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 zrównoleglanie gruboziarnistych zagnieżdzonych pętli programowych z parametrami do kodu wielowątkowego dla systemu Android. Eksperymenty przeprowadzono na zestawach pętli testowych (UTDSP i NPB) za pomocą czterorynkowego procesora ARM. Przedstawiono analizę wydajności i poboru mocy oraz pokrewnego rozwiązania. (Automatyczna ekstrakcja równoleżności dla urządzeń przenośnych)

Keywords: automatic parallelization algorithms, synchronization-free parallelism, code generation, mobile computing, multi-core processing, DSP applications, power consumption

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.

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 < 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 extract 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 multiprocessors. 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 sequential 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++)

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

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

---

**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 \([\text{input list}] \rightarrow [\text{output list}]\), 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 \(=, \leq, \geq, \equiv\) as binary relations. Standard operations on relations and sets are used, such as intersection (\(\cap\)), union (\(\cup\)), difference (-), domain (\(\text{dom}\)), range (\(\text{ran}\)), relation application (\(S^e = R(S)\); \(e \in S\) if \(e \in e \in R, e \in S\)), positive transitive closure of relation \(R^+ = \{e \rightarrow e' \mid e \rightarrow e' \in R \land (e' \rightarrow e' \in R^+)\}\), 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

Input: Set of relations $S_i = \{R_i\}$, $1 \leq i \leq n$, describing all dependences in a loop

Output: Code representing synchronization-free slices

1. $R = \bigcup_{i=1}^{n} R_i$
2. $S_{UDS} = \text{domain}(R) - \text{range}(R)$
3. $R_{USC} = \{\{e\rightarrow e'\} : e, e' \in S_{UDS}, e \neq e', (R^*(e) \cap R^*(e'))\}$
4. $S_{repr} = S_{UDS} - \text{range}(R_{USC})$
5. $S_{slice} = R^*((R_{USC})^*)^{-1}(e), e \in S_{repr}$
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 example, when $N=6$.

```
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]

$R_1 = \{[i,j] \rightarrow [i,j+1] : 1 \geq i \geq j \land 1 \geq j \land 1 \leq n \}$

$R_2 = \{[i,j] \rightarrow [i+2,j+1] : 1 \geq i \geq j \land 1 \geq j \land 1 \leq n \}$

The following relation $R_{USC}$ is calculated by means of the Omega calculator [13].

$R_{USC} = \{[i,j] \rightarrow [i',j'] : \text{Exists (alpha : 0 = i+2\alpha and } \land \land 1 \geq i \geq j \geq n \land 1 \geq i \geq j \geq n \}$

Next, the following sources of slices and elements of slices are produced by means of the Omega calculator [13].

$S_{repr} = \{[i,j] : 1 \geq i \geq j \land 1 \geq i \land 2 \geq n \}$

$S_{slice} = \{[i,j] : \text{Exists (alpha : i+2\alpha = 0 and } t1+2 \leq i \leq n \land 2 \leq j \leq n \land 1 \leq t1 \leq 2) \text{ OR Exists (alpha : t1+2\alpha = 0 and } i \leq j \leq n \land 1 \leq t1 \leq 2) \text{ OR Exists (alpha : t1 = i+2\alpha and } 1 \leq t1 \leq n \land 2 \leq j \leq n \land 2 \leq t1 \leq n \land 1 \leq i \leq n \land t1 \leq 2) \}$

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) {
    for (i=1; i<=n; 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 $N=6$.}

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 of the 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...
Table 1. Time, speed-up and efficiency.

UTDSP and NAS loops: a loop must be computationally 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, $S = \frac{T(1)}{T(P)}$, where $P$ is the number of processors. Efficiency, $E = \frac{S}{P}$, tells users about the usage of available processors by parallel code. Table 1 shows the times of loops execution (in milliseconds) for 1, 4 CPUs and 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; ii) speed-up occurs regardless of the number of loop iterations.

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 require 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_transfer_4 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.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Parameters</th>
<th>1 CPU (J)</th>
<th>4 CPUs (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress_2</td>
<td>B=500</td>
<td>4.979</td>
<td>4.685</td>
</tr>
<tr>
<td>Edge_detect_1</td>
<td>N=3000</td>
<td>0.603</td>
<td>0.550</td>
</tr>
<tr>
<td>Histogram_3</td>
<td>N=2500</td>
<td>0.407</td>
<td>0.284</td>
</tr>
<tr>
<td>FX_aux_fnct_2</td>
<td>N[1,2,3]=150</td>
<td>3.164</td>
<td>2.836</td>
</tr>
<tr>
<td>MG_mg_13</td>
<td>N[1,2]=1500</td>
<td>0.346</td>
<td>0.215</td>
</tr>
<tr>
<td>UA_diffuse_4</td>
<td>N[1,2,3,4]=100</td>
<td>26.177</td>
<td>17.009</td>
</tr>
<tr>
<td>UA_setup_16</td>
<td>N[1,2,3]=80</td>
<td>9.825</td>
<td>6.900</td>
</tr>
<tr>
<td>UA_transfer_4</td>
<td>N[1,2]=2500</td>
<td>0.465</td>
<td>0.381</td>
</tr>
</tbody>
</table>

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 intend 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

[28] Chen, J., et. al.: Automatic parallelization of multithreaded 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.

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