A New Approach for Java in Embedded Networks

Wolfgang Kastner and Christopher Krügel
Technische Universität Wien
Institut für Rechnergestützte Automation
Treitlstraße 1, A-1040 Wien, Austria
Email: k@auto.tuwien.ac.at

Abstract

This article starts with a short introduction of modern control networks and surveys severe problems, one faces when porting a Java Virtual Machine (JVM) to embedded components. Some possible solutions for those problems are presented, special purpose VMs, just-in-time (JIT) compilers or the use of compiled byte-code are evaluated. Finally, we assess a potential solution regarding a new approach for a Java embedded network that is currently under research and implemented in a future project.

1 Introduction

Looking closer on today's distributed control systems, it is easy to identify a common hierarchy of networks (Figure 1). On top we find a network of powerful *workstations*. The workstations can be divided into some, that run mighty applications for computer aided design (CAD), engineering (CAE), planning (CAP), testing (CAT), quality control (CAQ), recycling (CAR) and computer aided office (CAO), as well as others, that are more or less directly connected to the technical process. Usually, this upper network is a local area network (LAN). By means of gateways, the workstations have access to even higher networks, for instance a metropolitan area network (MAN) or the Internet.

Regarding workstations for the computer aided manufacturing (CAM), we distinguish between some fulfilling their tasks stand-alone and others delegating their job to further *control systems*. Depending on the underlying application, the control systems can either be reliable and fast microprocessor systems or established PLCs. In former days, sensors and actuators of the field were directly connected to these control systems. For large automation systems, this parallel wiring resulted in thick harnesses with lots of problems showing up when the cabling broke and had to be maintained. Fortunately, things have changed nowadays, where field devices can be connected to a common (serial) control network, better known under the term *fieldbus* system.

A distributed system based on such a fieldbus technology has several advantages compared to a so-called conventional centralized control system [4]. In our opinion the most important improvements are:

- *Setup* – Since there is just one single bus linking all automation components, there is a significant saving of cables. In accordance to this, the setup of such a system is easier and faster.
- *Maintenance* – As there is usually one (configuration) tool for all devices, it is easy to configure, parameterize, maintain, monitor and visualize the distributed system.

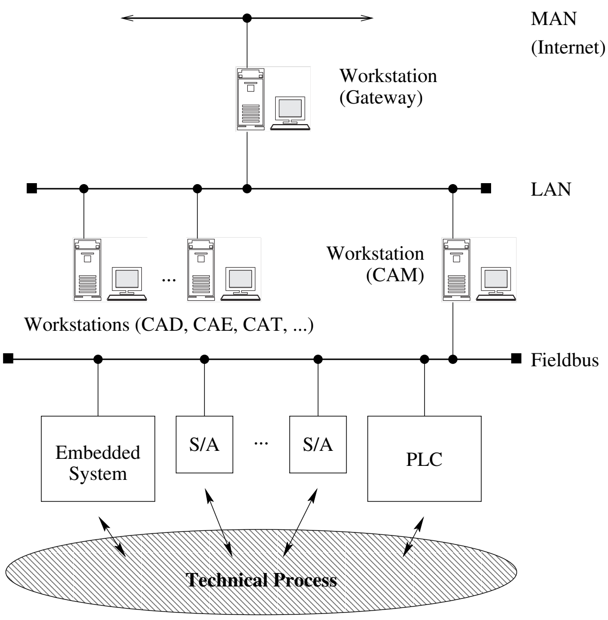

Figure 1. Distributed Automation Systems

According to the requirements of different automation installations, distinct fieldbus systems have been designed and implemented during the past years. Up to now leading manufactures of automation technology offer fieldbus interfaces for their devices.

Whereas in former days it was enough to setup the control system and fieldbus devices at the lower layer and then perhaps (re-)configure parts of the system, control or just monitor the ongoing technical process from a workstation at the top layer, today's users demand for real *(embedded) networking*, that is for instance controlling fieldbus systems via Internet, where typical interfaces are integrated as Java applets into common web browsers featuring a Java Virtual Machine (JVM).

A couple of different languages compete for their use in embedded, distributed systems. While C or C++ was the computer programming language of choice for the last two decades when implementing applications for small devices with scarce resources, things changed dramatically with the advert of *Java*. Java offers a number of advantages over C/C++, where only the most important are enumerated below.

- *Portability* – When using Java, an intermediate bytecode is produced, that can be run on any system which supports a compliant JVM without modification.
- *Safety* – As Java deallocates memory automatically, the well-known memory leaks in C/C++ cannot occur.
- *Library Support* – Java offers a lots of API functions, especially for easy network programming.

These benefits allow software projects to be completed faster and to run stable with far less debugging effort, causing more and more applications realized in Java. Unfortunately, the increase in convenience and safety comes with the price of a higher demand for resources (computational and memory).

This is no problem, when a standard PC, that runs a fully-fledged JVM, is used as a platform. Nevertheless, these demands become much more restrictive, when we think about networking small devices by means of middleware products based upon Java (e.g. Jini [2]). When a simple actuator or sensor should act as a Jini service, the scarce resources become a problem. It seems evident, that it is not useful to provide each component with enough computational power and memory to run Java on its own. Three different solutions are conceivable, which reduce the flexibility by lowering the demand for resources:

1. Using a gateway with a complete JVM, proxies act on behalf of simple devices. This variant needs a shared virtual machine, that runs Java programs and performs communication on behalf of its clients.
2. Implementing software in every device, that is just able to adapt to the middleware.
3. Implementing a complete JVM with a reduced need of resources by utilizing techniques like dynamic compilation, special purpose Java processors or automatic library down-loading. These techniques mainly combat Java's deficiencies and issues especially severe in real-time applications.

In a former project, targeting the integration of Jini in a home and building automation system (cf. [8], [9]), we used the first variant, i.e., a gateway solution with a shared virtual machine running on a workstation (PC), that hosts service proxies on behalf of their simple clients. The shared JVM provided a platform for software modules, that corresponded to a set of fieldbus devices and offered the devices' functionality. These software modules acted as proxies for fieldbus devices in the Jini federation, which they had to represent. The proxy-oriented system itself was designed for larger installations, where a dedicated workstation (PC) was used to realize the additional tasks and served as a Jini platform. When considering smaller installations (e.g. private homes), the overhead of a dedicated PC for a small fieldbus seems too much. It is simply too expensive to purchase an extra PC for every simple device facility. Here, our work offers a starting-point for further research: setting up an embedded Jini Controller, that allows to keep the needed Jini functionality at every device, simply by having every fieldbus gadget equipped with a JVM, that hosts the Jini classes itself. Once done, there would be no need for special proxy processes, that are hosted at remote locations and act on behalf of the devices. Instead of those, every device itself, respectively the user programs in the fieldbus installations, can directly interact with the Jini (and network) layer.

This paper may be seen as the starting point for our efforts to implement a Java execution environment for embedded systems, that is capable of hosting a Jini layer. We present an approach to build an efficient run-time system, that can cope with limited resources. The rest of this paper is structured as follows. First, we present an overview of drawbacks, when Java is used in embedded systems. Second, a number of state-of-the-art methods used to circumvent those disadvantages are shown. Third, we present a new approach to combat Java's deficits by combining compilation and interpretation into a hybrid execution environment.

2 Java Drawbacks

Most of Java's shortcomings have their origin in its biggest advantages. They are the price, that has to be paid for the ability of dynamic code downloading, automatic memory management by a garbage collector and portable classfiles by the use of standardized bytecode. We fully agree with [7] about the main weaknesses of using Java in embedded systems, i.e., weak execution performance, large memory footprint, unpredictable execution time and impossibility of direct memory access.

Obviously a well-known drawback is the poor performance of a Java system compared to a fully optimized C implementation, tailored exactly to a particular microcontroller, since the default method for executing bytecode by a JVM is interpreted execution. Whereas C/C++ applications are compiled to a native instruction set and are stored in a single executable file, Java applications are compiled to Java bytecode and are stored in separate class files for loading and later execution. When the Java application is run, the JVM loads the class files and interprets the bytecode as a stream of instructions. In embedded systems, where processing power is substantial weaker when compared to a standard desktop processor, it is often common to hand-optimize critical code parts and squeeze every processor cycle out of the implementation. Of course, that can't be done when interpreting code. However, new *just-in-time* (JIT) compilers help to close the performance gap. Unfortunately, when a JIT compiler is included, the memory needs are more increased, since for instance the code for the compiler itself has to be stored.

Using Java, all standard library classfiles have to be accessible, which for example takes up about 10 megabytes for the latest release 2. In order to cut the needed library size, Sun decided to provide official subsets of the standard Java API, that have been called PersonalJava [17] and EmbeddedJava (Micro Java) [14]. Besides a simple reduction of the API, several methods have been developed to reduce the classfiles size. One is called *ROMizing* and deals with creating an architecture dependent ROM image from standard Java classfiles [11]. Another obvious way of reducing the memory needs is to store non-critical classes on a common server within a network, where classes may be downloaded dynamically and, because they origin from a trusted source, do not need bytecode verification. Other improvements can be achieved by compressing classfiles. It might be possible to store a zipped variant of the classfile and decompress the needed parts on demand. While this approach can cut down the memory usage, the performance penalty may not be neglected and indeterminism is being introduced.

Unpredictable execution time or run-time indeterminism is an important issue in embedded systems, especially for those, that have to meet firm deadlines. Java's execution-time indeterminism mainly results from the automatic *garbage collection* (GC) and Java's thread model. A real-time GC has to free unused memory predictable and fast while still allowing high-priority tasks to run, because otherwise, important deadlines could be missed. Unfortunately, most GCs fail to fulfill any of these requirements. They usually work in batch or so-called *stop-the-world* mode. That means that the collector is only called, when the available memory is exhausted and an allocation fails. As it is very difficult to predict the exact point in time, when the GC gets activated, the whole system becomes unpredictable and unusable for real-time systems. To defeat that problem, some solutions suggest to run the GC as a concurrent task, which leads to the development of (sometimes preemptive) incremental GCs. However, this introduces new difficulties, like the well-known moving pointer problem (the interested reader should check [7] for a detailled problem description) and heap fragmentation. Hence, an optimal solution for an embedded system would be a concurrent, compacting, precise GC, offering best predictability.

As can already be seen from our last considerations, it is very difficult to find a proper balance between run-time performance and determinism on one side and storage and run-time memory needs on the other side. The introduction of a highly optimizing JIT-compiler speeds up the application and sophisticated GCs improve predictability, but they also use lots of additional memory. Hence, it is the task of the embedded system designer to decide, how much memory he can trade to gain the mentioned benefits.

The last disadvantage of Java, that results from the use of object references instead of pointers, is the impossibility of direct memory access. Java does not allow to define variables that hold memory addresses or access memory directly via references. This increases the security of the programming language, as illegal pointer accesses become impossible, but increases the overhead in case of a memory-mapped I/O operation. Many embedded systems handle data transfer between the microcontroller and external devices by mapping the I/O devices' registers into the microcontroller's address space directly. The microcontroller can access peripherals exactly like main memory by simply putting the desired address value on the memory bus. This is supported by conventional embedded systems languages like C by allowing pointer variables to hold arbitrary addresses. In Java, the problem can only be solved by leaving the boundaries, set by the runtime-environment. This can be done by using *Java Native Interface* (JNI) [16], which allows arbitrary machine code functions to be called or by utilizing non-standard language extensions. Unfortunately, the JNI method suffers again from a performance penalty, that is introduced when switching between JVM and native program execution. Using JNI involves problems with the GC, too, since for instance the GC has to be informed, which objects have been passed to native functions (those objects have to be considered in use and may not be reclaimed). Finally, all safety assertions that Java provides are temporary suspended when running a native function (which can lead to unexpected illegal memory references).

3 Java Embedded Solutions

To solve the shortcomings of Java for embedded systems, that have been described in the previous section, one basically has five different choices, we are going to examine in some detail in the following subsections. These solutions mainly try to target the memory usage and performance drawbacks, that have been presented. All solutions suffer from the shortcoming, that they cannot inherently remove the problem of direct memory access or concurrent garbage collection. But as will be shown, each approach offers some benefits.

3.1 Special purpose JVM

The obvious solution to minimize the memory needs of a Java run-time environment is to reduce the offered library functionality. When Sun realized, that the library file size will prevent Java from being used in the embedded sector, they invented the Java 2 *MicroEdition* (J2mE) [15]. This edition defines a minimal set of API functions, consisting of the core language's data types, threads, sockets and data streams, that the newly implemented JVM, called *Kilobyte Virtual Machine* (KVM) [13] has to support. For each class of devices, a new configuration can be defined, that exactly fits its demands.

Although, the KVM is designed to use as less memory as possible without significantly reducing the performance, portability is also an issue. Non-portable code has been reduced and concentrated into a single file. The non-portable code pieces mainly contain calls to obtain host system calls (like I/O calls to the local operating system). These invocations usually use JNI to perform their task, but have been replaced by functions inside the KVM to reduce the space needs. As a result, the support for native functions is rather limited and graphics-functions are nearly missing. Multithreading and the GC have been implemented in a completely platform-independent fashion.

In addition to Sun's approach of reducing the supported library functionality with a Java MicroEdition and thereby limit the number of resources, a couple of vendors try to implement a space efficient version of a standard JVM (e.g. *Jeode* [6], *ChaiVM* [5], *PERC* [11]). Most of these implementations run on different real-time operating systems as well as processor platforms.

3.2 Just-in-Time Compilation

JITs have been one of the first answers to get rid of Java's reputation of being a slow language. Bytecode interpretation doesn't only suffer from the problem that reoccuring code fragments have to be interpreted over and over again, but also from the fact that no global optimizations are possible. A JIT compiles methods *on-the-fly* when they are called the first time and stores the resulting machine code in memory for further use. This has three advantages. First, as a whole method is compiled at once, a JIT can perform local optimizations like a regular compiler. Second, as the quality and performance of the produced code is comparable to C/C++ compilers using highest optimization levels, the run-time of the compiled methods nearly reaches code written in those traditional embedded languages. Third, only methods, that are really executed, are considered by a JIT for compilation, thus saving the need of process functions which will never be invoked.

The performance gains come at a price, which is mainly paid by increased memory needs. Whereas the JIT compiler itself, which can be stored in ROM, needs only a few 100 KBytes, the stored machine instructions for compiled procedures also need significant RAM storage. Unfortunately, JIT compilers perform weak, when applications only have a short run-time and use most code parts only once. A further drawback for embedded systems is the introduced indeterminism of a JIT. Compiling an emergency routine with a short, hard deadline before executing might not be the best idea. Hence, in our opinion JIT compilers are perfectly suitable for desktop applications (e.g. Cacao [10]), where they can easily match C/C++ programs. However, the reduced environment of an embedded system most likely cannot fulfill their memory needs.

3.3 Ahead-of-Time Compilation

In contrast to a JIT compiler, that translates single methods before they are executed the first time, an *ahead-of-time* (AoT) compiler acts like a normal cross-compiler. Such a cross-compiler runs on the development host and accepts Java source or bytecode, which is then used to produce an executable for the target machine (the embedded system). Such a variant offers the advantage, that no virtual machine or additional compiler is needed at the target machine and code, comparable to C/C++ applications, can be created. An AoT compiler is very well suitable to combat the inefficiency and code size drawbacks of Java, especially when it is not necessary to keep the whole standard library at the embedded device. To circumvent the need for a GC, several compiler builders have changed Java's memory management and included a keyword similar to C++'s `delete` to remove the GC.

However, using an AoT compiler violates Sun's credo of 'write once, run everywhere'. It is not possible to download and dynamically execute new code from the network, as no interpreter or compiler is available and the library is missing. In our opinion the inefficiencies inherent in 'target independent' bytecode conflict with many basic embedded design principles. For example, embedded software is typically designed to interact with and manage a particular piece of hardware, often within stringent performance requirements and memory constraints. In this context, the costs associated with running target independent code on a Java VM collide with basic design and functionality goals. When no new code should be downloaded or the system is used in *stand-alone* applications, keeping the whole library around at every system would be a waste of resources. These facts indicate, that AoT compiler might offer an optimal solution for small and stand-alone devices, where Java is simply a new and very safe language to develop programs and possibly replace C/C++ applications. On the other hand, AoT solutions will not perform too well in embedded networked environments, where different services are connected and code downloading is very common. When a service user cannot run the code of the service provider's object, embedded networked advantages vanish and the whole system acts too static.

3.4 Dynamic Compilation

Similar to a JIT, dynamic compilation improves performance by *on-the-fly* compilation of classfiles, but unless a JIT, it only uses a limited amount of memory, which makes them more suitable for embedded systems. Dynamic compilers are much more selective of the bytecode they compile and can adopt to available system memory. When there is plenty of RAM available, more methods may be compiled; when system storage drops low, it can still rely on interpretation. During interpreted execution the JVM machine will monitor the application and determine where the execution bottlenecks reside. It will then invoke the adaptive dynamic compiler to compile the segment of bytecode, which is executing repeatedly. The resulting native instructions are stored in memory only for fast access. The JVM may then find and invoke compilation for the next most frequently executed code segment. This process is repeated until all available code buffers have been used. Because the dynamic compiler is adaptable, as it encounters a new code segment that is executing at greater frequency than some previously compiled and stored code, it will compile the new Java bytecode and store it in a code buffer containing less frequently used code. The user may configure the amount of system memory used for compilation, as well as the size and number of code buffers used for storing the compiled native instructions.

Dynamic compilation performs worst, when applications are very CPU-bound, don't contain small, inlineable functions and have their run-time equally distributed over the whole code. In such a case, every fragment will eventually be compiled and so a JIT, that optimizes every function immediately, will perform better. Since dynamic compilation is a rather new topic in the Java community, several issues are still open to be researched. In our opinion, a dynamic compiler cannot solve the problems of indeterminism or direct memory access, but it is an alternative worth considering for embedded systems, because it can produce fast code within given memory limitations.

3.5 Java Processors

Another alternative, the Java processor, was recognized almost immediately after Java was announced. There's no reason that the *processor architecture*, designed by Java's creators could not be realized in silicon. Sun and several other IC manufacturers soon set about doing just that. The speed of execution is comparable to that of any processor, executing code that has been compiled for it. Since there is no JVM, there is none of the associated memory overhead. Of course, the bytecode of the class libraries will still take up space in ROM.

Java is a stack-oriented language, where most operations manipulate operands on a virtual stack, so it seems natural to use processor registers to build a stack frame. Sun refers to its initial chip architecture as Java One, which uses *PicoJava* [12] as its core. The Java One architecture is designed in a way, that its native instruction set is the same byte code, generated by the Java byte-code compiler. That renders the JVM unnecessary, but getting the byte code to execute fast, even as a native instruction set, is still a challenging task. PicoJava handles memory-allocation restrictions by adding a byte-code instruction at the hardware level for reading and writing memory. That makes memory-mapped I/O possible, which doesn't improve performance, but makes memory appear contiguous to the operating system and running programs.

An important factor in the embedded system market is the price of the device itself. When a Java processor costs considerably more than a comparable microprocessor with a Java interpreter on-chip, economic chances are dramatically reduced. To make a microprocessor design a commercial success, it's necessary to establish a high-volume production run, so economies of scale pay for the chip design and fab setup. Without high volume, the cost of a special-purpose chip outweighs its benefits. Often, general-purpose chips gain performance advantages over time, that nullify the advantages of special-purpose chips. If general-purpose chips are fast and inexpensive enough, why should anyone use a special solution? Sun tries to overcome that obstacle by offering its PicoJava for about $25, which seems to be a quite competitive offer when compared to a standard microcontroller.

4 Hybrid Java Execution Environment

The previous sections have explained some advantages when using Java as a language for embedded systems, but have also enumerated certain problems originating in its execution model. Most important, embedded systems suffer from stringent resource constraints, in terms of computational power as well as available memory. This section presents a new variant of executing Java class files in such an environment and compares it to the introduced traditional solutions.

The basic idea of our hybrid Java execution environment (and in fact of our future *Hybrid Virtual Machine*, HVM) is the combination of interpretation and compilation techniques. Most of the classfile's bytecode is handled by a Java interpreter, that profiles the run-time behaviour and identifies heavy-used code parts. Those code parts are handed over to a compiler, that emits highly optimized machine code. Execution is then switched from the interpreter to native code to perform faster. In case of exceptions or when control flow leaves those code parts, execution is reset to interpretational mode.

This mechanism offers many benefits compared to traditional variants, as explained below.

- *JVM vs HVM* – When compared to a simple interpreter, *execution speed* is expected to rise dramatically, as heavily used code parts are translated to native instructions and executed on the CPU directly. The compiler can agressively perform optimizations, as exceptions or hard-to-optimize cases can simple be ignored, because the system can always revert to interpretation.

- *JIT vs HVM* – Compared to a JIT compiler, the *memory needs* in our approach can be kept reasonable low. Of course a JIT compiler might perform better than a hybrid system, because no interpretation takes place at all. On the contrary, the compilation process needs much memory, as data structures (e.g. control-flow graphs) have to be built for whole procedures. Many global optimizations run memory intense algorithms, that are focused to reduce execution time but do not care for used memory. Additionally, compiled methods have to be stored for later usage. Things are different for our compiler, which can dynamically adjust to available memory. When resources are dropping low, the number of compiled methods can be reduced; when plenty of memory exists, everything can be compiled. A mixed interpreter/compiler system can utilize run-time information to introduce additional optimizations. This can be used for loop unrolling, when upper bounds are known or constant folding/propagation, when variable values have been assigned. The whole range of dynamic compilation improvements are at hand. A significant part of research work will also be dedicated to identify compiler optimizations, that increase performance but demand only modest resources.

- *AoT vs HVM* – When our execution environment is compared to native code, produced by AoT compilers, it is clear, that such code will perform better most of the time. As no interpretational overhead or compile effort during run-time is needed, precompiled code only lacks the ability of using run-time information for optimizations. Nevertheless, such a variant suffers from a severe drawback. When classfiles are precompiled, no dynamic code downloading may take place. This might be no problem for stand-alone devices, but the future of embedded systems is found in *internetworks* between different application modules (e.g. Jini-enabled networks). Those small modules dynamically federate to solve common tasks and are linked by a network. Java is a language designed for use in such environments and when precompiled code (without an additional runtime environment) is used, possible advantages are forfeit.

Figure 2 shows the three most important aspects when embedding Java in a reduced environment (i.e., memory, efficiency and portability), three traditional techniques trying to solve the problems (i.e., reduced JVMs, JIT compilers and AoT compilers) and our approach in comparison to these techniques. It seems obvious that a hybrid execution offers a promising runtime performance while keeping resource demands reasonable low and maintaining the possibility of executing arbitrary bytecode. To realize the advantages, the items listed below seem to play a crucial role when designing such a system:

- fast and accurate *profiling* mechanism, that can identify heavy-used code parts,
- a good *heuristic*, that evaluates the profiling data and estimates compile costs and expected preformance increases to choose code for the compiler,
- agressive *compiler optimization* techniques, that produce results fast while using as little memory as possible,
- *dynamic compilation* techniques, that can make use of run-time information,
- fast *context-switches* between interpreter and native code (JNI optimizations and efficient stack-to-register mapping),
- reduction of *library size* by allowing to run compressed classfiles (bytecode compression, dynamic downloading from trusted server) and
- mechanisms to automatically access *memory locations* within the interpreter to access embedded device registers.

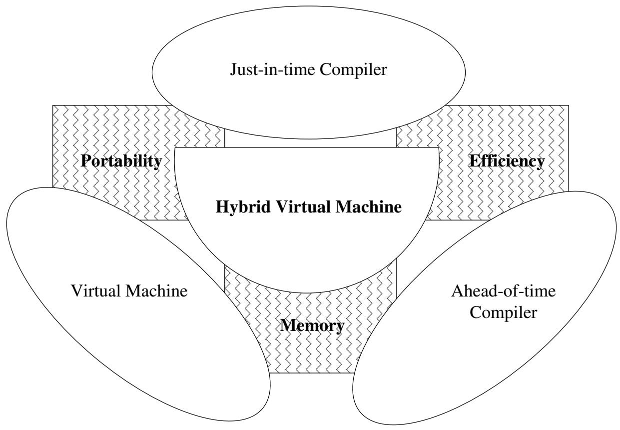

Figure 2. Hybrid Virtual Machine (HVM)

A profiling mechanism is necessary to provide the heuristic with information about code parts or procedures, that mainly contribute to the program's execution time. As the compiler should be able to optimize code parts, whole procedures and clusters of procedures, the run-time information has to be collected in different levels of detail. A call graph has to be built, that links each procedure with subroutines, it calls. Each node in that graph (representing a function) gets two counters, one representing the average number of executed cycles per invocation, the other one counting the number of invocations itself. This allows to pinpoint functions, which are responsible for a large fraction of the execution time. Such methods are further examined to identify parts of the code inside the procedure, that are heavily used. When certain hotspots are seen, only those parts are considered for compilation. In addition, the call graph can be used to perform interprocedural compilation. In contrast to JITs, that process each function independently, our system may combine heavy used function clusters into a single compilation unit and perform interprocedural optimizations (e.g. when allocating registers). Our *incremental* profiler allows to examine interesting procedures in more detail, while neglecting rarely used functions. Nevertheless, a number of closely related methods can be integrated into a single part of native code. When a profiler is available, its data might aid to estimate worst-case run-time performance. It is possible to mark certain methods as emergency routines and disallow the compiler to optimize it. In such cases, no additional indeterminism is introduced into the system. The used heuristic will have to consider three main aspects to decide, whether code should be compiled or interpreted:

1. *Interpretation speed* – It depends on the final implementation of the interpreter to determine the execution time, that is needed for each bytecode instruction.

2. *Compilation overhead* – This time is influenced by the implementation of the compiler itself, the applied optimizations and the code size, that has to be processed.

3. *Estimation of expected performance gain* – While empirical information (performance measurements) help to obtain accurate values for the first two points, the last one will be more challenging. A first approach might assume a constant speedup when moving from interpreted to native code. On the contrary, a more advanced heuristic should take loops into account, where loop invariant motion or loop unrolling can cause greater gains. The possibility of function inlining or constant/copy propagation is also more benefitial when applied to a loop body. It is therefore useful to let the heuristic choose functions, that spend most of their time executing tight loops or call small, inlineable functions.

The heuristic has to be adjusted to find code parts, where most compiler optimizations can be utilized. Those optimizations have to be carefully selected and coded to use as little memory as possible. It is desirable to prevent memory intense analysis (e.g. data dependency graph) where possible and limit optimizations to small code regions (e.g. peep-hole optimizations). Additionally, those optimization techniques have to be fast, because performance gains only result in net execution time reductions, when they can compensate compiler overhead. Loops have to be identified at run-time by monitoring branches, that jump to a previously executed basic block. Once a loop-header has been found, the control flow graph has to be traced to mark all basic blocks the loop consists of. When this analysis finishes, loop invariant code parts can be identified. This also helps to find induction variables, that can be used to remove array bound checks, inserted by the Java compiler. As each object reference and array access is dynamically checked, code can be significantly speed up by moving those checks outside the loop body. Such a technique has already been implemented for JIT compilers (e.g. [10]), where it provided significant speed up.

Closely related to compiler optimizations is the topic of dynamic compilation. Dynamic compilation uses run-time information to modify or create code to perform faster. According to [3], current dynamic compilation systems rely on some form of programmer direction to indicate where dynamic compilation would yield the highest profit. Some systems take an *imperative* (operational) approach to user direction, requiring the user to write programs that explicitly manipulate, compose, and compile program fragments at run time. This kind of system offers great flexibility and control to the programmer, at the cost of significant programmer effort. Alternatively, several dynamic compilation systems take a *declarative* (transformational) approach, with user annotations guiding the dynamic-compilation process.

In contrast to that, our system utilizes the profiling information gained during program interpretation and uses the above mentioned heuristic to identify interesting code parts automatically. This offers two advantages: First, it relieves the programmer of identifying heavily used code parts himself. Especially in larger projects, it is unrealistic to assume that every programmer has enough knowledge to estimate the execution frequency and performance impact of modules, he/she is responsible for. After the whole application has been merged and executed, our approach automatically identifies often-used code parts and can even adopt to dramatic changes in code utilization over time. Second, legacy applications without available source code that could be annotated or downloaded classfiles from unknown source are optimized, too. As no prior human intervention is necessary, the dynamic compilation is introduced automatically when needed.

Dynamic compilation enables optimizations based on the values of invariant data computed at run-time. Using the values of these run-time constants, a dynamic compiler can eliminate their memory loads, perform constant propagation and folding, remove branches they determine, and fully unroll loops they bound [1]. As the hybrid execution environment makes run-time information available, the compiler can easily profit from it.

An important aspect of our hybrid environment is the time, that is needed to switch between both execution variants. While the switch between the interpreter and the compiler itself should be fast, it is even more important to call native code from the interpreter with minimal overhead. As the compiler output is stored for later usage, we expect a compiled code fragment to be called more than once. Each of such an invocation has to proceed as fast as possible to prevent a performance reduction of the native code. Hence, a number of problems have to be solved when calling native functions. The following two seem to be the most severe:

1. *Possible interference with the GC* – When the native function allocates or frees memory, the GC has to be informed. As our compiler produces machine code from known bytecode, it can produce according information to prevent objects from being collected falsely.

2. *Interpreter stack values* – When switching to or away from the interpreter, stack values have to be kept in their correct register (or memory) locations. The compiler aims to put stack outputs into the correct registers and takes previous stack values from the correct places without inserting unnecessary spill code or register moves. The register allocator will use precolored nodes for initial and final stack values to reduce copy overhead. This method of fast switches between interpreter and native code can be used instead of the JNI for special annotated, precompiled functions. When a native function is compiled and the compiler produces the directives for the GC and adheres to our parameter passing schema, fast switches are possible for those functions as well.

As indicated before one major problem when using Java for embedded systems is the size of required libraries. We intend to allow the interpreter to contact predefined server to get needed library functions and immediately access them without prior classfile verification. This would allow several embedded devices, that are connected to a network, to share libraries on a common machine and to easily obtain needed code parts. Additionally, it is possible to compress the classfiles (or only the bytecode) and unzip it when needed. Actually, most optimizations performed by the ROMizer Tool could be applied to our system as well.

A last interesting point is the possibility of direct memory or register access by extending the original Java environment. As it has been mentioned before, Java doesn't know the notion of a memory address. Instead, references, which cannot be directly manipulated, are used to point to objects in memory. A execution environment for embedded systems could profit from an additional library, that would allow direct access to hardware resources. A special I/O library could be implemented, that allows objects to hold memory address and access them directly without the need to call a JNI function. The interpreter must be able to identify those objects and map a read/write to the corresponding memory location. While code that uses such a library could only be correctly executed on our HVM, the expected reduction of the overhead for a common operation in an embedded system may be worth such an extension (or incompatibility). The alternative would be a call to a native function that performs the access. Even when that function is optimized by using our fast context switch mechanism, a direct access from within the interpreter could be worth considering for embedded devices.

5 Conclusion

Implementing a Java runtime environment for an embedded system is not an easy task, as the stringent resource constraints limit the designer's freedom tremendously. The most important aspect when trying to cope with that reduced environment is to find a balanced solution which is fast, needs little memory and allows arbitrary code to be executed. The previously presented techniques only target two of those three requirements. A *just-in-time* compiler is fast and can run any code, but needs lots of memory. An *interpreter* uses little memory and can run arbitrary code, but is extremely slow. And *precompiled bytecode* is fast and memory efficient, but cannot provide a platform for new, downloaded bytecode. In our opinion, a *hybrid execution environment* offers the best balance between all three factors and allows much room for improvement when applying advanced compiler techniques. Such an environment could allow Java to enter the world of embedded systems and be the platform for Jini (or similar middleware layers), allowing the devices to easily integrate and communicate with each other. Therefore our future work might form the base for a new way of building embedded networks.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. *Proceedings Conference on Programming Language Design and Implementation*, 1996.

[2] K. Edwards. *Core Jini*. Prentice Hall, 1999.

[3] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Dyc: An expressive annotation-directed dynamic compiler for C. *Technical Report University of Washington*, 1998.

[4] Z. Hanzálek and T. Pácha. Use of the fieldbus systems in academic setting. *Proceedings of the 3rd IEEE Real-Time Systems Education*, 1998.

[5] Hewlett-Packard. Chai. *http://www.internetsolutions.-enterprise.hp.com/chai*, 2000.

[6] Insignia. Jeode. *http://www.insignia.com/4.0/products/-jeode.htm*, 2000.

[7] V. Ivanovic and M. Mahar. Circuit cellar ink. *Computer Application Journal*, 1999.

[8] W. Kastner and C. Krügel. Jini connectivity for EIB home and building networks – from design to implementation. *Proceedings EIB Scientific Conference*, 1999.

[9] W. Kastner, C. Krügel, and H. Reiter. Jini: ein guter Geist für die Gebäudesystemtechnik. *Proceedings Java Informations Tage JIT'99*, 1999.

[10] A. Krall. Efficient javavm just-in-time compilation. *Proceedings International Conference on Parallel Architectures and Compilation Techniques PACT'98*, 1998.

[11] NewMonics. Portable executive for reliable control. *http://www.newmonics.com/PERC*, 2000.

[12] Sun Microsystems. Picojava – the core. *White Paper*, 1998.

[13] Sun Microsystems. KVM – kilobyte virtual machine. *White Paper*, 1999.

[14] Sun Microsystems. Embedded java. *http://www.sun.com/-software/embeddedjava*, 2000.

[15] Sun Microsystems. Java 2 platform micro edition. *http://java.sun.com/j2me*, 2000.

[16] Sun Microsystems. Java native interface. *http://www.-sun.com/jni*, 2000.

[17] Sun Microsystems. Personal java. *http://www.sun.com/-software/personaljava*, 2000.