Path Clustering in Software Timing Analysis

Fabian Wolf, Rolf Ernst, *Member, IEEE,* and Wei Ye

***Abstract*—Verification of program running time is essential in system design with real-time constraints. Simulation with incomplete test patterns or simple instruction counting are not appropriate for complex architectures. Software running times of embedded systems are process state and input data dependent. Formal analysis of such dependencies leads to software running time intervals rather than single values. These intervals depend on program properties, execution paths, and states of processes, as well as on the target architecture. An approach to analysis of process behavior using running time intervals is presented. It improves our previous work by exploiting program segments with single paths and by taking the execution context into account. The example of an asynchronous transfer mode (ATM) cell handler demonstrates significant improvements in analysis precision. Experimental results show the superiority of the presented approach over well-established approaches.**

***Index Terms*—Performance analysis, real-time systems, software timing estimation, system level design.**

I. Introduction

ACCURATE software running time analysis is key to optimized system design. In general, imprecise estimation of software running time increases the design risk or leads to inefficient designs. The necessity to consider running time intervals for the design and verification of embedded digital systems becomes evident when looking at the limits of software simulation. Profiling and simulation are current practice in industrial design but since exhaustive simulation is impractical for more complex applications, simulation results can only cover part of the system behavior. This leads to unknown coverage of worst and best cases. Verification is a more complicated but attractive alternative. It provides lower and upper bounds reflecting data dependent control flow as well as data dependent instruction execution. In the past, these bounds were very wide due to a lack of efficient control flow analysis and architecture modeling techniques. In recent years, there has been significant progress in both areas such that formal software execution cost analysis has become practical. Power consumption analysis can use very similar techniques. It is crucial for battery lifetime prediction of hand-held devices. We will, therefore, use the general term *execution cost* in the following text.

Execution cost intervals depend to a certain extend on the input data and the state of a process (if it contains internal states). Input data and state values can be combined to define a process execution context. In other words, execution cost intervals of a process are context dependent. Fig. 1 gives an example of a system of communicating processes. It shows a simplified set of processes implementing the wireless internet protocol (IP) standard on a pico-cellular base station [1]. The solid lines represent the paths on which different data packets are routed through the process network running on the base station. Important questions the system architect can ask are the power consumption for sending a data packet or the time to set up a connection in the base station. This should take the system context into account since, for each packet type and destination, the processes react with a different control flow. Of course, simulation is always possible and statistical power and timing analysis are feasible, but the first approach is not reliable and the second one is just a rough approximation of the complex hardware activities when executing the software of a base station. We present an analysis approach which works on the source code level. It provides reliable and narrow software execution cost intervals for context dependent process execution with a minimum of user interaction. It allows to explore different target architectures in a very flexible way.

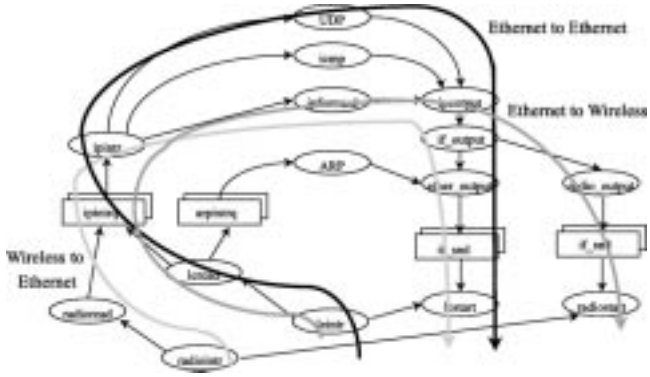


Fig. 1. Context dependent flow of execution in a base station.

We explain a new approach to execution cost and path analysis in Section II while the architecture modeling techniques are introduced in Section III. Experiments are presented in Section IV before we conclude in Section V.

II. Program Path Analysis

A. Process Representation

A system of communicating processes as shown in Fig. 1 is assumed. For simplicity, it is assumed that processes read data in the beginning and write data in the end. The system property intervals (SPI) model [2] is chosen as a system-level representation since it can consider process cost intervals.

B. Program Segments

In path analysis [3], a program is typically divided into basic blocks (BBs)[4]. Any well structured program can be partitioned into disjoint BBs. Then, the program structure is

Manuscript received April 7, 2000; revised November 3, 2000.

The authors are with the Institut für Datentechnik und Kommunikationsnetze, Technische Universität Braunschweig, Braunschweig D-38106, Germany (e-mail: {wolf; ernst; ye}@ida.ing.tu-bs.de).

Publisher Item Identifier S 1063-8210(01)08441-4.

represented as a directed control flow graph with BBs as nodes. For each BB, a cost interval with respect to each property, e.g., running time, can be determined. A longest and shortest path analysis on the control flow graph is then used to identify global intervals.

This procedure does not yet provide sufficient accuracy. For acceptable analysis precision one must identify all feasible paths through a program. A feasible program path or trace is a path in this flow graph corresponding to a possible sequence of BBs when the program is executed from the first to the last BB of a program.

Definition 1: A program segment (PS) *is a sequence of nodes in a control flow graph.*

This definition implies a hierarchy of program segments.

Definition 2: A program path segment (PPS) *is a program segment with at least one possible path.*

A false program path is a path which cannot be executed under any input condition. False path identification is essential for programs with loops since loops correspond to cycles in the graph, which can lead to an infinite number of potential paths and resulting infinite cost intervals.

C. Previous Work

The approaches by Mok [5], Puschner and Koza [6], and Park and Shaw [7] require iteration bounds for all loops in the program, which the user must provide by loop annotation. While making formal analysis feasible, loop bounding alone is not sufficient for accurate path analysis. In nested loops, conditions often depend on each other. These dependencies can be rather complex. As a second step in the approaches by Li and Malik [3] and Park and Shaw [7], the user is asked to annotate false paths. The number of false paths can be very large. The approach by Gong and Gajski [8] can partially consider false paths because the user can specify the branching probabilities. Instead of enumerating false paths or, conversely, feasible paths, a language for user annotation with regular expressions is introduced by Park and Shaw [7]. Still, the number of required path annotations can be extremely large in practice, as demonstrated with even small examples. A major step forward was the introduction of implicit path enumeration by Li and Malik [3].

This technique is based on the standard execution cost model for static analysis approaches, the sum-of-basic-blocks model (see, e.g., [9]). It can be used for timing as well as for power consumption analysis [10]. Let a program consist of N BBs with x_i execution count of BB bb_i and c_i execution cost (timing or power). Then, the *sum-of-basic-blocks* model defines for its execution cost interval

$$C = \sum_{i}^{N} c_i \times x_i.$$

In implicit path enumeration, the execution count x_i is constrained by linear equations or inequations. Structural constraints are derived from the program structure based on the fact that the execution count of a BB equals the execution count sum of all predecessors in the control flow. For each BB, the following is inserted:

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

Structural equations only encode the structure of a program, but not the feasible paths. Feasible paths have to be constrained by the designer who provides so-called functional constraints which are given as equations or inequations for a set of execution counts x_i [3] such as the relative frequency of "then" and "else" paths of an if statement. Together with the sum-of-basic-blocks cost function, these equations and inequations form an optimization problem. This problem is solved for the minimum and for the maximum cost value using an integer linear programming (ILP) solver. It provides a lower execution cost bound and an upper execution cost bound.

Theiling *et al.* [11] use this approach. They additionally apply abstract interpretation to the static prediction of cache and pipeline behavior. Gustafsson and Ermedahl use abstract interpretation to reduce excessive designer interaction for loop bounding [12]. A related approach is proposed by Healy *et al.* [13].

D. Execution Cost Model

The sum-of-basic-blocks model used in previous work is based on the execution count x_i and the execution cost per BB c_i. The execution cost can be determined by adding up the running time for each instruction in a BB, possibly with upper and lower bounds in case of data dependent running time, e.g., for multiplication. This instruction cost addition (ICA) approach leads to wide intervals in case of super scalar or pipelined architectures. Instruction execution overlap has been considered in recent work [14] modeling pipelined execution when adding up the instructions. It is still assumed that all executions of one BB take the same time.

Precise modeling of individual BBs only solves part of the problem since pipelining and superscalar execution extend over BB borders such that the running time depends on the program path through a sequence of BBs. Previous approaches must use very pessimistic intervals to be correct for all executions of one BB because empty pipelines have to be assumed, even when loop bounds are automatically determined. In [15], overlap blocks have been inserted between any two adjacent blocks. In this case, the sum-of-basic-blocks equation must be extended by the block transition frequency and cost [15]. This works for short pipelines, but increases analysis complexity.

To obtain higher precision, the analysis should be extended from BBs to longer program segments consisting of sequences of BBs. Such an approach leaves us with the following two problems:

1) the identification of suitable longer program segments;
2) the extension of the cost model to program segments.

We propose a coherent approach that covers both problem domains for very different architectures such as the StrongARM, the SPARC, or the 8051. We will show that it provides substantial precision improvements over previous work and, at the same time, reduces the number of functional constraints to be defined by the user.

We did not discuss the influence of caches. In general, cache analysis working on BBs such as [3] can be extended to longer program segments. The advantage is a reduced number of nodes

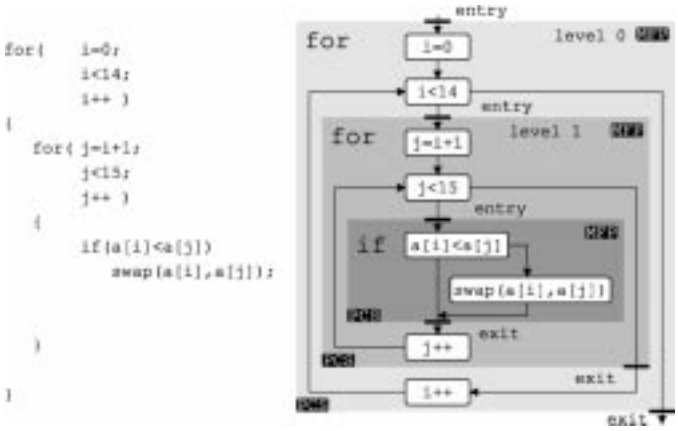
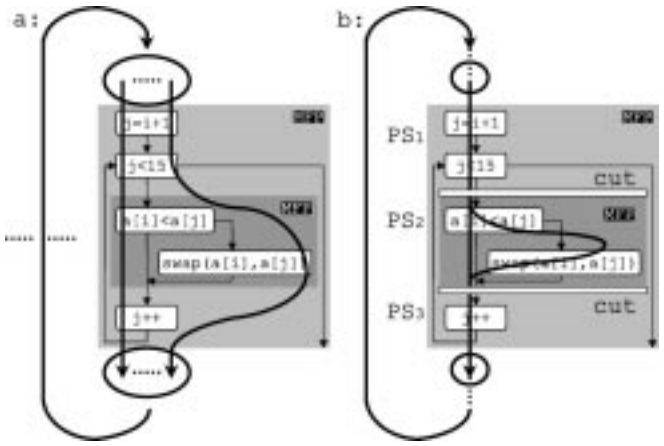


Fig. 2. Flow graph with a control structure hierarchy.



(a) (b)

Fig. 3. (a) Paths of bubble sort. (b) Separation of the `if` construct.

to be considered which has a drastic effect on analysis time [16]. Detailed investigations are currently being done [17].

E. Basic Path Classification

Program properties can be exploited to simplify path analysis for the determination of the execution cost through BB sequences. Large parts of typical embedded system programs have a single program path only. An finite impulse response (FIR) filter is a simple example and a fast Fourier transform (FFT) is a more complex one. There is only one path executed for any input pattern, even though this path may include many loops, conditional statements, and even function calls that are used for program structuring and compacting.

Definition 3: A program segment has a single feasible path (SFP) *when paths through the program segment do not depend on input data.*

A program segment with an SFP is an SFP segment. Previous 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, they may be accurate but require much designer interaction for SFP segments and still do not deliver the sequence-of-basic-blocks such as [3]. In case of SFP, execution chooses the one correct path and sequence for any input pattern without designer interaction. Most practical systems also contain non-SFP parts. These have multiple feasible paths (MFPs).

Definition 4: A program segment has MFPs *when paths through the program segment depend on input data.*

A program segment with MFP is an MFP segment. Isolation of SFP and MFP segments helps to exploit SFP by finding SFP and MFP nodes in the control flow graph. Embedded MFP are cut out and analyzed separately using ILP solving [3]. SFP are analyzed by simulating the running time or power consumption of the single path.

F. SFP Identification and Path Clustering

1) Hierarchical Flow Graph: For partitioning of SFP and MFP segments, the input program is mapped to a *hierarchical control flow graph* (HCFG) like the bubble sort example in Fig. 2. In this control flow graph, every control structure, such as `if` and `for` is a hierarchical node. Its associated BBs or hierarchical nodes on lower levels are dependent nodes, which refer to exactly one control structure. Each of the control constructs has an associated condition that decides which of the paths is executed. Control structures require BB nodes as well because an execution leading to a BB and, therefore, a node may be necessary for the evaluation of a condition, e.g., "$i < 14$." As in [3], structured programs are assumed so the HCFG can model the hierarchy of the program control segments.

Definition 5: A program control segment (PCS) *is a program segment with exactly one control structure.*

Control flow can only enter or leave the PCS at the current hierarchy level with its associated control structure so SFP and MFP segments must be disjoint. The shaded areas in Fig. 2 are the PCS with the associated BB and lower level hierarchical nodes of this example. Each control structure of the PCS as well as its nodes are classified as being either SFP or MFP at this stage.

2) SFP Identification: A depth first search algorithm on the flow graph using symbolic simulation of BBs [9] can be used to determine input data dependencies of conditions. Every control structure, which does not contain an input data dependent condition must be SFP. Leaf nodes are SFP by definition. If adjacent PCS on one level of hierarchy or child PCS are classified as SFP, they are joined to achieve longer sequences. If conditions contain input data, or symbolic execution is not successful due to the complexity of symbolic expansions, the flow graph nodes are classified as MFP. It only means that different methods for the determination of execution cost have to be applied leading to wider cost intervals as explained in Section II-C. This algorithm assigns a classification to each hierarchical node. PCS with MFP child nodes are classified as MFP because the multiple paths also enter and leave this hierarchical node when their control structure is independent of input data.

3) SFP Clustering: A conservative analysis assumes that the program paths branch at the `for` and the `if` statements [3] such that all the corresponding program control segments have the MFP property. In Fig. 3(a), two possible paths for every iteration of the loop can be seen, one of which is being taken for every iteration. When the condition in the `if` statement is evaluated, it can be recognized that values in $a[]$ are not known, meaning two potential paths for every loop iteration leading to $2^{\text{loop iterations}}$ potential paths through the program. The first major step is to

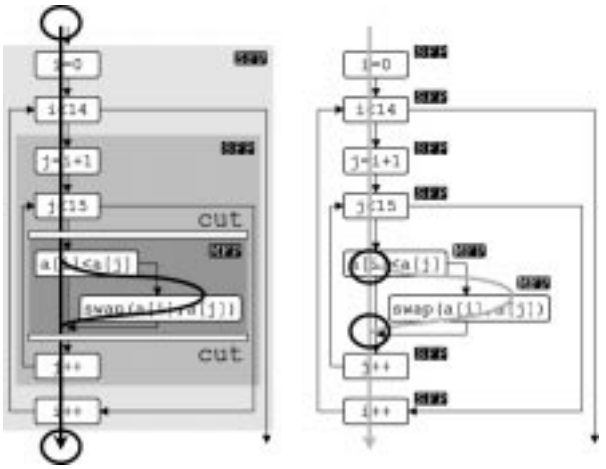


Fig. 4. Node classifications follow program control segments.

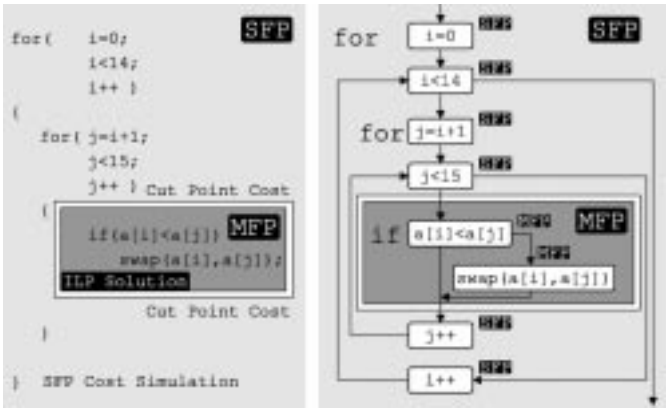


Fig. 5. Single feasible path with embedded multiple feasible path.

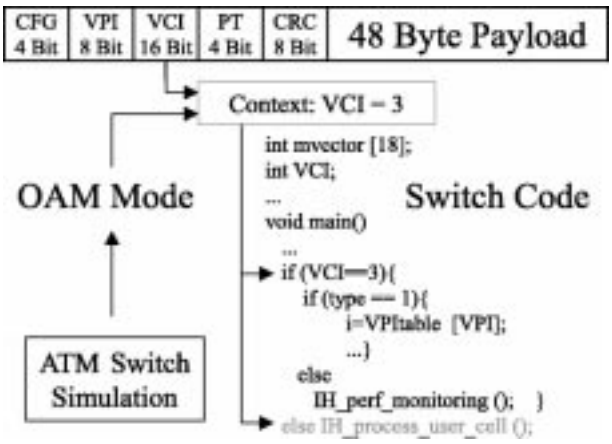


Fig. 6. Path selecting property of the OAM process mode.

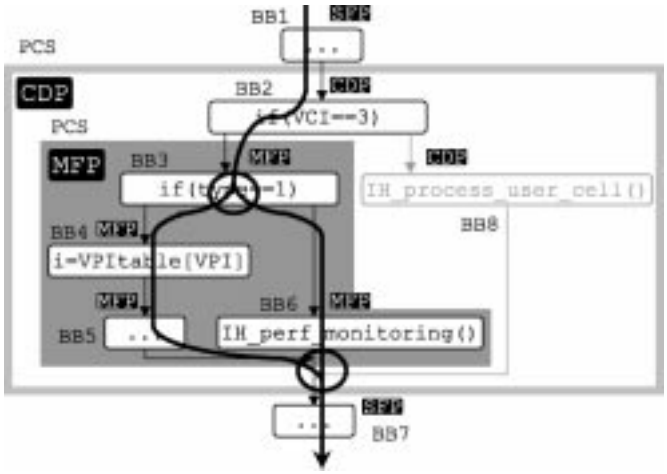


Fig. 7. PCS in the HCFG of the ATM switch component.

split the program into segments, the `if` construct and the rest. The two paths through PS_2 are now considered to be merged into a single MFP segment. As a consequence, the paths of Fig. 3(a) are merged into a single path through the segments PS_1 and PS_3 in Fig. 3(b), which winds around the two fixed and, hence, input data independent loops in Fig. 4.

In other terms, PS_1 and PS_3 become an SFP segment (SFP–PCS) which includes the MFP segment PS_2 (MFP–PCS). The MFP segment is isolated in the graph and then analyzed as a separate graph. The remaining SFP segment is analyzed using one of the cost models discussed before. The isolated MFP segment is now analyzed in the same way. This continues until we finally reach SFPs, at least at the level of BBs which are SFP by definition. For each isolated PCS, the execution cost is calculated.

In Fig. 5, only two subgraphs will remain, an MFP–PCS consisting of the condition BB, the comparison and the `swap`, and a single SFP segment consisting of the loops. The cut points contain the conservative overheads for merging different entry and exit paths each. In [18], we give a more formal algorithm and show that the resulting set of PCSs can be used to compute the execution cost with the same ILP approach as for BBs. In other words, we can apply the same implicit path enumeration technique, but on much larger and fewer blocks, namely PPS, than in the case of individual BBs.

G. Context Dependent Control Flow

The analysis quality can further be improved. In the introduction, we have argued that the designer is often interested in a context dependent process behavior. Here, context is defined to be a subset of input data and/or a subset of possible process states, often called process modes. 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 analysis. Global process representation models [2], [19] can support process modes such that the distinguishable contexts are known for cost analysis. A simple example for context dependent control flow that increases analysis precision in an ATM switch component is given.

One function of the ATM switch is to identify some of the cells in the data cell stream as so-called operation administration and maintenance cells (OAM), which control the ATM connection [20]. These cells do not carry user data so they are irrelevant for data transmission. Fig. 6 shows a code fragment to handle the OAM component of the switch. The control flow graph is shown in Fig. 7. In this "OAM mode," the shaded `else` program segment in Fig. 6 cannot be reached. It should not be included in further analysis of the OAM mode while in the "USER mode" only the `else` path is executed. For a given context, the

`if` node BB2 has a single path only. In other words, the contexts "VCI $= 3$" corresponding to the OAM mode and "not (VCI $= 3$)" corresponding to the USER mode turn an MFP-PCS into a PCS with a single path.

Definition 6: A context dependent path (CDP) *is a path through a PCS with control structures which only depend on context dependent input data.*

For analysis of the given context, the CDP is treated like an SFP–PCS. Where this approach is not applicable, the reduced path set of a given context can further be exploited via additional structural and functional constraints [3]. In both cases, context dependent behavior can be analyzed using the same techniques as described before. CDP segments are only found in segments formerly defined as MFP. At the transitions between SFP and CDP segments, PPS containing both SFP–PCS and CDP–PCS can be defined.

In general, the different contexts can lead to different CDPs and, therefore, a different set of equations. So, the ILP analysis for upper and lower bound must be repeated for each context. Currently values for contexts are read from annotation files that are obtained by using process modeling on the system level [2], [19].

The path analysis approach presented up to this point is target architecture independent as this can only influence local PPS cost. It is a general approach that improves state-of-the-art path analysis by an automatic detection of program properties and the consideration of process modes.

III. Architecture Modeling

The execution cost for an instruction, BB, SFP or PPS can be determined by simulation for a target architecture using one of the following two techniques.

A. Instruction Cost Addition

Instruction cost addition (ICA) uses a generalization of the standard sum-of-basic-block approach to calculate the execution cost of PPS consisting of SFP–PCS, CDP–PCS and BBs. For this purpose, the PPS just needs to be executed on a host system to derive the execution count for all BBs in the PPS. Since there is only a single path through the PPS, these instruction counts are unique. A sum-of-basic-blocks calculation provides the total cost of one PPS execution by using a cross compiler and instruction cost tables. This leads to accurate results for simple architectures without overlapping BBs effects, e.g., caused by pipelines or caches.

The main advantage compared to previous work is that there are no functional constraints required for SFP–PPS, alleviating the user in the error prone and tedious task of functional constraint definition. The user will only provide functional constraints for the remaining MFP. The experiments will show, however, that this is hardly ever necessary.

ICA faces the same issues for cache analysis and data dependent instruction execution times as standard basic-block-based path enumeration and the same solutions apply. If we use a cache tracing tool [21] with the target cache model when running the PPS on the host system, then even the instruction cache is modeled correctly within the PPS and can be used in cache analysis [3], [17]. This leads to a significantly smaller problem size since the number of PPS containing SFP and BBs is much smaller than the number of BBs as we can see in Section IV.

B. Program Segment Simulation

An alternative approach to execution cost analysis is to execute the PPS either by simulation or on the target architecture. Since the PPS execution path is fixed like in a BB, the execution costs are unique such that simulation with a single input data set is sufficient to determine the execution costs. This approach can consider overlapping BB effects. A conservative overhead is added to cover the worst case of all different entry paths into the PPS, which can represent different states for register allocation, pipelines, and caches. Greater overheads are needed to cover conservative PPS simulation startup for superscalar execution or branch prediction.

This overhead is also needed for single BBs so SFP improve analysis precision because the BBs are extended to PPS and intermediate overheads can be removed as the entry edge is known. As discussed before, target architecture simulation or execution can provide much higher precision since it correctly models the architecture. In case of data dependent instruction execution times, the result must be corrected for each execution of a data dependent instruction to obtain the correct cost interval.

Caches can be treated in the same way as in the ICA approach, either using a cache tracing tool or the target system cache model. An overhead for the PPS start assuming first misses is included. Both ICA and PSS require program execution. The input patterns must be selected such that all PPS are executed at least once. In a reasonable program test, the test patterns should have this property. Here, it does not matter if a program must be executed several times to reach all PPS since only a single PPS execution is regarded. When a PPS is not reached in simulation it is extracted and simulated separately. The pretty complex test environment which includes a flexible interface for off-the-shelf simulators and evaluation kits is presented in [22].

C. Software Power Consumption

The software power consumption of a PPS can be simulated using a simplification of the methodology presented in [10]. It proposes an ICA approach with base and transition energy values for a sequence of instructions given by host simulation. For reduced instruction set computer (RISC) architectures, experiments show that the simulation matches the measured power consumption. Influences of data values or cache behavior can be modeled via additional processor cycles that add to the instruction energy consumption. Details about recent trace-based or higher level software power analysis approaches are considered to be beyond the scope of this work. Software power analysis using such a recent approach can be integrated into our tool suite.

SFP identification improves software power analysis by removing conservative overhead assumptions like empty caches for BB beginnings and the resulting number of bus cycles for the misses. So it has a major impact on the global analysis of software power consumption because potentially dominant overheads are removed.

D. Cycle True SPARC Simulation

A cycle true processor model for a 32 bit superscalar SPARC RISC processor with four stage pipeline and floating point operations has been introduced in [15]. It implements PSS as explained before and is used in the experiments in Section IV. A GNU compiler translates PPS source code to assembly code for the simulator. Local cache simulation for PPS can be included by using [21] for the given address traces when a PPS is executed.

E. Instruction Cost Addition for i8051

The ICA approach has been implemented for a simple Intel eight bit 8051 processor. ICA is well suited for this architecture as no caches or pipelines are present. The sum-of-basic-blocks model delivers accurate results because no overlapping basic-block execution due to sophisticated architecture properties is present. We also do not encounter data dependent execution times. Such processors are widely used in microcontrol systems. A commercial cross compiler on a PC delivers the assembly code of the PPS while debug information is used to identify BBs in the assembly code. Profiling is running on a workstation that sets up a connection to the PC running the compiler. Results generated by ICA have been compared to the results of the commercial PSS simulator showing high precision for the ICA approach.

F. Cost Simulation Approach for StrongARM

As an example for PSS a StrongARM simulator core has been combined with the DINERO simulator [21] delivering both instruction and data cache behavior. Both source codes have been recompiled to one simulator to achieve better performance. The StrongARM cross compiler and the simulator source code have been given by Cygnus. Architecture modeling regarding timing has been derived from [23] while the energy dissipation model has been taken from [24] and [10]. The results for a PPS regarding timing and power are already intervals because data dependent instruction execution can be present. Cache simulation starts from both first hit or miss for the interval, representing the PPS overhead. Instruction cost tables and host tracing are also available to implement an ICA approach, which is, of course, less accurate due to the overlapping BB effects on a StrongARM RISC processor. The complete StrongARM tool suite can be used for simulation with test patterns given by the designer as well as for the simulation of an instrumented program.

G. Prototyping Approach for SPARClite

Processor simulators for the determination of the PPS execution cost are often slow, inaccurate or even not available. So the possibility of using commercial evaluation kits has been investigated. Details presented in [25] are out of scope of the presented approach while the results for the PPS are used. A Cygnus cross compiler translates PPS to assembly code. Timing and power consumption of a PPS are measured on a commercial SPARClite evaluation kit by inserting trigger points at PPS starts and ends. A trigger point is implemented by a store of the source code line number information to a defined trigger address in a noncached part of the memory space. The extension from BBs to PPS significantly improves measurement precision because the instrumentation overhead caused by trigger points is reduced. Trigger points are detected by a logic state analyzer and saved with a timestamp. Any commercial evaluation kit can be used for this purpose. Automatic download and measurement abstracts the evaluation kit to the same level as a software simulator. The approach is considered to be an enabling technology for static software analysis.

TABLE I
EXPERIMENTAL RESULTS FOR PATH CLUSTERING

Benchmark	CFG Nodes	PPS Nodes	Reduction	C Lines
3D-image	94	10	89%	164
diesel	65	1	98%	160
fft	78	1	99%	145
bsort	14	7	50%	25
smooth	48	10	79%	86
blue	80	28	65%	127
check-data	18	18	0%	44
whetstone	122	1	99%	251
line	101	83	18%	250

IV. EXPERIMENTS

The methodologies have been implemented in a tool suite symbolic timing analysis (SYMTA) that has been applied in a variety of experiments.

A. Path Clustering

In a first experiment, the reduction in analysis complexity through path clustering by SFP identification has been investigated. The first column in Table I shows experiments for benchmarks taken from [3] and [9]. The second column shows the number of BB and control nodes in the control flow graph which is reduced to the number of PPS including SFP–PCS and BBs in the third column. The fourth column gives the reduction of the complexity of the graph while the fifth column gives the number of lines in the source code. We have only analyzed the SFP and MFP properties of the graph, no execution cost or context dependency has been determined in this experiment.

The experiment reveals that many parts of programs contain SFP segments and can be clustered to PPS which can precisely be analyzed using simulation (PSS). We notice that through the identification of many SFP within the loops like in the "diesel" or the "FFT" benchmark the graphs lose most of their complexity. When only one PPS remains, the complete cost can be determined by simulation.

B. Timing Bound Analysis

In the following experiment, the timing bounds have been analyzed using SYMTA and have been compared to the results of accurate simulation. For this reason, we have selected programs where best-case and worst-case input data can be determined by hand to deliver the real bounds.

In Table II, only SFP identification without CDP identification or mode annotation has been applied for now. Program segments were simulated using a cycle true SPARC simulator to demonstrate PSS and an i8051 simulator to demonstrate ICA. No caches have been assumed for SPARC. For i8051, results

TABLE II
EXPERIMENTAL RESULTS USING SFP IDENTIFICATION ONLY

	Measured Bounds		Analyzed Bounds	
Benchmark	c_{min}	c_{max}	c_{min}	c_{max}
SPARC clock cycles using PSS				
3D-image	34908	37848	33874	38037
diesel	62944	62994	61445	63333
fft	1498817	1499176	1494650	1499290
bsort	4423	8938	4416	8938
smooth	3635651	4846511	3570227	4881135
blue	3564938	316865761	3345041	346541760
check-data	80	431	65	435
whetstone	2928459	3369459	2880230	3378098
line	514	1619	381	2035
8051 inst-cycles using ICA, one inst-cycle equals 12 clock cycles				
fft	26421460	26421460	26419338	26488288
bsort	9347	15045	7804	18167
smooth	9737378	9737516	9737469	9737522
check-data	68	559	63	588

are given in instruction cycles one of which consists of 12 clock cycles. The first column shows the benchmark under investigation followed by the measured cost bounds which are followed by the analyzed cost bounds. Analyzed bounds are tight while staying conservative with respect to simulated bounds.

C. Improvements to Previous Work

The state-of-the-art approach presented in [3] and the current approach are compared by their estimation errors in Table III. In the first three columns, estimation errors for SYMTA using PSS for SPARC are given followed by estimation errors for SYMTA using ICA for i8051. In the last four columns, results from [3] for i960 including estimation errors and the number of functional constraints given by the designer are presented. Caches have not been considered in this experiment. They would lead to much higher BB and PPS simulation startup overheads. No functional constraints are needed for the SYMTA approach, only program properties are exploited by SFP identification. Given functional constraints for the approach from [3] are an optimal selection which is hard to find for the designer. Estimation errors η are calculated for the upper bounds followed by the lower bounds in the second part of the table.

The SYMTA approach leads to tight estimation bounds without any functional constraints even for the MFP segments. It only exploits the program structure that implies most of the functional constraints. The designer is not burdened with error prone implicit path enumeration. For most benchmarks, estimated bounds are tighter than bounds delivered by an optimal selection of functional constraints because overlapping BB execution can exactly be modeled for SFP segments using target architecture simulation.

D. Case Study: OAM Component

The source code [20] of the top level of an ATM F4 implementation has been investigated with the proposed methodology. Two modes for this process exist. In the USER mode, the ATM cells are simply forwarded to the switch. In the OAM indicated by a special virtual channel identifier (VCI) in the cell, no user data is processed but special administration functions are triggered. In this example, the OAM mode implies context dependent input data for the VCI.

In Table IV, the different analysis approaches are evaluated by their worst-case bounds with respect to running time or power consumption for the given architectures as only worst-case assumptions have been used for program segment cost determination. The results are given for the OAM mode of the component, the switch itself has not been included and the USER mode has not been investigated. Architecture modeling has been done using StrongARM simulation, SPARClite software emulation and measurement as well as SPARC simulation. For all processors, core speeds of 80 MHz, bus speeds of 40 MHz and memory cycle times of 25 ns have been assumed to get comparable results. Caches have been intentionally switched off for program segment cost determination in this experiment.

In the first line, results have been determined with the methodology proposed by Li and Malik in [3] that is based on the analysis granularity of BBs using the sum-of-basic-blocks execution cost model. The control flow defined by the OAM mode has been annotated using a functional constraint for the according control structure. In the second line, the methodology basing on SFP clusters (SFP) [9] has been applied. The OAM mode has been annotated using a functional constraint, too. In the third line, the context sensitive methodology (CDP+SFP) has been applied in addition to SFP identification. The OAM mode can be considered without using functional constraints because it defines context dependent input data for the control structure. In the last line, the result for simulation with given worst case data as a reference is shown.

For the analysis of the OAM mode, SFP clustering delivers tighter bounds than the BB-based approach because original pipeline behavior for the SFP can be modeled. The context sensitive approach delivers even tighter bounds. This is caused by the possibility to consider the BB sequence across the context dependent control structure where the context dependent input data is defined by the OAM mode.

E. Case Study: Filter on Packet Data

The software execution cost analysis approach has been applied to a single process, which reads a packet and loads a picture. If the picture is addressed to the system component under investigation, it performs an “unlikely dot” filter on the picture data and sends it to another buffer. The significant parts of the source code are given in Fig. 8. The relevant program segments are marked using their original C source-code line numbers.

Different potentially context dependent control structures are present. For the program segment in line 89, the loop bounds depend on the number of pixels that can be context dependent. These context dependent input data can be selected by a process mode, e.g., the processing of a “large” or a “small” picture. If there is no such information for an execution of the process, loop bounds assuming designer knowledge about packet and picture size are annotated as functional constraints. They avoid an infinite cost interval for the resulting multiple feasible paths requiring BB-based analysis [3]. The same discussion applies to the program segment in line 124.

For the program segment in line 122, input data for the address can be context dependent when the address is known for a

TABLE III
COMPARISON OF SYMTA WITH LI AND MALIK BY IMPRECISION η

Benchmark	SYMTA-Approach SPARC(cycles)			SYMTA-Approach 8051(instruction cycles)			ILP with (in)equations i960 (cycles)			
Upper Bounds	c_{max}^{exact}	$c_{max}^{analyzed}$	η	c_{max}^{exact}	$c_{max}^{analyzed}$	η	c_{max}^{exact}	$c_{max}^{analyzed}$	η	fun const [3]
bsort	8938	8938	0.0%	15045	18167	20.75%	9.99e6	27.8e6	179%	6
fft	1499176	1499290	0.01%	26421460	26488288	0.25%	2.20e6	2.63e6	19%	11
check-data	431	435	0.93%	559	588	5.20%	430	471	10%	10
whetstone	3369459	3378098	0.26%	n.e.	n.e.	n.e.	6.93e6	10.5e6	52%	14
line	1619	2035	20.44%	n.e.	n.e.	n.e.	4836	6088	26%	2
Lower Bounds	c_{min}^{exact}	$c_{min}^{analyzed}$	η	c_{min}^{exact}	$c_{min}^{analyzed}$	η	c_{min}^{exact}	$c_{min}^{analyzed}$	η	fun const [3]
bsort	4423	4423	0.0%	9347	7804	16.51%	16942	13965	18%	6
fft	1498817	1494650	0.03%	26421460	26419338	0.08%	1.72e6	1.59e6	8%	11
check-data	80	65	18.8%	68	63	7.35%	35	35	0%	10
whetstone	2928459	2880230	1.65%	n.e.	n.e.	n.e.	6.94e6	5.97e6	14%	14
line	514	381	25.9%	n.e.	n.e.	n.e.	929	776	17%	2

n.e.: the code is not executable on the i8051 architecture due to a missing floating point unit and other restrictions
η(%): Imprecision = abs($c_{max}^{analyzed}$ - c_{max}^{exact}) / c_{max}^{exact} × 100% deviation for BCET and WCET

TABLE IV
UPPER EXECUTION COST BOUNDS FOR THE OAM MODE

Approach	SPARC	StrongARM	StrongARM	Sparclite	Sparclite
BB	1316 ns	11986 ns	1282 nWs	18.4 μs	26.5 μWs
SFP	1303 ns	11836 ns	1261 nWs	18.1 μs	26.1 μWs
CDP+SFP	1164 ns	9505 ns	911 nWs	14.9 μs	17.8 μWs
Exact	1128 ns	9471 ns	903 nWs	13.7 μs	16.4 μWs

```
/* Pseudo code of a packet receiver with filtering  */
89:  header = receive(INPUT, HEADER_SIZE);
     for all pixels                          /* Picture size */
         picture[y][x] = receive(INPUT, 1);
122: if(address == MY_ADDRESS){             /* Address match */
124:     for all pixels{
             for a 3*3 pixel window{
143:             if(without_center)  /* center calculation */
                     average = sum/8;
                 else average = sum/9;
             }
151:         if(abs(picture[y][x]-average)>threshold)
                  send(OUTPUT, average, 1);
             else send(OUTPUT, picture[y][x], 1);
         }
     }
```

Fig. 8. Pseudocode of the packet data filter.

mode. Two possibilities are a packet for another destination or an address match. No information about modes results in multiple feasible paths requiring BB-based analysis without further functional constraints for this control structure. For the program segment in line 143, these possibilities are the calculation of the average luminance including or excluding the center pixel.

For PSS, the StrongARM processor simulator with 80-MHz core frequency, 40-MHz bus frequency, and 25-ns memory cycle time including local cache simulation has been applied. Conservative first miss/hit scenarios have been assumed for local simulation. Communicated data, i.e., the number of sent and received bytes is delivered by the sum-of-basic-blocks model and the amount of data communicated by an instruction.

In Table V, execution cost intervals without any mode annotation or a resulting identification of context dependent control flow are given, only the identification of SFP program segments

TABLE V
COST INTERVALS $[c_{i,\min}, c_{i,\max}]$ WITHOUT MODES OR ANNOTATION

Line	PCS	Timing [ms]	Energy [mWs]	Sent kb	Rec. kb
89	MFP	[4.92,38.0]	[2.0,8.5]	[0,0]	[6.2,25.0]
122	MFP	[413ns,2475ns]	[50nWs,178nWs]	[0,0]	[0,0]
124	MFP	[39.5,329]	[17.5,72.6]	[0,0]	[0,0]
143	MFP	[1.54,131]	[0.65,14.7]	[0,0]	[0,0]
151	MFP	[16.7,182]	[2.85,20.4]	[0,24.4]	[0,0]
SFP	-	[4.955,680.8]	[2.099,116.2]	[0,24.4]	[6.2,25.0]
BB	-	[2.773,6368.0]	[1.855,582.2]	[0,24.4]	[6.2,25.0]

has been done. The first column shows the beginnings of relevant PCS identified by their first line numbers in the code; the second column shows their classifications. As the SFP program segments are not displayed in this table and no modes have been considered up to this point, only MFP classifications are present. The next four columns give the cost intervals with respect to timing, power consumption of the processor core and communicated data. The last but one line (SFP) shows the overall process cost intervals for this approach including SFP identification while the last line (BB) shows the results without SFP identification using the granularity of BBs.

Due to the loop bounds given by functional constraint annotation, the minimum, and maximum numbers of pixels are known but the PCS in line 124 stays an MFP segment. Intervals are wide because worst cases imply cache misses for the beginning of the segment while best cases imply cache hits to deliver PSS startup overhead. Without using SFP identification, intervals are far wider because of the BB analysis overheads in the nested loops.

In Table VI, a process mode has been annotated. For this mode, an "address match" is considered which leads to context dependent input data for the PCS in line 122. The picture size is "large," which leads to context dependent input data being the loop bounds in the PCS in line 89 and 124. The calculation of the luminance is done "with" the center pixel leading to context dependent input data for the program segment in line 143. All the according PCS become CDP–PCS because the context dependent input data is defined by the process mode. The

TABLE VI
COST INTERVALS $[c_{i,\min}, c_{i,\max}]$ WITH MODE ANNOTATION

Line	PCS	Timing [ms]	Energy [mWs]	Sent kb	Rec. kb
89	CDP	[19.2,38.0]	[8.4,8.5]	[0,0]	[25.0,25.0]
122	CDP	[164,329]	[72.6,72.6]	[0,0]	[0,0]
143	CDP	[15.7,22.6]	[4.4,5.00]	[0,0]	[0,0]
151	MFP	[64.9,182]	[11.8,20.3]	[24.4,24.4]	[0,0]
Mode	-	[264.60,572.01]	[97.31,106.51]	[24.4,24.4]	[25.0,25.0]
SFP	-	[4.955,680.8]	[2.099,116.2]	[0,24.4]	[6.2,25.0]
BB	-	[2.773,6368.0]	[1.855,582,2]	[0,24.4]	[6.2,25.0]

TABLE VII
COST INTERVALS $[c_{i,\min}, c_{i,\max}]$ WITH DIFFERENT PROCESS MODES

Modes	Timing [ms]	Energy [mWs]	Sent kb	Rec. kb
SFP	[4.955,680.8]	[2.099,116.2]	[0,24.4]	[6.2,25.0]
BB	[2.773,6368.0]	[1.855,582,2]	[0,24.4]	[6.2,25.0]
Small Picture	[4.955,66.71]	[2.099,24.61]	[0,5.9]	[6.2,6.2]
Large Picture	[19.24,575.1]	[8.474,107.5]	[0,24.4]	[25.0,25.0]
Address match	[38.49,660.2]	[21.03,112.1]	[5.9,24.4]	[6.2,25.0]
Small+match	[38.49,63.62]	[21.03,23.61]	[5.9,5.9]	[6.2,6.2]
Large+match	[264.6,572.0]	[97.3,106.5]	[24.4,24.4]	[25.0,25.0]

CDP–PCS in line 124 is clustered to a PPS with the CDP–PCS in line 122 and the SFP–PCS for the pixel window. All PCS except the MFP–PCS in line 151 can be clustered to PPS allowing execution cost determination by local simulation. This leads to tighter intervals for the mode because the execution path through the filter is known as well as the loop bounds for the picture leading to CDP–PCS. The only MFP–PCS is caused by the nested control structure depending on picture data. The results for the SFP approach and the BB-based approach are still included as a reference.

In Table VII, different scenarios for disjunct process modes have been explored. These lead to different context dependent input data for the control structures under investigation. The first column shows the given mode in which the process is executed. The next four columns show the behavioral intervals with respect to running time, energy consumption and communicated data.

Compared to the BB-based analysis approach and the approach using SFP identification without considering process modes, it can be noticed that modes deliver tighter specific intervals because of a more accurate path analysis. Even for worst-case modes, intervals are tightened because the control flow can be predicted for the corresponding control structures. Thus, overheads at the transitions between program segments can be removed. The remaining inaccuracy is caused by the remaining MFP–PCS in the loop nest, where a conservative first hit/miss scenario for the cache has to be assumed.

V. CONCLUSION

A static approach to software running time and power consumption analysis has been presented. It is an extension to the well known sum-of-basic-blocks approach with implicit path enumeration. The most important result is the transition from BB analysis to the analysis of complete program segments with a single execution path. This can be either input data independent or context dependent. The general path analysis approach is target architecture independent. We have presented techniques for running time and power analysis of such program segments. Experiments on a variety of different processor architectures and different environments demonstrate a significantly higher precision and far less designer interaction than in previous approaches. They emphasize the superiority of our approach exploiting program properties and process modes.

REFERENCES

[1] J. Liu, G. Maguire, M. Mateescu, A. Schmidt, and R. Ruppelt, "Document of network architecture strategies and tradeoffs," Esprit Media , Stockholm, Sweden, 1999.

[2] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele, "Combining multiple models of computation for scheduling and allocation," in *Proc. 6th Int. Workshop Hardware/Software Codesign* , Seattle, WA, Mar. 1998, pp. 9–13.

[3] Y. Li and S. Malik, *Performance Analysis of Real-Time Embedded Software*. Norwell, MA: Kluwer, 1999.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques and Tools. Reading, MA: Addison-Wesley, 1988, to be published.

[5] A. Mok, "Evaluating tight execution time bounds of programs by annotations," in *Proc. Workshop Real Time Operating Systems and Software*, Pittsburgh, PA, 1989, pp. 74–80.

[6] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *J. Real-Time Syst.*, vol. 1, no. 2, pp. 160–176, 1989.

[7] C. Y. Park and A. C. Shaw, "Experiments with a program timing tool based on source-level timing scheme," in *Proc. 11th IEEE Real-Time System Symp.*, Orlando, FL, 1990, pp. 72–81.

[8] J. Gong, D. Gajski, and S. Narayan, "Software execution from executable specification," *J. Computer and Software Engineering*, vol. 2, no. 3, pp. 239–258, 1994.

[9] W. Ye and R. Ernst, "Embedded program timing analysis based on path clustering and architecture classification," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD '97)*, CA, 1997, pp. 598–604.

[10] V. Tiwari, S. Malik, and A. Wolfe, "Instruction level power analysis and optimization of software," *J. VLSI Signal Processing*, vol. 13, no. 2/3, pp. 223–238, Aug. 1996.

[11] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separate cache and path analyses," *J. Real-Time Syst.*, vol. 18, no. 2/3, May 2000.

[12] J. Gustafsson and A. Ermedahl, "Automatic derivation of path and loop annotations in object-oriented real-time programs," *J. Parallel Distributed Computing Practices*, vol. 1, no. 2, June 1998.

[13] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. v. Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *J. Real-Time Syst.—Special Issue Worst-Case Execution-Time Analysis*, pp. 129–158, May 2000.

[14] T. M. Conte and C. E. Gimarc, *Fast Simulation of Computer Architectures*. Norwell, MA: Kluwer, 1995.

[15] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software cosynthesis," in *Proc. IEEE Int. Conf. Computer Design (ICCD '93)*, Cambridge, MA, 1993, pp. 452–457.

[16] A. Hergenhan and W. Rosenstiel, "Static timing analysis of embedded software on advanced processor architectures," in *Proc. Design, Automation, Test Eur. (DATE '00)*, Paris, France, Mar. 2000, pp. 552–559.

[17] F. Wolf and R. Ernst, "Data flow based cache prediction using local simulation," in *Proc. High-Level Design Validation and Test Workshop*, Berkeley , CA, 2000, pp. 155–160.

[18] ——, "Execution cost interval refinement in static software analysis," *J. Syst. Architecture—Special Issue Modern Methods Tools Digital Syst. Design*, vol. 47, no. 3-4, pp. 339–356, Apr. 2001.

[19] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst, "Interval-based analysis of software processes," in *Proc. ACM Workshop Languages, Compilers and Tools for Embedded Systems*, Snowbird, UT, June 2001, pp. 94–101.

[20] A. Doboli, J. Hallberg, and P. Eles, "A simulation model for the OAM functionality in ATM switches," Linköping, Sweden, 1995.

[21] M. Hill, "DINERO III cache simulator: Source code, libraries, and documentation,", www.ece.cmu.edu/ ece548/tools/dinero/src/, 1998.

[22] F. Wolf and R. Ernst, "Software timing and power estimation of telecom systems," ESPRIT MEDIA Rep., Tech. Univ. Braunschweig , Germany, 1999.

[23] S. Furber, *ARM System Architecture*. Reading, MA: Addison-Wesley, 1996.

[24] J. Montanaro, "A 160-MHz, 32-b, 0.5W CMOS RISC microprocessor," *The IEEE J. Solid State Circuits*, vol. 31, pp. 1703–1714, Nov. 1996.
[25] F. Wolf, J. Kruse, and R. Ernst, "Segment-wise timing and power measurement in software emulation," in *Proc. Design, Automation, and Test Eur. Conf., Designers' Forum*, Munich, Germany, Mar. 2001, pp. 165–169.

Fabian Wolf received the Diploma degree in electrical engineering from the Technical University of Braunschweig, Germany, in 1996.

Since 1996, he has been a Research Staff Member at the Technical University of Braunschweig, Germany. He is currently working on the development of SYMTA, an experimental system for static analysis of software running time and power consumption. His current research interests include software timing and power analysis as well as software emulation and cache analysis.

Rolf Ernst (M'89) received the Diploma degree in computer science and the Ph.D. degree in electrical engineering, both from the University of Erlangen-Nuremberg, Germany, in 1981 and 1987, respectively.

He was a Member of the Technical Staff at AT&T Bell Laboratories, Allentown, PA, from 1988 to 1989. Since 1990, he has been a Professor in the Department of Electrical Engineering at the Technical University of Braunschweig, Germany, where he heads the Institute of Computer and Communication Network Engineering. His main research interests are in embedded system design and embedded system design automation.

Wei Ye received the Diploma degree in computer engineering from the Technical University of Huazhong, China, in 1985.

He was a Research Staff Member with the Technical University of Braunschweig, Germany, from 1991 to 1997. Since 1998, he has been working as a Computer Engineer for Siemens AG in Erlangen-Nuremberg, Germany. His main research interests are in timing analysis of real-time systems, architecture modeling, processor emulation, and cache analysis.