Abstract

Embedded systems are widely used for supporting our everyday life. In the area of safety-critical systems human life often depends on the system’s correct behavior. Many of such systems are hard real-time systems, so that the notion of correctness not only means functional correctness. They additionally have to obey stringent timing constraints, i.e., timely task completion under all circumstances is essential. An example for such a safety-critical system is the flight control computer in an airplane, which is responsible for stability, attitude and path control.

In order to derive guarantees on the timing behavior of hard real-time systems, the worst-case execution time (WCET) of each task in the system has to be determined. Saarland University and AbsInt GmbH have successfully developed the aiT WCET analyzer for computing safe upper bounds on the WCET of a task. The computation is mainly based on abstract interpretation of timing models of the processor and its periphery. Such timing models are currently hand-crafted by human experts. Therefore their implementation is a time-consuming and error-prone process.

Modern processors or system controllers are automatically synthesized out of formal hardware specifications like VHDL or VERILOG. Besides the system’ functional behavior, such specifications provide all information needed for the creation of a timing model. But due to their size and complexity, manually examining the sources is even more complex than only looking at the processor manuals. Moreover, this would not reduce the effort nor the probability of implementation errors.

To face this problem, this paper proposes a method for semi-automatically deriving suitable timing models out of formal hardware specifications in VHDL that fit to the tool chain of the aiT WCET analyzer. By this, we reduce the creation time of timing models from months to weeks.

Categories and Subject Descriptors J.7 [Computers in other systems]: Real time; C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

General Terms Performance, Verification

Keywords Hard real-time, worst-case execution time, VHDL

1. Introduction

Embedded systems are the most used computer systems in our lives. They can be found in mobile phones, portable multimedia players, cars, etc. Most of them are subject to stringent timing constraints which are dictated by the surrounding physical environment. We are concerned with the problem of guaranteeing that the timing constraints of tasks when executed on a given processor architecture will be met ("timing validation").

Consequences of improper timing behavior are severe: often lives depend on correct working systems (like the flight control system in airplanes). So, it must be guaranteed that each real-time task in the system meets its given time frame to complete its execution even in the worst-case scenario. To ensure this, we need to perform a timing analysis that computes the worst-case execution time (WCET) of each real-time task in the system.

Systems show a variability of execution times depending on

- the input data: this has always been so and will remain so as it is a property of the algorithm,
- the initial execution state: this is caused by modern architectural features such as caches, pipelines, and speculation, and
- interference from the environment: preemptions and interrupts.

The unit-time (executing an instruction always takes exactly one time unit) or constant-time abstraction used in many approaches to timing validation is thus rendered obsolete by the advent of modern processors (with features like branch prediction, out-of-order execution or speculation, etc.).

In general, the state space of input data and initial states is too large to exhaustively explore all possible executions and so determine the exact worst-case execution time. Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions inevitably lose information, and yet must guarantee upper bounds for the worst-case execution time. Nevertheless, the theory of abstract interpretation [6, 7, 8] enables the application of static program analysis [27] for determining the timing behavior of embedded applications.

The computation of worst-case timing bounds for a program is realized by first employing an abstract processor model ("timing model") to compute a cycle-level abstract semantics of the program and, in a second phase, mapping resulting time bounds for program portions to an Integer Linear Program (ILP) whose maximal solution yields the final bound. This tool architecture has been successfully used to determine precise upper bounds on the execution times of real-time programs running on processors used in embedded systems [10]. A commercially available tool, aiT by AbsInt GmbH, cf. http://www.absint.de/wcet.htm, was implemented and is used in the aeronautics [31] and automotive industries [19].
The more compute-intensive first phase is a micro-architectural analysis and has the following three constituents:

1. Value analysis [30] attempts to compute information about data accesses and control flow, in particular it tries to identify infeasible paths, syntactically possible paths that will never be taken because of contradictory conditions.

2. Cache-behavior prediction [13] determines a safe and concise approximation of the contents of caches in order to classify memory accesses as definite cache hits or misses.

3. Pipeline-behavior prediction [33] analyzes how instructions pass through the pipeline taking cache-hit or miss information into account. At the end of simulating one instruction, a certain set of final states has been reached. The pipeline analysis starts the analysis of the next instruction in all those states. Here, the timing model introduces non-determinism that leads to multiple possible execution paths in the analyzed task. The pipeline analysis need to examine all of these paths.

Currently, these timing models are hand-crafted by human experts [33, 34, 12]. Therefore the model creation as well as the implementation of the corresponding pipeline analysis is a very time-consuming and error-prone process.

As modern processors are synthesized out of formal hardware description languages, like VHDL [3, 18, 17], in which their behavior (including the timing) is exactly specified, the timing model could be semi-automatically derived from them. This would avoid errors introduced by manual implementations due to human involvement, and it would speed up the process, too.

VHDL models of real-world processors are usually very big and complex making WCET determination a very difficult task [37]. Just generating a pipeline analysis that covers the whole micro-architectural behavior would additionally increase the state space. This would render the resulting timing analysis infeasible in terms of space and time consumption.

This paper describes how to derive timing models out of formal VHDL specifications in a semi-automatic way.

In a first step, we reduce the size of the model by pruning out all parts that do not contribute to the timing behavior. For example, we do not need information about each step within a multiplier unit. Instead, it suffices to know for many clock cycles an instruction occupies each stage of the multiplier pipeline.

The pruned model still contains a lot of detailed information about the processor state. But for practical reasons it is impossible to represent all state information in full detail. If we were to exactly record e.g., the contents of all memory cells or registers, the space required for the analysis would be prohibitive. Luckily, in many cases the exact knowledge about these things is not important as far as timing is concerned: an addition always takes the same amount of time, no matter what the arguments are. In other cases, the timing does depend on such information, but we may choose to lose the exact timing knowledge in order to make the analysis more efficient, or even to make it possible at all. One example for this are multiplications on some architectures, which are faster if one argument has many leading zero bits. By not keeping track of the arguments exactly, we have to assume an entire range of execution times for multiplication. The loss in precision is acceptable in this case, as the difference is usually only a few processor cycles and multiplications are rare.

Therefore, the second part of the timing model derivation is the abstraction of the processor state. Abstractions mean that we either leave out some details of the processor state or we approximate them. An example for an approximation is the replacement of concrete addresses by address intervals. For memory accesses, we do not need to know the exact address. We only need to know the type of memory that is accessed in order to simulate its timing behavior.

Using the methodology of abstract interpretation, one can trade precision of the analysis against efficiency by choosing different processor abstractions and concretization relations between the concrete processor state and the abstract one.

In theory, this was done in [34] by manually examining the VHDL sources of a memory controller. This shows the theoretical feasibility of our approach. The contribution of this paper is twofold. Firstly we provide a method that automates the manual process as much as possible. Secondly we apply our method to whole system’s specifications (the processor core including the peripheral components as e.g., the memory controller).

The implementation of the outlined sequence of model analyses and transformations is currently under development, so we cannot give any reasonable experimental results so far. The same holds for a comparison to a manually derived timing model.

The paper is structured as follows: Section 3 gives details about the hardware description language VHDL and its formal semantics. In Section 4, we give an overview about the problem of computing safe and precise upper bounds on the worst-case execution time of safety-critical applications. Section 5 and Section 6 then introduce transformations and abstractions on VHDL that are used to derive a timing model out of the original VHDL specification. Section 7 illustrates our automation process and its seamless integration into the aiT tool chain. Section 8 concludes.

2. Related Work

There are a variety of tools for computing worst-case execution time bounds for embedded system applications. According to [36], these tools can be categorized into two classes: static methods and measurement-based methods.

The first category employs static analyses to determine the timing bounds. Most of the tools in this category examine the fully linked executable. The hybrid prototype of TU Vienna [20] performs static analyses on source code level. There, it must be assured that the compiler does not significantly change the structure of the program code. Most of the static methods do not model complex processor pipeline features like out-of-order or speculative execution, static/dynamic branch prediction or caches in general. Examples are Bound-T [15], SWEET [9], Chalmers research prototype [24] or Chronos [22]. The most powerful and industrially usable tool in this context is aiT [11]. It can cope with highly complex processor architectures, computes precise WCET bounds and is user-friendly (e.g., warns about unrealistic code annotations).

The second category are the measurement-based WCET tools. Their methodology is not based on static analyses but on determining timing bounds by measuring the WCET path. The most promising and successful tools are RapiTime [4] and a research prototype from TU Vienna [28]. In contrast to the static analyses based tools, one needs to generate input vectors that trigger the worst-case execution path to be executed during measurement. This cannot be computed in general. Solutions are to generate test cases that provide a full execution coverage or to measure basic blocks separately and combine the basic block execution times to the time bound. In the latter case, the user needs to find the correct combination of execution paths of consecutive basic blocks.

In addition to the different WCET tools described above, there are many VHDL simulators like GHDL [32] or ModelSim [5]. Such simulators cannot be used to compute timing bounds by program execution simulation because:
entity counter is
  port (clk, rst : in std_logic;
  val : out std_logic_vector (2 downto 0));
end entity;

architecture rtl of counter is
  signal cnt : std_logic_vector (2 downto 0);
begin
  P1: process (clk, rst) is
    if (rst = '1') then
      cnt <= "000";
    elsif (rising_edge (clk)) then
      cnt <= cnt + '1';
      val <= cnt;
    end if;
  end process P1;
end:

Figure 1. 3-bit counter in VHDL.

- Modern real world processes (like Motorola PowerPC755 [16]) used in embedded systems are far too complex. The computational complexity of the simulation would render the method infeasible for industrial usage.
- Due to the complexity of the problem, we introduce non-determinism into the computation model. No traditional VHDL simulator can cope with that.

To the best of our knowledge, nobody has tried before to automate (at least partly) the development process of a timing model for processors with the goal of WCET determination.

3. VHSIC Hardware Description Language (VHDL)

VHDL is an IEEE Standard defined in IEEE 1076 [3, 18]. The focus of the language ranges from specifying circuits at wavefront level to describing large system behaviors with high-level constructs. As a result, the standard is huge, thus, this paper only considers the synthesizable subset of VHDL, defined in [17].

Figure 1 shows the specification of a simple 3-bit counter. The VHDL description of a circuit consists of an interface declaration defining the in- and output signals of the circuit and of one or more implementations. In VHDL, the interface declaration is called an entity, the implementation an architecture.

The entity declaration defines the input ports of the circuit (clk and rst). The counter is designed as a synchronous circuit, i.e., all computations are synchronized on the transitions of a global signal. This signal is referred to as the global clock (clk in the sample circuit). The current value of the counter is provided by the output port val which is a 3-bit binary number.

The implementation is given in form of a process (P1). A process executes its code, whenever one of the signals contained in the processes sensitivity list (clk and rst) changes its value. Thus, the sensitivity list of a process is an implicit wait-statement at its end. After execution of all statements, execution suspends until another change of at least one signal’s value.

VHDL also supports component-based circuit specifications. Figure 2 gives an example for hierarchical circuit composition. Here, a circuit for the logical implication \( a \rightarrow b \) for two inputs \( a \) and \( b \) and the output \( c \) is built from a logical-or gate or2bit and a negation gate invert, implementing the implication by the formula \( c = \neg a \lor b \). Note that or0 is an instance of the generic entity or2bit, as not0 is one of the entity invert.

Having a hierarchically composed specification of a circuit, elaboration has to be performed in order to get a flat definition of it. Elaboration does all the required renaming for unifying names, wires all structural descriptions, etc. The result is one large entity consisting of a number of processes and some locally defined signals.

A VHDL process consists of a set of local variables that are only accessible from inside the process. By contrast, local signals can be accessed by more than one process, but only one process is allowed to drive the value of a signal.\(^2\) Within a process, execution of statements is done sequentially.

VHDL makes a distinction between assignments to a variable and to a signal. Assigning a value to a variable takes effect immediately (i.e., the next reference of this variable returns the newly assigned value), whereas the assignment of a value to a signal is only scheduled to be the future value (i.e., the next reference returns the old value). E.g., in Figure 1, the signal assignment \( \text{cnt} \leftarrow \text{cnt} + '1' \); schedules the next value of \( \text{cnt} \) to be \( \text{cnt} + 1 \) but the next reference \( \text{val} \leftarrow \text{cnt} \); schedules the next value of \( \text{val} \) to be the current value of \( \text{cnt} \). These future values take effect as soon as all processes suspend their execution.

The semantics of a VHDL program, i.e., a set of processes, can be described as follows:

1. Execute processes until they suspend.
2. If all processes are suspended, make all scheduled signal assignments visible at once.
3. If there is a process being sensitive on a signal having changed its value, resume this process and go to step 1.
4. Otherwise, an external signal must change its value (e.g., the clock signal). If this happens, resume all processes waiting for this signal and go to step 1.

Thus, the semantics of VHDL can be seen as a two-level semantics: sequential process execution at its first, signal update and process revocation at its second level. A formal definition of the semantics of VHDL can be found in [25].

4. WCET Computation

Current state-of-the-art processors in embedded systems make use of instruction pipelines, i.e., the execution of consecutive instructions is overlapped. They pass simultaneously through different

\(^2\) In VHDL, the use of explicit wait-statements and sensitivity lists is exclusive. We assume, that the only place within a process, where wait-statements may occur, is at the end of the body of a process.

\(^3\) In full VHDL, resolution functions can be used for value computation of signals being driven by two or more processes.
pipeline stages: An instruction has to be first loaded from the memory. The residence time of the instruction in the fetch stage thereby is determined by the content of the instruction cache. After being loaded, the instruction can be decoded and dispatched to its corresponding execution unit (decode stage). While in the execution stage, instructions compete for resources, such as execution units and buses. After finishing its execution, the instruction waits for completion (writeback stage). Having passed this stage, all changes made to operands, etc., are part of the architectural state, thus, after this point, the instruction does not longer affect the behavior of the processor. This is called instruction retirement. Please note, that in modern processors, there can be several locations in the pipeline, where retirement (“leaving the pipeline”) can happen.

The number of cycles an instruction spends within one stage (and therewith its overall execution time) varies and depends on several criteria, namely

- the current state of the processors pipeline,
- the current environmental state, i.e., the content of the cache(s), etc. and
- the input program.

As stated before, pipeline-behavior prediction is based on the abstract interpretation of a timing model. The execution of a program is simulated by feeding instruction sequences from a control-flow graph to the timing model which computes the processor state changes at cycle granularity and keeps track of the elapsing clock cycles. Therefore the timing model is at a very detailed level.

Figure 4 (dashed box “Timing Analyzer”) shows the structure of the aiT timing analyzer that implements this theory. The starting point is the fully linked binary that contains the task(s) for which the WCET should be computed. After control-flow reconstruction, the above mentioned Micro-architectural Analyses are performed. Path Analysis computes the worst-case execution path based on the basic block execution times computed by the micro-architectural analyses. For further details on the aiT WCET analyzer, please refer to [10].

Abstract interpretation of the timing model is guaranteed to give upper WCET bounds for the execution of an instruction sequence [33]. The guarantee, however, is relative to the timing model itself. If the model fails to correctly describe the system’s behavior, the computed WCET bound may be incorrect.

As nowadays the system hardware is automatically synthesized from formal specifications (e.g., in VHDL or VERILOG), they determine the complete system’s behavior — including timing information. So, it is highly desirable to extract the information needed for the timing model creation from such specifications. This would ensure that the timing model is correct by construction and would speed up the creation time massively.

Unfortunately, VHDL specifications contain many details on processor state changes during program execution. If we would derive a timing model from a processor specification without any model preprocessing, the resource consumption of the resulting pipeline analysis would be prohibitive. Therefore, we need to reduce the size of the model.

Section 5 introduces some needed simplifications and transformations that compress a VHDL specification without loosing information about the timing behavior. Based on this, Section 6 describes the process of how to derive a computationally handable timing model from a VHDL specification.

### 5. Transformations of VHDL

A VHDL specification of a complete processor is quite large, e.g., about 70000 lines of VHDL for the Leon 2 processor [14]. Simply transforming the specification into a timing model would render the resulting timing analysis infeasible, especially in terms of space consumption. The design of a hard real-time system in hardware and software defines, which scenarios have to be modeled for timing analysis. E.g., if a CPU offers support for floating point instructions, but the software does not use them, the floating point unit does not have to be modeled for timing analysis. In the following, we describe some techniques to reduce the complexity of a VHDL specification in order to make timing analysis feasible.

#### 5.1 Environmental Constraints

Modern embedded architectures offer a huge variability concerning their configuration (e.g., dynamic or static branch prediction, bus pipelining enabled or disabled, write-through or write-back cache behavior, etc.) but also with respect to connected devices. Many of them allow to use a huge variety of memory types and therewith memory controllers, connected buses, etc. Those hardware settings are fixed during run time which renders some parts of the hardware specification obsolete for a timing analysis. The removal of those parts contributes to the reduction of the complexity of the created timing analysis.

Some events in modern architectures are either asynchronous to program execution (e.g., interrupts, DMA) or not predictable (e.g., ECC errors in RAM, exceptions) in the model. Timing analysis cannot deal with those events.

In order to make static timing analysis still applicable, it is necessary to make assumptions about the environment. An exception or interrupt brings the whole system to a state where no timing bound for the current task is needed. So for timing analysis, we can safely assume that those events do not happen.

Based on this assumption, some signals in the VHDL description never change their value, so we can “hardwire” these signals to their default value (i.e., the value during system reset). As a result, parts of the VHDL become unreachable (i.e., control never reaches these statements), so we can use dead-code elimination to remove those parts from the VHDL specification.

The same has to be done for asynchronous events that occur frequently (e.g., SDRAM refreshes). Their effect on the execution time has to be incorporated, e.g., by adding penalties based on the computed WCET and the worst-case occurrence of the events or by statistical means.

#### 5.2 Domain Abstraction

Registers in hardware are used for storing data processed by the CPU. For timing analysis, the concrete value stored in a register is often not important, e.g., we are not interested in the result of an addition but in the time it takes to add two registers. Thus register content can be abstracted from the VHDL model to make it smaller.

For other parts in the model, an approximated value of a register might suffice to model the correct timing behavior, e.g., when accessing the memory composed of a slow and a fast region, it suffices to abstract a concrete address by the information, which memory region will be accessed.

Same argumentation also holds for queues in the VHDL model, where not the content of a queue entry is interesting, but the size of the queue to determine if it is full or not. This type of transformation of VHDL we call domain abstraction. It is more or less a change of types of signals and variables in the VHDL model. Changing a domain type also necessitates the adaptions of functors working on the abstracted domain.

Domain abstraction might introduce non-determinism into the VHDL model (e.g., abstracting a concrete address to an address...
interval and checking it for a concrete value might result in three values: equal, not equal, and perhaps). It is the task of the timing analysis framework to cope with and handle all possibilities.

### 5.3 Process Substitution

As stated in Section 3, the active parts of a VHDL specification are processes. Normally, a process drives at least one signal containing the result of its computational task, e.g., the next address to be fetched from memory.

For timing analysis, the details on how a process derives its result might be uninteresting or modeling all its details might result in a model being too large to be used for timing analysis. Process substitution allows for replacing a concrete VHDL process implementation by a custom implementation modeling less details or using a powerful abstraction. E.g., this has been successfully done for caches and is known as the cache abstraction [13]. Thereby, the concrete cache is replaced by an abstracted cache storing the maximal ages of all lines that are definitively in the cache.

Process substitution might introduce non-determinism into the model. Whenever a decision depends on abstracted data and the abstraction is not able to precisely determine the result, the timing model has to follow all possibilities (cf. [25] for the semantics of abstracted VHDL).

### 5.4 Memory Abstraction

Processors execute programs that are kept in memory. For execution, instructions have to be fetched from main memory, decoded, executed, and the results are written back to memory.

Especially large memory arrays like main memory blow up the timing model. Content of registers, addresses of memory cells being accessed by a program, and content of memory cells can be computed using the value analysis and do not require a cycle-wise simulation of the processor’s behavior [30].

To make timing analysis applicable, large memory arrays must be abstracted. Otherwise, the resulting timing models cannot be used for WCET computation due to space limitations. As stated in Section 4, our timing analysis is based on a reconstructed control-flow graph of the executable program. Thus, instructions have to be inserted into the timing model using the information from the control-flow graph, so a pseudo VHDL process has to be added to the model acting as interface to this graph.

Due to domain abstraction and process substitution (cf. Section 5.2 and Section 5.3), and also due to static analysis, data access addresses of instructions might only be known as safe intervals, not as single addresses. Thus, the VHDL design must be adapted to utilize the information from the value analysis instead of the real computation of addresses. For this, all places where data access addresses are generated by instructions have to be identified. At these places pseudo VHDL processes have to be added that interfaces with the value analysis to retrieve the previously computed intervals. Consequently, addresses must be abstracted to address intervals (cf. Section 5.2).

### 6. Deriving the Timing Model

This section describes the derivation process starting from the VHDL model and ending in the corresponding timing model that can be used within the pipeline analysis of the WCET computation framework (cf. Section 4).

Figure 3 illustrates the process flow. At the top, there is the incoming VHDL model. As mentioned in the introduction and Section 4, the size of the model needs to be reduced. Most of this is done by the Model preprocessing step which is described below. Then the preprocessed model is the starting point for the search and application of Processor State Abstractions (cf. Section 6.2).

This iterative step results in the Timing Model used for the pipeline behavior prediction within the WCET computation framework presented in Section 4.

As the derived timing model is the basis for the determination of WCET guarantees for safety-critical systems, we need to prove that the resulting model after the application of our size-reducing transformations and state abstractions correctly approximates the timing behavior of the examined system. This is ongoing work in cooperation with researchers from the Electronic Design Automation Group of TU Kaiserslautern, Germany.

The next sections detail about the preprocessing step as well as the processor state abstractions.

### 6.1 Model Preprocessing

The task of the model preprocessing step mainly is to reduce the overall size of the VHDL model by eliminating parts that are not relevant for the timing behavior of the system.

Embedded systems are very specific, both in terms of the hardware and software components. Section 5.1 details about so-called environmental constraints that results from the fact that there are hardware features that are unused at all in the analyzed application or only in a limited way.

In the VHDL code, such configurable parts of the hardware are guarded by control signals that indicate whether the particular hardware feature is enabled or disabled. These control signals usually are constant, i.e., they are read but never written. If we analyze the model with an activated reset signal, we get some initial/default values for these control signals. Performing a constant propagation afterwards effectively “marks” parts of the model which process disabled hardware components/features as dead code. These can be pruned out by a dead code elimination. We call this model transformation Environmental Assumption Refinement (cf. Figure 3). A similar optimization that reduces the size of the original VHDL model is the Timing Dead Code Elimination. This also means the removal of model parts that does not contribute to the timing behavior of the system. In contrast to the Environmental Assumption Refinement, we can remove code pieces of hardware components that are enabled in the processor configuration. For example, the VHDL code of a multiplier unit describes the way this unit works in all details. This includes the algorithmic implementation of the execution units or the control logic that influence the result of the computation of a unit. We do not need to know, how exactly the functional units are designed. We only need detailed
information about the timing of the instruction flow through the pipeline. This means that it suffices to know how long each instruction stays in each stage of the processor pipeline.

In order to restrict the model to "timing-alive" code, we need to identify all locations where instructions can leave the pipeline, i.e., where instruction retirement happens. Please note that there can be several locations for instruction retirement due to early-out-conditions for some instruction classes. Having identified the instruction retirements, we compute backward slices from these locations. This yields all VHDL statements that influence the instruction retirement and therefore contribute to the instruction flow through the pipeline. All VHDL code that is not contained in the union over all computed backward slices is simply "timing-dead" and can be safely removed.

The third size reduction transformation is the Data Path Elimination. As already mentioned in the introduction, incorporating the data paths in the timing model would prohibitively increase the resource consumption during analysis. Fortunately, the latency of instructions is normally not affected by the content of registers and memory cells they use. Therefore, we can factor out (i.e., remove completely) all data paths from the model. The information about contents of registers and memory cells is then computed by an external value analysis. In contrast to the real address computation, this value analysis operates on a different domain, i.e., address intervals. On the one hand, this is a consequence of the lack of precise information when analyzing at the assembly level [33]. On the other hand, it is implied by the methodology of abstract interpretation [7]. This factorization of address/value information of course introduces the need of a domain abstraction from addresses to address intervals (cf. Section 5.4) which is used in the next section.

6.2 Processor State Abstractions

The model preprocessing described above reduces the size of the original VHDL model significantly so that the resulting pipeline analysis becomes computationally feasible. Depending on the complexity of the hardware model, this size reduction might not suffice.

So far, we removed parts of the model that do not contribute to the timing behavior. Further size reduction can only be achieved by approximating state information rather than precisely modelling it. In principle, such a lack of information about the system state might result in a loss of precision (i.e., safe upper bounds) in the computed WCET bound because we do not model the timing behavior exactly anymore. Fortunately, the timing often is not affected by this as for example an addition always takes the same amount of time independent of the argument values. In contrast to that multiplications can be finished faster if one argument has many leading zero bits. The lack of precise information about that forces us to assume an entire range of execution times for multiplication. But this might be acceptable as multiplications are rare.

In general, we can perform three different abstractions (from Section 5) to the processor state:

- Process Substitution,
- Domain Abstractions, and
- Memory Abstraction.

As shown in Figure 3, the first two of these three abstractions are employed iteratively until the resource consumption of the resulting pipeline analysis is acceptable. Certainly, the memory abstraction can only be applied once. The exact usage of process substitution and domain abstraction then is an engineering problem. It mainly depends on the actual complexity of the architecture to be modeled. For more simple architectures (like ARM7 [23]), we can model nearly everything precisely without getting trouble with computational resources. More complex systems however require more abstractions. However, there are common abstractions to be done for all processors. These are:

- **Cache abstraction:** This is an example for a process substitution where the update of the caches is replaced by a custom C++ implementation. Details can be found in [13].
- **Address intervals:** This is an example for a domain abstraction as we replace concrete addresses by address intervals. This replacement is implied by the removal of the data paths as mentioned above.

All of these abstractions [7] of the original VHDL model certainly must not semantically change the timing behavior. Moreover, they are very architecture specific in terms of applicability and necessity. As described in Section 5, imprecise values lead to several possibilities in the execution of a program/task. If the number of alternatives is large, the computational complexity might be higher as without the particular abstraction. So the introduced transformations must be chosen carefully.

The processor state abstractions can lead to the loss of precise information about the processor state. During the successor state computation of an input state, this possibly results in multiple successor states if precise information is needed but not available. In that case, the micro-architectural analysis computes multiple execution paths through the program and annotates each path with the execution times. In other words, the timing model might become non-deterministic by the introduction of processor state abstractions. As described in Section 7.3, our code generator needs to cope with that non-determinism when generating the pipeline analysis based on the computed abstracted timing model.

7. Automation of the Derivation Process

The previous section has given an overview of the steps needed to derive a timing model from a formal hardware description. Performing these steps by hand is very time-consuming on the one hand [34], and error-prone on the other hand. To eliminate these harms, automation of the derivation process is mandatory.

All parts of model preprocessing described in Section 6.1 are based on static analysis methods known from compiler construction. Since variables in VHDL are local w.r.t. a process and signal changes are delayed to the synchronization point, choosing a fixed order for the process execution is possible. Thus, a VHDL model
can be transformed into a sequential program [29] allowing the application of these static analysis methods.

Our proposed framework for the derivation process automation is shown in Figure 4. Starting from an initial VHDL processor specification, a set of static analyses and transformations can be performed to derive an abstracted timing model. A code generator then creates simulation code fitting into the timing analysis framework as part of the micro-architectural analysis.

### 7.1 VHDL Analysis Framework

In order to ease the process of analyzing a VHDL specification of a processor, [29] describes a framework for static analysis of VHDL. A general overview of this analysis framework is shown in Figure 5. Given a rtl processor specification, VHDL2Crl2 transforms it into an intermediate representation language called Crl2 [21]. Sequentialization as well as elaboration are performed during this translation phase. Thereby, Crl2 is hierarchically organized in instructions, basic blocks and routines, i.e., basic blocks are completely enclosed within routines and analogously instructions are completely enclosed within basic blocks.

The Crl2 representation is the input for each static analyzer. To ease the implementation of the different analyzers, the Program Analyzer Generator (PAG) [26] is used. PAG is based on abstract interpretation. The tool can generate fast and efficient static analyzers out of concise analysis specifications of a data flow framework in PAG specific languages, called OPTLA and DATLA [1].

In order to transform a VHDL description into a sequential program, a fixed ordering of all processes is automatically chosen by VHDL2Crl2. This order is embedded into a special simulation routine that is required by the analysis framework. The sensitivity lists of processes are represented by newly created guards within the simulation routine. Furthermore, a special wait node is inserted in the simulation routine representing the VHDL synchronization point (cf. Section 3).

Using this analysis framework enables the easy implementation and use of static analyses in order to perform environmental assumption refinement and timing dead code elimination as described in Section 6.1, but also eases the process of implementing domain abstractions. This is described in following.

#### 7.1.1 Obtaining Initial Values

When simplifying a VHDL design, we want to remove assignments to signals (or variables). Remaining read-references to these then read a constant value. This value is the initial value of the signal/variable being assigned during system reset.

Identifying the default values of signals or variables can be achieved by evaluating the code segments that are active during system reset. The contents of all signals and variables after the code segments have finished are the initial values. Note that the computation of the initial values may not happen in just one clock cycle after the reset signal is asserted. Several parts of a system may have a longer initialization sequence, e.g., memory controllers going through a sequence of activation steps for the attached memory chips.

Determining the set of instructions that are executed to compute the initial values can be automated by computing a forward slice [35]. A forward slice on VHDL determines the set of instructions that may be executed, depending on the values of a set of signals (or variables). This set of instructions is referred to as the slicing criterion. All instructions not in the slice are guaranteed not to depend on any of those signals. Certainly, the computation of a slice has to take into account transitive dependencies and dependencies introduced by the sensitivity lists.

Computing the forward slice for the criterion “reset is activated” yields a slice containing all instructions that may be executed during system reset. To obtain the default values for all signals/variables, a constant propagation analysis can be performed on this slice.

A constant propagation analysis determines for each program point, i.e., each statement, if a signal or variable has a constant value when execution reaches that point.

To show the effectiveness and simplicity of the VHDL analysis framework, the following example describes, how a constant propagation analysis on VHDL can be modeled. We introduce a mapping from signal/variable names to their corresponding value and extend it by a bottom and a top element to denote not yet considered program points and unknown values respectively.

\[
F \equiv (\text{identifier} \rightarrow (\text{value} \cup \top)) \cup \bot
\]  

(1)

As stated in Section 3, VHDL differentiates between signal and variable assignments. Thus, the domain of the data flow problem (see [27] for details on data flow analysis) for constant propagation has to cover current and future values of the identifiers used. Furthermore, to evaluate the condition of process guards (cf. Section 7.1), it is necessary to decide whether a signal has changed its value or not. Therefore, the domain has to be extended by the old values of signals. Thus, the domain \(dfi\) for the constant propagation analysis is:

\[
dfi \equiv F \times F \times F
\]  

(2)

The transition functions for updating an incoming data flow value

\[
dfi_{\text{pre}} = \langle \text{cur}_{\text{pre}}, \text{fut}_{\text{pre}}, \text{old}_{\text{pre}} \rangle
\]  

(3)

to the output value \(dfi_{\text{post}}\) for the different nodes can be directly defined as follows:

- **assignment node**
  The data flow value for an assignment can be computed from the incoming values in case of a variable assignment by updating the current and the future value of \(dfi_{\text{pre}}\) with the newly assigned value or \(\top\), if this value is statically not computable.
  In case of a signal assignment, only the future value has to be updated.

- **guard node**
  The guard encapsulates the sensitivity list of a process and is responsible for process revocation. A process is only re-executed, if one of the signals in its sensitivity list changes its value. This can be checked by comparing \(\text{cur}_{\text{pre}}\) with \(\text{old}_{\text{pre}}\).
  Based on this result, the data flow value is propagated into the process or not.

- **wait node**
  At this node, all scheduled signal assignments take effect and are made visible at once. The new data flow value is computed by copying the future values to the current one and the current ones to the old ones.

\[
dfi_{\text{post}} = \langle \text{fut}_{\text{pre}}, \text{fut}_{\text{pre}}, \text{cur}_{\text{pre}} \rangle
\]  

(4)
7.1.2 Identifying Timing-Dead Code

As stated in Section 6.1, eliminating timing dead code in a VHDL specification can be achieved by backward slicing as defined in [35]. A backward slice contains all instructions that may influence the value of a signal (or variable) at the end of process execution (i.e., at suspension time). Any instruction not in the slice is guaranteed not to influence the value of the signal (variable).

Computing backward slices with the criterion defined in Section 6.1, i.e., the set of points where instructions are retired in the VHDL model, results in a set of slices. All instructions not contained in the union over these slices do not have any influence on the timing of the processor. Thus, all of these statements can be safely removed from the model.

[12] describes how to compute backward slices by combining several data flow problems, namely reaching definitions, dominator and post-dominator analyses. Thus, the VHDL analysis framework can be easily used also for timing dead code elimination.

7.1.3 Domain Abstraction Analysis

One of the most common abstractions used for simplifying a VHDL processor specification is the domain abstraction as described in Section 5.2. Thereby, the type of a signal (variable) or even a whole type shall be changed to a different type. Naturally, changing the domain of a signal (variable) induces a change of all functions used by the signal (variable).

Due to procedure and function calls within the VHDL code, identifying all places that might have to be changed is not done on a syntactical level. Since VHDL is fully typed, and due to the fact, that this type information is still present in the intermediate representation created by VHDL2CRTL2, it is easy to use the analysis framework to construct an analysis finding all points of interest. We call this analysis domain abstraction analysis.

Given a mapping from old to new types, the automatically created analyzer reports all places, where functions need to be adjusted. If a functor for the new type already exists (e.g., during a sequence of abstractions, we first abstract the type of a signal, and in a second step, we want to change another signal to be of the same abstract type), this is also detected by the analyzer and the functor is suggested for usage. The result of the analysis forms the input for the domain transformation tool described in the next section.

7.2 Transformation Tools

As the idea for domain abstractions remains an engineering problem, human involvement in this step will never be eliminated completely. Nevertheless, the analyses and abstractions described in the previous section form the basis to develop support tools that automate as much as possible of the model transformations.

As illustrated in Figure 3, the way from the unmodified VHDL model to the timing model is an iterative process concerning the processor state abstractions. Each iteration consist of the following sub tasks:

1. Exploration and visualization of the model,
2. Abstraction specific analyses,
3. Model transformation, and
4. Timing dead code elimination.

In the exploration phase, the model can be visualized with the AbsInt graph viewer aiSee [2]. For this, we are able to emit the model in GDL format. These visualization and exploration techniques can assist a human expert in developing ideas for space saving abstractions (e.g., a domain abstraction or a process substitution).

The abstraction specific analyses are the ones mentioned in Section 7. They are used to find all points where the concrete abstraction implies transformations.

To automate the application of a domain abstraction, one can use the tool domain-abstract. Its input is the model itself as well as a specification for the domain abstraction. The latter specifies a type conversion of either the type of a single variable or all variables of a given type into a new type. domain-abstract then transforms the model accordingly and informs about newly needed operators on the new domain. These operators have to be provided manually. For the process substitution support, there is the tool process-replace. Its input is the model, the name of the process to be removed and the name of an update function that contains the custom simulation implementation for this process. process-replace then removes the specified process and adjusts all parts of the model that potentially trigger a reactivation of this process (e.g., writes to signals that are in the sensitivity list of the removed process).

Each abstraction results in one or several transformations of the model. After having applied the transformation, we need to recheck for newly timing dead code as the transformation itself made parts of the model timing dead. For example, removing statements due to abstractions can lead to constant signals if all signal writes were removed by that. Then, the only possible values for those signals are their defaults which marks some parts of the model as dead code.

After the model ran through all needed transformations, the code generator presented in the next section can then be used for generating the pipeline analysis out of it.

7.3 Code Generator

Having applied the several transformations and abstractions described before, we get a timing model from which C++ simulation code can be generated [25]. The generated code provides functionality to simulate the processor state update that results by exactly one core clock cycle (clock cycle updater). Remembering the two-level semantics of VHDL (cf. Section 3), this includes the process execution as well as the reevaluation level as both steps together form one clock cycle update. In addition to the clock cycle update functionality, framework code is generated that supports the seamless integration into the aiT tool chain (cf. Figure 4). Concretely, we need to generate code that simulates the process revocation level of the VHDL semantic (cf. Section 3).

The transformations and abstractions of the original VHDL model introduce non-determinism into the timing model, which then is called abstract timing model (cf. Section 6). In contrast to a traditional hardware simulation, where the simulated execution yields a trace of processor states, our simulation of the abstract timing model yields a tree of processor states (execution tree). A processor state can have multiple successor states due to the imprecisionness of parts of the processor state. In that case, all possible execution paths in the execution tree are simulated, so that the worst-case path can be selected by the path analysis (cf. Figure 4).

Please note that due to the abstractions, a possible execution path through the execution tree does not necessarily need to correspond to a real execution trace on the hardware. For example, if there are some exclusive conditional paths in the binary and due to some unknown memory accesses, all paths are combined although some of them are mutually exclusive. Such situations can lead to over-estimations of the real WCET if such a combined path is selected as the worst-case execution path.
One of the main advantages of our code generator is that it can cope with this non-determinism. In order to correctly generate simulation code for the abstracted timing model, we need a clear transition from the VHDL semantics to an abstract simulation semantics that define the state transitions from one abstract processor state to its successor states. This can be found in [25].

8. Conclusions

There are many safety-critical applications within the area of embedded systems, e.g., flight control computers in avionics or airbag control systems in cars. Certainly, failures within such systems can lead to severe damage or even loss of human life. Therefore, safety-critical applications need a validation of their timing behavior besides proving their functional correctness.

For a timing validation of an embedded system, we need to know the WCET of each systems task. Saarland University and AbsInt GmbH developed the aiT tool that can compute safe and precise upper bounds on the worst-case execution time of a task/program by performing an abstract simulation of the tasks execution within the so called pipeline analysis. This pipeline analysis is based on timing models (of the processor) that are currently hand-crafted by human experts. The implementation of the pipeline analysis out of the timing model is as well very time-consuming as error-prone.

In order to get rid of these disadvantages, this paper shows how to semi-automatically derive such timing models out of formal hardware specifications in VHDL. The derivation process starting from the VHDL model up to the generated combined cache/pipeline analysis consists of the following steps:

- Fully automatic transformation of the VHDL model into the intermediate representation Ctrl.
- Automatic transformation tools for generally needed model compactions.
- Semi-automatic transformation and analysis tools for supporting the introduction of processor state abstractions.
- Fully automatic simulation-code generation that seamlessly integrates into the existing aiT tool chain.

By this, we removed human involvement as far as possible. The remaining engineering task is the introduction of the processor state abstractions which are dependent on the particular processor and its complexity. This means that our method eliminates the possibility of implementation/modeling errors as well as dramatically decreasing the development time of a timing analyzer for a given processor architecture. The first is of utmost importance in the context of safety-critical applications and their certification while the lower development time faces the industrial time-to-market speedup of new embedded applications.

Future work besides the implementation of the presented approach is the comparison between a hand-crafted timing model and its correspondent semi-automatically derived one. This should confirm our claims about the advantages of the described timing model derivation.

Acknowledgments

The authors would like to thank Reinhard Wilhelm and Stephan Thesing for their help, support and advice in making this work possible.

Work reported here was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS), by the European Community’s 7th Framework Programme in the Network of Excellence “ArtistDe

References


