Włodzimierz BIELECKI, Krzysztof KRASKA
POLITECHNIKA SZCZECIŃSKA, WYDZIAŁ INFORMATYKI

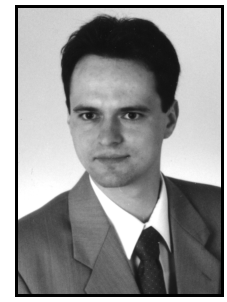
Zwiększenie lokalności programów równoległych wykonywanych w systemach osadzonych

Prof. dr hab. inż. Włodzimierz BIELECKI

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



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

Zwiększenie lokalności danych w programie jest niezbędnym elementem zwiększenia wydajności części programowych systemu osadzonego, zmniejszenia zużycia energii oraz redukcji rozmiaru pamięci w układzie. Przedstawiono komplementarne wykorzystanie metody szacowania lokalności danych wobec nowej metody ekstrakcji wątków, ich aglomeracji w celu dostosowania do możliwości docelowej architektury przy zastosowaniu różnych typów podziału iteracji pętli (mapowanie czasowo-przestrzenne) i z uwzględnieniem wpływu zastosowania znanych technik poprawy lokalności danych. Wybór najlepszej kombinacji transformacji kodu pod kątem lokalności danych umożliwia zwiększenie wydajności programu względem wskazanych czynników. Zaprezentowano podejście do analizy lokalności danych dla wybranych pętli, przedstawiono i omówiono wyniki badań eksperymentalnych a także wskazano kierunki dalszych prac.

Słowa kluczowe: lokalność danych, kompilatory, systemy osadzone, przetwarzanie równoległe, transformacje pętli programowych.

Increasing data locality of parallel programs executed in embedded systems

Abstract

Increasing data locality in a program is a necessary factor to improve performance of software parts of embedded systems, to decrease power consumption and reduce memory on chip size. A possibility of applying a method of quantifying data locality to a novel method of extracting synchronization-free threads is introduced. It can be used to agglomerate extracted synchronization-free threads for adopting a parallel program to a target architecture of an embedded system under various loop schedule options (space-time mapping) and the influence of well known techniques to improve data locality. The choice of the best combination of loop transformation techniques regarding to data locality makes possible improving program performance. A way of an analysis of data locality is presented. Experimental results are depicted and discussed. Conclusion and future research are outlined.

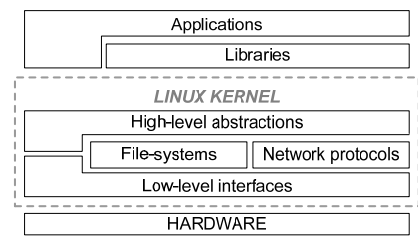
Keywords: data locality, compilers, embedded systems, parallel processing, loop transformations.

1. Wstęp

Współczesne systemy osadzone wspomagające proces przetwarzania (ang. embedded processing) składają się z programowalnych procesorów i przetwarzanych przez nie komponentów programowych oraz pozostającej w interakcji części sprzętowej najczęściej w postaci matryc FPGA. Części programowane umożliwiają szybkie wprowadzanie poprawek, elastyczną zmianę zastosowania danego programu, ponowne użycie kodu, przez co skracają czas dostarczenia produktu na rynek (ang. Time-to-Market). Programowalne procesory zużywają znacznie więcej energii a także są znacznie wolniejsze niż ich odpowiedniki sprzętowe. Rozwiązania sprzętowe zapewniają większą wydajność i mniejszy pobór mocy jednak czas ich projektowania jest długi a przez to również kosztowny [10].

Architektury wieloprocesorowe dla systemów osadzonych są powszechnie obecne na rynku. Dla przykładu układ FPGA Virtex-4FX firmy Xilinx. oferuje do 2 procesorów PowerPC405, układy Geode firmy National Semiconductor umożliwiają budowę wieloprocesorowych systemów osadzonych opartych o architekturę x86. Prowadzony przez Hewlett Packard projekt HPOC (ang. Hundred Processors, One Chip) zmierza do umieszczenia w jednym układzie nawet setek procesorów współdzielących pamięć [4].

Wśród systemów operacyjnych dla systemów osadzonych znacznym uznaniem cieszy się środowisko Linux, choć zgodnie z tradycyjnymi standardami dla systemów osadzonych, są to systemy klasyfikowane jako duże. Zastosowanie Linuxa w systemie osadzonym nie wymaga stosowania jego specjalnej wersji – do stworzenia systemu osadzonego wystarczy użyć, podobnie jak dla komputera ogólnego przeznaczenia, oficjalnego wydania jądra. Często jednak używa się takiej kompilacji jądra, która jest dostosowana dla specyficznego sprzętu i wspiera wymagane typy aplikacji. Linux obsługuje różne architektury dedykowane systemom osadzonym jak: PowerPC, MIPS, ARM, x86, M68k i inne [9]. Ogólną architekturę systemu osadzonego Linux przedstawia rysunek 1.



Rys. 1. Ogólna architektura osadzonego systemu Linux [9]
Fig. 1. The general architecture of an embedded Linux system [9]

Podobnie jak w przypadku każdego oprogramowania, tworzenie oprogramowania dla systemów osadzonych wymaga użycia takich narzędzi jak kompilator języka programowania, asembler czy debugger. Przykładem narzędzia do modelowania, projektowania i weryfikacji systemów elektronicznych jest, zatwierdzony jako standard IEEE Std.1666-2005, język SystemC™ zapewniający implementację przy użyciu notacji C++ zarówno części sprzętowych jak i programowych systemu osadzonego. Optymalna implementacja przez programistę oprogramowania dla wieloprocesorowego systemu osadzonego ma krytyczne znaczenie dla jego wydajności oraz zużycia energii. Powszechną natomiast cechą istniejącego oprogramowania, wykonującego zazwyczaj intensywne obliczenia na zbiorach danych w pętlach programowych, jest mała lokalność danych [6]. Ze względu na nieoptymalną postać kodu dane często muszą zostać ponownie dostarczone z wolniejszej pamięci zewnętrznej, zużywając dodatkową energię.

Nieoptymalny kod można poddać transformacji podczas kompilacji programu umożliwiając zwiększenie wydajności części programowych, zmniejszenie zużycia energii oraz redukcję rozmiaru pamięci w układzie.

Ponieważ pamięci cache najczęściej działają z pełną prędkością procesora zaś tańsze, lecz bardziej pojemne moduły pamięci zewnętrznej pracują z kilkukrotnie mniejszą częstotliwością taktowania, to czas dostępu do danych zawartych w cache jest wielokrotnie mniejszy. Zwiększenie lokalności przetwarzanych w programie danych skutkuje poprawą wykorzystania szybkiej pamięci cache i zmniejszeniem odwołań do wolniejszych modułów pamięci niższego poziomu, co przekłada się na ogólne zwiększenie wydajności programu.

2. Metoda szacowania lokalności danych

Dwa pojęcia określają lokalność danych: czasowa lokalność (ang. temporal locality) oraz przestrzenna lokalność (ang. spatial locality). Czasowa lokalność ma miejsce wówczas, gdy te same dane wykorzystywane są wielokrotnie w krótkim odstępie czasu. Przestrzenna lokalność ma miejsce wówczas, gdy w krótkim odstępie czasu wykorzystywane są inne dane aczkolwiek rozmieszczone obok siebie; takie dane z wysokim prawdopodobieństwem wciąż znajdują się w wierszach szybkiej pamięci podręcznej procesora obok danych uprzednio użytych. Lokalność danych wiąże się z ich ponownym użyciem (ang. reuse). Ponowne użycie danych zachodzi pomiędzy referencjami do tych samych lub innych miejsc pamięci danej tablicy. Własne ponowne użycie (ang. self reuse) określa czasową lub przestrzenną lokalność danych tej samej referencji do tablicy. Grupowe ponowne użycie (ang. group reuse) określa natomiast ponowne użycie danych pomiędzy różnymi referencjami do tej samej tablicy.

W [3] została przedstawiona metoda szacowania lokalności danych. Metoda znajduje zastosowanie, gdy referencje do elementów m-wymiarowych zmiennych tablicowych $X[f_1(i),\ldots,f_m(i)]$ wyrażone są w postaci funkcji afinicznych – system tych równań jest wówczas przedstawiany w kanonicznej formie macierzowej $Ai^T \le c$, gdzie $A^{m\times n}$ jest macierzą współczynników referencji w n pętlach, c jest wektorem offsetu. Dla dwóch różnych iteracji pętli i_1 oraz i_2:

a) własne-czasowe ponowne użycie (ang. self-temporal reuse) ma miejsce wówczas, gdy:

$$f_k(i_2) - f_k(i_1) = 0;\ 1 \le k \le m;$$

stąd:

$$Ai_1 + c = Ai_2 + c;\ A(i_2 - i_1) = 0;\ Ad = 0;$$

gdzie $d = (i_2 - i_1)$ jest wektorem odległości między iteracjami. Zbiór wszystkich rozwiązań tego równania jest przestrzenią nulową (ang. null space) A, oznaczaną jako *kernel A*. Pętla *k* niesie ponowne użycie, jeżeli istnieje wektor $d \in$ *kernel A* taki, że d_k jest pierwszym niezerowym elementem w d. Szczególnie interesujące są takie d, które są krotnościami jednostkowego wektora normalnego e_k.

b) własne-przestrzenne ponowne użycie (ang. self-spatial reuse) ma miejsce wówczas, gdy:

$$f_k(i_2) - f_k(i_1) = 0;\ 1 \le k \le m-1;$$

gdzie m jest wymiarem ciągłego obszaru pamięci (dla C/C++ jest to ostatni wymiar), stąd:

$$A_s i_1 + c = A_s i_2 + c;\ A_s(i_2 - i_1) = 0;\ A_s d = 0;$$

Zbiór wszystkich rozwiązań tego równania oznaczamy jako *kernel* A_s. Przestrzenne ponowne użycie dla pętli *k* zachodzi tylko, gdy $e_k \in$ *kernel* A_s a współczynnik indeksu w wyrażeniu dla wymiaru m jest mniejszy niż rozmiar linii pamięci cache.

c) grupowe-czasowe ponowne użycie (ang. group-temporal reuse) ma miejsce wówczas, gdy:

$$Ai_1 + c_1 = Ai_2 + c_2;\ A(i_2 - i_1) = c_1 - c_2;\ Ad = c_1 - c_2;$$

Jeżeli $d = 0$ wówczas grupowe-czasowe ponowne użycie zachodzi w tej samej iteracji. W przeciwnym wypadku ponowne użycie jest realizowane przez najbardziej zewnętrzną pętlę o pierwszym niezerowym elemencie d.

d) grupowe-przestrzenne ponowne użycie (ang. group-spatial reuse) ma miejsce wówczas, gdy:

$$A_s i_1 + c_{s,1} = A_s i_2 + c_{s,2};\ A_s(i_2 - i_1) = c_{s,1} - c_{s,2};\ A_s d = c_{s,1} - c_{s,2};$$

Grupowe-przestrzenne ponowne użycie zachodzi tylko w wymiarze ciągłego obszaru pamięci i tylko wówczas, gdy odległość ponownego użycia dla elementów tablicy jest mniejsza niż rozmiar linii pamięci cache. Ponadto, przestrzenne ponowne użycie dla wielu referencji do tej samej tablicy ulegnie poprawie tylko, gdy $GCD(a_{s,1}, a_{s,2}, \ldots, a_{s,n})$ współczynników wymiaru ciągłego obszaru pamięci jest większy od 1 oraz gdy $c_{s,1} - c_{s,2}$ nie jest krotnością GCD; w przeciwnym wypadku grupowe-przestrzenne ponowne użycie jest jedynie podzbiorem grupowego-czasowego lub własnego-przestrzennego ponownego użycia. Należy zwrócić uwagę na ograniczenie metody – dla grupowego ponownego użycia wyrażenia indeksowe dla wymiarów tablicowy posiadają tę samą macierz *A*.

Szacunkowy pomiar ponownego użycia może zostać dokonany przy użyciu współczynnika ponownego użycia (ang. reuse factor):

a) współczynnik własnego-czasowego ponownego użycia (ang. self-temporal reuse factor) dla pętli *k* jest równy liczbie iteracji tej pętli N_k jeżeli $e_k \in$ *kernel A* lub 1 w sytuacji przeciwnej;

b) współczynnik własnego-przestrzennego ponownego użycia (ang. self-spatial reuse factor) dla pętli *k* jest równy $\max(1, l/a_{sk})$, gdzie l jest rozmiarem linii pamięci cache zaś a_{sk} jest współczynnikiem indeksu w wyrażeniu dla wymiaru *m* lub 1 w sytuacji przeciwnej;

c) współczynnik własnego ponownego użycia (ang. self-reuse factor) R_k dla referencji w pętli *k* jest równy współczynnikowi czasowego ponownego użycia, jeżeli ten jest większy od 1 i równy współczynnikowi przestrzennego ponownego użycia w przeciwnej sytuacji;

d) łączny współczynnik własnego-ponownego użycia (ang. cumulative self-reuse factor) R_k^* dla referencji w pętli *k* jest produktem R_k dla tej referencji w pętli *k* i wszystkich wewnętrznych pętli zawierających tę referencję.

e) łączny współczynnik ponownego użycia (ang. cumulative reuse factor) jest sumą R_k^* dla wszystkich referencji w pętli;

f) współczynnik grupowego-czasowego ponownego użycia (ang. group-temporal reuse factor) wynosi ∞ dla wszystkich pętli jeżeli $d = 0$. W przeciwnym wypadku dla pierwszego niezerowego elementu d_k w d, współczynnik dla pętli *k* wynosi N_k / d_k i 1 dla pozostałych pętli. Aby wyliczyć współczynnik należy posegregować badane referencje w taki sposób, aby odległość pomiędzy sąsiednimi referencjami była leksykograficznie nieujemna i następnie dla każdej, prócz pierwszej, referencji dokonać obliczeń w stosunku do poprzedniej referencji na liście;

g) współczynnik grupowego-przestrzennego ponownego użycia (ang. group-spatial reuse factor) wynosi 1 dla wszystkich pętli jeżeli istnieje własne-przestrzenne ponowne użycie dla referencji lub gdy $GCD(a_{s,1}, a_{s,2}, \ldots, a_{s,n})$ dzieli $c_{s,1} - c_{s,2}$. Jeżeli $d = 0$ współczynnik wynosi N_k dla najbardziej wewnętrznej pętli i 1 dla pozostałych pętli. W przeciwnym wypadku dla pierwszego niezerowego elementu d_k w d, współczynnik dla pętli *k* wynosi N_k / d_k i 1 dla pozostałych pętli. Aby wyliczyć współczynnik należy posegregować badane referencje w taki sposób, aby odległość pomiędzy sąsiednimi referencjami była leksykograficznie nieujemna;

h) całkowity współczynnik grupowego-ponownego użycia (ang. total group-reuse factor) dla każdej referencji w każdej z pętli jest produktem grupowych-czasowych i grupowych-przestrzennych współczynników ponownego użycia dla pętli;
i) łączny współczynnik grupowego-ponownego użycia (ang. cumulative group-reuse factor) dla pętli jest produktem grupowych współczynników ponownego użycia dla danej pętli i wszystkich wewnętrznych pętli zawierających daną referencję;
j) ogólny współczynnik ponownego użycia (ang. generalized reuse factor) jest wyliczany poprzez pomnożenie łącznego współczynnika własnego-ponownego użycia przez łączny współczynnik grupowego-ponownego użycia.

Odcisk danych (ang. data footprint) dla danej referencji jest liczbą linii pamięci cache, które zostaną dostarczone przez daną pętlę. Jeżeli nie występuje ponowne użycie wówczas pętla będzie odwoływała się do jednego elementu w każdej iteracji, który ponadto będzie znajdował się w innej linii cache. Wartość odcisku danych na poziomie k z n zagnieżdżonych pętli jest obliczana zgodnie ze wzorem

$$F_k^* = l\prod_{i=k}^{n}\frac{N_k}{R_k},$$

gdzie: N_k jest równy liczbie iteracji k, l jest rozmiarem linii pamięci cache, R_k jest współczynnikiem ponownego użycia redukującym odcisk danych. Ogólny odcisk danych (ang. generalized data footprint) jest wyliczany poprzez podzielenie odcisku danych przez łączny współczynnik grupowego-ponownego użycia w celu uwzględnienia ponownego użycia pomiędzy różnymi referencjami. Ponowne użycie danych może zostać użyte jako miara lokalności dla pamięci cache, jeżeli odcisk danych mieści się w całości w pamięci cache.

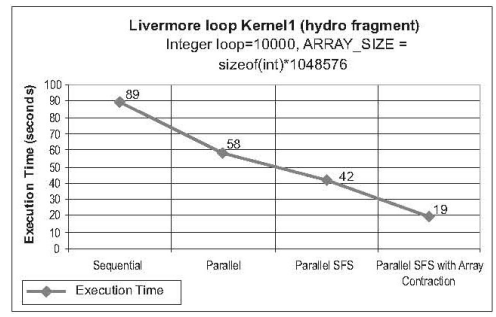
3. Analiza lokalności danych dla równoległych wątków pętli

W [1] przedstawiono nową metodę ekstrakcji równoległych wątków w pętlach programowych umożliwiającą zwiększenie poziomu zrównoleglenia. Metoda pozwala na ekstrakcję większej ilości równoległych wątków niż inne metody. Szacowanie ponownego użycia oraz odcisku danych przedstawione w rozdziale 2 umożliwia wybór takiego porządku pętli, który zwiększa lokalność danych programu. Z perspektywy metody ekstrakcji wątków zastosowanie szacowania lokalności danych jest elementem niezbędnym dla uzyskania kodu optymalnego pod kątem wydajności programu wykonywanego na docelowej architekturze – metoda zakłada ekstrakcję maksymalnej ilości równoległych wątków natomiast architektura docelowa systemu osadzonego posiada ustaloną, zazwyczaj mniejszą niż liczba wydzielonych wątków, ilość rdzeni CPU (np.: 2 rdzenie PowerPC405 dla Xilinx Virtex-4FX). Dlatego niezbędne staje się dostosowanie poziomu równoległości programu do możliwości docelowej architektury [11]. Przeprowadzone dotychczas badania z wykorzystaniem komputerów wieloprocesorowych wskazują, że ekstrakcja równoległych wątków oraz zastosowanie technik kontrakcji macierzy (ang. array contraction) i blokowania (ang. blocking | tiling) dla poszczególnych wątków może znacznie zwiększyć wydajność programu – w badaniach eksperymentalnych [2] dla kodu Kernel 1 (hydro fragment) pętli livermore [5] oraz podstawowego algorytmu mnożenia macierzy [6] uzyskano znaczące zmniejszenie czasu wykonania programów (rysunek 2a oraz rysunek 2b).

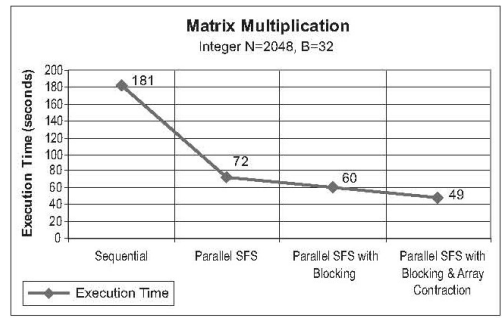
Z drugiej strony, przykład prostego kodu (rysunek 3) wykonanego w tym samym środowisku udowadnia, że ekstrakcja równoległych wątków może ograniczać wydajność programu – czas wykonania równoległego kodu (8 sekund) był o 30% dłuższy niż kodu sekwencyjnego (6 sekund).

Widocznym elementem kodu równoległego jest zmniejszenie przestrzennego-ponownego użycia dla referencji do tablicy *a*[] oraz konieczność synchronizacji pamięci cache procesorów (cache coherence protocol) w konsekwencji modyfikacji sąsiednich elementów pamięci przez odrębne procesory.

a)



b)



Rys. 2. Czas wykonania kodu: (a) Kernel 1 pętli livermore (b) mnożenia macierzy
Fig. 2. Execution time of: (a) Livermore Loops Kernel 1 (b) matrix multiplication

a) kod sekwencyjny

```
...
#define SIZE 100000000
int main(void)
{
   double *a = new double[SIZE];
   for(long i = 0; i < SIZE; i++) a[i] = (double)i+1;
   CTime curTime = CTime::GetCurrentTime();
   time_t time = curTime.GetTime();
   for(long i = 0; i < (SIZE-2) ; i++)
       a[i+2] = sin( (a[i]*SIZE + 1) / SIZE );
    curTime = CTime::GetCurrentTime();
    time = curTime.GetTime() - time;
    cout << "> Execution time: " << time << endl;
    return 0;
}
```

b) kod równoległy

```
#include <omp.h>
...
#define SIZE 100000000
int main(void)
{
    omp_set_num_threads(2);
    double *a = new double[SIZE];
    for(long i = 0; i < SIZE; i++) a[i] = (double)i+1;
    CTime curTime = CTime::GetCurrentTime();
    time_t time = curTime.GetTime();
    #pragma omp parallel for private(j,i) shared(a)
    for(long j=0; j < 2 ; j++)
         for(long i=j; i < (SIZE-3) ; i+=2)
              a[i+2] = sin( (a[i]*SIZE + 1) / SIZE );
    curTime = CTime::GetCurrentTime();
    time = curTime.GetTime() - time;
    cout << "> Execution time: " << time << endl;
    return 0;
}
```

Rys. 3. Zmniejszenie wydajności w równoległej wersji prostego kodu
Fig. 3. Decreased performance in the parallel version of a simple code

Zapewnienie optymalnej wydajności programu przy ograniczeniu poziomu równoległości do możliwości docelowej architektury wymaga iteracyjnej oceny poziomu lokalności danych różnych kombinacji aglomeracji wątków wraz z różnymi typami planowania przydziału operacji (iteracji pętli) do poszczególnych wątków (ang. space-time mapping – mapowanie czasowo-przestrzenne) i uwzględnieniem wpływu zastosowania technik poprawy lokalności danych. Wybór najlepszej kombinacji pod kątem lokalności danych zapewni optymalną wydajność programu względem wskazanych czynników.

Dla obu źródeł Kernel 1 (hydro fragment) pętli z zestawu Livermore Loops oraz podstawowego algorytmu mnożenia macierzy poziom równoległości (liczba utworzonych wątków) został dostosowany do docelowej architektury poprzez zastosowanie funkcji biblioteki OpenMP *omp_set_num_threads(omp_get_num_procs())*. Dla użytej architektury liczba utworzonych wątków wynosiła 4, przy czym 2 wątki wykonywane na jednym procesorze z technologią Hyper-Threading współdzieliły pamięć cache. Rozmiar linii DataCache-L1 użytego procesora wynosi 128-bajty, co odpowiada 32 elementom tablic. Ponieważ w obu programach dla dyrektywy *#pragma parallel for* nie określono sposobu planowania iteracji do wątków, kompilator samodzielnie zastosował typ *static* oraz przyjął dla każdego z 4 wątków równy rozmiar *chunk* wynoszący ¼ odpowiednio kolejnych iteracji ze wszystkich iteracji pętli [13].

Wartości lokalności danych uzyskane w wyniku wykorzystania metody przytoczonej w rozdziale 2 dla podstawowego algorytmu mnożenia macierzy kodu Kernel 1 (hydro fragment) oraz pętli livermore przedstawione zostały odpowiednio w tabeli 1 oraz tabeli 2 i tabeli 3.

Tab. 1. Współczynniki ponownego użycia dla algorytmu mnożenia macierzy
Tab. 1. Reuse factors for the matrix multiplication algorithm

Referencja	Współczynniki ponownego użycia														
	Czasowe			Przestrzenne			Własne-ponowne użycie			Łączne-własne ponowne użycie			Odcisk danych		
	k	j	i	k	j	i	R_k	R_j	R_i	R_k^*	R_j^*	R_i^*	F_k^*	F_j^*	F_i^*
z[i][j]	N	1	1	1	32	1	N	32	1	N	32N	32N	1	N/32	$N^2/128$
x[i][k]	1	N	1	32	1	1	32	N	1	32	32N	32N	N/32	N/32	$N^2/128$
y[k][j]	1	1	N/4	1	32	1	1	32	N/4	1	32	8N	N	$N^2/32$	$N^2/32$
Łączne							N+33	N+64	N/4+2	N+33	64N+32	72N	N+N/32 + 1	$N^2/32$ + N/16	$6N^2/128$

Tab. 2. Współczynniki własnego-ponownego użycia dla kodu Kernel 1
Tab. 2. Temporal reuse factors for the Livermore Loops Kernel 1

Referencja	Współczynniki ponownego użycia									
	Czasowe		Przestrzenne		Własne-ponowne użycie		Łączne-własne ponowne użycie		Odcisk danych	
	k	l	k	l	R_k	R_l	R_k^*	R_l^*	F_k^*	F_l^*
x[k]	1	loop	32	1	32	loop	32	32loop	n/32	n/128
y[k]	1	loop	32	1	32	loop	32	32loop	n/32	n/128
z[k+10]	1	loop	32	1	32	loop	32	32loop	n/32	n/128
z[k+11]	1	loop	32	1	32	loop	32	32loop	n/32	n/128
Łączne					128	4loop	128	128loop	n/8	n/32

Tab. 3. Współczynniki przestrzennego-ponownego użycia dla kodu Kernel 1
Tab. 3. Spatial-reuse factors for the Livermore Loops Kernel 1

Referencja	Współczynniki ponownego użycia					
	Własne-przestrzenne ponowne użycie		Grupowe-czasowe ponowne użycie		Łączne-grupowe ponowne użycie	
	k	l	k	l	k	L
z[k+11]	32	1	1	1	32	32
z[k+10]	32	1	1	loop	32	32loop

W przypadku algorytmu mnożenia macierzy nie występują referencje do tej samej tablicy, dlatego zastosowanie mają jedynie współczynniki zdefiniowane dla własnego-ponownego użycia. Dla pierwotnego wolumenu danych przy rozmiarze tablic $N \times N$, gdzie N=2048 elementy, oraz podziału liczby iteracji dla najbardziej zewnętrznej pętli L_i pomiędzy 4 równoległe wątki, wartość odcisku danych najbardziej zewnętrznej pętli F_i^* wielokrotnie przekraczała rozmiar dostępnej pamięci lokalnej DataCache-L1 skutkując odwołaniami do wolniejszego poziomu pamięci.

$$F_i^* = \frac{6 * N^2}{128};\ N = 2048;\ F_i^* = 196608 \cdot$$

Wartość F_i^*=196608 uwzględniająca 4-bajtowy rozmiar elementu tablicy daje 768KB wymaganej pamięci. Po zastosowaniu techniki blokowania i ustaleniu rozmiaru bloku B=32 elementy wartość odcisku danych uległa znaczącemu zmniejszeniu powodując, iż wolumen danych mógł w całości zostać umieszczony w pamięci DataCache-L1 procesora.

$$F_i^* = \frac{6 * B^2}{128};\ B = 32;\ F_i^* = 48 \cdot$$

Wartość F_i^*=48 uwzględniająca 4-bajtowy rozmiar elementu tablicy daje 192B wymaganej pamięci. Należy pamiętać, że pamięć DataCache-L1 procesora była współdzielona przez 2 równolegle wykonywane wątki, co w konsekwencji skutkuje podziałem dostępnego rozmiaru DataCache-L1 procesora pomiędzy tymi wątkami.

W przypadku kodu Kernel 1 (hydro fragment) pętli z zestawu Livermore Loops współczynniki własnego-ponownego użycia zarówno dla kodu reprezentującego drobno-ziarnistą równoległość jak i kodu z wydzielonymi wątkami metodą [1] są identyczne. Widoczne jest jednak to, że dla kodu z wydzielonymi wątkami dystans ponownego użycia jest mniejszy w przypadku, gdy pętla L_l jest najbardziej wewnętrzną.

W tym przypadku również występuje grupowe-ponowne użycie pomiędzy referencjami *z[k+11]* i *z[k+10]* ułożonymi w sposób, aby wektor odległości *d* był leksykograficznie nieujemny pomiędzy referencjami. Dla obu referencji mamy również własne-czasowe oraz własne-przestrzenne ponowne użycie. Grupowy-przestrzenny współczynnik ponownego użycia zgodnie z metodą nie jest wyliczany ze względu na istnienie własnego-przestrzennego współczynnika ponownego użycia.

Zgodnie z metodą dla pętli L_l dzielimy odcisk danych przez łączny-grupowy współczynnik ponownego użycia wynoszący *32*loop* w wyniku, czego uzyskujemy ogólny współczynnik ponownego użycia wynoszący:

$$n/(32^2\, loop) \cdot$$

4. Badania eksperymentalne

Badania eksperymentalne zostały wykonane przy użyciu dedykowanego do rozwoju oprogramowania dla systemów osadzonych programowego symulatora IBM PowerPC Multi-Core Instruction Set Simulator v1.29 (MC-ISS) [7] oraz komplementarnego debuggera IBM RISCWatch v6.0i kodu C dla architektury PowerPC 405/440/460 [8]. Poziom utylizacji cache procesorów został odczytany ze statystyk DCU (ang. Data Cache Unit) symulatora poprzez moduł integrujący RISCWatch z MC-ISS (sim readdcu).

Do wykonania badań użyto następującej konfiguracji symulatora: 2 procesory PowerPC405, 16KB two-way set-associative data-cache – linia cache: 8 słów (32 bajty), brak cache L2. Tym samym konfiguracja symulatora w przybliżeniu odpowiada np.: układowi Xilinx Virtex-4FX.

Badane źródła zostały opracowane w typowy dla wytwarzania oprogramowania osadzonego sposób, z wykorzystaniem międzyplatformowego środowiska złożonego z komputera i docelowego systemu [8]. Opracowane źródła C zostały skompilowane w środowisku Linux x86 przy użyciu kompilatora gcc-3.3.1 do formatu PowerPC Embedded ABI i poddane badaniom w docelowym środowisku operacyjnym z wykorzystaniem symulatora MC-ISS. Ze względu na możliwości docelowej architektury w źródłach wydzielono dwa wątki przetwarzania. Sposób planowania iteracji pętli do wątków określono w sposób charakterystyczny dla typu *static* przyjmując dla każdego z 2 wątków równy rozmiar *chunk* wynoszący ½ odpowiednio kolejnych iteracji ze wszystkich iteracji pętli.

Dla algorytmu mnożenia macierzy [6] wykonywanego w programowym symulatorze systemu osadzonego MC-ISS uzyskano wyniki zawarte w tabeli 4.

Dla kodu Kernel 1 (hydro fragment) pętli livermore [5] wykonywanego w programowym symulatorze systemu osadzonego MC-ISS uzyskano wyniki zawarte w tabeli 5.

Uzyskane wyniki badań eksperymentalnych potwierdzają wyniki uzyskiwane dla komputerów wskazując na znaczną poprawę skuteczności wykorzystania DCU procesora PowerPC405 dla kodu stosującego techniki zwiększania lokalności danych.

Tab. 4. Wyniki eksperymentalne utylizacji DCU dla algorytmu mnożenia macierzy (N=256, B=8)
Tab. 4. The experimental results of DCU utilization for the matrix multiplication code (N=256, B=8)

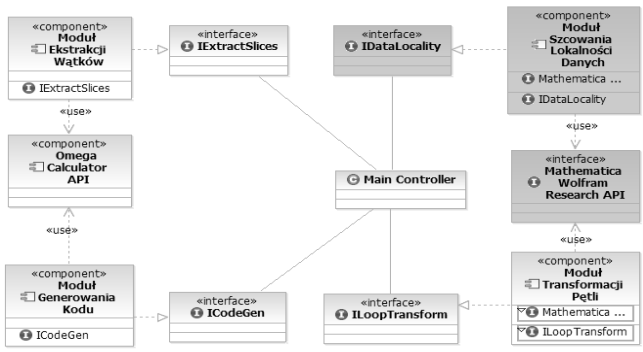
RISCWatch STATUS	Sequential		Parallel SFS		Parallel SFS with Blocking		Parallel SFS with Blocking & Array Contraction	
	CPU0	CPU 1	CPU0	CPU1	CPU0	CPU1	CPU0	CPU1
DCU total accesses	31852424	N/A	127044104	127044104	145012296	145012296	119846472	119846472
DCU misses	2160751	N/A	8634538	8634538	317789	317789	317789	317789
misses/total %	6,8%	N/A	6,8%	6,8%	0,22%	0,22%	0,27%	0,27%

Tab. 5. Wyniki eksperymentalne utylizacji DCU dla Kernel 1 (loop=100; array_size=8192*sizeof(int))
Tab. 5. The experimental results of DCU utilization for the Kernel 1 (loop=100; array_size=8192*sizeof(int))

RISCWatch STATUS	Sequential		Parallel		Parallel SFS		Parallel SFS with Array Contraction	
	CPU0	CPU1	CPU0	CPU1	CPU0	CPU1	CPU0	CPU1
DCU total accesses	11527637	N/A	5800687	5800687	11576472	11576472	8399916	8399916
DCU misses	309546	N/A	155799	155799	5130	5130	5131	5131
misses/total%	2,69%	N/A	2,69%	2,69%	0,04%	0,04%	0,06%	0,06%

5. Rozwijane oprogramowanie

Wyniki prac są rozwijane przez autorów w postaci oprogramowania narzędziowego o charakterze akademickim – kompilatora typu źródło-do-źródła (ang. source-to-source compiler). Poglądową strukturę tworzonego oprogramowania przedstawia rysunek 4.



Rys. 4. Rozwijane narzędzie implementujące uzyskiwane wyniki prac
Fig. 4. A structural overview of the software to build based on the results of research

Główny Kontroler steruje przepływem wykonania oprogramowania. Moduł Ekstrakcji Wątków implementuje przedstawioną w [1] metodę ekstrakcji równoległych wątków w pętlach programowych; moduł dokonuje ekstrakcji z postaci wejściowego pliku źródłowego C wykorzystując do analizy zależności bibliotekę Omega Calculator [12]. Kompilator następnie optymalizuje równoległe wątki pod kątem docelowej architektury wykonawczej. W tym celu Moduł Transformacji Pętli dokonuje wszelkich dostępnych kombinacji aglomeracji wątków z użyciem różnych typów planowania przydziału iteracji pętli do poszczególnych wątków i zastosowaniem technik poprawy lokalności danych: kontrakcji macierzy i blokowania. Moduł Szacowania Lokalności Danych, implementujący przedstawioną w artykule metodę, dla danej kombinacji postaci źródła szacuje współczynnik lokalności danych. Moduł Szacowania Lokalności Danych oraz Moduł Transformacji Pętli wykorzystują silnik algebry liniowej pakietu Mathematica firmy Wolfram Research. Moduł Generowania Kodu odpowiedzialny jest za wygenerowanie optymalnego kodu źródła dla docelowej architektury; przyjęto, że oprogramowanie generuje kod wynikowy zgodnie ze standardem OpenMP 2.0.

6. Wnioski

Szacowanie poziomu lokalności danych dla równoległych wątków wydzielonych za pomocą metody przedstawionej w [1] jest niezbędnym elementem zapewnienia optymalnej wydajności programu dostosowanego do i wykonywanego na docelowej architekturze. Wykonane badania eksperymentalne, w których uzyskano znaczące przyśpieszenia wykonania źródeł poddanych transformacji w celu ekstrakcji równoległych wątków i zwiększenia lokalności danych, potwierdzają zasadność rozważań analitycznych. Przedstawione w artykule badania eksperymentalne dla środowiska systemu osadzonego wskazują na tożsame korzyści uzyskiwane poprzez zwiększenie poziomu utylizacji DCU procesora dla źródeł wykorzystujących techniki zwiększania lokalności danych.

Istotnym spostrzeżeniem porównania wyników analizy lokalności danych oraz dotychczasowych badań eksperymentalnych jest brak możliwości określenia poziomu wykorzystania pamięci cache przez inne funkcjonujące podczas wykonania programu składniki programowe osadzonego systemu operacyjnego. Zasadnym jest przyjęcie założenia, że dostępny rozmiar pamięci cache jest różny dla różnych konfiguracji systemowych. W celu optymalizacji wykorzystania pamięci cache dla programu celowe jest określenie faktycznego dostępnego rozmiaru pamięci cache w środowisku systemowym, nie zaś wyłącznie kierowanie się parametrami fabrycznymi procesora.

Należy zaznaczyć, iż zawężenie obszaru zastosowania wyników pracy jest w istocie rezultatem ograniczenia metody przedstawionej w [3], która grupowe-ponowne użycia stosuje wyłącznie do przypadków, gdy wyrażenia indeksowe elementów tablicy odrębnych referencji tworzą taką samą postać macierzy A. Celem autorów jest rozwinięcie tej metody do postaci umożliwiającej zastosowanie w sytuacji, gdy:

$$A_1 i_1 + c_1 = A_2 i_2 + c_2 ;\ A_2 i_2 - A_1 i_1 = c_1 - c_2 .$$

Otwartym wyzwaniem przez zagadnieniami przetwarzania równoległego, które również przyświeca autorom, wciąż pozostają pętle programowe, w których wyrażenia indeksowe elementów tablicy dla referencji nie mają postaci równań afinicznych.

7. Literatura

[1] W. Bielecki, K. Siedlecki: Extracting synchronization free slices in perfectly nested uniform and non uniform loops, to be published in Electonic Modeling, (artykuł jest przyjęty do wydania).
[2] W. Bielecki, K. Kraska, K. Siedlecki: Increasing Program Locality by Extracting Synchronization Free Slices in Arbitrarily Nested Looks, Proceedings of the Fourteenth International Multi-Conference on Advanced Computer Systems ACS2007, 2007.
[3] M. Wolfe: High Performance Compilers for Parallel Computing, Addison Wesley, 1996.
[4] S. Richardson: MPOC. A Chip Multiprocessor for Embedded Systems, HP Laboratories Palo Alto, 2002. (http://www.hpl.hp.com/techreports/2002/HPL-2002-186.pdf)
[5] Netlib Repository at UTK and ORNL, http://www.netlib.org/benchmark/livermorec
[6] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, 2nd Edition, Addison-Wesley, 2006.
[7] IBM PowerPC Multi Core Instruction Set Simulator. User's Guide. Version 1.2, International Business Machines Corporation, 2008.
[8] IBM RISCWatch Debugger. User's Manual, International Business Machines Corporation, 2008.
[9] K. Yaghmour: Building Embedded Linux Systems, O'Reilly, 2003.
[10] A. Stasiak: Klasyfikacja Systemów Wspomagających Proces Przetwarzania i Sterowania, II Konferencja Naukowa KNWS'05, 2005.
[11] M. Griebl: Habilitation. Automatic Parallelization of Loop Programs for Distributed Memory Architectures, Iniversitat Passau, 2004.
[12] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott: The omega library interface guide. Technical Report CS-TR-3445, University of Maryland, 1995.
[13] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon: Paralel Programing In OpenMP, Morgan Kaufmann, 2001.

Artykuł recenzowany