Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs*

Daniel Cordes, Olaf Neugebauer, Michael Engel, Peter Marwedel
TU Dortmund University
Dortmund, Germany
firstname.lastname@tu-dortmund.de

***Abstract*—Heterogeneous multi-core platforms are increasingly attractive for embedded applications due to their adaptability and efficiency. This proliferation of heterogeneity demands new approaches for extracting thread level parallelism from sequential applications which have to be efficient at runtime. We present, to the best of our knowledge, the first Integer Linear Programming (ILP)-based parallelization approach for heterogeneous multi-core platforms. Using Hierarchical Task Graphs and high-level timing models, our approach manages to balance the extracted tasks while considering performance differences between cores. As a result, we obtain considerable speedups at runtime, significantly outperforming tools for homogeneous systems. We evaluate our approach by parallelizing standard benchmarks from various application domains.**

I. Introduction

The designs of state-of-the-art multiprocessor systems-on-chip (MPSoCs) show an interesting pattern: they depart from traditional homogeneous multicore architectures towards heterogeneity in processing speed. Still, many of these designs maintain binary compatibility between cores in order to ease software development. Early ideas on these so-called heterogeneous same-ISA multicores have been published by Kumar et al. [1]. Recently, the idea has caught on in industry, resulting in designs like ARM's big.LITTLE architecture [2], using Cortex A15 and A7 cores, as well as NVidia's Tegra 3 with five Cortex A9 cores at different clock speeds or Texas Instruments' OMAP4 using Cortex A9 cores and slower Cortex M3 cores for task offloading.

While multi-core architectures have been available for a significant time, the state of software development for these platforms leaves much to be desired. Even today, many applications are developed using a sequential mindset and subsequent manual parallelization. On homogeneous platforms, this approach has already shown to be time-consuming and error-prone, prompting the development of automatic parallelization tools that extract parallelism for multicore systems on different levels of granularity. For heterogeneous systems, the task of parallelizing a given application becomes even more complex than in the homogeneous case, since the execution time required for a given section of code differs between cores. Hence, manual parallelization of sequential code and balancing of tasks becomes a nearly infeasible problem for developers. As a consequence, efficient automatic parallelization becomes indispensable when targeting heterogeneous platforms. We show in Section II that most state of the art parallelization tools so far only consider homogeneous platforms or require a significant amount of manual interaction to target heterogeneous systems. While the heterogeneous systems considered behave identically on the ISA level, performance on a given core depends on the core's clock speed. Additional architectural parameters like, e.g., the pipeline structure, may also be influential. However, for performance reasons, evaluating highly precise low-level models of CPU cores with different architectural properties is usually infeasible due to the analysis overhead involved. Thus, efficient parallelization based on adequate high-level timing models is an interesting alternative.

In this paper, to the best of our knowledge, we present the first, efficient parallelization approach for heterogeneous MPSoC platforms. Our approach extracts task-level parallelism which is automatically balanced by the consideration of execution time differences for the various cores of a heterogeneous MPSoC. Its central property, the use of a Hierarchical Task Graph, is described in Section III. It allows the tool to reconsider different combinations of code sections on different granularity levels like instructions, loop iterations, or functions. This enables adaptation to heterogeneities in processing speed for different sections of code. The large design space inherent in parallelization for heterogeneous multicores is explored by using a state-of-the-art, Integer Linear Programming (ILP)-based approach, described in detail in Section IV. While ILP solving is NP-hard in general, the results presented in Section VI show that on the one hand, the ILP-based solution is able to provide significant speedups for a range of typical benchmark applications while on the other hand, the time overhead required to solve the ILPs at design time is still in the order of minutes on a current development platform. Here, our approach significantly outperforms existing approaches for homogeneous platforms. Details on the evaluation environment and additional tools used in our parallelization framework are described in Section V and an outlook to an integration of additional parallelization techniques is given in Section VII.

To summarize, the main contributions of this paper are:

1) To the best of our knowledge, this paper presents the first approach which uses ILP to exploit task-level parallelism for heterogeneous multiprocessor systems.
2) Our approach is enriched with an adequate cost model embedded in an ILP system. This enables automatic control of the granularity of the extracted parallelism.
3) The created tasks can be automatically balanced for processing units with different performance characteristics.

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

0190-3918/13 $26.00 © 2013 IEEE
DOI 10.1109/ICPP.2013.113

II. Related Work

In recent years, technical improvements as well as limitations in, e.g., processing speed and energy consumption led to an increasing adaption of multiprocessor systems-on-chip (MPSoC) devices, replacing traditionally used single-core platforms. However, most embedded applications are written in sequential C code so that coarse-grained software-based parallelism has to be extracted to benefit from multiple cores on one die. As a result, many techniques have been developed which automate or at least simplify this task.

A representative parallelization approach was published by Hall et al. [3] which automatically extracts task-level parallelism as part of the SUIF Parallelizing Compiler framework [4]. Their approach extracts parallelism which is not limited to function boundaries. A different approach was presented by Ceng et al. who developed a semi-automatic parallelization assistant [5]. The application code is transformed into a weighted statement control data flow graph which is subsequently processed by a heuristically clustering algorithm. The approach proposed by Ceng is semi-automatic since it requires a user feedback loop to steer the granularity of the parallelized program. Our previous publications [6], [7] presented two techniques which extract task-level parallelism with Integer Linear Programming and Genetic Algorithms for single and multiple objectives, like, e.g. execution time and energy consumption. Unfortunately, these approaches generate suboptimal results for heterogeneous architectures since they do not distinguish between different performance characteristics of the available processing units.

Several other approaches also try to extract coarse-grained task-level parallelism from sequentially written applications. E.g., Verdoolaege et al. [8] present a technique which transforms sequential applications into Kahn Process Networks which implicitly describe the parallelism of applications. This approach can only parallelize a restricted set of applications since all loops of the application have to be affine, which is not always given for real-life code. The approach is also used in the Deadalus framework [9] and the MADNESS project [10] to extract the required KPNs. Sarkar [11] has introduced parallelization techniques based on program dependence graphs and extensions of them. Polychronopoulos et al. demonstrated the usefulness of hierarchical task graphs for an automatic scheduling algorithm [12]. A slightly modified version of this hierarchical task graph is also employed by our own approach as described in Section III. Other frameworks, like, e.g., DOL [13], require already parallelized applications as input to map them to MPSoCs.

Raman et al. [14] and Tournavitis et al. [15] presented techniques which are able to automatically extract pipeline parallelism from loops of sequentially written applications. Both approaches split loops into different pipeline stages and further increase an application's performance by splitting stateless pipeline stages into additional tasks.

Many parallelization approaches have been discussed so far. However, all of them are restricted to homogeneous architectures since none of them considers problems which arise if applications should be parallelized for heterogeneous architectures. Thus, they will perform worse as soon as processing units vary in different performance characteristics. This observation was also made by Pienaar et al. [16] who presented an automatic heterogeneous pipelining (AHP) approach. Their framework uses sequentially written, annotated C++ code as input. Execution times for each task are extracted by profiling. As output, it generates pipelines, optimized in terms of throughput, and also creates a suitable schedule. However, the user has to manually extract and annotate parallel sections which should be moved to pipeline stages. In contrast, the approach presented in this paper automatically extracts task-level parallelism and balances it for processing units with different performance characteristics. The trend towards heterogeneous architectures can also be observed in state-of-the-art parallel programming languages. One prominent example is the addition of support for heterogeneous platforms to the widely used OpenMP API [17]. Using code annotations, the application designer is enabled to manually specify that a specific task should be executed on a fixed processing unit.

While previous work shows promising approaches for homogeneous platforms, our analysis of related work suggests that parallelization approaches optimized for heterogeneous platforms, especially embedded ones, mostly do not exist. One exception is the work of [16]. However, this approach does not automatically extract parallelism from sequentially written applications.

III. Framework & Approach

We have chosen our previously published parallelization framework [6], [7] as a starting point for the heterogeneous parallelization approach presented in this paper. Our previous techniques automatically balance extracted tasks for homogeneous architectures by combining estimated information about execution time, communication and task creation costs into high-level models. However, the current approaches are not able to distinguish different execution times for portions of the application depending on the executing processing unit. This is crucial to efficiently parallelize applications for heterogeneous architectures. Therefore, our new approach presented in this paper is able to consider different performance characteristics of the available processing units and to automatically balance the workload into appropriate tasks. In addition, it combines parallelism extraction with a pre-mapping of tasks to processor classes representing identical processor types of the heterogeneous architecture. Thus, tasks can be optimized for specific processing units directly in the parallelization process which was, to the best of our knowledge, not considered by existing approaches so far.

A. Augmented Hierarchical Task Graph

The parallelization process starts with the extraction of a so-called *Augmented Hierarchical Task Graph* from the application's source code (cf. Figure 1). The hierarchical structure of the graph correlates to the hierarchical structure of the application's source code. Each node of the graph represents one statement of the application. The graph contains *Simple Nodes* which do not incorporate any further hierarchical structures and represent, e.g., an assignment statement ($a = b$). In contrast, *Hierarchical Nodes* represent statements which do contain other statements deeper in the hierarchy, like, e.g., a loop containing other statements in its loop body. Hierarchical nodes contain a *Communication In-* and a *Communication*

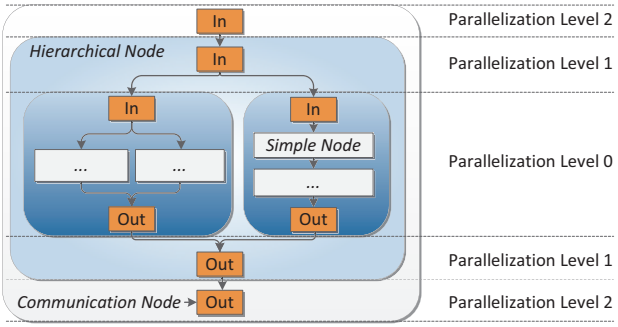


Fig. 1. Augmented Hierarchical Task Graph

Out-Node which encapsulate communication from outside of the hierarchical node to any child nodes and vice versa. Thus, each hierarchical node can be processed in isolation in the parallelization process which drastically reduces the complexity of the parallelization problem. The control flow of the application is directly represented by the hierarchical structure of the graph. Data-Flow edges are also part of the graph and denote communication if source and target node are executed in different tasks. By construction, all leaves of the graph are *Simple Nodes* and annotated with iteration counts and cost information like, e.g., execution costs. This information is automatically extracted by target platform simulation. However, to parallelize software for heterogeneous architectures, it is necessary to distinguish between different execution times depending on the allocated processor class. Thus, our new approach extracts this information once per processor class. All edges are also annotated with information like the amount of communicated data, the edge type, the iteration count, etc. We adapt the technique for extracting the Hierarchical Task Graph from sequentially written ANSI C-code from [6].

B. Parallelization Algorithm

The structure of the heterogeneous parallelization approach is depicted in Algorithm 1. The main function expects an intermediate representation of the source code and a target platform description as input parameters. The platform description [18] contains information about, e.g., performance characteristics of the available processing units and interconnects. After the Hierarchical Task Graph is extracted in line 2, the parallelization approach starts to extract parallelism in a bottom-up search strategy (cf. *Parallelization Level 0-2* in Figure 1) by calling the function PARALLELIZE for the root node in line 3.

The pseudo-code of the function PARALLELIZE starts in line 6. It returns the sequential version of the statements mapped to the different processing units as its only solutions (line 9) if it is a *Simple Node* without any nodes deeper in the hierarchy. If the node contains further hierarchical structures, the function is recursively called for all child nodes in line 12, first. This ensures that all child nodes are already processed before new parallelism is extracted on the current hierarchical level. For each node, a set of parallel solution candidates (res) is created containing all extracted solutions representing parallel versions of the node to be parallelized. Each parallel solution candidate is tagged by the processor class executing the main task and contains information about the extracted node-to-task mapping, the number of inner tasks,

Algorithm 1 Pseudo Code of Global Parallelization Algorithm

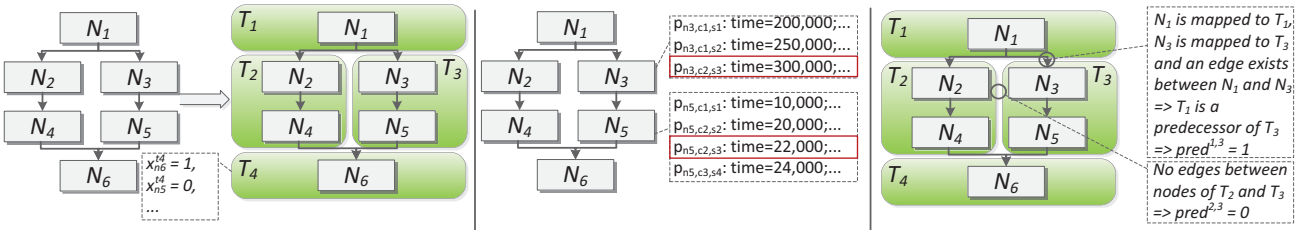
```
1:  function MAIN(IR ir, Platform pf)
2:      htg ← EXTRACTGRAPH(ir, pf)
3:      solutions ← PARALLELIZE(htg.getRootNode(), pf)
4:      IMPLEMENTBESTSOLUTION(htg, pf, solutions)
5:  end function
6:  function PARALLELIZE(Node n, Platform pf)
7:      res ← n.getSequentialSolutions(pf)
8:      if ISNOTHIERARCHICALNODE(n) then
9:          return res
10:     /* Parallelize bottom-up in hierarchy, first. */
11:     for all c ∈ n.getChildNodes() do
12:         PARALLELIZE(c, pf)
13:     /* Parallelize this node now (all child nodes processed). */
14:     for all seqPC ∈ pf.getProcClasses() do
15:         i ← pf.getNumCores()
16:         while i > 1 do
17:             r ← ILPPAR(n, n.getChildNodes(), seqPC, i, pf)
18:             res ← res ∪ {r}
19:             i ← NUMBEROFTASKS(r) − 1
20:         end while
21:     end for
22:     return res
23: end function
```

the execution time of the parallelized (or sequentially executed) node as well as the task-to-processor class mapping which is crucial if applications should be parallelized for heterogeneous architectures. These solutions are extracted by the ILPPAR function called in line 17 (described in Section IV) which extracts new tasks by moving child nodes to concurrently executed tasks. In addition, the ILP-based approach combines newly extracted tasks with tasks which were found deeper in the hierarchy – if it increases the performance. Since ILP-based approaches always return the best solution only, the approach is executed multiple times for different processing units ($seqPC$) used for the main task's execution. In addition, an upper bound of allocatable processing units (i) is also passed to the function in line 17 which is decreased in every iteration by at least one. Thus, the algorithm extracts multiple solutions with different allocated processing units. This yields great flexibility on the next hierarchical level when new tasks are combined with tasks which were found deeper in the hierarchy.

On the next hierarchical level, the ILP-based parallelization approach chooses one parallel solution candidate for each child node (which may contain parallelism which was found deeper in the hierarchy) and combines the candidates with new parallelism on the current hierarchical level, if the solution increases the performance. This step is repeated until the root node of the graph is reached so that the most efficient parallel solution candidate is finally implemented in line 4 of the MAIN function. It should be mentioned here, that the hierarchical structure of the algorithm drastically reduces the complexity of the parallelization problem which enables the usage of sophisticated parallelization algorithms, like the repetitively called ILP-based one described in the following Section.

IV. PARTITIONING AND MAPPING MODEL

Integer linear programming is a well-known technique for partitioning problems which makes it possible to parallelize sequentially written applications. Many commercial as well as open source solvers exist which are able to solve most



(a) 4.1 Node in Task Constraint **(b) 4.2 Parallel Set Constraint** **(c) 4.3 Predecessor Constraint**

Fig. 2. Graphical Representation of the Heterogeneous ILP-based Parallelization Approach (Part 1)

ILPs very efficiently, even if ILPs are NP-hard in the general case. Another advantage of using ILPs is that the solvers guarantee to find the optimal solution if one exists and they can determine that they found it. This is not the case for other optimization techniques like, e.g., Genetic Algorithms which just iterate until a given stopping criterion is met. Our new ILP-based parallelization approach is optimized for heterogeneous architectures and based on the homogeneous one presented in [6]. Therefore, we highlight the changes made and show in Section VI that our new approach significantly outperforms the existing homogeneous one. Our new ILP-based heterogeneous parallelization approach covers five main targets:

I) Map statements of direct child nodes into newly extracted, disjunctive tasks to reduce the overall execution time by parallel execution.
II) Combine newly extracted tasks with tasks which were extracted deeper in the hierarchy, if such a solution increases the overall performance (Parallel Set Mapping).
III) Keep track of dependencies which may change if child nodes representing statements are moved from one task to another one.
IV) Minimize the overall execution time by taking task creation and communication overhead as well as task execution costs depending on the mapped processor class into account.
V) Create a mapping of tasks to processor classes to take care that solutions are well balanced, even for architectures containing processing units with differing performance characteristics.

In the following ILP formulas decision variables are written in lower case letters, sets start with a capital letter and constants contain exclusively capital letters. Indices n and o are used for child nodes of the node to be parallelized, t and u represent indices for tasks, while c represents a processor class. A graphical representation of most equations is also given in Figures 2-5 which visualize the decision variables and constraints used. The sub-figures have the same titles as the corresponding subsections.

A. Node in Task Constraint

The ILP-based parallelization approach is executed for each hierarchical node in isolation. Target I of the approach is a mapping of child nodes to newly extracted, concurrently executed tasks. Therefore, a decision variable x_n^t is created in Equation 1 which denotes whether child node n is mapped to task t.

$$x_n^t = \begin{cases} 1, & \text{if node } n \text{ is mapped to task } t \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Since the presented approach extracts task-level parallelism, each child node should be mapped to exactly one task so that it is executed exactly once:

$$\forall n \in Nodes : \sum_{t \in Tasks} x_n^t = 1 \quad (2)$$

B. Parallel Set Constraint

As explained in the previous section, all child nodes are already processed by the ILP-based parallelization approach since the extraction algorithm parallelizes the application in a bottom-up manner. As a result, all profitable parallel solution candidates were collected in a so-called parallel set for each child node. The algorithm has to choose one solution for each child node which may contain tasks which were extracted deeper in the hierarchy (Target II). Thus, newly extracted tasks are combined with tasks which were found deeper in the hierarchy. Each solution candidate has a different execution time depending on the number of extracted tasks as well as its internal task-to-processor class mapping. This mapping dimension was added to our new approach since it is crucial to distinguish between different performance characteristics of the available processing units in heterogeneous architectures. Equation 3 defines variable $p_{n,c,s}$ which evaluates to 1 if parallel solution s of node n executed on processor class c is chosen.

$$p_{n,c,s} = \begin{cases} 1, & \text{if parallel solution } s \text{ of child node } n \\ & \text{executed on processor class } c \text{ is chosen} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

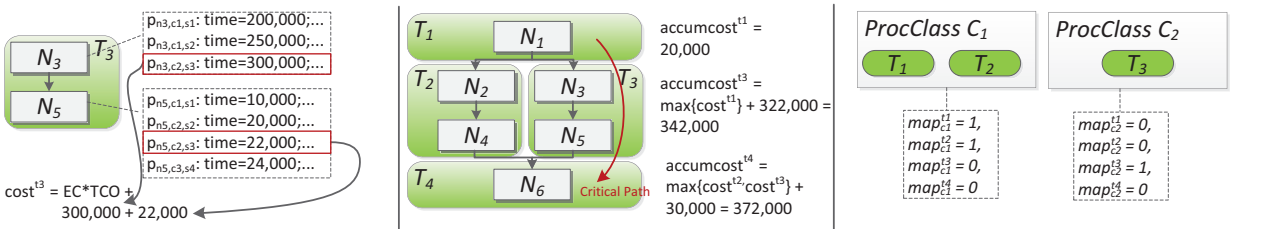
Equation 4 takes care that exactly one hierarchical parallel solution candidate is chosen for each child node.

$$\forall n \in Nodes : \sum_{c \in ProcClasses} \sum_{s \in Solutions_{n,c}} p_{n,c,s} = 1 \quad (4)$$

C. Predecessor Constraint

Parallel execution is often prohibited by data- or control-flow dependencies which create a predecessor and successor relationship between tasks. This relationship has to be explicitly modeled since the critical (or most expensive) path for the execution within the hierarchical node to be parallelized should be extracted. This is important since the ILP's objective is to reduce the execution time of the critical path by executing child nodes in parallel on different processing units. Equation 5 defines decision variable $pred^{t,u}$ which evaluates to 1 if task t is a direct predecessor of task u.

$$pred^{t,u} = \begin{cases} 1, & \text{if task } t \text{ is a predecessor of task } u \\ 0, & \text{otherwise} \end{cases} \quad (5)$$



(a) 4.4 Execution Costs of Task Constraint **(b) 4.5 Path Cost Constraint** **(c) 4.8 Mapping of Tasks to Processor Classes**

Fig. 3. Graphical Representation of the Heterogeneous ILP-based Parallelization Approach (Part 2)

The relation between tasks must be expressed in the ILP as claimed by Target III and depends on the node-to-task mapping, modeled by decision variable x_n^t (Equation 1). If a dependence edge from node n to node o exists ($EDGE_{n,o}$=1) and both nodes are mapped to different tasks t and u, then task t is a direct predecessor of u, like denoted in Equation 6.

$$\forall t, u \in Tasks : \forall n, o \in Nodes : t \neq u : n \neq o : \\ pred^{t,u} \geq EDGE_{n,o} * (x_n^t \wedge x_u^o) \quad (6)$$

If task t is a predecessor of task u then u has to wait until t has finished its execution, since u has to consume data produced by task t. The $\wedge$ operator used in Equation 6 is not part of regular ILP formulations. Nevertheless, it can be substituted by a new variable and three inserted constraints as shown in Equation 7.

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

D. Execution Costs of Task Constraint

The ILP-based parallelization approach should automatically balance the extracted tasks by respecting different execution times depending on the performance characteristics of the mapped processing unit. Therefore, costs for each task are calculated like shown in Equation 8.

$$\forall t \in Tasks : cost^t = EC * TCO + \\ \sum_n \sum_c \sum_s (x_n^t \wedge p_{n,c,s}) * COSTS_{n,c,s} \quad (8)$$

Also here, homogeneous approaches are not able to consider mapping decisions which have a big influence on the performance on heterogeneous architectures. Therefore, Equation 8 includes decision variable $p_{n,c,s}$ which selects from different execution costs $COSTS_{n,c,s}$[1] depending on the mapped processor class c. The execution costs $cost^t$ of task t consist of a configurable task creation overhead TCO multiplied by the execution count EC. This overhead is increased by the execution costs $COSTS_{n,c,s}$ of all nodes n which are executed on processor class c and mapped to task t depending on the chosen parallel solution candidate $p_{n,c,s}$. Thus, $cost^t$ contains all execution costs of task t by respecting the mapped processor class.

[1]Variables $COSTS_{n,c,s}$ were calculated deeper in the hierarchy and are thus constants in the parallel configurations.

E. Path Cost Constraint

Based on predecessor relationships and calculated costs of the extracted tasks, it is possible to describe path costs which contain the execution costs of all predecessor tasks and task t itself, like shown in Equation 9.

$$\forall t, u \in Tasks : pred^{u,t} = 1 \Rightarrow t \neq u : \\ accumcost^t \geq cost^t + accumcost^u + commcost^u \quad (9)$$

The accumulated path costs $accumcost^t$ of task t are equal to the execution costs of t itself increased by execution and communication costs of the *most expensive* predecessor task u. The precondition $pred^{u,t} = 1$ can be ensured by subtracting a constant from the right-hand side of the constraint whose value is greater than the sum of all other possible values, if the precondition is not met like shown in [6].

F. Cycle-Free Constraint

To avoid deadlocks and paths with infinite costs, the approach has to take care that the paths within the node to be parallelized are cycle-free. Therefore, all direct child nodes are topologically sorted by their dependencies and an ascending, unique id is generated for both, nodes ($nodeid$) and tasks ($taskid$). W.l.o.g., the generated task graph is cycle-free, if the $taskid$ of node n is greater or equal to the $taskid$s of all nodes o with a smaller $nodeid$. This is shown by Equation 10.

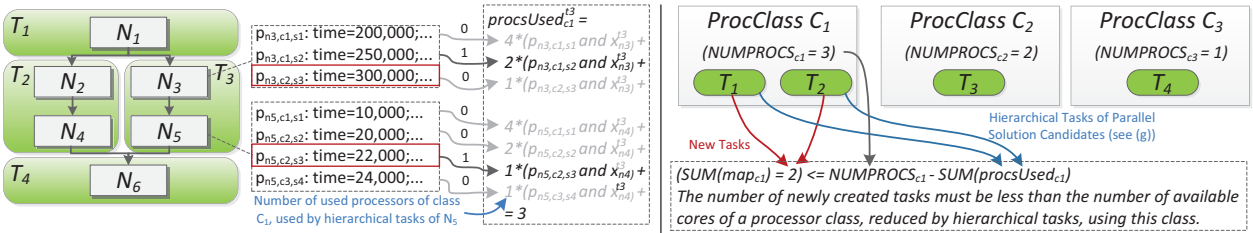
$$\forall n, o \in Nodes : nodeid_n \geq nodeid_o : \\ taskid_n \geq taskid_o \quad (10)$$

G. Objective Function

The main objective of the ILP-based parallelization algorithm is to minimize the execution time of the hierarchical node by moving statements of child nodes into concurrently executed tasks combined with tasks extracted deeper in the hierarchy. As already explained in Section III, each hierarchical node contains a Communication In- and Communication Out-Node. The latter one is a successor of all child nodes since it encapsulates the communication which leaves the hierarchical node. Thus, the overall execution time of the parallelized node is equal to the path costs of the task, which the sequential out-node is mapped to, like shown in Equation 11.

$$exectime = \min\{accumcost^{t_{seqOut}}\} \quad (11)$$

The objective function's execution time contains execution, task-creation, and communication costs as claimed by Target IV. The presented approach tightly couples task extraction



(a) 4.9 Available Processing Units per Class **(b) 4.10 Limit Allocated Processing Units**

Fig. 4. Graphical Representation of the Heterogeneous ILP-based Parallelization Approach (Part 3)

with mapping decisions which is crucial for heterogeneous architectures into one model. The remainder of this section presents the way mapping decisions are taken and how they are combined with the task extraction approach.

H. Mapping of Tasks to Processor Classes

Up to now, newly extracted tasks are not mapped to any processor class. This was not necessary for homogeneous architectures, but has a huge impact on the execution time for heterogeneous ones. Therefore, the presented approach of this paper combines the extraction of parallelism with a mapping of tasks to processor classes which represent identical processing units of the targeted heterogeneous architecture. This enables the extraction of well-balanced tasks which are optimized for a given processor class like demanded by Target V. A new decision variable map_c^t is introduced which evaluates to 1 if task t is mapped to processor class c like shown in Equation 12.

$$map_c^t = \begin{cases} 1, & \text{if task } t \text{ is mapped to processor class } c \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

Each task t has to be mapped to exactly one processor class c, which is ensured by Equation 13.

$$\forall t \in Tasks : \sum_{c \in ProcClasses} map_c^t = 1 \tag{13}$$

I. Available Processing Units per Class

By taking advantage of platform information in the task extraction step, it is possible to avoid additional scheduling overhead. Therefore, each processing unit is used by either newly extracted tasks or tasks which were extracted deeper in the hierarchy. The number of already allocated processing units of a processor class c used by chosen hierarchical solutions must be determined for each task t. The constant $USEDPROCS_{s,c}$ represents the number of allocated processing units of class c for hierarchical parallel solution candidate s. Equation 14 defines variable $procsused_c^t$ which stores the amount of already allocated processing units on basis of $USEDPROCS_{s,c}$ for all processor classes c, used by the child nodes mapped to task t.

$$\begin{gathered} \forall c \in ProcClasses : \forall t \in Tasks : \\ \forall n \in Nodes : \forall s \in Solutions_{n,c} : \\ procsused_c^t \geq USEDPROCS_{s,c} * (p_{n,c,s} \wedge x_n^t) \end{gathered} \tag{14}$$

With $procsused_c^t$ it is possible to calculate the amount of processors which are still available for allocation of newly extracted tasks, like shown in Equation 15.

$$\begin{gathered} \forall c \in ProcClasses : numPC_c = \\ NUMPROCS_c - (\sum_{t \in Tasks} procsused_c^t) \end{gathered} \tag{15}$$

The constant number of available processing units $NUMPROCS_c$ per processor class c is derived from the supplied platform information.

J. Limit Allocated Processing Units

So far, all newly extracted tasks could be mapped to the fastest processor class even if not enough processing units of this class for parallel execution would be available. Therefore, Equation 16 ensures that the number of newly extracted tasks t, mapped to processor class c, does not exceed the number of available processors for each processor class.

$$\forall c \in ProcClasses : \sum_{t \in Tasks} map_c^t \leq numPC_c \tag{16}$$

K. Restrict Solution Candidates

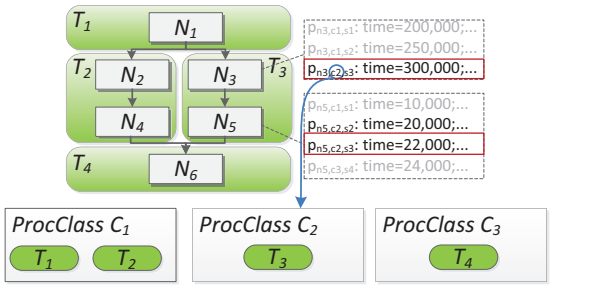
All solution candidates $p_{n,c,s}$ of child node n are tagged with a specific processor class c as explained in Section III. Thus, the ILP must be restricted to choose only one of those solution candidates using the same processor class as the task to which node n is mapped. Therefore, Equation 17 defines decision variable $nodeOnProcClass_{n,c}$ which evaluates to 1 if node n is executed on processor class c with respect to the node-to-task mapping x_n^t and task-to-processor class mapping map_c^t.

$$\begin{gathered} \forall n \in Nodes : \forall c \in ProcClasses : \\ nodeOnProcClass_{n,c} = \sum_{t \in Tasks} x_n^t \wedge map_c^t \end{gathered} \tag{17}$$

Finally, Equation 18 takes care that only those hierarchical parallel solution candidates can be chosen which are valid with respect to the task-to-processor class mapping.

$$\begin{gathered} \forall n \in Nodes : \forall c \in ProcClasses : \\ \sum_{s \in Solutions_{n,c}} p_{n,c,s} = nodeOnProcClass_{n,c} \end{gathered} \tag{18}$$

If the task executing node n is not mapped on processor class c, the sum of all hierarchical solution candidates' decision variables must be equal to zero, which avoids the selection of those solution candidates. Vice versa, if the task of node n is mapped on processor class c, the sum of all hierarchical solution candidates' decision variables must be equal to one,



All Parallel Solution Candidates of the child nodes belong to a specific processor class c. Thus, they can only be chosen if Task t, where node n ist mapped to, is executed on the same processor class.

4.11 Restrict Solution Candidates

Fig. 5. Graphical Representation (Part 4)

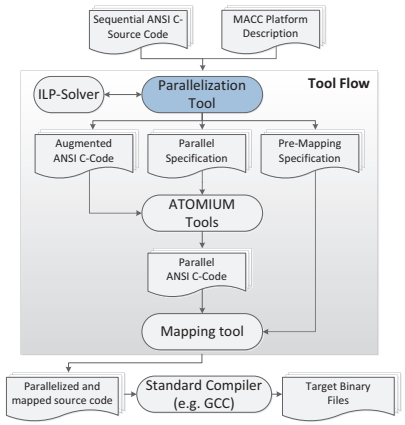


Fig. 6. Parallelization Tool Flow

so that one of those candidates must be chosen. *Note:* The parallel solution set of child node n contains at least one solution candidate for each processor class which represents the sequential execution on this processor class. Thus, it is guaranteed that the ILP finds a solution for this mapping.

L. Summary

The presented ILP-based parallelization approach combines parallelism extraction with a mapping of tasks to processor classes in a clear mathematical model. By using this model the approach is able to automatically find a good balancing of extracted tasks for heterogeneous processors with different performance characteristics and to combine this with a mapping of tasks to processor classes. Due to the use of a hierarchical approach, the runtime complexity of our technique increases only linear with the number of statements of the application to be parallelized. The following Section presents details about the parallelization tool flow, and Section VI shows that our new heterogeneous parallelization approach is able to significantly outperform homogeneous ones.

V. Experimental Environment

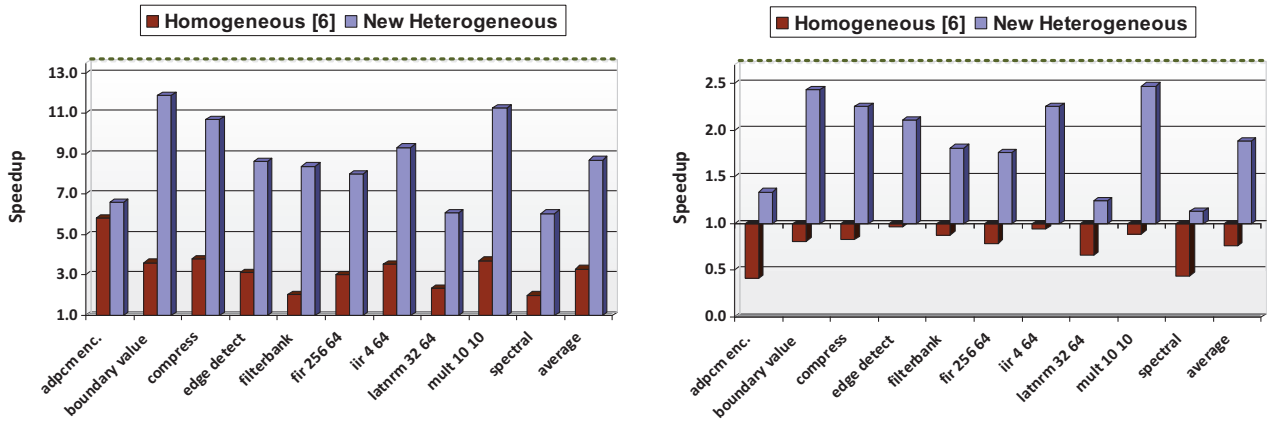
The techniques described in this paper are integrated into an automatic parallelization and compilation tool flow which is depicted in Figure 6. As can be seen, the tool flow expects sequential ANSI-C code together with a platform description of the targeted heterogeneous architecture as input. The parallelization tool automatically extracts the hierarchical task graph like described in Section III before the parallelization process (cf. Section IV) automatically extracts well balanced parallelism from the graph representing the source code of the sequentially written application. More details on the employed data-flow analysis and the extraction of execution times for the statements of the input program can be found in [6]. The generated ILPs which are used as a mathematical model of the parallelization problem are solved by state-of-the-art ILP solvers. In the current version of the tool, the user can choose between the freely available *lpsolve* [19] and IBM's commercially available *cplex* [20] solver. As a result, the parallelization tool annotates the source code of the application to describe the extracted parallelism. The format can either be compliant with the input specification of the *ATOMIUM (MPA)* tools [21] or an extension of *OpenMP* [17] which enables heterogeneous mapping. Figure 6 shows the tool flow which is used if the ATOMIUM tools are employed to implement the extracted parallelism. Here, a parallel specification which maps labeled statements of the application to tasks is also created by the parallelization tool. With both inputs, the ATOMIUM tools automatically implement the extracted parallelism which is further processed by a mapping tool. The presented parallelization tool of this paper optimizes the extracted tasks so that they are automatically balanced even for processing units with different performance characteristics, like in heterogeneous architectures. Therefore, a pre-mapping specification is generated which is passed to the mapping tool. This specification contains information about the extracted task-to-processor class mapping to ensure that tasks are mapped to processing units for which they are optimized.

All tools described so far perform source-to-source transformations. This has the big advantage that the designer is able to observe the applied code modifications after each step. In addition, a standard compiler can be used to compile and optimize the parallelized source code into binary files which are linked against a library that implements task creation and synchronization primitives. The tool flow also contains a link to the cycle accurate CoMET [22] and MPARM [23] MPSoC simulators, so that the sequentially written source code can fully automatically be parallelized, mapped and evaluated on several architectures without manual intervention.

VI. Experimental Results

To evaluate the efficiency of our new automatic parallelization approach for heterogeneous architectures, we present results achieved from the UTDSP benchmark suite [24] containing representative real-world embedded applications. In addition, we also evaluated other meaningful applications like, e.g., the so-called boundary value problem from a physical application domain. To emphasize the quality of our new approach, we compare it with results generated by the existing parallelization approach presented in [6]. Just like any other existing available parallelization approach, the one presented in [6] was optimized for homogeneous architectures.

The heterogeneous target platform was simulated with the cycle accurate CoMET MPSoC simulator [22]. Even though our parallelization approach would also perform well



(a) Accelerator Scenario

(b) Slower Cores Scenario

Fig. 7. Results for Platform Configuration (A): 100/250/500/500 MHz

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

A. Evaluation of Speedup

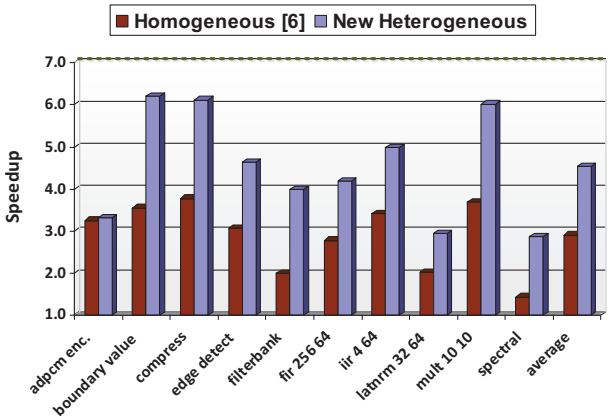
We evaluated the presented platforms for two different application scenarios: (I) The main processor of the platform is a slow core and the additional processing units are added as accelerators. (II) The main processor of the platform is a fast core and the slower processing units are added to the platform due to, e.g., power or thermal issues. The measurement baseline in both scenarios is the sequential execution on the main processor. Figure 7 depicts results for both evaluation scenarios on platform configuration (A) and compares our new heterogeneous parallelization approach to the one presented in [6]. The dashed line shows the theoretical maximum speedup limit for all evaluated platform configurations which can of course never be fully reached due to, e.g., inserted communication and task creation overhead.

Results for the accelerator scenario (I) are shown in Figure 7(a). As can be seen, both approaches increase the performance of all evaluated applications well. Since the homogeneous approach is not aware of different processor types, it tries to uniformly balance the workload for all available processors. Thus, a speedup between 3× up to 4× is achieved for most applications which is a very good performance increase for homogeneous architectures equipped with four processing units. However, the results do not exploit the potential of the targeted heterogeneous platform well. In contrast, results generated by our new heterogeneous approach are much more impressive. It automatically balances the extracted tasks by respecting different performance characteristics of the available processing units. Thus, the two processors with 500 MHz are automatically allocated with heavier workloads than the slower ones. This results in performance increases of up to 11-12× for some of the considered benchmarks (e.g., *boundary value*, *compress* and *mult*) which significantly outperforms the speedup of the homogeneous parallelization tool and is very close to the theoretical maximum speedup of 13.5×[2]. On average, the homogeneous parallelization tool increased the applications' performance by 3.3×. In contrast, our new heterogeneous approach reached an average speedup of 8.7×.

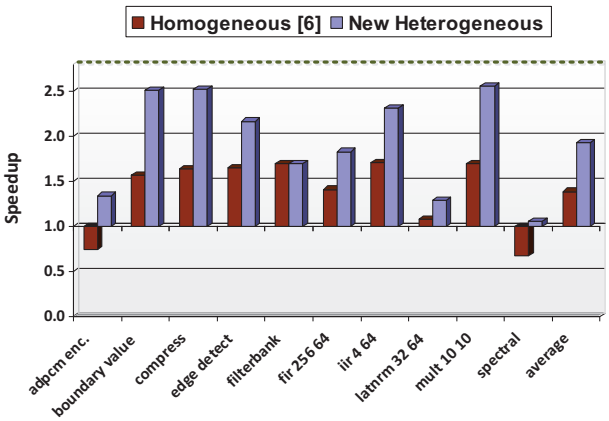
Figure 7(b) shows results for evaluation scenario (II) with a fast main processor (500 MHz) and slower additional cores. Here, the speedup produced by the homogeneous approach is less than one, meaning that the parallelized application performs slower than its sequential version. The reason is that the homogeneous approach uniformly distributes the work to the available processing units. Thus, the faster processors have to wait until the slower cores have finished their tasks. This shows that it is even more challenging to extract beneficial parallelism for an architecture with slower additional cores. However, in contrast to the homogeneous tool, our new heterogeneous parallelization approach was able to speed up the application by generating tasks that perfectly utilize the slower processing units so that all cores finish nearly at the same time. The speedup ranges between 1.2× and nearly 2.5× showing that the approach did not only allocate tasks to the 500 MHz cores and is also very close to the theoretical limit of 2.7×[3].

To highlight the adaptability of our new heterogeneous parallelization approach, Figure 8 on the next page shows additional results for platform configuration (B) which contains two 200 MHz and two 500 MHz cores. Both evaluation scenarios with a slow (I) and a fast (II) main processor are visualized, respectively. As can be seen in Figure 8(a), both approaches perform well for evaluation scenario (I). The homogeneous parallelization approach reached speedups of around 3× for most evaluated benchmarks. In contrast,

[2] $(1 * 100 + 1 * 250 + 2 * 500\text{MHz})/100\text{MHz} = 13.5\times$
[3] $(1 * 100 + 1 * 250 + 2 * 500\text{MHz})/500\text{MHz} = 2.7\times$



(a) Accelerator Scenario



(b) Slower Cores Scenario

Fig. 8. Results for Platform Configuration (B): 200/200/500/500 MHz

our new heterogeneous approach reached speedups of more than 6× for the benchmarks *boundary value*, *compress* and *mult*. The achieved speedups are not as high as the ones extracted for platform configuration (A), since the performance difference between the platform's processing units is not as large, here. The theoretical speedup limit of this platform is 7×[4], the one for platform configuration (A) was 13.5×. Thus, the quality of the results is comparable for both evaluated platforms even if the achieved speedups are different. The homogeneous approach reached an average speedup of 2.9× for this platform, while the heterogeneous one was able to increase the applications' performance by 4.5× on average.

The same observations were made for evaluation scenario (II) whose results are presented in Figure 8(b). The homogeneous parallelization approach reached speedups of up to 1.7×, while the heterogeneous one increased the applications' performance up to 2.6×. Also here, the results are very close to the platform's theoretical speedup limit of 2.8×[5] which emphasizes that our approach balances the workload between the available processing units well by respecting different performance characteristics. However, the performance of some benchmarks (e.g., *latnrm* or *spectral*) can still be improved. Those benchmarks have higher communication loads and the current approach extracts task-level parallelism only but the applications profit more from other parallelism types, like, e.g., pipeline parallelism. Thus, the adaptation of other techniques will be addressed in future work.

B. ILP Statistics

Table I on the following page summarizes collected data for all evaluated benchmarks and compares our new heterogeneous parallelization approach to the homogeneous one, presented in [6]. The table contains information about the time in minutes which was necessary to parallelize the applications with both approaches (*Time*), the number of generated ILPs (#ILPs), the number of created variables for all generated ILPs (*#Var*), and the overall number of created constraints (*#Constr*).

As can be seen, the ILP formulations are more complex in the heterogeneous case since a new dimension was added describing the task-to-processor type mapping. Our new heterogeneous parallelization approach also created and solved more ILPs than the homogeneous one. This is necessary since parallel solution candidates for different processor classes have to be extracted on each hierarchical level. Otherwise, the parallelization process on the parent hierarchical level would be limited which would drastically reduce the solution quality. Besides absolute numbers, shown in the first two blocks (labeled *Homogeneous approach [6]* and *New Heterogeneous approach*) of Table I, a third block (labeled *Factor*) shows the ratio between both approaches. As can be seen, the number of ILPs increases by factors between 2.4× and 7.4× if we move from the homogeneous to the heterogeneous case. The average increase over all evaluated benchmarks is 3.5×. The increase of newly created variables in the heterogeneous case ranges from 4.9× up to 14.8× (7.0× on average), while created constraints are increased by 4.1× up to 11.2× (5.5× on average). Thus, many new variables and constraints had to be added to parallelize applications for heterogeneous architectures. However, if the number of constraints is increased to 5.5× (average case) while the number of generated ILPs is increased to 3.5× (average case), the average increase of constraints per ILP is manageable low (<1.5×). This has also an impact on the time the parallelization approach needs to parallelize an application. The original homogeneous approach had parallelized an application in 8 seconds[6] on average, while the heterogeneous one needs 3:10 minutes[6]. Note that, due to the hierarchical approach, run-times are still remaining in an acceptable state. Nevertheless, the high speedups outweigh the higher execution times at compile time since the approach has to be executed only once in the compilation process. The new heterogeneous approach can also be applied to homogeneous architectures but due to higher execution times it makes more sense to use the approach which is optimized for homogeneous architectures in this case.

To summarize, the following results were achieved:

1) The presented heterogeneous parallelization approach utilizes heterogeneous platforms in an excellent way. Speedups of up to 11–12× could be achieved for evaluation platform (A) in scenario (I).

[4] $(2 * 200 + 2 * 500\text{MHz})/200\text{MHz} = 7\times$

[5] $(2 * 200 + 2 * 500\text{MHz})/500\text{MHz} = 2.8\times$

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

TABLE I. STATISTICS OF ILP-BASED PARALLELIZATION ALGORITHMS

	Homogeneous approach [6]				New Heterogeneous approach				Factor			
Benchmark	Time[7]	#ILPs	#Var	#Constr	Time[7]	#ILPs	#Var	#Constr	Time[7]	#ILPs	#Var	#Constr
adpcm enc.	00:03	23	6,933	12,686	00:15	78	50,631	70,488	5.0×	3.4×	7.3×	5.6×
bound. value	00:05	7	3,409	6,037	00:08	21	18,303	26,832	1.6×	3.0×	5.4×	4.4×
compress	00:21	59	16,348	31,022	12:12	438	242,382	347,448	34.9×	7.4×	14.8×	11.2×
edge detect	00:08	49	12,335	23,310	00:42	141	73,647	108,594	5.3×	2.9×	6.0×	4.7×
filterbank	00:07	6	3,723	6,700	06:27	20	27,918	38,962	55.3×	3.3×	7.5×	5.8×
fir 256	00:01	10	1,266	1,926	00:02	24	7,152	9,192	2.0×	2.4×	5.7×	4.8×
iir 4	00:02	10	7,387	12,877	00:08	24	35,817	52,950	4.0×	2.4×	4.9×	4.1×
latnrm 32	00:02	14	2,356	3,812	00:04	36	13,200	17,568	2.0×	2.6×	5.6×	4.6×
mult 10	00:02	11	2,276	4,307	00:06	45	16,005	23,157	3.0×	4.1×	7.0×	5.4×
spectral	00:24	33	13,427	25,994	11:40	102	74,595	111,212	29.2×	3.1×	5.6×	4.3×
average	00:08	22	6,946	12,867	03:10	93	55,965	80,640	14.2×	3.5×	7.0×	5.5×

2) The combination of mapping decisions with knowledge of heterogeneous performance characteristics in the parallelization approach is highly beneficial since tasks can be directly optimized for specific processing units.
3) Our new heterogeneous approach is able to increase the applications' performance even for platforms with cores which are much slower than the main processor.
4) In contrast to the homogeneous approach, our new heterogeneous one never generated speedups less than one for all benchmarks and significantly outperforms the homogeneous approach on heterogeneous architectures.
5) Even though ILP is NP-hard in the general case, we have shown that, due to the hierarchical approach, execution times of our approach still remain acceptable.

VII. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this is the first approach which combines automatic extraction of task-level parallelism with mapping decisions to efficiently balance created tasks for heterogeneous multiprocessor architectures. The efficiency of the tool was demonstrated on several real-world benchmarks from typical embedded application domains. The measurements, performed on a cycle-accurate MPSoC simulator, have shown that our new approach significantly outperforms existing state-of-the-art parallelization approaches.

In the future we intend to extend our heterogeneous parallelization framework to be able to extract other types of parallelism as well, like, e.g., pipeline parallelism to further increase the applications' performance. In addition, we will also consider taking other objectives into account, like, e.g., energy consumption or code size.

ACKNOWLEDGMENT

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

REFERENCES

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

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

[3] 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.

[4] 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.

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

[6] 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*, 2010.

[7] D. Cordes and P. Marwedel, "Multi-Objective Aware Extraction of Task-Level Parallelism Using Genetic Algorithms," in *Proc. of DATE*, 2012.

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

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

[10] E. Cannella, L. Di Gregorio, L. Fiorin *et al.*, "Towards an ESL design framework for adaptive and fault-tolerant MPSoCs: MADNESS or not?" in *Proc. of ESTIMedia*, 2011.

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

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

[13] L. Thiele, I. Bacivarov, W. Haid *et al.*, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Proc. of ACSD*, 2007.

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

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

[16] J. A. Pienaar, S. Chakradhar, and A. Raghunathan, "Automatic generation of software pipelines for heterogeneous parallel systems," in *Proc. of SC*, 2012.

[17] E. Stotzer, J. Beyer, D. Das *et al.*, "OpenMP Technical Report 1 on Directives for Attached Accelerators," OpenMP Architecture Review Board, Tech. Rep., 2012.

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

[19] M. Berkelaar, E. Kjell, and N. P., "lp solve 5.5," http://lpsolve.sourceforge.net/5.5/, 2013.

[20] IBM, "IBM - High-performance mathematical programming engine - IBM ILOG CPLEX - Software," http://www-01.ibm.com/software/integration/optimization/cplex/, 2013.

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

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

[23] 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.

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

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