2011 IEEE 17th International Conference on Parallel and Distributed Systems

Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming[1]

Daniel Cordes, Andreas Heinig, Peter Marwedel
TU Dortmund University
Dortmund, Germany
firstname.lastname@tu-dortmund.de

Arindam Mallik
Imec Belgium
Leuven, Belgium
arindam@imec.be

***Abstract*—The complexity and performance requirements of embedded software are continuously increasing, making Multiprocessor System-on-Chip (MPSoC) architectures more and more important in the domain of embedded and cyber-physical systems. Using multiple cores in a single system reduces problems concerning energy consumption, heat dissipation, and increases performance. Nevertheless, these benefits do not come for free. Porting existing, mostly sequential, applications to MPSoCs requires extracting efficient parallelism to utilize all available cores. Many embedded applications, like network services and multimedia tasks for voice-, image- and video processing, are operating on data streams and thus have a streaming-based structure. Despite the abundance of parallelism in streaming applications, it is a non-trivial task to split and efficiently map sequential applications to MPSoCs. Therefore, we present an algorithm which automatically extracts pipeline parallelism from sequential ANSI-C applications. The presented tool employs an integer linear programming (ILP) based approach enriched with an adequate cost model to automatically control the granularity of the parallelization. By applying our tool to real-life applications, it can be shown that our approach is able to speed up applications by a factor of up to 3.9x on a four-core MPSoC architecture, compared to a sequential execution.**

***Index Terms*—Automatic Parallelization, Embedded Software, Program Dependence Graph, Pipeline Parallelism, MPSoC, Cyber-Physical Systems**

I. Introduction

Recent years have shown a dramatic increase in the complexity of software for embedded and cyber-physical devices, resulting in a demand for more computational power. To satisfy these requirements, Multiprocessor System-on-Chip (MPSoC) architectures are gaining more and more importance in this domain. Compared to single-core platforms, MPSoCs can execute applications with reduced core frequencies by distributing the computations to multiple processing units. This results in lower energy consumption and less heat dissipation. Unfortunately, these benefits do not come for free.

Nowadays, most embedded applications are written in sequential C-code. Thus, to exploit the full advantages of MPSoC platforms, applications have to be split up into several concurrent tasks to enable parallel execution. In contrast to instruction level parallelism, task level parallelism has shifted the job of splitting up an application to the user. Since manual parallelization tends to be very error prone and time consuming, the designer should be relieved from the burden of manually parallelizing an application.

By analyzing the embedded market, it is noticeable that many embedded applications have a streaming-based structure. This is due to the fact that a large part of embedded devices execute applications in the domain of networking services, voice- and image processing as well as multimedia tasks like video decoding. All these applications have in common that most of their parallelism is hidden in loops containing different pipelining-based jobs. Thus, parallelization tools should be able to find this kind of parallelization, if they are optimizing embedded applications in this domain.

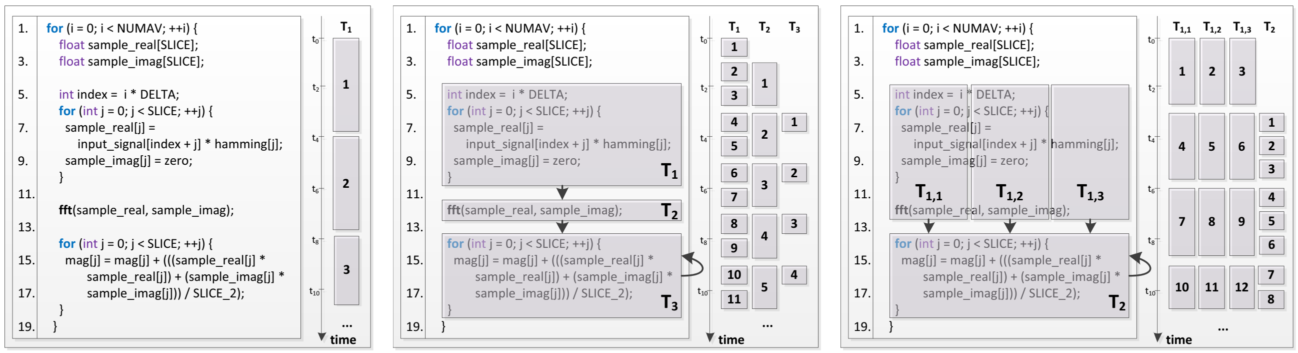
Despite the abundance of parallelism in streaming-based applications, it is a nontrivial task to find an efficient mapping of sequential applications to MPSoCs. Quite often, the gain of concurrent execution is overshadowed by too expensive communication costs between the created tasks. Thus, it is very important to obtain precise knowledge about execution and communication costs, which is neglected by most existing parallelization tools. Otherwise, the parallelization could even lead to a decrease of the application's performance.

Based on these observations, we developed a novel, fully automated loop parallelization methodology, which is extracting different pipeline stages from applications written in ANSI-C to split them efficiently into separate tasks. The algorithm is also able to split parts of these pipeline stages into additional concurrently executed tasks to further decrease the overall execution time. Our approach is based on integer linear programming (ILP), so that the results are optimal with respect to the underlying model. The ILP-based approach is enriched with an adequate cost model, so that it can estimate whether it is beneficial to distribute parts of the application to different processing units or not. To highlight the applicability for embedded and cyber-physical systems, results for embedded real-life applications are presented, executed on a cycle-accurate MPSoC simulator.

The main contributions of this paper are as follows:

1) To the best of our knowledge, this is the first approach which uses integer linear programming to combine the exploitation of pipeline parallelism with additional task splits in (nested) loops.
2) In contrast to the high-performance computing commu-

[1]The research leading to these results has received funding from the European Community's MNEMEE project as part of the Framework Programme FP7 under grant agreement no 216224 (Informatik Centrum Dortmund).



```
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.  }
```

(a) Sequential code (b) Horizontal loop split (c) Horizontal and vertical loop split

Fig. 1. Pipelining based loop level parallelization example

nity, this approach focuses on applications and restrictions of embedded and cyber-physical systems.

3) Our approach is enriched with an adequate cost model. This enables automatic control of the granularity of the extracted parallelism.

The rest of this paper is organized as follows: Section II describes the kind of parallelism extracted by our ILP-based approach. The program dependence graph (PDG), which is used as intermediate representation, is described in Section III. The parallelization step itself is presented in Section IV followed by a short overview of our tool flow in Section V. Section VI will then present experimental results for real-life applications. Afterwards, Section VII gives a survey of related work, before the paper is summarized in Section VIII.

II. Motivating example

This section describes the pipeline parallelism, which is extracted from flat or nested loops of applications by the tool presented in this publication. The code snippet shown in Figure 1(a-c) represents the nested main computational loop of the *spectral* benchmark [1]. The application calculates the power spectrum of an input sample of speech. This benchmark was chosen as example, since it has a representative structure for pipeline-based embedded applications. On the left hand side of each figure, the application's source code and the way the application is split up into separate tasks is shown. The right hand side demonstrates the time at which the iterations of the tasks are executed.

Due to the fact that the second inner loop starting in line 14 is reading elements of the *mag* array, written in the previous iteration of the outer loop, loop-carried dependencies exist between different iterations of the outer loop. Thus, it is not possible to split the complete outer loop into concurrently executed tasks. Instead, pipeline parallelism is splitting the loop into different tasks, which are executing disjunct parts, like visualized in Figure 1(b). The benefit of such a parallelization is that each task can start the next iteration of the loop, executing its assigned statements as soon as it has communicated its result to the tasks waiting for its output. In this way, a pipeline of calculations is created. Task T_1 starts with its first iteration at time t_0, and sends its output to T_2, so that T_2 can start its first iteration at time t_1. The next iteration of task T_1 can than be executed in parallel. Since the FFT call, mapped to task T_2, requires two time slots in our example, the first iteration of task T_3 starts at t_3. From now on, three tasks are executed in parallel, but due to the longer execution time of T_2, T_3 must always wait for its input data. These pipeline splits are also called *horizontal loop splits*.

Since more parallelism may be hidden in different pipeline stages, our approach is also able to split different iterations of each pipeline stage into further tasks. Figure 1(c) shows an example, where the statements of line 5-12 are mapped to one task, which is split up into the tasks $T_{1,1}$, $T_{1,2}$ and $T_{1,3}$. $T_{1,1}$ is now executing the iterations $\{1, 4, 7, ..\}$, while $T_{1,2}$ and $T_{1,3}$ are concurrently executing the iterations $\{2, 5, 8, ..\}$ and $\{3, 6, 9, ..\}$, respectively, of the statements assigned to the task. Due to the loop-carried dependency of Task T_2 to its next iteration no further improvements can be achieved by splitting up the iterations of this task. As can be seen in the figure, three FFT calls can now be executed in parallel and from time t_3 on, four well-balanced tasks are executing the loop in parallel. These further pipeline splits are also called *vertical loop splits*.

By combining horizontal and vertical splits, our algorithm is able to extract very efficient parallelization from loops, even if loop-carried dependencies inhibit traditional data parallelization. Our ILP-based approach provides both splits at the same time, so that it will not end up in a local optimum which may be found if the vertical splits would rely on a separated horizontal split. The example also shows that it is very important to take the execution time of the tasks into account to create a well-balanced task structure.

III. Program Dependence Graph

To enable automatic extraction of pipeline parallelism, an intermediate representation containing the dependencies of the application is required. Therefore, our approach uses a *program dependence graph (PDG)* like presented in [2]. The PDG combines control- and data-dependencies in a single representation which makes it most suitable for the extraction of parallelization. Our tool is able to automatically extract an augmented PDG from an application's source code.

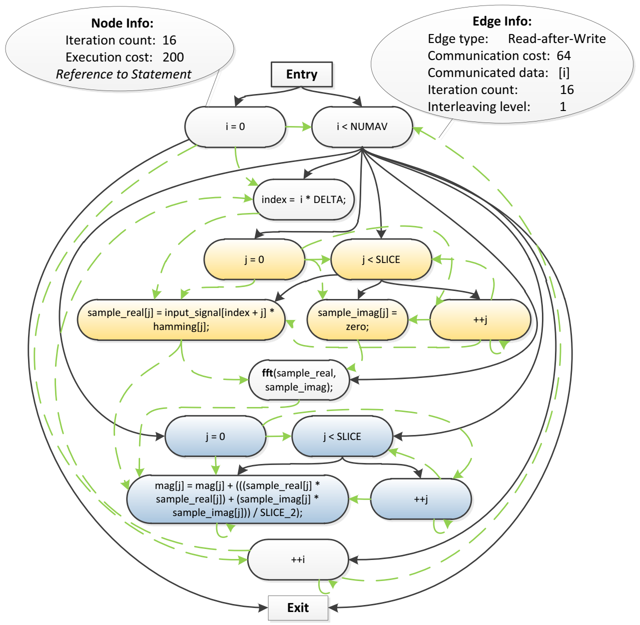


Fig. 2. Program dependence graph example

An example of such an augmented PDG is given in Figure 2. The graph represents the nested main computational loop of the *spectral* benchmark from Figure 1. Obviously, the graph contains one *entry* node, one *exit* node and several other nodes representing statements of the application. The nodes are connected by directed edges describing dependencies. Solid, black arrows represent control flow dependencies, while dashed arrows visualize data dependencies.

Formally, a program dependence graph $G = (V, E)$ is a directed graph with a set of nodes V, each representing a statement of the application. The nodes are connected with directed edges $E \subseteq V \times V$ denoting control- and data dependencies. For more details on how to construct the PDG, we would like to refer to [2].

In order to extract efficient parallelism from a sequential application, the created tasks have to be balanced, so that concurrently executed tasks will finish nearly at the same time. For this reason, each node of our PDG is augmented with the iteration count and execution costs of the statement represented by the node (cf. *Node Info* in Figure 2). It is also essential to have knowledge about the communication costs which have to be taken into account if the statements of the nodes are executed in separate tasks. Therefore, the edge type, communication costs, the communicated data, the iteration count as well as an interleaving level – describing the minimal amount of loop iterations which can be executed before the data is consumed at the target node – are annotated at the edges (cf. *Edge Info* in Figure 2).

IV. Parallelization Methodology

Based on an augmented PDG, our parallelization approach searches for pipeline parallelism in loops of applications.

```
 1: function DoParallelization(IR $ir$, int $maxTasks$)
 2:     $pdg \leftarrow$ ConstructPDG($ir$)
 3:     $loops \leftarrow$ CollectParallelizableLoops($pdg$)
 4:     $Sol \leftarrow \emptyset$
 5:     for $l \in loops$ do
 6:         $Sol \leftarrow Sol \cup$ Parallelize($l, maxTasks$)
 7:     end for
 8:     CombineBestResults($loops, Sol, maxTasks$)
 9: end function
10:
11: function Parallelize(Loop $l$, int $maxTasks$)
12:     $LoopPDG \leftarrow$ ConstructSubPDG($l$)
13:     $Solutions \leftarrow \{SequentialSolution\}$
14:     $i \leftarrow maxTasks$
15:     while $i > 2$ do
16:         $result \leftarrow$ ILPParallelizer($LoopPDG, i$)
17:         $Solutions \leftarrow Solutions \cup \{result\}$
18:         $i \leftarrow$ NumberOfTasks($result$) $- 1$
19:     end while
20:     return $Solutions$
21: end function
```

Fig. 3. Parallelization algorithm

A. Parallelization algorithm

The structure of our parallelization algorithm is shown in Figure 3. The approach starts by calling the function DoParallelization, which expects a high-level intermediate representation (IR) [3] of the application and an upper bound of concurrently executable tasks as input. By providing this upper task bound the user is always able to adjust the solution to his target platform. As can be seen in line 2, the function is first creating the program dependence graph based on the given IR. Then, the PDG is traversed searching for loops which can be parallelized by our algorithm. Afterwards, the function Parallelize is called for each loop in line 6. As a result, the function adds all extracted parallel solutions and the estimated execution costs for the parallelized loop to the set *Sol*. In a last step, the function CombineBestResults is executed in line 8, which picks out the best combination of parallelized loops, observing the maximum number of concurrent tasks.

The function Parallelize, starting at line 11 of Figure 3, is called for each loop. The function starts with the construction of a sub-PDG containing only those PDG nodes which are part of the loop. All dependencies coming from outside of the loop are starting at the *entry* node of the sub-PDG. All communication leaving the loop is redirected to the *exit* node of the sub-graph, respectively. In line 13, a set of possible solutions is created which is initialized with the sequential version of the loop. Thus, the function which is combining the results at the end (line 8) can always fall back to the sequential version of the loop if it finds other loops which are more beneficial to parallelize. Afterwards, the function ILPParallelizer is executed several times in a loop, starting with the maximum number of possible concurrently executed tasks downwards to only two available tasks. Thus, the combination function also has the option to combine, e.g., a solution with less than $maxTasks$ from this

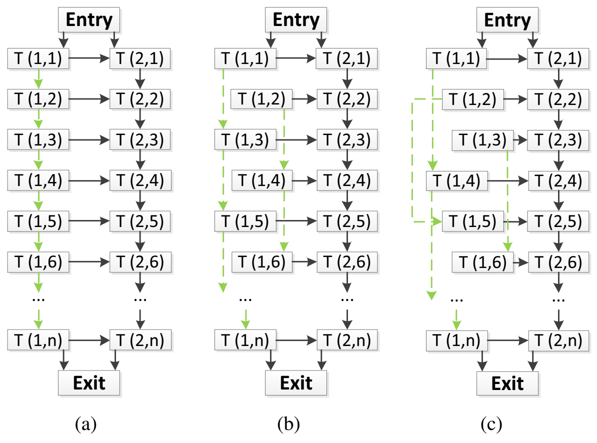


Fig. 4. Loop dependencies for vertical splits

loop and 3 tasks from another loop if it does not exceed the maximum number of concurrently executed tasks. The iterative call of the ILPParallelizer function is feasible since our local ILP formulations for each loop can be solved in less than a second for most cases. All solutions are collected in the solution set which is then returned as the result of the function in line 20.

B. Parallelization approach

As can be seen in line 16 of Figure 3, the algorithm is calling an ILP-based parallelization function, taking a PDG of a (nested) loop and an upper bound of concurrently executed tasks to generate as input, to find an efficient node-to-task mapping for each loop of the application.

To evaluate the performance of the found parallelization, the ILP is virtually unrolling the loops to minimize the most expensive execution path (*critical path*), starting from the *entry* to the *exit* node of the loop's PDG, by splitting the loop horizontally and vertically into different tasks. The computation of the critical path is based on an algorithm of Sarkar [4]. An example for two tasks and the dependencies of the *spectral* benchmark of Figure 1(c) is given in Figure 4(a-c). *T(x,y)* in the figure describes iteration y of task x. Figure 4(a) shows the dependencies which have to be taken into account if task 1 is not vertically split. Thus, each iteration of Task 1 depends on its previous iteration. The dependencies shown in Figure 4(b) represent the case in which task 1 is vertically split once. As can be seen, iterations $\{1, 3, 5, ..\}$ and $\{2, 4, 6, ..\}$ can be executed in parallel in that case. Figure 4(c) shows the case where task 1 is split into 3 tasks so that the iterations are grouped into $\{1, 4, 7, ..\}$, $\{2, 5, 8, ..\}$, and $\{3, 6, 9, ..\}$. Task 2 will not be split up into additional sub-tasks due to its loop-carried data dependency.

All cases and their dependencies, shown in Figure 4(a-c), are combined in one ILP formulation. In addition, the solid, black arrows of the figure, describing data dependencies, depend on the node-to-task mapping. Thus, the dependencies between the tasks may also change if one statement is moved from one task to another.

C. ILP Formulation

This section defines the ILP formulation for the pipeline-based parallelization approach described above. In the following, decision variables are written in lower case letters, sets start with a capital letter and constants contain only capital letters. The indices i and j are used for iterations of the loop to be parallelized, t and u will represent indices for tasks and n and m are used as indices for nodes of the PDG.

1) Horizontal task split constraint: One of the main results of the parallelization step is a mapping of PDG nodes to tasks, which is calculated by the ILP. This node-to-task mapping corresponds to the horizontal pipeline stage splits. Therefore, a decision variable $x_{n,t}$ is defined in Equation 1, which describes this relationship.

$$x_{n,t} = \begin{cases} 1, & \text{if node } n \text{ is mapped to task } t \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

The constraint in Equation 2 ensures that every node is mapped to exactly one of the possible tasks.

$$\forall n \in Nodes: \sum_{t \in Tasks} x_{n,t} = 1 \tag{2}$$

2) Vertical task split constraint: As explained in the previous section, the ILP has the option to split tasks (pipeline stages) vertically into several sub-tasks which are executing the iterations of the loop in an interleaved mode. This would mean that one sub-task may execute iterations $\{1, 3, 5, ..\}$ while another one would execute the iterations $\{2, 4, 6, ..\}$ of the statements, which are mapped to the given task. Therefore, a decision variable $split_{t,s}$ is added for each task and the number of possible splits. E.g., $split_{1,2} = 1$ means that task 1 is split twice, so that task 1 is composed of 3 sub-tasks. Equation 3 defines these decision variables.

$$split_{t,s} = \begin{cases} 1, & \text{if task } t \text{ is split } s \text{ times} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Equation 4 ensures that exactly one task split variable is chosen for each task. It is also possible to set the variable $split_{t,0} = 1$, which means that task t is not vertically split.

$$\forall t \in Tasks: \sum_{s \in \{0..\text{MAXTASKS}\}} split_{t,s} = 1 \tag{4}$$

3) Predecessor constraint: To minimize the critical or most expensive path from the *entry* to the *exit* node, the ILP formulation has to be extended by path information. As shown in Figure 4, the path costs are estimated by virtually unrolling the iterations of the loop. Thus, we have to create predecessor/successor variables for each iteration combination of two tasks. Equation 5 shows the definition of decision variable $pred_{t,i,u,j}$, which is added for each task t in iteration i and task u in iteration j.

$$pred_{t,i,u,j} = \begin{cases} 1, & \text{if task } t \text{ in iteration } i \text{ is predecessor} \\ & \text{of task } u \text{ in iteration } j \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

a) Task Split dependencies: As shown in Figure 4, constraints describing dependencies between different iterations of a task have to be added, depending on how often the task is split (dashed arrows). If, e.g., task t is split once, iterations 1 and 2 can be executed independently of each other (Figure 4(b)) – iff no data dependencies prevent the parallel execution. In addition, if the task is not split (Figure 4(a)), the iterations depend on each other since the task is then executed sequentially on the same core. These task-split dependencies are described in Equation 6, with $NUMITER$ depicting the number of loop iterations, which can be determined by e.g. [5].

$$\begin{gathered}\forall t \in Tasks : \forall i \in \{0,..,\text{NUMITER}\} : \forall j \in \\ \{0,..,\text{NUMITER}\} : (j - i \leq \text{MAXTASKS} \wedge j > i) \Rightarrow \\ pred_{t,i,t,j} \geq split_{t,j-i-1}\end{gathered} \tag{6}$$

b) Data and Control flow dependencies: Data and control flow dependencies between tasks (solid, black arrows of Figure 4) have to be taken into account as well which is done in Equation 7.

$$\begin{gathered}\forall t \in Tasks : \forall u \in Tasks : \\ \forall i \in \{0,..,\text{NUMITER}\} : \forall j \in \{i,..,\text{NUMITER}\} : \\ \forall n \in Nodes : \forall m \in Nodes : n \neq m : \\ pred_{t,i,u,j} \geq EDGE_{n,m,j-i} * (x_{n,t} \wedge x_{m,u})\end{gathered} \tag{7}$$

The predecessor variable $pred_{t,i,u,j}$ is created for all possible task and loop iteration combinations. This way, for all combinations of nodes, it is checked if node n is part of task t while node m has to be part of task u. If this is true and a directed edge from n to m exists with an interleaving level of $j - i$, denoted by $EDGE_{n,m,j-i}$, task u depends on t for the 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.

The $\wedge$ operator in Equation 7 is not part of regular ILP formulations, but can be modeled by the following constraints.

$$\begin{gathered}z = (x \wedge y) \in \{0,1\} \\ z \geq x + y - 1, \quad z \leq x, \quad z \leq y\end{gathered} \tag{8}$$

4) Execution costs of tasks constraint: The predecessor relationship enables to describe paths with respect to dependencies. Since it is very important to take execution and communication costs into account to create well-balanced tasks, we have to add the augmented cost information of the PDG (cf. Figure 2) to the ILP formulation. W.l.o.g. we distribute the overall execution costs of each node in equal parts over the different iterations of the loop. This saves a couple of decision variables, since we do not have to distinguish between different execution costs of tasks in different iterations.

$$\begin{gathered}\forall t \in Tasks : \\ cost_t \geq \sum_{n \in Nodes} x_{n,t} * (COST_n/\text{NUMITER})\end{gathered} \tag{9}$$

Equation 9 sets the lower bound of the costs for one iteration of task t to at least the sum of costs $COST_n$ of each node n, which is part of task t, divided by the number of loop iterations. The variable $cost_t$ is also part of the objective function, so that it is automatically minimized by the ILP solver if task t is part of the critical path.

5) Path cost constraint: Based on the knowledge of the execution costs of each task, it is now possible to describe the accumulated costs of the possible paths. Unfortunately, it is not known at this time, in which order the data will be communicated between two tasks. Therefore, a worst-case scenario is presented here which assumes that a task t has to wait for its data until all its predecessor tasks have communicated all data to the successor tasks, even if this data is not consumed by t. The ILP formulation of this worst-case scenario based path calculation is shown in Equation 10. It is also possible to change this worst-case optimization to e.g., an average-case based scenario, here.

$$\begin{gathered}\forall t \in Tasks : \forall u \in Tasks : \\ \forall i \in \{0,..,\text{NUMITER}\} : \forall j \in \{i,..,\text{NUMITER}\} : \\ accumcost_{t,j} \geq cost_t + accumcost_{u,i} + commcost_u - \\ \text{BIGCONST} + \text{BIGCONST} * pred_{u,i,t,j}\end{gathered} \tag{10}$$

Equation 10 ensures that the path costs $accumcost_{t,j}$ for task t in iteration j are at least as large as the costs $cost_t$ for the execution of one iteration of task t itself and the path costs of its most expensive predecessor $accumcost_{u,i}$, including all communication costs $commcost_u$ of task u. The last line automatically fulfills the constraint if task u in iteration i is *not* a predecessor of task t in iteration j by subtracting a big constant on the right hand side of the constraint which is bigger than all other values used within the ILP. The accumulated costs are also included in the objective function, so that it is automatically minimized by the ILP solver.

6) Additional constraints: There are also a couple of other constraints which are added to the ILP which, e.g., ensure that the maximum number of concurrently executed tasks is not exceeded and that the created parallelization ends up in a cycle-free graph. Due to space limitations, their explanation is skipped here. Nevertheless, they are not part of the objective function and are thus less important.

7) 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. Thus, we add additional constraints which statically set the *entry* node to be a predecessor of all tasks. The *exit* node will be a successor of all tasks, respectively. With the help of the additional constraints, it is easy to create the objective function, like shown in Equation 11.

$$\begin{aligned}\text{minimize} \quad & numtasks * \text{TASKOVERHEAD} + \\ & accumcost_{exit}\end{aligned} \tag{11}$$

Since the creation of tasks also increases the execution time, a constant task creation overhead, multiplied with the number of created tasks, is added. This 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. It is hence returned together with the node-to-task mapping as result of the parallelization step.

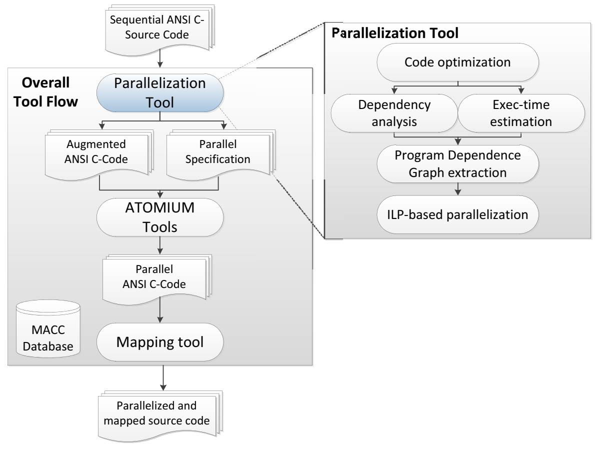
V. Experimental Environment

The techniques described in this paper were developed in the context of the MNEMEE European FP7 project [6]. All of them are implemented and integrated into an automated tool flow which is visualized in Figure 5(a). Here, only a subset of the tools developed in the MNEMEE project are shown, since we would like to focus on the benefit of the parallelization.

As can be seen, the tool flow starts with the source code of a sequential ANSI-C application, which is first parsed into a high-level intermediate representation, in our case an ICD-C IR [3]. All tools are developed on top of the MACC framework [7] which is used to facilitate the communication between all processing steps of the tool flow. The framework stores intermediate results, the application's IR and a description of the target platform in the MACC database, so that all tools can easily access this information.

The parallelization tool, described in this paper, is the first one in the presented tool flow. As a result, the tool annotates the given source code of the application to describe the extracted parallelism. Based on these annotations, a parallel specification is generated, complying with the input specifications of the ATOMIUM (MPA) tools [8], so that the extracted parallelization is implemented by them. Next, the resulting parallelized C-files are mapped to different processors of the target MPSoC platform. The resulting application can then be compiled and linked against a runtime library implementing the task creation and synchronization primitives.

The internal structure of the parallelization tool is shown in Figure 5(b) and is based on our parallelization framework which is described in [9]. Based on the IR of the application and the architectural information provided by the MACC framework, the sub-tools are applied in the given order to extract parallelism from sequential applications. The first one is the *code optimization tool*, which optimizes the source code regarding easier extraction of parallelism. Thereafter, the *dependency analysis tool* extracts data dependencies between different statements of the application. In the current version of our tool flow, a profiling-based approach is used to extract the data-dependencies. Nevertheless, this does not harm the correctness of our approach, since the ATOMIUM tool suite [8], which is based on safe static analysis techniques, is used to implement the parallelization found by this tool. The extracted information is extended with execution times of the statements which are extracted by the *execution time estimation* tool. All gathered information are then collected and combined with static knowledge of the program structure to extract the program dependence graph like described in Section III. Finally, the ILP-based parallelization tool is executed to extract the pipeline-based parallelism.



(a) (b)

Fig. 5. Implemented Tool Flow

VI. Experimental Results

To evaluate the effectiveness and efficiency of our parallelization approach, we present results for 12 benchmarks from the UTDSP suite [1], some additional embedded real-life applications like a JPEG encoder, and an industrial application used in INTRACOM TELECOM's Wimax system.

As target platform, we used the cycle-accurate MPARM simulator [10], which provides up to four separate single-core ARM processors, connected to local and shared memories. Due to the fact that each processor has its own memory the evaluated platform is representative for an MPSoC. Unfortunately, communication is in general much more expensive for these kinds of systems, compared to architectures with unified memories. Nevertheless, these facts are considered in our ILP-based parallelization approach.

To execute the benchmarks on the target platform we compiled the parallelized application with the platform-specific ARM-GCC and linked it to the RTEMS real-time operating system [11] which was extended by cross-core thread creation and other multi-core functionality.

The results for the parallelized benchmarks are shown in Figure 6. The measured execution time contains the execution time of the application without the initialization phase of the operating system and the runtime library, because this overhead affects both the sequential and the parallelized version of the application and has only to be done once. Thus, the results focus on the execution time of the application itself. As can be seen in the figure, we measured results for system configurations with two, three and four cores. The presented speedups are compared to the sequential execution times of the applications on a single core system. The amount of extracted, concurrently executed tasks was limited to the number of available cores. The presented results are measured by executing the benchmarks on their original input files, with an exception of *iir_4_64* and *mult_10_10*. For these two applications, the original input files are too small to achieve a benefit of additional tasks. Therefore, we increased the size

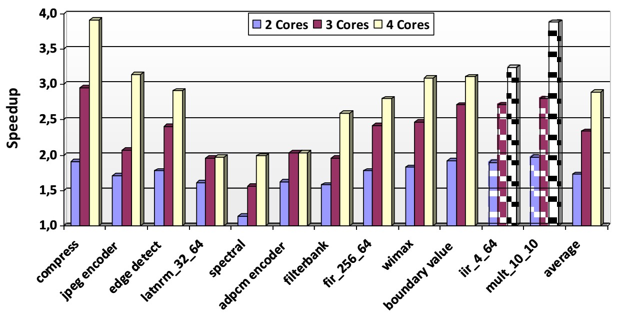


Fig. 6. Speedup of parallelization compared to sequential application code

of the input files by a factor of 8 for both benchmarks and shaded their bars in Figure 6.

As can be seen, the speedup scales very well with the given amount of processors for most applications. The highest speedup could be achieved for the *compress* benchmark with an increased performance of 1.9x, 2.9x and 3.9x compared to the execution of the sequential application. The high speedup results from the fact that the parallelization approach splits up the main computational loop into separate, concurrently executed tasks, so that nearly the whole application is executed in parallel. The speedup of the *spectral* benchmark amounts to 1.2x, 1.6x and 2.0x for a two-, three-, and four-core system configuration, respectively. Due to its simplicity and high amount of communication, the gain of the parallelization is not as large as for most other considered benchmarks. The parallelization implemented for two cores is visualized in Figure 1(b) with the difference that task T_1 and T_2 are merged. The parallelization for three and four cores is equivalent to the one shown in Figure 1(c). Due to the limited number of cores, T_1 is only split twice for the three core-version.

In addition to the presented real-life benchmarks, we also validated our approach with an industrially used application which is part of INTRACOM TELECOM's Wimax system. It was possible to increase the performance of this application by a factor of 1.8, 2.5, and 3.1 for the different numbers of available cores, which emphasizes the usability of the presented approach for industrial grade applications.

The average speedup of all 12 evaluated benchmarks is visualized by the last group of bars in Figure 6. It was possible to speed up the different applications on average by a factor of 1.8, 2.4 and 2.9, respectively.

Optimization time

The time required to parallelize an application with our tool flow strongly depends on the preprocessing steps of the tool flow. Processing the *edge detect* benchmark, for example, takes about 1:50 minutes, of which 1:29 minutes are used for the code optimization, dependency analysis and execution time estimation. For the parallelization step itself, 21 seconds are sufficient to parallelize all loops of the application with our ILP-based approach. This time contains the construction of the program dependence graph as well as the creation and solving times of 58 ILPs. The times were measured on a machine equipped with two quad-core AMD Opteron Processors @ 2.4 GHz, using one of the cores. This is also a proof of the applicability of the comprehensive ILP formulations.

VII. Related work

Due to the fact that parallel architectures were invented decades ago, many projects have focused on developing special programming languages or models like e.g., *MPI* [12], *PThreads* [13] and *OpenMP* [14] as well as semi- or fully automatic parallelization techniques. The exploitation of *instruction level parallelism* was also extensively [15] studied. Nevertheless, recent years have shown that software-based parallelization is becoming more and more important, since architectures provide multiple cores in one system. These parallelization techniques can be grouped into at least the categories of *task-level*, *data-level* and *pipeline parallelism*.

The first category is concerned with very coarse grained parallelism, which is e.g., executing two function calls on different cores. Hall et al. present a framework [16] which extracts this kind of parallelism automatically. It is based on an interprocedural analysis and was developed as part of the SUIF compiler system [17].

Ceng et al. developed a semi-automatic parallelization assistant [18]. The application is transformed into a weighted statement control data flow graph which is subsequently processed by a heuristically clustering algorithm, generating tasks after several iterations of a user-feedback loop.

Our previous publication [9] was also dealing with automatic exploitation of task level parallelism. The approach uses a hierarchical task graph as intermediate representation to reduce the search space. Compared to the approach presented in this paper, [9] is not able to extract parallelism from loops with loop-carried dependencies.

Creusillet [19] presents a framework called PAR4ALL which concentrates on the implementation of tasks. Thus, the user or an extraction tool has to identify parallelism, first.

In addition to this very coarse-grained task level parallelism, finer-grained data level parallelism is focusing on loops which can execute the same operations concurrently by splitting the amount of data to different tasks [20],[21].

The approach presented in this paper is extracting pipeline parallelism, combined with further splits of pipeline stages. Regarding this kind of parallelization, the work of Tournavitis et al. [22], [23] and Raman et al. [24], are most relevant to our work. Their approaches are also able to automatically extract pipeline parallelism. Compared to our work, their approaches are only able to split stateless pipeline stages into further tasks. In addition, the approaches lack a detailed cost model so that the parallelism is only extracted by a heuristic approach which is merging nodes based on a fixed threshold. Our work combines an adequate cost model with a clear, ILP-based mathematical description for the assignment of statements to tasks so that we can really balance the created tasks.

Liu et al. [25] present an approach which eliminates loop-carried dependencies by retiming the execution of statements in a loop. Therefore, they are 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. Just like the previous works, this one is also not aware of an accurate cost model which would make it possible to determine whether the parallelization of different loop iterations will increase the overall program performance.

Gordon et al. [26] present a compiler framework which combines the extraction of task-, data- and pipeline parallelism for applications written in the programming language StreamIt [27]. Since the user has to define independent actors, connected by explicit data channels, the designer has to extract tasks manually. The algorithms described in [26] are searching for parallelism in the given task structure. In contrast, our work extracts these tasks automatically based on sequential ANSI C-code, so that it can be applied to most existing embedded applications without first transferring it into another programming language.

Several other approaches also try to extract parallelism from sequential applications. E.g., Verdoolaege et al. [28] present a technique which transforms sequential applications into parallelized versions using process networks. Sarkar [29] and Ottoni [30] have introduced parallelization techniques based on program dependence graphs and extensions of them. Polychronopoulos et al. demonstrated the usability of hierarchical task graphs for automatic scheduling in [31].

With respect to the usage of integer linear programming in the parallelization domain, there is, to our best knowledge, only the work by Kadayif [32] and our previous publication [9]. However, Kadayif's approach does not partition the application, as done in this work. In fact, it is used to determine the most beneficial version composed of different parallel implementations.

VIII. Conclusions and Future work

To the best of our knowledge, this paper presents the first ILP-based pipeline parallelization approach tailored to the particular needs of embedded and cyber-physical systems. The tool is able to automatically control the granularity of the parallelization to find the most suitable solution with respect to the target platform by adding detailed knowledge of execution and communication cost to the program dependence graph. The efficiency of the tool was demonstrated on several real-life benchmarks from typical embedded system application domains. The measurements, performed on a cycle accurate simulator of an MPSoC, have shown that the tool is able to achieve speedups of up to 3.9x on a four-core system.

In the future we would also like to extend the presented approach to heterogeneous systems. In the current version, the tool takes only one execution time per statement into account while partitioning the application. If cores of different types are added to the same system, this approach has to be adapted to differing execution times, depending on the mapped core.

References

[1] C. G. Lee, "UTDSP Benchmark Suite," http://www.eecg.toronto.edu/˜corinna/DSP/infrastructure/UTDSP.html, July 2011.

[2] J. Ferrante, K. J. Ottenstein *et al.*, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, 1987.

[3] R. Pyka and J. Eckart, "ICD-C Compiler framework," http://www.icd.de/es/icd-c/icd-c.html, July 2011.

[4] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.

[5] P. Lokuciejewski, D. Cordes, H. Falk *et al.*, "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models," in *Proc. of CGO*. IEEE Computer Society, 2009.

[6] A. Mallik, S. Mamagkakis, C. Baloukas *et al.*, "MNEMEE - An Automated Toolflow for Parallelization and Memory Management in MPSoC Platforms," in *Proc. of DAC*. ACM, 2011.

[7] R. Pyka *et al.*, "Versatile System-level Memory-aware Platform Description Approach for embedded MPSoCs," in *Proc. of LCTES*, 2010.

[8] R. Baert, E. Brockmeyer *et al.*, "Exploring parallelizations of applications for MPSoC platforms using MPA," in *Proc. of DATE*, 2009.

[9] D. Cordes, P. Marwedel, and A. Mallik, "Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming," in *Proc. of CODES/ISSS*. ACM, 2010.

[10] L. Benini, D. Bertozzi, A. Bogliolo *et al.*, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *Journal of VLSI Signal Processing Systems*, 2005.

[11] RTEMS, "RTEMS Operating System — Real-Time and Real Free," http://www.rtems.com/, July 2011.

[12] M. Snir, S. Otto, S. Huss-Lederman *et al.*, *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, 1998.

[13] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[14] OpenMP, "The OpenMP API specification for parallel programming," http://www.openmp.org/, July 2011.

[15] G. Memik, G. Reinman, and W. H. Mangione-Smith, "Precise instruction scheduling," *Proc. of JILP*, 2005.

[16] 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.

[17] M. W. Hall, J. M. Anderson, S. P. Amarasinghe *et al.*, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, vol. 29, no. 12, 1996.

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

[19] B. Creusillet, "Automatic Task Generation on the SCMP architecture," http://www.par4all.org/, March 2011.

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

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

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

[23] G. Tournavitis, Z. Wang, B. Franke *et al.*, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine- learning based mapping," in *Proc. of PLDI*, 2009.

[24] E. Raman, G. Ottoni, A. Raman *et al.*, "Parallel-stage decoupled software pipelining," in *Proc. of CGO*. ACM, 2008.

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

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

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

[28] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: A Tool for Improved Derivation of Process Networks," *EURASIP Journal on Embedded Systems*, 2007.

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

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

[31] C. D. Polychronopoulos, "The hierarchical task graph and its use in auto-scheduling," in *Proc. of ICS*, 1991.

[32] I. Kadayif, M. Kandemir, and U. Sezer, "An integer linear programming based approach for parallelizing applications in On-chip multiprocessors," in *Proc. of DAC*, 2002.