A Note on Software Partitioning for Embedded Homogenous Multicore Systems

Bhaskar Das, Torsten Polle, Michael Uelschen

Advanced Driver Information Technology GmbH
Robert-Bosch-Straße 200
31139 Hildesheim

{bhaskar.das, tpolle, muelschen}@de.adit-jv.com

Abstract: The introduction of homogenous multicore systems for embedded devices in the automotive domain has been started recently. Driver information systems like car navigation are the first application. This paper shows how the software architecture should be designed in order to use the multicore technology efficiently. We will focus on two principles as scheduling algorithms and parallel programming to partition software in multicore systems.

1 Introduction

In Figure 1 the evolution of multicore processors with some historical examples is shown. 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.

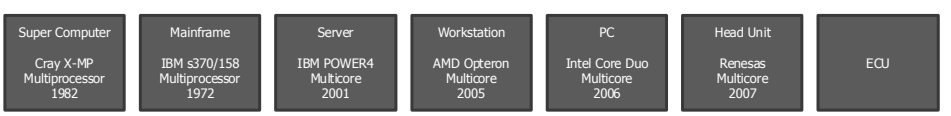

Figure 1: Evolution of multiprocessors and multicore systems from super computers to classical embedded electronic control units (ECU).

Now the multicore architecture is entering the embedded automotive domain. The first adopters are driver information systems (like car navigation or head units) followed by multicore systems for classical electronic control units (like body electronics).

Driver information systems combine functionality from the consumer electronics market (like MP3 or video playback functionality) as well as from the automotive domain (like CAN or MOST networking). Therefore it will be interesting to see if and how already established principles to use parallelism can be re-used for the automotive environment.

1.1 Embedded Automotive Multicore Systems

Homogenous multicore architectures are available for example from ARM and MIPS for Systems-on-Chip (SoC). Since 2007 commercial samples for navigation devices are offered by NEC and RENESAS [PU08].

In homogenous multicore systems, the shared access to the memory subsystem is symmetric and peripherals are identical from individual core point-of-view. Heterogeneous architectures constituting core connecting with a DSP are out of scope of this paper.

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

Deployment of new Functions. For the realisation 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.

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 in this approach.

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.

Convergence of Domains. 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 (see example from Toyota [Ta06]).

1.2 Parallelism

Parallelism can be separated into 4 levels [RR08]. In multi-purpose systems as well as in embedded devices further progress on bit-level or on instruction-level is limited. Since 1985 instruction pipelining is applied in most of the systems [HP03]. Data parallelism depends heavily on how the data can be processed by an algorithm and requires sophisticated programming languages and/or compilers that are seldom used in industrial solutions.

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.

2 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 condition if more than two threads of different priority go to RUN state at the same time.

2.1 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:

1. 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.

2. 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:

1. Deterministic behaviour 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.

2. Cache coherency, synchronization mechanism and shared data, limits application scalability.

3. Synchronization among threads compels execution across cores to become sequential.

2.2 Asymmetric Multiprocessing

SMP is the de-facto standard of multicore server and desktop operating systems [Kl08]. 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.

A promising approach is hybrid architecture: running just one RTOS but putting restrictions on the scheduling strategy.

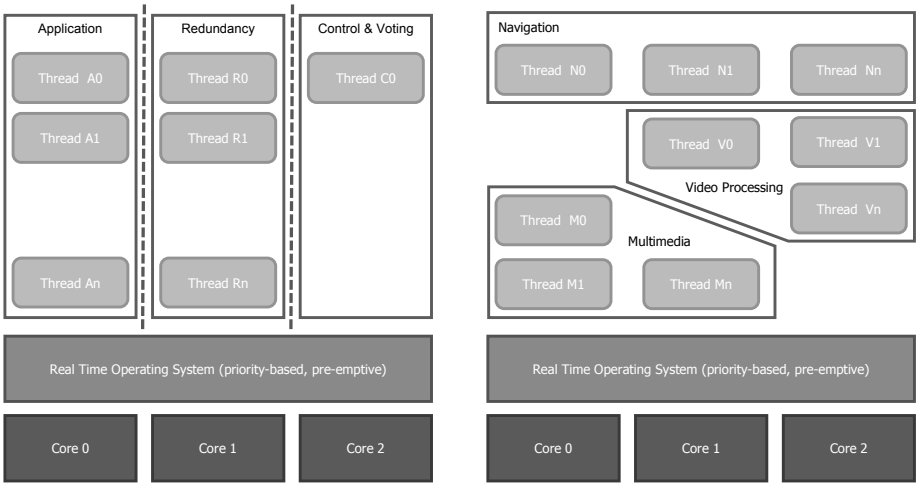

Figure 2: Vertical and horizontal partitioning. For the implementation of a redundant system vertical partitioning of the software is appropriate, running each redundant application on a separate core and the control application on the third. For combining several legacy applications horizontal partitioning gives the flexibility of load balancing and preserving of execution order.

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 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 multi-core systems.

In Figure 2 vertical and horizontal partitioning for the implementation of the described use cases is sketched.

Currently there are no standards on scheduling for multicore available. Some RTOS like eT-Kernel [Go06] or Neutrino [LC06] support both these flavours of SMP and AMP.

3 Parallel Programming

The described scheduling modes can support the porting of existing applications from single-core to multicore architectures. A better load balancing can be achieved if the parallel programming paradigm is applied at the implementation stage of the algorithms.

3.1 Parallelism of Algorithms

The challenge to get parallelized an application in an embedded device 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. However some design pattern from the non-embedded world [AR06, RR08] may also be applied for the embedded domain.

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 worker in a fork-join manner. For example sorting a large array can be implemented as parallel running worker threads quicksorting 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 gets 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 [Sm89]. Efficient parallelizing of such problem is much harder as the simple divide-and-conquer can not 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.

Using control parallelization requires a powerful library of thread functionality like Pthreads [Bu97] including synchronization objects like barriers or spin-locks. On a higher abstraction level a promising C++ library named Threading Building Blocks is available contributed by Intel for desktop OS [Re07]. Other approaches like OpenMP seem not suitable because of the different nature of embedded automotive domain.

3.2 Operating System and Device Driver Architecture

Parallel programming on thread level is the art to find units of execution that can be run concurrently. These units are not only found in applications but also in the operating system itself and device drivers. Especially in the automotive domain, the number of peripherals to be served like for example sensors is high, and also the integration into the car network, means the RTOS and device drivers have to work in parallel. Therefore even in embedded systems the trend goes towards kernel threads, which allows parallelising tasks in the RTOS and device drivers.

Another unit of execution is interrupt service routines. In some operating systems, an interrupt service routine is often coupled with a certain thread, the interrupt service thread. In the very extreme case, there is actually no longer a specific interrupt service routine but only the interrupt service thread.

If the time to service an interrupt is long and the interrupt frequency is high, there are two options: (1) Allow to service the second interrupt on another core concurrently; and (2) Use the techniques described in section 3.1 to parallelize the interrupt service thread.

4 Conclusion

Embedded systems are usually closed system in means that user interaction is limited and any direct interference like installing user-defined applications is prohibited. This gives the software architect for embedded multicore devices the opportunity to effectively use all cores in system by adopting software partitioning, a few of which have been covered as a part of this article.

References

[AR06] Akhter, S.; Roberts, J.: Multi-Core Programming. Intel Press, 2006.

[Bu97] Butendorf, D. R.: Programming with POSIX Threads. Addison-Wesley Boston, 1997.

[Go06] Gondo, M.: Blending Asymmetric and Symmetric Multiprocessing with a Single OS on ARM11 MPCore. In Information Quarterly, Volume 5, Number 4, 2006; pages 38-43.

[HP03] Hennessy, J. L.; Patterson, D. A.: Computer Architecture – A Quantitative Approach; 3rd Edition. Elsevier Science, 2003.

[Kl08] Kleidermacher, D.: Is symmetric multiprocessing for you? In Embedded Systems Design Europe, January-February 2008; pages 28-31.

[LC06] Leroux, P. N.; Craig, R.: Easing the Transition to Multi-Core Processors. In Information Quarterly, Volume 5, Number 4, 2006; pages 34-37.

[PU08] Polle, T.; Uelschen, M.: Softwareentwicklung für eingebettete Multi-Core-Systeme. In iX 3/2008; pages 124-131.

[Re07] Reinders, J.: Intel Threading Building Blocks. O'Reilly, 2007.

[RR08] Rauber, T.; Rünger G.: Multicore: Parallele Programmierung. Springer-Verlag, Berlin Heidelberg, 2008.

[Sm89] Smith, J.D.: Design and Analysis of Algorithms. PWS-KENT Publishing, Boston, 1989.

[Ta06] Takei, T.: Toyota Works on Own OS for Automotive Terminals. In Nikkei Electronics Asia, June 2006; http://techon.nikkeibp.co.jp/article/HONSHI/20061026/122752/.