Towards Composable Timing for Real-Time Software [*]

Peter Puschner and Raimund Kirner
Technische Universität Wien
Institut für Technische Informatik
A1040 Wien, Austria
{peter, raimund}@vmars.tuwien.ac.at

Robert G. Pettit
The Aerospace Corporation
Engineering and Technology Group
Chantilly, VA, USA
robert.g.pettit@aero.org

Abstract

Real-time software is increasing in size and complexity, precipitating the need for advanced modeling and analysis capabilities early in the software development process. One particular concern is the lack of sufficient methods and tools to effectively reason about the timing of software in such a way that software systems can be constructed hierarchically from components while still guaranteeing the timing properties. In this paper, we will discuss deficiencies in current real-time embedded hardware and software structures with respect to achieving our goal of composable and compositional timing behavior. To address these deficiencies, we will then discuss programming methods, code generation techniques, and ideas about hardware and software architectures that should help us in achieving a truly timing-composable and compositional engineering process for real-time software systems.

1. Introduction

Real-time software is increasing in size and complexity, precipitating the need for advanced modeling and analysis capabilities early in the software development process. Yet real-time software design and development practices lag behind the state of the art in non-real-time systems. One particular concern is the lack of sufficient methods and tools to effectively reason about the timing of software in such a way that software systems can be constructed hierarchically from components while still guaranteeing the timing properties. To create such a component-based real-time system, it is necessary that the timing of software units be both composable and compositional.

In this paper, we will show that the hardware and software structures currently used in real-time embedded systems do not support scalable and practical development methods for achieving composable and compositional timing behavior. To address this issue, we will discuss programming methods, code generation techniques, and ideas about hardware and software architectures that should help us in achieving a truly timing-composable and compositional engineering process for real-time software systems.

2. Desired Properties

In defining an approach for a composable and compositional development approach for real-time software systems, we propose that the following properties be met:

- *Support of Hierarchical Development Process*: The development process should support the hierarchical construction of the whole system from lower level components. With respect to timing behavior, composition of timing at higher levels from timing of lower levels should be straightforward.
- *Composability*: Timing of software components should be independent of the software context in which the components execute. That is, the timing of the individual components should hold regardless of the additional tasks introduced to the system and also be portable to other hardware configurations.
- *Compositionality*: The timing of a composite should be derivable by a simple formula from the timing of the components.
- *Predictability*: Timing analysis for the components and resulting system should be accurate and with a reasonable margin.
- *Stability*: Timing variation/jitter should be minimized with respect to both the component and the composed system.

[*]Parts of the research leading to these results have received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 214373.

- *Performance*: New strategies should not lead to significant performance losses when compared to state-of-the-art technologies.
- *Simplicity*: Timing analysis for the components and composite should be of low complexity.
- *Scalability*: The development approach should scale to real-world, industrial systems.

3. Timing in Hierarchical Development

The use of hierarchical development processes keeps the design and implementation of even complex applications simple. Starting from the top and most abstract level, a hierarchical process refines/solidifies an engineering problem and splits it into smaller sub-problems. These sub-problems are then addressed separately, which may involve further refinement, etc. The refinement is complemented by steps of integration in which the results of the lower development levels are combined to realize the services of the higher levels. Most practical development processes include iterations over refinement and integration steps with interleaving phases of construction and evaluation.

In this paper we focus on the question of how the timing of applications can be included into a hierarchical development process for real-time applications. Traditionally, real-time systems research decomposes the assessment of the timing of a computing node into two levels.

- At the *node level*, i.e., the higher level, the concern is that the load on a node can be handled such that the functional and the timing requirements of the application are met.

  In the node design phase, the requirements and the interface specification of the node are used to define a set of tasks and task relations. The worst-case execution times (WCET) of tasks are determined at the task level (see below). Once the task WCETs are known, schedulability tests and schedulers use the execution times and information about task constraints to test schedulability or make scheduling decisions.
- At the *task level* the goal is to write task code that does not exceed a given time limit during execution. A task itself consists of pieces of code (code segments) and the timing of the whole task is a function of the timing of its pieces. Recursively, those pieces can be decomposed again, until we arrive at the smallest code units of the selected programming language. Again, the timing of a larger code segment depends on the timing of its constituents.

The task interface separates the task level from the node level. It characterizes types, possible values and meaning of inputs and outputs, the semantics and the execution time of the task. To build systems that are (nearly) composable, the interactions of the subsystems – tasks in our case – via the interface should be weak, see [8].

What are the necessary properties to realize this divide and conquer approach to the planning and analysis of real-time systems timing, without introducing unnecessary complexity? Or put differently, what are the timing properties of tasks – characterized by the timing interface of the tasks – that make the planning and analysis for a correct timing at the scheduling level as simple as possible?

3.1. Desirable Timing Properties

We have found that the following properties of task execution times promote a low-complexity integration of task timing at the scheduling level:

- *Composability of Execution Times*: The timing of a task on a computer system must not be affected by other software running on this system. In particular, the execution time of a task observed during isolated execution must not differ from the timing of the task once other tasks have been added to the computer system.
- *I/O-Compositionality of WCETs* [7]: The WCET of a composition of tasks should be the sum of the WCETs of the two tasks. More formally, given two tasks (or code segments) A and B that are executed sequentially and a set of possible input data tuples $\mathcal{I}$ the following equation on the WCETs, denoted by T^W, holds

  $$T^W_{A;B}(\mathcal{I}) = T^W_A(\mathcal{I}) + T^W_B(\mathcal{O}_A(\mathcal{I}))$$

  where $\mathcal{O}_A(\mathcal{I})$ represents the possible outputs of A for the input sets $\mathcal{I}$.
- *Stability of Execution Times*: Execution times of tasks (or code segments) should be invariable.

The first property, composability, is mandatory for a meaningful decomposition, and thus for a hierarchical real-time systems engineering process, see [8]. If we do not ensure the composability of execution times then a local change in one of the tasks of a system necessitates that the timing of the whole system, including all other tasks, has to be re-analyzed and possibly re-designed in order to meet all timing constraints. The work reported in [9] shows that timing analysis that takes into account task side effects and still aims at keeping the pessimism acceptably low is highly complex.

For the second property, WCETs are generally lower I/O-compositional [7], i.e., $T^W_{A;B}(\mathcal{I}) \leq T^W_A(\mathcal{I}) +$

$T_B^W(\mathcal{O}_A(\mathcal{I}))$. In case the inequality holds a hierarchical devide-and-conquer analysis will lead to an undesirable over-estimation of the worst-case time consumption of a set of tasks or code segments. This would imply that by adding information about the relationship of the execution times across the borders of A and B, one could improve the result of the combined worst-case timing estimate for $A; B$. Allowing for an analysis across task borders does, however, require an extension of the timing interfaces of tasks (constructs) and makes timing analysis itself more complex, which are both undesirable properties. Thus, if a software development environment ensures that WCETs are I/O-compositional, one can be certain that even a simple hierarchical timing analysis does not lead to a costly over-estimation of resource needs.

Stability of execution times is assumed in most work on scheduling theory. When execution times of tasks are stable, i.e., constant, schedulers and schedulability tests only have to consider a small number of scenarios. Further, we can be sure that no undesireable situations caused by execution-time jitter like timing anomalies can occur.

3.2. Timing of Contemporary HW and SW

Many hardware and software architectures used today do not provide the listed properties.

Composability: Tasks running on the same processor system compete for shared resources like cache memories or branch target buffers whose access times are state dependent. The state of these resources, in turn, depends on the addresses of instructions and data in memory (spatial aspect), on the history of accesses (temporal aspect), and on the strategy that is used to update the state of the shared resources upon access (update strategy).

As a consequence, the memory layout of code and data, as well as the order in which code objects are referenced, influence the state of the computer system and thus the execution times. Conflicts that influence execution times occur between different code segments of a single task, between different tasks running on the same system, and even between different instances of a single task [6].

I/O-Compositionality: As mentioned above, WCETs are generally lower I/O-compositional. The reason for this is that due to data dependent control flow the worst-case scenarios of different code segments or tasks may exclude each other.

In contemporary hardware and software, stability of execution times is impeded by two main factors: First, the execution times of instructions vary when a code segment or task is executed with different inputs. Second, data dependent control decisions in code are responsible for different instruction sequences and memory access patterns, which in turn cause execution-time jitter.

4. Realizing the Desired Timing Properties

We showed that state of the art hardware and software does not allow us to build systems that support a hierarchical timing decomposition. We therefore investigate what type of restrictions and modifications of hardware and software architectures are helpful to obtain the above-mentioned important properties.

Composable Timing To ensure *composable execution times* we have to avoid the competition of tasks for shared resources that are stateful and have state-dependent timing. We see three ways in which this can be achieved.

In a *spatial isolation* the disputed resources are divided into partitions, and the partitions are exclusively assigned to the different tasks. The idea of *temporal isolation* is that every task gets assigned a window of time during which it has exclusive access to the resources so that it cannot be disturbed by other tasks. Finally one can *restrict the state-update strategy* of resources to obviate interferences. One extreme strategy of this type is freezing the state of a resource at a certain state. This is, for example, what cache locking does.

I/O-Compositionality of WCETs To implement this property we have to make sure that input data have no influence on the timing of the code segments of a task. This, in turn means that the algorithms implementing the instruction set of the hardware must have invariable execution times. Further, the input data to the task must not influence the execution times of the software implementing the task. In particular, branches in the control flow (as in alternatives or loops constructs) must not lead to input-data dependent timing variations.

Stability of Execution Times Provided that the execution times of hardware instructions are invariable, the class of programs with constant execution time consists of those programs where all data-dependent alternatives take the same amount of time and the programs that execute the same trace on each execution, i.e., single-path programs (see below).

4.1. Attaining I/O-Compositionality and Stability

Solutions should not only get us a good approximation of the ideal abstraction but should also fulfil the requirements that have been listed in Section 2.

A prerequisite for achieving the I/O-compositionality and the stability of execution times, which are tightly related goals, is that instruction execution times are constant.

Given that numerous CPU architectures exist that provide constant, single-cycle instruction execution, this issue does not require further discussion.

The second main issue in getting I/O-compositionality and stability is that we must find a way to make alternative code sections execute with equal time consumption. This can be addressed in two ways: either find ways to *make the timing of alternative traces equal* or *eliminate alternatives* that would otherwise cause variable timing.

Enforcing Equal Timing for Alternatives Given constant instruction execution times, an input-data dependent execution time can be traced back to some control-flow branch where different alternatives take a different amount of time. This difference can be eliminated by inserting so many single-cycle NOP instructions into the shorter (less time-consuming) alternative that the execution times of the alternatives become equal. A similar strategy can be applied to every loop with a non-constant but bounded number of iterations — insert another loop of identical iteration timing but empty functionality to compensate for non-taken interations in the original loop. So the number of iterations of both loops taken together is always the same. The same goal could be achieved by substituting the NOP sequences by a DELAY instruction, parameterized to stall execution for the equivalent amount of time the NOPs would take.

There are some drawbacks with this approach: First, the insertion of NOP respectively DELAY instruction increases code size. Second, the approach can only be used in architectures where there are no state-dependent mechansims that influence execution times. E.g., in architectures with instruction caches one cannot assign a fixed execution time to a set of instructions, because accesses times of instructions differ for hits and misses.

Eliminating Alternatives If we manage to generate code that follows the same execution trace for whatever input data it receives then obtaining composable and stable timing becomes almost trivial. This is the idea behind the single-path transformation [5]. The single-path transformation is a code tranformation strategy that extends the idea of if-conversion [1] to transform branching code into code with a single trace. Instead of using branches, the transformed code uses predicated instructions – comparable to instructions that hide the branches within – to control the semantics of the executed code. Loops are transformed following a similar idea. Loops with input-dependent iteration conditions are transformed into loops for which the number of iterations is known at compile time. Thereby the original termination condition is moved into the loop body and, again, branches in the control flow are removed by if-conversion (see [3] for details). The temporal I/O-compositionality of the so-constructed single-path code has been highlighted in [7]. As the serialization of alternatives may increase the number of instructions executed during a task run, thus leading to long execution times, it is advisable to use programming methods that aim at keeping the WCET low [4] before applying the single-path conversion.

4.2. Realizing Composability

As mentioned earlier there are three directions to avoid that the timing of a task is influenced by the behavior of other tasks – spatial isolation, temporal isolation, and choosing an appropriate update strategy for stateful shared resources. We will use hierarchical instruction memory to illustrate how these three strategies for realizing composability can be implemented.

Spatial Isolation A strategy for the spatial isolation has been proposed by Kirk et al. as early as 1989 [2]. In this work, the authors propose to split the instruction cache into equally sized partitions. Most of these partitions are statically assigned to tasks for exclusive use. In addition, a limited number of partitions is shared by tasks that need a common memory area to communicate and exchange data. Kirk et al. show that the partitioning of the cache is a means to avoiding temporal side effects between tasks. On the other hand, each task can only use part of the cache, which reduces the achievable speedup of memory accesses and makes the cache less effective.

Temporal Isolation The idea of the temporal isolation of tasks is derived from an ideal task model. In this model each task of constant execution time is assigned a time window which is dimensioned such that the task can run to completion without interruption by other tasks. Alternatively, the preemption of task windows is allowed and has no overhead. In both cases the time needed for the execution of a set of tasks is really the sum of the execution times of those tasks.

This model corresponds to the idealized task model we find in most of the published work on CPU scheduling for real-time systems. We therefore consider the model very important, and although it is generally impossible to implement it directly we will investigate ways and conditions to derive useful approximations thereof.

In the simpler, non-preemptive case, we have two obstacles to temporal composability: First, cache conflicts within a task invalidate the results of an isolated timing analysis of the conflicting parts of the task. Second, external accesses to the cache change the task state between different invocations. Thus tasks are run from different start states, which in turn leads to variations in execution time. In the preemptive case, the situation becomes even more complex as the time instants at which a task is preempted can hardly be predicted. The effects of preemptions on the task state and timing are therefore almost unpredictable (see [6]).

The abstraction of a temporal isolation of tasks can be simulated by storing the task state (cache state) whenever a task ends or the task is preempted and re-storing the task state upon a new invocation, respectively when the task is re-scheduled after a preemption. The overhead necessary for storing/restoring the state of the task should, of course, not become too high. Further, the overhead should be analyzable and controllable during the construction of the computer system. As a consequence, the execution model should be very restrictive and as many control decisions as possible should be taken at construction time.

The system we envisage therefore uses time-triggered control. Further all task sequences and preemptions are planned before runtime. This way the number and cost of store/restore operations and preemptions as well as the overhead for storing and restoring the hardware state to achieve task isolation can be exactly planned and analyzed. Complementing this rigid scheduling scheme with single-path tasks and a single-path operating system yields a computing system which does not only have composable timing but for which the behavior can be precisely predicted.

Update: The Prefetch Solution We have conceived a memory architecture that can help us reduce the overhead incurred by the cache state update needed to implement the temporal isolation of tasks. This strategy exploits the fact that the combination of static scheduling/planning and single-path code execution gives us for every point in time the exact information about which code the computer system will execute next (there are no branches that introduce uncertainty). We propose to use this detailed knowledge about the trace on which the software will execute as a means to control an intelligent prefetch unit that always prefetches the code that will be needed for execution next.

By splitting the hierarchical memory into a limited number of regions (e.g., two regions) and adding some hardware logic to allow for a parallel task execution in one region and state update in the other region the simulation of the temporal isolation could be made very effective. While one tasks executes in one memory region, the state for the next task is in parallel fetched into another region. A dedicated control unit manages the prefetch operations based on the system state and a prefetch plan that has been computed offline, by an automated tool component of the software engineering environment.

5. Summary and Conclusion

The goal of this paper was to discuss programming methods, code generation techniques and ideas about hardware and software that help us in building a structured hierarchical development process for complex real-time applications. We have shown that instruction execution times that are independent of operand values and the use of single-path code are means to generating code that has both I/O compositional timing and a stable (constant) execution time. Due to the single-path structure, the execution-time analysis is simple and well-predictable. Side effects that occur when tasks share restricted resources, in particular cache memories, are the central problem to achieving composable timing behavior. We showed how these side effects can be eliminated by a spatial or temporal isolation of tasks. Replacing the traditional cache-memory architecture by a partitioned hierarchical memory that supports pre-loading of partitions while code is in parallel executed from other partitions offers composable timing as well as the short memory access times that are the key to a good performance.

References

[1] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[2] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proc. 10th Real-Time Systems Symposium*, pages 229–237, Santa Monica, CA, USA, Dec. 1989.

[3] P. Puschner. Transforming execution-time boundable code into temporally predictable code. In B. Kleinjohann, K. K. Kim, L. Kleinjohann, and A. Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[4] P. Puschner. Algorithms for dependable hard real-time systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[5] P. Puschner and A. Burns. Writing temporally predictable code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[6] P. Puschner and M. Schoeberl. On composable system timing, task timing, and wcet analysis. In *Proc. 8th Euromicro Workshop on WCET Analysis*, pages 91–101. Austrian Computer Society, Jun. 2008.

[7] M. Schellekens. *A Modular Calculus for the Average Cost of Data Structuring*. Springer, 2008.

[8] H. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, 3rd edition, 1996.

[9] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proc. 17th Euromicro Conference on Real-Time Systems*, pages 41–48, Jul. 2005.