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

Reprezentacja przestrzeni iteracji pętli i opisu zależności w kompilatorze zrównoleglającym

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 artykule został przedstawiony proces optymalizacji kodu programów przeznaczonych do wykonania przez systemy osadzone lub wieloprocesorowe przy pomocy kompilatora optymalizującego. Głównym elementem, dla którego wykonywane są optymalizacje, są pętle, ponieważ to w nich wykonywane jest najwięcej instrukcji a ich optymalizacja ma wpływa na wykorzystanie dostępnych zasobów, co wpływa znacząco na ilość zużywanej przez układy energii. Zaprezentowana została również budowa kompilatora i istotne informacje przechowywane i wykorzystywane do analizy w trakcie kompilacji.

Słowa kluczowe: kompilator, systemy osadzone, analiza zależności.

Representation of iteration domain and description of dependencies in a parallelizing compiler

Abstract

A process of optimization of the program code to be performed by embedded or multiprocessor systems is very complex and time consuming. The code fragments whose optimization brings the greatest effects are loops because they process a large amount of data using the same instructions repeatedly. This makes the execution time of a loop a significant part of the overall program execution time. The time of each iteration affects use of available resources (memory, cache, etc.), which also influences the amount of energy. The program code is subjected to numerous transformations carried out on the basis of analysis of the relationships. This analysis allows determining which pieces of the code can be executed independently of each other. The aim of this paper is to present conversion mechanisms implemented in a paralellizing compiler. These mechanisms are focused on the representation format and description of the loop dependencies. The analysis of dependences allows the submission of many transitions and provides a more optimized version of the code when taking into account such factors as the energy aware code or the execution time.

Keywords: compiler, embedded system, dependency analysis.

1. Wstęp

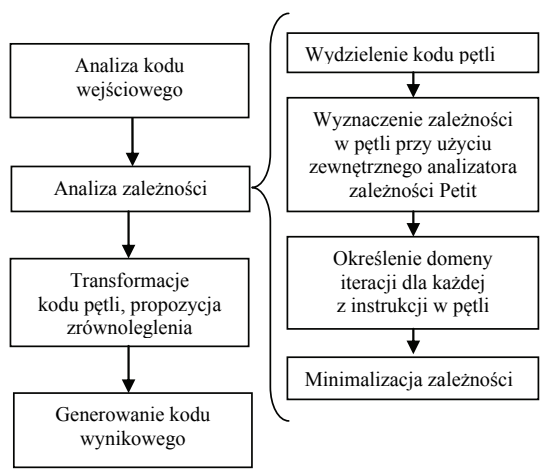
Proces optymalizacji kodu programów przeznaczonych do wykonania przez systemy osadzone lub wieloprocesorowe jest bardzo złożony i czasochłonny. Fragmentami kodu, których optymalizacja przynosi największe efekty są pętle, gdyż w pętlach przetwarza się duże ilości danych wykonując wielokrotnie te same instrukcje, co powoduje, że czas wykonania kodu pętli stanowi znaczną część czasu wykonania całego programu, zaś sposób wykonania poszczególnych iteracji ma wpływ na wykorzystanie dostępnych zasobów (pamięci podręcznej), co ma również wpływ na ilość zużywanej energii.

Kod programu jest poddawany licznym transformacjom przeprowadzanym na podstawie analizy zależności pozwalającej między innymi na określenie, które fragmenty kodu mogą zostać wykonane niezależnie od siebie, a które w celu zapewnienia deterministyczności muszą być wykonywane sekwencyjnie. Implementacje algorytmów realizujące transformacje pętli ze względu na złożoność wykorzystują różne biblioteki umożliwiające wykonywanie operacji na zbiorach (iteracji), a co za tym idzie różne formaty reprezentacji przestrzeni iteracji pętli i opisu zależności. Część algorytmów jako opisu zależności wymaga jedynie samego wektora zależności, w innych konieczne jest użycie relacji, a w jeszcze innych są używane wielościany i domeny iteracji.

Celem artykułu jest przedstawienie zaimplementowanych w kompilatorze zrównoleglającym mechanizmów zarządzania formatami reprezentacji przestrzeni iteracji pętli i opisu zależności, co umożliwia złożenie wielu transformacji i dostarczenie kilku wersji kodu zoptymalizowanego pod różnymi katami np. zużycia energii lub czasu wykonania.

2. Proces zrównoleglenia kodu

Proces zrównoleglenia kodu składa się z następujących po sobie faz (w kolejności jak na rys. 1): analizy kodu wejściowego, analizy zależności, transformacji, generowania kodu wynikowego.



Rys. 1. Schemat budowy kompilatora
Fig. 1. Schematic diagram of the compiler

Analiza kodu wejściowego - Analizie jest poddawany zapisany w pliku kod w języku C. We wstępnej fazie analizy kod wejściowy jest zapisywany w wewnętrznej strukturze w postaci drzewa parsowania. Następnie w drzewie parsowania wyszukiwane są pętle for, podlegające zrównolegleniu, dla których przeprowadzana jest analiza zależności.

Analiza zależności - Analiza zależności dla kodu pętli jest przeprowadzana przez analizatory zewnętrzne. Wyniki analizy zależności są zapisywane w drzewie parsowania dla każdej pętli, natomiast opis przestrzeni iteracji jest zapisywany dla każdej instrukcji w pętli.

Transformacje - Na podstawie informacji o opisie przestrzeni iteracji pętli i zależnościach dokonywane są transformacje. Wykonywane transformacje mogą zmieniać opis przestrzeni iteracji, natomiast zależności muszą być zachowane w transformowanej przestrzeni iteracji.

Generowanie równoległego kodu wynikowego - Drzewo parsowania zawierające fragmenty przekształcone w wyniku transformacji jest zapisywane w postaci kodu w języku C.

3. Analizatory zależności

Kompilator zrównoleglający do analizy zależności wykorzystuje analizatory zewnętrzne Petit [4] oraz Clan z Candl [1], które zostały zaimplementowane w różnych projektach i na potrzeby różnych algorytmów wyszukiwania równoległości. Główną zaletą implementowanego kompilatora zrównoleglającego jest możliwość wykorzystania algorytmów opracowanych w obu projektach. Możliwość ta została osiągnięta poprzez uspójnienie informacji dostarczanych przez oba analizatory i opracowanie konwersji danych między strukturami obu projektów.

Tab. 1. Analizatory zależności
Tab. 1. Dependency analyzers

	Analizator		
Informacja	Clan	Candl	Petit
Język opisu pętli	C/Java	C	petit
Domena iteracji	TAK		
Funkcja rozpraszająca (ang. scattering function)	TAK		
Informacje o odczycie i zapisie	TAK		
Treść wyrażeń	TAK		
Opis zależności		TAK	TAK
Zależności od-do		TAK	TAK
Domena zależności		TAK	TAK

Analizatory Clan i Candl są wykorzystywane w projekcie Pluto [7]. Analizator Clan [2] dostarcza reprezentacji kodu w postaci wielościanów. Na podstawie reprezentacji kodu w postaci wielościanu analizator Candl dostarcza informacji o zależnościach. Głównym ograniczeniem tego rozwiązania są ograniczenia gramatyki zastosowanej w analizatorze Clan.

Analizator Petit jest wykorzystywany w projekcie Omega [4]. Analizator dostarcza opisu zależności w postaci relacji zbiorów opisujących zależne iteracje. Głównym ograniczeniem analizatora Petit jest zastosowanie gramatyki języka Petit. W wersji analizatora wykorzystywanej w kompilatorze zrównoleglającym zmieniono gramatykę języka Petit na gramatykę języka C.

W tabeli 1 przedstawiono zestawienie analizatorów i dostarczanych przez nie informacji. Podstawowa różnica, rodzaj analizowanego języka, została wyeliminowana przez zastąpienie gramatyki języka Petit gramatyka języka C. Analizatory zależności Candl i Petit dostarczają identycznych informacji o zależnościach w różnych formatach danych. Rolę analizatora Clan przejął analizator kodu wejściowego kompilatora zrównoleglającego, którego dodatkowym zadaniem jest stworzenie opisu przestrzeni iteracji pętli (Iteration domain) w formacie zbiorów Omega [4]. Pozostałe informacje są dostępne bezpośrednio z drzewa parsowania.

Po przeprowadzonej analizie kodu wszystkie informacje w postaci opisów przestrzeni iteracji pętli i opisów zależności są dostępne w formacie zbiorów i relacji Omega [4] oraz zapisów w drzewie parsowania. W celu umożliwienia wykorzystania algorytmów bazujących na formacie danych w postaci wielościanów dokonywana jest konwersja, której mechanizm jest oparty o gotowe rozwiązania bazujące na projektach Omega [4], Isl [6], PolyLib [9].

4. Drzewo parsowania

Sposób przechowywania informacji o opisach przestrzeni iteracji pętli i zależnościach dla pętli nieidealnie zagnieżdżonej o wymiarze 3x3 (rys. 2) przedstawiono w postaci schematu drzewa parsowania (rys. 3).

```
          for(i=1;i<=10;i++)
          {
(S1)              b[i] = 0;
                  for(j=1;j<=10;j++)
                  {
(S2)                      a[i][j]=a[i+5][j];
                          if(j>=5)
                          {
(S3)                              c[j]=0;
                          }
                  }
          }
```

Rys. 2. Przykładowa pętla
Fig. 2. Example loop

```
|ParseTree1 (1)      (for(i=1;i<=10;i++))
|                \-Opis zależności w pętli dla S1, S2 i S3
|
|--ParseTree1.ParseTreeA (S1) (b[i] = 0;)
|                \-Domena iteracji dla S1
|
|-- ParseTree1.ParseTreeB (2)   (for(j=1;j<=10;j++))
\
 \---…ParseTreeB.ParseTreeA (S2) (a[i][j]=a[i+5][j];)
  |            \-Domena iteracji dla S2
  |
  |---…ParseTreeB.ParseTreeB (3)  (if(j>=5))
   \
    \--- ParseTreeB. ParseTreeA (S3) (c[j]=0;)
         \-Domena iteracji dla S3
```

Rys. 3. Reprezentacja przykładowej pętli
Fig. 3. Example loop represantation

Zależności między iteracjami związane z instrukcjami S2 i S3 są zapisane w węźle pętli drzewa parsowania.

Każda z trzech instrukcji jest wykonywana w innej przestrzeni iteracji:

instrukcja S1 dla {1<=i <=10 }

instrukcja S2 dla {1<=i <=10 && 1 <=j<=10}

instrukcja S3 dla {1<=i <=10 && 5 <=j<=10}

Opis przestrzeni iteracji dla każdej z instrukcji jest przechowywany w jej węźle.

5. Rozwiązania alternatywne

Do najbardziej znanych projektów wykorzystujących do analizy i transformacji kodu opisy w formatach zbiorów i relacji oraz wielościanów, zaliczamy projekty "integer set analysis framework" Isa [8] i "An automatic parallelizer and locality optimizer for multicores" PLUTO [7] oraz już nie rozwijany "The Omega Project" Omega [4]. W projektach Isa [8] i PLUTO [7] również wykorzystywane są zewnętrzne parsery kodu wejściowego i analizatory zależności. W przeciwieństwie do opisywanego w artykule rozwiązania we wspomnianych projektach przetwarzane i analizowane są tylko określone fragmenty kodu. Brak dostępu do informacji o całym kodzie w znacznym stopniu utrudnia przeprowadzenie części transformacji optymalizujących. Przykładem może być brak dostępu do informacji o typie tablic co utrudnia przeprowadzenie optymalizacji zwiększającej lokalność [3]. Proponowany w artykule kompilator zrównoleglający przechowuje informacje o całym kodzie w drzewie parsowania a wykorzystując analizatory zewnętrzne uzupełnia informacje o opis przestrzeni iteracji i występujące zależności, które są przechowywane w postaci zbiorów i relacji oraz wielościanów. Zaproponowana struktura umożliwia precyzyjną optymalizacje kodu w celu wpływu na sposób wykonania poszczególnych iteracji co ma wpływ na wykorzystanie dostępnych zasobów pamięci podręcznej, procesora a co za tym idzie również na ilość zużywanej energii.

6. Wyniki i wnioski

Do weryfikacji badań użyto źródeł ANSI-C z zestawu testów UTDSP w wersji 97.02.12 przeznaczonych do oceny jakości kodu cyfrowej obróbki sygnałów stosowanego w programowalnych procesorach [11]. Weryfikacji poddano również testy ze zbioru NAS[10] i Collective Benchmark (cBench) [12]. Dla zobrazowania działania kompilatora optymalizującego posłużono się testem z zestawu cBench[12]:

```
for (i = 1; i <= alphaSize; i++)
{
        j = weight[i] >> 8; j = 1 + (j / 2);weight[i] = j << 8;
}
real 0m31.143s   user 0m30.422s   sys 0m0.648s
```

Rys. 4. Fragment kodu przed tranformacją
Fig. 4. Code fragment before transformation

W przykładzie tym kompilator wykrył zależności pomiędzy instrukcjami pętli, nie ma natomiast zależności pomiędzy iteracjami. W związku z powyższym możliwe jest na przykład zastosowanie transformacji loop-unroll (rozwinięcie pętli), której wyniki przedstawiono na rys. 5. Osiągnięte rezultaty (architektura 2-rdzeniowa Intel E8400) przyśpieszenia (średnio 10%) są zadowalające w przypadku programów sekwencyjnych. Podstawowym celem prac jest jednak osiągnięcie zrównoleglenia programów sekwencyjnych poprzez wykrycie i zminimalizowanie zależności a następnie wybór odpowiednich transformacji. Dla prostych przykładów, jak ten z rysunków 4 i 5, możliwe jest osiągnięcie przyśpieszenia na poziomie wyznaczonym ograniczeniem wynikającym z prawa Amdahla. Dalsze prace nad opisywanym kompilatorem będą miały na celu zwiększanie liczby transformacji, umożliwienie analizy kodu zawierającego wskaźniki oraz umożliwienie doboru parametrów transformacji poprzez kompilacje iteracyjną.

```
for (i = 1; i + 4 <= alphaSize; i+=4)
{
  j = weight[i] >> 8; j = 1 + (j / 2); weight[i] = j << 8;
  j = weight[i+1] >> 8; j = 1 + (j / 2); weight[i+1] = j << 8;
  j = weight[i+2] >> 8; j = 1 + (j / 2); weight[i+2] = j << 8;
  j = weight[i+3] >> 8; j = 1 + (j / 2); weight[i+3] = j << 8;
}
for ( ; i <= alphaSize; i++) {
  j = weight[i] >> 8; j = 1 + (j / 2);
  weight[i] = j << 8;
}
real 0m29.050s   user 0m28.526s   sys 0m0.416s
```

Rys. 5. Fragment kodu po tranformacji
Fig. 5. Code fragment after transformation

7. Literatura

[1] Bastoul C.: Generating loops for scanning polyhedral, PRiSM, Versailles University, 2002/23.

[2] Bastoul C., Cohen, Albert and Girbal, Sylvain and Sharma, Saurabh and Temam, Olivier: Putting Polyhedral Loop Transformations to Work, Workshop on Languages and Compilers for Parallel Computing, 2003, LNCS, Springer-Verlag, College Station, Texas.

[3] Bielecki W., Kraska K.: Zwiększenie wydajności aplikacji wykonywanych w systemach osadzonych poprzez zwiększenie lokalności danych, Reprogramowalne Układy Cyfrowe, Szczecin 2007.

[4] The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs [online] http://www.cs.umd.edu/projects/omega/.

[5] The CLooG Code Generator in the Polyhedral Model's Home [online] http://www.cloog.org/.

[6] library for manipulating sets and relations of integer points [online] http://freshmeat.net/projects/isl.

[7] An automatic parallelizer and locality optimizer for multicores [online] http://pluto-compiler.sourceforge.net.

[8] integer set analysis framework [online] http://repo.or.cz/w/isa.git.

[9] A library of polyhedral functions [online] http://icps.u-strasbg.fr/polylib/.

[10] NASA Advanced Supercomputing Parallel Benchmarks Version 3.2 [online] http://www.nas.nasa.gov/Software/NPB/.

[11] Lee C.: UTDSP (University of Toronto Digital Signal Processing) Benchmark Suite [online] http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

[12] Collective Benchmark (cBench, dawniej MilepostGCC) http://ctuning.org/wiki/index.php/CTools:CBench.

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