Principles of Timing Anomalies in Superscalar Processors *

Ingomar Wenzel, Raimund Kirner, Peter Puschner, Bernhard Rieder
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
1040 Wien, Austria
{ingo, raimund, peter, bernhard}@vmars.tuwien.ac.at

Abstract

The counter-intuitive timing behavior of certain features in superscalar processors that cause severe problems for existing worst-case execution time analysis (WCET) methods is called timing anomalies.

In this paper, we identify structural sources potentially causing timing anomalies in superscalar pipelines. We provide examples for cases where timing anomalies can arise in much simpler hardware architectures than commonly supposed (i.e., even in hardware containing only in-order functional units). We elaborate the general principle behind timing anomalies and propose a general criterion (resource allocation criterion) that provides a necessary (but not sufficient) condition for the occurrence of timing anomalies in a processor.

This principle allows to state the absence of timing anomalies for a specific combination of hardware and software and thus forms a solid theoretic foundation for the time-predictable execution of real-time software on complex processor hardware.

1. Introduction

Due to the temporal constraints required for correct operation of a real-time system, predictability in the temporal domain of a real-time computer system is a stringent imperative to be satisfied. Therefore, it is necessary to determine the timing behavior of the tasks running on a real-time computer system. Worst-case execution time (WCET) analysis is the research field investigating methods to assess the timing behavior of real-time tasks.

* This work has been supported by the FIT-IT research project "Model-based Development of Distributed Embedded Control Systems (MoDECS)".

A central part in analyzing the execution time behavior of real-time tasks is modelling the timing behavior of the involved hardware within the analysis process.

The analysis of simple hardware architectures without caches, complex pipelines, or branch prediction mechanisms can be successfully performed with a number of well established WCET analysis methods [19, 21].

Besides these WCET analysis techniques designed for quite simple architectures, a number of solutions exist for architectures that we would like to call "moderately" complex. Moderately complex architectures are very restrictive as far as parallelism in instruction processing is concerned. Still they make use of caches and pipelines [11, 10, 2, 12, 6].

Recent research addresses the WCET analysis of the latest high-performance processor systems that take advantage of numerous performance enhancing mechanisms, especially including instruction-level parallelism and speculative execution. However, these features cause a lot of problems for WCET analysis [7]. The high parallelism and the interference of operations executed in the different units of these processor systems make it practically infeasible to evaluate exact worst-case timing models of reasonably-sized software in a monolithic timing model[1]. Instead, the analysis is usually decomposed into a number of steps that model the timing of different features of the computer system in separation, e.g., in [22]. The partial results of these steps are then combined to compute the final WCET bound for the code under consideration.

It is important to note that the divide-and-conquer strategy as proposed above will only work, if partial results can be safely combined to yield the total result. This means that the scenarios of the solutions found in one step of the analysis must never invalidate a result found in any other step. If this composability property does not hold, the final result of the analysis can never be guaranteed to be safe. This

1 By a monolithic timing model we mean a timing model that models all details of the entire processor system "simultaneously"

property is very important for WCET analysis, as the independence of the different steps of the analysis cannot be guaranteed for all kinds of processor systems. In particular, processor systems that allow for so-called *timing anomalies* to occur, cannot be safely analyzed with a simple divide-and-conquer approach as described above [14]. Due to complex inter-dependencies between the timing of parallel functional units, the safe application of the divide-and-conquer approach is impossible, or gets unmanageably complex.

In order to differentiate computer systems for which WCET analysis is highly complex from those whose timing can be analyzed with reasonable efforts, this paper investigates the phenomenon of the *timing anomaly* in detail. It is the purpose of this paper to establish a criterion that allows the safe identification of hardware/software combinations that can be stated to be free of timing anomalies and thus can be analyzed safely with reasonable effort and without introducing too much pessimism. In this way, this work also forms the theoretic basis for the safe application of newly developed measurement-based WCET analysis approaches [25] for complex processor architectures.

1.1. Contribution

First, we introduce properties required for the safe application of traditional WCET analysis methods on complex processors (Subsection 2.4).

Second, we identify structural sources of superscalar pipelines potentially causing timing anomalies (Section 3). We provide examples, showing that timing anomalies can arise in much simpler architectures than commonly supposed (i.e. even with hardware containing only in-order functional units).

Third, we elaborate the general principle behind timing anomalies and present a general criterion (the *resource allocation criterion*) that provides a necessary (but not sufficient) condition for the occurrence of timing anomalies in a processor. We explain how this principle can be used to identify problematic features of hardware (Section 4).

This simple and strong criterion allows two conclusions: (a) Whenever the processor contains resources that allow dynamic resource allocation decisions at runtime it might be vulnerable to timing anomalies. (b) However, if the actually executed instruction sequence does not cause any of these decisions at execution time, the actual combination of hardware and software can be guaranteed to be free of timing anomalies.

1.2. Structure of the Document

The rest of the paper is structured as follows: In Section 2 we provide an introduction to timing anomalies and give a precise explanation of the relationship of WCET analysis methods and timing anomalies. Section 3 presents the results of our investigation of structural architectural patterns that potentially cause timing anomalies. The observations are used for the development of the *resource allocation criterion* in Section 4. Finally, in Section 5 we critically review our results and the concept of timing anomalies in general.

2. Formulating the Concept of Timing Anomalies

In this section, we outline the concept of timing anomalies in detail. After introducing relevant related work, we characterize the idea behind a timing anomaly. Subsection 2.3 gives a formal definition of timing anomalies. Concluding this section, we elaborate on the impact of timing anomalies on WCET analysis.

2.1. Related Work

The term *timing anomaly* was introduced by Lundqvist and Per Stenström who were the first to discover this kind of problematic timing behavior when using modern processor hardware [14]. They present examples of timing anomalies and identify out-of-order resources as the characteristic feature that causes timing anomalies. Furthermore, they provide interesting ideas for strategies to avoid timing anomalies on the *PowerPC* architecture by modifying the program code.

Schneider developed a WCET analysis method for the *Motorola PowerPC 755* architecture in his PhD thesis [20]. His method handles timing anomalies occurring on that specific architecture; especially problems resulting from preemptions were resolved. He provides various examples of timing anomalies actually occuring on the *PowerPC* platform thus implicitly covering a wide range of anomalies. Further, he was the first who provided a real example for the *domino effect* [14]. The domino effect is an unbounded timing effect caused by different initial pipeline states propagating over loop boundaries.

An interesting approach for handling timing anomalies is the runtime re-configuration of processor features into a safe and predictable mode when timing anomalies are detected during runtime [1]. In this approach, a soft deadline mechanism is used to check the timely execution of code at runtime. When a deadline miss is detected, the processor hardware is re-configured into a safe-mode, thus ensuring that the execution of the remaining task code completes in time. This approach allows to gain the advantages of complex architectures (performance vs. saving energy) while regular operation. In the case timing anomalies being present, the safe-mode guarantees the timely completion of the execution.

2.2. Characterization of the Idea behind Timing Anomalies

The term "anomaly" denotes deviation of actual behavior from expected behavior. Hence, an anomaly never exists alone; it is necessarily embedded in some context (resulting in an "expectation"). Thus, the context has to be an explicit part of the analysis of an anomaly.

The context of timing anomalies is the set of WCET analysis methods. Within the first developed WCET analysis methods [18, 17], hardware modelling is an implicit part. When considering WCET analysis methods for more complex architectures that contain caches and pipelines, it makes sense to partition the WCET analysis process into subtasks and analyze different architectural features separately [22, 4].

The subtask of hardware modelling is deriving bounds of the execution time of some program code sequence executed on a specified hardware architecture starting with a well defined initial state.

Due to the lack of exact state information (e.g., a memory access cannot be precisely classified as cache hit or cache miss), worst-case assumptions are made (for instance, a cache miss is assumed). As Lundqvist et al. have shown, the expected consequences of these assumptions do no hold for dynamically scheduled pipelines [14].

From this viewpoint, a timing anomaly is the unexpected deviation of real hardware behavior contrasted with the modelled one, namely in the sense that predictions from models become wrong. This unexpected behavior could lead to erroneous calculation results by WCET analysis methods when actually implemented. Thus, the concept of timing anomalies rather relates to the WCET analysis modelling process and does not denote malicious behavior at runtime.

2.3. Definition of Timing Anomalies

To describe timing anomalies more formally, we assume a sequence of instructions $(I_1, I_2, ..., I_n)$ where each instruction has a corresponding *latency* [14, 13] t_{I_i} with $i \in 1...n$ in its according *functional unit*. The total execution time for the instruction sequence resulting from the execution of the instruction sequence and its related latencies is denoted by C. The latency of the first instruction t_{I_1} is varied by a value Δt and the future execution time C' resulting from the same instruction sequence but the changed latencies $t_{I_1} + \Delta t, t_{I_2}, ..., t_{I_n}$ is compared with the original execution time C. The difference between both times C and C' is defined to be $\Delta C = C' - C$, more intuitively $C' = C + \Delta C$. Definition 2.1 is semantically identical with Lundqvist's definition provided in [14, 13]:

Definition 2.1 *A* timing anomaly *is defined as a situation where according to the sign of* Δt *one of the following cases becomes true:*

a) Increase of the latency:
$\Delta t > 0 \rightarrow (\Delta C > \Delta t) \vee (\Delta C < 0)$

b) Decrease of the latency:
$\Delta t < 0 \rightarrow (\Delta C < \Delta t) \vee (\Delta C > 0)$

Whenever the term $\Delta t > 0 \rightarrow \Delta C < 0$ or $\Delta t < 0 \rightarrow \Delta C > 0$ is responsible for a timing anomaly as described in a) or in b) of Definition 2.1, we call this *counter-directive impact* of the latency variation Δt and the execution time change ΔC because $sgn(\Delta t) = -sgn(\Delta C)$.

If one from the remaining terms $\Delta t > 0 \rightarrow \Delta C > \Delta t$ or $\Delta t < 0 \rightarrow \Delta C < \Delta t$ is active, we call this a *strong impact timing anomaly*.

This means, a timing anomaly is said to be present when the expected change of the future execution time ΔC resides outside the interval $[0, \Delta t]$ whenever $\Delta t > 0$ or outside $[\Delta t, 0]$ with $\Delta t < 0$, respectively.

Immediately following from Definition 2.1, the *symmetry* property of timing anomalies can be identified: whenever the sequence of latencies $t_{I_1}, t_{I_2}, ..., t_{I_n}$ leads to a future execution time change ΔC using the latency variation Δt, then the future execution time change starting from latency sequence $(t_{I_1} + \Delta t), t_{I_2}, ..., t_{I_n}$ equals to $-\Delta C$ assuming an used latency variation of $-\Delta t$.

Note, that the definition of timing anomalies does not make any assumption about the initial processor state. It is implicitly assumed that the processor contains a particular state before we observe the execution of our instruction sequence. Thus, when we consider a sequence of instructions we always assume some initial hardware state. This can be an empty pipeline state, but there may be also some instructions executed before. The only relevant aspect when considering timing anomalies is that an "almost identical" initial internal state is assumed for the two compared scenarios (except the part of the state that causes the latency variation, e.g., a slightly different cache content).

Some examples for timing anomalies occurring on real hardware can be found in [20].

2.4. The Influence of Timing Anomalies on WCET Analysis

Timing anomalies make the WCET analysis process extremely complex, especially when the safeness of the WCET bound is required. Consequently, hardware allowing timing anomalies can only be analyzed safely using the pessimistic serial execution method [14] leading to useless results due to high overestimation or complex approaches [20].

With the aim being able to use well-established traditional analysis methods, the following - in these analysis methods inherently supposed assumptions - have to be satisfied.

Monotonicity assumption. Hardware involving timing anomalies introduces new problems and difficulties. The main reason identified for this explosion of complexity is the violation of the monotonicity assumption.

The monotonicity assumption means that when uncertain information is processed by a WCET analysis approach, it is assumed that a longer latency for an instruction necessarily imposes an at least equal or longer (bounded by the amount of the latency change) execution time for the overall instruction sequence under consideration. This assumption can be imagined being implicitly included in many WCET approaches and stipulates how uncertain information is incorporated in the hardware modelling process.

In the case of timing anomalies, this assumption does not hold. The situation can be compared with the anomalous behavior in bin packing algorithms [16].

Basic composability assumption. In many WCET calculation methods a basic composability[2] assumption is taken for granted. This assumption denotes the fact that WCET bound values for sub-paths can be safely composed by the WCET calculation method to compute the WCET bounds for the composite paths. However, this property of the validity of the sub-paths WCET bounds is lost when timing anomalies are present.

For instance, when in tree-based methods the WCET bound of a loop is calculated, the loop bound is multiplied with the WCET bound of the loop body. However, when due to a violation of the monotonicity and composability assumption (resulting from timing anomalies) the WCET bound of the loop body is underestimated in a particular execution context, this error will multiply during runtime, dependent on actual input data values. This phenomenon is called an *unbounded timing effect* [14, 13, 20, 24] and results from an insufficiently modelled hardware state. Effects lasting beyond the borders of the analyzed sub-paths are referred to as *late effects* [13] respectively *long timing effects* [5].

3. Structural Patterns of Timing Anomalies

After studying the examples presented by Lundqvist et al. [14], we systematically investigated abstract hardware structures in order to find out whether it is possible to identify minimal structural patterns allowing timing anomalies to arise [24]. The respective results are presented in the following.

First, in Subsection 3.1 and 3.2 we describe the basic terminology. Subsection 3.3 provides examples for timing anomalies that arise when using a single out-of-order resource, as this type of resources has been commonly supposed to be the reason for timing anomalies [14, 13]. Finally, in Subsection 3.4 we present the surprising result that timing anomalies even occur with structurally simpler hardware.

3.1. Hardware Related Terminology

A *superscalar* architecture can be characterized by the following two properties [8]:

1. The superscalar pipeline includes all features of a classical pipeline, but additionally, instructions may be executed simultaneously in the same pipeline stage, i.e., the stream of instructions can be distributed among different *functional units* available at the execute stage.
2. The execution of multiple instructions can be initiated simultaneously in one clock cycle. Such machines are often referred to as *multiple issue machines*. Instructions are *dynamically scheduled* (i.e., the actual instruction grouping is performed at runtime) in contrast to *VLIW* (very long instruction word) architectures [8].

The terms *dispatch* and *issue* are often used in a confusing manner in literature. In this paper, *dispatch* refers to the primary distribution of instructions among the particular subsystems of functional units (including possible buffers) whereas *issue* refers to the assignment of an instruction to a particular functional unit for immediate execution. Dispatch and issue coincide whenever there are no buffers between the dispatch and issue stage.

For our purposes (in order to show the principles behind a timing anomaly), we abstract from real hardware by introducing simplified hardware models. For instance, details of the mechanisms ensuring semantically correct operation of the hardware (e.g. Tomasulo's algorithm [23]) are lost due to abstraction. Indeed, we focus on changes of the resource allocations in these simplified models. Clearly, the abstractions do not cause any loss of generality.

3.2. In-Order and Out-Of-Order Resource Allocations

Resource models are applied in order to model the architectural state of a processor. Resources (e.g. registers, functional units) are *allocated* to instructions.

Lundqvist et al. divide resources into two disjunctive classes [14]:

---

2 An architecture supports *composability with respect to a specified property* whenever the property that has been established at the subsystem level remains valid during subsystem composition [9]. Composability is an important property in architectural design.

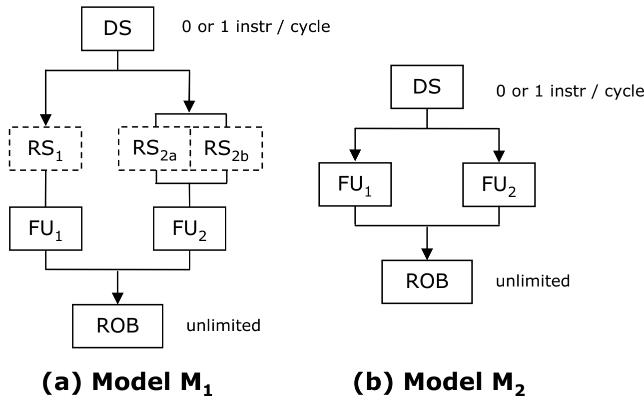

Figure 1. Model M_1 with two reservation stations allowing out-of-order allocation of functional unit FU_2. Model M_2 consisting of two non-equivalent functional units.

In-order resources. "In-order resources can only be allocated in program order to instructions. (...) Examples of in-order resources are registers that must be preserved in-order to guarantee that data dependencies in the program are not violated."

Out-of-order resources. "Out-of-order resources can be allocated to instructions dynamically, i.e. a new instruction can use a resource before an older instruction uses it according to some dynamic scheduling decision. Typical out-of-order resources are functional units that service instructions dynamically (out-of-order initiation)."

3.3. Timing Anomalies caused by Out-of-Order Resources

In this subsection we show examples for timing anomalies that are due to hardware containing of out-of-order resources. Even though we will show later (Subsection 3.4) that the original claim by Lundqvist et al., that the presence of out-of-order resources is a necessary condition for the occurrence of timing anomalies, is too strong, we consider out-of-order timing anomalies very essential and relevant.

We assume the abstracted hardware architecture depicted in Figure 1(a) including the following units: instructions are dispatched by the DS stage to its according reservation station RS_i. Whenever there is no free reservation station buffer available, dispatch stalls. Consecutively, instructions are issued to the respective functional unit. When the functional unit is idle on dispatch, dispatch and issue coincide in one cycle. Whenever an instruction is issued, its according reservation station entry still remains allocated until the instruction has finished its execution and can be moved into the reorder buffer ROB where it is completed as soon as possible (i.e. all previous instructions have completed).

Instruction	Required Functional Unit
A	FU_1
B	FU_2
C	FU_2
D	FU_1

Table 1. Resource requirements of the instruction sequence for model M_1

Model M_1 in Figure 1(a) uses the following issue policy: (i) the functional units serve disjunctive sets of instruction types and (ii) at most one instruction per cycle is assumed to be dispatched.

We constructed examples for timing anomalies involving multiple issue architectures, i.e., multiple instructions are issued in one clock cycle. However, as we will see in Subsection 3.4 and 3.3, even by excluding multiple issues through constraint (ii) in our issue policy, timing anomalies can arise in our hardware models.

In Table 1 the resource requirements for the instruction sequence (A, B, C, D) are depicted. Instructions A and D require functional unit FU_1, while B and C require FU_2.

Figure 2 depicts the execution of this sequence with model M_1 in a timing diagram. The arrows below the instruction labels visualize the instruction dispatch event. The instruction latencies and the latency variation can be obtained from the small boxes on the right side of the diagram. The arrows beside the latencies identify the dependency relationships between the instructions.

The diagram illustrates two situations: the first two rows shown are called case 1 and show the execution of the sequence using the latencies of the box besides. The bars illustrate the utilization of the according functional unit by an instruction, the dotted lines above the bars depicts the reservation station allocation by the instructions.

The two rows below will be referred to as case 2 and show quite the same situation like case 1 - the only difference is that instruction latency t_A is modified by Δt=2 and thus the execution of the instruction sequence results in another resource allocation. When comparing the total execution time of case 1 and case 2, we encounter the occurrence of a *counter-directive timing anomaly*.

For the same instruction sequence, in Figure 3 an example for a *strong impact timing anomaly* is provided.

The latencies of the instructions can been chosen relatively free. When trying to construct examples for timing

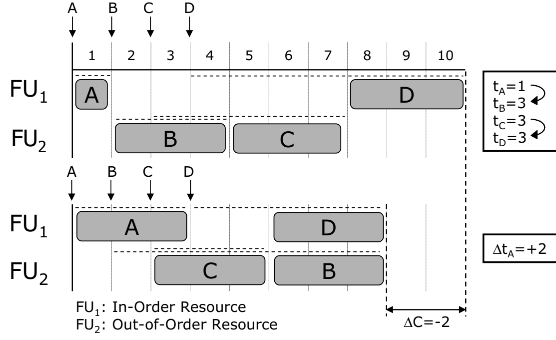

Figure 2. Example for a counter-directive timing anomaly in model M_1

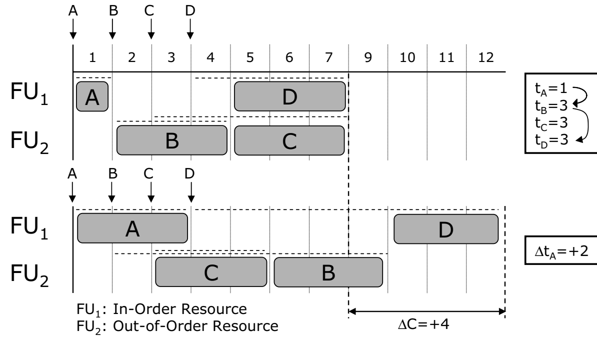

Figure 3. Example of a strong impact timing anomaly in model M_1

Instruction	Required Functional Unit
A	FU_1 or FU_2
B	FU_1 or FU_2
C	FU_1 or FU_2
D	FU_2

Table 2. Resource requirements of the instruction sequence of model M_2

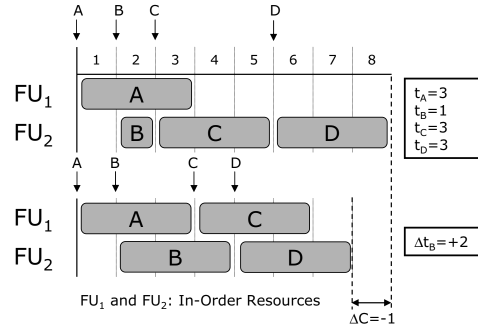

Figure 4. Example for a counter-directive timing anomaly in model M_2

anomalies as simple as possible, it turned out that timing anomalies even can occur for bigger and smaller instruction latencies (examples can be found in [24]). We selected basic latency values of 3 in order to provide demonstrative examples.

3.4. Timing Anomalies caused by In-Order Resources

In contrast to common and our former belief we found that timing anomalies can even occur in hardware architectures that only have in-order resources, like our abstract sample architecture depicted in Figure 1(b).

In model M_2 (overlapping functional units) we consider two functional units serving an overlapping set of instruction types without any reservation stations. FU_1 can serve all instructions of type $c \in IC_1$, FU_2 serves instructions of type $c \in IC_2$ (the set IC_1 contains generic types of instructions for functional unit i). For the instruction classes IC_1 and IC_2 the relation $IC_1 \subset IC_2$ holds. This simply means that FU_2 is able to serve more types of instructions than unit FU_1. Instructions dispatched to FU_1 could also be executed using FU_2, but the reverse is not true. Thus, we have to introduce a new issue policy in order to determine which functional unit should be used when both units are available. Therefore, we extend our issue policy by defining FU_1 as default unit.

Now consider the instruction sequence in Table 2. For each instruction the corresponding functional units are listed that are capable to serve this instruction.

Figure 4 shows an example for a *counter-directive timing anomaly* using model M_2 only employing in-order functional units.

Figure 5 depicts an example for a *strong impact timing anomaly* using model M_2.

Both functional units, FU_1 and FU_2, are allocated to instructions strictly in-order. Still, due to the different capabilities of both functional units, resource conflicts can arise causing timing anomalies.

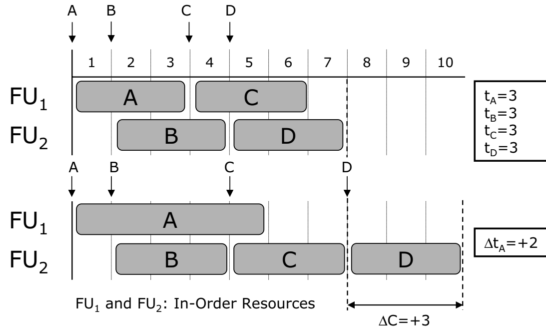

Figure 5. Example of a strong impact timing anomaly in model M_2

4. The Principle Behind Timing Anomalies

The previous section has shown that the differentiation between in-order and out-of order resources is not precise enough for the explanation of timing anomalies. Thus, we investigate in this section the common principle behind the timing anomalies occurring in both models presented in Section 3.

4.1. Terminology for Analyzing Resource Allocations

A *resource allocation* is defined as the assignment of an instruction $I_i \in S$ of a given instruction sequence $S = (I_1, I_2, ..., I_n)$ to hardware resources $u \in FU$ at particular points in discrete time, having FU denoting the set of functional units in the hardware model.

Furthermore, we define

$t_{ent}(I, u)$ ... entry timestamp of instruction I into functional unit u

$t_{I,u}$ .... duration of resource usage for instruction I in unit u (latency) with $t_{I,u} = t_{ent}(I, u) - t_{rel}(I, u)$

Next, we define a resource usage function $use(u) : FU \rightarrow \wp(I)$ describing the set of all instruction instances using a functional unit in an finite execution scenario. Note, that our viewpoint is that of an imagined outside observer thinking of the execution of an "appropriate" execution scenario. Thus, we do not deal with aspects like computability or the halting problem.

Since every functional unit can only serve one instruction at a time, according to the entry timestamps (and also to release timestamps, respectively) an order relation $<^u$ on $use(u)$ is implied.

Concluding, the resource allocation for a given hardware model depends on (i) the instruction type of the instructions in the sequence and their data dependencies, (ii) on the types of instructions that can be executed in each of the functional units and (iii) on the instruction's latencies in their respective functional units.

4.2. Possible Resource Allocation Decisions in Pipelines

In order to have a possibility for comparing resource allocation scenarios we consider tracing the instruction flow through each individual functional unit. As in the previous subsection described, for each resource $u \in FU$, an ordering relation $<^u$ is derived on the instructions passing through this resource considering the entry timestamps.

Second, the value of the latency is varied and for this slightly modified scenario the ordering relation is obtained, too.

Third, these ordering relations are compared whether they differ. In such a case, a different resource allocation has taken place.

Definition 4.1 *A* resource allocation decision *is defined to be possible in a hardware model M whenever an arbitrary instruction sequence S exists that may cause at least for one $u \in FU$ the instruction order relations $<^u$ and $<^{u'}$ to be different from each other ($<^u \neq <^{u'}$) due to a latency variation by Δt of one single instruction in sequence seq. The relation $<^u$ denotes the order relation of the instructions of sequence S implied by $t_{ent}(I, u)$ and the natural mathematical $<$ relation resulting from the execution of sequence S on model M using instruction latencies $t_{I,u}$[3]. The order relation $<^{u'}$ is defined analogously to $<^u$ with the difference that one single instruction has a different latency: $t'_I = t_I + \Delta t$ for $I \in use(u)$ and $t'_J = t_J$ for all $J \in use(u)$ with $I \neq J$.*

4.3. The Resource Allocation Criterion - a Necessary Condition for Timing Anomalies

According to our observations we have made in Subsection 3.3 and 3.4, the idea that possible resource allocation decisions as defined in Definition 4.1 are a necessary condition for timing anomalies to occur, leads to the formulation of Theorem 4.2.

Theorem 4.2 *The possibility of a* resource allocation decision *as defined in Definition 4.1 for a hardware model M is a necessary, but not sufficient condition for timing anomalies to be present.*

3 Instead of t_I, u often t_I is written for simplification reasons whenever the associated FU is clear and unambiguous.

The proof of Theorem 4.2 is provided in Proof 4.3.

Proof 4.3 *In the first step, the necessity condition is proven. Assume we have given an instruction sequence S and two latency scenarios t_I and t'_I with $t_I \neq t'_I$ for one single instruction I. Whenever all functional units are used in the same order, for both scenarios a latency increase may only result in a deferred execution initiation of dependent instructions. Therefore, no additional cycles can be imposed.*

Second, it has to be proven that the occurrence of a resource allocation decision *is not sufficient for a timing anomaly to take place. For this purpose, consider a model like M_2 but having two equivalent functional units. The resource allocation sequence may change, but since both units are equivalent considering the pipeline state this cannot have any impact on further instructions.* □

Finally, Corollary 4.4 provides our final conclusion:

Corollary 4.4 *In processors not allowing resource allocation decisions no timing anomalies can be present.*

Corollary 4.4 directly follows from Theorem 4.2.

5. Discussion and Future Work

The introduced resource allocation criterion allows two important conclusions:

First, the *resource allocation criterion* provides a simple and effective mask to diagnose the potential for timing anomalies. Whenever the change of a single latency can cause different resource allocations in an architecture, then timing anomalies may occur. We elaborated that not the ordering of the instructions on the resources is the precondition, but rather the potential for different resource allocation decisions at runtime. Whenever the processor contains resources that allow such runtime resource allocation decisions (e.g., out-of-order pipelines, functional units serving different instruction types) then latency variations of single instructions might cause timing anomalies further down the instruction stream on this particular type of hardware.

The reasons for latency variations of single instructions may result from caches [14], different operand values (e.g., floating points operations) or branch prediction mechanisms.

The information whether a processor contains hardware features that may cause timing anomalies can be obtained easily by looking into hardware manuals of the processor (e.g., by looking into the manual whether there are multiple functional units serving overlapping instruction sets or containing out-of-order resources).

Second, if the actually executed instruction sequence does not cause any dynamic resource allocation decision at execution time, an actual combination of hardware and software can be guaranteed to be free of timing anomalies. This condition can be ensured by analyzing the instruction streams of the real-time tasks and the resulting potential resource allocation decisions. If necessary, an according software rearrangement like the insertion of additional instructions [14] or an appropriate instruction reordering may be performed.

A simple example would be the execution of a sequence consisting purely of floating point instructions on the *PowerPC* platform. Due to the absence of potential resource allocation decisions (because there is only one single floating point unit in the *PowerPC 755*), no timing anomaly can occur. In this case and the additional satisfaction of the introduced monotonicity and composability assumption (Subsection 2.4), a time-predictable execution environment can be provided that allows to use complex processor hardware together with well-established WCET analysis methods.

In other words, it is possible to create a temporal predictable execution environment for safety-critical real-time code while using the advantage of powerful complex processor hardware. Especially, this is of high importance for mixed-criticality systems [3, 15]. It can be avoided to develop costly new hardware by introducing software architectural elements in those parts of the system that require temporal predictable execution and are subject to WCET analysis.

We extend published solutions dealing with timing anomalies [14] by analyzing the instruction streams and/or reordering the instructions if necessary. Only when needed, additional instructions have to be inserted.

Currently, we investigate simple code transformations (for the code of real-time tasks subject to execution on a specified hardware architecture) with respect to our resource allocation criterion in order to provide a method for "immunizing" a hardware/software combination against timing anomalies.

6. Conclusion

In this paper we explored the fundamental causes for the presence of timing anomalies in superscalar processors.

Instead of considering in-order and out-of-order allocations of processor resources as the sources of timing anomalies, our criterion investigates the potential that different resource allocation decisions may be taken at runtime. The existence of different possible resource allocations has been proven to be a necessary precondition for timing anomalies (Section 4).

The introduced simple and strong resource allocation criterion allows two conclusions:

1. Whenever the processor contains resources that allows dynamic resource allocation decisions, timing anomalies might occur on this particular type of hardware.

2. We provide a simple criterion for testing if a specific software running on given hardware may cause timing anomalies. When an actual hardware/software combination of a real-time task is free of timing anomalies, a time-predictable execution environment (by applying well-established WCET analysis methods) for safety-critical real-time code can be established while using the advantages of powerful complex processor hardware.

It is worth noting, that in-depth knowledge about the phenomena of timing anomalies is not only important for static WCET analysis of complex processor architectures. Our conclusions also form a solid base for the safe application of new measurement-based WCET analysis approaches that we are currently working on [25].

References

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 350–361. ACM Press, 2003.

[2] R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 28th Annual Simulation Symposium April 1995*, pages 172–181, 1995.

[3] B. Dutertre and V. Stavridou. A model of noninterference for integrating mixed-criticality software components. In *Dependable Computing for Critical Applications*, volume 7, pages 301–316, 1999.

[4] J. Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Real-Time Technology and Applications Symposium*, pages 46–55, 1999.

[5] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Acta Universitatis Upsaliensis, 2002.

[6] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[7] R. Heckman, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. In *Proceedings of the IEEE*, volume 91, pages 1038–1054, July 2003.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufman Publishers, 2003.

[9] H. Kopetz. *Real Time Systems*. Kluwer Academic Publishers, 3rd. edition, 1997.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[11] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. volume 21, pages 593–604, 1995.

[12] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *RTSS*, pages 334–345, 1998.

[13] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology Sweden, June 2002.

[14] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium, The 20th IEEE*, pages 12–21, December 1999.

[15] M. Morgan. Integrated modular avionics for next generation commercial airplanes. *Aerospace and Electronic Systems Magazine*, 6(8):9–12, Aug. 1991.

[16] F. D. Murgolo. Anomalous behavior in bin packing algorithms. *Discrete Applied Mathematics*, 21:229–243, 1988.

[17] C. Park and A. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[18] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, Sep. 1989.

[19] P. Puschner and A. Schedl. Computing maximum task execution times – a graph-based approach. volume 13, pages 67–91, Jul. 1997.

[20] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, Germany, December 2002.

[21] A. C. Shaw. Reasoning about time in higher-level language software. Technical report, July 1989.

[22] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *Real-Time Systems Symposium*, 19, pages 144–153, December 1998.

[23] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, number 11, pages 25–33, January 1967.

[24] I. Wenzel. Principles of Timing Anomalies in Superscalar Processors. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.

[25] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Design Automation and Test in Europe*, 2005. to be published.