Włodzimierz BIELECKI, Krzysztof KRASKA
POLITECHNIKA SZCZECIŃSKA, WYDZIAŁ INFORMATYKI

Zwiększenie wydajności aplikacji wykonywanych w systemach osadzonych poprzez zwiększenie lokalności danych

Prof. dr hab. inż. Włodzimierz BIELECKI

Jest kierownikiem Katerdry Technik Programowania Wydziału Informatyki Politechniki Szczecińskiej. Jego zainteresowania naukowe obejmują przetwarzanie równoległe i rozproszone, teorię kompilacji, języki VHDL i SystemC, syntezę sprzętu i kosyntezę sprzętowo-programową.

e-mail: wbielecki@wi.ps.pl

Dr inż. Krzysztof KRASKA

Ur. 29 grudnia 1974 r. Absolwent Wydziału Informatyki Politechniki Szczecińskiej. Tytuł magistra inżyniera uzyskał w 1999 roku. Stopień naukowy doktora w dyscyplinie informatyka specjalność kompilatory uzyskał w roku 2003 na Wydziale Informatyki Politechniki Szczecińskiej. Główne obszary jego aktywności zawodowej to: kompilatory, przetwarzanie równoległe, metody zwiększenia lokalności danych, języki programowania, języki opisu sprzętu, produkcja oprogramowania.

e-mail: krzysztof.kraska@wi.ps.pl

Streszczenie

Efektywne użycie pamięci jest krytycznym warunkiem uzyskania wysokiej wydajności przez oprogramowanie wykonywane na współczesnych architekturach z hierarchią pamięci. W systemach osadzonych efektywne wykorzystanie pamięci przez aplikacje umożliwia przede wszystkim zmniejszenie wymagań dla sprzętu przy ustalonych kryteriach wydajnościowych, redukcję rozmiaru pamięci jak i zmniejszenie zużycia energii. Wskazane czynniki bezpośrednio wpływają na koszt budowy systemu osadzonego. Osiągnięcie wysokiego poziomu efektywności użycia pamięci wymaga tworzenia oprogramowania uwzględniającego lokalność danych. Oprogramowanie intensywnie eksploatujące pamięć, takie jak chociażby aplikacje multimedialne, zazwyczaj przetwarza w pętlach programowych znaczne ilości danych umieszczonych w tablicach. Sposobem na zwiększenie lokalności takich programów jest transformacja pętli programowych do postaci bardziej optymalnego kodu. W artykule przedstawiono aktualny stan badań w zakresie metod transformacji programów zwiększających lokalność danych, dokonano analizy możliwości szacowania poziomu lokalności danych w pętlach programach i zwiększenia lokalności danych dla pętli doskonale zagnieżdżonych oraz przedstawiono wyniki badań obrazujących efektywność rozważanych transformacji.

Słowa kluczowe: lokalność danych, zrównoleglenie programów, transformacje pętli programowych, kompilatory.

Improving the application performance of embedded systems by increasing data locality

Abstract

The effective use of memory subsystem is the critical condition for software to achieve the high performance on the contemporary architectures with hierarchy of memory. In embedded systems the effective utilization of the memory subsystem mainly enables to decrease requirements for hardware with respect to established performance criteria, reduce the size of memory and decrease the energy consumption. The indicated factors influence on cost of building an embedded system directly. The achievement of high efficiency of memory subsystem requires creating of software with high data locality. Software that intensely explores memory, such as multimedia applications, usually processes within program loops considerable quantities of data placed in arrays. The transformation of program loops to more optimal code is the way on improvement data locality. In the paper, the state of the art of loop transformation methods improving data locality was presented. Additionally, the possibility of estimating a level of data locality and improving data locality for perfectly nested loop were examined. Finally, the results of analysis investigations were introduced illustrating the efficiency of considered transformations.

Keywords: data locality, parallelization of programs, loop transformations, compilers.

1. Wstęp

Współczesne zastosowania naukowe czy inżynierskie wymagają intensywnych obliczeniowo aplikacji skutkując zwiększeniem czasu wykonania kodu programu. Naturalnym sposobem uzyskania przyśpieszenia jest zrównoleglenie programów sekwencyjnych. Co więcej, o ile w ciągu kilku ostatnich dekad szybkość mikroprocesorów rosła w tempie ~50%-100% na rok to czas dostępu do pamięci ulegał poprawie w tempie rocznym zaledwie ~7%. Dlatego oprócz zrównoleglenia programów efektywne użycie pamięci jest równie krytycznym czynnikiem zapewnienia wysokiej wydajności dla aplikacji.

Rozbieżność pomiędzy prędkością procesora a prędkością pamięci częściowo łagodzą współczesne architektury z hierarchią pamięci, w których poziomy pamięci kolejno oddalone od procesora są większe pod względem rozmiaru, lecz cechują się dłuższym czasem dostępu. W przypadku architektur wieloprocesorowych z pamięcią rozproszoną każdy procesor dysponuje własną, szybką pamięcią lokalną a dostęp do pamięci innego procesora traktowany jest jako następny poziom hierarchii pamięci. Efektywne wykorzystanie hierarchii pamięci wymaga od aplikacji optymalizacji lokalności przetwarzanych danych (ang. *data locality*).

Mała lokalność danych jest powszechną cechą wielu istniejących aplikacji, które zazwyczaj intensywnie wykonują w wielu pętlach programowych afiniczne referencje do wielkich zbiorów danych (tablic) przekraczających swym rozmiarem szybką acz małą pamięć podręczną procesora. Ze względu na nieoptymalną postać kodu dane w takich aplikacjach często muszą zostać ponownie dostarczone z wolniejszej pamięci. Nieoptymalny kod można poddać transformacji do wydajniejszej postaci podczas kompilacji programu.

Zwiększenie lokalności danych jest istotne nie tylko dla maszyn równoległych, lecz również dla systemów osadzonych, w których programowe elementy systemu są przetwarzane przez programowalne procesory. Zwiększenie lokalności danych w tych systemach umożliwia redukcję wymaganego rozmiaru pamięci oraz zmniejszenie zużycia energii. Redukcja wymaganego rozmiaru pamięci konsekwentnie powoduje zmniejszenie opóźnienia dostępu do komórek pamięci oraz ograniczenie zajmowanego obszaru układu a przez to zmniejszenie zużycia energii. Zmniejszenie odwołań do pamięci zewnętrznej dodatkowo ogranicza aktywność energetyczną. Jednocześnie zwiększenie odwołań do szybkiej pamięci lokalnej znacząco zwiększa wydajność sprzętowych oraz programowych implementacji algorytmów w systemach osadzonych.

2. Metody zwiększania lokalności danych

Opracowano wiele metod optymalizacji wydajności pamięci poprzez zwiększenie lokalności danych. Najprostszą formą zwiększenia lokalności danych jest zmiana rozmieszczenia struktur danych programu. Dla przykładu, w prostej operacji mnożenia macierzy:

```
for(i = 0; i < n; i++)
   for(j = 0; j < n; j++)
      for(k = 0; k < n; k++)
         X[i,j] = X[i, j] + Y[i,k]*Z[k,j];
```

przy założeniu typowego dla języka C wierszowego porządku układu macierzy w pamięci, zmiana sposobu rozmieszczenia w pamięci kolumn macierzy Z do postaci obszarów ciągłych umożliwia zwiększenie ponownego wykorzystania danych dostarczanych z wierszy pamięci podręcznej procesora. Zastosowanie takiego podejścia jest jednak dość ograniczone, ponieważ w rzeczywistych programach macierze są przeważnie wykorzystywane w wielu różnych operacjach. W przypadku zamiany miejsc Z i Y w innej operacji mnożenia macierzy zmiana rozmieszczenia danych spowodowałaby znaczną utratę wydajności.

Generalizując, algorytmy zwiększania lokalności danych tworzą następujące grupy metod:

(1) blokowanie (ang. *blocking* lub ang. *tiling*) i transformacje pętli (ang. *loop transformations*), polegające na zmianie porządku operacji w programie w celu zwiększenia ponownego użycia danych,
(2) kontrakcja macierzy (ang. *array contraction*) polegająca na redukcji wymiarowości (ang. *dimensionality*) macierzy w programie.

2.1. Blokowanie i transformacje pętli

Czasami zwiększenie lokalności danych możliwe jest poprzez zmianę porządku wykonania instrukcji w pętlach tak, aby ponowne użycie danych następowało w krótkich odstępach czasu. Wiele z tych technik polega na wyborze ad *hoc serii* transformacji spośród siedmiu podstawowych transformacji pętli programowych, tj: *fusion*, *fission* (inaczej *distribution*), *re-indexing*, *scaling*, *permutation* (inaczej *interchange*), *reversal*, *skewing*. Przewidzenie łącznego efektu zastosowania serii tych technik bez wykonania transformacji i analizy powstałego kodu jest trudne a przez to wybór najlepszej sekwencji transformacji ze wszystkich możliwych kombinacji staje się problematyczny. Dlatego częstym podejściem jest wykonanie ustalonej sekwencji faz, w których realizowany jest tylko ograniczony zbiór transformacji. Niestety, współczesne kompilatory zazwyczaj dokonują optymalizacji równoległości oraz lokalności danych ograniczając się do pętli doskonale zagnieżdżonych [2].

Innym i najczęstszym sposobem reorganizacji porządku wykonania iteracji pętli zwiększającym lokalność danych jest blokowanie (ang. *blocking* lub ang. *tiling*) [1]. Technika ta polega na podziale macierzy na mniejsze bloki tak, aby mieściły się w całości w pamięci podręcznej procesora i wykonywaniu na nich operacji. W ten sposób dostęp do wszystkich elementów bloku odbywa się w krótkich odstępach czasu na szybkiej pamięci podręcznej. Technika blokowania może zostać zastosowana do każdego poziomu hierarchii pamięci w zależności od przyjętych rozmiarów bloku. Podobnie możliwe jest rozdzielenie bloków między procesory w celu minimalizacji transferu danych pomiędzy nimi.

Poniżej przedstawiono przykład zastosowania blokowania dla prostej operacji mnożenia macierzy, gdzie B równe jest długości wiersza pamięci podręcznej procesora.

```
for (i=0; i<n; i++)
 for (j=0; j<n; j++)
  for (k=0; k<n; k++)
   X[i,j]=X[i,j]+Y[i,k]*Z[k,j];
```

→

```
for (ii=0; ii<n; ii+=B)
 for (jj=0; jj<n; jj+=B)
  for (kk=0; kk<n; kk+=B)
   for (i=ii; i<ii+B; i++)
    for (j=jj; j<jj+B; j++)
     for (k=kk; k<kk+B; k++)
      X[i,j]=X[i,j]+Y[i,k]*Z[k,j];
```

Badania dowodzą, że blokowanie na maszynach jednoprocesorowych zwiększa wydajność trzykrotnie natomiast na maszynach wieloprocesorowych przyrost jest niemal liniowy. Jednak blokowanie znajduje zastosowanie jedynie do pętli doskonale zagnieżdżonych.

2.2. Kontrakcja macierzy

Kontrakcja macierzy jest techniką pierwotnie zaproponowaną do efektywnego przetwarzania tablic w konstrukcjach języka Fortran90 oraz HPF. Kompilator zazwyczaj przekształca każdą operację tablicową na pętlę programową oraz wykorzystuje tablice tymczasowe, w których utrzymuje chwilowe wyniki wyrażeń. Bezpośrednie uruchomienie takiego kodu na maszynach z pamięcią podręczną skutkuje kiepską wydajnością nie tylko ze względu na większą ilość działań na pamięci, ale również na większy obszar roboczy danych redukujący skuteczność pamięci podręcznej. Efektywną optymalizacją tak powstałych programów jest fuzja pętli tak, aby każdy element tablicy tymczasowej był definiowany i wykorzystywany w tej samej iteracji. Stąd zmienna skalarna może zostać użyta do utrzymywania wartości różnych elementów tablicy. Umieszczenie zmiennej skalarnej w rejestrze redukuje liczbę działań na pamięci oraz obszar roboczy danych programu.

Poniżej przedstawiono przykład kontrakcji macierzy dla prostego wyrażenia dodawania trzech n-elementowych macierzy – w wygenerowanym przez kompilator kodzie dwie pętle zostały połączone a tymczasowa macierz T „zawężona” (ang. contract).

```
R = X + Y + Z
```

→

```
for i = 0 to n-1
  T[i] = X[i] + Y[i]
for i = 0 to n-1
  R[i] = T[i] + Z[i]
```

→

```
For i = 0 to n-1
  T = X[i] + Y[i]
  R[i] = T + Z[i]
```

Konwersja macierzy T do postaci zmiennej skalarnej umożliwia jej umieszczenie w rejestrze procesora oraz ogranicza obszar roboczy danych programu.

Technika kontrakcji macierzy jest istotna dla optymalizacji istniejących aplikacji przeznaczonych dla komputerów wektorowych, gdzie programiści często umyślnie rozszerzali zmienne skalarne, aby kolejne iteracje pętli wykonywały operacje na kolejnych lokalizacjach pamięci. O ile proste zastosowanie techniki *fusion* może być odpowiednie dla konstrukcji językowych operujących na tablicach to wyszukane transformacje programu mogą ukazać znacznie więcej możliwości kontrakcji macierzy w programach.

3. Szacowanie poziomu lokalności danych

Lokalność danych określa stosunek referencji adresowych pamięci lokalnej do referencji adresowych pamięci zewnętrznej, tj: $R(local)/\ (R(local)+R(global))$. Wartość bliska 1 oznacza, że większość referencji odnosi się do pamięci lokalnej; wartość bliska 0 oznacza, że większość referencji odnosi się do pamięci zewnętrznej.

Zwiększenie odwołań do pamięci lokalnej we współczesnych architekturach w istocie polega na optymalizacji wykorzystania szybkiej pamięci podręcznej procesora. Przeprowadzenie szczegółowej analizy lokalności danych pętli programowej jest zbyt kosztowne, dlatego w praktyce stosuje się szacowanie lokalności danych na podstawie funkcji $C_M(L_i)$ – *kosztu pamięci najgłębszego zagnieżdżenia* pętli L_i (ang. *innermost memory cost*) [11]. Metoda polega na zidentyfikowaniu wszystkich referencji w gnieździe pętli i przypisaniu każdej referencji kosztu pamięci pod względem możliwych chybień w pamięci podręcznej, które byłyby ponoszone wyłącznie dla danego gniazda. Poszczególne ustalone koszty referencji są przemnażane przez współczynniki związane z pozostałymi pętlami zaś całkowity koszt jest wyliczany poprzez zsumowanie kosztów poszczególnych referencji.

4. Zwiększenie lokalności danych dla pętli doskonale zagnieżdżonych

W [10] zaproponowana została nowa metoda ekstrakcji wątków niewymagających synchronizacji dla pętli doskonale zagnieżdżonych, polegająca na:

(1) usunięciu nadmiarowych zależności pomiędzy iteracjami pętli,
(2) wyłonieniu zbioru wszystkich źródeł ostatecznych zależności a następnie wykonaniu jego podziału na dwa podzbiory:
 a) zawierający źródła wątków wymagających synchronizacji,
 b) zawierający źródła wątków niewymagających synchronizacji,
(3) znalezieniu iteracji należących do każdego wątku niewymagającego synchronizacji oraz wygenerowaniu kodu zawierającego w porządku leksykograficznym wątki niewymagające synchronizacji oraz iteracje każdego z wątków.

W sytuacji, gdy wątki niewymagające synchronizacji nie mogą zostać wydzielone z całej domeny pętli, algorytm próbuje dokonać ekstrakcji wątków w podprzestrzeni domeny pętli.

Metoda generuje kod pętli zawierającej wydzielone wątki według schematu:

```
(1) Kod wyliczający poszczególne wątki niewymagające synchronizacji
  (2) Kod wyliczający operacje wewnątrz poszczególnych wątków
```

Taki sposób generowania kodu pętli może zostać przedstawiony w następującej postaci pętli doskonale zagnieżdżonej:

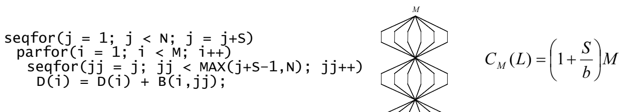
```
L1:      seq do I1 = p1, q1
         …
Lk:          par do Ik = pk, qk
Lk+1:            seq do Im = pm, qm
                     H(I1, …, Ik, Ik+1)
                 enddo
             enddo
         enddo
```

gdzie: sekwencyjne pętle $L_1,...,L_{k-1}$ to gniazda pętli programowej doskonale zagnieżdżonej, dla których nie można dokonać ekstrakcji wątków niewymagających synchronizacji, równoległa pętla L_k wylicza niezależnie wykonywane wątki natomiast sekwencyjna pętla L_{k+1} wylicza wykonanie zależnych iteracji wewnątrz poszczególnych wątków.

Zwiększenie lokalności danych dla kodu wygenerowanego przy użyciu metody przedstawionej w [10] może polegać na zastosowaniu blokowania na poziomie pętli wyliczającej operacje wewnątrz poszczególnych wątków. Należy zauważyć, iż kolejność pętli wyliczającej poszczególne wątki niewymagające synchronizacji oraz pętli wyliczającej operacje wewnątrz poszczególnych wątków nie może zostać zmieniona ze względu na definiujący przez metodę schemat odzwierciedlania wątków w kodzie programu.

Dla przykładu, pętla programowana o strukturze charakterystycznej dla metody ekstrakcji niezależnych wątków:

może zostać poddana blokowaniu (przy użyciu podstawowego algorytmu metody blokowania *stripmine&interchange* [11]) w gnieździe pętli wewnętrznej przyjmując postać:



Wykorzystując funkcję $C_M(L_i)$, ponieważ $S<N$ stąd oczywistym jest, iż:

$$\left(1 + \frac{S}{b}\right)M < \left(1 + \frac{N}{b}\right)M$$

co wskazuje na znaczne niższy szacowany koszt pamięci przy zastosowaniu blokowania. Należy jednak zauważyć, że szacunki nie uwzględniają kosztów związanych z częstszym tworzeniem i niszczeniem wątków.

Warto jednocześnie wskazać, że zastosowanie techniki blokowania w gnieździe pętli zewnętrznej parfor(i=1;i<M;i++) – bez względu na poprawność transformacji – w konsekwencji ogranicza liczbę równolegle wykonywanych niezależnych wątków co daje efekt przeciwny w stosunku do celu metody [10] maksymalizującej równoległość kodu programu.

5. Wnioski

W artykule została wykazana możliwość, sposób i efektywność zwiększania lokalności danych przy użyciu znanej z literatury techniki blokowania w pętlach programowych, w których zostały wydzielone niezależne wątki niewymagające synchronizacji przy użyciu nowej metody [10]. Stanowi to nowy element pracy rozwijający wyniki tej metody. Planowany zakres dalszych badań jest następujący. W pierwszej kolejności celem dalszych badań autorów jest przeprowadzenie analizy możliwości zastosowania dotychczas opracowanych algorytmów implementujących metodę blokowania (np.: [1, 3, 4, 5, 6]) oraz algorytmów kontrakcji macierzy (np.: [7, 8, 9]). Jednocześnie niezbędne jest określenie modelu wyboru najkorzystniejszej transformacji pod kątem architektury docelowego środowiska sprzętowego, na którym aplikacja będzie wykonywana, stąd model optymalizacji postaci kodu powinien uwzględniać przydział wątków do procesorów. Działania w tym zakresie wymagają wykonania szeregu badań eksperymentalnych weryfikujących skuteczność uzyskanych wyników analitycznych. Planowane rozwinięcie metody [10] na ekstrakcję wątków niewymagających synchronizacji w pętlach niedoskonale zagnieżdżonych implikuje *explicite* kierunek dalszego rozwoju badań nad zwiększeniem lokalności danych takiego kodu. Równolegle z postępem prac autorzy planują rozwijanie narzędzi implementujących uzyskane wyniki.

6. Literatura

[1] M.E. Wolf, M.S. Lam: A data locality optimizing algorithm, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, 1991.
[2] M. E. Wolf, M. S. Lam: A loop transformation theory and an algorithm to maximize parallelism, Transactions on Parallel and Distributed Systems, 1991.
[3] S. Carr, K. S. McKinley, C. W. Tseng: Compiler optimizations for improving data locality, Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.
[4] K. Kennedy, K. S. McKinley: Optimizing for parallelism and data locality, Proceedings of the 1992 ACM International Conference on Supercomputing, 1992.
[5] N. Ahmed, N. Mateev, K. Pingali: Synthesizing transformations for locality enhancement of imperfectly nested loop nests, Proceedings of the 2000 ACM International Conference on Supercomputing, 2000.
[6] I. Kodukula, N. Ahmed, K. Pingali: Data centric multi level blocking, Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation, 1997.
[7] V. Sarkar, G. R. Gao: Optimization of array accesses by collective loop transformations, Proceedings of the 1991 ACM International Conference on Supercomputing, 1991.
[8] G. R. Gao, R. Olsen, V. Sarkar, R. Thekkath: Collective loop fusion for array contraction, Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, 1992.
[9] Y. Song, R. Xu, C. Wang, Z. Li: Array contraction for memory reduction, Workshop on Solving the Memory Wall Problem, 2000.
[10] W. Bielecki, K. Siedlecki: Extracting synchronization free slices in perfectly nested uniform and non uniform loops, to be published in Electonic Modeling, 2007 (artykuł jest przyjęty do wydania).
[11] R. Allen, K. Kennedy: Optimizing Compilers for Modern Architectures: A Dependence based Approach, Morgan Kaufmann Publishers, 2001.

Artykuł recenzowany