JPURE - A PURIFIED JAVA EXECUTION ENVIRONMENT FOR CONTROLLER NETWORKS[1]

Danilo Beuche,
Lars Büttner,
Daniel Mahrenholz ,
Wolfgang Schröder-Preikschat,
Friedrich Schön*

University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
{danilo,lbuettner,mahrenho,wosch}
@ivs.cs.uni-magdeburg.de

* *GMD-FIRST*
Rudower Chaussee 5
D-12489 Berlin, Germany
fs@first.gmd.de

This paper presents an approach how to make microcontrollers able to execute Java applications with very small resource consumption compared to existing Java execution environments. The approach is based on the exploitation of the distributed computing power available in distributed controller network.

1. Introduction

About 98 % of the over eight billions processors produced in year 2000 will be used in the embedded systems market [11]. From these about 57 % will be 8-bit processors. Many of these microcontrollers will be interconnected using a networking technology that has little in common with the Internet. Rather special purpose technologies such as CAN, FireWire or BlueTooth are used to establish a controller network. Interesting

[1] This work has been partly supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1 and the Bundesministerium für Wirtschaft (BMWi), grant no. 01 MS 801/7 . measurements

examples for such networks, or distributed systems, are today's cars. As cars are mass production goods, each cent counts and there is always a pressure to use the cheapest technology possible.

Next generation cars will be connected to the Internet via gateways to receive information from and also provide information to the outside. Car users will be able to run different applications on the computing infrastructure (e.g. computer assisted navigation). These applications have much in common with normal desktop applications, i.e. the computational power needed is much higher than available in today's controller networks. This power will be provided by embedded processors which have the power of desktop processors. These supplementary processors have to cooperate with the controller network to access information about the car and to control its behaviour. The high processing power of the application processors allows the use of modern software techniques and languages.

The Java language and its supporting technologies like RMI are very interesting in this context as Java allows writing of portable code, which runs on different systems in the same way. So different cars can have different hardware, as long as it is able to run Java. For instance a car navigation application will run in all cars. Ideally all processors in the car should be able to run Java applications.

Unfortunately the microcontrollers currently used in the controller networks are not suited to run “normal” Java applications due to resource limitations. The controllers are not equiped with enough RAM and ROM to run a standard Java execution environment. Upgrading a node to make it ready for Java, is in many cases not feasible due to limited budget, electrical power consumption or heat dissipation.

In this paper we discuss different possible ways to make embedded nodes Java ready. We then present our solution for this problem which based on distributed computing principles. Based on measurements with our prototypical implementation we discuss the feasibility of the approach. The last section contains concluding remarks and presents some plans for future developments in the project.

2. The Network is the Computer

An escape from the above described dilemma is resource sharing in a distributed environment. To use Java with all its power even on small nodes with limited resources, different ways are possible:

Distributed applications. The application itself is designed distributed and some parts are remotely executed. This approach requires a full JVM with RMI (or similar mechanisms) support. The engineers have (even for simple) applications to worry about distribution or packages which hide the distribution from the programmer. Examples are BORG from Microsoft [3] which provides the view of a single JVM to applications running on different nodes. The cJVM is a similar solution for a CPU cluster from IBM [2].

Distributed libraries. The applications computation is done on the local node, but the libraries which form the API are implemented distributed. Because applications use only the standardized API, the distribution is transparent to them. But this requires reimplementation of the JDK API. The minimum requirements are a JVM and some means of communication system accessible from Java.

Distributed JVM. The JVM itself implements its services in a distributed manner. This allows the use of unmodified Java libraries and applications. Resource intensive parts of the JVM are located on different nodes. For example the class loader and byte code verifier could be a candidate for remote execution. The Sun kvm environment [4] e.g. uses a remote preverfier and packager for byte code preparation. The Jbed [4] Java enviroment can use a bytecode-to-native compiler which can be either run on the target node (for dynamic byte code compilation) or on a host computer (for precompilation into machine code)

Distributed JVM runtime support. The JVM requires a number of services to be provided by the base system. For instance the class loader requires access to files containing class implementations. Many of these services can be implemented by some kind of remote invocation of service on other machines.

Scalable JVM. The JVM itself is scalable. It provides only the functionalities required by the actual applications. E.g. if no object is ever destroyed no garbage collection support is necessary. Suns kvm is a good example for this approach. But the kvm approach doesn't work automatically, the user must decide which configuration is required. Another idea is to omit a real JVM altogether and use a Java-to-Native compiler. The compiler generates machine code from the Java application and the application linker puts together the required parts of the runtime system. An example for this approach is GCJ [10].

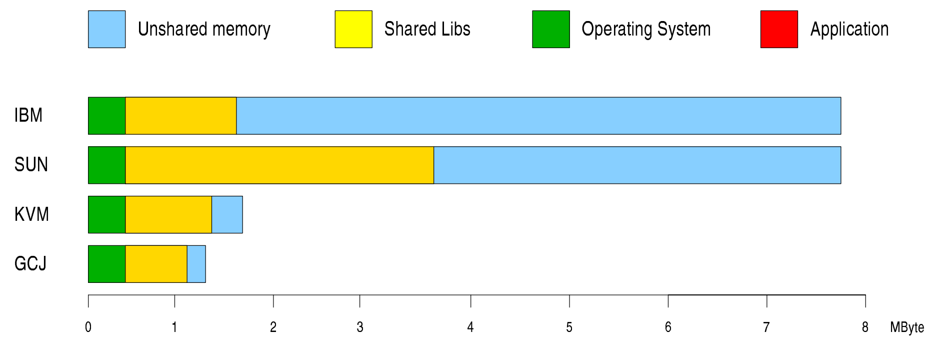

Figure 1: Memory usage of HelloWorld

These approaches are not mutually exclusive but can and should be used combined in order to achieve minimal resource usage on the node. Especially the approach which require a full JVM on a node are not stand-alone usable in most embedded contexts. Our measurements[2] depicted in figure 1 for a simple “HelloWorld” showed an enourmous amount of memory required by standard Java environments. The two JVM

[2] Figures were taken on a x86-linux system with IBM JDK 1.3.0, Sun JDK 1.3.0, Suns KVM from CLDC J2ME and GCJ 2.95.2

implementations required around 8 MBytes of memory. Even the `kvm`[3] configuration used more than a one Mbyte although Sun claimed in its whitepaper configurations with only 128k memory space are possible. The smallest amount of memory was used by the machine code executable generated with GCJ. But typical controllers have RAM below the 4k margin and ROM between 4k up to 256k.

Especially the approach to realize parts of the Java runtime support remotely can be very effective. For example Java makes use of TCP/IP protocols for communication purposes. Instead of using a local TCP/IP stack implementation on each node a remote stack could be used. Although it might look strange to use remote access to a communication stack, it can pay off. The communication between the node and the remote stack can be tailored to the network architecture of the controller network. The assumptions of a TCP/IP stack about networks are different from such a network, especially regarding reliability, communication latencies etc. So a much smaller implementation of communication layers are possible on embedded nodes compared to the typical size of a TCP/IP stack of 50k to 100k.

3. Tailoring Java

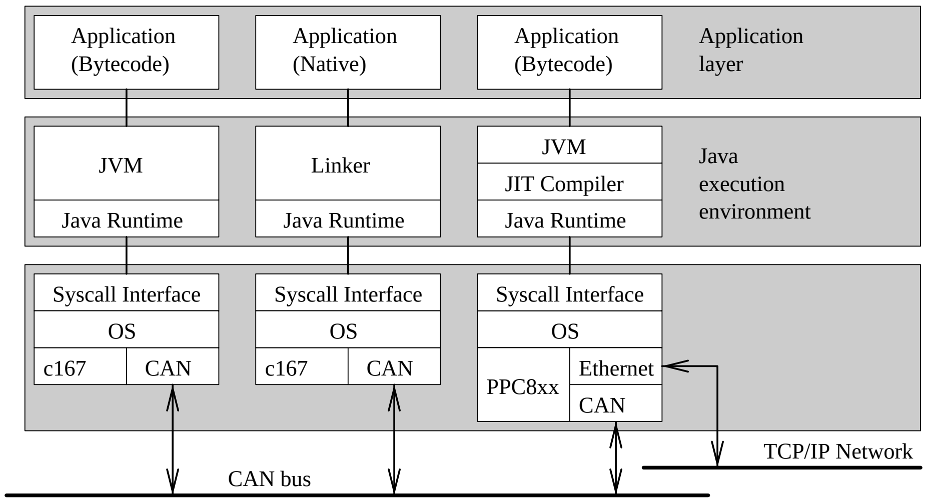

Figure 2: Sample embedded car network

One important constraint for our solution is the ability to support the complete Java language. Therefor the full JVM functionality must be available even on small nodes, not a restricted one like the JavaCardVM [6]. But closer examination shows that most applications in the embedded area do not make full use of the JVM but require only certain subset of functions. For instance many applications do not require dynamic class loading. Other applications do not need garbage collection. Some applications

[3] See the `kvm` whitepaper on [7] for a more detailed disscussion

may not need any of these services. So rather than implementing only one solution that fits all we propose to use a family of JVM implementations. Depending on the required functionality and available resources on the target node, different configurations can be used.

Figure 2 depicts a possible configuration. The rightmost system is the most powerful node and uses a normal JVM. It implements a gateway and acts as a host node that provides remote access to services. The two other nodes communicate via CAN bus protocols with that node and can access the Internet using the host node's TCP/IP stack. In general, those services which would not fit together with the application on a node need to be provided via a remote host node. In the shown setup the left node uses a JVM which executes bytecode and the middle node executes a Java application which has been compiled into native machine code.

3.1 Overview

Our solution for building an embeddable Java execution environment is based on the combination of open source tools and our embedded operating system family Pure [1]. The Java environment is provided by the GCJ Java-to-Native compiler with its runtime support library `libGCJ`. For the (optional) execution byte code the KaffeVM [5] can be embedded. The basic blocks of the our execution environment are shown in figure 3.

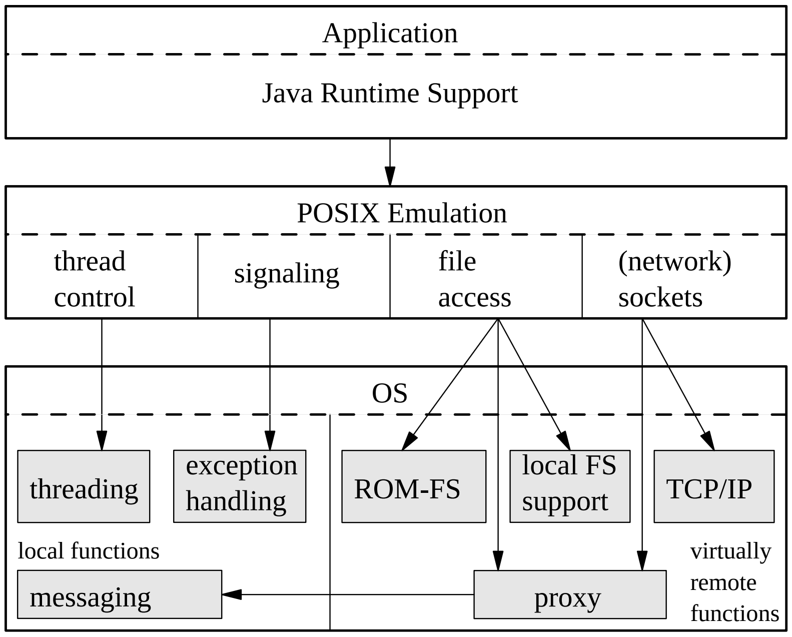

Figure 3: Runtime system building blocks

3.2 Java Runtime Support Layer

The `libGCJ` provides the runtime support for the machine code generated by the GCJ. It is a C++ library which provides the functionality of the JDK 1.1.8 from Sun with some JDK 1.2 extensions[4].

The original `libGCJ` is not suitable for embedded targets. Although it required the lowest amount of memory for our HelloWorld example, it is far from being able to run in deeply embedded platforms. The main problem is the close coupling of the different library functionalities. Even if a specific function is not used, it is included in the executable. For instance, standard stream objects are always intialized. This causes all scalar data type functions to be included (conversion functions etc.).

We modified the `libGCJ` to be more modular. Different methods were used for achieving this. The first step was to introduce conditional compilation statements into the code which allows for static, compile time decisions about functionalities provided by the library. By doing this we can do coarse grain configuration. We applied this technique for instance to the garbage collector or the floating point support. But this method is not applicable in every case. Using conditional compilation statements everywhere for configuration purpose can make the source code difficult to read and maintain. This is especially relevant for configurable properties which are not implemented in a single place but have their code distributed among many different parts of the system.

For further optimization of the existing `libGCJ` we use a combination of tools which analyze and modify binary object code stored in the library. This allows us to exchange a function in a library with a different implementation without changing the source. We use this to replace for instance the unwanted initalization functions for stream objects with a void function.

Further modifications which allow fine grained apdaptation of the `libGCJ` functionalities require replacement of parts of the library with new implementations of them. The idea is to replace the general-purpose one-size-fits-all implementation not by a single new implementation but rather use a family based approach [8] with a set of implementations.

3.3 Posix Emulation Layer

The `libGCJ` library requires a runtime system which provides a set of POSIX compatible system calls. The Pure operating system family doesn't have POSIX as its native API. To solve this problem we introduced new Pure family members. The members are able to translate the POSIX calls into the object-oriented world of Pure. The other, very important function of these members is the ability to redirect POSIX calls to other nodes. This is a realization of the distributed runtime approach described in section 2.

Closer examination of the functionalities required by `libGCJ` reveals that only a very limited number of them need to be provided locally. These are shown in the lower left box of figure 3. All other functions are subject to remote execution. They could either be handled locally if resources are available (e.g. access to a disk) or be forwarded using the system-call proxy.

[4] Functionality as at Autumn 2000

The POSIX call layer consists of a family of different implementations to execute such a call. One family extension supports local execution of a limited set of POSIX calls. A second family extensions supports the remote execution of POSIX calls on a different node. A third extensions allows for the switch between local and remote execution depending on the call arguments.

The implementation of the remote call execution uses are very simple remote procedure call (RPC) protocol. It supports only three data types: Integer, Character and Byte Array. If POSIX calls require structured data as arguments, then it is translated into a byte array on the client and the server has to known which data format the client uses. But in most case no structured data is needed. If the functionality provided by this implementation does not meet the requirements of the application, new family extensions may provide these additional functionality.

3.4 Operating System Layer

The lowest layer of JPure is provided by Pure. Pure is an operating system family for deeply embedded systems developed in our group. It is implemented in C++ and runs on many different processor types ranging from 8 bit (Atmel AVR) to 64 bit (Alpha). The family based design allows a maximal adaption of the operating system to the needs of the application(s) without unnecessary resource consumption. The result is high execution speed paired with low memory footprint.

4. Times and Sizes

To make our measurements of JPure comparable we had to choose a platform for which other Java implementations are easily available. A Linux system with a Pentium 166 CPU was therefor our target systems although it is not a real embedded target.

Function	*Size*	*Data part*	*Percentage*
libGCJ + App.	294k	94k	94%
Pure Core	6k	0.5k	2%
Pure Serial Driver	4k	0k	1%
Pure IOLib	5k	0.5k	2%
Remote Posix Calls	2k	0.5k	1%

Figure 4: Memory usage of HelloWorld on JPure

From the figures given in Table 4 it is easy to understand where the problems are. The runtime support for the `libGCJ` based on JPure takes only 6 percent of the used memory, the rest is dedicated to the `libGCJ` itself. The overall memory requirement looks not too bad compared to the numbers shown in figure 1, but unfortunately a third of it is valuable RAM. Most of the RAM is used by global `libGCJ` objects. Further reduction of this number therefore requires removal of unused and unnecessary global objects. Ways to achieve this were discussed in section 3.2. The server on the remote nodes needs about 388k on a Linux system (statically linked)

Java Environment	*Loop*	*Logic*	*Overall*
IBM JVM	10418	3939	5165
GCJ/Linux 2.96	5175	12219	3109
GCJ/Linux 2.95	5303	11111	2789
Jpure 2.95	2586	10818	2167
Sun JVM 1.3	2530	1712	1825
Sun JVM 1.2	1332	2250	1178

Figure 5: Selected Results for EmbeddedCaffeineMark Runs

The benchmark used to measure the Java performance was the EmbeddedCaffeineMark from Pendragon [9]. The results are quite interesting. The IBM JVM was the clear winner with a performance which is several times higher then any other Java version. From the individual test score for the loop test of the benchmark shown in table 5 it is obvious the IBM JVM includes a JIT which has special optimizations for this kind of benchmark. The second best result is the GCJ/Linux 2.96. As Jpure is currently based on GCJ version 2.95 we expected after a a port of our modifications to GCJ 2.96 similar results from JPure. For Version 2.95 JPure and GCJ/Linux have similar scores for all but one test. At the present time we are not able to explain the huge difference in the loop score. Both platforms used the same object file containing the loop test. We will do further investigations and hope to be able to solve or at least explain the result.

5. Current State and Future Development

The architecture sketched above is currently being implemented as part of a project together with a car manufacturer. The test case is a car which provides many telematic and multimedia services to the car users. The current setup consists of 5 PC which shall be replaced by smaller configurations based on PPC8xx and C16x. The host nodes will be running Linux (later WindowsNT will be possible too), while the other nodes are running JPure.

A prototypical implementation with RS232 based messaging running on x86 PC is complete. The JPure system itself is already running on other platforms but the communication drivers for CAN and ethernet are not yet fully functional so only selfcontained operation of JPure is possible on these platform right now.

Future extensions will include the partial distributed realization of standard Java libraries such as the Abstract Windowing Toolkit. An important extension will be the dynamic loading of class as native code into a running JPure system. The Java-to-Native compilation could be done on the JPure machine or if not enough resources are available on the local node, the compilation is done on a remote node and only the loader is local.

The emerging Java processors will be a very interesting target for our approach. Although these processors are able to execute bytecode natively with very high performance, they still need Java runtime software. If part of this software are moved to a remote node, very cheap and fast Java nodes with only small amounts of RAM are possible. This could make the use of Java feasible even in areas where today highly optimized assembly and C code are used.

References

[1] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, May 1999.

[2] IBM Corporation. *cJVM: A Cluster-Aware JVM*, 2000. `http://www.haifa.il.ibm. com/projects/systech/cjvm.html`.

[3] Microsoft Corporation. *SR Millennium Projects*, 2000. `http://research.microsoft. com/sn/millennium`.

[4] esmertec AG. *esmertec home page*, 2000. `http://www.esmertec.ch`.

[5] Tim Wilkinson et al. *The Kaffe virtual machine*, 2000. `http://www.kaffe.org`.

[6] SUN Microsystems. *Java Card Technology*, 2000. `http://java.sun.com/ products/javacard`.

[7] SUN Microsystems. *The K Virtual Machine*, 2000. `http://java.sun.com/ products/kvm`.

[8] D. L. Parnas. Designing Software for Ease of Extension and Contraction.*IEEE Transactions on Software Engineering*, SE-5(2):128-138, 1979.

[9] Pendragon Software. *CaffeineMark 3.0*, 2000. `http://www.pendragon- software. com/`.

[10] Open Source. *GCJ - Home Page*, 2000. `http://sourceware.cygnus.com/java`.

[11] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43:43-50, May 2000.