# Automatic Extraction of Multi-Objective Aware Pipeline Parallelism Using Genetic Algorithms

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

Olaf Neugebauer
Informatik Centrum Dortmund e.V.
Dortmund, Germany
neugebauer@icd.de

## ABSTRACT

The development of automatic parallelization techniques has been fascinating researchers for decades. This has resulted in a significant amount of tools, which should relieve the designer from the burden of manually parallelizing an application. However, most of these tools only focus on minimizing execution time which drastically reduces their applicability to embedded devices. It is essential to find good trade-offs between different objectives like, e.g., execution time, energy consumption, or communication overhead, if applications should be parallelized for embedded multiprocessor system-on-chip (MPSoC) devices. Another important aspect which has to be taken into account is the streaming-based structure found in many embedded applications such as multimedia and network services. The best way to parallelize these applications is to extract pipeline parallelism. Therefore, this paper presents the first multi-objective aware approach exploiting pipeline parallelism automatically to make it most suitable for resource-restricted embedded devices. We have compared the new pipeline parallelization approach to an existing task-level extraction technique. The evaluation has shown that the new approach extracts very efficient multi-objective aware parallelism. In addition, the two approaches have been combined and it could be shown that both approaches perfectly complement each other.

## Categories and Subject Descriptors

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

## General Terms

Algorithms, Design, Languages, Performance

## Keywords

Automatic Parallelization, Embedded Software, Genetic Algorithms, Multi-Objective, Pipeline Parallelism, Energy


*CODES+ISSS'12,* October 7-12, 2012, Tampere, Finland.


## 1. INTRODUCTION

As a result of continuously increasing demands imposed on modern embedded systems, today the market share of multiprocessor system-on-chip (MPSoC) devices is growing at a tremendous rate in different application areas. Higher frequencies of single processing units are no longer achievable due to physical limitations. By using multiple cores in one MPSoC system it is possible to increase the performance of the platform or to execute the same work with reduced CPU frequencies, compared to a single-core platform. Unfortunately, these benefits do not come for free. Nowadays, most embedded applications are written in sequential ANSI-C code and have to be partitioned manually into concurrently executed tasks to take advantage of the additional processing units of an MPSoC. Since manual parallelization tends to be very error-prone and time-consuming, many (semi-) automatic parallelization tools have been invented in the last decades, but most of them are not well applicable to resource-restricted embedded systems.

This is not surprising, since most of these tools were developed to extract parallelism for desktop or even high-performance architectures. For these systems, the minimization of the execution time is very often the only objective, since a huge amount of computational power and – seen from an embedded systems perspective – an almost inexhaustible amount of energy are available. The situation changes if one is looking at embedded devices. Most of them have, e.g., only a very limited amount of computational power, small memories, and are obtaining their energy from a small battery. Thus, multiple objectives have to be considered at the same time during parallelization for resource-restricted systems. For example, it might be a good idea to reduce the amount of extracted parallelism if a given timing criterion is still met, instead of extracting as much parallelism as possible like done by most existing parallelization tools. If less parallelism is extracted, some of the cores can be switched into idle mode and a platform with less cores can be used which saves a significant amount of energy.

In addition to the fact that software for embedded devices should be parallelized using multi-objective aware techniques, special characteristics of embedded software should also be taken into account. By analyzing these applications it is noticeable that many of them have a streaming-based structure. Examples include applications in the domain of network services, voice- and image-processing as well as multimedia tasks like video decoding. To efficiently parallelize these applications, different pipeline stages should be extracted from a (nested) loop body, which execute different

iterations of these stages in an interleaved way. Unfortunately, simple loop-level parallelism which is just executing different iterations of a loop in parallel can often not be applied, due to frequently occurring complex dependencies in embedded software. Thus, parallelization tools tailored towards special requirements of embedded devices should be able to extract so-called pipeline parallelism to generate efficient code for MPSoCs. Up to now, all previously published pipeline parallelization approaches are only optimizing a single-objective which limits their applicability to embedded devices.

Therefore, to the best of our knowledge, this paper presents the first multi-objective aware parallelization approach which is able to extract pipeline parallelism from sequential embedded applications. The new approach is based on genetic algorithms and has been compared against the previously published multi-objective aware task-level parallelization approach presented in [7]. The evaluation has shown that the new approach extracts very efficient parallelism for embedded real-world applications. In addition, the two approaches have been combined into one framework and it could be shown that both perfectly complement each other.

The main contributions of this paper are as follows:

1. To the best of our knowledge, this is the first automatic multi-objective aware parallelization approach which uses genetic algorithms to extract pipeline parallelism from sequential applications.

2. In contrast to high-performance computing, this framework focuses on requirements of constraint-driven embedded devices.

3. We use high-level models to evaluate different objectives without re-simulating the application, which drastically decreases the optimization time.

4. We combine this new approach with an existing task-level parallelism extraction technique and show how well both approaches complement each other to extract very efficient parallelism from embedded applications.

The rest of this paper is organized as follows: Section 2 gives a survey of related work, followed by a motivating example in Section 3. Afterwards, Section 4 gives a short explanation of the program dependence graph which is employed as intermediate representation. Section 5 then describes the multi-objective aware approach which is able to extract pipeline parallelism. The integration of this approach into an existing parallelization framework is described in Section 6, before the effectiveness and efficiency of the approach are evaluated in Section 7. Finally, Section 8 summarizes this paper and gives directions for future work.

## 2. RELATED WORK

In the last decades, many semi- and fully automatic techniques have been published that try to simplify the task of parallelizing a given sequential embedded application. While earlier publications on instruction-level parallelism (see, e.g., [29] and [13]) were targeting VLIW processors, recent multicore architectures rely on the extraction of more coarse-grained software-based parallelism. The latter one can be grouped into at least the three categories of *task-level-*, *loop-level-* and *pipeline parallelism* which are discussed in the following paragraphs.

A representative work of the first category, namely *task-level parallelism*, was presented by Hall et al. [11]. The authors present a technique which automatically extracts task-level parallelism as part of the SUIF Parallelizing and Optimizing Compiler framework [12]. Their approach is based on an interprocedural analysis and is not limited to function boundaries. Ceng et al. developed a semi-automatic parallelization assistant [3]. The application code is transformed into a weighted statement control data flow graph which is subsequently processed by a heuristical clustering algorithm, generating tasks after several iterations. The approach proposed by Ceng requires a user feedback loop to steer the granularity of the parallelized program.

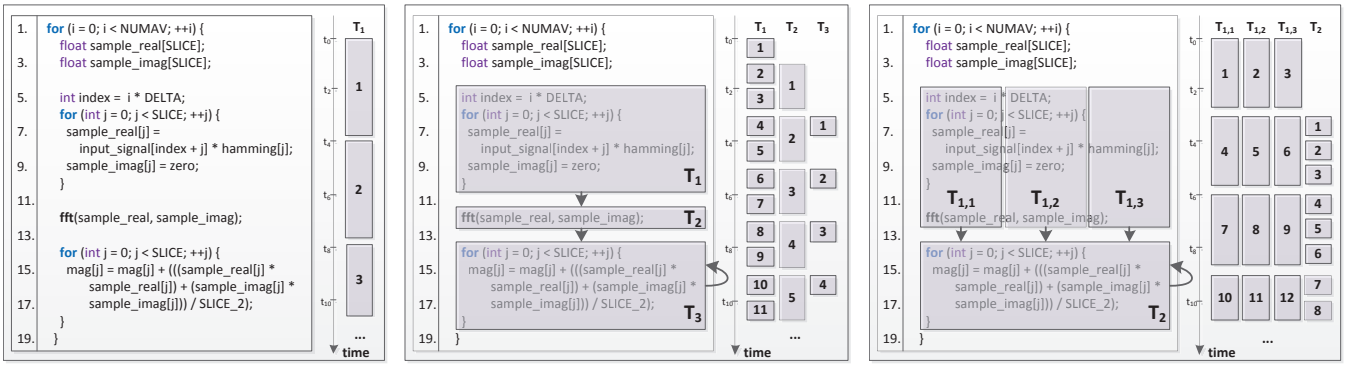
More recent work on automatic extraction of task-level parallelism was presented in our previous publications [8] and [7] describing two techniques based on Integer Linear Programming (ILP) and genetic algorithms. While the ILP-based approach in [8] is only able to optimize the execution time, the second work in [7] is able to extract task-level parallelism in a multi-objective aware manner. However, the potential of parallelizing loops with task-level parallelism is very limited, especially of those with loop-carried dependencies. Therefore, the approach presented in this paper extracts pipeline parallelism to provide a remedy for this restriction.

Other frameworks which extract task-level parallelism were presented by, e.g., Verdoolaege et al. [26], Nikolov et al. [17], Sarkar [22], and Ottoni [18].

Representative publications of the second category, fine-grained *loop-level parallelism*, were developed by Franke et al. [9] and Chandra et al. [4]. Franke describes an approach which automatically extracts loop-level parallelism from C applications for DSP architectures with multiple address spaces. This approach also applies program code recovery techniques which, e.g., replace pointer arithmetic with array accesses to enable more precise data analysis. Chandra parallelizes loops of sequential applications for CC-NUMA (cache-coherent non-uniform memory access) architectures.

Another popular technique to extract fine-grained data-level parallelism from sequential applications is the use of polytope models. Here, loops are transformed into polytopes, which describe the iteration space and dependencies of nested loops by linear inequalities spanning a geometric object in an $n$-dimensional space. 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. Papers by Lengauer [15] and Bondhugula et al. [2] are representative publications in this area.

The last mentioned category, *pipeline parallelism*, is most relevant for this work, since the presented approach of this paper aims at the extraction of this kind of parallelism. Raman et al. [20], Tournavitis et al. [24], [25], and our previous work in [6] presented different approaches to achieve this. They are able to automatically split loops into different pipeline stages and further increase the application's performance by splitting pipeline stages into additional tasks. While our approach in [6] uses a mathematical problem description based on Integer Linear Programming, Raman and Tournavitis lack detailed cost models and only use heuristic clustering approaches. However, in contrast to the work of this publication, all those approaches aim at the minimization of the execution time as the only optimization objective.




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

**Figure 1: Pipelining-based Loop Level Parallelization - Motivating Example**

Additional approaches in this area were presented by Liu et al. [16] which eliminate loop-carried dependencies by retiming the execution of statements in a loop. They are moving executions of statements to earlier iterations of the loop. Thus, dependencies may change which creates the opportunity to parallelize different iterations of the loops.

Gordon et al. [10] present a compiler framework which combines the extraction of task, data, and pipeline parallelism for applications written in the programming language StreamIt [23]. The designer has to extract tasks manually by defining independent actors connected by explicit data channels. The algorithms described in [10] search for parallelism in the given task structure.

All aforementioned approaches, with the exception of our previous publication in [7], which extracts a different type of parallelism, have in common that the minimization of the execution time is the only considered objective. Thus, their applicability to resource-restricted embedded devices is very limited. However, some more recent publications try to optimize the energy consumption of parallelized applications. Qiu et al. [19] propose an energy-aware loop parallelization method. This method optimizes performance and energy consumption while parallelizing an application. The approach follows a two-phase strategy. First, it parallelizes loops to optimize the performance of the application. Afterwards, it tries to decrease the energy consumption of the system based on the previously parallelized application. Due to the decoupled phases, this might lead to sub-optimal results. Wang et al. [27] also implement a two-phase strategy to parallelize streaming applications. Cho et al. [5] present theoretical formulas which describe the interplay of program performance and energy consumption of a parallelized application.

Thus, to the best of our knowledge, the work presented in this paper is the first one extracting pipeline parallelism, which is very efficient for many embedded applications, in a multi-objective aware manner.

## 3. MOTIVATING EXAMPLE

The extraction of coarse-grained task-level parallelism is a very efficient technique to parallelize large independent blocks of an application such as function calls which can be executed concurrently. However, those approaches lack the possibility to extract parallelism from loops, especially from those with loop-carried dependencies. Thus, the new approach presented in this paper extracts pipeline parallelism in a multi-objective aware manner.

Before the extraction algorithm is described, a small motivating example should demonstrate the potential of pipeline parallelism for an embedded real-world application. The code snippet depicted in Figure 1(a-c) represents the main computation loop of the spectral benchmark [14]. On the left-hand side of each figure, the application's source code and the way how the statements are partitioned into concurrently executed tasks are shown. The right-hand side shows the time at which the iterations of the tasks are executed.

Simple loop-level parallelism which is just executing different iterations of the whole outer loop in parallel cannot be applied here, due to a loop-carried dependency of the second inner loop starting in line 14 of Figure 1(a-c). There, the content of the mag-array ($mag[j]$) generated in the previous iteration of the outer loop is consumed which serializes the execution of loop iterations. One way to provide a remedy is the extraction of pipeline parallelism which splits the loop horizontally and vertically into concurrently executed tasks. While horizontal splits distribute the statements of the loop body into disjunctive tasks which are executed in a pipelined manner, vertical splits further partition the different loop iterations of the created tasks into additional sub-tasks.

An example where horizontal splits are applied to the spectral application is depicted in Figure 1(b). There, the loop body of the outer loop is divided into three tasks $T_1$, $T_2$, and $T_3$. The benefit of such a parallelization can be seen on the right-hand side of Figure 1(b). As soon as the first iteration of task $T_1$ has completed its work, the required data is sent to $T_2$. Now, $T_1$'s second and third iteration can be executed in parallel to the first iteration of task $T_2$. Afterwards, $T_2$ sends its data to $T_3$, so that all three tasks are executing their work in a pipelined manner.

As can be seen, this example is not well balanced, since $T_3$ has to wait for data of task $T_2$ after each iteration. To circumvent this problem, different loop iterations of the created tasks can be executed in parallel, generated by so-called vertical splits. The extracted parallelism in Figure 1(c) combines horizontal and vertical splits and is much more efficient (at least regarding execution time) than the solution which is only generating pipelined tasks, shown in Figure 1(b). Tasks $T_1$ and $T_2$ of Figure 1(b) are joined in this solution, but this new task is now vertically divided into three subtasks. These tasks are now executing the first, second and third loop iteration of the statements assigned to this task

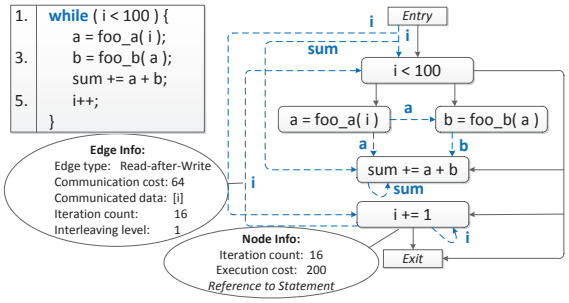



**Figure 2: Program Dependence Graph**

in parallel. As soon as all tasks have communicated their data to the consuming task $T_2$, the iterations four, five and six are executed in parallel to the first three iterations of task $T_2$ and so on. By combining both kinds of splits, the work-load is optimally balanced.

The extraction of this kind of parallelism has already been studied before, e.g., in [6] and [20]. However, in contrast to these earlier approaches, the one presented in this paper is able to consider multiple objectives at the same time while extracting pipeline parallelism. The solution in Figure 1(c) is very efficient regarding execution time, since all tasks execute their work in parallel without any delays. Nevertheless, four processing units are running at maximum speed, which consumes a significant amount of energy. In addition, a large amount of data has to be communicated concurrently which causes high bus contention and may block the execution of the tasks. Even if the solution shown in Figure 1(b) seems to be less efficient, a platform with fewer cores can be used to execute this solution which may save a huge amount of energy. Thus, if timing criteria are still met, solutions with less extracted parallelism may also be beneficial for other objectives.

# 4. PROGRAM DEPENDENCE GRAPH

The multi-objective aware pipeline parallelization approach presented in this paper employs an intermediate representation called augmented program dependence graph (PDG). The PDG is extracted for each loop of the application and combines both, control- and data-dependencies which makes it most suitable for our purposes. An example is given in Figure 2. The graph contains one entry, one exit and one node for each statement of the considered loop's body. Control flow dependencies are visualized by solid directed edges while data dependencies are represented by dashed arrows. Even for this small example many dependencies exist which makes it nontrivial to detect pipeline parallelism based on the graphical representation.

The presented parallelization approach of this paper uses high-level models to evaluate the benefit of a parallelized solution candidate of the considered loop in a very efficient way (c.f. Section 5.2). These models are used to determine the influence of a newly extracted task for all considered objectives, because task creation overhead and communication costs may prohibit efficient parallel execution. Therefore, necessary information like, e.g., estimated execution costs and the iteration count of the attached statement are annotated to the nodes of the graph (cf. *Node Info* in Figure 2). In addition, the edge type, communication costs, the communicated data, the iteration count and the interleaving level – describing the minimal amount of loop iterations which can be executed before the data is consumed by the target node – are annotated at the edges of the graph (cf. *Edge Info* in Figure 2). By combining the graphical representation with the additional cost information, all necessary information is available to extract well-balanced, efficient pipeline parallelism. The data is automatically extracted by the framework described in Section 6.

# 5. MULTI-OBJECTIVE AWARE EXTRACTION OF PIPELINE PARALLELISM

The parallelization algorithm presented in this paper is able to extract pipeline parallelism from loops of sequential ANSI-C applications in a multi-objective aware manner. Therefore, each loop(-nest) is transformed into an augmented program dependence graph and processed by a genetic algorithm presented in this section. As a result, a front of Pareto-optimal solutions representing parallelized versions of the considered loop is returned. The method to combine the different solutions of all loops to a global result is presented hereafter in Section 6.

## 5.1 Chromosome Representation

Genetic algorithms are favored for solving optimization problems in a multi-objective aware manner. They start with an initial population consisting of individuals representing possible solution candidates for the optimization problem. Each individual is characterized by a gene sequence called chromosome which describes the configuration of the optimization values. In each optimization step, some promising individuals are chosen to be mutated or recombined with other individuals to create a new population. This process is then repeated until a pre-defined termination criterion, e.g., a maximum number of generated populations, is met.

The challenge of using genetic algorithms is to map the optimization problem to genes of the individuals' chromosomes in such a way that they can be efficiently evaluated for different objectives. In the case of a genetic pipeline parallelization approach, disjunct pipeline stages of a loop(-nest) which are executed in an interleaved way, like shown in the example of Section 3, should be extracted. This can be achieved by horizontally splitting the loop's body. To further increase the application's performance, the genetic algorithm should also be able to vertically split these pipeline stages so that different loop iterations of the stage can be executed concurrently. Thus, both splits have to be part of each individual's chromosome, like depicted in Figure 3. As can be seen, the structure of the chromosomes is divided into two parts. The mapping of PDG nodes, representing statements of the loop's body, to tasks is shown on the left-hand side. Here, each node is mapped to exactly one task which represents one of the extracted pipeline stages due to horizontal splits. On the right-hand side, an integer variable declares how often each task, i.e., pipeline stage, is split into sub-tasks which are executing different loop iterations of the stage in parallel. In the example of Figure 3, the statements represented by nodes $N_1$ and $N_2$ are mapped to task (or pipeline stage) $T_1$, while node $N_3$ is mapped to task $T_2$. The first pipeline stage $T_1$ is split $S_1$ times, $T_2$ is split $S_2$ times, and so on. Each chromosome can be represented by an array of integers. The size of this array is as large as the number of statements contained in the loop's body plus the

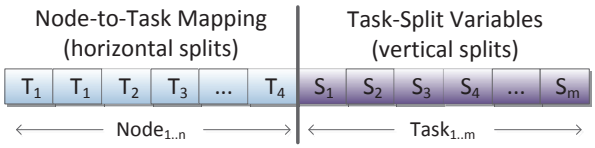



**Figure 3: Structure of Chromosomes**

number of maximal pipeline stages to generate. Thus, each chromosome can be decoded very efficiently which enables the generation of a huge amount of individuals.

The impact of a chromosome's configuration is visualized in more detail in Figure 4. As can be seen in the top part of the figure, nodes $N_1$ and $N_2$ are mapped to the first pipeline stage ($T_1$), $N_4$ and $N_5$ are mapped to stage $T_3$, while $N_3$ and $N_6$ are mapped to stages $T_2$ and $T_4$. Thus, $T_1$ starts with the execution of the first iteration of nodes $N_1$ and $N_2$. Afterwards, the generated data is sent to pipeline stages $T_2$ and $T_3$ so that the next iteration of $T_1$ is executed concurrently to the first iteration of $T_2$ and $T_3$, analogously to the example of Section 3. The dependencies between the different extracted pipeline stages rely on the node to task (i.e. node to pipeline stage) mapping. Thus, if one node is moved from one task to another, e.g., by mutating an individual, the dependencies between the stages may change which also influences the execution order of the tasks. If, e.g., node $N_2$ would be moved from $T_1$ to $T_2$, a new edge has to be added between the stages of tasks $T_2$ and $T_3$ which has to be taken into account while mutating and recombining individuals. Thus, even small changes in the mutation steps may have big influences for the evaluation of different objectives.

The genes of the vertical splits are shown in the bottom-left part of Figure 4. Here, the pipeline stages $T_1$ and $T_3$ are split once, resulting in two sub-tasks for both pipeline stages. Stages $T_2$ and $T_4$ are not split in this example so that each iteration of these stages is executed sequentially. The timing which belongs to the chromosome's configuration is visualized in the bottom-right corner of Figure 4. Due to the split of stage $T_1$, the first two iterations of this stage are executed concurrently. As soon as the results are available, the data is communicated to $T_2$ and both instances of task $T_3$. In the next time frame, iterations 3 and 4 are executed in parallel to the first and second iterations of $T_2$ and both sub-tasks of $T_3$. After four time slots, all six tasks are concurrently executing the statements assigned to their pipeline stage. Depending on task creation and communication costs, this configuration of the genes' values might represent a good solution candidate for the given example regarding executing time. Nevertheless, six tasks are executed in parallel so that many processing units have to execute their work concurrently which consumes significantly more energy if voltage scaling is not applied. Thus, other allocations of the chromosome's decision values might lead to solutions which reduce other limited resources as well.

## 5.2 Evaluation of Objectives

Since the quality of the final Pareto-optimal solutions returned by a genetic algorithm primarily depends on the population sizes used and the number of generated populations, the configuration of an individual's chromosome must be evaluated very efficiently. Thus, simulating each solution candidate which is generated by the genetic algorithm on the target platform is not an option since it would be too time

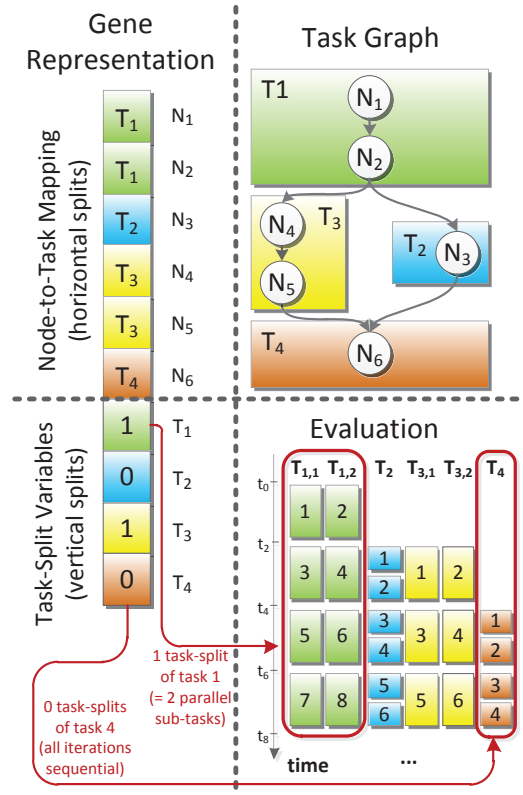



**Figure 4: Impact of Genes' Configuration**

consuming. Instead, the employed genetic parallelization algorithm uses high-level models to evaluate the different objectives. Here, we adapted the high-level models presented in [7] since these models can be evaluated very efficiently. In addition, the paper has also shown that the results of the objective values generated by the models are comparable to the ones extracted by the MPARM simulator [1], which is also used as target platform in this paper. The current implementation of our framework provides models for the objectives *execution time*, *energy consumption*, and the amount of *communicated data*. These models are described in the following.

Since different iterations of the same task may or may not depend on each other, the loop iterations are virtually unrolled for the calculation within these models. Like depicted in the right part of Figure 4, two circumstances can create dependencies between different tasks and their iterations. First, one iteration of task $T_i$ depends on an iteration of task $T_k$, if data- or control-dependencies between two statements with the matching interleaving level of these task exist, like shown in the top right part of Figure 4. Second, different iterations of a task $T_i$ may depend on each other controlled by the number of splits (see bottom-right part of Figure 4). Since task $T_1$ is split into two sub-tasks, iteration one and two of Task $T_1$ can be executed concurrently, while the third and fourth iteration depend on iteration one and two, respectively. If task $T_1$ would not be split, all iterations of this task would depend on each other. Thus, the variable $T_i^j$ denotes the $j$th iteration of task $i$ in the following.

Constants like, e.g., a constant task creation overhead or a communication multiplier can be adjusted in our framework

to support different target architectures. In addition, values like execution time or the amount of communicated data are extracted automatically by pre-processing tools of our framework and are annotated to the augmented program dependence graph of the loop to parallelize.

### 5.2.1 Objective 1: Execution time

To evaluate the objective value representing the execution time of a parallelized loop, high-level models based on the linear models of Sarkar [21] and our previous publication in [7] are used. Both models were developed to evaluate task-level parallelism so that we had to extend them to correctly handle different loop iterations and their dependencies for pipeline parallelism. The returned value of our proposed model is equal to the cycles of the longest (or most critical) execution path of the loop to parallelize. The following equations will describe the evaluation in a formal way.

The first component which is necessary to calculate the overall execution time of the parallelized loop is the execution time of task $T_i$ in iteration $j$. In our model, the execution time of a task $T_i$ is uniformly dispersed over all loop iterations of the task. Thus, the execution time $ET(T_i^j)$ of task $T_i$ in iteration $j$ is equal to the sum of the execution times $ETN(n)$ of the nodes $n$ which are mapped to $T_i$, divided by the number of loop iterations $LI$:

$$ET(T_i^j) = \sum_{n \in Nodes(T_i)} \frac{ETN(n)}{LI}$$

Based on the execution time of one iteration of a task, path costs can be calculated which denote the time until the $j$th iteration of a task $T_i$ is executed, including all iterations of its predecessors. The path costs $PC(T_i^j)$ of a task $T_i$ in its $j$th iteration are equal to the sum of $T_i$'s execution costs of the iteration $ET(T_i^j)$ and the path costs of the most expensive predecessor tasks $T_\ell^k$ including the costs of the communication $CC(T_\ell^k, T_i^j)$ between $T_\ell^k$ and $T_i^j$. The communication costs can be adapted to different architectures using a platform-dependent communication overhead. Due to the recursive structure of this formula, the costs of all indirect predecessors are also included:

$$\begin{aligned} PC(T_i^j) &= ET(T_i^j) + \\ &\quad \max\{PC(T_\ell^k) + CC(T_\ell^k, T_i^j) | T_\ell^k \in Pred(T_i^j)\} \end{aligned}$$

The overall execution costs are composed of the platform-dependent configurable task creation costs $TCO$, multiplied by the number of created tasks $NT$ plus the most expensive path costs $PC(T_i^{LI})$ of all tasks $T_i$ in their last iterations $LI$:

$$OverallET = TCO * NT + \max\{PC(T_i^{LI}) | \forall T_i \in Tasks\}$$

The changes made to the original model presented in [7] are mainly that the execution time of a task $T_i$ is now dispersed over its loop iterations $j$ and that each iteration is considered on its own to make the models applicable for pipeline parallelism. In addition, the task creation overhead is now added once before the extracted tasks are executed.

### 5.2.2 Objective 2: Energy consumption

The objective value describing the energy consumption consists of energy costs produced by task creation, communication overhead and the execution costs of statements which are mapped to the tasks.

The incoming communication energy costs $ICE(T_i^j)$ for task $i$ in iteration $j$ are equal to the sum of a constant incoming communication energy overhead $ICEO$ plus the number of transferred bytes $\#Bytes(d)$ multiplied by a platform-dependent communication energy factor $ICM$:

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

The outgoing communication energy $OCE(T_i^j)$ is calculated analogously to $ICE(T_i^j)$:

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

The energy $E(T_i^j)$ which is necessary to execute iteration $j$ of task $i$ contains both, the incoming ($ICE(T_i^j)$) and outgoing ($OCE(T_i^j)$) communication energy costs, increased by the energy $EN(n)$ which is necessary to execute the statements mapped to task $T_i$. The energy consumption is also dispersed in equal parts over all loop iterations of the task:

$$E(T_i^j) = ICE(T_i^j) + OCE(T_i^j) + \sum_{n \in Nodes(T_i)} \frac{EN(n)}{LI}$$

Finally, the overall energy consumption $OverallEnergy$ includes a constant energy overhead for task creation $TCE$ multiplied with the number of created tasks, increased by the energy which is consumed by all tasks $E(T_i^j)$ in each iteration:

$$OverallEnergy = TCE * NT + \sum_{i \in Tasks} \sum_{j \in \{0..LI\}} E(T_i^j)$$

Here, the main difference to the original model from [7] is that the energy consumption is modeled for each loop iteration separately.

### 5.2.3 Objective 3: Communication Overhead

The evaluation of the communication overhead is the simplest one and is identical to the original model presented in [7]. Here, all communicated data is summed up and multiplied with a platform-dependent communication factor:

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

## 5.3 Mutation and Recombination

Genetic algorithms generate new solution candidates (i.e. new individuals) by modifying values of the genes of existing individuals. This is done by a combination of mutation and recombination. In the first case, one gene of the chromosome is modified with a given probability. In our case, one statement is moved from one task to another – in case of a horizontal-cut change – or the number of splits of a newly created task is modified. Thus, dependencies between different tasks or at least between different iterations of one task may change in such a mutation step which has to be considered for the evaluation of the solution candidates.

In the second case, the recombination of individuals, also known as cross-over, cuts the chromosomes of two individuals at a random position to combine the left-hand side of the first chromosome with the right-hand side of the other side and vice versa. This operation simulates the combination of profitable individuals to new solution candidates.

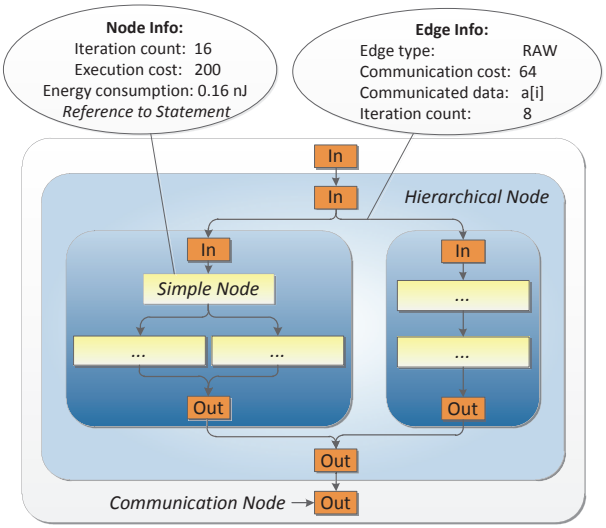



**Figure 5: Hierarchical Task Graph**

Both, mutation and recombination are processed until a given number of populations are evaluated so that a front of Pareto-optimal solutions of the parallelized loop is returned.

## 6. FRAMEWORK INTEGRATION

The task-level parallelization approach presented in [7] is the first framework that extracts parallelism in a multi-objective aware manner. The framework is able to automatically extract dependencies and additional information like, e.g., estimated execution times of statements of the application which are necessary to define our high-level models. In addition, our new multi-objective aware pipeline parallelization approach perfectly complements the approach presented in [7], since the older approach has only very limited support to parallelize loops of embedded applications.

The approach presented in [7] employs an annotated hierarchical task graph as intermediate representation like shown in Figure 5. The hierarchical structure of the graph is based on the hierarchical structure of the application's source code. Thus, e.g., function bodies, loops, or conditional statements create new hierarchical levels which can be parallelized in isolation. Unfortunately, the hierarchical task graph hides loop-carried dependencies which makes it unsuitable for the extraction of pipeline parallelism. The approach starts to extract task-level parallelism in a bottom-up search strategy which means that it starts to parallelize the inner-most hierarchical node. Each node is processed by a genetic algorithm which extracts a front of Pareto-optimal solutions which are added as solution candidates for the parallelized node. As soon as all nodes on the same level of the hierarchy are processed, the algorithm moves upwards in the hierarchy knowing that a Pareto-front of solution candidates is attached to each child node. At the next hierarchical level, the algorithm has the possibility to extract new tasks and to recombine this solution with parallel solutions which were found deeper in the hierarchy.

This encapsulated parallelization technique is the key which enables to integrate different parallelization approaches into one framework. Each approach is executed to return solution candidates representing beneficial parallelized versions of the statement represented by the node. The only requirement is that each solution can be evaluated for all considered objectives. Thus, a front of Pareto-optimal points from different approaches can be combined to be used in the following parallelization steps upwards in the hierarchy of the graph.

Here, we combined the multi-objective aware pipeline parallelization approach with the existing task-level technique. While the task-level parallelization approach can be applied to each hierarchical node of the graph, the pipeline parallelization approach is limited to loops. Thus, as soon as a node representing a loop is reached in the parallelization process, the corresponding program dependence graph is extracted and the genetic pipeline parallelization approach is executed. Afterwards, the results are added to the Pareto front. Due to the encapsulated technique, it is also possible to use only a subset of the parallelization approaches offered, like, e.g., only the pipeline parallelization approach.

As soon as the top node of the hierarchical graph is reached, the final front of Pareto-optimal solutions is returned to the application designer, who can choose the solution which represents the best trade-off for the given application scenario.

## 7. EXPERIMENTAL RESULTS

To evaluate the efficiency of our multi-objective aware parallelization framework we present results achieved from the UTDSP benchmark suite [14] containing representative real-world embedded applications. In addition, we also evaluated other meaningful embedded applications like, e.g., a JPEG encoder. As target platform we used the cycle-accurate MPARM simulator [1] which provides up to four single-core ARM processors. The simulator is also equipped with a detailed energy model called MEMSIM [28] which makes it most suitable for our purposes.

Figure 6(a-d) shows detailed results for four of the considered benchmarks. As explained in Section 5.2, our current framework is optimizing for the three objectives of speedup (execution time), energy consumption of the system and the communication overhead introduced by the extracted parallelism. These objectives are arranged on the $x$-, $y$- and $z$-axes in three dimensional diagrams. All axes are relative to the sequential solution which is located at the bottom-left point of the diagrams with a speedup of $1\times$, 100% energy consumption and zero communication overhead. For better readability we projected the points of the 3D-diagrams to the $x$-$y$-plane. Thus, the communication overhead of different solutions can be compared by the height of the vertical bars. Each diagram contains both, Pareto-optimal and Pareto-dominated points of the final solutions generated by our framework. Of course, only the Pareto-optimal ones are returned to the application designer. To compare the efficiency of the new multi-objective aware pipeline parallelization approach presented in this paper with the previously published multi-objective aware task-level parallelization technique contained in the framework presented in Section 6, different shapes are used for both types. In addition, the diagram also shows points which contain some parallel sections produced by task-level and some sections produced by pipeline parallelism. Thus, these points are produced by a combination of both approaches and are labeled as a MIXED solution.

By analyzing the results for the different benchmarks one can see that the number of Pareto-optimal solutions re-

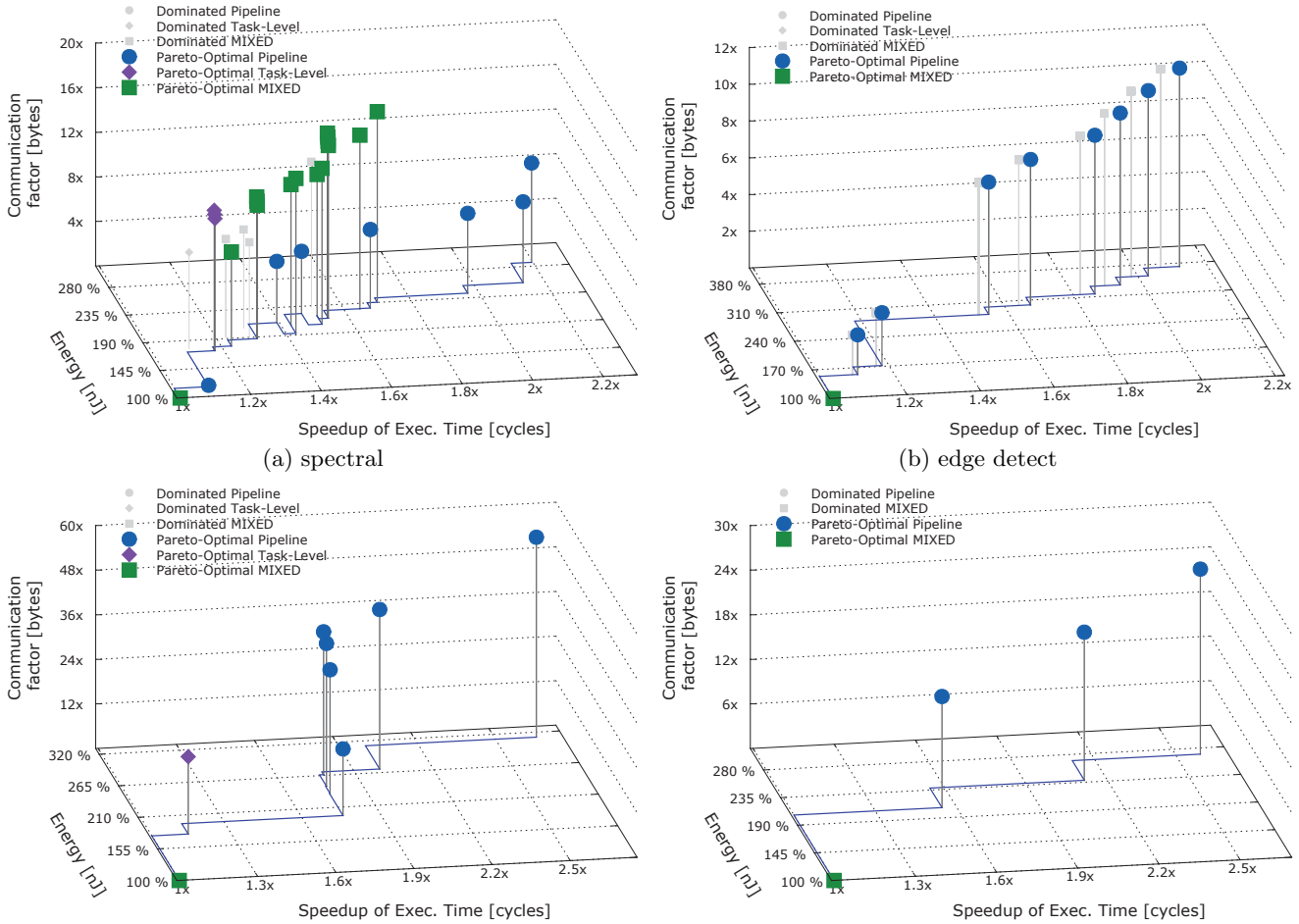



**Figure 6: Final parallel solutions returned by the parallelization framework**

turned to the application designer ranges from 4 up to 25 solutions which show huge optimization potential for different objectives. The solution with the highest speedup for, e.g., the filterbank benchmark (cf. Figure 6(c)) reduces the execution time by a factor of nearly 2.7×. Even if this solution drastically reduces the execution time of the application, it requires the highest communication overhead. In addition, all cores of the platform are executing threads in parallel which increases the energy consumption of the system to around 320% compared to the sequential solution. If the application designer knows that, e.g., a speedup of 1.9× is sufficient to meet his timing requirements he can choose a solution with less extracted parallelism. Thus, some of the cores can be switched into idle mode for some time and a platform with less processing units can be chosen. This reduces the energy consumption to less than 270% for this solution of the filterbank benchmark which is very efficient compared to 320% energy consumption for the solution with the highest speedup. In addition, the amount of inserted communication is also reduced. The solution with a speedup of 1.7× even reduces the energy consumption to less than 200% which highlights the optimization potential of our multi-objective aware approach. These observations can be made for the other evaluated benchmarks as well.

As already observed in [6], pipeline parallelism is able to generate solutions with a higher speedup compared to solutions which are extracted by task-level parallelism for many embedded applications. When looking at the results of our multi-objective aware framework, this observation is also confirmed, here. The solutions of Figure 6(a-d), which are extracting the highest speedup for the considered applications, are always generated by pure pipeline parallelism. However, these solutions also consume the highest amount of energy. This trade-off can be seen best in Figure 6(a) which shows the solutions of the spectral benchmark. This application was also used as the motiving example in Section 3. Here, 3 of 4 solutions with a speedup of less than 1.2x are generated by the already published coarse-grained task-level extraction approach. Even if the speedup is not as high as the speedup of the other approaches, only a small increase in energy consumption was observed. Nevertheless, a high amount of communication has to be inserted for this kind of parallelism depending on the structure of the application. 14 solutions with speedups between 1.2× and 1.7× are created by a combination of task-level and pipeline parallelism. Thus, the energy consumption is slightly increased but is at least lower than for solutions generated by pure pipeline parallelism. Finally, the solution candidates with more than 1.7× speedup are extracted by the pipeline parallelization approach presented in this paper. As can be

| Benchmark | Time[1] | #N | #Pop | #Ind | #Mut | #Cross | #S(TL,PL,MI) |
|---|---|---|---|---|---|---|---|
| adpcm encoder | 01:04 | 36 | 1,520 | 151,049 | 28,154 | 98,766 | 5(2,2,1) |
| boundary value | 01:11 | 12 | 644 | 83,331 | 15,804 | 54,032 | 4(0,3,1) |
| compress | 14:31 | 336 | 10,444 | 821,854 | 161,617 | 608,250 | 5(0,4,1) |
| edge detect | 04:48 | 105 | 2,872 | 196,720 | 38,125 | 137,788 | 9(0,8,1) |
| filterbank | 02:39 | 7 | 412 | 51,035 | 15,005 | 136,485 | 8(1,6,1) |
| fir 256 | 00:39 | 13 | 388 | 29,607 | 5,863 | 22,317 | 4(0,3,1) |
| h264_ldecode_block | 01:40 | 115 | 3,520 | 251,166 | 49,943 | 179,088 | 6(5,0,1) |
| h264_ldecode_macroblock | 02:30 | 51 | 2,168 | 238,007 | 45,870 | 170,351 | 5(2,1,2) |
| iir 4 | 14:35 | 13 | 852 | 103,294 | 21,298 | 92,830 | 4(0,3,1) |
| jpeg2000 | 04:49 | 62 | 2,868 | 313,047 | 62,630 | 231,390 | 45(0,27,18) |
| latnrm 32 | 01:34 | 17 | 636 | 53,462 | 11,931 | 46,358 | 4(0,3,1) |
| mult 10 10 | 02:45 | 36 | 1,060 | 70,442 | 14,984 | 60,399 | 4(0,3,1) |
| spectral | 03:04 | 51 | 2,260 | 211,023 | 41,667 | 160,477 | 25(3,7,15) |

**Table 1: Evaluation of Genetic Parallelization Algorithm**

seen in the figure, other Pareto-dominated solutions with task-level parallelism, pipeline parallelism and also a mixture of both techniques were generated but not returned to the application designer. Other benchmarks like, e.g., the edge detect benchmark in Figure 6(b) or the matrix multiplication application (mult 10 10) in Figure 6(d) profit even more from the new pipeline parallelization approach.

## 7.1 Genetic Algorithm Statistics

Due to limited space it is not possible to give results for all evaluated benchmarks in a graphical representation. Therefore, we sum up the results and statistics of the genetic algorithm of the other evaluated benchmarks in Table 1. The columns contain information about the time in minutes[1] which was necessary to parallelize the applications with the combination of the task-level and pipeline 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*) returned to the designer. The numbers shown in brackets in the last column show how many Pareto-optimal solutions were generated by the task-level- (*TL*) and pipeline parallelization (*PL*) approach as well as the number of solutions generated by combining both approaches (*MI*). The number of individuals and populations used to parallelize a node is determined dynamically, based on the number of child nodes. Thus, nodes with a smaller search space are processed much faster. The numbers of populations, individuals, mutations, etc., shown are summed up over all parallelized nodes and may marginally differ between different tool flow executions due to random decisions taken by genetic algorithms.

As can be seen in the last column of Table 1, most Pareto-optimal solutions are created by the new multi-objective aware pipeline parallelization approach presented in this paper. Nevertheless, some benchmarks also profit from the previously published task-level parallelization approach and also from the combination of both approaches. For example, 25 Pareto-optimal solutions are returned to the application designer for the spectral benchmark, which was used in the motivating example in Section 3. Three of these 25 solutions were generated by the task-level approach, while 7 solutions are purely based on extracted pipeline parallelism. Nevertheless, 15 solutions were generated by the combined approach containing parallel sections with both, task-level and pipeline parallelism. Thus, both approaches and also their combination are adding meaningful solutions which optimize at least one dimension of the Pareto-space. There are also benchmarks which profit either from the task-level parallelization approach, like, e.g., the block decoding of the h264 decoder, or from the pipeline parallelization approach, like, e.g., the latnrm benchmark. However, most benchmarks have shown that they profit from both approaches and their combination.

The table also presents some statistics of the genetic algorithm and the time which was necessary to parallelize the application. For the jpeg2000 application, for example, more than 300,000 individuals were created by mutation and recombination of old solutions. The whole parallelization approach took less than 5 minutes which correlates to less than a microsecond to mutate or recombine and also evaluate one of the individuals. Otherwise it would not be possible to generate a huge amount of solution candidates which would drastically reduce the quality of the solutions generated by the genetic algorithm.

## 7.2 Summary

To summarize, the following results could be confirmed by the evaluation:

1. A huge optimization potential exists if multiple objectives are considered at the same time in the parallelization process.
2. The new multi-objective aware pipeline parallelization approach presented in this paper extracts, in general, the most efficient parallelism regarding speedup of execution time for the considered embedded applications.
3. Solutions generated by the task-level parallelization approach are less efficient regarding speedup for many embedded applications, but they use less energy at the expense of more communication overhead.
4. The combination of task-level and pipeline parallelism produces highly beneficial solutions which find a good trade-off between the high speedup of pipeline parallelism and less energy consumption of task-level parallelism.

Thus, the results have shown that the multi-objective aware extraction of pipeline parallelism highly improves the quality of the solutions returned by the existing framework. In addition, the combination with task-level parallelism also extends the space of Pareto-optimal solutions.

[1]Measured on a system with two AMD Opteron quad-cores running at 2.4GHz

## 8. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this paper presents the first pipeline parallelization approach which uses genetic algorithms to divide loops of embedded applications horizontally and vertically into concurrently executed tasks in a multi-objective aware manner. In addition, we combined this approach with an existing multi-objective aware task-level parallelization technique to benefit from both types of parallelism. The evaluation was performed using real-world embedded applications and it could be shown that each approach and also their combination are able to add Pareto-optimal solution candidates to the final results. Compared to the state-of-the-art, our framework enables the possibility to select the parallel solution which perfectly matches to a specific application scenario instead of just returning the solution with the highest speedup on cost of other resources.

In the future, we would like to extend our framework and the models used to be aware of static or dynamic voltage frequency scaling 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 into our framework. In addition, we would also like to evaluate the influence of other objectives (e.g., code size) and adapt the tool flow to be also aware of heterogeneous architectures.

## 9. REFERENCES


[1] 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.

[2] U. Bondhugula, A. Hartono, J. Ramanujam, et al. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of PLDI*, 2008.

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

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

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *ISCA*, 1992.

[14] C. G. Lee. UTDSP Benchmark Suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, April 2012.

[15] C. Lengauer. Loop Parallelization in the Polytope Model. In *CONCUR '93, Lecture Notes in Computer Science 715*. Springer-Verlag, 1993.

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

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

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

[19] M. Qiu, J.-W. Niu, L. T. Yang, et al. Energy-Aware Loop Parallelism Maximization for Multi-core DSP Architectures. In *Proc. of GreenCom*, 2010.

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

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

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

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

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

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

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

[27] 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.*, 16, 2011.

[28] L. Wehmeyer and P. Marwedel. *Fast, Efficient and Predictable Memory Accesses*. Springer-Verlag New York, Inc., 2006.

[29] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4), 1991.