Włodzimierz BIELECKI, **Krzysztof SIEDLECKI**
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY, WYDZIAŁ INFORMATYKI
ul. Żołnierska 52, 71-210 Szczecin

# Wpływ redukcji zależności w pętlach programowych na zużycie zasobów w systemach wbudowanych

**Prof. dr hab. inż. Włodzimierz BIELECKI**

Jest kierownikiem Katedry Inżynierii Oprogramowania na Zachodniopomorskim Uniwersytecie Technologicznym. W badaniach koncertuje się na przetwarzaniu równoległym i rozproszonym, kompilatorów optymalizujących, ekstrakcji grubo- i drobnoziarnistej równoległości zawartej w pętlach programowych.

*e-mail: wbielecki@wi.zut.edu.pl*

**Dr inż. Krzysztof SIEDLECKI**

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, inżynieria oprogramowania.

*e-mail: ksiedlecki@wi.zut.edu.pl*

**Streszczenie**

Zaprezentowano wpływ redukcji zależności na zużycie zasobów dla pętli programowych zapisanych w języku ANSI-C. Do redukcji zależności wykorzystane zostały popularne techniki (redukcja zmiennych skalarnych, indukcja zmiennych, przekoszenie pętli, podział i łączenie pętli oraz rozszerzenie zmiennych skalarnych) jak również nowe pozwalające na redukcję zależności bez konieczności modyfikacji kodu pętli. Omówiono zużycie zasobów pamięciowych w zależności od zastosowanej transformacji. Dla przykładowych pętli przedstawiono analizę zużycia zasobów w kontekście systemów osadzonych.

**Słowa kluczowe**: redukcja zależności, transformacje pętli programowych, systemy osadzone.

## Impact of dependence reduction in programming loops on resource usage in embedded systems

**Abstract**

The influence of dependence removal techniques on computer resource utilization for program loops is investigated. Source loops are represented in the C language. Removing redundant dependence relations allows reducing time complexity of algorithms whose input is a set of dependence relations while output is a parallel program. In addition, removing dependences leads to reducing computer resource utilization. Well-known techniques and those proposed by the authors are examined. The following well-known techniques are investigated: scalar reduction, induction variable elimination, loop skewing, loop splitting, loop fissioning, and scalar expansion. All techniques are illustrated by means of examples. Additional techniques being examined are removing dependence relations describing the same dependences as well as removing dependence relations representing linear dependent distance vectors. For a chosen example, for each technique under examination, its effectiveness is presented and the effect of computer resource utilization is shown.

**Keywords**: redundant dependence removing, program loop transformation, embedded systems.

## 1. Wstęp

Systemy osadzone w ostatnim czasie stały się jednymi z najbardziej rozpowszechnionych systemów używanych w produktach konsumenckich, przemyśle jak i nauce. Współczesne systemy osadzone z jednej strony mogą pracować jako oddzielne jednostki odpowiedzialne za sterowanie pojedynczych urządzeń, z drugiej mogą zostać połączone w układ kilku a nawet kilku tysięcy mikrokontrolerów połączonych ze sobą przy pomocy sieci [1].

Tworzenie oprogramowania dla systemów osadzonych związane jest z użyciem dedykowanych narzędzi do konkretnego mikrokontrolera np. kompilator, debugger [2]. Obecnie producenci na rynku oferują wydajne wielordzeniowe mikroprocesory pozwalające na wykonywanie kodu w sposób współbieżny (ARM Cortex, Xilinx Virtex, Intel Atom).

Transformacja kodu sekwencyjnego do jego odpowiednika równoległego wiąże się z koniecznością przeprowadzenia analizy zależności. Wykryte zależności muszą być honorowane w kodzie wynikowym. W celu wyznaczenia jak największej równoległości w wielu przypadkach przed dokonaniem transformacji niezbędne jest zredukowanie zależności występujących w kodzie źródłowym. Zastosowanie redukcji zależności bardzo często wiąże się z koniecznością użycia większych zasobów pamięciowych systemu w celu zapewnienia poprawnego wykonania kodu. Miniaturyzacja systemów wbudowanych pociąga za sobą ograniczenia odnośnie ilości użytych zasobów i ewentualnych możliwości ich rozbudowy.

W artykule przedstawiono popularne techniki redukcji zależności oraz autorskie rozwiązania. Techniki zostały przeanalizowane pod kątem skuteczności oraz zużycia zasobów pamięciowych.

## 2. Reprezentacja zależności przy pomocy relacji

Pojedyncze wykonanie instrukcji s pętli dla konkretnej iteracji I określamy instancją instrukcji s(I).

Dwie instancje instrukcji s1(I) i s2(J) są zależne jeśli obie odwołują się do tej samej komórki pamięci i przynajmniej jedno z tych odwołań jest zapisem. Instancja instrukcji s1(I) nazywana jest początkiem zależności, natomiast s2(J) końcem z zapewnieniem że s1(I) jest leksykograficznie mniejsze od s2(J) ($s1(I) < s2(J)$).

Do opisu zależności wybrana została analiza zależności zaproponowana przez Pough'a oraz Wonnacott'a [8] gdzie zależności reprezentowane są przez relacje zależności. Ograniczenia relacji opisane są przy pomocy arytmetyki Presburgera zbudowanych z równań i nierówności afinicznych.

Relacja zależności opisuje mapowanie między dwoma krotkami gdzie wejście i wyjście to lista zmiennych i/lub wyrażeń używanych do reprezentacji iteracji wejściowej i wyjściowej, ograniczenia są formułami Presburgera nałożonymi na krotki wejściowe oraz wyjściowe.

## 3. Sposoby redukcji zależności

Proces wyznaczania zależności danych odbywa się poprzez analizę zmiennych programowych. Na podstawie przepływu danych do analizy brane są pod uwagę trzy rodzaje zależności (patrz tabela 2): prosta (ang. *flow*), odwrotna (ang. *anti*) oraz po wyjściu (ang. *output*). Zależność prosta występuje, gdy w instancji instrukcji I następuje zapis, natomiast w instancji instrukcji J następuje odczyt wartości zapisanej w I, przy czym $I \prec J$. Z zależnością odwrotną mamy do czynienia, gdy instancja instrukcji I odczytuje dane, a instancja instrukcji J zapisuje do tej samej komórki pamięci ($I \prec J$). W przypadku zależności po wyjściu instancje instrukcji I i J zapisują do tej samej komórki pamięci.

Redukcja zależności może odbywać się poprzez techniki modyfikujące kod wejściowy lub też wyeliminowanie nadmiarowych zależności bez konieczności transformacji kodu wejściowego. W pierwszej grupie odpowiednie transformacje ukierunkowane są na redukcję konkretnych typów zależności. Druga grupa transformacji redukuje zależności bez względu na ich typ.

W celu wyeliminowania zależności odwrotnej należy wykonać kopię modyfikowanej zmiennej w instancji S2, a w instrukcji S1 odczytać wartość z utworzonej kopi. Przykład transformacji został przedstawiony w tabeli 1 (pierwszy wiersz). W wynikowym kodzie została utworzona zmienna b1[] będąca kopią zmiennej b[]. Iteracje każdej z pętli mogą zostać wykonane równolegle, jednak wykonanie pętli względem siebie musi być sekwencyjne.

Tab. 1. Przykład redukcji zależności odwrotnej i po wyjściu
Tab. 1. Example of reducing anti and output dependency

| | |
|---|---|
| for (i=1; i<M; i++)<br>  b[i] = b[i+1]; | for(i=1; i<M; i++)<br>  b1[i] = b[i+1];<br>for(i=1; i<M; i++)<br>  b[i] = b1[i]; |
| for(i=1; i<10; i++)<br>{<br>  x = b[i+1] /2<br>  a[1] = x * 2 }<br>z = x + a[1] | #pragma omp for lastprivate(x,c)<br>for(i=1; i<10; i++) {<br>  x = b[i+1] /2<br>  c = x * 2 }<br>a[1] = c;<br>z = x + a[1]; |
| | |
| oryginalny kod | kod zmodyfikowany |

Zależność po wyjściu może zostać wyeliminowany poprzez stworzenie nowej zmiennej przechowującej wartość ostatniej instancji z zależności. Przykład transformacji został przedstawiony w tabeli 1 (drugi wiersz). W kodzie wynikowym utworzono zmienną pomocniczą, natomiast zapamiętanie ostatniej wartości uzyskano poprzez użycie modyfikatora lastprivate(c) zgodnie ze standardem OpenMP [5].

Zależność prosta to jedyna zależność, która w ogólnym przypadku nie może zostać wyeliminowana. Istnieje wiele technik modyfikujących kod do postaci, w której możliwe jest uzyskanie równoległości przy jednoczesnym respektowaniu tego typu zależności. Należą do nich: redukcja zmiennych skalarnych (ang. *scalar reduction*), eliminacja zmiennych indukcyjnych (ang. *induction variable elimination*), przekoszenie pętli (ang. *loop skewing*), podział pętli (ang. *loop splitting*), łączenie pętli (ang. *loop fissioning*) oraz rozszerzenie zmiennych skalarnych (ang. *scalar expansion*) [3]. Ze względu na ograniczenia objętościowe w artykule przedstawiono jedynie wybrane transformacje.

Technika eliminacji zmiennych indukcyjnych polega na wyznaczeniu zmiennych, których wartość aktualizowana jest poprzez iloczyn ze stałą, inkrementację o stałą lub inkrementację zmienną indeksową. W takim przypadku możliwe jest zastąpienie odwołań do zmiennej skalarnej poprzez odpowiednie wyrażenie na zmiennych indeksowych pętli. Przykład zastosowania techniki został przedstawiony w tabeli 2.

Tab. 2. Przykład zastosowania indukcji zmiennych
Tab. 2. Example of induction variable elimination

| | |
|---|---|
| idx = N/2 + 1<br>i_sum = 1<br>pow2 = 2<br>for(i=1; i<=N2; i++) {<br>  a[i] = a[i] + a[idx];<br>  b[i] = i_sum;<br>  c[i] = pow2;<br>  idx = idx + 1;<br>  i_sum = i_sum + i;<br>  pow2 = pow2 * 2 } | #par for<br>for(i=1; i<=N/2; i++) {<br>  a[i] = a[i] + a[i + N/2];<br>  b[i] = i * (i + 1) / 2;<br>  c[i] = 2 * i;<br>} |
| oryginalny kod | kod zmodyfikowany |

Kolejna technika pozwalająca na zrównoleglenie kodu z zależnościami prostymi jest przekoszenie pętli. Transformacji poddawana jest pętla, w której występują zależności przenoszone pętlą, w której występują zależności wewnątrz jednej iteracji. Przykład pętli zawierającej zależności proste przekoszone pętlą przedstawiony został w tabeli 3.

Tab. 3. Przykład zastosowania przekoszenia pętli
Tab. 3. Example of using loop skewing

| | |
|---|---|
| for(i=2; i<=N; i++) {<br>  b[i] = b[i] + a[i-1];<br>  a[i] = a[i] + c[i];<br>} | b[2] = b[2] + a[1];<br>#pragma omp parallel fo shared(a,b,c)<br>for(i=2; i<=N-1; i++) {<br>  a[i] = a[i] + c[i];<br>  b[i+1] = b[i+1] + a[i];<br>}<br>a[N] = a[N] + c[N]; |
| oryginalny kod | kod zmodyfikowany |

Zastosowanie wymienionych wyżej transformacji wiąże się ze zużyciem dodatkowych zasobów w celu zapewnienia poprawnego wykonania kodu. Ma to zasadnicze znaczenia w systemach, w których dodawanie nowych zasobów jest utrudnione, nieopłacalne lub wręcz niemożliwe np. systemy wbudowane.

Redukcja zależności może odbywać się również bez konieczności modyfikacji kodu pętli. Zaproponowano dwie metody: redukcja nadmiarowych zależności oraz redukcja relacji zależności o liniowo zależnych wektorach zależności. Algorytmy zostały opisane w [6, 7], ze względu na ograniczenia objętościowe nie zostały zawarte w publikacji.

Algorytm redukcji nadmiarowych zależności bazuje na założeniu, iż jedna z dwóch relacji opisujących zależności między dwoma tymi samymi instrukcjami może zostać usunięta bez utraty informacji o zależności. W takim przypadku redukcja zależności odbywa się bez konieczności transformacji kodu wynikowego, dzięki czemu nie ma dodatkowych narzutów na zasoby pamięciowe. Algorytm redukcji zależności o liniowo zależnych wektorach dystansu bazuje na założeniu, że jeśli dwie relacje opisują zależności dla tych samych instancji instrukcji oraz wektor zależności jednej z relacji może zostać przedstawiony jako krotność drugiej, to relacja o większym wektorze zależności może zostać usunięta. Informacje o zależnościach zostają zachowane gdyż zgodnie z pozostawiona relacją możemy osiągnąć każdą instancję instrukcji ze zredukowanej relacji.

## 4. Redukcja zależności dla wybranego przykładu

Jako przykład wybrana została jedna z bardziej skomplikowanych pętli z benchmarku NPB (ang. *NAS Parallel Benchmark*). Ze względu na ograniczony rozmiar artykułu ciało pętli zostało przedstawione jedynie w wybranych fragmentach.

Dla wybranej pętli, nie stosując żadnych technik redukcji zależności, przy pomocy narzędzia Petit wyznaczono 3228 wszystkich relacji zależności – tabela 4. Liczba relacji zależności w odniesieniu do ich rodzaju przedstawia się następująco: 1506 rel. zależności proste, 511 rel. zależności po wyjściu i 1211 rel. zależności odwrotne. Ograniczając zależności do zależności bazujących na wartościach danych (*ang. value-based dependence*) liczba wszystkich zależności spada do 279.

Zastosowanie złączenia pętli oraz zwinięcia instrukcji w pętle pozwoliło na znaczne zmniejszenie liczby relacji zależności. Zastosowanie transformacji wymaga rezerwacji niewielkich zasobów pamięciowych jedynie na zmienne indeksowe dla pętli przebierających kolejne iteracje. Do zastosowania transformacji niezbędne było zadeklarowanie nowej zmiennej skalarnej m1. Zastosowanie powyższej transformacji pozwoliło na zredukowanie 875 relacji prostych, 108 relacji po wyjściu, 1024 relacji odwrotnych (tabela 4, wiersz 3).

Tab. 4. Liczba zależności w zależności od zastosowanej transformacji
Tab. 4. Numer of dependences after using transfomation

| Transformacja | Wszystkie zależności | | | Value based | Redukcja |
|---|---|---|---|---|---|
| | proste | po wyjściu | odwrotne | | |
| brak | 1506 | 511 | 1211 | 279 | 2483 |
| prywatyzacja zmiennych skal. | 663 | 449 | 314 | 134 | 919 |
| złączenie pętli + zwinięcie instr. w pętle | 631 | 402 | 187 | 171 | 791 |
| zwiększenie wym. zm. tablicowych | 1137 | 842 | 270 | 279 | 1909 |
| zwiększenie wym. zm. tab. + dodanie nowych tab. | 1082 | 813 | 231 | 279 | 1858 |
| wszystkie możliwe transf. | 54 | 5 | 10 | 49 | 69 |

Kolejną wykorzystaną transformacją było stworzenie dodatkowej tablicy zmiennych poprzez zwiększenie już istniejących wymiarów zmiennych tablicowych. Zwiększenie wymiaru zmiennej tablicowej tmat() spowodowało, iż każde odwołanie do zmiennej tmat() dla dowolnej iteracji w zewnętrznej pętli odwołuje się do unikalnej komórki pamięci. Transformacja ta pozwoliła na zredukowanie o 369 i 941 relacji zależności prostych i odwrotnych. Koszt takiej transformacji może być znaczący. W tym przypadku zwiększenie wymiaru tablicy wymaga rezerwacji 5*5 tablic o wymiarze N1. Przyjmując, że tablica tmat() przechowuje zmienne typu double o wielkości 8 bajtów oraz N1 = 100, stworzenie dodatkowego wymiaru dla tablic wymaga 5*5*100 = 2500 bajtów. Jest to już znacząca wartość w szczególności dla systemów wbudowanych o ograniczonych możliwościach zwiększania zasobów.

Ostatnia z przeprowadzonych modyfikacji dotyczyła dodania nowych zmiennych tablicowych - do wcześniejszej transformacji - służących jako zmienne tymczasowe do przechowywania wartości dla kolejnych instrukcji. Zysk z transformacji został przedstawiony w tabeli 4 (przedostatni wiersz). Udało się zredukować 424 relacji zależności prostych oraz 980 relacji zależności odwrotnych. Koszt takiej transformacji jest jeszcze większy od poprzedniej, gdyż do już zwiększonych wymiarów zmiennych tablicowych dokładamy nowe zmienne tablicowe służące jedynie jako bufory.

Po zastosowaniu popularnych transformacji do pozostałych relacji zastosowano bezstratną redukcję zależności. Ostatnia kolumna tabeli 4 przedstawia uzyskane wyniki. Jak widać w większości przypadków udało się zwiększyć stopień redukcji zależności bez konieczności ponoszenia dodatkowych kosztów pamięciowych. Jednakże zastosowanie bezstratnej redukcji zależności bez uprzedniej transformacji kodu daje umiarkowane wyniki. W takim przypadku pozostało 2483 z 3228 relacji (patrz tabela 4, pierwszy wiersz).

## 5. Analiza zużycia zasobów w zależności od transformacji

Użycie każdej z wyżej przedstawionych transformacji dokonującej modyfikacji kodu źródłowego wiąże się z koniecznością zużycia pewnej ilości zasobów pamięciowych systemu. Dla systemów wbudowanych ma to zasadnicze znaczenie i często może być wyznacznikiem czy dana transformacje będzie mogła zostać zastosowana. W tabeli 5 przedstawiono zestawienie zużycia zasobów w zależności od zastosowanej transformacji. Pierwsza kolumna tabeli określa jaka transformacja została wykonana, w drugiej kolumnie przedstawiono zużycie pamięci konieczne do poprawnego wykonania transformacji, trzecia kolumna to skrócony komentarz do uzyskanych wyników.

W przypadku prywatyzacji zmiennych skalarnych ilość niezbędnej pamięci zależna jest od liczby wątków na których miałby zostać wykonany kod. W tym przypadku dla każdego wątku musi zostać prywatna kopia sześciu zmiennych typu całkowitego (4 bajty). Zgodnie z wynikami w tabeli 4 prywatyzacja pozwala na osiągniecie dość dobrych wyników jeśli chodzi o redukcję relacji zależności. Transformacja złączenia pętli i zwinięcia instrukcji w pętle wymaga stosunkowo mało dodatkowych zasobów. Dla wybranego przykładu należało stworzyć jedynie jedną zmienną indeksową dla dodatkowej pętli. Kolejna transformacja, zwiększenie wymiarów zmiennych tablicowych, wymaga już znacznych nakładów pamięciowych. W kodzie wynikowym zwiększono wymiary dwóch tablic: jedno- i dwuwymiarowej. Przy założeniu że N1=100 dla pierwszej tablicy niezbędne było zalokowanie 5 tablic o 100 elementach, natomiast dla drugiej tablicy zarezerwować pamięć dla 5*5 tablic o wymiarze 100 elementów. W sumie koszt transformacji dla całej pętli to 24000 bajtów. Dodanie kolejnych czterech tablic do poprzedniej transformacji pozwoliło na zredukowanie kolejnych 51 relacji zależności, jednakże niezbędne było zarezerwowanie dodatkowych 2000 bajtów.

Tab. 5. Koszty pamięciowe transformacji dla wybranego przykładu
Tab. 5. Transformation memory usage for a given example

| Transformacja | Dodatkowe narzuty pamięciowe (bajty) | Komentarz |
|---|---|---|
| brak | 0 | |
| prywatyzacja zmiennych skal. | 6*N*4 | 6 zmiennych prywatnych typu int (4 bajty) dla N wątków |
| złączenie pętli + zwinięcie instr. w pętle | 1*4 = 4 | jedna zmienna indeksowa dla pętli typu int (4 bajty) |
| zwiększenie wym. zm. tablicowych | (5*N1 + 5*5*N1) * 8 | zwiększenie wymiaru dwóch różnych tablic typu double |
| zwiększenie wym. zm. tab. + dodanie nowych tab. | (5*N1 + 4*5*N1 + 5*5*N1) * 8 | analogicznie jak poprzednia transformacja + stworzenie 4 nowych tablic |
| wszystkie możliwe transf. | 6*N1*4 + (6 + (5*N1 + 4*5*N1) + (5*5*N1 + 4*5*5*N1) )*8 | zastosowanie wszystkich wcześniejszych transformacji jednoczenie |

W wyniku połączenia wszystkich wcześniej wymienionych transformacji udało się zredukować 3159 relacji zależności. Koszt takiej transformacji to 122448 bajtów jakie trzeba dodatkowo zaalokować. Średnio aby wyeliminować jedną relację konieczne było zarezerwowanie dodatkowych 38 bajtów.

## 6. Literatura


[1] Ramming F. J.: Distributed and Parallel Embedded Systems, Kluwer Academic Publishers, 1999.
[2] Sha E. H. M.: Parallel Embedded Systems: Optimizations and Challenges, Lecture Notes in Computer Science, vol. 3824/2005, Springer, 2005.
[3] Vadim Maslv: Lazy Array Data-Flow Dependence Analysis, Univ. of Maryland, College Park, July 1993.
[4] Eric Stolz, Michael Wolfe: Detecting Value-Based Scalar Dependence, Departament of Computer Science and Engineering Oregon Graduate Institude of Science & Technology.
[5] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, Jeff McDonald: Parallel Programming in OpenMP, Morgan Kaufman Publishers.
[6] Bielecki W., Siedlecki K.: Finding Sources of Synchronization-free Slices in Perfectly Nested Loops, ELECTRONIC MODELING, vol.29, No.3, Kijów 2007, ss. 41-53, 2007.
[7] Bielecki W., Siedlecki K.: Extracting Synchronization-free Slices in Perfectly Nested loops, ELECTRONIC MODELING, vol.29, No.6, Kijów 2007, ss. 61-76, 2007.
[8] Pugh W. and Wonnacott D.: An exact method for analysis of value-based array data dependences. In In Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Springer-Verlag, 1993.


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