Java in Embedded Systems

Menno Lindwer

For several reasons, Java is not the most obvious language for embedded systems. It requires much more memory than most other languages and even with JIT compilers, it runs a lot slower. Therefore, systems running Java applications are more expensive than systems with the same functionality running natively. In many embedded systems industries (consumer electronics, networking, etc.), each additional cost is a market barrier. Besides that, Java is an inherently non-real-time language. However, some recent developments have turned Java into an obvious choice for those software tasks that do not require hard real-time operation, such as user interfaces. From the onset, Java was intended to reduce software development cost, software distribution cost, and lead time, which, in embedded systems, are rapidly becoming the dominant factors. Besides that, digital devices are increasingly required to interoperate with other devices and network servers. These new features require platform independent software, meaning that networked devices do not need to be aware of the internal architectures of their peers. Java comes with an extensive networking library and is compiled into a standardised platform independent distribution format, making ideal for such products. This leaves the reduction of Java execution cost in embedded systems as an interesting field for Research and and a challenge for Development...

Introduction

Java [1] is not the most obvious language for embedded systems. The reasons are manyfold. However, some recent developments have turned the tables.

Compared to conventional programming languages, such as C or C++, Java execution is expensive in terms of memory use, processor cycles, and power consumption. Until recently, the increased cost has proven to be a market barrier in embedded systems industries, such as consumer electronics and networking. To the users of many embedded systems, the increased functionality does not justify the increased cost. Besides that, the Java language is inherently 'real-time-unfriendly' [15]. It does not offer adequate constructs for specifying timing behaviour. The garbage collection and dynamic loading/linking features inhibit deterministic behaviour [15]. The programmer is shielded off from the underlying machine, giving him/her no handles to circumvent the problems.

However, recently system requirements (e.g. dynamic upgrade, networking, interoperability [6]) and business requirements (e.g. short time-to-market) have emerged that match quite perfectly with the mix of features offered by Java. This renewed interest has breathed new life into a number of optimisation efforts. It should be pointed out that none of the research topics are really specific to Java or invented specially for Java. But many have gained interest, funding, and momentum because of the possible application in Java. Some of those efforts, such as research into garbage collection algorithms goes back a long time [14]. Other developments, such as Just-In-Time (JIT) compilation [8], are quite recent. These optimisation ef-

forts in turn have brought the application of Java in cost-constrained embedded systems very near to commercial viability. In fact, at this moment several companies are introducing Java-enabled devices in one of the most cost-constrained industries: smartcards.

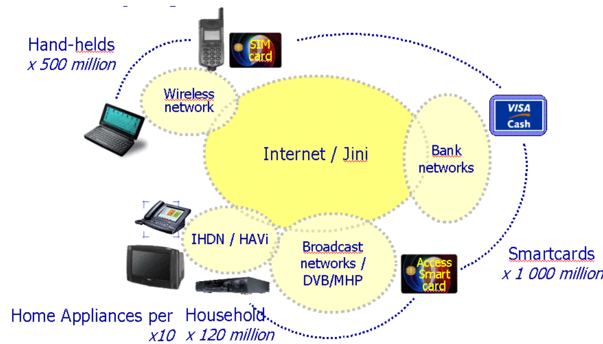

Figure 1: Markets for networked Java-enabled devices

Besides re-iterating the much-publicised software engingeering advantages of Java in an embedded context, this article aims to convey a deeper understanding of the performance and cost issues at play. I hope to show that Java's high execution and memory cost are not caused by singular features or failures in the Java system, but rather can be attributed to a multitude of deliberate considerations. This means that the law of 'retained misery' (Wet van behoud van ellende) almost always applies for attempts at optimisations. Therefore, good understanding of the issues is necessary, in order to prevent system designers from choosing solutions that exactly do not quite solve the problem or that are far more expensive than necessary.

This article is organised as follows: The next section discusses the business reasons, as related to the software development productivity gap, for applying Java. The second section investigates the impact of Java's feature set on execution cost. The third section takes us to the beef of the matter, namely the technical solutions for decreasing Java's execution cost. As dessert, the fourth section describes some hardware approaches for accelerating Java execution. The fifth section is the proverbial CFA (Conclusions, Future work, and Acknowledgements).

How is Java supposed to help increase software development productivity?

Java is an object oriented language [1]. Typical constructs from object oriented languages, such as inheritance, are usually considered benficial for software development productivity. However, this section is concerned with the features that are more specific to Java. Most of what is discussed below is not really new. But Java is the first widely used language that wraps them all in a nice, well-marketed package.

The following aspects of Java were included to improve software development productivity:

- **Language simplicity:** The language is easy to learn, i.e. the syntax is kept very close to that of C (C++). However, many of the constructs for which C++ is regarded as complex, have been left out. Apparently, this has not made the language less useful. A good example is the use of interfaces, instead of multiple inheritance (although some people question the use of either). Giving classes multiple interfaces is almost as powerful as multiple inheritance. However, it avoids problems such as classes inheriting the same class more than once.

  Because Java is easy to learn, it quickly gained a large developer base. This means that it will be more easy to find qualified software development personnel. Besides that, Java's simplicity allows other professionals than software developers to understand Java code. Therefore, other stakeholders can more easily participate in quality assurance of software projects.
- **Strong typing:** The Java language is strongly typed. This means that the compiler can statically check many commonly made mistakes, such as passing wrongly typed arguments, inadvertently losing or adding sign extensions at assingments, pointer arithmetic problems, etc.

  In fact, some researchers claim that, because of Java's strong typing compilers have more knowledge about the way the code will execute and can therefore apply more aggressive optimisations. This means that, in theory, Java code could run faster than C code...
- **Exception handling:** The software designer

can define the application level at which exceptions should be caught. The language offers constructs such that, at the levels below that one, developers can treat them transparantly (i.e. just pass them on). Of course the language also offers constructs for easy handling of exceptions at the level defined for that purpose. As an extra (and obvious) safety precaution, exceptions that do slip through that level will eventually be caught by the run-time environment.

- **Array boundary checking:** The code is guaranteed not to violate array boundaries. Software developers should still check for array boundaries. But if this fails, at least the state of the system does not get corrupted. The exception mechanism offers a standardised for handling those situations. Also, array boundaries are part of the language and can therefore be taken into account when writing loops. In fact, the exception catching mechanism can be legally used to end array handling loops (which is not to say that this is good programming practise :-)!
- **Automatic memory management:** Java does not have explicit memory allocation, nor can the programmer explicitly return memory. Memory is implicitly allocated during creation of objects. The language assumes that the operating environment contains a garbage collector that should appropriatly reclaim memory. Many languages (including C) actually do not have constructs for memory allocation and de-allocation; they are part of the library structure. In Java and C++, implicit memory allocation is part of the language. In Java, garbage collection is part of the language, in the sense that an explicit construct has been (purposely) omitted.
- **Platform independence:** This is achieved by compiling Java to an intermediary language (Java virtual machine language or Java bytecode, JBC). It is not a language feature. In fact, Java can very well be compiled directly into native code of any processor [5]. In principle, it is not possible to compile languages such as C and C++ to JBC, a.o. because Java does not require JBC to offer direct memory manipulation. Java Virtual Machine language interpreters (JVMs) are available for most (embedded) platforms. Combined with the next item, platform independence can result in enormous savings of development cost, because one does not need to maintain different software versions for different platforms.
- **Rich standardised set of APIs:** This set, including implementations (mostly in Java) was released together with the language. Software developers can safely assume implementations of these libraries to be available on any platform that runs the targeted flavour of Java. This means that developers can concentrate on solving the real problems. They do not have to spend effort studying APIs for many different platforms or implementing code for such basic operations, as set handling, sorting, hash tables, graphics primitives, etc.

The combination of these features is rumoured to result in a productivity increase per developer of a factor of 2.

Impact of Java's feature set on system cost

As far as the production cost of embedded devices is concerned, features such as language simplicity, strong typing, and exception handling come more or less for free. The other features come at a high cost.

Cost of performance penalty

The language specifies that every array access has to be checked against array boundaries. During an experiment on a real-life software system (a 15 KLoC simulation module, written in C), array bounds checking code could be switched off, increasing performance by 10This performance penalty translates into higher system cost. The processing elements inside an embedded system are usually dimensioned very carefully to exactly match the requirements of the software. Every unnecessary resource causes the eventual product to be more expensive and thereby lose market share. An exact factor for the increased system cost is difficult to give. It is usually not necessary to actually dimension the system 40 times larger than otherwise would be required. On the other hand, just scaling up the clock speed of the system is not enough. In order for a processor to actually benefit from higher clock speed, it should also have

bigger caches, wider memory lanes, faster on-board buses, more complex board designs, etc. A common way of increasing the Java performance is the application of a Just In Time (JIT) compiler, which reduces interpretation overhead (associated with executing JBC, Java's intermediary virtual machine language). However, even if a JIT compiler were to remove all interpretation overhead, Java is still about a factor 4 slower than native code (because of the other performance costs, such as array bounds checking and garbage collection).

Cost of increased memory requirements

The increased memory requirements are due to three factors:

- The JVM is a relatively large piece of software. The smallest full implementations have footprints of about 100KB. Because of optimisations, fancier threading mechanisms, and fancier user interface layers, this can increase to about 500KB. Since, in many cases, the JVM will be part of the firmware of a system, it will reside in ROM. ROM is much cheaper than RAM (about ??? times). Therefore, one would be tempted to discard this cost. However, RAM is faster than ROM, so that many embedded systems copy firmware to RAM, upon startup...
- Next to the JVM, a full Java system requires about 9MB of Java run-time libraries. For several reasons, it makes sense to place this code in rewritable memory. In a networked environment, this code definitely is eligible for updates. Besides that, for performance reasons, most JVMs modify the instructions as they execute them (turning dynamically linked code into semi-statically linked code). This requires the libraries to be placed in RAM.
- Java memory management is relatively expensive (in terms of memory utilisation). This is partly due to programming practises, partly it is inherent to the use of a garbage collector. Current programming practise results in the constant generation of many short-lived objects. For example, function results, as used in expressions often are objects, even though they could just as well be scalar types (integers, booleans). Returning an object causes that object to be created on the heap. However, immediately after evaluation of the surrounding expression, the returned function results become redundant.
- The garbage collector requires that the system contains more heap memory than strictly required by the application. Otherwise, the garbage collector would have to be activated whenever an object becomes redundant. Together, memory allocation and de-allocation require about 2MB of RAM, in order to run meaningful general purppose applications.
- However, mobile applications, provided by NTT DoCoMo's iMode (a Japanese mobile phone operator) show that careful design of Java software can result in useful applications that require only a few 10s of KB for dynamic memory allocations.

All-in-all, the minimum requirement for a full Java system is about 10MB of ROM and 2 MB of RAM. This comes on top of storage for the actual Java application code (which is assumed to be about the same as for the same application in native code[1]) and the requirements of the underlying operating system (which is still required when running Java).

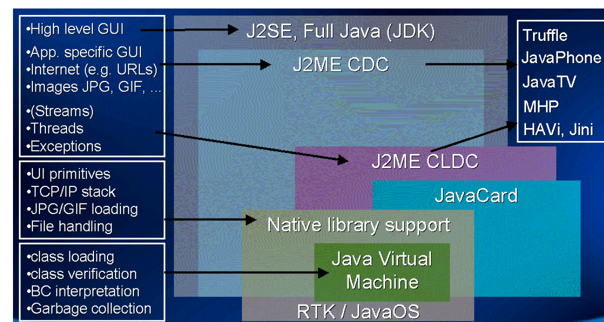

Figure 2: The complexity of the Java technology chart stems from the fact that it consists of many API sets, most of which are not precisely subsets of each other. The left column lists the functionality of the APIs. The rightmost box gives a number of product dependent API extensions

It should be noted that most embedded systems will not contain the full set of Java libraries. Part of the confusion around Java technology stems from the plethora of application domain specific subsets and extensions to the full Java API set. Experiments have shown that the full set can be brought

[1]On the one hand, JBC is about a factor 2 more compact than RISC code. On the other hand, JBC is packaged in Java class files, which contain a lot more data than just the JBC. Only some of that data gets discarded during loading.

back to about 500KB for mobile applications, by removing user interface and character conversion routines. When disregarding the performance penalty of ROM, and when using specially designed applications, the minimal footprint for a Java system ends up at about 1MB ROM and 100 KB RAM. Again, these costs come on top of the requirements for the actual (Java) applications and OS.

It is up to the system designer to choose the appropriate API set and live with the consequence of not being able to support all Java applications.

Cost of increased system complexity

These costs are difficult to quantify in a generic sense. But we can give an indication of the issues at play. What is meant here are the costs associated with having to design and maintain software and hardware components that are more complex than would be strictly required for native operation.

The simplest scenario is where efficient execution (i.e. interpreter performance) and graphics (user interface) are not required. There are few examples such systems, because devices without user interfaces usually constitute high-volume, low cost markets. Anyway, in that scenario, the only engineering cost is associated with porting a bare-bones Java interpreter to the target system. An experienced software engineer spends about half a man-year on porting, testing, and verifying a software stack like Sun's KVM. Given frequent updates, both in Java interpreter software technology and hardware platforms, the same cost will probably recur for maintenance on a yearly basis.

A more complex and realistic scenario would be the higher-end hand-held and mobile devices, in wich Java execution is added for user interface purposes and simple applications. Because of the kinds of applications, it is not required to have high-performance execution. And because of the market positioning, it is feasible to incorporate extra processing power. In this scenario, assuming that the platform already provides some degree of graphics support, the software development cost is increased by another 2 man-years for porting and verifying the native parts of the Java user interface toolkit (e.g. Sun's Abstract Windowing Toolkit, AWT). The maintenance cost will remain at about half a man-year, annually.

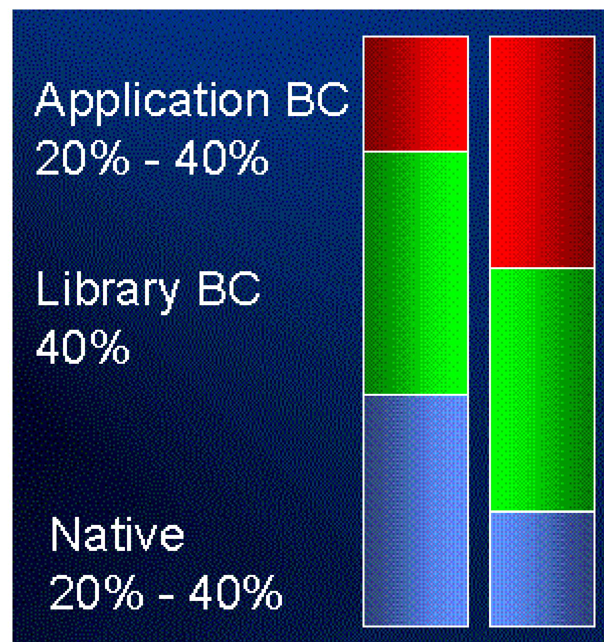

Figure 3: Applications spend 20% (large apps, right bar) to 40% (small apps, left bar) of execution time on native code. Consequently, 60% to 80% of time is spent on bytecode interpreting.

The next scenario are the medium to high-end consumer devices, such as set-top boxes. In the near future, they will adhere to standards such as Multimedia Home Platform (MHP [3]), Home Audio Video Interoperability (HAVi [6]), and Jini, which apply Java for complex tasks, combining system control and advanced user interface technology. Despite the market positioning (medium to high-end), the consumer price for such systems does not allow for the inclusion of PC-class hardware. In the first versions of these devices, the processor speed is limited to about 300MHz. Internal memory is in the range of 16MB to 32MB. Harddisks are not yet part of the package. The heavy use of Java in advanced user interfaces requires an optimised Java interpreter, sophisticated graphics stack, and native multithreading support. Several companies deliver speed-optmised interpreters, often in combination with JIT compilers. Because of their complexity, these systems require significant up-front and running licensing fees. Therefore, a choice for any package requires an extensive selection process. Usually, this selection process involves experimental ports of several rival software stacks onto simulators of the projected hardware system (during those preliminary experiments, the actual hardware is often still in the design phase). This selec-

tion phase may already involve several man-years work...

This paragraph dealt with some of the issues, surrounding the complexity of adding Java suppport to several types of embedded systems. Even though the list of issues per scenario and the set of scenarios are not complete, I hope this paragraph gives an idea of what to expect.

What Technologies are used to decrease Java's execution cost?

Obviously, the choice of technologies depends on the actual costs of the bottlenecks, as discussed in previous sections. For example, it makes no sense to optimise thread synchronisation for small embedded devices that are not expected to perform much multi-threading. However, in most cases, it does make sense to write the main interpreter loop in assembly, since this is where most JVMs spend about 80

When analyzing technologies, we can make several more or less orthogonal categories: hardware versus software, memory versus speed, and domain specific versus generic. Conveniently, this set of categories can be represented as a cube with more or less orthogonal sides. For example, JIT compilers are generic software enhancements, which impact the speed of the interpreter, at the cost of increased memory utilisation.

In the following sections, we will categorize and discuss a number of common optimisations to Java execution mechanisms. What we will see is that, as is usually the case, most optimisations involve trade-offs, where an improvement on one axis of the cube means a degradation on another axis.

JIT Compilers

JIT compilers [8] were already categorized as generic software solutions for increasing Java execution speed, at the cost of increased memory utilisation. It is therefore questionable whether they actuallly decrease execution cost. If memory is more expensive than processor silicon, this may not be the case.

Paradoxically, the pure JIT (Just In Time) compiler systems can also be called "Just Too Late", because they start compiling a (Java byte)code sequence at the exact moment the user/system (first) needs that particular function. Especially on embedded systems with relatively light processors, this initial call may take a long time. Also, this behaviour is particularly disruptive to real-time operation.

In order to prevent the Just Too Late behaviour and decrease memory cost of pure JIT compilers, profiling JIT compilers were introduced [HotSpot, 21]. Besides a compiler, such a system also contains a conventional interpreter-based execution mechanism. Initially, all code is executed by the interpreter. For every distinct code block (usually method), the frequency of its invocations is measured. When this frequency passes a certain threshold, the code block gets JIT-compiled. This apraoch generally decreases memory requirements, because no memory is waisted on the translation of blocks that are executed infrequently. However, we do have to take into account that the JVM has grown larger, because of the extra interpreter and profiling software. It also remains to be seen whether this approach performs as well as a JIT-only solution, since initial interpretation runs and profiling efforts may decrease overall performance.

Subset interpreters

These are domain specific software optimisations for reducing memory utilisation, usually at the expense of performance.

JavaSoft's KVM [21] and JavaCard [21] are examples of interpreters that do not support the full set of Java bytecodes.

The same goes for JVMs and library implementations that support only a subset of the standard Java APIs. Usually, those subsets are restricted in terms of user interface capabilities. For example, the Truffle [21] user interface library can only handle one application window at any time. It is implemented almost fully in Java, thereby reducing the required native functionality to a minimum (basically just pixel drawing). However, because almost all functionality is implemented in Java and supplied as Java bytecodes, Truffle is also relatively slow.

Specialised processors

These are generic hardware solutions for accelerating bytecode execution. Depending on (non-Java) legacy code requirements, the inclusion of a general purpose processor might still be necessary. In that case, the solution will come at the cost of increased silicon area and increased system complexity, both in terms of hardware and software system design.

Examples of specialised processors are PicoJava [16], Moon (Vulcan ASIC), and Shboom (Patriot Sciences). These are all processors that run the complete Java bytecode set natively. Keep in mind that the Java Virtual Machine language represents a Complex Instruction Set Computer (CISC). In fact, some Java bytecodes are extremely complex, involving memory allocation, initialisation, string table searches, and/or bytecode loading. On a regular software interpreter, this requires thousands of cycles. Normally, CISCs contain microcode, splitting complex operations in sequences of more basic operations. Microcode can be seen as a kind of processor-internal 'software'. Specialised Java processors can implement most bytecodes using microcode. However, the really complex bytecodes can not be implemented using such an extremely low-level language. Therefore, in spite of the promise of generic Java programmability, heavy investments in software development environments for those processors do have to be made.

And even if C/C++ software development environments are available for those specialised Java processors, they usually still don't run all the required legacy software. For example, because the legacy software was programmed in assembly or requires the support of an operating system that is not available for the Java processor. This would mean that a general purpose CPU needs to be added to the hardware system. If the project can afford to develop its own ICs, the additional direct cost is limited to a few euros worth of silicon per product. However, if the project has to rely on off-the-shelf hardware, extra ICs and increased circuit board size have to be added to the bill of material and product form factor. In terms of system design, going from a single-CPU to a multiprocessor solution adds a whole new set of problems, such as communication protocols, cache coherency protocols, and resource access arbitration. This gets aggravated in the case of heterogeneous multiprocessor designs, consisting of different types of processors.

Of course, a specialised Java processor (even a heterogeneous multiprocessor, incorporating a Java processor) probably contains less silicon than a single general purpose processor, offering the same Java performance. However, the question is, can't we find a more optimal approach, especially regarding the system design issues?

Bytecode accelerator hardware

Like specialised processors, these are hardware solutions for accelerating bytecode execution [11, 13]. However, they assist a general purpose processor in executing Java. Therefore, the complete solution always consists of a processor and an accelerator. However, since this processor is relieved of many of the Java execution tasks, it can be relatively small. Besides that, the accelerator module itself should be significantly smaller than the Java processors in the aformentioned heterogeneous designs.

In its simplest form [2], the accelerator is actually a translator from Java bytecodes to CPU native instructions. It can be seen as an instruction-level JIT compiler, implemented in hardware. Because it is implemented in hardware, it can perform its tasks in parallel to the processor doing the execution of the generated code. Because the translation takes place at instruction level, the system requires very little storage for intermediate results (a matter of several bytes, rather than several megabytes for a software JIT compiler).

One instance of such an accelerator will be discussed in more detail in the next section.

Graphics accelerators

Measurements have shown that, for meaningful Java applications, 2D graphics processing takes 10% to 20% of all processing time [20]. The reason is that most Java applications are user-interface intensive. After optimising Java bytecode processing, the relative impact of this factor will increase to 20% to 50% of all processing time. This means that graphics acceleration only becomes an issue after bytecode acceleration.

Graphics acceleration is a domain specific optimisation. It only has use in environments that require

media processing or have graphical user interfaces and large screens with some degree color depth.

Obviously, adding a graphics accelerator means higher hardware costs.

Multi-level and hardware garbage collectors

As was mentioned before, garbage collection also accounts for a significant amount of performance loss. As with graphics, this is very application dependent. Garbage collection seems to be a good candidate for acceleration through hardware. Some attempts have been made, including in the author's own projects [12]. In fact, it is not very difficult to implement certain garbage collection algorithms in hardware [14].

However, garbage collection algorithms themselves require substantial and variable amounts of memory. This can only be efficiently achieved by integrating the garbage collection logic with the memory devices. But the memory device business is very a cost-sensitive commodity market. Specially designed garbage collected memory chips can not be produced in sufficient numbers to make them commercially viable.

Another approach to at least alleviate the garbage collection bottleneck is to implement several types of software algorithms. Some algorithms are particularly good at quickly finding a large number of short-lived objects. Other algorithms are more thorough, but also more time consuming. Therefore, the heap is divided in a space for short-lived objects and a space for older ojects. The former ones are scanned quickly. Objects that have survived a number of those scans are moved to the latter space, which is scanned with the thorough procedure. The performance benefit results from the expensive procedure having to scan only part of the heap.

Optimised thread synchronisation

Java is a multithreaded language, heavily oriented towards re-use. This means that designers of Java classes alsways have to take into account that multiple threads may wish to concurrently access the internal data structures of those classes. Every object that may be accessed concurrently has to be protected against multiple threads interfering with each other's changes. Therefore, Java objects are synchronised very conservatively. The synchronisation operations involve threads performing operating system calls for claiming exclusive access, getting blocked as long as the claim can not be rewarded, and relinquising the claims when the operations have finished. These operating system calls are very expensive. A lot of time can be saved if one can utilise the fact that actual interference is very rare.

A Hardware approach to accelerating Java execution

At Philips Research, we've been working since the end of 1996 on hardware for Java acceleration in embedded systems. The work started from the following constraints:

- chip area increase should be minimal (e.g. much less than size of low-end 32-bit RISC CPUs),
- memory utilisation should not increase, compared to software interpreter,
- solution should be compatible with modern RISC CPUs (since general purpose CPUs remain necessary),
- solution should be modular (i.e. have minimal impact on other components in an embedded system), in order to facilitate re-use,
- performance increase should be at least a factor 5 over a regular software interpreter.

We found a solution in the form of a translator module, which assists general purpose CPUs in executing Java bytecodes. We called the module Virtual Machine Translator (VMI). Later, we found [2], which gives a good description of many of the concepts. VMI is very small. Essentially, it consists of tables that direct the translation. These tables can be implemented in a very compact way. VMI needs very little computational logic, since most computations take place on the general purpose CPU.

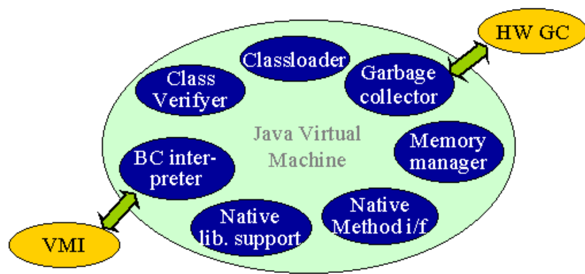

Figure 4: From a software point of view, the bytecode interpreter module is simply replaced by hardware (as the garbage collecture module might be)

Since part of the Java interpretation task is now implemented in hardware, the memory utilisation actually decreases slightly (we need less code to implement the Java interpretation software). Since the actual operations take place on the general purpose CPU (remember that VMI is only a translator), there are no problems with data coherency between the two processing elements. Contrary to most other accelerators, VMI has been developed completely separately from the CPU. CPU and VMI only communicate through the on-chip system bus. Most companies use standardised on-chip system buses. Therefore, building VMI for a specific on-chip system bus, means it is compatible with all CPUs that can be attached to that bus. The fact that VMI communicates only through a standardised bus also means no other parts of the hardware system need to be modified.

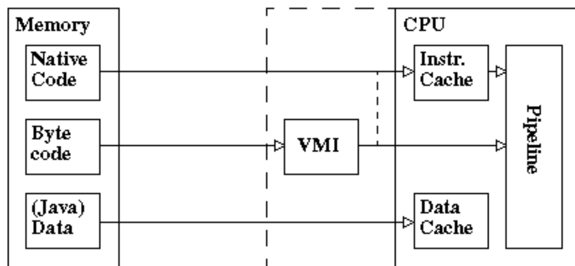

Figure 5: From an abstract hardware point of view, VMi is placed between the memory and the CPU pipeline, feeding the pipeline with translated bytecodes

After having indicated how the solution is intended to solve the problem, while keeping within the constraints, it is now time for some more technical detail.

Most computer systems contain at least a CPU (Central Processing Unit) and a memory. The CPU can be seen as a robot, which is able to execute sequences of instructions. For example, a car construction robot repeatedly executes instructions that tell it to move, pick up components, attach components, measure parts of the construction, etc. In order to assemble a complete car, such a robot executes thousands of those instructions. In the same way, CPUs execute billions of instructions for a simple task, such as drawing an image on a screen or printing a document. The CPU reads those instructions from the afore-mentioned memory. Thus we find the instructions for the Java applications in the memory and require the CPU to fetch and subsequently execute them. However, general purpose CPUs do not understand the Java instructions (also called 'bytecodes'). This is where the Java Virtual Machine software comes in. It translates the bytecodes into instructions that the CPU does understand. This means that next to the functionality of the bytecodes themselves, the CPU needs to spend time on the interpretation task. A very simple interpreter for some of the bytecodes could be programmed as follows:

```
{1.unsigned interpreter( char *pc ) {
2.  /* 'pc' points at bytecodes */
3.  unsigned sp[STACK_SIZE];
4.  /* 'sp' compute result stack */
5.  while(TRUE) {
6.    switch( *(pc++) ) {
7.      case push_const :
8.        *(sp++) = *(pc++);
9.        break;
10.      case pop :
11.        sp--;
12.        break;
13.      case add :
14.        *(sp-2)=*(sp-2)+*(sp-1);
15.        sp--;
16.        break;
17.     case ret :
18.        return *(sp-1);
19.        break;
20.    }
21.  }
22.}
}
```

The above code does not need check stack under/overflow or code overrun conditions, because in Java this is done statically.

Notice that the above instructions (push_const, pop, add, ret) are about as powerful as regular CPU in-

structions. However, the while-switch-case-break construction usually requires between 10 and 40 CPU instructions per iteration. The actual functionality of the bytecodes (involving sp in the above code) requires between 5 and 10 CPU instructions. The reason is that the stack pointer-relative addressing introduces an extra indirection and because the stack pointer itself needs to be updated. This means that a CPU needs to execute 15 to 50 instructions for operations for which it would normally require 1 or 2 instructions. This means a 7x to 50x interpretation and execution overhead per bytecode.

Going back to the accelerator concepts:

In order to reduce the interpretation overhead, the program counter is moved from the CPU into the accelerator. The accelerator now reads the bytecodes from the memory and determines the location in its translation tables of the corresponding sequence of CPU instructions. It performs this task within the time the CPU needs to execute the previous translation. Thereby, this bottleneck is completely removed.

In order to reduce the time needed for the actual functionality (remember that push, pop, add, and ret require 5 to 10 CPU instructions), the stack pointer is also moved from the CPU into the translator. Now, instead of just providing the corresponding sequence of translated instructions, including stack pointer indirections, the translator simplifies the translation by substituting the stack values in the instruction sequences [inspired by 4] and doing the stack pointer updates internally. The resulting translation sequences have an average length of about 2 CPU instructions. All-in-all, the translator provides a speed-up on the above bytecodes of at least a factor 15.

Conclusions, Future Work, and Acknowledgements

Java is becoming an important language for embedded systems programming. However, before Java-based products can become a success, the cost of Java execution mechansism has to be reduced.

Most companies providing Java execution mechanisms advertise their solutions citing a single benchmark (e.g. [17]). In this article, I hope to have made it clear that performance is not the only factor at stake and that JVMs are such complex systems that a single-point measurement of performance can not give an accurate indication of relative qualities.

The interest in incorporating Java in embedded systems is still increasing. Despite Moore's law (prescribing that compute power will steadily increase), there is a continuous need to taylor Java implementations to the strict requirements of embedded systems. Java acceleration technologies seem to offer interesting advantages, but their commercial viability still needs to be proven. On the short term (during 2001), JIT compilers will find their way into systems with little real-time and memory restrictions. On the somewhat longer term (before 2003), we will see bytecode accelerators opening up extremely constrained devices to the Java language. 2D graphics accelerators are already used in embedded systems with heavy user interfaces. The sophistication of garbage collection systems is constantly increasing, but much work remains to be done here. It is questionable whether garbage collection hardware will ever become viable.

I would like to thank the members of the Java Hardware Accelerator project at Philips Research for their enthousiasm, in particular Otto Steinbusch (currently at Philips Semiconductors), Narcisse Duarte (currently at Canal+), and Selim Ben-Yedder. I've also had many valuable discussions with Pieter Kunst, Nick Thorne, Harald van Woerkom, and Paul Stravers.

References

[1] K. Arnold, J. Gosling, D. Holmes, *The Java Language Specification*, Addison-Wesley 2000, ISBN 0-201-70433-1

[2] E.H. Debaere, J.M. van Campenhout, *Interpretation and Instruction Path Coprocessing*, The MIT Press, 1990, Cambridge MA, USA

[3] Digital Video Broadcast Multimedia Home Platform, http://www.mhp.org/html_index.html

[4] M.A. Ertl, *Implementation of Stack-Based Languages on Register Machines*, PhD thesis Technische Universitaet Wien, Vienna 1996

[5] The Free Software Foundation, *The GNU Compiler for the Java Programming Language*, http://www.gnu.org/software/gcc/java

[6] HAVi, http://www.havi.org

[7] J. Hoogerbrugge, L. Augusteijn, *Pipelined Java Virtual Machine Interpreters,* 9th International Conference on Compiler Construction, April 2000, Berlin, Germany

[8] A. Krall, R. Grafl, *CACAO - A 64 bit JavaVM Just-in-Time Compiler,* Institut fuer Computersprachen, Technische Universitaet Wien, Vienna, 1998

[9] M. Levy, *Java to Go: Part 1; Accelerators Process Byte Codes for Portable and Embedded Applications,* Cahners Microprocessor Report, February 2001

[10] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification,* Addison-Wesley, 1996-09

[11] M. Lindwer, *Versatile Java Acceleration Hardware,* 2001, to appear...

[12] X. Miet, *Hardware for (Java) garbage collection,* ENST, Paris, France, October 2000

[13] Nazomi, *Nazomi Communications; High Performance Java Technology for Mobile Wireless and Internet Appliances,* http://www.nazomi.com

[14] K. Nilsen, *Progress in Hardware-Assisted Real-Time Garbage Collection,* Iowa State University, 1995, http://www.newmonics.com/dat/iwmm_95.pdf

[15] K. Nilsen, *Issues in the Design and Implementation of Real-Time Java,* NewMonics, Inc., April 1996, http://www.newmonics.com/dat/rtji.pdf

[16] J.M. O'Connor, M. Tremblay, *PicoJava-I: The Java Virtual Machine in Hardware,* pages 45-57, IEEE Micro, 1997-03/04

[17] Pendragon Software, *CaffeineMark 3,* http://www.pendragon-software.com/pendragon/cm3/info.html

[18] Philips Research, *Mobile phones, set-top boxes, ten times faster with new Philips accelerator for Java,* January 2001, http://www.research.philips.com/pressmedia/010101.html

[19] Philips Semiconductors, *Java hardware accelerator for embedded platforms, Philips Semiconductors World News,* November 2000, http://www.semiconductors.philips.com/publications/content/file_680.html

[20] O.L. Steinbusch, *Designing Hardware to Interpret Virtual Machine Instructions; Concept and partial implementation for Java Byte Code,* Master's thesis, Eindhoven University of Technology, February 1998, TUE-ID363006

[21] Sun Microelectronics, *JavaSoft; The Source for Java Technology,* http://www.javasoft.com

Biography. Menno Lindwer is a Senior Scientist at Philips Research in Eindhoven (The Netherlands). He has been involved in hardware design (methodology) since 1991, graphics acceleration since 1995, and Java acceleration since 1996. Menno holds a Master's Degree in computing science from Twente University of Technology (1991) and a post master's degree in software technology from Eindhoven University of Technology (1993). Other interests include object oriented design, simulator technology, and system-on-silicon architecture. Menno joined Philips Research in 1995. Currently, he is in charge of the Platform Independent Processing and Java Hardware Acceleration projects at Philips Research in Limeil-Brevannes (France) and Eindhoven (The Netherlands). Previous work experience includes a.o. artificial intelligence systems, research in delay insensitive asynchronous circuits, and performance analysis of 3D graphics accelerators.