Piotr BŁASZYŃSKI, Maciej POLIWODA
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY, WYDZIAŁ INFORMATYKI, ul. Żołnierska 49, 71-210 Szczecin

Automatyczna ekstrakcja zależności i transformacje pętli w języku C do przekształceń optymalizujących w systemach osadzonych

Dr inż. Piotr BŁASZYŃSKI

Ukończył studia na Wydziale Informatyki Politechniki Szczecińskiej, obronił pracę doktorską w 2004 r. Jest adiunktem w Katedrze Inżynierii Oprogramowania. Jego zainteresowania naukowe to techniki kompilacji, języki programowania.

e-mail: pblaszynski@wi.ps.pl

Dr inż. Maciej POLIWODA

Ukończył studia na Wydziale Informatyki Politechniki Szczecińskiej, obronił pracę doktorską w 2002 r. Jest adiunktem w Katedrze Inżynierii Oprogramowania. Jego zainteresowania naukowe to programowanie równoległe, techniki kompilacji.

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

Streszczenie

W poniższym artykule zaprezentowano metody wyszukiwania pętli programowych i zależności między ich iteracjami. Analiza zależności pozwala na określenie, które fragmenty analizowanego kodu programu mogą zostać wykonane niezależnie. Przedstawiono analizę programów w ANSI C, ze wskazaniem możliwości wykorzystania wyników analizy do zmniejszenia użycia zasobów w systemach osadzonych. Zautomatyzowanie analizy pozwala określić klasy algorytmów i ich implementacje, które mogą być optymalizowane, oraz charakter tych optymalizacji.

Słowa kluczowe: kompilator, systemy osadzone, analiza zależności.

Automatic extraction of dependencies and loop transformations in C language for optimising transformations in embedded systems

Abstract

In the paper there are presented methods of searching for program loops and dependencies between iteration of these loops. Analysis of the dependencies allows determining which parts of the analysed code of the program must be executed sequentially, and which can be executed independently. There are given the results of the analysis of simple algorithms written in ANSI C language, indicating the possibility of using the analysis to reduce the use of resources in embedded systems. Automating the analysis process also allows specifying the types of algorithm classes and their implementation, which can be subjected to optimisation, as well as the nature of these optimisations. In the introduction there is discussed the range of topics dealt with in the paper. In the second paragraph the method for determining the dependencies between instructions is described. Fig. 2 shows the iteration space of the sample loop. The third paragraph presents the basic techniques of loop transformation, which are possible due to the dependency analysis between iterations. The above (and others obtained during the tests) results show the possibilities of allowing the code designer to improve devices and embedded systems. Transformation selection algorithms need different types of dependence description as an input data. The aim of the project described in the paper, which is devoted to the dependency analysis is to provide a variety of the dependency collection description.

Keywords: compiler, embedded systems, dependency analysis.

1. Wstęp

Proces optymalizacji kodu programów przeznaczonych do wykonania przez systemy osadzone jest bardzo złożony i czasochłonny. W procesie tym można wyróżnić kluczowe etapy, takie jak.

- Wyznaczenie fragmentu kodu do optymalizacji.
- Analiza zależności między instrukcjami w optymalizowanym kodzie.
- Wyznaczenie odpowiednich transformacji.
- Wygenerowanie kodu wynikowego.

Fragmentami kodu, których optymalizacja przynosi największe efekty są pętle. Jest tak, ponieważ w pętlach przetwarza się duże ilości danych wykonując wielokrotnie te same instrukcje, co powoduje, że czas wykonania kodu pętli stanowi znaczną część czasu wykonania całego programu. Sposób wykonania poszczególnych iteracji pętli ma wpływ na wykorzystanie dostępnych zasobów (pamięci podręcznej), co ma wpływ na ilość zużywanej energii. Automatyczne wykrywanie i oznaczanie fragmentów kodu będących pętlami jest procesem trywialnym i nie będzie dalej rozważanie w niniejszym artykule. Analiza zależności określa, które fragmenty analizowanego kodu programu muszą być wykonywane sekwencyjnie (instrukcja po instrukcji), a które mogą zostać wykonane niezależnie od siebie (kolejność wykonania instrukcji nie ma znaczenia). Na podstawie przeprowadzonej analizy zależności dokonuje się transformacji, przekształcenia kodu do postaci, w której zasoby systemu są lepiej wykorzystane. Natomiast wyniki działania kodu po transformacji są identyczne z wynikami otrzymanymi przed transformacją.

2. Zależności między instrukcjami i iteracjami

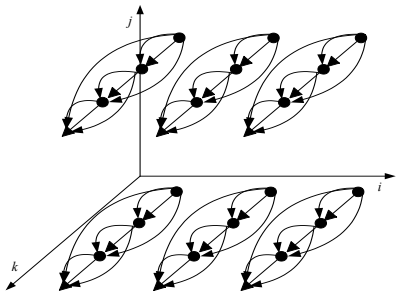
Określenie wszystkich zależnych iteracji i relacji między nimi pozwala na wyodrębnienie iteracji niezależnych i ich podział między niezależnie pracujące wątki. Metody analizy zależności między iteracjami były przedmiotem wielu prac [1, 2, 3], w których przedstawiono sposoby opisu zależności za pomocą wektorów zależności, relacji, grafów zależności.

Jeśli w dwóch iteracjach opisanych wektorami $\overline{I}$ i $\overline{J}$, spełniającymi leksykograficzną zależność $\overline{I} \prec \overline{J}$, następuje odwołanie do tego samego obszaru pamięci, przy czym jedno z tych odwołań musi dotyczyć operacji zapisu, mówimy, że są zależne. Zależność miedzy tymi wektorami można opisać za pomocą wektora zależności $\overline{K} = \overline{J} - \overline{I}$, który spełnia warunek $\overline{K} \succ \overline{0}$.

```
for (i = 0; i < s1; i++)
    for (j = 0; j < s2; j++)
        for (k = 0; k < s3; k++)
        a[i][j] += b[i][k] * c[k][j];
```

Rys. 1. Przykładowa pętla
Fig. 1. Example loop

Na rysunku rys. 2 zostały przedstawione zależności prosta, odwrotna i po wyjściu występujące między iteracjami pętli rys. 1, które opisuje wektor zależności $K = [0 \quad 0 \quad 1]^T$.



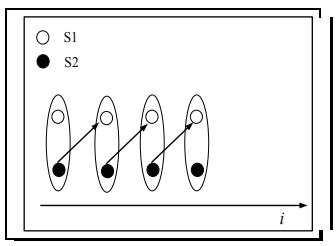
Rys. 2. Zależności w przestrzeni iteracji pętli
Fig. 2. Loop iteration space with dependencies

Pętla przedstawiona na rys. 3 ma zależności między iteracjami opisane przez wektor zależności $K = [1]^T$

```
for(i = 1; i ≤ 4; i++){
S1      ... = a[i];
S2      a[i+1] = ...;
}
```

Rys. 3. Przykładowa pętla
Fig. 3. Example loop

W pętli rys. 3 występują również zależności między instrukcjami $S1$ i $S2$. Na rysunku rys. 4 przedstawiono zależności między iteracjami i instrukcjami w poszczególnych iteracjach pętli.



Rys. 4. Zależności w przestrzeni iteracji pętli
Fig. 4. Loop iteration space with dependencies

3. Transformacje kodu pętli

Technikę znaną jako „hyperplane method", „wave-front-method" lub „Lamport method" przedstawiono w pracach [4, 5, 6]. Transformacje przeprowadzane podczas przekształcania pętli do postaci równoległej w metodzie hiperpłaszczyzn dzielimy na:

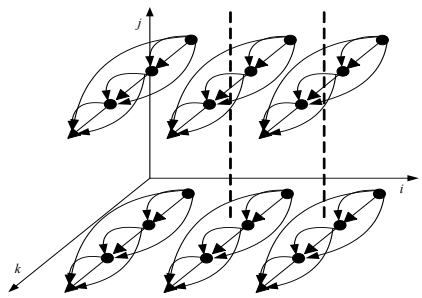
unimodularne:
- loop skewing
- loop interchange.
- loop reversal
- loop skewing

nieunimodularne:
- loop scaling

W przypadku wymienionych transformacji możemy dokonać zrównoleglenia jedynie pętli idealnie zagnieżdżonych, takich jak przykładowa pętla przedstawiona na rys. 1.

W pętli rys. 1. nie występują zależności między iteracjami najbardziej zewnętrznej pętli, które mogą być wykonane w sposób niezależny. Na rys. 5 przedstawiono zrównoleglony kod pętli, zapisany z użyciem standardu OpenMP.

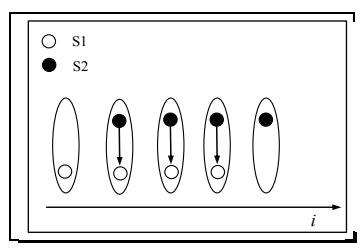
Przedstawione w pracach [12, 13, 14, 15, 16] transformacje pozwalają znaczne zwiększyć dziedzinę pętli nadających się do automatycznego zrównoleglenia. Zaliczamy do nich następujące transformacje:
- statement reordering
- loop distribution
- loop fusion
- loop alignment
- loop blocking (or tiling)
- loop interleaving
- index set splitting
- loop coalescing




```
#pragma omp parallel for private(j,k,sum) schedule(runtime)
    for(i=0;i<size;i++){
        for(j=0;j<size;j++){
            sum = 0;
            for(k=0;k<size;k++){
                sum+=b[i][k]*c[k][j];
            }
            a[i][j]+=sum;
        }
    }
```

Rys. 5. Zrównoleglone iteracje pętli zewnętrznej
Fig. 5. Parallel code outer loop parallelized

Pętla przedstawiona na rys. 3 nie może być poddana procesowi zrównoleglenia przez metodę hiperpłaszczyzn, gdyż nie jest zagnieżdżona. Po przeprowadzeniu transformacji przedstawionej na Rys. 6 zależności między iteracjami zostały usunięte. Iteracje pętli można wykonać w sposób równoległy.



Rys. 6. Zależności w przestrzeni iteracji pętli
Fig. 6. Loop iteration space with dependencies

Kod pętli po transformacji z rys. 6 przedstawiono na rys. 7, gdzie $S1(i)$ i $S2(i)$ przedstawiają wyrażenia $S1$ i $S2$ w i iteracji.

```
S1(1)     ... = a[1];
for(i = 1; i ≤ 3; i++){
S2(i)             a[i+1] = ...;
S1(i+1)           ... = a[i+1];
}
S2(4)   a[5] = ...;
```

Rys. 7. Przyspieszenie wykonania kodu pętli, kompilator ompi
Fig. 7. Loop speedup, ompi compiler

Niektóre transformacje jak "loop blocking" lub " loop interchange" poza umożliwieniem zrównoleglenia są wykorzystywane do poprawy efektywności kodu pętli.

4. Wyniki eksperymentu

W celu weryfikacji przydatności metod analizy zależności i transformacji pętli przeprowadzono eksperyment polegający na pomiarze przyspieszenia uzyskanego po zrównolegleniu pętli rys. 1. Do pomiaru wykorzystano komputer wyposażony w procesory 2 x Intel Quad Core Xeon E5310 1,60 GHz 1U. Pomiar przyspieszenia, rozumianego jako stosunek czasu wykonania kodu pętli Rys. 1 na jednym procesorze do czasy wykonania kodu pętli rys. 5 na P procesorach, wykonano dla różnych rozmiarów danych wejściowych.

Wraz ze wzrostem rozmiaru danych wejściowych przyspieszenie powinno rosnąć dążąc do P. Tymczasem dla danych o rozmiarze z przedziału 400 - 1000 przyspieszenie znaczne przekracza oczekiwaną wartość P, by dalej zachowywać się zgodnie z oczekiwaniami Rys. 10 (linia pari).

Zaskakująco wysokie przyspieszenie uzyskano dla wskazanego przedziału wielkości danych wejściowych, dzięki dobremu wykorzystaniu pamięci podręcznej.

Aby dobre wykorzystanie pamięci podręcznej miało miejsce w pełnym przedziale rozmiarów danych wejściowych dokonano dodatkowej transformacji (tiling) polegającej na podziale przestrzeni iteracji na mniejsze paczki.

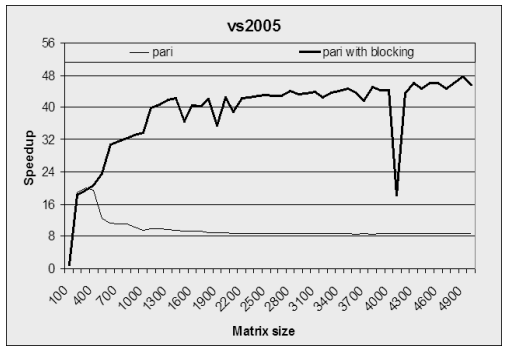
```
intgetIterationBlock
(intiterator,int*start,int*end,intloopInit,intloopEnd,intblockSize)
    *start=loopInit+iterator*bloockSize
    *end=*start+blockSize-1,
    *end=(((*end)>(loopEnd))?(loopEnd):(*end))
    return*start<=*end,
}
```

Rys. 8. Kod funkcji dzielącej przestrzeń iteracji pętli
Fig. 8. Function for dividing the loop iteration space

Na rys. 9 przedstawiono zrównoleglony kod pętli z podziałem przestrzeni iteracji na podprzestrzenie. Na rys. 8 przedstawiono funkcję dzieląca przestrzeń iteracji.

```
#pragma omp parallel private(k, j, i, ii, is, ie, jj, js, je, kk, ks, ke) shared(a, b, c)
 {
   for(ii = 0; getIterationBlock(ii,&is,&ie,0, size - 1, blockSize); ii + +){
     for(jj = 0; getIterationBlock(jj,&js,&je,0, size - 1, blockSize); jj + +){
       for(kk = 0; getIterationBlock(kk,&ks,&ke,0, size - 1, blockSize); kk + +){
#pragma omp for schedule (runtime)
         for(i = is; i <= ie; i + +){
           for(j = js; j <= je; j + +){
             for(k = ks; k <= ke; k + +){
               a[i][j] + = b[i][k] * c[k][j];
} } } } } } }
```

Rys. 9. Zrównoleglenie pętli zewnętrznej z podziałem przestrzeni iteracji
Fig. 9. Outer loop parallelised together with blocking (pari with blocking)



Rys. 10. Przyspieszenie wykonania kodu pętli, kompilator ompi
Fig. 10. Loop speedup, ompi compiler

Po wprowadzeniu transformacji wykorzystanie pamięci podręcznej poprawiło się dla całego przedziału rozmiarów danych wejściowych.. Na rys. 10 (linia pari with blocking) widać, że uzyskiwane przyspieszenie znacznie przekracza oczekiwaną wartość P.

5. Wnioski

Przedstawione powyżej (oraz inne uzyskane w trakcie badań) wyniki świadczą o możliwościach pozwalających na ulepszanie kodu przeznaczonego dla urządzeń i systemów osadzonych. Algorytmy wyboru transformacji potrzebują jako danych wejściowych różnego rodzaju opisu zależności. Celem części opisywanego w artykule projektu, która jest poświęcona analizie zależności jest dostarczenie różnorodnych opisów kolekcji zależności. Część algorytmów jako opisu zależności wymaga jedynie samego wektora zależności, w innych konieczne jest użycie relacji, a w jeszcze innych można używać wielościanów wypukłych. Przedstawione w artykule wyniki są używane przez dalsze części kompilatora optymalizującego, np. jako dane wejściowe do algorytmu slicingu. Dodatkowo, między innymi w związku ze zmniejszeniem ilości odwołań do pamięci cache układy osadzone mogą zużywać mniej energii na wykonywanie swoich funkcji

6. Literatura

[1] Allen, R, Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, 2001.
[2] Banerjee U.: Loop Transformations for Restructuring Compilers. Kluwer Academic, 1993.
[3] Wolfe M.: High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company,1995.
[4] Darte A., Robert Y., Vivien F.: Scheduling and Automatic Parallelization. Birkhäuser Boston, 2000.
[5] Lamport L.: The Parallel Execution of DO Loops. Communications of the ACM, Vol. 17, No.2, Feb. 1974, pp. 83-93.
[6] Wolf M. E. and Lam M. S.: A data locality optimizing algorithm. In Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, pages 30–44, June 1991.
[7] The Omega Project [online] http://www.cs.umd.edu/projects/omega/ [dostęp: 2009]
[8] PIPS: Automatic Parallelizer [online] http://www.cri.ensmp.fr/~pips/ [dostęp: 2009]
[9] Poliwoda M.: Automatyczne zrównoleglenie pętli programowych, implementacja metody hiperpłaszczyzn. Metody i narzędzia wytwarzania oprogramowania, 14-16 maja, Szklarska Poręba, Polska 2007.
[10] Schrijver: Theory of Linear and Integer Programming. Wiley, Chichester, 1986.
[11] Wei Li and Keshav Pingali: A singular loop transformation framework based on non-singular matrices. In 5th Workshop on Languages and Compilers for Parallel Computing, pages 249-260, Yale University, August 1992.
[12] Allen R., Callahan D. and Kennedy K.: Automatic decomposition of scientific programs for parallel execution. In Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages, pages 63-76, January 1987.
[13] Allen J. R. and Kennedy K.: Automatic translation of Fortran programs to vector form. ACM Transactions on Programming Languages and Systems, 9(4):491-542, October 1987.
[14] Vivek Sarkar and Radhika Thekkath: A general framework for iteration-reordering loop transformations. In ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, pages 175-187, San Francisco, California, Jun 1992.
[15] Steve Carr and Ken Kennedy: Compiler blockability of numerical algorithms. In Proceedings Supercomput-ing'92, pages 114-125, Minneapolis, Minnesota, Nov 1992.
[16] Polychronopoulos C.: Parallel Programming and Compilers. Kluwer Academic Publishers, 1988.

otrzymano / received: 09.05.2010
przyjęto do druku / accepted: 04.06.2010 *artykuł recenzowany*