Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming*

Daniel Cordes
Informatik Centrum Dortmund
Joseph-von-Fraunhofer-Str. 20
Dortmund, Germany
cordes@icd.de

Peter Marwedel
Informatik Centrum Dortmund
Joseph-von-Fraunhofer-Str. 20
Dortmund, Germany
marwedel@icd.de

Arindam Mallik
Imec Belgium
Kapeldreef 75
Leuven, Belgium
arindam@imec.be

ABSTRACT

The last years have shown that there is no way to disregard the advantages provided by multiprocessor System-on-Chip (MPSoC) architectures in the embedded systems domain. Using multiple cores in a single system enables to close the gap between energy consumption, problems concerning heat dissipation, and computational power. Nevertheless, these benefits do not come for free. New challenges arise, if existing applications have to be ported to these multiprocessor platforms. One of the most ambitious tasks is to extract efficient parallelism from these existing sequential applications. Hence, many parallelization tools have been developed, most of them are extracting as much parallelism as possible, which is in general not the best choice for embedded systems with their limitations in hardware and software support. In contrast to previous approaches, we present a new automatic parallelization tool, tailored to the particular requirements of the resource constrained embedded systems. Therefore, this paper presents an algorithm which automatically steers the granularity of the generated tasks, with respect to architectural requirements and the overall execution time reduction. For this purpose, we exploit hierarchical task graphs to simplify a new integer linear programming based approach in order to split up sequential programs in an efficient way. Results on real-life benchmarks have shown that the presented approach is able to speed sequential applications up by a factor of up to 3.7 on a four core MPSoC architecture.

Categories and Subject Descriptors

D3.4 [**Programming Languages**]: Processors—*Compilers*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Experimentation, Languages, Measurement, Performance

*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.

CODES+ISSS'10, October 24–29, 2010, Scottsdale, Arizona, USA.

Keywords

Automatic Parallelization, Embedded Software, Hierarchical Task Graph, Integer Linear Programming

1. INTRODUCTION

To meet the increasing demands that are imposed on modern embedded systems, plenty of computational power is needed. Multiprocessor System-on-Chip (MPSoC) architectures can be used to satisfy these requirements. Compared to a solution which increases the clock frequency of a single core architecture, it is possible to reach comparable computational power with lower energy consumption and less heat dissipation by using MPSoC platforms. Nevertheless, many new challenges have to be tackled, if existing sequential applications have to be ported to such multiprocessor systems. To exploit full advantages of these platforms, the applications have to be split up into several concurrent tasks to enable parallel execution on available processing units. However, by experience, manual parallelization tends to be a very error-prone and time consuming task. As a consequence, in most cases it is not possible to exploit the full potential provided by MPSoC platforms. Unfortunately, the complexity of embedded system's software is increasing, making it even more complicated to convert a sequential application into a parallelized one.

In general, it is essential to have detailed knowledge of the application to be parallelized, as well as of the underlying hardware platform. To mitigate these requirements, many models like *PThreads* [16], *MPI* [23] or *OpenMP* [17] were developed to support the designer in specifying the parallelism in his application. Nevertheless, the designer has to search for and to specify the parallelism of his application by hand in all details. Furthermore, depending on the used parallelization model, he is also responsible for introducing synchronization and communication between concurrent tasks, which is often a non-trivial exercise. As a consequence, the designer should be relieved of the burden of generating parallel code, so that he is able to focus primarily on modeling the behavior of the system.

In contrast to high performance computer systems, embedded systems have some specific characteristics which have to be addressed. On the one hand, they have a limited amount of available memory plus other hardware specific restrictions. On the other hand, specialized operating systems are often used to meet real-time constraints. To accommodate these constraints, in many cases the operating systems are very limited regarding e.g., the provided scheduling methodologies and threading support, which has to be

considered if parallel software should be mapped efficiently to such a platform. These observations were confirmed for the industrial embedded target platforms, considered in the MNEMEE European project, to which the presented work belongs. As a consequence, it is important to deliberately decide on how fine grained the parallelized code should be in order to get the most beneficial speedup on the MPSoC architecture.

Based on these observations, we developed a novel, fully automated parallelization tool which focuses on the special requirements of embedded systems. The proposed approach aims at a coarse grained, task level parallelism which can be limited to a given amount of concurrent tasks. Our techniques offer the possibility to e.g., limit the number of generated tasks to the number of available processors of the underlying hardware platform. To make the parallelization problem manageable, we introduce hierarchy into the task graph and search for parallelization opportunities on each level of the hierarchy by using integer linear programming (ILP). Although ILP may produce good or even optimal solutions to the parallelization with respect to the used model, it has not been employed in other works for this purpose so far. To highlight the applicability for embedded systems, results will be shown for embedded real-life applications, executed on a non-UMA based architecture.

The main contributions of this paper are as follows:

1. To our best knowledge, this is the first study which uses integer linear programming to find parallelism in sequential applications.
2. In contrast to the high performance community, this approach considers constraints, which are relevant for embedded system applications.
3. Our approach automatically steers the granularity of the extracted parallelism according to the underlying hardware platform.

The rest of this paper is organized as follows: Section 2 gives a survey of related work. The hierarchical task graph, which is used as intermediate representation during the parallelization step, is described in Section 3. The parallelization step itself is presented in Section 4, including the used ILP formulation. Section 5 gives a short overview of our tool flow, followed by experimental results in Section 6. Finally, Section 7 summarizes this paper and gives directions for future work.

2. RELATED WORK

Since parallel architectures were invented decades ago, much research has been done to develop automatic compiler techniques, tailored to the special needs of these platforms. One of these research areas aims at exploitation of *instruction level parallelism*, which includes optimizations like instruction scheduling or superscalar execution [14]. Its aim is to speed up the application by exploiting hardware level parallelism in isolation for each processor unit.

In contrast, *data level parallelism*, which is also well known as *loop-level parallelism*, focuses on splitting up an application to be computed by several cores concurrently. It concentrates on the dissemination of data to the available cores. This kind of parallelism can often be found in loops, whose iterations are therefore distributed. Some of these concepts are presented by Franke et al. [5] or Chandra et al. [4].

Our approach addresses the extraction of coarse grained *task level parallelism*. As well as for instruction and data level parallelism, additional research has been done on how to extract this kind of coarse grained parallelism from sequential applications. Hall et al. present a framework [7] which is able to do this automatically. It is based on an interprocedural analysis and was developed as part of the SUIF compiler system [8]. The work also presents advanced analysis techniques that are able to identify possible concurrency.

Ceng et al. developed a semi-automatic parallelization assistant [3]. The application code is transformed into a weighted statement control data flow graph which is subsequently processed by a heuristically clustering algorithm, generating tasks after several iterations. The approach proposed by Ceng requires a user-feedback loop to steer the granularity of the parallelized program. In contrast, our approach controls the granularity in a fully automatic way, depending on the given architectural description.

Several other approaches try to extract parallelism from sequential applications, as well. Verdoolaege et al. [25] present a technique which transforms sequential applications into parallelized versions using process networks. Therefore, all loops in the application have to be affine which is not always the case in real-life applications. Sarkar [22] and Ottoni [18] have introduced parallelization techniques based on program dependence graphs and extensions of them. Compared to their approach, we use a kind of hierarchical task graph that is similar to the one presented by Girkar et al. [6]. Polychronopoulos et al. demonstrated its usefulness for an automatic scheduling algorithm in [19].

With respect to the usage of integer linear programming in the parallelization domain, there is, to our best knowledge, only the work by Kadayif [12]. However, the described approach is not used to partition the application, as done in this work. In fact, it is used to determine the most beneficial version composed of different parallel implementations.

3. HIERARCHICAL TASK GRAPH

One of the main challenges of extracting parallelism from sequential code is the complexity of the solution space. The parallelization step of this work uses an integer linear programming based approach to exploit parallelism of an application. Since ILP systems are NP-hard in general, a good approximation of the problem description is essential. Without an approximation, the solution space is infeasible for even small sized applications. Hence, an abstraction level has to be introduced to make the parallelization step manageable.

One possibility, which is often used in literature, is the introduction of hierarchy in the underlying model. Based on a flat task graph, Girkar and Polychronopoulos developed a hierarchical task graph model [6], which is also used as intermediate representation in our proposed approach. However, some changes to the graph of Girkar are needed for the approach of this paper.

To automate the presented parallelization tool, we developed a technique which extracts a corresponding hierarchical task graph from sequential ANSI C code.

3.1 Structure

An example of a hierarchical task graph is given in Figure

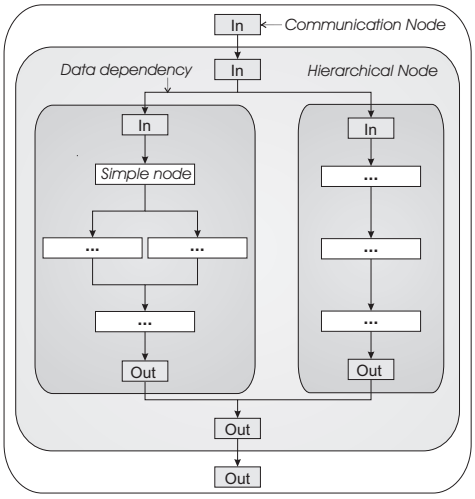

Figure 1: Hierarchical Task Graph Example

1. As depicted, the graph contains communication edges for data dependencies and four different kinds of nodes:

- *Simple nodes*
  These nodes correspond to one basic statement in the original C code. They do not contain any child nodes. Such a statement could be, e.g., $a = b$;
- *Hierarchical nodes*
  These nodes correspond to e.g., loop or function bodies in the original source code. All hierarchical nodes contain a communication in-node, a communication out-node and an arbitrary number of child nodes. These child nodes can be either simple nodes or other hierarchical nodes.
- *Communication in-nodes*
  These nodes are part of every hierarchical node. Communication from a node not contained in the hierarchical node to any inner node is redirected through this communication in-node.
- *Communication out-nodes*
  These nodes are also part of every hierarchical node. Communication from a child node of the hierarchical node to any node not contained in the hierarchical node is redirected through this communication out-node.

In contrast to the model of Girkar [6], we added the special communication nodes to encapsulate the communication between different levels of the hierarchy. As can be seen in Figure 1, communication between different levels of the hierarchy is always redirected through these special communication nodes. This fact will be used later on in the parallelization step, because it enables the extraction of parallelism for each node in isolation.

In addition, hierarchical task graphs have further beneficial characteristics. The control flow is already covered by the hierarchical structure of the graph except for special jump statements like `break`, `return` or `goto`. With the exception of the mentioned jump statements, no edges (especially back edges) have to be added to model the control

1: **while** $i \leq 10$ **do**
2: $a[i] \leftarrow 0$
3: $i \leftarrow i + 1$
4: **end while**

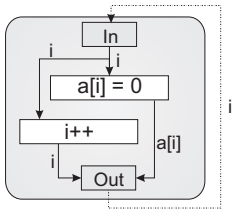

Figure 2: Communication Example

flow of the application. Jump statements are handled by communication edges without communication costs. In the current implementation, we distinguish between the following data dependency types:

- *Def-Use Edges:* read-after-write dependencies *(RAW)*
- *Use-Def Edges:* write-after-read dependencies *(WAR)*
- *Def-Def Edges:* write-after-write dependencies *(WAW)*

By construction, there is another advantage of the presented model. Figure 2 shows an example how loop carried dependencies are modeled within the hierarchical task graph. The statement of line 2 consumes data, produced by the previous iteration of statement 3. Instead of adding a communication edge in the opposite direction of the control flow, the value of variable i is send to the communication out-node of the hierarchical node, so that it is available at the communication in-node for the next iteration of the loop. The dashed edge visualizes an implicit communication, hidden in the hierarchy. Another communication edge is then added to send the value from the communication in-node to the node of the consuming statement of line 2. As a consequence, all cyclic dependencies are hidden in the hierarchical structure.

3.2 Additional Annotations

The information, which can be extracted from the presented task graph itself, is insufficient for the automatic parallelization technique described in this publication. Therefore, the task graph is automatically augmented with additional information which is described in more detail in the following.

In order to obtain an efficient parallelization for a sequential application, the created tasks have to be balanced, so that concurrently executed tasks should finish nearly at the same time, especially if they have to be synchronized. Otherwise, some of the tasks would have to wait for completion of other tasks. For this reason, weights must be added to the nodes of the graph. These weights denote execution times multiplied by the execution counts of statements, the node is representing. Section 5 describes how these execution times are extracted in our case.

In addition, it is also essential to have knowledge about the communication costs. These are computed as the amount of communicated bytes multiplied by a communication overhead, given by the description of the underlying hardware platform. This way, it is possible to employ the parallelization tool for different hardware platforms. The communication costs, as well as the communicated variables are annotated to the edges of the graph.

Having knowledge about both costs, our approach is able to generate a well balanced version of the application. To

enable an automatic tool flow, these additional annotations are collected automatically (cf. Section 5).

To summarize this section, the advantages of the presented hierarchical task graph are as follows:

- The parallelization problem becomes manageable by the abstraction of the hierarchical task graph.
- In general, no control flow edges have to be added.
- Since the back edges of a classical control flow graph are hidden in the hierarchy, all sub-graphs are cycle free. This fits well with the chosen optimization function of the parallelization step, which is presented in the next section.
- All communication of the child nodes is encapsulated by the introduction of the communication in- and communication out-nodes. This means that every node is self-contained and can be optimized without having detailed knowledge about other nodes in the graph.

4. PARALLELIZATION METHODOLOGY

Based on the extracted augmented hierarchical task graph, the parallelization tool is able to extract parallelism from the application's source code. Therefore, this section explains the details of the parallelization step. First of all, Section 4.1 gives an overview of the overall parallelization algorithm. Section 4.2 will then describe how the parallelization is determined for each node in the graph, followed by a detailed definition of the ILP formulation in subsection 4.3.

4.1 Parallelization algorithm

The functionality of the parallelization algorithm is shown in Figure 3. The function PARALLELIZE takes a node n and an upper bound on the number of creatable tasks (e.g., equal to the number of cores) as input to create parallel solutions – mappings of child nodes c_i to tasks T_j – as output. These solutions are called *parallel sets*. An example is given in Table 1.

Tasks	Task exec. times	Task mapping
1	$T_{Seq} \rightarrow 100$	$T_{Seq} \rightarrow \{c_1, c_2, c_3, c_4\}$
3	$T_{Seq} \rightarrow 40$, $T_1 \rightarrow 30, T_2 \rightarrow 30$	$T_{Seq} \rightarrow \{c_1, c_2\}$, $T_1 \rightarrow \{c_3\}, T_2 \rightarrow \{c_4\}$
4	$T_{Seq} \rightarrow 20, T_1 \rightarrow 20$ $T_2 \rightarrow 30, T_3 \rightarrow 30$	$T_{Seq} \rightarrow \{c_1\}, T_1 \rightarrow \{c_2\}$, $T_2 \rightarrow \{c_3\}, T_3 \rightarrow \{c_4\}$

Table 1: Parallel Sets Example

The first column contains the number of concurrently executed tasks for the created solutions of node n. The second one shows the execution time in CPU cycles per task, depending on the node-to-task mapping of the last column. The sequential task T_{Seq} is also added to the amount of concurrently executed tasks in the first column, since it must also be executed on one of the available cores. The number of concurrently executed tasks may also be increased if parallel versions of the child nodes c_i are used within the resulting parallel set.

Our parallelization process is started by calling the function PARALLELIZE with the root node of the graph. As can be seen in lines 2-11 of Figure 3, the function is called recursively in a depth-first-search-manner initially. As a consequence, the algorithm starts the parallelization step at the innermost nodes of the graph. By construction, these nodes of the presented hierarchical task graph are always *simple-* or *communication nodes* (cf. Section 3). Since it makes no sense to move one node to a separate task and wait for its completion, these nodes are skipped in the parallelization algorithm and a solution set containing only the sequential solution of this one task is returned (lines 2-5). The algorithm is then moving upwards in the graph hierarchy and will reach a hierarchical node. Since all child nodes have already been processed, parallel sets are available for them. These parallel sets are combined in line 13 to be used as input for the integer linear programming based parallelization approach (cf. Section 4.2).

```
1: function PARALLELIZE(Node n, int maxTasks)
2:     NewParSets ← {SequentialSet}
3:     if ISNOTHIERARCHICALNODE(n) then
4:         return NewParSets
5:     end if
6:     # Cluster bottom-up in hierarchy, first.
7:     ChildSets ← ∅
8:     for all c ∈ ChildNodes do
9:         childset ← PARALLELIZE(c, maxTasks)
10:        ChildSets ← ChildSets ∪ {childset}
11:    end for
12:    # Cluster this node now (all child nodes clustered).
13:    InputSets ← COMBINERESULTS(ChildSets)
14:    i ← maxTasks
15:    while i > 2 do
16:        set ← ILPPARALLELIZER(n, i, InputSets)
17:        NewParSets ← NewParSets ∪ {set}
18:        i ← NUMBEROFTASKS(set) − 1
19:    end while
20:    return NewParSets
21: end function
```

Figure 3: Pseudo code of parallelization function

As already described, one major characteristic of this parallelization algorithm is the option to limit the number of generated concurrently executed tasks. Therefore, the ILP based parallelization step is executed several times to compute solutions for $maxTasks$ down to 2 tasks, represented by the while loop in the lines 15-19. Since the parallelization step is based on ILP, the solution w.r.t. the model is always optimal. This means, that if the parallelization step returns a solution set with 4 tasks, even if we restricted the algorithm to generate at most 6 tasks, the iterations with 5 and 4 maximum tasks can be skipped, respectively. The generated *parallel sets* for node n are added to the $NewParSets$ set in line 17, before these are returned as the result of the parallelization step for node n. Once node n is processed, the algorithm continues by moving upwards in the hierarchy, until the root node is reached and all child nodes are processed. The root node contains the final parallelization result which is therefore returned.

4.2 ILP based parallelization approach

The previous section described the overall parallelization algorithm. As can be seen in line 14 of Figure 3, an ILP based approach is used to parallelize a node n for a given upper bound for the number of concurrent tasks and the already generated parallel sets of its child nodes. This sub-

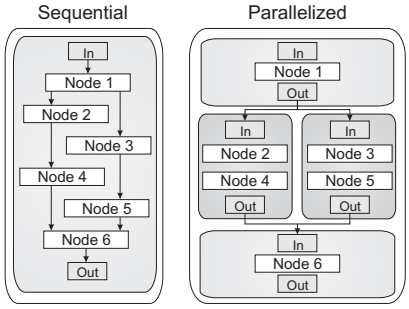

Figure 4: Parallelization Example

section presents the conceptual idea of the parallelization step, before the ILP formulation will be presented in more detail in Section 4.3.

As the objective, the parallelization step tries to minimize the *critical path* (or most expensive path) within a hierarchical node n. In particular, this means that the ILP-based approach aims at the minimization of the costs of the path from the hierarchical node's communication in- to its communication out-node by moving some of its child nodes to concurrent tasks. Since every child node has parallel sets with different execution times, depending on the number of tasks, deeper in the hierarchy, attached (cf. Table 1), the ILP solver is even able to pick solutions with different granularities for the child nodes, as long as the maximum number of concurrent tasks is not exceeded. The computation of the critical path is based on Sarkar [21].

To be able to adopt this approach to different hardware platforms, the user can specify the following parameters:

- *Task creation overhead*
  This overhead is added to the computed path for every created task. The parameter can be used to steer the granularity of the parallelization step, depending on the utilized hardware platform.
- *Communication costs*
  Communication costs have to be added, if data has to be transferred from one task to another. The communication overhead can also be changed by the user to model different hardware platforms. In this model, communication takes only place at the beginning and at the end of a task.

The parallelization approach divides the hierarchical node into three sections. All statements of the first section are executed sequentially on the same processor, which started the execution of the hierarchical node. The second one is the so called parallel section, where different concurrent tasks can be executed. In order to leave such a parallel section, all tasks have to synchronize and all data has to be communicated back to the parent task. The last section is again a sequential section, where statements without explicit data communication can be executed. This model is comparable to the MPA [1] or OpenMP [17] based *Sections* directive. In general, it is often useful to execute some statements before and after a parallel section due to high communication costs and less parallelism in some parts of the application.

An example of such a partitioning is shown in Figure 4. The input for the parallelization step is given on the left hand side. The sequential, hierarchical node, which should be parallelized, contains 6 child nodes, its communication in- and communication out-node and several data dependencies, which may produce communication, if the statements are executed in different tasks. The right hand side of Figure 4 shows a possible result of the ILP based approach. Here, the solver decided to map *Node 1* to the sequential task that is executed before the parallel section. Therefore, no data has to be communicated to execute *Node 1*. *Node 2* and *Node 4* are then moved to a separate task. The second concurrent task contains *Node 3* and *Node 5*. *Node 6* is finally executed after the two tasks have been synchronized. Thus, two tasks, each one containing two child nodes, are executed in parallel, which decreases the overall execution time of the hierarchical node. It should be mentioned here, that the inner-task communication edges are removed in the parallelized version of Figure 4, because data communication within the same task is superfluous.

4.3 ILP Formulation

Integer linear programming is a well known approach that is often used for partitioning problems. Even though ILP is NP-complete, the solutions can be determined very efficiently by commercial as well as by open source solvers for many real-life problems. The following sections define the ILP formulation for the approach described in Section 4.2.

4.3.1 Node in task constraint

The result of the parallelization step is a mapping of nodes to tasks, which is calculated by the ILP. Therefore, a decision variable $x_{i,j}$ is defined in Equation 1, which describes this relationship.

$$x_{i,j} = \begin{cases} 1, & \text{if node } i \text{ is mapped to task } j \\ 0, & \text{else} \end{cases} \tag{1}$$

The constraint in Equation 2 takes care that every node is mapped to exactly one of the given tasks.

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

4.3.2 Parallel sets constraint

As explained in the previous section, nodes, which should be mapped to tasks, have different execution times, depending on the number of concurrently executed tasks, deeper in the hierarchy. One parallel set has therefore a number of concurrent tasks and a mapping of nodes to tasks deeper in the hierarchy (cf. Table 1). The ILP has the possibility to choose one of the offered parallel sets, which is then used as basis for the newly generated parallel version of the active node. The decision variable for the chosen parallel set is defined in Equation 3.

$$p_k = \begin{cases} 1, & \text{if parallel set } k \text{ is taken} \\ 0, & \text{else} \end{cases} \tag{3}$$

Equation 4 takes care that exactly one parallel set is chosen.

$$\sum_{k \in ParSet} p_k = 1 \tag{4}$$

4.3.3 Predecessor constraint

To minimize the critical or most expensive path from the communication in- to the communication out-node, the ILP

formulation has to be extended by path information. Therefore, a predecessor/successor relationship between tasks l and m has to be added, that is realized by the decision variable $pred_{l,m}$ in Equation 5.

$$pred_{l,m} = \begin{cases} 1, & \text{if task } l \text{ is predecessor of task } m \\ 0, & \text{else} \end{cases} \quad (5)$$

In this case, it is not straightforward to describe this relationship, because the dependency information is currently only available for the nodes of the graph and not for the tasks. Moreover, the task dependencies hinge on the node to task mapping. As a consequence, the paths can change dynamically, if a node is moved from one to another task. This makes it hard to describe the predecessor relationship for tasks, given in Equation 6.

$$\begin{aligned} &\forall l \in Tasks : \forall m \in Tasks : l \neq m : \\ &\forall n \in Nodes : \forall o \in Nodes : n \neq o : \\ &pred_{l,m} \geq EDGE_{n,o} * (x_{n,l} \wedge x_{o,m}) \end{aligned} \quad (6)$$

The predecessor variable $pred_{l,m}$ is created for all possible task combinations. Furthermore, it checks for all node combinations if node n is part of task l while node o has to be part of task m. If this is true and a directed edge from n to o exists ($EDGE_{n,o} = 1$), task m depends on task l.

From the technical perspective, it should be mentioned that the constant $EDGE_{n,o}$ is known when the ILP is created. Therefore, constraints are only generated, if an edge between n and o exists. It should be mentioned, that we skipped the explanation of the different edge types in the path calculation with respect to the limited amount of pages.

Two further exceptions belong to the calculation of the correct path information: the sequential task right before the parallel section (cf. Figure 4) is a predecessor of all other tasks. Analogously, the sequential task after the parallel section is a successor of all other tasks or vice versa: each other task is a predecessor of the sequential task after the parallel section.

The $\wedge$ operator, used in Equation 6 is substituted by the constraints of Equation 7 and will also be used in the following equations.

$$\begin{aligned} z = (x \wedge y) &\in \{0, 1\} \\ z &\geq x + y - 1 \\ z &\leq x \\ z &\leq y \end{aligned} \quad (7)$$

4.3.4 Execution costs of tasks constraint

The predecessor relationship enables to describe paths with respect to dependencies. To weight these paths, costs have to be added to nodes as well as to edges. Equation 8 is responsible for this.

$$\begin{aligned} \forall j \in Tasks : cost_j \geq ITERCOUNT * OVERHEAD + \\ \sum_{i \in Nodes} \sum_{k \in ParSec} (x_{i,j} \wedge p_k) * COST_{i,k} \end{aligned} \quad (8)$$

Equation 8 sets the lower bound of the costs for each task j to at least the sum of costs $COST_{i,k}$ of each node i contained by j, depending on the chosen parallel set k. Furthermore, a constant overhead $OVERHEAD$ for task creation is added, which is multiplied by the iteration count $ITERCOUNT$ of the task. This is done to steer the granularity of the created tasks. The variable $cost_j$ is also part of the objective function, so that it is minimized by the ILP solver automatically, if task j is part of the critical path.

The costs of the sequential tasks are slightly different. Because they are executed on the same core as the previous nodes, the task creation overhead $ITERCOUNT * OVERHEAD$ of Equation 8 can be ignored.

4.3.5 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 p has to wait for its data until all its predecessor tasks have communicated all data to their successor tasks, even if this data is not consumed by p. The ILP formulation of this worst-case scenario based path calculation is shown in Equation 9. It should be mentioned here that it is an easy task to change the optimization function at this point to e.g., an average-case based scenario.

$$\begin{aligned} &\forall p \in Tasks : \forall q \in Tasks : p \neq q \\ &accumcost_p \geq cost_p + accumcost_q + commcost_q - \\ &BIGCONST + BIGCONST * pred_{q,p} \end{aligned} \quad (9)$$

Equation 9 ensures that the path costs $accumcost_p$ for task p are at least as high as the costs $cost_p$ of task p itself and the path costs of its most expensive predecessor $accumcost_q$, including all communication costs $commcost_q$ of task q.

Since there is no way to describe conditional constraints in an ILP formulation, the last line of Equation 9 takes care, that the equation is fulfilled automatically, if task q is *not* a predecessor of task p. If the variable $pred_{q,p} = 0$, a big constant $BIGCONST$, which is greater than the sum of all other possible values, is subtracted from the right-hand side of the constraint, so that it is fulfilled automatically for all variable assignments. On the other hand, if $pred_{q,p} = 1$, the last line of the constraint nullifies itself. The accumulated costs are also included in the objective function, so that it is automatically minimized by the ILP solver.

4.3.6 Number of hierarchical tasks constraint

Each child node may contain a number of concurrently executed tasks deeper in the hierarchy, depending on the chosen parallel set (cf. Table 1). To be able to restrict the overall number of concurrent tasks, the amount of hierarchical tasks $htasks_j$ has to be calculated for the newly created tasks j, based on the node-to-task mapping. Since all nodes within the same task are executed sequentially, the hierarchical tasks of these nodes will not overlap. Thus, the number of hierarchical tasks is as high as the maximum number of hierarchical tasks of all nodes mapped to the new task. This is expressed in the constraint of Equation 10.

$$\begin{aligned} &\forall j \in Tasks : \forall i \in Nodes : \forall k \in ParSet : \\ &htasks_j \geq (x_{i,j} \wedge p_k) * DEEPERHTASKS_{i,k} \end{aligned} \quad (10)$$

The number of hierarchical tasks $DEEPERHTASKS_{i,k}$ for a node i depends on the chosen parallel set k. Nevertheless, $DEEPERHTASKS_{i,k}$ is a constant, contained in the parallel sets (cf. first column of Table 1).

4.3.7 Max. number of concurrent tasks constraint

To limit the number of concurrently executed tasks, a new decision variable $taskused_j$ is introduced in Equation 11, which shows, if a task j is used.

$$taskused_j = \begin{cases} 1, & \text{if task } j \text{ is used} \\ 0, & \text{else} \end{cases} \quad (11)$$

Task j is used, if it contains at least one node, which is ensured by Equation 12.

$$\forall j \in Tasks : \forall i \in Nodes : taskused_j \geq x_{i,j} \quad (12)$$

The number of concurrently executed tasks is equal to the sum of the newly created tasks, increased by the number of hierarchical tasks. Here, only the parallel tasks are taken into account, since the executed tasks in the sequential parts are already synchronized, before the parallel section is entered. Equation 13 takes care, that the number of newly created and hierarchical tasks does not exceed the given upper bound $MAXTASKS$ of concurrently executed tasks.

$$MAXTASKS \geq \sum_{j \in Tasks} (taskused_j + htasks_j) \quad (13)$$

4.3.8 Cycle free constraint

To ensure that the created tasks are deadlock free, the nodes are sorted topologically by their dependencies. An ascending, unique ID is assigned to each of the tasks. W.l.o.g. the generated local task graph is cycle free, if the task ID of a node i has to be at least as high as the task ID of node n, if node i depends on node n. This is shown by Equation 14.

$$\forall i \in Nodes : \forall n \in Nodes : topsort_i \geq topsort_n : \\ taskID_i \geq taskID_n \quad (14)$$

4.3.9 Objective function

With all defined decision variables and constraints, it is now possible to describe the objective function. As mentioned before, the critical or most expensive path from the communication in- to the communication out-node should be minimized. The costs of this path are stored in the value of the variable *accumcost* of the sequential out task $t_{seq_{out}}$, because it is the successor of all other nodes in the created task graph. Thus, it should be minimized by the ILP solver:

$$exectime = \min\{accumcost_{t_{seq_{out}}}\} \quad (15)$$

The value of the objective function is equivalent to the execution time needed to execute the parallelized hierarchical node. It is hence returned together with the node-to-task mapping as the result of the parallelization step.

The number of constraints is exponential in the number of nodes. Nevertheless, the ILP can be solved quickly for the considered real-life benchmarks. In general, the hierarchical task graph reduces the amount of child nodes to five up to fifteen nodes. Empirical results have shown that most of the ILPs can be solved in less than a second, using the commercial ILP solver CPLEX [10].

5. EXPERIMENTAL ENVIRONMENT

All described techniques are developed as contribution to the MNEMEE European project. They are fully implemented and integrated into the MACC framework [20],

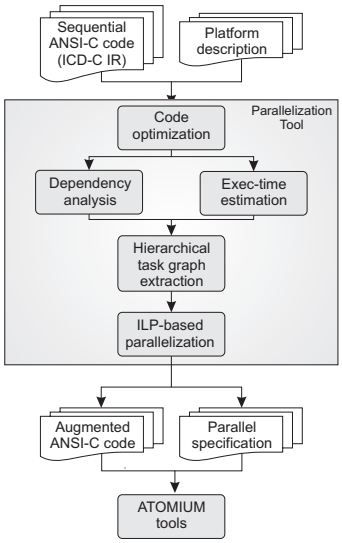

Figure 5: Implemented Tool Flow

which is used to facilitate the communication between all processing steps employed in the parallelization approach. Furthermore, the platform dependent information is also provided by the platform description of the framework, like e.g., the task creation- or communication overhead, steering the granularity of the ILP based parallelization algorithm.

The internal structure of the parallelization tool is visualized in Figure 5. As can be seen, the input to the tool is sequential ANSI C code, combined with the platform description of the MACC framework. The code is parsed by the ICD-C compiler framework [11] to extract a high-level intermediate representation (ICD-C IR) of the application's source code. Based on this intermediate representation and the architectural information provided by the MACC framework, the sub-tools of Figure 5 are applied in the given order to extract a parallelized version of the sequential source code. The sub-tools are now briefly described.

Code optimization

The code optimization tool prepares the code for the subsequent parallelization steps. These optimizations are e.g., *constant propagation*, *constant folding*, *dead code elimination* and other standard compiler optimizations described in [15]. Furthermore, the code optimization tool also applies optimizations which explicitly help the parallelization tool to find more effective parallelism in the application. One of these optimizations is a specialized version of *loop unrolling* which additionally helps this coarse grained parallelization approach to find parallelism in loops.

Dependency analysis

The dependency analyzer takes the intermediate representation of the optimized source code as input to extract all data dependencies of the application. This information is required to build the hierarchical task graph. In this case, a profiling based approach is used. Due to profiling driven analysis techniques, parallelization hints might ignore dependencies which are not manifest in the profiling run. This does not harm the correctness of the approach, since the ATOMIUM tool suite [1] is used, which is based on safe

static analysis techniques. ATOMIUM introduces synchronization and communication if necessary.

The profiling based approach has some advantages compared to a static analysis. For example, it delivers very detailed information about access patterns and very fine grained dependency information. Thus, by using the profiling based approach, it is possible to identify for example that loop iterations modifying arrays or pointers are independent and therefore possible parallelization candidates. Usually, this is very hard to detect via static analysis techniques. These observations are also reported in [24], where a profiling driven parallelization approach is presented.

Moreover, the dependency analyzer also returns the execution count of each statement.

Execution time estimation

To be able to attach weights to nodes of the hierarchical task graph, approximated information about the execution time of statements for a given hardware platform has to be known. Since we are currently using the cycle accurate MPARM simulator [2] as evaluation platform, the execution time estimation is specialized for this architecture.

This version of the execution time estimation simulates each statement on the given platform to extract an average execution time. Because node weights have to be extracted on statement level, the generated information is not as precise as an overall system simulation. Nevertheless, our experimental results (cf. Section 6) have shown that the extracted execution times are sufficient to create well balanced tasks. Compared to other approaches like Ceng [3] which just calculates weights according to an execution time per operation matrix, our model is more accurate.

Hierarchical task graph extraction

As soon as the previous steps have finished, all information which is required to extract the hierarchical task graph is available. The hierarchical structure of the graph is extracted statically with the use of the ICD-C framework. Then, the created nodes are connected with directed dependency edges, based on already gathered information of the dependency analyzer tool. Finally, the graph is augmented with weights of nodes and communication edges.

ILP based parallelization

Based on the extracted hierarchical task graph, the parallelization step is executed, as described in Section 4.

As a result, the parallelization tool annotates the given source code of the application. Based on these annotations, a parallel specification is generated. These annotations comply with the input specifications of the ATOMIUM tools, so that the extracted parallelization can be implemented by these tools automatically.

Implementation of parallelization

The implementation of the parallelization is done by the MPA tool of the ATOMIUM suite [1]. The ATOMIUM tools alone are not capable of automatically extracting parallelism from a sequential application. Typically, the user has to specify the parallelism on his own, which is now superfluous, because this information is automatically generated by the presented approach of this paper. The communication between both tools is fully automated to close the gap for an autonomously operating parallelization tool.

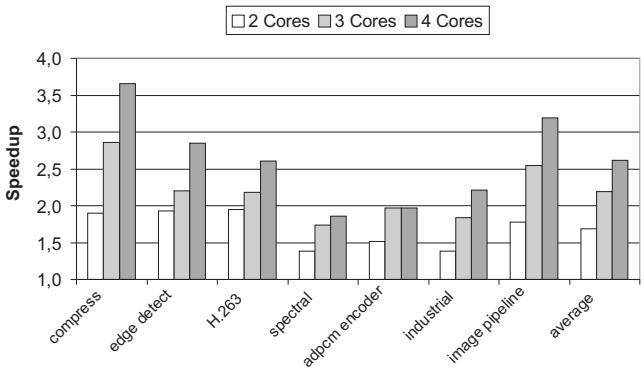

Figure 6: Speedup of parallelization compared to sequential application code

6. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our parallelization approach, we present results for seven benchmarks stemming from the UTDSP suite [13], some miscellaneous real-life applications like an implementation of the H.263 standard and an industrial application used in INTRACOM TELECOM's Wimax system.

To evaluate the performance gain of the parallelization tool, we used the cycle accurate MPARM simulator. Compared to a high-level simulator, this cycle accurate one yields highly precise results. Nevertheless, some additional work had to be done to execute the parallelized code on the simulator. The resulting code of the ATOMIUM tools requires a special run time library (RTLIB), which had to be ported to the MPARM platform. Furthermore, the RTEMS operating system had to be extended e.g., by cross-core thread creation and other multi-core functionality. To build a bridge between the operating system and the RTLIB, a library called R2G (RTEMS and RTLIB Glued Together) [9] was used.

The presented results include 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. Thus, the results focus on the execution time of the application itself. To summarize the potential of the presented approach, the measured execution times are visualized in Figure 6. The figure shows results for configurations of the simulator with one up to four cores. Due to restrictions of the simulator, at most four processors could be used simultaneously. The presented speedups are compared to the sequential execution times of the applications on a single core. The amount of extracted, concurrently executed tasks was limited to the number of available cores. The task creation- and the communication overhead were determined empirically for the MPARM platform and had been set to 20,000 and 100 units, respectively.

As can be seen in the figure, the speedup scales very well with the given amount of processor units for most of the benchmarks. Only the *adpcm encoder* does not benefit of a fourth core, because it was not possible to extract more coarse grained parallelism for that benchmark. Nevertheless, a speedup of 1.5x and nearly 2x could be reached on a two- and three-core architecture. The best utilization could be achieved for the *compress* benchmark of the UTDSP bench-

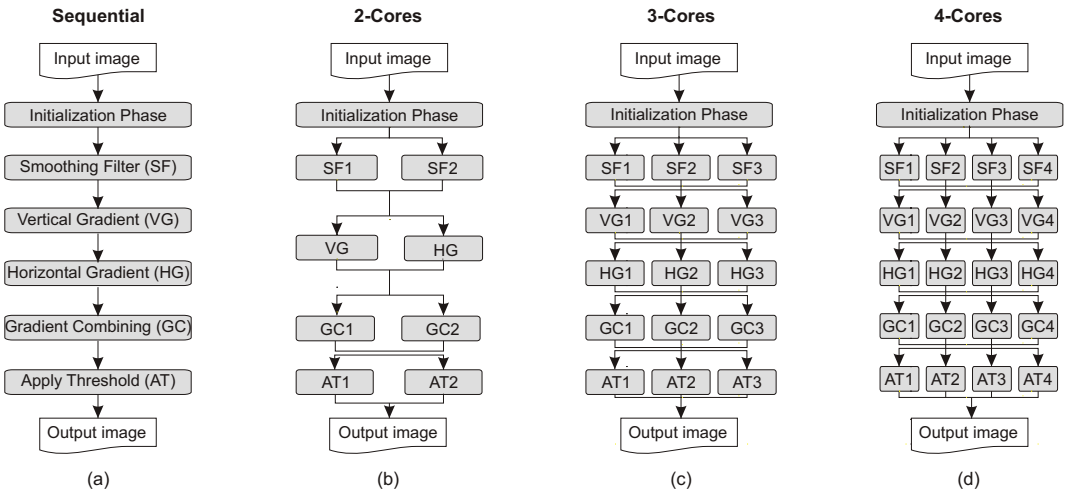

Figure 7: Parallelization of the *edge detect* benchmark

mark suite. The achieved speedup amounts to 1.9x, 2.9x and 3.7x for a two-, three- and four-core system, respectively. The high speedup results from the fact, that the parallelization tool was able to spread the main computational loop to concurrently executed tasks. In addition to the presented real-life benchmarks, we also validated our approach with an industrially used embedded application which is part of INTRACOM TELECOM's Wimax system. Compared to the sequential version, it was possible to accelerate even this application by a factor of 1.4, 1.9 and 2.2 for the given architectural configurations, which emphasizes the usability of the presented approach for industrial grade applications.

The average speedup of the presented benchmarks is visualized in the last column of Figure 6. It was possible to speed the different applications on average up by a factor of 1.8, 2.2 and 2.7, respectively. It should also be mentioned, that the tool did not violate any data dependencies by generating parallel solutions based on the profiling driven dependency analysis.

Exemplary Description of Parallelization

In order to present the conceptual options of possible parallel solutions exploited by our tool, the structural results of the *edge detect* benchmark are visualized in Figure 7. This application provides a good trade-off between computational work and simplicity in its structure.

The aim of the benchmark is to detect edges in a two-dimensional image. The structure of the sequential application is displayed in Figure 7(a). After a short initialization phase, three filters are applied to the original image in order to extract horizontal and vertical edges within the image. All three filters are applied by the same function `convolve2d` which is just called with different input filters. The result of the smoothing filter (the first call) is used as input for the second two function calls. The two filtered images are then combined and compared to a predefined threshold. This is done in a loop which is iterating over the image dimensions.

The result of the parallelization tool, limiting the number of concurrently executed tasks to two, is visualized in Figure 7(b). As can be seen, the `convolve2d` function is parallelized using two tasks to compute the result of the smoothing filter. In contrast to the most obvious solution of taking the same parallelized function for the two other filter calls, the parallelization tool decided to duplicate the `convolve2d` function, to still have one sequential version available. This one is then used for the horizontal and vertical filters, because the tool detected that maximum benefit is obtained by parallelizing both function calls instead of just parallelizing the functions itself. After both function calls returned, the final combining and threshold computation loop is also divided into two concurrently executed tasks. As can be seen, nearly the whole application is executed in parallel, while the number of concurrently executed tasks is limited to two. This results in the presented speedup of 1.9x.

The results for the three and four core versions are visualized in Figure 7(c) and 7(d). In contrast to the two core version, the parallel results of the vertical and horizontal function calls are changed. The version with two concurrent function calls is no longer used, since the speedup is less than using a parallelized version of the function, running three and four tasks concurrently. The achieved speedup with three concurrently executed tasks is 2.2x, while the measured speedup of four tasks is 2.9x. We also observed that the speedup is increasing for growing input image sizes, because the contribution of the initialization phase to the overall execution time is decreasing. In addition to these results, this example also demonstrated the described trade-off technique of our approach, which checks which parallelism is most suitable for the given architectural configuration.

Optimization time

The optimization time of the parallelization tool strongly depends on the profiling based dependency analysis. Since the results of the sub-tools can be stored within the MACC framework, they can be used for different executions of the parallelization step. The optimization time is visualized in Table 2, while the optimization time of the preprocessing tools (column two) includes the execution time of the *Code optimization*, the *Dependency analysis* and the *Execution time estimation* tools. The optimization time of the parallelization approach is given in column three, while column four lists the number of nodes in the hierarchical graph.

Benchmark	Preproc.	Parallelization	Nodes
compress	03:46 min	07 sec	1393
edge detect	00:33 min	06 sec	577
H.263	05:06 min	04 sec	210
spectral	00:22 min	02 sec	444
adcpm encoder	00:10 min	02 sec	352
industrial app.	11:30 min	22 sec	771
image pipeline	00:37 min	02 sec	117

Table 2: Optimization time

As can be seen in Table 2, the ILP based parallelization step performs very fast even if it generates and solves several ILPs for each node, whose number is given in the last column. This is also a proof for the applicability of the combination of the hierarchical task graph and the comprehensive ILP formulations.

7. CONCLUSIONS AND FUTURE WORK

To our best knowledge, this paper presents the first ILP based parallelization approach on hierarchical task graphs tailored to the particular needs of embedded systems. The introduced parallelization tool is able to automatically steer the task granularity for a given hardware platform to exploit the best suitable parallel version of the given sequential application. The efficiency of the tool was demonstrated on seven real-life benchmarks from typical embedded system application domains like e.g., audio-, image- and video-processing, as well as on an industrial application. It could be shown that the tool was able to achieve a speedup of up to 3.7x for a four core system.

In the future, we would like to extend the presented approach to heterogeneous systems. In the current version, the tool takes only one extracted execution time per node into account while partitioning the application. By adding different cores to the same system, this approach has to be adapted to differing execution times, depending on the mapped core. Furthermore, we would also like to combine this coarse grained approach with a finer grained loop level parallelization technique to achieve the most possible speedup for an application.

8. REFERENCES

[1] R. Baert, E. Brockmeyer, S. Wuytack, et al. Exploring parallelizations of applications for MPSoC platforms using MPA. In *Proc. of DATE*, 2009.

[2] L. Benini, D. Bertozzi, A. Bogliolo, et al. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing Systems*, 41(2):169–182, 2005.

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

[4] R. Chandra, D.-K. Chen, R. Cox, et al. Data distribution support on distributed shared memory multiprocessors. *ACM SIGPLAN Notices*, 32(5):334–345, 1997.

[5] 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.

[6] M. Girkar and C. D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[7] 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.

[8] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, et al. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, 1996.

[9] A. Heinig. R2G - RTEMS and RTLib Glued Together. http://ls12-www.cs.tu-dortmund.de/staff/heinig/research/projects/r2g/, April 2010.

[10] IBM. IBM - High-performance mathematical programming engine - IBM ILOG CPLEX - Software. http://www-01.ibm.com/software/integration/optimization/cplex/, April 2010.

[11] ICD e.V. ICD-C Compiler framework. http://www.icd.de/es/icd-c/icd-c.html, April 2010.

[12] 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.

[13] C. G. Lee. UTDSP Benchmark Suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, April 2010.

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

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

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

[17] OpenMP. The OpenMP API specification for parallel programming. http://www.openmp.org/, April 2010.

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

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

[20] R. Pyka, F. Klein, P. Marwedel, et al. Versatile System-level Memory-aware Platform Description Approach for embedded MPSoCs. In *Proc. of LCTES*, 2010.

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

[22] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5-6):779–804, 1991.

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

[24] 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.

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