Time-domain determinism using modern SoCs

OSPERT 2019

David Haworth

Elektrobit

Introduction

- What is “time-domain determinism”?
- What causes non-deterministic behavior?
- Overview of the AUTOSAR operating system

What is “time-domain determinism”?

- A deterministic program always produces the same output from the same input
- Deterministic means that what the system does is predictable
- In real-time systems, time is also important
  ... not only *what* output is produced, but *when* it is produced
- Variations in the timing are called “jitter”
- Hence determinism in the time domain (deadlines) as well as the data domain

Compare with spatial interference versus temporal interference (ISO26262)

Causes of jitter

Hardware

- Cache
- MMU behavior - table walks
- Bus contention
  ... in multi-core systems or systems with DMA

Software

- Configuration of time triggering mechanisms; interference
- Execution path variations, leading to execution time variations
- Exclusive areas (critical sections); synchronization primitives
- Cache thrashing

What can we do?

Hardware

- Cache locking when available; see [Borghorst & Spinczyk]
  ... but this doesn't address MMU behavior
- Avoid bus contention - use tightly-coupled memory

Software

- There's plenty of scope for avoiding jitter due to software
  ... but first some background

A very brief introduction to the AUTOSAR OS module

Characteristics

- Static configuration - no dynamic loading of code
- Real-time priority scheduling of tasks
- Entire system (including OS) in same address space

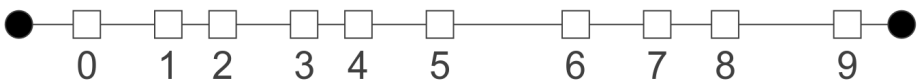
Configured elements

- Executable elements: tasks and ISRs
  ... tasks activated on demand or by means of time-triggering
  ... ISRs activated by hardware request (interrupt)
- Time-triggering elements: counters, alarms, schedule tables
  ... schedule tables and alarms are attached to (driven by) counters
  ... schedule tables and alarms can activate tasks

AUTOSAR schedule tables

A schedule table

- ... has a duration from start to end (black dots)
- ... can be “repeating” (at the end, the ST starts again from the beginning)
- ... has expiry points at configured times between the start and end
- ... expiry points can wake up one or more tasks
- ... expiry points need not be regularly spaced, subject to counter resolution



Comparison of performance variation on current hardware

- Features of OS used in the comparison
- Features of the hardware used in the comparison
- Comparison of performance between cache-clean and cache-preloaded

EB tresos Safety OS

Features

- Compatible with a subset of the AUTOSAR OS module
- Microkernel-based design
- Memory protection for all code (including itself)
- Reprograms (part of) memory protection hardware on context switch
  ... on hardware with MMU: flushes TLB (by ASID)
  ... does not flush cache

Performance

- Performance is quantifiable
  ... microkernel code shas no compile-time configuration dependencies
  ... API execution paths depend only on system state

Hardware used in the comparison

The following four types of microcontroller/SoC are compared

- Infineon TriCore Aurix processor (blue)
  ... typical embedded processor; RAM not cached, just code and read-only data
- ARM Cortex R - TI AR1642 (green)
  ... no cache, MPU programmed in software
- ARMv7 Cortex R - part of Renesas RCAR V3M (black)
  ... fully cached; MPU programmed in software
- ARMv8 Cortex A - part of Renesas RCAR V3M (red)
  ... fully cached; TLB loaded by hardware after page-table switch and invalidation

API execution time

ActivateTask() - without context switch

- The ActivateTask() API places a task in the ready state
  ... the task runs when it becomes the most eligible
- In this case the task has a lower priority so no context switch takes place

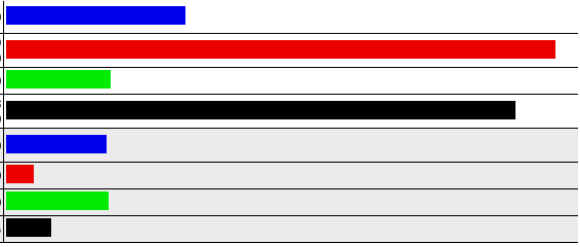
2	ActivateTask default -> LOW (cache COLD)	0	TRICORE_TC29XT	561	5 700	
			ARM64_RCARV3M	83	8 100	
			ARM_AR1642	292	2 520	
			ARM_RCARV3MCR7	2448	11 765	
3	ActivateTask default -> LOW (cache HOT)	1	TRICORE_TC29XT	342	3 430	
			ARM64_RCARV3M	11	1 000	
			ARM_AR1642	295	2 560	
			ARM_RCARV3MCR7	184	810	

API execution time

ActivateTask() - with context switch

- The ActivateTask() API places a task in the ready state
  ... the task runs when it becomes the most eligible
- In this case the task has a higher priority so a context switch takes place

0	ActivateTask default -> HIGH prio, (cache COLD)	0	TRICORE_TC29XT	628	6 440
			ARM64_RCARV3M	199	19 700
			ARM_AR1642	417	3 770
			ARM_RCARV3MCR7	3749	18 270
1	ActivateTask default -> HIGH prio (cache HOT)	1	TRICORE_TC29XT	360	3 630
			ARM64_RCARV3M	11	1 000
			ARM_AR1642	409	3 700
			ARM_RCARV3MCR7	349	1 635



Performance summary

Assumptions

- The cache (and TLB) is in a known state after cleaning (flush/invalidate)
- The cache state depends on the execution sequence

Observations from repeating the performance tests

- From a known state (clean cache), runtime is slow and predictable
- From a known state (filled cache), runtime is fast and predictable

Performance summary

Conclusion

- From an unknown cache state runtime is somewhere between clean and filled limits
- If the execution sequence is predictable, timing will be fairly predictable
- If the execution sequence is unpredictable, timing will be unpredictable
- Unpredictability of the software causes jitter in itself
- Unpredictability of the software amplifies hardware unpredictability
  ... on modern SoCs, hardware jitter might be more than software jitter
- It is therefore essential to control how the software behaves.
- This control is a fundamental feature of the system design
  ... it cannot simply be added later in the project

A study of a real project

- Description of system; what problems were experienced
- Causes and solutions; quick fix
- A deeper look into the system design

Data from a real project

Description

A fairly typical automotive application:

- A microcontroller based on ARM Cortex R4; single core, with cache
- Time-triggered scheduling using multiple schedule tables to activate the tasks
- Longest schedule table: 100 ms
- The main reported problem in the application was overall CPU load, not jitter
  ... maybe jitter wasn't important
  ... or perhaps just secondary to the CPU load problem
- The size of the RAM footprint was also a problem

Data from a real project

Causes of the problems

- Excessive interrupt load for the schedule tables
- The cache didn't perform as well as expected (h/w vendor's finding)
- The application ensured data consistency by making copies of data
  ... which contributed to the worse-than-expected cache performance
- Synchronization APIs (mutual exclusion and interrupt locks) used to ensure consistency while copying
  ... which contributed to the overall CPU load

Data from a real project

Solutions

- Excessive interrupt load and interference between the schedule tables was eliminated first by combining the multiple schedule tables into a single schedule table.
- Interrupt load reduced still further by “chaining” the tasks rather than activating all at the EPs
  ... this reduced the OS overhead and jitter at the EPs
- The chaining allowed sets of tasks to be assigned the same priority
  ... tasks with equal priorities can share the same stack region
  ... reduces the RAM footprint and improves cache performance slightly
- A couple of minor optimizations implemented in the microkernel

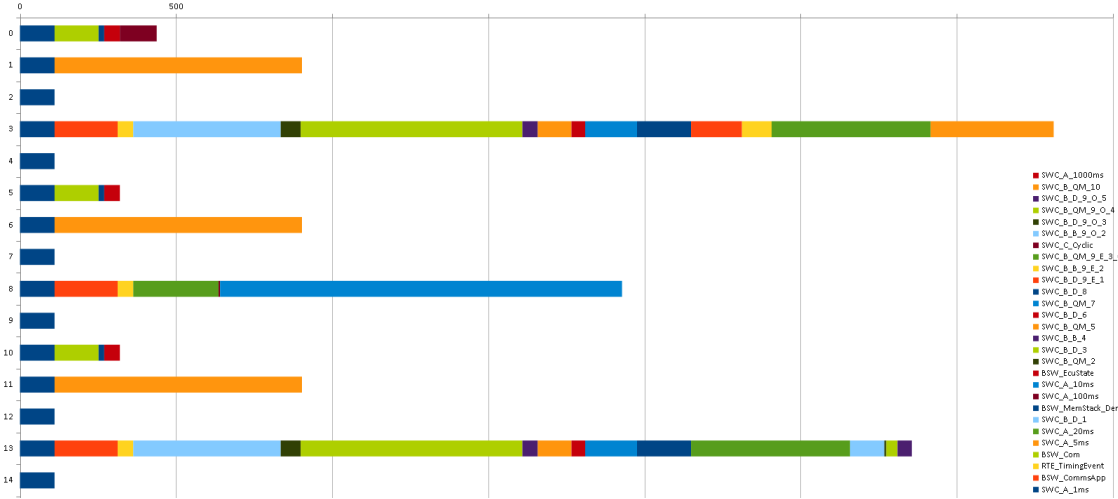
Data from a real project

End of EB's involvement

- This was a real project with real timescales
- The measures described above were sufficient to allow the application to perform acceptably
  ... so no further improvements were made. EB's involvement ended.
- However, the system was not well designed
  ... let's look at what we could do better ...

Elektrobit

Analysis of task execution versus time



Analysis of task execution versus time

Observations

- All tasks are activated by expiry points and execute by priority thereafter
- Expiry points configured at regular intervals
- OS overhead depends on number of activations at each expiry point
  ... the start time of the first task varies by expiry point
  (partially solved by task chaining, as mentioned earlier)
- Execution extends past 1000 us, next expiry point interrupts executing task
  ... leads to preemption; time of preemption is unpredictable
  ... leads to necessity for mutual exclusion (e.g. interrupt locks)
  ... causes more variation in the start time of the first task
- This project also has device ISRs (not shown) that can occur at any time

Analysis of task execution versus time

Summary

- All this variability means that it is impossible to predict the execution sequence
  ... and therefore the cache state
  ... which leads to even more variability in the timing
  
- How could we improve the predictability?
- Let's take a journey back in time, to the late 1970s and early 1980s ...

Comparison of the real automotive project with a historical project

- Description of the historical project
- Comparison with the modern project; similarities and differences
- Suggested improvements to the design of the modern project

Elektrobit

Here's a photo of an aeroplane ...

Photo ©Mike Freer, licensed under GNU Free Documentation License v1.2

Jaguar ACT flight control computer

Hardware characteristics

- Quadruplex system: four identical FCC boxes, most sensors and actuators quadruplex
- 16-bit processor based on Am2900 bit-slice ICs
- No cache, processor behaves fairly predictably, except ...
- Input and output exclusively by DMA
  ... no interrupts to transfer data to peripherals
- Execution time varies with the amount of DMA activity
  ... difficult to predict execution times and start times of functions exactly

Jaguar ACT flight control computer

System design

- System timing by means of a “master reset” (essentially an NMI) every 2.5 ms
  ... signals start of a “frame”
  ... complete processing cycle of 32 frames = 80 ms
- Minimal “operating system”:
  ... housekeeping, synchronization, fault reporting from previous frame
  ... run application “tasks” that are scheduled for the current frame
  ... on completion, wait for next reset; failure to reach this deadline is a serious fault
- Predefined DMA activity also starts with the reset:
  ... inputs, outputs and inter-FCC communication depends on frame number
  ... input data will be used in used subsequent frames
  ... in a few exceptions, at the end of the input frame
  ... output data has been computed in earlier frames

Jaguar ACT flight control computer

Software design

- 32 frame functions - one called each interval
- Each frame function calls a sequence of “link” functions
- Each link function performs a sequence of computations
- Distributed of link functions among frames depends on frequency/phase requirements
  ... e.g. a 10 ms link function would be called every fourth frame
- Distribution of link functions within a frame depends on data flow
  ... also considering arrival time of inputs that are used in the same frame

Jaguar ACT flight control computer

DMA-centric scheduling

- DMA schedule is predefined; not controlled by software running on the CPU
- DMA schedule places requirements on software schedule
- This way of using DMA means I/O timing behavior depends only on “master reset”
- Input data is always available when it is needed by the software
- Output data is always ready for transfer by DMA
  ... unless an earlier frame misses its completion deadline
- Jitter is eliminated

Jaguar ACT flight control computer

Execution time

- Theoretical worst-case execution time computed for each frame
  ... used an average instruction time that took worst-case DMA activity into account
  ... design rule: no more than 2.2 ms including housekeeping overhead
- In practice, computed WCET was shorter than 2 ms for most frames
- Measured real execution time for each frame
  ... in simulator: somewhat less than computed
  ... in real system: even less than in the simulator
- Modern tooling could probably compute worst-case execution time more accurately
- The average instruction times were probably too pessimistic
- The 2.2 ms guideline was almost certainly too pessimistic
- Could probably achieve a higher CPU load - but it wasn't needed

Comparison with the automotive project

Parallels

- The reset is essentially the regular schedule table interrupt
- The frame functions are essentially the expiry points of the schedule table
- The link functions are essentially the tasks

Differences

- 1 ms tick time versus 2.5 ms in FCC; but the modern processor is much faster
- Tasks are activated, versus links being called (necessary for memory protection)
- Expiry point allowed to interrupt tasks from previous expiry point(s) (bad practice)
- Device ISRs to interrupt tasks at unpredictable time (bad practice)
- Monitoring of deadlines and CPU load distribution is not possible

Improvement of the automotive project

Suggested application improvements

- Introduce and enforce a rule
  ... all tasks at an expiry point must complete before the next expiry point
- Add monitoring to measure worst case time for each EP's activations
  ... and to detect and report deadline violations
- Move the tasks that extend beyond the tick interval into the next EP
  ... repeat as necessary to eliminate deadline violations
  ... split up long tasks to allow them to be distributed

Improvement of the automotive project

Suggested application improvements

- Use DMA for I/O where possible
  ... I/O timing doesn't depend on software; DMA doesn't affect CPU state
- On a multi-core SoC, use a core as a dedicated I/O processor
  ... essentially a very sophisticated form of DMA
- Where DMA not possible, eliminate ISRs and use device polling instead
  ... alternatively, restrict ISRs to defined windows using interrupt source control API

Improvement of the automotive project

What do we gain?

- Critical sections are no longer needed; remove them
  ... eliminates jitter of expiry point interrupt
  ... reduces CPU load
- Most of the data copying is no longer needed; remove it
  ... reduces cache thrashing
  ... reduces CPU load
- Execution path after an expiry point is more predictable
  ... remaining differences are code paths in functions, but that's another story
- Cache state at any time is more predictable
  ... if necessary, can be improved further by cleaning at each EP
  ... cache maintenance may be needed for DMA inputs and outputs

Back to the future

- Improvements and new features in operating system
- Did we throw away a suitable OS?
- Summary and conclusion

How could we improve the operating system?

New features

- Expiry point “frame”; only activate first task
  ... each subsequent task at the EP is automatically activated when predecessor terminates
- Deadline monitor to detect frame overrun
- Measure execution time of each frame, store the longest
- ISR window feature
- Configurable cache maintenance as an activity in an expiry point

What about OSEK's time-triggered operating system?

Features of OSEKtime

- A single dispatcher round of configured duration
- An array of dispatcher events at configured times during the dispatcher round:
  ... activate a specified task or
  ... enable a specified ISR or
  ... task deadline; report an error if the specified task has not completed
- Tasks can be activated multiple times in the round
- Double-activation is a deadline error
- A newly-activated task pre-empts an executing task

What about OSEK's time-triggered operating system?

Comments on OSEKtime features

- No possibility to start a chain of tasks; no API to chain (or even activate) a task
- Task pre-emption causes the same uncertainty as in AUTOSAR OS
  ... could attempt to eliminate preemption by timing
  ... will result in lots of wasted time between tasks
- The ISR enable feature doesn't disable the ISR until it occurs:
  ... arrival time is unpredictable

Times have moved on since OSEKtime was specified. Its features don't really provide solutions to the problems we face on modern SoCs, and it doesn't offer features like memory protection that are essential for modern automotive applications.

Summary and conclusion

Summary

- Typical industry practice results in systems whose temporal behavior is not predictable
- Use of higher-performance SoCs can make the predictability worse
  ... even when the system is 100% time-triggered.

Conclusion

- Good system design - hardware *and* software - is essential for determinstic behavior
- New OS features in AUTOSAR could make the implementation easier

Useful links

- Borghorst & Spinczyk :
  CyPhOS - A Component-Based Cache-Aware Multi-core Operating System

  https://www.betriebssysteme.org/wp-content/uploads/2019/03/abstract-borghorst.pdf

  (abstract in German; full text in English available from Springer Professional)
- AUTOSAR OS specification

  https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_OS.pdf
- OSEKtime specification at archive.org

  https://web.archive.org/web/20110414223336/http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf

Elektrobit

Useful links

- Am2900 devices

  `https://en.wikipedia.org/wiki/AMD_Am2900`

- Jaguar ACT photo

  `https://commons.wikimedia.org/wiki/File:Sepecat_Jaguar_GR1-ACT,_AN1403546.jpg`

Disclaimer

Information about the Jaguar ACT flight control computer comes from personal recollection. The details may be inaccurate - the memory of events of 40 years ago is never perfect - but the general idea is correct.

The concepts that were developed were used in the Eurofighter Typhoon and in the Boeing 777 FBW.

The design principles probably became standard practice in the avionics industry, except when they forgot about them (*cough* MCAS *cough*).

Elektrobit

Jaguar ACT links

- Jaguar ACT at RAF museum

  https://www.rafmuseum.org.uk/research/collections/sepecat-jaguar-act-demonstrator/

- Jaguar FBW at Rochester Avionic Archives

  https://rochesteravionicarchives.co.uk/platforms/jaguar-fbw

- A photo of the aircraft taking off

  https://imgproc.airliners.net/photos/airliners/1/4/9/0763941.jpg?v=v40

- Two articles containing information about the software design

  https://apps.dtic.mil/dtic/tr/fulltext/u2/p002713.pdf

  https://apps.dtic.mil/dtic/tr/fulltext/u2/a161950.pdf starting on page 10

Get in touch!

david.haworth@elektrobit.com
www.elektrobit.com