Migration of Automotive Real-Time Software to Multicore Systems: First Steps towards an Automated Solution

Jörn Schneider, Michael Bohn, Robert Rößger
Dept. of Computer Science
Trier University of Applied Sciences
Trier, Germany
{*j.schneider, m.bohn, r.roessger*}*@fh-trier.de*

***Abstract*—With the advent of multicore systems in the automotive industry, a multitude of existing single-core applications wait to be migrated to multicore hardware. Manually migrating existing legacy software to multicore platforms in this industrial setting is practically infeasible, because it would be highly error prone, even if a host of skilled programmers were available. Therefore, tool support is strongly needed. This paper describes the challenges of migrating real-time software in the automotive context and our approach to use static program analysis to develop automated migration tools and methods.**

***Keywords*-real-time systems; multicore; multiprocessor; software migration; static program analysis**

I. Introduction

With the switch to multicore systems, the automotive industry faces a disruptive paradigm shift due to the introduction of true concurrency. The adaption of existing application and system software will be a very complex and yet tedious process leading to unpredictable errors, if done manually. The ProSyMig[1] project aims at developing methods and tools for the (semi-)automatic migration of real-time software to multicore systems.

We are not aware of other research groups addressing this issue. In [1] the migration to multi-core is considered, but with a focus on scheduling and partitioning.

A. System Model

Automotive real-time software running on multicore systems has to be compliant to the AUTOSAR standard [2]. Among other things AUTOSAR provides a middleware with standardized communication primitives, and a real-time operating system. The legacy software to be migrated is not necessarily AUTOSAR compliant. It usually still uses proprietary mechanisms, e. g. for communication and synchronization, and lacks a clear separation between hardware dependent system software and application software. The common ground of legacy and migrated software is the OSEK real-time operating system standard, as this is also the core of the AUTOSAR OS.

The latest release of AUTOSAR (4.0) appeared end of 2009 and includes for the first time some features for multiprocessing. As this is the assumed system model for this paper, we give this short characterization of these features: AUTOSAR prescribes bound multiprocessing, i. e. a partitioned system with tasks and interrupt service routines statically mapped to cores. Each core runs a fixed priority scheduler for its particular task set. It is possible to activate tasks and to send events (i. e. wake up signals for tasks in waiting state) across core boundaries. Unfortunately, the OS provides only spin locks as dedicated mechanism for mutual exclusion across cores.

B. Contributions

In this paper we will first highlight the problem of a frequent practice related to the migration to multicore systems (locking interrupts as measure for mutual exclusion) to exemplify our solution approach. Afterwards the scope is widened to a more comprehensive view by introducing three classes of constraints that need to be preserved when migrating a legacy system. We show that even with perfect information about the established constraints in the legacy system, it is likely that the migrated software wastes a lot of the processing power of multicore hardware. Even though it is impossible to achieve perfectly precise information about the existing constraints of the original system, we found a solution that, as we believe, will eventually provide automated migration to reasonably efficient multicore systems.

II. Problem description and approach

A. The problem

As an example, assume a single core system with three tasks A, B and C and two interrupt-service-routines (ISRs) ISR_1 and ISR_2. The priorities of the tasks are given as $Prio(A) > Prio(B) > Prio(C)$. The notional priorities of ISRs are above task priorities. Task B establishes a critical section by calling *Disable-* and *EnableAllInterrupts*. The developer's intention behind the disabling of interrupts in task B might be to protect a data structure that task B shares

This work is partly funded by the German Federal Ministry of Education and Research

[1] *Pro*gram and *Sy*stem Analysis Tools for the *Mig*ration of Embedded Software to Multicore Systems

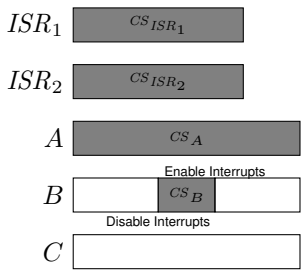

Figure 1. Potential critical sections by disabling interrupts in task B. Critical sections are shaded in gray

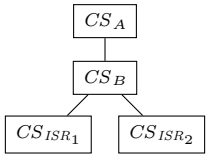

Figure 2. Mutual exclusion relation of critical sections in Figure 1

with ISR_1. To prevent a race condition between ISR_1 and task B he needs to use the interrupt disable mechanism.

If we (automatically) analyze the scenario described above, we do not know the developers intentions for the calls to *Disable-* and *EnableAllInterrupts*. We therefore have to conservatively identify *all possible* other critical sections that are in a mutual exclusion relation with the critical section in task B. Additionally to the actually intended mutual exclusion between the critical section in task B and ISR_1, we have to assume a mutual exclusion between ISR_2 and the critical section in task B. We assume also a mutual exclusion between task A and task B, because the disabling of interrupts in task B prevents task A from running concurrently with B. Note, that there is *no* mutual exclusion assumed between task C and the critical section in task B. This is, because task C has lower priority than task B. Task C could be preempted at any time by task B. Figure 1 shows the identified critical sections in the example tasks and ISRs. The mutual exclusion relations between the critical sections are depicted in Figure 2. The edges in this graph are the mutual exclusion relations.

The so identified critical sections are the *potential* critical sections. They *might* be used to protect a physical resource from being accessed concurrently. In the real system CS_B shares resources only with a subset of the critical sections in Figure 2, namely only the critical section CS_{ISR_1}. Because it is unknown (from a an analysis point of view) whether the protection is actually needed or just a side effect of the disabling of interrupts, the solely safe assumption is that all mutual exclusion relations need to be preserved.

The found mutual exclusion relations are potentially present in the single core legacy system. If this set is not reduced further (an approach for automatically achieving this is presented in the next subsection) the corresponding critical sections also have to be protected in the multicore system to prevent the introduction of race conditions. Whether a particular mutual exclusion relation is still existing in the multicore system without code changes depends on the mapping of tasks and ISRs to cores. Assume that task A and ISR_1 are bound to core zero and task B, C and ISR_2 to core one, some mutual exclusion relations still exist while others do not. The considered mapping eliminates the mutual exclusion relation between task A and CS_B. The exclusion of task A during CS_B was established by disabling interrupts. This does in no way hinder the execution of a task on a different core. Therefore, the potential critical section in task B is not protected against task A (and vice versa since the higher priority of task A does not prevent the scheduling of task B). If this mapping has to be realized, a different mechanism for protecting the critical section has to be used (e. g. a multiprocessor resource locking mechanism). For similar reasons, the mutual exclusion between ISR_1 and CS_B is no longer given.

Note that the critical sections in ISR_2 and task B are still protected against each other, because task B and ISR_2 are bound to the same core.

B. Conservative Solution

In a first step it is sufficient to derive the set of potential critical sections along with their mutual exclusion relations. For the considered mechanisms of task priorities and disabling interrupts this information can be automatically extracted from the AUTOSAR configuration files [3] and the C-code. The XML configuration files contain the information about the tasks and their priorities. To extract the critical sections that are established by calls to *Disable-* and *EnableAllInterrupts* we have to analyze the source code of the tasks and ISRs. Because calls to the interrupt API-functions of the OS are usually concealed in the system software, which is often closely entangled with the application, it is quite likely to find these calls nested inside conditional expressions or loops. Therefore, we currently develop a suiting dataflow analysis. Thereby it is guaranteed that no critical section of this type is missed, moreover, the analysis tries to extract the smallest critical section, but overapproximates the area of a critical section if the exact area cannot be inferred.

After all potential critical sections that are established by the disabling of interrupts have been identified, the relation graph to other potential critical sections for each such critical section can be build from the information of the analysis and the configuration file.

Note, that there are two levels of overapproximation present. The first one comes from the analysis that tries to find the smallest critical sections that are established by interrupt locking calls. Although, we are interested in reducing this overapproximation as much as possible, in general we cannot avoid it, since the underlying problem

is undecidable. The second overapproximation is the one that stems from the set of mutual exclusion relations in the original system that are not really necessary.

C. Improved Solution

The conservative approach in the previous section generates a lot of potential critical sections. Although every potential critical section might actually require protection, this is probably not the case in real systems. To reduce the set of potential critical sections in a safe way, we propose a static program analysis that derives the memory access behavior. The idea of this analysis is as follows. If we can determine the memory locations that *may* be accessed in each critical section, we can eliminate those mutual exclusion relations between critical sections that access disjunct memory areas only. At least we can thereby guarantee that the mutual exclusion relations are not needed to prevent concurrent access to shared memory or peripherals. Note that we can not deduce that the critical section is generally not needed. If, for example, two different actuators are controlled in the critical sections, two different memory locations are accessed for controlling the actuator. Although the memory analysis would conclude that the critical sections access disjunct memory locations, there might be the requirement that the two actuator positions are never changed concurrently. This requirement cannot be automatically inferred from code and has to be provided by the user.

The memory access analysis should comply with the following requirements. The result of the analysis should be all memory locations that *may* be accessed in the critical section. These accesses can be represented as intervals of memory addresses that are subsets of the whole memory. The accesses to the memory should be separately classified as read or write access. If the analysis cannot determine which memory locations are accessed it should assume that the whole memory is accessed.

A good starting point could be a value analysis, for example the sophisticated one implemented in ASTREE [4]. Value analyzers derive the potential values of variables at each program point. This is useful to decide which memory areas are accessed in case of arrays or pointers. A classical value analysis though does not provide all information we need. For example it is necessary to derive which ordinary global variables (not only arrays and pointers) are accessed and to distinguish between read and write accesses. This distinction is needed in the next step: the evaluation of the result of the memory access analysis.

After the intervals of memory access have been computed, the mutual exclusion relation between two critical sections can be evaluated. The test if two critical sections only access disjunct memory locations is trivial with the previous analysis result. We only have to test if all memory intervals in the set of memory accesses are pairwise disjunct. If we find out that the two critical sections never access the same memory we can *guarantee* that the mutual exclusion is not needed for memory access protection. If two critical sections potentially access the same memory location, it might still be possible to automatically eliminate the mutual exclusion relation. If the considered critical sections only read from the same memory locations, the mutual exclusion is not necessary assuming that concurrent reads can be allowed. Unless of course there are other reasons, for example the already mentioned, mutual exclusive positioning of two actuators, which has to be specified by the user.

If we find out that in all connected critical sections only a common memory location is read, all those critical sections can be dropped. This could be the case if a configuration parameter, is only written at the operating system startup and subsequently only read. On the other extreme, if we see that in each critical section a common memory location is written, all potential critical section still need to be protected to prevent race conditions.

In the case of multiple readers and writers present, we can decide locally if an edge in the graph in Figure 2 is still needed.

III. The big picture

A. Constraint classes

In the previous sections we presented our concept of using static program analysis to extract information from C-code and refine this information with further analyses for the example of locking interrupts. Apart from this there are many further mechanisms that provide mutual exclusion on single core systems but cannot guarantee this on multicore processors. We collected a list of such mechanisms and aim at identifying the mechanisms with practical relevance by studying typical automotive real-time software. Besides mutual exclusion we identified two more *constraint classes* and different mechanisms that can be used to realize these concurrency constraints in single core, yet not in multicore systems. The constraint classes and their relations are shown in Figure 3. The weakest constraint between code segments of jobs is the *none constraint*. If there is no constraint between code segments, there is no restriction for concurrent execution. The next constraint class is the mutual exclusion. If there is a *mutual exclusion constraint* between two code segments of jobs, these code segments are not allowed to be executed concurrently. That is, the first code segment can be executed before the other code segment or vice versa. A *precedence constraint* between code segments of jobs enforces a sequential execution order. If code segment A is in precedence constraint with code segment B, A must finish execution before B starts execution. The strongest constraint class that we consider is the *temporal constraint*. A temporal constraint between code segments of jobs is an interval $[x, y]$, where x is the minimum and y the maximum temporal distance between the code segments. The interval is denoted as the *temporal distance*.

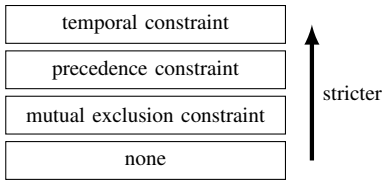

Figure 3. Hierarchy of concurrency constraints

The constraint classes are ordered by their strictness and form a hierarchy, where a more strict constraint implies a less strict constraint. The precedence constraint is a special form of the temporal constraint, where the temporal distance is implicitly set to $[0, \infty]$. Note that although these constraints are general in the sense that they are not limited to real-time systems, the actual mechanisms that are used to realize them in automotive electronic control units rely heavily on typical real-time mechanisms. For mutual exclusion most mechanisms rely on the fixed priority scheduling, e. g. two tasks sharing the same priority cannot preempt each other. Furthermore mutual exclusion is often used in combination with a temporal offset between jobs to achieve precedence. And precedence together with a certain best case execution time is sometimes used to realize a temporal constraint.

B. Realization approach

This section sketches the requirements for a toolset to help in the migration process. As already shown there exist various problem cases where detailed information has to be gathered in order to decide whether and how a certain code section has to be modified.

Not all information can be extracted directly from code files. As automotive real-time software is usually developed in a model-driven way, we also plan to analyze these models and configuration files, which we summarize under the term *artifacts*. This helps if specification knowledge is modeled on a higher level of abstraction and disappeared by being transformed into implicit mechanisms (e. g. locking interrupts) on code level.

Some information cannot be extracted at all. For example, if there is a temporal distance between code sections which stems from external requirements, we will hardly find the exact distance in any artifact. So the conservative assumption cannot be weakened, leading to potentially inappropriate overestimation. Therefore user input and user interaction is necessary to provide the possibility to incorporate external specification knowledge from the user. This means adequate visualizations of analysis results have to be found and mechanisms have to be implemented to make the system at least partially interactive.

Even a single use case like finding mutual exclusion depends on various analyses which are interconnected and potentially produce a huge amount of data. Collecting analysis data in an information repository could allow for caching mechanisms, i. e. each analysis stores its result in the repository and only needs to be run if its input data has changed. Successive analyses then can take their input from this source, so linking analyses becomes configurable on a data level. Additionally this abstraction supports interactive usage, as well as providing an interface to external queries for reporting services.

The high-level requirements for the analysis framework are accordingly: handling of various types of artifacts, efficient combining of analyses and ability for user interaction.

IV. Conclusion and future work

We identified three important constraint classes of real-time systems, that need special consideration when migrating to multicore hardware. As shown on the example of mutual exclusion, the mechanisms typically used in automotive real-time software establish significantly more constraints in single core systems than actually needed. It should be obvious that realizing all such concurrency constraining conditions in the multicore target system seriously limits the exploitation of parallelism. The described approach of combining different static program analyses to automatically detect potentially unnecessary constraint will, as we believe, significantly improve the efficiency of the migration process as well as the quality and performance of the migrated software.

This hypothesis has to be evaluated in our future work. Further challenges lie ahead. For instance extracting and improving the temporal constraints requires the combination of WCET, BCET and schedulability analysis to find constraints that potentially have to be preserved during the migration. The initially considered mutual exclusion constraint is also of importance here, since for example the WCET of critical sections is needed by the schedulability analysis.

Visualizing the extracted information and relations appropriately will also be important for the needed interaction with the user.

References

[1] F. Nemati, M. Behnam, and T. Nolte, "Efficiently migrating real-time systems to multi-cores," in *Proceedings of 14th IEEE ETFA'09*, September 2009.

[2] *AUTOSAR 4.0 Specification*, AUTOSAR Std., 2009.

[3] *AUTOSAR Specification of the Meta-Model*, AUTOSAR Std., 2009.

[4] P. Cousot, "The astrée static analysis tool," in *ES_PASS Workshop*, Berlin, Germany, 16–17 October 2007.