Modeling a System Controller for Timing Analysis

Stephan Thesing*
FR 6.2 - Informatik
Saarland University
D-66041 Saarbrücken, Germany
thesing@cs.uni-sb.de

ABSTRACT

Upper bounds on worst-case execution times, which are commonly called WCET, are a prerequisite for validating the temporal correctness of tasks in a real-time system. Due to the execution history sensitive behavior of components like caches, pipelines, buffers and periphery, the static determination of safe upper execution-time bounds is a challenging task.

A successful timing analysis approach developed at Saarland University/AbsInt GmbH uses abstract interpretation to derive safe WCET bounds based on *timing models* of the processor and periphery in a system. So far, WCET research has focused on *processor* timing behavior. System performance depends heavily on the performance of the periphery, namely the *system controller*, which includes the memory access logic. This paper is the first to describe experience in deriving a timing model for such a system controller. The starting point is the VHDL description from which the controllers FPGA implementation is synthesized. By a sequence of simplifications and abstractions we obtain an abstract VHDL model which can be translated easily into a timing model.

The evaluation of the derived WCET tool shows that the approach leads to a precise and efficient analysis. This opens up the perspective of automatically deriving timing models from VHDL descriptions also for processors.

Categories and Subject Descriptors

B.4.4 [**Hardware**]: Input/Output and data communications; B.5.1 [**Hardware**]: RTL implementation; C.3 [**Computer system organization**]: special-purpose and application-based systems; C.4 [**Computer system organization**]: Performance of systems; D.2.4 [**Software**]: Software engineering; J.7 [**Computer Applications**]: Computers in other systems

General Terms

Reliability,Verification

Keywords

WCET, VHDL, timing analysis, worst-case execution time, static analysis, verification, peripherals, avionics, aiT

1. INTRODUCTION

For hard real-time systems, the knowledge about safe *upper bounds* for execution times of all tasks is a prerequisite for systems validation. As a failure of the system (avionics, automotive, weapon guidance, etc) may have catastrophic consequences, the strictness of validation is high: one strives for a *proof* that all tasks meet their deadlines, thus one needs a proof that any derived bound is safe.

Obtaining execution-time bounds is a challenging task. Modern architectures with caches, pipelines, buffers, etc. are very sensitive to small changes in the execution state, because the performance of those components is very execution history sensitive, e.g. depends on cache contents, pipeline occupancy, etc.

Static analysis has been proposed and utilized to obtain guaranteed safe bounds [7, 14, 5, 12]. Bounds obtained by static analysis are safe, because they are based on invariants about all executions. These invariants are upper approximations. However, the approximations may loose some precision, i.e. the computed WCET bound may be larger than the WCET.

The **aiT** tool [1] employs an abstract interpretation based on a *timing model* of the system. This model consists only of the components relevant for timing. The model works at processor-cycle granularity and contains the necessary state of the processor components and the systems periphery. The principle is an abstract simulation of the model for consecutive processor cycles. The WCET analysis is proven to be safe w.r.t. this model.

Yet, there is still a weak point in the rigid strive for a correctness proof of the analysis: the models in [14] were designed from information taken from processor and periphery manuals and by performing experiments on the real hardware. There is a chance that the resulting model contains deviations from the behavior of the real hardware, as the underlying information sources cannot be called authoritative.

*This work 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) and the European Union by the Network of Excellence on Embedded Systems Design ARTIST2, IST-004527

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

While great efforts have been undertaken to validate the model by comparing its predictions against detailed hardware traces in numerous experiments, an unsatisfying gap in the proof chain remains.

This defect is not inherent in the methods applied, but could be avoided. Modern processors and periphery are generated (synthesized) from higher-level descriptions in *hardware description languages* (HDLs). VHDL [9], and Verilog [15] are two examples for such HDLs. The whole behavior of the component is described, including the timing behavior. A timing model in the style of [14] derived from e.g. a VHDL description would be an authoritative one: the resulting WCET analysis is guaranteed to be correct, closing the proof chain from hardware to WCET results.

Unfortunately, there are no VHDL or Verilog descriptions publically available for the processors modeled in [14], which where the targets for WCET analysis in the DAEDALUS project [4]. However, Airbus France gave us access to a VHDL specification of a system controller. This quite complex controller connects the CPU to main memory and several busses (PCI, etc). System performance for real-time control applications is mostly limited by the performance of peripherals, especially memory access times. The system controller's timing behavior has thus huge influence on overall program execution performance and on the timing predictability.

First, we tried to obtain a "high-level" model for this controller in the same style as the model for the PowerPC 755 was designed. The intention behind this was that such a high-level model would be more efficient for analysis. However, we were not able to obtain such a model, because the only high-level descriptions available were not exhaustive and only contained cases that engineers judged to be worst-case scenarios. Due to timing anomalies [10] in the PowerPC 755, one cannot rely on local worst-case descriptions but must also consider other behavior for a global worst-case analysis. Looking at the VHDL sources of the controller, it became obvious that the inner workings are quite complicated and interdependent.

Thus, a timing model for this controller was derived from the VHDL description. This method gave a model that accurately captured the controller's behavior. In this paper, we present our experiences with this process and the lessons learned. Compared to a high-level model, this approach has the possible danger of giving an inefficient model. After all, we do not try to extract "meaning" from the VHDL sources, e.g. that an access is a page hit in SDRAM, but rather (abstractly) simulate the execution of the VHDL code itself. On the other hand, correctness of the model is guaranteed with this approach. As it turned out, the efficiency of this "dump" approach is very good and does not pose a practical problem.

Section 2 presents a short overview about WCET analysis in general, especially about timing models. The concepts and semantics of VHDL are sketched in Section 3. The system controller is described in some detail in Section 4. The important contribution of this paper is handled in Section 5: how we obtained a timing model from the VHDL description. An experimental evaluation of integrating the model into **aiT** is given in Section 6. Section 7 concludes the paper and discusses related work.

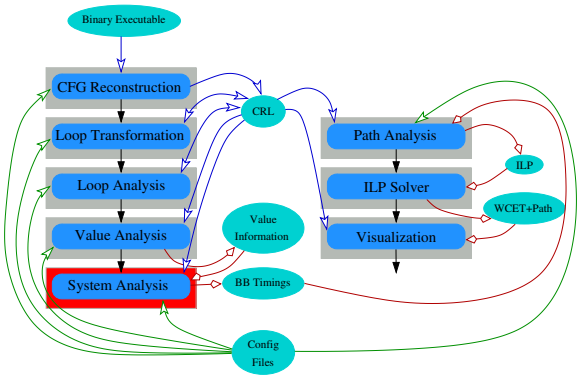


Figure 1: The aiT toolchain

2. THE WCET ANALYZER: AIT

The WCET tool **aiT** performs timing analysis in several phases, it reconstructs the control-flow from an executable, performs loop recognition, value analysis, system analysis and computes the global bound from basic block contributions, cf. Fig. 1. The details can be found in [7, 14]; relevant for this paper are only the value analysis and the system analysis. The value analysis computes an overapproximation for data addresses, an interval for each data-access instruction in the program. In most cases ($> 80\%$) the analysis is able to determine exact addresses (i.e. singleton intervals). For the others, this means that the next phase, the system analysis, has to assume that any memory location inside the interval may be accessed by the instruction.

The *system analysis* computes an upper bound for each basic block in each control-flow context. To compute the bound it determines at each program point a set of *abstract system states*. Each abstract system state *represents* a set of (concrete) states of the system (content of queues, pipeline stages, etc) such that every possible concrete system state at the program point is represented by one abstract state. In an abstract system state, there are no components, which do not influence the timing behavior. Some components that do influence the timing behavior are too costly to keep in an analysis, e.g. memory or register contents: the analysis assumes no information about their real contents and uses a conservative approximation. Other components, e.g. the caches, are approximated such that safe information is retained about their contents, although the information is not complete, cf. [6].

System analysis is an abstract simulation of the evolution of these abstract system states over processor cycles. The simulation for an instruction stops, when the instruction has left the pipeline. The maximal number of transitions from a starting state is the bound for that instruction. The resulting states at simulation end are the starting states for the successor instruction.

The transitions from one state to another are given by a *timing model*. This model describes the different *components* of a state and how it evolves. Following the examples for HDLs, the model structures components into *units* (like integer unit, load store unit, etc) communicating via *signals*. Our formalism uses two types of signals: *instantaneous* and *delayed*, which can both carry data. Instantaneous signals take effect as soon as they are emitted, i.e. in the same pro-

cessor cycle, receiving units react upon them immediately. Delayed signals are received only in the next processor cycle and thus carry events over processor cycles. Fig. 2 shows a top-level view of the model for the PowerPC 755 processor taken from [14]. Delayed signals are shown as dashed lines, instantaneous as solid lines. The complete definition of the model can be found in [14] here it is only important that the three delayed signals at the bottom of Fig. 2 handle the communication between the processor core (its Bus Unit, BU) and the system controller (Chip Set Unit, CSU). The $\mathbf{ts}(a, t, l)$ signal mimics the PowerPC 755's transfer start signal with address a, type t (data or instruction) and length l (1, 2, 4, 8 byte or burst access with 4×8 bytes). The **aack** and **ta** signals are the counterparts of the 755's address acknowledge and transfer acknowledge signals. The address parameter of the **ts** signal is not a simple 32 bit address, but rather an *address interval* computed by the value analysis. The bus protocol of the PowerPC 755 is divided into an address phase and a data phase. The processor requests an access by asserting **ts**, then waits for the **aack** signal from the system controller. This constitutes the address phase. The actual data is then transferred upon assertion of the **ta** signal by the controller; one assertion for single transfers, four assertions for burst accesses. This constitutes the data phase. While a data phase is in progress, another address phase may be performed (pipelining).

We want to derive a timing model for the CSU unit from its VHDL description. For this purpose, the next section briefly introduces VHDL.

3. VHDL

VHDL [9] models hardware by first specifying the interfacing of the components, called *entities*, and then giving an implementation, called *architecture*, for each entity. The interface consists of the *signals* that an entity receives or sends out; signals can carry data.

An architecture can either implement an entity by giving its subcomponents and their interconnection or by giving a number of parallel *processes*. A process is *sensitive* to a list of signals (from its interface or locally declared ones) and executes an imperative program every time the *value* of a signal in its sensitivity list changes. Besides signals, VHDL also provides *variables*. The difference between the two lies in the moment, at which an assignment takes effect. An assignment to a variable takes effect immediately, while an assignment to a signal is only *scheduled* to take effect later, when the process suspends. The reason for this differentiation lies in the resolution of conflicts during the parallel execution of all processes. All processes that have been triggered for execution by a signal change execute until they have reached the end of their program code[1].

When all processes have suspended, the new values for signals scheduled by assignments take effect and may cause several processes to restart. Variables are always local to a process, so no effects across processes can occur by variable assignment. As observed in e.g. [8], this form of parallelism can be simulated by executing the processes in an arbitrary order until they have suspended and then performing the signal assignments. It is checked if any process has to be restarted due to changes in the value of a signal in its sensitivity list and the evaluation begins anew. Fig. 3 gives a small VHDL example for a buffer.

[1]Or until they reach a `wait` statement. For synthesisable VHDL, such a statement is implicitly placed at the end of the process code and cannot occur anywhere else.

```
1    ENTITY buf IS
2      PORT(rd:  in std_logic; wr:  in std_logic;
3        dr:  out std_logic_vector(15 downto 0);
4        wd:  in std_logic_vector(15 downto 0);
5        reset:  in std_logic;
6        clk:  in std_logic );
7    END;
8    ARCHITECTURE rtl of buf IS
9      signal data:  std_logic_vector(15 downto 0);
10     work:  PROCESS(reset, clk) IS
11       IF (reset='1') THEN
12         -- clear data on reset
13         data <= "0000000000000000";
14       ELSIF (rising_edge(clk)) THEN
15         IF (wr='1') THEN
16           -- write request
17           data <= wr; -- new content
18         END IF;
19         IF (rd='1') THEN
20           -- read request
21           dr<=data; -- copy out data
22         END IF;
23       END IF;
24     END; -- of process work
25   END; -- of architecture
```

Figure 3: A buffer in VHDL

In this example, a small buffer can be read or written at the rising edge of the clock `clk`, while the `reset` signal clears the data (and takes precedence over reading/writing). A read request is denoted by a 1 on signal `rd`, write operation by signal `wr`; both can be active at the same clock edge. Data is input/output via `wd` and `rd`. Note that the value of signal `data` on line 21 holds the value the signal had at the start of process execution, as the potential assignment on line 17 has not yet been performed, but is only scheduled for execution at suspension time: when performing a simultaneous read and write access, the read data returned will be that of the *previous* write access (or 0), not the one currently being written. By changing the declaration of `data` to `variable data: std_logic_vector(15 downto 0);` and all assignments to it to use the variable assignment operator `:=` instead of `<=`, this behavior changes, as the reference in line 21 will then get the value written in line 17!

In the design we investigate, all processes are either sensitive to the rising edge of the systems bus clock, or are queues triggered by a read or write signal. Thus, in this case, all processes have to be executed at the rising edge of the clock until they finish and then execute the processes that handle the queues, if the respective read or write signal has been asserted. No recheck of process execution is necessary for our design. The next section gives an overview of the system controller's components and operation.

4. THE SYSTEM CONTROLLER

The system controller is automatically synthesized from a VHDL description. It contains four main parts, cf. Fig. 4.

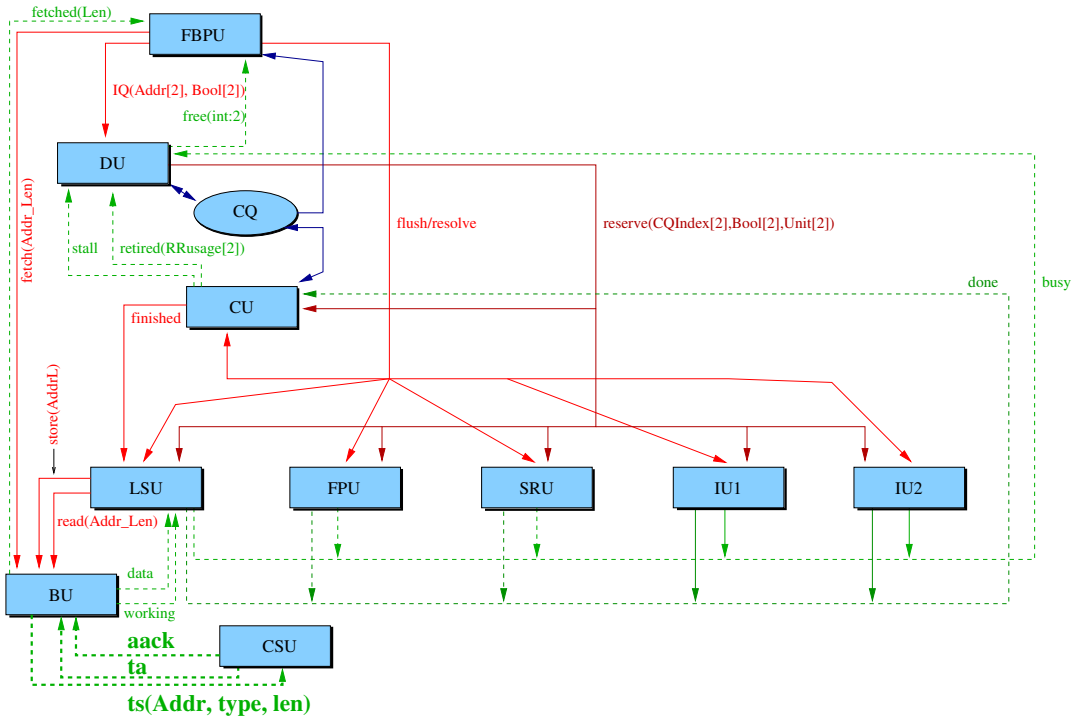
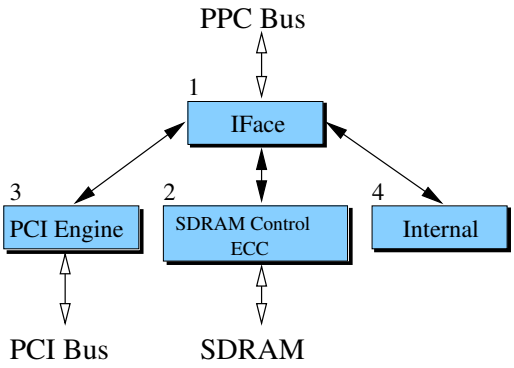


Figure 2: PowerPC 755 timing model

The PowerPC bus interface manages the PowerPC bus. Accesses on this bus are made up of two phases: address and data. During the address phase, initiated by the `ts` signal going active, the address, length and type of the access are put onto the bus by the CPU. The system controller finishes the address phase after latching the address and size/type information by asserting the `aack` signal. The data phase starts after the address phase is complete. Transfers are acknowledged by putting the data onto the data lines and asserting the `ta` signal. Depending on the transfer direction (read or write), data lanes are driven by the CPU or the system controller. The bus is pipelined, i.e. a new address phase can start as soon as the previous address phase has finished, even if the data phase of the first access is not yet completed. The PowerPC CPU (PPC 755) limits the pipelining depth to two outstanding accesses at any time. Also, the interrupt and exception inputs of the PPC 755 are generated by the controller.

The SDRAM part of the system controller is made up of a part that directly drives the SDRAM protocol lines (`RAS`, `CAS`, etc) and a checksumming part that provides the Error Correcting Code (ECC) capability of the memory. For each 64 bit (the native bus width of the PPC 755) of data in memory, 8 additional bits with checksums are stored. This allows to detect two bit errors in memory. Furthermore, the controller automatically corrects one bit errors during transfers. The disadvantage of using ECC is that it can only be computed for full 64 bit double words. For shorter accesses, writing only a part of a double word, the full double word must be read from memory, partly replaced and written back after computation of the new ECC. To avoid this performance penalty, the controller contains a write buffer. In case of multiple consecutive short write accesses to the same double word, only the first access has to read the data from



1. The interface to the PowerPC bus
2. A SDRAM controller with ECC capability
3. A PCI controller
4. Internal registers and features

Figure 4: The system controller

memory. Further accesses only have to perform ECC computation and double word writes. As usual with SDRAM controllers, the system controller keeps track of the pages it has open in the SDRAM memory chips. The controller can have up to four pages open. Access time to SDRAM heavily varies wether the accessed page is open or not. The necessary refresh logic is also part of the controller.

The PCI part connects to a slower (1/2 of the bus clock) PCI bus via a commercial PCI master interface. The interfacing also contains a write buffer for PCI writes. The

PCI transfer logic is handled by the commercial PCI master, whose VHDL was not available for analysis, which in turn is accessed by a simpler transfer protocol.

The internal functionality contains DMA engines, timers, exception and interrupt generation logic and internal registers. The PCI and internal registers are memory mapped into the PPC 755's address space. Due to the buffers, page registers and ECC machinery, the timings for accesses heavily depends on the sequence of accesses performed by the PPC 755 CPU and the intervals between consecutive accesses. No fixed timings can easily be derived and a higher level description of access timing based on the sequence of accesses being performed has not been possible with acceptable effort. Therefore, we choose to simply transform the VHDL itself into a timing model; the next section describes our approach.

5. TRANSFORMATIONS OF VHDL

The VHDL model of the complete controller is quite large, 18000 lines of VHDL in total. As the PPC 755 model itself is already quite complex, adding another complex part into the timing model would render the resulting timing analysis infeasible in terms of space and time consumption. The setting given by the design of the hard real-time system hardware and software determines, which scenarios have to be modeled for *timing analysis*. If, e.g., the design will never use a timer built into the system controller, then this timer doesn't have to be modeled. Other events, which are either asynchronous to program execution (interrupts, peripheral DMA) or are not predictable (ECC errors in RAM, exceptions) in the model cannot be dealt with in the analysis itself, thus they don't need to be in the model. Their effects on execution time has to be considered in a different manner, e.g. by statistical means or by adding penalties based on the computed WCET and worst-case occurrences of events. In the following, we list several issues the analysis can safely ignore or has to deal with differently.

Exceptions/interrupts: if the system controller signals an exception or interrupt to the CPU, the control software will shutdown the complete system. In this case, no bound for the tasks running when the exception happens is needed. We can therefore safely assume that no exceptions are signaled by the system controller to the CPU.

Asynchronous Events: SDRAM refreshes, rewrites due to ECC corrections, PCI slave device DMA transactions and SDRAM to SDRAM DMA engine transactions happen asynchronously w.r.t. program execution. It is not possible to precisely include the events in a static analysis. Instead, their effects can be incorporated into the analysis result afterwards by computing how often they could have happened in the WCET time interval and adding penalties for each occurrence.

Data Path Elimination: We cannot say anything about data paths in our abstract timing model, as we have already abstracted away registers and computations in the CPU model. Thus, we want to abstract away from data paths in the system controller model as well. At the same time, information is not always available precisely. For data accesses, the CPU model cannot always provide an exact address, but rather provides an interval of possible access addresses. Thus the input to our model via the `ts` signal is an interval. This means that for all parts that have to store the address, namely the write buffer and SDRAM open page registers, we also use intervals. In addition, flags that record if the data in a write buffer is valid are now not boolean, but can take the additional value of `?` for "don't know".

From this basis, our task for deriving a timing model from the VHDL code is threefold:

- throw away anything that is not relevant for timing analysis
- abstract away things we cannot represent (precisely)
- translate the VHDL processes into a timing model update program that performs the state update of the system controller unit (unit CSU in Fig. 2)

Most processes in the VHDL code of the system controller are structured as the example code in Fig. 3: one code block handles the initialization when the reset signal is asserted, another (mutually exclusive) code block handles the action that is triggered by the bus clock. The reset action assigns default values to all variables and signals in the design and is not important during normal execution of the code.

5.1 Obtaining the default signal values

When we simplify the VHDL design in a later step, we will remove assignments to signals or variables. Any remaining read-references to these then read a constant value. This value is the *initial value* of the signal or variable, which is assigned when the system controller goes through the system reset. To identify the default values of signals and variables, we evaluate the code segments that are triggered by the system reset signal. 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 the system have a longer initialization sequence, e.g. the SDRAM controller, which goes through a sequence of programming steps for the attached SDRAM chips. Therefore, we have to define a condition on the VHDL components that is true as soon as the system has reached its "ready" state. We then simulate VHDL execution until this condition becomes true and take the values of signals and variables at that point.

5.2 Finding important components by slicing

We only want to keep the parts of the system controller that can influence the timing behavior. We perform forward and backward *slicing* [16] on the VHDL code to find those parts.

A *forward slice* on VHDL determines those instructions in processes that *may* be executed, depending on the values of a set of signals (and/or variables). This set constitutes the slicing criterion. All instructions not in the slice are guaranteed to not depend on any signal (variable) in the set. Naturally, the computation of the slice has to take into account transitive dependencies and dependencies introduced by the sensitivity lists and process restarts of VHDL.

A *backward slice* contains those instructions that *may* influence the value of a signal (and/or variable) from a set of

```
foo:  PROCESS(x,y) IS      foo:  PROCESS(x,y) IS
1:IF x='1' THEN            1:
2:  z<='1';                2:
3:END IF;                  3:
4:w <='0';                 4:w<='0';
5:IF x='1' THEN            5:
6:  r<='1';                6:
7:END IF;                  7:
```

Figure 5: Sequential context elimination

those at the end of process execution (at suspension time). Any instruction not in the slice is guaranteed to not influence the value of the signal (variable).

The timing depends only on the timing of the activation of the transfer signals `aack` and `ta` from Fig. 2[2]. We perform a *backward slice* on these signals. All instructions not in this slice cannot have an effect on the timing of these signals and can thus be removed from the design. For the `ts` signal, we have to perform a *forward slice* to find all parts of the design that may be influenced by a new transaction on the bus and may thus influence the timing. The union of these two slices is the initial set of instructions we have to consider in the remainder. All other instructions never have an influence on timing.

5.3 Eliminating dead components

To reduce the size of the VHDL description, we have to find code sections that are never executed. As stated before, we assume that no interrupts, exceptions, error cases, etc. happen during normal execution. As these events are triggered by asserting signals (external on the bus or internal signals in the controller code), we can simply "hardwire" their values to the unasserted state. For those signals, we look at every place in the program, where the signal is being assigned to (their *definitions*). If a value is assigned that is different from the unasserted state, this assignment will never be executed during normal execution and is thus *dead code*. We can remove not only that assignment from the code, but also its *sequential context*. The sequential context of an assignment is the maximum section of the code that will be executed if and only if that assignment is being executed. Consider the code on the left in Fig. 5 and assume that signal `r` is hardwired to the unassigned state $'0'$. Thus, the assignment at line 6 is dead. The assignment is executed if and only if the signal `x` has the value $'1'$. Since `x` cannot have value $'1'$ during execution, this renders the assignment to `z` dead code (in fact, the whole if statement). This leaves only the code on the right in Fig. 5.

Note that the signal assignment semantics makes it very easy to search for dead code in a process: the value of signal `x` in the condition on line 5 is the same in the *whole* process. Furthermore, we can propagate the condition `x /= '1'` at every place in the design and find dead assignments for `x`.

Another reduction is the removal of variables or signals, that are not used in expressions, but are assigned to. Since nothing depends on their value, those variables or signals can be completely removed. In some cases, this leaves empty if-then-else constructs, where the branches only contained assignments to unused variables or signals. Removing the whole if-then-else construct removes uses of the variables or signals in the condition. Note that one must be careful not to remove such assignments if the signal being assigned occurs in the sensitivity list of a process; the later constitutes a use of the signals value.

[2] `ts` is driven by the CPU.

5.4 Value propagation

For the dead component elimination we have to know if expressions have a constant value. To statically evaluate all expressions as far as possible we can look at all the signals and variables in the program and the places, at which they are defined. If a signal or variable does not have a definition and if it is not an external signal, then we can replace any occurrence of the signal (variable) by its initial value and remove its declaration completely. After doing this, we can try to statically evaluate any expressions in the processes. Some of them will evaluate to a static value, e.g. conditions in if-then-else statements. For these conditionals, we can either remove the 'else' part (if the condition evaluates to `true`) or the 'then' part together with the condition test. This will remove additional definitions of variables, allowing to repeat the propagation process.

5.5 Abstraction

So far, this process follows traditional reduction techniques for HDL simulation or verification. It turns out, that the resulting model is still too large and contains elements which cannot be put to use in the analysis framework due to the abstractions already performed in the CPU part: removal of data elements and approximation of addresses by intervals.

By removing the data paths, the VHDL model shrinks considerably because all data buffers are eliminated, the ECC computation collapsed to a mere cycle delay component and data routing code is removed. The latter also removes uses of control signals.

The addresses in the design are replaced by address intervals, as this is what the other part of the timing model use as approximation. This step introduces non-determinism into the model. Whenever a decision depends on abstracted data and the abstraction is not able to precisely determine the condition[3], then both branches of the conditional execution have to be followed by generating new abstract system states for each outcome. For the write buffer, we would generate one state, where we assume the access hits in the buffer and another one that assumes a miss. This is a safe approximation which introduces more computation costs for the WCET analysis.

Consider the write buffer for SDRAM accesses: the code checks, if a write access hits in the buffer and then updates the buffer and schedules a 64 bit write to the ECC engine. Otherwise, a read of the double word is requested first. The address tag in the write buffer has to be changed to an interval. With address intervals we cannot *always* decide, if an access hits in the write buffer or not. If the address is the interval `[1024,1056]` and the write buffer tag is the interval `[1024,1032]` then the access may hit in the buffer or it may not: under this abstraction we cannot rule out either case. If, on the other hand, the write buffer contained the

[3] Note that this is a dynamic feature: for the write buffer we are able to determine exact hit/miss outcomes if the boundaries of the intervals are cooperative.

interval [512,520], then we know for sure that the access misses in the write buffer. As the PPC 755 shows timing anomalies [10], we cannot simply assume the local worst-case (buffer miss), but have to consider both possibilities in order to be sure to correctly approximate the global worst-case behavior.

The same principle can be used for the page open registers in the SDRAM controller itself and the determination of page hit or miss.

As we removed the data paths completely, another form of abstraction is possible. The controller design passes access requests in some parts via a queue of accesses (access pipelining). If we remove the data path, then entries in some queues are empty. That means that only the *number* of queue entries is important but not the contents of the entries anymore. We can safely abstract the queues to mere counters, which transforms the add and removal operations on the queues to counter increments and decrements.

For accesses to the internal register of the system controller, removing the data paths and hardwiring certain exception signals leads to a simplified structure of access timing. The register handling is distributed among the design, according to the functional block that are controlled by the registers. All access sequences in the different parts are a simple sequence of clock cycles until transfer acknowledges are generated. Thus, the register accesses can be centralized again and abstracted by a simple access timer that counts down clock cycles until the transfer acknowledge is asserted.

5.6 Iteration

The last three steps have to be repeated several times until the VHDL model doesn't change anymore. E.g., in the assignment `x<=y;` assume that value propagation determines that `y` can be replaced by its initial value '0' but this value contradicts the hardwired value '1' for `x`: the assignment becomes dead code, together with its sequential context.

5.7 Integration

The reduced and abstracted VHDL code has to be brought into the form necessary for our timing model framework using delayed and instantaneous signals. That is, we have to transform the VHDL semantics into the sequential unit update of the timing model.

We choose to implement VHDL signals and variables as mere unit components of our framework, cf. [14] for details; intuitively, components are like variables in VHDL. Luckily, the semantics of the `ts`, `aack` and `ta` signals is already VHDL compatible, as they are delayed signals checked only at bus clock intervals. As stated before, the design is mostly edge triggered on the bus clock, with only a few queue-access processes being triggered by queue read and write signals. Therefore, we can safely concatenate the code of the clock triggered processes into one sequential program. To handle the delayed signal assignments, we follow the usual approach in VHDL simulation and introduce for each signal a second copy, the "future" incarnation of the signal. Assignments are then simply assignments to these future incarnations, while references use the actual signal values. Then, after all clock triggered processes have finished, we can copy the future VHDL signal values to the actual ones. After this, the process code for queue accesses is executed, if the signals in the sensitivity lists of those changed, which can be translated as simple conditional checks. As the last step, the delayed signals `ta` and `aack` are generated, if they are active in the system controller components.

Table 1: Benchmark results

Prog. #	VHDL model WCET	VHDL model comp	Simple model WCET	Simple model comp	Ratio WCETs
1	79774	111	94673	120	1.19
2	20006	15	23521	16	1.18
3	48031	5	57138	6	1.19
4	19929	6	23586	3	1.18
5	75436	4	93215	6	1.24
6	21254	15	24856	10	1.17
7	36721	6	43670	5	1.19

Naturally, as accesses on the PCI bus depend on the peripherals connected to the bus which we don't know, the PCI accesses part at the bus uses user supplied information about access timings. That is, the user has to specify how long a PCI transaction, once initiated takes to complete for the devices mapped at specified addresses.

As our WCET analysis implements the actual cycle simulation in C, we had to translate the sequential unit update for the CSU into C. The result is 1300 lines in C; the original VHDL description has around 18000 lines. This transformation is straight forward and could have been automated. As the effort to write the code generator would have been higher than coding the model directly in C, this was not done. See Section 7 for our future plans on this topic.

6. EVALUATION

The timing model has been integrated into `aiT`, the WCET analysis tool with the PPC 755 implementation. To verify the hand coded implementation of the abstracted VHDL model in C, the predictions of `aiT` w.r.t. activation of the bus signals were compared to actual hardware traces on a large number of program runs. As could have been expected, the traces are perfectly predicted by the model.

To assess the costs and the precision gain of adding this detailed model of the system controller to the WCET analysis, we ran the analysis on several benchmark programs. As basis for comparison we designed a much simpler system controller which performs transfers which fixed timings, where the timings can be specified per memory area being accessed. We specified timing that we believe are upper approximations to the SDRAM, PCI and internal register access times.

The benchmarks are a representative set of programs from the avionics area used at Airbus France to evaluate the use of `aiT`. Table 1 gives the predicted WCET and computation time (in seconds) for the system analysis both for the VHDL derived timing model and for the simple model. The memory consumption is not given: we use a garbage collection based memory allocator. The tests were run on a 2Ghz Pentium IV machine with 1GB of main memory. We choose smaller benchmarks: each has between 8kB and 30kB of code. The cache is configured such that only 8kB are available for replacement, the remainder is filled with garbage data. Thus, the benchmarks are big enough to generate cache contention.

Using the VHDL derived timing model for the system controller does not significantly increase the analysis time: it

doubles at most for small execution times. Sometimes, the VHDL based analysis is even faster: this is because it can correctly predict that accesses are performed faster (write buffer hit). On the other hand, the predicted timing bounds increase by 17-24% with the simple analysis. Given that in [13] a precision of the timing analysis of around 30%-50% was given w.r.t. running times, this means that a simple model would overestimate around 50%-70%, i.e. half the predicted time bound. On the other hand, there may be cases when the simple model underestimates due to timing anomalies in the PPC 755.

This shows that the derivation and integration of a timing model from a VHDL description is not only possible but also efficient. The computed timing bounds are not only correct but also more tight than those computed with a different, simpler model.

7. CONCLUSIONS AND RELATED WORK

We presented a way to derive a timing model for a system controller from its VHDL description. The process involved simplification of the design and abstraction of several components by an abstract interpretation. The model was implemented and integrated into the **aiT** tool and compared against a simple model with approximated fixed times for memory accesses. It has been shown that the performance and precision of the new model are indeed very good. The advantage of deriving the model from VHDL are that the modeling source is authoritative.

We are continuing work in this direction. In the course of the AVACS SFB, we are working on a framework that can help to automate the steps necessary to derive a timing model. In this framework, VHDL designs can be parsed and transformed into a representation suitable for data-flow analysis. Slicing and other data-flow analyses, e.g. constant propagation are implemented using the PAG [11] framework. Furthermore, transformations on the VHDL are supported, guided by tools using slicing and data-flow results to automatize the steps mentioned in Sec. 5. Provisions are made to specify which parts of the design are to be replaced by abstractions. Furthermore, an automated code generation tool produces **C** code that can be directly plugged into the **aiT** tool to obtain the timing analyzer.

As a first goal, we are targeting the LEON SPARC processor together with simple periphery in this process.

Another area of active research is the question of reusing and combing models of peripherals. The AMBA bus has gained wide use as a periphery interconnect bus at chip level. Deriving models that all utilize this bus protocol in the same way allows to easily built more complex models for a whole chip from simpler building blocks.

7.1 Related Work

Modeling periphery access times has not played a role in WCET research so far. If varying access times for different memory areas are considered at all, they are assumed to be fixed.

A lot of work has been done in performing simplifications on HW designs both for synthesis or validation by model checking. For verification, usually some parts of the design are black-boxed or hidden behind a functional abstraction [2, 3]. When simplifying for synthesis the goal is usually to remove unused signals from the design to minimize the resulting net lists. This can also include a simplification of the design by hardwiring inputs and obtaining the backward/forward slices.

To the best of our knowledge, no work has been proposed for deriving timing models from VHDL or Verilog descriptions.

8. ACKNOWLEDGMENTS

We would like to thank Jean Souyris from Airbus France for his extensive and outstanding support and the opportunity to work on the modelization of the system controller. We thank Reinhard Wilhelm for his comments on drafts of this paper.

9. REFERENCES

[1] AbsInt GmbH. The ait WCET analyzer. URL http://www.AbsInt.com/ait, 2006.

[2] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-order Processor Verification. In *Proceedings of the Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, California, USA, 1998. Springer.

[3] J. Burch, E. Clark, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2), 1992.

[4] DAEDLAUS Project. Daedalus ist project 1999-20527. URL http://www.di.ens.fr/ cousot/projects/DAEDALUS/, 2002.

[5] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Science and Technology, Uppsala University, 2002.

[6] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[7] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of the first International Workshop on Embedded Software, EMSOFT 2001*, volume 2211, 2001.

[8] C. Hymans. Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 444–460. Springer, 2002.

[9] Institute of Electrical and Electronic Engineers, New York. *Draft IEEE Standard P1076 2000/D3 VHDL Language Reference Manual*, 2000.

[10] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis Using Instruction-Level Simulation Techniques. In F. Mueller and A. Bestavros, editors, *Proceedings of the ACM SIGPLAN Workshop Languages, Compilers and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15, 1998.

[11] F. Martin. PAG – an Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[12] T. L. N. Holsti and S. Saarine. Worst-Case Execution Time Analysis for Digital Signal Processors. In *Proceedings of the European Signal Processing Conference 2000 (EUSIPCO 2000)*, 2000.

[13] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th International Workshop on Worst-case Execution Time, WCET05*, Mallorca, Spain, 2005.

[14] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, November 2004.

[15] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, Massechusetts, 1991.

[16] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.