Time-Predictable Task Preemption for Real-Time Systems with Direct-Mapped Instruction Cache *

Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
{raimund,peter}@vmars.tuwien.ac.at

Abstract

Modern processors used in embedded systems are becoming increasingly powerful, having features like caches and pipelines to speedup execution. While execution speed of embedded software is generally increasing, it becomes more and more complex to verify the correct temporal behavior of software, running on this high-end embedded computer systems.

To achieve time-predictability the authors introduced a very rigid software execution model with distribution being realized based on the time-triggered communication model. In this paper we analyze the time-predictability of a preempting task-activation, running on a hardware with direct-mapped instruction caches. As one result we analyze why a task-preemption driven by a clock interrupt is not suitable to guarantee time-predictability. As a second result, we present a time-predictable task-preemption driven by an instruction counter.

1 Introduction

The number of embedded systems where the correct real-time behavior is of utmost importance is steadily increasing. Embedded systems also perform safety-critical tasks in application domains where human operators would be unable to provide the same system dependability. Human society gets more and more dependent on the correct operation of embedded systems. Thus, it is important to have verification methods that allow us to make sure that a computer system behaves correctly, also in the temporal domain.

Today, embedded systems become increasingly complex. Embedded processors are often equipped with performance enhancing features like caches or execution pipelines. As a result, the state space of the overall system becomes enormous, thus making the analysis of all possible behaviors of a piece of software running on such complex computer systems in general extremely difficult. Consequently, worst-case execution time (WCET) analysis of a piece of code running on such computer systems is already challenging, and the overall timing analysis of the system, including timing effects due to task preemptions requires unmanageably high efforts. This complexity problem not only applies to highly dynamic systems where most behavior is event-driven. Even in case of relatively simple systems with time-triggered task activation and scheduling the problem arises. To avoid the risk of missing a deadline, systems have to be over-dimensioned to reduce the risk of an unexpected temporal resource shortage during operation.

In the light of this unsatisfactory situation we started to think about simpler execution models for real-time systems. In recent work we proposed a system architecture that is time-predictable [7]. As the timing of the actions performed by a computer system depends on both, the software running on the computer and the properties of the hardware executing the software [5], we list software as well as hardware features that in combination allow us to make a computer system time-predictable. In this paper we provide evidence that the proposed architecture, which is summarized in Section 2, is time-predictable. A central point of the proposed architecture is the task activation. In Section 3 we describe our analysis by model

*This work has been supported in part by the European IST Network of Excellence ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

checking of our first attempt, a tasking model that allows for offline-scheduled, clock-driven task preemption. Finally, in Section 4 we analyze a more robust tasking model where preemption points are controlled by an instruction counter, not a clock. For the latter approach we show that it is fully time-predictable even when running on a hardware with direct-mapped instruction caches.

2 A Time-Predictable Application Computer

In this section we review the basic ideas of our aim to define a time-predictable computer system. Further details of the proposed time-predictable architecture including the communication subsystem to develop distributed time-predictable computer systems can be found in [7].

2.1 Hardware Architecture

A central idea of our approach is to obtain time predictability by using a software architecture that has an invariable control flow (see below). As a consequence of using this restrictive software model, we can allow for the use of hardware features that are otherwise considered as being "unpredictable" (e.g., instruction caches) and yet build systems whose timing is invariable. So the idea is to keep hardware restrictions and modifications within limits (e.g., we restrict caches to direct-mapped caches but do not demand special hardware modifications as, for example, needed for the SMART cache [2]). To support our execution model, the following hardware properties have to be fulfilled:

- The execution times of instructions do not depend on the values of the operands.
- The CPU supports a conditional move instruction or a set of predicated instructions that have invariable execution times.
- Instruction caches are direct mapped.
- Memory access times for data are invariable for all data items. (In our view, this is the strongest limitation at the moment. We will try to relax this in future work).
- CPU has a counter that counts the number of instructions executed. The counter can be reset and used to generate an interrupt when a given number of instructions has been completed. If such a counter is not available, as a work-around one could instrument the program code with software-traps to trigger task preemption.

2.2 The Software Architecture

To construct a time-predictable computer system while being not more restrictive about the hardware than explained above, we need to be very strict about the software structure. In fact, the proposed software architecture does not allow for any decisions in the control flow whose outcome has not already been determined before the start of the system. This property is true for both the application tasks and the operating system. Even task preemptions are implemented in a way that does not allow for any timing variation between different task invocations.

2.2.1 Task Model

The structure of all tasks follows the simple-task model found in [3]. Tasks never have to wait for the completion of an input/output operation and do never block. There are no statements for explicit input/output or synchronization within a task. It is assumed that the static schedule of application tasks and kernel routines ensures that all inputs for a task are available when the task starts and that outputs are ready in the output variables when the task completes. The actual data transfers for input and output are under control of the operating system and are scheduled before respectively after the task execution.

An important and unique property of our task model is that all tasks have only a single possible execution path. By translating the code of all real-time tasks into single-path code we ensure that all tasks follow the only possible, pre-determined control flow during execution and have invariable timing. For more details about the single-path translation see Section 2.3.

2.2.2 Operating System Structure

If not properly designed, the activities of the operating system can create a lot of indeterminism in the timing of a computer system. We have therefore been very restrictive in the design of the operating system and its mechanisms.

Predictability in the code execution of the operating system is achieved by two mechanisms. First, single-path coding is used wherever possible. Second, all data that are relevant for run-time decisions of the operating system are computed at compile time. These data include the pre-determined times for I/O, task communication, task activation, and task switching. They

are stored in static decision tables that the operating system interprets at runtime.

Task communcation and I/O is implemented by simple read and write operations to specific memory locations. As these memory accesses are pre-scheduled together with the application tasks, no synchronization and no waiting is necessary at run time.

The two greatest challenges in building a fully predictable operating system were in maintaining time-predictability in case of task preemptions and keeping the activities of the application computer in synchrony with its environment (the rest of the system).

- To maintain the deterministic timing in the presence of preemptions it was necessary to introduce a mechanism that allows for a precise preemption when a given number of instructions have finished execution, i.e., planning preemptions at specific times of the CPU clock turned out to be insufficient (see Section 3).
- The programmable time interrupt provided by the communication system is used to synchronize the operation of the application computer with the global time base [7].

2.3 Deterministic Single-Path Task Execution

As all branches in the control flow of a task may potentially cause variable timing, we translate the code of all tasks into so-called single-path code [6]. This translation can be done automatically. The code resulting from the single-path translation has only a single execution trace, hence the name single-path translation.

The strategy of the single-path translation is to remove input-data dependencies in the control flow. To achieve this, the single-path translation replaces all input-data dependent branching operations in the code by predicated code. It serializes the input-dependent alternatives of the code and uses predicates (instead of branches) and, if necessary, speculative execution to select the right code to be executed at runtime.

For pieces of code with an if-then-else semantics, a similar transformation, called *if-conversion*, has been used before to avoid pipeline stalls in processors with deep pipelines [1]. In addition to code with if-then-else semantics the single-path translation transforms loops with input-data dependent control conditions. This transformation yields loops with constant iteration counts, again with a single execution path [4].

As a prerequisite for the single-path translation of a piece of code, the upper bounds for the number of iterations of all loops have to be available. These numbers can either be computed by a semantic analysis of the code or can be provided by the programmer in the form of annotations, in case an automated analysis is not possible or available.

3 Analysis of Clock-Driven Task Preemption

To complete our proposed predictable software architecture, we need a time-predictable task preemption model. The idea of the predictable preemption is to preempt each task that needs to be preempted at the same points in time in each execution cycle of the static schedule. By doing so, the overall timing of all repetitive executions of the cyclic schedule would also be invariable.

Our original plan was to implement the predictable task preemption by using the CPU clock for task preemptions, i.e., preempt tasks always when the CPU clock assumed one of the values given in the preemption-time tables of the operating system. The motivation was that with exact clock-driven task preemptions the cache behavior will eventually stabilize, resulting in constant code locations where task preemption occurs and thus resulting in constant execution times of each execution cycle.

To get a time-predictable system it is important to avoid by design interrelations between task activation and the other components that may cause non-predictable effects. In the following we describe our analysis of whether task activation by clock-driven task preemption in combination with the hardware and software patterns described in Section 2 will ensure time-predictable operation.

We imposed the execution times of instructions to not depend on the operand values. However, a potential source of timing interrelation with the task activation is the instruction cache, which we assume to be direct-mapped.

3.1 Model-Checking of the Task Preemption

We analyzed the clock-driven task preemption by model checking. The model checker we used was SAL [1] from SRI international.

We built a model of the task preemption and the other hardware and software mechanisms. The basic strategy on the development of the model was to get a quite flexible model so that the model checker is able to automatically find and test different system configurations. For example, the model checker analyzes the

[1] `http://sal.csl.sri.com/`

cache behavior for varying numbers of tasks, instructions per task, task activation times, cache size, and instruction-to-memory mapping.

Applying the model to the model checkers yielded the interesting result that it may happen that for certain system configurations the timing behavior of the system will never stabilize. A useful feature of model checking is that in case a property to be checked is violated, a concrete counterexample is provided.

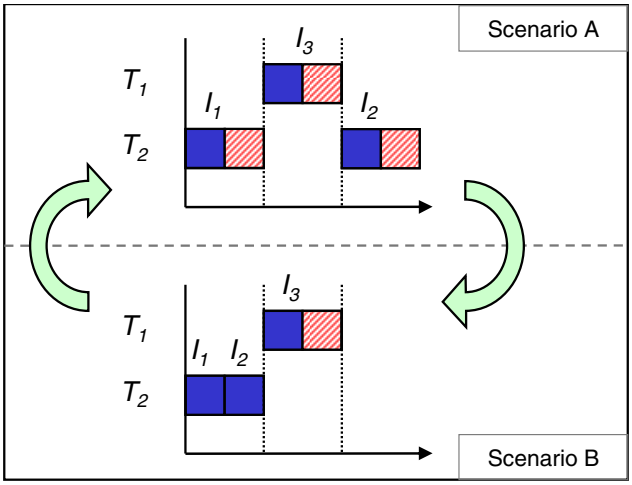

Figure 1. Oscillating Clock-Driven Activation (calculated by the Model Checker)

The counterexample, found violating the stabilization property is given in Figure 1. It is a system configuration consisting of two tasks, T_1 and T_2. T_1 preempts T_2 at the pre-scheduled time marked by the dashed line on the left. T_2 resumes after T_1 has completed its execution. Task T_1 consists of only one instruction I_3, while task T_2 consists of two instructions, I_1 and I_2. Further, the model checker calculated the instruction cache conflict relations shown in Figure 2 of a direct-mapped cache. A drawn-through line between two instructions means denotes a cache conflict, i.e., the two instructions reside in different memory locations that map to the same cache line of the instruction cache. A dotted line between two instructions denotes that the two instructions map to the same memory location, i.e., if one of these instructions is executed after the other instruction without executing a conflicting instruction between, it will be a cache hit. For example, instructions executed in different iteration of a loop behave like this. Also fractions of neighboring basic blocks mapping to the same cache entry can have this behavior. The basic execution time of the instructions is marked by dark boxes. Extra time spent by waiting due to an instruction cache miss is given by striped boxes.

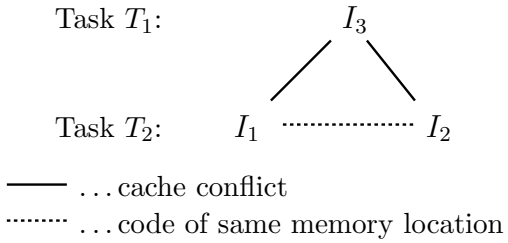

Figure 2. Cache Conflict Pattern

Let us assume the very first activation of our schedule leads to the execution shown in Scenario A. The first access of T_2 to the conflicting cache line leads to a cache miss, and so do the other accesses by T_1 respectively T_2 (The latter misses are due to the order in which the tasks access memory).

When the schedule is repeated, T_2 has a cache hit on the first memory access. So T_2 makes faster progress and the second access to the conflicting address occurs before T_2 is preempted, thus resulting in a hit, too. T_1 then executes with a cache miss, and as T_2 has already completed its two critical memory accesses, the instruction of T_1 remains in cache (see Scenario B). On the next execution of the schedule, T_2 has a cache miss when accessing the conflicting address, and so Scenario A is repeated. Following Scenario A, Scenario B happens again, and so on. The timing does not stabilize. Instead, the task execution times are oscillating. Currently we do not know in general the lowest upper bound of the length of this repetitive pattern of cache behavior.

It is a convenient feature of model checking that typically counterexamples of short length are calculated. But in our case this does not mean that these short counterexamples show unrealistic behavior that cannot occur in systems of real size. For example, the code pattern of the counterexample given in Figure 1 can also cause the same oscillation effect on tasks consisting of much more instructions. Figure 3 shows a possible instantiation of this counterexample. To have the oscillating effect there, it is sufficient that in task T_2 no other instruction between I_i and I_j is in conflict with them and the task activations are at critical time instances.

As a consequence, clock-driven task preemption is not suitable to provide robust and predictable task execution times.

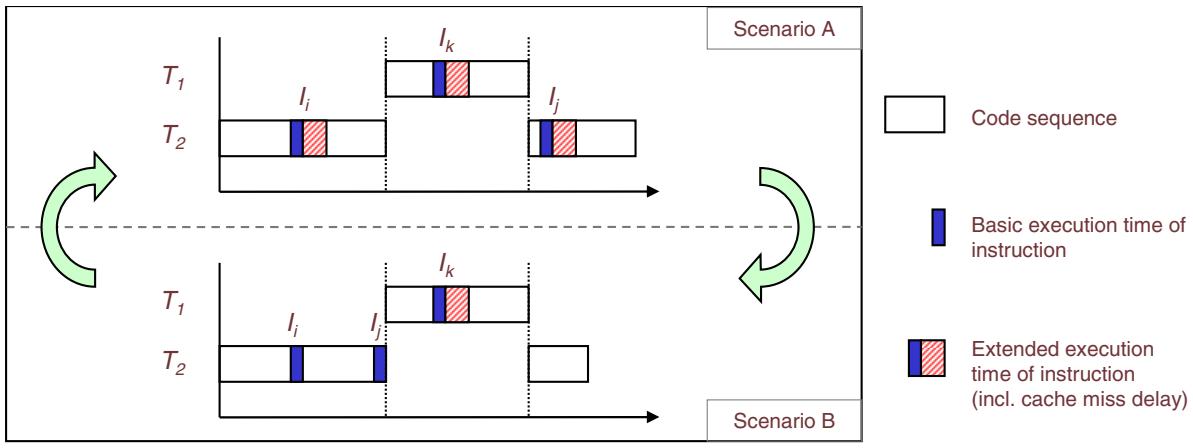

Figure 3. Task Preemption by Clock Interrupt (instantiation of the counter-example given in Figure 1)

4 Predictable Task Preemption

We found that preempting tasks based on the number of instructions executed yields the desired time-predictable behavior [7]. So instead of using a clock we count the number of instructions completed. Preemptions happen when the value of this counter matches an entry in the scheduling table.

Figure 4 shows the schedule for our example, using an instruction counter. Still, the timing of the second execution of the schedule differs from the first, in which we have an initial cache miss. From the second execution on, however, the execution always starts from the same cache state and has a constant execution time.

This stabilization of execution time is guaranteed in systems with direct-mapped instruction caches. Since in direct-mapped caches each memory reference maps to a single cache line, the new cache state after applying a sequence of memory references one time is the same as applying this sequence multiple times. As a result, from the second execution of an instruction sequence, the instruction cache behavior is constant.

Thus, direct-mapped instruction caches can be used in a time-predictable computer system as described in Section 2 in combination with task-preemption driven by instruction counting to ensure time-stability.

5 Summary and Conclusion

In this paper we analyzed the task activation mechanism of a software architecture for safety-critical hard real-time systems. This software architecture uses a static task activation scheme of a cyclic executive. The operating system design and the single-path translation of code support the construction of time-predictable computer systems based on modern, powerful state-of-the-art hardware.

Within this paper we first analyzed why static, offline-scheduled task preemption driven by a clock interrupt is not guaranteed to be time-predictable on a computer architecture with instruction cache. We then demonstrated that time-predictable task preemption can be achieved, if the clock based preemption is replaced by a preemption model that uses an instruction counter to control preemption points. Using this new model all task instances are always preempted after the same number of instructions executed and their execution times stabilize, i.e., after the very first execution the execution times of the tasks become and remain invariable.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Efficient path profiling. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 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] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer, 1997. ISBN: 0-7923-9894-7.

[4] 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

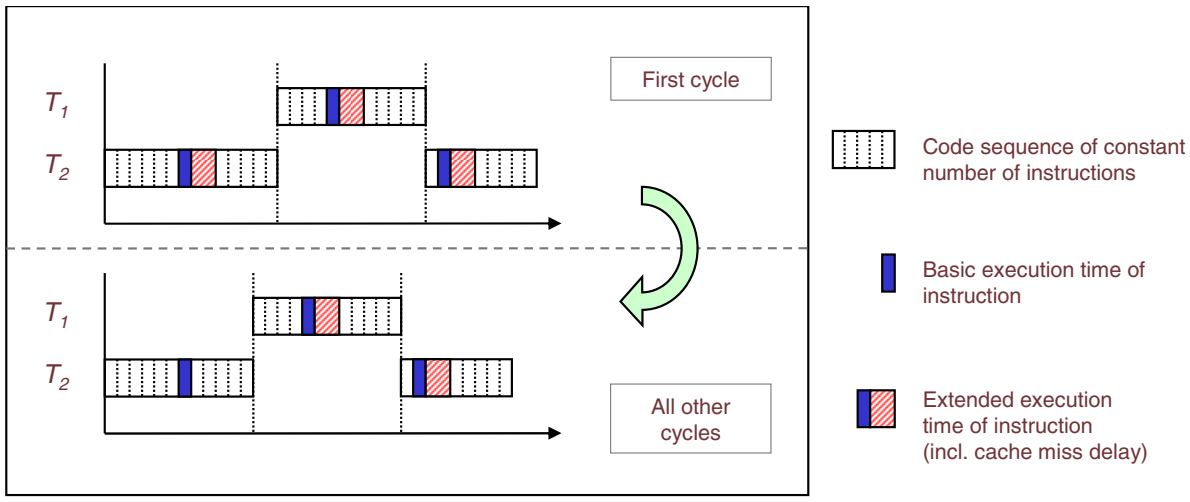

Figure 4. Task Preemption by Instruction Counter (with constant instruction count at each preemption point)

Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[5] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[6] 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.

[7] P. Puschner and R. Kirner. From time-triggered to time-deterministic real-time systems. In *Proc. 5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 115–124, Braga, Portugal, Oct. 2006.