Java Virtual Machines für ressourcenkritische eingebettete Systeme und Smart-Cards

Frank Golatowski · Hagen Ploog · Ralf Kraudelt
Tino Rachui · Olaf Hagendorf · Dirk Timmermann

Universität Rostock
Fachbereich Elektrotechnik und Informationstechnik
Institut für Angewandte Mikroelektronik und Datentechnik
http://www-md.e-technik.uni-rostock.de

Abstract. In diesem Beitrag werden Erfahrungen beim Entwurf und der Realisierung verschiedener Java-basierter Systeme beschrieben. Diese Systeme sind auf verschiedene Anforderungen und Anwendungsgebiete zugeschnitten. Zuerst wird die Realisierung einer virtuellen Javamaschine (JVM) für eingebettete PC-basierende Systeme beschrieben. Dabei werden die notwendig gewordenen Modifikationen hervorgehoben, um den Einsatz auf solchen ressourcen-kritischen Systemen zu ermöglichen. Einen wesentlichen Schwerpunkt bildet dabei die Umsetzung von E/A-Zugriffen über Standard-Unix-Gerätefunktionen. Als weiteres wird eine JVM für ein mikrocontroller-basiertes System vorgestellt. Als Mikrocontroller kommt hier ein 8-Bit-Prozessor zum Einsatz. Ziel dieser Implementierung ist es, Voraussetzungen zu schaffen und Erfahrungen zu sammeln für die Entwicklung eines Java-Prozessors. Diesen stellen wir als dritte Komponente vor. Dieser Spezialprozessors ist für den Einsatz als Smart-Card-Prozessor vorgesehen. Die besondere Aufgabenstellung besteht darin, die Objektorientierung hardwaretechnisch umzusetzen. Die beiden virtuellen Maschinen wurden beispielhaft für 80x86 (mindestens 80386) bzw. 8051-Prozessoren implementiert. Sie können aber auch auf andere Prozessoren portiert werden.

1 Einleitung

Die Programmiersprache Java kann in kleinen eingebetteten Systemen und auch in offenen Automatisierungssystemen eingesetzt werden. Allerdings sind dabei einige Voraussetzungen zu erfüllen. Client-Systeme, die auf dieser Technologie basieren, können Anwendungen bei Bedarf dynamisch übers Netzwerk (Intranet, Internet) von beliebigen Standorten nachladen. Jene Systeme sind unabhängig von dem zugrundeliegenden Betriebssystem und werden folglich auch als offene Systeme bezeichnet. Java erfüllt die Anforderungen, die an eine portable, robuste, sichere und leistungsstarke Programmiersprache gestellt werden. Java-Programme werden als Applets bezeichnet. Applets werden typischerweise übers Netzwerk geladen und vom Client-Rechner, der Java-Code interpretieren kann, ausgeführt. Normalerweise wird dazu ein Java-fähiger Web-Browser verwendet. Beim Einsatz in eingebetteten Systemen kann zwischen Systemen unterschieden werden, in denen das Laden übers Netzwerk sinnvoll ist, und jenen, in denen ein Laden übers Netzwerk nicht erwünscht ist.

2 Stand der Technik

Der Einsatz von Java in eingebetteten und in Echtzeitsystemen steht im Mittelpunkt zahlreicher Arbeiten.
Zu den ersten Arbeiten, die sich mit dem Einsatz von Java in zeitkritischen eingebetteten Systemen beschäftigen, gehören die Arbeiten zu Real-Time Java von Kevin Nilsen [15, 16]. Hier werden Erweiterungen zur Sprache Java vorgestellt, um die Sprache als Echtzeitsprache einzusetzen. Wesentliches Augenmerk wird dort auf den Garbage Collector, den Zugriff auf Prozeßperipherie und auf die Einhaltung harter Echtzeitbedingungen [3] gelegt. Seit Mitte 1999 arbeitet die „Real Time Java Experts Group“ [21] innerhalb des von SUN initiierten „Java Community Process“ [22] an der Ausarbeitung einer Echtzeit-Java-Spezifikation.
Weitere Arbeiten beinhalten eine Neuimplementierung der JVM. Eine Neuimplementierung hat in der Regel die Verkleinerung der monolithischen Original-JVM [23], die Schaffung von Schnittstellen für den Zugriff auf Prozeßperipherie und die Ablauffähigkeit von Java-Programmen auf eingebetteten Systemen zum Ziel. Zu diesen Arbeiten gehört die eingebettete virtuelle Javamaschine von Hewlett Packard [8]. Sie ist voll kompatibel zur kompletten Java-Spezifikation von Sun [23] und für den Einsatz in Konsumgütern (elektronische Geräte, z.B. Drucker) entwickelt und optimiert. Die für eine jeweilige Anwendung erforderlichen Java-Bibliotheken sind konfigurierbar. Die virtuelle Maschine wurde von mehreren führenden Echtzeitbetriebssystemherstellern (u.a. Lynx, WindRiver, ISI, etc.) und von Microsoft für WindowsCE lizensiert. Sie umfaßt alle wesentlichen Komponenten der SUN-JVM.
Die Systeme Kaffe [32] und Ghost [13] sind offene JVM-Implementierungen. Während Kaffe für den Einsatz auf Desktop-Systemen vorgesehen ist, wurde Ghost für den Einsatz auf kleinen mobilen Geräten entwickelt. In diese Kategorie kann auch der Software-Coprozessor von NSI [17] eingeordnet werden. Weitere Arbeiten beschreiben Möglichkeiten der Übersezung von Java-Code. Während in [10] die Übersetzung von Java in Maschinencode vorgenommen wird, wird in [9] eine Methodik zur Konvertierung von C in Java vorgestellt.
Eine ebenso stürmische Entwicklung findet im Bereich der Entwicklung von Java-Prozessoren statt. Diese können in echte Java-Prozessoren und Prozessoren, die die Ausführung von Java-Programmen unterstützen (s. z.B. [31]) unterteilt werden.
Von SUN selbst werden verschiedene Java-Prozessor-Klassen hergestellt. PicoJava ist ein konfigurierbarer Java-CPU-Kern. Die Prozessoren sind optimiert für die Ausführung von Java-Code, unterstützen aber auch die Ausführung von C und C++ Code. Das Ziel dieser Unterstützung besteht in der Migration dieser Programmiersprachen [25, 28, 6].
Neben diesen echten Java-Prozessoren gibt es Prozessoren, die durch ihre Architektur geeignet sind, JVMs effizient zu implementieren und dadurch das Ausführen von Java-Code beschleunigen. Dazu gehört der PSC1000 von Patriot Pacific [18]. Der PSC1000 ist ein stackbasierter 32-Bit-RISC-Prozessor. Durch die stackbasierte Architektur ist der Prozessor geeignet, um JVM und Just-In-Time-Compiler zu implementieren. Er verzichtet auf leistungssteigernde Architekturmerkmale, wie z.B. Pipeling, und enthält keine Programm- oder Daten-Caches.

Weiterhin wird ein operandenloser Befehlssatz verwendet und auf breite Befehlsstrukturen mit Mehrfachoperanden verzichtet. Ein Befehl ist 8 Bit breit. Operanden werden im Operandenstack erwartet. Der Prozessor enthält einen separaten I/O-Prozessor für die Entkopplung zeitkritischer Vorgänge von reinen Rechenaufgaben.

3 JVM für eingebettete PC-Systeme

In den folgenden beiden Abschnitten werden, ausgehend von der Beschreibung einer allgemeinen JVM, die erforderlichen und realisierten Modifikationen für zwei verschiedene virtuelle Java-Maschinen (JVM) beschrieben. Die erste JVM ist lauffähig auf eingebetteten PC-Systemen und wird im weiteren als miniJava bezeichnet. Die zweite auf 8051-Systemen ausführbare Maschine bezeichnen wir als smartJava.

3.1. Virtuelle Java-Maschine

Die virtuelle Java-Maschine JVM ist erforderlich zur Ausführung von Java-Programmen. Compilierte Java-Programme werden nicht direkt auf der CPU ausgeführt, sondern werden von der JVM ausgeführt. Die JVM ist das Softwareinterface zwischen dem compilierten Java-Programm und der Hardware. Die JVM kann als virtueller Prozessor aus der Sichtweise des Programmierers betrachtet werden. Die JVM interpretiert jedoch nicht nur die Java-Befehle, sondern führt auch Sicherheitsverifikationen und andere Aktivitäten aus. Die JVM besteht aus den Komponenten: Bytecode-Interpreter, natives Interface, Exception-Behandlung, Multithreading, Garbage Collection, dynamischer Klassenlader und Bytecode-Verifier. Abbildung 1 zeigt den Aufbau einer JVM.
Für den Entwurf der virtuellen miniJava Maschine waren zwei Hauptpunkte zu beachten:

- Minimierung der Speicheranforderungen
- Portabilität

Es ist kaum möglich, Teile der Spezifikation aus dem Design zu streichen. Um Java Applikationen ohne Einschränkungen ablaufen lassen zu können, müssen unbedingt alle Teile implementiert werden. Es gibt nur relativ wenig Ausnahmen. Das Modul zur Überprüfung des Class-Files und des Bytecodes muß nur in Teilen vorhanden sein, damit die JVM einsatzfähig ist. Die Folge ist allerdings eine Sicherheitslücke. Für eingebettete Systeme, die geschlossen sind, und für die ein unkontrollierter Eingriff nicht zu erwarten ist, kann man dies akzeptieren. Auch ist es möglich, auf das Multithreading zu verzichten.

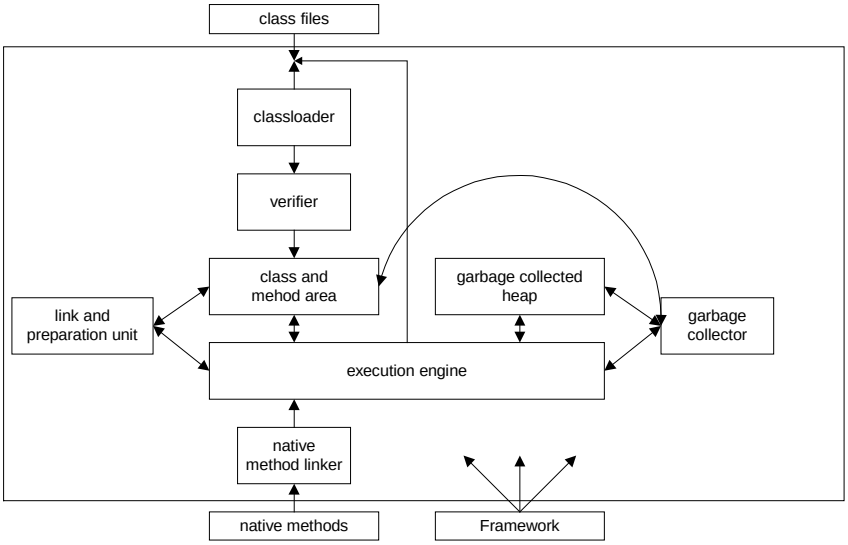

Abbildung 1. Bestandteile einer JVM

Class-Loader

Der Class-Loader erwartet ein sich schon im Speicher befindendes Abbild eines class files. Dieses Speicherabbild wird sequentiell abgearbeitet und entsprechend der JVM Spezifikation interpretiert. Dabei wird eine interne Datenstruktur, ein Class-Pool Eintrag angelegt und begonnen zu initialisieren. Der Class-Pool spielt eine wichtige Rolle im gesamten Design. Er enthält für jede geladene Klasse genau einen Eintrag. Dieser wiederum enthält die kompletten Informationen für die jeweilige Klasse. Alle weiteren Teile der JVM, die Informationen über eine Klasse benötigen, auf Methoden und auf Felder zugreifen, beziehen sich auf diese Strukturen.

Die Datenfelder korrespondieren weitestgehend direkt mit den entsprechenden Strukturen im Class-File (siehe [12] [14]). Für die interne Verwaltung sind zwei Datenfelder hinzugekommen:

- Head

zeigt auf das Objekt, das diese Klasse repräsentiert; im Laufzeitsystem muß für jede geladene Klasse genau eine Instanz von `java.lang.class` existieren. Diese ist in großen Teilen mit dem class pool Eintrag identisch.

- State

kennzeichnet den aktuellen Status im Laden/Linken/Vorbereiten

Verifier

Der Verifier ist das einzige Modul, das nicht in allen Teilen zwingend erforderlich ist. Selbst der originale Java Interpreter aus dem JDK von Javasoft führt nur einige Teile der Class-File-Verifikation standardmäßig aus.

Die gesamten Überprüfungen haben einen umfangreichen Programmcode und lange Laufzeiten zur Folge. Da sie nur dann wirklich notwendig sind, wenn die Applikation auf unsichere Klassen zugreift und dies für die Klasse der von uns betrachteten eingebetteten Systemen ausgeschlossen werden kann, ist es nicht notwendig, sie in einem Minimalsystem zu implementieren.
In miniJava gibt es keinen direkten Verifier. Alle unbedingt notwendigen Überprüfungen werden in anderen Modulen durchgeführt. Die Überprüfungen, die zur Laufzeit ausgeführt werden sind:

- Initialisierung von Klassen, falls noch erforderlich
- Typüberprüfung bei Zuweisungen
- Test, ob Methoden und Felder, auf die zugegriffen werden soll, existieren
- Test, ob die gerade aktive Methode Zugriffsrechte auf referenzierte Methoden und Felder hat

Link- und Init-Unit
Dieses Modul muß eng mit dem Class-Loader und mit der Execution-Engine zusammenarbeiten. Die Hauptaufgabe besteht darin, die vom Class-Loader bereits teilweise initialisierten Datenstrukturen in einen Zustand zu überführen, so daß sie durch die Execution-Engine ausgeführt werden können. Alle Felder müssen in einen definierten Ausgangszustand überführt und die <cinit> Methode aller gebundenen Klassen aufgerufen werden. Diese spezielle Methode ist eine normale Java Methode, die vom Compiler für die Initialisierung einer Klasse oder eines Interface erzeugt wird.
In miniJava erfolgt eine komplette Umsetzung dieses Moduls.

Nativer Methoden-Linker
Dieses Modul dient der Verbindung der nativen Bibliothek und der virtuellen Maschine. Die virtuelle Maschine erhält Zugriff auf die Funktionen, die nicht in Java, sondern in einer plattformspezifischen Form implementiert wurden. Weiterhin wird mit diesem Modul der Zugriff auf bestimmte Ressourcen der Java Laufzeitumgebung ermöglicht.

Stack-Frame
Der Stack-Frame ist eine weitere wichtige Datenstruktur innerhalb der JVM. Wie bei vielen anderen Teilen ist auch der Stack-Frame nur in seiner grundsätzlichen Struktur und Funktion definiert. In miniJava wurde der Stack-Frame aus Effizienzgründen in zwei Teile aufgeteilt:

- der Stack-Frame selbst enthält alle notwendigen Daten (Programmzähler, Stackpointer, Zeiger auf oberstes Stackelement, Zeiger auf lokale Variablen und eine Reihe von verschiedenen temporären Variablen, die zur Ausführung bestimmter Bytecodes benötigt werden)
- die Exception-Liste ist eine einfach verkettete Liste, deren Knoten Zeiger auf das dazugehörige Objekt, die darauf ausgeführte Methode und den Stack enthält

Die beiden Datenstrukturen werden hauptsächlich in zwei Modulen der VM verwendet, in denen jeweils eine hohe Geschwindigkeit erforderlich ist:
Der erste Teil wird bei der Abarbeitung der Bytecodes verwendet und enthält den vollständigen Zustand des Interpreters zu einem bestimmten Zeitpunkt.

Der zweite Teil bildet die Grundlage für die Suche nach ‘lebenden’ Objekten im Garbage Collector.
Durch die Aufteilung in zwei Datenstrukturen ist es möglich, in beiden Modulen nur eine einfache Verzweigung zu verwenden. Der entstehende Geschwindigkeitsverlust bei einer doppelten Initialisierung ist gegenüber den Vorteilen vernachlässigbar klein.

Garbage Collector
Der Garbage Collector gehört zwar nicht direkt zur Virtuellen Maschine, ist aber so eng mit den Datenstrukturen und der Funktionsweise dieser verbunden, daß er an dieser Stelle mit behandelt wird.
In miniJava wird ein Tracing-Algorithmus wegen seiner einfachen Implementierung eingesetzt. Dieser Algorithmus geht von einer oder mehreren statischen Wurzeln aus. Das Garbage Collecting erfolgt in zwei Phasen. In der ersten wird der komplette Baum der Objektreferenzen von den Wurzeln aus durchsucht und alle erreichbaren Objekte markiert. Nachdem der komplette Baum durchsucht wurde, können in der zweiten Phase alle nicht markierten Objekte freigegeben werden.
Der Vorteil dieses Algorithmus liegt darin, daß zur Laufzeit kein zusätzlicher Aufwand getrieben werden muß, um die Objekte zu verwalten. Der Nachteil ist aber der relativ hohe Rechenaufwand, der erforderlich ist, um den Objektbaum zu durchsuchen und löschbare Objekte zu markieren.
Aus diesem Grund ist miniJava nicht für Echtzeitsysteme mit harten Zeitanforderungen [3] einsetzbar.

Execution-Engine
Die Execution-Engine bildet den eigentlichen Kern der Virtuelle Maschine. Durch dieses Modul werden die Bytecodes direkt abgearbeitet. Er arbeitet eng mit den anderen Teilen der VM zusammen beziehungsweise greift auf von diesen erzeugte Datenstrukturen zurück.
Um das Laufzeitverhalten zu testen, wurde die innere Interpreterschleife in zwei unterschiedlichen Versionen implementiert:
- Switch-Case-Konstrukt
- Array mit Zeigern auf Funktionen; die Indizierung erfolgt direkt über den Befehlscode

Im Vorfeld wurde angenommen, daß die Lösung mit dem Array aus Funktionszeigern eine wesentliche Geschwindigkeitssteigerung bringen würde. Dies ist nicht der Fall. Eine Analyse des erzeugten Assemblercodes ergab, daß das Switch-Case Konstrukt von den verwendeten Compilern so umgesetzt wurde, daß es mit einigen wenigen Vergleichen auskommt und die Funktionsblöcke selbst mit einfachen Sprüngen erreicht werden. Im Gegensatz dazu erfolgt bei der zweiten Lösung ein vollständiger Kontextwechsel auf dem Stack, der bei den überwiegend sehr kleinen Funktionsblöcken einen erheblichen Geschwindigkeitsverlust gegenüber dem Vorteil des kürzeren Aufrufs bringt (siehe Leistungstest in Abschnitt 3.4).
Falls bei einer Portierung auf ein anderes System der Compiler zu dieser Optimierung nicht in der Lage ist, kann durch eine einfache Änderung einer Präprozessordefinition das Verhalten des Interpreters geändert werden.

3.2. Framework

Das Framework besteht aus mehreren Modulen, die nicht direkt zur virtuellen Maschine gehören. Sie stellen eine Basisfunktionalität zur Realisierung einer leicht portierbaren und möglichst plattformneutralen Java Laufzeitumgebung bereit.
Zum Framework gehören der Class-File-Loader, die Gerätetreiberschnittstelle, der IP-Stack, die Stringverwaltung und Speicherverwaltung. Nachfolgend wird für den Class-File-Loader und die Gerätetreiberschnittstelle eine Kurzcharakterisierung vorgenommen.

3.2.1 Class-File-Loader

Der Class-File-Loader ist stark plattform- und anwendungsabhängig. Für verschiedene Anforderungsprofile sind für die Class-Files folgende Varianten implementiert worden:

1. Class-Files befinden sich in einem Verzeichnis auf einer Diskette/Festplatte oder einem ähnlichen Massenspeicher (nur in der Win32 Version aufgrund langer Dateinamen lauffähig)
2. Class-Files werden mit der Laufzeitumgebung in einer Applikation integriert
3. Class-Files werden über ein IP Netzwerk mit dem TFT Protokoll von einem Applikationsserver geladen

Die erste Variante ist für den Einsatz auf Desktop-Systemen, u.a. auch für Cross-Entwicklungen geeignet.
Die zweite Variante ist für den Einsatz in einem Einzelsystem prädestiniert. Während der Entwicklung der Applikation wird mit einem Zusatzwerkzeug festgestellt, welche Klassen benötigt werden. Diese werden dann mit dem Laufzeitsystem zusammengebunden. Damit ist es möglich, ein speicheroptimiertes System zu erstellen, das nur die benötigten Klassen enthält.
Die dritte Variante ermöglicht bei Vorhandensein eines Netzwerkanschlusses das Nachladen benötigter Klassen mittels der Standardprotokolle BOOTP und TFTP. Sie erfordert aber zusätzlich einen Applikations- und Bootserver.

3.2.2 Gerätetreiberschnittstelle

Von entscheidender Bedeutung ist der Zugriff auf E/A-Geräte. Hierfür wurde eine an Unix-Systeme angelehnte offene Schnittstelle umgesetzt.
Der Hauptunterschied besteht jedoch darin, daß die miniJava Gerätetreiber nicht in ein Dateisystem integriert sind und damit auch nur eine Untermenge der üblichen Funktionen (siehe [29]) zur Verfügung stellen. Diese Untermenge ist aber für die Funktionalität, die für das Laufzeitsystem benötigt wird, vollkommen ausreichend.
Die einzelnen Geräte und deren Funktionsblöcke werden eindeutig durch eine Major und Minor-Nummer gekennzeichnet. Jeder Treiber muß bei seiner Initiali-

sierung eine Datenstruktur mit Funktionszeigern füllen. Über diese Struktur spricht die Gerätetreiberschnittstelle die Funktionen eines einzelnen Treibers an.
Folgende Funktionen sind vom Treiber zu implementieren:

- `read` liest n Bytes vom Gerät
- `write` schreibt n Bytes zum Gerät
- `select` testet, wie viele Bytes ohne Blockierung gelesen werden können
- `sync` schreibt alle eventuell gefüllten Puffer zum Gerät
- `ioctl` führt gerätespezifische Kommandos aus
- `open` öffnet ein Gerät
- `release` gibt das Gerät frei

Die Gerätetreiberschnittstelle wurde in der neuen Klassenbibliothek java.devices implementiert.

3.3. JAVA-Klassenbibliothek

Aus der folgenden Tabelle gehen die vorgenommen Modifikationen an den Klassen-Bibliotheken hervor.

Tabelle 1. Für miniJava modifizierte JAVA-Klassenbibliothek

Klassenbibliothek	Bemerkung
java.awt	nicht implementiert
java.lang	• SecurityManager, Process, Compiler, Thread, ThreadGroup, ClassLoader nicht implementiert • StdIO mit eigener Stringverwaltung
java.io	• komplette Implementierung • zusätzliche Listenklasse
java.net	• Zweiteilung: – niedere Netzwerkfunktionen – hohen Netzwerkfunktionen • keine HOST-Datei und kein DNS-Service Direkte Angabe • Sockets als Java-Klasse nachgebildet
java.util	• Klassen für den Zugriff auf Dateisystem des Massenspeichers nicht implementiert
java.devices	NEU eingeführte Klassenbibliothek

3.4. Leistung und Ressourcenanforderungen des Prototyps

Zum entwickelten miniJava-Prototyp gehören, neben der beschriebenen JVM und der modifizierten und erweiterten Klassenbibliothek, die Entwicklungswerkzeuge und ein Applikationsserver. Als Entwicklungswerkzeuge wurde Visual C++ 4.2 verwendet. Für das Targetsystem wurde eine DOS-Portierung des GNU C/C++ Compilers Vers. 2.7.2.1 verwendet [2].
Die JVM wurde mit Visual C++ entwickelt und nachfolgend mit GNU C auf DOS portiert. Mit dem Benchmark „Sieb des Eratosthenes“ wurde ein Leistungsvergleich zwischen der Java- und der C-Version durchgeführt (s. Tabelle 2). Außer-

dem erfolgt ein Vergleich für zwei verschiedene Implementierungen der inneren Schleife der Execution-Engine (s. 3.1). Bei der Ausführung des Benchmarks entspricht eine Iteration einer vollständige Suche im Zahlenbereich 0 bis 8191.

Tabelle 2. Geschwindigkeitsvergleich Java vs. C. (Algorithmus: Sieb der Eratosthenes)

	Compiler	Iterationen
C Version	DJGPP gcc v2.7.2.1	4732
	MS VC++ v4.2	4753

	Compiler	Iterationen
Java Version :	DJGPP gcc v2.7.2.1	101
inner loop enthält Switch-Case-Konstrukt	MS VC++ v4.2	133

	Compiler	Iterationen
Java Version :	DJGPP gcc v2.7.2.1	68
inner loop enthält Array aus Funktionszeigern	MS VC++ v4.2	88

Speicherplatzbedarf

Der Speicherplatzbedarf wird in Tabelle 3 angegeben.

Tabelle 3. Speicherplatzbedarf

	Speicher-anforderung	Speichertyp
Laufzeitsystem ohne interne Klassen	368 KByte	ROM
Laufzeitsystem mit allen Systemklassen	531 KByte	ROM
Laufzeitsystem mit Applikation und benötigten Systemklassen (optimiert)	468 KByte	ROM
Arbeitsspeicherbedarf		
Initialisierung	280 KByte	RAM
Applikation	40 KByte	RAM

Der Arbeitsspeicherbedarf ist bei allen drei Varianten identisch, da nur die benötigten Klassen geladen und initialisiert werden.
Die Größe der kompletten Laufzeitumgebung läßt sich etwa wie folgt auf die Einzelmodule aufteilen:

- Laufzeitumgebung 180 KByte
- native Klassenbibliothek 70 KByte
- Netzwerkschnittstelle 45 KByte
- Gerätetreiberschnittstelle 20 KByte

Werden ein oder mehrere Module nicht benötigt, können diese entfernt werden. Die native Klassenbibliothek ist notwendig. Es ist aber möglich, Teile aus dieser

Bibliothek selbst zu entfernen, um damit die Größe der Laufzeitumgebung zu verringern.

4 JVM für kleine Mikrocontroller-basierende Systeme

Eine besondere Herausforderung bei der Implementierung von JVMs auf ressourcen-kritischen Systemen ist neben der geringen Taktfrequenz u.a. auch der extrem begrenzte Speicher. Derartige Limitierungen gestatten es nicht, die komplette JVM effizient auf dem Mikrocontroller zu realisieren. Eine Lösung bietet das Aufsplitten der JVM in einen offline-Anteil (vorbereitende Arbeiten), der auf einem leistungsfähigen Rechner abläuft und einen online Anteil (ausführende Arbeiten) der zur Laufzeit auf dem Mikrocontroller verbleibt.
Das Prinzip der Aufplittung zeigt Abbildung 2.

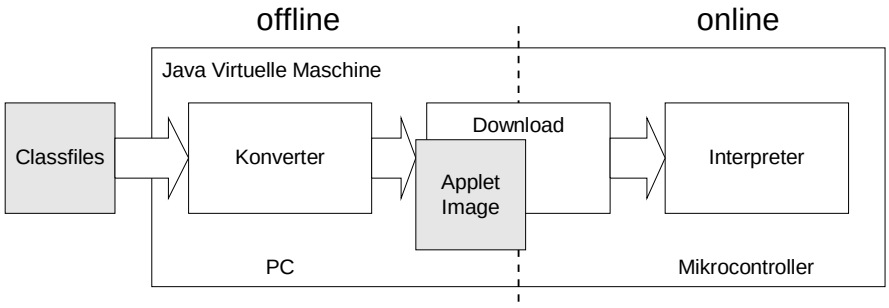

Abbildung 2. Die geteilte Java Virtuelle Maschine

Der in der JVM integrierte Bytecode-Verifier ist dem Konverter zugeordnet und steht durch die Aufteilung der JVM nicht mehr für eine Laufzeitüberprüfung des Bytecodes im Mikrocontoller zur Verfügung. Eine formale Verifikation der bei einem Methodenaufruf übergebenen Objekte kann aus Performancegründen nicht durchgeführt werden.
Das Aufteilen stellt somit einen Schwachpunkt im JAVA-Sicherheitskonzept dar. Durch den Einsatz von Lookup-Tables (LUT) kann zumindest sichergestellt werden, daß bei Auftreten illegaler Befehlscodes im Bytecode des Applets ein deterministischer Ausstieg gewährleistet werden kann.
Die konkrete Implementierung der JVM (smartJava) auf einen 8051-Mikroprozessor wurde in Standard-C geschrieben und basiert auf der JavaCard 2.0-Spezifikation [24]. Bei der Realisierung wurde statt der üblichen großen „case”-Schleife (s.o.) eine Modularität verwendet, wie sie bei der Beschreibung digitaler Systeme verwendet wird. Die Aufteilung der JVM erfolgt zu Lasten der Performance. Anstatt die Abarbeitung direkt nach Identifikation des Bytecodes durchzuführen, erfolgt beispielsweise ein Funktionsaufruf in der Form: ALU (opcode, parameter1, parameter2). Dadurch sind alle arithmetischen und

logischen Befehle leicht zu identifizieren und nicht über den gesamten Sourcecode verteilt.
Die so realisierte JVM benötigt ca.14 KByte ROM, wobei durch die zusätzliche Modularität ein ROM-Mehrbedarf von ca. 3 KByte entsteht.

5 Smart-Card-Java-Prozessor

Die Performance einer JVM kann signifikant verbessert werden, indem die Leistung der Basishardware gesteigert wird. Es macht allerdings wenig Sinn, für jede noch so kleine Applikation den jeweils leistungsfähigsten Rechner einzusetzen. Eine wesentlich größere Leistungssteigerung läßt sich erreichen, wenn der Prozessor JAVA-Bytecode direkt ausführen kann, da dann auf den Schritt der Interpretation verzichtet werden kann.
Bei der Implementierung sind verschiedene Varianten möglich:

1. Unterstützende Hardware
   Bei der Neuentwicklung eines Prozessors wird die interne Struktur soweit wie möglich für die effiziente Abarbeitung von Java-Bytecode optimiert (z.B. PSC1000 [18])
2. Coprozessor
   Ein Coprozessor, übernimmt die Abarbeitung des Java-Bytecodes.
3. Java Silicon Machine (JSM)
   Die JSM ist ein echter Prozessor, der ausschließlich für die Abarbeitung von Java-Bytecode entwickelt worden ist (z.B. picoJava-I, picoJava-II, microJava-Prozessor [25, 26, 27])

Direkte Zugriffe auf Peripherie werden in JAVA durch Aufruf spezialisierter Systemfunktionsrufe (API-Calls) oder Nutzung spezieller Spracherweiterungen realisiert (s. auch 3.2.2.) Für den eigentlichen Zugriff wird die JAVA-Ebene verlassen und auf die Ebene der Maschinensprache des Prozessors gewechselt. Im Befehlssatz eines Standardprozessors sind entsprechende Befehle vorhanden, die den direkten Zugriff auf die Peripherie gestatten (Abbildung 3).
Während die beiden ersten Varianten von der JAVA-E/A-Zugriffs-Problematik unberührt bleiben, müssen für den HW-Zugriff unter JAVA bei der dritten Lösung aufgrund einer fehlenden Hierarchiestufe neue Möglichkeiten geschaffen werden. Eine Reihe von Lösungen sind bereits bekannt, aufgrund des bis jetzt noch fehlenden Standards aber noch proprietär. Zur Realisierung wird der für JAVA-Bytecode zur Verfügung stehende Befehlsraum mit neuen Befehlen erweitert. So wurden z.B. beim PicoJava-II 50% mehr Befehle implementiert, als tatsächlich in der JAVA-Spezifikation vorgesehen ist. Diese verborgenen (hidden) Befehlscodes können nicht von einem normalen JAVA-Compiler erzeugt werden,

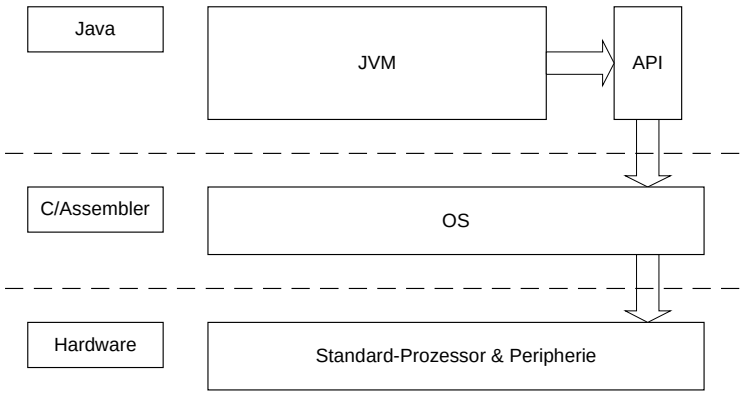

Abbildung 3. Hardware-Zugriff unter JAVA

d.h. auch das Bitmuster der Befehlscodes kann nur in trusted-libs vorkommen. Sobald Implementierungsdetails veröffentlicht werden, können allerdings sog. TROJANER-Programme diese Sperre umgehen und ggf. selbst auf die Hardware zugreifen. Im Bereich der eingebetteten Systeme (s.o.) kann dieses Verhalten u.U. sogar gewollt sein, führt aber auf Smart-Cards zu akuten Sicherheitsproblemen (Ausspähen von PINs).

Unsere Untersuchungen zur Konzeption einer JSM [1] haben gezeigt, daß durch Erweiterung der API-Funktionalitäten, letztlich nur die Befehle IO-Read und IO-Write zusätzlich zu implementieren sind. Alle darüber hinaus zusätzlich vorhandenen Befehlscodes beschleunigen zwar die Abarbeitung, verletzen aber die Sicherheitsidee von JAVA. Eine Möglichkeit, dieses Sicherheitsproblem für Smart-Cards zu lösen, besteht darin, den Aufrufort der Bytecodes aufzuzeichnen. Während Applikations-Applets im reprogrammierbaren Speicher liegen, kann das Java-Card-Runtime-Enviroment (JCRE) wahlweise im ROM oder bei gewünschter Update-Fähigkeit in einem gesonderten EEPROM untergebracht sein. Durch physikalisch unterschiedliche Adressen wird es möglich, den Aufrufort der Bytecodes aufzuzeichnen. Dadurch kann, im Sinne eines minimalen Online-Verifiers, beim Auftreten der neuen Bytecodes festgestellt werden, von wo der Aufruf erfolgte und ggf. die Ausführung verweigert werden [19].

Soweit Implementierungsdetails bekannt sind, basieren alle vorgestellten JSMs auf einer Microcode-Plattform, d.h. die Funktionalität wird über Tabellen gesteuert. Genau diese lassen sich aber mittels Mikroskop sehr leicht auf dem Chip identifizieren und bei Vorhandensein entsprechender Gerätschaften auch modifizieren.

6 Ausblick und Zusammenfassung

Wir haben gezeigt, daß durch Beschränkungen der vollständigen JVM, sich JAVA-Laufzeitumgebungen für den eingebetteten und für den Smart-Card-Bereich realisieren lassen.
Gerade das bis Februar 1999 nicht definierte Austauschformat der Applets innerhalb der geteilten Maschine, machte eine eigene Definition erforderlich. Diese hat Einfluß auf den Gesamtentwurf des Prozessors. Für standardgerechte Prozessorimplementierung muß das Austauschformat angepaßt werden.
Die Änderungen beim Smart-Card-Prozessor fallen nicht allzu gravierend aus, da diesem eine Microcode-Implementierung zugrunde liegt. Somit sind Änderungen lediglich in der Steuertabelle vorzunehmen.
Nach der vollständigen Validierung der VHDL-Beschreibung ist eine Probesynthese auf einem APTIX System Explorer MP3C mit vier XILINX XCV1000-FPGA geplant.

Literatur

[1] Bannow, N.: Konzeption eines Java-Prozessors. Studienarbeit, Institut für Mikroelektronik und Datentechnik, Universität Rostock, 1999

[2] Delorie, D. J.: A free 32-bit development system for DOS. http://www.delorie.com/

[3] Golatowski, F., Timmermann, D.: Using Hartstone Uniprocessor Benchmark in a Real-Time Systems Course, Proceedings of the Third IEEE Real-Time Systems Education Workshop, Poznan, Polen, S. 77-84, 1998

[4] Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley, 1996

[5] Gosling, J., McGilton, H.:The Java Language Environment. Addison-Wesley, 1996

[6] Hangal, S., O'Connor, M.: Performance Analysis and Validation of the picoJava Processor. IEEE Micro, 19 (3), S. 66-72, 1999

[7] Hagendorf, O.: Entwurf und Prototypimplementation einer Java-Laufzeitumgebung für kleine eingebettete Systeme, Diplomarbeit, Institut für Mikroelektronik und Datentechnik, Universität Rostock, 1997

[8] Hewlett Packard, http://www.hpconnect.com/embeddedvm/, 1998

[9] Huelsbergen, L.:http://cm.bell-labs.com/who/lorenz/,1997

[10] Esmertec AG, Jbed Whitepaper: Component Software and Real-Time Computing, http://www.jbed.com/, Zürich, 1999

[11] Kraudelt, R.: Entwicklung und Implementierung einer JAVA virtuellen Maschine (JVM) für den Einsatz in besonders ressourcenkritischen Systemen (Smartcards). Diplomarbeit, Institut für Mikroelektronik und Datentechnik, Universität Rostock, 1999

[12] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Bonn, 1996

[13] McDirmid, S.: Ghost Machine: A Distributed Virtual Machine Architecture for Mobile Platforms. http://www.cs.utah.edu/~mcdirmid/ghost/, 1998

[14] Meyer, J., Downing, T.: Java Virtual Machine. O'Reilly, Sebastopol, 1997

[15] Nilsen, K.: Java for Real-Time. Real-Time Systems Journal, S. 197-205, 1996

[16] Newmonics Inc.: Discussions on Real-time Java. http://www.newmonics.com/WebRoot/technologies/java.html, 1996

[17] NSI Corp: JSCP-Software Co-Processor for Java. Or-Yehuda, Israel, http://www.nsicom.com, 1998

[18] Patriot Scientific: Java Processor PSC1000. elektronik industrie, H. 2, S. 51.f, 1998
[19] Ploog, H., Rachui, T., Timmermann, D.: Design Issues in the development of a JAVA-processor for small embedded applications, ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA´99, Monterey, 1999
[20] Rankl, W., Effing, W.: Handbuch der Chipkarten: Aufbau - Funktionsweise - Einsatz von Smart-Cards. Hanser, München, Wien, 1996
[21] Real Time Java Experts Group, http://www.rtj.org
[22] Sun Microsystems: http://java.sun.com/aboutJava/communityprocess/index.html
[23] Sun Microsystems: The Java™ Language: An Overview. 1995
[24] Sun Microsystems: JavaCard 2.0 Language Subset and Virtual Machine, Specification. http://www.javasoft.com/products/javacard, 1997
[25] Sun Microsystems: picoJava-I Microprocessor Core Architecture. Datenblatt, http://www.sun.com/microelectronics/picoJava/, 1998
[26] Sun Microsystems: picoJava-II. Datenblatt, http://www.sun.com/microelectronics/datasheets/picoJava-II/, 1998
[27] Sun Microsystems: microJavaTM-701 Processor Evaluation Platform, http://www.sun.com/microelectronics/microJava-701
[28] O'Connor, J., Tremblay, M.: picoJava-I: The Java Virtual Machine in Hardware. IEEE Micro, 17 (2), S.45-53, 1997
[29] UNIX System Laboratories Inc.: Device Driver Reference UNIX SVR 4.2. Prentice Hall International Inc., New Jersey, 1992
[30] Venners, B.: Inside the JAVA Virtual Machine. McGRAW-HILL, 1998
[31] Vijaykrishnan, N., Ranganathan, N., Gadekarla, R.: Object-Oriented Architectural Support for a Java Processor. 12th European Conference on Object-Oriented Programming, Brüssel, Belgien, 1998
[32] Wilkinson, T.: Kaffe. http://www.kaffe.org, 1997