

AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Dominik Czarnota

kierunek studiów: **informatyka stosowana**

Wpływ sposobu organizacji w pamięci złożonych struktur danych na wydajność kodu wynikowego

Opiekun: **dr inż. Bartosz Mindur**

Kraków, styczeń 2016

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

Spis treści

| | |
|--|----|
| 1. Wstęp | 5 |
| 2. Teoria | 7 |
| 2.1. Przetwarzanie potokowe (ang. <i>pipelining</i>) | 7 |
| 2.1.1. Superpotokowość – zwiększenie liczby etapów potoków | 8 |
| 2.1.2. Superskalarność | 8 |
| 2.1.3. Superskalarność i superpotokowość | 8 |
| 2.1.4. Zależności między instrukcjami | 9 |
| 2.1.5. Gałęzie wykonania (ang. <i>branches</i>) | 9 |
| 2.2. Jednoczesna wielowątkowość | 10 |
| 2.3. Instrukcje wektorowe – SIMD | 11 |
| 2.4. Problem dostępu do pamięci | 12 |
| 2.4.1. Hierarchia pamięci | 13 |
| 2.4.2. Pamięć podręczna procesora | 13 |
| 2.4.3. Poziomy pamięci podręcznej | 14 |
| 2.4.4. Implementacja pamięci podręcznej | 15 |
| 2.4.5. Asocjacyjność | 16 |
| 2.4.6. Tagi w pamięci cache | 17 |
| 2.5. Pamięć wirtualna | 18 |
| 2.6. Wczesne pobieranie | 18 |
| 2.7. Języki programowania a ułożenie danych | 19 |
| 2.8. Wyrównanie danych | 19 |
| 2.8.1. Wyrównanie danych w strukturach | 20 |
| 2.8.2. Zmiana wyrównania | 23 |
| 3. Testy wydajności | 27 |
| 3.1. Przejście po macierzy | 28 |
| 3.2. Tablica struktur a struktura tablic | 33 |
| 3.2.1. Dostęp sekwencyjny | 33 |

| | |
|--|-----------|
| 3.2.2. Dostęp sekwencyjny – „kompaktowa” struktura | 37 |
| 3.2.3. Dostęp swobodny | 40 |
| 3.3. Przetwarzanie warunkowe | 43 |
| 3.4. Przetwarzanie równoległe..... | 48 |
| 3.5. Wyrównanie danych | 55 |
| 4. Podsumowanie oraz wnioski | 59 |

1. Wstęp

W obecnych czasach, gdy prędkość przesyłania danych nie nadąża za częstotliwością taktowania procesora, wiele jego cykli jest marnowanych na bezczynne oczekiwanie na dane. Biorąc pod uwagę sposób działania sprzętu, na którym wykonywane jest dane oprogramowanie, możliwe jest napisanie programu w taki sposób, aby zminimalizować czas bezczynności procesora.

W niniejszej pracy zbadano wpływ ułożenia danych na prędkość działania programu. Zebrano dotychczasowe badania na ten temat, rozpoczynając od teorii działania współczesnych procesorów. Następnie zaprezentowano i przeanalizowano zestaw przykładów. Na końcu sformułowano wnioski, podsumowujące przedstawione metody przyspieszenia działania programu, czasem bardzo niewielkim kosztem.

Opisane optymalizacje i wnioski mogą zostać wykorzystane w projektach, dla których wydajność ma kluczowe znaczenie – przykładowo, gdy przetwarzane są bardzo duże ilości danych, przez co nawet niewielkie zmiany mogą wywrzeć zauważalny wpływ na szybkość działania. W takiej sytuacji warto zatem rozważyć każdą możliwość optymalizacji.

Jednym z przykładów takiego oprogramowania jest ATLAS Trigger. Eksperyment ATLAS (*A Toroidal LHC ApparatuS*) to jeden z detektorów LHC (*Large Hadron Collider*, Wielki Zderzacz Hadronów), który analizuje i rejestruje zderzenia cząstek. W wyniku tego powstają ogromne ilości danych, które muszą być zredukowane, zanim będą mogły zostać trwale zapisane – ponieważ obecne oprogramowanie jest w stanie przetworzyć tylko niewielki odsetek zdarzeń, potrzebne jest odfiltrowanie na początku tych najbardziej interesujących.

Innym przykładem jest pisanie silników gier komputerowych oraz samych gier. Muszą one być w stanie wykonywać jak najwięcej obliczeń w jak najkrótszym czasie. Ciekawym aspektem tego przykładu są konsole. Gry pisane na nie są zazwyczaj lepiej zoptymalizowane, niż te pisane na PC, ponieważ twórcy gier mogą się skupić na optymalizacji pod daną konfigurację sprzętu, a nie, tak jak w przypadku komputerów osobistych, brać pod uwagę wiele rodzajów sprzętu.

Z uwagi na specyfikę problemu, wszystkie przytoczone przykłady zostały napisane w języku C++, gdyż języki wyższego poziomu nie dają programiście możliwości manipulowania tym, jak dane ułożone są w pamięci.

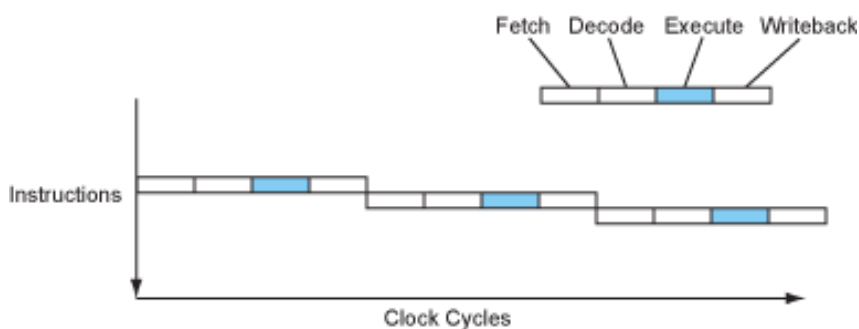
2. Teoria

Aby móc zagłębić się w temat, należy najpierw wyjaśnić, jak działają poszczególne rozwiązania wykorzystywane w procesorach oraz jak dane są pobierane przez procesor.

2.1. Przetwarzanie potokowe (ang. *pipelining*)

Wykonywanie instrukcji przez procesor składa się z kilku etapów. W zależności od procesora może być ich różna ilość. Etapy, które musi wykonać procesor, to:

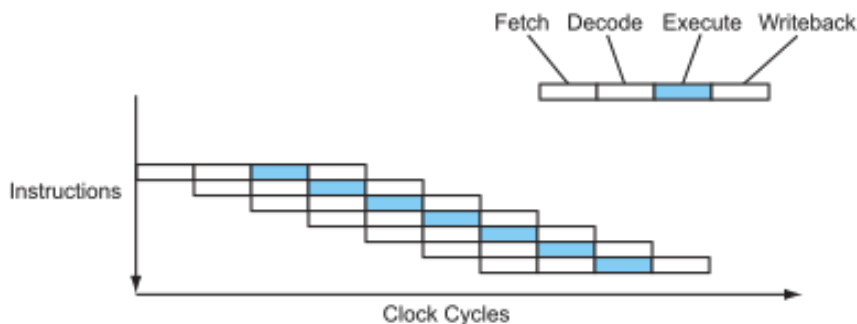
- Pobieranie instrukcji (ang. *IF* – *instruction fetch*),
- Dekodowanie instrukcji (ang. *ID* – *instruction decode*),
- Wykonywanie instrukcji (ang. *EX* – *execute*),
- Zapisywanie rezultatu (ang. *WB* – *writeback*).



Rys. 2.1. Przebieg wykonywania instrukcji przez procesor sekwencyjny [1].

Nowoczesne procesory, zamiast przetwarzać instrukcje sekwencyjnie, wykonują każdy etap równoległe, ale dla różnych instrukcji. Gdy rezultat jednej instrukcji jest zapisywany, to inna się wykonuje, inna jest dekodowana oraz kolejna jest pobierana. Proces ten nazywamy przetwarzaniem potokowym i przypomina on linię produkcyjną:

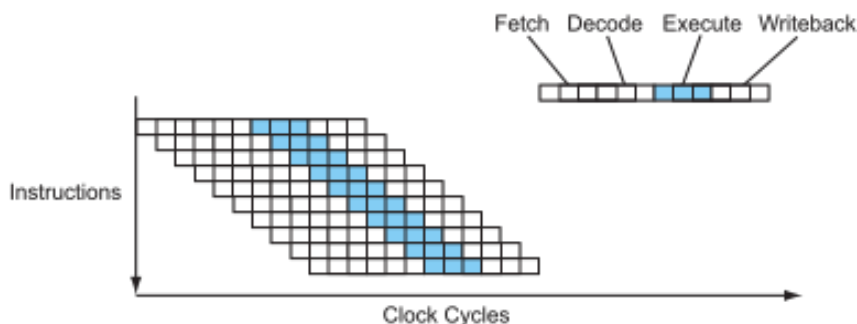
W takiej sytuacji procesor kończy wykonywać jedną instrukcję na cykl (w procesorze sekwencyjnym trwało to cztery cykle). Uzyskano zatem czterokrotne przyspieszenie, nie zmieniając taktowania procesora.



Rys. 2.2. Przepływ wykonywania instrukcji przez procesor potokowy [1].

2.1.1. Superpotokowość – zwiększenie liczby etapów potoków

Z uwagi na to, że prędkość wykonywania wielu instrukcji jest limitowana między innymi przez czas najwolniejszego etapu w potoku, etap taki można podzielić na mniejsze. W ten sposób wykonywanie kolejnej instrukcji będzie mogło rozpocząć się szybciej. Co prawda, z powodu podziału instrukcje mogą wykonywać się przez więcej cykli (większe opóźnienie), ale procesor wciąż będzie kończył jedną instrukcję na cykl (większa przepustowość), zatem wykona więcej instrukcji w tym samym czasie.



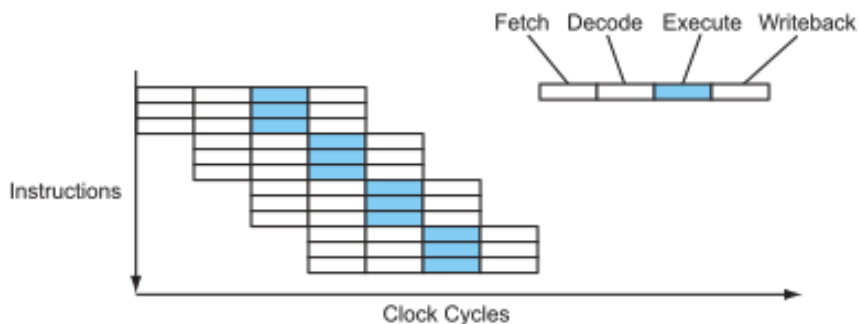
Rys. 2.3. Przepływ wykonywania instrukcji przez procesor superpotokowy [1].

2.1.2. Superskalarność

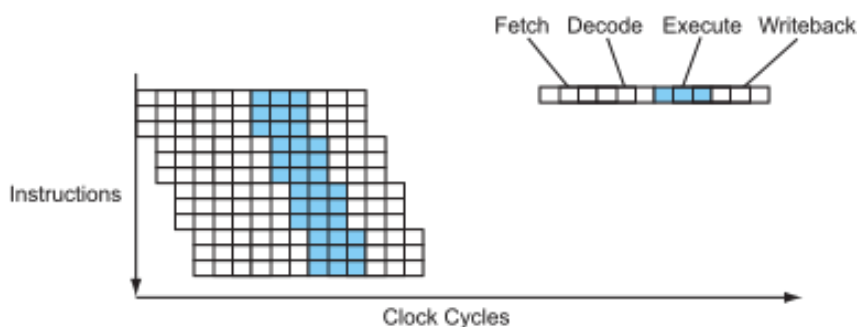
Kolejnym rozszerzeniem potoku jest zwiększenie liczby skalarnych jednostek wykonawczych. Dzięki temu taki superskalarny potok jest w stanie wykonywać kilka instrukcji jednocześnie.

2.1.3. Superskalarność i superpotokowość

Procesory mogą być także jednocześnie superskalarne i superpotokowe. Praktycznie wszystkie obecne procesory są projektowane w ten sposób. Zazwyczaj nazywa się je po prostu superskalarnymi, ponieważ superpotokowość to w rzeczywistości potokowość z większą liczbą etapów w potoku [1].



Rys. 2.4. Przepływ wykonywania instrukcji przez procesor superskalarny [1].



Rys. 2.5. Przepływ wykonywania instrukcji przez procesor superskalarny i superpotokowy [1].

2.1.4. Zależności między instrukcjami

Przetwarzanie potokowe musi sprostać wielu problemom. Jednym z takich problemów są zależności między instrukcjami. Przykładowo:

```
a = b * 2;
d = a + e;
```

Druga instrukcja jest zależna od pierwszej, zatem nie można zacząć jej wykonywać do czasu, aż zostanie zapisany wynik pierwszej. W takim przypadku superskalarność nie zwiększy wydajności.

2.1.5. Gałęzie wykonania (ang. *branches*)

Kolejnym problemem dla potoku są instrukcje warunkowe. Przykładowo:

```
if (a == 3)
    b = c;
else
    b = d;
```

Co po przetłumaczeniu na instrukcje asemblera będzie wyglądać mniej więcej tak:

```
cmp a, 3 ; a == 3 ?
jne L1   ; skocz do L1, jeśli porównanie zwróciło fałsz
mov c, b ; przypisz b = c
jmp L2   ; skocz do L2
```

```
L1:    mov d, b ; przypisz b = d
L2:    ...
```

Aby nie marnować cennych cykli, procesory posiadają specjalną jednostkę odpowiedzialną za dynamiczną predykcję gałęzi (ang. *dynamic branch prediction*). Jej działanie polega na tym, że procesor próbuje przewidzieć – poprzez zapamiętywanie informacji w trakcie działania programu – która z gałęzi powinna zostać wykonana, i tę zaczyna wykonywać.

Wynik instrukcji danej gałęzi nie zostaje zapisany do czasu wykonania się instrukcji warunkowej. Wtedy okazuje się, czy procesorowi udało się przewidzieć i kontynuuje pracę, czy musi na chwilę ją przerwać i usunąć z potoku wszystkie instrukcje, które pochodziły ze źle przewidzianej gałęzi. Następnie wykonuje on instrukcje poprawnej gałęzi.

Pracę procesora może także wspomóc programista (a w zasadzie kompilator) poprzez tak zwane statyczne przewidywanie gałęzi (ang. *static branch prediction*). Kompilator może oznaczyć daną gałąź jako taką, którą program wykonuje częściej. Nowoczesne kompilatory (np. gcc czy clang) pozwalają skompilować program tak, żeby podczas jego działania zapisywane były informacje, które z gałęzi zostały wykonane częściej¹. Następnie takie dane można wykorzystać podczas kolejnej kompilacji programu.

Istnieją także instrukcje, które eliminują gałęzie. Wcześniejszy przykład można zapisać tak:

```
cmp    a, 3 ; a == 3 ?
mov    c, b ; przypisz b = c
cmovne d, b ; przypisz b = d, jeżeli ostatnie
           ; porównanie jest fałszem
```

Wprowadzono tutaj nową instrukcję `cmovne` (ang. *conditional move if not equal*). Instrukcja ta zapisuje swój wynik tylko wtedy, gdy warunek jest spełniony – w tym przypadku, jeżeli flaga ZF (ang. *zero flag*) w rejestrze stanu jest nieustawiona (jest ona ustawiana przez instrukcję `cmp`, gdy wartości są równe).

Zastosowanie nowej instrukcji z jednej strony skróciło kod, a z drugiej wyeliminowało problematyczne zjawisko predykcji gałęzi, które mogłoby spowodować zmarnowanie cykli procesora.

Wykorzystanie takiej instrukcji nie zawsze jest optymalizacją. W przypadku, gdy dana gałąź wykonania jest przewidywalna dla procesora, lepiej jest skorzystać z instrukcji skoku warunkowego. Jest tak ponieważ instrukcja przypisania warunkowego wprowadza więcej zależności między danymi.

2.2. Jednoczesna wielowątkowość

Jeżeli podczas wykonywania potoku procesor nie może wykonać kilku instrukcji niezależnie (na przykład ze względu na zależności między nimi), to potencjalnym źródłem instrukcji niezależnych mogą być inne wątki w tym samym programie lub inne programy. Jednoczesna wielowątkowość

¹Taka optymalizacja to z ang. *PGO – profile guided optimization* – czyli optymalizacja opierająca się na profilowaniu aplikacji.

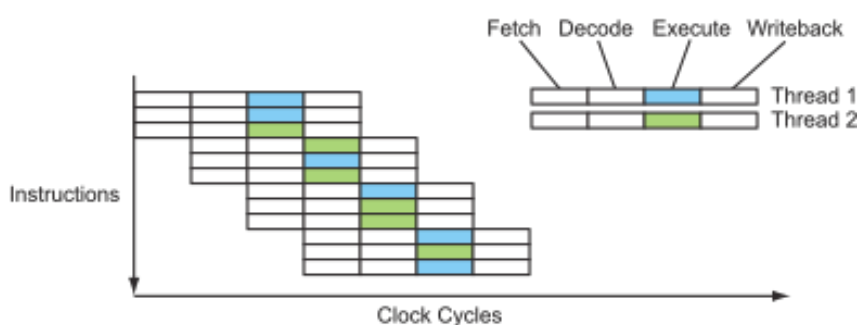
(ang. *SMT – Simultaneous multithreading*) jest techniką projektowania procesorów, która wykorzystuje taki typ zrównoleglania na poziomie wątków.

Idea ta polega na tym, żeby zastąpić „puste” etapy w potoku (na przykład czekające na wykonanie zależnej instrukcji) instrukcjami pochodzącymi z innych wątków wykonujących się w tym samym czasie, na tym samym rdzeniu procesora. Procesor korzystający z tej techniki w rzeczywistości korzysta z jednego fizycznego rdzenia. Z poziomu systemu operacyjnego prezentuje się on jako dwa lub więcej logiczne procesory.

Technika taka jest dużo wydajniejsza od wielordzeniowego procesora pod względem wykorzystania przestrzeni układu elektronicznego, kosztu wykonania, zużycia energii czy rozpraszania ciepła.

Implementacja SMT wymaga powielenia niewielkiej części komponentów procesora – między innymi rejestrów procesora oraz tablicy TLB (ang. *Translation Lookaside Buffer*) mapującej adresy pamięci wirtualnej na adresy fizyczne. Większe i bardziej złożone komponenty, takie jak pamięć podręczna (ang. *cache*), czy jednostki wykonania (ang. *execution units* czy *functional units*) są współdzielone między wątkami jednego rdzenia.

Takie rozwiązanie pozwala na ogólne zwiększenie wydajności procesora poprzez relatywnie mały wzrost kosztów produkcji jednego rdzenia.



Rys. 2.6. Przepływ wykonywania instrukcji przez procesor wykorzystujący SMT [1].

Wadą technologii SMT jest fakt, że może ona zmniejszyć wydajność przetwarzania poprzez zmniejszenie ilości zasobów dostępnych dla wątku (pamięci podręcznej i tablicy TLB). Ze względu na to może się okazać, że dany program bądź grupa programów będzie działać szybciej, gdy wyłączymy SMT.

Jedną z implementacji technologii SMT jest technologia firmy Intel zwana Hyper-Threading, która dzieli każdy z rdzeni procesora na dwa wątki. [1]

2.3. Instrukcje wektorowe – SIMD

Nowoczesne procesory posiadają specjalne rozszerzenia, dodające nowe, duże rejestry oraz instrukcje, pozwalające na wykonanie danej operacji na danych, znajdujących się w tych rejestrach (przykładowo, na zsumowanie czterech liczb z innymi czterema liczbami jednocześnie).

Instrukcje operujące na rejestrach zawierających kilka danych nazywa się instrukcjami wektorowymi i określa się je jako SIMD (ang. *single instruction multiple data*).

W zależności od rozszerzenia, rejestry wektorowe mają różny rozmiar:

- SSE (ang. *Streaming SIMD Extensions*) – 128 bit
- AVE (ang. *Advanced Vector Extensions*) – 256 bit

Rejestry te pozwalają na wykorzystanie różnych typów danych – na przykład, w rejestrach XMM dodanych w rozszerzeniu SSE, można przechowywać:

- 4 liczby typu float (32-bitowe liczby zmiennoprzecinkowe)
- 2 liczby typu double (64-bitowe liczby zmiennoprzecinkowe)
- 16 liczb typu int8 (8-bitowe liczby całkowite)
- 8 liczb typu int16 (16-bitowe liczby całkowite)
- 4 liczby typu int32 (32-bitowe liczby całkowite)
- 2 liczby typu int64 (64-bitowe liczby całkowite)
- jedną liczbę typu int128 (128-bitowa liczba całkowita)

Wykorzystanie instrukcji wektorowych pozwala na uzyskanie dużego przyspieszenia. Kompilatory z najbardziej agresywną optymalizacją potrafią czasami zamienić kod posiadający rozgałęzienia – które potrafią być kłopotliwe z perspektywy wydajności – na taki, który ich nie ma oraz wykorzystuje instrukcje wektorowe. Przykład takiej optymalizacji został przedstawiony w przykładzie opisanym w sekcji 3.3.

2.4. Problem dostępu do pamięci

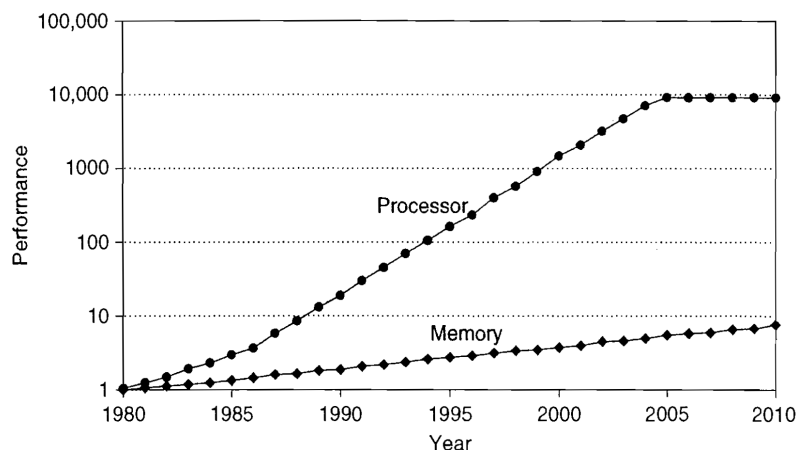
Około jedna czwarta instrukcji wykonywanych przez procesor to ładowanie danych z pamięci. Aby procesor mógł wydajnie działać, przesyłanie danych musi być dostatecznie szybkie.

Obecnie procesory są dużo bardziej skomplikowane niż 35 lat temu. W tamtych czasach taktowanie procesorów wynosiło tyle, co taktowanie szyny pamięci. W konsekwencji dostęp do pamięci był niewiele wolniejszy od dostępu do rejestrów procesora. Ta tendencja zmieniła się we wczesnych latach 90., kiedy to zwiększano taktowanie procesora. Taktowanie szyny pamięci oraz wydajność kości RAM² nie rosły proporcjonalnie do prędkości procesora [2], co przedstawia rysunek 2.7.

Ze względu na powyższe stworzono pamięć podręczną procesora (ang. *cache*), do której ładowane są najczęściej wykorzystywane dane. Dostęp do takiej pamięci jest dużo szybszy od dostępu do RAM-u. Wynika to z dwóch faktów – po pierwsze, jest ona wykonana w technologii SRAM³, w przeciwieństwie

²ang. Random Access Memory – pamięć o dostępie swobodnym

³SRAM – ang. Static Random Access Memory – statyczna pamięć o dostępie swobodnym



Rys. 2.7. Wykres przedstawia lukę w wydajności na przestrzeni lat, mierzoną jako różnica w czasie pomiędzy odwołaniem do pamięci przez procesor (pojedynczy procesor lub rdzeń) a opóźnieniem w dostępie do pamięci DRAM, przy przyjęciu osiągnięć z roku 1980 jako podstawy [3].

do pamięci RAM, która jest pamięcią DRAM⁴. Po drugie, fizycznie znajduje się bliżej procesora niż RAM, więc dane muszą zostać przesłane na krótszą odległość.

2.4.1. Hierarchia pamięci

W komputerach osobistych wyróżniamy następującą hierarchię pamięci (od najszybszej do najwolniejszej):

- rejestry procesora – znajdują się wewnątrz każdego rdzenia procesora; to na nich procesor wykonuje obliczenia,
- pamięć podręczna procesora – może być jej kilka poziomów,
- pamięć RAM,
- pamięć zewnętrzna SSD/HDD.

W tej pracy skupiono uwagę głównie na pamięci podręcznej procesora, bo uwzględniając fakt jej istnienia i to, jak działa, programiści są w stanie pisać szybsze programy.

2.4.2. Pamięć podręczna procesora

Pamięć podręczna nie jest bezpośrednio dostępna dla programisty czy dla systemu operacyjnego, a zamiast tego całkowicie zarządza nią procesor. Jest ona wykorzystywana do tworzenia tymczasowych kopii danych, które prawdopodobnie będą używane w niedługim czasie przez procesor.

Proste obliczenia pozwalają wykazać, jak w teorii efektywna może być pamięć podręczna. Na potrzeby przykładu można założyć, że dostęp do głównej pamięci zajmuje 200 cykli procesora, a dostęp

⁴DRAM – ang. Dynamic Random Access Memory - rodzaj ulotnej pamięci półprzewodnikowej

do pamięci podręcznej – 15 cykli. Wtedy kod, który używa 100 elementów, każdy po 100 razy, spędzi 2 000 000 cykli na dostępie do pamięci, gdy pamięć podręczna nie jest dostępna, a tylko 168 500 cykli, gdy pamięć podręczna pomieści wszystkie dane. Fakt posiadania pamięci podręcznej zredukował ilość cykli o 91.5% [2].

2.4.3. Poziomy pamięci podręcznej

Obecnie procesory nie pobierają danych bezpośrednio z RAM-u. Zamiast tego, pobierają je z wbudowanej pamięci podręcznej. W zależności od procesora, ilość i rozmiar pamięci podręcznych może się różnić; obecnie są to najczęściej trzy poziomy pamięci podręcznej – L1, L2, L3. Procesor pobiera dane z pamięci podręcznej L1, która pobiera je z L2, a ta z kolei z L3. Dane do L3 pobierane są bezpośrednio z pamięci RAM.

Wielopoziomowa pamięć podręczna wynika z faktu, że im bliżej procesora fizycznie znajduje się pamięć podręczna oraz im mniejszy ma rozmiar, tym jest szybsza⁵. Pamięć podręczna pierwszego poziomu – L1 zazwyczaj ma rozmiar 8-64 kB oraz dzieli się na L1d (ang. *L1 data cache* – pamięć podręczna danych) oraz L1i (ang. *L1 instruction cache* – pamięć podręczna instrukcji). Fizycznie znajduje się ona wewnątrz każdego rdzenia. L2 przechowuje od kilkuset kilobajtów do kilku megabajtów danych i tak samo jak w przypadku L1, każdy rdzeń procesora posiada swoją pamięć L2. L3 natomiast ma rozmiar kilku do kilkudziesięciu megabajtów. W przeciwieństwie do L1 oraz L2, pamięć ta jest współdzielona między rdzeniami.

Transfer danych pomiędzy poziomami pamięci podręcznej oraz pamięcią RAM odbywa się w blokach o stałym rozmiarze – najczęściej 32 lub 64 bajty, nazywanych liniami cache (ang. *cache line*). Dzieje się tak dlatego, aby sprawdzenie, czy potrzebne dane znajdują się w pamięci podręcznej, nie było zbyt kosztowne [1].

Przykładowe hierarchie pamięci w nowoczesnych procesorach zostały przedstawione w tabelach 2.1 oraz 2.2.

⁵Szerzej opisane w sekcji 2.4.4

Tabela 2.1. Hierarchia pamięci w procesorach Core i*4 Haswell [1].

| Rodzaj pamięci | Rozmiar | Opóźnienie [cykle] | Lokalizacja fizyczna |
|----------------|---------|--------------------|---|
| L1 cache | 32 KB | 4 | wewnątrz każdego rdzenia |
| L2 cache | 256 KB | 11 | obok każdego rdzenia |
| L3 cache | 6 MB | 21 | współdzielone między wszystkimi rdzeniami |
| L4 E-cache | 128 MB | 58 | oddzielny układ eDRAM |
| RAM | 4+ GB | 117 | kości SDRAM DIMM na płycie głównej |
| Swap | 100+ GB | 10000+ | dysk HDD lub SSD |

Tabela 2.2. Hierarchia pamięci w procesorach Apple A8 w iPhone 6 [1].

| Rodzaj pamięci | Rozmiar | Opóźnienie [cykle] | Lokalizacja fizyczna |
|----------------|---------|--------------------|---|
| L1 cache | 64 KB | 4 | wewnątrz każdego rdzenia |
| L2 cache | 1 MB | 20 | obok dwóch rdzeni |
| L3 cache | 4 MB | 107 | obok kontrolera pamięci |
| RAM | 1 GB | 261 | kość SDRAM |
| Swap | N/A | N/A | stronicowanie oraz swapowanie nie są wykorzystywane przez iOS |

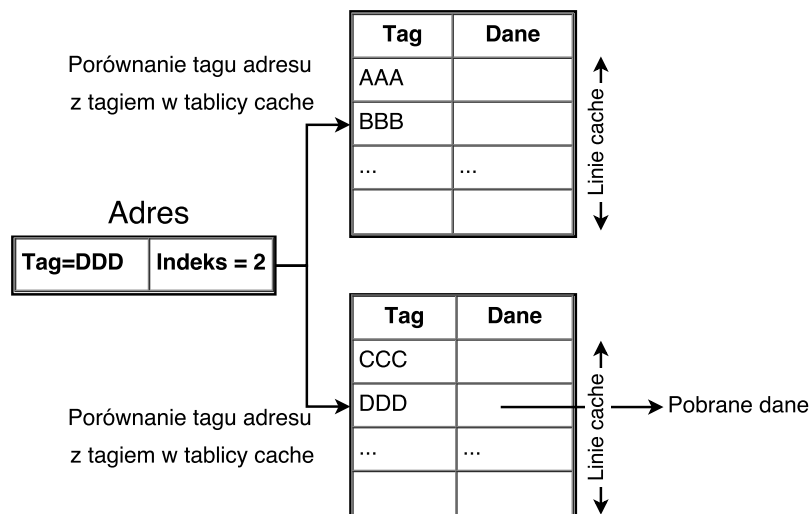
Efektywność pamięci podręcznej wynika stąd, że pisząc odpowiednio aplikację, dostęp do pamięci zajmie tylko kilka cykli zamiast kilkuset. Jest to możliwe dzięki temu, że w wielu programach zarówno kod programu, jak i dane są lokalne przestrzennie (ang. *spatial locality*) oraz czasowo (ang. *temporal locality*) [2, 4]:

- Lokalność przestrzenna – podczas korzystania z danej komórki pamięci występuje duże prawdopodobieństwo, że program będzie odwoływał się do pamięci o bliskiej adresacji względem poprzednich danych (na przykład iteracja po tablicy).
- Lokalność czasowa – podczas korzystania z danej komórki pamięci występuje duże prawdopodobieństwo, że program w krótkim czasie ponownie odwoła się do tych samych danych.

2.4.4. Implementacja pamięci podręcznej

Z punktu widzenia sprzętu pamięć podręczna stanowi kilka dwukolumnowych tabel. W pierwszej kolumnie przechowywane są tagi służące do adresowania, a w drugiej dane. Kolejne wiersze są kolejnymi liniami cache. Tag to górna część adresu danych, które szukane są w pamięci podręcznej. Dolna część adresu jest wykorzystywana jako indeks w tabelach pamięci podręcznej.

Sprawdzenie, czy dany adres znajduje się w pamięci podręcznej, to w praktyce porównanie, czy na danym indeksie w którejś z tabel znajduje się dany tag. Przykład tego został przedstawiony na rysunku 2.8.



Rys. 2.8. Sprawdzenie, czy dane znajdują się w pamięci podręcznej posiadającej dwie tablice.

Sytuacja znalezienia danych w pamięci podręcznej nazywana jest trafieniem (ang. *hit*) – wtedy to odpowiednie komórki przesyłane są do procesora. W przeciwnym razie dane wyszukiwane są w wyższych poziomach pamięci podręcznej lub pobierane z RAM-u. Sytuację niezalezienia danych w cache nazywa się chybieniem (ang. *miss*) i powoduje ona utratę wydajności procesora, ponieważ musi on poczekać na dane.

2.4.5. Asocjacyjność

Jak można wywnioskować z rysunku 2.8 pamięć podręczna nie przechowuje stricte najczęściej używanych danych, gdyż oznaczałoby to, że dane o dowolnych adresach można przechowywać w dowolnych liniach cache. Z perspektywy zasad lokalności takie rozwiązanie byłoby bardzo korzystne. Niestety, oznaczałoby to też utratę szybkiego dostępu do pamięci podręcznej, ze względu na potrzebę sprawdzenia wszystkich linii cache, czy dana komórka pamięci się w danej linii znajduje.

Zamiast tego, dany adres w pamięci może znajdować się w jednej z kilku linii cache. Implementuje się to w ten sposób, że pamięć podręczną danego poziomu dzieli się na kilka tablic. Dzięki temu sprawdzenie, czy dany adres znajduje się w cache polega na wykonaniu tylu porównań, ile jest tych tablic. Sytuacja taka została przedstawiona na rysunku 2.8.

Pamięć podręczną zaprojektowaną w ten sposób nazywa się pamięcią zbiorowo-skojarzeniową (inaczej zbiorowo-asocjacyjną – ang. *set-associative cache*). Nazwa pochodzi stąd, że sprawdzenie, czy dany adres znajduje się w pamięci, działa poprzez skojarzenie – to znaczy, każdy adres w pamięci RAM jest skojarzony ze zbiorem lokacji (linii cache) w pamięci podręcznej.

Z tego powodu, gdy wiele komórek pamięci mapuje się do tego samego indeksu, naprzemienny dostęp do nich będzie wolny, ponieważ procesor będzie łąadował je naprzemiennie do pamięci podręcznej z pamięci podręcznej wyższego poziomu lub z RAM-u. Taką sytuację nazywamy konfliktem (ang. *cache conflict*) lub też zaśmiecaniem (ang. *thrashing*) – ponieważ, pomimo wykorzystywania tych samych danych, główne założenia pamięci podręcznej nie przynoszą korzyści.

Rozwiązaniem tego problemu jest po części zwiększenie ilości tablic pamięci podręcznej. W ten sposób można ograniczyć efekt zaśmiecania kosztem utraty wydajności związanej z równoległym sprawdzaniem, czy dane znajdują się w którejś z kilku tablic.

W nowoczesnych procesorach pamięć podręczna instrukcji zazwyczaj jest w dużym stopniu skojarzeniowa, ponieważ opóźnienie, które wynika z dodatkowej logiki na równoległe sprawdzenie wielu tablic, jest ukryte przez pobieranie i buforowanie we wczesnych etapach potoku procesora.

Z drugiej strony, pamięć podręczna danych jest w mniejszym stopniu skojarzeniowa, aby zminimalizować opóźnienie związane z ładowaniem danych (które było istotnym powodem stworzenia pamięci podręcznej).

Większość procesorów posiada cztery tablice w zbiorowo-skojarzeniowej pamięci podręcznej, ale istnieją też takie, które posiadają dwie (na przykład Athlon, Athlon 64/Phenom, PowerPC G5 oraz Cortex-A15/A57). Są też takie, które posiadają ich 8 – PowerPC G4e, Pentium M oraz jego wielordzeniowi następcy.

Ostatnia „deska ratunku” przed odwołaniem się do pamięci RAM, czyli duża pamięć podręczna L2 lub L3, jest zazwyczaj także wysoko skojarzeniowa – zawiera 12 lub 16 tablic [1].

2.4.6. Tagi w pamięci cache

Projektując pamięć podręczną, jako tag można wykorzystać adresy pamięci fizycznej albo wirtualnej (opisanej w sekcji 2.5).

W przypadku wykorzystania adresów pamięci wirtualnej, problem będzie sprawiał fakt, że różne programy mogą używać tych samych adresów pamięci wirtualnej do mapowania innych adresów fizycznych. Aby to naprawić, pamięć podręczna musi zostać wyczyszczona (ang. *flushed*) podczas każdej zmiany kontekstu (ang. *context switch*)⁶.

Z drugiej strony, użycie adresów fizycznych jako tagów oznacza, że podczas sprawdzania, czy adres znajduje się w pamięci podręcznej, należy przeprowadzić translację adresu wirtualnego na fizyczny. Taka operacja spowalnia ów test.

Powszechnym sposobem jest zastosowanie adresów wirtualnych do indeksowania pamięci podręcznej oraz adresów fizycznych jako tagów. Mapowanie adresów wirtualnych na fizyczne – poprzez mechanizm *TLB lookup* (ang.) – może być wtedy wykonane równoległe z wyciągnięciem tagu o podanym

⁶Zmiana kontekstu to proces polegający na zapisie stanu danego procesu lub wątku i odczycie zapisanego stanu kolejnego. Dzięki temu wykonanie procesu lub wątku może zostać później wznowione z miejsca, w którym został on zatrzymany. W taki sposób wiele procesów może wykonywać się „równoległe” na jednym procesorze. Taką funkcjonalnością cechują się wielozadaniowe systemy operacyjne. [5]

indeksie z pamięci podręcznej, dzięki czemu będzie on szybciej dostępny i porównany z tagiem. Takie rozwiązanie nazywa się wirtualnie indeksowaną, fizycznie tagowaną pamięcią podręczną (ang. *virtually-indexed physically-tagged cache*) [1].

2.5. Pamięć wirtualna

Pamięć wirtualna jest mechanizmem zarządzania pamięcią, dostępnym we współczesnych systemach operacyjnych ogólnego przeznaczenia, który daje procesom wrażenie pracy w ciągłej przestrzeni adresowej. Polega on na mapowaniu adresów pamięci używanych przez proces, nazywanych adresami wirtualnymi, na adresy fizyczne w RAM-ie lub w pamięci zewnętrznej.

Dzięki temu rozwiązaniu, program działający w systemie widzi pamięć operacyjną, jakby w całości należała do niego. Jest w ten sposób odizolowany od innych procesów i nie może modyfikować należących do nich obszarów pamięci⁷. Ułatwia to znacząco pisanie programów, gdyż programiści nie muszą dbać o to, czy odwołują się do pamięci zajmowanej przez inne procesy, czy nie.

Wirtualna przestrzeń adresowa jest implementowana zarówno sprzętowo, jak i programowo – w procesorze znajduje się specjalna jednostka zarządzania pamięcią (ang. *MMU – Memory Management Unit*), która przeprowadza translację z adresów pamięci wirtualnej na adresy pamięci fizycznej, a system operacyjny odpowiada za wypełnianie tabelę tzw. stron pamięci.

W rzeczywistości procesory mapują adresy „stron” pamięci, czyli bloków o konkretnym rozmiarze. Typowym rozmiarem strony w systemach 32-bitowych, jak i 64-bitowych jest 4 kB. Wartość tę da się zmienić – przykładowo, w systemach opartych na jądrze Linux, wymaga to rekompilacji jądra systemu.

Aby każde odwołanie do zmiennej w programie nie wymagało translacji adresu wirtualnego na fizyczny, przetłumaczone adresy stron zapisywane są w specjalnej pamięci podręcznej TLB (ang. *translation lookaside buffer*) [2].

2.6. Wczesne pobieranie

Wczesne pobieranie (ang. *prefetching*) jest techniką „ukrywania” opóźnień związanych z dostępem do pamięci. Polega ono na tym, że dane pobierane są z pamięci na krótko przed tym, gdy procesor ich faktycznie potrzebuje. Wyróżniamy dwa rodzaje wczesnego pobierania:

- Sprzętowe – procesory posiadają specjalne układy, które monitorując schematy dostępu do danych, przewidują, które dane kolejno będą używane i pobierają je do procesora. Proces ten jest automatyczny, programista nie ma na niego bezpośredniego wpływu. Układy takie nazywane są jednostkami wczesnego pobierania (ang. *prefetcher*).

⁷Oczywiście systemy operacyjne dostarczają specjalny mechanizm pamięci współdzielonej, dzięki której procesy mogą się ze sobą komunikować.

- Programowe – procesory posiadają specjalne instrukcje, informujące procesor, że pamięć o danym adresie będzie niedługo potrzebna – wadą tego podejścia jest narzut na wykonanie instrukcji.

Większość dzisiejszych procesorów posiada kilka układów wczesnego pobierania – na przykład procesory firmy Intel z rodzin Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell oraz Broadwell posiadają ich cztery – dwa dla pamięci podręcznej pierwszego poziomu oraz dwa dla drugiego poziomu [6, 7].

2.7. Języki programowania a ułożenie danych

Poniżej przedstawiono kilka faktów na temat możliwości ułożenia danych w kilku najpopularniejszych językach programowania według indeksu Tiobe z grudnia 2015 [8].

- Python, PHP, Perl, Ruby, JavaScript – jako języki interpretowane, czyli programy działające jak maszyny wirtualne, nie dają wielu możliwości. Niektóre z nich mają mechanizmy pozwalające na przykład zmniejszyć ilość pamięci zajmowanej przez obiekty zdefiniowanego typu (np. mechanizm `__slots__` w języku Python [9]).
- Java, C#, Visual Basic .NET – programy w tych językach są uruchamiane wewnątrz maszyny wirtualnej oraz posiadają mechanizm automatycznego zarządzania pamięcią. Z perspektywy tworzenia kolekcji obiektów, nie ma pewności, czy będą one ułożone kolejno w pamięci. Co ciekawe, jednym z etapów mechanizmu odśmiecania (ang. *garbage collector*) jest kompaktowanie obiektów – czyli przemieszczanie ich w celu defragmentacji pamięci [10, 11]⁸.
- C, C++ – dzięki bezpośredniej możliwości operowania na wskaźnikach oraz braku jakiegokolwiek maszyny wirtualnej, programista ma największy wpływ na ułożenie danych w pamięci.

2.8. Wyrównanie danych

Wyrównanie danych to sposób ułożenia oraz dostępu do danych w pamięci. Zmienna jest „naturalnie wyrównana”, jeśli znajduje się pod adresem, który jest wielokrotnością jej rozmiaru. Dla przykładu: 32-bitowa zmienna jest naturalnie wyrównana, jeżeli znajduje się pod adresem, który jest wielokrotnością 4 (ponieważ 32 bity to 4 bajty).

Procesory oparte o niektóre architektury (np. bazujące na architekturach Alpha, IA-64, MIPS oraz SuperH) nie pozwalają na odczyt niewyrównanych danych. Inne zaś zezwalają na taki dostęp, często tracąc przy tym na wydajności [13].

⁸W przypadku języka C# istnieje różnica między strukturami oraz klasami – instancje struktur są tworzone na stosie. Elementy tablicy struktur będą ułożone w ciągłym obszarze pamięci. Zarówno w Javie, jak i w C# tablica obiektów klas przechowuje w ciągłym obszarze pamięci referencje (wskaźniki) do właściwych elementów. Te mogą być dowolnie ułożone w pamięci. Osoby zaangażowane w rozwój języka Java chcą tę sytuację poprawić i stworzyć tzw. *value objects* (ang.) [12], które są odpowiednikiem struktur w C#.

Kompilatory języków takich jak C czy C++ wyrównują dane w sposób przedstawiony w tabeli 2.3.

Tabela 2.3. Typowe wyrównanie danych na 32 oraz 64 bitowych procesorach na systemie Linux przez kompilator firmy Intel [14].

| Typ danych | Wyrównanie na procesorze 32-bit (w bajtach) | Wyrównanie na procesorze 64-bit (w bajtach) |
|------------------|--|--|
| char | 1 | 1 |
| short | 2 | 2 |
| int | 4 | 4 |
| long | 8 | 8 |
| float | 4 | 4 |
| double | 8 | 8 |
| long long | 8 | 8 |
| long double | 4 | 16 |
| dowolny wskaźnik | 4 | 8 |

2.8.1. Wyrównanie danych w strukturach

Gdy struktura przechowuje elementy różnych typów, kompilator wstawia nieużywaną pamięć pomiędzy nie, aby je wyrównać. Proces ten nazywany jest dopełnieniem (ang. *padding*). Zwiększa on wydajność dostępu do danych kosztem zwiększenia używanej pamięci.

Dopełnienie może także wystąpić na końcu struktury, w celu wyrównania jej samej (dzieje się tak w przykładzie zawartym w listingach 2.1, 2.2 oraz 2.3) [14].

Aby poznać faktyczne ułożenie danych przez kompilator, można skorzystać z odpowiedniej flagi kompilatora clang lub dodatku do debuggera gdb o nazwie pahole-gdb (dostęp w dniu 2015-12-15). Istnieje także dedykowana aplikacja pahole, lecz nie wspiera ona standardu c++11.

W listingach 2.2 oraz 2.3 przedstawiono ułożenie danych przez kompilator clang++-3.6 oraz g++-5.2.1 dla przykładu z listingu 2.1. Program został skompilowany na 64-bitowym systemie Ubuntu 15.10.

Listing 2.1. Program z przykładową strukturą danych.

```
1 struct S {
2     int a;
3     bool b;
4     int* c;
5     bool e;
6     int f;
7     bool g;
8     int h;
9     bool i;
10 };
11
12 int main() {
13     // Poniżej wykorzystano obiekt struktury oraz jego rozmiar.
14     // Instrukcje te są wymagane dla funkcjonalności wyświetlenia
15     // organizacji struktury przez clang oraz pahole-gdb.
16     S s;
17     return sizeof(S);
18 }
```

Listing 2.2. Ułożenie danych oraz padding przez kompilator clang++-3.6. Pierwsza kolumna oznacza z ang. *offset* – przesunięcie względem początku struktury. Dodatkowe informacje o strukturze prezentowane są na końcu⁹.

```
$ clang++-3.6 -cc1 -fdump-record-layouts struct_S.cpp
```

```
*** Dumping AST Record Layout
0 | struct S
0 |   int a
4 |   _Bool b
8 |   int * c
16 |  _Bool e
20 |   int f
24 |  _Bool g
28 |   int h
32 |  _Bool i
   | [sizeof=40, dsize=40, align=8
   |   nvsize=40, nvalign=8]
```

⁹Sizeof – rozmiar struktury. Dsize – rozmiar danych (bez dopełnienia na końcu struktury). Align – wyrównanie obiektów struktury. Nvsize – rozmiar struktury bez uwzględnienia wskaźnika na tablicę funkcji wirtualnych (jeśli struktura takowy posiada). Nvalign – wypełnienie bez uwzględnienia wskaźnika na tablicę funkcji wirtualnych.

Listing 2.3. Ułożenie danych oraz padding przez kompilator g++ 5.2.1. Dodatek pahole-gdb w przypadku linii zawierającej deklarację struktury wyświetla jej całkowity rozmiar. W przypadku linii zawierających pola, wyświetlane są dodatkowo przesunięcie względem początku struktury, rozmiar danego pola oraz – jeżeli występuje – informacja o rozmiarze „dziury”, czyli dopełnienia, aby następne pole było wyrównane.

```
$ g++ -std=c++11 -g struct_S.cpp -o exec
$ gdb --quiet ./exec
Reading symbols from ./exec...done.
(gdb) pahole S
/* 40      */ struct S {
/* 0      4 */   int a
/* 4      1 */   bool b
/* XXX 24 bit hole, try to pack */
/* 8      8 */   int * c
/* 16     1 */   bool e
/* XXX 24 bit hole, try to pack */
/* 20     4 */   int f
/* 24     1 */   bool g
/* XXX 24 bit hole, try to pack */
/* 28     4 */   int h
/* 32     1 */   bool i
}
```

Jak można zaobserwować, oba kompilatory ułożyły dane tak samo. W obu przypadkach, na dopełnienie trzech zmiennych typu `bool` zmarnowane zostały 72 bity.

Zmarnowane miejsce można zaoszczędzić, zmieniając kolejność pól w strukturze danych – ustawiając je od największego rozmiaru pola do najmniejszego ¹⁰.

Oszczędność miejsca w ten sposób zwiększa oczywiście wydajność – przykładowo, w przypadku wykorzystania tablicy struktur, więcej elementów zmieści się w jednej linii cache. Ułożenie takie zostało przedstawione w listingu 2.4 wraz z organizacją struktury – dla kompilatora clang w listingu 2.5 oraz g++ w listingu 2.6.

Listing 2.4. Wydajniejsze ułożenie danych. Pola zostały rozlokowane od największego do najmniejszego rozmiaru.

```
struct S {
    int* c;
    int a;
    int f;
    int h;
    bool b;
    bool e;
    bool g;
```

¹⁰Jest to generalna zasada. Są jednak przypadki, w których, pomimo nie zastosowania się do tej zasady, da się uzyskać optymalny rozmiar struktury (zawierający najmniejsze możliwe dopełnienie lub jego brak).

```

        bool i;
};

```

Listing 2.5. Ułożenie danych przez kompilator clang. Pola zostały rozlokowane od największego do najmniejszego rozmiaru.

```
$ clang++-3.6 -ccl -fdump-record-layouts struct_S_good.cpp
```

```

*** Dumping AST Record Layout
0 | struct S
0 |   int * c
8 |   int a
12 |  int f
16 |  int h
20 |  _Bool b
21 |  _Bool e
22 |  _Bool g
23 |  _Bool i
   | [sizeof=24, dsize=24, align=8
   |  nvsize=24, nvalign=8]

```

Listing 2.6. Ułożenie danych przez kompilator g++. Pola zostały rozlokowane od największego do najmniejszego rozmiaru.

```

$ g++ -std=c++11 -g struct_S_good.cpp -o exec
$ gdb --quiet ./exec
Reading symbols from ./exec...done.
(gdb) p/ahole S
/* 24      */ struct S {
/* 0      8 */   int * c
/* 8      4 */   int a
/* 12     4 */   int f
/* 16     4 */   int h
/* 20     1 */   bool b
/* 21     1 */   bool e
/* 22     1 */   bool g
/* 23     1 */   bool i
}

```

Tym razem w obu przypadkach struktura zajmuje 24 bajty i nie posiada żadnego dopełnienia.

2.8.2. Zmiana wyrównania

Istnieją uzasadnione przypadki, gdy należy zmienić wyrównanie struktury, bądź jej pól. Przykładem takim może być implementacja niskopoziomowych sterowników, gdy dane urządzenie wymaga odpowiedniej organizacji danych.

W ogólnym przypadku raczej się tego nie stosuje, ze względu na utratę wydajności, co zostało pokazane w rozdziale 2.8.

W kompilatorze Microsoft Visual C++, wyrównanie można zmienić poprzez dyrektywę preprocesora `#pragma pack`, a w gcc oraz clang oprócz `#pragma pack` można także wykorzystać specjalny atrybut `__attribute__((packed))`.

W listingach 2.7, 2.8 oraz 2.9 pokazano organizację danych upakowanej struktury z listingu 2.1.

Listing 2.7. Upakowana struktura danych.

```
1 // ustawienie wyrównania do jednego bajtu
2 #pragma pack(1)
3 struct S {
4     int a;
5     bool b;
6     int* c;
7     bool e;
8     int f;
9     bool g;
10    int h;
11    bool i;
12 };
13 // przywrócenie domyślnego ustawienia wyrównania
14 #pragma pack()
15
16 int main() {
17     S s;
18     return sizeof(S);
19 }
```

Listing 2.8. Upakowana organizacja danych przez kompilator clang.

```
$ clang++-3.6 -cc1 -fdump-record-layouts struct_S_packed.cpp
```

```
*** Dumping AST Record Layout
0 | struct S
0 |   int a
4 |   _Bool b
5 |   int * c
13 |  _Bool e
14 |   int f
18 |  _Bool g
19 |   int h
23 |  _Bool i
   | [sizeof=24, dsize=24, align=1
   |   nvsize=24, nvalign=1]
```


Listing 2.9. Upakowana organizacja danych przez kompilator g++.

```
$ g++ -g -std=c++11 struct_S_packed.cpp -o exec
$ gdb --quiet ./exec
Reading symbols from ./exec...done.
(gdb) pahole S
/* 24      */ struct S {
/* 0      4 */   int a
/* 4      1 */   bool b
/* 5      8 */   int * c
/* 13     1 */   bool e
/* 14     4 */   int f
/* 18     1 */   bool g
/* 19     4 */   int h
/* 23     1 */   bool i
}
```


3. Testy wydajności

Do testów wydajności wykorzystano przykłady napisane przez Joaquín M^a López Muñoz, omówione na konferencji `using std::cpp 2015` [15], z pewnymi modyfikacjami.

Główna modyfikacja została przedstawiona na listingu 3.1. Polega ona na zmianie funkcji mierzącej czas, tak, aby mierzyła ona tylko czas danej operacji – wywołania funkcji szablonowej `f()`, przekazywanej przez argument szablonowy `lambda`. Test wydajności polega na wykonaniu 50 iteracji, w których mierzony jest czas wykonania przekazanej funkcji. Wyniki kolejnych pomiarów są wypisywane na standardowe wyjście, a następnie uśredniane przez skrypt uruchamiający testy.

Pozostałe modyfikacje polegały głównie na zmianie interfejsu programu, tak, aby dało się uruchomić test wydajności dla odpowiedniego przypadku oraz rozmiaru danych (zamiast liczyć wszystko jednocześnie), dzięki czemu można dokonać analizy narzędziami takimi jak `perf`, `cahegrind` czy `Intel VTune Amplifier`.

Listing 3.1. Kod funkcji mierzącej czas. Na podstawie [15].

```
1  template <int iterations=50, typename Size, typename F>
2  void measure(Size n, F f) {
3      using namespace std::chrono;
4      volatile decltype(f()) res{0}; /* to avoid optimizing f() away */
5      high_resolution_clock::time_point t1, t2;
6
7      std::array<long double, iterations> timings;
8
9      for(auto i=0; i<iterations; ++i) {
10         t1 = high_resolution_clock::now();
11         res += f();
12         t2 = high_resolution_clock::now();
13         timings[i] = (long double)(t2 - t1).count() / (long double)n;
14     }
15     std::cerr << "result value used against compiler optimization - ignore this: ↵
16         " << res << std::endl;
17
18     for(int i=0; i<iterations-1; ++i)
19         std::cout << timings[i] << ", ";
20     std::cout << timings[iterations-1] << std::endl;
21 }
```

Przedstawione testy zostały przeprowadzone na dwóch różnych maszynach o parametrach przedstawionych w tabeli 3.1.

Tabela 3.1. Parametry maszyn na których zostały przeprowadzone testy wydajności.

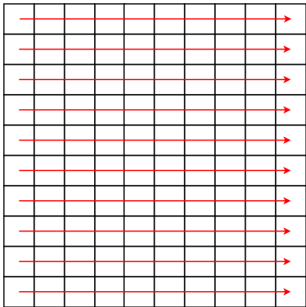
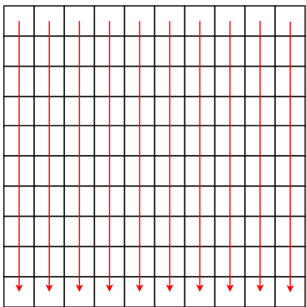
| Parametr | Maszyna 1 | Maszyna 2 |
|--|--------------------------------|------------------------|
| Model procesora | Intel® Core™ i7-4720HQ | Intel® Xeon® W3565 |
| Taktowanie | 2.60 GHz | 3.20 GHz |
| Taktowanie turbo | 3.6 GHz | 3.46 GHz |
| Turbo włączone | nie | nie |
| Liczba rdzeni | 4 | 4 |
| Hyper-Threading | tak | tak |
| Liczba wątków | 8 | 8 |
| Pamięć podręczna L1 danych (dla każdego rdzenia) | 32 kB | 32 kB |
| Pamięć podręczna L1 instrukcji (dla każdego rdzenia) | 32 kB | 32 kB |
| Pamięć podręczna L2 (dla każdego rdzenia) | 256 kB | 256 kB |
| Pamięć podręczna L3 (współdzielona między rdzeniami) | 6 MB | 8 MB |
| System operacyjny | Ubuntu 15.10 | Debian 7.8 (wheezy) |
| Wielkość strony pamięci | 4 kB | 4 kB |
| Wersja kompilatora gcc | 5.2.1 (Ubuntu 5.2.1-22ubuntu2) | 4.7.2 (Debian 4.7.2-5) |

3.1. Przejście po macierzy

W poniższym przykładzie przeanalizowany został problem zsumowania elementów macierzy [16]. Porównane zostały dwa sposoby przejścia macierzy, przedstawione w tabeli 3.2.

Wyniki testu wydajności przedstawiono na rysunku 3.1. Najlepszy wynik uzyskano wykorzystując najbardziej agresywną optymalizację – `-O3`. W każdym przypadku przejście wierszami było wydajniejsze od przejścia kolumnami. Przyczyna jest bardzo prosta – z uwagi na to, że elementy macierzy są ułożone w ciągłym obszarze pamięci wierszami, to dzięki wczesnemu pobieraniu oraz pamięci podręcznej, kolejne wiersze są pobierane do cache, podczas gdy procesor sumuje elementy danego wiersza (a w zasadzie danej linii cache).

Tabela 3.2. Testowany kod operacji sumowania elementów. Komórka na rysunku przedstawia element macierzy w pamięci RAM, która jest reprezentowana od lewej do prawej.

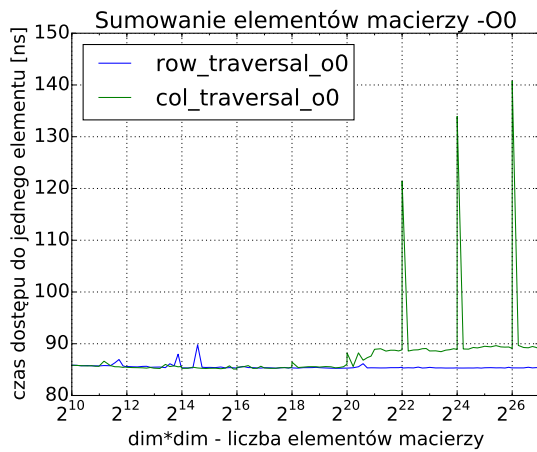
| Przejście macierzy wierszami | Przejście macierzy kolumnami |
|---|---|
| <pre> 1 long int sum = 0; 2 for(auto i=0; i<dim; ++i) 3 for(auto j=0; j<dim; ++j) 4 sum += matrix[i][j]; 5 return sum; </pre> | <pre> 1 long int sum = 0; 2 for(auto j=0; j<dim; ++j) 3 for(auto i=0; i<dim; ++i) 4 sum += matrix[i][j]; 5 return sum; </pre> |
|  |  |

Gdy rozmiar macierzy jest dostatecznie mały – do 2^{16} elementów (czyli 256 kB – rozmiaru pamięci podręcznej L2), wydajność przetwarzania wierszami jest o około 30-40% większa od przetwarzania kolumnami.

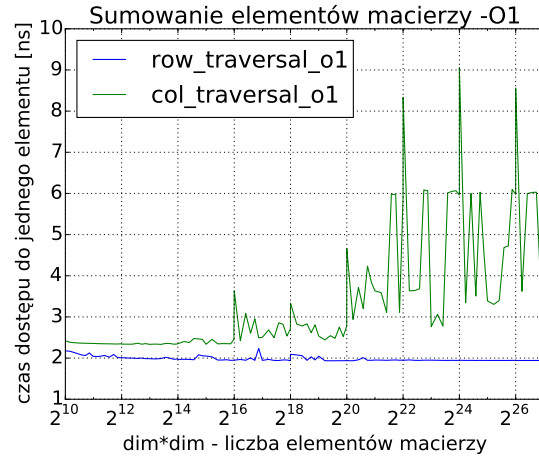
Po przekroczeniu tego rozmiaru, program stara się wykorzystać pamięć podręczną L3. Ze względu na fakt, że pamięć ta nie jest dostępna na wyłączność programu (ponieważ jest współdzielona między rdzeniami), to począwszy od 2^{16} elementów wykres dla przypadku przetwarzania kolumnami jest poszarpany. Po przekroczeniu rozmiaru 2^{21} elementów, wykres ten staje się jeszcze bardziej nieregularny. Przyczyną tego mogą być omówione w rozdziale 2.4.5 konflikty oraz zaśmiecanie pamięci podręcznej – może to być spowodowane konkurencją o miejsce w pamięci podręcznej L3 z innymi procesami, bądź po prostu niefortunne ułożenie adresów kolejnych kolumn macierzy (np. mapujących się do tych samych indeksów w cache).

W tabeli 3.3 przedstawiono wyniki programu perf dla wybranych rozmiarów macierzy. Ciekawymi przypadkami są punkty 2^{22} , 2^{24} oraz 2^{26} – charakterystyczne piki na wykresach wydajności. Mają one ponad 90% chybień do pamięci podręcznej L3, stąd prawdopodobnie występuje w nich największy efekt konfliktów lub zaśmiecania pamięci podręcznej.

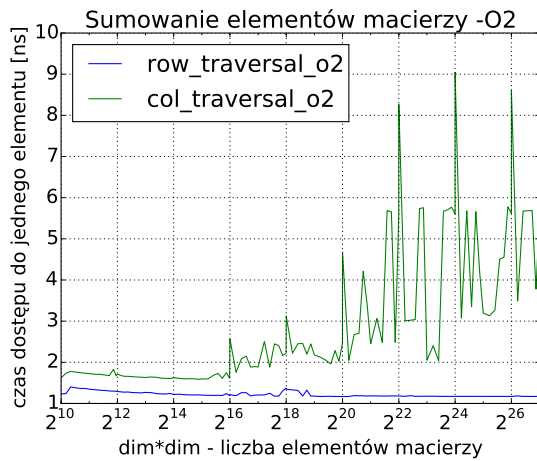
Na rysunku 3.2 zostały przedstawione wykresy pomiarów wykonanych na maszynie z procesorem Intel Xeon W3565. Wykresy te są dużo mniej poszarpane, co może wynikać z mniejszego obciążenia systemu. Na wykresie 3.2e można także zaobserwować trzy charakterystyczne piki omówione wcześniej. Być może ma to związek z rozmiarem strony pamięci (4 kB), gdyż odległości między kolejnymi elementami, po których program przechodzi wynoszą 8 kB, 16 kB oraz 32 kB. Przypadki te wymagają dogłębniejszej analizy.



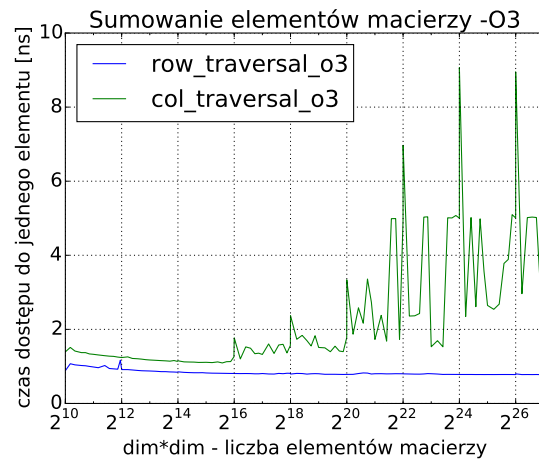
(a) Kompilacja z flagą -O0



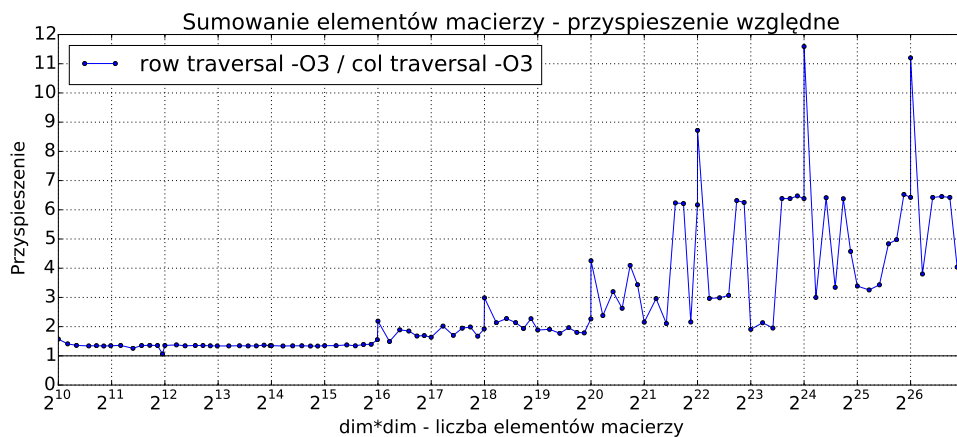
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2

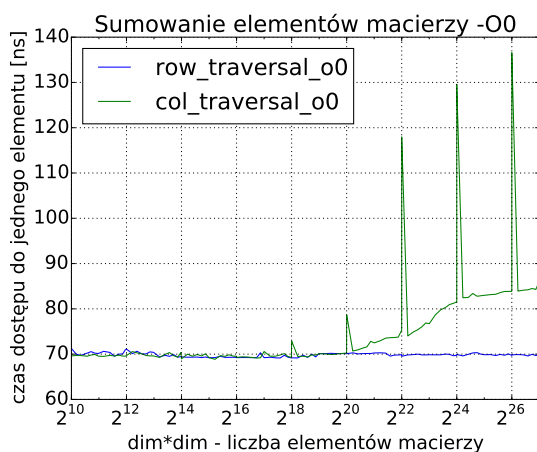


(d) Kompilacja z flagą -O3

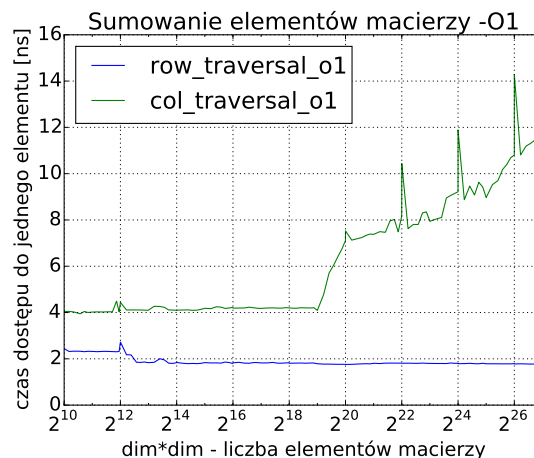


(e) Wydajność przejścia po wierszach względem przejścia po kolumnach dla flagi -O3.

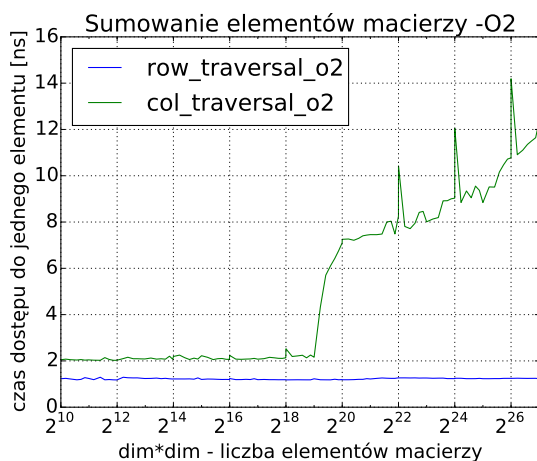
Rys. 3.1. Wyniki testów sumowania elementów macierzy (z sekcji 3.1), dla procesora Intel i7-4720HQ.



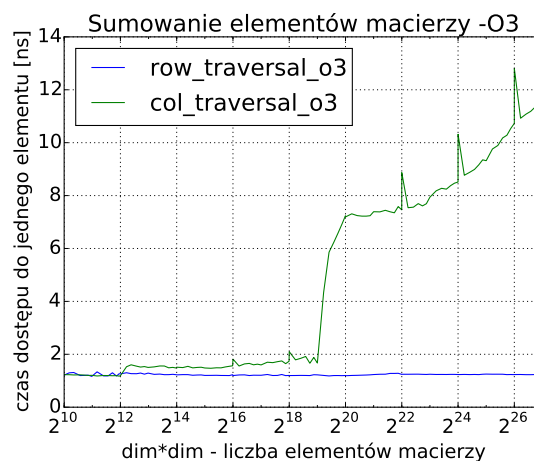
(a) Kompilacja z flagą -O0



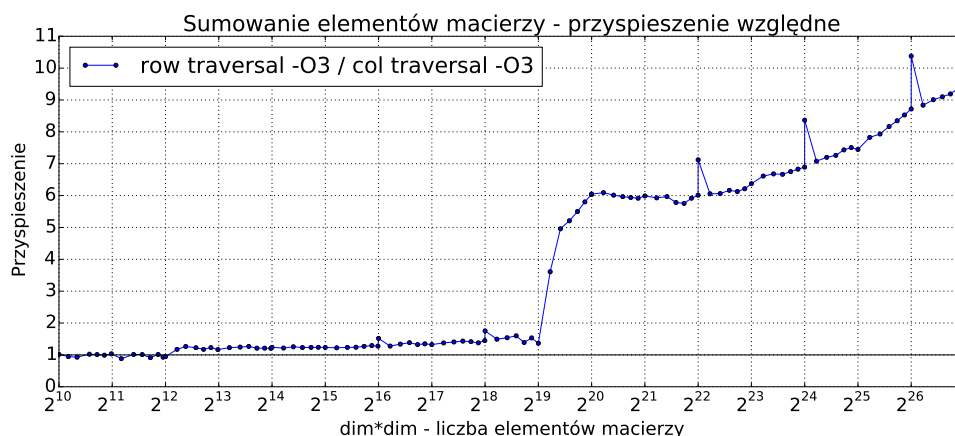
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2



(d) Kompilacja z flagą -O3



(e) Wydajność przejścia po wierszach względem przejścia po kolumnach dla flagi -O3.

Rys. 3.2. Wyniki testów sumowania elementów macierzy (z sekcji 3.1), dla procesora Intel Xeon W3565.

Tabela 3.3. Statystyki programu perf dla problemu z sekcji 3.1 w wersji z optymalizacją -O3. W kolejnych komórkach znajdują się uśrednione statystyki z 20 przebiegów programu (flaga --repeat programu perf) dla wybranych rozmiarów macierzy oraz rodzaju przejścia. Na szaro oznaczono przejście kolumnami, a na białą wierszami.

1 – liczba elementów macierzy oraz sposób przejścia po niej.

2 – średni czas dostępu do jednego elementu [ns].

3 – liczba wykonanych instrukcji.

4 – liczba chybionych gałęzi (ang. *branch misses*) [%].

5 – liczba chybień do cache L1d (danych) [%].

6 – liczba chybień do cache L3 [%].

| 1 | 2 | 3 | 4 | 5 | 6 |
|----------|------|----------------|------|-------|-------|
| 2^{20} | 0,6 | 528 426 650 | 0,05 | 6,26 | 12,76 |
| 2^{20} | 3,01 | 692 766 880 | 0,04 | 99,55 | 6,62 |
| 2^{22} | 0,62 | 2 129 768 639 | 0,02 | 6,35 | 53,32 |
| 2^{22} | 7,14 | 2 752 463 132 | 0,02 | 99,48 | 90,97 |
| 2^{23} | 0,61 | 4 150 161 533 | 0,01 | 6,15 | 41,94 |
| 2^{23} | 1,32 | 5 527 987 761 | 0,01 | 98,52 | 98,08 |
| 2^{24} | 0,61 | 8 614 182 665 | 0,01 | 6,11 | 38,47 |
| 2^{24} | 8,79 | 11 055 702 862 | 0,01 | 98,45 | 94,81 |
| 2^{25} | 0,61 | 17 209 919 277 | 0,01 | 6,16 | 46,21 |
| 2^{25} | 2,11 | 22 291 499 139 | 0,0 | 96,18 | 6,64 |
| 2^{26} | 0,61 | 34 382 549 263 | 0,0 | 6,43 | 44,93 |
| 2^{26} | 8,58 | 44 467 984 425 | 0,0 | 97,17 | 99,59 |

3.2. Tablica struktur a struktura tablic

Drugim zbadany problemem jest organizacja danych w strukturach. Rozpatrzono trzy przypadki – dostęp sekwencyjny, dostęp sekwencyjny z mniejszą strukturą danych oraz dostęp swobodny [17, 18, 19].

3.2.1. Dostęp sekwencyjny

W poniższym przykładzie porównano czas wykonania prostego algorytmu, który polega na zsumowaniu pozycji w trzech wymiarach (x, y, z). Na listingach 3.2 oraz 3.3 przedstawiono dwa podejścia do organizacji danych – „AoS” (z ang. *array of structures*) – tablica struktur oraz „SoA” (z ang. *structure of arrays*), czyli struktura tablic. Z uwagi na zmianę organizacji danych między przypadkami, sama implementacja algorytmu również uległa zmianie. Jego wynik jest oczywiście taki sam w obu przypadkach.

Listing 3.2. Organizacja danych oraz implementacja algorytmu dla AoS.

```
1 struct particle {
2     int x, y, z;
3     int dx, dy, dz;
4 };
5 ...
6 auto ps = create_particle_aos(n);
7
8 long int res = 0; // testowany algorytm
9 for(const auto &particle: ps)
10     res += particle.x + particle.y + particle.z;
11 return res;
```

Listing 3.3. Organizacja danych oraz implementacja algorytmu dla SoA.

```
1 struct particle_soa {
2     std::vector<int> x, y, z;
3     std::vector<int> dx, dy, dz;
4 };
5 ...
6 auto ps = create_particle_soa(n);
7
8 long int res = 0; // testowany algorytm
9 for (std::size_t i = 0; i < n; ++i)
10     res += ps.x[i] + ps.y[i] + ps.z[i];
11 return res;
```

W tabelach 3.4 oraz 3.5 przedstawiono organizację obu struktur danych w trzech kolejnych liniach cache pobieranych przez procesor.

Tabela 3.4. Organizacja tablicy struktur (AoS) w kolejnych liniach cache pobieranych przez procesor dla algorytmu z listingu 3.2. Wiersze oznaczają linie cache (każda ma 64B). Na niebiesko zaznaczone zostały dane wykorzystywane przez algorytm, na szaro te, które są pobierane niepotrzebnie.

| | | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| x1 | y1 | z1 | dx1 | dy1 | dz1 | x2 | y2 | z2 | dx2 | dy2 | dz2 | x3 | y3 | z3 | dx3 |
| dy3 | dz3 | x4 | y4 | z4 | dx4 | dy4 | dz4 | x5 | y5 | z5 | dx5 | dy5 | dz5 | x6 | y6 |
| z6 | dx6 | dy6 | dz6 | x7 | y7 | z7 | dx7 | dy7 | dz7 | x8 | y8 | z8 | dx8 | dy8 | dz8 |

Tabela 3.5. Organizacja struktury tablic (SoA) w kolejnych liniach cache. Wiersze oznaczają linie cache (każda ma 64B). Wszystkie dane znajdujące się w pobieranych liniach cache są wykorzystywane przez algorytm.

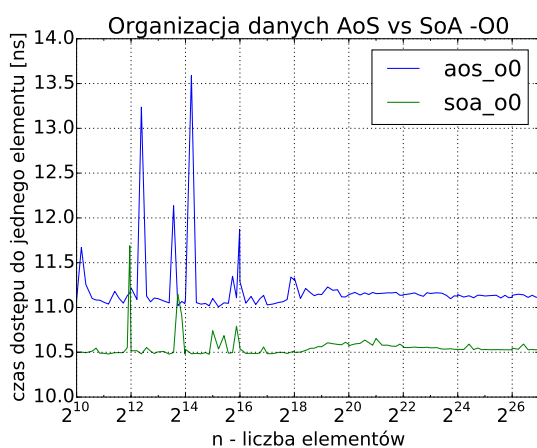
| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|
| x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | x10 | x11 | x12 | x13 | x14 | x15 | x16 |
| y1 | y2 | y3 | y4 | y5 | y6 | y7 | y8 | y9 | y10 | y11 | y12 | y13 | y14 | y15 | y16 |
| z1 | z2 | z3 | z4 | z5 | z6 | z7 | z8 | z9 | z10 | z11 | z12 | z13 | z14 | z15 | z16 |

Implementacja AoS (z listingu 3.2) iteruje po kolejnych elementach tablicy struktur – procesor widzi, że dostęp do danych odbywa się sekwencyjnie, więc pobiera kolejne dane. Problem tego rozwiązania polega na tym, że struktury (czyli wszystkie jej pola) są ułożone w pamięci sekwencyjnie. Zatem, aby zsumować pozycję 16 cząsteczek, procesor musi skorzystać z 6 linii cache (bo jest w stanie pobierać dane tylko w formie bloków o rozmiarze linii cache – czyli tutaj 64B).

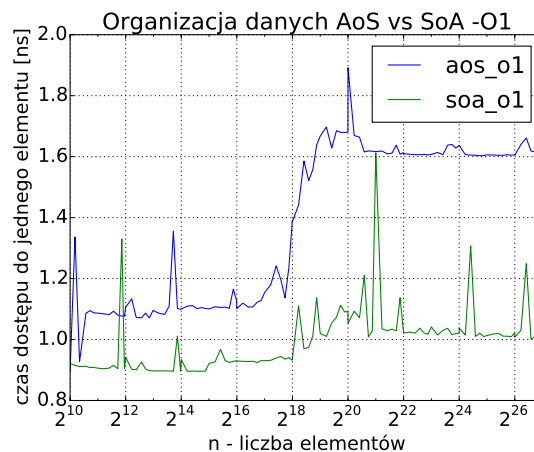
Dla podejścia SoA (z listingu 3.3) algorytm iteruje po trzech wektorach równocześnie. Dzięki zmianie organizacji struktury na taką, która zawiera potrzebne elementy w osobnych tablicach, procesor nie musi pobierać niepotrzebnych danych. Stąd, aby zsumować pozycję 16 elementów, wymagane są tylko 3 linie cache.

Dla najbardziej agresywnej optymalizacji – O3 – podejście SoA jest około dwukrotnie szybsze (w zależności od liczby elementów), co widać na rysunku 3.3. Wynika to z dwóch rzeczy. Po pierwsze, w podejściu AoS procesor pobiera więcej danych. Po drugie, mechanizm wczesnego pobierania w przypadku SoA pobiera kolejne elementy wektorów x , y oraz z równolegle. Efekt ten widać również w kolejnej sekcji (3.2.2) – gdzie ze struktury wyrzucono niewykorzystywane dane.

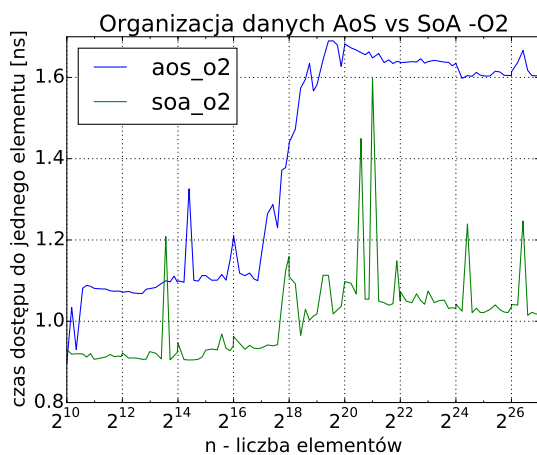
Na rysunku 3.4 przedstawiono wyniki testów dla procesora Intel Xeon W3565. Jak można zauważyć, uzyskane przyspieszenia są bardzo zbliżone.



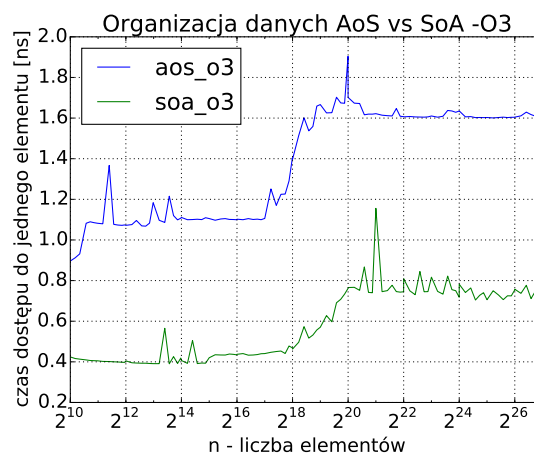
(a) Kompilacja z flagą -O0



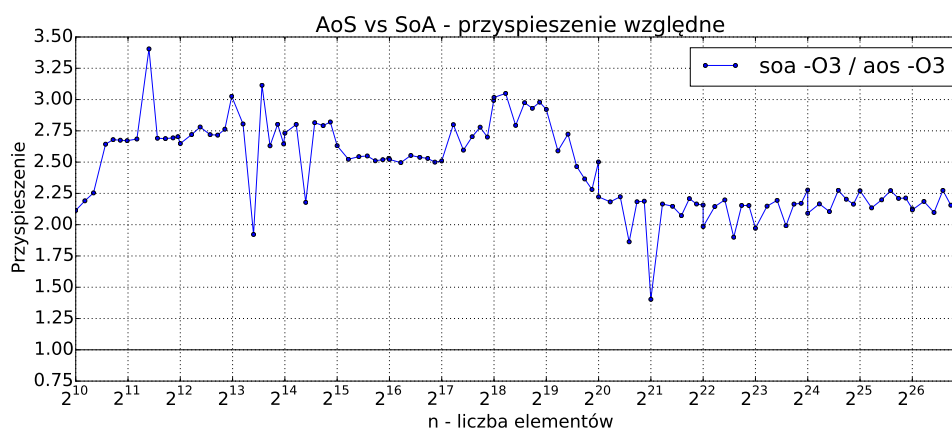
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2

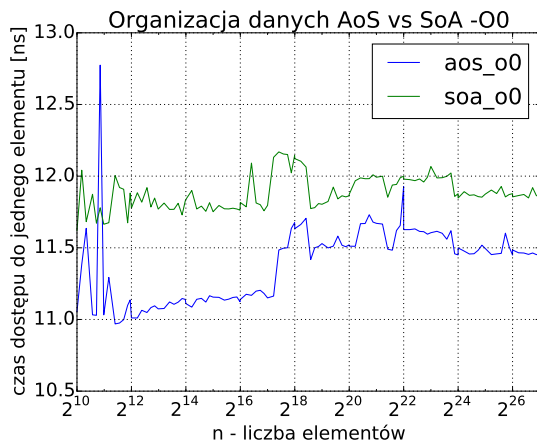


(d) Kompilacja z flagą -O3

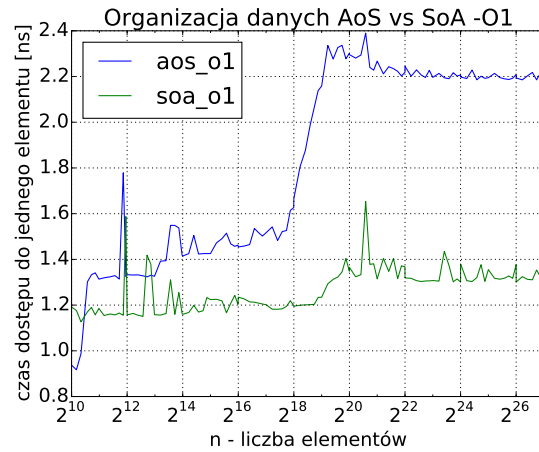


(e) Wydajność SoA względem AoS dla flagi -O3.

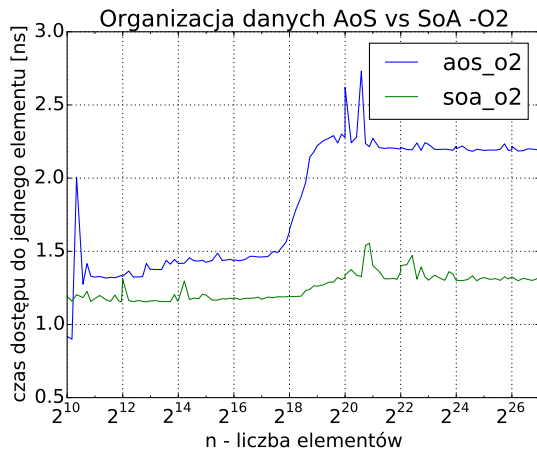
Rys. 3.3. Wyniki testów AoS vs SoA (z sekcji 3.2.1), dla procesora Intel i7-4720HQ.



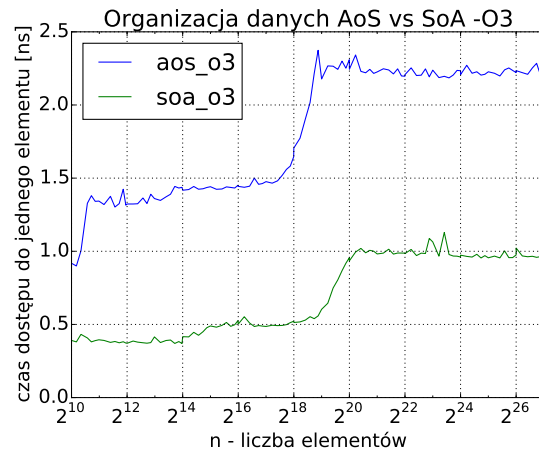
(a) Kompilacja z flagą -O0



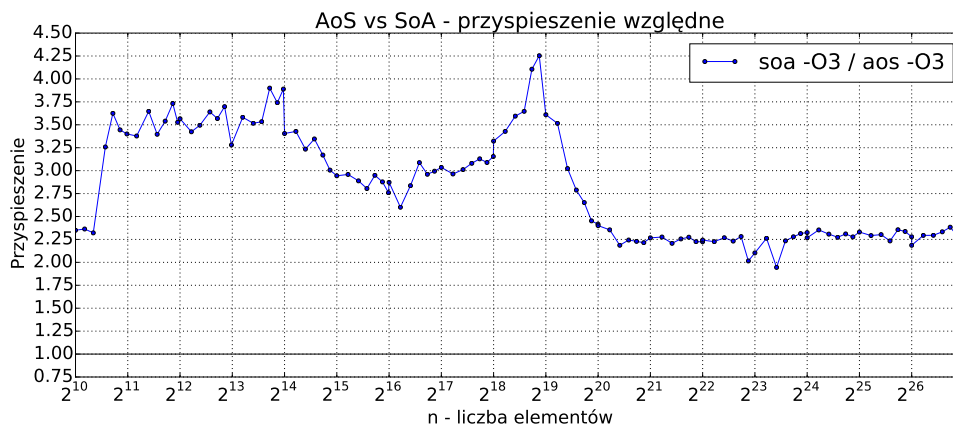
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2



(d) Kompilacja z flagą -O3



(e) Wydajność SoA względem AoS dla flagi -O3.

Rys. 3.4. Wyniki testów AoS vs SoA (z sekcji 3.2.1), dla procesora Intel Xeon W3565.

3.2.2. Dostęp sekwencyjny – „kompaktowa” struktura

W kolejnym przykładzie ze struktur AoS oraz SoA usunięto dane niewykorzystywane przez opisany w sekcji 3.2.1 algorytm (pola `dx`, `dy` oraz `dz`). Struktury te w „kompaktowej” wersji przedstawiono na listingach 3.4 oraz 3.5.

Listing 3.4. Organizacja danych kompaktowej struktury dla AoS.

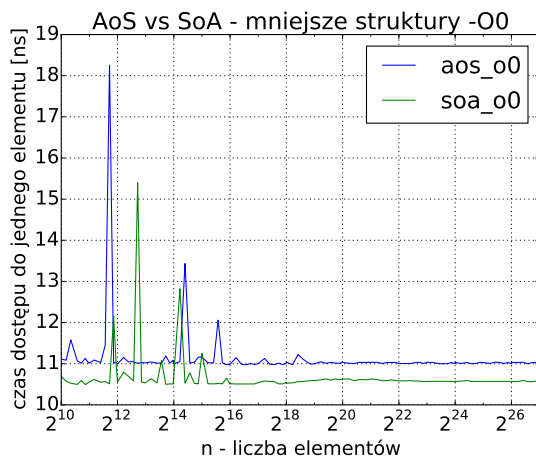
```
1 struct particle {  
2     int x, y, z;  
3 };
```

Listing 3.5. Organizacja danych kompaktowej struktury dla SoA.

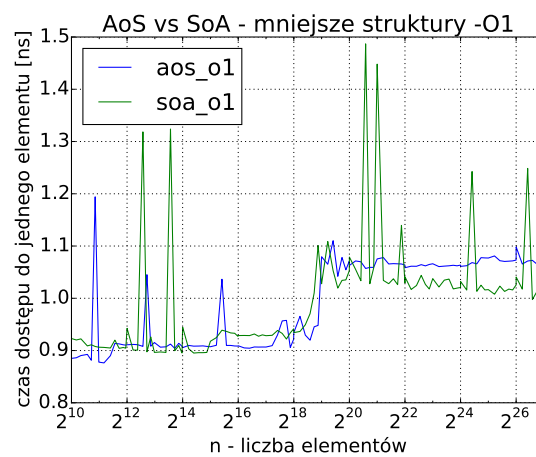
```
1 struct particle_soa {  
2     std::vector<int> x, y, z;  
3 };
```

Na rysunku 3.5 przedstawiono wyniki. Jak można zauważyć, dla najbardziej agresywnej optymalizacji, gdy ilość danych nie przekracza około 1 MB (czyli n równe 2^{18}), wciąż przyspieszenie jest znaczące – około dwukrotne. Po przekroczeniu tego progu, z uwagi na współdzielenie pamięci podręcznej L3, przyspieszenie nie jest tak duże – spada do około 30%, co widać na rysunku 3.5e.

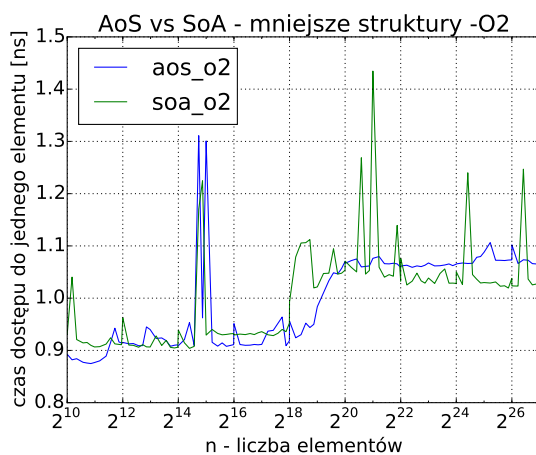
Dla procesora Intel Xeon W3565 sytuacja wygląda podobnie, co widać na rysunku 3.6. Zauważalną różnicą jest większy spadek przyspieszenia po przekroczeniu rozmiaru pamięci podręcznej L2 – czyli ponad 2^{14} elementów. Wynika on stąd, że procesor Intel Xeon W3565 posiada większą pamięć podręczną L3 (8 MB), niż procesor Intel i7-4720HQ (6 MB), przez co dostęp do niej jest wolniejszy.



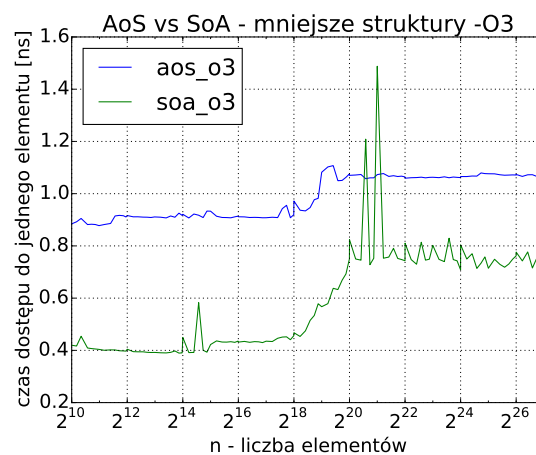
(a) Kompilacja z flagą -O0



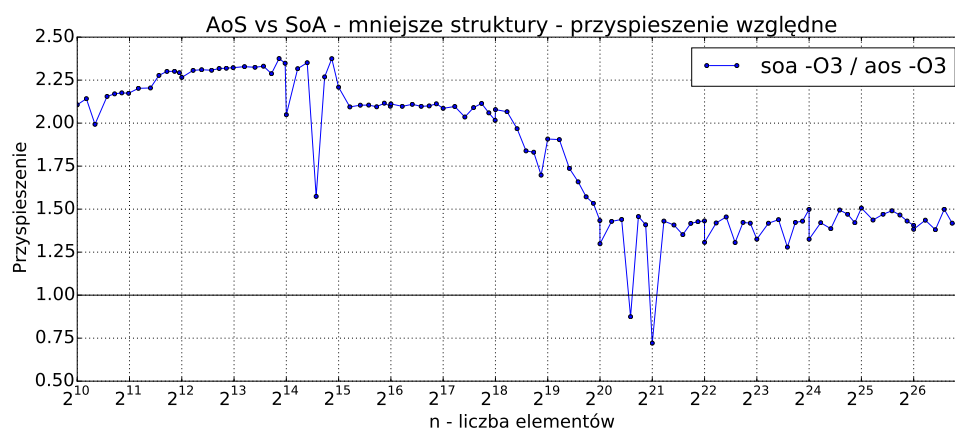
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2

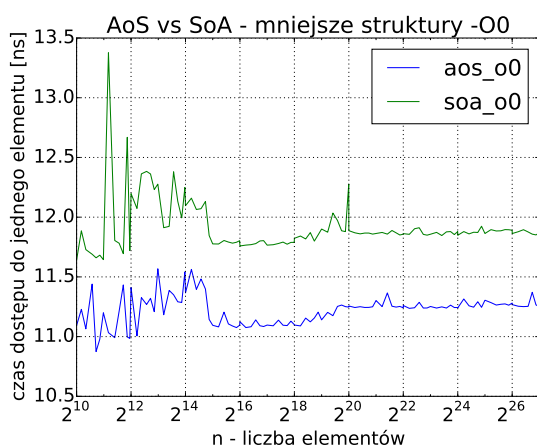


(d) Kompilacja z flagą -O3

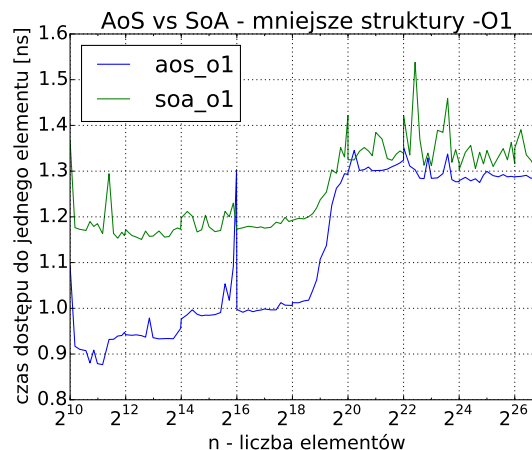


(e) Wydajność SoA względem AoS dla mniejszych struktur, dla flagi -O3.

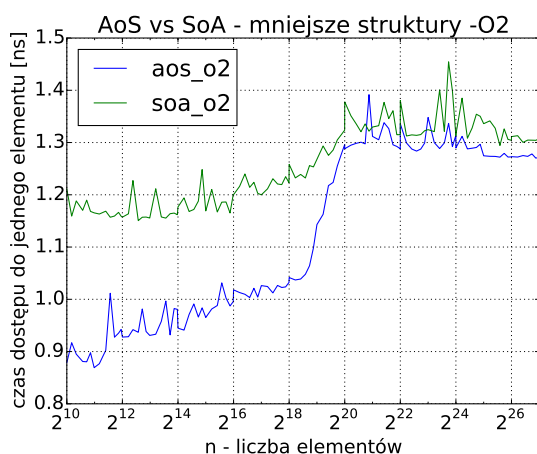
Rys. 3.5. Wyniki testów AoS vs SoA dla mniejszych struktur (z sekcji 3.2.2), dla procesora Intel i7-4720HQ.



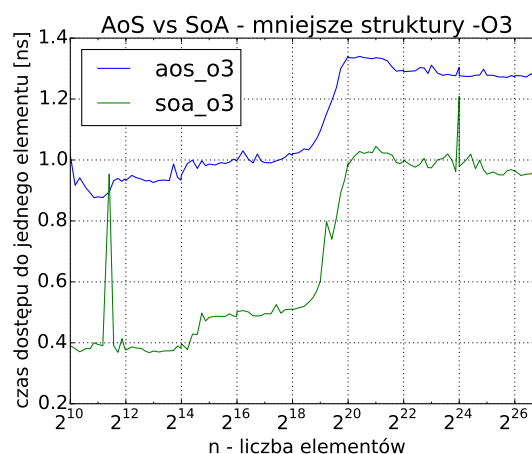
(a) Kompilacja z flagą -O0



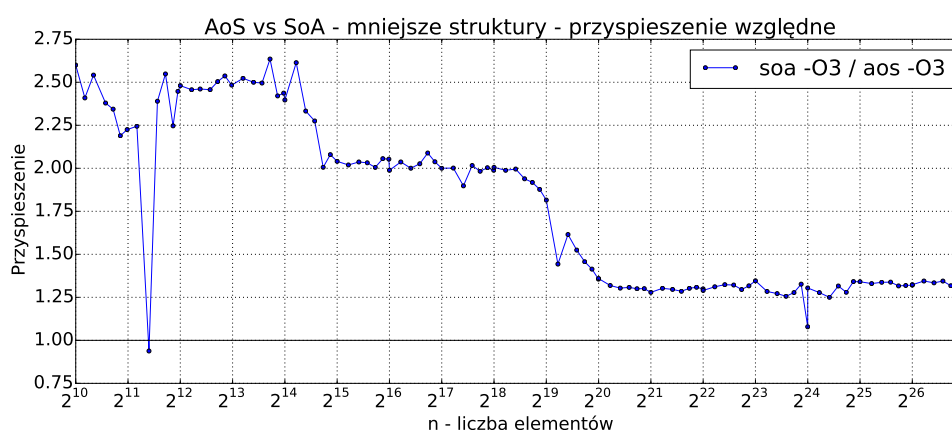
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2



(d) Kompilacja z flagą -O3



(e) Wydajność SoA względem AoS dla mniejszych struktur, dla flagi -O3.

Rys. 3.6. Wyniki testów AoS vs SoA dla mniejszych struktur (z sekcji 3.2.2), dla procesora Intel Xeon W3565.

3.2.3. Dostęp swobodny

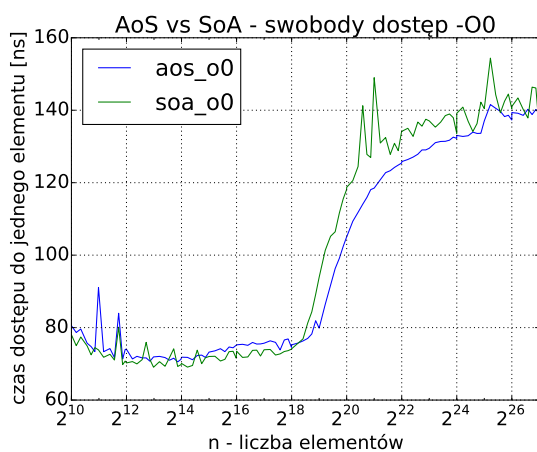
W poniższym przykładzie przeanalizowano swobodny dostęp do kompaktowych struktur danych AoS oraz SoA, przedstawionych wcześniej na listingach 3.4 oraz 3.5. W tym celu zmieniono algorytm sumowania elementów – przedstawiono go na listingu 3.6.

Listing 3.6. Zmodyfikowany algorytm z listingów 3.2 oraz 3.3 – dostęp odbywa się losowo, zamiast sekwencyjnie.

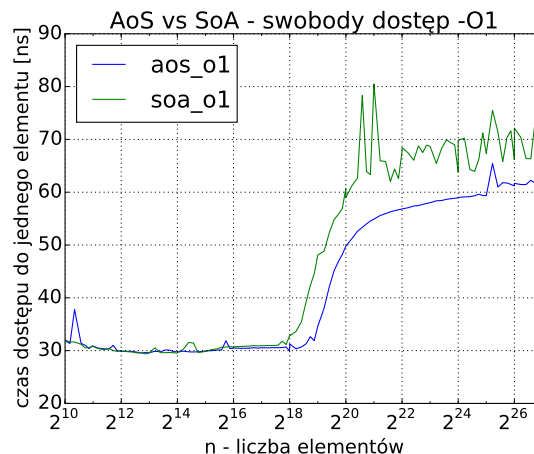
```
1  std::mt19937 gen;
2  std::uniform_int_distribution<> rnd(0, n - 1);
3  long int res = 0;
4  for (std::size_t i = 0; i < n; ++i) {
5      auto idx = rnd(gen);
6      // operacja sumowania zależna od podejścia - dla AoS:
7      res += ps[idx].x + ps[idx].y + ps[idx].z;
8      // lub w przypadku SoA:
9      res += ps.x[idx] + ps.y[idx] + ps.z[idx];
10 }
11 return res;
```

Na rysunkach 3.7 oraz 3.8 przedstawiono wyniki dla dwóch różnych procesorów. Jak można zauważyć, w przypadku swobodnego dostępu dla dużej liczby elementów, lepiej wykorzystać „standardowe” podejście obiektowe, czyli tablicę struktur (AoS), ponieważ dla SoA traci się około 20% wydajności, co widać na wykresach 3.7e oraz 3.8e.

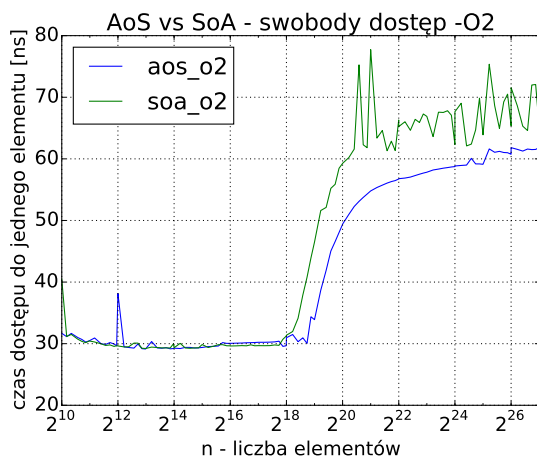
W przypadku gdy dane wciąż mieszczą się w pamięci podręcznej, a dostęp do danych jest powtarzalny, wydajność jest na zbliżonym poziomie.



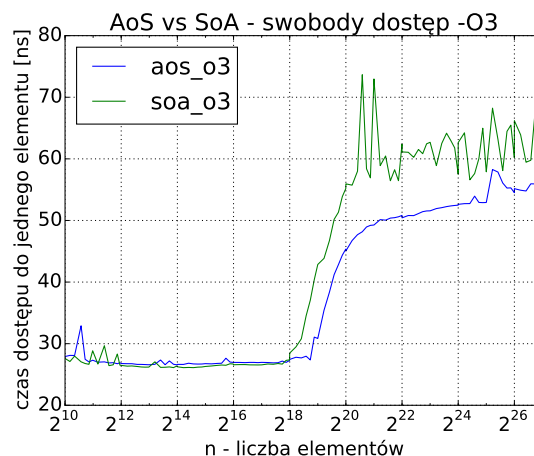
(a) Kompilacja z flagą -O0



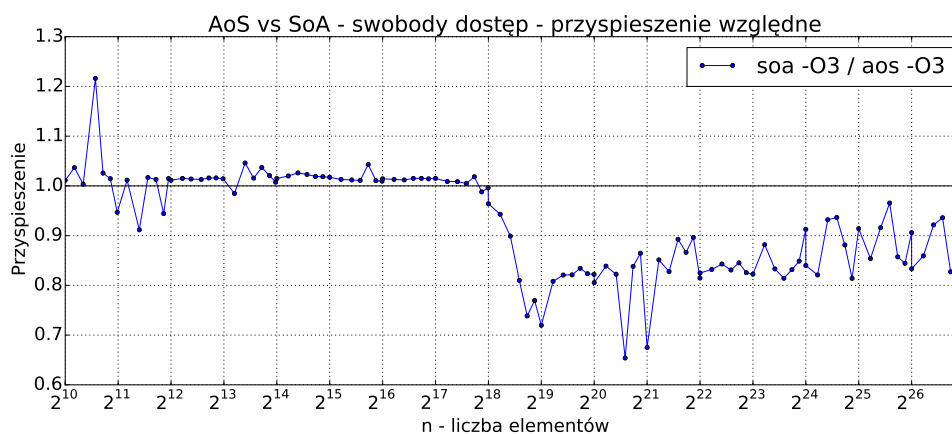
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2

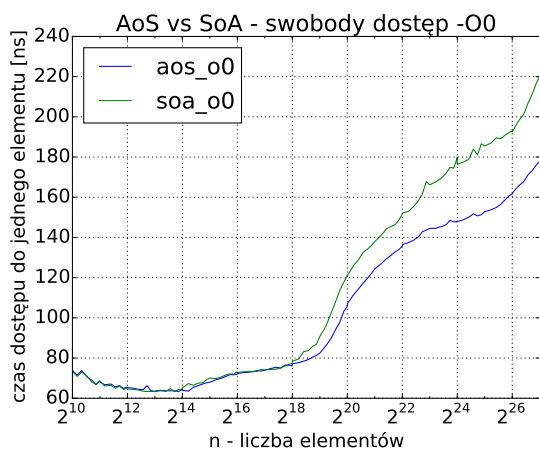


(d) Kompilacja z flagą -O3

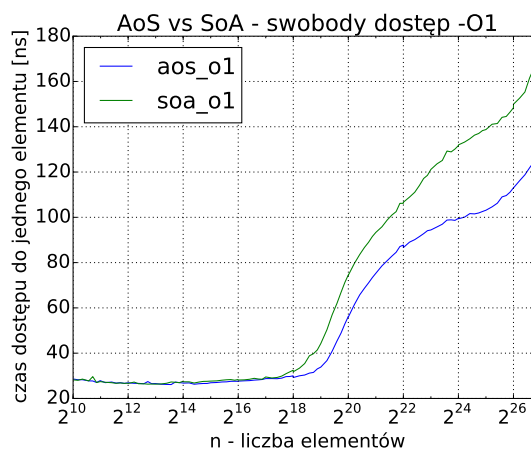


(e) Wydajność SoA względem AoS dla swobodnego dostępu, dla flagi -O3.

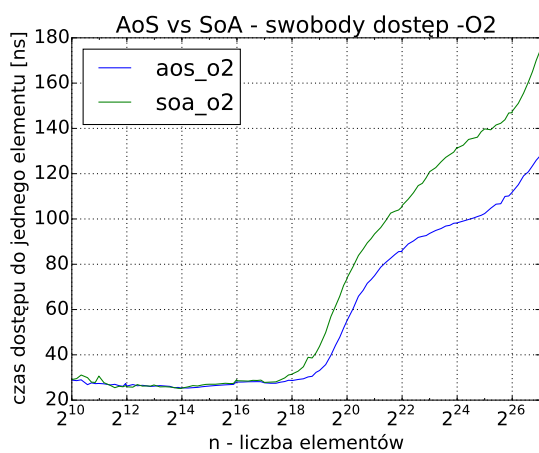
Rys. 3.7. Wyniki testów AoS vs SoA dla swobodnego dostępu (z sekcji 3.2.3), dla procesora Intel i7-4720HQ.



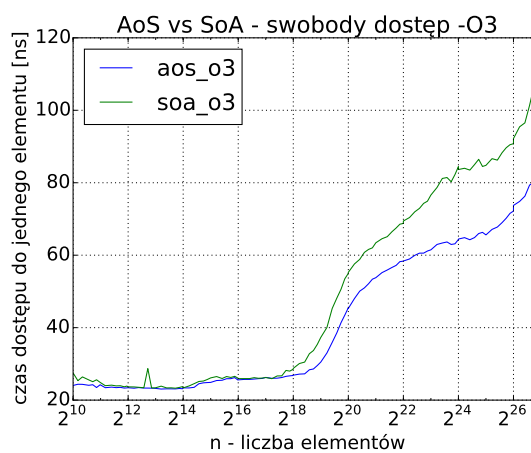
(a) Kompilacja z flagą -O0



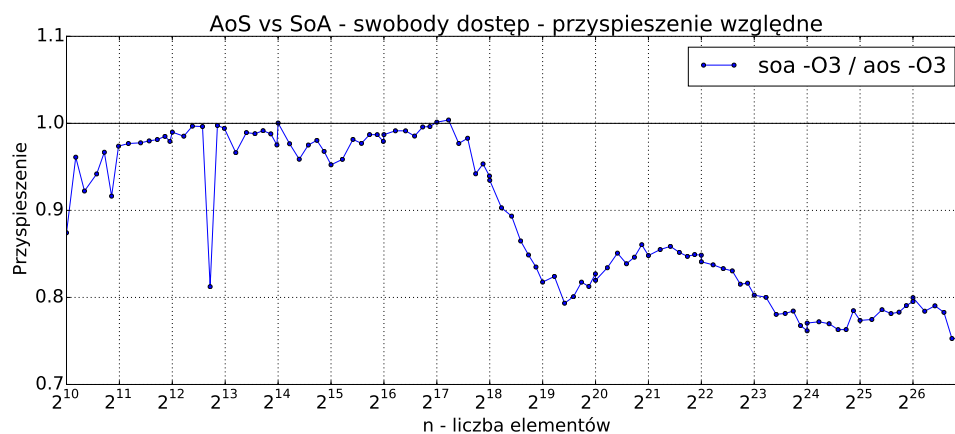
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2



(d) Kompilacja z flagą -O3



(e) Wydajność SoA względem AoS dla swobodnego dostępu, dla flagi -O3.

Rys. 3.8. Wyniki testów AoS vs SoA dla swobodnego dostępu (z sekcji 3.2.3), dla procesora Intel Xeon W3565.

3.3. Przetwarzanie warunkowe

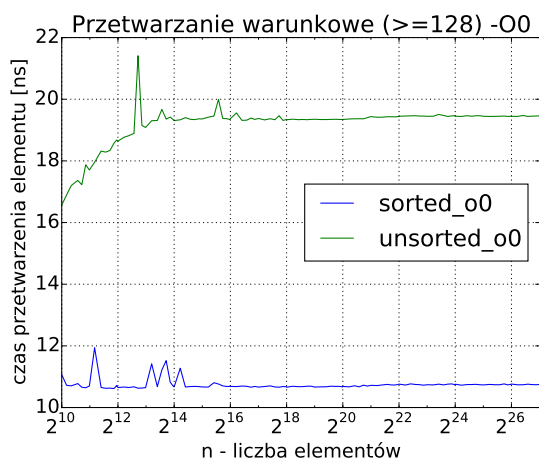
Przetwarzanie warunkowe polega na przetworzeniu danych, bazując na określonym warunku. Dla tego zagadnienia zbadano problem polegający na sumowaniu elementów wektora, które są większe lub równe 128 [20]. Wektor został wypełniony losowymi elementami od 0 do 255¹. Porównano dwa podejścia – gdy wektor jest nieposortowany oraz gdy został posortowany. Samo sortowanie nie zostało wliczone w czas wykonania. Testowana funkcja to lambda f z linii 11-17 listingu 3.7.

Listing 3.7. Kod sumowania elementów większych niż 128.

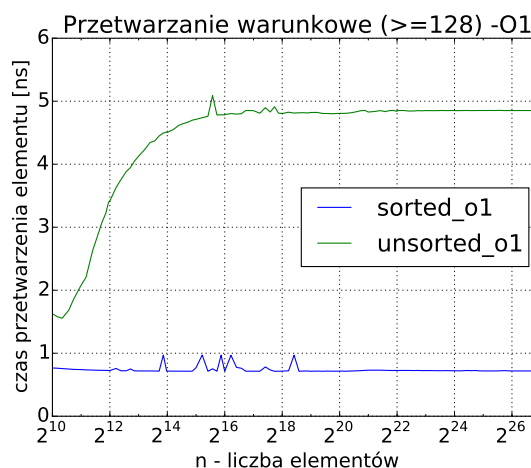
```
1 std::vector<int> v;
2
3 // wypełnienie wektora losowymi elementami z zakresu [0, 255]
4 std::mt19937 gen;
5 std::uniform_int_distribution<> rnd(0, 255);
6 v.reserve(n); // n - liczba elementów w wektorze
7 for (std::size_t i = 0; i < n; ++i)
8     v.push_back(rnd(gen));
9
10 // mierzona funkcja
11 auto f = [&]() {
12     long int res = 0;
13     for (int x: v)
14         if (x >= 128)
15             res += x;
16     return res;
17 };
```

Na rysunkach 3.9 oraz 3.10 zostały zaprezentowane wyniki testów wydajności dla dwóch omówionych procesorów, a w tabeli 3.6 statystyki uzyskane programem perf dla wybranego rozmiaru wektora dla kompilacji z flagą optymalizacji `-O2` dla procesora Intel i7 4720hq (przyczyna wybrania flagi jest opisana poniżej).

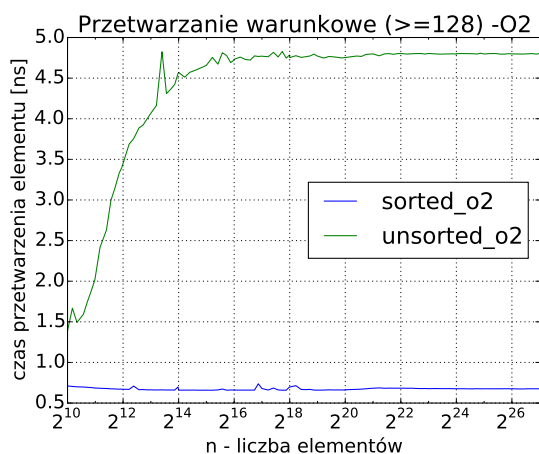
¹W obu przypadkach tymi samymi, ponieważ ziarno generatora liczb pseudolosowych pomiędzy wywołaniami programu jest takie samo.



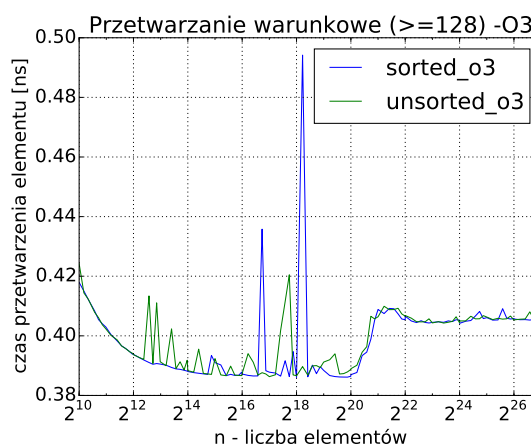
(a) Kompilacja z flagą -O0



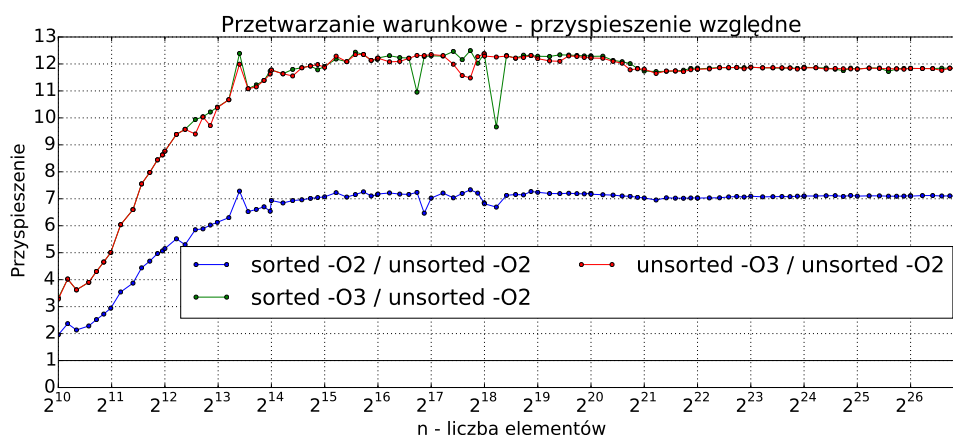
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2

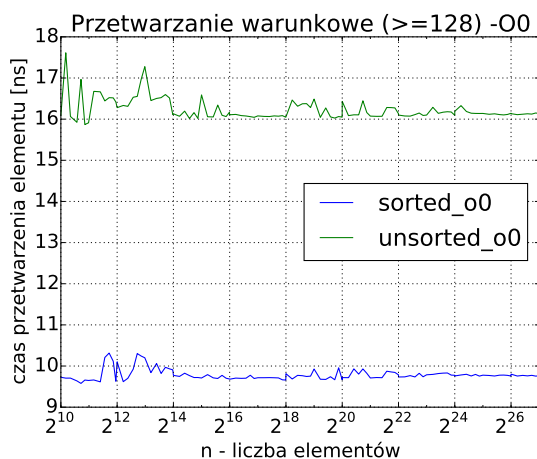


(d) Kompilacja z flagą -O3

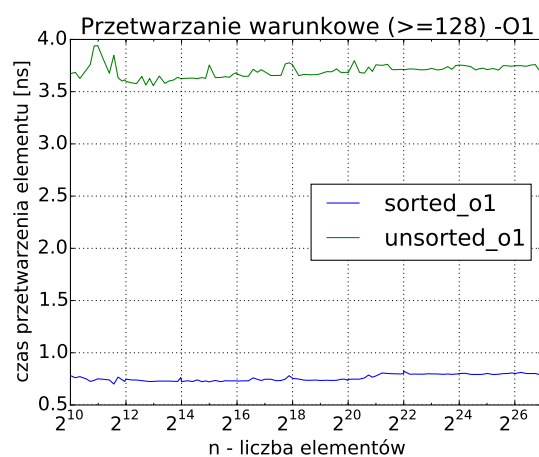


(e) Przyspieszenie uzyskane względem przypadku nieposortowanych danych przy kompilacji z flagą -O2.

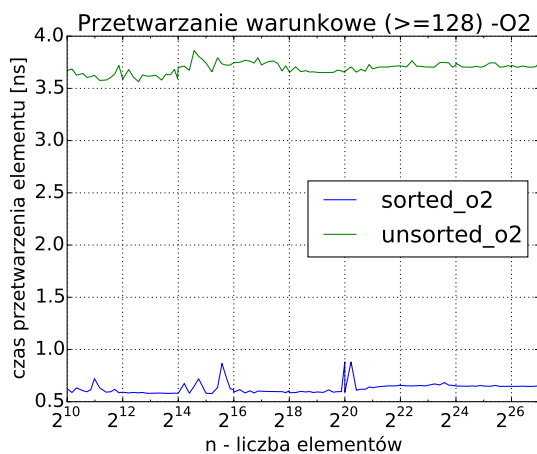
Rys. 3.9. Wyniki testów przetwarzania warunkowego (z sekcji 3.3), dla procesora Intel i7-4720HQ.



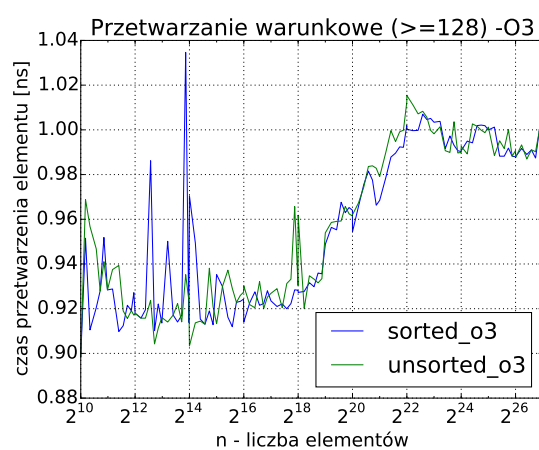
(a) Kompilacja z flagą -O0



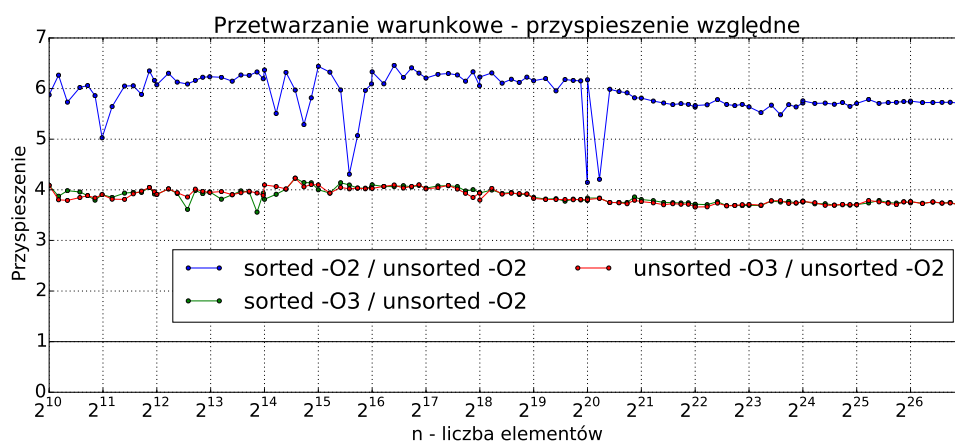
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2



(d) Kompilacja z flagą -O3



(e) Przyspieszenie uzyskane względem przypadku nieposortowanych danych przy kompilacji z flagą -O2.

Rys. 3.10. Wyniki testów przetwarzania warunkowego (z sekcji 3.3), dla procesora Intel Xeon W3565.

Tabela 3.6. Statystyki programu perf dla problemu z sekcji 3.3 w wersji z optymalizacją -O2. W kolejnych komórkach znajdują się uśrednione statystyki z 20 przebiegów programu dla wybranych rozmiarów wektora. Szare wiersze oznaczają posortowane dane.

1 – liczba elementów wektora.

2 – średni czas dostępu do jednego elementu [ns].

3 – liczba wykonanych instrukcji.

4 – liczba chybionych gałęzi (ang. *branch misses*) [%].

5 – liczba chybień do cache L1d (danych) [%].

6 – liczba chybień do cache L3 [%].

| 1 | 2 | 3 | 4 | 5 | 6 |
|-----------------|------|---------------|-------|------|-------|
| 2 ¹⁸ | 0,61 | 178 052 185 | 2,93 | 6,19 | 3,64 |
| 2 ¹⁸ | 3,63 | 124 947 938 | 20,44 | 6,21 | 16,23 |
| 2 ¹⁹ | 0,51 | 244 598 497 | 1,67 | 1,7 | 1,98 |
| 2 ¹⁹ | 3,71 | 192 833 009 | 22,22 | 5,24 | 15,66 |
| 2 ²⁰ | 0,5 | 589 022 167 | 2,81 | 3,07 | 22,91 |
| 2 ²⁰ | 3,65 | 434 235 010 | 22,48 | 4,78 | 62,49 |
| 2 ²¹ | 0,53 | 1 237 576 961 | 2,24 | 3,64 | 46,43 |
| 2 ²¹ | 3,65 | 870 883 135 | 21,98 | 5,43 | 74,07 |
| 2 ²² | 0,52 | 2 448 258 888 | 2,6 | 4,29 | 38,15 |
| 2 ²² | 3,62 | 1 768 991 087 | 22,33 | 4,88 | 91,38 |
| 2 ²³ | 0,53 | 5 036 140 404 | 2,46 | 4,02 | 37,7 |
| 2 ²³ | 3,65 | 3 531 412 473 | 22,3 | 4,73 | 94,77 |

Jak można zaobserwować, dla optymalizacji różnej od `-O3` przetwarzanie posortowanych danych jest dużo szybsze. Jest tak, ponieważ procesor stara się przewidzieć wynik operacji warunkowej `if (x >= 128)` i łąduje kolejne instrukcje do potoku przetwarzania. W przypadku, gdy dane są nieposortowane, procesor w ponad 20% przypadków zgaduje źle (co widać w tabeli 3.6 w kolumnie 4), przez co musi porzucić instrukcje z potoku przetwarzania, co powoduje narzut czasowy (zostało to szerzej opisane w sekcji 2.1.5). Jak widać na rysunkach 3.9e oraz 3.10e, traci na wydajności około 6-7 razy (w zależności od procesora) w stosunku do przetwarzania posortowanych danych.

Optymalizacja z flagą `-O2` wykonuje instrukcję porównania wartości w rejestrze z wartością 127 – `cmp $0x7f,%edx`, a następnie instrukcję skoku warunkowego – `jle`, która przeskakuje instrukcje dodawania wartości z wektora do sumy, jeżeli wartość w rejestrze była mniejsza.

Najbardziej agresywna optymalizacja `-O3` pozbyła się instrukcji warunkowej wykorzystując odpowiednie instrukcje wektorowe (opisane w sekcji 2.3). Dzięki temu wydajność dla posortowanych, jak i nieposortowanych danych jest niemalże taka sama. W listingu 3.8 zaprezentowano przebieg mierzonej funkcji w formie pseudokodu asemblera dla optymalizacji `-O3` dla nieposortowanego wektora o rozmiarze 13. Elementami wektora po inicjalizacji są liczby: 208, 34, 231, 213, 32, 248, 233, 56, 161, 78, 24, 140, 71.

Listing 3.8. Pseudokod mierzonej funkcji z przykładu opisanego w sekcji 3.3 dla optymalizacji `-O3`.

```
(1) PXOR xmm2, xmm2           // zerowanie xmm2
(3) PXOR xmm4, xmm4           // zerowanie xmm4
(2) xmm5 = 127, 127, 127, 127 // przypisanie stałych do rejestru xmm5
(4) xmm1 = 208, 34, 231, 213   // załadowano cztery pierwsze elementy wektora
(5) xmm0 = xmm1                // kopiowanie wartości xmm1 do xmm0

(6) PCMPGTD xmm5, xmm0         // porównanie "większe niż" integerów 32 bitowych ↔
    w xmm5 z xmm0, wynik zapisywany w xmm0 (255 jeśli element był większy, 0 ↔
    jeśli nie był)
Wynik w xmm0 = 255, 0, 255, 255

(7) PAND xmm1, xmm0            // operacja bitowa and – w xmm0 otrzymano liczby ↔
    >127
Wynik w xmm0 = 208, 0, 231, 213

(8) xmm1 = xmm4                // Skopiowanie xmm4, obecnie zeruje to xmm1
(9) xmm3 = xmm0                // Skopiowanie liczb większych niż 127 do xmm3

(10) PCMPGTD xmm0, xmm1        // obecnie nic to nie zmienia

(11)** PUNPCKLDQ xmm1, xmm3     // Zapisanie do xmm3 dwóch "niższych" int32 z ↔
    xmm1 oraz xmm3
xmm1 = 0, 0, 0, 0
xmm3 = 208, 0, 231, 213
Wynik w xmm3 = 208, 0, 0, 0
```

```

(12) PUNPCKHDQ xmm1, xmm0          // Zapisanie do xmm0 dwóch "wyższych" int32 z ←
    xmm1 oraz xmm0
xmm1 = 0, 0, 0, 0
xmm0 = 208, 0, 231, 213
Wynik w xmm0 = 231, 0, 213, 0

(13) xmm2 += xmm3                  // Sumowanie elementów w rejestrach.
Wynik w xmm2 = 208, 0, 0, 0
(14) xmm2 += xmm0
Wynik w xmm2 = 439, 0, 213, 0

(15) // Skok warunkowy przed instrukcję (1) - iteracja powtarzana jest, aż ←
    zsumowane zostanie 12 z 13 elementów wektora większych niż 127. Pozostałe ←
    elementy (które nie zmieściły się do jednostki wektorowej) są sumowane ←
    niewektorowo. Wynik powyższych operacji jest zapisany w rejestrze rsi - ←
    którego wartość to 1434.

(16) CMP 127, eax                  // Sprawdzenie, czy element wektora jest większy ←
    niż 127.
(17) CMOVL rdx, rax                // Warunkowe zapisanie wartości ostatniego ←
    elementu wektora do rejestru rax.
(18) ADD rax, rsi                  // Dodanie 0 do rsi (gdyby element wektora byłby ←
    większy od 127, to dodalibyśmy go, zamiast 0).

// Następnie odbywa się skok na koniec mierzonej funkcji (aby zmierzyć czas ←
    wykonywania) lub kolejne instrukcje podobne do 16-18 (jeśli pozostałych ←
    elementów było więcej).

```

Ciekawym faktem jest, że na procesorze Intel i7 4720HQ wykorzystanie jednostek wektorowych przez optymalizację `-O3` jest najwydajniejsze, a dla Intel Xeon W3565 najwydajniejsza była optymalizacja `-O2` wraz z obliczaniem warunku. Przyczyn takich wyników może być wiele. Prawdopodobnie wynika to z różnicy w implementacji instrukcji wektorowych w procesorach lub z różnych wersji kompilatora.

3.4. Przetwarzanie równoległe

Kolejnym omówionym przykładem jest przetwarzanie równoległe [21]. Zbadano w nim wpływ adresów, pod jakie wątki zapisują wyniki, na wydajność.

Testowany kod przedstawiono na listingu 3.9. Tworzy on zadaną liczbę wątków, które iterują po danej części wektora, wykonując algorytm z lambdy zapisanej w zmiennej `th` (linie 2-8). W trakcie wykonywania algorytmu wątki zapisują wyniki pod osobnymi adresami, które w zależności od przypadku są adresami leżącymi blisko lub daleko od siebie, co zostało pokazane na listingu 3.10. Teoretycznie, skoro adresy, do których piszą wątki, są różne, nie powinno to wpływać na wydajność.

Listing 3.9. Kod mierzonej funkcji dla problemu opisanego w sekcji 3.4.

```

1  auto f = [&](volatile int **result_pointers) {
2      auto th = [](volatile int *result, int *first, int *last) {
3          *result = 0;
4          while (first != last) {
5              int x = *first++;
6              *result += x % 2;
7          }
8      };
9
10     std::vector<std::thread> threads;
11     const auto elements_per_thread = n / threads_count;
12     const auto rest_elements = n % threads_count;
13     decltype(n) data_index_start = 0;
14
15     for (int thread_index = 0; thread_index < threads_count; ++thread_index) {
16         auto do_elements = elements_per_thread;
17
18         if (thread_index < rest_elements)
19             do_elements += 1;
20
21         const decltype(n) data_index_end = data_index_start + do_elements + 1;
22         threads.push_back(
23             std::thread{
24                 th,
25                 result_pointers[thread_index],
26                 v.data() + data_index_start,
27                 v.data() + data_index_end
28             }
29         );
30         data_index_start = data_index_end;
31     }
32
33     for (auto &t: threads)
34         t.join();
35
36     auto sum = 0;
37     for (auto i = 0; i < pointers_count; ++i)
38         sum += *result_pointers[i];
39     return sum;
40 };

```

Listing 3.10. Kod przedstawiający ulokowanie wskaźników, pod które wątki zapisują wyniki dla problemu opisanego w sekcji 3.4.

```

1  int res[500];
2  // pierwszy przypadek - wątki piszące do adresów ulokowanych blisko siebie

```

```
3 volatile int *near_pointers[] = {res, res + 1, res + 2, res + 3, res + 4, res + ↵  
    5, res + 6, res + 7};  
4  
5 // drugi przypadek - wątki piszące do adresów odległych od siebie  
6 volatile int *far_pointers[] = {res, res + 64, res + 128, res + 192, res + 256, ↵  
    res + 320, res + 384, res + 448};
```

Na wykresach z rysunków 3.11, 3.12, 3.13 oraz 3.14 przedstawiono testy wydajności dla dwóch różnych procesorów dla różnej liczby wątków wykorzystujących odległe (*far*) lub bliskie (*near*) wskaźniki.

Jak można zauważyć na wykresach z rysunków 3.11 oraz 3.13, dla małej liczby elementów przetwarzanego zbioru (od 2^{10} do 2^{12}) nie opłaca się uruchamiać więcej niż jednego wątku, ponieważ narzut związany z utworzeniem wątków marginalizuje przyspieszenie wynikające z zrównoleglenia.

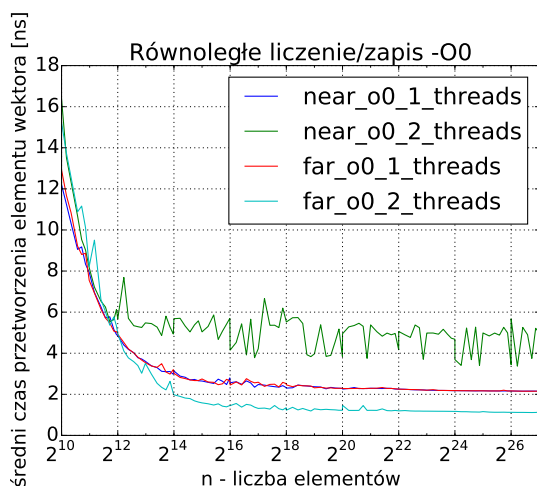
Gdy wątki wykorzystują bliskie adresy, wydajność jest gorsza. Jest tak, ponieważ gdy adresy są blisko siebie w programie, to znajdują się one również w jednej linii cache, a gdy różne rdzenie zapisują dane do jednego bloku pamięci (linii cache), to muszą go synchronizować między sobą². Problem ten nazywa się „fałszywym współdzieleniem” (ang. *false sharing*), gdyż z pozoru wątki nie współdzieliły danych, a w rzeczywistości, pod powłoką, tak się dzieje.

Pomimo tego, że testy wydajności dla obu procesorów zostały przeprowadzone z włączoną technologią hyper-threading (czyli na jednym rdzeniu procesora działają dwa wątki), wydajność programu dla dwóch wątków nadal jest lepsza dla odległych adresów. Prawdopodobnie wątki te zostały rozdzielone na dwa różne rdzenie procesora, aby mogły wykorzystywać więcej pamięci cache.

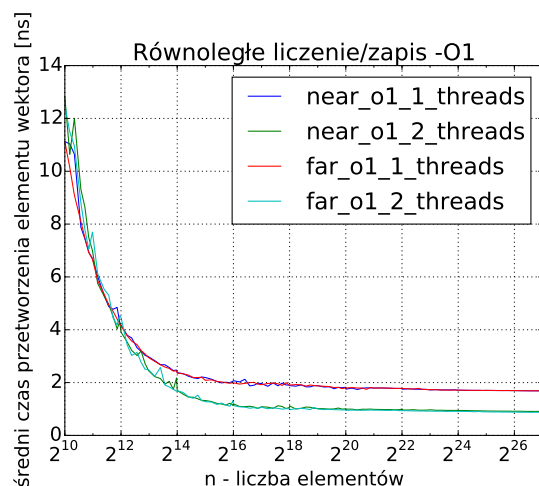
Jak można zauważyć na wykresach 3.11e oraz 3.13e dla dużej liczby danych przyspieszenie związane z zapisem do odległych adresów dla 8 wątków jest około dwukrotne, względem zapisu do bliskich adresów.

Ciekawy może być fakt, że gdyby w listingu 3.9 wyrzucić słowa kluczowe `volatile`, różnicy wydajności nie byłoby dla kompilacji z odpowiednią optymalizacją. Wynik obliczeń zostałby zapisywany do tymczasowego rejestru, a zapis pod dany adres odbywałby się dopiero pod koniec całej iteracji. Niemniej jednak nie każdy przypadek fałszywego współdzielenia zostanie automatycznie wyeliminowany przez kompilator.

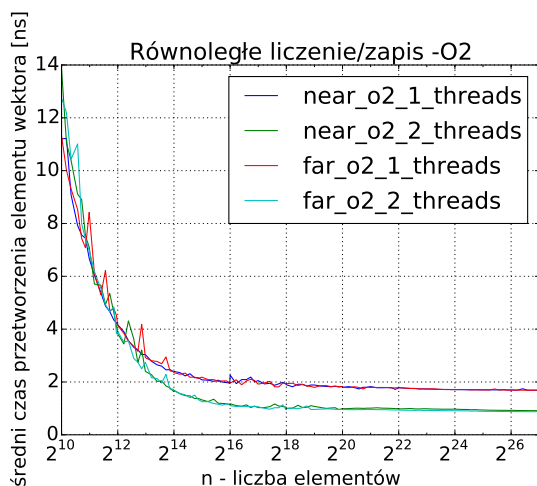
²Taka synchronizacja w przypadku procesorów firmy Intel odbywa się za pomocą protokołu MESIF.



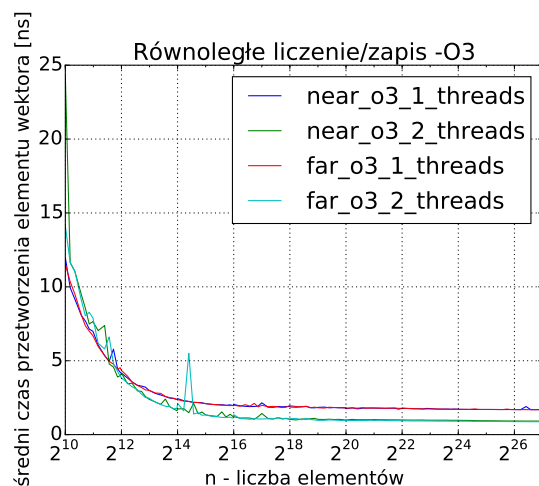
(a) Kompilacja z flagą -O0



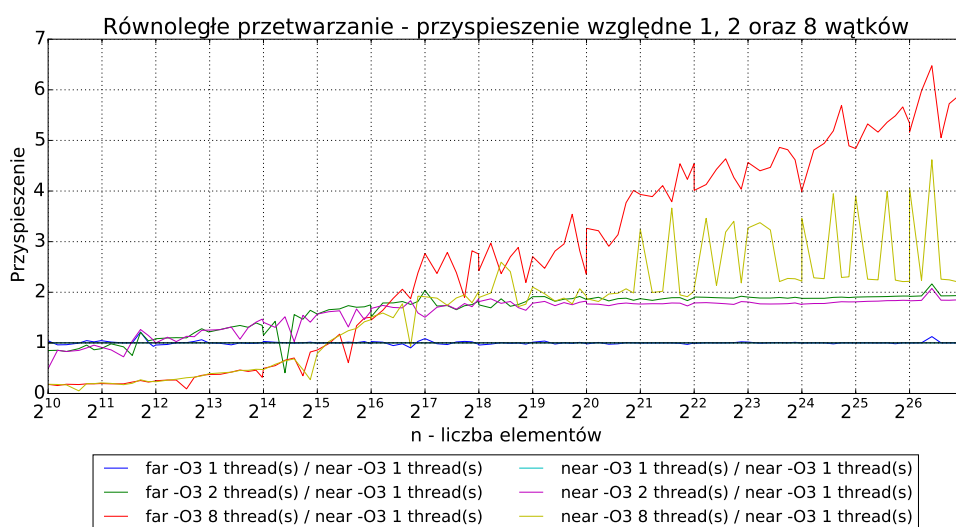
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2

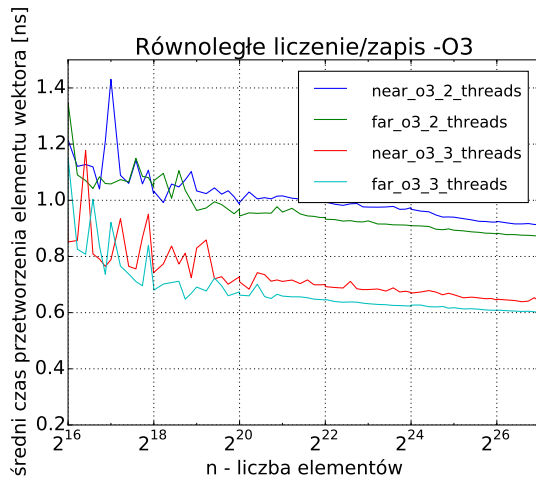


(d) Kompilacja z flagą -O3

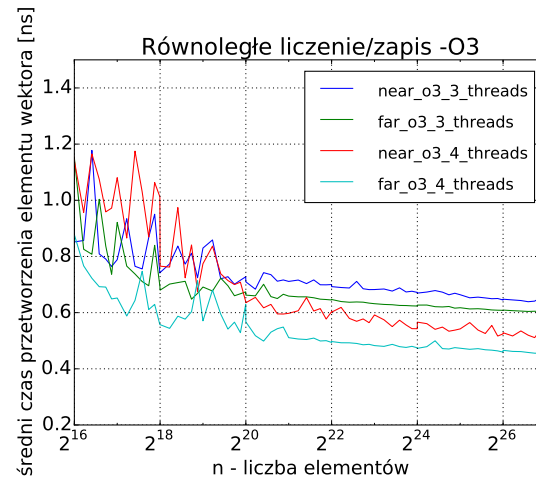


(e) Przyspieszenie uzyskane względem bliskich wskaźników dla jednego wątku, dla kompilacji z flagą -O3.

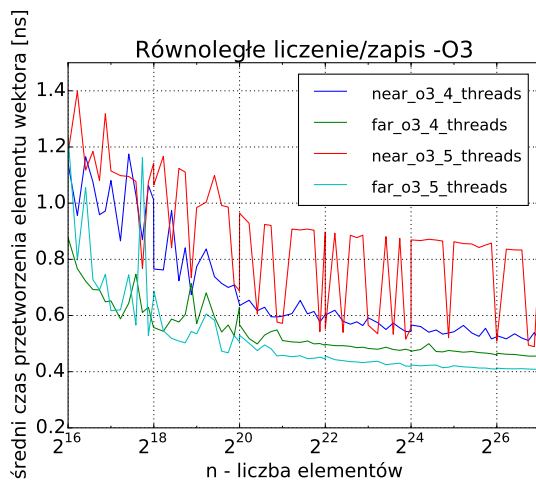
Rys. 3.11. Wyniki testów przetwarzania równoległego (z sekcji 3.4) dla różnej liczby wątków, dla procesora Intel i7-4720HQ.



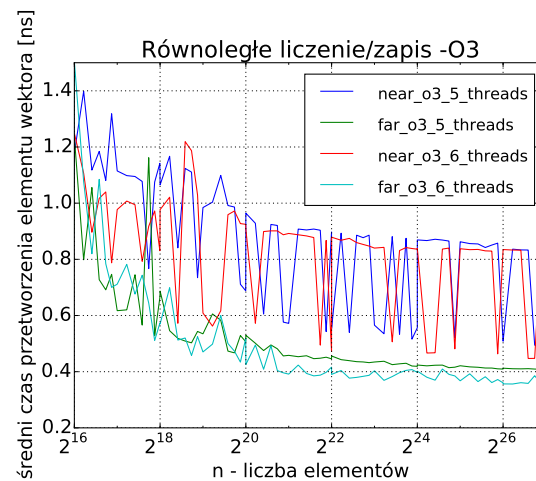
(a) Wydajność 2 i 3 wątków.



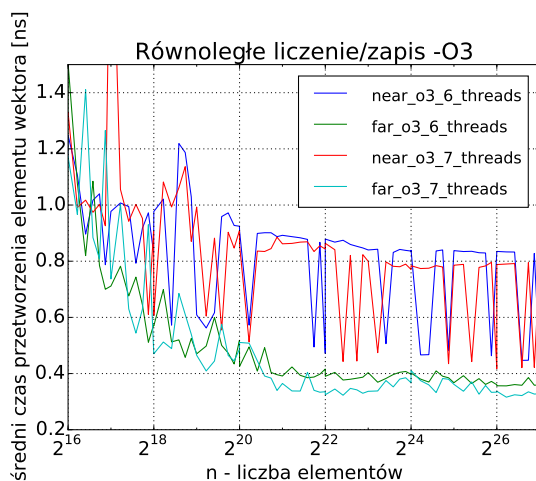
(b) Wydajność 3 i 4 wątków.



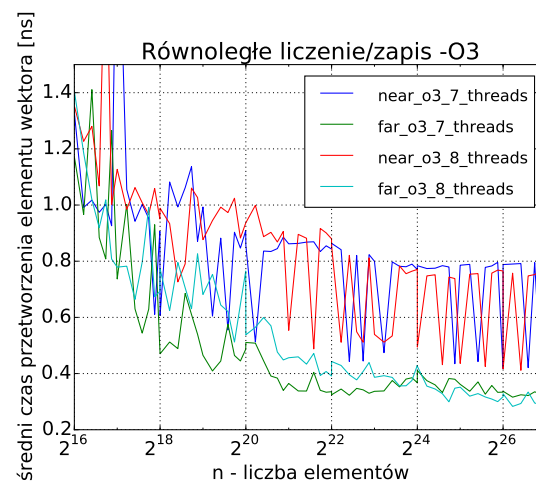
(c) Wydajność 4 i 5 wątków.



(d) Wydajność 5 i 6 wątków.

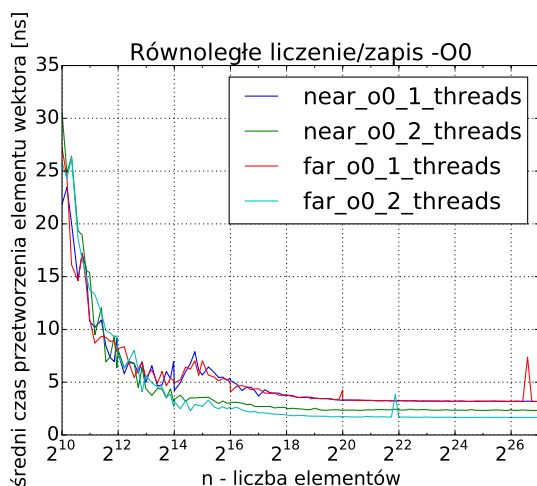


(e) Wydajność 6 i 7 wątków.

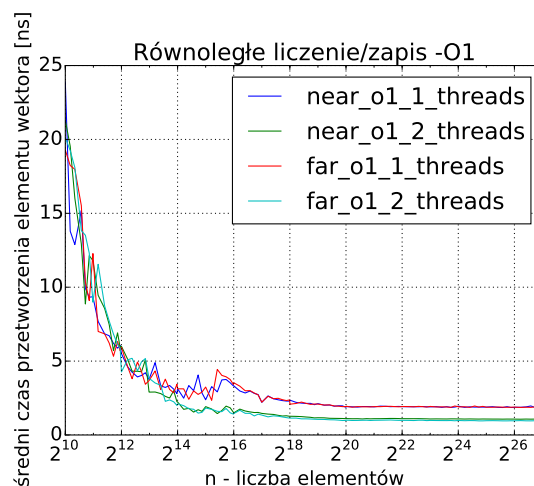


(f) Wydajność 7 i 8 wątków.

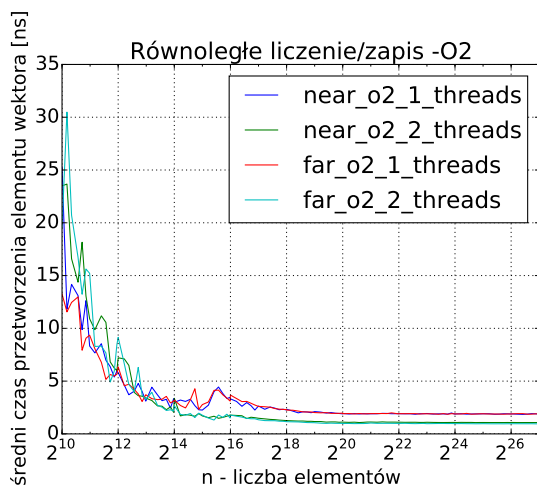
Rys. 3.12. Bardziej szczegółowe wyniki testów przetwarzania równoległego (z sekcji 3.4) dla różnej liczby wątków, dla procesora Intel i7-4720HQ.



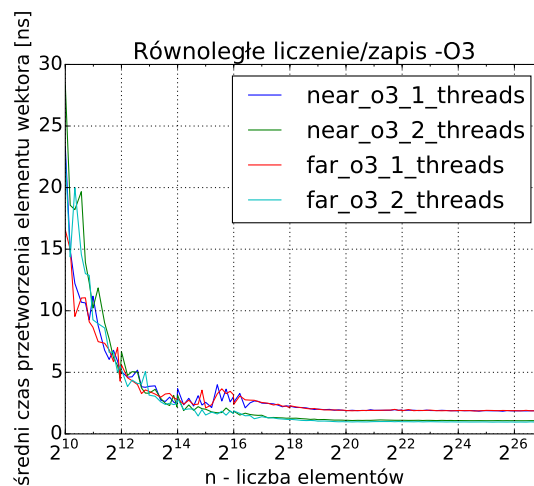
(a) Kompilacja z flagą -O0



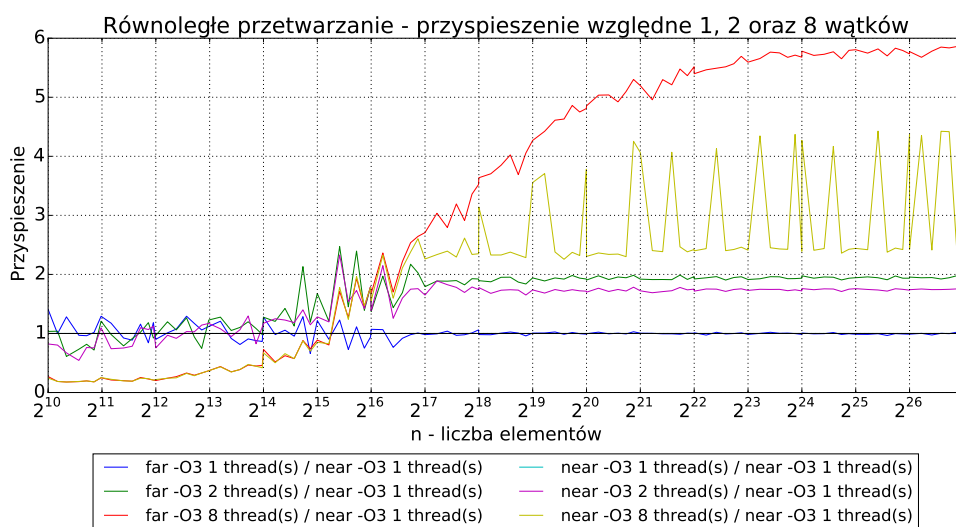
(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2

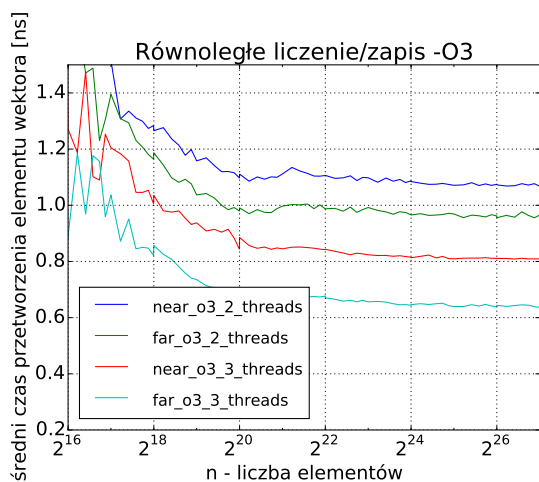


(d) Kompilacja z flagą -O3

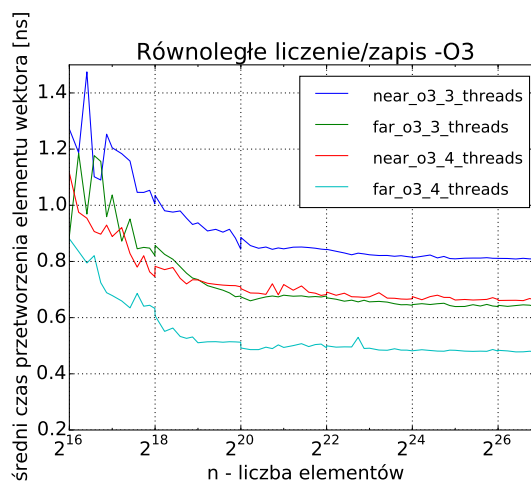


(e) Przyspieszenie uzyskane względem bliskich wskaźników dla jednego wątku, dla kompilacji z flagą -O3.

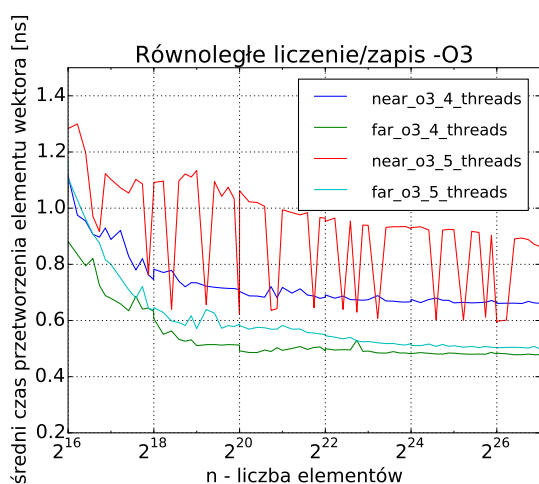
Rys. 3.13. Wyniki testów przetwarzania równoległego (z sekcji 3.4) dla różnej liczby wątków, dla procesora Intel Xeon W3565.



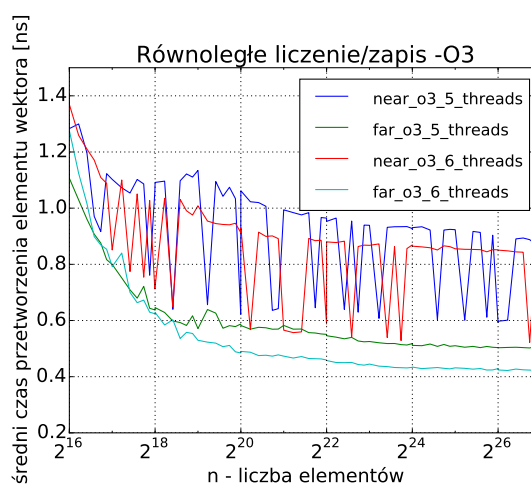
(a) Wydajność 2 i 3 wątków.



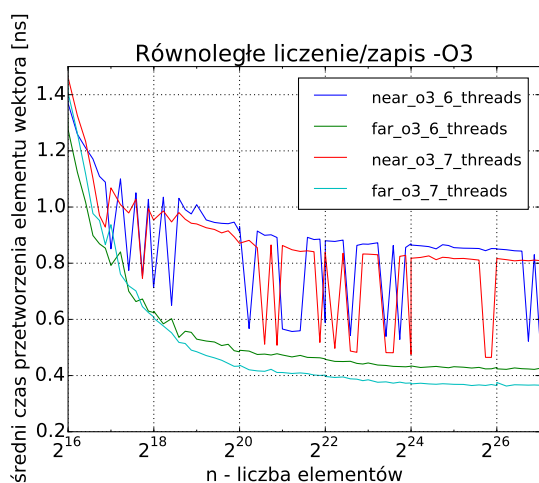
(b) Wydajność 3 i 4 wątków.



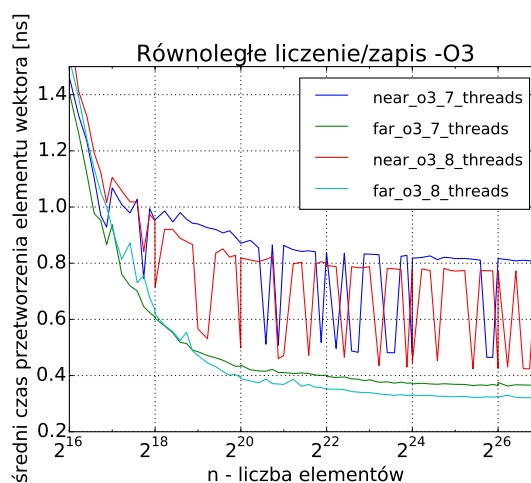
(c) Wydajność 4 i 5 wątków.



(d) Wydajność 5 i 6 wątków.



(e) Wydajność 6 i 7 wątków.



(f) Wydajność 7 i 8 wątków.

Rys. 3.14. Bardziej szczegółowe wyniki testów przetwarzania równoległego (z sekcji 3.4) dla różnej liczby wątków, dla procesora Intel Xeon W3565.

3.5. Wyrównanie danych

Kolejnym testem jest porównanie wydajności dostępu do wyrównanych oraz niewyrównanych danych, czyli problemu opisanego w sekcji 2.8.

Na listingu 3.11 przedstawiono dwie struktury o takim samym rozmiarze (72 bajty) – domyślnie wyrównaną – `PaddedList` oraz wyrównaną do jednego bajtu – `PackedList`. Struktury te są węzłami listy jednokierunkowej. W drugim przypadku, wskaźnik na kolejny element listy jest ułożony w pamięci tak, że znajduje się pomiędzy dwoma liniami cache – w tym celu zastosowano omówioną w sekcji 2.8 specjalną dyrektywę preprocesora – `#pragma pack`.

Na listingu 3.12 znajduje się kod mierzonej funkcji szablonowej. Polega ona na przejściu po elementach danego typu listy, który jest argumentem szablonu.

Listing 3.11. Kod wyrównanej i niewyrównanej struktura danych dla problemu wyrównania danych opisanego w sekcji 3.5. Obie struktury zajmują 72 bajty.

```
1 struct PaddedList {                // struktura danych wyrównana domyślnie
2     char padding[59];
3     struct PaddedList* next;
4 };
5
6 #pragma pack(1)                    // ustawienie wyrównania do jednego bajtu
7 struct PackedList {                // struktura danych wyrównana do 1 bajtu
8     char padding[59];
9     struct PackedList* next;
10    char fill[5];
11 };
12 #pragma pack()                     // ustawienie wyrównania na domyślne
```

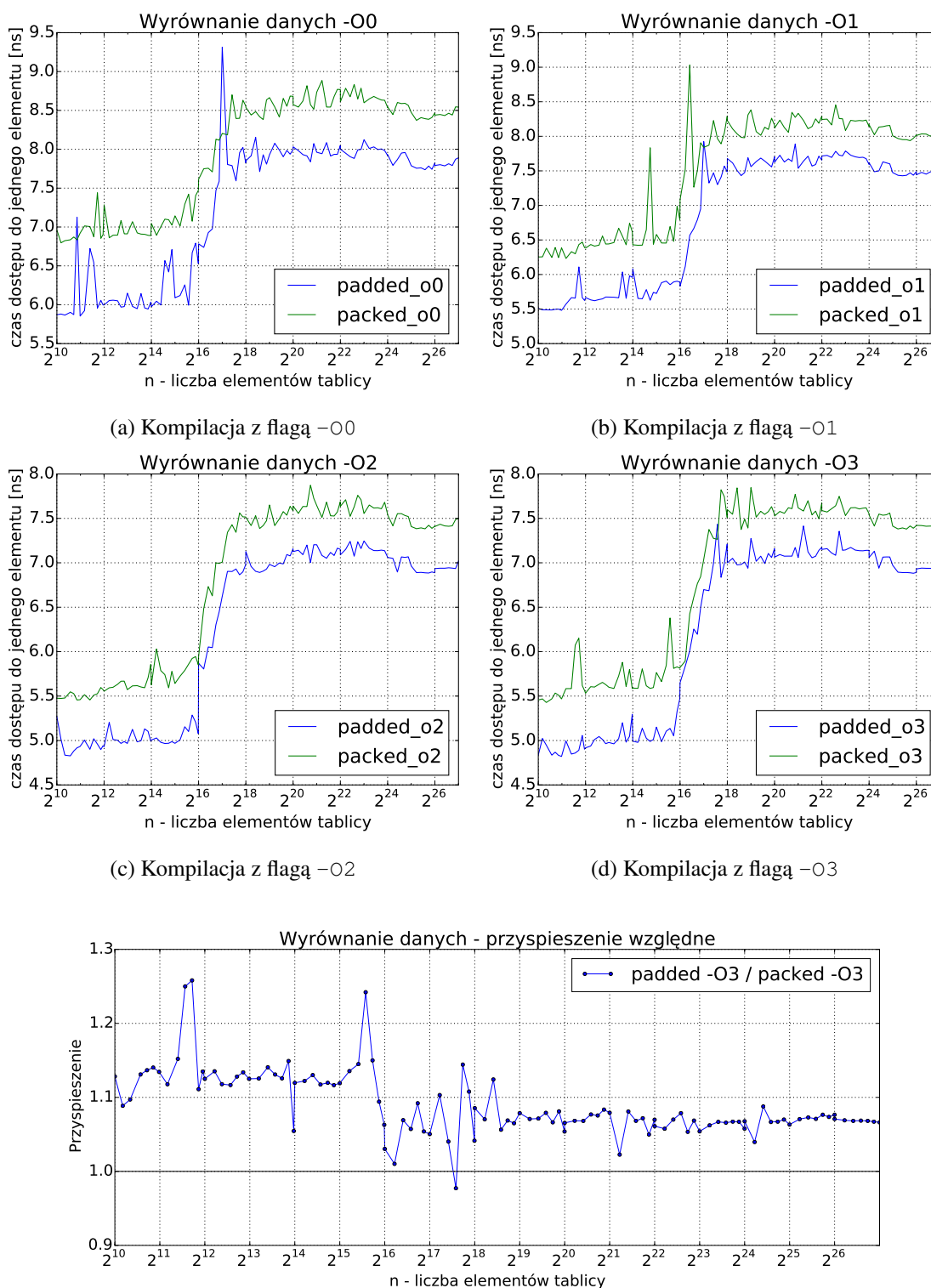
Listing 3.12. Kod mierzonej funkcji dla problemu wyrównania danych opisanego w sekcji 3.5.

```
1 template <typename ListType>
2 long double benchmark(std::size_t n) {
3     ListType* list = new ListType[n];
4
5     for(int i=0; i<n-1; ++i)        // inicjalizacja listy
6         list[i].next = &list[i+1];
7     list[n-1].next = list;
8
9     ListType* ptr = list;
10
11    // pomiar czasu (kod mierzonej funkcji wewnątrz lambda)
12    auto timing = measure(n, [&]() {
13        for (auto i = 0; i < n; ++i)
14            ptr = ptr->next;
15        return ptr-list;
16    });
17 }
```

```
16     });  
17     // ...  
18 }
```

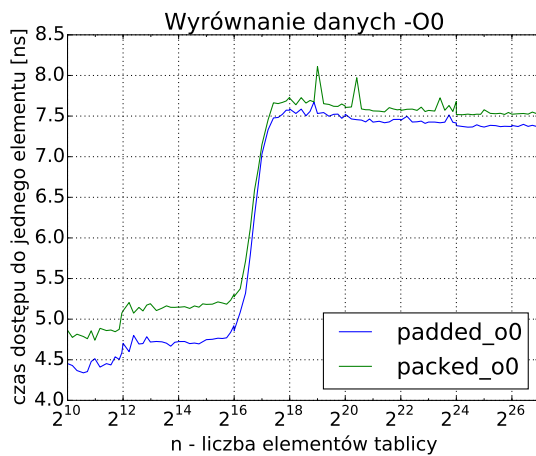
Na rysunku 3.15 przedstawiono wyniki testu wydajności. Zgodnie z oczekiwaniami dostęp do danych wyrównanych jest szybszy od dostępu do niewyrównanych, w szczególności, gdy dane mieszczą się w pamięci podręcznej.

Jak można zauważyć na wykresach 3.15e oraz 3.15e, uzyskane przyspieszenie jest niewielkie (przynajmniej w stosunku do innych omawianych testów) – rzędu 10-15%. Dla procesora Intel Xeon W3565 widać, że po przekroczeniu rozmiaru pamięci podręcznej L3 (8 MB), czyli liczby elementów między 2^{16} oraz 2^{17} , przyspieszenie jest niezauważalne. Dla procesora Intel i7 4720HQ przyspieszenie wciąż wynosi wtedy około 10%.

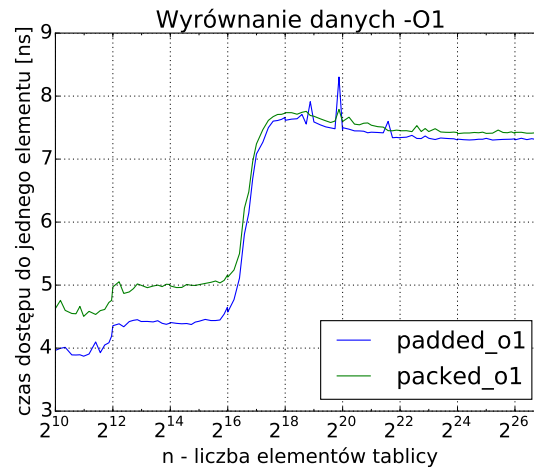


(e) Przyspieszenie uzyskane względem przypadku upakowanych danych dla kompilacji z flagą -O3.

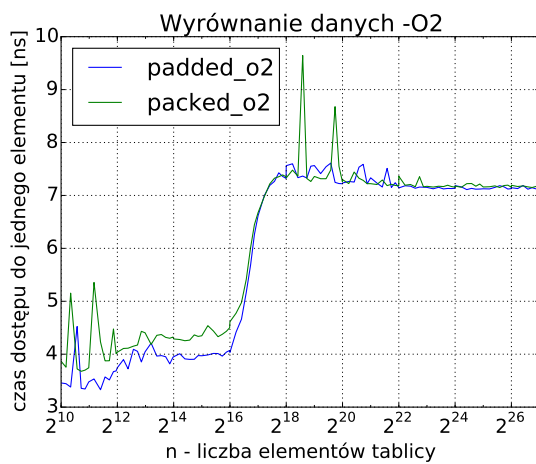
Rys. 3.15. Wyniki testów wyrównania danych (z sekcji 3.5) dla procesora Intel i7-4720HQ.



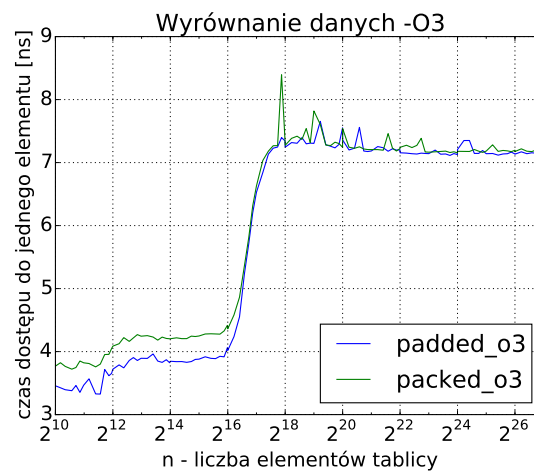
(a) Kompilacja z flagą -O0



(b) Kompilacja z flagą -O1



(c) Kompilacja z flagą -O2



(d) Kompilacja z flagą -O3



(e) Przyspieszenie uzyskane względem przypadku upakowanych danych dla kompilacji z flagą -O3.

Rys. 3.16. Wyniki testów wyrównania danych (z sekcji 3.5) dla procesora Intel Xeon W3565.

4. Podsumowanie oraz wnioski

Celem niniejszej pracy było zbadanie wpływu organizacji w pamięci złożonych struktur danych na wydajność kodu wynikowego. Problem ten jest nietrywialny i powiązany zarówno ze sprzętem, systemem operacyjnym, jak i wersją oprogramowania czy bibliotek. Widać to w niektórych testach, gdzie wyniki różnią się pomiędzy maszynami i oprogramowaniem, na którym były uruchamiane.

Jak można było się wielokrotnie przekonać podczas omawiania przeprowadzonych testów, najbardziej efektywną optymalizacją dla algorytmów przetwarzających dane często jest wykonywanie operacji sekwencyjnie, niezależnie od faktu, czy algorytm jest zrównoleglany, czy nie.

Naturalnie, wiele rzeczywistych problemów jest bardziej złożonych, niż te przedstawione w pracy. Mimo tego, na podstawie przeprowadzonych testów można wysnuć kilka wniosków:

- Podczas projektowania oprogramowania warto pomyśleć o tym, aby dobrze wykorzystywać pamięć podręczną procesora. Nie marnować jej zatem na dane, które nie są potrzebne w danym algorytmie, jak zostało pokazane w zagadnieniu tablicy struktur oraz struktury tablic.
- W przypadku stosowania wielowątkowości należy być świadomym efektu false sharing. Problem ten nie jest oczywisty i trudno go wykryć. Można go za to prosto wyeliminować, tworząc lokalne kopie zmiennych dla wątków.
- Jeżeli nie ma dobrego uzasadnienia dla zmiany wyrównania elementów w pamięci, to nie należy tego robić. Może to prowadzić do potrzeby pobierania większej liczby linii cache przez procesor, niż wynikałoby to z rozmiaru pobieranych danych. Dodatkowo, niektóre procesory nie zezwalają na taki dostęp.

Niektóre z testów wydajności – przetwarzanie warunkowe, przetwarzanie równoległe, czy wyrównanie danych – to tylko dotknięcie pewnych problemów i ich zasygnalizowanie. Aby uzyskać bardziej całościowy obraz, należy przeprowadzić więcej różnych testów. Możliwym kierunkiem rozwoju jest dogłębnějšíe zbadanie tych przypadków.

Przykładowo, można by zbadać wydajność przetwarzania warunkowego, polegającego na posiadaniu flagi w obiekcie klasy. Innym możliwym rozwiązaniem tego problemu jest przechowywanie licznika informującego o obiektach, w których flaga jest włączona oraz sortowanie obiektów na podstawie tego licznika. W przypadku wyrównania danych, można by przeanalizować wpływ wydajności tego

efektu na przetwarzanie równoległe. Prawdopodobnie taki zrównoleglony algorytm (operujący oczywiście na osobnych danych), byłby wolniejszy, ze względu na ograniczoną liczbę jednostek wczesnego pobierania oraz faktu, że niewyrównane dane mogą wymagać wykorzystania większej liczby linii cache. Ten sam efekt można by przeanalizować dla przypadku tablicy struktur oraz struktury tablic.

Kolejnymi dwoma zagadnieniami, które można by poruszyć, jest wpływ różnych alokatorów na efekt false sharingu dla różnych rozmiarów obiektów oraz kwestię wykorzystania dużej ilości pamięci przez program. Mogą wtedy występować problemy z miejscem w pamięci podręcznej TLB, które można zredukować, wykorzystując duże strony pamięci (ang. *hugepages*).

Podczas optymalizacji dowolnego algorytmu bądź oprogramowania nie należy oczywiście zapominać o najważniejszej kwestii – „*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*” (ang. powinniśmy zapomnieć o małych przyspieszeniach, powiedzmy w 97% przypadków: przedwczesna optymalizacja jest źródłem wszelkiego zła) [22]. Aby zoptymalizować aplikację, w pierwszej kolejności należy ją sprofilować i starać wyeliminować się tak zwane wąskie gardła, czyli miejsca w kodzie, które działają wolno. Nie należy ślepo optymalizować miejsc, które wydają się działać wolno. Z drugiej strony, warto pomyśleć o kwestii wydajności programu już na etapie jego projektowania.

Bibliografia

- [1] Jason Robert Carey Patterson. *Modern Microprocessors - A 90 Minute Guide!* URL: <http://www.lighterra.com/papers/modernmicroprocessors/> (term. wiz. 2015-12-16).
- [2] Ulrich Drepper. „What Every Programmer Should Know About Memory”. W: (21 list. 2007), s. 13.
- [3] J. L. Hennessy i D.A. Patterson. *Computer Architecture. A Quantitive Approach*. 5 wyd. Morgan Kaufmann, 2012, s. 73.
- [4] Peter J. Denning. „The Locality Principle”. W: *Commun. ACM* 48.7 (lip. 2005), s. 19–24.
- [5] *Context switching*. URL: https://en.wikipedia.org/wiki/Context_switch (term. wiz. 2015-12-20).
- [6] *Disclosure of H/W prefetcher control on some Intel processors*. 24 wrz. 2014.
- [7] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Rozd. 7.
- [8] *TIOBE Index for December 2015*. URL: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html> (term. wiz. 2015-12-07).
- [9] *The Python Language Reference - Data model*. URL: <https://docs.python.org/3/reference/datamodel.html#slots> (term. wiz. 2015-12-07).
- [10] *Java diagnostics guide - Understanding Memory Management*. URL: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html%5C#wp1086917 (term. wiz. 2015-12-06).
- [11] Microsoft. *.NET Framework Development Guide - Fundamentals of Garbage Collection*. URL: https://msdn.microsoft.com/en-us/library/ee787088%5C#what_happens_during_a_garbage_collection (term. wiz. 2015-12-06).
- [12] *Java Enhancements Proposals 169: Value Objects*. URL: <http://openjdk.java.net/jeps/169> (term. wiz. 2015-12-20).
- [13] *RISC Processor Data Alignment: About Data Alignment*. URL: [https://msdn.microsoft.com/en-us/library/ms253949\(VS.80\).aspx](https://msdn.microsoft.com/en-us/library/ms253949(VS.80).aspx) (term. wiz. 2015-12-25).
- [14] *Coding for Performance: Data alignment and structures*. URL: <https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures> (term. wiz. 2015-12-25).

- [15] Joaquín M^a López Muñoz. *using std::cpp 2015: Mind The Cache*. URL: <https://github.com/joaquintides/usingstdcpp2015> (term. wiz. 2015-12-16).
- [16] *Mind The Cache - Matrix Sum*. URL: https://github.com/joaquintides/usingstdcpp2015/blob/master/matrix_sum.cpp (term. wiz. 2015-12-14).
- [17] *Mind The Cache - aos vs soa*. URL: https://github.com/joaquintides/usingstdcpp2015/blob/master/aos_vs_soa.cpp (term. wiz. 2015-12-14).
- [18] *Mind The Cache - compact aos vs soa*. URL: https://github.com/joaquintides/usingstdcpp2015/blob/master/compact_aos_vs_soa.cpp (term. wiz. 2015-12-14).
- [19] *Mind The Cache - Random access aos vs soa*. URL: https://github.com/joaquintides/usingstdcpp2015/blob/master/random_access_aos_vs_soa.cpp (term. wiz. 2015-12-14).
- [20] *Mind The Cache - Filtered Sum*. URL: https://github.com/joaquintides/usingstdcpp2015/blob/master/filtered_sum.cpp (term. wiz. 2015-12-14).
- [21] *Mind The Cache - Parallel Count*. URL: https://github.com/joaquintides/usingstdcpp2015/blob/master/parallel_count.cpp (term. wiz. 2015-12-14).
- [22] Donald E. Knuth. „Structured Programming with Go to Statements”. W: *ACM Comput. Surv.* 6.4 (grud. 1974), s. 261–301.