Marek PAŁKOWSKI[1]

West Pomeranian University of Technology (1)

doi:doi:10.15199/48.2015.11.40

Automatic Extraction of Parallelism for Mobile Devices

Abstract. *This paper presents the Iteration Space Slicing (ISS) framework aimed at automatic parallelization of code for Mobile Internet Devices (MID). ISS algorithms permit us to extract coarse-grained parallelism available in arbitrarily nested parameterized loops. The loops are parallelized and transformed to multi-threaded application for the Android OS. Experimental results are carried out by means of the benchmark suites (UTDSP and NPB) using an ARM quad core processor. Performance benefits and power consumption are studied. Related and future work are discussed.*

Streszczenie. *Artykuł przedstawia ekstrakcję niezależnych fragmentów kodu dla urządzeń przenośnych. Narzędzie pozwala na zrównoleglenie gruboziarniste dowolnie zagnieżdżonych pętli programowych z parametrami do kodu wielowątkowego dla systemu Android. Eksperymenty przeprowadzona na zestawach pętli testowych (UTDSP i NPB) za pomocą czterordzeniowego procesora ARM. Przedstawiono analizę wydajności i poboru mocy oraz pokrewne rozwiązania.* ***(Automatyczna ekstrakcja równoległości dla urządzeń przenośnych)***

Keywords: automatic parallelization algorithms, synchronization-free parallelism, code generation, mobile computing, multi-core processing, DSP applications, power consumption
Słowa kluczowe: automatyczne zrównoleglanie kodu, równoległość bez synchronizacji, generowanie kodu, przetwarzanie na urządzeniach przenośnych, przetwarzanie wielordzeniowe, aplikacje DSP, pobór mocy

Introduction and task definition

To meet the increasing demands that are imposed on modern embedded systems, plenty of computational power is needed. The utilization of multi-core processors in the embedded market segments offers unique challenges: (i) not all embedded OSes and the software tools available on these OSes fully support multi-core processors; (ii) many legacy embedded applications do not move easily to modern multi-core processors. Embedded designs featuring converged and heterogeneous cores increase the programming and communication complexity. Another trend is the movement of multi-core units into new market segments such as Mobile Internet Devices [1].

To exploit full advantages of embedded platforms, the applications have to be split up into several concurrent tasks to enable parallel execution on available processing units. The lack of automated tools permitting for exposing such parallelism decreases the productivity of parallel programmers and increases the time and cost of producing a parallel program.

Most computations are contained in program loops. Because mobile devices more often are equipped with multi-core processors, the automatic extraction of parallelism from loops is extremely important for these multi-core systems, allowing us to produce parallel code from existing sequential applications and to create multiple threads that can be easily scheduled by a load balancers achieving a high system performance.

Loop parallelization is not trivial and dependence analysis is needed. Ignoring loop dependencies causes that parallel code can produce not correct output. Two statement instances I and J are *dependent* if both access the same memory location and if at least one access is a write. I and J are called the *source* and *destination* of a dependence, respectively, provided that I is lexicographically smaller than J ($I \prec J$, i.e., I is always executed before J).

Different techniques have been developed to extract coarse-grained parallelism that is represented with synchronization-free slices of computations available in loops, for example, those presented in papers [2, 3]. Unfortunately, these techniques fail to parallelize loops in some cases [4, 5]. Hence, potential parallelism is left unexploited.

In this paper, Iteration Space Slicing algorithms extracting coarse-grained parallelism from program loops are discussed. Experimental results are carried out in order to check the speedup and efficiency of generated parallel code for the Android OS and an ARM processor with four cores. This software and low-power microprocessors are mainly used in mobile internet devices, smartphones and tablets. Signal processing and parallel benchmarks are used in the experiments.

Related parallelization techniques and problem specification

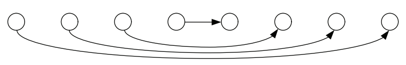
The results of the paper are within the Iteration Space Slicing Framework (ISS) introduced by Pugh and Rosser [6]. That paper examines one of possible uses of ISS, namely how to optimize interprocessor communication. However, Pugh and Rosser do not show how synchronization-free slices can be extracted. Different techniques and compilers based on the polyhedral models [11] have been developed to extract coarse-grained parallelism available in loops.

An automatic parallelization of embedded software using hierarchical task graphs and integer linear programming (ILP) was presented in paper [24]. The tool is able to extract parallelism from the application's source code and focuses on the special requirements of embedded systems. The approach uses an integer linear programming to exploit parallelism of an application. Due to the fact that ILP systems are NP-hard in general, an approximation of the problem description is supported by the framework. The implementation of the parallelization is done by the MPA tool of the ATOMIUM suite.

HELIX [25] presents a technique for automatic parallelization of irregular programs for chip multiprocessing. It uses thread-level parallelism (TLP) among loop iterations that is able to achieve significant speedups without constraints such as a loop nesting limit, regular memory accesses, or regular control flow. The iterations of each parallelized loop run in round-robin order on cores of a single processor. HELIX applies code transformations to minimize the inefficiencies of sequential segments, data transfer, signalling, and thread management. Choosing the right loops to parallelize is one key to the success of HELIX, which combines an analytical model of loop speedup with profiling data to choose the most profitable loop sets.

By extending prior work on critical-path analysis (CPA) to incorporate real-world constraints, the Kremlin tool [26] implements a practical oracle that predicts outcomes for sequential-code parallelization. The tool takes in an unmodified serial program and a few representative inputs, and outputs an ordered list of the regions that are likely to be the most productive for the user to parallelize. Kremlin extends CPA to capture the impact of key parallelization constraints such as the number of available cores on the target, the ex-

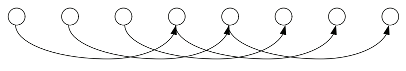
```
for (i=0; i<n; i++)
  A[i] = A[n-1-i]*B[i];
```



$$C_1 \times i + c_1 = C_1 \times (n-i-1) + c_1$$

```
//par
for(t1=1; t1<=intDiv(n-2,2); t1++)
{
 a[t1]=a[n-1-t1]*b[t1];
 if(2*t1 <= n-2 && t1 >= 1) {
  a[n-t1-1]=a[t1]*b[n-t1-1];
 }
}
```

```
for (i=0; i<n; i++)
  A[i+k] = A[i]*B[i];
```



$$\begin{cases} C_{11} \times i + c_1 = C_{11} \times (i-k) + c_1 \\ C_{12} \times i + c_1 = C_{11} \times (i+k) + c_1 \end{cases}$$

```
//par
for(t1=1; t1 <= min(n-k+1,k); t1++)
{
 a[t1+k]=a[t1]*b[t1];
 if (t1 >= 1) {
  for(t1_=t1+k; t1_<= n-1; t1_+= k)
  {
   a[t1_+k]=a[t1_]*b[t1_];
}}}
```

Fig. 1. Simple original loops and ISSF code exhibiting data dependences with iteration spaces, when n = 8 and k = 3.

ploitability of parallelism, and parallelization overhead.

The affine transformation framework (ATF), considered in papers [2, 3] unifies a large number of previously proposed loop transformations. The ATF framework is implemented in the project Pluto [10, 12]. Pluto transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling and fusion, but not limited to those. However, the affine transformation framework does not exploit all parallelism with synchronization-free slices in some cases of loops [4]. The tool does not parallelize the loops illustrated in Figure 1.

In order to find an affine transformation $\Phi_s = C_s I_s + c_s$ extracting parallelism for these examples, we have to solve the following equations in Figure 1 The solution in both cases is [0 c], for all positive integer c, means that there does not exist any affine transformation extracting two or more slices for these loops [4]. Dependences of these examples were described for the Omega Test in paper [20].

The polyhedral method was invented by Paul Feautrier [3] and implemented in the Automatic Parallelizer and Code Transformation Framework (PIPS) [22, 23]. PIPS is a source-to-source compilation framework for analyzing and transforming C and Fortran programs. Program transformations of the tool include loop distribution, scalar and array privatization, atomizers (reduction of a statements to a three-address form), loop unrolling (partial and full), strip-mining and others. The authors are going to develop code generation for mobile and embedded systems (Tilera, Kalray MPPA, ST P2012, EdkDSP) [22]. Notwithstanding this, the tool fails to extract parallelism with slices for some loops including the example from section 2 and the loops in Figure 3.

The Cetus tool provides an infrastructure for research on multi-core compiler optimizations that emphasizes automatic parallelization [21]. The compiler targets C programs and supports source-to-source transformations. It performs loop dependence analysis and generates parallel loop annotations. However, a loop parallelizer of the tool is limited only to induction variable substitution, reduction recognition and array privatization.

Parallelism extraction using Iteration Space Slicing

Iteration Space Slicing (ISS) was introduced by Pugh and Rosser [6] as an extension of a program slicing proposed by Weiser [7]. It takes dependence information as input to find all statement instances that must be executed to produce the correct values for the specified array elements. A dependence graph refers to extensive set of dependence of a loop nest, described by dependence relations in the Presburger arithmetic. The algorithms presented in paper [4] show the usage of the Iteration Space Slicing for coarse-grained parallelization. Coarse-grained code is presented with synchronization-free slices or with slices requiring occasional synchronization. An (iteration-space) slice is defined as follows.

Definition. Given a dependence graph defined by set of dependence relations, a slice S is a weakly connected component of this graph, i.e., a maximal subgraph such that for each pair of vertices in the subgraph there exists a directed or undirected path.

Iteration Space Slicing (ISS) requires an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects a dependence if and only if it actually exists. To describe and implement the algorithm, we chose the dependence analysis proposed by Pugh and Wonnacott [16] where dependences are represented by dependence relations.

A dependence relation is a tuple relation of the form [*input list*]$\rightarrow$[*output list*]: *formula*, where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples and *formula* describes the constraints imposed upon *input list* and *output list* and it is a Presburger formula built of constraints represented with algebraic expressions and using logical and existential operators.

Presburger arithmetic PA is the first-order theory of the integers in the language *L* having 0, 1 as constants, +,- as binary operations, and equality =, order $<$ and congruences $\equiv_n$ modulo all integers n$\geq$1 as binary relations. Standard operations on relations and sets are used, such as intersection ($\cap$), union ($\cup$), difference (-), domain (dom *R*), range (ran *R*), relation application ($S' = R(S)$: $e' \in S'$ iff exists *e* s.t. $e \rightarrow e' \in R, e \in S$), positive transitive closure of relation *R*, $R+ = \{[e] \rightarrow [e'] : e \rightarrow e' \in R \vee \exists e'', e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R+\}$, transitive closure $R^* = R+ \cup I$. In detail, the description of these operations is presented in [16, 13].

Let us to remind an ISS algorithm presented in [4]. It extracts coarse grained parallelism represented with slices

Algorithm Iteration Space Slicing for Program Loops Parallelization

```
Input : Set of relations $S=\{R_i\}$, $1 \leq i \leq n$, describing all dependences in a loop
Output : Code representing synchronization-free slices

1. R = $\bigcup_{i=1}^{n} R_i$
2. $S_{UDS}$ = domain(R) - range(R)
3. $R_{USC}$ = $\{[e] \rightarrow [e'] :\ e, e' \in S_{UDS},\ e \neq e',\ (R^*(e) \cap R^*(e'))\}$
4. $S_{repr}$ = $S_{UDS}$ - range($R_{USC}$)
5. $S_{slice}$ = $R^*((R_{USC})^*(e)$, $e \in S_{repr}$
6. Generate parallel code scanning synchronization-free
   slices by means of set $S_{slice}$ and a loop generator, for
   example the Omega Library [20] or the Barvinok tool [14].
```

and consists of the following steps:

- find set S_{repr} of representative sources as domain(R) - range(R);
- reconstruct slices from their representatives and generate code scanning these slices using S_{slice}.

The approach to extract synchronization-free slices relies on the transitive closure of an affine dependence relation describing all dependences in a loop. An ultimate dependence source is a source that is not the destination of another dependence. Ultimate dependence sources and destinations represented by relation R can be found by means of the following calculations: domain(R) - range(R). The set of ultimate dependence sources of a slice forms the set of its sources. The representative source of a slices is its lexicographically minimal source. The following listing includes steps of the ISS algorithm. More details can be found in paper [4].

Let us clarify the ISS technique by means of the following parametrized loop. Figure 2 presents dependences of the loop and two synchronization-free slices.

```
for(i=1; i<=N; i++)
  for(j=1; j<=N; j++)
     a[i][j] = a[i][j-1] + a[i-2][j-1];
```

There are the dependence relations returned by Petit [8]
$R_1 = \{[i,j] \rightarrow [i,j+1] : 1 \geqslant i \geqslant n \wedge 1 \geqslant j < n\}$.
$R_2 = \{[i,j] \rightarrow [i+2,j+1] : 1 \geqslant i \geqslant n\text{-}2 \wedge 1 \geqslant j < n\}$.
The following relation R_{USC} is calculated by means of the Omega calculator [13].
R_{USC} = {[i,j] → [i',j'] : Exists (alpha : 0 = i+i'+2alpha ∧ j = 1 ∧ j' = 1 ∧ 1 ⩾ i ⩾ i'-2 ∧ i' ⩾ n)}.
Next, the following sources of slices and elements of slices are produced by means of the Omega calculator.
S_{repr} = {[i,j] : j = 1 ∧ 1 ⩾ i ⩾ 2 ∧ 2 ⩾ n}.
S_{slice} = {[i,j] : Exists (alpha : i+t1+2alpha = 0 ∧ t1+2 ⩽ i ⩽ n ∧ 2 ⩽ j ⩽ n ∧ 1 ⩽ t1 ∧ t1 ⩽ 2) OR Exists (alpha : t1+i+2alpha = 0 ∧ j = 1 ∧ 1 ⩽ t1 ⩽ i-2 ∧ i ⩽ n ∧ t1 ⩽ 2) OR Exists (alpha : t1 = i+2alpha ∧ 1 ⩽ t1 ⩽ i ⩽ n ∧ 2 ⩽ j ⩽ n ∧ 2+i ⩽ 2j+t1 ∧ 1 ⩽ t1 ⩽ 2) OR j = 1 ∧ t1 = i ∧ 1 ⩽ t1 ⩽ 2}

Applying the algorithm for independent slices extraction [4] and *codegen* function from the Barvinok library [14], the following parallel code is generated:

```
if (n >= 2) {
  // parallel for
  for (i=1;i<=2;i++) {
    for (j=1;j<=n;j++) {
        a[i][j] = a[i][j-1]+a[i-2][j-1];
```

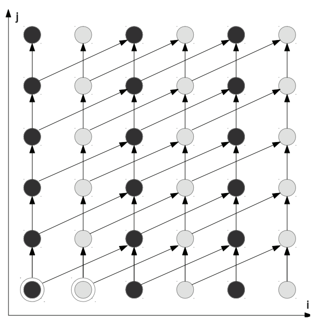


Fig. 2. Dependences of the loop example, when N=6.

```
      }
    if (i >= 1) {
      for (i_1=i+2;i_1<=n;i_1+=2) {
       for (j=1;j<=n;j++) {
         a[i_1][j]=a[i_1][j-1]+a[i_1-2][j-1];
         }
}}}}
```

Experiments

The presented technique was implemented in a tool which uses the Petit dependence analyser. The sources of programs in Java have been transformed by means of TRACO 0.1. TRACO includes the ISL library and the Omega Calculator framework for Presburger arithmetic calculation, Cloog for code generation. TRACO is designed to x86 and x86-64 architectures with Linux OS. It transforms code in C/C++ grammar (Java, C#). The output code of program loops is parallelized and transformed to multi-threaded applications for the Android OS [9].

Experiments were carried with an Google Nexus 5, processor: Qualcomm Snapdragon 800 2.3 GHz with 4 cores, 2 GB RAM, Android 4.4. The UTDSP Benchmark [18] and the NAS Parallel Benchmark (NPB 3.2) suites [17] were a subject the of experiments.

The first benchmark was created to evaluate the quality of code generated by a high-level language compiler targeting a programmable digital signal processor (DSP). The benchmark suite consists of six kernels and ten real-life applications for image processing and communication. The effectiveness of exploiting parallelism in kernels dominates the overall performance. In other words, the compiler must generate efficient code for kernels to maximize the utilization of the hardware resources in the model architecture. DSP application benchmarks are commonly used in embedded and mobile systems [19].

The loops of the NAS Parallel Benchmark suite are a small set of programs designed to help evaluate performance of parallel machines. The test suite, which is derived from computational fluid dynamics (CFD) applications, consists of five kernels and three pseudo-applications [17].

From 77 loops of the UTDSP benchmark suite, Petit is able to analyse 43 loops, and dependences were found in 34 loops (the rest 9 loops do not expose any dependence). For these loops, the presented approach is able to extract parallel threads for 18 (52,9%) loops.

From 431 loops of the NAS benchmark suite, Petit is able to analyse 257 loops, and dependences were found in 134 loops (the rest 123 loops do not expose any dependence). For these loops, the presented approach is able to extract parallel threads for 116 (86,5%) loops.

To assess the efficiency of code produced by the ISS, the following criteria were taken into account for choosing

Loop	Parameters	1 CPU (ms)	4 CPUs (ms)	S	E
Compress_2	B=300	5749	2467	2.330	0.582
	B=400	13822	5600	2.468	0.617
	B=500	29090	11090	2.623	0.655
Edge_detect_1	N=1500	1043	456	2.287	0.571
	N=2000	2402	683	3.516	0.879
	N=3000	3981	1309	3.041	0.760
Histogram_3	N=1000	416	112	3.714	0.928
	N=2000	1131	370	3.056	0.764
	N=2500	1745	766	2.278	0.569
FX_aux_fnct_2	N1,N2,N3=100	888	497	1.786	0.446
	N1,N2,N3=150	4871	1532	3.180	0.795
	N1,N2,N3=200	12093	4010	3.016	0.754
MG_mg_13	N1,N2=750	187	75	2.493	0.623
	N1,N2=1000	583	388	1.503	0.376
	N1,N2=1250	816	255	3.200	0.800
UA_diffuse_4	N1,N2,N3,N4=50	1265	540	2.343	0.586
	N1,N2,N3,N4=75	7747	1876	4.130	1.032
	N1,N2,N3,N4=100	32313	7002	4.615	1.154
UA_setup_16	N1,N2,N3=20	174	65	2.677	0.669
	N1,N2,N3=50	3622	1020	3.551	0.888
	N1,N2,N3=80	26673	6001	4.445	1.111
UA_transfer_4	N1,N2=1500	371	128	2.898	0.725
	N1,N2=2000	646	209	3.091	0.773
	N1,N2=2500	1128	404	2.792	0.698

Table 1. Time, speed-up and efficiency.

UTDSP and NAS loops: a loop must be computatively heavy (there are many benchmarks with constant upper bounds of loop indices, hence their parallelization is not justified), code produced by the algorithm must be parallel, structures of chosen loops must be different (there are many loops of a similar structure).

Applying these criteria, the following loops were selected:

- the UTDSP Benchmark Suite:
  - *Compress_2* - Image compression using discrete cosine transform (DCT),
  - *Edge_detect_1* - Edge detection using 2D convolution and Sobel operators,
  - *Histogram_3* - Image enhancement using histogram equalization.
- the NAS Parallel Benchmark:
  - *FT_auxfnct_2* - Fast Fourier Transform Benchmark,
  - *MG_mg_13* - Multigrid methods for solving differential equations,
  - *UA_diffuse_4*, *UA_transfer_4* and *UA_setup_16* - Unstructured Adaptive Benchmark.

To check the performance of parallel code, speed-up and efficiency are taken into account. Speed-up is the ratio of sequential time and parallel time, S=T(1)/T(P), where P is the number of processors. Efficiency, E=S/P, tells users about the usage of available processors by parallel code. Table 1 shows the times of loops execution (in milliseconds) for 1, 4 threads with speed-up and efficiency for three different numbers of iterations.

Figure 3 illustrates the speed-up for 2 and 4 CPUs in a graphical way. The results demonstrate that parallel loops formed on the basis of parallel code produced by the ISS framework i) permit for utilizing cores of the mobile multi-core processor; 2) speed-up occurs regardless of the number of loop iterations.

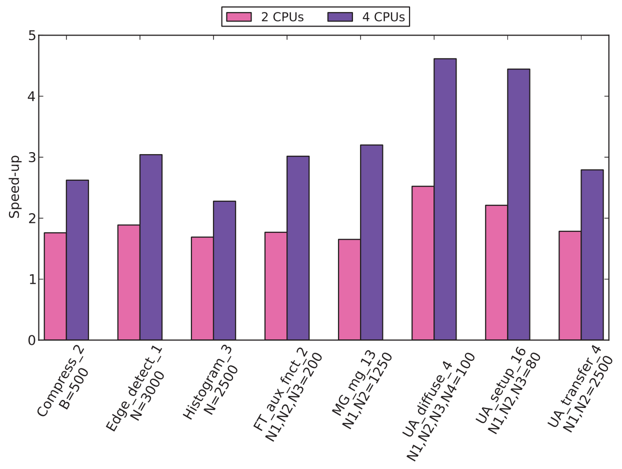


Fig. 3. Speed-up of the parallelized loops by means of the mobile and quad-core processor.

The power consumption of sequential and parallel program loops has been studied by means of the PowerTutor tool [27] which implements the Sesame mechanisms. Sesame is able to generate system energy models of 95% accuracy at one estimation per second and of 88% accuracy at one estimation per 10 ms. The PowerTutor has been chosen to carry out experiments because it does not requires an external assistance; it incurs low overhead and complexity (PowerTutor is intended to run when the system is being used); the tool achieves higher accuracy and rate than the battery interface and adapts to changes either in hardware or usage [27].

The results of experiments are presented in Table 2. The mobile processor consumes more power of its battery for two or four cores. However, the time period of computing is shorter. For *UA_diffuse_4*, *UA_setup_16* and

FX_aux_fnct_2, significant reduction of power consumption can be observed. It corresponds to good speed-up and longer time of computing. We expect that super-linear speed-up allows us to achieve better energy saving even more. Hence, we are going to study such transformations like loop tiling basing on a transitive closure operation for our compiler. Synchronization-free slices extraction with tiling will be developed in future.

Loop	Parameters	1 CPU (J)	4 CPUs (J)
Compress_2	B=500	4.979	4.665
Edge_detect_1	N=3000	0.603	0.550
Histogram_3	N=2500	0.407	0.284
FX_aux_fnct_2	N[1,2,3]=150	3.164	2.836
MG_mg_13	N[1,2]=1500	0.346	0.215
UA_diffuse_4	N[1,2,3,4]=100	26.177	17.009
UA_setup_16	N[1,2,3]=80	9.825	6.900
UA_transfer_4	N[1,2]=2500	0.465	0.381

Table 2. Power consumption.

Conclusion and future technique

The paper demonstrates that ISS algorithms extracts coarse-grained parallelism and generates code for mobile systems. The efficiency of the implemented tool was demonstrated on real-life benchmarks from typical embedded system application domains like e.g., audio-, image- and video-processing. Loops of the NAS and UTDSP benchmarks are divided on many slices which are mapped to cores of the ARM processor as threads. Coarse-grained parallelism advantage is no synchronization or occasional synchronization between threads. It allows users to achieve significant speed-up of parallel programs on mobile and memory-shared machines with multi-core processors.

Power saving is also possible. The power consumption is different because two units compute the problem instead of one unit. Time of computation is shorter. However, two units consume the power almost double. For other transformations like loop tiling, power reduction may correspond to speed-up significantly.

In the future, we indent to develop source-to-source tools of multi-threaded code generation for embedded devices by means of loop tiling based on the transitive closure operation. We consider improving locality of produced code for particular multi-core mobile systems with one and more processors. Furthermore, we would also use optimization loop techniques to achieve the most possible speedup for an application and less power consumption. The implementation of the ISS framework and studied examples can be found at the website `http://traco.sourceforge.net`.

REFERENCES

[1] Domeika M. : Software Development for Embedded Multi-Core Systems, A practical guide for using Intel embedded systems, Newnes, 2008.

[2] Lim, A., Lam, M., Cheong, G. : An affine partitioning algorithm to maximize parallelism and minimize communication. In ICS'99, ACM Press, pp. 228-237, 1999.

[3] Feautrier, P. : Some efficient solutions to the affine scheduling problem, part I and II, one and multidimensional time, International Journal of Parallel Programming 21, pp. 313-348 and pp. 389-420, 1992.

[4] Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K. : Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel Computing, 37, pp. 479-–497, 2011.

[5] Griebl, M., Feautier, P., Lengauer, C. : Index Set Splitting, International Journal of Parallel Programming, 28, pp. 607-631, 1999.

[6] W. Pugh, E. Rosser : Iteration space slicing and its application to communication optimization. In International Conference on Supercomputing: pp. 221–228, 1997.

[7] M. Weiser : Program slicing. In IEEE Transactions on Software Engineering, pp. 352–357, 1984.

[8] Kelly, W., Pugh, W., Maslov, V., Rosser, E., Shpeisman, T., Wonnacott, D. : New User Interface for Petit and Other Extensions. User Guide, 1996.

[9] Android Developers Guide - Processes and Threads : [web page] `http://developer.android.com/guide/components/processes-and-threads.html`, 2012. [Accessed on 31 Jun. 2014.].

[10] PLUTO - An automatic parallelizer and locality optimizer for multicores, [web page] `http://pluto-compiler.sourceforge.net`, 2011. [Accessed on 31 Jun. 2014.].

[11] Bondhugula, U., Baskaran, M., et al. : Affine transformations for communication minimal parallelization and locality optimization of arbitrarily-nested loop sequences, Lecture Notes in Computer Science, Volume 4959/2008, pp. 132–146, 2008.

[12] Bondhugula, U., Hartono, A., Ramanujan, J., Sadayappan, P. : A practical automatic polyhedral parallelizer and locality optimizer. ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08), 2008.

[13] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D. : The omega library interface guide. Technical report, College Park, MD, USA, 1995.

[14] Verdoolaege, S. : Barvinok: User Guide v. 035, [web page] `www.kotnet.org/~skimo/barvinok/barvinok.pdf`, 2011. [Accessed on 31 Jun. 2014.].

[15] Moldovan, D. : Parallel Processing: From Applications to Systems, Morgan Kaufmann Publishers, Inc, 1993.

[16] Pugh, W., 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.

[17] The NAS benchmark suite, [web page] `http://www.nas.nasa.gov`, 2012. [Accessed on 31 Jun. 2014.].

[18] Lee, C.G. : The UTDSP Benchmark Suite, [web page] `http://www.eecg.toronto.edu/~corinna`, 2002. [Accessed on 31 Jun. 2014.].

[19] Peng, S.H. : UTDSP: A VLIW Programmable DSP Processor, Graduate Department of Electrical and Computer Engineering, University of Toronto, 1999.

[20] WonnacottD. : A Retrospective of the Omega Project, Haverford College Computer Science Tech Report 2010.

[21] Chirag, D., Hansang, B., Seung-Jai, M., Seyong, L., Eigenmann, R., Midkiff, S., : Cetus: A Source-to-Source Compiler Infrastructure for Multicores, IEEE Computer, pp. 36–42, 2009.

[22] Amini, M., Ancourt, C., et al., : PIPS Documentation [web page] `http://pips4u.org/doc`, 2012. [Accessed on 31 Jun. 2014.].

[23] Amini, M., et al., : PIPS Is not (just) Polyhedral Software. In First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011). Chamonix, France, 4/2011, 2011.

[24] Cordes, D., Marwedel, P., Malik, A. : Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming, Proceeding CODES/ISSS '10 Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 267–276, 2010.

[25] Campanoni, S., et. al. : HELIX: automatic parallelization of irregular programs for chip multiprocessing, Proceeding CGO'12 Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 84–93, 2012.

[26] Garcia, S., et. al. : The Kremlin Oracle for Sequential Code Parallelization, Micro, IEEE, Volume: 32, Issue: 4, pp. 42–53, 2012.

[27] Dong, M. and Zhong, L. : Self-constructive, high-rate energy modeling for battery-powered mobile systems, in Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys), June 2011.

***Authors**: Ph.D. Marek Palkowski, Faculty of Computer Science, West Pomeranian University of Technology, 70210, Zolnierska 49, Szczecin, Poland, email: mpalkowski@wi.zut.edu.pl*