Automatic Extraction of Multi-Objective Aware Parallelism for Heterogeneous MPSoCs*

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

***Abstract*—Heterogeneous MPSoCs are used in a large fraction of current embedded systems. In order to efficiently exploit the available processing power, advanced parallelization techniques are required. In addition to consider performance variances between heterogeneous cores, these methods have to be multi-objective aware to be useful for resource restricted embedded systems. This multi-objective optimization requirement results in an explosion of the design space size. As a consequence, efficient approaches are required to find promising solution candidates. In this paper, we present the first portable genetic algorithm-based approach to speed up ANSI-C applications by combining extraction techniques for task-level and pipeline parallelism for heterogeneous multicores while considering additional objectives. Using our approach enables embedded system designers to select a parallelization of an application from a set of Pareto-optimal solutions according to the performance and energy consumption requirements of a given system. The evaluation of a large set of typical embedded benchmarks shows that our approach is able to generate solutions with low energy consumption, high speedup, low communication overhead or useful trade-offs between these three objectives.**

I. Introduction

The proliferation of heterogeneous multiprocessor systems on chip (MPSoC) poses a new challenge for embedded system designers. Heterogeneity allows to obtain speedups while saving energy by executing tasks on slower cores. However, it is crucial to balance parallel tasks on these differing cores. Otherwise, a system would waste computation time and energy due to cores that idle while waiting for data from other tasks. A typical use case in embedded systems is to extract as much speedup as required while staying within a given energy limit.

Previous research has shown that the extraction of efficient parallelism from existing source code for heterogeneous systems is a complex task, since optimizations have to take the difference in processing speed between differing cores into account. When considering additional objectives such as energy consumption, the number of possible solutions explodes, which renders traditional sophisticated optimization approaches infeasible due to the excessive growth in required computing time to find feasible solutions.

In this paper, we present a portable approach based on genetic algorithms that enables embedded system designers to obtain solutions for this optimization problem quickly. Compared to existing parallelization methods, our approach is able to extract both coarse-grained task-level parallelism as well as fine-grained pipeline parallelism found in typical embedded applications. By evaluating cost models, both parallelization granularities can be combined to achieve significant speedups.

The central data structure used for optimization is a hierarchical task graph (HTG), which reflects the structure of a program starting with simple statements at its leaves up to the complete program at its root. The use of an HTG enables solutions that mix coarse- and fine-grained partial solutions in order to match the execution time characteristics of the different cores. When assessing the possible speedup obtainable by parallelization, it is important to also consider the overhead introduced by task creation and communication requirements of the extracted tasks. Spending too much time on communication, e.g., by creating too many fine-grained tasks, might render all improvements obtained by using more processor cores void. For this reason, our optimization approach considers task creation and communication costs, based on high-level cost models, in addition to differences in execution speed. All parameters are configurable to enable an easy adaption of our new approach to various architectures.

For multi-objective optimization, the nodes of the HTG are augmented with information on execution time as well as energy consumption of a given piece of code extracted by target platform simulation. This information forms the basis for finding efficient, Pareto-optimal multi-objective solutions using a genetic algorithm approach.

The evaluation of a set of typical embedded benchmarks shows that using our optimization, speedups close to the theoretical maximum speedup as well as energy-optimized solutions providing reasonable speedups with low communication overhead are obtainable. The generation of a set of Pareto-optimal solution candidates then allows the embedded system designer to select a parallelization solution tailored to the specific requirements of the given system.

To summarize, the main contributions of this paper are:

1) To the best of our knowledge, this paper presents the first approach to extract parallelism for heterogeneous MPSoCs that considers multiple optimization objectives at the same time.
2) Our approach combines the extraction of task-level as well as pipeline parallelism in order to be efficient for embedded systems.

*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

978-1-4799-1010-6/13/$31.00 © 2013 IEEE

3) Our approach is enriched with adequate cost models to evaluate execution time, energy consumption and communication overhead. This enables automatic control of the granularity of the extracted parallelism. Furthermore, the created tasks can be automatically balanced for processing units with different performance characteristics.

The rest of this paper is structured as follows. We give an overview of related work in Section II. A description of our approach and the framework behind it is given in Section III. Section IV details our multi-objective aware parallelization approach, followed by an evaluation in Section V. Finally, Section VI concludes the paper and gives an outlook to future research ideas.

II. Related Work

A lot of effort has been invested in the last decades in research projects developing approaches and tool support to automatically parallelize sequentially written applications for multi-processor systems. The kind of parallelism extracted by those tools can be grouped in different categories. Here, we will discuss two of them, namely *task-level* and *pipeline parallelism* since the approach described in this paper extracts these two kinds of parallelism.

A typical approach which is able to extract coarse-grained *task-level* parallelism fully automatically was presented by Hall et al. [1]. Their technique extracts this kind of parallelism based on an interprocedural analysis. It was later integrated in the SUIF Parallelizing Compiler framework [2]. The approach presented by Hall et al. also applies various code optimizations, like, e.g., privatizations and reduction recognition for arrays and scalar variables, to extract even more efficient parallelism.

A different approach was published by Ceng et al. [3]. Their parallelization assistant works in a semi-automatic manner so that the user has to decide how fine-grained the application should be partitioned. The approach was integrated into the MAPS framework and operates on a weighted statement control data flow graph which is subsequently processed by a heuristically clustering algorithm.

Sarkar [4] also presented an automatic coarse-grained parallelization approach extracting task-level parallelism from sequentially written applications. His approach was integrated into IBM's PTRAN compiler. The parallelism extraction technique employed a so called forward control dependence graph (FCDG) as intermediate representation and evaluated the extracted parallelism by high-level models. However, only simple heuristics were used to merge nodes of the graph to coarser-grained tasks.

Other parallelization approaches extracting task-level parallelism were, e.g., presented by Ottoni [5]. Their approach employs a program dependence graph and extensions of it to detect parallelism in sequentially written applications. Polychronopoulos et al. demonstrated the usefulness of hierarchical task graphs for an automatic scheduling algorithm in [6]. The HELIX system [7], presented by Campanoni et al., is able to parallelize loops automatically from sequentially written application code. A heuristical speedup model extending Amdahl's Law [8] is used to determine loops to be parallelized. Profiling data is used to extract the execution time required by the model.

Up to now, all discussed approaches are optimized to extract coarse-grained task-level parallelism. However, task-level parallelism is of limited use for extracting efficient parallelism from loops, especially from those with loop-carried dependencies. This was also observed by Raman et. al. [9] and Tournavitis et al. [10] who presented techniques which are able to extract pipeline parallelism from loops of sequentially written applications. Both approaches split loops into different pipeline stages which can further be divided into concurrently executed sub-tasks if the stages are stateless. Unfortunately, both approaches lack cost models which are necessary to balance the extracted tasks.

Polytope models are also often used to extract parallelism from sequentially written loops. The iteration space of the loops is transformed into polytopes including all dependencies of nested loops by linear inequalities. Those approaches have the drawback that they can only analyze affine loops so that already existing applications can, in general, not be parallelized without manually adapting the source code. A representative work in this area was published by Lengauer [11].

All approaches mentioned so far have the drawback that they are not optimized for resource restricted embedded MPSoCs. First, they extract either coarse-grained task-level parallelism or finer-grained pipeline parallelism. Second, they only consider the reduction of execution time as their only optimization objective on cost of other resources like, e.g., energy consumption. Finally, all of them were developed for homogeneous architectures so that they are not able to handle different performance characteristics of the cores available in a heterogeneous MPSoC. In contrast, the approach presented in this paper combines the extraction of coarse-grained task-level parallelism with finer-grained pipeline parallelism while considering different objectives in a multi-objective aware optimization. In addition, our approach also considers different execution times for statements executed on the available processing units which is indispensable if applications should be parallelized for heterogeneous architectures.

However, our previously published approaches [12] and [13] consider multiple objectives at the same time while extracting task-level and pipeline parallelism. Comparable to the approach presented in this paper, [12] and [13] employ genetic algorithms and high-level cost models to automatically balance the extracted tasks. It could be shown that the approaches work well for homogeneous architectures. But in contrast to the work presented in this paper, heterogeneous architectures with different performance characteristics of the available processing units are not supported. Thus, large energy savings or massive speedups cannot be achieved by the previously published tools since all processing unit are treated to be identical.

To conclude, to the best to our knowledge, so far no parallelization approach exists which extracts task-level and pipeline parallelism in a multi-objective aware manner, combined in one approach which is also optimizing for heterogeneous architectures.

III. Framework & Approach

Our previous publications in [12], [13] have shown that the presented parallelization approaches, based on genetic

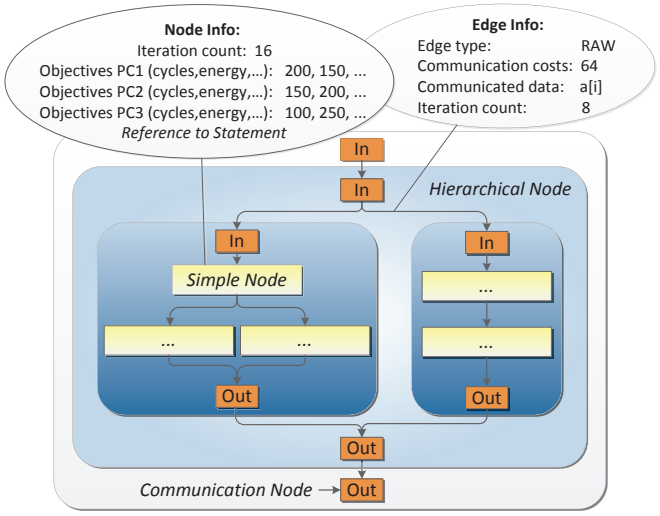


Fig. 1. Augmented Hierarchical Task Graph

algorithms, are able to extract efficient parallelism from sequentially written ANSI-C code for homogeneous embedded devices. Thus, we decided to use this framework as a starting point for our new multi-objective aware parallelization approaches optimized for heterogeneous MPSoCs. However, the existing approaches of [12], [13] are not aware of variances in the objective values for statements to be parallelized depending on the executing processing unit. This is not necessary for homogeneous architectures since all cores behave identically but it is crucial to efficiently parallelize applications for heterogeneous systems. For example, processing units with a slower frequency are, in general, much more energy efficient than a processing unit with a higher frequency at the cost of slower execution times. Therefore, our new parallelization approach takes advantage of platform specific information to automatically balance extracted tasks for processing units with differing execution behavior. Multiple objectives are considered at the same time and a front of Pareto-optimal solutions is returned to the application designer so that the solution which fits best to a specific application scenario can be chosen as final implementation. In addition, our tool also combines parallelism extraction with a pre-mapping of tasks to processor classes which represent identical processing types of the heterogeneous architecture. This enables the optimization of tasks for specific processing units directly in the parallelization process. The extracted pre-mapping information is later passed to a mapping tool to ensure that the extracted tasks are mapped to the type of processing units for which they are optimized.

The rest of this section presents a brief description of the hierarchical task graph in Section III-A which is employed as intermediate representation followed by an overview of the global parallelization algorithm in Section III-B.

A. Augmented Hierarchical Task Graph

An intermediate representation is indispensable to extract well-balanced parallelism from sequentially written applications. In this paper, we employ an *Augmented Hierarchical Task Graph* which is automatically extracted from the application's source code as intermediate representation (an example is depicted in Figure 1). The hierarchical structure

Algorithm 1 Pseudo code of global parallelization algorithm

```
1:  function MAIN(IR ir, Platform pf)
2:      htg ← EXTRACTGRAPH(ir, pf)
3:      pfront ← PARALLELIZE(htg.getRootNode(), pf)
4:      solution ← CHOOSESOLUTION(pfront)
5:      IMPLEMENTSOLUTION(solution)
6:  end function
7:  function PARALLELIZE(Node n, Platform pf)
8:      res ← {SequentialSolutions(n)}
9:      if ISNOTHIERARCHICALNODE(n) then
10:         return res
11:     end if
12:     /* Parallelize bottom-up in hierarchy, first. */
13:     for all c ∈ ChildNodes(n) do
14:         PARALLELIZE(c, pf)
15:     end for
16:     /* Parallelize this node (all children processed). */
17:     if ISLOOPSTMT(n.getStmt()) then
18:         pfront ← GAPIPELINEPARALLELIZER(n, pf)
19:     else
20:         pfront ← GATASKLEVELPARALLELIZER(n, pf)
21:     end if
22:     res ← res ∪ {pfront}
23:     return res
24: end function
```

of the graph is equivalent to the hierarchical structure of the application's source code. Thus, statements which contain other statements deeper in their hierarchy, like, e.g., a loop, are represented by *Hierarchical Nodes*. In contrast, *Simple Nodes* represent statements which do not contain any further hierarchical structures, such as, e.g., an assignment statement ($a = a + 1$).

All hierarchical nodes contain a *Communication In-* and a *Communication Out-Node* to encapsulate all communication edges coming 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 application's control flow is implicitly modeled by the hierarchical levels of the graph so that explicit control flow edges only have to be added in special cases like, e.g., *break*- or *return*-statements. Data-Flow edges are also modeled by the hierarchical task 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 are annotated with iteration counts and cost information of the considered objectives (like, e.g., execution time and energy consumption) depending on the mapped processing class (PC) (see *Node Info* in Figure 1). This information is automatically extracted by our framework by target platform simulation. All edges of the graph are also annotated with information like the amount of communicated data, the edge type, the iteration count, etc. (see *Edge Info* in Figure 1). Techniques which can be used to extract a Hierarchical Task Graph from sequentially written ANSI C-code can be found in [12].

B. Parallelization Algorithm

The overall structure of our new multi-objective aware parallelization approach for heterogeneous MPSoCs is shown

in Algorithm 1. The main function expects an intermediate representation [14] of the application's source code as well as a description of the target platform as input parameters. The target platform description [15] contains information about, e.g., the number and performance characteristics of the available processing units, interconnects, and also of the memory subsystems. After the Augmented Hierarchical Task Graph is extracted in line 2, the parallelization approach starts to extract parallelism in a bottom-up search strategy by calling the function PARALLELIZE for the root node of the hierarchical graph in line 3.

The pseudo-code of the function PARALLELIZE starts in line 7. The function returns the sequential version of the assigned statements as its only solution (line 10) if the node does not contain any further hierarchical nodes (*Simple Nodes*), since no meaningful parallelism can be extracted from a separate statement. The sequential version of the application is returned once per processor class so that it could be mapped to each of the available processing units. If the node contains further hierarchical structures, the function is recursively executed for all child nodes in line 14, first. This ensures that all child nodes have already been processed before new parallelism is extracted on the current hierarchical level.

This paper presents and combines two multi-objective aware parallelization approaches. The first one extracts coarse-grained task-level parallelism (presented in Section IV-A) which may for example execute two function calls concurrently. Task-Level Parallelism is very efficient in parallelizing large independent chunks of the application but its applicability to loops – especially those with loop-carried dependencies – is very limited. Therefore, the task-level approach is combined with a pipeline parallelization approach, presented in Section IV-B. As can be seen in Algorithm 1, the multi-objective aware pipeline parallelization approach is executed for each node representing a loop of the application in line 18, while the coarse-grained task-level parallelization approach is executed for each non-loop node in line 20. Both approaches return a front of Pareto-optimal solutions containing beneficial versions of the node to be parallelized. These results are combined with the sequential versions of the node in line 22. This set is finally returned as a result of the parallelization function.

On the next hierarchical level, the GA-based parallelization approach is able to choose one parallel solution candidate for each child node (which may contain parallelism which was found deeper in the hierarchy). In addition, these solutions are combined with new parallelism extracted on the current hierarchical level, if the solution optimizes at least one of the considered objectives. This step is repeated until the root node of the graph is reached. As a result, the front of Pareto-optimal solutions is returned to the application designer in line 4. Thus, the solution which fits best to the considered application scenario can be chosen and finally implemented in line 5. It should be mentioned here, that the hierarchical structure of the algorithm drastically reduces the complexity of the parallelization problem. In addition, the complexity scales only linear with the program size since new nodes are added for each statement but the number of statements processed per hierarchical level does not increase. This enables the use of sophisticated parallelization algorithms, like the GA-based one presented in the next Section.

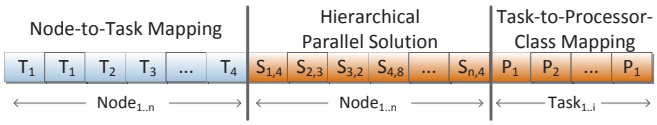


Fig. 2. Structure of Heterogeneous Task-Level Individual

IV. MULTI-OBJECTIVE AWARE EXTRACTION OF PARALLELISM

Genetic algorithms are favored for solving optimization problems in a multi-objective aware manner. An optimization developer has to provide a comprehensive representation of the individuals' structure as well as three methods implementing mutation, recombination and evaluation of possible solution candidates. A genetic algorithm starts to create an initial population containing individuals representing possible solution candidates. Afterwards, the individuals are evaluated for the considered objectives. Based on these results, some individuals are chosen to survive or are used for mutation and recombination (which is also called cross-over) to create new solution candidates for the next population. This step is repeated and several populations are created until a given stopping criterion, like, e.g., a maximum number of created populations, is fulfilled. Finally, a front of Pareto-optimal solutions is returned as the result of the optimization step. This makes genetic algorithms well applicable to extract parallelism in a multi-objective aware manner. Our approach uses the PISA framework [16] for Selection and Variation purposes.

In the following, Section IV-A presents the individuals' structure used to extract task-level parallelism while the structure used to extract pipeline parallelism is described in Section IV-B. Afterwards, Section IV-C defines the functions used to evaluate the considered objectives before some hints about portability are given in Section IV-D. Finally, the proposed mutation and cross-over functions are described in Section IV-E.

A. Extraction of Task-Level Parallelism

The first challenge in using genetic algorithms is to map the values of the solution space to genes of the individuals' chromosomes in such a way that they can be changed and evaluated efficiently. Figure 2 shows the structure employed by our novel approach to extract task-level parallelism from a hierarchical node of the application and map the extracted tasks to processing units of a heterogeneous MPSoC. As already explained in Section III-B, each hierarchical node is processed in isolation which drastically reduces the complexity of the parallelization problem. Due to the bottom-up approach, all nodes deeper in the hierarchy are already processed, so that a front of Pareto-optimal solutions $S_{i,j}$ exists for each child node N_i containing solution candidates which might implement parallelism deeper in the hierarchy.

The first part of the employed chromosome structure maps each child node of the node to be parallelized to newly extracted tasks (cf. Figure 2). The second part of the individuals' chromosome structure chooses one hierarchical solution for each child node while the third part maps newly extracted tasks to processor classes of the targeted MPSoC. In the example of Figure 2, nodes N_1 and N_2 are mapped to task T_1 while node N_3 is mapped to task T_2 and so on. In addition, solution $S_{1,4}$ of

child node N_1's Pareto front is chosen as hierarchical solution while $S_{2,3}$ is chosen for child node N_2. Task T_1 is mapped to processor class P_1 while task T_2 is mapped to processor class P_2.

Each chromosome is represented by an array of integers. The size of each chromosome is twice as large as the number of direct child statements n contained in the hierarchical node to be parallelized ($1\times$ node to task mapping + $1\times$ hierarchical solution) plus the number of maximal extractable tasks i[1]. Thus, each chromosome can be encoded very efficiently by $2 \times n + i$ integers.

The impact on the evaluation of the individuals' objectives is depicted in Figure 3. The example parallelizes a hierarchical node with seven child nodes which can be mapped to four newly created tasks on a platform providing four processing units grouped into three processor classes. The figure shows the genes' values on the left-hand-side and their impact on the evaluation on the right-hand-side. The upper part of the figure shows the task graph representation of the node to be parallelized according to the node-to-task mapping defined on the left-hand-side. As can be seen, nodes N_1 and N_2 belong to task T_1 while node N_3 belongs to task T_2. Edges between the created tasks depend on the node-to-task mapping. Here, a dependence edge between node N_2 and N_3 exists which also adds a dependence edge between tasks T_1 and T_2. Thus, the execution of task T_2 has to wait for completion of T_1 since data has to be communicated between both tasks before T_2 can start with its execution. These task execution orders as well as inserted communication costs should be considered in the evaluation functions for the applied objectives.

The second part of the chromosome representation contains the selection of hierarchical parallel solution candidates for all child nodes. Due to the bottom-up approach, all child nodes are already processed by the GA-based parallelization technique. Thus, a front of Pareto-optimal solutions exists for each child node evaluated by the high-level functions presented in Section IV-C. The frontiers contain solution candidates with parallelism which was found deeper in the hierarchy. The approach has to choose one solution candidate from each child nodes' Pareto-frontier providing different objective values for the corresponding node. A solution with more extracted parallelism may, for example, reduce the overall execution time at the cost of the system's energy consumption. Thus, this part of the chromosome's structure also influences the objectives' evaluation.

The last part of the chromosomes' structure defines the task-to-processor class mapping which is crucial if applications should be parallelized for heterogeneous MPSoCs. Execution time, energy consumption, and other objectives depend on the processing unit used for a task. Therefore, these gene values map the tasks created in the first part of the chromosomes to processor classes representing identical processing units. In the example shown, task T_1 which contains nodes N_1 and N_2 is mapped to processor class C_1, while task T_3 which executes the statements of nodes N_4 and N_5 is mapped to processor class C_2. Also here, the genes' values directly influence the evaluation of all objective values.

[1]The maximum number of extractable tasks is set to the number of available processing units by default but can be changed by user.

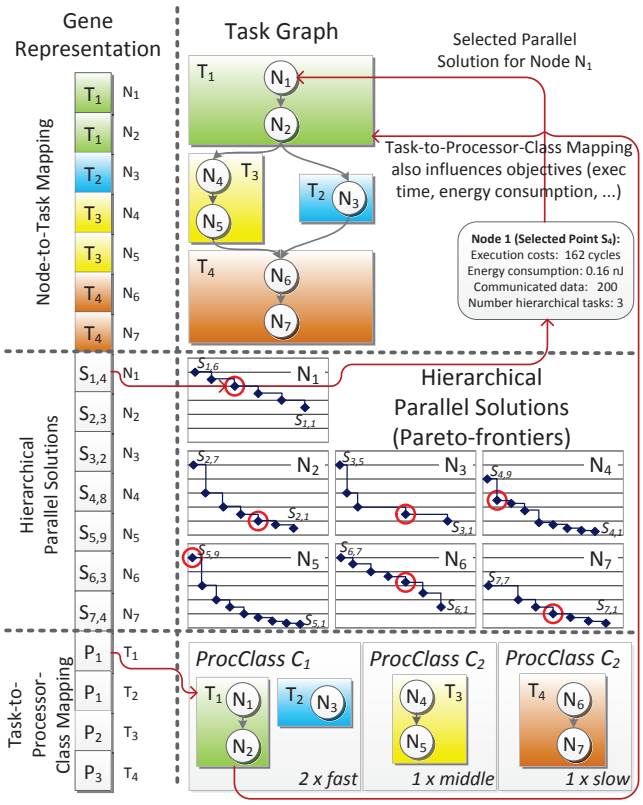


Fig. 3. Structure of Heterogeneous Pipeline Individual

To summarize the gene representation, new tasks can be extracted, mapped to processor classes, and can also be combined with tasks which were found deeper in the hierarchy. More details on the evaluation of the considered objectives are presented in Section IV-C.

B. Extraction of Pipeline Parallelism

Even though task-level parallelism is very efficient for large independent blocks of the application, its applicability to loops of embedded applications, especially those with loop-carried dependencies, is limited. Therefore, this section describes a second parallelization approach which is able to extract multi-objective aware pipeline parallelism for heterogeneous architectures. As already described, both approaches can also be applied in a combined manner.

The main target of pipeline parallelism is to split a loop's body horizontally into different pipeline stages so that the statements in the loop body belong to disjunct tasks. The benefit of such a parallelization is that a task T_1 can start with its next iteration as soon as it has communicated its output data to its predecessor task T_2 waiting for the data (and the data for the next iteration of task T_1 itself is also available). Hence, a pipelined execution behavior can be created even for loops with loop-carried dependencies. However, the amount of parallelism which can be extracted by pure pipeline creation is limited. Therefore, we also allow to distribute the iterations of a pipeline stage to concurrently executed sub-tasks. Thus, e.g., iterations 1, 2, and 3 of task T_1 can be executed in parallel if no data dependencies between the different iterations exist.

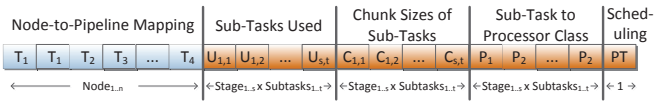


Fig. 4. Structure of Heterogeneous Pipeline Individual

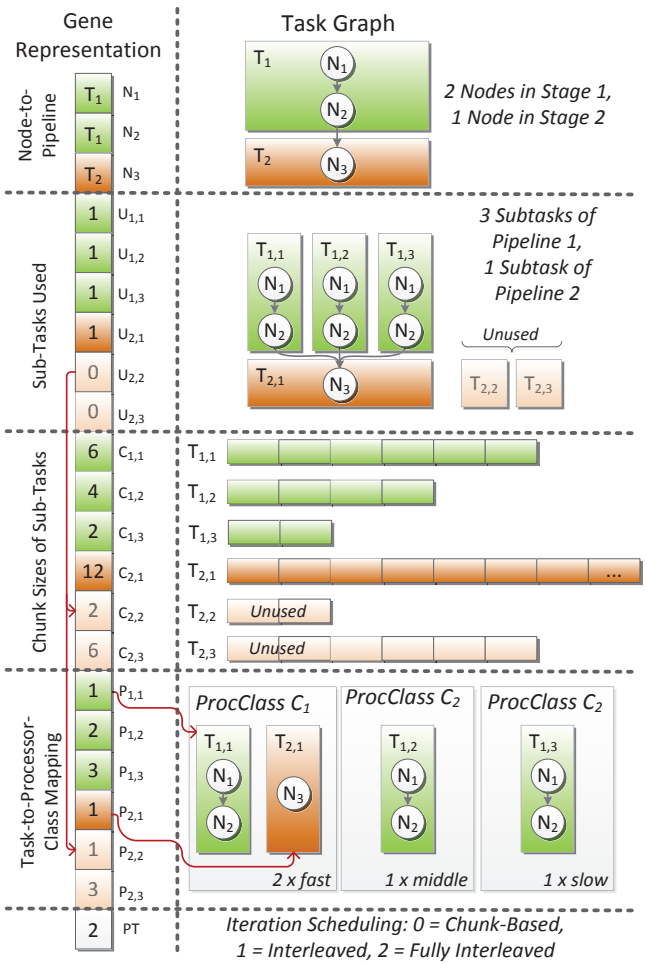
These targets as well as iteration balancing and processor class mapping have to be extracted by a GA-based parallelization approach targeting heterogeneous MPSoCs.

The employed chromosome structure implementing such a parallelization approach based on a genetic algorithm is depicted in Figure 4. The first part of the chromosome maps child nodes of the loop to be parallelized to disjunct pipeline stages. An example using this chromosome representation is shown in Figure 5(a). As can be seen, nodes N_1 and N_2 are mapped to task T_1 while node N_3 is mapped to task T_2. Thus, two stages are extracted which can be executed in a pipelined manner. Unfortunately, a dependence edge exists between both tasks which drastically limits the achievable speedup of this solution. Therefore, the second part of the chromosome structure (called Sub-Tasks Used) defines whether sub-tasks are used for a given task [2]. In our example, three sub-tasks are used for T_1 processing this task in parallel to faster supply task T_2 with the required input data. Task T_2 uses only one sub-task and is not split into concurrently executed sub-tasks.

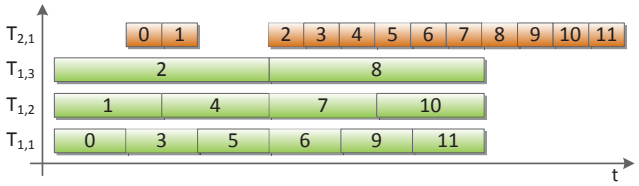
So far, pipeline stage extraction and sub-task generation are encoded in the chromosome structure. This structure is – in general – sufficient if applications are to be parallelized for homogeneous architectures. For heterogeneous ones, however, the execution time of a task's iteration may change depending on the executing processing unit. Therefore, the third gene block allocates chunk sizes of the pipeline stages' iterations to the according sub-tasks. Thus, sub-tasks which execute larger chunks can, e.g., be executed on faster processing units to balance the overall execution behavior. In the example of Figure 5(a), sub-task $T_{1,1}$ executes 6 iterations while sub-tasks $T_{1,2}$ and $T_{1,3}$ execute 4 and 2 iterations, respectively. Sub-task $T_{2,1}$ of pipeline stage T_2 executes all 12 iterations of the loop. The chunk sizes of $T_{2,2}$ and $T_{2,3}$ are ignored since the sub-tasks are not used according to the zero values in the last two positions of the *Sub-Tasks Used* genes. Note that the chunk sizes are not directly taken from the genes' values because the probability that the sum of all chunk sizes per pipeline stage is equal to the number of loop iterations is quite low. Since all loop iterations of a pipeline stage must be executed exactly once by the generated sub-tasks, a solution with less or more executed iterations is invalid. To avoid this, the chunk sizes are translated into percentage-values so that the iterations are mapped to the sub-tasks according to these percentages.

The fourth block of gene values maps the extracted sub-tasks to the processor classes of the targeted heterogeneous MPSoC. In our example, sub-task $T_{1,1}$ is mapped to processor class C_1 which contains two fast processors. This is a good choice for this scenario, since $T_{1,1}$ executes more iterations then both other sub-tasks of pipeline stage T_1. $T_{1,2}$ and $T_{1,3}$ are mapped to processor class C_2 and C_3 with respect to their execution load.

[2]The maximum number of generated subtasks can also be defined by the user and is set to the number of available processing units by default.



(a) Structure of Heterogeneous Pipeline Individual



(b) Timing Diagram of Solution Candidate

Fig. 5. Heterogeneous Pipeline Parallelization

The last position in the chromosome's structure is reserved for iteration scheduling. In our current approach we support three different scheduling strategies, namely *chunk-based*, *interleaved* and *fully-interleaved*. The first one maps all iterations continuously to a sub-task respecting its chunk size. Thus, sub-task $T_{1,1}$ would be allocated with iterations $\{0, .., 5\}$ while $T_{1,2}$ would be allocated with iterations $\{6, .., 9\}$, respectively. This scheduling policy has very good cache locality but all succeeding iterations are executed by the same task, so that tasks waiting for the generated data do not obtain the data of iterations concurrently. The *interleaved* policy executes smaller chunk sizes in an interleaved manner. Thus, task $T_{1,1}$ could for example execute iterations $\{0{-}2, 6{-}8\}$ while $T_{1,2}$ and $T_{1,3}$ could execute iterations $\{3{-}4, 9{-}10\}$ and $\{5, 11\}$, respectively. This scheduling policy is a good trade-off between

cache locality and a faster response time for waiting tasks. The last scheduling policy, namely *fully-interleaved*, schedules the iterations in a way that all tasks start with the execution of a pipeline stage's iteration as soon as possible while respecting the configured chunk sizes. A possible schedule for this example with this scheduling methodology could be $\{0, 3, 5, 6, 9, 11\}$, $\{1, 4, 7, 10\}$, and $\{2, 8\}$ for tasks $T_{1,1}$, $T_{1,2}$, and $T_{1,3}$, respectively (cf. Figure 5(b)). All three tasks execute one of the first three iterations concurrently. Afterwards, task $T_{1,3}$ starts with the next iteration after task $T_{1,1}$ has executed 3 iterations, like defined by the chunk size. In the future, we would like to integrate additional scheduling strategies as well.

Taken all genes together, it is possible to extract well-balanced pipeline parallelism optimized for heterogeneous MPSoCs. The timing behavior of the genes' values presented in Figure 5(a) is depicted in Figure 5(b). As can be seen, the node-task-mapping, the sub-task creation, the different chunk-sizes, the processor-class-mapping as well as the iteration scheduling are considered in the diagram. As soon as the first six iterations of pipeline stage T_1 have been executed, all sub-tasks are executing without any interruptions. This behavior would improve further if the loop would be executed for more than the shown 12 iterations. Note that the solution depicted in Figure 5(a) has a good execution behavior regarding execution time. However, our approach considers other objectives, such as energy consumption, as well. For this objective, other solutions with, e.g., less used cores may produce better results. Therefore, our GA-based approach evaluates all considered objectives for all individuals so that a front of Pareto-optimal solutions is generated which also contains results with good trade-offs between the different objectives.

C. Evaluation of Objectives

To be able to choose individuals for mutation and recombination, a genetic algorithm must be able to evaluate the different objective values for the individuals of the current population. Therefore, this section defines the evaluation functions for the three objectives which are currently considered by our framework. They are based on the ones defined in [4] and [12]. The authors have shown that their models are accurate enough to be used for homogeneous architectures and have verified them with target platform simulation. However, the presented models do not distinguish between different performance characteristics of processing units used in heterogeneous architectures. We present the models used to evaluate the task-level individuals here. The difference to the models used to evaluate pipeline individuals is that the latter one has to split up the nodes' evaluation to different iterations. Due to limited space, the description of the evaluation models used for pipeline individuals has to be skipped here.

1) Execution time: The proposed models used to determine the individual's execution time return a linear execution time estimation. Based on the models presented in [12], the execution time is equivalent to the execution time of the longest path within the hierarchical node to be parallelized. With respect to the node-to-task-mapping shown in the example of Figure 3, the longest execution path is either $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6 \rightarrow N_7$ or $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_6 \rightarrow N_7$ depending on the nodes' execution times – which also depend on the task-to-processor-class mapping – and the communication delay of the tasks created. The following equations define the calculation of the execution time in a more formal way.

The execution time $ET(T_i)$ for task T_i is equal to the sum of the execution times $ETN(n, S_{n,j}, c)$ and a constant task creation overhead TCO. $ETN(n, S_{n,j}, c)$ is calculated for each child node n which is mapped to task T_i. The execution time of the child nodes ETN also depends on the chosen hierarchical parallel solution $S_{n,j}$ and the processor class c executing the task's statements:

$$ET(T_i) = TCO + \sum_{n \in Nodes(T_i)} ETN(n, S_{n,j}, c)$$

The path costs $PC(T_i)$ of task T_i are recursively defined and equal to the sum of the execution time $ET(T_i)$ of the task itself plus the path costs $PC(t)$ of the most expensive predecessor task t including the communication costs $CC(t, T_i)$:

$$PC(T_i) = ET(T_i) + \max\{PC(t) + CC(t, T_i) | \forall t \in Pred(T_i)\}$$

Finally, the overall execution time of the node to be parallelized is equal to the longest execution path:

$$OverallET = \max\{PC(t) | \forall t \in Tasks\}$$

2) Energy consumption: The energy consumption of a task-level individual contains energy costs which arise due to task spawning, statement execution on specific processing units and communication, like shown in the following equations.

The energy consumption $ICE(T_i)$ caused by incoming communication of task T_i is calculated by summing up a static overhead for the incoming data $ICEO$ (for setting up the communication channels etc.) and a factor ICM per communicated byte:

$$ICE(T_i) = \sum_{d \in InData(T_i)} ICEO + \#Bytes(d) * ICM$$

The energy consumption $OCE(T_i)$ for the outgoing communication is similar to $ICE(T_i)$:

$$OCE(T_i) = \sum_{d \in OutData(T_i)} OCEO + \#Bytes(d) * OCM$$

The total amount of energy $E(T_i)$ consumed by each task T_i is the sum of a constant task creation overhead TCE, the energy which has to be spent to execute the statements of the task $EE(T_i, c)$ on the mapped processing unit c and the energy for the incoming and outgoing communication:

$$E(T_i) = TCE + EE(T_i, c) + ICE(T_i) + OCE(T_i)$$

Thus, the overall energy consumption for a chromosome's configuration is equal to the sum of the energy consumption of all tasks:

$$OverallEnergy = \sum_{t \in Tasks} E(t)$$

3) Communication overhead: The third considered objective is the overhead introduced by communication. This objective value is equal to the sum of the communicated bytes of all tasks multiplied by a specified communication delay:

$$CommOverhead = \sum_{data \in Comm} \#Bytes(data) * Costs$$

D. Portability to Multiple Target Platforms

To enable portability to multiple target platforms, the hierarchical task graph is annotated with various cost information, like, e.g., execution time and energy consumption per processor class. Thus, as long as these values can be extracted, our approach is portable to various target platforms.

In addition, all constants of the evaluation functions presented in the previous section can be configured in our framework, as well. The communication costs $CC(t, T_i)$ are determined by multiplying a configurable communication factor with the amount of communicated bytes. The task creation overheads for execution time TCO and energy consumption TCE can also be configured, to mention only some of the configurable constants. By combining these configurable parts with the extracted objective values it should be easy to adapt our framework to various target platforms.

E. Mutation and Cross-Over Functions

Genetic algorithms create new solution candidates by mutating a profitable existing one or by recombining two of them. These functions have to be provided by the optimization developer. To generate new parallel solution candidates we implemented the mutation function with a 1-bit flip mutation strategy. This means that a random position in the chromosome of the individual which should be mutated is changed so that for example one child node is moved from one task or pipeline stage to another one. It can also happen that, e.g., a created task is re-allocated to a different processor class.

The employed cross-over function (also called recombination) splits two individuals at a random position and joins the left-hand-side of the first one with the right-hand-side of the second one and vice versa. This can easily be achieved since all chromosomes of the same node to be parallelized have the same length and the split position is the same for both individuals.

So far, both functions seem to be simple state-of-the-art implementations. However, we have observed that a huge amount of invalid solutions ($> 98\%$) is created by such simple mutation and cross-over functions which drastically reduces the solution quality of the employed genetic algorithm. Solutions are invalid if, e.g., a deadlock is created by mutation or recombination so that one task is waiting for data of another one and vice versa. This can happen if a child node is moved from one task to another one since dependencies between the tasks may change as well. Our algorithm also rejects solutions with more concurrently executed tasks mapped to the same processor class as processing units are available in this class. Thus, additional scheduling overhead at runtime can be avoided.

To circumvent the creation of too many invalid solutions, our mutation and cross-over functions are enriched with smart correction algorithms. As soon as a valid solution gets invalid after mutation or recombination, the algorithm determines the source of the problem and tries to fix it with a subsequently executed mutation. Thus, to fix, e.g., a deadlock, the target node which causes the deadlock is moved to a different task to solve the problem. If, for example, too many tasks are allocated to a specific processor class after mutation, one of the other tasks is moved to a different processor class. These steps are repeated until the solution gets valid again or a maximum number of fixing steps is reached.

By applying this strategy, the number of invalid solutions could be reduced from more than 98% to around 5%, which significantly improved the solution quality.

V. Experimental Results

To evaluate the applicability of our new multi-objective aware parallelization approach for embedded heterogeneous MPSoCs we have chosen several benchmarks included in the UTDSP benchmark suite [17] covering test cases from typical real-world embedded application domains. In addition, we also evaluated other representative applications which are often used on MPSoCs like a JPEG encoder. As target platform for our model-based evaluation we have chosen a same ISA-multi-core platform (like [18]) configured with four ARM cores running at 100 MHz (1x), 250 MHz (1x) and 500 MHz (2x) to simulate a platform with large performance variances.

Detailed results for four of the considered applications can be found in Figure 6. In the current version, our framework supports three objectives, namely speedup of the execution time, energy consumption and inserted communication overhead. These objectives are arranged on the x-, y-, and z-axes, accordingly. Other objectives, like, e.g., the reduction of thermal issues or the size of allocated memory may be added in the future as well. The sequential version of the application, executed on the slowest processing unit, is located at the bottom-left of each diagram. This solution is the slowest one with a speedup of $1\times$ and consumes the lowest amount of energy since only one core – the slowest and most energy efficient one – is used so that it forms the base-line, here. For improved readability, vertical bars are added to the diagrams to project the points into the x-y-plane. In addition, a solid line marks the front of Pareto-optimal solutions, also projected to the x-y-plane. *Note*: The projected Pareto-frontier is not in a straight echelon form due to the third objective. Even if a solution is worse in execution time and energy consumption it may be added to the front of Pareto-optimal solutions if it adds less communication overhead. Each diagram contains both, Pareto-optimal and Pareto-dominated solutions. Of course, only the first ones are finally returned to the application designer as possible solution candidates. The diagram contains three different shapes to mark solution candidates containing parallel sections generated by the task-level parallelization approach (cf. Section IV-A), the pipeline parallelization approach (cf. Section IV-B) and a mixture of both approaches. All objective values of the presented solutions are based on the high-level models presented in Section IV-C. The input for these models (e.g., execution time and energy consumption) was extracted by executing all statements of the target application in an isolated way on a high-level simulator.

The visualized benchmarks have been selected since they show different behaviors with respect to the evaluated parallelization approaches and the maximum objective values. The *edge detect* application, for example, shown in Figure 6(a) profits most from the presented pipeline parallelization approach. Only a few solutions generated by the task-level parallelization approach are part of the Pareto-frontier. In contrast,

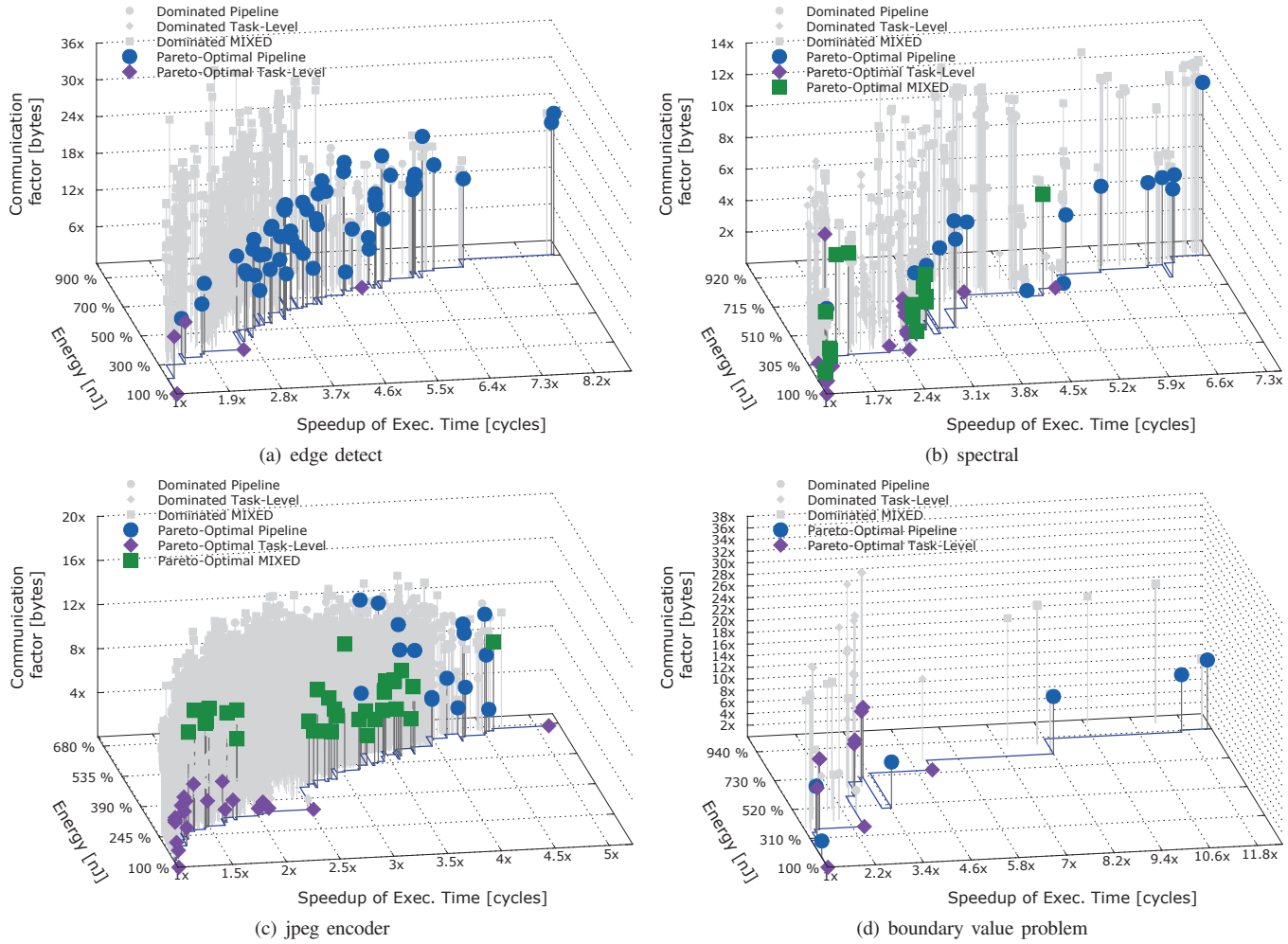


Fig. 6. Front of Pareto-Optimal Solutions Returned by the Parallelization Framework

the solutions generated for the *spectral* benchmark shown in Figure 6(b) and the *JPEG encoder* shown in Figure 6(c) profit from both approaches and also from a mixture of them. The last visualized application is the *boundary value problem* in Figure 6(d). Here, fewer solutions are generated but it was possible to extract a speedup of over 12× which is very close to the theoretical speedup limit of the targeted MPSoC. However, the solution increases the energy consumption to 940% compared to the slowest solution which is executed sequentially on the slowest processing unit, only. This highlights the trade-offs which are enabled by our parallelization framework due to our multi-objective aware approach. If the application designer knows that a speedup of, e.g., 3.4× is sufficient for the considered application scenario, a solution which uses less and also slower processing units can be chosen so that the energy consumption is only increased to around 500% instead of 940%. If 2.2× are sufficient, a solution with an energy consumption of 320% could also be chosen to further decrease the system's overall energy consumption. Similar trade-offs were also observed for the other evaluated benchmarks. The fastest solutions, extracted for the edge-detect benchmark shown in Figure 6(a), increase the system's energy consumption to 900% while gaining a speedup of 9×. If a speedup of 5× is sufficient, more energy efficient cores can be used for execution to reduce the energy consumption to 700%.

Most solutions generated by the presented pipeline parallelization approach extract the highest speedups at the cost of the system's energy consumption. In addition, a lot of data has to be communicated for this approach. The solutions generated by task-level parallelism are the ones producing only a small speedup but are much more energy efficient. The solutions which contain both, parallel sections based on task-level and pipeline parallelism, are mostly a good trade-off between higher speedups and lower energy consumption. This shows that both approaches are able to extract efficient parallelism from sequentially written embedded applications.

A. Statistics of the GA-based approaches

Due to limited space it is not possible to present figures with Pareto-frontiers for all considered applications. Therefore, Table I summarizes the results for all evaluated benchmarks while providing additional statistics about timing and the employed genetic algorithm. The columns contain information about the required time to extract the final solution space in minutes and seconds (*Time*), the number of parallelized hierarchical nodes (*#Nodes*), the number of created populations (*#Populations*), the number of created individuals (*#Individuals*), the number of performed mutations (*#Mutations*) and cross-over operations (*#Cross-Over*) as well as the number of Pareto-optimal solutions (*#Solutions*) returned to the applica-

TABLE I. EXECUTION TIME AND STATISTICS OF THE COMBINED, GENETIC ALGORITHM-BASED PARALLELIZATION APPROACHES

Benchmark	Time[3]	#Nodes	#Populations	#Individuals	#Mutations	#Cross-Over	#Solutions
adpcm encoder	01:31	36	1,520	153,453	30,729	104,962	63
boundary value	01:17	12	644	83,973	17,900	54,964	34
edge detect	02:43	105	2,872	200,371	40,002	133,096	118
filterbank	03:24	7	412	50,779	12,229	44,164	47
fir 256	00:30	13	388	29,889	6,629	21,515	44
iir 4	03:02	13	852	105,224	22,631	79,206	63
JPEG encoder	05:13	62	2,868	312,242	68,142	246,427	333
latnrm 32	01:11	17	636	53,642	11,462	39,831	54
mult 10 10	01:01	36	1,060	70,855	14,635	57,226	90
spectral	02:25	51	2,260	213,230	44,696	158,624	114

tion designer. All numbers are summed up over all performed parallelization steps. The population sizes as well as the number of populations are determined dynamically so that nodes with a larger number of child nodes are processed longer than nodes with a smaller number of child nodes.

As can be seen, the number of finally returned Pareto-optimal solutions ranges between 34 solutions for the boundary value problem up to 333 solutions for the JPEG encoder. Note that some of the solutions are so close to each other that they overlap with other solutions in the diagrams shown in Figure 6. Nevertheless, the number of extracted solutions shows the huge optimization potential for trade-offs served by our multi-objective aware approach. Another important aspect is the time the approach needs to parallelize an application. The time varies between 30 seconds for the fir benchmark up to 5 minutes for the JPEG encoder measured on a system with four AMD-Opteron cores running at 2.4 GHz. For the latter one, over 300,000 solution candidates were created, evaluated and mutated or recombined. This means that creation and evaluation of one individual could be done in less than a millisecond due to the use of the proposed high-level models.

To summarize, the following results were achieved:

1) The consideration of differing performance characteristics of processing units available in a heterogeneous MPSoC combined with mapping decisions in the parallelization process is highly beneficial.
2) The presented framework is able to provide the application designer a large number of beneficial solution candidates for trade-offs between different objectives.
3) The combination of task-level and pipeline parallelization approaches optimized for heterogeneous MPSoCs in one framework is advantageous since both approaches perfectly complement each other.

VI. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this paper presents the first multi-objective aware parallelization approach combining the extraction of task-level and pipeline parallelism in one framework, optimized for heterogeneous MPSoCs. The huge optimization potential exploitable by our tool was demonstrated on several real-world applications employed in typical embedded application domains. By using our proposed framework, the application designer is able to apply a parallelized version of the application optimized for a specific application scenario. In contrast to other frameworks, a huge amount of energy can be saved if the designer does not select the solution with the highest speedup, returned as the only solution by most existing parallelization frameworks.

In the future we would like to extend our approaches to other objectives, like, e.g., code size. In addition, we would also like to choose between different communication strategies in the parallel code sections. DVFS techniques can also be integrated into the models to save even more energy.

[3]Measured on a system with four AMD Opteron cores running at 2.4GHz

REFERENCES

[1] 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.

[2] 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.

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

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

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

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

[7] S. Campanoni, T. Jones, G. Holloway *et al.*, "Helix: automatic parallelization of irregular programs for chip multiprocessing," in *Proc. of CGO*, 2012.

[8] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of AFIPS (Spring)*, 1967.

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

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

[11] C. Lengauer, "Loop Parallelization in the Polytope Model," in *CONCUR '93, Lecture Notes in Computer Science 715*, 1993.

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

[13] D. Cordes, M. Engel, P. Marwedel *et al.*, "Automatic extraction of multi-objective aware pipeline parallelism using genetic algorithms," in *Proc. of CODES+ISSS*, 2012.

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

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

[16] S. Bleuler, M. Laumanns, L. Thiele *et al.*, "PISA — A Platform and Programming Language Independent Interface for Search Algorithms," in *Proc. of EMO*. Springer, 2003.

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

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