Interactive Parallelization of Embedded Real-Time Applications Starting from Open-Source Scilab & Xcos

Oliver Oey, Michael Rückauer, Timo Stripf, Jurgen Becker, Clément David, Yann Debray, David Müller, Umut Durak, Emin Koray Kasnakli, Marcus Bednara, et al.

Interactive Parallelization of Embedded Real-Time Applications Starting from Open-Source Scilab & Xcos

Oliver Oey, Michael Rückauer, Timo Stripf, Jürgen Becker
emmtrix Technologies GmbH
Karlsruhe, Germany
{ oey, rueckauer, stripf, juergen.becker }@emmtrix.com

Clément David, Yann Debray
ESI Group
Paris, France
{ clement.david, yann.debray }@esi-group.com

David Müller, Umut Durak
German Aerospace Center (DLR)
Braunschweig, Germany
{ david.mueller, umut.durak }@dlr.de

Emin Koray Kasnakli, Marcus Bednara, Michael Schöberl
Fraunhofer Institute for Integrated Circuits (IIS)
Erlangen, Germany
{koray.kasnakli, marcus.bednara, michael.schoeberl}@iis.fraunhofer.de

***Abstract*—In this paper, we introduce the workflow of interactive parallelization for optimizing embedded real-time applications for multicore architectures. In our approach, the real-time applications are written in the Scilab high-level mathematical & scientific programming language or with a Scilab Xcos block-diagram approach. By using code generation and code parallelization technology combined with an interactive GUI, the end user can map applications to the multicore processor iteratively. The approach is evaluated on two use cases: (1) an image processing application written in Scilab and (2) an avionic system modeled in Xcos. Using the workflow, an end-to-end model-based approach targeting multicore processors is enabled resulting in a significant reduction in development effort and high application speedup. The workflow described in this paper is developed and tested within the EU-funded ARGO project focused on WCET-Aware Parallelization of Model-Based Applications for Heterogeneous Parallel Systems.**

***Keywords*—multi-core processors; embedded systems; automatic parallelization, code generationg, Scilab, Xcos, model-based design**

I. INTRODUCTION

Developing embedded parallel real-time software for multicore processors is a time-consuming and error-prone task. Some of the main reasons are

1. It is hard to predict the performance of a parallel program and therefore hard to determine if real-time timing constraints are met.
2. New potential errors like race conditions and dead locks are introduced. These errors are often hard to reproduce and therefore hard to test for.
3. The parallelization approach of an application is optimized for a specific number of cores resulting in a high porting effort when there is need for changing the number of cores.

In this paper, we want to demonstrate parts of the ARGO approach to show a simple way for the development of applications with real-time constraints. The major benefits of this flow are the more abstract modelling of the application (compared to plain C), making the programming easier, automatic algorithms that handle many error prone tasks automatically and a great flexibility regarding the target platforms.

A. State of the Art Model-Based Design

Model-Based Design refers to the development of embedded systems starting from a high-level mathematical system model. It is a subset of a larger concept called Model-Based System Engineering. Model-based design has seen a rising interest from the industry in the last couple of decades, especially in Aeronautics, Automotive and Process industries, using more and more electronics and software.

The main reason for this trend is the possibility to manage the development process from a higher point of view, making abstraction of the low-level design of systems. This results in the gain of time and costs, but has disadvantages in terms of control on the knowhow. With the rising complexity of the systems integrated in today's and tomorrow's products, this abstraction layer shifts the design challenges to the tools vendors and technology providers, and the real-time requirements needs to be addressed in a collaborative manner on both hardware and software level.

We recently observed a consolidation on the market of tools vendors, in the favour of Product Lifecycle Management players, such as Dassault Systèmes and Siemens PLM (the latter acquiring Mentor Graphics in 2017 for 4.5 billions $). Two specialists in the segment of Simulation and Analysis remain independent and provide the more appealing solutions for Model-Based Design for both Aeronautics and Automotive, namely

Ansys Scade (from the acquisition of Esterel Technologies) and Matlab Simulink.

Scilab Xcos represents an open-source alternative to those dynamic system modelling & simulation solutions (for both time continuous and discrete systems). It is also packaged with domain specific libraries for signal processing and control systems. It bases on the same kernel than Matlab, for matrix computation and linear algebra LAPACK& BLAS [1]. Xcos provides a graphical block diagram editor in order to model systems. The blocks contain functional description of the components of the system, in the form of transfer functions, and the blue links between the blocks convey signal data at every step of the clock synchronizing the simulation. Time synchronization is propagated to the blocks requiring this information in their behaviour, by red links from the clock (special block). The particularity of Xcos in comparison with Simulink is its asynchronous behaviour. Indeed it is possible in Xcos to represent different time sampling clocks to represent asynchronism of embedded systems.

B. State of the Art Parallel Programming with real-time constraints

In practice, the real-time embedded implementations for imaging applications are achieved in the following way: Starting from a high-level model in MATLAB or Scilab, the algorithms are modified for constant runtime. Especially with complex algorithms, data dependent computation is present. These data-dependent processing elements need to be identified and conditional execution needs to be re-written. For example, execution of both branches and mask-based combination of the results must be manually implemented. This code is then ported to an embedded C/C++ code and further optimized for the target platform. The parallelization is carried out manually, by distributing the work on the target architecture. This manual process is time consuming, error-prone and the result is fixed to a single architecture.

Parallelizing applications for embedded systems with real time constraints is a broad topic with several different approaches. The parMERASA project uses well-analyzable parallel design patterns [2] to parallelize industrial applications [3]. The patterns cover different kinds of parallelism (e.g. pipeline, task or data) as well as synchronization idioms like ticket locks or barriers. In doing so, existing legacy code can be parallelized and executed on timing-predictable hardware with real time constraints. Using these well-known parallel design patterns eases the calculation of the worst-case execution time (WCET).

The work of [4] proposes compiler transformations to partition the original program into several time-predictable intervals. Each such interval is further partitioned into memory phase (where memory blocks are prefetched into cache) and execution phase (where the task does not suffer any last-level cache miss and it does not generate any traffic to the shared bus). As a result, any bus transaction scheduled during the execution phases of all other tasks, does not suffer any additional delay due to the bus contention.

The work of [5] attempts to generate a bus schedule to improve both the average-case execution time (ACET) and the worst-case execution time (WCET) of an application. This technique improves the ACET while keeping its WCET as small as possible

Other approaches define extensions for programming languages in order to describe different kinds of parallelism within the program. In [6] an OpenMP inspired infrastructure is introduced that allows annotating parallelism in the source code in order to automatically extract data dependencies and insert synchronization.

In this paper, we introduce a semi-automatic, interactive parallelization approach for applications written in an abstract programming language or model. It covers a subset of the ARGO toolchain and although lacking complete WCET analysis for the sequential and parallel program, transformations that optimize the WCET and WCET aware scheduling, can already be used for applications with real time requirements.

II. Application Use Cases

A. Polarization Image Processing

This application is a specialized image processing system for image data originating from a novel polarization image sensor (POLKA) developed at Fraunhofer IIS [7]. This camera is used in industrial inspection, for example in in-line glass [8] and carbon fiber [9] quality monitoring. Polarization image data is significantly different from 'traditional' (i.e. color) image data and requires widely different – and significantly more computation intensive - processing operations as shown in Fig. 1.

A gain/offset correction is performed on each pixel to equalize sensitivity and linearity inhomogeneity. For this purpose, additional calibration data is required (G/O input in Fig. 1).

Since each pixel only provides a part of the polarization information of the incoming photons, the unavailable information is interpolated from the surrounding pixels (similar to Bayer

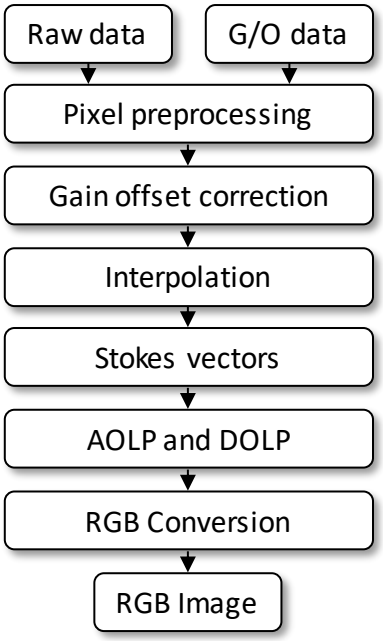

Fig. 1 Exemplary polarization image processing pipeline

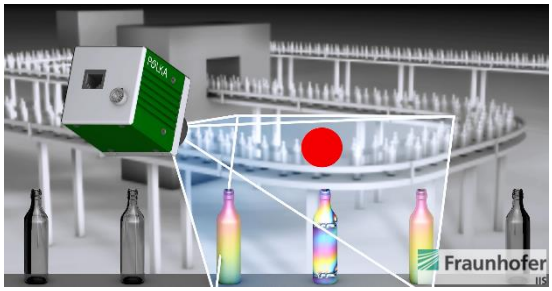

Fig. 2 Inline glass inspection with PolKa

pattern interpolation on color image sensors). From the interpolated pixel values we now compute the Stokes vector, which is a vector that provides the complete polarization information of each pixel. By appropriate transformations, the Stokes vectors are converted into the degree of linear polarization (DOLP) and angle of linear polarization (AOLP). These parameters are usually the starting point for any further application dependent processing (which is not shown here). For demonstration purposes, we convert AOLP and DOLP into a RGB color image that can be used for visualizing polarization effect.

Polarization image processing is currently used in industrial inspection. For example, inline glass inspection is depicted in Fig. 2 and Fig. 3. Glass products are transported at up to 10 items per second and images are captured. Typically, a single inspection PC will handle multiple cameras and requires at least 20 fps processing capabilities. Currently, for one camera, this rate can be achieved, but in case of multiple camera outputs processed by one PC or in case of different use-cases where the number of output measurement frames increases, it can drop to 6-10 fps. This obligates for each use-case to reconsider/investigate further optimization possibilities. Our aim is to achieve a minimum of 25 fps as a hard constraint independent of use-case and processing elements in the algorithm chain. This is a hard constraint knowing that without any optimizations and parallelization, we can only achieve around 6 fps.

Fig. 2 shows the POLKA Polarization Camera with glass measurements performed in a single shot per item. Since this is a measurement device, the precision of the measured data is of uttermost importance. Therefore, the standard algorithm is further adapted for each sensor and polarization data is further processed for different use cases. Especially trigonometric computation leads to a large computation overhead.

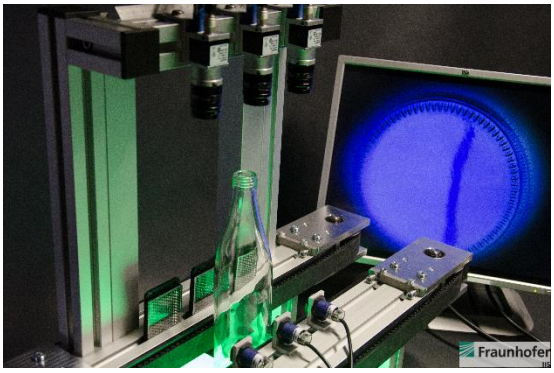

Fig. 3 Inline glass inspection with COTS cameras

An alternative based on a number of COTS cameras is shown in Fig. 3. This system complements the POLKA capabilities with increased spatial resolution and lower system cost. This construction, however, requires additional image fusion. The required registration and alignment further increase the computational complexity of the measurement operation [10].

In both cases, their underlying algorithms need to be adapted to each use case, starting from the Scilab high-level algorithmic description, all the way down to the embedded C / VHDL implementation.

B. Enhanced Ground Proximity Warning System

An Enhanced Ground Proximity Warning System (EGPWS) is one of various Terrain Awareness and Warning Systems (TAWS) and defines a set of features, which aim to prevent Controlled Flight Into Terrain (CFIT). This type of accident was responsible for many fatalities in civil aviation until the FAA made it mandatory for all turbine-powered passenger aircraft registered in the U.S. to have TAWS equipment installed [11]. There are various TAWS options available in the market for various platforms in various configurations. The core feature set of an EGPWS is to create visual and aural warnings between 30 ft to 2450 ft Above Ground Level (AGL) in order to avoid controlled flight into the terrain. These warnings are categorized in 5 modes:

1. Excessive Descent Rate: warnings for excessive descent rates for all phases of flight.

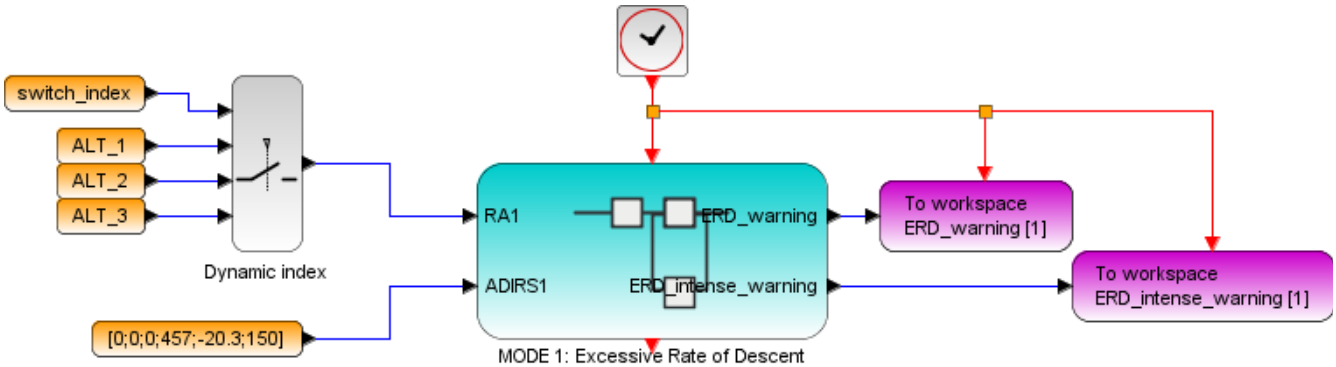

Fig. 4 Reduced ARGO EGPWS Scilab Xcos block diagram

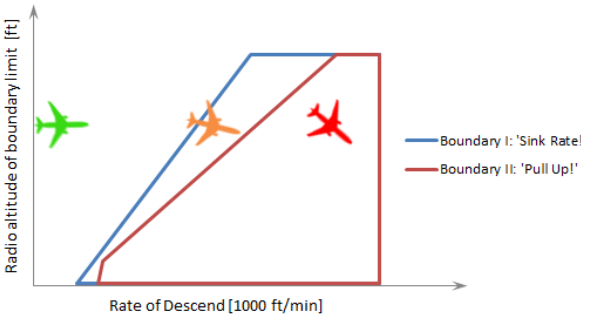

Fig. 5 Graph depicting the foundation for the implementation of Mode 1, Excessive Descent Rate

2. Excessive Terrain Closure Rate: warnings to protect the aircraft from impacting the ground when terrain is rising rapidly with respect to the aircraft.
3. Altitude Loss After Take-off: warnings when a significant altitude loss is detected after take-off or during a low altitude go around.
4. Unsafe Terrain Clearance: warnings when there is no sufficient terrain clearance regarding the phase of the flight, aircraft configuration and speed.
5. Excessive Deviation Below Glideslope: warnings when the aircraft descends below the glideslope.

Additionally, an EGPWS provides some enhanced functions, like the Terrain Awareness Display and Terrain Look Ahead Alerting based on a terrain database.

Fig. 4 shows a reduced Scilab Xcos model as it was used for debugging during the development of the ARGO EGPWS, in this case for the Mode 1 block. Fig. 5 gives an understanding of the corresponding algorithm. The three aircraft have the same altitude of about 2000 ft, but different Rates of Descent, which is demonstrated by their position in the graph. While the green aircraft is in a safe flight state, the orange one's Rate of Descent causes a warning. The red aircraft, however, is sinking much too fast considering its low altitude, requiring immediate action by the pilot.

Most important among the Terrain Awareness features is the Terrain Awareness Display. It is not a separate device, but an enhancement to the Navigation Display (ND) that is already existent in a conventional airliner cockpit. As a background to the displayed information, an abstracted image of the terrain ahead can be turned on by the push of one button. The range of the ND can be as little as 10 nm and as much as 160 nm (18.5 km or 296 km, respectively), which then also applies to the radius of the semicircular terrain image.
The first step for the terrain visualization is the extraction of an area of interest from the database, based on the position and orientation of the aircraft. The range set on the ND is also important, as it determines the size of the AOI. Given the level of detail of the database, which is just above 90m between data points, the AOI's size can range between 200 by 400 and 6400 by 12800 points. The elevation data of each point in the AOI is compared to the aircraft's altitude to create a color map, which has to be converted to an image with a much lower resolution in order to be displayed on the ND. The conversion yields the highest elevation point in a given part of the AOI to make sure that no critical elevation information is lost.

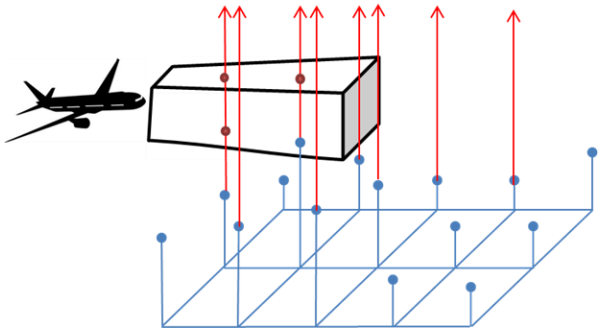

Fig. 6: Collision detection based on comparison of terrain database with a box shaped flight path prediction

Another feature is the Terrain Look Ahead Alerting. A virtual box predicting various possible flight paths for the next 60 seconds flies ahead of the aircraft. By checking the box for collisions with the covered terrain points in the AOI, the system is able to alert the pilot early enough before a terrain collision will occur. The principle is shown in Fig. 6.

III. Interactive Parallelization Workflow

The interactive parallelization workflow as shown in Fig. 7 is designed to assist the user with the parallelization process

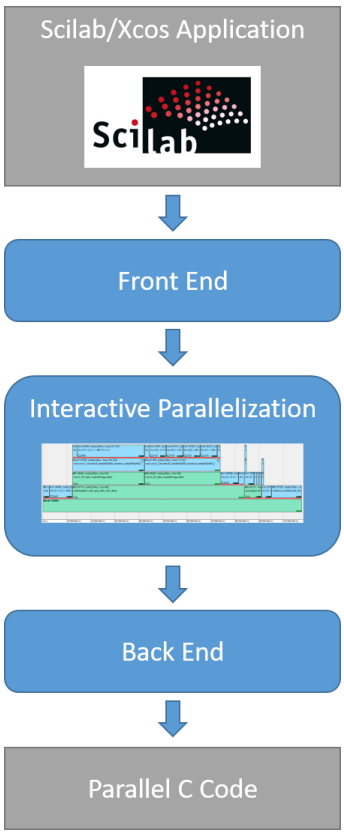

Fig. 7: Overview of the interactive parallelization flow

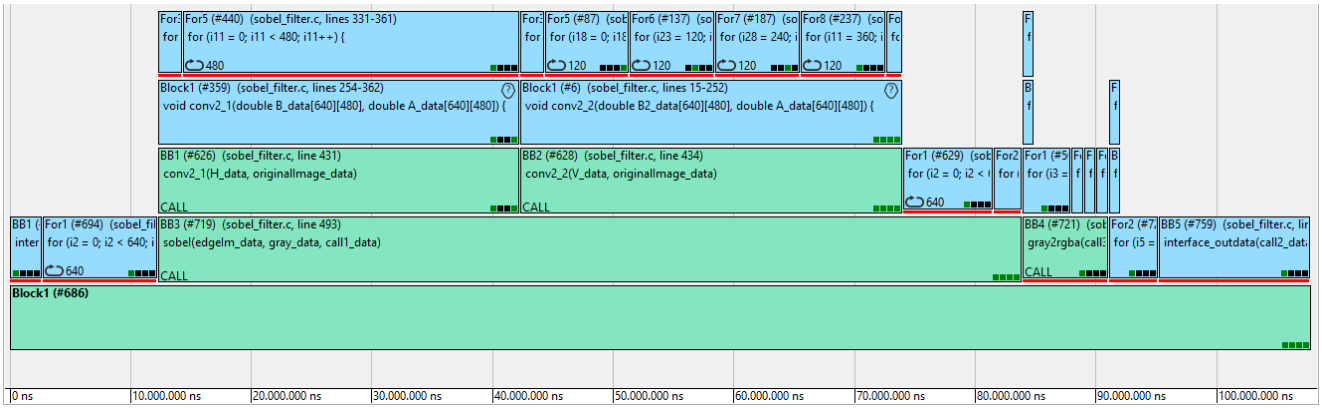

Fig. 8 Hierarchical representation of a sample program

through abstraction and automation. Algorithm development can be performed using abstract, mathematical programming languages like Scilab or MATLAB or their respective model-based extensions Xcos or Simulink. This allows focusing on the functionality while timing and hardware-specific optimizations will be handled later in the tool chain.

A. Front end

The front-end tools parse Scilab and Xcos files in order to transform them into a functionally equivalent sequential C code representation. Constraints from the end user are taken into account for front end transformations and potential additional information from the Scilab source code is preserved as pragma-based source code annotations. The generated C representation uses a subset of the C99 standard excluding constructs like function pointers and pointer arithmetic, which can dramatically reduce the compile time predictability.

The Xcos to Scilab code generation is a Scilab toolbox reusing Xcos model transformation. It takes an Xcos diagram, a sub-system name and a configuration Scilab script as input and outputs Scilab code for both the scheduling and block implementations for the selected sub-system. The generated Scilab code is later used as an input to generate C code using the Scilab to C frontend.

The Scilab to C code generation generates efficient, comprehensible and compact embedded C code from Scilab code. It supports a wide range of the Scilab language features and extensions as well as embedded processor architectures. Developers can easily integrate the C code into existing projects for embedded systems or test it as standalone application on the PC as the code has not yet any optimizations for any specific target platforms.

The C code generator can analyze the worst-case execution count of each block of the generated C code. The analysis uses value range information from sparse condition constant propagation (SCC). The value range information contains the maximum values of variables that effect e.g. the maximum or worst-case execution count of for loops. If no worst-case information can be derived automatically, special functions can be used to manually specify worst-case information within the Scilab code. The result of the analysis is generated as pragmas into the generated C code. Furthermore, all data accesses are taken into account in order to generate code with static memory allocation.

B. Parallelization

The parallelization tool generates statically scheduled parallel C code for a specific target platform. A user can control the process through a graphical representation of the program as can be seen in Fig. 8. The width of the blocks represents the duration of the sequential program as calculated using a performance model of the hardware platform. Hierarchies on the Y-axis show different control structures like function calls, loops or if blocks. We use the term task to describe a unit of work. During the later code generation, tasks will be clustered for the individual cores and depending on the configuration or the targets operating system form threads or processes.

A user can interact with the parallelization process in several ways:

- Assigning core constraints to tasks in order to enforce or forbid the execution of a task on a specific core.
- Setting cluster constraints in order to limit the granularity on which the automatic parallelization algorithm works.
- Applying code transformations to specific code blocks of the program. More details about this concept are described later in this section.

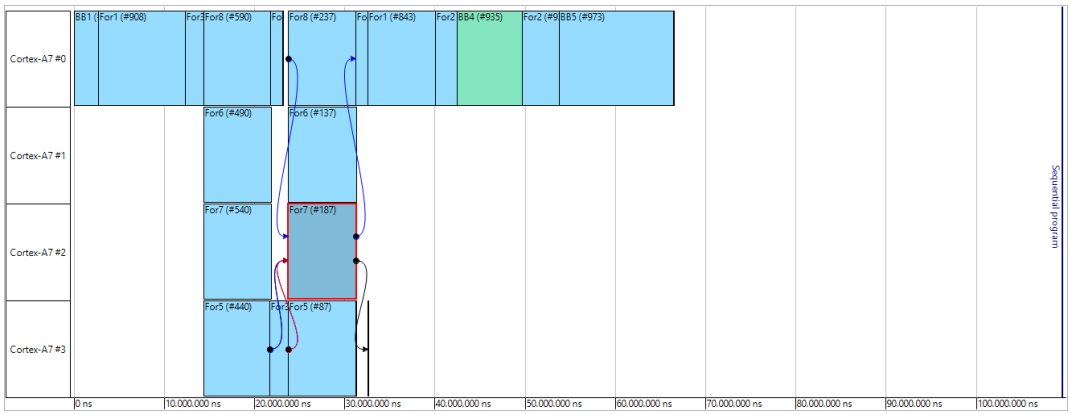

Fig. 9 Example for a scheduling view

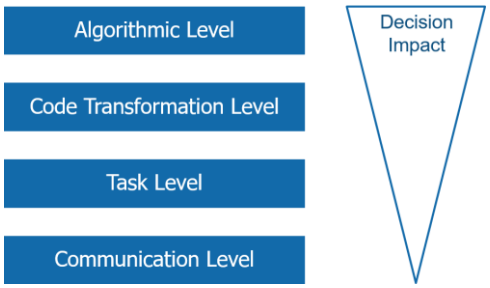

Fig. 10 Parallelization levels

As basis for the graphical representation and for the automatic scheduling the well-known hierarchical task graphs (HTG) [12] are used. Their main concept is hiding complexity caused by cyclic dependencies through the introduction of hierarchies. For each loop, a new hierarchy level is created and the loop is placed inside. Task dependencies can only connect tasks on the same hierarchy level. By introducing these new hierarchies for loops, cycles on the same level are avoided. This representation eases the analyzability of the whole program and enables more accurate predictions of the performance, which are necessary to meet the real time constraints.

We handled the scheduling with a modified version of the established Heterogeneous Earliest Finish Time (HEFT) algorithm [13]. It prioritizes the execution of tasks with a high rank, which is defined by its computing cost, its number of succeeding tasks and the overall communication costs for the necessary variables. Being a greedy algorithm, it can fail to find the optimal solution but has the advantage of a fast execution time. This is key for the interactivity with the user. The modified HEFT algorithm is able to handle hierarchical structures and to take into account core and cluster constraints assigned by the user. An example of a resulting mapping and schedule can be seen in Fig. 9. For each core of the target platform, the mapping of tasks over the time is shown. Arrows represent data and control dependencies to guide the user with the parallelization process. As a reference, the sequential execution time of the program is shown on the right hand side of the figure. All user interaction described with the HTG view is also applicable for the scheduling view. By generating a static schedule of the whole program, our flow does not rely on the scheduler of the operating system.

The performance estimation used for the parallelization is based on the worst-case execution count as determined by the front-end tools. The data is acquired by a combination of static analysis of the source code and profiled execution on the host platform. In doing so, the number of iterations for each loop can be determined. Additionally, a performance model of the execution times of instructions on the platform is used to perform a static analysis of the complete sequential program in order to determine the runtime of the program on the target platform. The execution times of instructions were directly measured on the target platforms. These measurements also take into account different types of cores and memory configurations.

In compiler design, a code transformation is typically applied to the whole program. In the context of ARGO and parallelization, this behavior is problematic. A transformation exposing coarse grain parallelism makes only sense on code regions that require more coarse grain parallelism. On all other locations, it would have negative effect i.e. performance overhead, larger code size or memory footprint or the incompatibility to other optimization transformations. An example of this is splitting a for-loop into several independent for-loops. The potential for parallelism is increased as these new loops can be executed in parallel. However, this usually comes at the cost of additional temporary or duplicated variables, which have a negative impact on the performance and/or the memory footprint of the application when all loops are executed sequentially. Therefore, we need a concept for selectively applying code transformations to code regions only where it makes sense from the global schedule point of view. Thereby, we must solve the phase ordering problem since the code transformations are applied before scheduling and mapping. Furthermore, on a specific code region or code position the order of applying code transformation must be controllable.

We solve the problem by using a code transformation framework that applies all potential transformation in a single pass. In a top down approach, the pass visits each task where first all potential transformations are analyzed for applicability. E.g. a simple loop unrolling transformation can only be applied to “for” loop blocks matching a specific init, step and condition template. If a transformation candidate is found, the task is marked to have a potential transformation, which can then be set in the HTG view. In parallel, it is checked if a decision value is set for the transformation in the GUI from a previous iteration. Based on the value the corresponding transformation is applied to the task. Afterwards, all children of the block are visited. This approach opens up a large design space where several transformations can be applied to different tasks of the program. In this first iteration of the flow, the user can dynamically select transformations for tasks and will get feedback about the performance impact through the scheduling view. All available transformations like loop splitting, tiling, fission or unrolling preserve the predictability of the program as the new execution times can be calculated from the existing data.

Parallelization can be categorized into different levels like shown in Fig. 10. We already covered the code transformation level and the task level as well as their impact on the predictability of the program. Above these two, there is the algorithmic level. Many problems can be solved by different algorithms which may have different performance, memory requirements or can be parallelized differently. A common example is the Fast Fourier Transformation (FFT) where a 1024-point FFT can also be calculated with two 512-point FFTs. Both can be calculated in parallel and we therefore have a different behavior regarding the parallelization. Within our tool flow, the user can make the choice of the algorithm in the interactive view. However, the selection presented to the user is already made by the front end, which recognizes functions/algorithms with different implementations, and provides them to the interactive GUI. When the user selects a different implementation, the flow starts from the beginning with the new selection, thus recalculating all necessary performance information.

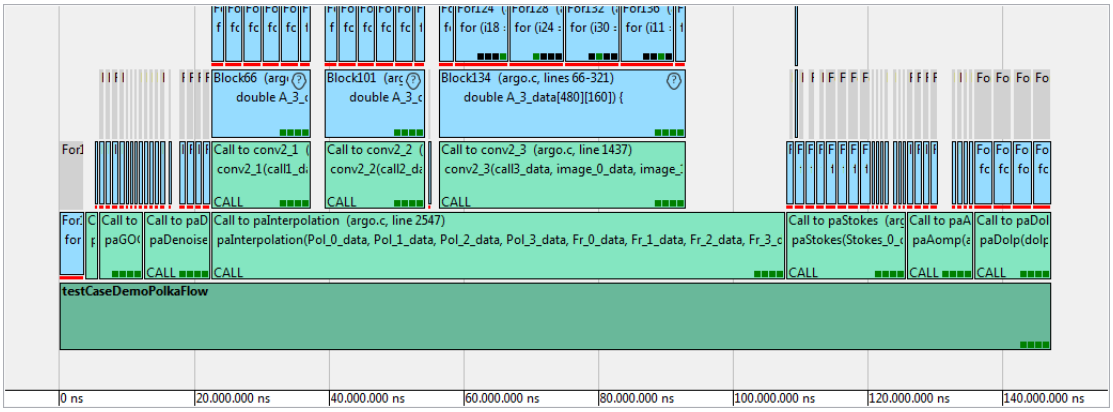

Fig. 11 Graphical representation of polarization imaging

Changes in the communication level of the application are taken into account in the back end of the flow. During the scheduling, only a rough estimation of the chosen communication model is used for the performance prediction.

With all these different parallelization methods, the user is able to iteratively optimize the performance of the application.

C. Back end

The back end of the tool flow covers the communication/synchronization and the generation of parallel C code. Currently, we use a distributed memory model for all target platforms. This means, that each core that needs access to a variable has its own copy of it. Data dependencies are analyzed using a static single assignment representation [14] so that all edges, which have different cores for the definition and usage of a variable, can be used to insert communication in the program. As both cores have their own version of the variable, this explicit communication is necessary and has the benefit of avoiding access to the same memory areas from different cores. This greatly enhances the predictability of the resulting parallel program. The timing estimation of the communication overhead is closely coupled with the target platform and its capabilities. Important factors are the operating system, how the data is transferred and whether the system load affects the timing or not. Two different access patterns can be differentiated:

- Multiple cores read the same data: in this case, one core is the owner of the data either by calculating it or by acquiring it from an external interface. The core will then send the data to all other cores that need access to it. The order of the communication is determined by the static schedule to minimize the waiting times of the receiving cores. When the data is initialized at the beginning of the program, this procedure is performed on all cores and further communication is not necessary.
- Multiple cores modify the same data: as each core has its own copy of a variable, the modifications do not directly affect each other. When data needs to be modified in a specific order, the values are synchronized between the corresponding cores before the modifications. In the case that the variable is an array or a matrix of values, it is split into several independent variables and joined back into one after the processing.

To improve the predictability of parallel programs, which contain control structures like loops that are partially executed by multiple cores, the back end duplicates the control flow on all involved cores. This means rebuilding control structures like loops or if-blocks as well as statements like break or continue. When necessary, each iteration of the loop is synchronized by evaluating the condition on one core and sending the result to the other cores. In doing so, each core has the same amount of iterations which eases predicting the performance of the loop.

The generated C code is compiled into a single binary that is executed on each core.

IV. Parallelization of Applications

A. Polarization Imaging

The algorithm is fairly simple, however computationally cumbersome in parts such as 2D convolutions and intensity mappings in demosaicing. Nevertheless, the uniformness of the computation over the data array permits a high potential for data parallelization.

The graphical representation of the parallelization is shown in Fig. 11. As can be seen in the hierarchical view, the main processing is a chain of several consecutive steps. Most of them can be parallelized using loop transformations on the task parallelization layer. The resulting schedule can be seen in Fig. 12. All four cores of the target platform are occupied through most of the program resulting in a speedup of up to around 3 compared to the sequential execution.

In order to achieve such a tight schedule on core mapping, different tiles of the input image should be able to be processed independently of each other. The toolchain allows us to exploit this, without changing the original Scilab code in overall, but by adding the necessary functions, provided by the front end in user-defined positions in the algorithm. Thus, the end-user

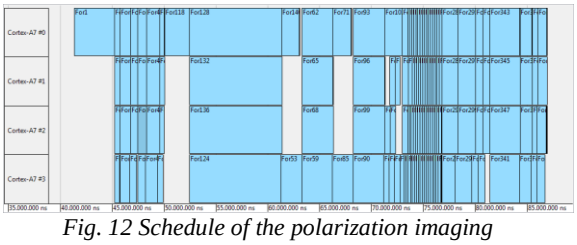

Fig. 12 Schedule of the polarization imaging

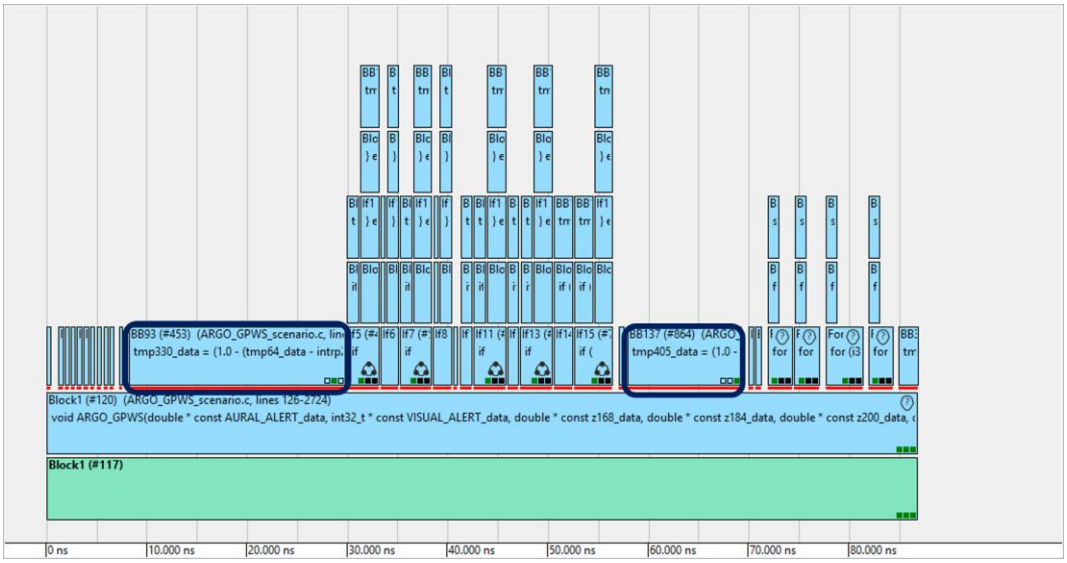

Fig. 13 Hierarchical view of the EGPWS

gains access through the Scilab code to scheduling and mapping in hardware level for any desired parts of the code without requiring the knowledge of underlying specifics for parallelization.

In our processing pipeline, we divide the image data after the step Pixel Preprocessing in Fig. 1 into 4 tiles with a variable splitting transformation, each mapped to one core and use loop splitting to run the tiles in separated loops for each tile. Interpolation step introduces 4 additional image arrays each of size of the input image array. These are divided into 4 tiles again and mapped to corresponding cores. The Stokes Calculation step is a linear combination of its predecessor step and reduces the array number to 3. These arrays are again mapped to 4 cores and so on and so forth, until output arrays AOMP and DOLP are calculated. Thus, there is a high degree of locality of the data, since the same tile stays on the same core among processing steps.

Once a hierarchical view of the algorithm is built, different constellations of computation blocks on cores can be played with for achieving speed-up from within the available GUI. By clicking on the desired computation block and updating the scheduling view, the corresponding scheduled timing for this block can be zoomed in and the tasks on other processing cores can be observed for that time instance continuously.

For example, it is important to control the 2D convolution part of our algorithm, such that it is distributed on (4) cores in a tight schedule. This can only be achieved by trial and error from simulating for the previously mentioned different constellations. Fortunately, the GUI of the toolchain provides also an assessment about the expected speed-up by showing the estimate of the runtime-ratio between the sequential and parallel program on the right side, to get a feeling what is worth to run to test for performance on the target hardware.

B. Enhanced Ground Proximity Warning System (EGPWS)

The EGPWS algorithm is fundamentally different from the image processing algorithm from the first test case. It is implemented in Xcos and by the dataflow-oriented description, one would assume a lot of task-level parallelism.

However, the performance estimation of the algorithm outlined that the Mode 1 to 5 calculations are not computational intensive. The hot spots are the Terrain Awareness Display and Terrain Look Ahead Alerting calculation requiring many checks to the terrain database. Both hotspots are implemented as a Scilab function. To gain sufficient task parallelism, we further parallelized the hotspots by loop transformations and partition of variables. This results in the hierarchical view of the application as shown in Fig. 13 where one can see a broad structure of the whole program. The transformations created many small tasks with two blocks that stand out because of their length. One of them is located in Mode 2 and the other in Mode 4. Each of them does a bilinear interpolation, which means that for a given x- and y-value an appropriate z-value is calculated. For this, they use many mathematical operations with double precision, which takes more time than most of the other tasks in the application, like the linear interpolations that are done in Mode 1 and 5. As the two interpolations do not have a direct dependency between them, they can be assigned to two different cores. All other shorter tasks will then be scheduled automatically to reduce the amount of communication between

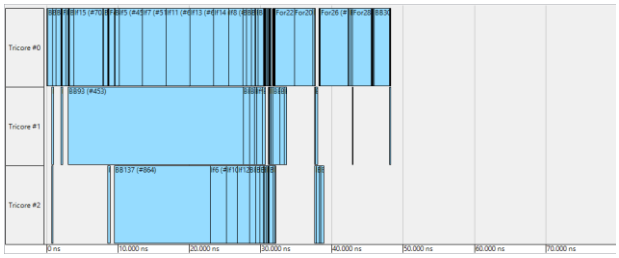

Fig. 14 Schedule of the EGPWS

them. Fig. 14 shows an example schedule for the EGPWS on the Aurix Tricore TC297 processor with three cores. The speedup of the whole application is around 1.8.

Even though the ARGO EGPWS is intended for use only within DLR's Air Vehicle Simulator (AVES), this use case can in addition serve to examine the ARGO toolchain with respect to software and software tool development regulations like DO-178C [15] and its supplements DO-330 [16] and DO-331 [17]. This, however, is currently out of scope and regarded as a future work.

V. Evaluation

For evaluation of the performance and correctness of the parallel application, we used as a first step a Raspberry Pi 2 running a linux operating system equipped with a 4-core ARM Cortex-A7 as target architecture. We used locking-free queues to apply a message-based communication between the four processor cores. In the next step, we used an Aurix Tricore TC297 running a FreeRTOS operating system. Similar locking free queues were used for the message-based communication between the three cores. These two platforms are not the target architectures intended to run the applications but are taken as examples to evaluate whether the workflow is feasible or not.

During evaluation, we could show three major results:

1. Using the interactive parallelization workflow, we could increase the performance of the applications from two different domains between the factors 1.8 (on the Aurix Tricore) and 3 (on the Raspberry Pi 2) compared to the sequential execution. The three respective four cores of the target platform could all be utilized to improve the performance of the application.
2. We could enable an iterative parallelization approach for optimizing the application. Going from Scilab or Xcos to a parallel application running on the target architectures, was possible in less than one minute. In each iteration, the end users could explore an alternative parallelization or further optimize the input code.
3. The implementation effort for parallelizing applications was reduced by over 50% compared to a manual parallelization effort. Furthermore, the workflow enables a model-based development approach for multicore processor that was not possible before. Analyses from literature show that a model-based development approach can reduce the overall development effort by 50% for single-core processor. This results in reduction of the overall effort by 60-80% for multi-core processors. By using two different target architectures, we could also show that using this approach allows fast switching of the platforms as many decisions that were made for the parallelization can easily be ported to other platforms. Oftentimes, a simple change of a parameter, e.g. splitting a loop into 3 instead of 4 independent loops, is enough, to get a better performance for another platform.

VI. Conclusion

In this paper, we have shown that applications with real time constraints, which are modelled in open source programs like Scilab and Xcos, can be used as a basis for a tool flow for the generation of efficient parallel C code for embedded target platforms. By starting at a higher abstraction level, the intermediate sequential C code can be generated to be optimized for automatic analysis passes. It also allows the option to select different implementations of algorithms depending on the target platform.

The iterative flow enables fast and simple optimizations of the program until the desired timing is achieved. This can directly be verified by testing on the actual hardware. Switching to a different platform can also be done without changes to the original model or source code and is mostly handled by the framework.

During the ARGO project, the concepts shown in this paper will be enhanced by an integration of WCET into the flow. They are computed for the sequential C code and are directly taken into account by the scheduling algorithms. Additional transformations allow further optimizations of the WCET of the program.

ARGO (http://www.argo-project.eu/) is funded by the European Commission under Horizon 2020 Research and Innovation Action, Grant Agreement Number 688131.

References

[1] Univ. of Tennessee; Univ. of California, Berkeley; Univ. of Colorado Denver; and NAG Ltd., LAPACK — Linear Algebra PACKage, http://www.netlib.org/lapack, 2017.

[2] M. Gerde, R. Jahr and T. Ungerer, "parMERASA Pattern Catalogue: Timing Predictable Parallel Design Patterns," Augsburg, 2013.

[3] T. Ungerer, C. Bradatsch and M. Frieb, "Parallelizing Industrial Hard Real-Time Applications for the parMERASA Multicore," *ACM Transactions on Embedded Computing Systems (TECS)*, July 2016.

[4] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *In IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.

[5] J. Rosén, C. Neikter, P. Eles, Z. Peng, P. Burgio and L. Benini, "Bus access design for combined worst and aver-age case execution time optimization of predictable real-time applications on multiprocessor systems-onchip," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.

[6] O. Neugebauer, M. Engel and P. Marwedel, "A parallelization approach for resource-restricted embedded heterogeneous MPSoCs inspired by OpenMP," *Journal of Systems and Software*, pp. 439-448, March 2017.

[7] S. Junger, W. Tschekalinkij, N. Verwaal and N. Weber, "On-chip nanostructures for polarization imaging and multispectral sensing using dedicated layers of modified CMOS processes," in *Proceedings*

of the SPIE conference on Photonic and Phononic Properties of Engineered Nanostructures, 2011.

[8] A. Nowak, "In-line residual stress measurement," *Glass Worldwide*, pp. 72-73, 2015.

[9] M. Schöberl, K. Kasnakli and A. Nowak, "Measuring Strand Orientation in Carbon Fiber Reinforced Plastics (CFRP) with Polarization," in *19th World Conference on Non-Destructive Testing*, Munich, 2016.

[10] T. Richter, C. Saloman, N. Genser, K. Kasnakli, J. Seiler, A. Nowak and A. S. M. Kaup, "Sequential Polarization Imaging Using Multi-View Fusion," in *IEEE International Workshop on Signal Processing Systems*, Lorient, France, 2017.

[11] Federal Aviation Administration, *Advisory Circular 23-18, Installation of Terrain Awareness and Warning Systems (TAWS) Approved for Part 23 Airplanes*, Washington, D.C.: U.S. G.P.O., 2000.

[12] M. Girkar and C. D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Transactions on Parallel and Distributed Systems*, pp. 166-178, 1992.

[13] H. Topcuoglu, S. Hariri and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, pp. 260-274, 2002.

[14] B. Rosen, M. N. Wegman and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLANT-SIGACT symposium on Principles of programming languages*, San Diego, CA, USA, 1988.

[15] RTCA, Inc., „DO-178C "Software Considerations in Airborne Systems and Equipment Certification",“ 2012.

[16] RTCA, Inc., „DO-330 "Software Tool Qualification Considerations",“ 2012.

[17] RTCA, Inc., „DO-331 "Model-Based Development and Verification Supplement to DO-178C and DO-278A",“ 2012.