社団法人　情報処理学会研究報告　　2008－MBL－44　(8)
IPSJ SIG Technical Report　　2008－UBI－17　(8)
2008／3／5

携帯端末向け Java 省メモリ実行環境の検討

岡田　英明　　清原　良三
三菱電機株式会社　情報技術総合研究所

筆者らは、携帯端末をユーザ自身の手で、より自由に安全にカスタマイズできるアドイン環境の基盤としての Java 実行環境の検討を行っている。アドイン環境実現のためには、現在はゲーム等の単一のアプリケーション実行用に用いられることが多い Java 実行環境に対して、複数アプリケーション実行機能の追加が必要である。マルチ VM 方式による複数 Java アプリケーション実行の実現は、シングル VM 方式に比べて独立性の点で有利であり、アドイン環境に適しているが、より多くのメモリ(RAM)を必要とする問題がある。本報告では、省メモリ対策を行ったマルチ VM 方式による複数 Java アプリケーション実行環境を提案し、性能評価の結果を示す。10 アプリケーション同時実行時の評価により、その有効性を確認した。

Small Footprint Java Runtime Environment for Mobile Devices

Hideaki Okada　　Ryozo Kiyohara
Information Technology R&D Center, Mitsubishi Electric Corp.

Add-in software environment, which enables users to customize their mobile phones by adding downloadable software, is expected to be available in the mobile phone market. We consider Java is the key technology to realize the add-in software environment. It is necessary to add multiple application execution function to Java runtime environment. Multi-VM method for this function is more robust than Single-VM method, but requires more memory. We propose small footprint Multi-VM method for multiple application execution function and confirm the validity by evaluation.

1. はじめに

様々な機能やセンサを内蔵し、インターネットへのアクセス機能を持つ携帯端末、中でも携帯電話に対し、ユーザ自身の手でソフトウェアを追加(アドイン)することにより、これらの機能を自由に利用できるようにカスタマイズしたいというユーザニーズがある。端末内部の情報に対するセキュリティを確保しつつ、このようなニーズに応える手段として、Java アプリケーションのアドイン機能がある。しかし、現在の Java アプリケーション実行環境では、同時にひとつのアプリケーションしか動作させられない、バックグラウンドで動作させられないなど、PC のソフトウェア環境と比べると多くの制限事項が存在する。

2007 年、SDK が公開された Android[1]は、このようなアプリケーションのアドインに適したプラットフォームとして期待されている。また、Android は、アドイン用のアプリケーションだけではなく、電話機能などの携帯電話に組み込まれている基本機能も同じ環境上のアプリケーションとして実現しているという特徴も有している。

筆者らは、Java 技術を利用した、ユーザによるカスタマイズ用ソフトウェアおよび組み込み用ソフトウェアの双方に対応可能なアドイン環境の実現を検討している。本環境においては、フォアグランド、バックグラウンド、常駐、などの形態で、複数のアプリケーションを同時に実行できる必要がある。

本論文では、アドイン環境における省メモリ対策を行った JavaVM の実現方式を提案する。10 アプリケーション同時実行時の評価によりその有効性を確認した。以降、2 章において、アドイン環境における JavaVM に求められる要件を定義する。アドイン環境では前述したように複数のアプリケーションの実行が可能であるが、3 章において、その基本実現方式の検討を行うとともに省メモリ対策の必要性を述べる。4 章において、前記要件の関連研究を紹介する。5 章において、アドイン環境に向けた省メモリ JavaVM の実現方式を提案し、6 章においてその評価を行う。最後にまとめと今後の課題を述べる。

2. アドイン環境における JavaVM の要件

アドイン環境実現のため、Java 実行環境に求められる JavaVM の要件を整理する。

実行速度

現状の携帯電話向け Java 実行環境と同等の性能を実現できることが必要である。

既に Java 実行の高速化のため、JIT(Just-In-Time), AOT(Ahead-Of-Time)等の Java バイトコードをネイティブコードにコンパイルする方式や、ハードウェアアクセラレータ[16]等が広く利用されている。したがって、これらの技術適用を妨げたり、性能を劣化させてはならない。

起動速度

現状の携帯電話向け Java 実行環境と同等以上の性能を実現できることが必要である。

Java アプリケーションが起動される回数が著しく増大すること、ゲーム等のアプリケーションだけでなく大小さまざまなアプリケーションが利用されることから、これまで以上に起動速度の向上が必要である。

メモリ使用量

RAM の使用量を最小化することが必要である。携帯電話は、ROM、RAM 容量ともに制約があり、これらリソースの節約が必要である。特に複数アプリケーションの同時実行によって RAM の使用量が増大する。同時実行可能なアプリケーション数を増大させ、また、バックグラウンドや常駐アプリケーションが動作したままで大容量メモリを利用するアプリケーションが実行できるためには、アプリケーションが必要とする以外のメモリ使用量を削減する必要がある。

独立性

アプリケーションリソースの保護・分離、セキュリティ権限の管理、耐障害性等の点において、各アプリケーション間の独立性を保障することが必要である。

現在の携帯電話では、ソフトウェア開発を効率化し、セキュリティを向上させるため、Linux、SymbianOS 等のプロセスによるアプリケーション間の保護・分離機能を利用している。さらに SymbianOS Platform Security[17]のようなセ

キュリティ権限の管理機能も開発されている。これらの仕組みを利用し、アプリケーション間の独立性を保つことが必要である。

本アドイン環境においては、端末内のすべてのソフトウェアを Java で実現することは想定していないため、リアルタイム性を要件とはしない。しかし、アプリケーション間の独立性が不十分であることにより、応答性を損なってはならない。

3. 複数 Java アプリケーション実行方式の検討

前記要件のアドイン環境について検討する前に、複数 Java アプリケーションの実行に関する基本方式について検討を行う。

3.1. シングル VM 方式とマルチ VM 方式

複数の Java アプリケーション実行を実現する場合に、JavaVM 実装の上で最も容易な方式は、1 つの Java アプリケーションは単一の OS プロセス、JavaVM 上で動作し、これらが複数動作する方式(以下、マルチ VM 方式)である。

一方、関連研究として、単一の OS プロセス上で動作する単一の JavaVM 上で複数の Java アプリケーションが動作する方式(以下、シングル VM 方式)[2]がある(図 1 参照)。

アプリケーション間の独立性を保つため、同じ

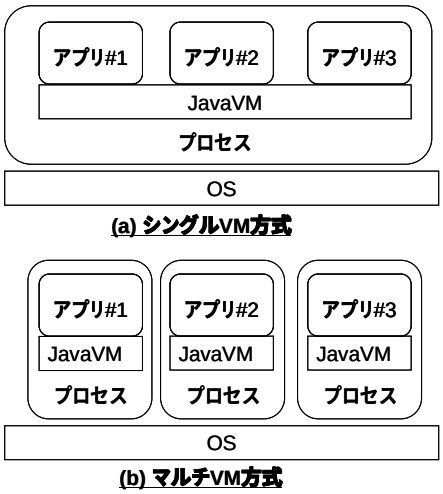

図 1 シングル VM 方式とマルチ VM 方式

VM 上で動作する別のアプリケーションのデータを参照することは出来ず、JavaVM 終了メソッドのような JavaVM 全体に影響を及ぼす動作はそのアプリケーションにのみ影響するように変更される[3]。したがって、各アプリケーションは、シングル VM 方式専用に記述する必要はない。

3.2. 複数 Java アプリケーション実行方式

シングル VM 方式の場合、Java 実行に必要な各種データが自然にアプリケーション間で共有されることにより、アドイン環境の要件において、様々な利点をもたらす。

- アプリケーションが使用するクラスには、アプリケーション固有のクラスと、端末にプレインストールされているクラス(以下、システムクラス)がある。システムクラスのうち、Java アプリケーション実行に必須であるクラスや、画面描画に関連するクラス、通信に関連するクラスは、各アプリケーションから共通的に利用される可能性が高く、メモリ使用量を削減できる。
- ネイティブコードに変換して実行速度を高速化する場合、上記と同様にコンパイルされたコードを共有できる可能性が高く、メモリ使用量を削減できる。
- Java 実行において、中間言語であるバイトコードの実行以外に負荷の高い処理として、VM 自身の初期化処理と、クラスのロード・ベリファイ処理がある。あるアプリケーションによってロードされたクラスが別のアプリケーションによって利用される場合、再度ロードする必要がない。したがって、実行速度、特に起動速度が向上する可能性が高い。

一方、独立性の点では問題がある。本アドイン環境では、ユーザカスタマイズ用のアプリケーションだけではなく、組み込み用のアプリケーショ

ンにも対応し、数多くの異なるセキュリティ要件、品質要件を持ったアプリケーションが動作する。それぞれの Java アプリケーションは、OS から見れば 1 つのプロセスであり、保護・分離、セキュリティ権限の管理、耐障害性等の点において独立性に劣る。

両方式の得失を比較すると、表 1 のようになる。シングル VM 方式は、マルチ VM 方式と比較して主にリソース使用の点で利点を有している。しかし、独立性において本質的に問題がある。

したがって、本検討においては、マルチ VM 方式を採用した上で、シングル VM との差、特にリソース使用量を最小化することに注力して、アドイン環境に適した JavaVM を検討する。

表 1 シングル VM 方式とマルチ VM 方式の比較まとめ

特性	シングル VM 方式	マルチ VM 方式
実行性能	通常 VM と同じ	通常 VM と同じ
起動性能(2 アプリケーション目以降)	優れている	通常 VM と同じ
メモリ使用量	優れている	通常 VM と同じ
独立性	劣る	通常 VM と同じ

4. 関連研究

前記要件についての関連研究、既存技術を紹介する。

4.1. 実行速度高速化技術

動的コンパイル方式

既に述べたようにネイティブコードにコンパイル・実行することで実行速度を向上させることが出来る。しかし、メモリ量、CPU 性能に制約のある携帯端末においては、コンパイルしたネイティブコードによるメモリ使用量の増大、コンパイル処理による応答性の劣化が問題となる。我々は、これらメモリ使用量、応答性の問題を解決した動的コンパイル方式を実現している[11]。

しかし、複数アプリケーション実行環境においては、それぞれの JavaVM において同じクラスのコンパイルコードが存在する可能性があり、端末全体としてメモリ利用量の無駄が発生する。

世代別 GC

一般に、メモリオブジェクトには一時的な使用目的等のため短時間で生成削除されるオブジェクト、比較的長く存在するオブジェクトの異なる特性を持つメモリオブジェクトが混在する。世代別 GC[18]、メモリ領域を新世代、旧世代に分離し、新世代領域に対しては頻繁に GC を行う一方、一定回数の GC を経て存在するオブジェクトを旧世代領域に移動し、旧世代領域に対してはあまり GC を行わないことにより、全体において GC 性能を向上させる。

マルチ VM 方式においては、それぞれの JavaVM に新世代領域が存在する。したがって、新世代領域に無駄の多い実装においては、端末全体としてメモリ利用量の無駄が発生する。

4.2. 起動高速化技術

既に述べたように、クラスのロード・ベリファイは負荷の大きい処理である。図 2 に JavaVM の内部構造を簡略化したものを示す。図中のクラスデータは、Java クラスファイルを JavaVM が読み込み、JavaVM が管理しやすい形に表現や配置を変更したものであり、中間言語であるバイトコードやシンボル情報等が含まれる。クラスのロード・ベリファイ処理は、クラスファイルを読み込んでその正当性を検証し、このクラスデータを作成することである。

既存の携帯端末向け JavaVM[6][7][8]では、ROMIZE 機能と呼ばれる機能が提供されている。ROMIZE 機能とは、JavaVM のビルド過程において、指定クラスに対して、クラスデータを作成し、それを JavaVM の実行ファイルに取り込む

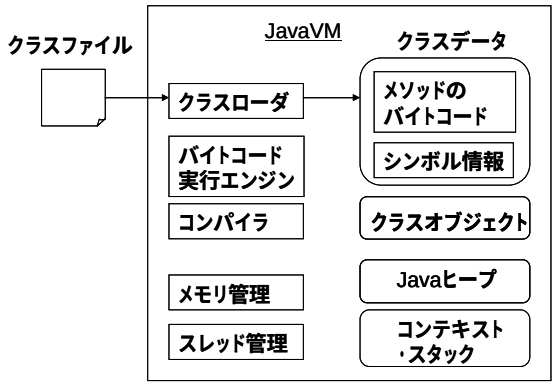

図 2 JavaVM の内部構造とデータ

機能である。指定されたすべてのクラスは、通常の JavaVM がロードする処理と同じようにロードされたものとして存在する。ただし、多くの場合、クラスの初期化は行われない。

ROMIZE 機能には、性能上の欠点が存在する。Java 言語の特徴として、シンボル解決を Java プログラムのコンパイル・リンク時に行わず、実行時に行う。例えば、getstatic という Java バイトコードは、指定するクラスの指定する静的フィールドの値を取得する。クラスおよびフィールドの指定は、文字列により行われ、実行時に名前解決を行う。

名前解決は負荷の大きい処理であり、また、一度行えば十分である。そこで、実行速度向上のため、getstatic というバイトコードを fast_getstatic という別のバイトコードに変更する quick 化という処理が行われる。この場合、fast_getstatic の引数として、解決したクラス静的フィールドのアドレスを間接的に指定する場所を指すインデックス値などを指定するようにバイトコードを変換する。なお、この書き換えを行う場合には、アクセス先のクラスが初期化済みである必要がある。

ROMIZE されたクラスデータを、書き込み不可メモリに配置した場合には、上記の quick 化を行うことが出来ないため、実行性能が劣化する。

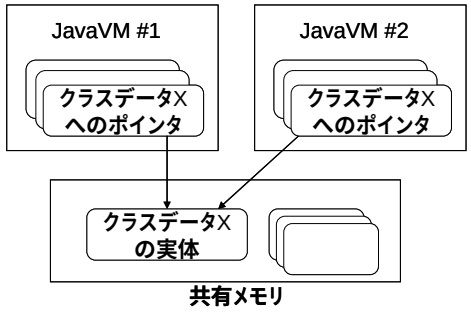

図 3 共有メモリによるクラス共有

4.3. メモリ使用量削減技術

クラス共有(ShMVM-B)

ShMVM-B[9]は、複数の JavaVM 間でクラスデータの共有を実現している。各 JavaVM が読み書き可能な共有メモリを確保し、その共有メモリ中にクラスをロードし、クラスデータを構築し、各 JavaVM から共有される(図 3 参照)。

共有することにより、メモリ使用量を節約できる。また、起動速度の向上にも効果がある。

しかし、異なるプロセス上にある JavaVM 間で排他制御しつつ共有データを管理する必要があるため、実行性能がやや劣化する。また、堅牢性に劣るため、独立性にも問題がある。

クラス共有(SLVM)

SLVM[10]は、クラスデータおよび関連するデータを DLL の形式に変換し、複数の JavaVM プロセスからメモリマップ機能により共有するものである。DLL のうち、クラスデータなどの各 JavaVM 間で完全に共有できるものは書き込み不可で共有し、JavaVM 毎にデータが異なる可能性のあるものは書き込み可で共有して書き込み発生時に copy-on-write により各 JavaVM にローカルなデータとする。本技術もメモリ使用量削減に効果がある他、起動性能向上に効果がある。

コンパイルコード共有(ShMVM-C)

ShMVM-C[9]は、ShMVM-B の考えを、JIT

等でコンパイルしたコードに適用したものである。

ShMVM-B と同様に、メモリ使用量や起動性能に効果がある一方、実行性能、独立性に問題がある。

5. 省メモリマルチ VM 方式

省メモリ対策を行ったマルチ VM 方式による、アドイン環境に適した JavaVM として、以下の機能を持つ JavaVM を提案する。

5.1. クラスデータの共有

前記 ROMIZE 機能により、共有可能なクラスデータは、Java 実行環境の実行ファイルのコード(テキスト)領域に含まれることになる。そこで、UNIX や Windows 等におけるコードの共有のようにプログラムコードをマップして共有し、クラスの共有を実現する。

クラス共有によるメモリ使用量削減効果と、ROMIZE による起動性能の向上が得られる。一方、ROMIZE による性能劣化の影響がある。

5.2. コンパイルコードの共有

クラスデータと同様にコンパイルコードの共有を行う。すなわち、AOT コンパイル方式により事前にコンパイルコードを生成しておき、OS のコード共有機能により共有する。

コンパイルコード共有によるメモリ使用量削減効果と、起動性能の向上が得られる。一方、ROMIZE による性能劣化と同じ理由により、コンパイルしたコードの実行性能がやや劣化する。

コンパイラの性能やネイティブの命令セットに依存するが、一般にバイトコードをネイティブコードに変換すると、5〜10 倍のサイズに増大する[11]。したがって、すべてのコードをコンパイルするのは望ましくなく、コンパイルによる性能向上が大きく、頻繁に利用されることが期待されるコードのみを選択して[12]、コンパイルする。

5.3. Java ヒープ伸縮

携帯端末向けの多くの JavaVM の実装においては、実装効率化のため、PC 用などの JavaVM 実装[5]と異なり、Java ヒープのサイズを JavaVM の起動時に決定する。単一アプリケーションの場合にはこのような実装でも問題は発生しないが、複数アプリケーションの同時実行の場合、各アプリケーション毎に必要な最大の Java ヒープサイズを指定しなければならず、Java ヒープによる無駄が生じる。

したがって、Java ヒープが伸縮する JavaVM 実装を行い、Java ヒープが縮んだ場合に OS に対して Java ヒープ用に使用していたメモリ領域を返却する。アプリケーションがフォアグラウンドからバックグラウンドに移行する、アイドル状態になる、などの契機においてヒープ縮小、メモリ返却を行うことにより、メモリ使用量削減に効果がある。

5.4. 世代別 GC における未使用領域の削減

携帯端末に搭載が開始された当初の Java 実行環境では Java ヒープは小さく、GC の方式として、主にマーク＆スイープ(およびコンパション)を採用していた[6]。しかし、携帯端末上の Java アプリケーションが普及し、Java ヒープサイズが大型化し、また、Java 実行環境が搭載される携帯端末のハードウェア仕様が向上するにつれ、世代別 GC などの GC 方式を採用した携帯端末向けの JavaVM 実装も登場している[7][8]。

これらの JavaVM における世代別 GC の実装においては、実装効率化のため、例えば 8MB の旧世代ヒープに対して、1MB の新世代ヒープを固定的に割り当てるなどの実装も見られる[7]。このような実装においては、各アプリケーション毎に存在する新世代ヒープにおける使用されていない領域が積算することによる無駄が生じる。

したがって、Java ヒープ内で新世代ヒープ・旧世代ヒープの境界が動的に変更する実装や、新

世代ヒープにおいても前節に述べたようなヒープの伸縮をサポートする JavaVM 実装を行い、メモリ使用量の削減を行う。

6. 評価

省メモリ対策の効果について評価を行う。

対策 1, 2 による効果

対策 1)クラス共有、および 2)コンパイルコード共有による効果について評価を行った結果を図 4 に示す。

評価プラットフォームとして、CLDC[13]ベースの MIDP[14]リファレンス実装を使用した。同プラットフォーム上で、ゲームやアドレス帳など異なるタイプの 10 個のアプリケーションの動作ログを採取し、同ログからクラスデータおよびコンパイルデータによるメモリ使用量を評価した。

評価対象は、a) シングル VM 方式、b) 省メモリ対策のないマルチ VM 方式、c)省メモリ対策のあるマルチ VM 方式、である。評価項目は、ロードされたシステムクラスが消費するメモリ容量、およびシステムクラスからコンパイルされたコンパイルコードが消費するメモリ容量とする。

なお、本評価においては、コンパイルはメソッド単位で行うものとし、10 回以上呼ばれた時点でコンパイルを行うものとした。AOT 方式により事前にコンパイルするメソッドは、5.2 節の考え方に沿って設定した。

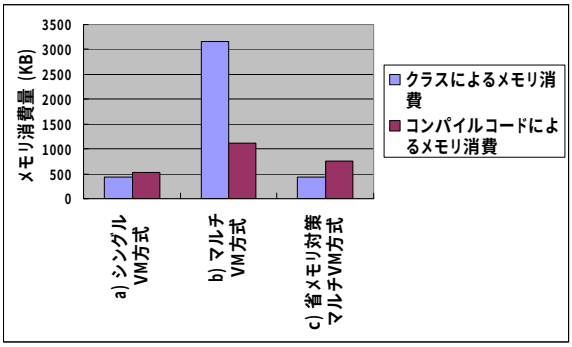

図 4 省メモリマルチ VM 方式の効果

なお、使用した MIDP プラットフォームのシステムクラスのサイズは Jar 形式において 2.2MB 程度である。CLDC より高機能な Java 仕様として CDC[15]があること、アドイン環境に必要、あるいは求められる機能の追加を考慮すると、このシステムクラスのサイズは、アドイン環境としては最低限度のものであろうと考える。

クラスによるメモリ消費に対しては大きな効果が見られる。これは、MIDP リファレンス実装において、アプリケーション起動時に多くの共通的に利用されるクラスがロードされるためである。

一方、コンパイルコードによるメモリ使用量については、効果は認められるが、例えば、アドイン環境を製品化する際に本対策を製品に適用してすべきか、といった命題に明確に判断できるような結果は得られなかった。よりアドイン環境に近い、Android 実装を利用した環境等における評価が必要である。

対策 3, 4 による効果

対策 3)Java ヒープ伸縮、4)世代別 GC 未使用領域の削減については、ヒープの使用量はアプリケーションが本来必要とするメモリと同程度となるため、メモリ削減の効果はある。特にアプリケーションの動作が定常状態となり、大きなメモリ使用の変動がない場合に顕著である。

一方、メモリ使用量が変動している場合には、メモリ確保処理や GC 処理のため、実行速度や応答性の劣化が発生する。これらの劣化を発生させないためにヒープに持たせる余裕領域の評価については今後の課題である。

7. おわりに

本報告では、アドイン環境の実現方式として、省メモリマルチ VM 方式を提案した。

クラスデータやコンパイルコードの共有による省メモリについては、シングル VM 方式ほど

の効果は得られないものの、マルチ VM 方式の利点を損なわない範囲で一定の効果が得られていることを確認した。共有による省メモリ対策によるシステムクラスのコード実行性能の劣化は、1 つのバイトコードあたり数命令であり、大きなものではないが、積算されることによる影響がある。本影響の評価は今後の課題である。

一方、ヒープ管理方式の変更による省メモリについては、固定ヒープ方式に比較すればその効果は明らかであるものの、その運用は必ずしも容易ではないと考えている。ヒープ使用量については、適切な初期および最大ヒープ使用量をアプリケーションの属性として指定する、Java 実行環境が平均的なメモリ使用量を測定してアプリケーション管理機構に保存して利用すること、などの対応が考えられる。バックグラウンド以降時にヒープ縮小を行う対策などと同様に、それぞれ単独では実現容易であるが、実際に効果的な方法の検討は、今後の課題である。

参考文献

[1] Android - An Open Handset Alliance Project, http://code.google.com/android/

[2] Laurent Daynès and Grzegorz Gzajkowski, “Sharing the Runtime Representation of Classes across Class Loaders”, Proc. 19th European Conference on Object-Oriented Programming (ECOOP 2005), July, 2005.

[3] JSR-121 Application Isolation API

[4] Vesa-Matti Hartikainen, Pasi P. Liimatainen, and Tomii Mikkonen, “On Mobile Java Memory Consumption”, Proc. 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing(PDP'06), Feb. 2006.

[5] JDK6 Documentation, Sun Microsystems, Inc.

[6] The K virtual machine, Sun Microsystems, Inc.

[7] CLDC HotSpot™ Implementation Architecture Guide, Sun Microsystems, Inc.

[8] CDC Runtime Guide, Version 1.0, Sun Microsystems, Inc.

[9] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom, “Code Sharing among Virtual Machines”, Proc. 16th European Conference on Object-Oriented Programming (ECOOP 2002), June 2002.

[10] Bernard Wong, Grzegorz Czajkowski, and Laurent Daynès, “Dynamically Loaded Classed as Shared Libraries: an Approach to Improving Virtual Machine Scalability”, Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS), April, 2003.

[11] 高橋克英, 清原良三, “携帯端末向けの Java 高速化手法とその評価”, 情報処理学会論文誌 Vol.48 No.2, 2007 年 2 月

[12] 岡田英明, 高橋克英, 清原良三, “携帯端末向け Java 動的コンパイラのチューニング手法の検討”, 情報処理学会 第 67 回全国大会講演論文集

[13] JSR-139 Connected, Limited Device Configuration 1.1

[14] JSR-118 Mobile Information Device Profile 2.0

[15] JSR-218: Connected Device Configuration 1.1

[16] “High performance Java on embedded devices”, http://www.arm.com/pdfs/JazelleDBX_WhitePaper_2007v1p1.pdf”, ARM Limited

[17] Craig Heath, “SymbianOS Platform Security”, John Wiley & Sons.

[18] 中西正和, 田中詠子, “世代別ごみ集め”, 情報処理 Vol.35 No.11, 1994 年 11 月