Proceedings of the
2007 IEEE Intelligent Vehicles Symposium
Istanbul, Turkey, June 13-15, 2007

A Real-Time-capable Hard- and Software Architecture for Joint Image and Knowledge Processing in Cognitive Automobiles

Matthias Goebl and Georg Färber

Abstract— **Cognitive automobiles consist of a set of algorithms that cover a wide range of processing levels: from low-level image acquisition and feature extraction up to situation assessment and decision making. The modules implementing them are naturally characterized by decreasing data rates at higher levels, because raw data is discarded after evaluation, and increasing processing intervals, as knowledge based levels require longer computation times.**
The architecture presented in this papers offers a method to interchange information with different temporal resolutions liberally among modules with distinct cycle times and real-time demands. It allows effortless buffering of raw data for subsequent data fusion and verification, facilitating innovative processing structures. The paper is completed by measurements demonstrating the achieved real-time capabilities on our selected hardware architecture.

I. INTRODUCTION

This paper presents a real-time-capable method for an open integration of modules written by different researchers to construct software for a cognitive automobile. It illustrates the chosen hardware architecture and a method to attain a tight cooperation between image processing (mostly real-time) and knowledge processing (often non real-time) tasks. Every software module is able to access the published data of all other modules, there is no way for an information gap to arise between two module types, like real-time and non real-time or image and knowledge processing. So we promote open data interfaces in order to encourage exchange of ideas between researchers. At the same time the architecture guarantees hard real-time for modules requiring it.

A. Requirements of a 'collaborative research center'

Within the Transregional Collaborative Research Centre 28 (TCRC28) "Cognitive Automobiles"[1], established at the beginning of the year 2006 by the german research foundation (DFG), researchers from several research disciplines work together to develop a theory of machine cognition and to demonstrate it with the autonomous driving of so-called cognitive automobiles. For a more precise description see [2].

Every involved research institute has its own set of methods, tools and software, most apparently shown by the choice of different programming languages like C, C++, Java, Ada, and others. A prospering collaboration requires common standards, interfaces and tools. The results presented here are the work of the research project C3 within the TCRC28 that is responsible for the hard- and software architecture of the cognitive automobile. It has the important mission to support the overall software integration.

M. Goebl and G. Färber are with Technische Universität München, Institute for Real-Time Computer Systems, Arcisstr. 21, 80333 München, Germany. {goebl,faerber}@rcs.ei.tum.de

B. Previous approaches

There are several architectures for autonomous systems, whose main scope of application are autonomous robots. However, a vehicle in motion on public roads cannot stop for a prolonged cognition cycle like a robot, because of its non negligible inertia. Architectures for autonomous vehicles must consider highly dynamic environments and fulfill stringent real-time requirements.

Well-known middleware concepts like CORBA bring along an overhead in software having memory and processing requirements that are a magnitude too large for our vehicles. Even derivatives like RT-CORBA are quite resource demanding.

A system architecture specially designed for visually guided road vehicles has been proposed in [3]. It is, however, designed for systems that implement the 4D-approach [4]. An agent-based architecture has been presented in [5], that has been changed later to support hard real-time control processes [6]. Both require the application modules to be integrated into a dedicated framework, that has its own scheduler and management tools. A case study in [7] considered the suitability of databases for vehicle control systems.

C. Proposed architecture

This paper presents a comprehensive architecture for cognitive automobiles. It satisfies the requirements of the TCRC by providing interfaces for easy integration. It allows the joint work of real-time and non real-time modules and provides a temporal decoupling. The underlying hardware architecture can be easily duplicated at a reasonable price for every interested research group for simultaneous work.

This contribution is organized as follows: Sec. II gives a brief functional overview of the architecture, sec. III outlines the selection of the hardware architecture, sec. IV presents the developed software architecture including its methods and applications. Sec. V presents figures and evaluation results. Sec. VI summarizes our results and concludes the paper.

II. FUNCTIONAL ARCHITECTURE

Fig. 1 gives a rough overview over the functional architecture of the cognitive automobile. It is based on the architecture proposed in [3] and integrates modules that implement the mentioned 4D-approach [4]. However the developed architecture also supports the investigation of

1-4244-1068-1/07/$25.00 ©2007 IEEE.

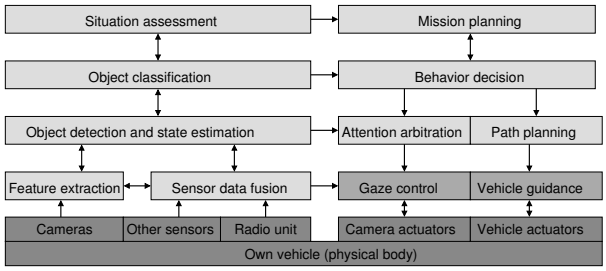

Fig. 1. Functional architecture

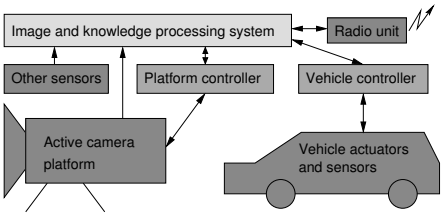

Fig. 2. Hardware architecture

other approaches and allows the combination of them to reach even better cognitive results.

III. HARDWARE ARCHITECTURE

Fig. 2 shows the overall hardware architecture that will be used throughout the TCRC28. The dark gray parts form the physical body of the vehicle (cf. fig.1) and consist of

- various sensors like cameras, RADAR, LIDAR, GPS, inertial, the vehicle's odometry, tachometer, wheel speeds and steer angle, and any future sensors
- the active camera platform as described in [8]
- the vehicle's actuators like steering, throttle and brake control
- a radio unit for the communication with other cognitive automobiles and for remote supervision.

The two medium gray boxes are two embedded systems that guarantee the lowest possible latencies for reliable feedback control loops: The active camera platform is controlled by a board with a MPC565 processor from Freescale Semiconductors. It exhibits the necessary response times needed for an inertial stabilization of the cameras and offers a rich set of hardware input/output-interfaces (I/O) to handle all motors and platform sensors [9]. Commands to the MPC565 are sent over a dedicated CAN link. Due to its safety relevance the control of the vehicle actuators is sourced out into another embedded system, a dSpace AutoBox that is exclusively operated by the responsible TCRC subproject.

The upper left object in fig. 2 is a PC system that hosts all image and knowledge processing modules shown in light gray in fig.1. It must therefore meet sizably requirements:

- Image processing requires fast computation speeds.
- Image acquisition from several cameras at once only succeeds if there is enough I/O-bandwidth available on all traversed busses.
- Knowledge processing needs fast access to large memory areas in order to rapidly find inferences.
- Because most cognitive functions are carried out on this system a parallel execution is necessary.
- The optimal cooperation of all modules requires an efficient inter-process communication with low latencies.
- For logging purposes a powerful storage system is desirable.

The hard- and software architecture of the PC system will be the focus for the remainder of this paper.

A. PC system architecture

Personal computers (PCs), assembled from commercial-off-the-shelf (COTS) components, today are the most frequently used systems for machine cognition. The real-time characteristics of modern multiprocessor PCs have been sufficiently studied in [10], [11], [12] so we have chosen to base our hardware architecture on COTS components. The mass market's demand for consequent backward compatibility conveniently guarantees a smooth migration of our software onto future systems.

In [13] vision based autonomous driving has been accomplished using a cluster of four dual-processor PCs connected with scalable coherent interfaces (SCI). Due to the low mass market penetration the price for SCI remains high.

The TCRC needs a system that can be easily duplicated at an affordable price. So all interested project participants can buy their own laboratory system that is identical to the real target system in the vehicle. A multi-PC solution could lead to an early partition of software modules on different PC, prohibiting a later rearrangement of modules and shifting cooperation from software to less flexible hardware interfaces.

B. Opteron system architecture

In [9] we compared modern multiprocessor architectures and selected the AMD Opteron as the best scaling architecture for growing processor numbers. Fig. 3 shows the internal architecture of an AMD Opteron system. Each node consists of a central processing unit (CPU) with its own (local) memory (RAM) connected by an internal crossbar switch (X). Dual-core processors (not shown for simplicity) contain two CPUs. The nodes are networked by hypertransport links between their crossbar switches. Each hypertransport link provides a bandwidth of $3.2 \cdot 10^9 Byte/second$[1] per direction. The latency for accessing remote memory has been measured[2] to be well below 1μs. Hypertransport hubs and tunnels (T) connect the nodes to the peripheral busses AGP, PCI, PCI-X and PCI express. They interlink the peripheral interfaces (IEEE1394, CAN, other I/O). To sum it up, the chosen system provides enough I/O-bandwidth to saturate a considerable number of video streams.

[1] Our own measurements revealed a usable memory throughput of $3.11 \cdot 10^9 Byte/second$ for the local node and $2.62 \cdot 10^9 Byte/second$ for a remote node on our AMD Opteron 275HE (2.2GHz, DDR-400 RAM) executing the following *not optimized code*, compiled by GCC 3.4.5 with "-O3" and no further optimizations:

```
volatile int32_t mem[2000000];
for(j=0;j<100;j++) for(i=0;i<2000000;i++) mem[i]++;
```

[2] <110 ns in [14] and bounded at $330ns+130ns\cdot(n-1)$ for n subsequent accesses in [15]

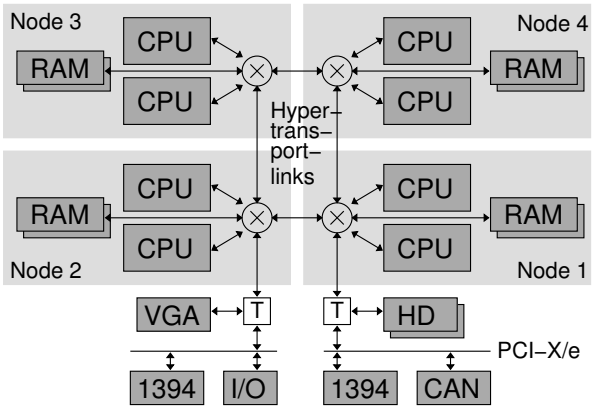

Fig. 3. Architecture of AMD Opteron systems

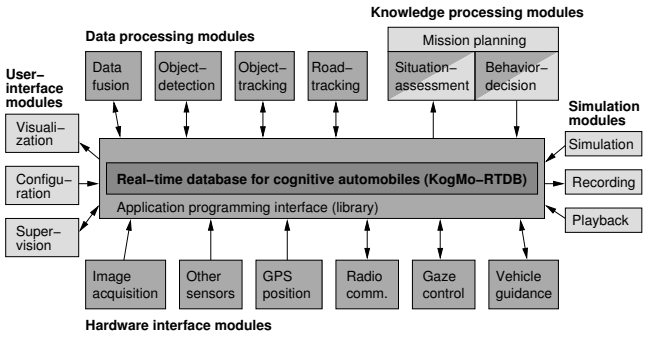

Fig. 4. KogMo-RTDB architecture

The Opteron in this architecture can be regarded as a "cluster-in-a-box". Compared with a real cluster it only needs infrastructure components once: Storage media, console devices and power supply. Scalability is yet limited to the current development in multi-core processors (quad-core is announced) and boards (quad-CPU boards are already available on the mass market), so a quad-quad-system will be soon possible. In contrast to mobile processor families, the Opteron is optimized for server performance and not power saving. But there are selected CPUs with a thermal design power (TDP) of 55 Watt (named 'HE') and 30 Watt (named 'EE') available.

For the cognitive automobile we currently use two 2.2GHz dual-core 275HE[3] low-power CPUs that consume a total of just 160W including the mainboard and 4GB RAM. The complete system costs below $4500.

IV. SOFTWARE ARCHITECTURE

The cognitive automobile can only navigate safely through real traffic if the timely processing of sensor data and its translation into actuator commands it ensured. On one hand this requires efficient algorithms for cognition. On the other hand this needs a runtime environment to provide the essential computational and memory resources. The following section points out how we reached that goal. It gives a short overview and then studies the involved building blocks in detail.

As basic operating system (OS) the free UNIX derivate Linux has been selected. Due to its open source nature it can be easily extended and allows a deep analysis of its internals and critical execution paths necessary for implementing real-time functionality. Our integration framework sits mainly on top of the OS. It serves as interface between all cognitive modules in the vehicle.

A. Real-time database KogMo-RTDB

Fig. 4 visualizes the developed integration framework. Its center exposes our real-time database for cognitive automobiles named "KogMo-RTDB". It serves as central hub for information. All relevant information available within the automobile is published in this database, where it can be openly accessed by each other module. The only interface between modules are database objects, so a maximum transparency is warranted. This kind of communication prevents unnecessary and time consuming data forwarding between modules from different cognitive layers.

Below the database there are interface modules whose only jobs should be feeding raw data from the cameras and other sensors into the database, and forwarding commands for the vehicle and camera platform to the actuators. However it is not strictly forbidden by the architecture for interface modules to preprocess acquired images. This has to be negotiated by the participating researchers.

Ideally the image and knowledge processing modules shown above get their input data from the central database and publish their results, so the vehicle's knowledge is represented in the database. This open concept makes it easy to visualize the vehicle's situation just by querying the database. It also allows us to feed the database with simulation data, and to record and replay situations.

The KogMo-RTDB needs an easy-to-learn interface, because it is used by all participating researchers and their students. We designed an intuitive database-like interface, available as a shared library, and provide methods to

- publish and delete data
- update data
- search and retrieve data
- wait for updated and new data from other modules.

All data is organized as "objects", that need not necessarily be C++ objects. Every database object is structured as depicted in tab. I: The static block is generated at the creation of an object, the dynamic block contains the actual user data that is normally updated in every cognitive cycle. OID denominates an object identifier that is kept unique and never reassigned after object deletion. One could use bit slicing like in IPv6 and 64 bit to maintain uniqueness across vehicles. PID refer to the internal OID of process objects. TS are timestamps, see below. $DATA_{user}$ are n_{bytes} bytes of user data with a maximum size of $n_{bytes,max}$ bytes.

The definition of new objects is the responsibility of all project participants. New objects are defined by the respective TCRC project that wants to publish its results into

[3]EE-CPUs are not yet available as dual-core

TABLE I
STRUCTURE OF A RTDB OBJECT

static information
OID, $name$, TID, $permissions$, $n_{bytes,max}$, $T_{history}$, t_{cycle}, $TS_{created}$, $PID_{created}$, $TS_{deleted}$, $PID_{deleted}$
dynamic data
$TS_{committed}$, $PID_{committed}$, TS_{data}, $DATA_{user}$, n_{bytes}

the RTDB. The definitions are organized in a TCRC-wide source code repository and coordinated in working groups. There is no need to touch the database internals to support new objects. The RTDB is responsible solely for managing the data. Functions to work with the data contents can be submitted into the source code repository for common use.

B. History buffer

For the data block of every object the RTDB creates a circular buffer. It manages the temporal decoupling of accessing modules. We assign it the following amount of history slots:

$$n_{slots} = \frac{T_{history}}{t_{cycle}} + 1$$

$T_{history}$ denominates the desired time span that old data should remain available. t_{cycle} is the minimal expected update cycle time of the object. It must be given at object creation time and defaults to the parameter $t_{process}$, the minimal cycle time of the cognitive module, that accompanies every database connection.

The KogMo-RTDB features a consequent use of timestamps (TS): Every single action within the database is stamped with the current time and date, every query requires a timestamp. When performing multiple queries on the same situation it is important to include the same timestamp, so the database will yield the corresponding data from the history buffer, that has been valid at the given point of time. That method provides slower cognitive modules with an coherent view at the situation represented in the database for one particular moment. Due to the nature of the ring buffer, information older than $T_{history}$ will be overwritten, so the worst case execution time $t_{WCET,reader}$ for a reader's algorithm accessing a particular set of n current objects $\mathcal{D} = \{D_1, \ldots..., D_n\}$ must obey to

$$t_{WCET,reader} < \min_{D \in \mathcal{D}} \{T_{history,D}\}$$

It is possible to intentionally retrieve object data from a past time t_{age}, for example in order to search for patterns and calculate trends. However that data is only available for an even shorter certain period of time:

$$t_{WCET,reader} < \min_{D \in \mathcal{D}} \{T_{history,D}\} - t_{age}$$

C. Timestamps and clock synchronization

For the resolution of the timestamp different possibilities have been taken into consideration:

resolution	size	validity
millisecond	32 bits	$<$ 50 days
microsecond	32 bits	$<$ 2 hours
nanosecond	32 bits	$<$ 5 seconds
nanosecond	64 bits	$<$ 585 years

A millisecond resolution is too inaccurate, a microsecond timestamp will wrap-around to often with 32 bits, so we decided to use 64 bit timestamps. That allows us to provide a nanosecond resolution. Even when we use a signed value it is valid for over 292 years. We use signed absolute timestamps beginning at 1970–01–01[4], that will be valid until 2262. Another benefit is that all produced data including videos will carry that absolute value, so from just looking at the timestamp and GPS position we can for example deduce the season. Because the RTDB adds the current timestamp to all data committed to the database, we can calculate the runtime of modules just by looking at their timestamps.

The timestamps' actual accuracy is yet limited by the clocks being used. In our system we use the timestamp counter (TSC), that exists in every x86 compatible CPU since the Pentium family. The TSC synchronization of all CPUs is accomplished by the Linux kernel at boot time with an accuracy that is within the order of transferring a cache line to another CPU [16]. According to sec. III-B this is well below 1μs. During the runtime of the system all CPUs are driven by the same oscillator, so when not using energy saving techniques that manipulate the cpu frequency scaler, all TSC stay synchronized.

The external synchronization can be assured by GPS. The resulting accuracy is determined by the interrupt latency, that we have measured for our Opteron at 6.2μs-26.4μs.

D. Database locking and consistency

The main intention for the RTDB was to encourage module developers to publish their data. So it is important not to disadvantage them when doing so with prolonged blocking times for write operations. To satisfy this requirement a non-blocking write protocol similar to [17] has been developed. This method makes a very efficient use of the circular buffers mentioned earlier. However it relies on the data writer to stay within its specified $t_{cycle,object}$ for object updates. If the writer publishes updates at a higher rate $t_{cycle,writer}$ and the object has no 'cycle-watch'-flag set, the valid history time T_{valid} shrinks to

$$t_{valid} = \frac{T_{history,object} \cdot t_{cycle,writer}}{t_{cycle,object}}$$

All database operations are performed in the context of the calling module, so the copying of data from the database can be preempted by modules with a higher priority and thus prolong it by a factor $\tau_{preempt}$. So a requested data block with an given age $t_{age,object}$ must be valid until the end of

[4] 1970-01-01 00:00:00 UTC (the UNIX 'epoch') without leap seconds

the copy process whose runtime depends on the object size $n_{bytes,object}$:

$$(t_{copy,header} + t_{copy,byte} \cdot n_{bytes,object}) \cdot \tau_{preempt} < \frac{T_{history,object} \cdot \min(t_{cycle,writer})}{t_{cycle,object}} - t_{age,object}$$

As $t_{copy,byte} \cdot n_{bytes,object}$ grows significantly for huge blocks like video images, it is wise to first examine the data on the blackboard directly using a 'direct-read'-pointer given by the RTDB before asking for a local copy. Depending on the configured properties of an object it can automatically activate locks e.g. when allowing more than one module to update it.

E. Dynamic storage allocation

Another important aspect in database systems is dynamic storage allocation (DSA) for new objects and its deallocation after deletion. Standard DSA algorithms are designed for a good average performance and can expose significant worst case execution times (WCET). Masmano et al. [18] proposed a bounded time O(1) good-fit allocator called 'two-level segregated fit memory allocator' (TLSF) that has been integrated into the RTDB. If an object of the same type is allocated and deallocated very often, the RTDB allows the user to keep its allocated storage for faster reuse. That method ensures optimal results for the steady state.

F. Application programming interface

The KogMo-RTDB provides a C-interface, because this is a common denominator that most other programming languages can work with. It contains C++ classes for convenience and data generated by them can be read back from any other language. An older version of the database has been successfully used in conjunction with a knowledge processing module described in [19] and written in Java using the SWIG toolkit. By using GNAT a successful interface experiment has been passed for Ada.

Fig. 5 gives a short example in C++ to understand the simplicity of the programming interface. The given code retrieves camera images from the RTDB, runs a road-tracker on it and writes the result back to the database. The `RTDBConn` objects opens a connection to the local system database `local:system`. On our simulation system we can run several instances of KogMo-RTDB with different names. The objects `C3_Image` and `A2_RoadKloth` have been defined by the TCRC projects C3 and A2. `RTDBReadWaitNext()` implements a notification mechanism and prevents an inefficient polling. The road-tracker (not shown here) is described in [20]. Please note the transfer of TS_{data} for a correct temporal alignment of the results in later processing stages.

```
// Establish a connection to the KogMo-RTDB as a module
// named a2_roadtracker with a cycle time of 33 milliseconds
RTDBConn DBC ( "a2_roadtracker", 0.033,
               "local:system" );

// Wait for an object with an image of the camera video
// provided by the research project C3
C3_Image Video ( DBC );
Video.RTDBSearchWait ( "c3_camera_left" );

// Insert an Road-Object that describes the lanes with
// a model of clothoid segments
A2_RoadKloth Road ( DBC, "a2_egolane" );
Road.RTDBInsert ()

for(;;) {
 // Wait until the next camera image is available and fetch it
 Video.RTDBReadWaitNext ();

 // Run our road-tracker on the given image
 Roadtracker.Run ( Video.getImage () );

 // Copy the resulting clothoid segments into the road object
 Road.setKloth ( Roadtracker.getKloth () );

 // Transfer the data timestamp
 Road.setTimestamp ( Video.getTimestamp () );

 // Commit the new road data to the KogMo-RTDB and make it
 // available for subsequent modules that wait for it
 Road.RTDBWrite ();

 // Signal own vitality and give a simple synchronization point after
 // updating more than one object and for monitoring/debugging
 DBC.CycleDone ();
}
```

Fig. 5. Example application using the KogMo-RTDB API

G. Hard real-time operation system

The kernel of the Linux operating system has not been designed to fulfill hard real-time requirements. However there are approaches to add real-time capabilities. RTLinux [21] and RTAI-Linux [22], [23] implement a dual kernel technique by adding a real-time domain that is strictly isolated from the standard Linux system. The real-time modules are run in kernel space, giving them the best possible interrupt latencies and direct hardware access. On the flipside the real-time modules have no memory protection, are difficult to debug and have limited C++ abilities. So it would not be feasible to ask all TCRC project partners to write dedicated Linux kernel modules.

A possible solution shows RTAI/LRXT [24]: It allows the execution of real-time tasks in user space, giving them a memory protection and better C++ abilities. But debugging is difficult and it needs its appropriate real-time API.

Therefore we decided to use Xenomai [25], formerly know as RTAI/fusion: It offers a better integration with the Linux kernel and can profit from future Linux soft real-time enhancements. Application are started as ordinary Linux tasks, having only soft real-time capabilities in a mode called 'secondary mode'. As soon as they issue real-time calls they are being switched to the 'primary mode' and scheduled by a real-time scheduler. As long as they do calculations and issue only further real-time calls they are granted real-time conditions. It possible to use Linux system calls like `printf()` and standard unix debugging techniques, but this causes them switch back to the non real-time mode. Xenomai offers a POSIX skin that allows applications that use POSIX real-time calls to gain hard real-time capabilities. It requires relinking of the application to intercept the POSIX calls.

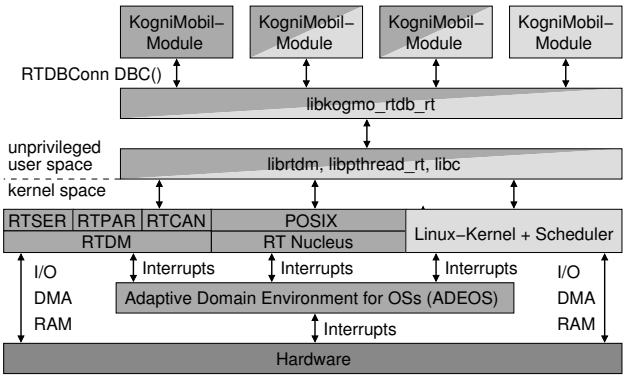

Fig. 6. KogMo-RTDB real-time architecture

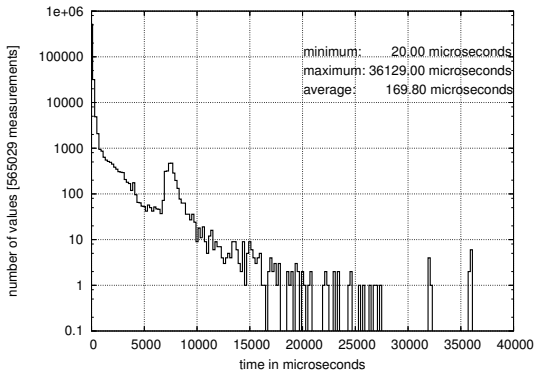

Fig. 7. KogMo-RTDB: Non real-time IPC latency (heady system load)

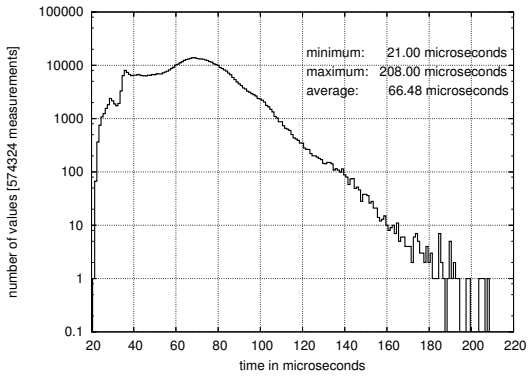

Fig. 8. KogMo-RTDB: Real-time IPC latency (heady system load)

H. Hard real-time architecture for KogMo-RTDB

Fig. 6 shows the hard real-time architecture for the real-time database KogMo-RTDB. On the top there are the modules that host the cognitive algorithms, implemented as standard Linux processes. They call the KogMo-RTDB API packed into `libkogmo_rtdb_rt`. The KogMo-RTDB operates on top of the Xenomai libraries. On the real-time side (dark gray) we use the ADEOS nanokernel plus the Xenomai real-time nucleus and its POSIX skin. For hardware access we prefer drivers conforming to the real-time driver model (RTDM) [26]. The system may consist of a mixture of real-time modules (dark gray), non real-time modules (light gray) and even modules that alternate (light and dark). The corresponding colors in fig. 4 illustrate our proposed assignment.

Note that all cognitive modules are implemented in user space, protecting them one against each other and allowing easy supervision by a watchdog module (not shown). If one module crashes, the watchdog can even initiate an emergency break maneuver using the last known situation data available in the database.

For offline experiments and development we also provide `libkogmo_rtdb` (without `_rt`), a non real-time version of the KogMo-RTDB. Its main advantage is that it does not need a modified Linux system. We went even further and took the binary of a Qt visualization application compiled against `libkogmo_rtdb`, preloaded it with `libkogmo_rtdb_rt` on a real-time platform and had a successful visualization. In this case the system switches to real-time mode for all KogMo-RTDB API calls, making use of all necessary real-time methods. So the library safely obeys the priority inversion protocol when using mutexes within the database. That method ensures that modules cannot violate our real-time regulations.

V. EXPERIMENTAL RESULTS

We evaluated our architecture by measuring the execution times of key operations of our real-time database. This includes the influences caused by our selected hardware architecture and underlying operation system. We used different system configurations as shown in tab. II: In the ‘idle’ configuration only the module under measurement is started. No effort has been made to isolate it, there are still some other applications running (system services and user interface). So the relevant figure is the average time that shows the low overhead of the architecture. The ‘heavily loaded’ configuration shows the necessity for the real-time variant of the developed architecture. The system is running a synthetic compilation of applications in order to stress I/O, memory and CPU. The measurements depict the efficiency of our real-time capabilities. In the non real-time mode the system reveals rising response times, whereas in the real-time mode the systems responds within certain bounds. The operations we evaluated are inserting and deleting whole objects, and reading and writing object data. Additionally we measured the time between the update of object data by one module and the notification and data retrieval by another module that is subscribed to object changes. This characterizes the latency time for inter-process communication (IPC). Fig. 7 shows the histogram for the distribution of IPC times under the given system load within the non real-time configuration, fig. 8 contrasts these times with our real-time configuration. Regarding these figures our system architecture offers a low overhead in all configurations and is able to guarantee real-time conditions even in heavily loaded situations.

TABLE II

MEASURED EXECUTION TIMES FOR KOGMO-RTDB OPERATIONS

	system configuration											
	non real-time						real-time					
	idle			heavily loaded			idle			heavily loaded		
operation	**measured execution time** in μs											
	average	min.	max.	average	min.	max.	average	min.	max.	average	min.	max.
insert object	60.5	53	154	122.5	39	93109	45.3	38	77	75.6	38	273
delete object	5.2	4	28	18.5	4	41250	9.0	8	20	18.5	8	131
read object	4.6	4	16	17.0	4	10721	5.2	4	18	16.8	4	62
write object	8.3	5	51	22.6	5	181681	10.3	8	28	25.6	6	134
receive notification	29.6	23	241	169.8	20	36129	31.6	27	96	66.5	21	208

VI. CONCLUSIONS AND FUTURE WORKS

A. Conclusions

We presented a comprehensive architecture for cognitive automobiles. A PC system delivers the necessary computational power, that at the same time meets our latency time requirements. We selected an extensible real-time operation system that we use to combine hard and soft real-time tasks. We developed a real-time database that provides transparent interfaces between modules for cognitive processes. It successfully serves as an integration platform for a TCRC, a locally distributed research center. It has been well adopted: Participants successfully created interfaces to existing applications like a vision system, a simulation environment, a behavior generator and a vehicle controller. It has been deployed in our vehicles and also used for simulations. Even a suitable 3D visualization application has been developed.

B. Future Works

Future work will make use of the observation capabilities of the KogMo-RTDB: The execution times of modules can be read directly from the database and used for developing a scheduling strategy. The OS scheduler will be modified to enforce the strategy, profiting from the non-blocking kernel access to all published data. We also work on real-time image capture methods that conform to the real-time driver model.

VII. ACKNOWLEDGMENTS

The authors gratefully acknowledge support of this work by the Deutsche Forschungsgemeinschaft (German Research Foundation) within the Transregional Collaborative Research Centre 28 'Cognitive Automobiles'. We also thank our colleagues for valuable discussions.

REFERENCES

[1] Transregional Collaborative Research Centre 28. [Online]. Available: http://www.kognimobil.org

[2] C. Stiller, G. Färber, and S. Kammel, "Cooperative Cognitive Automobiles," in *Proc. IEEE Intelligent Vehicles Symposium*, 2007.

[3] M. Maurer and E. D. Dickmanns, "System architecture for autonomous visual road vehicle guidance," in *IEEE Intelligent Transportation Systems*, 1997, pp. 578–583.

[4] E. D. Dickmanns, "The 4D-Approach to Dynamic Machine Vision," in *Proc. IEEE Decision and Control*, vol. 4, 1994, pp. 3770–3775.

[5] S. Görzig and U. Franke, "ANTS - Intelligent Vision in Urban Traffic," in *IEEE Conf. Intelligent Transportation Systems*, 1998, pp. 545–549.

[6] A. Schmidt, S. Görzig, and P. Levi, "ANTSRT - Eine Software-Architektur für Fahrerassistenzsysteme," in *Autonome Mobile Systeme 2003*. Springer-Verlag, 2003, pp. 44–55.

[7] D. Nystrom, A. Tesanovic, C. Norstrom, J. Hansson, and N. Bankestad, "Data management issues in vehicle control systems: a case study," in *Proc. IEEE Euromicro Conference on Real-Time Systems*, 2002, pp. 249–256.

[8] T. Dang, C. Hoffmann, and C. Stiller, "Self-calibration for Active Automotive Stereo Vision," in *Proc. IEEE Intelligent Vehicles Symposium*, 2006, pp. 364–369.

[9] M. Goebl, S. Drössler, and G. Färber, "Systemplattform für videobasierte Fahrerassistenzsysteme," in *Autonome Mobile Systeme 2005*. Springer-Verlag, 2006, pp. 187–193.

[10] J. Stohr, A. v. Bülow, and M. Goebl, "Einflüsse des PCI–Busses auf das Laufzeitverhalten von Realzeitsoftware," Institute for Real-Time Computer Systems, Technische Universität München, Tech. Rep., 2003.

[11] J. Stohr, A. von Bülow, and G. Färber, "Controlling the Influence of PCI DMA Transfers on Worst Case Execution Times of Real–Time Software," in *Proc. 4th Intl. Workshop on WCET Analysis in conj. with the 16th Euromicro Conference on Real–Time Systems*, 2004.

[12] J. Stohr, A. von Bülow, and G. Färber, "Bounding Worst-Case Access Times in Modern Multiprocessor Systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[13] R. Gregor, M. Lutzeler, M. Pellkofer, K. H. Siedersberger, and E. D. Dickmanns, "EMS-Vision: a perceptual system for autonomous vehicles," in *Proc. IEEE Intelligent Transportation Systems*, vol. 3, 2002, pp. 48–59.

[14] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The Opteron processor for multiprocessor servers," in *IEEE Micro*, vol. 23, 2003, pp. 66–76.

[15] J. Stohr, "Auswirkungen der Peripherieanbindung auf das Realzeitverhalten PC-basierter Multiprozessorsysteme," Ph.D. dissertation, Institute for Real-Time Computer Systems, Technische Universität München, Mar. 2006.

[16] Sources of the Linux operating system kernel. [Online]. Available: http://www.kernel.org

[17] H. Kopetz and J. Reisinger, "The non-blocking write protocol: solution to a real-time synchronization problem," in *Proc. IEEE Real-Time Systems Symposium*, 1993, pp. 131–137.

[18] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *16th Euromicro Conference on Real-Time Systems*, 2004, pp. 79–88.

[19] A. D. Lattner, J. D. Gehrke, I. J. Timm, and O. Herzog, "A Knowledge-based Approach to Behavior Decision in Intelligent Vehicles," in *Proc. IEEE Intelligent Vehicles Symposium*, 2005, pp. 466–471.

[20] S. Neumaier, P. Harms, and G. Färber, "Videobasierte Umfelderfassung zur Fahrerassistenz," in *4. Workshop Fahrerassistenzsysteme*, Löwenstein, Oct. 2006.

[21] V. Yodaiken, "The RTLinux manifesto," in *Proc. of The 5th Linux Expo, Raleigh, NC*, Mar. 1999.

[22] E. Bianchi, L. Dozio, G. Ghiringhelli, and P. Mantegazza, "Complex Control Systems, Applications of DIAPM–RTAI at DIAPM," in *1st Realtime Linux Workshop*, Vienna, 1999.

[23] "RTAI - official website." [Online]. Available: http://www.rtai.org

[24] E. Bianchi and L. Dozio, "Some Experiences in fast hard realtime control in user space with RTAI-LXRT," in *2nd Realtime Linux Workshop*, Orlando, 2000.

[25] P. Gerum, "Xenomai - implementing a RTOS emulation framework on GNU/Linux," Apr. 2004. [Online]. Available: http://www.xenomai.org

[26] J. Kiszka, "The Real-Time Driver Model and First Applications," in *Seventh Real-Time Linux Workshop*, Lille, Nov. 2006.