A_016

メモリダンプとロードによる組込みJava™実行環境の起動高速化
Boot Acceleration of Java™ Runtime Environment with Memory Dump and Load

川崎 進一郎† Shin-ichiro KAWASAKI　池原 潔† Kiyoshi IKEHARA　井奥 章† Akira IOKU

1. はじめに

携帯電話、Blu-Rayレコーダ、カーナビなど、コンシューマ向け組込み機器にJava技術の適用が進められている。これらのコンシューマ機器では、アプリケーションの起動時間が問題になっている。例えば携帯電話Java向けのゲームには起動に１０秒程度かかるものが多く見受けられる（表 1）。CPU処理性能の制限に加え、Javaクラスロード、画像ロードの処理時間が起動に時間がかかる原因の一端となっている。

一方、起動高速化の手法として、メモリダンプ方式が知られている。例えば Lisp 処理系をベースに実現されている Emacs エディタでは、Lisp により記述されたエディタ初期化処理を事前に実行しておき、完了時に処理系のメモリイメージをダンプする。次回実行時はダンプされたメモリイメージをロードして実行を開始することで、初期化処理を省略している。

表 1　au 携帯電話向け Java アプリの起動時間

#	ゲーム名	起動時間(秒)
1	アドベンチャーゲームA	4
2	アドベンチャーゲームB	4
3	アクションゲームA	5
4	アクションゲームB	9
5	アクションゲームC	12
6	シミュレーションゲーム	13
7	アクションパズルゲーム	14

※　A5303H を利用し、ストップウォッチにより計測

本稿では、メモリダンプ方式を用いて組込み Java 実行環境の起動時間を短縮する方法を検討、試作した結果について述べる。

† 日立製作所(株) 中央研究所 組込みシステム基盤研究所

2. メモリダンプによる起動時間短縮

検討にあたり、OS は、携帯電話への採用が進む Linux を仮定した。また、Java 実行環境のスレッド実装として、Linux シングルスレッド上で複数の Java スレッドを実現するグリーンスレッド方式を仮定した。試作に用いた Java 実行環境 CLDC は、これらの条件を満たす実装が公開されている。以下、検討の結果採用した処理内容を示す。

2.2. 処理手順

ダンプおよび再起動の手順は６段階からなる（図 1）。

①Java 実行環境の ELF バイナリが Linux 環境におけるローダである ld.so によりメモリ上へロードされる。
②Java プログラムの実行を開始する。クラスや画像が読み込まれ、初期化処理が実行される。
③特定キーの押下などのトリガに応じ、メモリイメージをダンプする。
④ダンプされたイメージと元々の Java 実行環境プログラムから、高速起動用 Java 実行環境バイナリを生成する。
⑤ld.so により高速起動用のプログラムをロードする。
⑥ロード後に、③ダンプ処理時の状態を復帰する。
⑦復帰した状態から実行を再開する。

処理①と②が従来の起動処理であり、処理⑤と処理⑥が高速な起動処理である。以下、Java 実行環境を構成するデータ要素毎に、処理内容を説明する。

2.3. コードとデータ

コード領域(.text セクション)の内容はダンプ前後で変わらない。データ領域(.data および.bss セクション)は、起動後の初期化処理によって変更される。ダンプした領域を高速起動用 Java 実行環境にとりこむ。

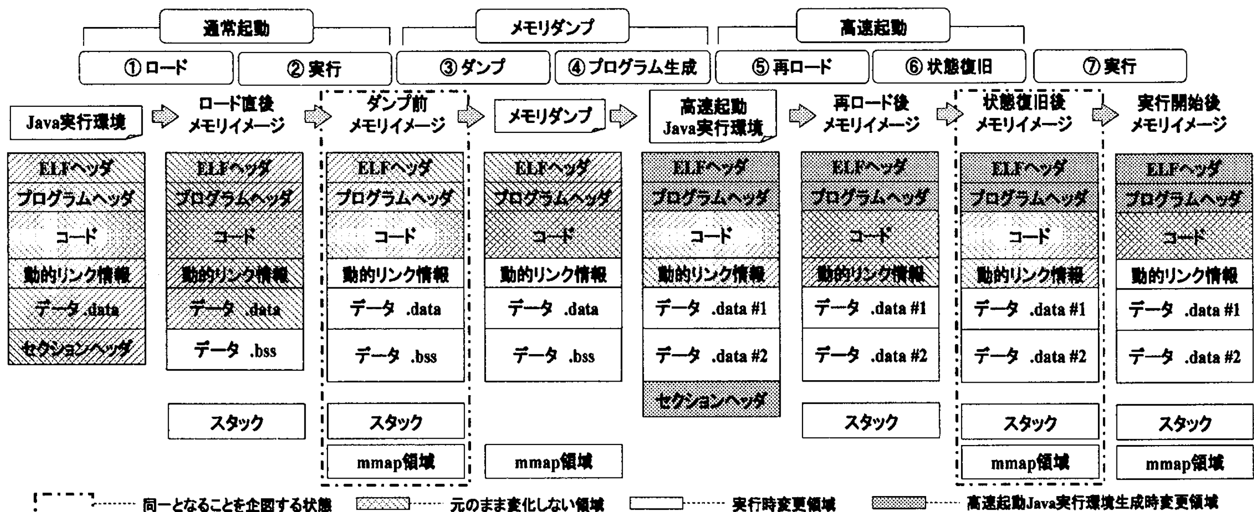

図 1　メモリダンプ手順とプログラムデータ構成

初期化データを持たない.bss セクションは、ロード時に確保される。ダンプした.bss セクションを、高速起動用 Java 実行環境の２つ目の.data セクションとして取り込む。

2.4. ヒープ

Java オブジェクトなどが配置されるヒープ領域は、malloc や mmap により確保される。malloc などのＣ言語標準関数により確保されるヒープは、.bss 領域に後続する領域に配置される。.bss 領域と一括してダンプし、.data セクションとして高速起動 Java 実行環境に取り込むことで復旧できる。ヒープを管理する malloc ルーチンの状態は、glibc 内に管理されている点に留意が必要である。再起動時に実行される glibc の初期化処理により、malloc ルーチンも初期化されるため、復旧されるヒープ領域と malloc ルーチンの状態に不整合が生じてしまう。この不整合は、glibc の malloc ルーチン(Doug Lea's malloc) の提供する状態保存・復帰用 API を用いることで回避できる。

一方Ｃ言語標準関数ではなく mmap システムコールにより確保された領域は、システムコールをフックすることで領域のリストを把握できる。これらの領域は別ファイルにダンプし、再起動時に読み込むことで対応できる。

2.5. スタック

Java 実行環境の状態をＣスタック上に保持しない実装であれば、Ｃスタックの復帰は不要である。

3. 試作

3.1. 試作環境

表 2に試作に用いた環境を示す。携帯電話向けプロセッサであるSH-Mobile3、OSにはLinuxを用いることとした。

表 2　試作環境

ハードウェア	Solution Engine(MS73180CP01) [1]
CPU	SH73180 (SH-Mobile3) [2]
動作周波数	216 MHz
メモリ	64Mbyte SDRAM
OS	Linux 2.4.21
ファイルシステム	NFS
クロスコンパイラ	gcc 3.2.4
C ライブラリ	glibc 2.2.5
Java 実行環境	CLDC 1.1 参照実装[3]

3.2. 実装結果

携帯向け Java 実行環境の持つ画面やキー入力を持たない CLDC を実装対象としたため、一般の Java アプリケーションを用いた動作検証は未済である。代わりに CLDC 上で動作可能な Embedded Caffeine Mark 3.0(eCM)を用いて検証した。eCM は約２０秒でベンチマーク計測を完了するアプリケーションであるが、この計測処理の一部を起動処理に見立てる。計測途中の状態をメモリダンプし、ダンプイメージから高速起動用 CLDC を生成した。生成された高速起動用 CLDC を再起動し、正常に計測処理を継続できることを確認した。

図 2に eCM 実行時間を示す。通常起動時、ダンプ時、高速起動時の３ケースを計測し、これらの差分から推定される内訳時間を図示している。高速起動時には、起動処理に見立てた計測処理時間が省略されている。また、ダンプ時には３秒程度の中断が必要であった。

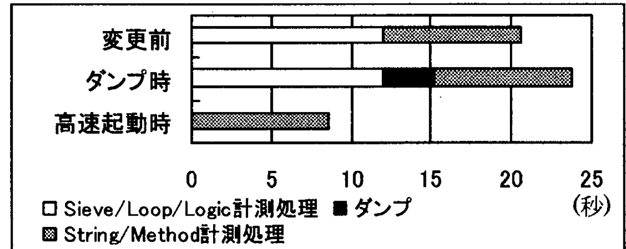

図 2　メモリダンプ適用による eCM 実行時間の変化

ソースコード変更量は CLDC 全体の４％であった。変更前の CLDC のバイナリサイズは 304Kbyte であったが、高速起動用 CLDC のバイナリサイズはヒープ領域分(272Kbyte)増加し、576Kbyte となった。

4. 課題

上述の eCM は、ファイル I/O やネットワーク I/O を伴わないプログラムであった。プログラムが I/O を通じて可変データの入力を受け取った場合、プログラムの状態は入力データに依存することになり、メモリダンプ方式では適切に状態を復帰できない。このため、I/O を監視し、メモリダンプ処理の適用可否を判定する必要がある。

また、実際に出荷されている携帯電話の Java 実行環境は、CLDC に加えて MIDP やキャリア独自ライブラリを備えている。これらのライブラリの管理する状態を再起動時に復元する方法について、別個に検討が必要となる。

これらの課題を含め、表 3に示す制約事項がある。

表 3　制約事項

制約対象	制約内容
ファイル/ネットワーク	可変データ読込後はダンプ不可。
ライブラリ	CLDC 以外については別途検討が必要。
データ領域	高速起動用 CLDC を保持する領域が新たに必要となる
セキュリティ	高速起動用 CLDC はアプリケーションの実行状態を含むため、機器使用者から参照されない領域に配置する必要がある。

5. まとめと今後の課題

携帯電話を主ターゲットとする Java 実行環境 CLDC に対してメモリダンプによる起動高速化方法を検討、試作し、適切に動作することを確認した。

(i)可変データの入力以降は本方式を適用できないという制限と、(ii)ライブラリ拡張時に各ライブラリの状態保存および復帰方法の実装が必要となる点が課題である。

参考文献

[1] CLDC HotSpot™ Implementation Virtual Machine, pp.16-17, http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf

[2] 永野他, portable dumper; アーキテクチャに依存しない Emacs の起動時間短縮手法, Linux Conference, 2002

[1] (株)日立超LSIシステムズ [2] (株) ルネサステクノロジ
[3] Sun Microsystems, Inc.