Zeszyty Naukowe WWSI, No 28, Vol. 17, 2023, s. 7-26
DOI: 10.26348/znwwsi.28.7

Dekompozycja instrukcji rozgałęziających w procesie optymalizacji kodu

Rafał Aleksander[*]
Adam Piórkowski[**]

Akademia Górniczo-Hutnicza,
Katedra Biocybernetyki i Inżynierii Biomedycznej, Kraków, Polska

Streszczenie

Tematem niniejszego artykułu była ocena możliwości ręcznej modyfikacji kodu źródłowego w celu optymalizacji czasu wykonania programu. W ramach projektu inżynierskiego zaproponowano autorskie techniki optymalizacyjne, wykorzystujące dekompozycję instrukcji rozgałęziających. Eksperymentalnie zbadano efektywność prezentowanej dekompozycji dla procesora z rodziny Intel x86, wykonującego operacje w sposób potokowy. W ramach niniejszej pracy przebadano kompilatory z rodziny *Clang* oraz *GCC* dla języka *C* i *C++* oraz środowisko *JVM*.

Słowa kluczowe – optymalizacja kodu, dekompozycja kodu, instrukcje rozgałęziające, analiza wydajności optymalizacji

[**] E-mail: rafal.m.aleksander@gmail.com
[**] E-mail: pioro@agh.edu.pl

Zgłoszono do druku 4 kwietnia 2023 r.

1. Wstęp

Instrukcje rozgałęziające, inaczej również nazywane instrukcjami warunkowymi lub *instrukcjami sterującymi przepływem* [1], to grupa instrukcji bezpośrednio decydujących o tym, który zestaw kolejnych, sekwencyjnie następujących instrukcji kodu maszynowego zostaje wykonany. Instrukcje te dzielą się na dwie grupy [2]:

- instrukcje rozgałęziające bezwarunkowo,
- instrukcje rozgałęziających warunkowo.

Pierwsza z grup zajmuje się bezpośrednim oddelegowaniem dalszego przebiegu programu do innego miejsca (*adresu*) [3]. Grupa instrukcji warunkowych niejako pozwala zdecydować na podstawie *wyznaczonego* warunku, który zestaw instrukcji kodu maszynowego zostanie wykonany w następnej kolejności.

1.1. Przykładowe instrukcje rozgałęziające

Przykładem instrukcji rozgałęziającej bezwarunkowo dla rodziny procesorów Intel x86 może być instrukcja ***JMP*** – Jump, natomiast do grupy najbardziej rozpoznawalnych instrukcji rozgałęziających warunkowo zaliczają się [2]:

- ***JE*** – skok przy ustawionej fladze równości (ang. ***J**ump if **E**qual)*,
- ***JG*** – skok przy ustawionej fladze większości (ang. ***J**ump if **G**reater*),
- ***JNE*** – skok przy ustawionej fladze braku równości (ang. ***J**ump if **N**ot **E**qual*).

Nazwy poszczególnych instrukcji różnią się między wybranymi rodzinami architektur procesorów, a także w zależności od konkretnego modelu i generacji procesora może różnić się ich liczba. Należy również spodziewać się, iż różne platformy mogą wykonywać przedstawione operacje logiczne w inny sposób. Przenoszenie optymalizacji wykorzystujących charakterystyczne instrukcje maszynowe między różnymi architekturami procesora może mieć skutki przeciwne do oczekiwanych.

Przeprowadzone badania analityczne [4] najczęściej wykonywanych instrukcji języka maszynowego *Intel x86* zdecydowanie wskazują na to, że sumarycznie najczęściej wykonywanymi instrukcjami maszynowymi są operacje przenoszące, np. instrukcja *MOV*. W tym miejscu należy wspomnieć, iż w wyniku przytoczonych badań oraz w związku ze złożonymi problemami optymalizacyjnymi dla przetwarzania potokowego we współczesnych procesorach powstała specjalna grupa instrukcji rozgałęziających warunkowo, zajmujących się operacjami przenoszącymi. Pierwszy raz

instrukcje z tej rodziny pojawiły się w rodzinie *x86* w roku 1995, wraz z wydaniem przez firmę Intel na rynek procesorów *Intel Pentium Pro*. Grupa instrukcji, o której mowa, to instrukcje przeniesienia warunkowego ***CMOVcc***.

1.2. Rodzina instrukcji *CMOVcc*

Instrukcje z rodziny ***CMOVcc*** sprawdzają stan jednej lub wielu flag z rejestrów procesora, a następnie przeprowadzają operację przeniesienia w zależności od spełnienia zadanego warunku [5]. Należy również zaznaczyć, że instrukcje nie porównują bezpośrednio danych, tylko wykorzystują flagi z wcześniejszych porównań. Różnicę między standardową operacją warunkowego przeniesienie, a operacją **CMOVcc** przedstawiono na listingach:

```
1 :  cmp eax, ebx
2 :  jne skip
3 :  mov ecx, edx
```

Listing 1. Sekwencja przeniesienia warunkowego bez instrukcji ***CMOVcc***

```
1 :  cmp eax, ebx
2 :  cmove ecx, edx
```

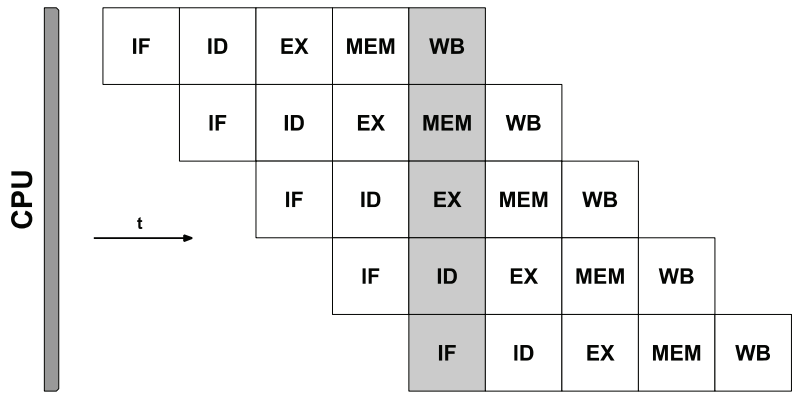
Listing 2. Sekwencja przeniesienia z użyciem instrukcji ***CMOVcc***

Obecnie do grupy instrukcji ***CMOVcc*** zalicza się ponad 80 rożnych operacji [5], m.in.:

- **CMOVZ** – przeniesienie przy fladze ustawionej na zero (ang. ***M****ove if* ***Z****ero*),
- **CMOVE** – przeniesienie. przy ustawionej fladze równości (ang. ***M****ove if* ***E****qual*),
- **CMOVP** – przeniesienie przy wyzerowanej fladze *parity* (ang. ***M****ove if* ***P****arity*),
- **CMOVNO** – przeniesienie przy wyzerowanej fladze przepełnienia (ang. ***M****ove if* ***N****ot* ***O****verflow*).

1.3. Instrukcje sterujące przepływem a przetwarzanie potokowe

Dostępne obecnie procesory przetwarzają dane w sposób sekwencyjny – procesory wykonują operacje w zespołach specjalnie przygotowanych grup instrukcji [6]. Grupy te wykonywane są w odpowiedniej narzuconej kolejności, tworząc w ten sposób potok. Kolejność jest niezwykle ważna, gdyż następujące kolejno grupy operacji niejako pozwalają procesorowi na równoległą kontynuację pracy na już zmodyfikowanych danych. Każda z kolejnych grup operacji jest na innym, odpowiednio przesuniętym etapie wykonania. Sposób potokowego przetwarzania operacji przedstawiony jest na diagramie (rysunek 1).



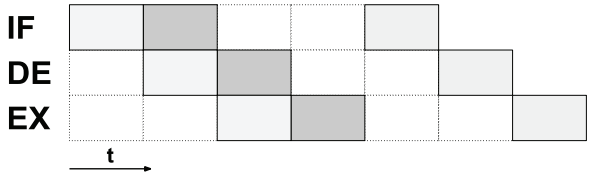
Rysunek 1. Schemat potokowego wykonania operacji przez procesor. W schemacie przyjęto następujące oznaczenia: **IF** – ang. ***I**nstruction **F**etch* (pobranie instrukcji z pamięci); **ID** – ang. ***I**nstruction **D**ecode* (zdekodowanie instrukcji); **EX** – ang. ***EX**ecute* (wykonanie instrukcji); **MEM** – ang. ***ME**-**M**ory access* (dostęp do pamięci); **WB** – ang. ***W**rite **B**ack* (zapisanie wyników działania).

Największym problemem potokowego wykonania operacji są instrukcje rozgałęziające. Trudność określenia właściwego wyboru następnej instrukcji w momencie, gdy warunek jeszcze nie jest wyznaczony, tworzy zauważalne opóźnienia. Na chwilę obecną szacuje się, że rodzina architektur *x86* jest na to bardzo narażona, gdyż stwarzające problem operacje skoku warunkowego pojawiają się w wygenerowanym kodzie maszynowym co kilkanaście instrukcji.

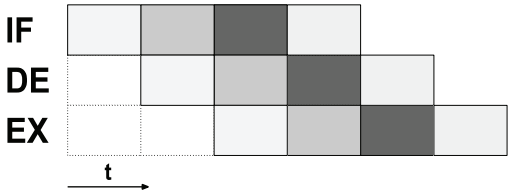
1.3.1. Predykcja statyczna

W podstawowej realizacji przetwarzania potoku logiczne rozgałęzienie przepływu programu rozwiązywane jest przez ścisłe, sekwencyjne zachowanie kolejności wykonywanych operacji w obrębie danej kolejki. Zawsze jako pierwszy wyliczany jest warunek rozgałęzienia, a następnie na podstawie jego wartości wybierana jest określona ścieżka wykonania programu. Rzeczywisty przebieg wykonania wygląda analogicznie do przypadku wskazanego na rysunku 2 – całkowity brak informacji o tym co powinno wykonać się jako kolejna operacja skutkuje dłuższym czasem wykonania.

W czasie wykonania programu współczesne procesory z rodziny *Intel x86* wykorzystują mechanizm *predykcji statycznej* [7]. W trakcie przetwarzania kolejnych instrukcji procesor w miejscu logicznego rozgałęzienia programu wybiera w sposób naiwny jeden z możliwych wariantów przepływu kodu, dla którego następnie przygotowuje odpowiedni potok, przebieg tej operacji zilustrowano na diagramie (rysunek 3). Właściwa kolejność następujących operacji nie jest jednak znana do momentu wyznaczenia wartości warunku. W sytuacji, kiedy warunek nie zostanie spełniony, czyli załadowany potok jest niewłaściwy, zostaje on w całości *wycofywany* (rysunek 3).

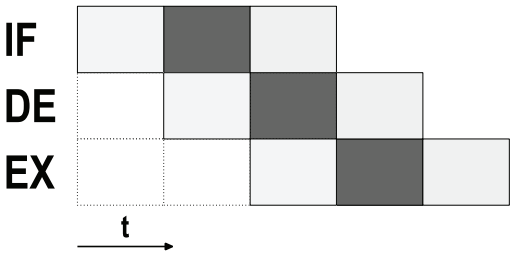


Rysunek 2. Uproszczony schemat potokowego sposobu wykonania kodu z listingu 1 – przypadek, gdy warunek skoku został spełniony



Rysunek 3. Uproszczony schemat potokowego sposobu wykonania kodu z listingu 1 – przypadek, gdy warunek skoku nie został spełniony

W naiwnych przypadkach tj. np. ciągłego niespełnienia warunku, predykcja statyczna powinna gwarantować najlepsze czasy wykonania [7]. Bardzo zbliżona sytuacja miała miejsce dla jednego z eksperymentalnych przypadków testowych, którego wyniki zostały zaprezentowane w dalszej części artykułu. W obliczu wad prezentowanej predykcji *statycznej* zostały zaproponowane instrukcje stosujące predykcję cząstkową, które to w naiwny sposób naśladują predykcję dynamiczną, występującą w procesorach bezinstrukcyjnych, sterowanych przepływem danych. Dla procesorów z rodziny *Intel x86* instrukcjami tego typu są zbiory instrukcji ***CMOVcc*** oraz ***FCMOVcc***, które w czasie wykonania programu wykorzystują określone wartości rejestrów do wykonania przypisania w przypadku spełnienia wybranego warunku [6]. Brak konieczności modyfikacji potoku wykonania w czasie pracy pozwala na uniknięcie wielu związanych z tym opóźnień.



Rysunek 4. Uproszczony schemat potokowego sposobu wykonania kodu z listingu 2 – przypadek wykorzystujący instrukcję przeniesienia warunkowego ***CMOVE***

Różnice związane z wygenerowanymi opóźnieniami zilustrowano na rysunkach 2 oraz 3. Prezentują one sposób wykonania kodu bez użycia instrukcji przeniesienia warunkowego – kod z listingu 1. Rysunek 4 obrazuje sposób wykonania kodu, w którym znalazła się instrukcja przeniesienia warunkowego ***CMOVE*** – kod listingu 2. Diagramy prezentują uproszczony trójstopniowy potokowy sposób wykonania kodu względem kolejnych instrukcji kodu maszynowego. Kolor najciemniejszy opisuje instrukcję przenoszącą.

2. Dekompozycja instrukcji rozgałęziających

Większość współczesnych popularnych środowisk programistycznych posiada zestaw odpowiednich przełączników decydujących o tym, czy ma zachodzić, i jeśli tak, to jak silna optymalizacja.

2.1. Ograniczenia optymalizacyjne kompilatorów

Wspomniane mechanizmy optymalizacyjne w większości przypadków bardzo dobrze sobie radzą z doborem odpowiednich instrukcji rozgałęziających dla warunków logicznych kodu wysokiego poziomu. Biorąc za przykład kod z listingu 3, który został skompilowany z flagą *-O2* można zauważyć, że kompilator *GCC* w wersji *8.2.1* wybrał specjalnie stworzoną do takich zadań instrukcję ***CMOVE***.

```
1 :  int i; int res = 0;
2 :
3 :  for (i = 0; i < range; ++i)
4 :     if (VALUE == A[i])
5 :         res = i;
```

Listing 3. Funkcja znajdująca ostatni indeks elementu w tablicy o zdanej wartości

```
1 :  nopl  rax
2 :  mov   rcx,rdx
3 :  cmpb  rdi,rdx,1
4 :  ea    rdx,rcx
5 :  cmove edx,eax
6 :  cmp   rdx,rsi
```

Listing 4. Fragment kodu maszynowego pętli *for* z Listingu 3

Problem stanowią zadania bardziej złożone. Jeśli nie są rozważane operacje wykonywane tylko w oparciu o zmianę wartości na jednym rejestrze, który dodatkowo używany jest w czasie iteracji pętli, kompilator nie ma do dyspozycji odpowiedniego

zestawu flag. W takiej sytuacji w praktyce nie jest możliwe wytypowanie odpowiednich dla danej akcji instrukcji predykcji odgałęzienia.

2.2. Optymalizacja instrukcji sterujących przepływem w językach wysokiego poziomu

Do prezentacji proponowanego zabiegu optymalizacyjnego został wykorzystany problem sortowania tablicy przy użyciu algorytmu sortowania bąbelkowego. Kod funkcji sortującej w wersji podstawowej (próba kontrolna) został zamieszczony na listingu 5.

```
1 :   int i; int j; int tmp;
2 :
3 :   for (i = 0; i < N; ++i)
4 :    for (j = 0; j < N - i; ++j)
5 :     if(A[j] > A[j+1]) {
6 :       tmp = A[j];
7 :       A[j] = A[j+1];
8 :       A[j+1] = tmp;
9:    }
```

Listing 5. Podstawowa funkcja sortująca przed optymalizacją w wariancie dla języka *C* oraz *C++*

W ramach analizy zachowania badanych kompilatorów w obliczu prezentowanej modyfikacji kodu źródłowego przedstawiono również przykładowy zdekompilowany kod wewnętrznej pętli *for* funkcji sortującej dla przypadku standardowego – wariant A oraz optymalizowanego – wariant B.

W wersji zdekompilowanej usunięto dodatkowe wygenerowane etykiety niezwiązane z analizowanym problemem oraz uproszczony został zapis wyliczanych przesunięć; tak przygotowane kody języka maszynowego zostały odpowiednio zamieszczone na listingach 6 oraz 8.

W ramach niniejszej pracy sprawdzane były różne kompilatory w różnych wersjach oraz różne opcje kompilacji. W standardowej wersji algorytmu żaden z wykorzystanych kompilatorów na chwilę obecną nie jest wstanie rozwinąć kodu wysokopoziomowego do szukanych instrukcji kodu maszynowego. Niezależnie od

zastosowanego mechanizmu sterującego przepływem, tj. użycia operatora *trójargumentowego*, instrukcji *switch*, czy instrukcji *if* w dowolnej kombinacji, nie skutkowało wygenerowaniem odpowiednich instrukcji kodu maszynowego. W obliczu przedstawionego problemu została zaproponowana metoda ręcznej optymalizacji, przedstawiona na listingu 7.

```
1 : .LBB0_2:
2 :     mov ebx, dword ptr [rdi]
3 :     cmp r8d, 1
4 :     jne .LBB0_5
5 : .LBB0_6:
6 :     mov ebx, dword ptr [...]
7 :     mov edx, dword ptr [...]
8 :     cmp ebx, eax
9 :     mov ebp, eax
10:     cmovle ebp, ebx
11:     mov dword ptr [...], ebp
12:     cmovl ebx, eax
13:     cmp edx, ebx
14:     mov eax, ebx
15:     cmovle eax, edx
16:     cmovge ebx, edx
17:     mov dword ptr [...], eax
```

Listing 6. Kod maszynowy dla wewnętrznej pętli *for* w kodzie z listingu 5

```
1 :  int i; int j; int tmpA; int tmpB;
2 :
3 :  for (i = 0; i < range; ++i)
4 :    for (j = 0; j < range - i; ++j) {
5 :      tmpA = A[j]; tmpB = A[j+1];
6 :      A[j] = tmp_B < tmpA ? tmpB : tmpA;
7 :      A[j+1] = tmpB < tmpA ? tmpA : tmpB;
8 :  }
```

Listing 7. Proponowana optymalizacja funkcji sortującej w wariancie dla języka *C* oraz *C++*

Wstępnie analizując listing kodu maszynowego (Listing 8) można zauważyć, iż zastosowanie dodatkowych zmiennych, czyli bezpośrednie użycie potrzebnych rejestrów, umożliwia późniejsze wykorzystanie ich przez kompilator. Pozwala to na wygenerowanie instrukcji z rodziny ***CMOVcc*** przy kompilacji kodu z flagami optymalizacyjnymi.

```
1 : .LBB0_2:
2 :     mov ebx, dword ptr [rdi]
3 :     cmp r8d, 1
4 :     jne .LBB0_5
5 : .LBB0_6:
7 :     mov ebx, dword ptr [...]
8 :     mov edx, dword ptr [...]
9 :     cmp ebx, eax
10:     mov ebp, eax
11:     cmovle ebp, ebx
12:     mov dword ptr [...], ebp
13:     cmovl ebx, eax
14:     cmp edx, ebx
15:     mov eax, ebx
16:     cmovle eax, edx
17:     cmovge ebx, edx
18:     mov dword ptr [...], eax
19:     mov dword ptr [...], ebx
```

Listing 8. Kod maszynowy dla wewnętrznej pętli *for* w kodzie z listingu 7

2.2.1. Alternatywne przypadki testowe

Istnieje słuszne pytanie, czy samo użycie operatora trójargumentowego przyspieszy wykonanie kodu i czy dodatkowe operacje przypisania na wygenerowanych zmiennych jedynie przeszkadzają. Zostały zatem zaproponowane dwa przypadki badanego kodu, mające na celu rozwianie wątpliwości związanych z proponowanym zabiegiem optymalizacyjnym.

Alternatywne wersje zostały przedstawione na listingach 9 oraz 10 – algorytm przedstawiony w tych przypadkach testowych został odpowiednio zmodyfikowany, każda iteracja zewnętrznej pętli *for* zeruje zmienne tymczasowe.

```
1 : int i; int j; int tmp;
2 :
3 :   for (i = 0; i < N; ++i, tmp = 0)
4 :   for (j = 0; j < N - i; ++j) {
5 :    if(A[j] > A[j+1]) tmp = A[j];
6 :    if(A[j] > A[j+1]) A[j] = A[j+1];
7 :    if(tmp > A[j+1])  A[j+1] = tmp;
8 :  }
```

Listing 9. Wersja doświadczalna z instrukcjami warunkowymi w postaci *if*

```
1 : int i; int j; int tmp;
2 :
3 : for (i = 0; i < N; ++i, tmp = 0)
4 :  for (j = 0; j < N - i; ++j) {
5 :   tmp = A[j] > A[j+1] ?
6 :         A[j] : tmp;
7 :   A[j] = A[j] > A[j+1] ?
8 :          A[j+1] : A[j];
9 :   A[j+1] = tmp > A[j+1] ?
10:            tmp : A[j+1];
11:   }
```

Listing 10. Wersja doświadczalna z instrukcjami warunkowymi w postaci operatora trójargumentowego

3. Środowisko testowe

W celu maksymalnego ujednolicenia testów wszystkie badania zostały przeprowadzone na platformie z systemem operacyjnym *Linux* w wersji jądra 4.18. Na potrzeby przeprowadzenia pomiarów zostały ręcznie wyłączone specjalistyczne funkcje procesora tj. *Intel Hyper-Threading* oraz *Intel Turbo Boost*. Z uwagi na metodę pomiarów został wyeliminowany czynnik związany z czasem uruchomienia środowiska oraz niestabilnościami systemowymi. Wyniki uzyskane w ten sposób nadają się do porównywania jedynie w obrębie jednej technologii — testowany jest proponowany zabieg optymalizacyjny, niestety nie jest możliwe porównanie wydajności różnych środowisk.

3.1. Specyfikacja komputera używanego w badaniach

Zebrane pomiary zostały przeprowadzone na komputerze o następujących parametrach:

- **Procesor:** Intel Xeon Processor E5-2680 v2
- **Pamięć RAM:** 8x Patriot Signature Line DDR3 4GB 1600 CL9 (32GB)
- **Płyta główna:** Asus P9X79

3.2. Badane kompilatory oraz środowiska maszyn wirtualnych

3.2.1. Język C oraz C++

Wszystkie cztery zaprezentowane kody testowe w wersji dla *języka C* oraz *C++* zostały skompilowane z flagą optymalizacyjną *-O2*. Wykorzystane kompilatory to *gcc/g++* w wersji 8.2 oraz *clang/clang++* w wersji 7.0.

3.2.2. Język Java

Język *Java* został wybrany jako alternatywa kodu wykonywanego przez maszynę wirtualną. Testowane optymalizacje zostały kosmetycznie dopasowane do syntaktyki języka. Testy zostały przeprowadzone przy użyciu maszyny wirtualnej *OpenJDK* w wersji 10.0.

4. Eksperymentalne testy wydajności proponowanej dekompozycji

Z uwagi na różnice syntaktyczne między wybranymi językami, badaniom zostały poddane odpowiednie kody źródłowe, dostosowane do technologii, analogiczne do prezentowanych na listingach 5, 7, 9 oraz 10. W dalszych opisach badań zostały one oznaczone odpowiednio literami: A, B, C, D – zgodnie z nagłówkami prezentowanego kodu.

4.1. Metodyka badań

Dla każdego przypadku testowego wykonano 12 pomiarów, następnie wyselekcjonowano wyniki środkowe z pominięciem pierwszej oraz ostatniej obserwacji. Uzyskane w ten sposób 10 pomiarów posłużyło do wyliczenia średniego czasu wykonania kodu dla określonej próbki oraz pozwoliło oszacować niestabilność pomiarów przez obliczenie odchylenia standardowego. Przykładowe zestawy wyników dla *języka C* zostały przedstawione w tabelach 1 i 2.

Tabela 1. Wyniki pomiarów testów kodu w wersji języka *C* dla kompilatora *gcc*

	Dane posortowane		**Dane odwrotnie posortowane**		**Dane *losowe***	
	Czas wykonania [s]	Odchylenie standardowe	Czas wykonania [s]	Odchylenie standardowe	Czas wykonania [s]	Odchylenie standardowe
A	3,6144	0,00646	8,9844	0,01815	17,6281	0,02365
B	6,6725	0,04351	6,6529	0,02010	6,6395	0,00529
C	5,3734	0,02154	8,9550	0,10740	17,1616	0,02111
D	11,0991	0,04018	11,4413	0,25376	11,1078	0,01673

Tabela 2. Wyniki pomiarów testów kodu w wersji języka *C* dla kompilatora *clang*

	Dane posortowane		**Dane odwrotnie posortowane**		**Dane *losowe***	
	Czas wykonania [s]	Odchylenie standardowe	Czas wykonania [s]	Odchylenie standardowe	Czas wykonania [s]	Odchylenie standardowe
A	2,6912	0,00433	4,4819	0,01362	15,2483	0,02978
B	6,9806	0,02459	6,9876	0,01529	6,9890	0,05597
C	4,6000	0,02459	6,2720	0,00688	17,4015	0,01820
D	7,2144	0,01700	6,3570	0,05931	16,8540	0,01968

Z uwagi na charakterystykę algorytmu wykorzystywanego do prezentacji proponowanej optymalizacji, były testowane następujące trzy warianty danych wejściowych:

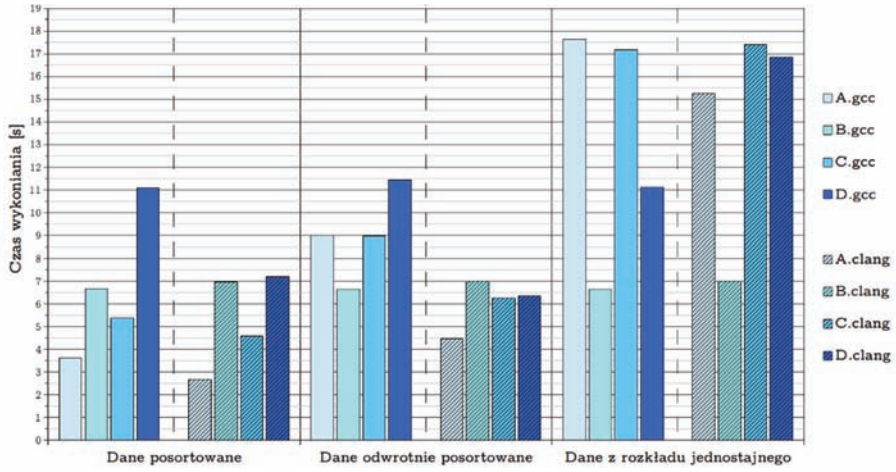
- dane w wersji posortowanej,
- dane w wersji odwrotnie posortowanej,
- dane wygenerowane przez generator liczb losowych o rozkładzie jednostajnym.

W tym miejscu należy również nadmienić, że do wszystkich badań używany był ten sam, wcześniej wygenerowany, zbiór danych wejściowych o liczebności 100 tys. elementów.

4.2. Wyniki oraz wnioski dla kodów kompilowanych AOT

4.2.1. Testy wydajności dla języka C

Średnia arytmetyczna czasów wykonań kodu w wersji dla języka *C* dla kompilatorów *gcc* oraz *clang* została zamieszczona na rysunku 5.



Rysunek 5. Porównanie wydajności proponowanych optymalizacji dla języka *C*

4.2.2. Analiza wyników badań dla języka C

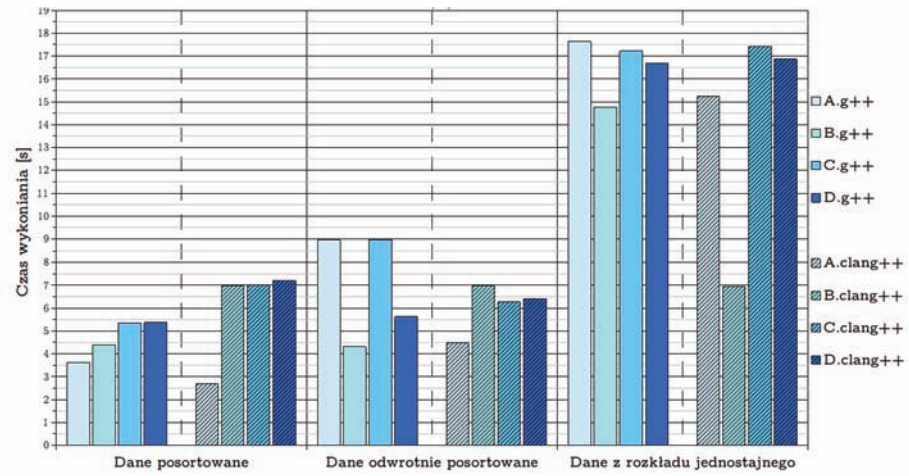
Wstępnie analizując uzyskane wyniki można zauważyć, że za wyjątkiem testów dla danych posortowanych oraz danych odwrotnie posortowanych, gdzie do kompilacji wykorzystywany był kompilator *clang*, proponowana optymalizacja – wariant B jest najwydajniejsza. Kolejnym bardzo ważnym wynikiem badań, może być stały, niezależny od danych wejściowych, czas wykonania operacji sortowania dla algorytmu

używającego instrukcji ***CMOV***. W tym miejscu należy również nadmienić, że testy wykazywały się wysokim poziomem stabilności, gdyż odchylenie standardowe dla większości obserwacji mieściło się w granicy 3% średniej. Wyniki przedstawiono w tabelach 1 i 2.

Czasy wykonania dla kodu optymalizowanego wydają się również nie zależeć od wybranego kompilatora, dla *języka C* proponowana metoda optymalizacji jest uniwersalna. Oczywistym jest, że odpowiedni dobór algorytmu używanego w tej prezentacji pozwolił bardzo dobrze uwydatnić działanie proponowanej optymalizacji. Należy jednak zauważyć, że dana technika optymalizacyjna może być wykorzystywana do wydajniejszego wykonywania bardzo wielu operacji przypisania – wszędzie, gdzie operacja ta zależy od warunku *wyliczanego* z danych. Badanie to pozwala również w sposób bezpośredni udowodnić korzyści, jakie dla procesorów przetwarzających dane potokowo niesie zastosowanie instrukcji przeniesienia warunkowego.

4.2.3. Testy wydajności dla języka C++

Analogicznie do prezentacji wyników dla *języka C* na wykresie (rysunek 6) zaprezentowana średnią arytmetyczną czasów wykonań kodu we wszystkich testowanych.



Rysunek 6. Porównanie wydajności proponowanych optymalizacji dla języka *C++*

4.2.4. Analiza wyników badań dla języka C++

Wyniki z pomiarów czasu uzyskanych dla kodu *C++* prezentują jeszcze lepszą stabilność pomiarów. Odchylenia standardowe średnio mają wielkość około 1% średniej arytmetycznej. Proponowany algorytm sortujący w wersji optymalizowanej w każdym przypadku daje wyniki sortowania w bardzo zbliżonym, stałym czasie. Pojawia się natomiast zdecydowana różnica w czasie wykonania poszczególnych wersji testowych między różnymi kompilatorami.

4.2.5. Rozbieżność między badanymi kompilatorami

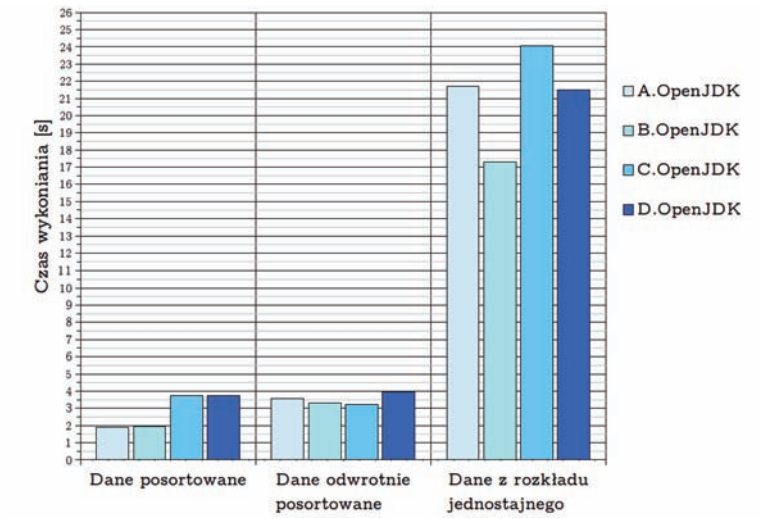
Kod maszynowy wygenerowany przez kompilator *clang* dla danych pesymistycznych oraz rzeczywistych w przypadku wielu badanych próbek jest zdecydowanie szybszy. Diametralną różnicę widać dla ostatniego zestawu danych oraz wariantu testowego B (proponowanej optymalizacji). Zaobserwowana rozbieżność pomiarów możne wskazywać na to, który kompilator lepiej sobie radzi z optymalizacją języka, z oczywistą przewagą dla kompilatora *clang*. Przytoczoną zależność można potraktować jako kolejny wniosek z badań, język *C++* z uwagi na wyższy poziom trudności kompilacji, objawia słabsze możliwości optymalizacji. Należy również wspomnieć o tym, że kod testowy w obu przypadkach był skompilowany w taki sam sposób.

Niezależnie jednak od przytoczonych problemów, badany wariant *B* charakteryzuje się lepszą wydajnością od pozostałych, a przede wszystkim stałym czasem wykonania dla wszystkich przypadków w obrębie kategorii testowych. Prezentowany algorytm wykorzystujący zaproponowaną dekompozycję wykazuje najwyższą stabilność czasów wykonania.

4.3. Wyniki oraz wnioski dla wybranych środowisk maszyn wirtualnych

4.3.1. Testy wydajności dla języka Java

Ostatnim testowanym środowiskiem uruchomieniowym była maszyna wirtualna *JVM*; wyniki pomiarów czasu wykonania algorytmu zostały przedstawione na rysunku 7.



Rysunek 7. Porównanie wydajności proponowanych optymalizacji dla języka *Java*

4.3.2. Analiza wyników badań dla języka Java

Dla przypadków testowych z danymi sekwencyjnymi maszyna wirtualna *Java* radzi sobie zdecydowanie najlepiej. Syntetyczne testy wyraźnie pokazują przewagę i zalety dodatkowego kroku optymalizacji kodu dla kompilacji *JIT*. Próbki analizowane w obrębie tej technologii cechują się również bardzo niską średnią wartością odchylenia standardowego czasów wykonania – poniżej 1% wyniku obserwacji.

W przypadku danych z rozkładu jednostajnego zaobserwować można spore różnice w czasie wykonania. Dodatkowa warstwa abstrakcji, wynikająca z wykorzystania maszyny wirtualnej, wyraźnie pokazuje swój narzut we wszystkich wynikach obserwacji. Mimo delikatnie wyższych niestabilności pomiarów (odchylenie standardowe o wartości maksymalnie 3% wartości pomiaru czasowego), po raz kolejny proponowana optymalizacja okazuje się być wydajniejsza.

5. Podsumowanie

Tematem niniejszych badań było przedstawienie techniki optymalizacyjnej wykorzystującej dekompozycję instrukcji rozgałęziających. Została przedstawiona przy-

kładowa dekompozycja kodu zawierającego instrukcje warunkowe w postaci równoważnej, wymuszającej wybór przez kompilator instrukcji przeniesienia warunkowego. Działanie prezentowanego rozwiązania wykorzystano do modyfikacji standardowego algorytmu sortowania bąbelkowego. W trakcie badań wykazano, że optymalizowany kod źródłowy funkcji sortującej dla większości próbek danych wejściowych był zdecydowanie szybszy niż wersja standardowa.

Dodatkowo warto również wspomnieć, że badany kod dla języka *C* – (Listing 7) w przypadku obu testowanych kompilatorów wykonywał się w stałym czasie, niezależnie od danych wejściowych. Podobna sytuacja miała miejsce kodu napisanego w języku *C++* i skompilowanego przy użyciu kompilatora *clang*.

Uzyskane wnioski potwierdzają, że w chwili obecnej w sytuacjach analogicznego rozgałęzienia przepływu programu wykonywanego przez procesor z rodziny *Intel x86* w celu optymalizacji warto dokonywać ręcznej dekompozycji kodu tak, aby wskazać kompilatorowi możliwość użycia instrukcji przeniesienia warunkowego – ***CMOVcc***.

Literatura

[1] J. Loomis. (2008) *Branch Instructions.* [Online]. http://www.johnloomis.org/microchip/pic32/calc/branch.html

[2] S. Mittal, "A survey of techniques for dynamic branch prediction", *Concurrency and Computation: Practice and Experience* Vol. 3, No. 1, p. e4666, 2019. [Online]. https://arxiv.org/pdf/1804.00261.pdf

[3] *CMOVcc–Conditional Move*. [Online]. https://www.jaist.ac.jp/iscenter-new/mpc/altix/altixdata/opt/intel/vtune/doc/users_guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/mergedProjects/instructions/instruct32_hh/vc35.htm

[4] J. Huang, T.-C. Peng, "Analysis of x86 instruction set usage for DOS/Windows applications and its implication on superscalar design", *IEICE Transactions on Information and Systems* Vol. 85, No. 6, pp. 929-939, 2002. [Online]. https://doi.org/10.1002/cpe.4666

[5] A. Fog. *The microarchitecture of Intel, AMD, and VIA CPUs. An optimization guide for assembly programmers and compiler makers*. 2023 [Online]. https://www.agner.org/optimize/microarchitecture.pdf

[6] S.A. Mahlke, R.E. Hank, J.E. McCormick, D.I. August, W.-W.W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors", *SIGARCH Comput. Archit. News*,Vol. 23, pp. 138-150, May 1995. [Online]. https://doi.org/10.1145/225830.225965

[7] A. Piórkowski, M. Żupnik, "Loop optimization in managed code environments with expressions evaluated only once", *TASK Quarterly. Scientific Bulletin of Academic Computer Centre in Gdansk* Vol. 14, No. 4, pp. 397-404, 2010. [Online]. https://bibliotekanauki.pl/articles/1955308.pdf

Decomposition of Branching Instructions in a Code Optimization Process

Abstract

The subject of this article was the manual modification of source code in order of reducing average execution time of certain repeated conditional move operations. This paper is concerned with proposed optimization by utilizing proper decomposition of branching instructions. This article described in detail conditional assignment operations as well as conditional operations and adequate CPU execution in regard of specific processor family architecture. Every discussed topic was supported with simplified comp diagrams. Experiments were conducted based of analysis of decompiled machine code and advanced analysis of CPU pipelining and branch predication mechanisms. Proposed optimized solutions were investigated based on processor from Intel x86 family.

Examples of applications of mentioned code manipulations with adequate code snippets were presented with adequate performance analysis. As part of this work, the optimization capabilities of the *Clang* compiler for *C* and *C++* were analysed and validated. Effectiveness of proposed optimization was experimentally tested also for compilers from collections *GCC* and for the most popular virtual machine environments such as *JVM*. Gathered cleared results were presented after carried out statistical analysis on column charts and in simplified tables.

Keywords: *code optimization, code decomposition, branching instruction, branching methods, optimization performance analysis*