The legacy ECU software problem - approach and research challenges

Thomas Heinz, Jörn Schneider
Robert Bosch GmbH, Stuttgart
{*thomas.heinz2, joern.schneider*}*@de.bosch.com*

Abstract: While life cycles of electronic components tend to become ever shorter, automotive suppliers need to keep up ECU supply for up to 30 years. This causes high storage costs or requires redesigning legacy ECUs in a way that obsolete components are replaced by newer ones. These redesigns may cause tremendous software redevelopment efforts, e.g. to guarantee correct real-time behaviour. This paper presents a both academically and industrially challenging approach inspired by the legacy ECU software problem, namely static binary translation, which aims at automatic retargeting of real-time software. Eventually, the approach has the potential to significantly cut the redevelopment costs and facilitate the interchangeability of electronic components.

1 The legacy ECU software problem

Automotive ECUs are typically supported over a period of 10 to 30 years. This timeframe exceeds the life cycles of many of its electronic components (e.g. microcontrollers) by far. To enable production of ECUs in the original configuration beyond the point of discontinuation of its electronic components, these components need to be stockpiled in advance. However, it is very hard to come up with a tight estimate of the actual number of ECUs needed within the entire product lifetime. Hence, it can be necessary to redesign the ECU hardware such that obsolete components are replaced by up-to-date ones with similar functionality. This usually makes it necessary to port the software as well.

Porting and testing for correct functionality and real-time behaviour is tedious, very costly and comes with the significant risk of introducing errors. This inspires a both academically and industrially challenging approach, namely **automatic software retargeting** which is discussed in the next section.

2 Automatic retargeting of embedded real-time software

The legacy software problem is not specific to ECU software but is prevalent in virtually any real-time and non-real-time software field. A successful approach for non-real-time and soft real-time software is **emulation**, i.e. representing the original programming environment (e.g. instruction set architecture, device registers, application binary interface) on the target machine and instruction-wise interpreting the original machine code. Ideally, emulation is a black box approach in the sense that it does not require any knowledge about the legacy code. It is highly flexible but inherently inefficient which can be compensated by translating software such that it can be natively executed on the target machine. An established method that builds upon this idea is **binary translation**, a compilation technique that transforms machine code for one architecture into machine code for another architecture while retaining its functional behaviour. Binary translation can be performed either at runtime (dynamic) [CLU02] or offline (static) [CvERL01]. The former takes advantage of runtime information which enables efficient code generation and simplifies some problems of the static approach, e.g. control flow analysis in the presence of branches whose target address depends on runtime data (indirect branches/calls). However, a major issue that renders dynamic binary translation unsuitable for legacy ECU software (or more general software involving hard real-time constraints) is its unpredictable temporal behaviour. This is because the translator is invoked at unpredictable times at runtime and the temporal behaviour of the translated code (= runtime data) cannot be predicted statically. In particular, the dynamic translator is not amenable to existing real-time scheduling methods because its invocation cannot be postponed (highest priority) and its minimum interarrival time cannot be reasonably bounded. A major advantage of the static approach is its amenability to program analysis as the target code is available offline.

A solution to automatic retargeting of embedded real-time software must ensure **preservation of functional and temporal behaviour** of the original software. The following sections discuss the suitability of **static binary translation (SBT)** with respect to these requirements and present promising approaches and open challenges.

3 Static binary translation - a functional point of view

Legacy ECU software is composed of application software and basic software (OS, drivers). Apart from the standardized operating system (OSEK), ECU software is proprietary and contains a considerable amount of architecture dependent low level code. Hence, SBT must address the entire instruction set architecture and computational resources (e.g. registers, CPU flags, memory, I/O ports, interrupts) of the source machine. Moreover, the functional behaviour of the devices on the source machine (e.g. SPI, CAN, ADC, timer) must be mapped onto appropriate target devices.

Figure 1 illustrates common transformation steps to translate the source binary into a functionally equivalent target binary. In a first step, the binary is disassembled into code and data sections. At this point, code sections may still be intermixed with data (e.g. switch table, constants). The subsequent control and data flow analysis separates code and data and builds a control flow graph (CFG). To enable architecture independent SBT and optimizations, the source CFG is mapped to an intermediate CFG based on an architecture independent intermediate language. Finally, the code generator turns the intermediate CFG into a target specific CFG on which optimizations tailored to the target architecture can be applied. To obtain the target binary, two simple steps need to be performed, namely translating the target specific CFG into target assembly and finally into machine code. Note that it is possible to transform debugging information associated with the source binary such that the translated binary can be related to the original source code.

SBT faces a number of challenges some of which can be reduced to theoretically undecid-

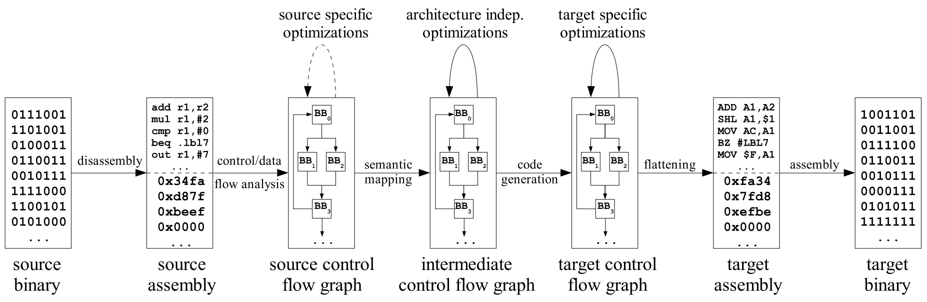

Figure 1: Common SBT transformation steps.

able problems. Among are:

- control flow analysis in the presence of indirect branches/calls
- efficient code generation
- data translation to enable native data access (alignment, data representation)
- separation of intermixed code and data
- handling of machine dependent issues (e.g delayed branches)
- distinguishing memory mapped I/O instructions from regular memory accesses
- instruction atomicity in the presence of hardware interrupts (a single source instruction may correspond to a sequence of target instructions)
- retargetability of the translator

Theoretically undecidable problems are approached by various program analysis techniques which compute safe approximations of runtime information (e.g. interval analysis of register values). Where not automatically deducible, the user has to provide the relevant information, e.g. in terms of source code annotations which are treated by the compiler. Particularly, these analyses are essential for efficient code generation. Note that SBT is not capable of handling self-modifying code. However, this is not a problem for automotive software and presumably neither for hard real-time software in general.

Recent advances in control flow analysis (CFA) [The02] and the success of static timing analysis tools, which face the same problem, show that CFA is reasonably mastered to be successfully applied to real world ECU software.

4 Temporal proximity

When replacing an ECU by its redesigned version, it is of crucial importance that the observable behaviour at its interfaces to sensors, actuators and the car network remains essentially unchanged. This includes the temporal behaviour in particular.

Before one can come up with methods to preserve the temporal behaviour, it is necessary to determine the level of accuracy as a criterion for precision of the preservation. Here, four levels of accuracy are distinguished.

Cycle accuracy: The functional behaviour of each source machine clock cycle is precisely reflected on the target machine. This is required for development of real-time systems in absence of the target hardware. Instead a cycle accurate system simulator is used. Real-time capability cannot be achieved using a fully software-based approach.

Instruction accuracy: The start or end of execution of each source instruction and its corresponding sequence of target instructions is synchronous on source and target machine. This level of accuracy is still too high. Moreover, to achieve real-time capability, the target machine must be vastly superior to the source machine.

Basic block accuracy: The scope is extended from a single instruction to a basic block. This accuracy level might already be too imprecise, e.g. in case a basic block contains an I/O instruction whose execution point in time is crucial to the correct behaviour of the software.

Synchronization point (SP) accuracy: SP accuracy is motivated by the observation that certain instruction sequences can be executed arbitrarily fast without affecting the correctness of the temporal behaviour, e.g. arithmetic computations involving only local variables. Hence, instruction accuracy is not demanded for each program point but only for certain critical program points, e.g. an I/O instruction, a system instruction or a store operation to a shared memory location. A naive way to safely approximate the set of SPs for a program is to divide the instruction set into potentially critical and non-critical instructions and to mark each program point containing a potentially critical instruction as SP. The number of SPs can be reduced by program analysis or user information.

Given a safe approximation of the SPs, SP accuracy is precise enough to capture the temporal behaviour of the original software that is to be preserved. It remains to devise a synchronization method that is real-time capable. Traditional approaches implement a dynamic delay mechanism that keeps track of the execution time on the source machine [Cog95]. The delay ensures that after the execution of an SP on the target machine, the system is idle until the execution of the respective instruction would have been finished on the source machine. This is efficient for simple architectures where the execution time of each instruction is constant but does not scale for machines with pipelines or caches as the execution time of an instruction depends on the execution history and thus all execution time affecting entities would have to be simulated on the target machine up to an extent.

As a solution, we propose an offline method to compute a set of delay constants $\{d^p_{c_1}, \ldots, d^p_{c_n}\}$ for each program point p where each constant is associated with a context c_i. A context allows to distinguish several control flows that may lead to the execution of the associated program point (see [The02] for details). The synchronization code that is executed at runtime mainly consists of determining in which context c_i the program point p is currently executed and waiting the precomputed constant time $d^p_{c_i}$. The problem of computing the delay constants is cast into an optimization problem whose objective is to minimize the maximum **temporal displacement** of all program points. Intuitively, the temporal displacement of a program point p is the maximum time that the execution on the target machine is ahead or behind the execution on the source machine for p. Moreover, the temporal displacement is a metric for the quality of the achieved SP accuracy.

The precise description of the optimization problem is beyond the scope of this paper. The basic idea is motivated by the integer linear program (ILP) used to compute an upper bound of the worst case execution time of real-time tasks [Wil05]. Essentially, the ILP constraints are used to describe a safe approximation of the temporal displacement for each program point, i.e. an upper bound of the actual temporal displacement. For this, an inner approximation of the execution time of each program point on the source machine is computed by measurements whereas on the target machine, an outer approximation is computed based on abstract interpretation. Note that an inner approximation $[s^i, t^i]$ of an execution time interval $[s, t]$ satisfies $[s^i, t^i] \subseteq [s, t]$ and an outer approximation $[s^o, t^o]$ of $[s, t]$ satisfies $[s^o, t^o] \supseteq [s, t]$. The approach is currently being implemented and refined as part of the Ph.D. thesis of the primary author.

Clearly, this approach does not guarantee perfect SP accuracy but it provides a guaranteed bound of the temporal displacement for each program point and thus quantifies the degree of **temporal proximity** between the execution of the original and the translated code.

5 Conclusion

The legacy ECU software problem motivates a number of challenging theoretical and practical problems. The approach presented in this paper seems very promising. However, as the temporal behaviour of the original software cannot be precisely reflected on the target machine, the deviation with respect to the temporal behaviour which can be tolerated must be quantified such that it is possible to decide whether correct real-time behaviour is achieved for the given temporal displacement. Note that a quantification of the tolerance cannot be deduced from the code but requires system knowledge, e.g. about the physical tolerance of the system.

Last not least, it remains to show that SBT is real-time capable for a reasonable pair of source and target CPU/microcontroller and thus prove its practicability. For this, it is essential to devise a method to safely and tightly approximate the necessary SPs.

References

[CLU02] Cristina Cifuentes, Brian T. Lewis, and David Ung. Walkabout - A Retargetable Dynamic Binary Translation Framework. Technical report, Sun microsystems, 2002.

[Cog95] Bryce Howard Cogswell. *Timing insensitive binary-to-binary translation*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, April 1995.

[CvERL01] Christina Cifuentes, Mike van Emmerik, Norman Ramsey, and Brian Lewis. *The University of Queensland Binary Translator (UQBT) Framework*. The University of Queensland, Sun Microsystems, Inc, 2001.

[The02] Henrik Theiling. *Control Flow Graphs For Real-Time Systems Analysis*. PhD thesis, Universität des Saarlandes, 2002.

[Wil05] Reinhard Wilhelm. Determining Bounds on Execution Times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.