Marek PALKOWSKI [1],

West Pomeranian University of Technology, Faculty of Computer Science (1)

doi:10.15199/48.2015.02.31

Parallel Code Generation for Mobile Devices

Abstract. *Mobile computing is driven by pursuit of ever increasing performance. Multicore processing is recognized as a key component for continued performance improvements. 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 the ARM dual core processor. The related parallelization techniques are discussed, in particular for embedded systems. The future work is outlined.*

Streszczenie. *Przetwarzanie obliczeń za pomocą urządzeń mobilnych wiąże się z rosnącym zapotrzebowaniem na moc ich procesorów. Artykuł przedstawia zastosowanie narzędzia ISS (podziału przestrzeni iteracji pętli programowych) do wyznaczenia równoległego kodu dedykowanego dla urządzeń mobilnych (MID). Algorytmy pozwalają na wyznaczenie równoległości gruboziarnistej dla dowolnie zagnieżdżonych pętli i wygenerowanie wielowątkowego kodu dla systemu Android. Wyniki eksperymentalna dla zestawów pętli testowych NAS i UTDSP przeprowadzono wykorzystując dwurdzeniowy procesor ARM. Prace pokrewne i przyszłe zadania przedstawiono na końcu artykułu.*
Generowanie kodu równoległego dla urządzeń przenośnych

Keywords: automatic parallelization algorithms, synchronization-free parallelism, code generation, mobile computing, multi-core processing, DSP applications
Słowa kluczowe: algorytmy wyznaczające automatycznie równoległość. generowanie kodu, przetwarzanie mobilne, równoległość gruboziarniste, programowanie wielordzeniowe, aplikacje do przetwarzania sygnałów (DSP).

Introduction

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].

Using multiple cores in a single system enables to close the gap between energy consumption, problems concerning heat dissipation, and computational power. Multi-core processors offer developers the ability to apply more computer resources to a particular problem. The increasing use of multi-core microprocessors necessitates the increasing need to expose coarse-grained parallelism available in sequential algorithms.

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 [5-6]. Unfortunately, these techniques fail to parallelize loops in some cases [2]. 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 the ARM processor with two 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.

Parallelism extraction using Iteration Space Slicing

Iteration Space Slicing (ISS) was introduced by Pugh and Rosser [3] as an extension of a program slicing proposed by Weiser [4]. 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 [2] 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 1. Given a dependence graph, D, defined by a set of dependence relations, S, a *slice* is a weakly connected component of graph D, i.e., a maximal subgraph of D 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 [7] where dependences are represented by dependence relations.

A dependence relation is a tuple relation of the form {[*input list*] $\rightarrow$ [*output list*] : *constraints};* where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples and *constraints* is a Presburger formula describing constraints imposed upon *input list* and *output list*.

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≤1 as binary relations. Standard operations on relations and sets are used, such as intersection ($\cap$), union ($\cup$), difference (-), domain of relation (domain(*R*)), range of relation (range(*R*)), relation application (given a relation *R* and set *S*, *R*(*S*) = {[*e'*]: $\exists e \in S, e \rightarrow e' \in R$), positive transitive closure (given a relation *R*, $R^+ = \{[e] \rightarrow [e']: e \rightarrow e' \in R \,||\, \exists e'' \text{ s.t. } e \rightarrow e'' \in R \,\&\, e'' \rightarrow e' \in R^+\}$), transitive closure ($R^* = R^+ \cup I$, where *I* is the identity relation). These operations are described in detail in [11].

Let us to remind the ISS algorithm presented in [1]. The framework extracts coarse grained parallelism represented with slices consists of the following steps:

- find set S_{repr} of representative sources;
- 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 slice is its lexicographically minimal source. The following listing includes steps of the ISS algorithm. More details can be found in paper [1].

Algorithm: Iteration Space Slicing for Program Loops Parallelization

Input: Set of relations, S={R_i}, 1≤i≤n, describing all dependences in a loop

Output: Code scanning synchronization-free slices

1. $R = \bigcup_{i=1}^{n} R_i$
2. S_{UDS} = *domain(R) – range(R)*
3. $R_{USC} := \{[e] \rightarrow [e'] : e, e' \in S_{UDS}, e \prec e', R^*(e') \cap R^*(e)\},$
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 [19] or the Barvinok tool [13].

Let us clarify the ISS technique by means of the following parameterized loop. Figure 1 presents dependences of the loop and two synchronization-free slices.

Example 1

```
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 following dependence relations returned by Petit.

```
R1 = {[i,j]->[i,j+1] : 1<=i<= n && 1<=j< n},
R1 = {[i,j]->[i+2,j+1] : 1<=i<=n-2 && 1<=j<n}.
```

The following relation R_{USC} is calculated by means of the Omega calculator [7].

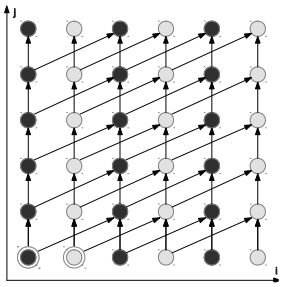

Fig.1. Dependences of the loop example, when N=6.

```
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 [2] and [12] function from the Barvinok library [13], 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];
  }
  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];
    }
}}}}
```

The presented technique was implemented in a tool by means of the Petit analyser. The input code of program loops is parallelized and transformed to multi-threaded applications for the Android OS [8].

The experiments were carried with the Samsung I9100 Galaxy S II, processor: Exynos 4210 1.2 GHz with 2 cores - ARM Cortex A9, 1 GB RAM, Android 4.0.3, kernel: 3.0.15. The UTDSP Benchmark Suite [17] and the NAS Parallel Benchmark (NPB 3.2) [16] were subject 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 loops of the NAS Parallel Benchmark 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 [16].

Table 1. Speed-up and efficiency of parallel loops

Loop	Parameters	1 CPU ms)	2 CPUs (ms)	S	E
Compress_2	B=300	5749	3540	1.624	0.812
	B=400	13822	7978	1.733	0.866
	B=500	29090	16532	1.760	0.880
Edge_detect_1	N=1500	1043	686	1.520	0.760
	N=2000	2402	1205	1.993	0.997
	N=3000	3981	2109	1.888	0.944
Histogram_3	N=1000	416	203	2.049	1.025
	N=2000	1131	675	1.676	0.838
	N=2500	1745	1033	1.689	0.845
FT_aux_fnct_2	N1,N2,N3=100	888	680	1.306	0.653
	N1,N2,N3=150	4871	2632	1.851	0.925
	N1,N2,N3=200	12093	6368	1.768	0.884
MG_mg_13	N1,N2=750	187	115	1.626	0.813
	N1,N2=1000	583	433	1.346	0.673
	N1,N2=1250	816	494	1.652	0.826
UA_diffuse_4	N1,N2,N3,N4=50	1265	745	1.698	0.849
	N1,N2,N3,N4=75	7747	3166	2.447	1.223
	N1,N2,N3,N4=100	32313	12808	2.523	1.261
UA_setup_16	N1,N2,N3=20	174	96	1.813	0.906
	N1,N2,N3=50	3622	1868	1.939	0.969
	N1,N2,N3=80	26673	12062	2.211	1.106
UA_transfer_4	N1,N2=1500	371	200	1.855	0.928
	N1,N2=2000	646	390	1.656	0.828
	N1,N2=2500	1128	632	1.758	0.892

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 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 seconds) for 1, 2 threads with speed-up and efficiency for three different numbers of iterations. Figure 2 illustrates the speed-up presented in Table 1 in a graphical way. The results in Table 1 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.

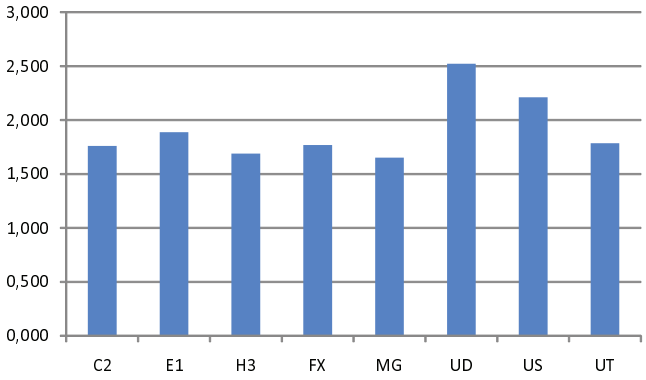

Fig.2. Speed-up of the parallelized loops by means of dual-core Exynos 4210 1.2 GHz

Related Work

The results of the paper are within the Iteration Space Slicing Framework (ISS) introduced by Pugh and Rosser [3]. 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.

An automatic parallelization of embedded software using hierarchical task graphs and integer linear programming (ILP) was presented in paper [23]. 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 [24] 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, signaling, 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 [20] 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.

The affine transformation framework (ATF), considered in papers [5-6] unifies a large number of previously proposed loop transformations. The ATF framework is implemented in the project Pluto [9-10]. 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 [2].

The polyhedric method was invented by Paul Feautrier [6] and implemented in the Automatic Parallelizer and Code Transformation Framework (PIPS) [21-22]. 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, loop interchange and others. The authors are going to develop code generation for mobile and embedded systems (Tilera, Kalray MPPA, ST P2012, EdkDSP) [21].

Conclusion

The paper demonstrates that the ISS algorithms extracts coarse-grained parallelism and generates code for mobile systems. The efficiency of the 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.

In the future, we indent to develop source-to-source tools of multi-threaded code generation for embedded devices. We consider improving locality of produced code for particular multi-core mobile systems. Furthermore, we would also like to combine this coarse grained approach with a finer grained loop level parallelization technique to achieve the most possible speedup for an application. The implementation of the ISS framework can be found at the website http://issf.sourceforge.net.

REFERENCES

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

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

[3] Pugh W.,Rosser E., Iteration space slicing and its application to communication optimization. In International Conference on Supercomputing: 221--228, (1997).

[4] Weiser, M., Program slicing. In IEEE Transactions on Software Engineering: 352--357, (1984).

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

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

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

[8] Android Developers Guide - Processes and Threads : http://developer.android.com/guide/components/processes-and-threads.html, (2012).

[9] PLUTO - An automatic parallelizer and locality optimizer for multicores, http://pluto-compiler.sourceforge.net, (2011).

[10] 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, 132-146, (2008).

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

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

[13] Verdoolaege, S. : Barvinok: User Guide v. 035, www.kotnet.org/~skimo/barvinok/barvinok.pdf, (2011).

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

[15] 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).

[16] The NAS benchmark suite, http://www.nas.nasa.gov, (2012).

[17] Lee, C.G. : The UTDSP Benchmark Suite, http://www.eecg.toronto.edu/~corinna, (2002).

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

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

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

[21] Amini, M., Ancourt, C., et al., : PIPS Documentation http://pips4u.org/doc, (2012).

[22] 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).

[23] 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).

[24] 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).

***Authors**: dr inż. Marek Pałkowski, Zachodniopomorski Uniwersytet Technologiczny, Katedra Inżynierii Oprogramowania, ul. Żołnierska 49, 71-210 Szczecin, E-mail: mpalkowski@wi.zut.edu.pl;*