

Embedded System Design using the SPI Workbench

Marek Jersak, Dirk Ziegenbein, Fabian Wolf, Kai Richter, Rolf Ernst

Technische Universität Braunschweig

`{jersak|ziegenbein|wolf|richter|ernst}@ida.ing.tu-bs.de`

Frank Cieslok, Jürgen Teich

Universität Paderborn

`{cieslok|teich}@date.uni-paderborn.de`

Karsten Strehl, Lothar Thiele

ETH Zürich

`{strehl|thiele}@tik.ee.ethz.ch`

Abstract

1. Introduction

The design of complex embedded systems typically requires combining multiple models of computation for different problem domains [16]. Two approaches exist to model such heterogeneous systems. In the first approach, a single super-language is used. In the second approach, multiple models of computation or languages are used, each for a different part of the system. This approach has several advantages: the models of computation or languages are optimized for a specific problem domain, tools and languages are well known to designers, companies have made large investments in those tools, and many design libraries are available [7].

However, while each of these models of computation has properties which can be exploited for analysis and optimized implementation, these properties are different for each model, thus inhibiting system analysis and optimization across language boundaries. In order to solve this problem, we propose a design-flow where the multi-language input specification is translated into a common representation, the SPI model. SPI (System Property Intervals) [26, 25, 20] is an internal high-level representation that facilitates global, system-level analysis, optimization and synthesis of heterogeneously specified embedded systems.

Since research in system design automation requires a high effort to build the necessary environments to obtain results for relevant examples, and because of the large number of input languages, possible analysis and synthesis approaches, the

SPI Workbench is being built as an open research platform offering opportunities for original contributions, exchange of algorithms and access to demonstrators related to the SPI model.

SPI is based on the model of communicating processes, but with process function abstracted into a set of *externally visible properties* necessary for system-level analysis. A major contribution to the high semantic flexibility of the model is the use of behavioral intervals, e.g. time-intervals or data-rate intervals, to capture data-dependent behavior. Behavioral intervals also allow to incorporate incomplete specifications or legacy code, whose internal details are only partially known.

Most high-level languages or models of computation are well suited for *functional* specification and simulation, but typically lack formalisms to model timing requirements and constraints. As a result, timing is tested today by simulation late in the design flow. However, it is practically impossible to test complex systems completely, and extremely hard to find test-patterns that cover the corner cases. The satisfaction of timing constraints thus cannot be guaranteed.

Since this is one of the most time-consuming problems in embedded system design, the current focus of our work is on the modeling of timing-constraints, timing-analysis, and scheduling strategies that satisfy all timing constraints.

In Sec. 2, the SPI workbench is described in detail, followed in Sec. 3 by a formal introduction of the most important SPI concepts. A modeling example is introduced in Sec. 4 that is used throughout the paper. In Sec. 5, a translation from an input specification into SPI is described using the tool Simulink as an example. The annotation of timing-constraints is shown. In Sec. 6, a static approach is explained as an example for process-level timing analysis. A cyclo-static technique is

used as an example for system-level timing analysis and scheduling in Sec. 7, followed by a conclusion in Sec. 8.

1.1. Related Work

An outline of the state of the practice and the state of the art in the area of hardware/software code-sign is given in [7]. There, the insufficient coherency of the different languages, methods and tools is identified as a substantial obstacle on the way to a higher design productivity and to a reliable design process. A comparison of many models of computation accepted in industrial design and supported by an extensive set of design tools can be found in [5].

Ptolemy II [9] is a framework that supports input specifications with multiple models of computation. It thus facilitates common simulation and functional verification at a high level of abstraction. However, there is no clear path to system implementation. In particular, behavioral intervals are not considered which inhibits efficient modeling of data-dependent behavior.

Several hardware/software co-design environments are available from academia. Both POLIS [1] and COSYMA [8] supports co-design for reactive systems, but are limited to a small set of input languages, scheduling and allocation strategies. As with Ptolemy II, behavioral intervals and timing are not formalized.

Simulink [19] is used as an example input tool for SPI in this paper. Its underlying model of computation is time-driven. Real-Time Workshop [18] is the standard software code-generator for Simulink. It can be used for prototyping or as a basis for production code, but lacks the ability to guarantee timing. Timing is also a problem for other Simulink code-generators, such as TargetLink [11].

In static process timing analysis, a well known approach is based on implicit path enumeration [17]. The user provides linear (in)equations to define false paths. To evaluate these (in)equations, the upper and lower bound identification is mapped to two ILP problems. However, estimated bounds are wide because automatic path analysis is not supported as in [24, 23], which is the approach used here.

For heterogeneous hardware-software systems, typically neither purely static scheduling policies such as those developed for synchronous dataflow (SDF) [15], nor purely dynamic scheduling policies such as EDF (earliest deadline first) are appropriate. Instead, combinations of static and dynamic policies, e.g. quasi-static scheduling [14]

usually provide a reasonable compromise.

Techniques related to quasi-static scheduling have been developed for specification models such as dynamic data flow graphs [2], actors with data-dependent execution times [10], and free choice Petri nets [21]. In [22], we have proposed a symbolic scheduling approach for a mixed data flow and control flow specifications. The generated uni-processor schedules consist of statically scheduled blocks which are dynamically called at run-time.

Here, we present a complementary analysis technique that checks timing constraints in the presence of latency intervals, and if possible returns a cyclo-static multiprocessor schedule.

2. The SPI Workbench

In this section, the concepts, structure and implementation of the SPI (System Property Intervals) workbench are presented.

Fig. 1 shows the workbench structure. Input is a system with its system function captured in domain-specific *input languages or tools* with different underlying models of computation (Dataflow, StateCharts, ...), and some coupling information between the domains. There are several advantages of such a multi-language representation compared to using a uniform system specification language: each model of computation or language is optimized for a specific problem domain, languages and design tools are well known to designers, companies have made large investments in those tools, and large design libraries are available [7].

Domain-specific optimizations like transformations of signal flow graphs or composition and decomposition of state-based descriptions are best performed in these input languages. Simulation and *functional verification* can be performed on combinations of differently described system parts, independent of the SPI representation, using existing tools and co-simulation approaches [7]. However, the modeling of timing, in particular with behavioral intervals and across input languages, as well as the specification of timing constraints are not possible.

To enable system-level analysis, in particular of timing, input languages as well as IP and legacy blocks are transformed into a SPI representation. In this process, *system structure* (functional elements, states, channels, interfaces) as well as their *externally visible properties* (data rates, execution rates, activation functions) are captured, while functional details are abstracted. After the

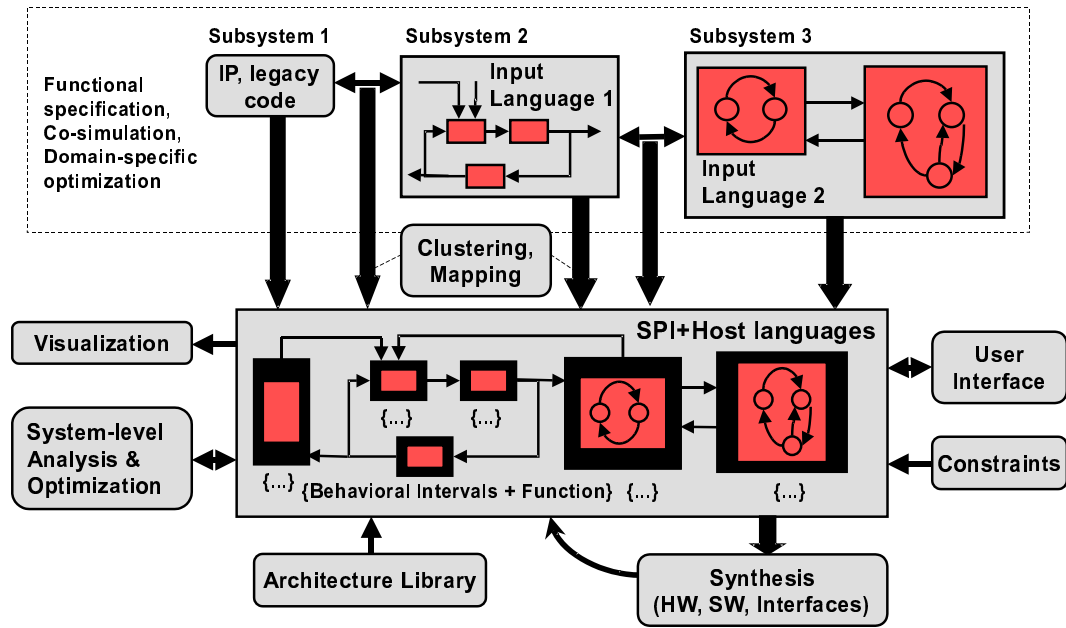


Figure 1. SPI Workbench Structure

transformation to SPI, the heterogeneously specified subsystems have been merged into a uniform representation that facilitates *combined analysis and system synthesis*. SPI is thus a system *coordination language*.

The input languages also have to be translated into *host languages* (C/C++, HDLs, ...) that implement the function of the abstract models used by the input languages, and that are well suited for implementation.

The high level of detail which is common to the host languages makes them suitable for *model execution* and *analysis of behavioral intervals*, in particular architecture- and data-dependent timing. Code-generators available for the various modeling tools typically generate output in these host languages and can be re-used in the SPI workbench, potentially with slight modifications.

Clustering is important to control the granularity of the generated processes. It should be independent of the hierarchy often used in block diagram oriented tools, since this is typically a structuring concept to facilitate comprehension and navigation of the design. However, this kind of structuring usually does not yield the best clustering for implementation.

The coupling information between input languages must be modeled in SPI with sufficient detail to allow process scheduling as well as memory sizing. One solution is message passing between processes via abstract channels, since this is

a close match with the SPI model and is supported by most code generators. Because of the compatibility of the subsystems this step is substantially simpler than the coupling for (co)simulation where the interface must provide the transition between different semantics.

Timing analysis is necessary to guarantee correct implementation of embedded real-time systems. On the SPI level, timing constraints (sensor to output latency, deadlines, execution rate jitter, ...) can be specified exactly as needed, also across input language boundaries. The result is a formal design space description that captures both function and constraints, and allows to apply a variety analysis and synthesis techniques.

Estimation and analysis of target architectures is necessary to obtain execution times (as well as other information relevant to synthesis, e.g. power dissipation). Analysis, estimation and synthesis are not part of the workbench but part of external tools and environments to account for the large number of target architectures and possible analysis and synthesis approaches. Therefore, the SPI workbench is open, and we only provide interfaces to read the SPI representation of a design and to back-annotate results into a SPI graph.

For timing analysis of single processes, profiling and simulation with selected test patterns are the state-of-the-art in industry, but since exhaustive simulation is impractical, simulation results can only cover part of the system behavior, often with

unknown coverage of critical corner cases. Static analysis is a more complicated but attractive alternative. It provides lower and upper bounds reflecting data dependent control flow as well as data dependent statement execution timing.

For system-level scheduling and allocation, a system architecture model is necessary. We are currently working on a set of architecture parameters that present a suitable architecture abstraction for our purposes. System-level timing analysis can then be performed using the set of system parameters and the system structure captured in SPI, timing constraints, process-level timing estimation and the architecture model.

Visualization shall be used for SPI graph manipulation and synthesis control. It offers additional possibilities for debugging of the workbench function itself as well as of the implemented synthesis techniques.

2.1. Implementation

Two implementations of the SPI data structures are available which can be transformed into each other. The first is in C++ and is used for efficient navigation and manipulation of the SPI representation of a design. The second is in XML [12] and is used for easy, textual interfacing between various tools and the SPI workbench. The correctness of a SPI graph can be validated by a *SPI.DTD* (a grammar file used to define XML-tags and their properties).

The specification of clustering and timing constraints is also done in XML to facilitate integration with the SPI representation of a system.

3. The SPI Model

In this section, the main concepts of the SPI (System Property Intervals) model are introduced using a small example. Since the main focus of the paper is on the SPI workbench and not on the SPI model, the introduction is restricted to the extent necessary for the understanding of the presented methodology. A formal definition of the SPI model can be found in [?, ?].

In the SPI model, a system is represented as a set of concurrent processes which communicate tokens via unidirectional channels that are either FIFO-ordered queues (destructive read) or registers (destructive write). Processes as well as channels are not characterized by their exact internal functionality but by their abstract external behavior. This behavior is captured by a set of parameters that enable the adaptation to different input languages or models of computation.

These parameters include *data rates* denoting the number of tokens consumed or produced by a process per execution on a certain channel, *latency times* denoting the time between start and completion of a process, and *activation functions* determining based on the tokens on incoming channels whether a process is ready for execution. For example, process P_1 in Figure 2 consumes 1 data token and produces 2 data tokens at each execution, and the latency time of P_1 is 1ms. Since no activation function is explicitly specified for P_1 , it is assumed by default that P_1 is activated, i. e. ready for execution, if there are enough tokens for one execution (in the example at least one) on its incoming channel.

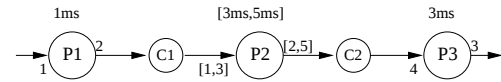


Figure 2. SPI Example

Process P_1 is completely determinate and all parameters are fixed in value. This is not necessarily the case for all processes, since the process behavior may depend on incoming data or an internal state. Thus, the SPI parameters may be specified as *behavioral intervals*, e.g. time intervals and communication intervals, i.e. intervals of produced and consumed data capturing data-dependent communication. Moreover, behavioral intervals allow the specification of upper and lower bounds in the input language. This enables the integration of processes whose internal functional details are only partially known, particularly "legacy code".

In the example, Process P_2 is specified with behavioral intervals. These intervals represent lower and upper bounds for the actual value of the corresponding parameter. Thus, P_2 consumes at least 1 and at most 3 tokens from channel C_1 and produces at least 2 and at most 5 tokens on channel C_2 , respectively. The latency time is between 3ms and 5ms.

Due to the use of behavioral intervals, the correlation between process parameters and the causal coupling of process activations is lost. This may lead to worst-case estimations that are not based on the desired system behavior and may cause false rejection or inefficient implementation of the system. Thus, the concept of *process modes* [?] was introduced that enables the explicit modeling of different execution paths within a process. For this purpose, a set of modes is associated to each process, where a mode is a tuple of data rates and latency time describing one or a subset of execu-

tion paths. For example, process P_2 may be represented as having two alternative modes:

$$\begin{aligned} m_1 &= (3\text{ms}, 1, 2) \\ m_2 &= (5\text{ms}, 3, 5) \end{aligned}$$

Then e. g. in mode m_1 process P_2 's latency is 3ms, it consumes 1 token and produces 2 tokens, etc. Nevertheless, without specifying when process P_2 shows a behavior described by one of the modes, the behavior of P_2 is still as uncertain as with the specification using the behavioral intervals.

Examples show that in many systems, there are distinct execution paths also across process boundaries. One of these examples is an MPEG2-Encoder, where the behaviors of its functional blocks depends on the coding type of the currently processed image. To enable to representation of these dependencies, *mode tags* are attached to tokens. These mode tags represent the relevant correlation information already inherent in the communicated data but not captured by the abstract SPI tokens.

To utilize this correlation information, the activation function is enhanced by a possibility to select the mode according to which the process will execute. Thus, the activation function is no longer only dependent on the numbers of available data tokens but also on the values of their attached mode tags. In the example, the activation function may be formulated as a set of rules. These rules map input token predicates to modes. For process P_2 , these rules may be:

$$\begin{aligned} a_1 &: (c_1.\text{num} \geq 1) \wedge ('a' \in c_1.\text{tag}) \mapsto m_1 \\ a_2 &: (c_1.\text{num} \geq 3) \wedge ('b' \in c_1.\text{tag}) \mapsto m_2 \end{aligned}$$

Assuming that process P_1 attaches one of the tags 'a' or 'b' to all produced tokens, the behavior of P_2 is completely determinate. If there is at least 1 available token on channel C_1 and if the tag 'a' is included in the tag set of this token, process P_2 is activated in mode m_1 . Analogously, if there are at least 3 tokens available on C_1 and the first one has 'b' in its tag set, P_2 is activated in mode m_2 .

For the implementation of embedded systems, not only the system itself but also its environment has to be regarded. To enable the representation of system and environment in a single model, *virtual* processes and channels are introduced that have the same semantics as non-virtual model elements[?]. Since they are not part of the system function, they do not have to be implemented, they rather provide additional information for synthesis. Besides the representation of the environment, these virtual model elements allow the modeling and combination of a variety of models of

computations with different activation principles in a closed form.

Part of the environment are also constraints that the implementation of the system has to fulfill. Of particular importance are timing constraints, that can be modeled in SPI by means of *latency path constraints*. Latency path constraints limit for all causal chains of data tokens on a certain path the time between their production the first channel of the path and their consumption from the last channel of the path [?]. Other timing constraints, e. g. rate constraints, can be modeled by latency constraints over virtual channels[?].

example for constraints

closing paragraph mentioning shown representations of MOCs and function variants

4. Example

The example used here is a system for evaluating codebooks for a Code Excited Linear Prediction (CELP) algorithm. An incoming speech signal is filtered by a Linear Prediction Coding (LPC) analysis filter to produce a noise-like residual. After determining the filter-coefficients on a frame of 80 speech samples, these samples are filtered one by one. Blocks of 40 filter outputs are compared with a codebook of 1024 reference blocks.

The number of the best-matching codebook entry is send serially through the channel. At the receiving side, the speech signal is reconstructed from the appropriate vector out of the codebook and the corresponding filter-coefficients which are also transmitted to the receiver.

A SPI representation of the system is shown in Fig. 3. The number of tokens consumed by process $LPC_anal.$ (a) and by $LPC_synth.$ (s) on their upper input channels is not a constant but equals *one* each 80th firing and *zero* otherwise. To model this kind of behavior, process a and b both have two modes ($m_{a,1}, m_{a,2}$ and $m_{s,1}, m_{s,2}$, respectively.) Mode-tags sent via virtual feedback channels C_{aa} and C_{ss} are used to switch between the two modes as shown in the mode-tag production rules in Fig. 3.

5. Input Specification in Simulink

Simulink [19] is a time-driven industry standard tool for simulating mixed reactive/transformative dynamic systems. It supports continuous-time, discrete-time (also multi-rate) or a hybrid of the two. The basic execution model is extended by additional semantics, such as enabledsubsystems.

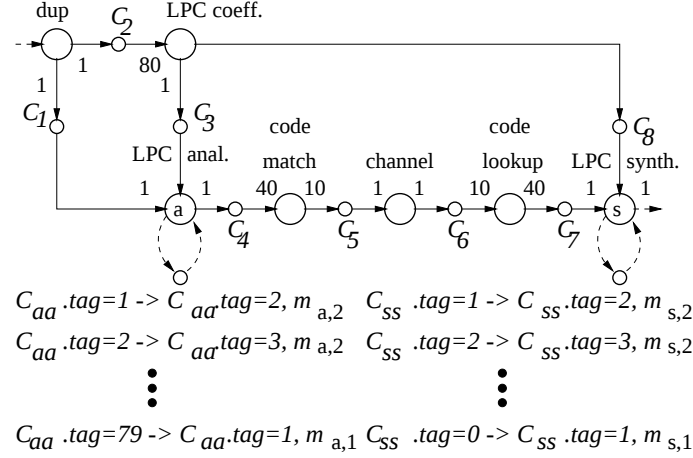


Figure 3. SPI graph for CELP algorithm

Several C-code generators are available ([18], [11]). Their main weakness is their inability to guarantee timing.

5.1. Simulink Model of Computation

In Simulink, values are communicated between blocks over directed edges that have *register semantics*. Consequently, in multi-rate designs a value on an edge can be read multiple times, or it can be overwritten before having been read.

A Simulink system is executed at certain points in time depending on the solver selected. An idealized timing model is used for block execution and communication. Both happen *infinitely fast at exact points in simulated time*. All values on edges are constant in between time steps.

5.2. Translation to SPI

While the Simulink model of time is suitable for simulation, it obviously cannot be implemented in an embedded system, explains the problems current code generators have with timing. Our goal is to relax the restrictive Simulink timing without violating the functional semantics of Simulink. It then becomes possible to specify critical timing constraints as needed. This results in a larger design-space for exploration and implementation.

1. Each Simulink block (or cluster of Simulink blocks, Sec. 5.4) is mapped into one SPI process.
2. Each Simulink edge is mapped into one SPI register channel to maintain Simulink destructive write, non-destructive read semantics. One token is written (read) on each

register channel per activation of the writing (reading) process.

3. A pair of virtual FIFO-queues is generated between every two processes that communicate over a register channel. Activation of the generated SPI processes is enabled by availability of tokens on those virtual FIFO-queues. The time-driven Simulink model of computation is thus transformed into a data-driven model which is supported by SPI.
4. Relative execution rates and partial ordering between Simulink blocks are maintained by writing (reading) the appropriate number of tokens to (from) each virtual queue, and by the number of initial tokens on each virtual queue, as specified in the following equations.

$$\begin{aligned}
 r_{virt}(P_i) &= t_s(B_i) \\
 n_{C_j(P_{wr} \rightarrow P_{rd})} &= r_{virt}(P_{rd}) - 1 \\
 n_{C_j(P_{rd} \rightarrow P_{wr})} &= r_{virt}(P_{wr})
 \end{aligned}$$

$r_{virt}(P_i)$ is the number of tokens written and read by process (P_i) per execution on each of its virtual channels. $t_s(B_i)$ is the sample time of block i . n_{C_j} is the number of initial tokens on virtual queue C_j . The direction of queue C_j is indicated by indices P_{wr} and P_{rd} , which refer to the writing and reading processes of the corresponding register channel.

A simple Simulink system and its translation to SPI can be seen in Fig. 4 and 5.

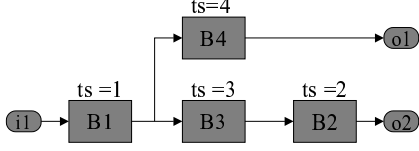


Figure 4. A simple Simulink block diagram

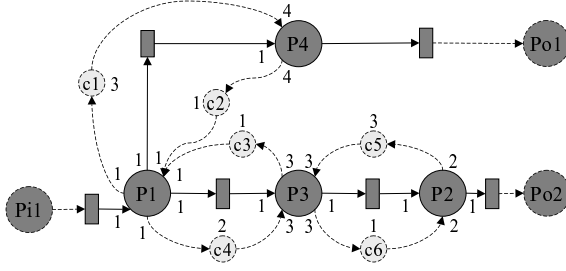


Figure 5. Generated SPI graph

5.3. (Re)introducing timing constraints

Once the design has been translated to SPI, timing constraints can be specified exactly as needed, also across input language boundaries. For example, an exact latency constraint on a virtual “self”-channel of a process (such as process *LPC anal.* in Fig. 3) forces an exact periodic activation of this process. This automatically produces the maximum possible execution time intervals for *all* processes coupled to this process through pairs of virtual queues [13].

5.4. Example: LPC Analysis

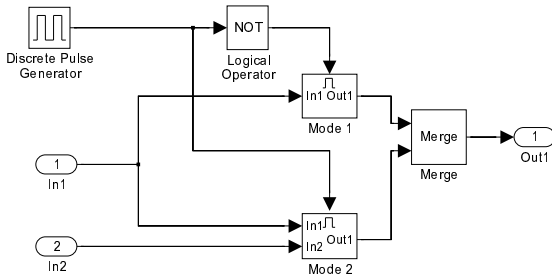


Figure 6. Simulink design with two modes

Fig. 6 shows a Simulink system that can be mapped into the process *LPC anal.* in Fig. 3. It consists of two subsystems that model the two

possible modes of process *LPC anal.*, and are enabled alternatively by the signal *en*.

Since Simulink blocks communicate over register channels, the FIFOs in Fig. 3 are modeled in Simulink as additional blocks. An additional optimization stage is necessary to convert them into SPI FIFO channels.

6. Process-Level Timing Analysis

For further process modeling on an abstract level, conservative execution timing intervals for all possible sets of input data are needed. Processes under investigation usually have a set of unpredictable input data, a compilable source code and may have different execution modes that abstract a subset of the input data to the process under investigation.

6.1. Static Timing Analysis

The execution time model in [17] is established as a standard model for static approaches which is called the sum-of-basic-blocks model in [24, 23]. Let a program consist of N basic blocks with x_i the execution count of basic block bb_i and c_i its execution time. Then, the execution time is given by the sum of all basic block execution counts x_i multiplied with their execution times c_i given by architecture modeling. Both x_i and c_i are intervals with respect to the best case and worst case bounds. The *sum-of-basic-blocks model* defines the program execution timing interval C as:

$$C = \sum_i^N c_i \times x_i$$

For the execution count intervals x_i , the designer provides an implicit description of the possible paths by means of linear equations. These functional constraints relate the execution counts x_i of the basic blocks in the control flow graph [17] to each other. The structural constraints define another set of equations: The execution count inflow d of a basic block equals its execution count x and its execution count outflow.

$$\sum_{bb} d_{inflow} = x_{i,bb} = \sum_{bb} d_{outflow}$$

These (in)equations for the upper and the lower execution count bound are mapped to two ILP problems which can be solved to derive the conservative execution count interval for each basic block which delivers the overall execution timing interval. It is assumed that all executions of one

basic block have the same cost c_i . However, data dependent instruction execution and super-scalar or super-pipelined architectures with overlapping basic block execution, as well as unpredictable cache behavior lead to widely varying local path cost with respect to latency time. For these architectures, the sum-of-basic-blocks model cannot provide close bounds, but must be pessimistic to be correct. For higher accuracy, basic block sequences in program segments must be considered.

6.2. Single Feasible Paths

Program properties can be exploited to simplify path analysis for the determination of the execution timing along basic block sequences [24, 23]. Large parts of typical embedded system programs have a single program path for any input data, even though this path may wrap around many loops, conditional statements and even function calls which are used for program structuring and compacting. Examples are an FIR filter, an FFT or the LPC analysis/synthesis with input data independent control flow. A program segment has a *Single Feasible Path SFP*, when paths through this segment are not depending on input data.

Previous analysis approaches give more than one execution path for SFP segments because they do not distinguish between input data dependent control flow and program structuring aids. In the best case, path analysis may be accurate but requires much designer interaction for SFP program segments and still does not deliver the path segment timing with overlapping basic block execution such as [17]. In case of SFP, execution would choose the one correct path and sequence for any input pattern without designer interaction.

6.3. Multiple Feasible Paths

Most practical systems also contain non-SFP parts. A program segment has *Multiple Feasible Paths MFP*, when paths through the program segment are depending on input data. SFP are exploited by finding SFP and MFP nodes in the control flow graph. Embedded MFP are cut out and analyzed separately using the approach in 6.1 while SFP are analyzed by simulating the timing of the only path with an off-the-shelf processor simulator exploiting the given sequence of basic blocks. The ILP approach in 6.1 can integrate SFP as well as basic blocks for the c_i . In our LPC analysis/synthesis example, only the control structure selecting the part of the code executed in the filter mode or in the coefficient update mode lead to an

MFP for now. The program segment for the filter and the coefficient update are completely clustered to one SFP each. The process execution timing is an interval bound by the lower and upper execution timing c of the two SFP multiplied with their execution counts x according to 6.1, regardless which mode it is executed in.

6.4. Context Dependent Paths

We have argued that the designer is often interested in a context dependent process behavior, referred to as process mode. In each context, only a subset of paths through a program segment can be executed. This potentially means reduced cost bounds which could be exploited for process analysis. For a given context, control structures depending on the input data defined by the context have a single path only. In other words, the contexts corresponding to certain modes turn an MFP segment into a segment with a single path. We will call such a segment a *Context Dependent Path* program segment *CDP*. For analysis of the given context, it is treated like an SFP-segment. For different modes, SFP segments and functional constraints for the remaining MFP segments stay the same, while a different block of CDP can be extracted from the MFP segment.

In our example, the filter mode and the coefficient update mode both turn the MFP from 6.3 into a CDP. This is clustered with the SFP of the according mode. As there is only one path per context, the context dependent execution timing interval is reduced to a single value per mode as no MFP analysis is necessary and each execution timing can be delivered by simulation.

7. Global Timing Analysis and Scheduling

In [3], we present necessary and sufficient conditions for detecting whether a SPI model graph has cyclo-static behavior or not. Cyclo-static behavior [6] means that the consumption and production rates of the processes in the SPI graph are such that the system returns to the same initial buffer state within a finite number of actor firings of each process and subsequently repeats this behavior forever.

Our now following global analysis technique allows to check timing constraints, i.e., latency path constraints of all legal execution sequences of SPI models with cyclo-static firing behavior.

Here is a short summary of our analysis technique: First, a given SPI graph is converted into an

equivalent marked graph¹ [4] by unfolding each actor as many times as is necessary in order to develop a periodic (cyclo-static) behavior. For the example of the CELP algorithm represented by the SPI graph in Fig. 3, process LPCcoef has to appear once, dup, LPCanal and LPCsynth each 80 times, codematch, codelookup 2 times, and the process named channel has to be activated 20 times within a cyclo-static execution period, see also Table 1.

In order to 1) check latency path constraints, 2) derive a multiprocessor schedule, and 3) minimize some objective, for instance the latency (period) of such a periodic activation, we formulate and solve an ILP (integer linear program) based on the unfolded marked graph equivalent.

A *periodic schedule* (with period P) of a marked graph $G = (V, A, s)$ is a function $t : V \rightarrow \mathbf{N}_0$, assigning a start time $\tau(v_i, k) = t(v_i) + k * P, \forall k \in \mathbf{N}_0$ to each vertex $v_i \in V$, so that for all edges $(v_i, v_j) \in A : t(v_j) - t(v_i) \geq w_i - s_{ij} * P$ where $\tau(v_i, k) = 0$ for all $k < 0$. $\tau(v_i, k)$ is the start time of the k^{th} iteration of vertex v_i and $s : A \rightarrow \mathbf{N}_0$ assigns the number of initial tokens $s_{i,j}$ to each edge $(v_i, v_j) \in A$. w_i is the execution time (latency) of vertex v_i . Now, given a marked graph $G = (V, A, s)$, a function $f_w : V \rightarrow \mathbf{N}_0, \forall v \in V$ denoting the latency of v the minimum possible period P_{min} may be found by solving the optimization problem

$$P_{min} = \min\{P \mid (\vec{t}, P) * \begin{pmatrix} \vec{C} \\ \vec{s} \end{pmatrix} \geq \vec{w}\}$$

in which $\vec{C}$ is the *incidence matrix* of G of dimension $|V| \times |A|$. For fulfilling a number of given latency path constraints $paths = \{LC_1, \dots, LC_p\}$, the following inequalities must be satisfied:

$$t_{lat,min,n} \leq t_{path,n} \leq t_{lat,max,n}, \quad n = 1, \dots, p$$

In [3], we have shown that this can be achieved by adding for each path constraint LC_n the two inequalities to the linear program:

$$\begin{aligned} t_{end,n} - t_{start,n} + \sum_{e_i \in path_n} s_i * P &\geq \\ t_{lat,min,n} + w_{start,n} - w_{end,n} \\ t_{start,n} - t_{end,n} - \sum_{e_i \in path_n} s_i * P &\geq \\ -t_{lat,max,n} - w_{start,n} + w_{end,n} \end{aligned}$$

Here, $t_{end,n}$ ($t_{start,n}$) denote the start time of the last (first) node in the path of constraint LC_n .

¹In a marked graph, each actor appears only once in a periodic schedule.

Finally, the latencies of each actor are intervals for general SPI graphs. So, the element of the vector $\vec{w}$ become variables, too, bounded by

$$\vec{w}_{min} \leq \vec{w} \leq \vec{w}_{max}$$

based on the interval descriptions of each process. In [3], we propose also a number of variants of this linear program such as to minimize implementation costs, to satisfy resource constraints (here, it is assumed that each process is implemented on a dedicated resource).

For the CELP algorithm, Fig. 7 shows the result of a feasible multiprocessor scheduling each actor of the unfolded SPI (marked) graph. The process execution times per activation are given in Table 1. Note that LPCcoef, LPCanal and channel are the only processes with an interval specification of their execution time. There is an interval [5, 6] for the latency of LPCcoef and LPCanal depending on whether existing coefficients or new coefficients are processed. For the channel process, we want to check latency path constraints in case the channel transport execution time may vary in between 20 to 50 time units per activation, see Table 1. Finally, let us formulate a latency path constraint $LC_{path} = [0, 1500]$ for any token traveling on the computation path $path = (c1, c4, c5, c6, c7)$. Obviously, the latency path constraint is satisfied for the cyclo-static schedule shown in Fig. 7 for an assumed execution time of 20 time units for each activation of process channel. However, for an assumed execution time of 50 time units, the ILP becomes infeasible, hence the latency path constraint will not be satisfied if the channel requires the upper bound of its latency interval.

Process	Latency Lat		Activations per period
	min	max	
dup	1	1	80
LPCcoef	200	200	1
LPCanal	5	6	80
codematch	150	150	2
channel	20	50	20
codelookup	75	75	2
LPCsynth	5	6	80

Table 1. Execution times of CELP processes in time units.

8. Conclusion

(Marek, Dirk, max 0.5 pages)

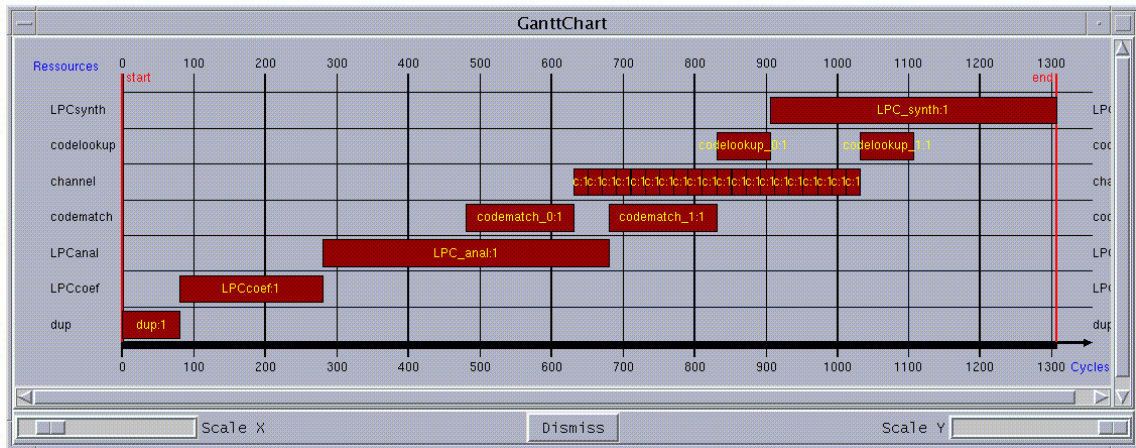


Figure 7. Gant-chart of a cyclo-static schedule for the CELP algorithm (one period)

References

- [1] Felice e.a. Balarin. *Hardware-Software Co-Design of Embedded Systems. The POLIS Approach*. Kluwer Academic Publishers, 1997.
- [2] J. T. Buck. Scheduling dynamic dataflow graphs with bounded memory using the Token Flow Model. Technical Report UCB/ERL 93/69, Ph.D dissertation, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1993.
- [3] F. Cieslok and J. Teich. Timing analysis and scheduling of process graphs with uncertain execution times. Technical report, TR No. 1/00, Computer Engineering Laboratory, University of Paderborn, March 2000.
- [4] F. Commoner and A.W. Holt. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.
- [5] e. A. Lee and Th. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, May 1995.
- [6] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-Static Data Flow: Model and implementation. In *Proc. 28th Asilomar Conf. on Signals, Systems, and Computers*, pages 503–507, Pacific Grove, CA, 1994.
- [7] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design & Test of Computers*, April 1998.
- [8] R. Ernst, J. Henkel, and Th. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, December 1993.
- [9] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. CAD*, June 1999.
- [10] S. Ha and E. A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Trans. on Computers*, 46(7):768–778, July 1997.
- [11] H. Hanselmann, U. Kiffmeier, L. Köster, and M. Meyer. Automatic generation of production quality code for ECUs. Technical report, dSPACE GmbH, March 1999. Distributed during Embedded Intelligence, Nürnberg, Germany.
- [12] E. R. Harold. *XML Bible*. IDG Books Worldwide, 1999.
- [13] Marek Jersak, Ying Cai, Dirk Ziegenbein, and Rolf Ernst. A transformational approach to constraint relaxation of a time-driven simulation model. In *Proceedings 13th International Symposium on System Synthesis*, Madrid, Spain, September 2000.
- [14] E. A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In R. W. Brodersen and H. S. Morowitz, editors, *VLSI Signal Processing III*, pages 330–340, IEEE Press, New York, 1988.
- [15] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- [16] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. CAD*, December 1998.
- [17] Y. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.
- [18] The MathWorks, Inc. *Real-Time Workshop User's Guide, Version 3*, January 1999.
- [19] The MathWorks, Inc. *Using Simulink, Version 3*, January 1999.
- [20] K. Richter, D. Ziegenbein, R. Ernst, J. Teich, and L. Thiele. Representation of function variants for embedded system optimization and synthesis. In *Proceedings 36th Design Automation Conference (DAC '99)*, New Orleans, USA, June 1999.
- [21] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using free-choice petri nets. In *Proc. of the Workshop on Hardware Design and Petri nets (HPWN'98)*, 1998.
- [22] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich. Scheduling hardware/software systems using symbolic techniques. In *Proc. CODES'99, the 7th Int. Workshop on Hardware/Software Co-Design*, pages 173–177, Rome, Italy, May 1999.
- [23] F. Wolf and R. Ernst. Intervals in software execution cost analysis. In *Proceedings 13th International Symposium on System Synthesis*, Madrid, Spain, September 2000.
- [24] W. Ye and R. Ernst. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings ICCAD '97*, San Jose, USA, 1997.
- [25] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings Sixth International Workshop on Hardware/Software Co-Design (Codes/CASHE '98)*, Seattle, USA, March 1998.

- [26] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *Proceedings International Conference on Computer-Aided Design (ICCAD '98)*, San Jose, USA, November 1998.