Tailoring and Optimising Software for Automotive Multicore Systems

Torsten Polle and Michael Uelschen

Advanced Driver Information Technology GmbH
Robert-Bosch-Straße 200
31139 Hildesheim
{tpolle,muelschen}@de.adit-jv.com

Abstract. The software architecture of embedded systems is heavily influenced by limitations of the underlying hardware. Additionally, real-time requirements constrain the design of applications. On the other hand, embedded systems implement specific functionalities and hence give the designer the opportunity to optimize the system despite of limitations. Multicore systems compromise the predictability of real-time requirements. Again, with the knowledge of the application the software design can benefit from the multicore architecture. This paper discusses how to decide on software design based on use-cases and shows new avenues how to efficiently implement the design with an example.

Keywords: Multicore, Scheduling, Automotive Embedded Systems, Producer-Consumer Pattern.

1 Introduction

Starting with a multiprocessor architecture at the super computer and main frame classes in the 1970's and 1980's the technology process got improved, which enabled chipset manufacturers to layout several cores on one die. The year 2005 was the inflection point when the increase of the clock frequency got restricted around 4 GHz, primarily because of huge power consumption and it was then, when multicore technology hit the consumer market.

Now the multicore architecture is entering the embedded automotive domain. The first adopters are driver information systems followed by multicore systems for classical electronic control units (ECU). Head units like car navigation systems combine functionality from the consumer electronics market (like MP3 or video playback functionality) with increasing complexity as well as from the automotive domain (like CAN or MOST networking).

Therefore an ever-increasing demand for processing power has to be satisfied. Although a multicore architecture has the potential to sate this demand, the change of paradigm forestalls the efficient usage of the provided processing power. Certainly, embedded systems can borrow from the approaches taken in the consumer electronics market. But there are differences like priority-based scheduling and optimisation techniques, which have to be taken care of. Therefore it will be interesting to see if

T. Margaria and B. Steffen (Eds.): ISoLA 2008, CCIS 17, pp. 71–81, 2008.
© Springer-Verlag Berlin Heidelberg 2008

and how already established principles to use parallelism can be re-used in the automotive domain.

This paper extends [1] by covering the entire software design process. The starting point is six "meta" use-cases. These use-cases are in contrast with those that describe what the developed software is doing; they rather describe how to change the software. A multicore architecture offers several cores, on which software can be executed. The question "Which parts of the software are executed where?" is a new aspect of software design. The use-cases offer criteria on such decisions. Finally, it is shown that well established optimisation techniques have to be replaced for multicore systems.

2 Use-Cases for Automotive Multicore Systems

For the investigation of use-cases it is assumed that a functional single-core system is already available. This means migration of legacy code is a major requirement. This is important especially for considerations outside the academic world since car makers and suppliers usually cannot afford to start such complex systems from scratch.

In this paper the focus is on homogenous multicore systems [2]. The shared access to the memory subsystem is symmetric and peripherals are identical from individual core point-of-view. Heterogeneous architectures, e.g. constituting core connecting with a DSP, are out of scope of following sections.

The major observed trends or use-cases for multicore in the automotive domain can be classified as:

2.1 Use-Case 1: Deployment of New Functions

For the realization of upcoming features additional computing power is required. This is to support technology advancement, on one hand (like the European satellite navigation system Galileo). On the other hand, more rigid laws and standards (like the European eCall) have to be implemented. The use case also covers the refactoring of existing application, e.g. in order to increase performance or precision.

2.2 Use-Case 2: Redundant Systems

For automotive control units that require high safety and reliability, the use of multicore is a cost-efficient approach. However a failure of the underlying hardware will affect the entire system. Further investigations are necessary to verify which application can be designed for this approach.

2.3 Use-Case 3: Concentrating of Functions

Since cars are equipped with up to 100 ECUs, the minimization of the amount is a strong objective. This is driven by many factors, hardware costs being one of them. Also the configuration management and therefore the test and release process gets less complex and time-consuming as the amount of possible combinations of software and hardware versions goes down.

2.4 Use-Case 4: Convergence of Domains

Beside the requirement to reduce the amount of devices in the car it turns out that the different automotive domains are getting closer to each other. In an example by Toyota [3] the availability of the current position as well as the calculated route to some destination can be used for other applications in the car like vehicle control. The domain of driver information gets closer to driver assistance or to the powertrain domain. Functionalities from the consumer electronics domain are getting into the car. This leads to several challenges since the consumer and automotive domain follow different rules, e.g. innovation cycles [4]. The requirements focus mainly on 2 fields: visualization (human-machine-interface) and connectivity (mobile phones, media player). Solutions known from the desktop world will be integrated in future infotainment systems [5].

2.5 Use-Case 5: Architecture Harmonization

The introduction of two or more identical cores reduces the need of specialized hardware like a DSP. This approach is based on the assumption that a harmonized hardware as well as software architecture increases flexibility. This also might simplify the usage of tools (as compilers and debuggers) and programming languages.

2.6 Use-Case 6: Parallel Algorithms

Often algorithms can be parallelized to solve some problem more natural. Having a hardware environment that supports parallelism avoids sequential refactoring of a parallel algorithm. In contrast to fields of high performance computing the nature of algorithms for car applications is different. For example it might be beneficial to parallelize the route calculation for getting shorter computation cycles in a navigation system. But analysis of the current devices has shown that the access to the external map data (e.g. on CD or DVD) rules the system performance.

Parallelism on control level utilizes (light-weight) threads, as provided by the underlying real time operating system (RTOS) to get computing done concurrently. In order to schedule these threads for execution on the individual cores, the kernel of the operating system has several options. It turns out that different mechanisms need to be considered for partitioning software on embedded systems compared to the desktop world.

3 Scheduling

Scheduling in desktop or server systems for user level programs is round-robin in nature, to give enough justice to all user programs under execution. This scheduling mechanism is non-deterministic as the operating systems (OS) steals control from threads to realize round-robin scheduling. In case of embedded systems, the scheduling is generally priority based pre-emptive. And in such scheduling schemes, an application may misbehave or lead to data race conditions if more than two threads of different priority go to RUN state at the same time.

3.1 Symmetric Multiprocessing

In case of priority-based, pre-emptive scheduling on SMP kernels, the kernel provides flexibility to decide, which thread runs on which core. Dynamic load balancing is one of the properties of SMP mode.

The advantages of symmetrical multiprocessing are:

- The operating system manages automatic dynamic load balancing. The OS decides how to distribute threads across processors/cores to assure effective usage of all processors/cores.
- Inter-core communication can be implemented very easily using inter-processor interrupts as memory is visible to all processors/cores. No explicit message passing mechanism is required.

The drawbacks of symmetrical multiprocessing are:

- Deterministic behavior gets degraded because of automatic load balancing. Also the load balancing algorithm consumes more CPU time as the number of processors/cores in the system, increases.
- Cache coherency, synchronization mechanisms and shared data, limits application scalability.
- Synchronization among threads compels execution across cores to become sequential.

3.2 Asymmetric Multiprocessing

SMP is the de-facto standard of multicore server and desktop operating systems [6]. For the embedded world also other architectures are under consideration. On systems with asymmetric multiprocessing (AMP) different operating systems or several instances of the same are executed in parallel sharing the same physical hardware.

In this case load balancing is not supported and the communication between the cores is costly. On the other hand porting of existing single-core applications is less difficult.

The usage of several operating systems on a multicore system requires a communication channel for the synchronization of shared resources (e.g. memory or I/O). After virtualization was successfully introduced to the server and workstation domain, there are initial attempts to apply similar concepts to embedded systems [7, 8].

3.3 Hybrid Multiprocessing

A promising approach is hybrid architecture: running just one RTOS but putting restrictions on the scheduling strategy. Such hybrid configuration is supported by the clever design of the scheduler logic and its associated data structures available as a part of the kernel:

- Single Core. A core is configured in the way that a set of threads is defined to run exclusively on a specific core. Neither migrating of threads from this, nor to this core is allowed. The scheduling strategy on this core is priority-based. Load balancing is not possible.

- Execution Order Preserving. Threads are pooled to a partition with dependencies. The scheduler assures that the execution order of the depending threads is kept. If two threads have no dependencies, the scheduler is allowed to run these in parallel for load balancing reasons.
- Core Affinity. A thread is bound to a specific core. The scheduler does not migrate the thread for execution even if a different core is idle. Other threads can migrate to this core and will be scheduled priority-based.

The most flexibility is given if the RTOS supports the combination of SMP and the described AMP modes. For example, on a three core system, the system designer can configure at boot-time one core as one scheduling unit and the remaining two cores together, as another scheduling unit.

Another configuration for a three core system could be that each core is treated as one scheduling unit. Here the situation tends to be like an AMP system. Such configuration gives flexibility to port existing legacy applications from single-core systems to multicore systems.

Currently there are no standards on scheduling for multicore available. Some RTOS like eT-Kernel [9] or Neutrino [10] support both these flavors of SMP and AMP.

4 Application Binding

Without detailed understanding of the system requirements no general rules can be given how to apply the different multiprocessing modes to the described use-cases. Therefore in this section only examples can be given how to bind an application to the cores. A major open issue is the predictability of real-time requirements.

Based on the different use-cases the software partitioning varies. In contrast to multi-purpose systems like desktop computers or the high performance computing domain the symmetric multiprocessing design may be inappropriate. Table 1 summarizes the mapping of the use-cases to different software partitioning.

A hybrid multiprocessing approach enables the designer to keep dependencies and timings of the existing application and combine new functions. If algorithms can be parallelized and a short execution time is required, then a pure symmetric scheduling strategy fits. A hybrid approach does not gain additional benefits.

Table 1. The symmetric multiprocessing approach is most beneficial for load-balancing reason or if an algorithm is following the single program-multiple data pattern

Use Case	Asymmetric	Symmetric	Hybrid
1: Deployment of new Functions		**	***
2: Redundant Systems	***		**
3: Concentrating of Functions	**		***
4: Convergence of Domains	**		***
5: Architectural Harmonization	*	**	***
6: Parallel Algorithms		***	

Rating: * suitable with restrictions; ** suitable; *** most benefial approach

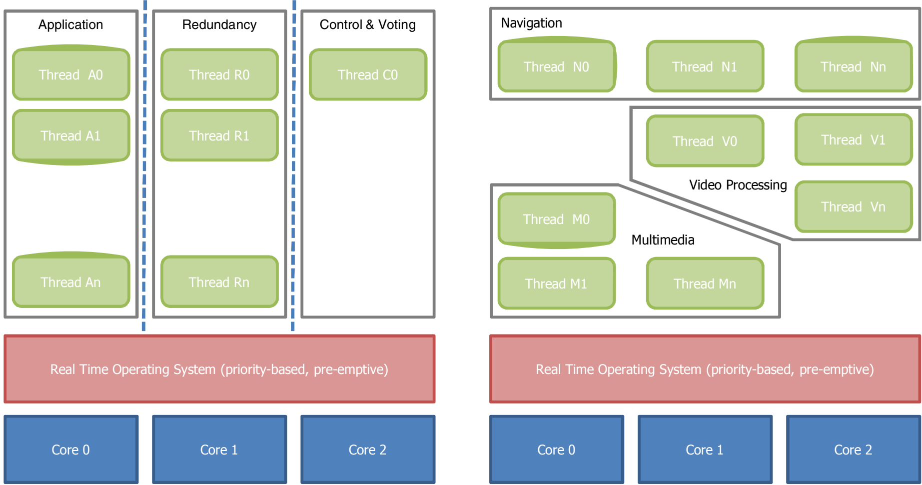

Fig. 1. For the implementation of a redundant system vertical partitioning of the software is appropriate, running each redundant application on a separate core and the control application on the third. For combining several legacy applications horizontal partitioning gives the flexibility of load balancing and preserving of execution order.

Usually redundant systems should be separated mutually. Therefore an asymmetrical partitioning minimizes the influence of a system to the other. Depending on the level of separation also single-core mode in a hybrid multiprocessing is appropriate (cf. figure 1).

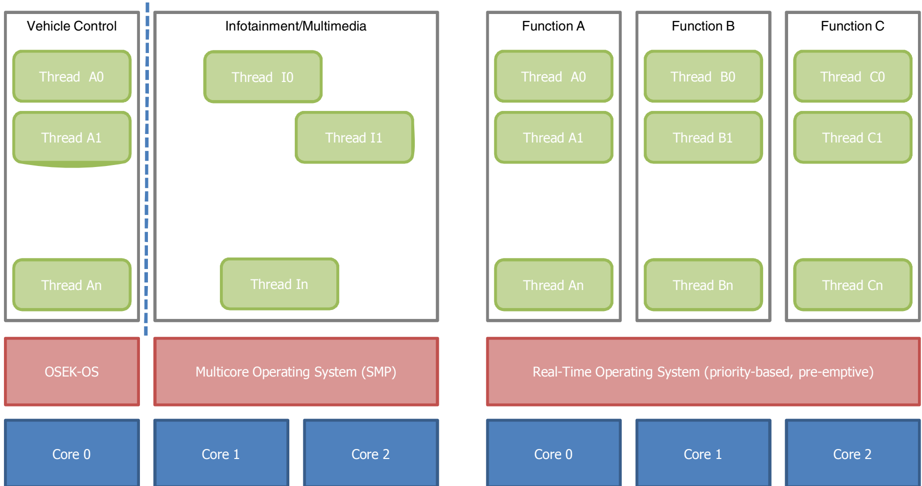

Fig. 2. Using asymmetric multiprocessing two operating systems are running in parallel representing different automotive domains (left). A real-time operating system that supports hybrid multiprocessing can bind functionalities to dedicated cores (right).

In case of replacing dedicated DSP functionality for harmonization reasons by a multicore architecture a hybrid approach can be applied. The ported DSP application can be moved in a hybrid environment running in single-core mode. But also an asymmetrical multiprocessing system may be appropriate as first step of a migration strategy. The drawback of asymmetrical multiprocessing is to have two operating systems running in parallel that need to get synchronized in an appropriate manner. But in specific cases this can be beneficial.

If pure computation power is required and the data can be arranged in a way that the calculation is symmetrical a pure symmetrical multiprocessing scheduling is the most appropriate. Again, the nature of automotive applications is usually not that way.

Figure 2 shows how different automotive domains can be mapped to multicore systems in different ways. Combining vehicle control functionality with the infotainment domain leads to the sketched architecture. On a single-core the OSEK operating system schedules threads (or tasks in OSEK nomenclature). Having a second symmetrical multiprocessing operating system both domains can run on a joint multicore hardware. Migrating different single-core applications to a multicore system can be achieved in single-core mode of a hybrid environment.

5 Design Patterns

The task to decide on the layout of an application across several cores is governed by the question to what degree the application can be parallelized. The challenge to get an application in an embedded device parallelized is as complex as on the desktop or server domain. No general guideline can be given since control parallelism always requires specific knowledge on the problem space. Design patterns are a well-accepted technique in software design. Some design patterns from the non-embedded world [11, 12] may also be applied for the embedded domain. But special attention has to be bestowed on the implementation.

5.1 Parallel Design

If a problem can be divided in the way that the algorithm can work in parallel and independent on separate chunks of data, then the master-worker pattern is an appropriate approach. A master thread controls a set of workers in a fork-join manner. For example sorting a large array can be implemented as parallel running worker threads quick-sorting sub-arrays. Merging the workers' output by the master thread finalizes the algorithm. Since the locality of the sub-arrays is very high, negative effects to the cache can be avoided. The speed-up is high. Other prominent examples are matrix calculations like multiplication or solving of linear equations on a mesh for fluid dynamics. Usually the nature of an embedded automotive application is not that way.

A central time-consuming algorithm of a navigation device is the route calculation. Finding the shortest path in a street network can be computed efficiently by Dijkstra's greedy algorithm [13]. Efficient parallelizing of such problem is much harder as the simple divide-and-conquer cannot be used easily. In order to achieve load balancing the pipelining programming pattern which works like an assembly line seems to be a more beneficial approach. In case of route calculation the data reading from some

medium like DVD-Rom or SD-card can be arranged in the way that always a buffer of the next edges of the street network is available for the shortest path calculation. Using two threads on separate cores will speed-up the overall performance.

5.2 Efficient Implementations

Parallel design patterns require efficient implementations. The producer-consumer pattern is a well-known pattern and can be used for the communication between components. One component, the producer, generates data, which is used by another component, the consumer. A simple and efficient implementation can be achieved, when a ring buffer is used. As depicted in figure 3, the producer adds data at the position write pointer, and the consumer reads data at the read pointer.

The producer has to take care that the write pointer never overtakes the read pointer, while the consumer has to make sure that the read pointer does not overtake the write pointer.

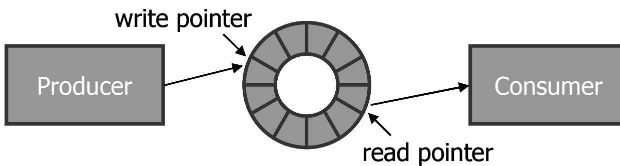

Fig. 3. The producer-consumer pattern is realised with a ring buffer

Although the implementation does not need further synchronisation mechanisms for just one producer and one consumer, the implementation does not work, if multiple producers or consumers enter the scene, because updates of the write pointer or read pointer are not atomic and therefore open to race conditions. In this case, e.g. a mutex must be used to protect the access to the ring buffer respectively the write and read pointer. The introduction of the mutex comes with the additional cost of a system call. In embedded systems these costs are often not acceptable. Therefore less "expensive" implementations are chosen. E.g. instead of using a mutex to protect the access to the ring buffer, CPU interrupts are masked as long as the access to the ring buffer is performed. But unfortunately in a multicore system, this approach does not work. When disabling interrupts for one core, another core is not prevented from accessing the ring buffer. To make the implementation work for multicore systems, the implementation can use a spinlock. If the spinlock is taken on a core, other cores cannot execute code in the critical section. They perform a busy wait until the lock is released. Alas, a simple spinlock is not sufficient, because the system might end in a dead lock. If the spinlock is taken and as many tasks as cores are available want to take the spinlock as well and these tasks have a higher priority than the task holding the spinlock, these tasks will on the one hand side wait for the lock to be released and on the other hand side prevent the task, which holds the lock, from releasing the lock. Therefore before the spinlock is taken, the task has to disable interrupts, in order not to be interrupted. The interrupts are enabled after the spinlock has been released.

The instructions to enable and disable interrupts are often privileged instructions, which only can be executed when the processor is in privileged mode. Often it is not desirable to run the processor in privileged mode. Instead user mode should be used whenever possible.

Therefore an implementation, which does not rely on masking interrupts, is necessary. To this end, atomic update operations like test and swap or load link and store can be used. But these operations cannot cover the entire access to the ring buffer. They can only be used to protect the update of the write and read pointer. First, a component reserves space in the ring buffer by updating the write pointer and then fills the ring buffer with data. Special care has to be taken, when a consumer wants to read data from the ring buffer. Since the write pointer is updated before the data in the ring buffer becomes ready, invalid data might be read by the consumer. If for example two components reserve space in the ring buffer, they will update the write pointer to hold first the value wp_i and then the value wp_{i+1} (cf. figure 4).

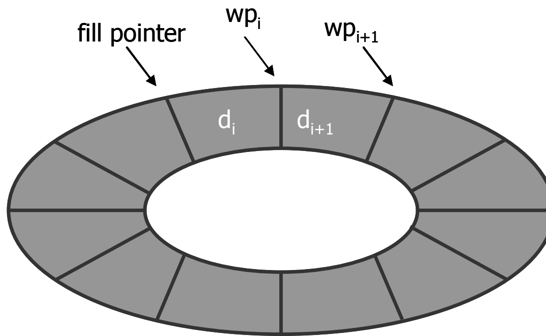

Fig. 4. Multiple Producers

Afterwards they each fill the reserved space with data d_i and d_{i+1}, respectively. At this point, it is not known which of the data is written first. To keep track of the point where data has been completely written, a fill pointer can be introduced. The fill pointer is updated after d_i has been written. In order to know whether the data d_{i+1} has already been written or not, an indicator is necessary. One way to realise such an indicator is to build up a list for the data, which has been filled, without updating the fill pointer. E.g. the element corresponding to data d_{i+1} holds the information wp_{i+1} and length(d_{i+1}) (see figure 5).

Fig. 5. Filled List

The elements are stored in ascending order of the write pointer value. The insertion of elements into the list has to be done by test and swap operations.

Although the algorithm presented comes with an overhead for the insertion into the filled list, the performance improvement is significantly compared to a solution using a mutex. On a system with a processor clock of 400 MHz, the operating system T-Kernel needs about 3µsecs for a system call. Whereas the implementation presented above takes only 100 nsecs, if no element is in the list. If there are already elements in the list when a new element is inserted, the traversal of each element takes 50 nsecs. Additionally, the algorithm is non-blocking, hence a producer can also run in the context of an interrupt.

Certainly, the optimisation can only be employed if the number of conflicting accesses to the ring buffer is an exception rather than the normal case.

6 Conclusion

Embedded systems are usually closed systems in the sense that user interaction is limited and any direct interference like installing user-defined applications is prohibited. This gives the opportunity to tune and optimise software. A multicore architecture takes away some optimisation techniques like efficient locking through interrupt masking, but at the same time offers new ways to gain performance like binding applications or threads to specific cores.

Based on use-cases this paper focuses on how to apply different operating system modes. However multicore systems compromise the predictability of real-time requirements. Further studies should focus on porting existing applications in order to get more evidence that hybrid multiprocessing is a feasible approach to support keeping such real-time conditions.

References

1. Das, B., Polle, T., Uelschen, M.: A Note on Software Partitioning for Embedded Homogenous Multicore Systems. In: Informatik 2008, München (2008) (accepted as conference submission)
2. Polle, T., Uelschen, M.: Softwareentwicklung für eingebettete Multi-Core-Systeme iX 3, 124–131 (2008)
3. Takei, T.: Toyota Works on Own OS for Automotive Terminals. Nikkei Electronics Asia (2006), http://techon.nikkeibp.co.jp/article/HONSHI/20061026/122752/
4. Lucke, H., Schaper, D., Siepen, P., Uelschen, M., Wollborn, M.: The Innovation Cycle Dilemma. In: Koschke, R., Herzog, O., Rödiger, K., Ronthaler, M. (eds.) Informatik 2007. LNI, vol. 110, pp. 526–530. Gesellschaft für Informatik, Bonn (2007)
5. Microsoft Auto 3.0, http://www.mircosoft.com/windowsautomotive
6. Kleidermacher, D.: Is symmetric multiprocessing for you? Embedded Systems Design Europe, January-February, 28–31 (2008)
7. Widmann, P.: Multi-Core-Systeme sinnvoll nutzen. Elektronik 13, 66–69 (2008)
8. Domeika, M.: Software Development for Embedded Multi-Core Systems: A Practical Guide Using Embedded Intel Architecture. Butterworth Heinemann (2008)

9. Gondo, M.: Blending Asymmetric and Symmetric Multiprocessing with a Single OS on ARM11 MPCore. Information Quarterly 4, 38–43 (2006)
10. Leroux, P.N., Craig, R.: Easing the Transition to Multi-Core Processors. Information Quarterly 4, 34–37 (2006)
11. Akhter, S., Roberts, J.: Multi-Core Programming. Intel Press (2006)
12. Rauber, T., Rünger, G.: Multicore: Parallele Programmierung. Springer, Heidelberg (2008)
13. Smith, J.D.: Design and Analysis of Algorithms. PWS-KENT Publishing, Boston (1989)