AJACS : Applying Java™ to Automotive Control Systems

Jérôme Charousset, Antonio Kung (Trialog)
Jerome.charousset@trialog.com Antonio.kung@trialog.com
25 rue du Général Foy, F-75008 Paris Tel: +33 1 44 70 61 00 Fax : +33 1 42 94 80 64
Thilo Gaul (University of Karlsruhe)
Gaul@ipd.info.uni-karlsruhe.de
Postfach 6890, D-76131 Karlsruhe Tel : +49 (721) 608 - 60 88 Fax : +49 (721) 69 14 62

Keywords : Java, Automotive Control, Real-Time, OSEK/VDX.

Abstract

This paper presents AJACS (Applying Java to Automotive Control Systems), a two-year initiative to specify, develop and demonstrate an open technology allowing the use of Java in deeply embedded automotive systems such as engine control systems (http://www.ajacs.org). This initiative is jointly carried out within the High Integrity Profile working group of the J consortium with the objective to define a J consortium specification.

It first discusses the trends in the automotive industry to go for global systems design : vehicle-level functions (e.g. cruise control) are viewed as a collection of collaborating sub-functions (e.g. measure speed, read brake pedal…) with very precise requirements. The basic aim is to reduce costs by being able to reuse the same sub-function for the implementation of several vehicle-level functions, and thereby to avoid hardware or software redundancies.

A brief introduction on the OSEK/VDX initiative led by car manufacturers and OEMs to support this trend is then presented, followed by a discussion on how the Java technology could usefully complement it. Particular constraints for industrial acceptance are listed. One of them is adaptability to deeply-embedded configurations that are used in the industry today (e.g. system footprints with 256Kb ROM, 16Kb RAM). Technical issues are also presented.

The paper then sketches the overall approach for the definition of a Java-based programming environment suitable to automotive control systems that is being defined in the High Integrity Profile for Automotive control (HIPA) specification work. It finally elaborates on the generation approach that will be used in AJACS.

Automotive Control Systems Today

Today vehicles include an increasing number of electronics systems. It has been estimated by Dataquest that the average semiconductor content of a vehicle will reach $240 by 2001, with consumption of DSPs, microcontrollers and microprocessor reaching $4.9 billion. ***Electronic control units*** or ECUs now play a crucial part in all the functional areas of a vehicle such as infotainment / multimedia (e.g. radio system, road guidance, cellular phone), body control (e.g. instrument panel, window lift, automatic door lock), or vehicle control (e.g. engine management, transmission, brakes).

This was made possible through the advent of networking technology such as CAN. Typical vehicles consist of several interconnected networks that more or less reflect distinct functional areas. For instance, the Volvo S80 includes «18 ECUs connected via six networks: a low-speed body electronics CAN bus (125kbit/s), a high-speed powertrain CAN bus (250kbit/s) and four other networks».

ECUs are typically designed and developed by OEMs according to requirements set up by car manufacturers. Until recently, each ECU was dedicated to a single user-function (e.g. climate control) and OEMs had entire freedom for the implementation (hardware and software). Car manufacturers now try to introduce more flexible development processes that support the breakdown of the user function into fine-grain sub-functions with very precise requirements, in particular at the software level. Two examples of such breakdown (“Display speed” and “Cruise Control” user functions) are presented in Figure 1. At some point in the development process, the resulting sub-functions will be mapped to the hardware subsystems, the ECUs, that form the physical architecture (see Figure 2). The intent of the car manufacturer is that the functional

specification and the physical architecture design tasks can be carried on in parallel with definite synchronization points (the tentative physical allocations of sub-functions).

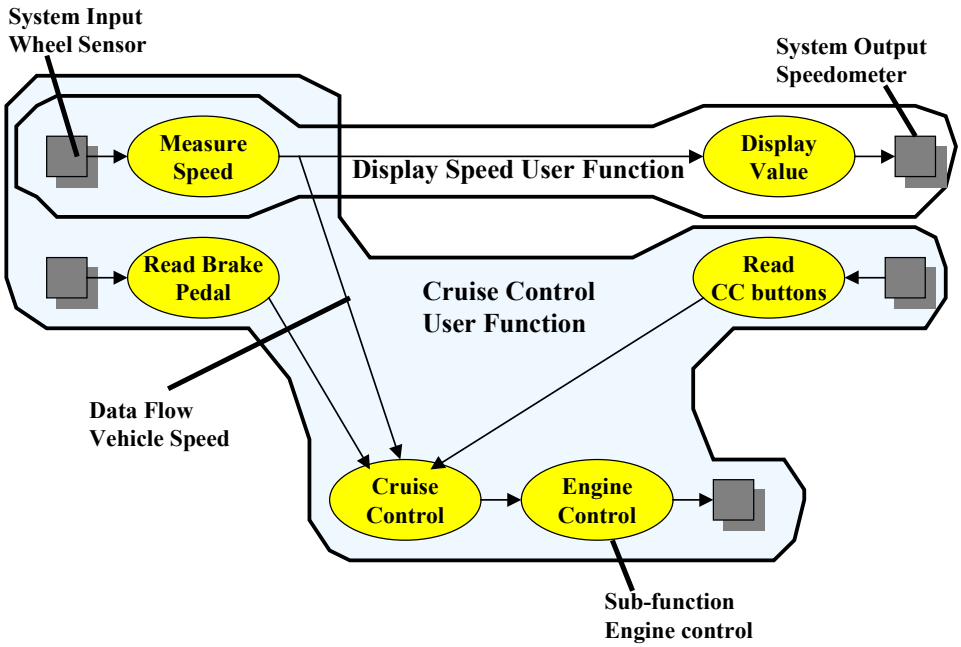

Figure 1 - Example of User Functions

The car manufacturers expect many benefits of such a development process : having more control on the specification of the user functions, protecting trade secrets by keeping some crucial sub-functions under their only control, reducing development duration and reducing the resulting bill of materials for the vehicle electronics.

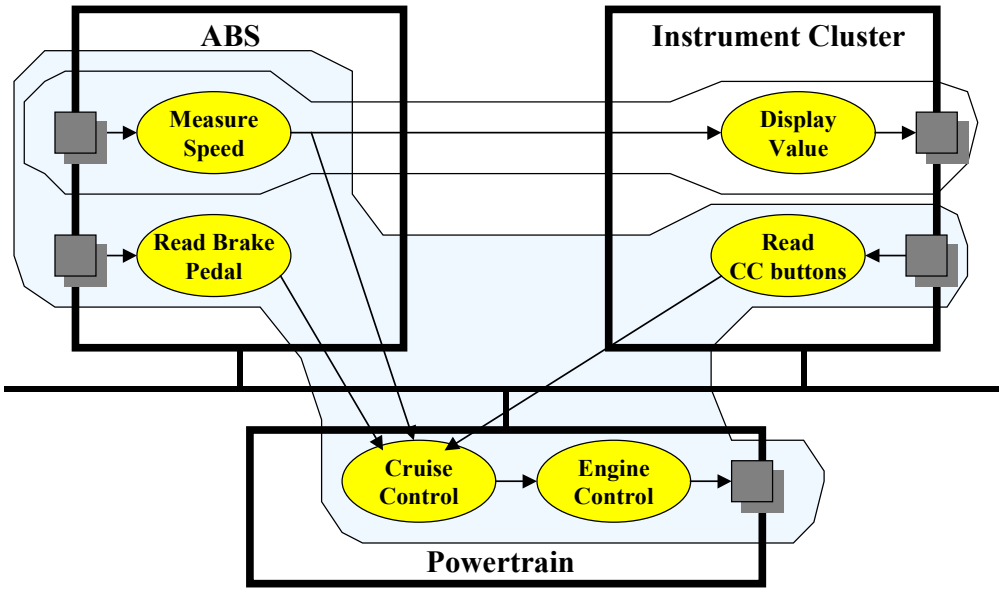

Figure 2 - Example of Physical Allocation

Resulting from this trend, the relationship between car manufacturers and OEMs is thus starting to change. OEMs may be subcontracted for the development of software components only (e.g. an entire user function, or all the sub-functions sitting in the engine control ECU). Or OEMs may be subcontracted for the provision of an “incomplete ECU”, that is an hardware and software platform

on which sub-functions can be further added by the car manufacturer or some third-party (e.g. the engine control ECU without the sub-functions related to automatic gear shift).

From a different perspective, the OEMs also seek for a more flexible development process. Many of them try to set up generic hardware and/or software platforms that would fulfill all the general requirements for a particular functional domain (e.g. climate control) ; the intent is that a large range of products can be easily and cheaply derived from these generic platforms by a quick tailoring to the specific requirements of each individual project. At the software level, such generic platforms usually rely on a definite set of fine-grained components that can be interconnected in various ways, depending on the actual requirements to be fulfilled.

To address the trend towards advance electronic architectures, the whole industry has identified the need to provide guidance on the transition towards advanced electronic architectures. It has been pushing from the start for the definition of open systems, with the definition of corresponding interfaces (APIs) as in the OSEK/VDX (1) initiative presented below. It has also been pushing for the use of advanced software engineering methods (such as OMT, UML), and approaches promoting software reuse (such as object-oriented programming).

The OSEK/VDX Initiative

OSEK, an abbreviation for the German term "Offene Systeme und deren Schnittsellen für die Elektronik in Kraftfahrzeug"[1], is a joint project started in 1993 by the German automotive industry. Initial project partners were BMW, Bosch, Daimler-Benz, Opel, Siemens, Volkswagen and the IIIT of the University of Karlsruhe as co-ordinator. French car manufacturers Peugeot and Renault joined OSEK in 1994 introducing their VDX-approach ("Vehicle Distributed Executive") which was a similar project within the French automotive industry. The first results of the harmonization effort were presented by the ***OSEK/VDX*** group in 1995.

An ISO submission process started in early 2000 for the latest release of the specifications. As of today, OSEK/VDX is probably the most successful undertaking concerning standardization in the embedded software industry: there are more than 10 different providers of OSEK/VDX components and many more proprietary implementations from car manufacturers and OEMs.

The scope of the OSEK/VDX specifications includes :

- ***OSEK/OS***, a specification of behavior and APIs for a real-time operating system that makes provision for the execution and synchronization of application tasks and interrupt routines.
- ***OSEK/COM***, a specification of APIs and network protocols that supports exchange of data messages between software components, either through local communication (i.e. the transmitter and the receiver are located on the same ECU) or through a network.
- ***OSEK/NM***, a specification of APIs and protocols that serve as a basis for network wide negotiated management functions. Basically, it supports a distributed monitoring scheme for the ECUs, thus allowing a particular sub-function to be informed about the current availability of the other sub-functions it relies on.
- ***OSEK/OIL***, a language to describe the configuration of a particular software component or of a complete application. This configuration includes for instance the definition of the operating system tasks, as well as the description of the messages to be received and/or transmitted.

OSEK/VDX provides a standard architecture for distributed control units in vehicles. It meets the two stringent automotive requirements we have previously mentioned: real-time support and small footprints. This is reflected in OSEK/VDX characteristics :

[1] English translation is "Open Systems and the Corresponding Interfaces for Automotive Electronics".

- It is a static system. All entities are known and declared in advance. Furthermore, all data structures within OSEK/VDX are defined and initialized statically using the OIL language. For instance a task descriptor contains information such as a starting address or priority which are static. Such information is stored in ROM and not in RAM. In order to bring some flexibility, the notion of mode management is also defined. Applications may switch from one mode to another. This switch involves the initialization of the whole system to another set of predefined objects.
- Many OSEK/VDX-based applications are interrupt intensive applications. As a matter of fact, some of them could be entirely interrupt-oriented. This is why OSEK has defined the notion of basic tasks or entities which can be considered as logical interrupts.
- Many applications need to react to cyclic events, although not periodic. This is the case of engine control. OSEK provides services for associating events with a flexible notion of counters, which could be a timer counter or any type of counter (e.g. an event from a sensor).
- Because of the wide range of applications to be supported, the technology offers a number of options: basic tasks (logical interrupt routines) or extended tasks (standard tasks with waiting states), preemption or not, multiple activation or not of the same task.
- A common mutual exclusion service with PCP (Priority Ceiling Protocol) is supported for basic tasks and extended tasks.
- Extended tasks may wait for events. Events are thus associated with configurations of extended tasks.

Java

Three factors have contributed to the immense success of Java in the industry : the programming language itself, the virtual machine capability, and of course the huge collection of freely available libraries.

First of all, the Java language syntax is familiar and easy to grasp, especially for programmers with a C/C++ background. It provides clean support for object-orientation. Complex and/or error-prone language constructs (e.g. multiple inheritance, pointer arithmetic) are banned. It includes built-in support for powerful concepts like multitasking or automatic garbage collection, thus relieving applications of error-prone programming aspects. Robustness is ensured through strong type checking at compilation time, as well as built-in support for runtime error checking (e.g. arithmetic overflow, division by zero, incorrect array index). Last but not least, attention has been paid to prevent any unspecified or implementation dependent behavior.

The virtual machine capability means that compiled Java programs and modules can be directly executed by any implementation of the ***Java Virtual Machine*** (JVM) specification. The traditional cross-compilation step is no longer required. This feature, also called “write once, run anywhere” (WORA), further opens the way to dynamic downloading and linking of software components.

The AJACS Initiative

Because of the success of Java in the industry, and the increasing interest of the real-time community to use it, it was felt that Java technology could also be used in automotive applications and complement the OSEK/VDX technology :

- Through its object-orientation, Java makes it easy to design software component with strictly defined interfaces, even at the source code level. Methodologies like OMT and UML can be adopted with little effort.

- The Java language is designed to be platform independent. Thus hardware independence and portability requirements are largely fulfilled.
- Easiness and robustness attributes contribute to high programmer productivity and low defect rates. This will help the development team focus on high-level activities like component-based design.

To this end, the AJACS initiative[2] was started in February 2000 with the following objectives:

- Define an open technology which relies on existing standards of the automotive industry, such as OSEK/VDX.
- Retain the benefits expected from object-oriented language programming in terms of software structuring, reusability, dependability in particular retain the portability, and robustness attributes associated with Java.
- Address technical issues created by drawbacks of the Java language in terms of real-time and determinism support for embedded systems with high integrity constraints. In the case of automotive control, this means supporting the same kind of real-time constraints which non-Java based ECUs currently handle and targetting the type of memory footprints that are acceptable in the automotive industry (e.g. 256 Kbytes ROM, 16 Kbytes RAM).

Overview of Technical Issues

AJACS has to focus on a number of technical issues. First of all, Java is a dynamic execution platform while OSEK/VDX is a static execution platform. Automotive control systems, like many embedded systems, are static or closed systems. Programming resources are predetermined and allocated only one time. The list of tasks, the list of drivers, the usage of memory is fixed and will not change during the life of the system. Identification of entities is typically predetermined (e.g. task 3). There are at least two reasons for using static systems in the industry: it is easier to have deterministic systems, and it helps to meet small footprint constraints.

To see why static orientation somewhat conflicts with the main features of the Java language, consider for instance multithreading and memory management:

- Threads are Java objects which are created dynamically. A Java thread is identified through a dynamically created reference. In an underlying static system, this reference must be associated somehow to an existing static identifier (e.g. task 3). This has two consequences. First Java applications must only create predetermined threads at initialization time. Second a mechanism must be provided to ensure the association between the Java entity and the underlying static entity.
- Memory management in Java is fully dynamic. Objects which are no longer referenced are automatically reclaimed. Since, in an underlying static system, entities are predefined, this generally means that the corresponding Java object will always be referenced. Thus garbage collection of such objects is not necessary. However, the language itself contains classes which have no links with the underlying system. Java exceptions and Java strings are examples of such classes. Java programmers may also introduce "collectable" objects, which - unless controlled by the programmer - must be released automatically. Such classes are often used in such a way that significant dynamic memory allocation is needed, implying the need for dynamic de-allocation mechanisms. In such case, one might wonder whether garbage collection should be included. We do not believe this will be accepted by the automotive industry, even though it is feasible to use real-time garbage collection algorithms. Other approaches have been proposed

[2] AJACS includes Trialog, PSA, Centro Fiat Riserche, Mecel AB, U.Karlsruhe/IPD. AJACS is partially funded by the European Commission.

by the real-time community such as the allocation of objects in the stack, provided some scope rules are followed, or the explicit de-allocation by the programmer.

Adapting Java to static systems, in particular when part of the application is legacy C code, means that an approach must be defined for mapping the underlying static configuration to whatever remains dynamic in Java. In the case of OSEK/VDX, it means in particular that the OIL configuration language must be combined with Java. All OSEK entities are described in OIL, and the OIL-based building tool should be also be able to generate the appropriate configuration structures related to Java.

The second issue is the level of CPU performance. The standard way to execute programs in the Java environment is to interpret the platform independent Java Byte Code (JBC) in a virtual machine (see Figure 3). The JBC is compiled from the original Java sources with a standard Java compiler (JavaC).

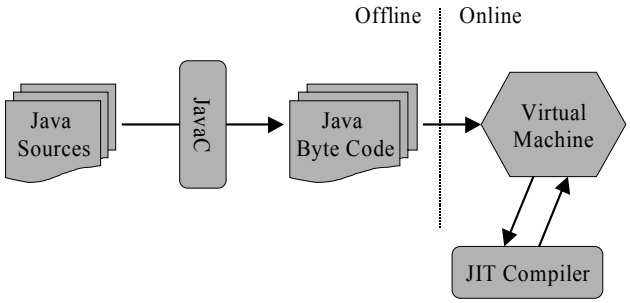

Figure 3 - Standard Java Compilation Process

The virtual machine itself uses a built-in interpreter to run the JBC code. This interpretation mode is obviously the one with the least runtime efficiency, and even on workstations execution times are magnitudes away from traditional ones. A first level of optimisations can take place at runtime by just-in-time (JIT) compilations of atomic parts of the JBC program, i.e. expressions or methods, to native machine code. Usually only local optimisation steps are performed, because the cost of the JIT compilations must be measured as execution times of the program. In addition there are two major problems in using JIT compilers for embedded systems: both the amount of memory needed and the CPU time required for the run-time optimiser are difficult to determine.

As an alternative to the pure interpretation mode, we can get back to a traditional off-line compilation process generating full native binary code for the target system. This allows one to perform global optimisations of the code and specialised code selection for the target architecture which results in a better code quality. The traditional compilation approach has pros and cons. First, execution speed is improved by orders of magnitude. Further, it can help in having more static memory layouts, thus reducing garbage collection overhead. On the other hand, we loose a number of features such as dynamic overloading of classes, or the fact that Java programs have to be specifically recompiled for each target. We no longer have the “compile once, run anywhere” capability.

The traditional compilation approach still allows us to meet software engineering requirements such as the replacement of software modules and the support of inspection and debugging interfaces. Last but not least, we still have portability, i.e. the “write once, compile to many platforms” capability.

Because of bill of material reasons, the virtual machine capability (e.g. dynamic downloading of classes) is not likely to be the mainstream for vehicle control and body control.

The HIPA Specification

The above issues are being addressed within the J consortium (2), in particular in the ***High Integrity Profile*** (HIP) working group which tackles safety critical, high integrity and fault-tolerant systems as well as distributed systems. This specification subsets Java in a suitable version where all entities are statically defined and where no garbage collection is needed. The AJACS consortium will specialize the resulting HIP specification to take into account automotive control aspects and in particular the support of OSEK/VDX. We now describe the resulting specification or HIPA specification (High Integrity Profile for Automotive Control).

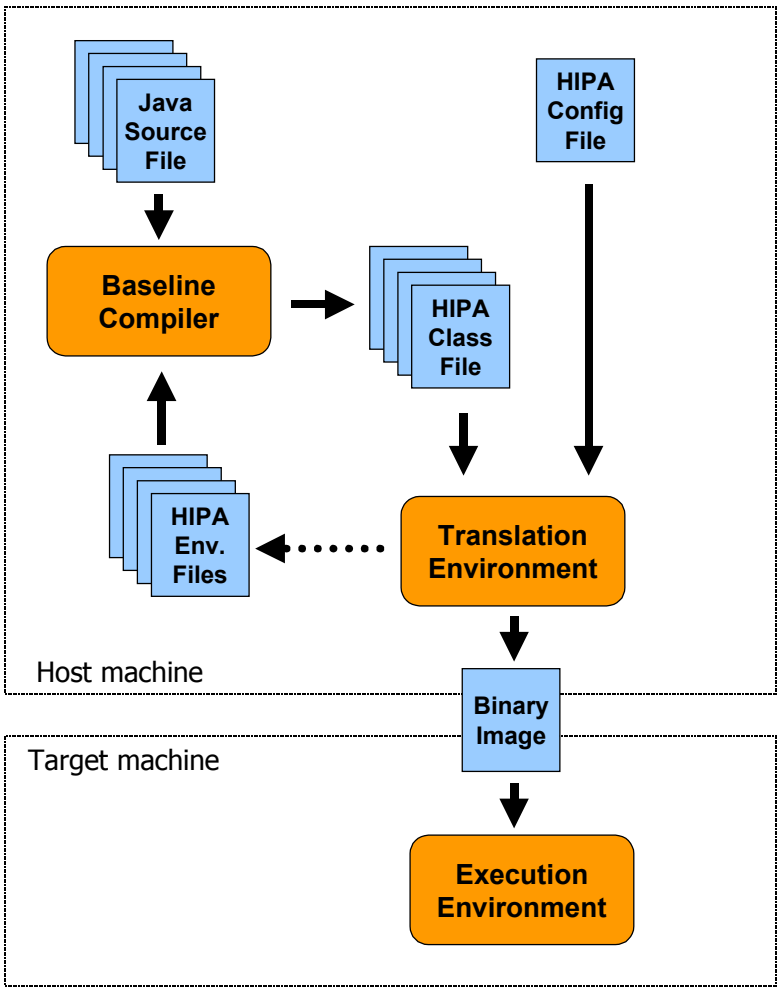

Figure 4 - Overview of a Typical HIPA Development Environment

A HIPA compliant implementation is conceptually divided into two data-processing environments (see Figure 4), the *translation environment*, which comprises the set of development tools needed for the conversion of an HIPA applications into some binary representation for the target machine, and the *execution environment*, which comprises all software components (including libraries) needed for the execution of the binary representation of the program on the target machine.

The following files are involved :

- HIPA Java source files contain the automotive application code
- HIPA configuration files include data about the system static resources. Static entities supported are those of OSEK/VDX such as tasks, resources, events, delays. The HIPA configuration files are the counterpart of the OIL files in C.
- HIPA environment files are generated by the translation environment to reflect the impact on Java programming of configuration aspects (i.e. HIPA configuration files)

- HIPA class files are the result of Java compilation.

The HIPA class files and the HIPA configuration files are finally used to generate a binary image suitable to the underlying execution environment. This binary image can be the result of mixing C and Java code. This is because a HIPA execution environment supports all OSEK/VDX entities. The following APIs are defined :

- A configuration API. This API provides the glue between Java code and the configuration vision of the underlying OSEK/VDX kernel. This means that all OSEK/VDX entities identifications are declared through this API. For instance, if the OSEK/VDX entities include a task called “Foo”, the translation tool will generate a Java identification called “Foo” which can be used by the Java programmer to bind a Java task to its underlying OSEK/VDX task.
- A task management API corresponding to OSEK/VDX tasks semantics.
- An interrupt management API corresponding to OSEK/VDX interrupts semantics.
- An event management API corresponding to OSEK/VDX events semantics.
- A synchronisation API corresponding to OSEK/VDX resource semantics based on the use of Java synchronised blocks or methods.

In terms of conformance, HIPA implementations follow the OSEK/VDX conformance approach :

- Complete run-time checking capability is defined, with the possibility for the programmer to explicitely switch off certain checks.
- The four OSEK/VDX classes of conformity are retained : BCC1, BCC2, ECC1, ECC2. The OSEK/VDX (1) specification explains in details these classes. They influence aspects such as event management support, the number of tasks per priority or task activation semantics.

AJACS Generation Approach

Native Code Approach

The main benefits of compiling Java to native code are improved code quality and better memory layout. Both result in significantly improved execution efficiency without additional effort on the target machine since the whole process works off-line. The native virtual machine no longer executes the generated code, but is more likely to be a standard runtime environment extended with Java functionality. This will be the approach used in AJACS. Figure 5 illustrates the native code compilation process used.

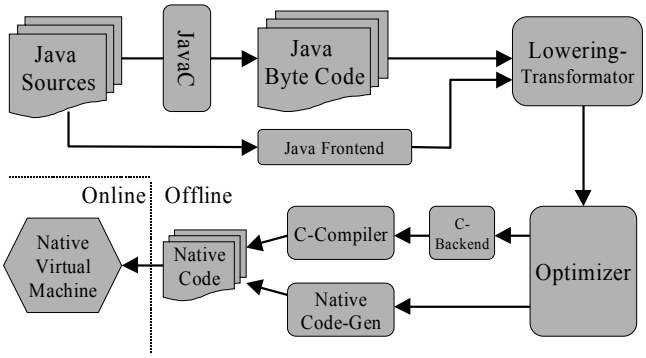

Figure 5 - Java Native Compilation Process

The lowering transformator gets input from a specialised Java front-end, or Java Byte Code (JBC), generated with a usual Java compiler. We also adapted the widely used IBM Jikes compiler (7) to

directly produce input for the AJACS compiler framework. The transformator lowers the high-level Java constructs, which are also still present in JBC, to a low-level SSA (Single Static Assignment) intermediate form. On this representation, general and target dependent optimisations are performed. The AJACS framework offers two possibilities to produce target code from this representation: It produces either low level C or native machine code with an optimising code generator. The latter usually improves code quality, though it is more costly because every target architecture family has to have its own code generator. The C code generation is typically preferred, if a C compiler of high quality is already available.

Optimisation Technology to be used in AJACS

The object-oriented programming paradigm offers a lot of valuable features for the designer and developer, but makes it even harder for the compiler to get rid of inefficiencies when compiling to the machine.

Java programs, like other object-oriented languages, usually have a number of runtime efficiency relevant drawbacks: indirections in data accesses and calls are not well handled, memory layout is overly fragmented, and many small basic blocks are used. It is known (2) that compared to usual imperative programs, object oriented programs have 5 times more procedure/method calls, 60% more memory accesses, and 20 times more indirect calls due to polymorphism.

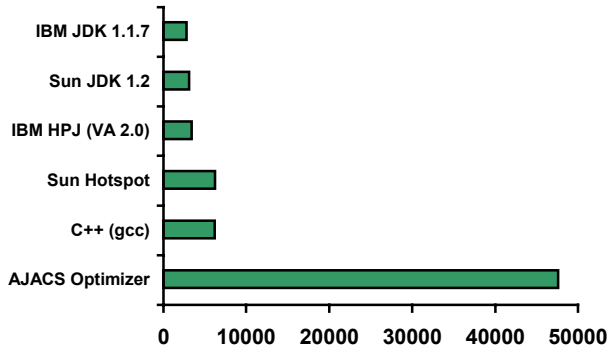

Figure 6 - Performance of OO Native Code Systems

An additional severe problem is that standard optimisation techniques cannot be applied in the presence of these program features. To eliminate the resulting inefficiencies, a number of optimising transformations have to be provided. They can be summarised mainly in two categories: the conversion of heap objects to local states, and aggressive code placement and in-lining. An adaptive grained value flow analysis allows to eliminate typical indirections, polymophism, and data accesses. It is possible to apply expensive optimisations by using an iterative analysis technique with adaptive accuracy. The more object-oriented program features can optimised away, the more standard optimisations can be applied to the resulting code.

The main optimisations are performed through explicit dependency graphs (EDG), a SSA form where optimisations correspond to graph rewritings on this representation. We have already carried out initial experiments. The results based on standard object-oriented benchmarks are very promising (see Figure 6).

Back-End Generator Technology Used in AJACS

Writing optimising compiler back-Ends by hand is no longer needed because well established generator technologies and tools are available for code generation.

AJACS will use a code-generator generator technology that uses *bottom-up-rewrite/bottom-up-pattern-match* techniques to implement optimising code selectors from specifications. This

technology simplifies the task of the code generator developer because it allows him to concentrate on local aspects, while still fulfilling global optimisation criteria. Further, he neither has to be bothered with traversal mechanism nor with register allocation implementations. Another advantage of the technology is that formal verification of code selection correctness is possible.

This technology is provided by the Back-End Generator (BEG) tool from Uni Karlsruhe (3). It has already been applied in several projects and assures easy retargeting to different embedded architectures (6). The technology is based on the use of one unique integrated processor/code-select description, a covering mechanism that assures optimisation in both code size or execution time, the provision of several optimised register allocators built-in, the support of consistency and type checks on specification, and finally the generation of an instruction scheduler from specification.

Conclusion

The HIPA specification will be finalized in 2001. The AJACS consortium will carry out an implementation and validate the whole approach through two applications. The objective is to have a global CPU performance and memory footprint which is compatible with today's C-based application.

References

(1) OSEK-VDX: www.osek-vdx.org

(2) Calder, B., Grundwald, D. and Zorn, B.: *Quantifying behavioral differences between C and C++ programs*. Technical report, University of Colorado, 1994.

(3) Emmelmann, H. and Schröer, F.-W. and Landwehr, R.: *BEG - A Generator for Efficient Back-Ends*. Proceedings of the Sigplan '89 Conference on Programming Language Design and Implementation, 1989.

(4) JSR-000001: www.rtj.org

(5) J consortium : www.j-consortium.org

(6) Gaul, T. and Schumacher, G.: *Advanced Generator Techniques for Embedded Compilers*, Proceedings of the EMMSEC'99 Conference, Stockholm, Sweden

(7) IBM Java Compiler *Jikes* : www.jikes.org
