozkazu z listy rozkazów procesora związane jest z jego zamianą na sekwencję realizowanych mikro-rozkazów)
- wykonania rozkazu

Wykonanie rozkazu może być związane z pobraniem danych wejściowych rozkazu z rejestrów lub pamięci głównej, zapisem danych wyjściowych lub zmianą wewnętrznego stanu procesora.

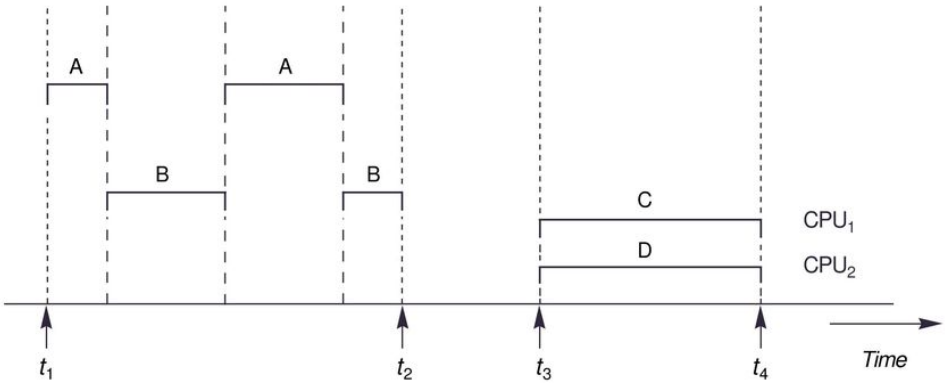
W klasycznej maszynie von Neumanna całość układu przetwarzania można schematycznie przedstawić jak na rys. 1.4. Poza pamięcią (*Memory*) i procesorem, składającym się z jednostki sterującej (*Control Unit)* oraz jednostki wykonania rozkazów (*Arithmetic-Logic Unit*) z wyróżnionym pojedynczym rejestrem (*Accumulator*), schemat obejmuje także urządzenia wejścia/wyjścia (*Input/Output*).

1.2.3 Jednostki wykonania rozkazów

Rozszerzenia i modyfikacje pierwotnej architektury von Neumanna obecne we współczesnych mikroprocesorach obejmują bardzo szeroki zakres, tak jeśli chodzi o liczbę elementów składowych procesora, jak i o stopień ich złożoności. Pierwszym z analizowanych elementów jest układ wykonywania rozkazów.

Przetwarzanie potokowe

Wykonanie rozkazów przez współczesne procesory odbywa się zawsze z wykorzystaniem potoków przetwarzania rozkazów (*instruction pipeline*). Przetwarzanie pojedynczego rozkazu jest rozbijane na etapy, każdy z etapów wykonywany jest przez odrębne układy, dzięki czemu procesor współbieżnie



Rysunek 1.5: Wykonanie współbieżne: w przeplocie (procesy/wątki A i B) oraz równoległe (procesy/wątki C i D) [źródło: Wikipedia]

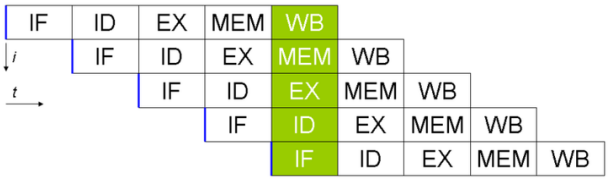
Rysunek 1.6: Schemat przetwarzania bez wykorzystania współbieżności [źródło: Wikipedia]

przetwarza kilka rozkazów. W dalszej części książki współbieżność zawsze będzie oznaczała realizację wielu działań (w tym przypadku przetwarzanie wielu rozkazów, dalej także wykonanie wielu wątków lub procesów), w taki sposób, że rozpoczęcie realizacji kolejnego działania rozpoczyna się przed zakończeniem poprzedniego. Współbieżność może mieć wiele postaci, począwszy od realizacji wielu działań w przeplocie (kiedy w konkretnej chwili realizowane jest tylko jedno działanie), poprzez realizacje potokową, kiedy równocześnie może być realizowane kilka działań, ale każde znajduje się w innej fazie, aż po pełną równoległość, kiedy wiele jednocześnie realizowanych działań może znajdować się w tej samej fazie (rys. 1.5).

Przyjmując podział przetwarzania rozkazu przez procesor na przykładowe etapy (przy czym w rzeczywistości, różne rozkazy mogą mieć różne etapy – czasem jest ich więcej, czasem mniej niż w przedstawionym przykładzie): IF – pobranie rozkazu (*instruction fetch*), ID – dekodowanie rozkazu (*instruction decode*), EX – wykonanie operacji składających się na rozkaz (*instruction execute*), MEM – dostęp do pamięci (*memory access*) oraz WB – zapis efektu realizacji rozkazu (*write-back*), schemat klasycznego przetwarzania sekwencyjnego rozkazów można zilustrować jak na rys. 1.6.

Wykorzystanie podziału procesora na pracujące równolegle podukłady związane z realizacją poszczególnych faz przetwarzania rozkazu prowadzi do przetwarzania potokowego pokazanego na rys. 1.7. Widoczne jest, że procesor w jednej chwili czasu (biegnącego wzdłuż osi poziomej) przetwarza kilka rozkazów, z których każdy znajduje się w innej fazie (na rysunku pojedynczy rozkaz złożony z przykładowych faz jest pojedynczym poziomym blokiem, a pionowy zielony blok odpowiada przykładowej chwili - pojedynczemu taktowi procesora).

Porównując z przetwarzaniem nie wykorzystującym współbieżności (rys. 1.6), widać zyski czasowe związane z przetwarzaniem potokowym. Jeśli przyjmiemy, że każdy etap przetwarzania potokowego zajmuje ten sam odcinek czasu t_e, to przetwarzanie rozkazu o k etapach zajmuje $k \cdot t_e$ czasu. Dla sekwencji n rozkazów daje to w przypadku nie wykorzystywania współbieżności czas wykonania $n \cdot k \cdot t_e$. W przypadku przetwarzania potokowego, czas wykonania pierwszego rozkazu to $k \cdot t_e$, podczas gdy ukończenie pozostałych $n-1$ rozkazów zajmuje dodatkowo $(n-1) \cdot t_e$, co ostatecznie daje czas wykonania



Rysunek 1.7: Schemat klasycznego przetwarzania potokowego [źródło: Wikipedia]

$(n + k - 1) \cdot t_e$. Dla odpowiednio długiej sekwencji rozkazów, $n \gg k$, skrócenie czasu wykonania związane z zastosowaniem potokowości, $\frac{n \cdot k \cdot t_e}{(n+k-1) \cdot t_e}$, zmierza więc do k.

W praktyce czas wykonywania rozkazów wyraża się często w taktach (cyklach) zegara procesora (*clock cycle*). Często założeniem jest takie zaprojektowanie procesora, aby wykonanie jednego etapu przetwarzania zajmowało jeden takt zegara. Dzięki temu w przetwarzaniu potokowym możliwe staje się uzyskanie sytuacji, w której po każdym takcie zegara kończone jest przetwarzanie jednego rozkazu procesora (tak jak przedstawia to rys. 1.7).

Przetwarzanie potokowe ma jeszcze inny aspekt istotnie wpływający na wydajność. Staranne zaprojektowanie potoków przetwarzania (z kilkunastoma lub kilkudziesięcioma etapami) umożliwia znaczne zwiększenie częstotliwości pracy procesorów. W taki sposób wprowadzane w latach 80-tych XX wieku procesory z rodziny RISC (*Reduced Instruction Set Computers*), procesory o zredukowanej liście uproszczonych, ale dobrze dostosowanych do przetwarzania potokowego, rozkazów) wypierały, dzięki wyższej częstotliwości i wydajności pracy, dawniejsze architektury, nazwane później procesorami CISC (*Complex Instruction Set Computers*).

Rozróżnienie procesorów na rodziny CISC i RISC straciło współcześnie na znaczeniu – mikroprocesory posiadające rozkazy typowe dla CISC na swojej liście rozkazów (np. mikroprocesory z rodziny *x86*), zazwyczaj wewnętrznie transformują je na sekwencję (mikro-)rozkazów typowych dla RISC.

Miary wydajności - opóźnienie i przepustowość przetwarzania potokowego

Z przetwarzaniem rozkazów przez procesor związana jest jedna z klasycznych miar wydajności – liczba taktów na rozkaz (*CPI, cycles per instruction*). Ideą wprowadzenia tej miary jest chęć szacowania czasu wykonania programu (liczonego w taktach procesora), jako iloczynu liczby wykonanych rozkazów i miary CPI.

W pierwotnych ujęciach, bez uwzględnienia przetwarzania potokowego, miarę CPI można było wiązać z liczbą taktów wymaganych do realizacji pojedynczego, izolowanego rozkazu. Tak definiowany czas wykonania związany jest z pojęciem opóźnienia (zwłoki, *latency*). Ogólnie, opóźnienie związane z wykonaniem konkretnej operacji można definiować jako czas od rozpoczęcia realizacji operacji do jej zakończenia (w dalszej części książki pojecie opóźnienia stosowane będzie do różnych komponentów systemów komputerowych i różnych realizowanych operacji).

W przypadku przetwarzania zilustrowanego na rys. 1.6, można przyjąć, że opóźnienie każdego z rozkazów wynosi k taktów zegara. Zastosowanie miary CPI powiązanej z opóźnieniem prowadziłoby do jej wartości równej k i poprawnego czasu wykonania n rozkazów: $n \cdot k$. Jednak przyjęcie CPI równego k prowadzi do błędnego oszacowania czasu wykonania dla przypadku przetwarzania potokowego z rys. 1.7. Zamiast poprawnego czasu $n + k - 1$ uzyskuje się znacznie wyższą wartość $n \cdot k$, zaniżającą wydajność przetwarzania.

Ze względu na fakt, że współczesne procesory odrębnie dokonują pobierania i dekodowania rozkazów, a odrębnie ich wykonania, opóźnienie przy przetwarzaniu rozkazów określa się zazwyczaj tylko w odniesieniu do faz wykonania (np. EX, MEM, WB). Wynosi ono zazwyczaj kilka taktów, choć dla złożonych rozkazów może wynosić kilkanaście lub nawet kilkadziesiąt (dane na temat opóźnienia dla

konkretnych rozkazów można znaleźć w podręcznikach programowania i optymalizacji dla konkretnych procesorów). Nadal jednak powiązanie miary CPI z tak definiowanym opóźnieniem nie nadaje się do szacowania czasu wykonania w przypadku przetwarzania potokowego, wciąż znacząco zaniżając wydajność.

Poprawne oszacowanie można uzyskać analizując przypadek z rys. 1.7. Widać, że dla odpowiednio dużej liczby rozkazów istotna dla czasu wykonania jest liczba taktów, jaka mija pomiędzy chwilami zakończenia kolejnych rozkazów. Gdyby za miarę CPI przyjąć tę właśnie liczbę (w przypadku rys. 1.7 równą 1), czas wykonania w taktach rzeczywiście byłby równy iloczynowi CPI i liczby rozkazów n (czyli n, dla odpowiednio długiej sekwencji rozkazów, $n \gg k$, dobrze przybliżający wartość dokładną $n + k - 1$, z dokładnością do kilku $(k - 1)$ początkowych taktów).

W efekcie, przy uwzględnieniu przetwarzania potokowego, szacowanie czasu wykonania przestaje być powiązane z opóźnieniem pojedynczego rozkazu, a staje się zależne od możliwości jak najefektywniejszego zrealizowania przez sprzęt dużej liczby operacji. Możliwości te określane są za pomocą miar przepustowości (*throughput*) wykonywania operacji, definiowanych wprost jako stosunek liczby wykonanych operacji do czasu wykonania. Przepustowość określana jest zazwyczaj dla wykonywania nieskończonego strumienia operacji lub strumienia wystarczająco dużej liczby operacji, pozwalającej pominąć opóźnienia związane z pojedynczą operacją. Często także, określenia "przepustowość" używa się w przypadku optymalnych warunków przetwarzania, kiedy sprzęt uzyskuje swoją maksymalna wydajność.

Przetwarzanie potokowe jest jedną z technik ukrywania opóźnienia (*latency hiding*) – zwiększania wydajności przetwarzania (przepustowości) bez konieczności zmniejszania opóźnienia pojedynczej operacji. Ukrywanie opóźnienia pojawia się w rozmaitych dziedzinach techniki, także techniki obliczeniowej, gdzie w trakcie wykonywania dużej liczby operacji usiłuje się uzyskać czas realizacji krótszy niż wynikający z sumowania opóźnień pojedynczych operacji.

Przyjęcie miary CPI równej 1 dla przykładowego przetwarzania potokowego z rys. 1.7 odpowiada sytuacji idealnej, kiedy nic nie zaburza przetwarzania, a procesor w żadnej chwili nie jest zmuszony do wstrzymania realizacji któregokolwiek ze współbieżnie wykonywanych rozkazów. Takie idealne sytuacje, rzadko obserwowane w praktyce, służą często do określania maksymalnych wydajności sprzętu.

Jedną z miar, które można wykorzystać w tym celu jest miara IPC, liczba rozkazów na takt (*instructions per cycle*), która definiowana jest jako liczba rozkazów kończonych w każdym takcie przez procesor [1]. Dla różnych procesorów można próbować oszacować maksymalne wartości IPC, związane z konkretnymi typami rozkazów. Popularne jest określanie maksymalnej liczby operacji zmiennoprzecinkowych, których wykonanie może skończyć procesor w każdym takcie. Liczba ta jest związana z liczbą i charakterem potoków przetwarzania operacji zmiennoprzecinkowych i służy do określania maksymalnej wydajności (przepustowości) wykonywania operacji zmiennoprzecinkowych przez procesor.

Rozwiązaniem pośrednim pomiędzy miarami związanymi z opóźnieniem przetwarzania rozkazów (zaniżającymi zazwyczaj wydajność) i maksymalną przepustowością przetwarzania (najczęściej zawyżającą wydajność) może być określanie miar wydajności w oparciu o rzeczywiste parametry wykonania programu. Takie miary można definiować jako przeciętne, uśrednione wartości w czasie realizacji obliczeń. W dalszej części książki obie miary IPC i CPI oznaczały będą takie właśnie miary uśrednione. IPC definiowane będzie jako iloraz liczby zrealizowanych rozkazów (l^{ins}) przez liczbę taktów zegara, jaką zajęło wykonanie rozkazów (c^{ins}), dając w efekcie, dla konkretnej sekwencji wykonanych rozkazów,

[1] Dla skrócenia opisu stosowane będą określenia "wydanie rozkazu" (*instruction issue*) jako rozpoczęcie fazy wykonania rozkazu przez procesor (po pobraniu, zdekodowaniu i ewentualnych innych wstępnych operacjach) oraz "kończenie rozkazu" (*instruction retirement*) jako kończenie realizacji rozkazu (skończony rozkaz, *retired instruction*, to rozkaz, którego realizacja została zakończona, przy czym zakończenie może obejmować dodatkowe operacje po opuszczeniu potoków przetwarzania, np. przemianowanie rejestrów).

liczbę rozkazów kończonych przeciętnie w pojedynczym takcie zegara. W przykładzie z rys. 1.7 daje to wartość IPC równą $\frac{n}{(n+k-1)} \to 1$, dla $n \to \infty$.

Uśrednione miary uzyskiwane są, jak widać z powyższego wzoru eksperymentalnie, na podstawie pomiaru czasu wykonania programu. Nie mogą więc służyć do szacowania tego czasu. Ideą użycia miar uśrednionych jest uzyskanie informacji o stopniu wykorzystania sprzętu (stosunku miar uśrednionych do miar ekstremalnych) i przeprowadzanie na ich podstawie wnioskowania o możliwych wartościach miar dla innych programów, dla których nie dokonuje się pomiarów.

Uśredniona miara CPI (*average CPI*), dotycząca całości wykonania programu i będąca prostą odwrotnością miary IPC, definiowana jest jako

$$\mathrm{CPI} = \frac{1}{\mathrm{IPC}} = \frac{c^{ins}}{l^{ins}}$$

W takim ujęciu CPI oznacza średnią liczbę taktów zegara przypadającą na pojedynczy wykonywany rozkaz (uzyskaną jako iloraz całkowitej liczby taktów przez liczbę zrealizowanych rozkazów). W efekcie, przetwarzanie bez współbieżności charakteryzować się będzie wartościami CPI większymi niż jeden, natomiast idealne przetwarzanie potokowe dążyć będzie do wartości CPI równej jeden.

Powyższe definicje i analizy są podstawą tzw. równania wydajności (*performance equation*), realizującego ideę użycia miary CPI do szacowania czasu wykonania. Czas wykonania programu jest rozbijany na iloczyn trzech czynników:

$$\text{czas wykonania} = \frac{\text{liczba sekund}}{\text{program}} = \frac{\text{liczba rozkazów}}{\text{program}} \cdot \frac{\text{liczba taktów}}{\text{rozkaz}} \cdot \frac{\text{liczba sekund}}{\text{takt}}$$

Pierwszy czynnik zależy od kodu źródłowego i strategii doboru rozkazów procesora przez kompilator. Drugim czynnikiem jest parametr CPI dla konkretnego wykonania programu, a trzecim czas trwania pojedynczego taktu procesora, będący odwrotnością częstotliwości jego pracy.

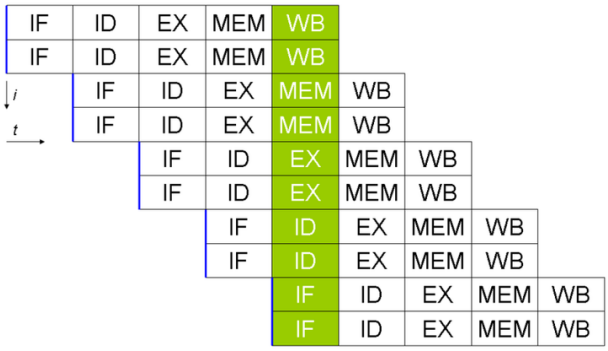
W praktyce uzyskanie wartości liczbowych każdego z czynników napotyka rozmaite trudności. Nawet dla tego samego kodu źródłowego różne kompilatory, w szczególności stosując różne opcje optymalizacji, produkują różne sekwencje rozkazów procesora. Współczynnik CPI, definiowany w sposób określony powyżej, uśrednia wartości, które są nie tylko różne dla różnych rozkazów, ale także, uwzględniając możliwe opóźnienia przetwarzania potokowego, mogą być różne dla tego samego rozkazu, w zależności od tego jakie rozkazy są przetwarzane bezpośrednio przed i bezpośrednio po nim. Wreszcie, częstotliwość pracy współczesnych procesorów nie jest wartością stałą lecz zmienia się w czasie wykonania programów, najczęściej stosownie do realizowanej strategii oszczędności energii.

Niemniej równanie wydajności pozostaje istotną wskazówką optymalizacji, rozumianej jako dążenie do redukcji czasu wykonania. Aby ją osiągnąć należy:

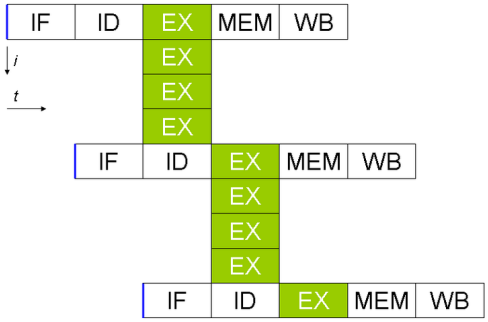
- zmniejszać liczbę rozkazów w kodzie (lub używać bardziej wydajnych rozkazów – np. wektorowych)
- umożliwiać procesorom (rdzeniom) sprawne realizowanie przetwarzania potokowego (maksymalizacja rzeczywistego, uśrednionego IPC i minimalizacja CPI)
- zwiększać częstotliwość pracy procesora (rdzenia)

Superskalarność

Przypadek CPI=IPC=1 nie jest maksymalną wydajnością współczesnych procesorów. Zwielokrotnienie liczby tranzystorów umieszczanych w pojedynczym układzie scalonym, umożliwiło budowanie procesorów o zwielokrotnionych jednostkach funkcjonalnych. Jeśli założymy, że podwojone są wszystkie układy uczestniczące w przetwarzaniu potokowym, otrzymujemy możliwość w pełni równoległego



Rysunek 1.8: Schemat przetwarzania potokowego superskalarnego[źródło: Wikipedia]



Rysunek 1.9: Schemat przetwarzania potokowego SIMD [źródło: Wikipedia]

przetwarzania dwóch rozkazów, nazywanego dwu-drożnym przetwarzaniem superskalarnym (*2-way superscalar processing*), zilustrowanego na rys. 1.8. W przykładzie tym, na każdym etapie przetwarzania potokowego znajdują się dwa rozkazy i w konsekwencji w każdym takcie zegara kończone są dwa rozkazy. Praktycznie stosowane procesory miały i mają możliwości superskalarnego przetwarzania ograniczone do kilku rozkazów. Drożność procesorów, liczba równolegle przetwarzanych rozkazów, zazwyczaj nie przekracza ośmiu, stąd w praktyce spotyka się procesory dwu-, cztero-, sześcio- czy ośmiodrożne (*2-, 4-, 6-, 8-way superscalar*). We współczesnych procesorach, o rozdzielonych podukładach pobierania i wstępnego przetwarzania rozkazów oraz zróżnicowanych potokach przetwarzania, możliwości przetwarzania superkalarnego odnosi się najczęściej do przetwarzania potokowego, liczby potoków i liczby rozkazów wydawanych w pojedynczym takcie (*multiple-issue processors*).

W typowych procesorach, np. z rodziny *x86*, selekcji do wykonania superskalarnego dokonuje układ procesora, niezależnie od kompilatora i kodu źródłowego. Architektury, gdzie kompilator w pojedynczym rozbudowanym słowie przekazuje do wykonania równoległego kilka rozkazów (*very long instruction word, VLIW, architectures*), okazały się w praktyce mniej wydajne od standardowych architektur RISC.

Jeśli wprowadza się zwielokrotnienie liczby potoków wykonania dla pojedynczego zdekodowanego rozkazu, jak na rys. 1.9, przetwarzanie staje się typowym działaniem dla architektury SIMD (*single instruction multiple data*). Jeden pobrany i zdekodowany rozkaz dotyczy wielu egzemplarzy danych i wykonywany jest przez wiele jednostek wykonania. Obie modyfikacje, superskalarność i przetwarzanie SIMD (nazywane także przetwarzaniem wektorowym[2]), powodują, że teoretyczna możliwa do uzyskania

[2]Historycznie nazwa przetwarzania wektorowego i procesorów wektorowych odnosiła się do procesorów zawierających starannie zaprojektowane skalarne potoki fazy wykonania rozkazów zmiennoprzecinkowych, podczas gdy przetwarzanie SIMD charakteryzowało tzw. procesory macierzowe – w obu wypadkach wysoka przepustowość wykonania opierała się na realizacji

0	1	2	3	4	5	6	7	8	9	10	11
IF	ID	EX	MEM	WB							
	IF	*	*	*	ID	EX	MEM	WB			
		IF	*	*	*	ID	EX	MEM	WB		
			IF	*	*	*	ID	EX	MEM	WB	
				IF	*	*	*	ID	EX	MEM	WB

Rysunek 1.10: Przestój przetwarzania potokowego [źródło: Wikipedia]

przez procesor wartość CPI spada poniżej 1, natomiast wartość IPC sięga kilku lub nawet kilkudziesięciu (jak np. w przypadku procesorów graficznych).

Problemy przetwarzania potokowego

Wysoka teoretyczna wartość IPC współczesnych procesorów rzadko kiedy osiągana jest w praktyce. Jedną z przyczyn są zaburzenia idealnego przetwarzania potokowego. Rysunek 1.10 przedstawia sytuację przestoju przetwarzania potokowego (*pipeline stall*), w której procesor nie jest w stanie wykonać fazy ID drugiego rozkazu, do momentu zakończenia fazy WB rozkazu pierwszego.

Istnieje szereg przyczyn przestojów przetwarzania potokowego. Określane są one jako hazardy (*hazards*), czyli sytuacje, w których standardowe (niemodyfikowane i niewstrzymywane) przetwarzanie potokowe prowadzi do ryzyka wystąpienia błędów przetwarzania. Istnieje kilka typów hazardów:

- hazardy zasobów – kiedy dwa lub więcej rozkazów chce jednocześnie korzystać z tych samych zasobów procesora
- hazardy sterowania – kiedy w kodzie pojawia się instrukcja skoku (warunkowego lub bezwarunkowego)
- hazardy danych – kiedy poprawne wykonanie kolejnego rozkazu wymaga znajomości wyniku wcześniejszego rozkazu, co wyklucza współbieżność (taka definicja obejmuje tylko hazardy odczyt-po-zapisie, *read-after-write*, nie obejmuje możliwych do usunięcia przez odpowiednie transformacje kodu hazardów zapis-po-zapisie, *write-after-write*, i zapis-po-odczycie, *write-after-read*).

Unikanie hazardów i związanych z nimi opóźnień przetwarzania potokowego jest z jednej strony celem projektowania procesorów, a z drugiej techniką optymalizacji kodu. W tym ostatnim przypadku najistotniejsze jest unikanie hazardów danych, które na poziomie kodu źródłowego przejawiają się w postaci zależności danych pomiędzy instrukcjami kodu (np. rzeczywistej zależności odczyt-po-zapisie, kiedy jedna instrukcja korzysta z wartości zmiennej uzyskanej w wyniku wykonania wcześniejszej instrukcji)[3].

Na poziomie projektowania procesorów istnieje szereg technik unikania hazardów i opóźnień, z których większością nie daje się sterować poprzez optymalizację kodu przez programistę. Dwie spośród technik najważniejszych dla wydajności przetwarzania, to przewidywanie rozgałęzień (skoków, *branch prediction*) i wykonywanie poza kolejnością (*out-of-order execution*).

pojedynczego rozkazu dla wielu egzemplarzy danych.

[3] Zależność danych pomiędzy dwoma instrukcjami kodu powstaje, kiedy obie korzystają z tej samej komórki pamięci (tej samej zmiennej) i choć jedna z nich dokonuje zapisu – współbieżne wykonanie takich instrukcji prowadzi do niedeterministycznego wykonania programu (wynik zależy od kolejności wykonania instrukcji, która nie jest poddana kontroli).

Pierwsza z technik ma na celu uniknięcie przestojów spowodowanych hazardami sterowania w przypadku skoków warunkowych. W przetwarzaniu niezmodyfikowanym, w sytuacji kiedy w potoku pojawia się rozkaz skoku warunkowego, informacja o tym, który rozkaz zostanie wykonany jako następny staje się dostępna dopiero po obliczeniu wartości logicznej warunku. W technice przewidywania rozgałęzień wykorzystuje się fakt, że większość rozgałęzień (skoków) w kodzie (np. związanych z wykonywaniem pętli) wykonywana jest wielokrotnie, a częstość spełniania lub nie warunku układa się w powtarzalny schemat (np. przy wykonywaniu długich pętli, skok na początek pętli realizowany jest miliony czy miliardy razy, natomiast przejście dalej tylko raz).

W przewidywaniu rozgałęzień, na podstawie wcześniejszych wartości warunku związanego ze skokiem, przewidywana jest wartość aktualna i realizowany skok związany z tą wartością. W przypadku kiedy wartość jest przewidziana prawidłowo eliminuje się przestój potoku. W przypadku błędnej predykcji, wykonanie cofa się ponownie do rozkazu skoku, a potok jest czyszczony (*flushed*) z efektów wszystkich rozkazów, które nastąpiły po nim. Technika ta, będąca przykładem wykonania spekulatywnego (*speculative execution*), powoduje w przypadku błędnego przewidywania skoków większe opóźnienie niż w przypadku wyłącznie wstrzymania przetwarzania niezmodyfikowanego, dlatego ostateczna przydatność przewidywania rozgałęzień zależy od jego skuteczności.

Istnieje szereg praktycznych realizacji mechanizmu przewidywania rozgałęzień. Potrafią one osiągać we współczesnych procesorach skuteczność, dla przeciętnych programów, rzędu 80-90 procent przypadków, a dla specyficznych fragmentów kodu (jak wspomniane wcześniej długie pętle) sięgać blisko 100 procent. Ma to istotne znaczenie dla wydajności, ponieważ w przeciętnych programach rozkazy skoku warunkowego (ze względu na m.in. powszechność stosowania pętli) mogą stanowić kilka-kilkanaście procent całkowitej liczby rozkazów wykonywanych w trakcie realizacji programu.

Druga z ważnych technik unikania opóźnień przetwarzania potokowego, wykonywanie poza kolejnością, wymaga istotnych zmian w układach procesora. Pobierane rozkazy gromadzone są w odpowiednich strukturach danych, przeglądane, a następnie dokonywane jest ustalenie kolejności wykonania i przeprowadzane są ewentualne modyfikacje rozkazów (np. zmiana nazw rejestrów rozkazu). Kolejność wykonania rozkazów przez procesor jest ustalana wewnętrznie, jednak efekt zewnętrzny musi być identyczny jak w przypadku wykonania w kolejności wynikającej z zapisu kodu binarnego w pamięci (dotyczy to jednak tylko pojedynczego wątku, a więc pojedynczej sekwencji rozkazów). W związku z tym istnieje jeszcze jeden etap, porządkujący efekty pracy procesora, etap opuszczania procesora przez rozkaz (*instruction retirement*).

Układy wykonywania poza kolejnością zabierają znaczną liczbę tranzystorów i powierzchni procesora, co np. oznacza mniejszą liczbę tranzystorów na układy wykonania rozkazów, jednak ich działanie jest na tyle ważne, że stanowią element wszystkich procesorów ogólnego przeznaczenia (o specjalnych procesorach, w których rozkazy wykonywane są w kolejności, mowa będzie w dalszej części książki).

Ostatnią omawianą, choć w praktyce zazwyczaj najważniejszą, przyczyną opóźnień przetwarzania potokowego jest przyczyna leżąca poza układem wykonania rozkazów przez procesor. Wstrzymanie przetwarzania ma miejsce w sytuacji, kiedy nie zostały dostarczone dane potrzebne do wykonania rozkazu. Częściowo problem może zostać rozwiązany przez wykorzystanie wykonania poza kolejnością (lub wielowątkowość, o czym będzie mowa w kolejnych punktach), jednak podstawowe znaczenie ma optymalizacja układu dostarczania danych, związana z organizacją pamięci w systemie komputerowym.

1.2.4 Wielopoziomowa organizacja pamięci

W klasycznej architekturze von Neumanna, dla której wciąż jeszcze tworzony jest w przypadku większości procesorów kod binarny (z możliwym zapisem w językach asemblera), istnieją tylko dwa typy pamięci jako argumenty rozkazów: rejestry i pamięć główna.

Rejestry stanowiące wewnętrzną pamięć procesora, w szczególności ich liczba i typy (określające najczęściej rozmiar w bitach oraz przeznaczenie), decydują w istotnej mierze o możliwościach wydajnościowych procesora. Możliwość jawnego wykorzystania rejestrów w programowaniu istnieje tylko w przypadku stosowania asemblera lub specjalnych rozszerzeń języków programowania, występujących często w postaci zbliżonych do asemblera wstawek (*intrinsics*) rozpoznawanych przez kompilatory języków wyższego poziomu. W niniejszej książce, jako podstawowe, analizowane będą techniki modyfikacji standardowego kodu źródłowego i ich wpływ na wydajność wykonania, bez jawnego stosowania programowania w asemblerze. Niemniej jednak, badanie możliwego wykorzystania rejestrów przez procesor będzie istotnym elementem analizy wydajności i optymalizacji programów.

W języku asemblera dostęp do pamięci głównej odbywa się poprzez użycie rozkazów procesora, w których jako argument występuje adres pamięci, przechowywany w rejestrze lub obliczany na podstawie wartości przechowywanych w kilku rejestrach, dla złożonych trybów adresowania. Rozmiar używanych w tym celu rejestrów (8, 16, 32, 64 bity) jest związany z rozmiarem dostępnej przestrzeni adresowej (2^n komórek pamięci, gdzie n jest liczbą bitów rejestrów), wpływa na konstrukcję i funkcjonowanie układów pamięci oraz magistral łączących procesor z pamięcią.

Jedną z podstawowych, ze względu na wydajność, modyfikacji standardowego modelu pamięci architektury von Neumanna, jest wprowadzenie pamięci podręcznej (*cache memory*). Jej całościowemu omówieniu poświęcony jest jeden z kolejnych podrozdziałów książki. W tym miejscu zaznaczony jest tylko jej podstawowy schemat: zamiast jednego poziomu pamięci (a więc sytuacji, kiedy wartość konkretnej zmiennej w programie przechowywana jest zawsze w jednej tylko lokalizacji w pamięci), istnieje wiele poziomów, tak że z jedną zmienną w programie może być związane kilka lokalizacji (w pamięci głównej i różnych poziomach pamięci podręcznej), z wartością najbardziej aktualną przechowywaną tylko w niektórych lokalizacjach.

Najbliżej procesora (pomijając rejestry) znajduje się poziom L1 pamięci podręcznej. W najpopularniejszych architekturach procesorów, poziom ten zorganizowany jest w sposób odpowiadający tzw. architekturze harvardzkiej. Pamięć główna (DRAM), zgodnie z architekturą von Neumanna, przechowuje w jednej przestrzeni adresowej kod programu i jego dane, w architekturze harvardzkiej istnieją osobne pamięci dla kodu i dla danych.

We współczesnych procesorach pobranie lub zapis dowolnego argumentu rozkazu asemblera z lub do pamięci głównej jest złożonym procesem. Adres zapisany w rejestrze nie jest adresem fizycznym wykorzystywanym przy sprzętowym dostępie do układów pamięci i musi zostać na taki adres przetłumaczony. Stosuje się w tym celu rozmaite techniki, zależne od systemu operacyjnego, z których najpopularniejszą dzisiaj jest technika pamięci wirtualnej (omówiona szerzej w p. 1.3.1). Architektury procesorów wspierają stosowanie pamięci wirtualnej, poprzez przechowywanie związanej z nią tablicy stron, wspomagającej tłumaczenie adresów z wirtualnych na fizyczne, w pełnej hierarchii pamięci. Tablica stron posiada więc specjalne dedykowane pamięci podręczne, zwane także tablicami translacji adresów (TLB, *Translation Lookaside Buffer*), w tym pamięć poziomu L1 bezpośrednio w procesorze.

1.2.5 Przetwarzanie SIMD i wektoryzacja

Wspomniane w punkcie 1.2.3, omawiającym modyfikacje klasycznego przetwarzania potokowego, przetwarzanie SIMD jest na poziomie procesora związane z wykorzystaniem rejestrów wektorowych. Rejestr wektorowy to odpowiednio szeroki rejestr (w klasycznej architekturze x86 i jej rozszerzeniach od 64 do 512 bitów), który może być użyty do przechowania kilku egzemplarzy liczb określonego typu. Przykładowo, rejestr 128-bitowy może pomieścić cztery liczby 32-bitowe, całkowite lub zmiennoprzecinkowe pojedynczej precyzji, 2 liczby 64-bitowe lub większą liczbę zmiennych o typach wymagających 8 lub 16 bitów.

Z rejestrami wektorowymi określonego typu (czyli określonej szerokości) związany jest specjalny zestaw rozkazów na liście rozkazów procesora. Współczesne procesory udostępniają po kilka zestawów rejestrów (różne procesory oferują różną liczbę i różny zakres szerokości rejestrów), często opatrując zestawy rozkazów operujących na danych rejestrach specjalnymi nazwami (np. MMX, 3DNOW, SSE). Odwołania w tych nazwach do pojęć związanych z renderowaniem grafiki (np. MMX – *Multimedia Extensions*), wynikają z pierwotnego ukierunkowania wykorzystania rozkazów wektorowych wyłącznie na przetwarzanie grafiki. Jednak dzisiaj, w sytuacji kiedy rejestry wektorowe osiągają rozmiar 512 bitów, każda dziedzina zastosowań powinna starać się maksymalizować wykorzystanie rejestrów wektorowych. W podanym przykładzie rejestrów 512-bitowych, wykorzystanie jednego rozkazu wektorowego dla liczb całkowitych jest równoważne realizacji 16 odpowiadających operacji skalarnych. Odpowiednia organizacja obliczeń pozwala ukryć narzut związany z wykonaniem wektorowym, wynikający m.in. z konieczności pakowania i rozpakowywania kilku czy kilkunastu egzemplarzy danych do i z rejestrów wektorowych, i uzyskać przyspieszenie obliczeń wprost związane z szerokościa rejestrów, czyli w rozważanym przypadku przyspieszenie 16-krotne.

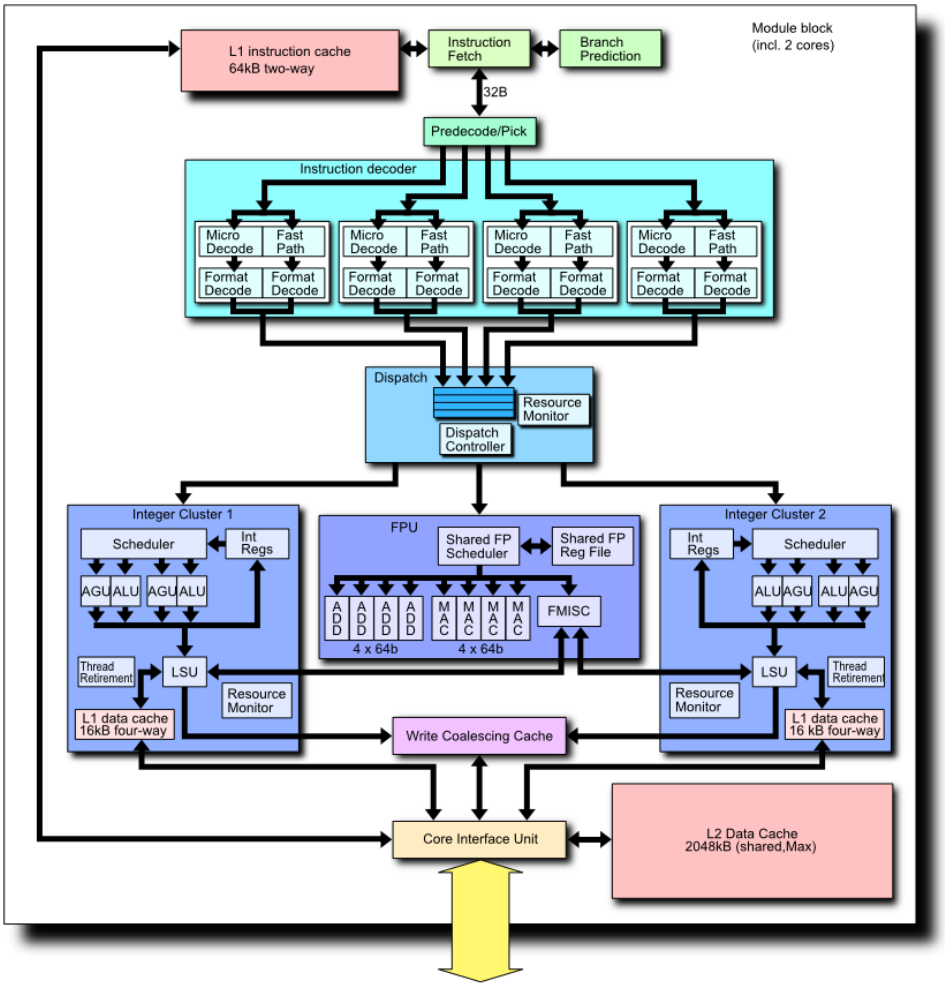
Patrząc na ten aspekt z innej strony, można uznać, że nie zastosowanie rozkazów wektorowych oznacza kilku- lub kilkunastokrotne zmniejszenie wydajności w stosunku do maksymalnej oferowanej przez procesor.

Z przetwarzaniem SIMD wiąże się jeszcze jedno pojęcie wygodne w analizie wydajności. Rozkazy wektorowe są wykonywane na argumentach będących rejestrami wektorowymi, przy wykorzystaniu specjalnych wektorowych potoków przetwarzania. Działanie takiego potoku można sobie wyobrazić jako działanie, w pełnej synchronizacji sprzętowej, kilku pojedynczych potoków przetwarzania skalarnego. Tak wyodrębniony potok odpowiadający przetwarzaniu skalarnemu będzie dalej nazywany ścieżką SIMD (*SIMD lane*).

W prezentowanym ujęciu, wykonanie pojedynczego rozkazu wektorowego jest związane z wykorzystaniem tylu ścieżek SIMD, ile egzemplarzy danych jest spakowane w pojedynczym rejestrze wektorowym, natomiast wykonanie dowolnej operacji skalarnej jest związane z pojedynczą ścieżką SIMD. Występowanie na schemacie procesora potoku przetwarzania skalarnego określonego typu oznacza istnienie w procesorze pojedynczej ścieżki SIMD tego typu. Występowanie potoku przetwarzania wektorowego oznacza istnienie kilku ścieżek SIMD, jednak ich liczba zależy od typu rejestrów i zestawu powiązanych rozkazów. Potok przetwarzania rozkazów 256-bitowych jest równoważny istnieniu 8 ścieżek SIMD dla zmiennych 32-bitowych i 4 ścieżek dla zmiennych 64-bitowych. Jest to prawdziwe w sytuacji kiedy te same potoki wektorowe obsługują różne typy danych. W wielu architekturach występują odrębne potoki przetwarzania dla różnych typów danych, w szczególności 32 i 64 bitowych. W takim przypadku liczba ścieżek SIMD jest odrębnie określana dla każdego z typów danych.

Ścieżki SIMD są wygodnym narzędziem charakteryzowania maksymalnej wydajności przetwarzania dla rozkazów konkretnego typu. Maksymalna wydajność procesora to łączna wydajność wszystkich ścieżek SIMD danego typu pracujących jednocześnie. Liczba ścieżek SIMD to liczba jednocześnie wykonywanych operacji skalarnych danego typu. Często występującym przypadkiem jest sytuacja kiedy maksymalna wydajność osiągana jest w wyniku pracy kilku potoków wektorowych. Wtedy wydajność procesora dla danego typu operacji jest iloczynem liczby potoków i liczby egzemplarzy danych w jednym rejestrze wektorowym danego typu (równa liczbie ścieżek SIMD w potoku). Wydajność taka określa maksymalną liczbę skalarnych operacji danego typu (np. zmiennoprzecinkowych pojedynczej lub podwójnej precyzji) kończonych w pojedynczym takcie procesora (zakładając, że potoki pozwalają na kończenie jednego rozkazu wektorowego w każdym takcie). Po pomnożeniu przez częstotliwość pracy procesora uzyskujemy maksymalną wydajność procesora w liczbie operacji na sekundę.

W dalszej części książki, jako sprzęt na którym uruchamiane są przykładowe programy, wykorzystywane jest kilka mikroprocesorów. Jednym z nich jest 4-rdzeniowy mikroprocesor Intel Core i7-4790, z rdzeniami o architekturze Haswell i nominalną częstotliwością pracy 3.60 GHz. Każdy rdzeń posiada



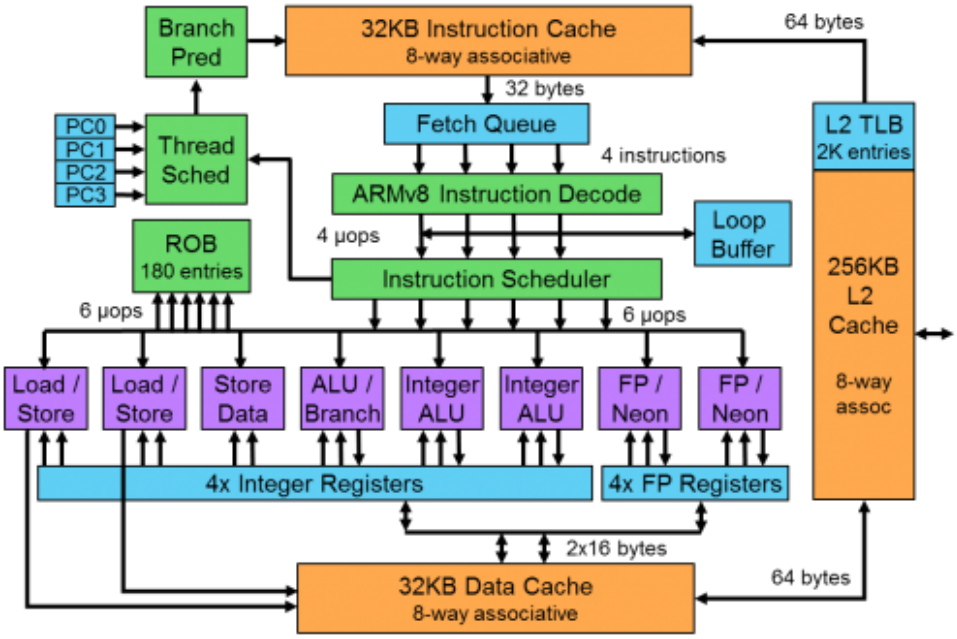
Rysunek 1.11: Architektura AMD Bulldozer [źródło: Wikipedia]

dwa potoki przetwarzania rozkazów 256-bitowych. Każdy z potoków potrafi w jednym takcie kończyć jedną połączoną operację mnożenia i dodawania (*fused multiply-add – FMA*). Dla zmiennych podwójnej precyzji oznacza to 8 (4·2) skalarnych operacji kończonych w każdym takcie. Ostateczna maksymalna wydajność pojedynczego rdzenia wynosi $8 \cdot 2 \cdot 3.6 \cdot 10^9 = 57.6 \cdot 10^9$ operacji arytmetycznych podwójnej precyzji na sekundę, czyli 57.6 GFlops (*floating point operations per second*), co prowadzi do maksymalnej wydajności czterech rdzeni mikroprocesora 230.4 GFlops.

1.2.6 Przykłady mikroarchitektur procesorów

Rysunki 1.11, 1.12 i 1.13 przedstawiają przykłady diagramów blokowych pojedynczych rdzeni współczesnych mikroprocesorów, jako odpowiedników klasycznej maszyny von Neumanna.

Większość współczesnych procesorów (rdzeni), w tym przykładowe (mikro-)architektury AMD Bulldozer, ARMv8 i Intel Core 2 (a także nowsze), posiada zbliżoną do siebie budowę. Można w nich wyróżnić dwie podstawowe grupy układów: system pobierania i dekodowania rozkazów oraz system wykonywania rozkazów, a także występujący pomiędzy nimi układ planowania (*schedule*) i rozdysponowania (*dispatch*) rozkazów. Poniżej omówione są podstawowe elementy przykładowych architektur (stosując wspólne nazwy układów, a w przypadku gdy oznaczenia układów na diagramach są różne, zazwyczaj



Rysunek 1.12: Architektura ARMv8 [źródło: Wikipedia]

używając w tekście kolejności odpowiadającej kolejności rysunków), z pominięciem niektórych bloków oraz bardziej szczegółowych informacji zawartych na diagramach .

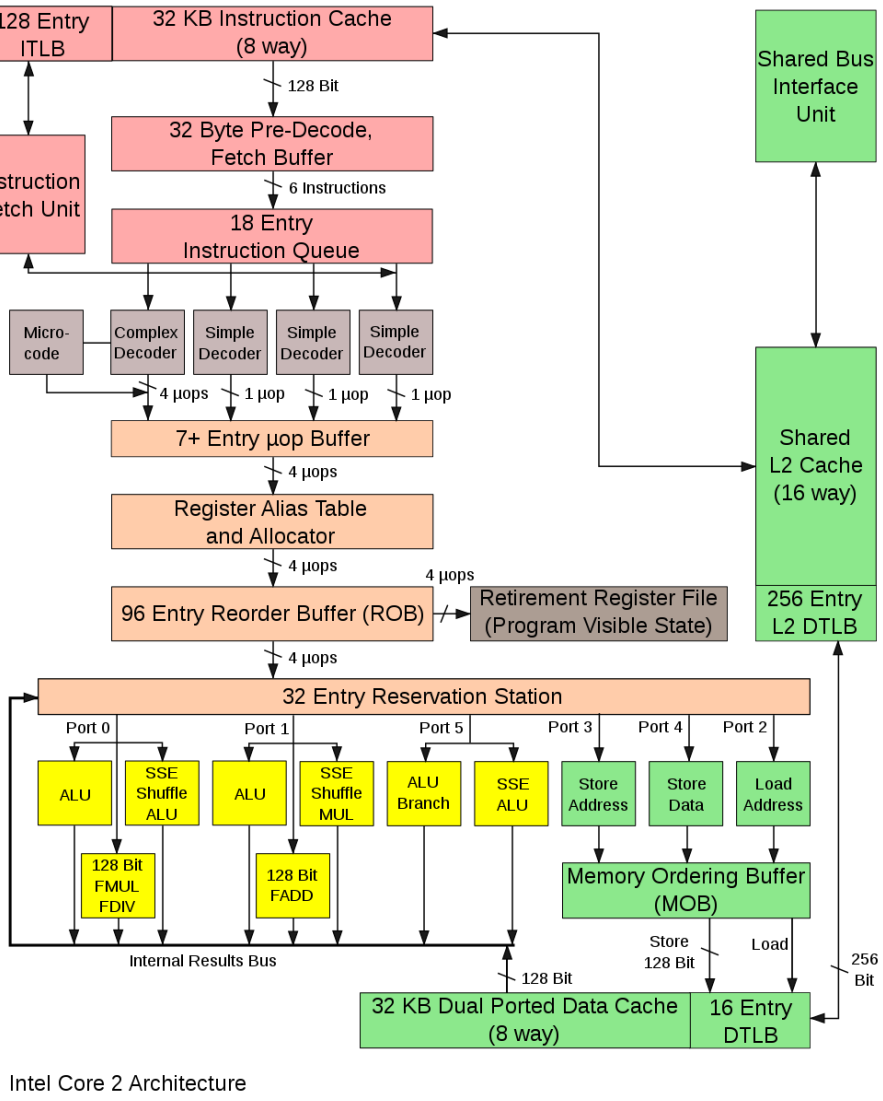
Analizując budowę procesorów w kolejności odpowiadającej kolejności przetwarzania pojedynczego rozkazu, jako pierwsze występują układy pobierania rozkazów. Źródłem rozkazów jest pamięć podręczna L1 rozkazów, oznaczana na diagramach jako blok *(L1) Instruction Cache*. Rozkazy z pamięci podręcznej, mającej typowe rozmiary 32, 64 kB, trafiają do układu pobierania, który przechowuje jednocześnie wiele rozkazów w odpowiednich strukturach danych (bloki: AMD – *Instruction Fetch*, ARM – *Fetch Queue*, Intel – *Fetch Buffer, Instruction Queue, Instruction Fetch Unit*). Pobieranie rozkazów wymaga translacji ich adresów, do czego służą układy pamieci TLB, występujacej także w wariancie dedykowanym wyłącznie dla rozkazów (Intel – *ITLB*).

Umieszczenie zbioru rozkazów w strukturach danych procesora umożliwia ich wstępne zdekodowanie, przeglądnięcie zawartości i operacje takie jak przewidywanie rozgałęzień (bloki *Branch Prediction*, w architekturze Intel wewnątrz *Instruction Fetch Unit*).

Dalszym etapem przetwarzania jest pełne dekodowanie rozkazów, do postaci właściwej dla rozdysponowania (wydania) do poszczególnych potoków wykonania. Wykonanie rozkazów odbywa się przez potoki specjalnie zaprojektowane do realizacji różnych typów rozkazów. Każdy rozkaz określonego typu rozbijany jest na właściwą sobie liczbę faz i wykonywany przy użyciu odpowiednich zasobów procesora. Na przykładowych diagramach znajdują się potoki:

- ALU (*Arithmetic-Logic Unit*), *Integer ALU* – klasyczne potoki realizacji operacji arytmetyczno-logicznych, wykonywanych na liczbach całkowitych

- AGU (*Address Generation Unit*), LSU (*Load/Store Unit*), *Load/Store Adress*, *Load/Store Data* – jednostki pobierania i zapisywania danych oraz rozkazów, z i do hierarchii pamięci, w tym ewentualnie odrębne jednostki tłumaczenia i obliczania (generowania) adresu (związane m.in. z faktem stosowania pamięci wirtualnej i występowania w asemblerze rozkazów ze złożonymi trybami adresowania, w których adres jest obliczany za pomocą zestawu prostych operacji arytmetycznych)



Rysunek 1.13: Architektura Intel Core 2 [źródło: Wikipedia]

- FP, (F)ADD, (F)MUL, (F)MAC, F(MAC), (F)DIV – jednostki wykonywania operacji na argumentach zmiennoprzecinkowych (dodawanie, mnożenie, dzielenie, łączne dodawanie i mnożenie – oznaczane jako MAC – *Multiply-Accumulate*, MAD – *Multiply-Add* lub FMA – *Fused Multiply-Add*), obejmujące także jednostki wykonywania rozkazów SIMD (o nazwach specyficznych dla rodziny procesorów, np. SSE, Neon, AVX, VMX)

- *Branch*, *Shuffle* – jednostki wykonywania specjalnych typów rozkazów

Występowanie wielu, w praktyce niezależnych, potoków wymusza istnienie rozbudowanych układów pomiędzy podsystemem pobierania i dekodowania rozkazów, a jednostkami (potokami) wykonywania rozkazów (stosuje się także pojęcie portów, jawnie zaznaczonych na diagramie mikroarchitektury firmy Intel, jako jednostek rozdysponowania rozkazów). Organizacja i nadzorowanie pracy jednostek przetwarzania potokowego wymagają szeregu działań, takich jak: rozdzielanie rozkazów pomiędzy potoki, zarządzanie przydziałem rejestrów dla rozkazów (co związane może być z przemianowaniem rejestrów, *register renaming*), porządkowanie opuszczania procesora przez rozkazy, *instruction retirement*.

Działania te realizowane są przez układy oznaczane na diagramach jako *Dispatcher* (dyspozytor), *Scheduler* (planista) i wykorzystują dodatkowe, zaznaczone na diagramach podukłady i struktury danych (jak np. ROB, *Reorder Buffer* lub *Retirement Register File*).

Optymalizacja przetwarzania przez pojedynczy rdzeń mikroprocesora

Jak widać na przykładach zaprezentowanych mikroarchitektur, budowa pojedynczego rdzenia mikroprocesora jest na tyle złożona, że optymalizacja jego pracy przy realizacji konkretnego programu wymaga uwzględnienia szeregu aspektów, takich jak między innymi:

- liczba rozkazów w kodzie asemblera – im wyższa tym większe wymagania szybkości pobierania i dekodowania rozkazów przez procesor, w szczególności w przypadku rozkazów o krótkim czasie wykonania

- złożoność rozkazów, w tym złożoność trybów adresowania – wpływające na wymagania szybkości dekodowania, a także na liczbę pojedynczych operacji związanych z realizacją rozkazów (pośrednio na liczbę potoków przetwarzania zaangażowanych w realizację rozkazu – zazwyczaj potoki przetwarzania związane są z mikro-rozkazami na które rozbijane są rozkazy z listy rozkazów procesora)

- sposób organizacji obliczeń – powiązany z możliwościami efektywnego użycia układów przewidywania rozgałęzień i wykonywania poza kolejnością

- wzajemne zależności między rozkazami – określające hazardy przetwarzania, a więc i opóźnienia przetwarzania potokowego

- organizacja przechowywania struktur danych oraz sposoby korzystania z pamięci głównej

Ostatni z wymienionych aspektów w przypadku wielu algorytmów okazuje się być decydującym o ostatecznej wydajności. Jest jednakże jednym z aspektów najbardziej złożonych – obejmuje nie tylko liczbę dostępów do danych, ale także organizację tych dostępów, ich umiejscowienie w kodzie asemblera i wzajemne uporządkowanie, co w dalszej kolejności określa korzystanie z pamięci podręcznej rozmaitych poziomów, a także pamięci wirtualnej z obsługą tablicy stron i związanych z nią pamięci podręcznych ITLB i DTLB

Z punktu widzenia wydajności idealną sytuacją jest pełne wykorzystanie wszystkich jednostek potokowych procesora, kiedy wszystkie jednostki pracują równolegle, a ich potoki nie mają przestojów. Układ pobierania i dekodowania rozkazów oraz układ rozdysponowania rozkazów są w stanie dostarczać w każdym takcie każdemu potokowi rozkazy do wykonania wraz z potrzebnymi danymi. Taka wizja pracy procesora prowadzi do modelu analizy wydajności, w którym w przypadku, gdy wydajność odbiega od optymalnej, badane jest, które elementy okazały się wydajnościowym wąskim gardłem (*performance bottleneck*), decydującym o wydłużeniu czasu realizacji obliczeń.

Powyższy model analizy ma swoje wady. Bada wykorzystanie procesora jako całości, bez względu na specyfikę wykonywanego programu. Czasem postać kodu źródłowego jawnie determinuje, które elementy procesora będą wykorzystane, a które nie. Na przykład, w przypadku kodu o zdecydowanej przewadze operacji zmiennoprzecinkowych, potoki przetwarzania liczb całkowitych będą mało wykorzystane, a uśrednione wyniki wykorzystania wszystkich potoków mało reprezentatywne dla rzeczywistej wydajności przetwarzania.

Z tych względów, w niniejszej książce przyjęty jest inny kierunek badania. Punktem wyjścia są algorytmy, a nie poszczególne układy procesora. Istotne (przynajmniej dla określonych dziedzin zastosowań)

algorytmy i ich implementacje są analizowane, określane są czynniki decydujące o ich wydajności, tworzone modele wydajności i poszukiwane efektywne metody optymalizacji. Dla każdego programu rozważane są tylko potoki przetwarzania istotne dla przetwarzania tego właśnie programu, w konkretnym momencie jego wykonania.

Specyfika analizowanych w książce algorytmów sprawia, że nie pojawia się problematyka pobierania i dekodowania rozkazów – w przypadku rozważanych programów etapy te nie stanowią czynnika ograniczającego wydajność dla współczesnych procesorów.

1.2.7 Liczniki sprzętowe

Cennym narzędziem wspomagającym badania funkcjonowania mikroprocesorów, oraz współpracujących z nimi innych układów, są liczniki sprzętowe (*hardware counters*), wspomniane już w p. 1.1.2. Idea liczników sprzętowych polega na wyodrębnieniu szczegółowych zdarzeń związanych z realizacją programów i zliczaniu ich wystąpień, najczęściej w trakcie wykonania konkretnego programu, a czasem, w określonym przedziale czasu, dla całego systemu lub wybranego układu (np. pojedynczego rdzenia mikroprocesora). Najważniejszym odnotowywanym zdarzeniem jest zawsze pojedynczy takt zegara – dzięki temu można określić liczbę taktów w trakcie wykonania programu. Liczba taktów wykorzystywana jest także do uzyskania pochodnych miar wydajności, wyrażanych w liczbie zdarzeń przypadających na pojedynczy takt (lub liczbie taktów na pojedyncze analizowane zdarzenie).

Mechanizm sprzętowy wykorzystywany w licznikach to specjalne dedykowane rejestry oraz odpowiednie układy zwiększania wartości rejestrów w przypadku zajścia konkretnego zdarzenia. Dla każdego rdzenia mikroprocesora istnieje tylko kilka rejestrów do zliczenia zdarzeń, co oznacza, że dla każdego wykonania programu można śledzić częstość występowania tylko kilku zdarzeń. Jednak rejestry w większości są rejestrami ogólnego przeznaczenia (z wyjątkiem paru, np. rejestrów do zliczania taktów zegara) i przy każdym uruchomieniu programu można zlecać zliczanie innych zdarzeń. Istnieje kilka możliwych sposobów dostępu do rejestrów zliczających zdarzenia (ich resetowania, przypisywania do konkretnych zdarzeń, odczytywania wartości), tak z poziomu rozkazów asemblera, funkcji systemowych, jak i odpowiednich programów narzędziowych.

Początkowo liczniki sprzętowe obejmowały podstawowe zdarzenia, jak np. wykonywane operacje arytmetyczne z podziałem na całkowite i zmiennoprzecinkowe, skoki warunkowe i bezwarunkowe itp. Z czasem liczba zdarzeń możliwych do zliczania dla konkretnego procesora/rdzenia rosła, osiągając dziś kilkaset i obejmując także zdarzenia zachodzące poza rdzeniem. Często zdarzenia zdefiniowane dla procesora są bardzo szczegółowe (np. ”liczba taktów w trakcie gdy mikrooperacje inicjowane przez bufor dekodowania są dostarczane do kolejki dekodowania rozkazów, podczas gdy układ kolejkowania rozkazów jest zajęty”), co związane jest z faktem, że podstawową rolą liczników sprzętowych jest wspieranie procesu analizowania i projektowania budowy procesorów/rdzeni oraz ich pojedynczych detali architektonicznych.

Z punktu widzenia analizy wykonania programów, użycie liczników sprzętowych może napotykać trudności. Pewne zdarzenia, których zliczanie byłoby istotne dla analizy wydajności, czasem nie są definiowane (np. dla ostatnich generacji procesorów firmy Intel nie jest zdefiniowane zdarzenie wykonania operacji zmiennoprzecinkowej). Często interesujące zdarzenia są uwikłane w złożone mechanizmy działania procesora/rdzenia, przez co relatywnie prostym instrukcjom w programie mogą odpowiadać różne zdarzenia lub kombinacja kilku zdarzeń (dzieje się tak np. kiedy procesor wykonuje działania niewynikające bezpośrednio z kodu, jak w przypadku realizacji ścieżki wybranej przez układ przewidywania rozgałęzień, która w rzeczywistości nie jest wybrana, lub pobierania danych, które nie są później wykorzystywane w programie[4]).

[4]W ramach mechanizmu pobierania z wyprzedzeniem omawianego w dalszej części książki

Niemniej liczniki sprzętowe, ze względu na swoją fundamentalną rolę, są coraz powszechniej wykorzystywane w analizie wydajności programów, przy zachowaniu jednak szeregu środków zapobiegających możliwym błędom pomiaru lub interpretacji wyników.

Narzędzia umożliwiające dostęp do liczników sprzętowych

Pierwszym ze sposobów dostępu do rejestrów zliczających konkretne zdarzenia jest wykorzystanie w programach wstawek kodu asemblera zapisujących i odczytujących wartości odpowiednich rejestrów. Rozwiązanie to, poza brakiem przenośności między architekturami procesorów, może także wymagać wyższych uprawnień przy wykonywaniu kodu, niż dostępne standardowemu użytkownikowi. Z powodu tych wad wprowadzane są narzędzia wyższego poziomu. W systemie Linux istnieje narzędzie *perf*, korzystające z modułu wbudowanego w jądro Linuxa i służące do wspomagania analizy wydajności programów. Pozwala ono między innymi na uzyskiwanie wartości liczników sprzętowych dla zbioru wstępnie zdefiniowanych zdarzeń lub zdarzeń zadanych przez użytkownika, związanych z wykonywanym programem. W książce wykorzystywana będzie komenda *perf stat*, zwracająca za pomocą prostego interfejsu wartości liczników sprzętowych dla kilku wybranych, podstawowych zdarzeń w trakcie wykonania programu określonego jako jej argument.

Do uzyskania wartości liczników sprzętowych dla fragmentów kodu używane będzie w książce narzędzie PAPI (*Performance Application Programming Interface*). Jego kolejne wersje dostarczają przenośny interfejs oraz realizującą go bibliotekę, umożliwiające dostęp do liczników sprzętowych dla kolejnych generacji mikroprocesorów, włączając w to także procesory graficzne. PAPI udostępnia wartości liczników dla wszystkich możliwych do zaprogramowania zdarzeń dla konkretnych procesorów, posługując się także zestawem wybranych, podstawowych zdarzeń (tzw. predefiniowane zdarzenia PAPI), traktowanych jako powszechnie występujące we wszystkich typach architektur.

Prosty przykład wykorzystania liczników sprzętowych

Pierwszy prezentowany przykład wykorzystania liczników sprzętowych dotyczy przewidywania rozgałęzień. Badany fragment kodu ma postać:

```
for(i=0;i<1000000;i++)
  x = drand48();
  if(x<0.5) {
    // proste operacje arytmetyczne
  }
}
```

W każdej z miliona wykonywanych pętli znajdują się dwa skoki warunkowe:

- jeden, prosty do przewidzenia, skok na początek pętli związany z warunkiem dotyczącym zmiennej `i` (tylko raz nie wykonany, pozostałe razy wykonany)
- drugi, związany z wartością zmiennej `x`, niemożliwy do przewidzenia (funkcja `drand48` zwraca losową wartość z przedziału $(0, 1)$)

Do sprawdzenia działania układu przewidywania rozgałęzień, kod, po kompilacji do pliku wykonywalnego *a.out*[5], uruchamiany jest w ramach narzędzia *perf*:

```
$ perf stat a.out
```

[5]Przy kompilacji należy zwrócić uwagę czy kompilator nie zastosował optymalizacji zmieniających charakter wykonywanego kodu, np. dokonując rozwinięcia pętli, zamiany skoku na operacje arytmetyczne lub tp. (patrz p. **??**)

Jedną ze zwracanych przez `perf stat` wartości jest wartość licznika związanego ze zdarzeniem niepoprawnego przewidywania rozgałęzień (*branch misses*), która w przypadku badanego kodu osiąga wartość zbliżoną do przewidywanej liczby 500000, połowy wszystkich iteracji:

```
Performance counter stats for 'a.out':
...
        368727897     cycles
...
        500846        branch-misses
...
        0,092846243   seconds time elapsed
```

Zwraca uwagę fakt, że praktycznie wszystkie skoki związane z realizacją pętli są przewidziane prawidłowo (z 2000000 skoków warunkowych w programie, ok. 1500000 jest przewidzianych prawidłowo, z tego ok. 1000000 związanych z realizacją pętli i ok. 500000 z realizacją instrukcji warunkowej *if*). Oznacza to, że układ przewidywania skoków jest w stanie niezależnie uwzględniać co najmniej dwa miejsca wystąpienia skoków w kodzie asemblera.

1.2.8 Testowanie opóźnienia i przepustowości przetwarzania rozkazów

Testem mierzącym opóźnienie wykonania wybranych rozkazów może być wykonanie przez procesor sekwencji takich rozkazów (także np. w pętli), gdzie każdy następny rozkaz jako daną wejściową wykorzystuje daną wyjściową rozkazu poprzedzającego (jest to relatywnie łatwe do uzyskania dla operacji arytmetycznych). W przypadku tak jawnie realizowanej zależności danych, a także w ogólnym przypadku, gdy procesor nie może korzystać ze współbieżności przetwarzania, opóźnienie staje się czynnikiem decydującym o wydajności (w praktyce, dla rozbudowanych programów, sytuacja taka zdarza się rzadko – kompilatory optymalizujące starają się usunąć zależności, a same procesory korzystają np. z przekazywania argumentów pomiędzy rozkazami bezpośrednio w ramach potoków wykonania (*operand forwarding*) oraz innych technik optymalizacji wydajności omawianych w dalszej części rozdziału).

Do badania opóźnienia przetwarzania rozkazów wykorzystany będzie fragment kodu, zawierający istotne z punktu widzenia obliczeń technicznych operacje zmiennoprzecinkowe dodawania i mnożenia podwójnej precyzji:

```
  for(i=0;i<1000000;i++)
    a = 1.000001*a+0.000001;
  }
```

Kompilacja z włączoną opcją optymalizacji `-O3` (szerzej o optymalizujących kompilatorach w p. **??**) daje w wyniku następujący kod asemblera (dla kompilatora *gcc*):

```
.L3:
mulsd %xmm0, %xmm1
subl $1, %eax
addsd %xmm2, %xmm1
jne .L3
```

Kompilator (podobne warianty uzyskuje się dla obu stosowanych w pracy kompilatorów *gcc* i *icc*) do wykonywania operacji arytmetycznych zastosował rozkazy wektorowe `mulsd` i `addsd`, operujące na 128-bitowych rejestrach `xmm` (dodatkowo zmodyfikował także sprawdzanie warunku końca pętli, co jednak nie wpływa na badanie przetwarzania rozkazów zmiennoprzecinkowych).

Pomiar czasu wykonania kodu na komputerze wyposażonym w omawiany wcześniej procesor Intel Core i7-4790 (z rdzeniami o architekturze Haswell i nominalną częstotliwością pracy 3.60 GHz) daje wydajność ok. 1 GFlops (10^9 operacji zmiennoprzecinkowych na sekundę). Jak widać jest to wydajność kilkadziesiąt razy mniejsza od maksymalnej nominalnej wydajności pojedynczego rdzenia (obliczonej w p. 1.2.5 jako 57.6 GFlops).

Program uruchomiony jest w postaci jednowątkowego procesu na pojedynczym rdzeniu, co pozwala na osiągnięcie zwiększonej częstotliwości pracy rdzenia, ok. 4 GHz. Wyniki wskazują na wydajność średnią CPI=4 (4 takty na pojedynczą operację – bardziej szczegółowe badania wskazują na opóźnienie 3 takty dla `addsd` i 5 taktów dla `mulsd`). Co powoduje tak niską wydajność przetwarzania?

Kod napisany jest w taki sposób (co dobrze widać analizując postać asemblera), że w każdej operacji zmiennoprzecinkowej wykorzystywany jest ten sam rejestr zawierający modyfikowaną wartość zmiennej *a* (w konkretnym przypadku użytych kompilatorów jest to rejestr `xmm1`). Oznacza to, że kolejny rozkaz zmiennoprzecinkowy w kodzie nie może zostać wykonany, dopóki nie zostanie zakończony rozkaz poprzedzający. Procesor/rdzeń nie jest w stanie w pełni ujawnić swoich możliwości przetwarzania związanych z technikami ukrywania opóźnienia (*latency hiding*): przetwarzaniem potokowym i wieloma potokami przetwarzania rozkazów.

W celu wykrycia maksymalnej wydajności (przepustowości) przetwarzania pojedynczego rdzenia, testowy kod jest modyfikowany:

```
for(i=0;i<1000000;i++)
  a = 1.000001*a+0.000001;
  b = 1.000001*b+0.000001;
  c = 1.000001*c+0.000001;
  // itd. dla dalszych zmiennych
}
```

Dodawanie kolejnych zmiennych w kodzie pozwala na wykorzystanie większej liczby potoków oraz sprawniejsze przetwarzanie potokowe, kiedy każdy potok może współbieżnie przetwarzać nie powiązane ze sobą rozkazy dotyczące różnych zmiennych. Dodanie pojedynczej zmiennej początkowo zwiększa wydajność przetwarzania o ok. 1 Gflops (do 5 zmiennych). Następnie przyrosty wydajności są już mniejsze, osiągając maksymalną wydajność ok. 7.9 GFlops dla 9 zmiennych.

Wydajność taka wciąż jest daleka od teoretycznych możliwości rdzenia. Jedną z wad dotychczasowego rozwiązania jest użycie odrębnych rozkazów dodawania i mnożenia, podczas gdy rdzeń potrafi wykonać połączone dodawanie i mnożenie (*FMA, fused multiply-add*) w takiej samej liczbie taktów jak każdą z pojedynczych operacji. Zamiana odrębnych rozkazów na pojedynczy połączony rozkaz następuje po przekazaniu do kompilatora opcji jawnie wskazującej na typ architektury rdzenia, w tym wypadku opcji: `-march=core-avx2`. Po zastosowaniu opcji wydajność wzrasta do ok. 15.7 GFlops. Po uwzględnieniu liczby taktów zegara w czasie wykonania programu, uzyskuje się wydajność ok. 4 operacji zmiennoprzecinkowych na pojedynczy takt (dokładnie 2 operacji FMA, co odpowiada 4 klasycznym operacjom arytmetycznym). Miara CPI (uśrednionego, odpowiadającego przepustowości) dla operacji FMA na pojedynczym argumencie podwójnej precyzji wynosi więc 0.5.

Kolejnym brakiem powstałego kodu jest działanie skalarne. Wprawdzie w asemblerze pojawiają się rejestry i instrukcje wektorowe, ale w każdym rejestrze 128-bitowym znajduje się tylko jedna liczba podwójnej precyzji i operacja na takim rejestrze jest efektywnie tylko jedna operacją zmiennoprzecinkową. W celu umożliwienia kompilatorowi pełnego wykorzystania możliwości przetwarzania wektorowego konieczne jest dokonanie dalszych modyfikacji kodu. Zamiast pojedynczych zmiennych wykorzystane zostają małe tablice o rozmiarze 16:

```
for(i=0;i<1000000;i++)
```

```
    for(k=0; k<16; k++){
      a_tab[k] = 1.000001*a_tab[k]+0.000001;
      b_tab[k] = 1.000001*b_tab[k]+0.000001;
      c_tab[k] = 1.000001*c_tab[k]+0.000001;
      // itd. dla dalszych zmiennych
    }
  }
```

Już użycie pojedynczej tablicy zwiększa wydajność przetwarzania do ok. 25 Gflops. Składa się na to pełne wykorzystanie czterech 256-bitowych rejestrów AVX do przechowywania całej tablicy, pozwalające rdzeniowi na kończenie 3 skalarnych operacji FMA na pojedynczy takt. Użycie dwóch tablic zwiększa wydajność do ok. 50 GFlops, a najwyższą praktycznie uzyskiwaną wydajność 63 GFlops zapewnia wykorzystanie 3 tablic (dokładnie jak w przedstawionym kodzie), odpowiadające 16 skalarnym operacjom arytmetycznym (8 operacjom FMA) w pojedynczym takcie.

Obliczona wcześniej nominalna maksymalna wydajność pojedynczego rdzenia analizowanego procesora, obliczana na podstawie danych producenta, wynosi 57.6 GFlops (3.6 GHz x 16 operacji zmiennoprzecinkowych w pojedynczym takcie, jako efekt wykorzystania 2 potoków 256-bitowych rozkazów wektorowych FMA). Zwiększenie do wartości uzyskanej eksperymentalnie (dla trzech tablic i przetwarzania wektorowego) związane jest z podwyższeniem częstotliwości przetwarzania do ok. 4 GHz, możliwym w przypadku użycia wyłącznie jednego rdzenia.

Uruchomienie powyższego kodu w postaci wielowątkowej (np. z wykorzystaniem biblioteki wątków POSIX, *pthreads*) pozwala osiągnąć wydajność ok. 240 GFlops (przy częstotliwości pracy ok. 3,8 GHz).

W obu przypadkach, jedno i wielowątkowym, wydajność przetwarzania uzyskana eksperymentalnie jest zbliżona (z dokładnością do ok. 2%) do maksymalnej wydajności 16 operacji zmiennoprzecinkowych kończonych w pojedynczym takcie przez pojedynczy rdzeń procesora. W ostateczności obliczona średnia miara CPI dla jednego rdzenia związana z pojedynczym rozkazem FMA na 256-bitowym rejestrze wektorowym osiąga optymalną wartość teoretyczną 0.5 (IPC równe 2).

Warto jeszcze zwrócić uwagę na płynący z powyższego przykładu wniosek o relatywnym znaczeniu współczynnika CPI, dotyczący nie tylko różnicy między rozkazami skalarnymi i wektorowymi. Obserwacja, że kończenie w każdym takcie jednej operacji skalarnej (CPI=1) oznacza kilkukrotnie niższą wydajność niż kończenie jednej pełnowartościowej operacji wektorowej (CPI także równe jeden), jest wnioskiem słusznym i oczywistym. Mniej oczywisty jest fakt, że kompilator może użyć operacji (potoków) przetwarzania wektorowego w przypadku nie w pełni wykorzystanych rejestrów wektorowych. Wtedy wartość współczynnika CPI (nawet uwzględniając, że dotyczy rozkazów wektorowych) ponownie nie odpowiada rzeczywistej wydajności programu – istotnym staje się obsadzenie rejestrów użytecznymi danymi programu i efektywna liczba użytecznych operacji kodu źródłowego realizowanych w pojedynczym rozkazie wektorowym.

"Ciśnienie na rejestry" i rozdzielanie pętli

Ciekawym faktem związanym z przedstawionym w poprzednim punkcie kodem jest gwałtowne zmniejszenie wydajności przetwarzania w przypadku użycia czterech tablic.

```
for(i=0;i<1000000;i++)
  for(k=0; k<16; k++){
    a_tab[k] = 1.000001*a_tab[k]+0.000001;
    b_tab[k] = 1.000001*b_tab[k]+0.000001;
    c_tab[k] = 1.000001*c_tab[k]+0.000001;
    d_tab[k] = 1.000001*d_tab[k]+0.000001;
```

```
    }
  }
```

Kod asemblera dla trzech tablic wygląda następująco (tym razem dla kompilatora *icc*):

```
..B1.6:
        vfmadd213pd %ymm0, %ymm13, %ymm12
        incl        %eax
        vfmadd213pd %ymm0, %ymm13, %ymm11
        vfmadd213pd %ymm0, %ymm13, %ymm10
        vfmadd213pd %ymm0, %ymm13, %ymm9
        vfmadd213pd %ymm0, %ymm13, %ymm8
        vfmadd213pd %ymm0, %ymm13, %ymm7
        vfmadd213pd %ymm0, %ymm13, %ymm6
        vfmadd213pd %ymm0, %ymm13, %ymm5
        vfmadd213pd %ymm0, %ymm13, %ymm4
        vfmadd213pd %ymm0, %ymm13, %ymm3
        vfmadd213pd %ymm0, %ymm13, %ymm2
        vfmadd213pd %ymm0, %ymm13, %ymm1
        cmpl        $10000000, %eax
        jb          ..B1.6
```

Podczas gdy dla czterech tablic otrzymuje się:

```
..B1.7:
        vmovupd     64(%rsp,%rdx,8), %ymm2
        vmovupd     192(%rsp,%rdx,8), %ymm3
        vmovupd     320(%rsp,%rdx,8), %ymm4
        vmovupd     448(%rsp,%rdx,8), %ymm5
        vfmadd213pd %ymm0, %ymm1, %ymm2
        vfmadd213pd %ymm0, %ymm1, %ymm3
        vfmadd213pd %ymm0, %ymm1, %ymm4
        vfmadd213pd %ymm0, %ymm1, %ymm5
        vmovupd     %ymm2, 64(%rsp,%rdx,8)
        vmovupd     %ymm3, 192(%rsp,%rdx,8)
        vmovupd     %ymm4, 320(%rsp,%rdx,8)
        vmovupd     %ymm5, 448(%rsp,%rdx,8)
        addq        $4, %rdx
        cmpq        $16, %rdx
        jb          ..B1.7
```

W drugim przypadku, kompilatorowi brakuje rejestrów, żeby w nich umieścić wszystkie zmienne występujące w pojedynczej iteracji, i w efekcie, zamiast efektywnego przetwarzania z wykorzystaniem wyłącznie rejestrów, procesor realizuje w każdej iteracji 8 dostępów do pamięci (za pomocą wektorowych wariantów rozkazu *mov*), co nawet w przypadku wyłącznego korzystania z pamięci podręcznej najbliższej rdzeniowi, znacznie spowalnia wykonanie programu.

Powyższe negatywne zjawisko, użycia wewnątrz pętli zbyt wielu zmiennych, uniemożliwiające efektywne korzystanie z rejestrów, zwane jest ”ciśnieniem na rejestry” (*register pressure*). Unikanie ”ciśnienia na rejestry” jest jednym z wskazań przy tworzeniu wysoko wydajnego kodu.

Dla konkretnego rozważanego przykładu, techniką optymalizacji pozwalającą na likwidację "ciśnienia na rejestry" i osiągnięcie wyższej wydajności jest rozdzielenie pętli (*loop fission*). Przeprowadzenie rozdzielenia pętli z indeksem i na dwie odrębne pętle:

```
for(i=0;i<1000000;i++){
  for(k=0; k<16; k++){
    a_tab[k] = 1.000001*a_tab[k]+0.000001;
    b_tab[k] = 1.000001*b_tab[k]+0.000001;
  }
}
for(i=0;i<1000000;i++){
  for(k=0; k<16; k++){
    c_tab[k] = 1.000001*c_tab[k]+0.000001;
    d_tab[k] = 1.000001*d_tab[k]+0.000001;
  }
}
```

nie powoduje zmiany wyniku (rozdzielane obliczenia są od siebie całkowicie niezależne), a pozwala na podniesienie wydajności z ok. 28 GFlops (kod z czterema tablicami w każdej iteracji wewnętrznej pętli) do ok. 50 GFlops (kod z dwiema tablicami w każdej wewnętrznej iteracji). W przypadku sześciu tablic, *loop fission* pozwala na zwiększenie wydajności z ok. 30 GFlops (wszystkie sześć tablic w jednej pętli z indeksem *k*) do ok. 63 GFlops po rozdzieleniu na dwie podwójne pętle.

"Rozciąganie" tablic dla optymalizacji przetwarzania wektorowego

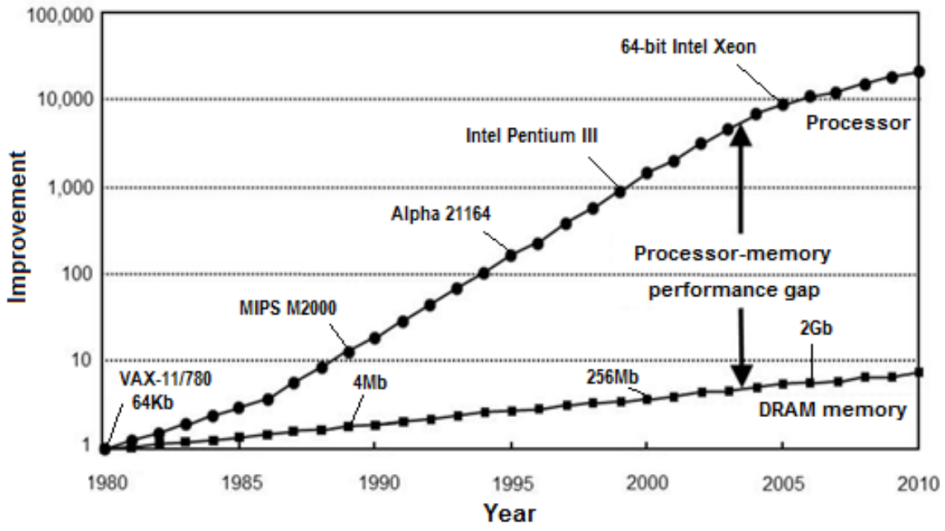
Wyobraźmy sobie sytuację, kiedy z wymagań aplikacji wynika, że w każdej z tablic `a_tab`, `b_tab` i `c_tab` mamy do przechowania 15 egzemplarzy danych, na których mamy wykonać operacje, takie jak w dotychczasowym kodzie. Naturalne zaprojektowanie kodu z rozmiarem tablic 15 i taką samą liczbą operacji w pętli wewnętrznej, powoduje zmniejszenie wydajności przetwarzania do ok. 75 GFlops. Analiza kodu asemblera wyprodukowanego przez kompilator (*icc*) pokazuje, że zastosował on, podobnie jak w przypadku ciśnienia na rejestry, dostępy do pamięci, zamiast czystego przetwarzania z użyciem rejestrów (dodatkowo zamiast wyłącznie rejestrów 256-bitowych, użył dodatkowo rejestrów 128-bitowych wypełnionych tylko pojedynczymi liczbami podwójnej precyzji).

W celu przywrócenia przetwarzania zbliżonego do optymalnego można w tym momencie zastosować technikę rozciągania (rozpychania) tablic (*array padding*), omówioną w p. 1.1.1. Będzie ona polegać na użyciu tablic większych niż wymaga tego aplikacja, tak aby umożliwić kompilatorowi i sprzętowi działanie prowadzące do wyższej wydajności.

Zastosowanie rozciągania tablic jest w tym przypadku niezwykle proste. Należy zaalokować tablice o rozmiarze 16 i zwiększyć liczbę iteracji w pętli wewnętrznej także do 16. Oznacza to wykonywanie operacji także dla 16-go elementu w tablicach, który dobrze jest zainicjować (np. wartością zero), tak aby uniknąć ewentualnych (choć mało prawdopodobnych) problemów przetwarzania wartości nietypowych (np. NaN).

Po kompilacji i uruchomieniu okazuje się, że czas wykonania programu uległ znacznemu skróceniu, mimo że w całym programie wykonywana jest większa liczba operacji niż przed optymalizacją.

W konsekwencji konieczne jest zmodyfikowanie sposobu liczenia wydajności. Mimo wykonywania 16 iteracji w pętli wewnętrznej, efektywna praca na potrzeby aplikacji dotyczy tylko 15 iteracji i 15 elementów każdej z tablic. Co oznacza, że praca sprzętu z wydajnością ok. 240 GFlops (uzyskana w układzie optymalnym), na potrzeby aplikacji daje tylko ok. 227 GFlops (czyli ok. 240*15/16). Ta ostatnia wartość jest przyjmowana jako efektywna wydajność, zgodnie z konwencją przyjętą w książce.



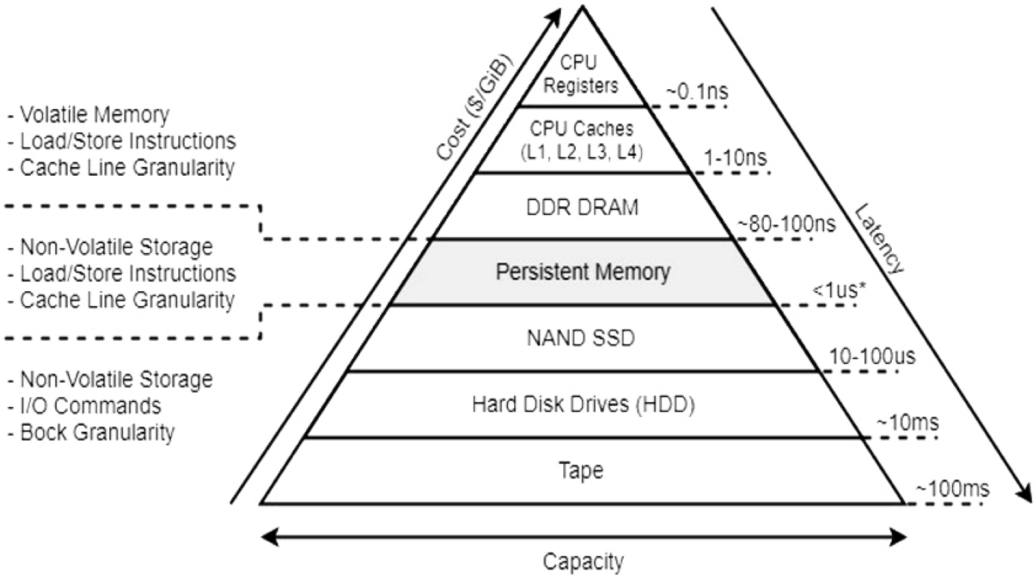
Rysunek 1.14: Przyrost wydajności procesorów (rdzeni) i modułów pamięci DRAM w latach 1980-2010 [źródło: Wikipedia].

1.3 Układ pamięci

Pojedynczy wątek pracujący na pojedynczym rdzeniu współczesnego mikroprocesora realizując dostępy do zmiennych korzysta z hierarchii poziomów pamięci. Zmiennej w kodzie źródłowym odpowiada nazwany obszar pamięci operacyjnej (pamięci głównej – *primary storage, main memory*), najczęściej fizycznie realizowanej w technologii DRAM (*dynamic random access memory*), stąd dalej często używane będzie określenie pamięć DRAM na oznaczenie pamięci głównej. W rozkazach asemblera jako argumenty, obok zawartości rejestrów i argumentów bezpośrednich, używane są adresy zmiennych w pamięci operacyjnej (bezpośrednio zapisane w rejestrach lub obliczone na podstawie zawartości rejestrów, w złożonych trybach adresowania).

Od wielu lat obserwowane jest zjawisko znacznie wolniejszego przyrostu wydajności modułów pamięci DRAM w stosunku do wydajności procesorów (rdzeni), tzw. *memory wall*. Szczególnie jest to widoczne w odniesieniu do opóźnienia jako miary wydajności, bez uwzględnienia szeregu technik ukrywania opóźnienia przy dostępie do pamięci (*memory latency hiding*), o których będzie mowa w dalszej części książki. Zjawisko *memory wall* ilustruje rys. 1.14, pokazujacy jak od początku lat 80-tych XX wieku, przyjętych jako punkt odniesienia, różnica w wydajności między procesorami CPU a modułami DRAM powiększyła się ponad tysiąc razy.

Zjawisko *memory wall* stało się motywacją do zastosowania pośrednich poziomów pamięci, z wykorzystaniem szybszej technologii SRAM (*static random access memory*). Powstała w ten sposób hierarchia pamięci, począwszy od rejestrów, poprzez pośrednie poziomy pamięci podręcznej (*cache memory*), najczęściej realizowane właśnie w technologii SRAM, aż do pamięci głównej DRAM, a nawet, poprzez mechanizm pamięci wirtualnej, do pamięci zewnętrznej (*external memory*). Każdy kolejny poziom w tej hierarchi charakteryzuję się niższą wydajnością, ale w zamian za to także mniejszym kosztem jednostkowym (na pojedynczy przechowywany bajt) i w związku z tym większą pojemnością efektywnie umieszczaną w układach procesora i całego komputera. Ilustruje to rys. 1.15, na którym poza poziomami pamięci wykorzystywanymi do przechowywania danych w trakcie programu znajdują się także pamięci stosowane wyłącznie do archiwizacji danych i nowy proponowany poziom, oznaczany jako *Persistent memory*, realizowany np. w jednym z rozwijanych obecnie wariantów technologii NVRAM (NRAM, MRAM itp.). Ilustracja, poza zaznaczeniem pojemności (*capacity*), kosztu (*cost*) i czasu dostępu do



Rysunek 1.15: Hierarchia pamięci współczesnych systemów komputerowych [źródło: Wikipedia].

danych (*latency*) dla każdego poziomu pamięci, wskazuje na różnice w trwałości zapisanych danych (pamięci ulotne, *volatile*, i nieulotne, *non-volatile*), w sposobie dostępu (za pomocą rozkazów procesora, *load/store instructions* i poleceń systemu operacyjnego, *I/O commands*), a także w jednostkowym rozmiarze danych (*granularity*) przy praktycznej realizacji dostępu (dostęp do linii pamięci podręcznej, *cache line* i do bloku pamięci zewnętrznej). Największe pojemności oferują: pamięć zewnętrzna (drugiego rzędu, *secondary storage*), najczęściej występująca jako twarde dyski HDD lub układy SSD[6], gdzie pojemności sięgają terabajtów pamięci, oraz taśmy magnetyczne, o pojemnościach rzędu petabajtów.

We współczesnych procesorach stosuje się kilka (najczęściej 2 lub 3, rzadziej 4) poziomy pamięci podręcznej (poziom czwarty bywa wykonany w technologi eDRAM). W pamięciach podręcznych, do których nie ma bezpośredniego dostępu z poziomu kodu źródłowego w standardowych językach programowania, przechowuje się kopie wartości zmiennych z pamięci głównej. Zasadą organizacji pamięci podręcznej jest umieszczanie, w miarę oddalania się od potoków przetwarzania, kolejnych poziomów pamięci (L1, L2, L3 - *level 1, level 2, level 3*), z których każdy następny charakteryzuje się mniejszą szybkością działania[7].

W ramach omawiania hierarchii pamięci w niniejszej książce, uwzględnione będą tylko pamięć główna i pamięć podręczna. Pamięć zewnętrzna jest zazwyczaj o co najmniej jeden rząd wielkości wolniejsza od pamięci DRAM, stąd przy optymalizacji wydajności należy unikać konieczności jej użycia. Istnieje szereg aplikacji, w których korzystanie z pamięci zewnętrznej jest jednak konieczne, np. ze względu na rozmiar używanych struktur danych. W takich wypadkach, przy optymalizacji wydajności przydatne mogą być niektóre z technik opisywanych poniżej dla pamięci DRAM i pamięci podręcznych (np. zwiększanie lokalności odniesień w programie), często też optymalizacja będzie istotnie zależna od

[6]W dalszej części ze względu na popularność twardych dysków, określenie to będzie czasami zamiennie uzywane z określeniem pamięć zewnętrzna.

[7]Mimo oznaczania kolejnych poziomów pamięci podręcznej symbolami L1, L2, L3, co można przyjąć za hierarchię pamięci od najniższego poziomu do najwyższego, funkcjonuje także konwencja uznawania pamięci L1 za poziom najwyższy, a kolejne za coraz niższe. W niniejszej książce przyjęta jest konwencja określania poziomów jako znajdujących się bliżej lub dalej od potoków przetwarzania (z L1 jako poziomem najbliższym). Ostatni poziom przed pamięcią DRAM (zazwyczaj L3, ale bywa także L2 lub L4), oznaczany często jako LLC (*last level cache)*, jest w takim ujęciu poziomem najdalszym (nazywany będzie dalej także poziomem ostatnim).

specyfiki aplikacji (np. od rodzaju konkretnych struktur danych użytych do przechowywania zmiennych aplikacji). Specjalne algorytmy, nieopisywane w niniejszej pracy, korzystające z pamięci zewnętrznej (jawnie, bez pośrednictwa mechanizmu pamięci wirtualnej, i z założenia w sposób dążący do optymalnego) nazywane są algorytmami *out-of-core* (*out-of-core algorithms*).

1.3.1 Pamięć wirtualna

Pierwszym z mechanizmów, który zostanie omówiony w kontekście projektowania i optymalizacji korzystania z danych w pamięci operacyjnej, jest mechanizm pamięci wirtualnej ze stronicowaniem, stosowany przez praktycznie wszystkie współczesne systemy operacyjne ogólnego przeznaczenia.

Przykładowy rozkaz dostępu do pamięci, zapisany w języku assemblera zgodnie z omówionymi wcześniej konwencjami dla procesorów z rodziny $x86$, można zapisać następująco:

```
mov -4(%ebx,%ecx,2), %eax
```

Oznacza on pobranie wartości zmiennej przechowywanej w pamięci, w komórkach o adresie początkowym obliczonym na podstawie zawartości rejestrów `%ebx` oraz `%ecx`, i zapisanie jej w rejestrze `%eax` (liczbę pobranych bajtów określa dodatkowa litera dodana na końcu nazwy rozkazu, np. `movl` oznacza pobranie 4 bajtów)[8]. Obliczony adres jest adresem wirtualnym, mieszczącym się w zakresie charakteryzującym dany procesor i tryb jego pracy. W przypadku współczesnych procesorów rodziny $x86$ podstawowy zakres wynosi od 0 do $2^n - 1$, z jednostką adresowania w postaci pojedynczego bajtu i n równym 32 lub 64, w zależności od typu rejestrów używanych do obliczania adresu (w powyższym przykładzie użyto rejestrów 32-bitowych). W praktyce zakres może być modyfikowany, a procesor może udostępniać kilka zakresów, np. poprzez sztuczne ograniczenie przestrzeni adresowej dla procesorów 64-bitowych.

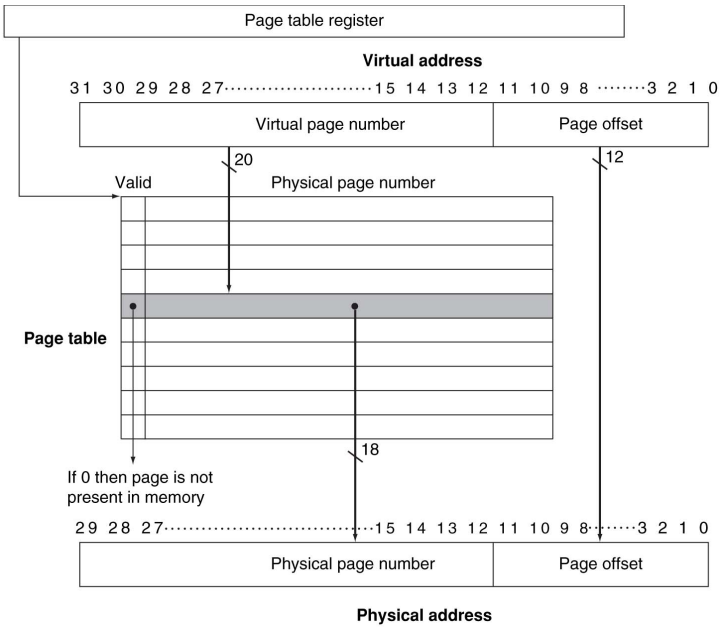
Rozmiar pamięci DRAM współczesnych procesorów jest najczęściej mniejszy niż dopuszczalne wirtualne maksimum[9]. W mechanizmie pamięci wirtualnej ze stronicowaniem, całość przestrzeni adresowej (pamięci wirtualnej) jest dzielona na strony o określonym rozmiarze (system operacyjny pozwala na wybór rozmiaru strony, z najczęściej stosowaną wartością 4 kB). Pojedynczej stronie odpowiada pojedyncza ramka w pamięci DRAM, oznaczająca układ fizycznie przechowujący dane. W pamięci DRAM nie jest przechowywana cała przestrzeń adresowa procesu (wykonywanego programu), ale tylko wybrane jej strony. Jest to naturalną konsekwencją faktu, że przeciętne programy używają dla swojego kodu i danych tylko drobnego ułamka dostępnej przestrzeni adresowej.

Adres wirtualny jest tłumaczony na adres fizyczny związany z konkretną lokalizacją w modułach pamięci DRAM. Obliczenie adresu fizycznego składa się z kilku kroków. Adres wirtualny jest podzielony na sekcje o określonej liczbie bitów. W najprostszym przypadku są dwie sekcje: jedna służąca do określenia numeru strony oraz druga dla wewnętrznego adresu w ramach strony. Uzyskany numer strony służy do znalezienia lokalizacji odpowiadającej ramki pamięci DRAM. W tym celu stosowana jest tablica stron (*page table*). Informacja przechowywana w tablicy stron wskazuje czy zawartość strony przechowywana jest w ramce pamięci DRAM, i jeśli tak, to zawiera także fizyczny adres odpowiadającej ramki 1.16.

W przypadku, gdy program (lub kilka współbieżnie wykonywanych w systemie programów) używa więcej pamięci niż jest dostępne w modułach DRAM, zawartość niektórych stron pamięci wirtualnej jest

[8] Ściśle, adres obliczony zgodnie z regułami adresowania pośredniego, stosowanego często przy dostępie do tablic, dla zapisu *disp(base, index, scale)* (w przyjętej w pracy notacji assemblera $x86$) wynosi: baza (*base*) adresu (zawarta w odpowiednim rejestrze) + indeks (*index*) w tablicy zapisany także w rejestrze * odstęp pomiędzy kolejnymi elementami w tablicy (*scale*), równy 1, 2, 4 lub 8 + przesunięcie *disp*, *adres* = *base* + *index***scale* + *disp*. Specjalne potoki wykonania służące do obliczania adresów w powyższej formie są udostępniane także dla rozkazów, w których nie dokonuje się dostępu do pamięci (np. `lea`, *load effective address*, obliczający adres i zapisujący go w rejestrze). Rozkazy te bywają wykorzystywane przez optymalizujace kompilatory do wykonywania czysto arytmetycznych operacji, np. zawierających mnożenie przez 2, 4 lub 8.

[9] Historycznie, gdy dominowały procesory 32-bitowe, oznaczało to mniej niż 4GB pamięci (co dzisiaj jest coraz rzadziej spotykane), natomiast granica dla procesorów 64-bitowych 16384 PB (petabajtów, 10^{15}B) nie jest dzisiaj praktycznie osiągalna.



Rysunek 1.16: Wykorzystanie tablicy stron do obsługi pamięci wirtualnej [źródło: Wikipedia].

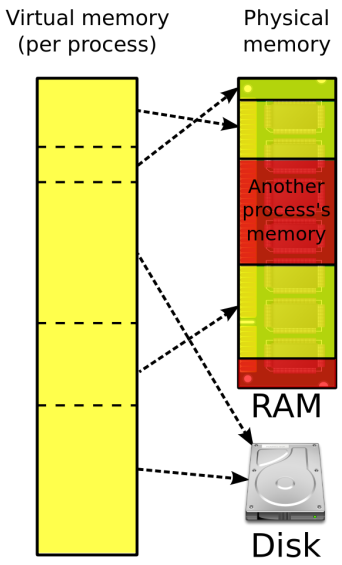
usuwana z pamięci głównej i zapisywana w pamięci zewnętrznej. Gdy zawartość strony jest przechowywana na twardym dysku, dostęp do żądanego adresu wirtualnego wymaga najpierw pobrania zawartości strony do pamięci DRAM, a dopiero potem możliwy jest dostęp do konkretnej komórki pamięci.

Rysunek 1.17 przedstawia prostą ilustrację mechanizmu działania pamięci wirtualnej. Część stron z pamięci wirtualnej wykonywanego programu (procesu) przechowywana jest w pamięci DRAM, a część na twardym dysku. Rysunek pokazuje także, jak pamięć DRAM wykorzystywana jest jednocześnie przez inny, współbieżny proces – mechanizm pamięci wirtualnej jest wygodnym sposobem zapewnienia bezpieczeństwa wykonania wielu współbieżnych procesów w ramach systemów wielozadaniowych.

Zarządzaniem pamięcią wirtualną, w szczególności wykorzystaniem pamięci zewnętrznej, drugiego rzędu, do przechowywania stron pamięci wirtualnej usuniętych z pamięci głównej, zajmuje się system operacyjny. W przypadku kiedy procesor chce uzyskać dostęp do zmiennej i okazuje się, że odpowiadajacy jej obszar pamięci nie ma przydzielonej ramki w pamięci głównej, zgłaszany jest błąd strony (*page fault*) i uruchamiana procedura jego obsługi. Często błąd strony oznacza tylko konieczność przydzielenia ramki stronie pamięci wirtualnej, bez konieczności odwoływania się do pamięci zewnętrznej, np. kiedy strona nie znajdowała się dotąd w pamięci głównej, a pamięć posiada wolne zasoby. Taki mniejszy błąd strony (*minor page fault*), typowy dla początkowego okresu realizacji programu oraz dla dynamicznej alokacji pamięci, nie stanowi znaczącego narzutu na czas wykonania programu. Bardzo kosztowny jest natomiast większy błąd pamięci (*major page fault*) związany z przeładowaniem zawartości strony pamięci pomiędzy pamięcią zewnętrzną, a pamięcią główną (*page swap*).

Szczególnie niekorzystnym zjawiskiem przy wykonaniu programu są powtarzające się z dużą częstotliwością żądania dostępu do zbyt wielu stron pamięci, w stosunku do rozmiaru pamięci DRAM, co powoduje nieustanne błędy stron i przeładowania zawartości pomiędzy pamiecią główną i zewnętrzną, określane jako szamotanie (*thrashing*). Szamotanie może być konsekwencją zbyt dużej liczby stron wykorzystywanych bieżąco przez pojedynczy program, może także być związane ze zbyt dużą liczbą uruchomionych programów (z których każdy wymaga częstego dostępu do określonego zbioru stron pamięci wirtualnej).

System operacyjny, w ramach sterowania mechanizmem pamięci wirtualnej, zarządza także tablicą

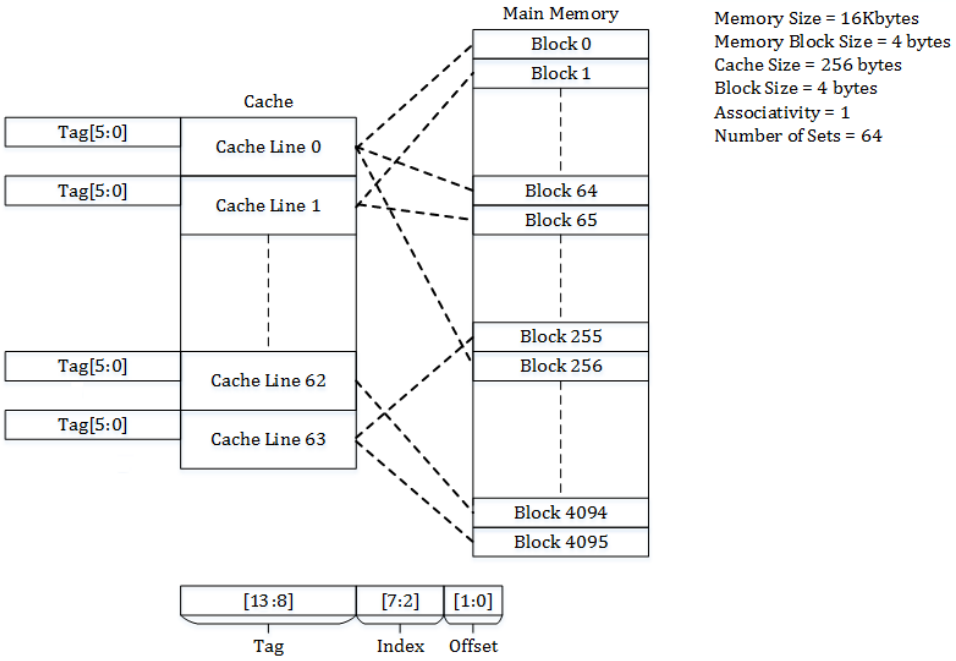


Rysunek 1.17: Mechanizm działania pamięci wirtualnej, odwzorowanie sekwencji stron pamięci wirtualnej procesu w zbiór ramek pamięci DRAM, połączone z przechowywaniem zawartości niektórych stron na twardym dysku [źródło: Wikipedia].

stron. W praktyce współczesne systemy operacyjne stosują wiele tablic stron, np. odrębne dla każdego procesu lub dla większych obszarów pamięci zwanych segmentami (wtedy adres wirtualny zawiera więcej sekcji, np. sekcję dla katalogu tablic stron). Tablice stron przechowywane są w pamięci DRAM, a także w szybkiej pamięci podręcznej TLB (opisywanej już przy omawianiu architektur rdzeni mikroprocesorów). W skrajnym przypadku bardzo obszerna tablica stron może także, w ramach pamięci wirtualnej, znajdować się częściowo w pamięci zewnętrznej.

Testowanie mechanizmu podmiany stron

Prostym testem stosowania przez system operacyjny podmiany stron (*page swapping*), który należy realizować jednak z ostrożnością, gdyż może prowadzić do zawieszenia sytemu, jest zaalokowanie i specjalne wykorzystywanie w programie tablicy o rozmiarze zbliżonym lub lekko przekraczającym rozmiar pamięci DRAM komputera. Specjalne wykorzystanie, powodujące podmiany stron, można uzyskać wielokrotnie realizując dostęp do wszystich elementów tablicy, za pomocą odpowiednio skonstruowanych pętli. Śledzenie podmian stron pamięci wirtualnej można osiągnąć na różne sposoby. Można wykorzystać jedną z procedur systemowych (np. *getrusage* w Linuxie), zwracających dotychczasową liczbę błędów stron w dowolnym momencie wykonania programu. W Linuxie można także wykorzystać narzędzie systemowe *time* (*/usr/bin/time*, w odróżnieniu do polecenia powłoki *time*), które zwraca informacje o błędach stron dla całego programu, przekazanego jako argument polecenia. Kolejnym sposobem może być śledzenie aktywności systemu, np. za pomocą polecenia *top* (wymaga to odpowiednio długiego czasu wykonania programu, nawet przy zadaniu częstości próbkowania dla polecenia *top* co jedną sekundę, za pomocą *top -d 1*). Pojawienie się narzutu związanego z podmianą stron widoczne będzie poprzez zmniejszenie procentu czasu CPU przydzielanego wykonywanemu procesowi oraz pojawienie się, ze znacznym udziałem czasu CPU, procesów odpowiedzialnych za obsługę błędu strony.



Rysunek 1.18: Mechanizm umieszczania zawartości komórek pamięci głównej w liniach pamięci podręcznej i związany z nim sposób wykorzystania kolejnych bitów adresu, dla pamięci odwzorowanej bezpośrednio [źródło: Wikipedia].

1.3.2 Pamięć podręczna

Podstawowy mechanizm działania pamięci podręcznej i lokalność odniesień

Obliczony w ramach mechanizmu pamięci wirtualnej adres fizyczny służy do uzyskania dostępu do zawartości odpowiadającej mu komórki pamięci DRAM[10]. W praktyce dostęp taki jest zawsze realizowany za pośrednictwem pamieci podręcznej, przechowującej kopie danych. Opis mechanizmów funkcjonowania pamięci podręcznej rozpoczniemy od sytuacji występowania tylko pojedynczego jej poziomu, rozszerzając go następnie na przypadek kilku poziomów.

Pamięć podręczna składa się z linii. Dla różnych poziomów pamięci rozmiar pojedynczej linii może się różnić, przy czym najczęściej spotyka się linie o rozmiarze 64 lub 128 bajtów (co oznacza kilka zmiennych podwójnej precyzji lub kilkanaście zmiennych całkowitych i pojedynczej precyzji). Pamięć DRAM podzielona jest na bloki, każdy o rozmiarze odpowiadającym rozmiarowi linii pamięci podręcznej. W najprostszym modelu, pamięci odwzorowanej bezpośrednio (*direct mapped*), każdy blok pamięci DRAM jest przyporządkowany do pojedynczej linii pamięci podręcznej, co oznacza, że kopia zawartości bloku może znajdować się tylko w tej linii. Kolejny następujący po nim blok jest przyporządkowany kolejnej linii, itd. Całkowity rozmiar pamięci podręcznej jest znacznie mniejszy od rozmiaru pamięci DRAM, stąd w pewnym momencie kolejny blok zostaje przyporządkowany pierwszej linii, następujący po nim drugiej itd. W końcowym efekcie, pojedynczej linii pamięci podręcznej odpowiada wiele bloków pamięci DRAM.

Rysunek 1.18 pokazuje zasadę umieszczania danych w pamięci podręcznej dla hipotetycznej sytuacji, kiedy pamięć operacyjna adresowana jest za pomocą 14 bitów (jej rozmiar wynosi więc 16 kilo-

[10]W praktyce dostęp do pamięci odbywa się często do pewnego stopnia równolegle, poprzez zastosowanie odpowiednich technik, z translacją adresu z wirtualnego na fizyczny

bajtów), a pamięć podręczna składa się z 64 linii o rozmiarze 4 bajtów. Pamięć główna dzielona jest więc na 4-bajtowe bloki, pierwsze 64 bloki (o numerach od 0 do 63) odwzorowane są w kolejne 64 linie pamięci podręcznej, po czym blok o numerze 64 odwzorowany jest ponownie w pierwszą linię pamięci podręcznej (linia 0), o numerze 65 w drugą, kolejne bloki w kolejne linie, aż do bloku o numerze 127, a cały proces powtarza się dla każdego zbioru 64 bloków pamięci. Przy rozmiarze 16 kB, całkowita liczba bloków wynosi 4096, czyli 64 sekwencje kolejnych 64 bloków (w jedną linię pamięci podręcznej odwzorowane są więc 64 bloki pamięci głównej).

Adres dowolnej jednobajtowej komórki pamięci składa się z dwóch bitów, które oznaczają miejsce w czterobajtowym bloku pamięci DRAM, a więc także w czterobajtowej linii pamięci podręcznej (*offset*), kolejnych 6 bitów oznaczających numer bloku w zbiorze 64 kolejnych bloków (czyli numer odpowiadającej linii pamięci podręcznej – *index*) oraz ostatnich 6 bitów określających numer zbioru bloków (*tag*). W każdej linii pamięci podręcznej może znajdować się blok z dowolnego 64-blokowego zbioru, numer zbioru (*tag*) musi być więc przechowywany wraz z linią pamięci podręcznej, aby umożliwić sprawdzenie, który blok pamięci głównej jest aktualnie przechowywany w pamięci podręcznej.

Mechanizm działania pamięci podręcznej jest następujący. Procesor, po wygenerowaniu adresu zmiennej, do której chce uzyskać dostęp, sprawdza czy w pamięci podręcznej znajduje się aktualna kopia wartości zmiennej[11]. W tym celu oblicza na podstawie odpowiednich bitów adresu (*index*), w której linii może znajdować się kopia bloku pamięci głównej zawierającego zmienną, a następnie na podstawie innych bitów (*tag*) sprawdza czy rzeczywiście w linii znajduje się ten blok. Jeśli aktualna wartość zmiennej jest w pamięci podręcznej (co będziemy określali jako trafienie w pamięci podręcznej, *cache hit*), procesor realizuje szybki dostęp do zmiennej. Jeśli blok zawierający zmienną nie znajduje się w pamięci podręcznej (lub jeśli jego zawartość nie jest aktualna), co określane jest jako chybienie (*cache miss*), zawartość całej linii jest podmieniana, odpowiedni blok pamięci DRAM jest kopiowany do linii pamięci podręcznej, po czym następuje realizacja dostępu do zmiennej w pamięci podręcznej. Załadowany blok pozostaje w pamięci podręcznej, dopóki nie zostanie podmieniony przez inny blok, odwzorowany w tę samą linię.

W oczywisty sposób pierwszy dostęp do każdego bloku pamięci głównej w trakcie działania programu powoduje chybienie w pamięci podręcznej (tzw. **chybienie konieczne**, *compulsory miss*). Jednak kolejne dostępy mogą prowadzić do trafień lub chybień. Opis działania pamięci podręcznej wskazuje na kilka podstawowych z punktu widzenia wydajności faktów:

1. jeśli w trakcie wykonania programu dostęp do pewnej zmiennej następuje wielokrotnie w krótkich odstępach czasu (tzw. **lokalność czasowa**, *temporal locality*), to rośnie prawdopodobieństwo, że kolejne dostępy (poza pierwszym) realizowane będą przy użyciu kopii w pamięci podręcznej, a nie wartości w powolnej pamięci DRAM (warunkiem jest, aby odpowiednia linia pamięci podręcznej zawierająca kopię zmiennej, nie została podmieniona lub zdezaktualizowana pomiędzy kolejnymi dostępami)

2. jeśli w trakcie wykonania programu w krótkim odstępie czasu następują dostępy do różnych zmiennych przechowywanych w tym samym bloku pamięci, odwzorowanym w pojedynczą linię pamięci podręcznej (tzw. **lokalność przestrzenna**, *spatial locality*), to rośnie prawdopodobieństwo, że kolejne dostępy do zmiennych (poza pierwszym dostępem do zmiennej z danego bloku) realizowane będą przy użyciu kopii w pamięci podręcznej, a nie wartości w powolnej pamięci DRAM

3. jeśli następuje chybienie w pamięci podręcznej, to narzut czasowy na obsługę chybienia (*miss penalty*) jest znaczący, ze względu na **konieczność podmiany całej linii pamięci podręcznej**

[11] Znaczenie określenia aktualna kopia wyjaśnione jest przy omawianiu mechanizmów utrzymania spójności pamięci podręcznej w p. **??**.

Skuteczność pamięci podręcznej, jako mechanizmu optymalizacji, zależy od tego jak wiele programów i w jak dużym stopniu wykazuje czasową i przestrzenną lokalność dostępów (odniesień) do pamięci (*locality of reference*). Okazuje się, że zdecydowana większość programów w naturalny sposób posiada te dwie cechy i w praktyce korzysta z istnienia pamięci podręcznych, uzyskując dzięki nim redukcję czasu wykonania.

W praktyce wykorzystuje się szereg dodatkowych szczegółowych mechanizmów funkcjonowania pamięci podręcznych, z których niektóre są omówione w dalszej części rozdziału. Już jednak podstawowy sposób działania pamięci podręcznej implikuje najważniejsze wskazanie dotyczące optymalizacji dostępów do pamięci przy wykonaniu programów na współczesnych systemach komputerowych: **należy zawsze dążyć do maksymalizacji lokalności czasowej i przestrzennej odniesień do pamięci w trakcie wykonania programów**. Należy także zawsze pamiętać o tym, że **dostęp do pamięci DRAM (pobranie lub zapis danych) standardowo dotyczy całej linii pamięci podręcznej, a nie pojedynczej zmiennej**.

Drożność pamięci podręcznej

Mechanizm bezpośrednio odwzorowanej pamięci podręcznej przedstawiony dotychczas ma istotną wadę. Rozważmy prostą pętlę algorytmu obliczania iloczynu skalarnego dwóch wektorów:

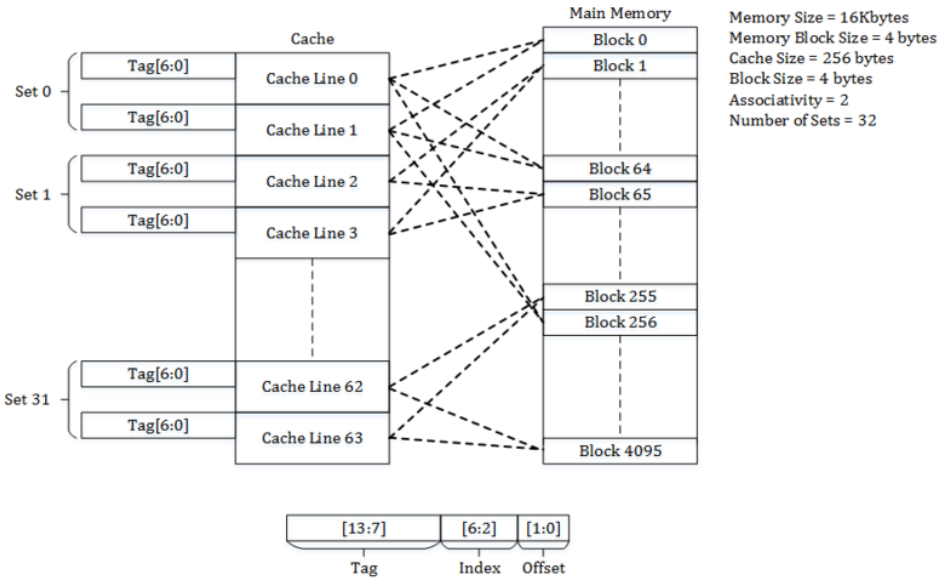
```
for (j=0; j<N; j++) il_skal += b[j] * c[j];
```

Wydaje się, że algorytm będzie realizowany optymalnie, ze względu na dużą lokalność przestrzenną odniesień (skok równy 1 dla każdego z wektorów). Jednak może się zdarzyć, że początki tablic `b` i `c` zostaną odwzorowane w to samo miejsce pamięci podręcznej (dla rys.1.18 mogłoby się tak zdarzyć, gdyby np. tablica `b` zaczynała się w bloku 0, a tablica `c` w bloku 256). Wtedy w pierwszej iteracji, po pobraniu elementu tablicy `b`, przy próbie pobrania elementu tablicy `c` następuje chybienie w pamięci podręcznej i kosztowna czasowo podmiana całej linii. W drugiej iteracji, przy próbie dostępu do drugiego elementu tablicy `b`, ponownie następuje chybienie, mimo że element ten przed chwilą był już w pamięci podręcznej, jednak jego linia została podmieniona przez linię związaną z tablicą `c`. W tej samej drugiej iteracji następuje w chwilę później kolejne chybienie, przy dostępie do drugiego elementu tablicy `c`, który znowu przed chwilą był w pamięci podręcznej, lecz jego linia została podmieniona elementami tablicy `b`. Dzieje się tak dla każdej kolejnej iteracji, więc w efekcie w każdej iteracji pętli występują dwa chybienia w pamięci podręcznej i dwie podmiany linii.

Aby temu zapobiec powszechnie stosuje się tzw. pamięci sekcyjno-skojarzeniowe (*set associative*). Pojedynczy blok pamięci DRAM jest przyporządkowany więcej niż jednej linii pamięci podręcznej, w praktyce najczęściej 2, 4 lub 8 liniom, co nazywane jest pamięcią dwudrożną (*two-way*), czterodrożną (*four-way*) i ośmiodrożną (*eight-way*).

Rysunek 1.19 pokazuje schemat działania pomięci dwudrożnej. Cała pamięć podręczna dzielona jest na zbiory (*set*) po dwie linie w zbiorze (dla pamięci czterodrożnej byłyby to zbiory po cztery linie, dla ośmiodrożnej po osiem linii, itd). Każdy blok w pamięci głównej (o rozmiarze pojedynczej linii pamięci podręcznej) może być przechowywany w jednej z dwóch linii zbioru, w który został odwzorowany. Na rys.1.19 oznacza to np. że blok 0 pamięci głównej może być przechowywany w linii 0 lub linii 1, stanowiących zbiór 0. Z kolei linie 2 i 3 tworzą zbiór 1, w który odwzorowany jest blok 1 pamięci głównej. W przykładowej zilustrowanej pamięci podręcznej znajdują się 32 zbiory (dla 64 linii), co oznacza, że blok 31 pamięci głównej odwzorowany jest w zbiór 31, a blok 32 (podobnie jak blok 64, 96, 128 itd.) w zbiór 0.

Dla przykładu algorytmu mnożenia skalarnego pamięć dwudrożna usuwa niebezpieczeństwo ciągłych podmian linii w pamięci podręcznej, które groziły w przypadku pamięci bezpośrednio odwzorowanej. Nawet jeśli początek tablicy `b` odwzorowany będzie w blok 0, tak samo jak początek tablicy `c`,



Rysunek 1.19: Mechanizm umieszczania zawartości komórek pamięci głównej w liniach pamięci podręcznej i związany z nim sposób wykorzystania kolejnych bitów adresu, dla dwudrożnej pamięci sekcyjno-skojarzeniowej [źródło: Wikipedia].

to poprawnie działający mechanizm podmiany linii w pamięci podręcznej, umieści w trakcie wykonania algorytmu iloczynu skalarnego początkowe wyrazy tablicy `b` w jednej linii zbioru, a początkowe wyrazy tablicy `c` w drugiej.

W kolejnych iteracjach, bloki pamięci głównej zawierające kolejne fragmenty tablic będą umieszczane w kolejnych zbiorach linii pamięci podręcznej. Linie w zbiorach będą wypełniane danymi, na których algorytm dokonuje operacji, aż do momentu dojścia do ostatniego zbioru. W takiej sytuacji wszystkie linie będą wypełnione danymi z tablic. Jeśli rozmiary tablic będą odpowiednio duże, przejście do kolejnej iteracji i do kolejnych elementów tablic, wymusi podmianę linii w pamięci podręcznej. Przyczyną tej podmiany będzie fakt, że rozmiar danych algorytmu przekroczył rozmiar pamięci podręcznej. Podmiana w takiej sytuacji następuje w wyniku chybienia w pamięci podręcznej, wymuszonego przez zbyt małą pojemność pamięci podręcznej. Mówimy wtedy o **chybieniu pojemnościowym** (*capacity miss*).

Pamięć dwudrożna rozwiązuje problem możliwych częstych chybień w przypadku algorytmu iloczynu skalarnego, jednak nie usuwa zagrożenia w przypadku np. algorytmu dodawania wektorów:

```
for (j=0; j<N; j++) a[j] = b[j] + c[j];
```

Jeśli teraz początki wszystkich trzech tablic odwzorowane zostaną w ten sam zbiór dwóch linii, w każdej iteracji będzie dochodziło do chybień w pamięci podręcznej. Już w pierwszej iteracji po pobraniu z pamięci DRAM początkowych wyrazów dwóch tablic – do dwóch linii w pojedynczym zbiorze, próba pobrania wyrazów trzeciej tablicy doprowadzi do konieczności podmiany jednej z linii zbioru. Stanie się tak, mimo że większość pamięci podręcznej jest w tym momencie niewykorzystywana przez algorytm, a więc nie jest przekraczana pojemność pamięci przez dane algorytmu. Taki typ chybienia w pamięci podręcznej, powodującego konieczność podmiany linii na skutek odwzorowania wielu bloków pamięci DRAM w ten sam zbiór linii pamięci podręcznej, nazywany jest chybieniem na skutek konfliktu (*conflict miss*). Takie **chybienia konfliktowe** są niekorzystne, powodują konieczność podmiany linii mimo pozostawania części pamięci podręcznej niewykorzystanej, utrudniają także analizę i optymalizację algorytmów, łatwiejszą w przypadku chybień pojemnościowych.

W przypadku pamięci odwzorowanej bezpośrednio, ryzyko chybień konfliktowych jest relatywnie duże. Maleje ono w przypadku pamięci dwudrożnej, a dalsze zwiększanie drożności prowadzi do coraz mniejszego niebezpieczeństwa wystąpienia chybień konfliktowych. Im wyższa drożność tym mniejsze prawdopodobieństwo, że w konkretnym programie dla pewnego zbioru zmiennych, intensywnie wykorzystywanych i przyporządkowanych temu samemu zbiorowi linii pamięci podręcznej, liczba linii w zbiorze okaże się zbyt mała, co będzie prowadzić do częstych chybień konfliktowych i podmian linii, podczas gdy pozostałe zbiory linii nie będą w pełni wykorzystane.

Całkowita eliminacja chybień konfliktowych następuje w przypadku tzw. pamięci w pełni skojarzeniowej, *fully associative*, dla której każdy blok pamięci może być przechowywany w dowolnej linii pamięci podręcznej. Daje to gwarancję, że zawsze wykorzystywana będzie pełna pojemność pamięci, nie doprowadzając do marnowania zasobów.

Jednak w praktyce tego typu pamięci stosuje się rzadko, zazwyczaj drożność pamięci podręcznej jest ograniczona. Wynika to z badań statystycznych zysku, jaki przynosi zwiększenie drożności dla wydajności dostępów do pamięci. Zwiększając drożność, zmniejsza się częstotliwość chybień konfliktowych i przez to zmniejsza narzut czasowy obsługi chybień na czas dostępu do danych. Jednak jednocześnie zwiększa się koszt obsługi pamięci podręcznej, i to dla każdego dostępu do pamięci, nie tylko w przypadku rzadkich chybień konfliktowych.

Każdy dostęp do pamięci przechodzi przez fazę sprawdzenia, czy dane których dotyczy znajdują się już w pamięci podręcznej. Na podstawie odpowiednich bitów adresu (na rys.1.19 to bity tworzące *index*) znajdowany jest zbiór linii, w które może być odwzorowany blok pamięci zawierający dane. Następnie na podstawie bitów charakteryzujących konkretny blok (bity tworzące *tag* na rys.1.19) sprawdza się czy zawartość danego bloku znajduje się w jednej z linii zbioru (porównując z wartościami *tag* związanymi z każdą linią). Sprawdzenie to powoduje narzut czasowy, który będzie rósł w miarę rosnącej drożności pamięci, liczby linii w pojedynczym zbiorze (skutkuje także zwiększonym wydatkiem energii). Dla pewnej wartości drożności, narzuty te zaczynają przeważać nad zyskami w postaci redukcji liczby chybień konfliktowych.

Fakt ograniczenia drożności pamięci podręcznych ma znaczenie dla optymalizacji wydajności, w szczególności dla algorytmów operujących na tablicach (wektorach, macierzach). Z jednej strony, w przypadku wielu tablic należy sprawdzać czy ich położenie nie prowadzi do chybień konfliktowych. Z drugiej strony, chybienia konfliktowe możliwe są także wtedy, gdy algorytm operuje na małej liczbie tablic, a nawet na pojedynczej tablicy. Dzieje się tak, kiedy dostęp do tablic realizowany jest nie wyraz po wyrazie, ale z pewnym skokiem. Przy pewnych wartościach skoku niebezpieczeństwo chybień konfliktowych rośnie.

Dalsze szczegóły funkcjonowania pamięci podręcznej

Poza omówionymi powyżej podstawowymi mechanizmami, funkcjonowanie pamięci podręcznej obejmuje szereg szczegółowych rozwiązań, takich jak np. działanie nieblokujące, wykonanie potokowe, wielobankowość. Jednym z ważniejszych aspektów jest stosowanie pamięci zawierających się w sobie (*inclusive caches*) i pamięci odrębnych (*exclusive caches*). W pierwszym przypadku, jeśli zawartość bloku pamięci DRAM znajduje się w pamięci bliższej potokom (np. L1), to ten sam blok znajduje się w pamięci kolejnego poziomu (np. L2). Oznacza to, że cała zawartość pamięci bliższej potokom, jest przechowywana także w kolejnej pamięci (czyli w efekcie np. L1 jest zawarta w L2). Pobranie bloku z pamięci DRAM powoduje kolejno wypełnianie linii w L3, L2 i L1.

Inaczej jest dla pamięci odrębnych, których przykładem jest pamięć *victim cache*, służąca do przechowywania podmienionych linii z pamięci bliższej potokom wykonania. Jeśli rolę *victim cache* pełni np. L2, wtedy pobranie z L3 dokonywane jest bezpośrednio do L1, a podmiana w L2 następuję w efekcie usunięcia linii z L1 i jej zapisu do L2. Zawartości L1 i L2 są różne – dostępna pojemność szybkiej

pamięci podręcznej dla programu tym samym rośnie (jest sumą pojemności L1 i L2, a nie wyłącznie pojemnością L2), choć komplikuje się mechanizm dostępu[12].

Innym, istotnym aspektem funkcjonowania pamięci wielodrożnych, jest fakt, że podmiana zawartości w pamięci, w celu pobrania nowego bloku, może dotyczyć dowolnej linii ze zbioru linii, w które odwzorowany jest ten blok. Istnieją różne strategie podmiany linii, niektóre faworyzujące szybkość działania nad maksymalizację liczby trafień, jak np. podmiana losowa lub wykorzystanie prostej kolejki FIFO, inne bardziej złożone, jak np. podmiana linii najdawniej użytej (*LRU, least recently used*) lub najrzadziej używanej (*LFU, least frequently used*). Zagadnienie wyboru i implementacji strategii podmiany, choć niewątpliwie mające znaczenie dla wydajności, nie jest zazwyczaj analizowane w kontekście optymalizacji kodu źródłowego i modelowania wydajności. W niniejszej książce nie są przeprowadzane szczegółowe badania mechanizmów podmiany linii, czasem jednak uwzględniany będzie ich możliwy wpływ na wydajność.

Przykładowym, często spotykanym w praktyce przypadkiem, jest wykonywanie pętli, w której uzyskuje się dostęp do kolejnych elementów jednej lub kilku tablic, czyli jednej lub kilku sekwencji komórek w pamięci DRAM. Wydajność może być w takim przypadku niezależna od drożności pamięci podręcznej, zakładając odpowiednią strategię podmiany linii. Strategia powinna umożliwiać wypełnianie najpierw pierwszych linii w każdym zbiorze linii w pamięci podręcznej, następnie drugich, potem kolejnych, aż do wypełnienia całej pamięci podręcznej. W takiej sytuacji nie występują chybienia konfliktowe, a każde chybienie staje się chybieniem pojemnościowym.

Cel taki udaje się relatywnie prosto zrealizować dla pojedynczej tablicy i pojedynczej sekwencji komórek w pamięci. Opisane wyżej warunki redukcji chybień konfliktowych spełnia każda ze strategii podmiany, za wyjątkiem najprostszej, losowej.

Dla wielu tablic, o różnych długościach i wzorcach dostępu, redukcja chybień konfliktowych może być znacznie trudniejsza. W dalszej części książki, badając wydajność algorytmów, rozważany będzie możliwy wpływ na wydajność różnych możliwych strategii podmiany, także bardziej złożonych od powyższych. W ich świetle szacowany będzie czas wykonania i możliwe optymalizacje kodu.

Pamięć podręczna a kod źródłowy

Każdy z omówionych dotychczas mechanizmów funkcjonowania pamięci podręcznych ma wpływ na wydajność, jednak do ich wykorzystania nie ma jawnych mechanizmów programowania. Już sam fakt istnienia pamięci podręcznej nie jest zaznaczony w praktycznie żadnym z najważniejszych języków programowania. Optymalizacja użycia pamięci podręcznej odbywa się poprzez zmiany kodu źródłowego, co do których przewiduje się, że będą miały wpływ na wydajność (alternatywą jest korzystanie z wybranych rozkazów z listy procesora operujących jawnie na pamięci podręcznej, które jednak także nie dają możliwości sterowania większością aspektów jej funkcjonowania). Im bardziej szczegółowy mechanizm działania pamięci tym trudniej jest optymalizować jego wykorzystanie.

Dodatkowym efektem istnienia szczegółowych mechanizmów funkcjonowania pamięci podręcznej jest zawsze przybliżony charakter przewidywań dotyczących czasu dostępu do zmiennych w programie. Złożoność mechanizmów i szereg interakcji w jakie wchodzą w trakcie wykonania programu powodują także, że czas wykonania jest dla wielu algorytmów różny przy praktycznie każdym wykonaniu programu, i to z różnicami dochodzącymi do kilkudziesięciu procent. W dalszej części książki omawiane będą techniki umożliwiające w pewnych przypadkach uzyskanie, nie tylko minimalizacji czasu obliczeń, ale także stabilności wydajności.

[12] W nowszych generacjach procesorów pojawia także zastosowanie pamięci odrębnej dla poziomu L3, co znacząco wpływa na sposoby analizy i optymalizacji wydajności pamięci podręcznych.

1.3.3 Praktyczne aspekty badania parametrów wydajnościowych pamięci

Realizacja dostępów do pamięci we współczesnych systemach komputerowych obejmuje, poza omówionymi powyżej zasadami funkcjonowania pamięci podręcznej i wirtualnej, cały szereg innych elementów sprzętowych i mechanizmów obsługi. W praktycznych badaniach parametrów wydajnościowych pamięci przeprowadzanych w książce uwzględniane będą wybrane mechanizmy, o istotnym znaczeniu dla wydajności i możliwości optymalizacji działania poprzez modyfikacje kodu źródłowego. Elementami takimi są miedzy innymi wyrównanie zmiennych w pamięci i spekulatywne pobieranie danych przez procesory (pobieranie z wyprzedzeniem, *prefetching*).

Położenie zmiennych w pamięci głównej

Powszechną jednostką adresowania pamięci operacyjnej jest pojedynczy bajt. Jednak większość typów danych (poza znakami z pierwotnego zestawu ASCII i specyficznymi typami zmiennych całkowitych) ma rozmiar pojedynczej zmiennej będący wielokrotnością jednego bajtu. Wiąże się to najczęściej z cechami konkretnej maszyny (jako układu procesor-pamięć), które w przeszłości determinowały długość tzw. słowa maszynowego. Długość słowa w szczególności związana była z typowymi dla danego sprzętu: rozmiarem rejestrów procesora (zwłaszcza przechowujących adresy w pamięci) i szerokością magistrali łączącej procesor z układami pamięci. Pojęcie słowa maszynowego pojawiało się np. w rozkazach procesora, gdzie rozmiar argumentów był wyrażany w wielokrotności słowa (w architekturze *x86* ostatnie litery rozkazu często określają rozmiar argumentu: *w* – słowo (*word*), *dw* – podwójne słowo (*doubleword*), *qw* – poczwórne słowo (*quadword*), gdzie rozmiar słowa wynosi 16 bajtów).

Na skutek rozwoju architektur procesorów, przy jednoczesnej chęci utrzymania kompatybilności listy rozkazów, tradycyjne powiązanie rozmiaru słowa maszynowego z cechami sprzętu zniknęło. Adresy przechowywane są najczęściej w rejestrach 32 lub 64-bitowych, magistrale mają także najczęściej 64 bity szerokości. Powyższe cechy sprzętu mają w oczywisty sposób istotne znaczenie dla wydajności operacji na pamięci. Znajduje to swoje odzwierciedlenie m.in. w pojęciu wyrównania danych (*data alignment*).

Standardowo, kompilatory działają tak, aby pojedyncze zmienne konkretnego typu (mające najczęściej rozmiary 4 lub 8 bajtów) miały adresy swoich początków będące wielokrotnością swojego rozmiaru. Mówimy wtedy o wyrównaniu na granicy 4- lub 8-bajtowej[13].

W przypadku tablic wpływ na wydajność ma położenie kolejnych wyrazów w pamięci. Pojawia się wtedy problem odpowiedniego wykorzystania nie tylko elementów sprzętowych takich jak rejestry, magistrale i moduły pamięci DRAM, ale także pamięci podręcznej. Ze względu na rozmiar linii pamięci podręcznych będący wielokrotnością 8 bajtów, przy zmiennych wyrównanych standardowo w pamięci, nie zachodzi niebezpieczeństwo odwzorowania jednej zmiennej częściowo do jednej, a częściowo do kolejnej linii pamięci podręcznej. Dla krótkich tablic, lub dla algorytmów, które operują na fragmentach tablic, znaczenie może mieć jednak jak sekwencja elementów odwzorowana jest w zbiór linii pamięci podręcznej.

Najczęściej zakłada się, że optymalne ułożenie w pamięci oznacza początek tablicy pokrywający się z początkiem bloku odwzorowanego w pojedynczą linię pamięci podręcznej. Aby to uzyskać wystarczy, aby początkowy adres tablicy był wielokrotnością rozmiaru linii pamięci podręcznej. W trakcie wykonania programu, pierwszy wyraz w tablicy znajduje się wtedy na pewno na początku pewnej linii pamięci podręcznej. W praktyce stosuje się w takim przypadku wyrównanie na granicy 64-bajtowej (rzadziej 128-bajtowej).

[13] Możliwą konsekwencją takiej praktyki jest występowanie niewypełnionych danymi przestrzeni wewnątrz złożonych struktur danych, np. struktur języka C. Jeśli struktura jest wyrównana na granicy 4-bajtowej, jej pierwszym elementem jest pojedyncza zmienna znakowa, a drugim zmienna np. całkowita, to przy wyrównaniu zmiennej całkowitej na granicy 4-bajtowej, pomiędzy pierwszym i drugim elementem struktury pozostaną trzy bajty niezapełnione danymi programu.

Można także stosować wyrównanie na granicy będącej wielokrotnością rozmiaru strony pamięci wirtualnej. Mamy wtedy optymalizację zmierzającą do redukcji liczby błędów strony w trakcie wykonania programu.

Do alokowania w pamięci dynamicznej tablic wyrównanych na odpowiedniej granicy służą rozmaite narzędzia, najczęściej nie będące składową języka programowania. W przypadku wielu kompilatorów języka C można posłużyć się przenośną funkcją standardu POSIX:

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Jedyną istotną różnicą w stosunku do standardowej alokacji w języku C (poza innym sposobem zwracania wskaźnika `*memptr` do zaalokowanej pamięci) jest podanie, oprócz rozmiaru alokowanej pamięci `size` (w bajtach), także żądanego wyrównania `alignment`, także wyrażanego w bajtach.

W niniejszej książce, przy opisach działania sprzętu i wykonania rozmaitych algorytmów, a także w programach związanych z książką, standardowo zakładane jest wyrównanie wszystkich tablic na granicy 64-bajtowej.

Pobieranie z wyprzedzeniem przy realizacji dostępów do pamięci

Wspominany już uprzednio, jako przykład techniki wykonania spekulatywnego, mechanizm sprzętowy pobierania z wyprzedzeniem (*hardware prefetching*) w istotny sposób wpływa na funkcjonowanie dostępów do pamięci we współczesnych procesorach (stosują go praktycznie wszystkie mikroprocesory, czasem w sposób zaawansowany, np. z kilkoma układami współbieżnie realizującymi tego typu pobieranie). Ma on między innymi znaczenie przy wszelkich próbach tworzenia i wykorzystania mikrobenchmarków do pomiaru wydajności pamięci, a także tworzenia modeli wydajnościowych wykonania programów. Z tego względu poniżej omówione zostaną podstawowe zasady jego funkcjonowania oraz ich konsekwencje, na przykładzie prostego algorytmu, często stosowanego przy badaniu parametrów pamięci podręcznej.

Analizowanym algorytmem jest wykonanie prostej pętli, w której sumuje się wartości w tablicy liczbowej `tab` o rozmiarze `rozmiar_tab`:

```
  for (j=0; j<rozmiar_tab; j++) suma += tab[j];
```

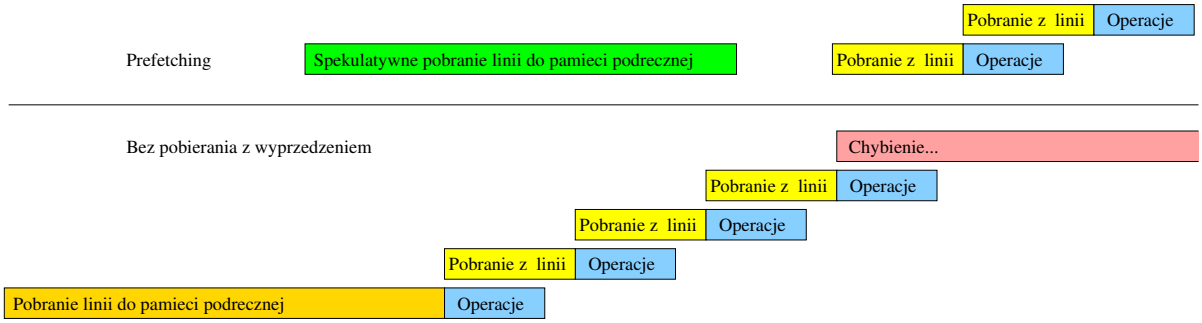
Założeniem badania jest wyrównanie tablicy na granicy będącej wielokrotnością rozmiaru linii pamięci podręcznej. Dzięki temu można założyć, że pierwszy element tablicy jest pierwszym w linii pamięci podręcznej, w którą zostanie odwzorowany.

Kod asemblera dla badanej pętli może wyglądać następująco:

```
.L5:
        addsd   0(%rbp,%rdx), %xmm7
        addq    $8, %rdx
        cmpq    %rax, %rdx
        jne     .L5
```

Nie wchodząc w możliwe warianty kodu (obejmujące użycie innych rozkazów procesora, rozwinięcie pętli przez kompilator, czy zastosowanie innych typów rejestrów), podstawowe działania w każdej iteracji pętli przebiegają w ten sam sposób: realizowany jest dostęp do pamięci (dodanie zawartości elementu tablicy do sumy przechowywanej w rejestrze `%xmm7`), następnie wykonywane operacje na indeksie pętli (rejestr `%rdx`), po czym sterowanie powraca na początek pętli.

Wiedząc o charakterystyce działania pamięci, można przewidzieć, że początkowy dostęp do pamięci spowoduje chybienie w pamięci podręcznej i podmianę linii, co zajmie czas wielokrotnie dłuższy niż wykonanie późniejszych operacji w pętli.



Rysunek 1.20: Schemat mechanizmu pobierania z wyprzedzeniem (*prefetching*)

Naiwne działanie procesora mogłoby polegać na wykonywaniu rozkazów w każdej iteracji i dopiero po przejściu do kolejnej iteracji rozpoczynanie wykonywania rozkazów z nią związanych. Wiedząc, że procesor przeprowadza przewidywanie skoków, wykonywanie poza kolejnością i wykorzystuje zestaw potoków przetwarzania można założyć pewien stopień współbieżności jego działania. Schematycznie i przykładowo przedstawia to rys. 1.20. Na rysunku zaznaczono często występującą sytuację, kiedy na skutek zastosowania powyższych mechanizmów czas wykonania jest w całości determinowany przez czas operacji na hierarchii pamięci.

Na schemacie pokazanym poniżej poziomej linii, kiedy nie stosuje się pobierania z wyprzedzeniem, czas wykonania nie jest optymalny. Żądania dostępu do pamięci DRAM (zakładając dla uproszczenia tylko jeden poziom pamięci podręcznej), są generowane dopiero po wystąpieniu chybienia w pamięci podręcznej, kilka iteracji po pobraniu ostatniego bloku z pamięci DRAM.

Nie uwzględnia to możliwości współczesnych układów pamięci, które potrafią obsługiwać współbieżnie wiele żądań dostępu do pamięci DRAM i tylko w takiej sytuacji wykazują najwyższą wydajność działania. Kolorem zielonym powyżej poziomej linii zaznaczony jest schemat działania z pobieraniem z wyprzedzeniem (*prefetching*), którego celem jest wykorzystanie tych możliwości.

Pobieranie z wyprzedzeniem jest przykładem wykonania spekulatywnego, kiedy procesor lub inny układ sprzętowy realizuje rozkazy i operacje, co do których nie wiadomo czy rzeczywiście będą wymagane w programie. Występuje przewidywanie jak będzie wyglądało wykonanie w przyszłości i działanie na podstawie takich spekulacji (wymaga to także istnienia mechanizmów, które powodują poprawne wykonanie programu, kiedy spekulacje okazują się błędne, co w przypadku *prefetchingu* jest relatywnie proste i polega na pominięciu niepotrzebnie pobranych danych).

Podstawą ogólnego mechanizmu pobierania z wyprzedzeniem jest obserwacja, że dostępy do pamięci są często realizowane przez programy według pewnego wzorca. W przypadku analizowanej powyżej pętli wzorcem tym jest dostęp do kolejnych komórek pamięci oddalonych o stały, ściśle określony odstęp. Układy pobierania z wyprzedzeniem, na podstawie dotychczasowych dostępów do pamięci, spekulatywnie przewidują, gdzie nastąpi kolejny i, zanim jeszcze procesor dotrze do odpowiednich rozkazów pobrania (i związanego z nimi chybienia w pamięci podręcznej), uruchamiają odpowiednie mechanizmy pobierania dla odpowiedniego bloku pamięci DRAM.

W efekcie, bezpośrednio po pobraniu linii pamięci podręcznej, współbieżnie z wykonywaniem operacji na kolejnych elementach linii, procesor może spekulatywnie pobrać zawartość kolejnej linii. Tak właśnie przebiega działanie przedstawione na rys. 1.20, gdzie uwzględniono możliwe pobieranie z wyprzedzeniem, realizowane przed wystąpieniem chybienia w pamięci podręcznej, które teoretycznie powinno uruchamiać pobranie z pamięci DRAM.

W konsekwencji analiza wydajności wykonania pętli może zakładać realizację dostępów do pamięci DRAM jako jedynych elementów wykonania, przy czasie dostępów do pamięci podręcznej i wykonywania operacji na danych całkowicie ukrytym w czasie pobierania danych z pamięci DRAM (fakt ten ma

swoje odzwierciedlenie w omawianych dalej modelach wydajności wykonania kodu).

Pobieranie z wyprzedzeniem zazwyczaj znacząco zwiększa wydajność większości programów. Czas obsługi chybienia w pamięci podręcznej dla nowo pobieranej linii zostaje zredukowany, jeśli linia znajduje się już w drodze z pamięci głównej lub całkowicie ukryty jeśli linia znajdzie się w pamięci podręcznej przed wygenerowaniem żądania dostępu do niej przez procesor. Oczywiście, w przypadku błędnej spekulacji pobrane z wyprzedzeniem dane nie są wykorzystywane, a samo pobieranie z wyprzedzeniem może zaburzać efektywną realizacje dostępów jawnie wskazanych w kodzie, stanowiąc narzut wykonania spekulatywnego.

Współczesne procesory mogą posiadać kilka układów pobierania z wyprzedzeniem, dotyczących np. transferów pomiędzy różnymi poziomami pamięci podręcznej i pamięcią DRAM. Układy te można wyłączać z poziomu systemu operacyjnego, jeśli np. chce się uzyskać informację o pracy konkretnego komponentu sprzętowego, a pobieranie z wyprzedzeniem zaburza uzyskiwanie takiej informacji. W przykładach w niniejszej książce wyłączanie pobierania z wyprzedzeniem nie będzie stosowane, natomiast uwzględniany i badany będzie jego możliwy wpływ na wydajność analizowanych algorytmów. Odpowiada to punktowi widzenia przeciętnego użytkownika systemów komputerowych, który przy wykonywaniu programów nie ma możliwości uniknięcia efektów pobierania z wyprzedzeniem.

1.3.4 Eksperymentalne określanie charakterystyk pamięci głównej i pamięci podręcznych różnych poziomów

W przypadku zdecydowanej większości algorytmów, których wydajność uzależniona jest od czasu dostępów do pamięci, złożoność mechanizmów obsługi pamięci przez współczesne procesory praktycznie uniemożliwia szacowanie czasu wykonania na podstawie parametrów teoretycznych sprzętu, bez wspomagania eksperymentem. Złożoność mechanizmów oznacza nie tylko podstawowe algorytmiczne aspekty funkcjonowania pamięci podręcznych i różne warianty pobierania z wyprzedzeniem, ale także szczegóły funkcjonowania układów SRAM i DRAM, a także ewentualnych dodatkowych układów stosowanych przez producentów procesorów w celu usprawnienia dostępów do pamięci (jak np. działanie układów udostępniania elementów podmienianej linii w pamięci podręcznej, jeszcze przed zakończeniem pobierania całej linii, potokowość układów pamięci, wielobankowość pamięci itp.). Mimo braku pełnego teoretycznego modelu wydajności, można jednak skonstruować mikrobenchmarki, które pozwalają na wykrycie różnic w czasie wykonania dla rozmaitych kombinacji parametrów utworzonych programów i wnioskowanie, na podstawie tych różnic, o wybranych aspektach mechanizmów obsługi pamięci, o wpływie tych mechanizmów na wydajność programu i o możliwych optymalizacjach kodu.

Przykładowym mikrobenchmarkiem wykorzystywanym w kolejnych badaniach jest, najczęściej wielokrotne, wykonanie pętli, w której odwiedzane są kolejno wyrazy pewnej tablicy, oddalone od siebie o `skok` elementów:

```
for (j=0; j<rozmiar_tab; j+= skok) tab[j]++;
```

Tablica jest wyrównana na granicy odpowiadającej długości linii pamięci podręcznej, a analiza mechanizmu pobierania z wyprzedzeniem i potokowego przetwarzania rozkazów dostępu do pamięci wskazuje, że jedynym składnikiem istotnym dla całkowitego czasu wykonania jest czas operacji na hierarchii pamięci. Dla wydajności wykonania znaczenie będzie miał tylko charakter dostępów do pamięci w pętli, inaczej będzie w przypadku odczytu wartości, inaczej w przypadku zapisu, inaczej w zastosowanej w przykładzie modyfikacji wartości, oznaczającej pobranie i zapis.

Eksperymentalne wykrywanie rozmiarów pamięci podręcznych różnych poziomów

W celu ustalenia rozmiaru pamięci podręcznych kolejnych poziomów wykorzystana zostanie wersja kodu przedstawionego w poprzednim punkcie:

```
for (j=0; j<rozmiar_tab; j++) tab[j]++;
```

Badana pętla wykonywana jest dla różnych rozmiarów `rozmiar_tab` tablicy `tab`, z wielokrotnym powtórzeniem dla każdego z rozmiarów (wielokrotne wykonanie dla każdego z rozmiarów jest dodatkowo powtarzane kilka razy w celu uzyskania większej dokładności pomiaru czasu oraz dla uniknięcia efektu zaburzeń przy pierwszym wykonaniu).

W badaniu tym, skok pomiędzy elementami tablicy modyfikowanymi w kolejnych iteracjach pętli wynosi 1. Jest to optymalna wartość, najczęściej pojawiająca się w standardowych algorytmach, dla której i kompilator, i sprzęt mogą zastosować szereg technik i mechanizmów optymalizacji. Przykładowo kompilator może zastosować rozkazy wektorowe, a sprzęt pobieranie z wyprzedzeniem.

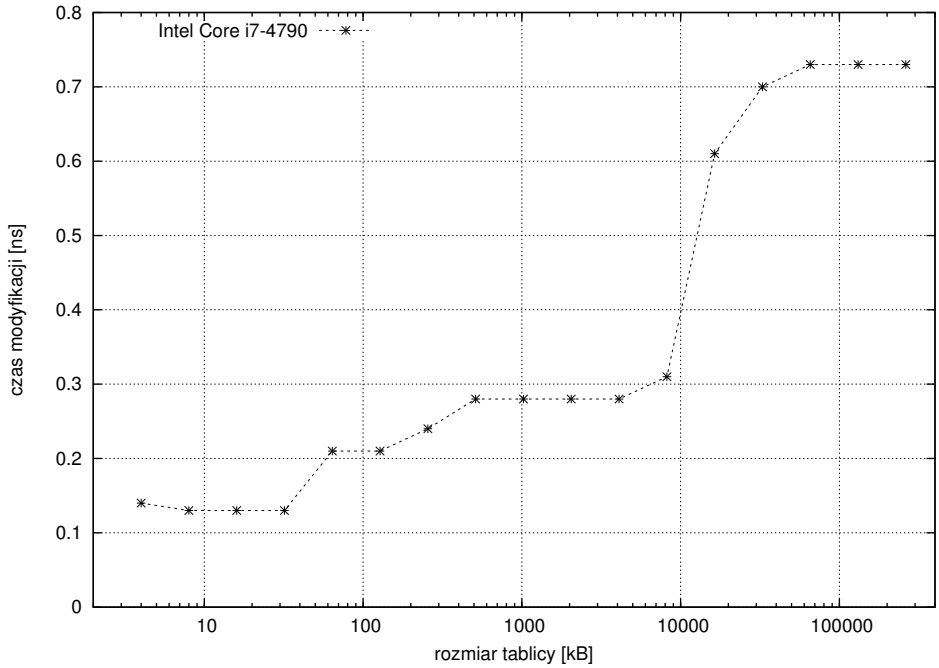
W trakcie wykonania pętli, pobierane z pamięci i modyfikowane są następujące po sobie wyrazy tablicy. Występuje lokalność przestrzenna, kolejne wyrazy znajdują się w pamięci bezpośrednio po sobie. Po chybieniu w pamięci podręcznej związanym z dostępem do pierwszego elementu w bloku pamięci DRAM i pobraniu bloku do linii pamięci podręcznej, dostęp do kilku następnych elementów (ilu to zależy od rozmiaru linii i rozmiaru danych) spowoduje trafienie i szybki transfer z pamięci podręcznej. Wyrazy wypełniają linie pamięci podręcznej w sposób gęsty, każda linia jest wypełniana zawartością bloku pamięci, każdy pobrany z pamięci operacyjnej wyraz jest wykorzystywany w algorytmie. Przepustowość magistrali i układów DRAM nie jest marnowana na przesyłanie danych niewykorzystywanych w obliczeniach, pracują one (między innymi także dzięki pobieraniu z wyprzedzeniem) z pełną wydajnością.

Jeżeli cała tablica mieści się w pamięci podręcznej, to kolejne wykonanie pętli będzie pracować tylko na elementach już pobranych z pamięci DRAM, bez chybień w pamięci podręcznej (dla każdego pobranego wyrazu zachodzić będzie wystarczająca dla danego poziomu pamięci podręcznej lokalność czasowa). Tym sposobem uśredniony czas dostępu do pojedynczej danej (przy odpowiednio dużej liczbie powtórzeń praktycznie niwelujący wpływ pierwszego pobrania z pamięci DRAM), będzie czasem dostępu do pamięci podręcznej.

Jeśli tablica nie mieści się w pamięci podręcznej, to w pewnym momencie nowe wyrazy tablicy zaczynają podmieniać wyrazy już pobrane. Dla prostego przykładu analizowanej pętli efekt podmian będzie taki, że w miarę zwiększania rozmiaru tablicy liczba chybień w każdym przebiegu pętli będzie rosła, przez co będzie rósł średni czas dostępu do pojedynczej zmiennej. W przypadku istnienia pamięci podręcznej kolejnego poziomu, o większym rozmiarze, tablica wciąż może w całości mieścić się w tej pamięci. W efekcie, czas dostępu do pojedynczej danej stanie się czasem dostępu do pamięci podręcznej kolejnego poziomu. Dla dalej rosnącego rozmiaru tablicy cały proces będzie się powtarzał, aż dla największych tablic czas dostępu stanie się czasem dostępu do pamięci DRAM.

Wyniki wydajnościowe opisanego wyżej eksperymentu obliczeniowego dla stosowanego w książce procesora Intel Core i7-4790 przedstawia rys. 1.21. Na osi poziomej znajduje się rozmiar tablicy wielokrotnie modyfikowanej w pętli, a na osi pionowej czas modyfikacji pojedynczego elementu, obliczony jako iloraz liczby dostępów, wynikającej z kodu źródłowego, podzielonej przez zmierzony czas wielokrotnego wykonania pętli (liczba powtórzeń pętli była w trakcie wykonywania pomiaru inna dla każdego rozmiaru, tak aby uzyskać miarodajne wyniki w rozsądnym czasie). Kolejne rozmiary tablicy na rys. 1.21 są kolejnymi potęgami 2, od 4 kB do 256 MB.

Z rysunku odczytać można dla jakich rozmiarów tablicy następuje odejście od stałych czasów dostępu na płaskich fragmentach wykresu, wskazujące na przekroczenie rozmiaru kolejnego poziomu pamięci podręcznej. Widać cztery takie płaskie fragmenty, pierwszy z czasem dostępu niewiele ponad 0.1 ns, kończący się w 4 punkcie wykresu (co przy przyjętych założeniach eksperymentu odpowiada rozmiarowi $2^4 * 2$ kB), drugi, z czasem ok. 0.2 ns i końcem dla $2^6 * 2$ kB, trzeci z czasem poniżej 0.3 ns i końcem dla $2^{12} * 2$ kB, i wreszcie, czwarty, ostatni z czasem ponad 0.7 ns. W efekcie (zakładając



Rysunek 1.21: Średni czas modyfikacji pojedynczego wyrazu tablicy, dla wielokrotnie wykonywanej pętli odwiedzania jej kolejnych wyrazów, w zależności od rozmiaru tablicy.

standardową praktykę konstruowania pamięci podręcznych z rozmiarami będącymi potęgami 2^{14}), dane wykresu prowadzą do następującego oszacowania rozmiarów pamięci: L1 - 32 kB, L2 - 256 kB, L3 - 8 MB. Ostatni płaski fragment wykresu, odpowiadający pamięci DRAM, kończy się dla tablicy o rozmiarze 256 MB – dalszy wzrost rozmiaru może spowodować pewne wachania czasu dostępu, związane z innymi mechanizmami funkcjonowania pamięci, takimi jak rozmiar stron, rozmiar pamięci TLB itp.

Uzupełniającym badaniem rozmiaru pamięci podręcznych, może być analiza wykorzystująca ten sam eksperyment, ale przeprowadzająca wnioskowanie na podstawie liczby chybień w pamięciach podręcznych, a nie wyników wydajnościowych (choć oczywiście jedno jest powiązane z drugim).

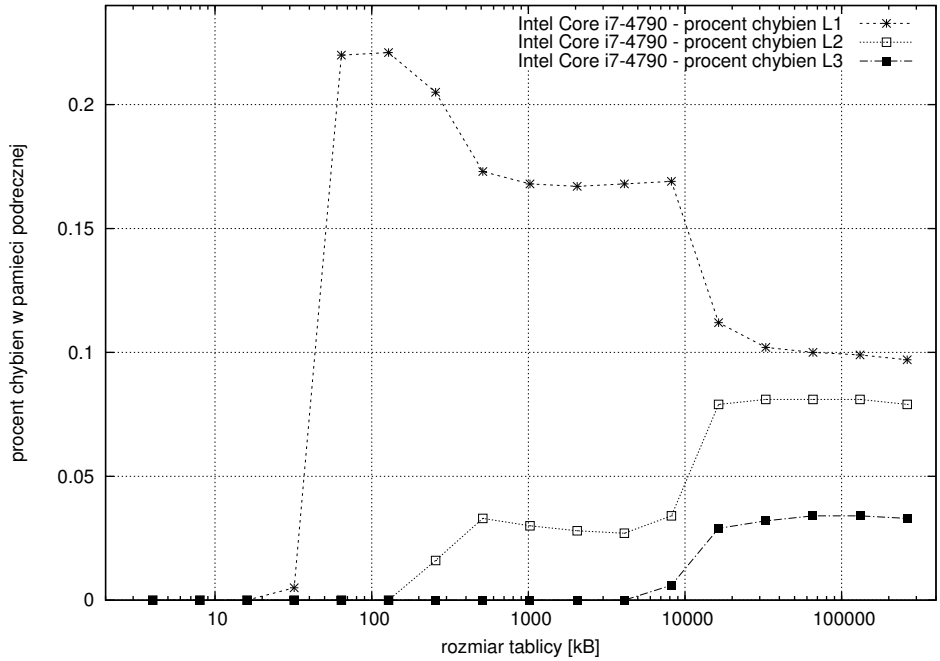
W trakcie wykonania pętli zbierane są dane z liczników sprzętowych dla zdarzeń rozkazów pobrania danych oraz dla chybień w pamieci podręcznej:

- MEM_UOPS_RETIRED.ALL_LOADS,
- MEM_UOPS_RETIRED.L1_MISS,
- MEM_UOPS_RETIRED.L2_MISS,
- MEM_UOPS_RETIRED.L3_MISS.

Następnie obliczany jest procent chybień, jako stosunek trzech ostatnich wartości do pierwszej. Rysunek 1.21 przedstawia krzywe uzyskane, jak zwykle dla testowej platformy z mikroprocesorem Intel Core i7-4790.

Wyczerpująca analiza wykresów wymagałaby uwzględnienia szeregu szczegółów funkcjonowania pamięci podręcznych. Na potrzeby badania rozmiaru pamięci wystarczające jest zaobserwowanie punktów, dla których następuje wzrost procentu chybień dla konkretnego rodzaju pamięci. Widać, że znaczący

[14] Praktyka taka dotyczy głównie pamieci L1 i L2, pamięci L3 mogą mieć bardziej zróżnicowane rozmiary



Rysunek 1.22: Procent chybień w pamięci podręcznej różnych poziomów, dla wielokrotnie wykonywanej pętli odwiedzania kolejnych wyrazów tablicy, w zależności od rozmiaru tablicy.

przyrost, po którym następuje osiągnięcie relatywnie stałej wartości (aż do osiągnięcia rozmiaru kolejnego poziomu pamięci podręcznej) pojawia się przy rozmiarze 32 kB dla pamięci L1, 256 kB dla L2 i 8 MB dla L3, co potwierdza rezultaty osiągnięte w badaniu wydajnościowym.

Eksperymentalne wykrywanie rozmiaru pojedynczej linii pamięci podręcznej

Chcąc znaleźć rozmiar pojedynczej linii pamięci podręcznej można wykonać kolejny eksperyment posługując się ponownie prostym algorytmem:

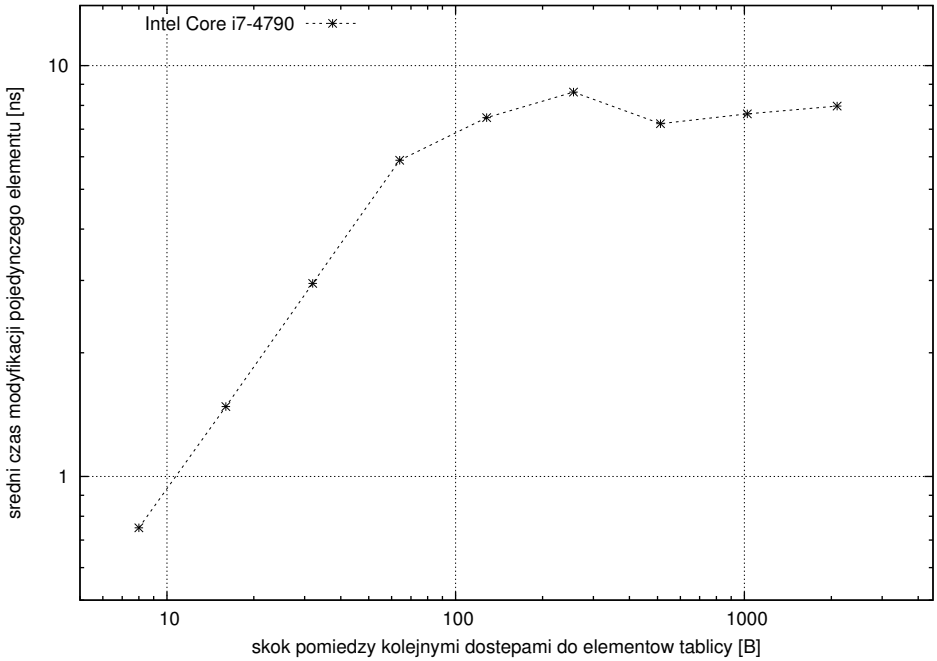
```
for (j=0; j<rozmiar_tab; j+=skok) tab[j]++;
```

Tym razem, rozmiar tablicy jest stały i na tyle duży, żeby nie mieściła się ona w pamięci podręcznej żadnego z poziomów. Istotny jest rozmiar pojedynczego elementu tablicy, w przeprowadzanych eksperymentach równy 8 bajtów (zmienne podwójnej precyzji). Badanie dotyczy pamięci L1 i polega na przeprowadzeniu serii obliczeń dla rosnących wartości zmiennej `skok`, znowu jako potęg liczby 2.

Średnie czasy dostępu na platformie testowej dla tablicy o rozmiarze 256 MB wyglądają następująco:

`skok`	1	2	4	8	16	32	64	128	256
średni czas dostępu	0.75	1.48	2.95	5.88	7.47	8.62	7.22	7.63	7.97

Dla każdej wartości zmiennej `skok` pobranie pierwszego elementu tablicy oznacza chybienie w pamięci podręcznej (wszystkich poziomów) i pobranie linii z pamięci DRAM. W przypadku wartości `skok` równej 1 program wykorzystuje do obliczeń wszystkie pobrane wartości – liczba dostępów wynikająca z kodu źródłowego odpowiada liczbie pobranych elementów. Dla wartości `skok=2` pobrana również zostanie cała linia, ale wykorzystany w programie do modyfikacji będzie tylko co drugi element. Liczba efektywnych dostępów w kodzie będzie dwa razy mniejsza niż liczba pobranych danych. Zakładając ten sam czas pobrania linii co dla `skok=1` oznacza to dwukrotnie wyższy czas przypadający



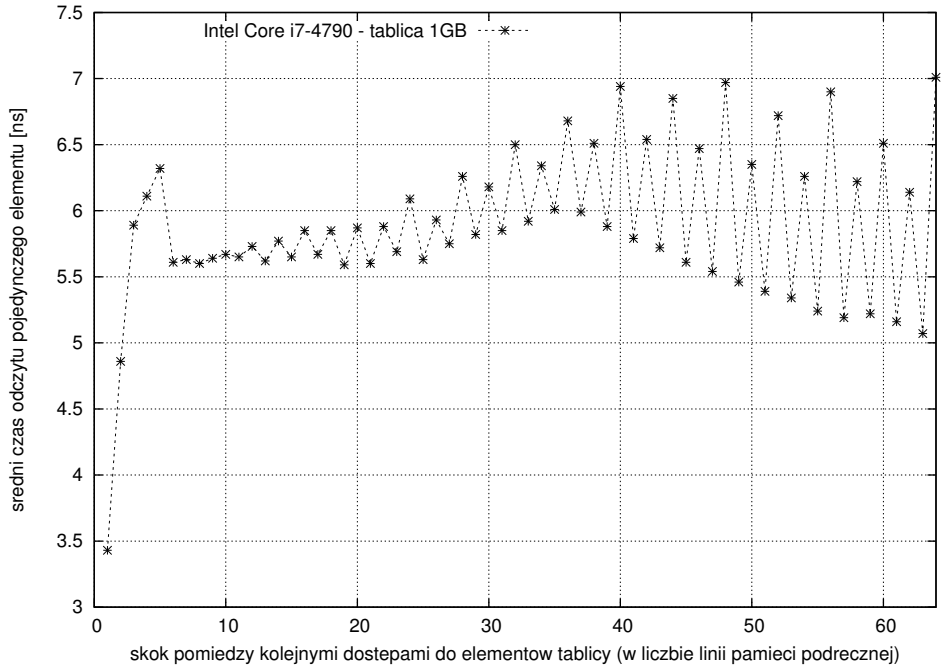
Rysunek 1.23: Średni czas dostępu do pojedynczego elementu tablicy jako funkcja skoku pomiędzy elementami tablicy modyfikowanymi w kolejnych iteracjach pętli

na pojedynczy efektywny dostęp. Podobnie dla wartości `skok=4` – efektywny czas dostępu powinien znów dwukrotnie wzrosnąć. Będzie się tak działo, aż do osiągnięcia przez odstęp pomiędzy dwoma kolejno wykorzystywanymi w pętli elementami (mierzony w bajtach w pamięci DRAM), długości linii pamięci podręcznej (w przypadku obliczeń testowych odstęp jest iloczynem wartość zmiennej `skok` i rozmiaru pojedynczego elementu tablicy, czyli `8*skok`). Kiedy odstęp w pamięci DRAM między kolejno odwiedzanymi elementami przekracza rozmiar linii pamięci podręcznej, efektywnie wykorzystany w programie jest zawsze tylko jeden element tablicy z całej pobranej linii. Narzut wynikający z tego faktu jest nadal zmienny, jednak zależność czasu dostępu od wartości `skok` przestaje być prostą zależnością liniową (w grę zaczyna wchodzić fakt, że nie wszystkie bloki pamięci DRAM są wymagane do pobrania do pamięci podręcznej). W zamieszczonej powyżej tabeli widać, że wartość graniczna rosnącego czasu dostępu do pojedynczej zmiennej (a więc malejącego procentu wykorzystywanych elementów w linii) osiągana jest dla zmiennej `skok` równej 8 (pomiędzy wartością `skok=4` a `skok=8` czas wzrasta nieomal dokładnie dwukrotnie, pomiędzy wartościami 8 i 16 tylko o ok. 30%). Można z tego wyciągnąć wniosek, że długość pojedynczej linii pamięci podręcznej to 8 liczb podwójnej precyzji, czyli 64 bajty.

Charakter powyższych zależności dla platformy testowej dobrze ilustruje wykres na rys. 1.23, odpowiadający danym z zamieszczonej powyżej tabeli. Widać jak początowo zależność średniego czasu modyfikacji pojedynczego elementu tablicy rośnie liniowo jako funkcja skoku pomiędzy kolejnymi elementami tablicy odwiedzanymi w pętli testowego algorytmu. Na osi x skok jest wyrażony w bajtach co pozwala odczytać rozmiar linii pamięci podręcznej, jako miejsce gdzie krzywa zmienia swój charakter. Po przekroczeniu wartości 64 B następuje spowolnienie wzrostu czasu modyfikacji elementu tablicy, a w dalszej kolejności ustabilizowanie czasu dostępu w zakresie ok. 7-8 ns.

Algorytm użyty do badania rozmiaru pamięci podręcznych i rozmiaru pojedynczej linii pamięci podręcznej, mimo swojej prostoty, pozwala na dokładne śledzenie wpływu lokalności odniesień, tak czasowej, jak i przestrzennej, na wydajność dostępów do pamięci.

W pierwszym przypadku lokalność przestrzenna jest zawsze taka sama, niezależnie od rozmiaru



Rysunek 1.24: Zależność czasu dostępu do pojedynczego wyrazu tablicy od skoku pomiędzy dwoma kolejno odwiedzanymi wyrazami – odczyt dla całej tablicy o rozmiarze 1GB

tablicy (każda wartość pobrana do pamięci podręcznej jest jednokrotnie wykorzystana w pojedynczym wykonaniu pętli), a wydajność (jako odwrotność czasu dostępu) zależy wyłącznie od lokalności czasowej oraz proporcji rozmiaru tablicy do pojemności pamięci podręcznej konkretnego poziomu.

W drugim przypadku, kiedy z powodu dużego rozmiaru tablicy nie występuje lokalność czasowa, wydajność zależy wyłącznie od lokalności przestrzennej, sterowanej przez wartość zmiennej `skok`.

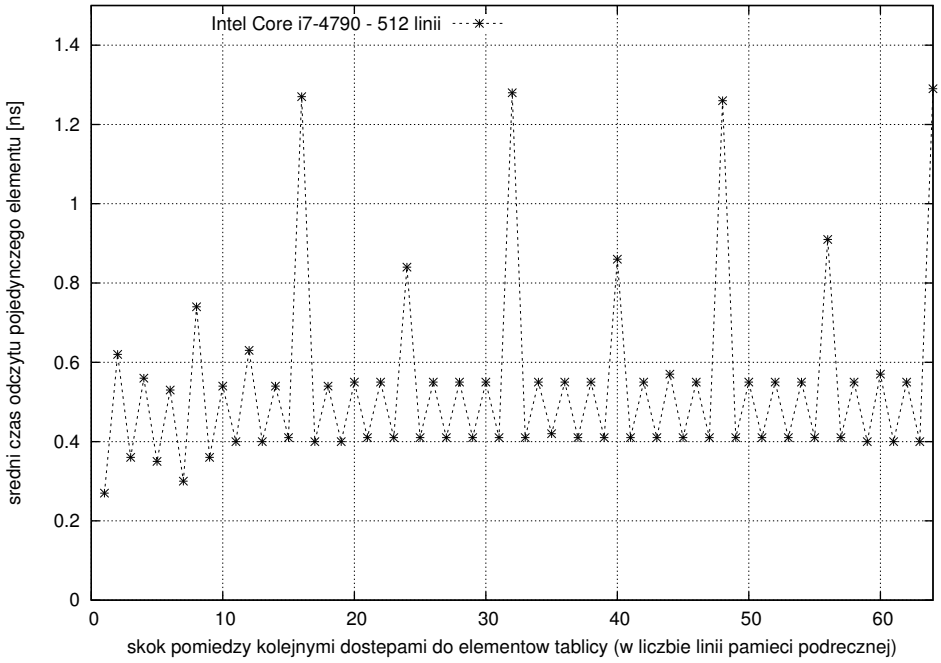
Badanie zależności czasu dostępu do pamięci od wzorca dostępu

Do badania zależności czasu dostępu do pamięci od wzorca dostępu może posłużyć kolejna wersja prostego algorytmu przeglądania tablicy jednowymiarowej, tym razem zapisana jako:

```
for(blok=0; blok < LICZBA_LINII*SKOK; blok+= SKOK){
  suma += a[ blok*(ROZMIAR_LINII)];
}
```

Parametr `ROZMIAR_LINII` odpowiada liczbie elementów tablicy w pojedynczej linii pamięci podręcznej. Dzięki takiej wartości, w każdej iteracji pętli odwiedzany jest inny blok pamięci DRAM, powiązany z pojedynczą linią pamięci podręcznej. Parametr `SKOK` decyduje o odstępie miedzy kolejnymi pobieranymi elementami tablicy `a`, a w konsekwencji o odstępie (liczonym w liczbie bloków, a więc i linii pamięci podręcznej) pomiędzy kolejno odwiedzanymi elementami i zawierającymi je blokami.

Rysunki 1.24 i 1.25 pokazują średni czasu dostępu (oś y na wykresie) przy wielokrotnym wykonaniu rozważanej pętli (co daje dodatkowy efekt możliwej lokalności czasowej odniesień). Pomiary wykonywane są dla odpowiednio długiej tablicy (zaalokowanej z rozmiarem ponad 1 GB) oraz różnych wartości skoku (oś x na wykresie), mierzonego w liczbie linii pamięci podręcznej (a więc 1 odpowiada skokowi 64 B, 2 skokowi 128 B, itd.). Pomiary jak zwykle przeprowadzone są na maszynie z procesorem Intel Core i7-4790 o architekturze Haswell.



Rysunek 1.25: Zależność czasu dostępu do pojedynczego wyrazu tablicy od skoku pomiędzy dwoma kolejno odwiedzanymi wyrazami – odczyt dla pierwszych 512 linii tablicy o rozmiarze 1GB

Wykres 1.24 pokazuje wyniki dla eksperymentu odczytu dla całego zakresu tablicy (liczba odwiedzanych linii jest odwrotnie proporcjonalna do skoku pomiędzy dostępami), a wykres 1.25 wyniki dla odczytu z pierwszych 512 linii oddzielonych odpowiednim skokiem (zakres indeksów odwiedzanych elementów w tablicy rośnie więc dla kolejnych punktów na osi x, zawsze do 512-krotności skoku).

W pierwszym eksperymencie, przeglądania całego zakresu tablicy, liczba odwiedzanych linii, choć maleje dla kolejnych wartości na osi x, jest na tyle duża, że ich sumaryczny rozmiar każdorazowo kilkakrotnie przekracza rozmiar pamięci L3. Niemniej od pewnej wartości skoku, zjawisko lokalności czasowej dla pamięci L3 zaczyna się pojawiać, co skutkuje malejącym średnim czasem dostępu.

Sumaryczny rozmiar 512 linii w drugim eksperymencie wynosi 32 kB (zakładając długość linii 64 B), co oznacza możliwość zmieszczenia wszystkich odwiedzanych linii w pamięci L1 procesora. Na skutek opisanego wyżej sposobu funkcjonowania pamięci wielodrożnych i odwzorowania bloków pamięci głównej w różne zbiory linii pamięci podręcznej, zależnie od wartości skoku, nie zawsze wykorzystana jest całą pamięć L1 i pojawiają się chybienia konfliktowe. Liczba odwiedzanych linii jest jednak na tyle mała, że praktycznie w każdym przypadku zachowana jest lokalność czasowa dla pamięci L2 (brak chybień pojemnościowych i konfliktowych), dzięki czemu czasy dostępu są znacznie niższe niż w eksperymencie pierwszym (ośmiodrożna pamięć L2 składa się z 4096 linii w 512 zbiorach).

Dokładna analiza pokazanych wyników musiałaby obejmować szereg dalszych aspektów, poza lokalnością czasową oraz chybieniami pojemnościowymi i konfliktowymi, takie jak np. pobieranie z wyprzedzeniem czy mechanizmy obsługi pamięci wirtualnej (dla odpowiednio dużego skoku pomiędzy odwiedzanymi elementami tablicy, każdy z nich może znajdować sie na innej stronie pamięci wirtualnej, co ma wpływ tak na funkcjonowanie pamięci TLB, jak i mechanizm pobierania z wyprzedzeniem).

Z punktu widzenia praktyki ważna jest obserwacja czasów dostępu, które dla wybranych przypadków mogą różnić się nawet kilkukrotne. W obu eksperymentach, dla odpowiednich zakresów wartości skoku, widać różnicę w czasie dostępu pomiędzy odczytami ze skokiem o parzystą liczbę linii, a odczytami ze skokiem o nieparzystą liczbę linii. Dodatkowo, dla eksperymentu odwiedzania 512 linii szczególnie

długim czasem dostępu wyróżniają się dostępy ze skokiem będącym wielokrotnością 8 linii pamięci podręcznej, a zwłaszcza ze skokiem będącym wielokrotnością 16 linii. Prowadzi to do wniosku, że w programach ukierunkowanych na wysoką wydajność obliczeń należy szczególnie zwracać uwagę na sytuacje kiedy w następujących po sobie iteracjach dochodzi do dostępów do tablic w lokalizacjach odległych o wielokrotności charakterystycznych potęg 2.

Przykład przeciwdziałania wzrostowi opóźnień przy dostępach do pamięci poprzez rozciąganie (rozpychanie) tablic (*array padding*)

Kolejnym badanym przykładem zależności czasu dostępu do pamięci podręcznej od wzorca dostępu, mającym pewne znaczenie praktyczne i zasługującym na analizę i optymalizację, jest prosty algorytm polegający na wielokrotnym obliczaniu transpozycji macierzy za pomocą pętli:

```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
    at[i*n+j] = a[j*n+i]; // AT[i][j] = A[j][i]
  }
}
```

W badanym algorytmie występują dwie macierze: będąca źródłem danych macierz A oraz zapisywana macierz A^T, mająca być transpozycją A. W pętli założone jest przechowywanie macierzy wierszami w tablicach jednowymiarowych `a` i `at` (patrz p. 1.1.1). Dla uproszczenia przyjęto macierze kwadratowe *n*x*n*.

Z natury operacji transpozycji, $A^T_{ij} = A_{ji}$, wynika, że wiersze jednej z macierzy są kolumnami drugiej. Jeśli odwiedzamy w standardowej podwójnej pętli wszystkie elementy obu macierzy, to dla jednej z nich odwiedzanie będzie odbywać się wierszami (w kolejnych iteracjach dostęp do kolejnych wyrazów wiersza), a dla drugiej kolumnami (w kolejnych iteracjach dostęp do kolejnych wyrazów kolumny). W przykładowej, zaprezentowanej powyżej, implementacji, dostęp wierszami dotyczy macierzy A^T. Dostęp do A polega na tym, że w kolejnych iteracjach wewnętrznej pętli po zmiennej `j`, odczytywane są wyrazy tablicy `a` odległe o *n*, a więc wyrazy w kolejnych wierszach aktualnej *i*-tej kolumny (`a[j*n+i]` oznacza wyraz A_{ji}).

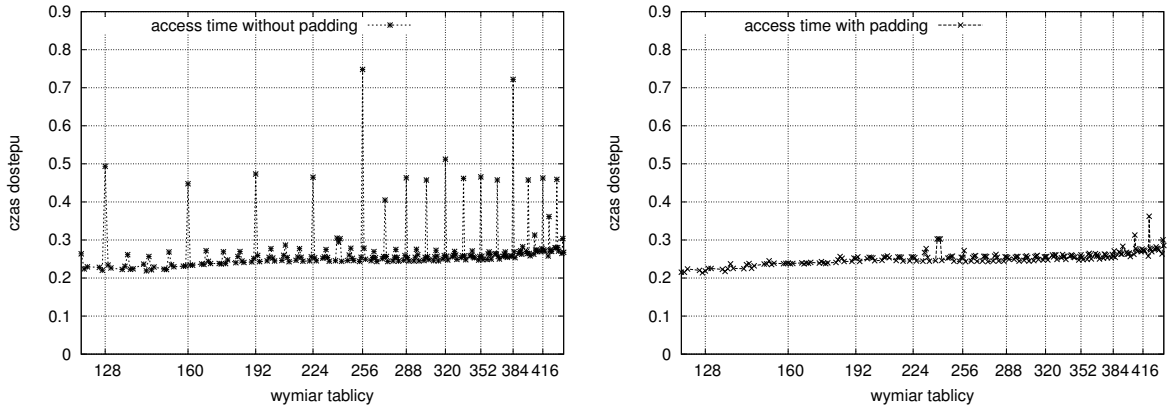
Zgodnie z omawianymi dotychczas zasadami działania pamięci, wielokrotne odczytywanie tablicy ze skokiem pomiędzy kolejno odwiedzonymi wyrazami może stać się źródłem problemów wydajnościowych, dla specyficznych wartości skoku. Rys. 1.26 przedstawia po lewej stronie wykres średniego czasu dostępu do pojedynczego wyrazu tablic dla badanego kodu i różnych wymiarów macierzy. Widać, że dla niektórych wymiarów *n* czas ten rośnie znacząco – dwu, a nawet trzykrotnie. Wszystkie te przypadki odpowiadają wymiarom macierzy będącym wielokrotnościami 8 (czyli wierszom o długości będącej wielokrotnością 64 bajtów, a więc szerokości linii pamięci podręcznej).

W celu uniknięcia znaczącego wydłużenia czasu działania dla wybranych rozmiarów tablic można zastosować technikę "rozpychania" tablic omówioną w p. 1.1.1.

Wykorzystywane w tej technice alokowanie zamiast oryginalnej tablicy, tablicy o wydłużonym wierszu powoduje, dla macierzy przechowywanych wierszami, zmianę odstępu pomiędzy kolejnymi wyrazami w kolumnie, a więc zmianę wzorca dostępu do pamięci, w przypadku odwiedzania elementów w tej samej kolumnie.

Zakładając alokację, zamiast tablicy o rozmiarze $n \times n$, tablicy o rozmiarze $n \times (n + o)$, można optymalnie dobrać parametr `o`, tak aby uniknąć niekorzystnego wzorca dostępów do pamięci.

W badanym algorytmie przyjęto, że zamiast dla tablicy $n \times n$, w przypadku kiedy n jest podzielne przez 8, alokuje się pamięć dla tablicy $n \times (n + 1)$ (każdorazowa długość wiersza przechowywana jest w algorytmie w zmiennej `WYMIAR`). W celu realizacji transpozycji, pętle algorytmu pozostają bez



Rysunek 1.26: Średni czas dostępu do pojedynczego elementu tablic liczb podwójnej precyzji, podczas wielokrotnego obliczania transpozycji macierzy, dla różnych wymiarów macierzy

zmian, nadal operuje on na *n*n* wyrazach tablic (dodatkowe elementy tablicy `a` mogą nawet pozostać niezainicjowane), zmienia się tylko zapis dostępu do elementów oryginalnej tablicy `a`:

```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
    at[i*n+j]=a[j*WYMIAR+i]; // AT[i][j] = A[j][i]
  }
}
```

Po prawej stronie na rys. 1.26 znajduje się wykres średniego czasu dostępu do pojedynczego wyrazu tablic dla zmodyfikowanego kodu i różnych wymiarów macierzy. Różnice w czasie dostępu dla różnych przypadków są znacząco zmniejszone w stosunku do oryginalnego programu.

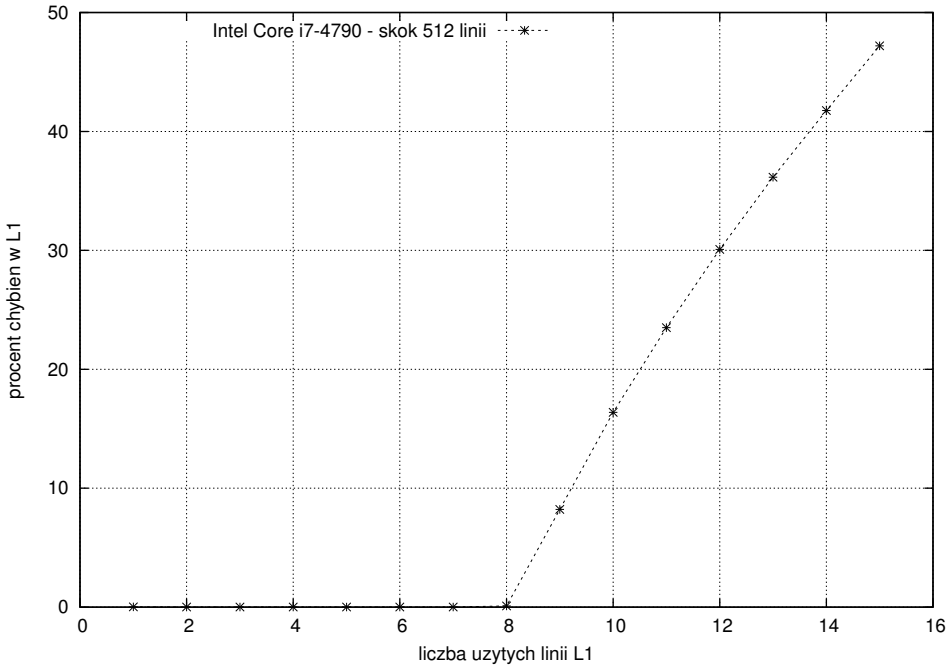
Dane zaprezentowane na wykresach dotyczą wielokrotnego dokonywania transpozycji relatywnie małych macierzy, a więc specyficznego przypadku algorytmu. W przypadku jednokrotnego wykonania transpozycji znika znaczące (rzędu kilku razy) zróżnicowanie pomiędzy wydajnością dla poszczególnych zbliżonych wymiarów macierzy. Czasy dostępu nadal zależą od wymiaru, jednak nieznacznie tylko rosną, w momencie kiedy rozmiary tablic przekraczają wartości prowadzące do chybień pojemnościowych dla kolejnych pamięci podręcznych (od rzędu ok. 0.5 ns dla wymiarów rzędu kilkuset, do ok. 1.7 ns dla wymiarów rzędu kilkunastu, kilkudziesięciu tysięcy).

Wyjątkiem są przypadki kiedy pojedynczy wiersz macierzy zajmuje wielokrotność pojedynczej strony pamięci wirtualnej (wtedy wymiar jest wielokrotnością 512 dla stron wielkości 4 kB), a średni czas dostępu rośnie do kilku nanosekund, co oznacza, że czas wykonania programu rośnie kilkukrotnie. Dla większych wymiarów macierzy kilkukrotny wzrost czasu dostępu dotyczy także (choć w mniejszym stopniu) rozmiarów wierszy będących wielokrotnościami 256 bajtów. Zastosowanie opisanego wyżej rozciągania tablic, w każdym przypadku redukuje kilkukrotnie zwiększony czas dostępu z powrotem do standardowego dla danego zakresu wymiarów macierzy.

Eksperymentalne wykrywanie drożności pamięci podręcznej

Do zbadania drożności pamięci podręcznej użyta może zostać (jak zwykle wykonywana wielokrotnie) ta sama pętla co w punkcie poprzednim:

```
for(blok=0; blok < LICZBA_LINII*SKOK; blok+= SKOK){
  suma += a[ blok*(ROZMIAR_LINII)];
```



Rysunek 1.27: Zależność procentu chybień w L1 od liczby używanych linii tablicy przy odwzorowaniu wszystkich linii w ten sam zbiór linii w pamięci L1

```
}
```

Tym razem parametry dobrane są tak, żeby wszystkie bloki odwzorowane były w ten sam zbiór linii pamięci. Dla maszyny testowej wyposażonej w pamięć L1 o pojemności 32 kB (512 linii o rozmiarze 64 B) wystarcza do tego przyjęcie wartości `SKOK = 512`. Przeskok o cały rozmiar pamięci przy odwiedzeniu kolejnego wyrazu tablicy, gwarantuje umieszczenie kolejnego odwiedzanego bloku pamięci DRAM w tym samym zbiorze linii.

W eksperymencie zliczana jest częstość występowania chybień w pamięci L1 (jako proporcja liczby zdarzeń MEM_UOPS_RETIRED.L1_MISS i MEM_UOPS_RETIRED.ALL_LOADS) dla różnych wartości parametru `LICZBA_LINII`. Dla `LICZBA_LINII = 1`, algorytm odczytuje wartości tylko z pierwszego elementu tablicy (do odpowiedniego zbioru linii trafia tylko jeden blok pamięci DRAM). Dla `LICZBA_LINII = 2` odczytywane są dwie wartości z dwóch bloków – obu odwzorowanych w ten sam zbiór linii, dla `LICZBA_LINII = 3` z trzech itd.

Rys. 1.27 przedstawia wyniki eksperymentu. Dla pierwszych kilku wartości parametru `LICZBA_LINII` chybienia w pamięci L1 praktycznie nie występują, liczba odwiedzanych bloków pamięci DRAM jest mniejsza niż liczba linii w pojedynczym zbiorze linii. Nagły wzrost procentu chybień pojawia się po przekroczeniu wartości `LICZBA_LINII = 8`. Prostym wnioskiem jest, że drożność pamięci L1 procesora wynosi 8.

1.3.5 Pomiary opóźnienia i przepustowości elementów hierarchii pamięci

Dotychczasowe eksperymenty obliczeniowe stosowały różne algorytmy operujące na tablicach, w których czas wykonania zakładany był jako w całości poświęcony na odczyty i zapisy danych, a średnie czasy dostępu do pojedynczej danej znacząco się różniły (od ok. 0.1 ns do ok. 8 ns). Z punktu widzenia analizy możliwych do osiągania w praktyce wartości wydajności operacji na pamięci, ciekawe jest skonstruowanie benchmarków, w których czasy te osiągałyby wartości ekstremalne.