Krzysztof KRASKA, Tomasz WIERCIŃSKI, Agnieszka KAMIŃSKA
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY, WYDZIAŁ INFORMATYKI,
Żołnierska 49, 71-210 Szczecin

Obliczeniowe szacowanie lokalności danych dla programów ANSI-C

Dr inż. Krzysztof KRASKA

Dr inż. Krzysztof Kraska zatrudniony jest na stanowisku adiunkta w Katedrze Inżynierii Oprogramowania na Wydziale Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie. Do jego zainteresowań badawczych należą: kompilatory optymalizacyjne, przetwarzanie równoległe, lokalność danych.

e-mail: kkraska@wi.zut.edu.pl

Dr inż. Tomasz WIERCIŃSKI

Dr inż. Tomasz Wierciński zatrudniony jest na stanowisku adiunkta w Katedrze Inżynierii Oprogramowania na Wydziale Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie. Do jego zainteresowań badawczych należą: optymalizacja kompilacji, języki opisu sprzętu, automatyczna synteza sprzętu (VHDL, SystemC).

e-mail: twiercinski@wi.zut.edu.pl

Streszczenie

Krytycznym czynnikiem warunkującym wydajność obliczeniową oprogramowania jest lokalność dostępu do danych. Dlatego oczekuje się od narzędzi kompilacji automatyzacji procesu przekształcenia nieoptymalnego kodu do postaci charakteryzującej się wysoką lokalnością danych. W artykule przedstawiono podejście pozwalające na oszacowanie lokalności danych programów na podstawie kodu źródłowego w języku ANSI-C. Omówiono wyniki przeprowadzonych badań eksperymentalnych oraz wskazano kierunki dalszych prac.

Słowa kluczowe: lokalność danych, pamięć podręczna, blokowanie, kompilatory optymalizujące.

Estimation of data locality for ANSI-C source codes

Abstract

Good data locality, comprehended as such placement of program data in memory that program data requested by the processor are available immediately on demand, is a critical software requirement for achieving high efficiency in data processing. One of the ways to achieve good data locality is to transform source codes at the compilation stage so as to improve their usage of the cache memory and, thus, fully benefit from the concept of memory hierarchy. Modern compilers are expected to carry out this kind of optimization automatically, by adopting relevant transformations. In order to select the transformation which is best for this purpose for a given source code, the compiler should be able to compare, from this point of view, the available transformations and indicate the one that produces a semantically identical code of the shortest execution time possible. The paper briefly describes Wolfe's method of estimating data locality based on calculations carried out directly on the source code under analysis, without any need to carry out time consuming compilation of the source code to its executable form and to collect memory access metrics at run time. The paper also presents in outline how the authors implemented in C++ a software module estimating data locality for ANSI-C source codes based on Wolfe's method. The paper discusses the results of adopting the proposed approach to some selected source codes and indicates directions of further works.

Keywords: data locality, cache memory, tiling, optimizing compilers.

Mgr inż. Agnieszka KAMIŃSKA

Mgr inż. Agnieszka Kamińska jest doktorantką w Katedrze Inżynierii Oprogramowania na Wydziale Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie. Do jej zainteresowań badawczych należą: optymalizacja kompilacji, lokalność danych, przetwarzanie równoległe.

e-mail: agkaminska@wi.zut.edu.pl

1. Wstęp

Jednym z istotniejszych problemów współczesnego sprzętu komputerowego jest występowanie znaczącej dysproporcji pomiędzy szybkością procesorów a czasem dostępu do podsystemu pamięci, co skutkuje ograniczeniem wydajności przetwarzania danych [1]. Hierarchiczna organizacja pamięci stwarza znaczące możliwości minimalizacji negatywnych konsekwencji wskazanej rozbieżności.

W idealnej sytuacji, wszystkie dane potrzebne podczas wykonywania programu znajdują się w pamięci położonej możliwie najbliżej procesora. Ponieważ dla większości programów pojemność położonych najwyżej w hierarchii pamięci rejestrów procesora jest zbyt mała w stosunku do rozmiaru przetwarzanych danych, stąd szczególnego znaczenia nabiera pamięć podręczna jako następny poziom hierarchii.

Aby efektywnie wykorzystać pamięć podręczną, niezbędne jest zapewnienie dobrej lokalności znajdujących się w niej danych. Oznacza to, że dane potrzebne procesorowi mają być dostępne natychmiast po zgłoszeniu na nie zapotrzebowania. W tym kontekście, dobrą lokalność danych można utożsamiać z odpowiednim rozmieszczeniem danych w pamięci podręcznej podczas wykonywania programu, co z kolei można osiągnąć odpowiednio przekształcając jego kod źródłowy. Szczególne znaczenie dla poprawy lokalności mają pętle programowe, ponieważ są one tym miejscem w programie, gdzie realizowana jest większość operacji związanych z przetwarzaniem danych. Problem stanowi jednak ocena lokalności danych poszczególnych postaci równoważnych kodu źródłowego (bez kompilacji do postaci wykonywalnej oraz jego uruchamiania) celem wybrania wersji, której czas wykonania będzie najkrótszy.

Celem niniejszego artykułu jest: (1) przedstawienie zaproponowanej przez Wolfe'a [2] metody obliczeniowego szacowania lokalności danych, (2) omówienie implementacji autorskiego modułu szacowania lokalności danych wg metody Wolfe'a, (3) przedstawienie i omówienie wyników zastosowania modułu dla wybranych pętli programowych oraz sformułowanie na podstawie uzyskanych wyników kierunków dalszych prac.

2. Szacowanie lokalności danych

Przetwarzanie danych w pętlach programowych polega na wielokrotnym wykonaniu ustalonej sekwencji operacji na zmieniających się danych. Pętla może wiele razy odwoływać się do danych pochodzących z tej samej lokalizacji w pamięci lub lokalizacji sąsiadujących. Sytuacje te określa się odpowiednio jako czasowe ponowne użycie (ang. temporal reuse) oraz przestrzenne ponowne użycie danych (ang. spatial reuse) [2, 3]. Pętle programowe są tymi miejscami w kodzie programu, które cechują się potencjalnie największym ponownym użyciem danych.

Występowanie ponownego użycia danych w pętli programowej stwarza możliwość efektywnego wykorzystania pamięci podręcznej, ponieważ można wielokrotnie odwoływać się do danych raz pobranych i znajdujących się pamięci podręcznej, zamiast każdo-

razowo odwoływać się do pamięci głównej. Im większe ponowne użycie danych, tym potencjalnie większe możliwości wykorzystania pamięci podręcznej, co w praktyce przekłada się na potencjalnie lepszą lokalność danych i krótszy czas wykonywania programu. Wspomniana poprawa lokalności danych dotyczy zarówno lokalności czasowej jak i lokalności przestrzennej.

Zaproponowana przez Wolfe'a [2] metoda szacowania lokalności danych opiera się na powyższych obserwacjach. Metoda ma zastosowanie do pętli programowych, w których przetwarzane są dane wyrażone poprzez skalary i tablice o indeksach afinicznych. Istotą tej metody jest ocena lokalności danych poprzez współczynniki ponownego użycia danych i wyznaczony na ich podstawie odcisk danych (ang. data footprint). Współczynniki ponownego użycia wyznaczane są dla wszystkich referencji pojawiających się w ciele analizowanych pętli.

Dla każdej referencji obliczane są współczynniki: własnego-czasowego ponownego użycia (ang. self-temporal reuse factor), własnego-przestrzennego ponownego użycia (ang. self-spatial reuse factor) oraz własnego-ponownego użycia (ang. self-reuse factor). Jeżeli dla określonej referencji występuje własne-czasowe lub własne-przestrzenne ponowne użycie to liczba operacji transferu danych z pamięci głównej do pamięci podręcznej, związanych z referencją, zostanie zredukowana:

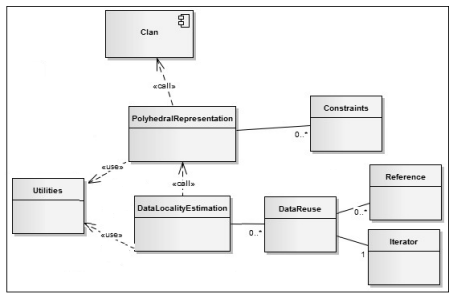
- w przypadku własnego-czasowego ponownego użycia – w najlepszym razie z całkowitej liczby iteracji pętli do 1,
- w przypadku własnego-przestrzennego ponownego użycia – w najlepszym razie o liczbę elementów mieszczących się w jednym wierszu pamięci podręcznej.

Odcisk danych dla określonej referencji wskazuje, ile linii pamięci podręcznej musi zostać nadpisanych danymi pobieranymi z pamięci głównej, aby pobrać wszystkie wartości potrzebne w każdej z iteracji pętli. W przypadku, gdy nie występuje ponowne użycie, odcisk danych jest równy liczbie iteracji pętli; w przeciwnym wypadku będzie zredukowany o wartość ponownego użycia.

Biorąc pod uwagę wszystkie referencje można obliczyć zbiorczy odcisk danych. Zbiorczy odcisk danych określa minimalną pojemność pamięci podręcznej (wyrażoną liczbą linii lub, po odpowiednim przeliczeniu, liczbą bajtów) niezbędną, aby pomieścić wszystkie dane przetwarzane w rozpatrywanej pętli. Jeżeli wartość zbiorczego odcisku danych jest nie większa niż rozmiar pamięci podręcznej i w analizowanym kodzie występuje ponowne użycie danych, wówczas przełoży się to na występowanie lokalności danych na poziomie pamięci podręcznej [2].

3. Oprogramowanie szacujące lokalność danych

Ze względu na brak ogólnie dostępnego oprogramowania szacującego lokalność danych wg metody Wolfe'a, na potrzeby prowadzonych badań został zaimplementowany w języku C++ własny moduł szacujący lokalność danych wg ww. metody [4]. Moduł wykorzystuje zewnętrzną bibliotekę Clan [5]. Rysunek 1 przedstawia strukturę zaimplementowanego modułu.



Rys. 1. Struktura zaimplementowanego modułu do szacowania lokalności danych
Fig. 1. Structure of the implemented module for data locality estimation

Klasa PolyhedralRepresentation odpowiada za przekształcenie wskazanych pętli do modelu wielościennego, z wykorzystaniem zewnętrznej biblioteki Clan. Parametry występujące w granicach rozpatrywanych pętli są zapamiętywane w wektorze elementów typu Constraint.

Klasa DataLocalityEstimation wylicza wartości współczynników ponownego użycia i odcisku danych dla wszystkich rozpatrywanych pętli. Informacje na temat ponownego użycia i odcisku danych dla każdej z rozpatrywanych pętli przechowywane są w wektorze złożonym z elementów typu DataReuse. DataReuse jest strukturą przechowującą informacje o konkretnej pętli, m.in. o iteratorze związanym z tą pętlą (klasa Iterator) oraz o referencjach pojawiających się w ciele tej pętli (wektor elementów typu Reference). Każdy element typu Reference zawiera informacje o opisywanej referencji, m.in. ponownym użyciu oraz odcisku danych.

4. Badania eksperymentalne

Do realizacji badań eksperymentalnych wykorzystano środowisko sprzętowo-programowe przedstawione w tabeli 1.

Tab. 1. Charakterystyka środowiska testowego
Tab. 1. Parameters of the test environment

Procesor	Intel Core 2 Quad Q6600
Liczba rdzeni / wątków	4 / 4
Pamięć podręczna L1 Data	4 x 32 KB, 8-way set associative, 64-byte line size
System operacyjny	Linux Ubuntu 9.04 - Jaunty Jackalope
Kompilator	gcc (Ubuntu 4.3.3-5ubuntu4) 4.3.3
Optymalizacja kompilacji	Wyłączona (kompilacja z opcją –O0)

Badania eksperymentalne zostały przeprowadzone dla algorytmów numerycznych rozwiązywania układów równań liniowych metodami Gaussa-Seidla oraz Jacobiego. Dane przetwarzane w obu kodach były typu float; jeden element typu float zajmował 4 B pamięci.

Testy przeprowadzono dla bloków mieszczących się w całości w pamięci podręcznej. Maksymalny rozmiar kwadratowego bloku mieszczącego się w 32 KB pamięci L1D spełnia zależność: $BLOCK\_SIZE^2$ [elementów] $\leq$ 32 768 [B] / 4 [B/element], czyli BLOCK_SIZE $\leq$ 90,51 (zatem maksymalny rozmiar bloku kwadratowego elementów typu float to 90 x 90).

Każdy z badanych kodów został poddany blokowaniu. Zastosowane zostały bloki kwadratowe o wymiarach BLOCK_SIZE x BLOCK_SIZE, przy czym przyjęto następujące wielkości BLOCK_SIZE: 16, 32, 48, 64, 80, 90. Dla każdego rozmiaru bloku i dla każdego z badanych kodów przeprowadzono po 5 niezależnych pomiarów czasu wykonania programu z blokowaniem, a otrzymane wyniki uśredniono w celu dalszej analizy. Dodatkowo, dla każdego z kodów i dla każdego rozmiaru bloku dokonano oszacowania odcisku danych

Wyjściowe kody źródłowe i kody po blokowaniu, wraz z rezultatami przeprowadzonych testów, zostały zamieszczone poniżej.

Gauss-Seidel

Wersja kodu bez blokowania:

```
float a[500][500];
for(t=0; t<=499; t++)
  for(i=1; i<=498; i++)
    for(j=1; j<=498; j++)
      a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1]
              + a[i][j-1] + a[i][j] + a[i][j+1]
              + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
```

Wersja kodu z blokowaniem:

```
#define MATRIX_SIZE 500
float a[MATRIX_SIZE][MATRIX_SIZE];
for(t=0; t<=499; t++)
  for(it=1; it<=MATRIX_SIZE-2;it=it+BLOCK_SIZE)
    for(jt=1;jt<=MATRIX_SIZE-2;jt=jt+BLOCK_SIZE)
      for(i=it; i<min(it+BLOCK_SIZE,MATRIX_SIZE-1); i++)
        for(j=jt; j<min(jt+BLOCK_SIZE,MATRIX_SIZE-1); j++)
          a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1]
                  + a[i][j-1] + a[i][j] + a[i][j+1]
                  + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
```

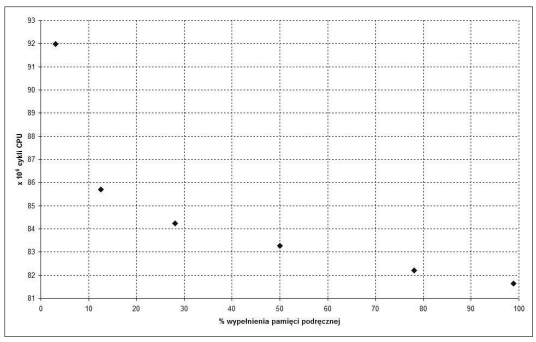
Podsumowanie wyników testów (średnia z 5 niezależnych pomiarów) przedstawia tabela 2. Graficzną prezentację wyników przedstawia rysunek 2.

Dla najmniejszego zbadanego bloku, tj. 16 x 16 elementów typu float, szacunkowy odcisk danych wyznaczony zgodnie z metodą Wolfe'a wynosił 1024 B, co stanowi 3,125% dostępnej pamięci podręcznej L1D. Oznacza to, że przy takim rozmiarze bloku, wykorzystanie pamięci podręcznej jest bardzo słabe. Przekłada się to na czas wykonania programu który, dla bloku o wymiarze 16, wynosił średnio 91,982 x 10^6 cykli CPU.

Ze wzrostem rozmiaru bloku, co jest tożsame ze wzrostem stopnia wypełnienia pamięci podręcznej danymi przetwarzanymi w programie, zaobserwowano skrócenie czasu wykonania programu. Krzywa pokazująca czas wykonania programu w zależności od stopnia wypełnienia pamięci podręcznej przez blok była ściśle monotoniczna i malejąca – najlepszy wynik (tj. najkrótszy czas wykonania programu) zaobserwowano dla największego bloku, wypełniającego pamięć podręczną w ok. 100% (blok 90 x 90 elementów typu float – szacunkowy odcisk danych, wyznaczony zgodnie z metodą Wolfe'a, wynosił 32400 B).

Tab. 2. Wyniki testów – Gauss-Seidel
Tab. 2. Test results – Gauss-Seidel

Wymiar bloku BLOCK_ SIZE	Średni czas wykonania programu [cykle CPU x 10^6]	Odcisk danych		Wypełnienie pamięci podręcznej [%]
		Wyrażony w liniach pamięci podręcznej	Wyrażony w B	
16	91,982	16	1024	3,125
32	85,712	64	4096	12,5
48	84,248	144	9216	28,125
64	83,276	256	16384	50
80	82,208	400	25600	78,125
90	81,634	506,25	32400	98,877



Rys. 2. Gauss-Seidel – stopień wypełnienia pamięci podręcznej przez blok danych a czas wykonania programu
Fig. 2. Gauss-Seidel – percentage of the cache memory filled by a data block vs. execution time

Jacobi

Wersja kodu bez blokowania:

```
float a[1000][1000];
for(p=0; p<8000; p++)
  for(t=1; t<1000; t++)
    for(i=1; i<999; i++)
      a[t][i]=0.33*(a[t-1][i] + a[t-1][i-1] + a[t-1][i+1]);
```

Wersja kodu z blokowaniem:

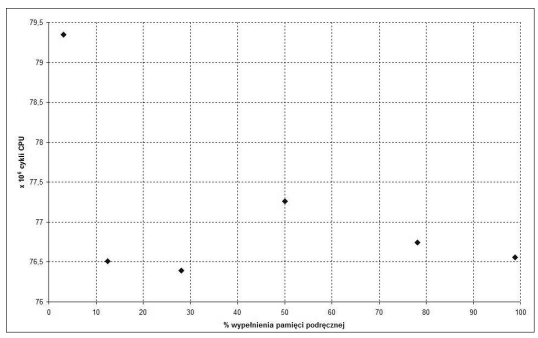
```
#define MATRIX_SIZE 1000
float a[MATRIX_SIZE][MATRIX_SIZE];
for(p=0; p<8000; p++)
  for(tt=1; tt<MATRIX_SIZE; tt=tt+BLOCK_SIZE)
    for(it=1; it<MATRIX_SIZE-1; it=it+BLOCK_SIZE)
      for(t=tt; t<min(tt+BLOCK_SIZE,MATRIX_SIZE); t++)
        for(i=it; i<min(it+BLOCK_SIZE,MATRIX_SIZE-1); i++)
          a[t][i]=0.33*(a[t-1][i] + a[t-1][i-1] + a[t-1][i+1]);
```

Podsumowanie wyników testów (średnia z 5 niezależnych pomiarów) przedstawia tabela 3. Graficzną prezentację wyników przedstawia rysunek 3.

Ze wzrostem rozmiaru bloku, co jest tożsame ze wzrostem stopnia wypełnienia pamięci podręcznej danymi przetwarzanymi w programie, zaobserwowano generalnie skrócenie czasu wykonania programu z pewnymi drobnymi wahaniami od tego trendu. Jednakże, ta sytuacja wynika z właściwości samej pamięci podręcznej i faktu, że blokowanie jest optymalizacją o charakterze probabilistycznym [2].

Tab. 3. Wyniki testów – Jacobi
Tab. 3. Test results – Jacobi

Wymiar bloku BLOCK_ SIZE	Średni czas wykonania programu [cykle CPU x 10^6]	Odcisk danych		Wypełnienie pamięci podręcznej [%]
		Wyrażony w liniach pamięci podręcznej	Wyrażony w B	
16	79,35	16	1024	3,125
32	76,51	64	4096	12,5
48	76,394	144	9216	28,125
64	77,264	256	16384	50
80	76,746	400	25600	78,125
90	76,55667	506,25	32400	98,877



Rys. 3. Jacobi – stopień wypełnienia pamięci podręcznej przez blok danych a czas wykonania programu
Fig. 3. Jacobi – percentage of the cache memory filled by a data block vs. execution time

5. Wnioski

Uzyskane w przeprowadzonych badaniach wyniki szacowania lokalności danych współczynnikiem odcisku danych, obliczonym na podstawie metody Wolfe'a, są zbieżne z wydajnością programu zmierzoną rzeczywistym czasem wykonania. W obu badanych przykładach do określenia najlepszej, pod kątem lokalności danych, postaci kodu wystarczy oszacować współczynnik odcisku danych na podstawie kodu źródłowego, bez konieczności czasochłonnej kompilacji programu do postaci wykonywalnej i gromadzenia metryk wykorzystania pamięci podręcznej w trakcie jego wykonania. W przyszłych pracach planowane jest przeprowadzenie badań eksperymentalnych dla znacznie bardziej licznego zbioru pętli pochodzących z uznanych benchmark'ów (NAS, UTDSP) w celu sformułowania bardziej uogólnionych wniosków. Jednocześnie autorzy planują rozszerzenie modelu Wolfe'a do szacowania lokalności danych dla kodu równoległego. Uzyskane wyniki badań w intencji autorów stanowić będą autorski wkład w rozwój najbardziej zaawansowanych narzędzi kompilacji, takich jak kompilator PLUTO [6].

6. Literatura

[1] Stallings W.: Organizacja i architektura systemu komputerowego. Projektowanie systemu a jego wydajność, Wydawnictwa Naukowo-Techniczne, 2004.
[2] Wolfe M.: High Performance Compilers for Parallel Computing, Addison Wesley, 1996.
[3] Wolf M.E., Lam M.S.: A Data Locality Optimizing Algorithm. Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada, 1991.
[4] Kraska K., Kamińska A.: Koncepcja metody zwiększania lokalności danych na poziomie pamięci podręcznej oparta na transformacjach pętli programowych. Metody Informatyki Stosowanej, Nr 2/2010, s. 63-72.
[5] Bastoul C., Cohen A., Girbal S., Sharma S., Temam O.: Putting Polyhedral Loop Transformations to Work, LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958, 2003.
[6] Uday Bondhugula: Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model, Ohio State University 2008.

otrzymano / received: 13.05.2011
przyjęto do druku / accepted: 04.07.2011 *artykuł recenzowany*