

Transactional Consistency in the Automotive Environment

Patrick Schmidt, Stefan Frenz, Steffen Gerhold, Peter Schulthess

Department of Distributed Systems

University of Ulm, Oberer Eselsberg, 89069 Ulm

Germany

Email: patrick.schmidt@uni-ulm.de

Abstract—This paper examines the feasibility of a distributed shared memory in an automotive environment with respect to communication and resource consumption. We briefly present the Plurix cluster operating system and its compiler demonstrating the advantages of a distributed shared memory concept for medium scale computation scenarios. We then apply the DSM concept to a specific use-case from the area of automotive embedded systems. A prototype system was implemented and shown to offer higher bandwidth and lower resource utilisation than conventionally layered designs.

I. PLURIX

Plurix is a distributed shared memory operating system developed at the Institute of Distributed Systems at the University of Ulm. Primary goals of the Plurix project were to simplify the development of distributed applications (see [2]) and offer recoverable persistent objects (see [1]). Plurix is based on the Distributed Shared Memory (DSM) paradigm introduced by Keedy (see [3]) and Li (see [4]) and implements transactionally consistent memory combination with an optimistic synchronization scheme. The system is currently rebuilt to support 64 bit multi-core processors and to offer additional memory consistency options.

II. SJC

The Small Java Compiler SJC is a lean but sophisticated compiler translating (a subset of the) Java language into native code for different architectures targeting 8, 16, 32 and 64 bit processors (see [7]). The special class "MAGIC" grants low-level hardware access beyond the traditional Java sandbox.

SJC thus reconciles driver & OS-level programming with the benefits of a type-safe language avoiding potential programming errors due to memory arithmetic or misused pointers (see [5]). The runtime structures created by the compiler discriminate between primitive data-types (called scalars) and reference variables. Objects are therefore dual-headed (see [6]) and simplify the inspection of references in tasks like relocation or garbage collection. There are two categories of scalar objects depending on the intended memory consistency model: "direct scalars" are directly allocated in the object descriptor and "indirect scalars" are allocated in a separate allocation pool.

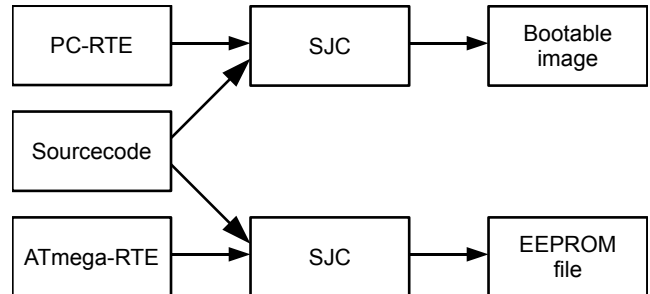


Fig. 1. SJC comprising the complete toolchain

The Compiler was also implemented in Java¹ with appropriate packaging to allow the addition of a new language frontend, the provision of additional codegenerations (backend) for different target machines. The creation of disk-bootable images, hexfiles for EEPROM flashing or executable files for Linux and Microsoft Windows¹ works smoothly due to the clean packaging of the compiler. The complete SJC tool chain (see figure 1) from source code to output file is modularized, well arranged and easily teachable.

III. AUTOMOTIVE SOFTWARE

Whereas desktop applications tend to be rather heavy-weighted and resource-consuming, the software used in automotive and embedded environments typically must meet stricter requirements with respect to memory consumption, processor speed and power consumption. In brake or steering control systems for example it is unacceptable to delay data or the calculation of essential results.

Another typical property of automotive and embedded applications is their distribution to a group of micro-controllers connected through a field bus system. This distribution of resources forces the software engineer to deal with an ensemble of asynchronously cooperating CPUs instead of merely programming a single embedded control unit. Traditionally the intricacies of data distribution, asynchronous communication & asynchronous execution are

¹Windows, Java and ATmega are registered trademarks.

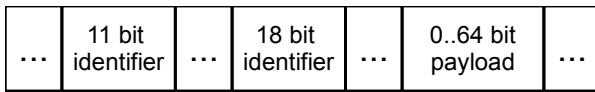


Fig. 2. Id and payload field in a CAN frame (2.0B)

handled by run-time libraries, tool chains and specialized IDEs. In practice, the developer is simultaneously dealing with the intrinsic complexity of the application task and the added complexity of the tool chain. Standards for the development of automotive software resulted in the OSEK-standard and finally in its successor, the AUTOSAR standard (see [9]). The main goals were exchangeability and interoperability of software modules by respecting given interfaces and conventions. Regarding the software architecture, OSEK defines several layers, starting with the application software, runtime environment, basic software, hardware abstraction and finally the micro-controller-specific layer with well defined interfaces. Communication is basically carried out by a message-passing mechanism combined with a listener concept. In order to meet the given real-time requirements, the time-sequence of transmitted messages is set up at compile time and takes into account all software-modules participating on the bus. Several refinements exist in the definition of precise timing, scheduling, protection mechanisms, etc... But the implementation is left to the component manufacturer, following the motto cooperation on standards, competition on implementation. AUTOSAR, emerged from the OSEK-standard, also proposes a layered structure for the software design dividing the development into three components: the software comprising a software component description (providing information about ROM/RAM-consumption, API,...), an ECU resource description and a set of constraints to be applied to the software. These elements are processed by a tool partitioning the tasks and finally creating the ECU-software. The resulting ECU-software is rather inflexible, as changes made to a single component affect other components, the latter being interweaved by the partitioning tool. Additionally, this approach tends to create static communication-schemes, complicating a potential replacement of modules.

To interconnect the ECUs the controller area network (CAN) became accepted in the automotive environment [9]. The CAN standard defines data-rates from 10kbit/s up to 1 Mbit/s providing bus access to the participants by means of an id encoded in the header of a CAN message. Each CAN message contains up to 8 bytes of data. In case of collisions the message with the highest priority i.e. the lowest id prevails.

An industry partner provided a scenario to demonstrate the feasibility of a distributed shared memory scheme for automotive environments. Three stations/ECUs were connected by CAN-bus at a data-rate of 600 kbit/s. Each station owns 200 bytes of data stored in the DSM and

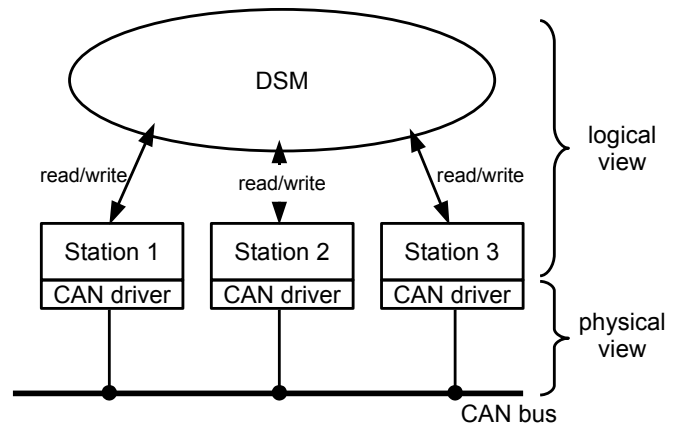


Fig. 3. DSM-Scenario with logical and physical view

processes them by accessing 2-byte-portions of the data in a deterministic manner. Consequently the DSM spans a memory containing 600 bytes. After transmission, data is checked for consistency on the other stations. The requirements to be met with regard to time response demanded that 400 bytes are exchanged every 10ms and 200 bytes in a period between 100 and 1000ms. The processor clock rates should not be more than 50 MHz. These requirements combine two essential aspects of automotive software development. On one hand items are exchanged between different nodes in given rigid time patterns representing the real-time requirements and on the other hand a consistent perspective of the data is guaranteed and verified. The data traffic imposes high demands considering that the effective net data rate amounts to approx. 35 kB/s.

IV. DSM IN AUTOMOTIVE

As an alternative to the library centered approach of the current standards we suggest the use of distributed shared memory. In this scenario, participants of the distributed system (such as sensors, etc...) simply store their data in the DSM and make it thereby visible to the other participants. Data exchange is performed implicitly by accessing the software DSM. The prototype implementation is completely programmed in Java and based on the minimal operating system PicOS (see [7] and [8]) and drivers providing access to the CAN bus adapters.

Consistency is guaranteed by a global DSM time for each set of data indicating the validity of the changes made to it. According to the given scenario we implemented a single-writer/multiple-reader situation and a distinction between writer and checker tasks, the former writing given areas in the software DSM, the latter verifying whether the remote value currently readable is valid. Data transmission is managed by an interrupt-driven driver which is in charge of propagating to the other participants of the distributed

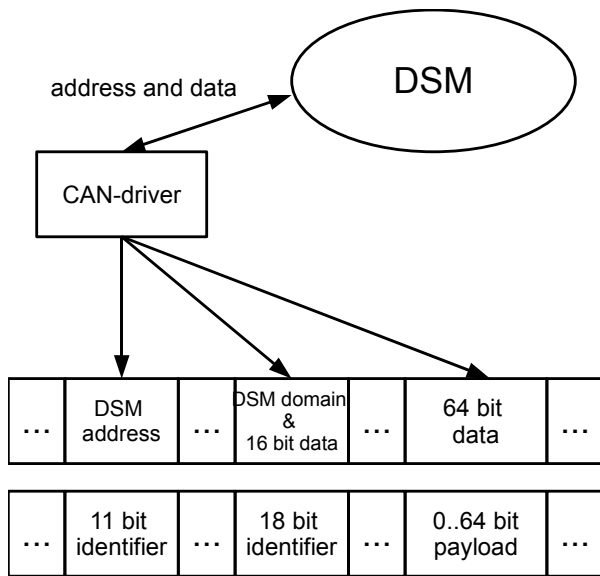


Fig. 4. Mapping between DSM address; usage of CAN messages in this scenario and default CAN frame

scenario the changes made to a set of data (write-update). As a networking standard we adopted the popular CAN version 2.0B (see [9]). In order to maximize the data-rate we extended the payload of a single CAN-frame to 10 bytes by taking 2 bytes from the extended identification field (see figure 4). The driver provides a mapping between the address of an element of the software DSM and the id in the CAN-frame and currently supports CAN-interfaces equipped with the Philips SJA1000 CAN-controller chip.

V. MEASUREMENTS

To meet the specified use-case requirements the software DSM was divided into three partitions; each participant, i.e. station holding the ownership of one partition. The first, straight-forward implementation consisted of three tasks per station, one sending the locally hosted partition of the DSM, the others checking values of the other two remote partitions. Observation of the timing behaviour and inspection of the bus-signals showed that incrementing and transmitting one DSM partition by a single station caused considerable delays. The transmit delays were introduced by the lack of double buffering in the Philips SJA1000 CAN controller. While the packet buffer of the controller was filled the CAN bus was idle. Having only one transmit buffer in the CAN controller, coordination and execution order of the tasks and access to the CAN controller is essential to achieve maximum data rates.

The second, more sophisticated approach distributed the execution of the tasks and the transmission of the data on two stations providing more parallelism. This interleaving minimized the gap between two CAN frames to a nearly minimum interframe-gap specified by the CAN-standard

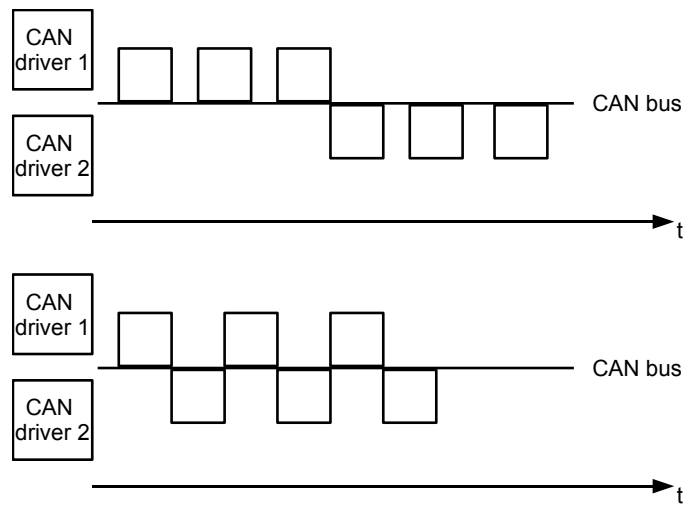


Fig. 5. Burst transmission of CAN packets leaving gaps between messages and interleaved transmission without gaps

consisting of 3 bits (see figure 5). The hardware used in this scenario consisted of one laptop, equipped with an Intel 486 DX/2-66 MHz processor and two desktop PCs equipped an Intel Pentium 133 MHz and an Intel 486 DX/2-50 MHz comparable to the performance of today's micro-controllers. The prototype managed to meet the given requirements with an effective CAN bus data rate of 570 kbit/s, still reaching an idle-time percentage of the CPU of the slowest station of up to 70%.

While current approaches usually leave communication issues to the programmer, the prototype demonstrates a DSM component providing an implicit and therefore simple data exchange between nodes reducing the communication complexity apparent to the programmer. The application programmer only needs to write a set of local processing tasks periodically reading, processing and writing the readily available data. The focus on consistency of the shared data reduces the risk of errors at the level of application logic. CAN messages are particularly suitable for distribution of shared storage entities because the CAN id can be directly mapped to the DSM address. Destination addresses are not required and the transmitted data volume is accordingly reduced.

Actual measurements showed that - depending on the interface used to drive the CAN controller - the gap between CAN messages burst transmission from a single station amounted up to 26 micro-seconds suggesting the use of an interleaved transmission.

The distribution of the tasks and the interleaved packet transmission on more than one station drives the CAN bus to a throughput figure of almost 99% and allows concurrency between execution and packet transmission. The primary requirement in our scenario is to transmit 400 bytes in at most 10 ms. Transmitting 10 bytes per CAN packet allows us to meet this limit and yields a packet rate of 40 packets

| task | time in micro-seconds per execution | CPU load percentage |
|-----------|-------------------------------------|---------------------|
| writer | 55.0 | 0.6% |
| checker | 104.8 | 1.1% |
| scheduler | 4.4 | 6.2% |

Fig. 6. Time measured on an Intel 486 DX/2 66 MHz

in 10 ms (4000 packets per second). The following load is generated on an Intel 486 DX/2 at 66 MHz. Considering the fact, that a CAN frame according to the CAN 2.0B standard amounts to 130 bits (without bitstuffing), the data volume amounts to 5200 bits per 10 ms or 520000 bits/s. Without bitstuffing a data rate of 520 kbit/s would suffice, but leaving an allowance for larger packets caused by bitstuffing the data rate must be raised to 570 kbit/s.

The prototype easily meets the imposed performance requirements and provides ample resources for more complex computations at the application level. This is achieved with a simple shared memory model, with a simplified packet transmission and with an efficient but light-weight tool-chain.

VI. CONCLUSION & FUTURE WORK

The prototype demonstrates the effectiveness of distributed memory schemes in embedded or automotive environments. The periodic data exchange implied by the memory consistency model relieves the programmer from the burden of message based synchronisation and selective station addressing while still providing the functional flexibility found in current systems. The use of a compiler which integrates a complete toolchain and a powerful package concept allows the creation of more flexible communication schemes. A conceptually type-safe language like Java instead of C reveals programming errors at an early stage of the development process. Contrary to common belief a properly designed compiler can generate competitive code - both in terms of execution speed and code density. As the language semantics are platform independent initial program development can occur on a desktop system and then be migrated to a small-scale ECU.

Modularity and extensibility is offered by the object orientation in the compiler. New devices can be added by providing a compatible extension of the current device class. Other communication media might be supported by replacing the packet driver class.

A prototype of an automatic/dynamic configuration facility for new devices is being planned where the different components can register by name, by attribute and by communications requirement - doing away with the static configuration requirement of current systems.

Separating communication issues from application issues paves the way to self-configuring devices with "plug&work" capabilities. New devices may be connected to the bus and reserve space in the DSM for the data they are contributing.

The data is accessible by name after registering at an indexing or naming service, which can be examined by all participants of the distributed system. This service also takes care of assigning DSM areas to single participants avoiding ambiguous allocations. Consequently a change in the layout of the distributed system does not trigger its recompilation or restructuring.

Beyond these structural aspects, the notion of data consistency is of utmost importance in distributed systems. In the automotive environments, introducing different consistency domains in the DSM for data with different refresh rates gives best utilization of available bandwidth while still guaranteeing the validity of data.

In order to step beyond the prototype stage in our experiment the prototype will be ported to microcontrollers used in the automotive environment. These microcontrollers usually comprise communication features contributing to the potential performance of the prototype. Due to the modular structure of SJC, it is easy to support new architectures. Currently, we are working on a backend for Atmel microcontrollers targeting support for AT90CAN128¹ microcontrollers with built-in CAN controller. Both, the compiler and the prototype represent a complete testing framework, the first with respect to other CPU architectures, the latter with respect to other bus systems and their drivers.

REFERENCES

- [1] M. Schoettner, S. Frenz, R. Goeckelmann, P. Schulthess: Checkpointing and Recovery in a transaction-based DSM Operating System, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, Innsbruck, Austria, 2004.
- [2] S. Frenz, M. Schoettner, R. Goeckelmann, P. Schulthess: Parallel Ray Tracing with a Transactional DSM, 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, Chicago, USA, 2004.
- [3] J. L. Keedy and D. A. Abramson: Implementing a Large Virtual Memory in a Distributed Computing System. In Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, 1985.
- [4] K. Li.: IVY: A Shared Virtual Memory System for Parallel Computing. International Conference on Parallel Processing, 1988.
- [5] N. Wirt and J. Gutknecht: Project Oberon, Addison-Wesley, 1992.
- [6] R. Goeckelmann, S. Frenz, M. Schoettner, P. Schulthess: Compiler Support for Reference Tracking in a type-safe DSM: Proceedings of the Joint Modular Languages Conference, Klagenfurt, Austria, 2003.
- [7] <http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/compiler.html>
- [8] <http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/picos.html>
- [9] W. Zimmermann, R. Schmidgall: Bussysteme in der Fahrzeugtechnik, Vieweg, 2006.