# Interval-Based Analysis of Software Processes

D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, R. Ernst*
Institute for Computer and Communication Network Engineering
Technical University of Braunschweig
Hans-Sommer-Str. 66
38106 Braunschweig, Germany

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

## ABSTRACT

A typical characteristic of complex embedded systems is their large software share that consists of software processes either being directly written in an implementation language like C, or being created from abstract modeling tools (e. g. Simulink or StateMate) using standard code generators, or being reused from previous designs (e. g. legacy code). A major challenge is the safe integration of these separately designed system parts. This paper focuses on the formal analysis of software processes with respect to their non-functional properties like timing or power consumption. The proposed approach yields safe upper and lower bounds on these properties and has advantages over previous work in terms of accuracy and efficiency. Further, it is shown how the results of this process-level analysis can be utilized to generate a model for the system-wide validation of non-functional properties. The applicability of the approach is demonstrated using an example of a filter process operating on a packet stream.

## Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; C.4 [**Computer Systems Organization**]: Performance of Systems; D.4.8 [**Operating Systems**]: Performance—*Operational analysis, modeling and prediction*

## General Terms

Algorithms, Design, Measurement, Performance, Theory

## Keywords

Behavioral intervals, software execution cost analysis, system-level timing validation

*This work was supported by the German DFG.

## 1. INTRODUCTION

Complex embedded systems such as mobile communication devices and automotive control or multi-media systems typically have a large software share that commonly consists of several software processes. The sources of these processes can be classified firstly into functions written directly in an implementation language, traditionally C and more recently C++ based language extensions. Secondly, standard code generators produce software processes from more abstract modeling tools that are established industry praxis in certain application domains (e. g. signal processing or communication protocols). Thirdly, reused components (legacy code, IP) are increasingly included in the design to reduce the amount of design work required. For the safe integration of these different heterogeneous system parts not only the function but also non-functional properties of the software processes have to be analyzed and taken into account.

While there are several cosimulation-based approaches from both academia and industry that are well suited for functional validation and yield an easy understanding of the complete system function, these approaches fail to reliably validate non-functional constraints e. g. with respect to timing and power consumption. This is mainly due to their lack to consider input data dependent behavior, which is quite common for embedded systems and leads to intervals instead of single values for non-functional properties. These intervals are called *behavioral intervals*. Since exhaustive simulation is infeasible in most cases, cosimulation can only cover part of the system behavior and thus can not guarantee that a corner case resulting e. g. in the worst case latency time of a process is covered.

We propose a formal two-level approach that focuses on the analysis of non-functional properties, in particular timing and power consumption. The validation of the system function is not regarded. The intended analysis flow is shown in Figure 1. At the process-level, behavioral interval analysis is performed separately for each process, i. e. that upper and lower bounds on non-functional properties of the processes are obtained. Since these properties may depend on the target architecture (i. e. the processor the process will be executed on), the implementation has to be considered in this step. The sources of behavioral intervals can be input data dependent process behavior (due to input data dependent control structures) or limited analyzability of the target architecture (due to features like pipelining, caches etc.).

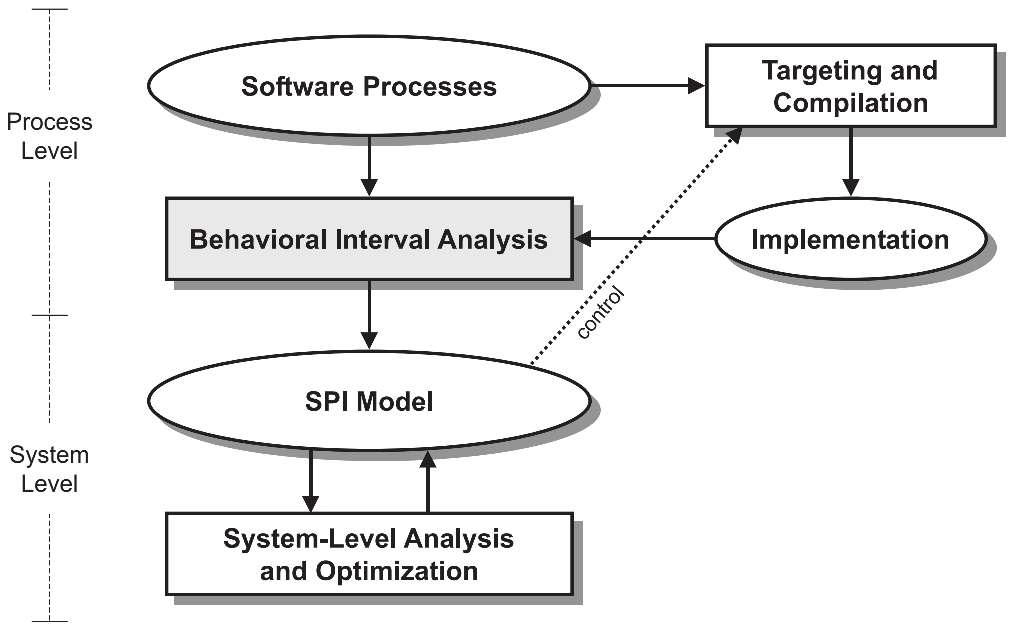



**Figure 1: Proposed Two-Level Analysis Flow**

At the system-level, the first step is to build a homogeneous model of the whole system. This is done by means of an intermediate design representation called the SPI (System Property Intervals) model [18, 19] that is specially targeted to analysis and synthesis of heterogeneously specified embedded systems. In the SPI model, process behavior is represented by parameters such as latency times, power consumption and the communicated amount of data. The values of these parameters are the behavioral intervals obtained by the process-level analysis. Furthermore, SPI supports the explicit specification of input data dependent behavior by means of the concept of process modes.

The created system model then serves as a starting point for the system wide analysis. Additionally, this system model can also be used for optimization steps such as scheduling or load balancing. This is denoted by the dotted line in Figure 1 representing the control of implementation steps. A more detailed description of the SPI methodology can be found in [8].

The consideration of behavioral intervals is of integral significance for our approach, since for many system-level analysis approaches not only upper but also lower bounds on system properties are needed to guarantee the compliance with non-functional constraints. Well-known examples are real-time scheduling anomalies as mentioned e. g. by Gerber et al. [5].

This paper focuses on the behavioral interval analysis of software processes. The proposed analysis method extends the well-established sum-of-basic-blocks method [10], a formal static software execution cost analysis approach where the overall process execution cost is the sum of all basic block execution costs multiplied by the corresponding execution count for each of the basic blocks. The added value of our approach is the automated path analysis that identifies and clusters sequences of basic blocks that have a single input data independent control flow. Furthermore, user information on the execution context can be considered to enable the modeling of internal as well as inter-process control flow at the system-level. Both steps lead not only to more accurate results but also raise the granularity level and thus reduce the problem size of the embedded ILP problem.

Our approach does not only yield behavioral intervals for timing and power consumption but also bounds the communicated amount of data per process execution. This is a prerequisite for system-level steps like communication scheduling and memory allocation. Based on the obtained behavioral intervals, approaches like [9] can easily estimate the latency times of communications between processes.

The rest of the paper is organized as follows. In Section 2, the basic concepts of the SPI model including behavioral intervals and context dependent behavior are presented. Then in Section 3, the analysis of behavioral intervals for software processes is introduced. After the example application of our approach on a filter process in Section 4, the paper is concluded.

## 2. THE SPI MODEL

In this section, the basic concepts of the SPI model are introduced, but the main goal of this section is to motivate the need for behavioral intervals. A more formal definition of SPI can be found in [18, 19].

### 2.1 Behavioral Intervals

In the SPI model, the system is represented as a set of concurrent processes which communicate via unidirectional channels that are either FIFO-ordered queues or registers. While each channel node simply transfers data from the sender to the receiver without any transformation, the functionality of process nodes can be of arbitrary complexity. However for scheduling, allocation and performance analysis, knowledge about the detailed functionality of a process is not needed. It is sufficient to know for each process the resource requirements and the interaction with its environment.

These properties of the processes and channels are captured by parameters that are annotated to the corresponding elements. This allows an easy adaption of the model to include all required information for a certain optimization goal or task in the design flow. In the context of this paper, we consider data rates, latency times and power consumption.

The parameters need not be fixed but can be specified using behavioral intervals, i. e. they are constrained by an upper and lower bound. The sources for this non-determinism can be abstraction of input data dependent functionality of a process (e. g. due to `if-then-else` structures depending on input data) or limited analyzability of the input model, on the one hand, or incomplete specification resulting in estimation of parameters, on the other hand. While for the abstraction and limited analyzability the parameter may switch between all possible values of the interval at run-time, the non-determinism of the incomplete specification will be eliminated before run-time such that it can be assumed that the parameter will take just one of the possible values of the behavioral interval. For implementation dependent parameters like timing and power consumption, limited analyzability of the target architecture (e. g. caches, pipelining, out-of-order execution) is another source of non-determinism. In the SPI model, however, the different types of non-determinism are not distinguished since the differences analysis and optimization methods could utilize are minimal.

For the extraction of behavioral intervals from software processes, not only the input data dependencies of control structures but also the limited analyzability of the target architecture have to be considered as sources for behavioral intervals.

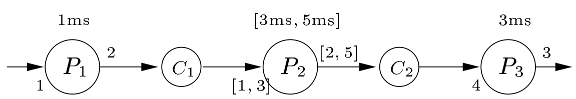



**Figure 2: SPI Example**

An example of a SPI model is depicted in Figure 2. At each execution, processes consume input data and produce output data. However, since we are not interested in the function performed by a process, the communicated data is only represented by the amount of data which is important for communication scheduling or memory allocation. For example, process $p_1$ consumes 1 data token and produces 2 data tokens at each execution. The latency of $p_1$ is 1 ms. Process $p_2$ is an example for a process that is specified using behavioral intervals, as it 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 execution latency is between 3 ms and 5 ms.

The SPI (System Property Intervals) model is specifically targeted to heterogeneous system design and allows to capture the coordination semantics of various established models of computation, such as dataflow, finite state machines, time-driven models or Petri nets. Therefore, the basic structure and properties of the SPI model resemble the structure and coordination semantics of those models. The main difference between SPI and those models is the fundamental support of behavioral intervals that makes the SPI model an excellent choice for our analysis approach.

## 2.2 Context Dependent Behavior

Behavioral intervals of a process depend to a certain extend on the process control flow which in turn depends on process input data. Hence, behavioral intervals are *context dependent* where a context is a value definition for a subset of input data variables. Thus, a context determines control structures and thereby selects a subset of execution paths through the process. The analysis of such a path set typically yields behavioral intervals with much narrower bounds. But even the analysis of a single execution path may still yield behavioral intervals and no fixed values for implementation dependent parameters due to limited analyzability of target architecture features like caches, pipelines or data dependent instruction execution times.

Context dependent behavior can be modeled in SPI using the concept of process modes. Each process mode represents the behavioral intervals for a certain context. For example, the analysis of process $p_2$ with two different contexts may yield the following two alternative modes:

$$m_1 = ([3, 3.8]\text{ms}, 1, 2)$$
$$m_2 = ([4.5, 5]\text{ms}, 3, 5)$$

Then in mode $m_1$ process $p_2$'s latency is between 3 ms and 3.8 ms and $p_2$ consumes 1 token and produces 2 tokens, while in mode $m_2$ its latency is between 4.5 ms and 5 ms and it communicates 3 and 5 tokens, respectively. Nevertheless, without also representing the context that corresponds to the process execution described by the respective mode, the behavior of process $p_2$ is still uncertain since $p_2$ may show either the behavioral intervals represented by $m_1$ or $m_2$.

As mentioned above, in the SPI model data is abstracted to tokens without information about values. However, contexts have to be visible in order to analyze the reaction of processes or systems on certain input data. A typical example is a base station for wireless communication, where there are several paths on which different data packets are routed through a network of processes. Important questions for the system designer can be the power consumption for sending a data packet or the time to set up a connection in a base station. This should take the system context into account, since for each packet type the processes react with a different control flow.

To model such inter-process correlations, *virtual mode tags* may be associated to data tokens to represent contexts. The correspondence between contexts and their respective process modes is modeled by a set of *activation rules* that is associated with each process. These rules map input token predicates to modes. A predicate in this case is either ‘true’ or ‘false’ depending on the number of tokens and the tags of the first tokens on the input channels of the process. For process $p_2$ of the above example, these rules could be:

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

Assuming that process $p_1$ adds one of the tags 'a' or 'b' to the tag set of all produced tokens, the communication 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 $m_2$.

The designer typically has knowledge of execution contexts and process correlations for the system or system part she designed. By utilizing this knowledge for analysis and parameter extraction, the behavioral intervals not only of a single process but also of the whole system can be substantially narrowed.

# 3. BEHAVIORAL INTERVAL ANALYSIS

In this section, our approach to behavioral interval analysis including its limitations and advantages over previous approaches is presented. We consider latency time, power consumption and the sent and received amounts of data of a software process. Since the overall methodology for the determination of these different properties is very similar, we will use the general term *execution cost* in the following text and will only refer to each property where the methods differ.

Our approach is performed off-line, i. e. before the deployment of the system, and consists of the following three steps

- Identification and classification of possible paths across basic block boundaries called process segments (Section 3.1)
- Determination of execution cost intervals for each process segment (Section 3.2)
- Combination of process segment execution costs using the execution count intervals obtained by an ILP approach (Section 3.3)

in order to determine the overall behavioral intervals of a given software process. In Section 3.4, the SYMTA tool that implements the approach is presented. A more formal introduction of the process-level behavioral interval analysis approach can be found in [16].

## 3.1 Static Path Analysis

Behavioral intervals for process execution cost can be highly input data dependent. This may be caused by input data dependencies of control flow and of instruction execution, e. g. for microcoded multiplication or library functions. The test pattern selection for the coverage of the extreme cases to obtain exact bounds on the execution cost using simulation is an undecidable problem. Formal analysis of input data dependencies is a feasible alternative because it can guarantee that the actual execution cost interval falls into the predicted behavioral interval. This can be independent of any designer decision like test pattern selection. As all standard formal software analysis approaches (e. g. [10]), our approach assumes structured programs, i. e. code without the use of jump-like statements. Thus, `goto` is not allowed as well as `break` statements within loops. Another restriction is that pointers are currently not supported. However, the approach could be extended in a way similar to [13] where pointers and also dynamic memory allocation are resolved.

### 3.1.1 Basic Approach

The latency time model in [10] is established as a standard model for static approaches, which is also called the *sum-of-basic-blocks* model. Here, the overall process execution cost is the sum of all basic block execution costs multiplied by the corresponding execution count for each of the basic blocks. Evidently, the execution cost of a basic block depends on the target architecture whereas the execution count is architecture independent.

Both values, cost and count, are intervals representing the worst case and best case bounds. It is assumed that all executions of one basic block have the same cost interval. However, data dependent instruction execution and pipelined architectures as well as unpredictable cache behavior and register allocation lead to widely varying basic block execution cost. This effect is referred to as overlapping basic block execution. For these architectures, the sum-of-basic-blocks model cannot provide close bounds, but must use very pessimistic basic block cost intervals to be correct for all executions of the basic block because empty pipelines or cache flushes have to be assumed for basic block beginnings. Many other approaches are also based on the analysis granularity of basic blocks [3], single basic block transitions [6] or require complex modifications to cost determination [11].

A way to achieve a higher analysis accuracy is to consider *basic block sequences* for the determination of execution costs. This way, pipeline and cache behavior between basic blocks in such a sequence can be exactly modeled without assuming worst cases in between them.

### 3.1.2 Single Feasible Paths

Large parts of typical embedded system processes have a single path independent of input data, even though this path may wrap around many loops, conditional statements and even function calls which are used for source code structuring and compacting. Examples are an FIR filter or a Fast Fourier Transformation. These input data independent paths are called *Single Feasible Paths (SFP)*.

A *Process Segment (PrS)* is a sequence of nodes with exactly one first and one last basic block. This follows the definition of basic blocks in [1]. The execution cost of a PrS is delimited by an upper and lower bound. The key to finding SFP process segments (SFP-PrS) is to distinguish between input data dependent control flow and source code structuring aids. SFP-PrS are characterized by input data independent control structures. We use a very efficient depth-first search algorithm on the syntax graph which is sufficiently accurate in praxis combined with symbolic execution of basic blocks to determine these dependencies. The syntax graph has been chosen because it contains the hierarchy of control structures. In the approach in [10], path analysis by functional constraint annotation for SFP-PrS may be accurate but requires much tedious and error-prone designer interaction while the approach still uses the sum-of-basic-blocks cost model with its drawbacks regarding accuracy. Basic block sequences are easy to consider for SFP-PrS because their execution cost can be determined using simulation. For process-level analysis, SFP-PrS can be treated like basic blocks.

Obviously, most practical systems also contain non-SFP process segments. A process segment is said to have *Multiple Feasible Paths (MFP)*, when paths through the process segment depend on input data. An MFP-PrS only consists of a control structure with input data dependent control flow. A more formal classification of PrS is given in [16].

#### 3.1.3 Context Dependent Paths

As already mentioned in Section 2.2, processes often have context dependent behavior. A context is a value definition for a subset of input values that is specified by the designer. In each context, only a subset of paths through a process segment can be executed, since the control flow is defined by context dependent input data. This usually means tighter execution cost bounds for a given context. In other words, the contexts turn an MFP-PrS into an SFP-PrS. We call such a path a *Context Dependent Path (CDP)*. Thus, for further analysis of the given context, a CDP-PrS is treated like an SFP-PrS.

## 3.2 Process Segment Execution Cost

In the following, different approaches to obtain execution costs of a single process segment and their application to the determination of latency times, power consumption and communicated data are presented.

The *Instruction Cost Addition* (ICA) approach uses a generalization of the sum-of-basic-blocks approach to calculate the execution costs of a PrS. Since an SFP-PrS or basic block has a single path only, it just needs to be executed on a host system to derive the execution counts for all basic blocks in the PrS. There is no input data dependent control flow in such an SFP-PrS, so this execution yields the exact instruction counts. A sum-of-basic-blocks calculation provides the total cost of one PrS execution by using a cross compiler and instruction cost tables with respect to the target architecture. This leads to good results for simple architectures without overlapping basic block execution.

The execution cost analysis approach to choose for architectures with overlapping basic block execution is cycle true *process segment simulation* (PSS). Since the SFP-PrS execution path is fixed like in a basic block, the execution costs are fixed such that a single simulation of the SFP-PrS is sufficient. A conservative overhead is added to cover the worst case of all different entry paths into the SFP-PrS which can represent different states for register allocation, pipelines and caches. The major improvement is the extension of basic blocks to SFP-PrS, so an execution cost simulation of this segment with any off-the-shelf processor simulator automatically chooses the one correct path and exploits the basic block sequence without designer interaction. Too pessimistic overheads between basic blocks can be replaced by the exact pipeline and cache behavior.

Both, ICA and PSS require process execution or simulation. Clearly, the input patterns should be selected such that all PrS are executed at least once but some process segments may be difficult to reach. These are automatically detected and can be simulated separately to guarantee full code coverage. In case of input data dependent instruction execution cost, the result must be corrected for each execution of a data dependent instruction to obtain the correct cost interval. In both approaches the influence of caching is accounted for by using a cache tracing tool [7] or the target system cache model. An overhead assuming first misses for the SFP-PrS start must be included that can be refined using global data flow analysis for cache line contents [15].

#### 3.2.1 Latency Time

For ICA, latency time intervals for single machine instructions can be taken from data books. When using PSS, most off-the-shelf simulators deliver the execution timing of the single machine instructions as well as the complete process segment. Thus, the latency time of a PrS is straightforward to determine.

#### 3.2.2 Power Consumption

The software power consumption of a PrS can be simulated using a simplification of the methodology presented in [14]. It proposes an ICA approach with base and transition power values for a sequence of instructions given by host simulation. For RISC architectures, experiments show that the simulation adequately matches the measured power consumption. Influences of data values or cache behavior can be modeled via additional processor cycles that add to the instruction power consumption.

#### 3.2.3 Communicated Data

For the analysis of the communicated data of an SFP-PrS, we assume only explicite communication. The amount of data sent or received by a process influences timing and power consumption of communication components like buffers, busses or memories. We can use an ICA approach to determine the data sent or received on an SFP-PrS where the cost is given by the size of the communicated data block.

## 3.3 Process-level Analysis

The process-level execution cost is an interval bound by the lower and upper execution costs given by the SFP-PrS cost intervals multiplied with their execution count intervals. In previous approaches (e. g. [10]), the designer has to provide an implicit description of the possible paths by means of linear equations in order to determine the execution count intervals. These so called *functional* constraints specify the relations of the execution counts of the basic block nodes in the control flow graph to each other. Another set of equations captures *structural* constraints, e. g. that the execution count inflow of a basic block node equals its execution count and its execution count outflow. These (in)equations are mapped to two ILP optimization problems, one for the upper and one for the lower execution count bound. These are solved to derive a conservative execution count interval for each basic block.

In the presented approach, basic block nodes are clustered to SFP-PrS. Since the ILP problem treats SFP-PrS execu-

```
header = receive(INPUT, HEADER_SIZE);
for all pixels              /* Context: Size    */
  picture[y][x] = receive(INPUT, 1);

if(address == MY_ADDRESS) { /* Context: Address */
  for all pixels {
    for a 3*3 pixel window {
      if(without_center)
        average = sum/8;
      else average = sum/9;
      }
    if(abs(picture[y][x]-average)>threshold)
      send(OUTPUT, average, 1);
    else send(OUTPUT, picture[y][x], 1);
  }
}
```

**Figure 3: Pseudo code of the filter process**

tion cost in the same way as basic block execution cost, it only needs to be solved for the input data dependent control structures between SFP-PrS. For different contexts, SFP-PrS and additional functional constraints for the remaining MFP-PrS stay the same, while a different set of CDP-PrS can be extracted from the MFP-PrS. When all input data that influences control flow is context dependent, no ILP analysis is necessary and each context dependent execution cost can be obtained by simulation.

Besides the higher achieved analysis accuracy that leads to tighter behavioral intervals, the use of process segments instead of basic blocks has another major advantage. Due to the raised granularity, the size of the ILP problem for the process-level solution can be significantly reduced. This yields a shorter computation time of the ILP solution, but more importantly enables the application of our approach to much more complex processes as compared to previous approaches.

## 3.4 The SYMTA Tool

The presented approach is implemented in the SYMTA tool suite (SYMbolic Timing Analysis) that in contrast to its name is not only capable to determine behavioral intervals for latency times but also for power consumption and communicated data. A major advantage of our approach is its flexibility with respect to its front end (possible input languages) as well as to its back end (possible target architectures).

Since the path analysis of our approach is based on a syntax graph and a control flow graph, we do not restrict ourselves to a specific input language. Currently, one front end has been implemented for $C^x$ [2], a C derivative that extends ANSI-C by generic `send` and `receive` functions which implement the inter-process communication. Very similar notations can also be found in more recent C++ derivatives targeted to system-level design like SpecC [4] and SystemC [12]. Here, interface methods (e. g. `write` and `read`) are used to access external channels. Our analysis approach can be easily enhanced to also support these languages.

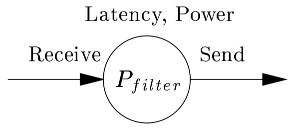
For process segment cost analysis, ICA and PSS have been implemented for a set of target architectures covering very different domains. ICA has been implemented for Intel 8051 where it can deliver accurate results because no overlapping basic block execution is present. PSS has been implemented for StrongARM and SPARC including pipeline and cache tracing [16]. Due to the drawbacks regarding performance and availability of cycle-true simulators for complex architectures, the PrS execution cost can also be obtained by measuring the corresponding physical values. Then, code instrumentation and bus observation are used to measure execution costs for a specific PrS between predefined and automatically inserted trigger points. The results are automatically back annotated to be used in SYMTA. This measurement approach has been implemented for a SPARClite evaluation board [17]. The possibility to use off-the-shelf simulators as well as evaluation platforms enables us to explore different target architectures in a very flexible way.

# 4. EXAMPLE: FILTER ON PACKET DATA

The behavioral interval extraction has been applied to a single filter process that reads a packet containing a picture. If the packet is addressed to its system component, the filter process performs an "unlikely dot" filtering on the picture data and sends it to another process. Possible execution contexts are the processing of a "large" or a "small" picture and address match or miss. The pseudo code of the process is given in Figure 3.

Tables 1 and 2 show the behavioral intervals of the filter process with respect to latency time, power consumption of the processor core and communicated data for all combinations of execution contexts. The intervals are of the form $[b_{min}, b_{max}]$ where $b_{min}$ denotes the lower bound on the value of the respective property while $b_{max}$ denotes the upper bound. The values were obtained using the SYMTA tool for a StrongARM with 80 MHz core frequency, 40 MHz bus frequency and 25 ns memory cycle time including local cache simulation.

When comparing the values for the different combinations of contexts, it can be seen that the utilization of specified contexts during analysis of the filter process helps to substantially narrow the extracted behavioral intervals. For the communicated amount of data, this context dependent analysis even yields deterministic values i. e. the upper and lower bounds of an interval are equal.




| | Latency [ms] | Power [mWs] | Send [kB] | Recv. [kB] |
|---|---|---|---|---|
| *Values for SPI representation with single mode* | | | | |
| $m_{\text{single\_mode}}$ = ( | [6, 572], | [2, 107], | [0.0, 24.4], | [6.2, 25.0] ) |
| *Values for SPI representation with four modes* | | | | |
| $m_{\text{small,match}}$ = ( | [38, 64], | [21, 24], | [5.9, 5.9], | [6.2, 6.2] ) |
| $m_{\text{small,miss}}$ = ( | [6, 13], | [2, 4], | [0, 0], | [6.2, 6.2] ) |
| $m_{\text{large,match}}$ = ( | [265, 572], | [97, 107], | [24.4, 24.4], | [25.0, 25.0] ) |
| $m_{\text{large,miss}}$ = ( | [20, 39], | [8, 9], | [0, 0], | [25.0, 25.0] ) |

**Figure 4: SPI representations of filter process**

| Latency [ms] Power [mWs] | | Address not considered | Address miss | Address match |
|---|---|---|---|---|
| Size not considered | Latency | [5, 681] | [6, 40] | [38, 681] |
| | Power | [2, 117] | [2, 9] | [21, 117] |
| Large Picture | Latency | [19, 572] | [20, 39] | [265, 572] |
| | Power | [8, 107] | [8, 9] | [97, 107] |
| Small Picture | Latency | [5, 67] | [6, 13] | [38, 64] |
| | Power | [2, 25] | [2, 4] | [21, 24] |

**Table 1: Behavioral intervals for latency and power consumption of the filter process**

| Send Data [kB] Receive Data [ms] | | Address not considered | Address miss | Address match |
|---|---|---|---|---|
| Size not considered | Snd | [0, 24.4] | [0, 0] | [5.9, 24.4] |
| | Rec | [6.2, 25.0] | [6.2, 25.0] | [6.2, 25.0] |
| Large Picture | Snd | [0, 24.4] | [0, 0] | [24.4, 24.4] |
| | Rec | [25.0, 25.0] | [25.0, 25.0] | [25.0, 25.0] |
| Small Picture | Snd | [0, 5.9] | [0, 0] | [5.9, 5.9] |
| | Rec | [6.2, 6.2] | [6.2, 6.2] | [6.2, 6.2] |

**Table 2: Behavioral intervals for sent and received data amounts of filter process**

An interesting effect is that the maximum latency of 681 ms without considering any execution context (see upper left element of Table 1) is not contained in the behavioral intervals for neither the large nor the small picture context (572 ms and 67 ms respectively). This is due to the fact, that for each of both contexts two process segments have merged such that the worst case assumptions on the cache and pipeline state for the beginning of the second segment can be dropped.

In comparison to our approach, a standard basic block based approach that does not exploit SFP properties yields far wider intervals due to the overlapping basic block effects in the nested loops. Using the approach from [10] including functional constraint annotations for the loops with respect to both picture sizes, the worst case bounds on the latency of the filter process have been calculated. The results were 6368 ms and 887 ms for a large and a small picture respectively compared to 572 ms and 67 ms obtained by our approach.

Further experiments comparing the results of our approach and of the approach from [10] can be found in [16]. These experiments validate the improved accuracy of the SYMTA tool. The range of improvement is usually in the range of about 20% and increases to one order of magnitude in the presence of nested loops as common for signal processing algorithms.

Based on the obtained behavioral intervals, a SPI representation of the filter process can be created that may be used to analyze the overall system performance. Two SPI representations of the filter process using the obtained behavioral intervals with a single mode and four modes are depicted in Figure 4.

Note that for the representation with a single mode, the used behavioral intervals for latency ([6, 572] ms) and power consumption ([2, 107] mWs) are narrower than the intervals obtained without considering any context depicted in the upper left element of Table 1 ([5, 681] ms and [2, 117] mWs). This reduction of the interval width is possible due to a *complete* context dependent analysis, i. e. if all possible paths through a context dependent control structure have been covered by different contexts, the overall behavioral intervals can be reduced to the union of the obtained behavioral intervals for these contexts. In case of several context dependent control structures, the cross product of all contexts has to be considered. Thus for the example, the overall behavioral intervals can be combined from the four table elements in the bottom right corner of tables 1 and 2, e. g. the lower bound on the latency equals the minimum of 20, 265, 6, and 38.

## 5. CONCLUSION

We have presented a formal analysis approach for software processes that yields safe behavioral intervals for key process properties like timing, power consumption and communicated data. Due to the automated path classification and clustering that raises the granularity level of the analysis from basic blocks to process segments, our approach is advantageous in terms of accuracy (narrower intervals because of less pessimistic worst case assumptions) and efficiency (smaller problem size of embedded ILP formulation).

User information on the execution context of processes can be utilized to further narrow the obtained behavioral intervals and to enable the explicit modeling of internal as well as inter-process control flow. The approach has been implemented in a modular way so that it is very flexible with respect to the possible input languages, the possible target architectures, and the execution cost determination methods.

The presented approach is part of the SPI workbench, a system-level analysis and optimization approach for the design of heterogeneously specified embedded systems that is currently being implemented in an international cooperation.

## 6. REFERENCES


[1] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, GB, 1988.

[2] Th. Benner, A. Österling, and R. Ernst. Comparison of Context Switching Methods for Fine Grain Process Scheduling. Technical Report CY-96-1, Institute of Computer Engineering, Technical University of Braunschweig, Germany, 1996.

[3] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[4] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, 2000.

[5] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time taks. *IEEE Transaction on Computers*, 44(3):471–479, March 1995.

[6] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, pages 53–70, January 1999.

[7] M. Hill. DINERO III Cache Simulator: Source Code, Libraries and Documentation. www.ece.cmu.edu/ ece548/tools/dinero/src/, 1998.

[8] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, R. Ernst, F. Cieslok, J. Teich, K. Strehl, and L. Thiele. Embedded system design using the spi workbench. In *Proceedings 3rd Forum on Design Languages*, Tübingen, Germany, September 2000.

[9] P. V. Knudsen and J. Madsen. Communication estimation for hardware/software codesign. In *Sixth International Workshop on Hardware/Software Codesign CODES/Cashe '98*, pages 55–59, Seattle, USA, March 1998.

[10] Y. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software.* Kluwer Academic Publishers, 1999.

[11] T. Lundquist and P. Stenström. Integrating path and timing analysis using instruction level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[12] Open SystemC Initiative. *SystemC.* http://www.systemc.org/.

[13] L. Semeria, K. Sato, and G. De Micheli. Resolution of dynamic memory allocation and pointers for the behavioral synthesis from c. In *Design Automation and Test in Europe DATE'00*, pages 312–319, Paris, France, March 2000.

[14] V. Tiwari, S. Malik, and A. Wolfe. Instruction level power analysis and optimisation of software. *The Journal of VLSI Signal Processing*, 13(2/3):223–238, August 1996.

[15] F. Wolf and R. Ernst. Data flow based cache prediction using local simulation. In *Proceedings of the IEEE High Level Design Validation and Test Workshop*, pages 155–160, Berkeley, USA, November 2000.

[16] F. Wolf and R. Ernst. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture, The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, 47(3-4):339–356, April 2001.

[17] F. Wolf, J. Kruse, and R. Ernst. Segment-wise timing and power measurement in software emulation. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference, Designers' Forum*, pages 165–169, Munich, Germany, March 2001.

[18] 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)*, pages 9–13, Seattle, USA, March 1998.

[19] 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.