家電向け仮想マシンアーキテクチャ

春名修介[†] 金丸智一[†] 吉田力[†]
和氣裕之[†] 富永宣輝[†]

家電機器に対するアプリケーション配信を実現するために，Javaは有望なアーキテクチャとみなされている．しかし従来のJavaは，仮想マシンを搭載する機器に多大なメモリ量と高性能のCPUを必要とするため，低コスト・低消費電力要求が厳しい家電機器に適用するのは困難であった．本論文では，少ない資源しか持たない機器上でもJavaアプリケーションを実行できるようにした，家電向け仮想マシンアーキテクチャについて述べる．家電機器に対するアプリケーション配信は，インターネットのような開かれたネットワークではなく，特定のアプリケーション配信元と家電機器のみが連結される閉じられたネットワークにより行われる．本アーキテクチャはこの特性に着目し，静的に実行できる検証処理やリンク処理を仮想マシンから分離し，これらの処理を機器外で行うことにより，仮想マシンの処理の負荷を軽減した．その結果，従来のJavaVMと比較して，仮想マシンの実装のために必要なメモリ量を約十分の一に，配信される実行形式のサイズを従来の約40%に縮小したアーキテクチャを実現した．

Virtual Machine Architecture for Consumer Electronics

Shusuke Haruna,[†] Tomokazu Kanamaru,[†] Chikara Yoshida,[†]
Hiroyuki Waki[†] and Nobuki Tominaga[†]

Java is considered to be a promising architecture for the distribution of programs to consumer electric appliances. However, as conventional Java requires a large amount of memory and a highly efficient CPU in appliances running virtual machines, it was difficult to apply it to consumer electric appliances where low cost and low power consumption are a prime concern. This paper describes a virtual machine architecture for consumer electric appliances which allows Java applications to run even on appliances with limited resources. Distribution of applications to consumer electric appliances is achieved not by an open network like the Internet, but by a closed network in which only specific application distributors are connected to the appliances. This architecture focuses on these characteristics, and separates verification processing and link processing, which can be performed statically, from the virtual machine. By performing this processing outside the device, it decreases the processing load on the virtual machine. As a result, we were able to obtain an architecture which, in comparison to conventional JavaVM, requires only about 1/10 of the memory, and wherein the size of the distributed executable program is reduced to about 40% of its previous value.

1. はじめに

近年，ネットワークを利用した新サービス提供などの目的から，家電機器に対するアプリケーション配信機能搭載の要求が高まっている．

Java仮想マシン（JavaVM）[1]は，機器のハードウェアやOSに依存せずにアプリケーションを動作させることができるため，家電機器に対するプログラム配信を実現するのに有望なアーキテクチャとみなされている[2]．また，開発言語としてのJava言語[3]は，簡素で理解容易なネットワーク対応のオブジェクト指向言語として，広く普及している．

本稿は，従来のJava言語とその開発環境をそのまま利用でき，かつ，少ない資源しか持たない家電機器上でもJavaアプリケーションを実行できるようにした，家電向け仮想マシン（Virtual Machine for Consumer Electronics，以下CVMと記す）アーキテクチャの設計と実装について述べたものである．

本稿では以下，2章でJavaVMを一般の家電機器に適用する場合の問題点について述べ，3章で問題点に対するアプローチ，特にネットワークの特性の考察に基いて行った，アーキテクチャの設計について説明する．4〜6章でCVMアーキテクチャの各構成要素

† 松下電器産業株式会社 マルチメディア開発センター
Multimedia Development Center, Matsushita Electric Industrial Co.,Ltd.

について解説し，7章で，実装したCVMアーキテクチャに関しての評価を述べる．

2. JavaVMの問題点

家電機器にはコンピュータ分野に比べて，以下のようなハードウェア制約が存在する．

- 低コスト化が重視される．必要最小限の性能のマイコンと小容量のメモリで機器の機能を実現する必要がある．
- 低消費電力要求がある．特に携帯電話などの機器では，消費電力の低減は必須であり，動作周波数は可能な限り低くする必要がある．

このように家電機器には省資源性が強く要求される．現在，全世界で民生機器用に出荷されている組み込み用マイコンの個数のうち，9割以上を占めるのが4bit〜16bitのMCU（Micro Controller Unit）であり，これを搭載する，携帯電話や，冷蔵庫，電子レンジ，エアコンなどに代表される家庭電化製品は，**表1**に示される通り，数十kByte前後のメモリしか搭載しないものがいまだ主流である[4]．

通常のJavaVMは，**表2**に示される通り，その動作のために高性能のCPUと多大なメモリ量を必要とする．よってJavaVMを家電機器に搭載するためには，メモリ量などの使用する資源を削減することが不可欠である．

従来のJavaVMのメモリ削減のアプローチは，実行環境のうちJavaVMの仕様には変更を加えず，実行環境に搭載するクラスライブラリをサブセット化することによって，必要メモリ量を削減するものがほとんどであった[5]．

その代表的なものがSun Microsystems社のPersonalJava（PJava）およびEmbeddedJava（EJava）である☆1．しかし表2からも判る通り，PJavaやEJavaにおいて要求される資源量は，依然として一般の家電機器には過大なものである☆2．

表1 MCUと共に搭載される典型的なメモリ容量
Table 1 Typical memory size embedded with MCU.

MCUのタイプ	メモリ	8bit	16bit	32bit
標準	RAM	0.5kB	1kB	2kB
	ROM	48kB	64kB	96kB
大型	RAM	2kB	4kB	4kB
	ROM	64kB	128kB	192kB

表2 JavaVMの動作のために必要な資源量[6]
Table 2 Resources required for Java virtual machine.

Parameter	Java JDK（通常のJava）	Personal Java	Embedded Java
RAM size	4MBytes	1MBytes	512kBytes
ROM size	4-8MBbytes	2MBytes	512kBytes
CPU type	100MHz +	50MHz +	25MHz +

同様のアプローチを採用したものに，Hewlett Packard社のChaiVM☆3やアクセスのJV-Lite☆4などがあるが，いずれの取り組みも，使用メモリ量は通常で600kバイト以上，最小構成で230kバイト程度という，大部分の家電機器にとっては過大な値に留まっている．

また，分散処理の考え方をネットワークに適用しようとする試みとしてJini[7]があるが，Jiniも従来のJavaVMの使用を前提としているため，これを一般の家電機器のネットワークに適用するのは困難である．

メモリ量が数十kByte程度の家電機器上でJavaアプリケーションを動作させるには，クラスライブラリを削減する方法のみでは実現が難しく，JavaVMが行う処理自体を家電機器向けに見直す必要がある．

3. 本研究のアプローチ

JavaVMがその動作のために多くの資源を必要としているのは，従来のコンピュータ分野の流通ネットワークの特性に起因している．我々は家電機器の流通ネットワークを考察し，その特性に基づきJavaVMアーキテクチャを再設計することによって，一般の家電機器上でも動作可能な仮想マシンを実現した．

3.1 流通ネットワークの特性

ネットワークの特性の差異について説明するために，ここでは「開かれたネットワーク」と「閉じられたネットワーク」という2つのモデルを用いる．

開かれたネットワーク（図1）とは，インターネットのように，ネットワークに接続する者であれば誰でも，流通するアプリケーションをダウンロードできるが，アプリケーションの正当性の保証はないものを指す．

従来のJavaVMは，開かれたネットワークでの使用を前提としているため，アプリケーションが危険な

☆1 Sun Microsystems, http://www.javasoft.com/products/（1999年12月10日現在）

☆2 Sun Microsystems社は，PJavaやEJavaとは別に，メモリ制約システムに対応可能なKVMを1999年6月に発表しているが，詳細な仕様は決定していない．1999年11月現在，日米欧のメーカによるワーキンググループによって仕様策定中である．

☆3 ChaiVM: http://www.hp.com/emso/products/（1999年12月10日現在）

☆4 JV-Lite: http://www.access.co.jp/（1999年12月10日現在）

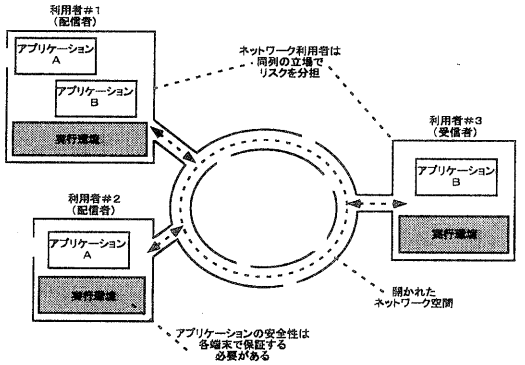


図 1 開かれたネットワーク
Fig. 1 Opened network for program distribution.

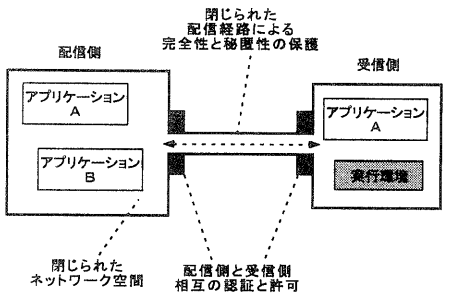


図 2 閉じられたネットワーク
Fig. 2 Closed network for program distribution.

動作をしないことを実行前に調べる検証処理を行う必要があった．この検証処理はフロー解析やシンボル参照の繰り返しなど多大なステップを要する処理である．

一方，閉じられたネットワーク（図 2）とは，以下の特長を持つものを指す．

- 配信側と受信側で相互認証を行うことによって，接続先が特定できる．
- 暗号化などの手段によって，配信経路が保護される．

家電機器は，高い信頼性を保持する必要があり，アプリケーション配信の際には，データの保護が必須である．すなわち，課金が正確に行われること，アプリケーション使用者の個人情報が漏洩しないこと，配信されるデータが改竄されたりしないこと，などが求められる．そのためには，家電機器を接続するネットワークは，電子署名や暗号化などの手段を用いて，接続先が正しいことや，盗聴が行われないことを保証する，閉じられたネットワークでなければならない[8] ☆．

3.2 CVM アーキテクチャの構成

閉じられたネットワークでは，接続先が常に信頼できるため，機器内での検証処理を省略し，検証済みのアプリケーションを配信することができる．つまり，この検証処理を機器外（アプリケーション配信側）で行うことが可能となる．

この家電ネットワークの特性に着目し，我々は，家電向け仮想マシン（CVM）アーキテクチャを設計した．

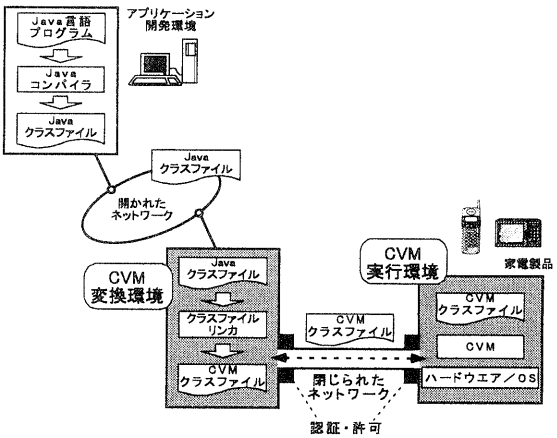


図 3 CVM アーキテクチャにおける開発から実行までの流れ
Fig. 3 Process from development to execution in CVM.

CVM アーキテクチャは，機器外の処理である CVM 変換環境と，機器内の処理である CVM 実行環境に分かれる．本アーキテクチャのアプリケーション開発から実行までの流れを図 3 に示す．

CVM のアプリケーション開発は，従来の Java のアプリケーション開発と同様に行われる．アプリケーションは Java 言語により記述され，Java クラスファイルへと変換される．

CVM 変換環境では，Java クラスファイルに対して，以下の処理を行う．

- アプリケーションの正当性を検証する．
- 複数の Java クラスファイルを結合し，1 つのファイルモジュールである，CVM クラスファイルを生成する．

これらの処理は，クラスファイルリンカと呼ばれる変換系によって行われる．CVM クラスファイルは閉じられたネットワークを通じて，CVM 変換環境から CVM 実行環境へ配信され，家電向け仮想マシン

☆ 現状の家電機器の接続を想定したネットワークとしては，
Bluetooth: http://www.bluetooth.com/
HomeRF: http://www.homerf.org/
VESA: http://www.vesa.org/
（1999 年 12 月 10 日現在）などが，代表的な例として挙げられる．

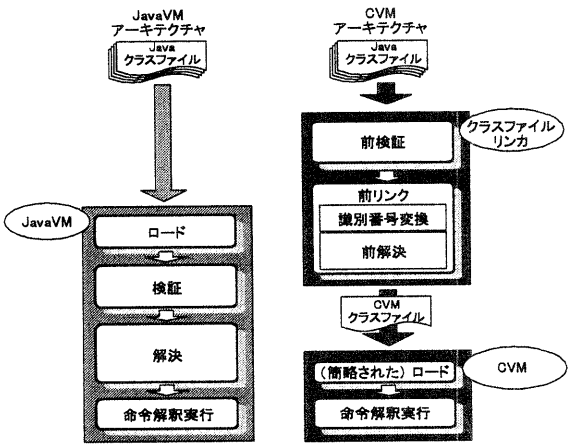


図 4 仮想マシンアーキテクチャにおける処理
Fig. 4 Activities in virtual machine.

(CVM) によって実行される.

CVM 変換環境は，携帯電話の基地局や，家庭内のサーバ機器などの，開かれたネットワークと閉じられたネットワークの接点に構築される．CVM 変換環境で正当性の検査を行うことで，例えば，携帯電話でのデータサービス [9] に見られるように，家電機器から開かれたネットワーク上の豊富なコンテンツを利用するシステムに，CVM アーキテクチャを適用することも可能である．

以降の章では，CVM アーキテクチャの各構成要素に関して説明する．クラスファイルリンカの章では，主に CVM の処理の負荷の軽減に関して，CVM クラスファイルの章では，クラスファイルのサイズの削減に関して，CVM の章では，主に家電機器上への実装に関連した事項について述べる．

4. クラスファイルリンカ

クラスファイルリンカは，JavaVM アーキテクチャでは動的に行われていた処理の一部を先に実行することにより，CVM の処理の負荷を軽減する．この処理は大きく，前検証処理と，前リンク処理に分けることができる．図 4 にクラスファイルリンカと CVM が行うおおまかな処理を示す.

4.1 前検証処理 (pre-verification)

JavaVM における検証とは，Java クラスファイルが正しいフォーマットを保持し，危険な動作を行うものではないことを検査する処理である．この正しいフォーマットは制約項目 (静的制約および構造体制約) によって規定される [1].

制約項目の大部分は，静的に検査することが可能であるが，従来の JavaVM はこの検査を全て動的に行っていた．しかし制約項目の中には，その検査のためにデータフロー解析等の多大なステップを必要とする，例えば以下のようなものがある.

- オペランドスタックはオーバーフローやアンダーフローを起こさない.
- 命令が指定する型のオペランドがスタックに積まれている.
- 命令が指定する型のオペランドがローカル変数に格納されている.

JavaVM における検証は，実装に必要なメモリ量を増大させるだけでなく，Java クラスファイルの命令解釈実行が開始されるまでの起動時間を長くする原因にもなっていた.

CVM アーキテクチャでは，クラスファイルリンカの前検証処理によって，静的に検査が可能な全ての制約項目を検査する．これにより CVM で検証を行う必要はなくなり，CVM の実装に必要なメモリ量の削減や，アプリケーションの起動時間の短縮が実現される.

4.2 前リンク処理 (pre-linking)

前リンク処理はさらに大きく，識別番号変換と，前解決処理に分けることができる.

4.2.1 識別情報変換

JavaVM では，使用されるクラス，メソッド，フィールドの識別を，UTF-8 の名前文字列によって行っている．これは Java クラスファイル中のコンスタントプールに記録されている.

CVM ではこれらの識別を，より小さなサイズの情報である，2 バイトの識別番号によって行う．クラスファイルリンカは，アプリケーションを構成している Java クラスファイルを解析し，使用される全てのクラス名，メソッド名，フィールド名に対して，一意に定まる識別番号を付加する.

4.2.2 前解決処理 (pre-resolution)

JavaVM が行う解決とは，一般に次の 2 つの処理を指す.

1. オペランドにシンボル参照を持つ命令の実行に先立って，その参照が正しいかどうかを検査する.
2. シンボル参照が間接的なもの，つまり何回もシンボル参照を繰り返すものである場合，これをより直接的な参照に置き換える.

JavaVM における解決の具体例を説明する．ある命令が，他のクラスファイルに定義されているメソッドへの参照を持つとする．この命令のオペランドはコンスタントプールへのシンボル参照である．この命令が解釈実行されると，命令はシンボル参照を追跡し，

UTF-8 文字列であるメソッド名を得る．次に同じメソッド名を記録している他の Java クラスファイルを探索することにより，メソッドの定義を探し出す．以上が 1 の処理である（図 5 (a)）．

1 の処理が終了した命令には，通常，続いて 2 の処理が行われる．JavaVM は，2 の処理を実現する仕組みとして，_quick 変換の機能を備える☆1)．これは，1 の処理を済ませた命令を，_quick 命令と呼ばれる，より直接的なシンボル参照を行う命令に置きかえる変換である（図 5 (b)）．これにより，同じシンボル参照に対して，何度も 1 の処理を行わずに済むようになる．

しかし，JavaVM では，1 と 2 の処理は全て命令の解釈実行時に動的に行われるものであった．

CVM アーキテクチャでは，クラスファイルリンカの前解決処理によって，1 と 2 の処理の双方ともを静的に行う．まず 1 の処理では，Java クラスファイルに記録された全ての命令について，それが保持するシンボル参照を追跡し，その参照の正当性を検査する．

2 の処理では，1 の処理を終えた命令を，Java バイトコードを拡張した独自の命令に置き換える．この命令は_quick 命令とは異なり，コンスタントプールを介さずに識別番号を直接オペランドに指定するものである（図 5 (c)）．CVM の命令拡張に関しては，6.2 節で述べる．

CVM は動的に 1 と 2 の処理を行う必要がないため，これらの処理に要するコストが除かれ，高速に命令の解釈実行を行うことができる．加えて，拡張命令は，コンスタントプールを介さない，_quick 命令よりもさらに直接的な参照を行うため，クラスやメソッドの参照の手順そのものも高速化される．

5. CVM クラスファイル

CVM クラスファイルのサイズは，同じアプリケーションの Java クラスファイルの総サイズと比較して，大幅な削減がなされている．

CVM クラスファイルのサイズが減少することにより，CVM 実行環境に必要なメモリ量を削減できるのと同時に，配信されるデータの伝送量そのものも少なくなる．よって転送速度があまり高くない家電ネットワーク環境に対しても，CVM アーキテクチャの適用は容易になる．

CVM クラスファイルのサイズの削減は，主に，コンスタントプールの構造の変更により達成される．両者

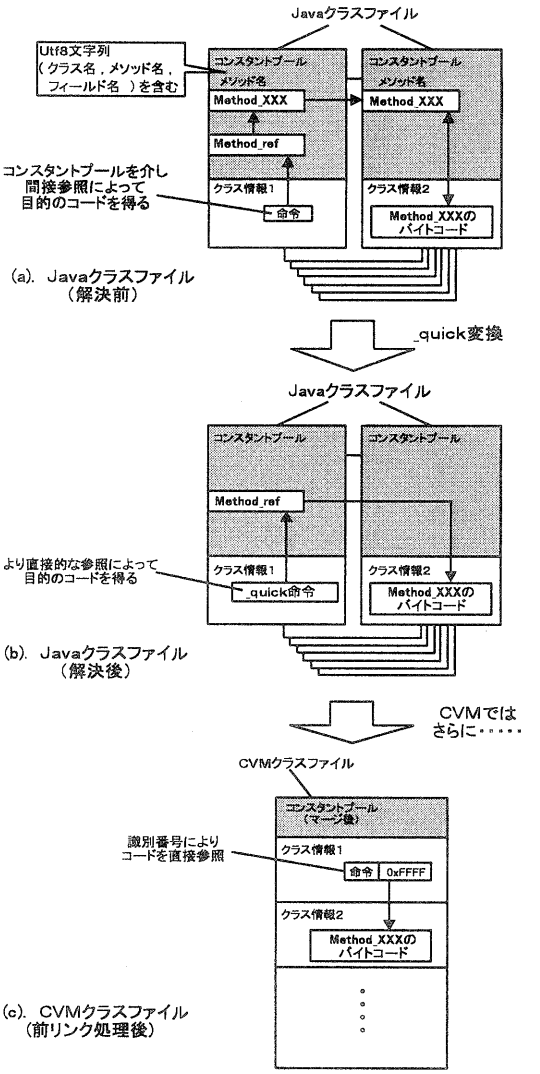


図 5 解決処理および前解決処理の例
Fig. 5 Example of resolution and pre-resolution.

を比較した場合，CVM クラスファイルのコンスタントプールの構造には以下の変更が加わっている（図 6）．

1. 識別情報（UTF-8 の名前文字列）を記録しているエントリをコンスタントプールから削除した．
2. クラス毎に存在していた複数のコンスタントプールを 1 つにマージした．
3. エントリの種類を示すタグを削除し，固定長サイズと可変長サイズのエントリを種類毎に分類してテーブル化した．

特に 1 の点は，CVM クラスファイルのサイズ削減に大きな効果をもたらす．これは 4.2 節で解説した識別情報変換に関連している．

☆ JavaVM の仕様では，_quick 変換は必須の機能ではないが，アプリケーションの実行の高速化のために実装している例が多い．

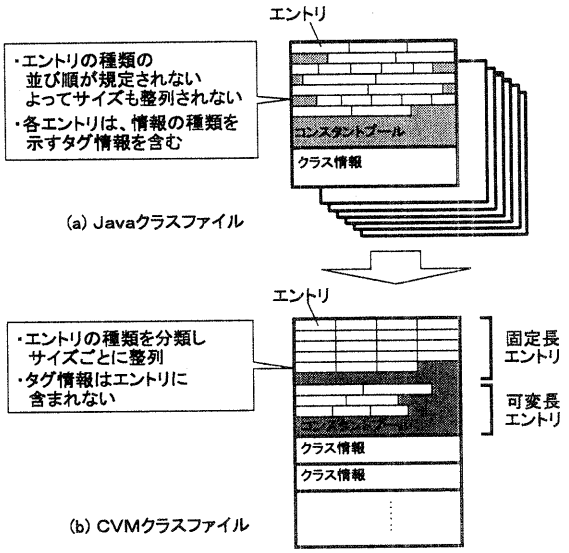


図 6 コンスタントプールの構造
Fig. 6 Structure of constant pool in class file.

ここで識別情報に関して，さらに説明を加えておく．Java クラスファイルが識別情報を名前文字列として保持する理由は，これが開かれたネットワークの環境の中でも，一意に定まる識別情報として使用可能なためである．名前文字列は 1 エントリの記録に，平均して十数バイトのサイズを必要とする識別情報であるため，この特徴は Java クラスファイル全体のサイズを肥大させる原因にもなっていた．

しかし，CVM クラスファイルは，閉じられたネットワークの中で流通されるファイルである，このため識別情報は，クラスファイルリンカと CVM の間でのみ一意に定まるものを保持していればよい．クラスファイルリンカによって識別情報変換が行われた後は，コンスタントプールに名前文字列を記録する必要はなくなるため，CVM クラスファイルからこれを削除できる．

CVM クラスファイルのコンスタントプールの構造は，先に挙げた 3 点の変更により，Java クラスファイルのそれよりも簡単な構造になっている．このことは CVM のロード処理を単純化することにも貢献している．Java クラスファイルのコンスタントプールのエントリ構造や，ロード処理の詳細に関しては文献 [1),10)] を参照されたい．

6. 家電向け仮想マシン（CVM）

6.1 実　　装

CVM を構成する基本部品，すなわちオペランドスタック，スタック，ヒープ領域などの構成は，従来の JavaVM のそれと変わるところはない．

しかし，家電機器はシステム構成のバリエーションが広く，CVM を搭載する機器の種類によっては，使用しない機能や，使用資源の効率の面から実装しないほうが良い機能が存在する．CVM は以下の機能の実装・非実装の選択を，コンフィグレーションによって指定可能とすることで，家電機器全般に柔軟に対応した実行環境を提供している．

- 基本型（float, long, double）
- 多次元配列
- ガーベジコレクション（GC）
- スレッド

上記の中でも，ガーベジコレクション（GC）の実装は重要な項目である．家電機器ではしばしば対話性や高いリアルタイム性が求められ，GC によるアプリケーションの解釈実行の中断は，これを妨げるものとして問題とされている [11)]．

我々の実装では，GC が中断可能であり，他の処理と平行して時分割で進めることができるマークアンドスィープ GC と，比較的簡単なメモリ構成で実現が可能な保守的 GC [12)] の 2 種類のガーベジコレクションを実装し，これらを選択可能とした．

6.2 命令セット

CVM の命令セットは，JavaVM の命令セットを基にして設計されているが，いくつかの命令には拡張が加えられている．主要なものをここで述べておく．

6.2.1 ロード・ストア命令

家電機器向けのアプリケーションには，メモリ使用の効率の問題から 1 バイトや 2 バイトサイズの変数が頻繁に用いられる．従来の JavaVM はデータに対し 4 バイトサイズ単位でしかメモリアクセスができなかったが，上記の特性を有効に利用するため，CVM では他のサイズのデータもメモリアクセスできるように命令を拡張した．

これによりオブジェクトのフィールドに値を格納する命令として，従来の putfield 命令（4 バイトアクセス）以外に，putfield_b（1 バイトアクセス），putfield_c（符号なし 2 バイトアクセス），putfield_s（符号付き 2 バイトアクセス），putfield_d（8 バイトアクセス）の 4 つの命令を，新たに命令セットに加えた．

同様に，getfield, putstatic, getstatic 命令に関しても，以下の命令を新たに命令セットに加えた．

getfield_b, getfield_c, getfield_s, getfield_d, putstatic_b, putstatic_c, putstatic_s, putstatic_d, getstatic_b, getstatic_c, getstatic_s,

getstatic_d

6.2.2 定数ロード命令

コンスタントプールの構造の変更に伴い，コンスタントプールに記録された定数をアクセスする命令にも変更を加えた．5章で述べた通り，従来のコンスタントプールのエントリに付加されていたタグ情報が，CVMクラスファイルのコンスタントプールのエントリでは削除されたため，定数をアクセスする際には，オペコードの種類によって定数の型を指定する．これにより，従来の定数ロード命令である ldc, ldc_w, ldc2_w 命令に代わって，以下の 10 種類の命令を新たに命令セットに追加した．いずれの命令も，オペランドにはCVMクラスファイルのコンスタントプールのインデクスを指定する．

ldc_s, ldc_i, ldc_f, ldc_w_s, ldc_w_i, ldc_w_f,
ldc2_l, ldc2_d, ldc2_w_l, ldc2_w_d

6.2.3 動的束縛

JavaVMは，動的束縛（dynamic binding）を実現する命令として invokevirtual や invokeinterface 命令を持つ．これらの命令のオペランドは，コンスタントプールに対するシンボル参照であり，呼び出されるメソッドの決定は，コンスタントプールを解析し，クラスの継承の階層を追跡することによってなされる．しかしこの解析は動的に行われるため，アプリケーションの解釈実行の速度を低下させる原因となっていた．

CVMアーキテクチャでは，クラスファイルリンカによってクラスの継承関係をあらかじめ解析し，クラスと呼び出されるメソッドの組み合わせに対するディスパッチ表を作成し，CVMクラスファイルに記録する．invokevirtual, invokeinterface 命令はそれぞれ，オペランドによりこのディスパッチ表を参照するように，変更を加えた．呼び出されるメソッドの決定のために要するコストは，従来のコンスタントプールの探索によるものから，ディスパッチ表に対する参照のそれに減らすことができるため [13]，より高速に動的束縛を実現できる．

6.2.4 その他の拡張

4.2.2節で述べた通り，従来の Java における，オペランドにコンスタントプールのエントリを指定する命令の大部分は，CVMでは，参照するクラスやメソッドの識別番号を直接オペランドに指定する，同名の命令に変更された．具体的には，以下の命令がそれに該当している．

new, anewarray, multinewarray, checkcast,
instanceof, invokestatic, invokespecial

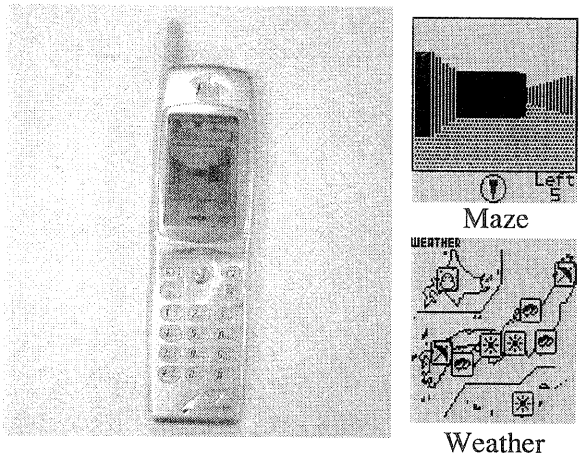


図7 CVMの携帯電話への適用例
Fig. 7 Implementation of CVM on celluler phone.

7. 評価

本研究の応用検討の一環として，携帯電話試作機（組み込み用 16bit マイコン搭載，動作クロック 3MHz）に CVM を搭載し，Java アプリケーションを実行させた．図7左に応用例の実機，図7右上および右下に，稼動しているアプリケーションの例を示す．この試作機上への実装例をもとに，CVMアーキテクチャの使用資源量の削減の効果を，使用メモリ量，クラスファイルのサイズ，アプリケーションの動作速度という3つの指標を用いて評価した．

7.1 使用メモリ量

CVMを機器に搭載するために使用するメモリ量を計測した．CVMはC言語によりコーディングされ，組み込みマイコン用クロスコンパイラを用いて実行コードが生成された．CVMの実行コードおよび基本クラスライブラリはROMに，その他の作業用データはRAMに配置される．CVMは，6.1節で述べた機能について，実装・非実装の選択が可能であるが，ここでは4種類の異なったコンフィグレーション（**表3**）を選び，それぞれのCVMについて計測を行った．

比較の対象として，他社製の JavaVM を同等の条件で動作させることは困難であるため，以下の2つのデータを併せて示した．

- EJava: 表2で示されている，EJavaにおける従来の JavaVM の使用メモリ量のデータ [6]．
- JVM0: 我々が独自に実装した JavaVM を，試作機上に搭載したときの使用メモリサイズ．基本クラスライブラリは各CVMに搭載するものと同じセットを使用している．

表 3 CVM のコンフィグレーション
Table 3 Configuration of CVM.

コンフィグレーション	基本型・多次元配列	GC	スレッド
CVM-full	○	○	○
CVM-GC	○	○	−
CVM-normal	○	−	−
CVM-minimum	−	−	−

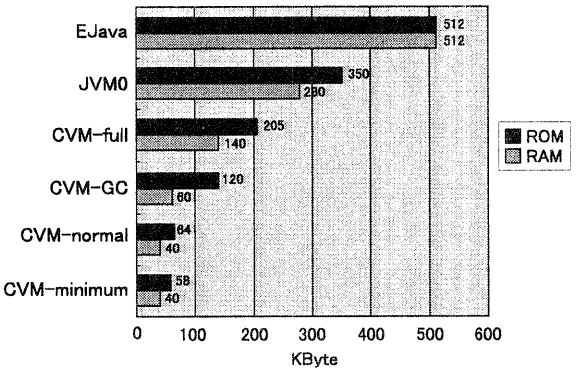


図 8 CVM の動作のために必要なメモリ量
Fig. 8 Memory size required for CVM.

本評価では，各 VM の使用メモリ量の差異を明確にするため，GC を備えた 3 種類の仮想マシン，JVM0, CVM-full, CVM-GC について，いずれも GC の方式を保守的 GC として評価データを得た．表 3 の GC の項の，○は保守的 GC を選択実装したこと，−は GC を実装していないことを意味している．

図 8 に計測結果を示す．JavaVM（EJava）で必要とされているメモリ量と比較して，最小構成時で約十分の一の使用メモリ量で CVM の搭載が可能なことが確認された．CVM は JavaVM とは異なり，検証・解決を行わないため，仮想マシンの実行コードだけではなく，作業用データのための必要メモリ量も少なくなる．この点が ROM と RAM の両方のサイズ削減に効果を現わしたものと考えられる．

CVM の各コンフィグレーションを比較すると，スレッドの機能を備えた場合には，CVM の必要メモリ量は最小構成時と比較して，約 3 倍に増大することが判る．CVM の機器への搭載の際には，アプリケーションが使用する機能と，使用可能な資源量に合わせて CVM のコンフィグレーションの指定を行うことが重要である．

今回，応用例として作成したアプリケーションは，いずれもスレッドを利用せず，GC を必要としない（実行中にヒープ領域に多量のオブジェクトを生成しない）ものである．試作機に搭載した CVM は，コンフィグレーションには CVM-normal を指定し，ROM64kB, RAM40kB という使用メモリ量でアプリケーションの実行を実現している．

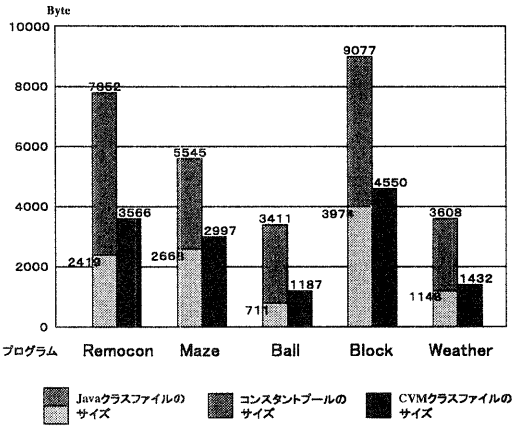


図 9 クラスファイルのサイズ
Fig. 9 Size of classfile.

7.2 クラスファイルのサイズ

アプリケーションを構成する Java クラスファイルの総サイズと，同じアプリケーションをクラスファイルリンカで結合した後の CVM クラスファイルのサイズとの比較を行った．評価用アプリケーションとしては，携帯電話応用向けに作成した，以下のものを選んだ．

- 現状の家庭内に浸透している AV 機器制御用リモコン（Remocon）
- 実行性能評価用の各種ゲーム（Maze, Block, Ball）
- データサービスを想定した天気予報表示（Weather）

図 9 に計測結果を示す．Java クラスファイルに関しては参考値として，そのサイズのうちコンスタントプールの領域が占める割合も示した．比較の結果，CVM クラスファイルのサイズは，クラスファイルリンカで変換される前の Java クラスファイルの総サイズと比較して，30〜50%程度，平均約 40%の削減がなされていることを確認した．全体として，Java クラスファイルの中でコンスタントプールの占めるサイズの割合が大きいアプリケーションほど，CVM クラスファイルのサイズ削減の効果は大きく現れている．

7.3 動作速度

試作機上でサンプルアプリケーションの実行時間を計測することによって，動作速度を評価した．CVM のコンフィグレーションは 7.1 節で説明した CVM-normal とした．比較の対象として，他社製の JavaVM を同等の条件で動作させることは困難であるため，我々が独自実装した JavaVM を用いた．

評価用のアプリケーションには，前節で述べたもの

表 4 アプリケーションの実行時間
Table 4 Execution time of sample application.

プログラム	JavaVM	CVM
Maze	15.76	12.38
Ball	67.69	22.08
Block	22.44	16.05

単位：Sec

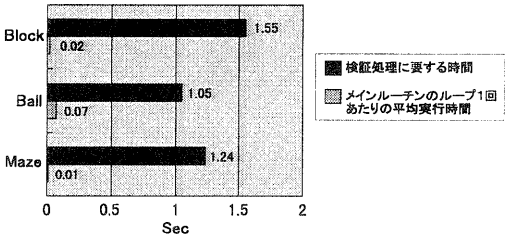


図 10 検証処理の実行時間
Fig. 10 Execution time of verification.

のうち，Maze, Ball, Block の 3 種類を用い，それぞれメインルーチンを 1000 回ループさせて終了した際の，仮想マシンの処理開始から終了までの実行時間を，試作機上で計測した．

計測結果を表 4 に示す．全体の処理時間として JavaVM と CVM とを比較した場合，少なくとも 1.2 倍，最大で 3.0 倍の処理速度向上が確認された．4.2.2 節で説明した通り，JavaVM と CVM それぞれの，命令解釈実行時のシンボル参照時にかかる処理コストの差が，この違いを生み出している．

CVM の前リンク処理の効果が最も大きく現れているのが Ball である．このアプリケーションは動的束縛を行うものであり，JavaVM は動的にクラスの継承関係を解析しなければならないため，実行時間は大幅に増大している．CVM は 6.2 節で述べた通り，ディスパッチ表を用いて動的束縛を行う命令拡張を行っているため，高速実行を実現している．

JavaVM における検証処理は，アプリケーションの起動時に一度しか行われないため，表 4 の計測結果だけから CVM の前検証処理の効果を評価することは難しい．この効果を明確にするために，JavaVM の処理の中から検証処理だけを抜き出し，各アプリケーションに対する検証処理の開始から終了までに要する時間を計測した．

計測結果を図 10 に示す．参考値として，各アプリケーションのメインルーチンのループ 1 回あたりの実行時間の平均を同じ表に示す．検証処理は，アプリケーションの命令解釈実行に要する時間と比較しても，大きな実行時間を費やす処理であることがわかる．CVM では前検証処理によって，実行時間からこのコストが除かれていることになる．

クラスファイルリンカによる前検証処理は主に，CVM のアプリケーションの起動時間の短縮に貢献し，対話性が重視される家電機器のアプリケーションにとっては効果が大きい．また，検証処理に要する時間はクラスファイルのサイズに応じて大きくなることから，大規模なアプリケーションほど前検証処理の効果は大きく現れるといえる．

8. おわりに

家電向け仮想マシン（CVM）は，家電機器へのアプリケーション配信に適したネットワークの特性を基に設計されたアーキテクチャである．検証処理やリンク処理を仮想マシンから分離し，機器外の変換環境でこれらを先に処理することにより，プログラムを実行する環境の省資源化を達成した．実装および評価では，CVM は従来の JavaVM と比較して，約十分の一のメモリ量しか使用せずに Java アプリケーションを実行できることや，配信される実行形式のサイズを従来の約 40 %に縮小できることを確認した．今後は CVM アーキテクチャの，さまざまなネットワーク家電機器への応用を検討していく予定である．

謝辞

本論文をまとめるにあたりご指導頂いた，早稲田大学理工学部深澤良彰教授，神戸大学工学部瀧和男教授，本研究の機会を与えて頂き日頃ご指導頂くマルチメディア開発センター櫛木好明所長，南方郁夫室長，ならびに開発や議論に参加頂いた，マルチメディア開発センターの諸氏に深く感謝いたします．

参考文献

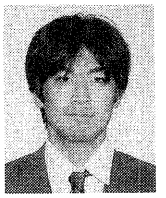
1) Lindholm, T. and Yellin, F.: *The Java^{TM} Virtual Machine Specification*, Addison Wesley (1997).
2) 枝洋樹, 五十嵐幸雄: 組み込み Java 独り立ち, 日経エレクトロニクス, Vol. 729, pp. 99–123 (1998).
3) Gosling, J., Joy, B. and Steele, G.: *The Java^{TM} Language Specification*, Addison Wesley (1997).
4) 中本幸一, 高田広章, 田丸喜一郎: 組み込みシステム技術の現状と動向, 情報処理, Vol. 38, No. 10, pp. 871–878 (1997).
5) 臼井和敏: 技術者のための最新 Java 技術入門, Interface 9 月号, Vol. 267, pp. 52–58 (1999).
6) Mulchandani, D.: Java for Embedded Systems, *IEEE Internet Computing, May, June*, pp. 30–39 (1998).
7) Arnold, K., O'sullivan, B., Scheifler, R. W.,

Waldo, J. and Wollrath, A.: *The Jini^{TM} Specification*, Addison Wesley (1999).
8) 野沢哲生: 新規配線なしの SOHO ワイヤリング, 日経コミュニケーション, Vol. 299, pp. 79–95 (1999).
9) 安井晴海, 高槻芳: 手のひらに乗るインターネット, 日経コミュニケーション, Vol. 289, pp. 98–117 (1999).
10) Meyer, J. and Downing, T.: *Java Virtual Machine*, O'Reilly&Associates (1998).
11) 村山敏清: 組み込み Java の世界, *Java Press*, Vol. 5, pp. 6–19 (1999).
12) Boehm, H. and Jackson, F.: Garbage Collection in an Uncooperative Environment, *Software Practice and Experience*, Vol. 18, No. 9, pp. 807–820 (1988).
13) Aigner, G. and Hölzle, U.: The Direct Cost of Virtual Function Calls in C++, *ECOOP'96 Conference Proceedings*, pp. 142–166 (1996).

(平成 11 年 10 月 15 日受付)
(平成 12 年 2 月 22 日採録)

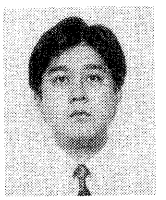
春名 修介 (正会員)
昭和 29 年生. 昭和 52 年神戸大学工学部電子工学科卒業. 同年松下電器産業 (株) 入社. 現在, マルチメディア開発センター勤務. マイクロコンピュータアーキテクチャ, コンパイラ・OS 等の基本ソフトウェア, ソフトウェア開発環境に関する研究開発に従事. 電子情報通信学会, ACM 各会員.

金丸 智一
昭和 47 年生. 平成 9 年早稲田大学大学院理工学研究科情報科学専攻修士課程修了. 同年松下電器産業 (株) 入社. 現在, マルチメディア開発センター勤務. プログラミング言語処理系の研究開発に従事.



吉田 力
昭和 47 年生. 平成 9 年京都大学大学院工学研究科情報工学専攻修士課程修了. 同年松下電器産業 (株) 入社. 現在, マルチメディア開発センター勤務. プログラミング言語処理系の研究開発に従事.

和氣 裕之
昭和 41 年生. 平成 2 年東京電機大学大学院理工学研究科数理学専攻修士課程修了. 同年松下電器産業 (株) 入社. 現在, マルチメディア開発センター勤務. プログラミング言語処理系の研究開発に従事.



富永 宣輝
昭和 38 年生. 昭和 63 年京都大学大学院工学研究科数理工学専攻修士課程修了. 同年松下電器産業 (株) 入社. 現在, マルチメディア開発センター勤務. コンピュータアーキテクチャ, 基本ソフトウェアおよびソフトウェア開発環境 (コンパイラ, OS, VM 等) の研究開発に従事.