Multi-Objective Aware Extraction of Task-Level Parallelism Using Genetic Algorithms

Daniel Cordes
TU Dortmund University
Dortmund, Germany
daniel.cordes@tu-dortmund.de

Peter Marwedel
TU Dortmund University
Dortmund, Germany
peter.marwedel@tu-dortmund.de

***Abstract*—A large amount of research work has been done in the area of automatic parallelization for decades, resulting in a huge amount of tools, which should relieve the designer from the burden of manually parallelizing an application. Unfortunately, most of these tools are only optimizing the execution time by splitting up applications into concurrently executed tasks. In the domain of embedded devices, however, it is not sufficient to look only at this criterion. Since most of these devices are constraint-driven regarding execution time, energy consumption, heat dissipation and other objectives, a good trade-off has to be found to efficiently map applications to multiprocessor system on chip (MPSoC) devices. Therefore, we developed a fully automated multi-objective aware parallelization framework, which optimizes different objectives at the same time. The tool returns a Pareto-optimal front of solutions of the parallelized application to the designer, so that the solution with the best trade-off can be chosen.**

***Index Terms*—Automatic Parallelization, Embedded Software, Multi-Objective, Genetic Algorithms, Task-Level Parallelism, Energy awareness**

I. Introduction

The demands that are imposed on modern embedded devices are continuously increasing, making multiprocessor system on chip (MPSoC) devices more and more important in the domain of embedded systems. By providing multiple cores in one system, it is possible to fulfill the performance requirements of today's embedded applications without drastically increasing the cores' frequencies. Unfortunately, these benefits do not come for free. To efficiently map these mostly sequentially written applications to MPSoCs, the source code has to be split up into concurrently executed tasks. Therefore, many (semi-) automatic tools exist, but most of them are not applicable for constraint-driven embedded devices.

In the domain of desktop or even high-performance computing, much computational power and a huge amount of energy are available. Thus, it is not surprising that most parallelization tools are only optimizing the execution time of applications at the expense of other plentiful available resources. In contrast, many objectives have to be considered at the same time if an application should be efficiently mapped onto an embedded MPSoC. Certainly, a lot of execution time can be saved if multiple cores of an MPSoC are executing an application fully in parallel. But this requires a huge amount of energy since all cores must be supplied with power. If the designer knows that a certain amount of speedup is sufficient for the execution of an application, it may be a good idea to reduce the amount of implemented parallelism in order to put some of the cores into idle mode or switch to a platform with less cores which saves a lot of energy.

While multi-objective aware optimizations are widely used in the domain of mapping or scheduling algorithms (e.g., [1], [2] and [3]), they have not been employed to parallelization tools so far. In general, most frameworks are first extracting massive parallel parts of the applications to reduce execution time before multi-objective or energy aware mapping tools try to minimize other objectives as well. At this point, a lot of optimization potential is wasted since mapping tools rely on the parallelized application, which is given as input.

Therefore, we developed a novel, fully automated multi-objective aware task-level parallelization framework which optimizes different objectives at the same time by using genetic algorithms. In the current implementation, execution time, energy consumption and communication costs are evaluated by the parallelization approach. While the first two objectives are in general contrary to each other, the communication costs may be correlated with both objectives. The parallelization algorithm determines a Pareto-front of optimized solutions from which the designer can choose the most suitable solution for his scenario.

The main contributions of this paper are as follows:

1) To the best of our knowledge, this is the first parallelization approach which uses genetic algorithms to partition applications in a multi-objective aware manner.
2) In contrast to the high-performance computing community, this framework focuses on the requirements for constraint-driven embedded devices.
3) We use high-level models to evaluate different objectives without re-simulating the whole application.

The rest of the paper is organized as follows: Section II gives a survey of related work, followed by a short explanation of the hierarchical task graph in Section III, employed as intermediate representation. Section IV gives an overview of the overall parallelization algorithm, before the multi-objective aware parallelization approach is described in Section V. Afterwards, Section VI gives an overview of our framework which is evaluated in Section VII. Finally, Section VIII summarizes this paper and gives directions for future work.

II. Related work

The first Multi-Core architectures were invented decades ago. As a result, special programming languages or models like, e.g., *MPI*, *PThreads* and *OpenMP* as well as semi- or fully automatic parallelization techniques were developed. Most of these techniques can be grouped into at least the categories of *task-level*, *data-level* and *pipeline parallelism*.

The first category of tools extracts very coarse grained parallelism by, e.g., moving independent function calls into concurrently executed tasks. The frameworks of Hall et al. [4], Ceng et al. [5] and our previous publication [6] are dealing with the extraction of this kind of parallelism in a semi- and fully-automatic manner. Other approaches like the ones published by Verdoolaege et al. [7], Sarkar [8] and Ottoni [9] are also extracting task-level parallelism.

In addition to task level parallelism, the finer-grained data level and pipeline parallelism techniques are focusing on loops in applications. While, in general, data-level parallelism splits the amount of data being processed into equal tasks [10], [11], pipeline parallelism splits the body of a loop into disjunctive pipelined stages. Raman et al. [12], Tournavitis et al. [13] and our previous publication in [14] are examples of frameworks which are able to extract this kind of parallelism.

The framework presented in this publication belongs to the first category of tools which extract task-level parallelism. Compared to our work, the abovementioned ones are only optimizing the performance and disregard other objectives, like e.g., energy consumption or bus load. Qiu et al. [15] present an energy-aware loop parallelization method, which is most relevant to our work. They are optimizing performance and energy consumption while parallelizing an application. In contrast to our publication, [15] implements a two phase strategy, which first parallelizes loops to optimize the performance of an application. Afterwards, they try to reduce the energy consumption of the system based on the previously parallelized application. Instead, our work is observing different objectives at the same time, so that optimizations are not based on decoupled phases, which may lead to sub-optimal results. Wang et al. [16] are also implementing a two phase strategy to parallelize streaming applications. Cho et al. [17] present theoretical formulas describing the interplay of program performance and energy consumption of a parallelized application.

Thus, best to our knowledge, our work is the first one which uses genetic algorithms to take multi-objective aware decisions while extracting task-level parallelism from sequential applications for MPSoCs.

III. Hierarchical Task Graph

The complexity of the solution space is always a problem for tools which are dealing with automatic parallelization. Since our multi-objective aware approach is optimizing for at least three objectives at the same time, good approximations in the intermediate representation are essential. Hence, we are using an augmented hierarchical task graph as intermediate representation (cf. Figure 1). The graph contains several hierarchical levels, which correspond to the hierarchical structure of an application. Thus, complex statements like, e.g., loops, selections or function bodies are represented by hierarchical nodes. Expression statements like, e.g., $a = b$ are represented by simple nodes, without additional hierarchical levels.

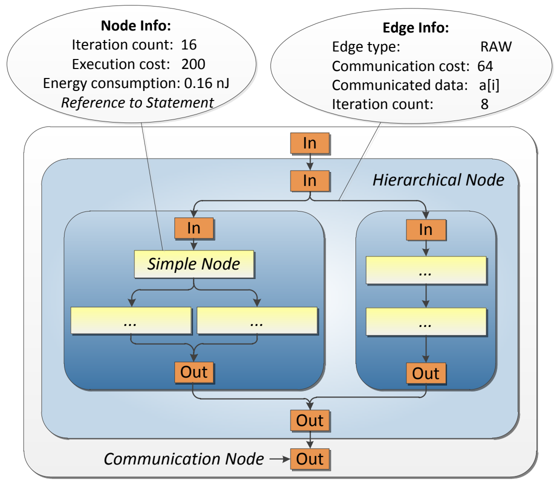


Fig. 1. Hierarchical Task Graph Example

Dependencies which may lead to inter-task communication are depicted by directed edges between the nodes of the graph. Each hierarchical level contains a communication in- and out-node which encapsulates the communication of each hierarchical level. Thus, each hierarchical node can be parallelized in isolation, so that the approach does not have to consider the whole application at once. Both, nodes and edges of the graph are augmented with information like execution counts, execution times, energy consumption, etc., which are extracted by our framework.

The hierarchical task graph is extracted fully automatically from sequential ANSI-C code, to enable an automatic parallelization tool flow. For more information on the structure and how to extract the graph, we would like to refer to [6].

IV. Tool Flow

As already mentioned, the multi-objective aware parallelization tool is able to parallelize sequential applications which are written in ANSI-C code. In a first step, our framework analyzes the application to extract information which is necessary to create the corresponding hierarchical task graph. As soon as the graph is extracted, the parallelization process starts to extract parallelism in a bottom-up search strategy in the hierarchical structure of the graph. Each hierarchical node is processed in isolation. The genetic algorithm is moving child nodes of the hierarchical nodes to tasks which are then evaluated for all considered objectives. As a result, a front of Pareto-optimal solutions for the different objectives is attached to the hierarchical nodes. As soon as all nodes on the same level of the hierarchy are processed, the parallelization algorithm continues with the parent node. There, the algorithm is also able to move child nodes to new tasks. In addition, it also chooses one of the solutions of the Pareto-front of each child node which may contain additional tasks deeper

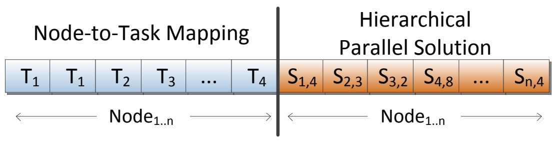


Fig. 2. Individual's Gene Structure

in the hierarchy. This procedure is continued until each node in the hierarchical task graph is processed and the top node is reached. The Pareto-front of parallel solutions is then returned to the designer so that the solution with the best trade-off for his scenario can be chosen.

V. Multi-Objective Aware Parallelization

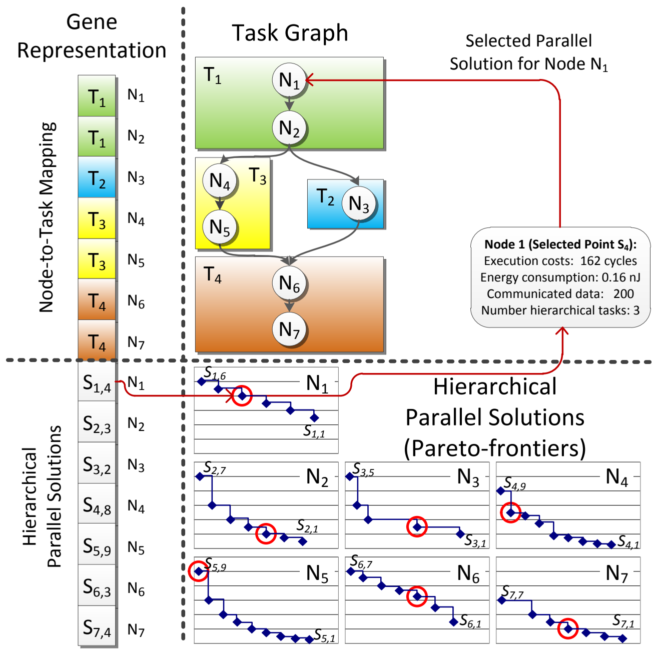
Genetic algorithms are well known for their applicability to multi-objective optimizations. They combine natural selection methods with genetic mutation of individuals. Each individual represents a solution candidate with fitness values for each observed objective. Populations consist of individuals. In each optimization step, a couple of individuals of the population are selected to survive with a probability proportional to their fitness values. These individuals are then mutated and recombined to generate the next population. This process is continued until a pre-defined stopping criterion is met.

A. Gene Structure

As already explained in Section IV, each hierarchical node of the graph is processed in isolation. Due to the bottom-up direction of the parallelization process, all child nodes deeper in the hierarchy are already processed, so that a Pareto-front of different solutions which may contain additional tasks is attached to each child node. The challenge of parallelizing a hierarchical node is to find an efficient node-to-task mapping for all direct child nodes and to select one of the parallel solutions for each child node. Thus, the parallelization algorithm is able to extract new parallelism at the current level of the hierarchy which can be combined with parallelism found deeper in the hierarchy.

The gene structure of the individuals for a hierarchical node to be parallelized is depicted in Figure 2. It consists of two parts. The first part maps a node N_i to a task T_j while the second part selects a parallel solution $S_{i,k}$ for each child node N_i. Thus, the individual consists of $NumberOfChildNodes * 2$ positions and can be efficiently implemented by an array of integer values. The gene in Figure 2, for example, specifies that nodes N_1 and N_2 are mapped to task T_1, while node N_3 is mapped to task T_2. In addition, the hierarchical parallel solution $S_{1,4}$ is chosen for node N_1, parallel solution $S_{2,3}$ is chosen for node N_2, etc.

The impact of the gene's configuration is visualized in Figure 3. The inner task graph structure for the hierarchical node to be parallelized is shown on the top right of the figure, according to the node-to-task mapping of the gene on the left hand side. Since data and control dependencies between the created tasks hinge on the chosen node-to-task mapping, the structure of the graph may change if one node



Gene Representation
Task Graph
Selected Parallel Solution for Node N1
Node-to-Task Mapping
T1 N1
T1 N2
T2 N3
T3 N4
T3 N5
T4 N6
T4 N7
Node 1 (Selected Point S4):
Execution costs: 162 cycles
Energy consumption: 0.16 nJ
Communicated data: 200
Number hierarchical tasks: 3
Hierarchical Parallel Solutions
S1,4 N1
S2,3 N2
S3,2 N3
S4,8 N4
S5,9 N5
S6,3 N6
S7,4 N7
Hierarchical Parallel Solutions (Pareto-frontiers)

Fig. 3. Impact of Gene Configuration on the Parallelized Hierarchical Node

is moved to another task. Thus, if node N_5 is moved to task T_2, for example, a new dependency between T_3 and T_2 arises which has to be taken into account for the evaluation of the objective's fitness values. Since all child nodes $N_1, .., N_7$ may also be hierarchical nodes, different parallel solutions may exist which were extracted deeper in the hierarchy. Thus, the parallelization algorithm has to choose one of these solutions from the Pareto-fronts of the child nodes. In the example of Figure 3, the hierarchical parallel solution $S_{1,4}$ is chosen for Node N_1, solution $S_{2,3}$ is chosen for Node N_2 and so on. Since all child nodes were processed in isolation, each one has its own Pareto-front of possible parallel solutions. The selected parallel solution contains a task graph for the child node which directly influences the execution time, the energy consumption and other objectives for the evaluation of this node. Thus, if a solution with more generated hierarchical tasks for node N_1 is chosen, the execution time for this node might be decreased, while the energy consumption is increasing due to, e.g., more communication. The next subsection explains the high-level models which are used to evaluate the fitness values of the different objectives.

B. Objective Evaluation

In the current implementation, we observe three objectives, namely *the execution time*, which reflects the speedup of the application, the *energy consumption* of executing the application on the embedded device, and the *communication overhead*, which gives a hint on the bus load of the parallelized application.

1) Execution time: The evaluation of the fitness value for the execution time of a gene is based on the algorithm of Sarkar [18] and our previous publication [6] and returns a linear execution time estimation. Due to the hierarchical structure and the cycle-freeness of the hierarchical sub-graphs, the fitness value is equal to the execution time of the longest (or *most critical*) path through the sub-graph of the parallelized hierarchical node. In the example of Figure 3, the longest path may be $T_1 \rightarrow T_3 \rightarrow T_4$, depending on the execution time and the communication delay of the tasks. The following Equations show the calculation of the fitness value for 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 $ETN(n, S_n)$ of the child nodes n which are mapped to task T_i and a constant task creation overhead TCO. The execution time of the child nodes depends on the chosen hierarchical parallel solution S_n:

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

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 is equal to the longest execution path of the node's sub-graph:

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

2) Energy consumption: The fitness value of the energy consumption of a gene's configuration contains energy costs which arise due to task spawning, statement execution and communication, like shown in the following equations.

The energy consumption $ICE(T_i)$ for the incoming communications 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 communications 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 $EE(T_i)$ which is annotated in the graph (it also depends on the chosen hierarchical parallel solution of the nodes) and the energy for the incoming and outgoing communication:

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

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

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

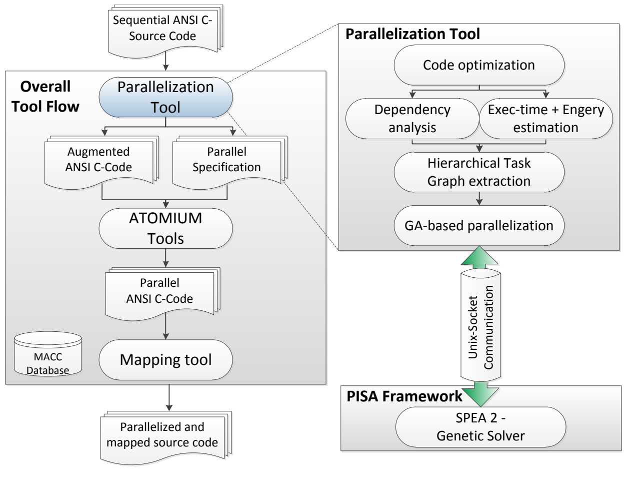


Fig. 4. Implemented Tool Flow

3) Communication overhead: In our current implementation, the evaluation of the communication overhead's fitness value is equal to the sum of the communicated bytes of all tasks multiplied by a specified communication delay:

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

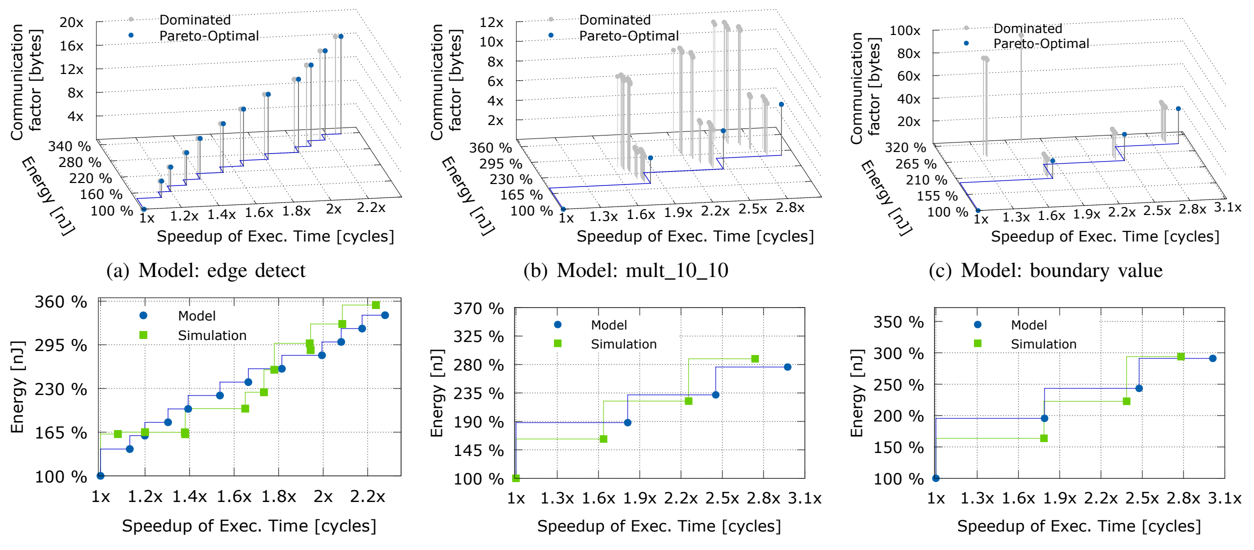
C. Gene-Mutation & Cross-Over

The mutation of a selected gene is done by modifying one position of the individual's gene with a fixed probability. Thus, one statement is moved from one task to another, or a different hierarchical parallel solution is chosen for one of the child nodes. The most challenging problem in the mutation step is to avoid the creation of cyclic – and thus invalid – solutions regarding the dependencies of the created task graph, since it is very easy to create such cyclic solutions. If, for example, node N_5 of Figure 3 is moved to task T_2 and N_6 is moved to T_3 within two mutation steps, T_2 would depend on T_3 and T_3 would depend on T_2 if communication is only allowed at the beginning and at the end of each task. Therefore, we added special checks which prevent that such a solution is created in the mutation and cross-over steps.

Recombining two different genes (cross-over) is done by cutting two genes at a randomly chosen position so that the left side of one gene is joined with the right side of the other one and vice versa. Here again, we are also preventing the generation of cyclic solutions.

VI. Experimental Environment

All described techniques are fully implemented and integrated in our parallelization framework which was previously published in [6] (cf. Figure 4(a)). The sequential application's source code which is used as input for the tool flow is parsed into a high-level intermediate representation. All tools are developed on top of the MACC framework [19] which is used to facilitate the communication between all processing steps of the tool flow. The framework stores intermediate results,



(a) Model: edge detect (b) Model: mult_10_10 (c) Model: boundary value

(d) Simulation: edge detect (e) Simulation: mult_10_10 (f) Simulation: boundary value

Fig. 5. Final parallel solutions based on the used models (a-c) compared to the evaluation on the MPARM/MEMSIM simulator (d-f)

the application's IR and a description of the target platform in a database, so that all tools can easily access this information.

The parallelization tool described in this publication extracts the parallelism of the application in a multi-objective aware manner and annotates the application's source code to express the parallelism of the chosen Pareto-point. This annotated source code and a parallel specification is passed to the ATOMIUM (MPA) tool suite [20], which is then implementing the extracted parallelism. Finally, a mapping tool is used to map the different tasks to the target platform.

The internal structure of the parallelization tool is depicted in Figure 4(b). In a pre-processing step, high-level optimizations like, e.g., constant propagation are applied to the source code to enable the extraction of more efficient parallelism. Afterwards, a dependency analysis and a platform dependent execution time and energy estimation tool are executed. The second tool is extracting this information by simulating the statements of the application, so that this information can be annotated to the hierarchical task graph, which is extracted in the succeeding step (cf. [6]). Finally, the presented GA-based algorithm can be executed to extract the parallelism of the application. The selection of individuals which should be mutated or recombined and the generation of the Pareto-fronts are done by the SPEA2 genetic solver [21] (part of the PISA framework).

VII. Experimental Results

To evaluate the validity of the results produced by our parallelization framework, we present results for benchmarks from the UTDSP suite [22], as well as additional meaningful embedded applications like a JPEG encoder. As target platform, we used the cycle-accurate MPARM simulator [23] which provides up to four separate single-core ARM processors. The simulator is equipped with a detailed energy model called MEMSIM, which makes it most suitable for our purposes.

Figure 5(a-c) shows the parallel solutions for three of the evaluated benchmarks which are determined by our novel parallelization framework based on the presented high-level models. The three objectives, namely speedup, energy consumption and communication, are visualized in a 3D-point diagram. The 3D-points are projected to the 2D plane to improve readability. The axes of the diagram are normalized to the solution with the lowest speedup and the minimal energy consumption. Figure 5(a-c) shows both, the Pareto-optimal solutions, as well as the dominated ones, for the top hierarchical node. The designer is now able to choose one of the Pareto-optimal solutions which complies with the best trade-off for his scenario. If, e.g., a speedup of 1.4x might be sufficient for the parallelization of the *edge detect* benchmark (c.f. Figure 5(a)), the amount of consumed energy (compared to the sequential execution) increases to around 200%. Nevertheless, if the solution with the highest speedup were chosen – like done by most existing parallelization frameworks, which are only optimizing the speedup – the energy consumption would increase to over 340%. Thus, 140% percent of energy could be saved here, by using our multi-objective aware framework. Similar optimization potentials can be observed for the benchmarks *mult* (cf. Figure 5(b)) and *boundary value* (cf. Figure 5(c)). Compared to the results of the edge detect application, it can be seen that less Pareto-optimal points were found, since many solutions are Pareto-dominated. Nevertheless, a speedup of up to 2.9x with 280% energy consumption (mult) and 3.1x with 310% energy consumption (boundary value) create a huge optimization potential for our multi-objective aware trade-offs.

Since the applicability of our framework strongly depends on the soundness of the used high-level models, Figure 5(d-f) opposes the Pareto-optimal points of our model to the simulated results on the target platform. The third objective (amount of communication) is not shown here, since it cannot be measured by the simulator. Nevertheless, the most important ones are speedup and energy consumption. We have configured the MPARM simulator to have the same amount

TABLE I
EVALUATION OF GENETIC PARALLELIZATION ALGORITHM

Benchmark	Time	#N	#Pop	#Ind	#Mut	#Cross	#S
adpcm enc.	00:49	27	1,376	146,003	27,164	91,616	5
bound. value	01:04	6	532	79,933	15,122	50,492	4
compress	09:07	131	6,552	592,124	110,426	371,048	5
edge detect	02:08	47	2,088	175,203	32,506	109,528	12
filterbank	01:36	4	212	22,198	4,223	14,012	6
fir 256	00:18	7	292	26,462	4,883	16,620	4
iir 4	00:51	7	564	82,224	15,373	51,996	6
jpeg2000	04:36	43	2,468	294,970	55,655	186,988	42
latnrm 32	00:13	11	460	37,122	6,884	23,172	4
mult 10	00:14	10	404	34,511	6,529	21,500	4
spectral	01:36	38	1,948	190,266	35,498	120,084	33

of cores as concurrently executed tasks appearing in the parallelized application. Thus, a solution with 2 concurrently executed tasks is executed on a platform with 2 cores. Of course, there can be several parallel regions in the application. All cores which are not executing threads at a given timeframe are put into idle mode to save energy. As depicted in the figures, the trend of increasing energy consumption for more expressed parallelism was confirmed by the simulator. In addition, the figures show that the points of our high-level model are comparable to the simulated ones which makes the model accurate enough to be used for our purposes.

We have summarized the results for all evaluated benchmarks in Table I. The columns contain information about the time in minutes which was necessary to parallelize the applications with the presented parallelization approach (*Time*), the number of processed nodes (*#N*), the number of generated populations (*#Pop*), the overall number of generated and evaluated individuals (*#Ind*), the number of mutated (*#Mut*) and recombined (*#Cross*) individuals and the number of offered Pareto-optimal solutions (*#S*) which are returned to the designer. The number of individuals and populations used to parallelize a hierarchical node is determined dynamically, based on the number of child nodes and the number of found hierarchical solutions. Thus, nodes with probably less parallel solutions are processed much faster. The shown numbers of populations, individuals, mutations etc. are summed up over all parallelized hierarchical nodes. As can be seen, our approach is able to create and evaluate individuals very fast due to the usage of the presented high-level models. To parallelize, e.g., the compress benchmark, more than half a million individuals were generated and evaluated in about 9 minutes which is only possible due to the usage of our accurate high-level models. The number of offered Pareto-optimal solutions varies between 4 and 42, depending on the available parallelism of the application. This also shows that our framework is able to give the designer a huge amount of freedom in finding good trade-offs for the parallelization process.

VIII. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this paper presents the first parallelization approach which uses genetic algorithms to partition embedded applications in a multi-objective aware manner. We have shown that our high-level models are accurate enough to enable trade-offs between the considered objectives, which are namely speedup, energy consumption and communication overhead. The applicability of the framework was demonstrated on several real-life benchmarks from typical embedded system application domains.

In the future, we would like to extend our framework and the used models to be capable of dynamic voltage scaling (DVS) to further optimize the energy consumption of the embedded device. Even if this might lead to a decrease of the application's speedup, the trade-off can be perfectly integrated in our framework. In addition, we would also like to evaluate the influence of other objectives (e.g., code size) as well.

REFERENCES

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

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

[3] S. Ha, "Model-based Programming Environment of Embedded Software for MPSoC," in *Proc. of ASP-DAC*. IEEE Computer Society, 2007.

[4] 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.

[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*. ACM, 2010.

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

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

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

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

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

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

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

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

[15] M. Qiu, J.-W. Niu, L. T. Yang *et al.*, "Energy-Aware Loop Parallelism Maximization for Multi-core DSP Architectures," in *Proc. of GreenCom*. IEEE Computer Society, 2010.

[16] Y. Wang, H. Liu, D. Liu *et al.*, "Overhead-aware energy optimization for real-time streaming applications on multiprocessor System-on-Chip," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, 2011.

[17] S. Cho and R. G. Melhem, "On the Interplay of Parallelization, Program Performance, and Energy Consumption," *IEEE Trans. Parallel Distrib. Syst.*, 2010.

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

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

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

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

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

[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.