Trace-based KPN Composability Analysis for Mapping Simultaneous Applications to MPSoC Platforms

Jeronimo Castrillon*, Ricardo Velasquez†, Anastasia Stulova*, Weihua Sheng*, Jianjiang Ceng*, Rainer Leupers*, Gerd Ascheid*, Heinrich Meyr*

*Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany
†ALaRI Institute, Lugano, Switzerland
maps@iss.rwth-aachen.de

Abstract—Nowadays, most embedded devices need to support multiple applications running concurrently. In contrast to desktop computing, very often the set of applications is known at design time and the designer needs to assure that critical applications meet their constraints in every possible use-case. In order to do this, all possible use-cases, i.e. subset of applications running simultaneously, have to be verified thoroughly. An approach to reduce the verification effort, is to perform composability analysis which has been studied for sets of applications modeled as Synchronous Dataflow Graphs. In this paper we introduce a framework that supports a more general parallel programming model based on the Kahn Process Networks Model of Computation and integrates a complete MPSoC programming environment that includes: compiler-centric analysis, performance estimation, simulation as well as mapping and scheduling of multiple applications. In our solution, composability analysis is performed on parallel traces obtained by instrumenting the application code. A case study performed on three typical embedded applications, JPEG, GSM and MPEG-2, proved the applicability of our approach.

I. INTRODUCTION

Embedded Multi-Processor Systems on Chip (MPSoCs) are here to stay. They have the potential to provide high performance with low energy consumption, and the flexibility to support different applications and products by using programmable cores. With all these, MPSoCs are well equipped to address an embedded community populated with battery-powered devices and characterized by stringent time-to-market constraints. However, leveraging this potential is a complex task that can only be accomplished with tool support.

Interactive tool support for parallelizing applications written in a sequential way was presented in [1]. Due to the amount of existing legacy code and the sequential programming heritage, such support is badly needed. Nonetheless, in order to overcome the difficulty of (semi-)automatic parallelism extraction, parallel programming models need to be supported as well [2]. Moreover, with embedded devices becoming more and more complex, an MPSoC programming framework has to be able to deal with multiple applications, taking into account different application classes, e.g. real-time and non real-time. In this paper we present a flow built on top of the compiler-centric framework presented in [1] that supports multiple applications described using a parallel dataflow programming model [3], [4].

We argue that current research on tools for single and multiple parallel applications lacks pragmatism. On the one hand, new parallel programming models are being proposed that extend the expressiveness of simple Models of Computation (MoCs), like Synchronous Dataflow (SDF) Graphs [5], without losing some of their formal properties [6]. These models are however intricate and can only be handled by a handful of experienced programmers. On the other hand, most research on multiple applications still focuses on the SDF MoC [7], [8], [9], [10], [11], [12], notwithstanding its restrictions. Our parallel specification is based on the Kahn Process Networks (KPN) MoC [13], which is more expressive and easier to use, although more difficult to analyze. To compensate for that, we provide an extensible framework to analyze KPN applications and derive schedules for them based on traces. These traces are also used to perform composability analysis [11] in reasonable time. We provide a compact way of describing use-cases in form of an Applications Concurrency Graph (ACG).

The framework does not assume full knowledge of the execution time of processes, but includes a Sequential Time Estimation (STE) module. We generate retargetable code by using an abstract Application Programming Interface (API) which is compatible with the HVP [14] simulator. This allows for fast functional validation through simulation.

To our knowledge, this is the first framework to handle multiple KPN applications and to provide fine-grained composability analysis through traces. The rest of this paper is organized as follows. Background knowledge and related work are presented in Sections II and III. The overall tool flow is described in Section IV and details on how the traces are used to analyze multiple applications are given in Section V. Section VI presents a case-study using JPEG, GSM and MPEG-2 and conclusions are given in Section VII.

II. BACKGROUND

A. The MAPS Framework

Our framework is integrated into the MPSoC Application Programming Studio (MAPS) [1], which consists of a set of
tools for MPSoC programming with a graphical user interface (GUI) built upon the Eclipse environment [15]. In [1] we describe the tools that help a programmer to find parallelism in sequential applications. A key component of this flow is an analysis phase where the data and control flows are computed statically and dynamically. Dynamic data flow analysis is enabled through a profiling/tracing mechanism built on [16] that provides valuable information for the subsequent code partitioning and parallelism extraction phase. In this paper we use the analysis facilities of the MAPS framework and apply them to support explicit parallel programming. Although not in the focus of this paper, sequential analysis can still be performed to extract additional parallelism inside the parallel tasks.

**B. Dataflow Models: KPN and SDF**

A KPN application is represented as a graph $G = (V, E)$. Nodes are called *processes* and represent a certain computation. Edges represent unbounded FIFO channels through which processes communicate by sending data items or *tokens*. Processes can only become blocked by reading from an empty input channel - *blocking reads* semantics. A KPN is *determinate*, i.e. the history of tokens produced on the communication channels does not depend on the scheduling [13]. The behavior of a process can be data dependent and feature arbitrary complex control flow. For this reason, it is not possible to compute a plausible static schedule for a KPN application. Scheduling of KPNs has been studied in [3], [17], [18].

An SDF [5] application is also represented as a graph $G = (V, E, W)$. In this graph, nodes are called *actors* and $W = \{w_1, \ldots, w_{|E|}\} \subset \mathbb{N}^3$ associates 3 integer constants $w_e = (p_e, c_e, d_e)$ to every channel $e = (v_1, v_2) \in E \subset V \times V$. $p_e$ represents the number of tokens produced by every execution of actor $v_1$, $c_e$ the number of tokens consumed in every execution of actor $v_2$ and $d_e$ the number of tokens (called delays) initially present on edge $e$.

SDFs are more amenable to analysis, since it is possible to compute a static schedule and the questions of *termination* and *boundedness* are decidable [3]. This has made SDFs very popular among researchers. However, apart from having a restricted expressiveness, predicting timing behavior of multiple concurrent SDFs is no longer trivial [11] and only pessimistic guarantees can be computed [7]. The research trend for multiple SDFs is to relax guarantees in order to obtain a better platform utilization [11]. Due to these limitations, we built our framework to support the more general KPN MoC. Still, with our dataflow characterization it is possible to identify regular accesses to channels and exploit SDF or hybrid scheduling techniques.

**III. RELATED WORK**

Several frameworks support parallel specifications for single applications in a similar way that we do. HOPES [19] receives a parallel specification of the application in the form of a task graph. Data level parallelism is specified by OpenMP pragmas [20] within tasks. HOPES is retargetable, i.e. it generates code with an abstract API that can be implemented for different platforms. The DOL framework [21] uses analytical performance estimation for KPN applications and employs evolutionary algorithms to compute the mapping. Daedalus [22] is a framework for Polyhedral Process Networks (PPN), PPNS can be derived from a set of restricted sequential programs by using the tool presented in [23] and feature better properties than KPNs. This framework accepts mappings generated by the *Sesame* tool [24] which uses evolutionary algorithms as well. Additionally, Daedalus provides means for performing design space exploration and HW synthesis. SystemCoDesigner [25] is similar to Daedalus, but includes also performance evaluation. This framework allows to implement different MoCs with a library-based approach based on SystemC.

Multiple applications have been mainly analyzed by Bekooij et al. [7], [8], [26] and by Corporaal et al. [9], [11]. The common approach is to include dynamic arbitration at runtime, e.g. Time Division Multiplexing (TDM), Round-Robin With Skipping (RRWS), with resource enforcement. They have analyzed the composability problems of SDFs and developed algorithms to determine, for example, the optimal slot duration of a TDM schedule. These approaches are orthogonal to our approach, since they can be integrated in our extensible framework.

**IV. TOOL FLOW**

The overall tool flow is presented in Figure 1. As inputs, the tool receives an XML description of the architecture, the KPN specification of every application including their constraints and the use-cases described in the so-called *Applications Concurrency Graph (ACG)*.

The architecture model includes a list of Processing Elements (PEs) with their types and a description of the communication architecture. Currently, for each PE type, a cost table is associated that is used for sequential performance estimation. The communication architecture is represented as a set of physical channels among processors, each of which with a function that maps transferred bytes to cycles. The KPN specification can be directly modeled within the GUI of the tool as shown in Figure 1b. The ACG can be also generated within the GUI by dragging and dropping applications. It is a graph $ACG = (V, E)$ in which nodes represent applications, and an edge $e = (a_1, a_2)$ indicates that applications $a_1$ and $a_2$ may run simultaneously.

Figure 1a shows the multi-application flow, which follows the *composibility* approach [11], i.e. individual schedules are computed for each application in isolation and are later composed to cover a use-case. After composition, several configurations are recorded for each use-case into a run-time database. The flow finishes with the code generation phase in which an abstract API for communication, synchronization and scheduling is used. This API was implemented on top of the HVP-API [14].

\(^1\)Also possible through custom extensions to the C language.
The single application flow is shown in Figure 1b. In this flow, the application is translated into an Intermediate Representation (KPN IR) that is suitable for analysis. Thereafter, the application is profiled and several scheduling configurations are generated and stored into scheduler descriptors. Figures 1c,d show the profiling and scheduling flows in more detail.

In the profiling and tracing flow (Figure 1c), a Code Generation Engine (CGE) is available that generates pthread [27] code that can be compiled and run on the host machine. The pthread implementation is instrumented to record the history of tokens on the communication channels. Since the token history does not depend on the scheduling, we let the operating system schedule the threads freely. The CGE also generates a sequential application in which the processes are executed in isolation with the recorded tokens (Isolated Processes in Figure 1c). This sequential application makes it possible to re-use the tracing and analysis mechanisms of the MAPS framework, which generates a single sequential trace and annotated Statement Control Data Flow Graphs (aSCDFG) [1] for the processes. The sequential trace is then processed by the KPNTracer, which finds the exact locations in the aSDFGs where the channels are accessed and interfaces with the Sequential Time Estimator (STE) module to extract the time elapsed between channel accesses for different PE types.

The STE has a simple interface: It receives a portion of IR code and returns an estimate of its execution on a given PE type. In the current implementation, the STE uses the cost tables from the architecture model to perform source level performance estimation. Different implementations of the STE could allow direct annotation (for HW accelerators) or interface with more elaborate performance estimation tools to handle heterogeneity better.

The mapping and scheduling flow is shown in Figure 1d. Similar to compiler back-ends, our framework has two scheduling phases surrounding the mapping phase. In the first scheduling phase, a scheduling configuration is computed without mapping information (e.g. process priorities). In the mapping phase, processes are assigned to PEs. The last phase is a place holder for schedule variations (e.g. load balancing) and to determine parameters that depend on the mapping (e.g. buffer bounds). In order to compute and derive schedules, a Trace Replay Module (TRM) is included in this flow. The TRM can replay a portion of the parallel traces according to a scheduling configuration. This module estimates the application makespan, the throughput, the buffer sizes and the platform utilization.

Through configuration files, the user can constrain the search space of the scheduling and mapping algorithms by, for example, fixing a scheduling policy or the mapping of some processes to PEs. Currently, the tool supports preemptive priority-based, non-preemptive data driven (FIFO), Round-Robin (RR), RRWS and self-timed with static ordering [4]. The outcome of each scheduling trial is recorded into a scheduler descriptor XML file. Apart from keeping the scheduling configuration, the scheduler descriptors store the platform utilization and the buffer sizes obtained from the TRM. Additionally, bottleneck processes [17] are also identified together with an estimate of the required buffer size increment to solve the deadlock. This information can be fed to a run-time deadlock resolution routine.

V. TRACE-BASED SCHEDULING FOR KPNs

The problem of computing a static schedule for KPNs is undecidable [3], hence dynamic scheduling policies are applied. In order to derive plausible scheduling configurations and analyze if two or more applications can be executed simultaneously we use profiling traces.

Given a process $P_x$ with associated $aSCDFG = (V, E, W)$, a trace is a sequence of segments $T = \{S_1, S_2, \ldots\}$, where a segment is a path $S = \{v_0, \ldots, v_{i-1}, v_i, \ldots, v_k\}$, $(v_{i-1}, v_i) \in E$ in which only one channel is accessed (read or write), and the access takes place in $v_k$. Different dataflow models differ mainly in their access pattern to channels, thus an analysis on the traces helps to identify an efficient scheduling approach. An example of a trace is shown in Figure 2. Figure 2a shows an example of an aSCDFG for a process $P_x$ with several input and output channels. A segment of this process trace is shown in Figure 2b. In this trace, read accesses $r(I_i)$ are visualized at the left side of the trace and write accesses $w(O_i)$ to the right. Figure 2b also shows how the sequential trace is analyzed to find out which statements were executed between
Fig. 2. Trace Example. a) A process and its aSCDFG. b) Trace illustration. c) Example of extracting behaviors from traces.

channel accesses. In the example, the segment is composed of the nodes \{2, 4, 6, 2, 5\} with a read access to the input channel \(I_1\) in node 5. In Figure 2c a simple illustration shows how a trace can be used to detect regular behaviors. The trace on the left hand side of the figure displays SDF behavior, i.e. a process that always reads two tokens and writes three tokens. The trace on the right hand side displays in turn two SDF modes.

A. Computing a Single Application Schedule

If the traces of all channels display regular behavior as in the example discussed above, then the tool (or the user after inspection) can decide to compute a static order scheduling. For more general cases, the tool supports randomized search and a heuristic-based approach. In randomized search, the user simply gives a number of trials and the tool reports the best performing configuration. Randomized search supports the scheduling policies described above: priority-based, RR, RRWS and FIFO.

To solve the problem of computing a mapping and a schedule for a KPN application, several authors have proposed to use evolutionary algorithms [21], [25], [24]. We instead, use lightweight heuristics that run in linear time. The heuristic for computing priority-based and FIFO schedules for best-effort applications is based on list-scheduling [28]. The set of traces of all processes are considered as streams of instructions and PEs are treated as functional units. The latency of every segment is retrieved from the STE. At every iteration, the list scheduler computes the \textit{ready set}, selects a segment and \textit{maps} it to PE that provides the earliest finishing time. The ready set computation is greatly simplified by the simple synchronization mechanism of KPNs. As with traditional list scheduling algorithms, the performance highly depends on the way the \textit{select} step chooses a ready segment. We have implemented several heuristics that use the fan-in/out, the relative total execution time in stand-alone mode or the length of the path to the sink process. These heuristics are also used to assign the final priority of a process in the case of priority-based scheduling. Based on the list scheduling pass, a final mapping is computed so that process migrations are avoided. In order to do this, the list schedule is traversed and every process is assigned to the processor that holds the majority of its segments.

For throughput constrained applications, we have implemented a simple algorithm that generates platform subsets and iterates along them performing best-effort scheduling until the constraint is met. A platform with

![Fig. 3. Illustration of the composition function](image)

\[ T \text{ processor types is represented as a set } P = \{PE_1^1, \ldots, PE_N^1, \ldots, PE_1^j, \ldots, PE_N^j\}, \] where \(PE_i^j\) is the \(j\)-th PE of type \(i\) of which a total of \(N_i\) cores are available. The algorithm incrementally computes a list of subsets \(L = \{P'_1, P'_2, \ldots, P'_f\} \subseteq \varphi(P)\), where \(\varphi(P)\) is the power set of \(P\), on which the application is mapped. In order to avoid exponential growth due to the power set computation, the subsets are grown monotonically, i.e. \(|P'_{t-1}| \leq |P'_t|\), which strives at finding the mapping with the lowest resource utilization that meets the constraint [9], [8].

Growing the subsets is trivial for homogeneous MPSoCs but, in order to deal with heterogeneous ones, the subset creation algorithm assigns every PE a probability of being added to the next subset. At a given step, with \(N_c\) representing the number of PEs of any class \(c\) left to be used, a PE of class \(i\) will be added to the next subset with a probability given by \(p_i = \frac{\sum_{j<i} N_j}{N_c}\). By doing this, we ensure that scarce resources are seldom used, and are therefore reserved for other processes that might demand a fixed mapping.

B. KPN Composition

As already mentioned in Section IV, the user describes all possible use-cases via the ACG. In the composition phase, all use-cases, i.e. subgraphs of the ACG, are analyzed and, if possible, a schedule configuration is generated for each one of them. The composition step is aware of three different application classes: hard real-time, soft real-time and best effort. Different applications classes are arbitrated separately by using hierarchical scheduling [29]. The hierarchical schedule privileges real-time applications over non-real time applications. Among applications belonging to the same class, the user can decide to assign a priority for each application as a whole or use automatically generated Weight-Fair-Queue (WFQ) schedules. The weights of the WFQ are computed from the utilizations observed during the single application flow.

In order to determine if the schedules of two applications can run simultaneously we perform composability analysis. A simple approach consists of adding average utilization values and checking if the result surpasses a given threshold [11]. In our approach, additional to this simple test, we use the time traces from the TRM to perform finer analyses. Consider two applications \(a_1\) and \(a_2\) with sampled utilization functions over time for every PE \(p\) given by \(f_p(t)\) and \(g_p(t)\) respectively. The user can then select a \textit{displacement analysis} that computes:

\[
h_p(t, \tau) = f_p(t) + g_p(t - \tau)\\
H_p(\tau, T) = \int_0^\infty ((h_p(t, \tau) - T) \cdot \mu(h_p(t, \tau) - T)) \, dt
\]
Where $\mu$ is the Heaviside step function. The functions $H_p(t)$ measure to what extent the utilization of a PE goes beyond a threshold $T$, when considering all possible offsets $\tau$. The offset represents a relative starting time of application $a_2$ w.r.t. $a_1$. Depending on the values of $H_p(t)$, the tool provides a qualitative feedback of the feasibility of the current configuration. Provided with this information, the user can decide whether to keep the configuration or try another one for the current use-case. For applications with cyclic behavior, a circular shift is used instead of the time shift shown in the equations above. This computation is illustrated in Figure 3 for periodic utilization vectors and a threshold of 0.8. In this illustration, the displacement was performed in discrete time intervals. For the case in the left hand side, the utilization values of function $g_p$ are higher than those in the right hand side. As a consequence, the function $H_p$ has smaller values. Ideally $H_p = 0$, but the user can decide to select a schedule configuration for which $H_p \neq 0$ and validate later through simulation.

VI. Case Study

In order to test the framework, three applications were selected: JPEG (encoder and decoder), GSM (encoder) and MPEG-2 (decoder). The KPN representation of these applications is shown in Figure 4. The JPEG application was manually parallelized by creating a processing branch for each one of the three color components (see Figure 4a). In each branch, the rectangular-to-block ($r2b$) transformation, the discrete cosine transform ($dct$) and the quantization ($qnt$) are performed. The streams are merged by the zig-zag ($zz$) block and further decoded using the run-length-code ($rlc$). The GSM application was taken from the reference implementation in [30] and was partitioned into five data intensive processes ($pproc, lpc1, lpc2, subframe$ and $bitstream$) and two control processes as shown in Figure 4b. $pproc$ performs a 2nd order high pass filter. $lpc1$ includes the autocorrelation computation, the Levinson-Durbin algorithm and the parameter interpolation and quantization. $lpc2$ computes the weighted input speech of the whole frame and open-loop pitch delays.

![Fig. 4. Benchmarks: a) JPEG decoder. b) GSM encoder. c) MPEG-2 decoder. d) Channel accesses for the trf process in MPEG2.](image)

2 Variable names correspond to those in the reference implementation.

| Subframe itersates over the subframes and computes the pitch as well as the innovation parameters. bitstream converts the vector of analysis parameters into serial bits. Finally, an already parallelized version of the MPEG-2 application was taken from the benchmark suite of the Artist Network of Excellence [31]. Its partition and a portion of the trace of the trf process are displayed in Figures 4c,d. As target platform, a simple test architecture was created on the HVP with 3 PEs. The virtual processing elements (VPEs) were modeled as RISC processors.

After describing the applications in our framework, each of them was processed by the single application flow. Table I summarizes the results of the mapping and scheduling flow for different number of PEs in best-effort mode (Heu.). The mapping and scheduling phase took about 10 minutes on a Dual Core AMD Opteron running at 2.6 GHz for the three applications. The table also includes the results obtained by randomized search (Rand.) after 100 trials and the PE utilizations. The complete randomized search took around 25 hours. During this phase, the framework proved to be very useful for quickly rewriting the applications that were not in a parallel form. JPEG and GSM were partitioned and transformed into KPN in less than one week starting from the reference implementation. Additionally, and more importantly, the simple heuristics implemented produced good results in reasonable time, outperforming the randomized search by 18% and 9% for JPEG and 3% and 17% for MPEG-2 in 2 and 3 PEs respectively. The GSM partition has a very low parallelism and the speed-up is therefore not considerable.

A fully connected ACG for this case study was created and passed to the tool. GSM and MPEG-2 were marked as constrained with a throughput of 20 ms and 40 ms respectively3. Figure 5a shows the results of the composition step for four different configurations, having all of them GSM mapped to only one PE. In the first configuration, the complete MPEG-2 was mapped to the same processor of GSM and therefore the $H_p$ function displays very high values. The second and third configurations use 2 PEs, and variate the way MPEG-2 shares a processor with GSM. The third configuration features low values of $H_p$ and could be selected. Finally, the last configuration shows the results on three PEs for which $H_p = 0$. Code was generated for this configuration and the result was validated through simulation on the HVP simulator (see Figure 5b). In the simulations, the deadlines

### Table I: Single Application Flow

<table>
<thead>
<tr>
<th>Configuration</th>
<th>PE</th>
<th>Heu. (mega-cycles)</th>
<th>Rand. (mega-cycles)</th>
<th>Utilization(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 PEs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSM</td>
<td>Heu.</td>
<td>175</td>
<td>30</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Rand.</td>
<td>175</td>
<td>30</td>
<td>57</td>
</tr>
<tr>
<td>MPEG-2</td>
<td>Heu.</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Rand.</td>
<td>100</td>
<td>29.1</td>
<td>39</td>
</tr>
<tr>
<td>Utilization(%)</td>
<td></td>
<td>65, 99</td>
<td>65, 37</td>
<td>90, 97</td>
</tr>
<tr>
<td>3 PEs</td>
<td>Heu.</td>
<td>86</td>
<td>28.8</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Rand.</td>
<td>94</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>Utilization(%)</td>
<td></td>
<td>94, 44, 99</td>
<td>65, 40, 48</td>
<td>92, 43, 37</td>
</tr>
</tbody>
</table>

3 Constraints achievable on a RISC running at 750MHz.
were met even when adding the JPEG application (due to the hierarchical approach). Different mappings of JPEG produced different latencies, 320, 240, 235 mega-cycles for 1, 2 and 3 PEs. The composition phase quickly analyzed and discarded several configurations. For this simple case study, a total of 30 configurations were analyzed in less than one minute.

With this approach, the user can select configurations that achieve a higher platform utilization, since it is not bound to average or worst-case analysis. Being a profile-based approach, guarantees cannot be provided, and therefore one has to use it carefully when considering hard real-time applications.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we presented a complete flow for multiple applications represented using the KPN MoC in which scheduling decisions are based on profiling traces. Throughout the flow, single applications are analyzed separately and are later combined in a composition phase that is aware of the possible use-cases through an ACG. The analysis phase provides tracing and profiling of the parallel applications and allow further compiler analyses inside the processes. This phase includes an extendable mapping and scheduling flow. We tested the applicability of our framework with three typical embedded applications, JPEG, GSM and MPEG-2, for which code was generated and simulated on the HVP. The case study demonstrated how the tool can (1) ease the design effort of parallel applications, (2) produce good scheduling and mapping configuration even with simple heuristics based on traces and (3) perform fast composability analysis for KPNs providing enough information to select multi-application configurations.

In the future we plan to generate code for virtual and real platforms to further test the framework on more complex platforms. We will perform further research on KPN composability analysis and mapping heuristics. Finally, we will investigate the effects and implications of sequential partitioning within processes in already parallelized applications.

ACKNOWLEDGMENTS

This work has been supported by the UMIC Research Centre, RWTH Aachen University.

REFERENCES

[16] K. Karuri and et al, “Fine-grained application source code profiling for asip design,” in DAC ’05. ACM.