Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-Core Platforms [*]

Daniel Cordes, Michael Engel, Olaf Neugebauer, Peter Marwedel

TU Dortmund University
Dortmund, Germany
firstname.lastname@tu-dortmund.de

Abstract

Automatic parallelization of sequential applications is the key for efficient use and optimization of current and future embedded multi-core systems. However, existing approaches often fail to achieve efficient balancing of tasks running on heterogeneous cores of an MPSoC. A reason for this is often insufficient knowledge of the underlying architecture's performance.

In this paper, we present a novel parallelization approach for embedded MPSoCs that combines pipeline parallelization for loops with knowledge about different execution times for tasks on cores with different performance properties. Using Integer Linear Programming, an optimal solution with respect to the model used is derived implementing tasks with a well-balanced execution behavior. We evaluate our pipeline parallelization approach for heterogeneous MPSoCs using a set of standard embedded benchmarks and compare it with two existing state-of-the-art approaches. For all benchmarks, our parallelization approach obtains significantly higher speedups than either approach on heterogeneous MPSoCs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Design, Languages, Performance

Keywords Automatic Parallelization, Heterogeneity, MPSoC, Embedded Software, Integer Linear Programming, Pipeline

1. Introduction

Heterogeneity is expected to be one of the leading design principles for future embedded multi-core systems, since it offers promising approaches to solve upcoming problems especially in the areas of power, heat dissipation, reliability, and security. In embedded systems, however, usually there is neither support for parallel programming languages nor are inherently parallel problems prevalent like in many high-performance computing settings. Efficient use of the computing resources available in embedded multi-processor systems on chip (MPSoCs) thus requires approaches to extract parallelism from a given sequential description.

When targeting heterogeneous MPSoCs, the complexity to be handled by parallelization tools increases significantly. Whereas in homogeneous systems all processing units of an MPSoC can be considered identical regarding the execution time for a given piece of code, execution times that differ for separate processors or processor classes have to be taken into account in the heterogeneous case. Therefore, it is essential to *balance the execution time* of all tasks running in parallel in order to achieve the best possible utilization of computing resources. Currently available automatic parallelization methods for embedded MPSoC systems, however, either do not consider heterogeneous execution times at all, e.g., [19], or are only capable of extracting coarse-grained parallelism for heterogeneous systems [5]. In the first case, this results in a suboptimal use of processing resources, since the incorrect assumption of identical execution times on each processing unit leads to faster processing cores waiting for slower cores to complete their respective workload. In the second case, the amount of parallelism extracted can be suboptimal, which may result in the generation of only a small number of parallel tasks, e.g., using task-level parallelism. This, in turn, may leave some of the available cores unused.

In this paper, we present a parallelization approach that is able to extract more fine-grained parallelism for heterogeneous systems from sequential C programs. It makes use of the fact that a large number of embedded applications consist of multiple nested loops which, e.g., iterate over vectors or matrices of data. Examples for this class of applications are audio and video decoders and other signal processing applications like wireless baseband processing. Single iterations of these loops can only be executed in parallel if it is ensured that there are no loop-carried data dependencies between the loops' iterations. Our approach is able to consider these dependencies when extracting a set of parallel tasks, resulting in a program implementing pipeline parallelism. In contrast to previous solutions implementing such an approach, our method employs a cost model that allows incorporating differing execution times for loop iterations due to the underlying heterogeneous platform. This way, the balancing of parallel tasks that implement loop iterations can be achieved. Taken together, pipeline parallelism and consideration of execution time heterogeneity enable our parallelization approach to utilize each processing unit according to its performance characteristics in a system as well as the utilization of as many processing units as possible at the same time.

Selecting the appropriate combinations of tasks to be executed in parallel requires the consideration of two constraints. On the one hand, the execution time of each parallel task should be as similar to all other tasks' execution times as possible, while, on the other hand, the set of tasks to be executed in parallel is restricted

[*] Parts of the work on this paper have been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3. URL: http://www.sfb876.tu-dortmund.de

by the data dependencies inherent in the original sequential code. We solve this problem by applying an Integer Linear Programming (ILP)-based approach. While ILP-based approaches are known to be NP-hard in general, we can show that for a comprehensive set of benchmark programs we selected, the time required for analysis and parallelization allows for the integration of our parallelization tool in a typical embedded development tool flow.

To summarize, the main contributions of this paper are:

1. To the best of our knowledge, this is the first approach which uses Integer Linear Programming to exploit pipeline parallelism for embedded heterogeneous architectures.
2. In contrast to approaches from high-performance computing, our approach focuses on applications and restrictions of embedded systems.
3. Balancing tasks is essential for heterogeneous MPSoCs. To facilitate this, our approach integrates an appropriate cost model which enables automatic control of the granularity of the extracted parallelism. This allows tasks to be mapped to processing units with different performance characteristics.

The paper is structured as follows. Section 2 discusses related work. The general idea behind our parallelization approach is motivated in Section 3, followed by a description of the central data structure used, the Program Dependence Graph, in Section 4. Section 5 presents details of the ILP-based parallelization methodology used, augmenting the formal descriptions with graphical representations that visualize the different steps involved. Experimental results are discussed in Section 6, and Section 7 concludes the paper and gives an outlook to future research ideas.

2. Related Work

Many semi- and fully automatic parallelization techniques have been proposed in the last decades. All of them aim at simplifying the task of parallelizing sequentially written applications for (embedded) multi-core platforms. Early approaches optimize fine-grained instruction-level parallelism (see, e.g., [24]) for VLIW processors. However, recent architectures provide multiple cores on one die and require techniques which extract more coarse-grained thread-level parallelism. This kind of parallelism can be grouped into at least three categories, namely *task-level-*, *loop-level-* and *pipeline parallelism* that are discussed below.

A representative approach of *task-level parallelism* was presented by Hall et al. [8]. This technique automatically extracts task-level parallelism and is integrated into the SUIF parallelizing and optimizing compiler framework [9]. Ceng et al. published a semi-automatic parallelization assistant in [2]. Their approach transforms the application code into a weighted statement control data flow graph which is subsequently processed by a heuristically clustering algorithm. The algorithm generates tasks after several iterations and requires a user feedback loop to control the granularity of the parallelized program. Further approaches which extract this kind of parallelism were presented by Sarkar [20], Ottoni [15], and Nikolov et al. [14].

All approaches mentioned so far have in common that their applicability is limited to homogeneous architectures. Our previous publication [5] also extracts task-level parallelism but significantly differs from the other mentioned approaches since it considers heterogeneous architectures in the parallelization process. It could be shown that it is very important to take platform information of the heterogeneous target platform into account if applications should be parallelized for those architectures. However, the earlier presented approach is only able to extract coarse-grained task-level parallelism with limited potential of parallelizing loops, especially of those with loop-carried dependencies. Therefore, the approach presented in this paper focuses on the extraction of pipeline parallelism for heterogeneous architectures.

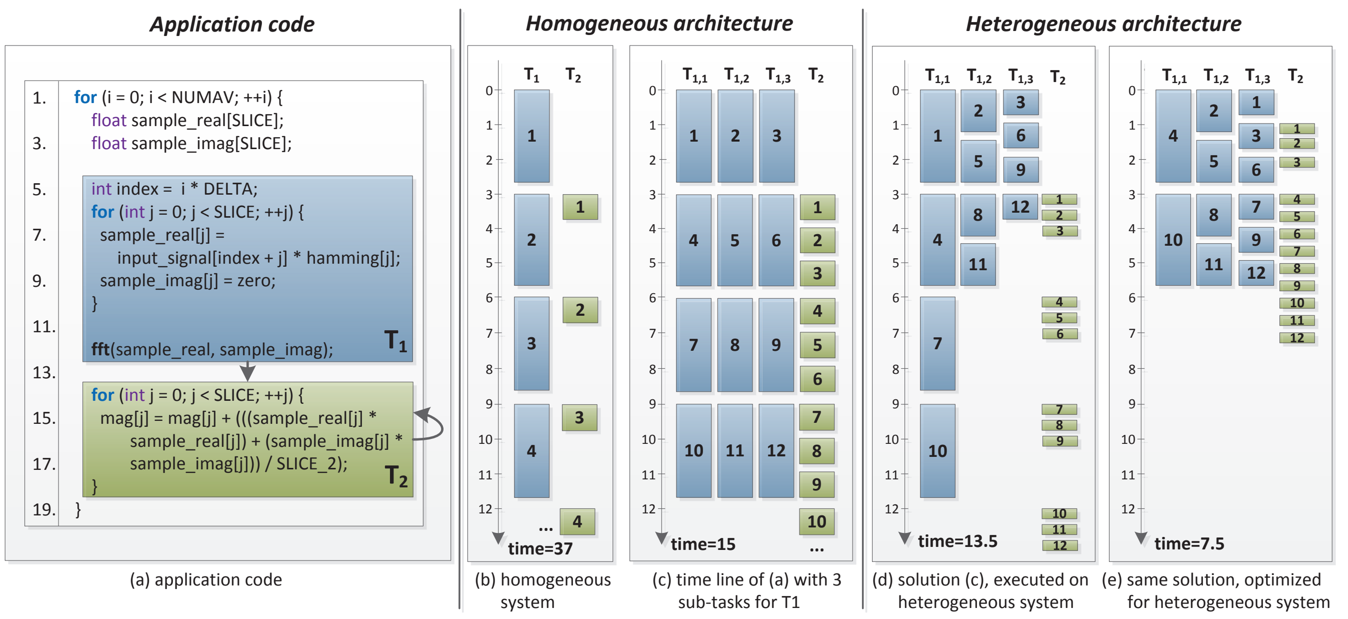
Approaches to extract fine-grained *loop-level parallelism* were developed by Chandra et al. and Franke. Chandra et al. [3] describe an approach which parallelizes loops of sequential applications for CC-NUMA (cache-coherent non-uniform memory access) architectures. Franke [6] presents an approach which is applicable for C applications on DSP architectures with multiple address spaces. Their approach also applies program code recovery techniques which enable more efficient dependency analyses. Another popular technique to extract fine-grained data-level parallelism from sequential applications is based on polytope models. Polytopes are used to represent loops of the application with its iteration space and dependencies of nested loops. Approaches using this technique have the drawback that they can only analyze affine loops. Thus, already existing applications can, in general, not be parallelized without manual code rewriting. Representative publications in this area were, e.g., published by Bondhugula et al. [1] and Lengauer [12].

The last mentioned category, *pipeline parallelism*, is most relevant for this work, since the presented approach of this paper aims at the extraction of this kind of parallelism for heterogeneous embedded architectures. Raman et al. [19], Tournavitis et al. [23] and our previous publication in [4] present different approaches to achieve this. The approaches are able to automatically split loops into different pipeline stages and further increase the application's performance by splitting pipeline stages into additional sub-tasks. Again, all these approaches have in common that they are optimizing for homogeneous platforms which makes it hard for them to balance the extracted tasks for heterogeneous architectures. This distinguishes them from the pipeline parallelization approach presented in this paper. Our new approach takes performance differences of heterogeneous architectures into account and is also less restrictive in the generation of sub-tasks. An additional approach in this area was presented by Liu et al. [13]. Their approach eliminates loop-carried dependencies by re-timing the execution of statements in a loop by moving executions of statements to earlier iterations of the loop. Thus, dependencies may change which creates the opportunity to parallelize different iterations of the loops. In addition, Gordon et al. [7] present a compiler framework which combines the extraction of task, data, and pipeline parallelism for applications written in the programming language StreamIt [22]. The designer has to extract tasks manually by defining independent actors connected by explicit data channels. The algorithms described in [7] search for parallelism in the given task structure. However, all described pipeline parallelization approaches are not optimized for heterogeneous architectures which drastically limits the achievable speedup of the parallelized applications.

To summarize, many previously published approaches have been presented but only the task-level parallelization approach presented in [5] takes performance differences of the available processing units of heterogeneous architectures into account. Unfortunately, many embedded applications are organized in a pipelined program structure so that pipeline parallelism is often able to extract more efficient parallelism than task-level parallelization methodologies. Even though automatic pipeline parallelization approaches exist (like, e.g., [19], [23], and [4]), none of these is optimized for heterogeneous architectures. The approach presented in this paper intends to fill this gap by introducing a new pipeline parallelization approach which considers differences in execution time between cores of heterogeneous embedded architectures.

3. Motivating Example

As already mentioned in the introduction, many embedded applications are structured in a pipelined manner. Dependencies between



Application code
Homogeneous architecture
Heterogeneous architecture
T1
T2
time=37
time=15
time=13.5
time=7.5
(a) application code
(b) homogeneous system
(c) time line of (a) with 3 sub-tasks for T1
(d) solution (c), executed on heterogeneous system
(e) same solution, optimized for heterogeneous system

```
1.   for (i = 0; i < NUMAV; ++i) {
       float sample_real[SLICE];
3.     float sample_imag[SLICE];

5.     int index =  i * DELTA;
       for (int j = 0; j < SLICE; ++j) {
7.       sample_real[j] =
           input_signal[index + j] * hamming[j];
9.       sample_imag[j] = zero;
       }
11.
       fft(sample_real, sample_imag);
13.
       for (int j = 0; j < SLICE; ++j) {
15.      mag[j] = mag[j] + (((sample_real[j] *
             sample_real[j]) + (sample_imag[j] *
17.          sample_imag[j])) / SLICE_2);
       }
19.  }
```

Figure 1. Pipeline parallelization example on homogeneous and heterogeneous architectures

these pipeline stages make it hard to extract efficient parallelism by applying, e.g., task-level or simple loop-level parallelization methodologies. Therefore, this section describes a motivating example, shows how embedded applications profit from pipeline parallelism and discusses capabilities and limitations of existing approaches.

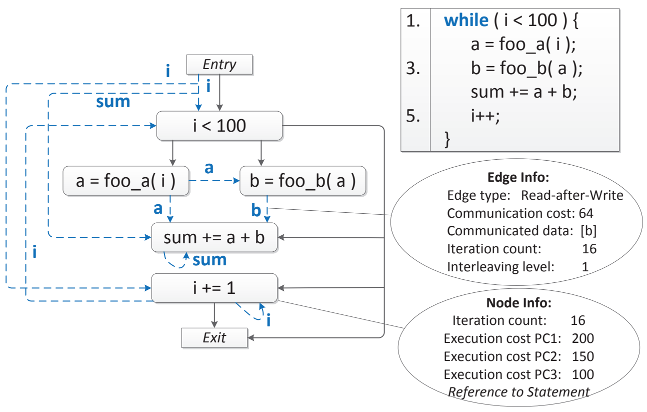
Figure 1 shows an example for the main computational loop of the spectral benchmark from the UTDSP benchmark suite [11]. It is a representative embedded application that calculates a power spectrum of an input speech sample. The application code is shown in Figure 1(a). The outer loop contains two inner loops and a call to an *fft* (Fast Fourier Transform) function between both loops. The second inner loop contains a loop-carried dependency to its previous iteration, since it reads from $mag[j]$ (in line 15) which was written in its previous iteration of the outer loop. Thus, it is not possible to execute all loop iterations in parallel. Instead, the statements of the loop are grouped into two disjunct pipeline stages T_1 and T_2, like shown in the figure. In this example, a data dependency exists between both pipeline stages, since T_2 processes data generated by T_1. Therefore, both tasks cannot be executed fully in parallel. The benefit of such a parallelization is that each task can start its next iteration of the loop as soon as it has communicated the generated data to the tasks waiting for its output.

The timing behavior of the two pipeline stages is visualized in Figure 1(b) assuming that each task is executed by a separate processing unit. As can be seen, pipeline stage T_1 starts its first iteration at time t_0. At the end of iteration 1 it sends its output data to stage T_2. Now, the first iteration of T_2 can be executed in parallel to the second iteration of T_1 and so on. Thus, the extracted tasks are executed in a pipelined manner. In this example we assume that the loop has 12 iterations. With the given timing behavior of 1(b), the application would be accelerated to use 37 instead of 48 time units to execute the loop with two extracted pipeline stages.

The solution shown in Figure 1(b) has two disadvantages. First, pipeline stage T_2 is a lot faster than T_1 which results in long idle phases for T_2 and an unbalanced execution behavior. In addition, most architectures provide more than only two cores. Thus, more tasks should be extracted to circumvent these problems and gain additional performance. One possibility is to divide the loop into more pipeline stages. However, the number of such stages is often limited due to high communication costs. Instead, sub-tasks of pipeline stages can be extracted which execute different iterations of the stages in parallel. Figure 1(c) shows an example, in which pipeline stage T_1 is split into three subtasks, so that iterations 1, 2 and 3 as well as 4, 5 and 6, etc. are executed in parallel. In this example, T_2 obtained a sufficient amount of data to execute its loop iterations continuously without waiting for T_1 after the first data has arrived. As can be seen, the solution is well balanced for a homogeneous architecture with four cores. Approaches which are able to extract such a solution were presented in, e.g., [19] and [4]. However, many embedded devices are heterogeneous and contain several cores with differing performance characteristics.

Figure 1(d) shows what may happen if solution (c) was mapped to such a heterogeneous architecture with different performance characteristics of the available processing units. Most existing parallelization tools do not possess any information about the targeted architecture and are, in addition, not aware of those performance differences. Rather, they would have to assume a homogeneous architecture. This leads to a very unbalanced timing behavior, as can be seen in Figure 1(d). Task $T_{1,3}$ has executed all mapped iterations in 4 time units, while $T_{1,1}$ needs 12 time units since it is mapped on a slower processing unit. Thus, the cores executing $T_{1,2}$ and $T_{1,3}$ are idle for a long time which drastically reduces the performance of the extracted solution. In addition, task T_2 also often has to wait for data. Solution (c), executed on the homogeneous architecture, needs 15 time units, while the execution of solution (d) took 13.5 time units. This shows that the potential of the accelerated cores cannot be used by parallelization approaches which are not optimized for heterogeneous architectures.

To circumvent this problem, the approach presented in this paper handles different execution times for all statements of the application (and thus, also for created tasks) depending on the mapped processing unit. In addition, the approach performs a premapping of tasks to processor classes, representing identical processing types of the heterogeneous architecture. Another difference is the way how the approaches map iterations to subtasks. The previously published approaches map all iterations with a different offset to the same subtask. In contrast, the approach presented in this



```
while ( i < 100 ) {
    a = foo_a( i );
    b = foo_b( a );
    sum += a + b;
    i++;
}
```

Figure 2. Program Dependence Graph

paper freely maps iterations to subtasks. As can be seen in Figure 1(e), $T_{1,3}$ executes iterations $\{1, 3, 6, 7, 9, 12\}$, while task $T_{1,2}$ executes $\{2, 5, 8, 11\}$. This freedom of decision highly increases the complexity of the solution space but enables the extraction of well-balanced tasks. All subtasks of T_1 finish at the same time and provide task T_2 with input data in an optimized way which reduces the execution time from 13.5 to 7.5 time units. Thus, compared to the previously published approaches, our new approach is the first one which takes advantage of heterogeneous architectures for the extraction of pipeline parallelism. It should be mentioned here, that the extraction of pipeline stages and the mapping of iterations to subtasks is done at the same time, so that the algorithm will not remain in a local optimum. We demonstrate this in the result section by comparing our results to the homogeneous state-of-the-art pipeline parallelization approach presented in [4]. The results show that our new approach is able to significantly outperform the existing one on heterogeneous architectures.

4. Program Dependence Graph

The pipeline parallelization approach for embedded heterogeneous architectures presented in this paper employs an augmented program dependence graph (PDG) as intermediate representation. Each loop of the application is transformed into a PDG which combines both control- and data-dependencies. An example of a PDG for a small code snippet is depicted in Figure 2. The graph contains one node for each statement of the considered loop, one entry and one exit node. Control flow dependencies are visualized by solid directed edges while data dependencies are represented by dashed ones. Even for this small example many dependencies exist which demonstrates that it is nontrivial to detect pipeline parallelism.

The presented approach of this paper is based on Integer Linear Programming (ILP) and uses a clearly defined mathematical model to evaluate the possible improvement achieved by a solution candidate. Therefore, necessary information like, e.g., estimated execution costs and iteration counts of the statements, are required and annotated to the nodes of the graph (cf. *Node Info* in Figure 2). Since processing units may differ in their performance characteristics in heterogeneous systems, execution costs are extracted and annotated for each processor class PC_i (representing identical processing units), separately. Thus, the approach is able to evaluate the benefit of a parallel solution depending on the mapping of tasks to processor classes which is also extracted by the presented approach.

At the edges of the graph, the edge type, communication costs, the communicated data, the iteration count and the interleaving level – describing the minimal amount of loop iterations which can be executed before the data is consumed by the target node – are also annotated (cf. *Edge Info* in Figure 2). The annotated data is automatically extracted by target platform simulation. By combining the graphical representation with additional cost information, all necessary information is available to extract well-balanced pipeline parallelism for heterogeneous MPSoCs.

```
 1: function DoParallelization(IR $ir$, Platform $pf$)
 2:     $loops \leftarrow$ CollectParallelizableLoops($ir$)
 3:     $sol \leftarrow \emptyset$
 4:     for $l \in loops$ do
 5:         $sol \leftarrow sol \cup$ Parallelize($l, pf$)
 6:     end for
 7:     CombineBestResults($loops, sol, pf$)
 8: end function
 9:
10: function Parallelize(Loop $l$, Platform $pf$)
11:     $loopPDG \leftarrow$ ConstructPDG($l$)
12:     $result \leftarrow$ SimpleParallelizer($loopPDG, pf$)
13:     if $result = \emptyset$ then
14:         $result \leftarrow$ ILPParallelizer($loopPDG, pf$)
15:     end if
16:     $solutions \leftarrow \{result, sequentialSolution\}$
17:     return $solutions$
18: end function
```

Figure 3. Parallelization algorithm

5. Parallelization Methodology

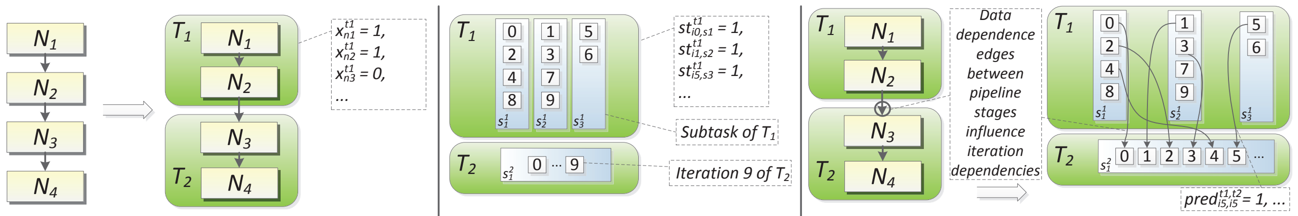
This section gives a global overview of the parallelization algorithm in Section 5.1 followed by a formal definition of the Integer Linear Programming-based pipeline parallelization approach in Section 5.2. Finally, a simple loop parallelization methodology is presented in Section 5.3 which is combined with the ILP-based approach to extract parallelism from loops without loop-carried dependencies in a less computational intensive way.

5.1 Parallelization Algorithm

The overall structure of the presented pipeline parallelization approach for embedded heterogeneous architectures is shown in Figure 3. The algorithm starts with the function DoParallelization in line 1. An intermediate representation (IR) [17] of the application's source code as well as a description of the target platform are processed as input. The target platform description [18] contains information about, e.g., the number and performance characteristics of the available processing units and interconnects and also of the memory subsystems.

First of all, a call to the function CollectParallelizable Loops traverses the IR of the application in line 2 and returns a list of loops which may profit from pipeline parallelism. Afterwards, the function Parallelize is called (line 5) for each loop in isolation to extract pipeline parallelism. The solutions are collected and the best combination of parallelized loops is determined and implemented by the function CombineBestResults. The approach may of course implement parallelism for more than only one loop.

The function Parallelize, starting in line 10, is called for each loop in isolation. First, a PDG is created in line 11 which contains only those nodes which are part of the loop(-nest) to be processed. Since the complexity of the ILP-based pipeline parallelization approach is very high, a call to SimpleParallelizer (described in Section 5.3) tries to parallelize the loop in a less complex way which can only be applied if the loop does not contain any loop-carried dependencies. If a solution can be found, it is combined with the sequential version of the loop and returned as its solution candidates. Otherwise, the more complex ILP-based parallelization approach presented in Section 5.2 is started and its result is returned together with the sequential version of the loop. The



(a) 5.2.1 Pipeline Stage Mapping

(b) 5.2.2 Iterations of Pipeline Stages to SubtaskMapping

(c) 5.2.3 Definition of Predecessor Relationships (Data and Control Flow Dependencies)

Figure 4. Graphical representation of the ILP-based pipeline parallelization approach (part 1)

sequential version of each loop is always added, so that the call to COMBINEBESTRESULTS at the end of the algorithm always has the option to use the sequential version, if another loop increases the overall performance in a more efficient way.

5.2 ILP-based pipeline parallelization approach

Integer Linear Programming (ILP) is a well-known technique that is often used for partitioning problems. Even though ILP is NP-hard, solutions can be determined very efficiently by commercial as well as by open source solvers for many real-world problems. This section defines the ILP-based pipeline parallelization approach for embedded heterogeneous architectures which is called in line 14 of Figure 3. The approach covers five main targets:

I) Extract different pipeline stages by mapping statements of the loop's body into disjunctive stages (see Figure 1(a,b)).

II) Divide pipeline stages into subtasks which execute different iterations of the stages in parallel (see Figure 1(c,d)).

III) Keep track of dependencies which may change if statements are moved from one pipeline stage to another one or if iterations of pipeline stages are mapped to different subtasks.

IV) Create a mapping of tasks to processor classes of the targeted embedded heterogeneous architecture (see Figure 1(e)).

V) Minimize execution costs by taking into account task creation, communication, and task execution costs which depend on the processor class a task is mapped to.

In the following paragraphs, decision variables are written in lower case letters, sets start with a capital letter and constants contain exclusively capital letters. Indices n and o are used for nodes of the PDG, i and j are used for iterations of the loop to be parallelized, t and u represent indices for pipeline stages, while c represents a processor class and s is used for concurrently executed subtasks of a pipeline stage. A graphical representation of most equations is also given in Figures 4-6. The sub-figures have the same name as the corresponding subsections which describe the equations.

5.2.1 Pipeline Stage Mapping

Target (I) of the ILP-based pipeline parallelization approach is a mapping of PDG nodes to pipeline stages. To perform this, decision variable x_n^t is defined in Equation 1.

$$x_n^t = \begin{cases} 1, & \text{if node } n \text{ is mapped to pipeline stage } t \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

The constraint defined in Equation 2 ensures that every child node (representing statements of the loop to be parallelized) is mapped to exactly one pipeline stage.

$$\forall n \in Nodes: \sum_{t \in Stages} x_n^t = 1 \tag{2}$$

5.2.2 Iterations of Pipeline Stages to Subtask Mapping

Target (II) of the ILP-based pipeline parallelization approach is a mapping of loop iterations of the created pipeline stages to concurrently executed subtasks. This is expressed by decision variable $subtask_{i,s}^t$ which is defined in Equation 3.

$$subtask_{i,s}^t = \begin{cases} 1, & \text{if iteration } i \text{ of pipeline stage } t \\ & \text{is mapped to subtask } s \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

To be compliant with the original program semantics, the ILP has to take care that each loop iteration (NI = number of loop iterations) is executed exactly once for each pipeline stage, which is ensured by Equation 4.

$$\forall t \in Stages: \forall i \in \{0,..,NI–1\}: \\ \sum_{s \in SubTasks^t} subtask_{i,s}^t = 1 \tag{4}$$

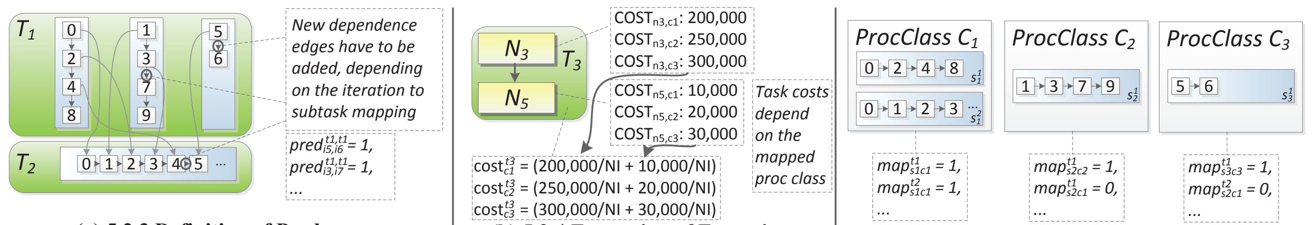
5.2.3 Definition of Predecessor Relationships

The main objective of the pipeline parallelization approach is the reduction of the execution time by moving statements of the loop's body into disjunctive pipeline stages. The loop iterations of each stage can also be executed concurrently in different subtasks. In order to minimize the execution time, the critical or most expensive path from the *entry* to the *exit* node of the loop's PDG (cf. Figure 2) has to be extracted. Therefore, the algorithm has to define predecessor/successor relationships between different pipeline stages which depend on data and control flow dependencies of the child nodes as well as on the mapping of loop iterations to subtasks (cf. Target (III)). Equation 5 defines decision variable $pred_{i,j}^{t,u}$, which is created for all pipeline stages t and u in iterations i and j.

$$pred_{i,j}^{t,u} = \begin{cases} 1, & \text{if pipeline stage } t \text{ in iteration } i \text{ is pre-} \\ & \text{decessor of pipeline stage } u \text{ in iteration } j \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

Data and Control Flow Dependencies Up to now, decision variables $pred_{i,j}^{t,u}$ are defined to express the predecessor/successor relationship between different pipeline stages. Now, constraints have to be added which take care that the decision variables evaluate to 1, iff pipeline stage t in iteration i is a predecessor of pipeline stage u in iteration j. One situation which leads to such a predecessor relationship relies on data and control flow dependencies of the mapped child nodes. Thus, if a data or control flow edge from node n to node m exists and both nodes are mapped to different pipeline stages, the latter one has to wait until the first one has completed its execution. This is ensured by Equation 6.

$$\forall t,u \in Stages: \forall n,m \in Nodes: \\ \forall i \in \{0,..,NI–1\}: \forall j \in \{i,..,NI–1\}: n \neq m: \\ EDGE_{n,m,j-i} = 1: pred_{i,j}^{t,u} \geq (x_n^t \wedge x_m^u) \tag{6}$$



T_1
T_2
New dependence edges have to be added, depending on the iteration to subtask mapping
$pred^{t1,t1}_{i5,i6} = 1$, $pred^{t1,t1}_{i3,i7} = 1$, ...
N_3
N_5
T_3
COST$_{n3,c1}$: 200,000
COST$_{n3,c2}$: 250,000
COST$_{n3,c3}$: 300,000
COST$_{n5,c1}$: 10,000
COST$_{n5,c2}$: 20,000
COST$_{n5,c3}$: 30,000
Task costs depend on the mapped proc class
cost$^{t3}_{c1}$ = (200,000/NI + 10,000/NI)
cost$^{t3}_{c2}$ = (250,000/NI + 20,000/NI)
cost$^{t3}_{c3}$ = (300,000/NI + 30,000/NI)
ProcClass C_1
ProcClass C_2
ProcClass C_3
s^1_1
s^2_1
s^1_2
s^1_3
$map^{t1}_{s1c1} = 1$, $map^{t2}_{s1c1} = 1$, ...
$map^{t1}_{s2c2} = 1$, $map^{t1}_{s2c1} = 0$, ...
$map^{t1}_{s3c3} = 1$, $map^{t2}_{s2c1} = 0$, ...

(a) 5.2.3 Definition of Predecessor Relationships (Subtask Dependencies)

(b) 5.2.4 Extraction of Execution Costs of Pipeline Stages

(c) 5.2.5 Mapping of Subtasks to Processor Classes

Figure 5. Graphical representation of the ILP-based pipeline parallelization approach (part 2)

The predecessor variable $pred^{t,u}_{i,j}$ is created for all possible pipeline stage and loop iteration combinations. This way, for all combinations of nodes, it is checked if node n is part of pipeline stage t while node m has to be part of pipeline stage u. If this is true and a directed edge from n to m exists with an interleaving level of $j - i$, denoted by $EDGE_{n,m,j-i} = 1$, pipeline stage u depends on t for iterations i and j. If there exists an edge from, e.g., node n to m in iteration 1 and 3, the interleaving level is 2. From a technical perspective, it should be mentioned that the constant $EDGE_{n,m,j-i}$ is known when the ILP is created. Therefore, constraints are only generated if an edge between n and m with the matching interleaving level exists.

The $\wedge$ operator used in Equation 6 can be modeled by the following constraints.

$$z = (x \wedge y) \in \{0, 1\}$$
$$z \geq x + y - 1, \qquad z \leq x, \qquad z \leq y \tag{7}$$

Subtask Dependencies In addition to data and control flow dependencies, dependencies between different iterations of the same pipeline stage have to be considered as well. They depend on the loop iterations to subtask mapping. If, e.g., iterations 1 and 3 of pipeline stage t are mapped to the same subtask, both iterations are executed sequentially since a subtask cannot evaluate different iterations in parallel. Thus, iteration 1 of pipeline stage t is a predecessor of iteration 3 of pipeline stage t. Those dependencies are created for all iteration combinations of the different pipeline stages like defined in Equation 8.

$$\forall t \in Stages : \forall i \in \{0, .., NI{-}1\} : \forall j \in \{i + 1, .., NI{-}1\} :$$
$$pred^{t,t}_{i,j} \geq subtask^t_{i,s} \wedge subtask^t_{j,s} \tag{8}$$

5.2.4 Extraction of Execution Costs of Pipeline Stages

To calculate path costs, execution costs of the different pipeline stages have to be determined. In the homogeneous case, it would be sufficient to sum up the execution costs of the nodes which are added to the pipeline stage. Since it is important to consider different performance characteristics of the different processing units if an application should be parallelized for heterogeneous architectures, execution costs should depend on the processor class executing the given pipeline stage. This is done in Equation 9 which creates one cost variable $cost^t_c$ for all pipeline stages t, executed on a processor class c.

$$\forall c \in ProcClasses : \forall t \in Stages :$$
$$cost^t_c \geq \sum_{n \in Nodes} x^t_n * (COST_{n,c}/NI) \tag{9}$$

This variable contains costs $COST_{n,c}$ of all nodes n mapped to the corresponding pipeline stage t for the execution of one iteration on processor class c. The overall execution costs of each node are distributed in equal parts over the iterations NI of the loop. This saves several decision variables, since the ILP does not have to distinguish between different execution costs of pipeline stages in different iterations. The execution costs of node n on processor class c are annotated at the nodes of the PDG and are automatically extracted by the framework as described in [4].

5.2.5 Mapping of Subtasks to Processor Classes

Variable $cost^t_c$ contains the execution costs of one iteration of pipeline stage t if it is executed on processor class c. But, up to now, pipeline stages and their subtasks are not mapped to any processor classes. This is not necessary for homogeneous architectures, but has a huge impact on the execution time for heterogeneous ones. Therefore, the presented approach of this paper combines the extraction of parallelism with a mapping of subtasks to processor classes to create well-balanced solutions like demanded by Target (IV). This mapping is implemented by decision variable $map^t_{s,c}$ defined in Equation 10.

$$map^t_{s,c} = \begin{cases} 1, & \text{if subtask } s \text{ of pipeline stage } t \\ & \text{is mapped to processor class } c \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Each subtask s of pipeline stage t has to be mapped to exactly one processor class c so that it is executed exactly once which is ensured by Equation 11.

$$\forall t \in Stages : \forall s \in SubTasks^t :$$
$$\sum_{c \in ProcClasses} map^t_{s,c} = 1 \tag{11}$$

5.2.6 Used pipeline stages

The algorithm has the capability to extract as many pipeline stages as processing units are available. Nevertheless, task creation and communication costs as well as different performance characteristics of the available processing units may result in the ILP extracting fewer tasks if such a solution leads to a higher reduction of the overall execution time. Thus, some of the pipeline stages may not be used which can be evaluated by decision variable $stageused^t$ defined in Equation 12.

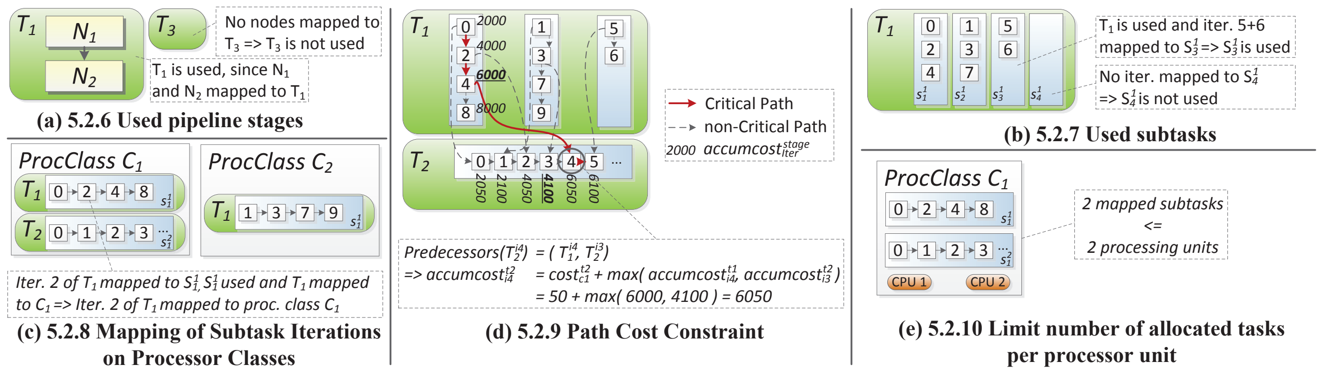
$$stageused^t = \begin{cases} 1, & \text{if pipeline stage } t \text{ is used} \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

Pipeline stage t is used if at least one node n is mapped to it, like defined in Equation 13.

$$\forall n \in Nodes : \forall t \in Stages :$$
$$stageused^t \geq x^t_n \tag{13}$$

5.2.7 Used subtasks

Like already mentioned, the approach offers the possibility to extract as many subtasks s for a pipeline stage t as processing units are available. The extraction of subtasks directly influences the overall execution costs since task creation costs are added for each



No nodes mapped to T3 => T3 is not used
T1 is used, since N1 and N2 mapped to T1
(a) 5.2.6 Used pipeline stages
ProcClass C1
ProcClass C2
Iter. 2 of T1 mapped to S1, S1 used and T1 mapped to C1 => Iter. 2 of T1 mapped to proc. class C1
(c) 5.2.8 Mapping of Subtask Iterations on Processor Classes
Critical Path
non-Critical Path
2000 accumcost
Predecessors(T2) = (T1, T2)
=> accumcost = cost + max(accumcost, accumcost)
= 50 + max(6000, 4100) = 6050
(d) 5.2.9 Path Cost Constraint
T1 is used and iter. 5+6 mapped to S3 => S3 is used
No iter. mapped to S4 => S4 is not used
(b) 5.2.7 Used subtasks
2 mapped subtasks <= 2 processing units
CPU 1
CPU 2
(e) 5.2.10 Limit number of allocated tasks per processor unit

Figure 6. Graphical representation of the ILP-based pipeline parallelization approach (part 3)

created subtask. To determine the amount of created subtasks, a decision variable $subtaskused_s^t$ is created for each subtask s of pipeline stage t like shown in Equation 14.

$$subtaskused_s^t = \begin{cases} 1, & \text{if subtask } s \text{ of pipeline stage } t \text{ is used} \\ 0, & \text{otherwise} \end{cases} \tag{14}$$

Subtask s of pipeline stage t is used if at least one iteration i is mapped to it and pipeline stage t itself is used like ensured by Equation 15.

$$\forall t \in Stages : \forall s \in SubTasks^t : \forall i \in \{0,..,NI{-}1\} : \\ subtaskused_s^t \geq subtask_{i,s}^t + stageused^t - 1 \tag{15}$$

5.2.8 Mapping of Subtask Iterations on Processor Classes

Up to now, only a relation between subtasks and their mapped processor classes exists, like defined in Equation 10. For the calculation of the overall execution costs the relation between iteration i of pipeline stage t and its mapped processor class is also necessary. Therefore, Equation 16 defines decision variable $iterOnPC_{i,c}^t$.

$$iterOnPC_{i,c}^t = \begin{cases} 1, & \text{if iteration } i \text{ of pipeline stage } t \\ & \text{is mapped to processor class } c \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

Iteration i of pipeline stage t is mapped to processor class c if it is part of subtask s and s is mapped to processor class c if subtask s is really used. Equation 17 evaluates to one if this is true.

$$\forall t \in Stages : \forall c \in ProcClasses : \\ \forall s \in SubTasks^t : \forall i \in \{0,..,NI{-}1\} : \\ iterOnPC_{i,c}^t \geq subtask_{i,s}^t + map_{s,c}^t + subtaskused_s^t - 2 \tag{17}$$

5.2.9 Path Cost Constraint

Based on the knowledge of the execution costs of each pipeline stage depending on the mapped processor class and the mapping of loop iterations to processor classes, it is now possible to describe the accumulated costs of the possible paths to determine the overall execution time. Equation 18 defines $accumcost_j^t$ and ensures that it contains the execution costs of all executed predecessors as well as the execution costs of the pipeline stage's iteration itself.

$$\forall t, u \in Stages : \forall c \in ProcClasses : \forall i \in \{0,..,NI{-}1\} : \\ \forall j \in \{i,..,NI{-}1\} : pred_{i,j}^{u,t} = 1 \wedge iterOnPC_{j,c}^t = 1 \Rightarrow \\ accumcost_j^t \geq cost_c^t + accumcost_i^u + commcost_u \tag{18}$$

Equation 18 ensures that the path costs $accumcost_j^t$ for pipeline stage t in iteration j are at least as large as the costs $cost_c^t$ for the execution of one iteration of pipeline stage t itself executed on processor class c and the path costs of its most expensive predecessor $accumcost_i^u$, including all communication costs $commcost_u$ of pipeline stage u. Preconditions like $pred_{i,j}^{u,t} = 1$ can be modeled in ILPs like, e.g., shown in [5]. The accumulated costs are included in the objective function, so that it is automatically minimized by the ILP solver.

5.2.10 Limit number of allocated tasks per processor unit

A platform is equipped with a limited number of processing units. By taking advantage of platform information in the parallelization step, it is possible to avoid additional scheduling overhead. Therefore, each processing unit should execute only one subtask of a pipeline stage in our model at a time. Thus, the constant number of available processing units $NUMPROCS_c$ of a processor class c must be at least as high as the number of mapped subtasks $map_{s,c}^t$ if they are used $subtaskused_s^t$. This is ensured by Equation 19.

$$\forall c \in ProcClasses : \sum_{t \in Stages} \sum_{s \in SubTasks^t} \\ map_{s,c}^t \wedge subtaskused_s^t \leq NUMPROCS_c \tag{19}$$

5.2.11 Objective Function

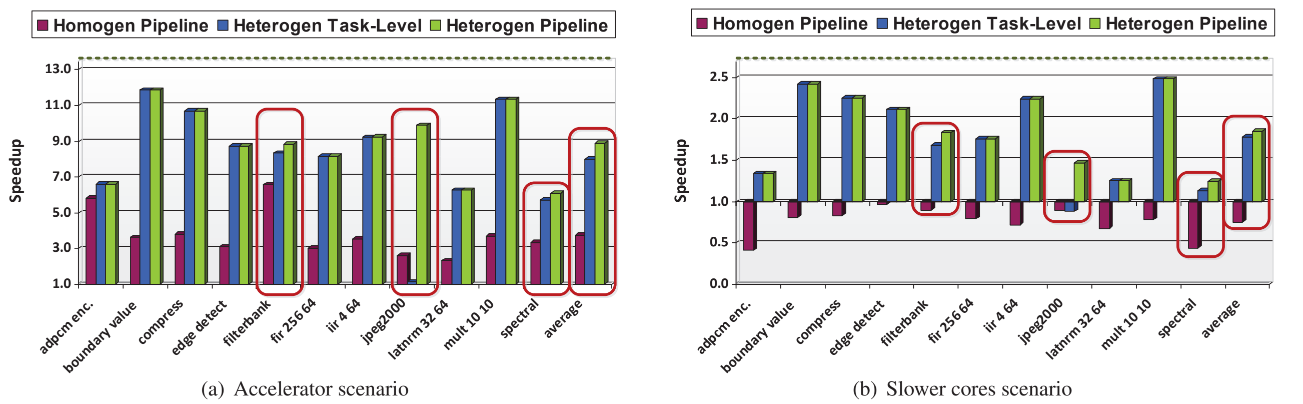
With all decision variables and constraints defined, it is now possible to describe the objective function. As mentioned before, the most expensive execution path from the *entry* to the *exit* node of the loop's PDG should be minimized like defined by Target (V). Thus, additional constraints are added which statically set the *entry* node to be a predecessor of all pipeline stages. The *exit* node will be a successor of all pipeline stages, respectively. With the help of all defined constraints, it is easy to create the objective function, like shown in Equation 20.

$$\text{minimize} \quad numtasks * \text{TASKOVERHEAD} + \\ accumcost_{exit} \tag{20}$$

The variable $numtasks$ contains the number of extracted subtasks used. Since the creation of such tasks increases the execution time, a constant task creation overhead, multiplied with the number of created subtasks, is added to the objective value.

$$numtasks = \sum_{t \in Stages} \sum_{s \in SubTasks^t} subtaskused_s^t \tag{21}$$

The task creation overhead can be defined in the platform description together with a communication cost factor. By defining these platform dependent parameters, it is easy to adapt the cost model of the ILP to different architectures. The value of the objective function is equivalent to the execution time of the parallelized loop on the targeted heterogeneous architecture. It is hence returned



(a) Accelerator scenario

(b) Slower cores scenario

Figure 7. Results for platform configuration (A): 100/250/500/500 MHz

together with the node-to-pipeline-stage mapping, the mapping of the stages' loop iterations to subtasks and the mapping of subtasks to the processor classes of the targeted heterogeneous platform as result of the parallelization step.

5.3 Simple Loop Parallelization Approach

The ILP-based pipeline parallelization approach described in the previous section automatically balances extracted tasks from loops of embedded applications and also combines it with a mapping of those tasks to processor classes of an embedded heterogeneous architecture. Due to the complexity of the problem's solution space, the algorithm may need a long time to find a good solution. Therefore, a fast but simplified algorithm is executed by the parallelization approach first (cf. Section 5.1) which just divides the different iterations of the loop into concurrently executed tasks. The generated results are a special kind of pipeline parallelism (and could also be extracted by the ILP-based approach) and contain only one pipeline stage divided into several subtasks. This approach can only be applied if the loop does not contain loop-carried dependencies since dependencies between different iterations would sequentialize the execution in such a case.

An example for a loop with 80 iterations, parallelized for a platform with four different cores is given in Table 1. In a first step, the approach calculates how long each processing unit takes to execute one iteration of the loop (cf. column *Exec time*). This time includes the time to execute one loop iteration on the specific processor as well as task creation and communication costs for the task created. Based on those execution times, a factor is calculated (cf. *Factor*) which denotes the number of iterations executed on processing unit c while one iteration is executed on the slowest processing unit:

$$Factor_c = \max_{p \in Processors}\{ExecTime_p\}/ExecTime_c \quad (22)$$

In a third step, the percentage of executed loop iterations is calculated by dividing each factor by the sum of all factors (cf. *Percentage*):

$$Percentage_c = (Factor_c / \sum_{p \in Processors} Factor_p) * 100 \quad (23)$$

Finally, the iterations of the loop are distributed to the different CPUs, depending on the calculated percentages (cf. *Iterations*). If the sum of assigned iterations is less than the number of loop iterations, the remaining iterations are assigned to the fastest processing units (numbers in brackets). Thus, the number of loop iterations assigned to a processing unit is automatically balanced according to the processing unit's performance characteristics.

CPU	Exec time	Factor	Percentage	Iterations
CPU 1	1,153,280	5.13	38.61	30 (+1)
CPU 2	1,281,280	4.62	34.76	27 (+1)
CPU 3	2,332,160	2.54	19.11	15
CPU 4	5,920,320	1.00	7,52	6
Sum	-	13.29	100.0	78 (80)

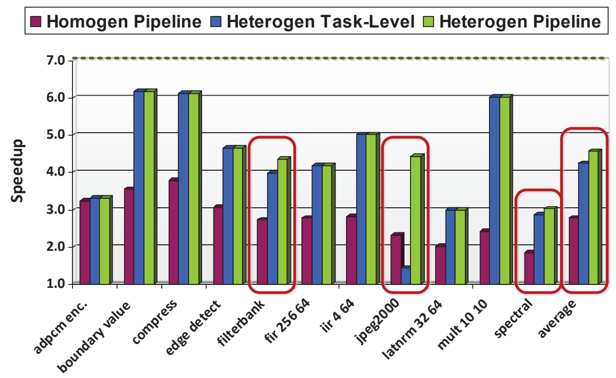
Table 1. Simple Loop Parallelization Approach Example

By combining this simple but fast loop parallelization approach with the rich but complex ILP-based one the framework presented in this paper is able to extract efficient pipeline parallelism for embedded heterogeneous architectures as shown in the next section.

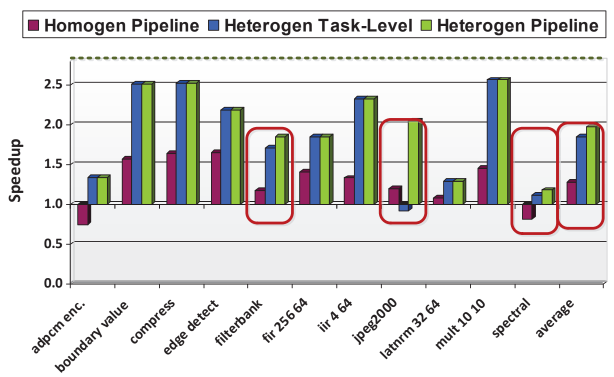
6. Experimental Results

To highlight the efficiency of the solutions generated by our new automatic pipeline parallelization approach for embedded heterogeneous architectures, we present results from the UTDSP benchmark suite [11] containing representative real-world embedded applications. In addition, we also evaluated other meaningful embedded applications like, e.g., a JPEG encoder which is often applied in embedded systems. To emphasize the quality of our new approach, we compare the extracted results with two existing state-of-the-art pipeline parallelization tools. The first one is a pipeline parallelization approach [4] which is optimized for homogeneous architectures. The second one extracts a different kind of parallelism (task-level parallelism) but is optimized for heterogeneous architectures [5].

The heterogeneous embedded target platform was simulated with the cycle-accurate CoMET simulator [21]. Even though our parallelization approach would also perform well for different instruction sets and specialized processing units since it uses different execution costs for each statement, we have chosen same-ISA multi-core platforms [10] for evaluation purposes. They are used in emerging products, like, e.g., ARM's big.LITTLE platform [16] or NVIDIA's Tegra 3. To emphasize the adaptability of our approach to various architectures, we present results for at least two platform configurations. Platform configuration (A) contains four ARM cores running at 100 MHz (1×), 250 MHz (1×), and 500 MHz (2×). This configuration shows that our approach works well for architectures with large performance variances. All cores are connected to a level-two cache on a high performance bus to enable fast memory accesses for shared data. Platform configuration (B) contains two 200 MHz and two 500 MHz cores to simulate a performance discrepancy of approximately 2.5×. This is also the average performance difference of ARM's big.LITTLE platform [16] with two Cortex-A7 and two Cortex-A15 cores.



(a) Accelerator scenario



(b) Slower cores scenario

Figure 8. Results for platform configuration (B): 200/200/500/500 MHz

6.1 Evaluation of Speedup

We evaluated the presented platforms for two application scenarios: (I) The main processor of the platform is the slowest one (e.g. 100 MHz) and the additional cores are added as accelerators. (II) The main processor of the platform is the fastest one (e.g. 500 MHz) and the other (slower) processors are added to the platform due to, e.g., power or thermal issues. The measurement baseline in both scenarios is the sequential execution on the main processor for all evaluated approaches. Figure 7 depicts results for both evaluation scenarios for platform configuration (A) (100/250/500/500MHz) and compares our new heterogeneous parallelization approach to the ones presented in [4] and [5]. The dashed line shows the theoretical maximum speedup of the considered target platforms.

Results for platform configuration (A) and the accelerator scenario (I) are shown in Figure 7(a) on the previous page. As can be seen, all three approaches increase the performance of all evaluated applications well. The homogeneous approach [4] uniformly balances the workload for all available processors. Thus, a speedup between $3\times$ up to $4\times$ is achieved for most applications which is very good for a homogeneous architecture with four processing units. However, results generated by [5] and our new heterogeneous pipeline parallelization approach are much more impressive. They automatically balance the extracted tasks by respecting different performance characteristics of the available processing units. Thus, the two processors with 500 MHz are automatically allocated with heavier workloads than the slower ones. This results in performance increases of up to 11-12$\times$ for some of the benchmarks (e.g., *boundary value*, *compress* and *mult*) which significantly outperforms the homogeneous pipeline parallelization tool [4] and is very close to the theoretical maximum speedup of $13.5\times$ [1]. However, even if the approach presented in [5] extracts comparable speedups to our newly presented pipeline parallelization approach of this paper for most applications, it is outperformed for three of the considered benchmarks (*filterbank*, *jpeg*, *spectral*). The highest difference was observed for the JPEG encoder. Here, even the homogeneous pipeline parallelization tool extracted a more efficient parallel solution ($2.6\times$) than the heterogeneous task-level approach (only $1.1\times$). This shows that for some embedded applications pipeline parallelism is most efficient. Thus, the technique presented in this paper is able to extract a speedup of nearly $10\times$ which significantly outperforms both existing ones. On average, the homogeneous tool increased the applications' performance by $3.8\times$ while the heterogeneous task-level one reached speedups of on average $8\times$. In contrast, our new approach reached an average speedup of nearly $9\times$.

Figure 7(b) shows results for scenario (II). Here, the speedup produced by the homogeneous approach is slower than one which means that the parallelized application performs slower than its sequential version. The reason is that the homogeneous approach uniformly distributes the work to the available processing units. Thus, the fast main processor has to wait until the slower processing units have finished their tasks. Instead, the heterogeneous parallelization approaches were able to speed up the applications by generating tasks that perfectly utilize the slower processing units so that all cores finish nearly at the same time. Again, the speedup of the applications *filterbank*, *jpeg*, and *spectral* could be increased by our new approach which outperforms both existing ones. It should also be mentioned, that our new approach has extracted a higher speedup for all evaluated benchmarks in both application scenarios compared to the existing pipeline parallelization approach. In addition, it performed better for more than one quarter of all evaluated benchmarks compared to the efficient heterogeneous task-level parallelization tool and otherwise never generated slower solutions.

These observations could also be confirmed for platform configuration (B) with two 200 MHz and two 500 MHz cores (cf. Figure 8). The performance difference between all three approaches is less than the performance difference for platform configuration (A) since the theoretical speedup limit is lower for platform configuration (B). Nevertheless, the relation between increased speedups and the theoretical speedup limit is comparable between both platform configurations. Here, the three already mentioned applications *filterbank*, *jpeg*, and *spectral* profit most from our new heterogeneous pipeline parallelization approach. The average speedup for scenario (I) is $2.8\times$, $4.2\times$ and $4.6\times$ for the evaluated approaches with a theoretical speedup limit of $7\times$ of the platform. The speedup of Scenario (II) with the slower additional cores is at $1.3\times$, $1.8\times$ and $2\times$ with a theoretical limit of $2.8\times$, respectively.

To summarize, the following results could be achieved:

1. Our newly presented heterogeneous pipeline parallelization approach is able to utilize heterogeneous platforms in an excellent way (speedups of up to 11-12$\times$ were measured).
2. The integration of mapping decisions and platform information in a heterogeneous parallelization approach is highly beneficial.
3. Our new approach outperformed two existing state-of-the-art parallelization approaches. For the JPEG encoder, a speedup of nearly $10\times$ could be measured compared to $2.6\times$ and $1.1\times$.

6.2 Execution Time

Of course, those results do not come for free. The newly presented ILP-based pipeline parallelization approach for heterogeneous ar-

[1] $(1 * 100 + 1 * 250 + 2 * 500\text{MHz})/100\text{MHz} = 13.5\times$

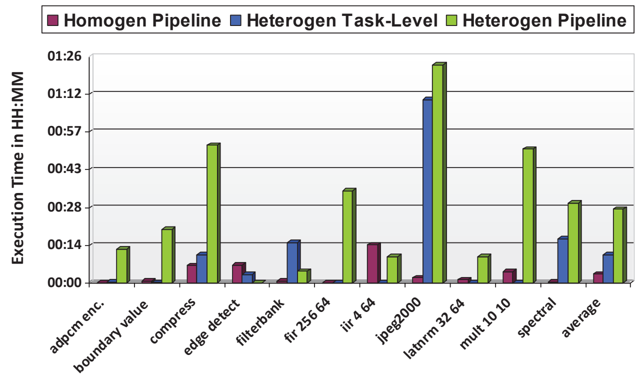


Figure 9. Exec-time of parallelization approaches in HH:MM

chitectures has the longest execution time compared to the other approaches. The time to parallelize the evaluated benchmarks with the three opposed approaches is visualized in Figure 9[2]. The timings shown are those which were necessary to extract the parallel solution for platform configuration (A) with evaluation scenario (I).

As can be seen, the homogeneous pipeline parallelization approach performs faster for most benchmarks than both other approaches while the heterogeneous task-level approach performs faster for most of them than our newly presented heterogeneous pipeline parallelization approach. This is due to the fact that the complexity of the solution space is increasing between all three approaches from left to right. Nevertheless, the quality of the solution also increases with the complexity of the respective approach. On average, the homogeneous pipeline parallelization approach took around 3 minutes to parallelize the considered applications. But most of them could be processed in less than a minute. In contrast, the heterogeneous task-level parallelization approach took 10 minutes on average to parallelize the applications. The newly presented heterogeneous pipeline parallelization approach processed the considered applications in 28 minutes on average while most applications were parallelized in around 10 minutes. One possibility to counteract the higher execution times is to parallelize the parallelization approach itself. Since all loops are processed in isolation first (cf. Figure 3), the approach is highly parallelizable which can significantly reduce the overall execution time. Nevertheless, the high speedups outweigh higher execution times in most cases and are acceptable since parallelization has to be done only once in the compilation process.

7. Conclusions and Future Work

To the best of our knowledge, this paper presents the first approach which combines automatic extraction of pipeline parallelism with mapping decisions to efficiently balance created tasks for embedded heterogeneous MPSoCs. The efficiency of the tool was demonstrated using several real-world benchmarks from typical embedded application domains. The measurements, performed on a cycle-accurate MPSoC simulator, have shown that our new approach outperforms two existing state-of-the-art parallelization approaches. Speedups of up to 11-12× could be measured for some of the considered benchmarks on a heterogeneous embedded platform with a theoretical speedup limit of 13.5×.

In the future we would like to extend our approach to take other objectives into account as well, like, e.g., energy consumption or code sizes, to be able to create even more efficient code for embedded MPSoCs.

[2] Measured on an AMD Opteron core running at 2.4 GHz

Acknowledgments

The authors would like to thank Synopsys for the provision of the MPSoC simulator CoMET.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, et al. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of PLDI*, 2008.

[2] J. Ceng, J. Castrillon, W. Sheng, et al. MAPS: an integrated framework for MPSoC application parallelization. In *Proc. of DAC*, 2008.

[3] R. Chandra, D.-K. Chen, et al. Data distribution support on distributed shared memory multiprocessors. *ACM SIGPLAN Notices*, 1997.

[4] D. Cordes, A. Heinig, P. Marwedel, et al. Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming. In *Proc. of ICPADS*, 2011.

[5] D. Cordes, M. Engel, O. Neugebauer, et al. Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs. In *In Proc. of PSTI*, 2013.

[6] B. Franke and M. O'Boyle. Compiler parallelization of C programs for multi-core DSPs with multiple address spaces. In *Proc. of CODES+ISSS*, 2003.

[7] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of ASPLOS-XII*, 2006.

[8] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, et al. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proc. of Supercomputing*, 1995.

[9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, et al. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12), 1996.

[10] R. Kumar, D. M. Tullsen, P. Ranganathan, et al. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. of ISCA*, 2004.

[11] C. G. Lee. UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/`, 2013.

[12] C. Lengauer. Loop Parallelization in the Polytope Model. In *CONCUR '93, Lecture Notes in Computer Science 715*. Springer-Verlag, 1993.

[13] D. Liu, Z. Shao, M. Wang, et al. Optimal loop parallelization for maximizing iteration-level parallelism. In *Proc. of CASES*, 2009.

[14] H. Nikolov, M. Thompson, T. Stefanov, et al. Daedalus: Toward composable multimedia MP-SoC design. In *Proc. of DAC*, 2008.

[15] G. Ottoni, R. Rangan, A. Stoler, et al. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proc. of MICRO 38*, 2005.

[16] Peter Greenhalgh, ARM. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. `http://www.arm.com/files/downloads/big.LITTLE_Final.pdf`, 2013.

[17] R. Pyka and J. Eckart. ICD-C Compiler framework. `http://www.icd.de/index.php/en/es/icd-c-compiler/icd-c`, 2013.

[18] R. Pyka et al. Versatile System-level Memory-aware Platform Description Approach for Embedded MPSoCs. In *Proc. of LCTES*, 2010.

[19] E. Raman, G. Ottoni, A. Raman, et al. Parallel-stage decoupled software pipelining. In *Proc. of CGO*, 2008.

[20] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 1991.

[21] Synopsys. CoMET, Virtual Prototyping Solution. `http://www.synopsys.com`, 2013.

[22] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of CC*. Springer, 2002.

[23] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proc. of PACT*, 2010.

[24] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4), 1991.