

携帯エミュレーターを用いたアプリケーションの観察と考察

東京工業大学  
理学部  
情報科学科

岩橋弥生  
(0403382)

平成19年度卒業論文

指導教官 佐々 政孝 教授

2月14日

# 目次

<b>第1章</b>	<b>はじめに</b>	<b>4</b>
1.1	研究動機 . . . . .	4
1.2	研究概要 . . . . .	4
<b>第2章</b>	<b>Java と Java Platform</b>	<b>5</b>
2.1	概要 . . . . .	5
2.2	Java Platform . . . . .	5
<b>第3章</b>	<b>Java Platform Micro Edition</b>	<b>6</b>
3.1	概要 . . . . .	6
3.2	Connected Limited Device Configuration . . . . .	6
3.3	Mobile Information Device Profile . . . . .	7
<b>第4章</b>	<b>Sun Java Wireless Toolkit</b>	<b>8</b>
4.1	概要 . . . . .	8
4.2	ツールキットの機能 . . . . .	8
4.3	サポートされている技術 . . . . .	9
4.4	エミュレータの使用法 . . . . .	11
4.4.1	エミュレータの環境設定 . . . . .	11
4.4.2	アプリケーションの監視 . . . . .	11
<b>第5章</b>	<b>実験</b>	<b>15</b>
5.1	アプリケーションの高速化 . . . . .	15
5.2	for 文の変更 . . . . .	16
5.2.1	原理 . . . . .	16
5.2.2	実験 - テストプログラマー . . . . .	19
5.2.3	実験 - Life3D に適用した場合 - . . . . .	21
5.3	ループの展開 . . . . .	23
5.3.1	原理 . . . . .	23
5.3.2	実験 - Life3D に適用した場合 - . . . . .	23
5.4	メソッド呼び出しの展開 . . . . .	25
5.4.1	原理 . . . . .	25
5.4.2	実験 - Life3D に適用した場合 - . . . . .	25
5.5	ビット演算の使用 . . . . .	28
5.5.1	原理 . . . . .	28
5.5.2	実験 - Life3D に適用した場合 - . . . . .	28

5.6	実験のまとめ . . . . .	31
第 6 章	関連研究	32
6.1	携帯端末向けの Java 高速化手法とその評価 . . . . .	32
第 7 章	まとめと今後の課題	33

# 目次

4.1	K Toolbar	8
4.2	アプリケーションのパッケージ化	9
4.3	アプリケーションの開発とテスト	10
4.4	エミュレータの外見	12
4.5	プロファイラ	13
4.6	メモリーモニター	14
5.1	ソースコード	17
5.2	i++のクラスファイル	18
5.3	i-のクラスファイル	18
5.4	for(int i = 0; i < n; i++)の結果	20
5.5	for(int i = n - 1; i >= 0; i--)の結果	20
5.6	変更前のデータ	21
5.7	変更後のデータ	21
5.8	変更前に対する変更後の割合 (%)	21
5.9	変更前のデータ	23
5.10	変更後のデータ	23
5.11	変更前に対する変更後の割合 (%)	23
5.12	変更前のデータ	25
5.13	変更後のデータ	26
5.14	変更前に対する変更後の割合 (%)	26
5.15	bindTargetGraphics, drawStringのデータ	26
5.16	変更前のupdateNeighbours	29
5.17	変更後のupdateNeighbours	30
5.18	変更前のデータ	31
5.19	変更後のデータ	31
5.20	変更前に対する変更後の割合 (%)	31

# 第1章 はじめに

## 1.1 研究動機

今日の情報技術の中で、私たちはプログラミング言語というものをを用いてコンピュータを動作させている。この数年間で瞬く間に世の中に浸透した携帯電話もまた、こうしたプログラミング言語によって動いている。今の携帯電話に使われている最も主流なプログラミング言語は、Java である。私は、大学の授業で初めて Java のプログラミングを習ったときに、それがパソコンだけでなく、携帯電話や家電製品にも使用されていることにとっても興味が湧いた。普段大学の授業で動かしている Java プログラミングのアプリケーションが、携帯電話などの小型機器上にどのようにインストールされて実行されるのか、詳しい仕組みを知りたくなったのである。これが、この論文「エミュレータを用いた携帯用アプリケーションの監視と考察」を作成するきっかけである。

## 1.2 研究概要

本論文では、携帯電話のアプリケーションの最適化の中でも特に、シューティングゲームなどのプログラムの実行速度が重要となっているアプリケーションを対象とした。まずはじめに、プログラムの実行速度を速くするいくつかの方法について考察し、次にそれをあとで説明する J2ME Wireless Toolkit のデモアプリケーションである Life3D というプログラムに適用し、その効果を観察した。実験には J2ME Wireless Toolkit のエミュレータを用い、プログラムの観察にはエミュレータの機能であるプロファイラとメモリーモニターを使用した。実験結果から、おおむね理論どおりプログラムを高速化することができたが、プログラムを変更した副作用により逆に計算速度が遅くなる可能性もあることが分かった。

## 第2章 JavaとJava Platform

### 2.1 概要

JavaはSun Microsystem社が開発したプログラミング言語であり、それまで主要であったC言語とは違う特徴を持った言語である。その主な特徴としては以下のことがあげられる。

- プラットフォームの違いに影響されない言語であり、一度ソースコードを書けば様々なプラットフォーム上で実行することが出来る
- Javaで書いたプログラムは、Java仮想マシンと呼ばれるソフトウェア上で実行される
- データの集合とそれに対する手続きをセットにして管理する、オブジェクト指向の言語である

Javaは今日、情報記述の広範囲にわたって使われており、携帯電話などの組み込み機器や会社や家庭内で使われているコンピュータ、更に大規模なスーパーコンピュータにまでJavaの技術は浸透している。このように様々な種類の計算機に使われているJavaであるが、その中身を詳しく見ると更に3つに分類することが出来る。次の節でその3つの分類について説明する。

### 2.2 Java Platform

Java Platformとは、Java仮想マシン、ライブラリのセットなど、Javaプログラムを実行する上で必要な実行環境のことである。Java Platformは、以下の3つに大きく分けることができる。

- Java Standard Edition(Java SE)
  - 汎用的な用途に使われるもので、他の2つのプラットフォームの基盤ともなっている
- Java Enterprise Edition(Java EE)
  - Java SEにサーバー用の諸機能を付け加えたもので、業務システムなどの大規模なサーバーを対象としたもの
- Java Micro Edition(Java ME)
  - 家電製品やPDA(携帯情報端末)、携帯電話などの組み込み機器を対象としたもの

次の章では、携帯電話に使われているJava Micro Editionについて詳しく説明する。

## 第3章 Java Platform Micro Edition

### 3.1 概要

Java Platform Micro Edition とは、携帯電話、組み込み式デバイスなど、リソースに制限のある小型機器のための Java Platform であり、小型機器の制限のあるメモリサイズ、画面、CPU のパワーなどを考慮した上で開発されたものである。

Java ME の技術は、主に次の 3 要素からなる。

- 最も基本的なライブラリのセットと、さまざまなデバイスに対応できる VM
- 特定のデバイスをサポートする API
- API を更に発展させた追加のパッケージ

JavaME は、更に以下の 2 つのフレームワークを持つ。

- Connected Limited Device Configuration (CLDC)
  - 携帯電話などの限られたリソースを持つ機器を対象としたフレームワーク。Mobile Information Device Profile (以下 MIDP) のライブラリを用いて、携帯電話などの機器に対する Java 実行環境を提供する。
- Connected Device Configuration (CDC)
  - PDA (携帯情報端末) などの、CLDC の対象とする機器よりもリソース制限の少ない機器を対象としたフレームワーク。

今回の実験は携帯電話が対象であったので、次の節では携帯電話に使用されている CLDC について説明する。

### 3.2 Connected Limited Device Configuration

Connected Limited Device Configuration (以下 CLDC) は、Java Platform, Micro Edition の基盤となるアーキテクチャの一つである。J2ME は、configuration, profile, optional package の 3 つの API によって動作する。J2ME のアプリケーションは、CLDC のような configuration と MIDP のような profile を持っており、更に optional package を付けると、無線通信などの機能も付けることができる。

CLDC が対象とするデバイスは、主に次のような性質を持つ。

- 16bit または 32bit のプロセッサで、16MHz 以上のクロックスピードを持っている

- CLDC ライブラリと VM に割り当てられるメモリ量が 160KB 以上
- Java Platform に割り当てられるメモリ量が 192KB 以上
- 大半がバッテリー電源により動作し、少ない電力消費量
- (通常は無線で断続的な接続、帯域幅に制限のある) ネットワークに接続可能

通常の携帯用アプリケーションは、J2ME プラットフォーム上で CLDC と MIDP のセットを組み立てて動作する。次の節では MIDP について説明する。

### 3.3 Mobile Information Device Profile

Mobile Information Device Profile (MIDP) は、Java Platform, Micro Edition の主要なプロファイルである。今日の主要な携帯電話や PDA は、前節で説明した CLDC と本節の MIDP を組み合わせることによって実装されている。日本の携帯電話では、au の EZ アプリやソフトバンクモバイルの S! アプリに MIDP が使われている。また NTT ドコモの i アプリは、NTT ドコモが独自に開発した DoJa というプロファイルを使用している。

CLDC と MIDP は、標準化された Java 実行環境と Java API のセットによって携帯用小型機器のプログラミングをサポートしている。MIDP で書いたプログラムは、様々な種類の小型機器の上で動かすことができる。MIDP で書かれたアプリケーションは MIDlet と呼ばれている。

また MIDP アプリケーションの開発をサポートする開発ツールとしては Sun 社により提供されている、Sun Java Wireless Toolkit などがある。au とソフトバンクモバイルがそれぞれ EZ アプリ、S! アプリの開発ツールとして Sun Java Wireless Toolkit をあげており、一方 NTT ドコモの i アプリは同社独自の DoJa エミュレータを提供している。

次の節では、Sun Java Wireless Toolkit の主な機能について説明する。

# 第4章 Sun Java Wireless Toolkit

## 4.1 概要

Java Wireless Toolkit は、携帯電話や PDA などの組み込み機器用のアプリケーションを開発するための一連のツール群である。Java Wireless Toolkit は、CLDC、MIDP とその他いくつかのオプションパッケージをサポートしている。また、開発を効率よく行うために、携帯電話および PDA のエミュレータ、アプリケーションの監視のためのツール、そしていくつかのデモアプリケーションなどが備え付けられている。

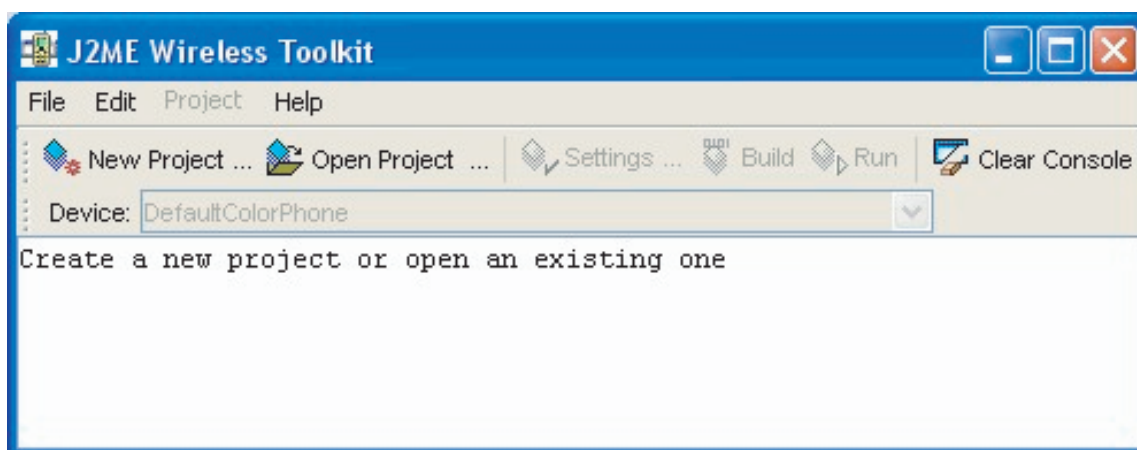


図 4.1: K Toolbar

## 4.2 ツールキットの機能

Java Wireless Toolkit は次のような主要機能を備えており、MIDP アプリケーションの作成をサポートしている。アプリケーションの開発とテストの主な流れを図 4.3 に示す。

- ビルドとパッケージ化
  - ユーザーはソースコードを作成するだけで、残りの処理はツールキットに任せることができる。ボタンをクリックすると、ソースコードのコンパイル、クラスファイルの実行前検証、および MIDlet スイート（複数の MIDlet の集合）のパッケージ化がツールキットによって行われる。
- 実行と監視

- MIDlet スイートは、エミュレータで直接実行するか、実際にデバイスにアプリケーションをインストールするときと同様の手順でインストールすることができる。MIDlet の動作を分析するために、メモリーモニター、ネットワークモニター、およびメソッドプロファイラが用意されている。

- MIDlet スイートの署名

- ツールキットには、MIDlet スイートに暗号で署名するツールが含まれている。このツールは、さまざまな保護のドメインで MIDlet の動作をテストするのに役立つ。

実行と監視の詳細については、後の「エミュレータの使用法」、「アプリケーションの監視」の節で説明をする。

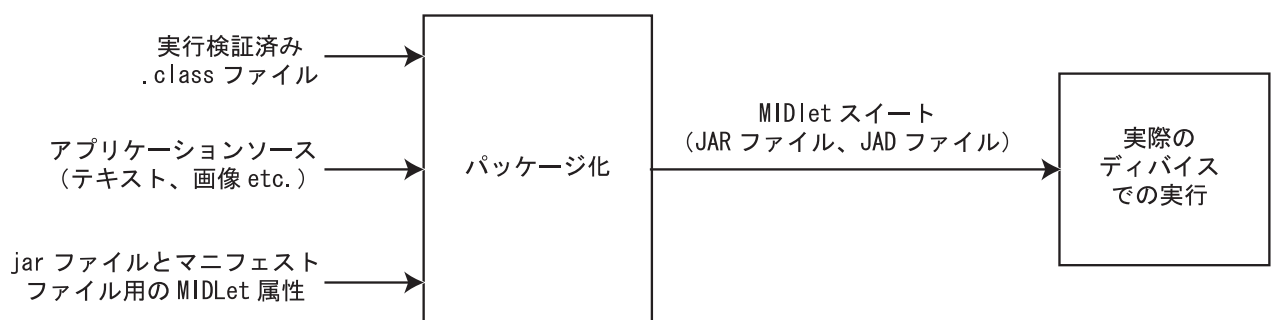


図 4.2: アプリケーションのパッケージ化

### 4.3 サポートされている技術

Java Wireless Toolkit では、以下のアプリケーションプログラミングインターフェース (API) がサポートされている

- Connected Limited Device Configuration (CLDC)
- Mobile Information Device Profile (MIDP)
- Java Technology for the Wireless Industry (JTWI)
- Wireless Messaging API (WMA)
- Mobile Media API (MMAPI)
- PDA Option Package for the J2ME Platform (PIM and File)
- Java API for Bluetooth (Bluetooth and OBEX)
- J2ME Web Services Specification
- Mobile 3D Graphics API for J2ME (3D Graphics)

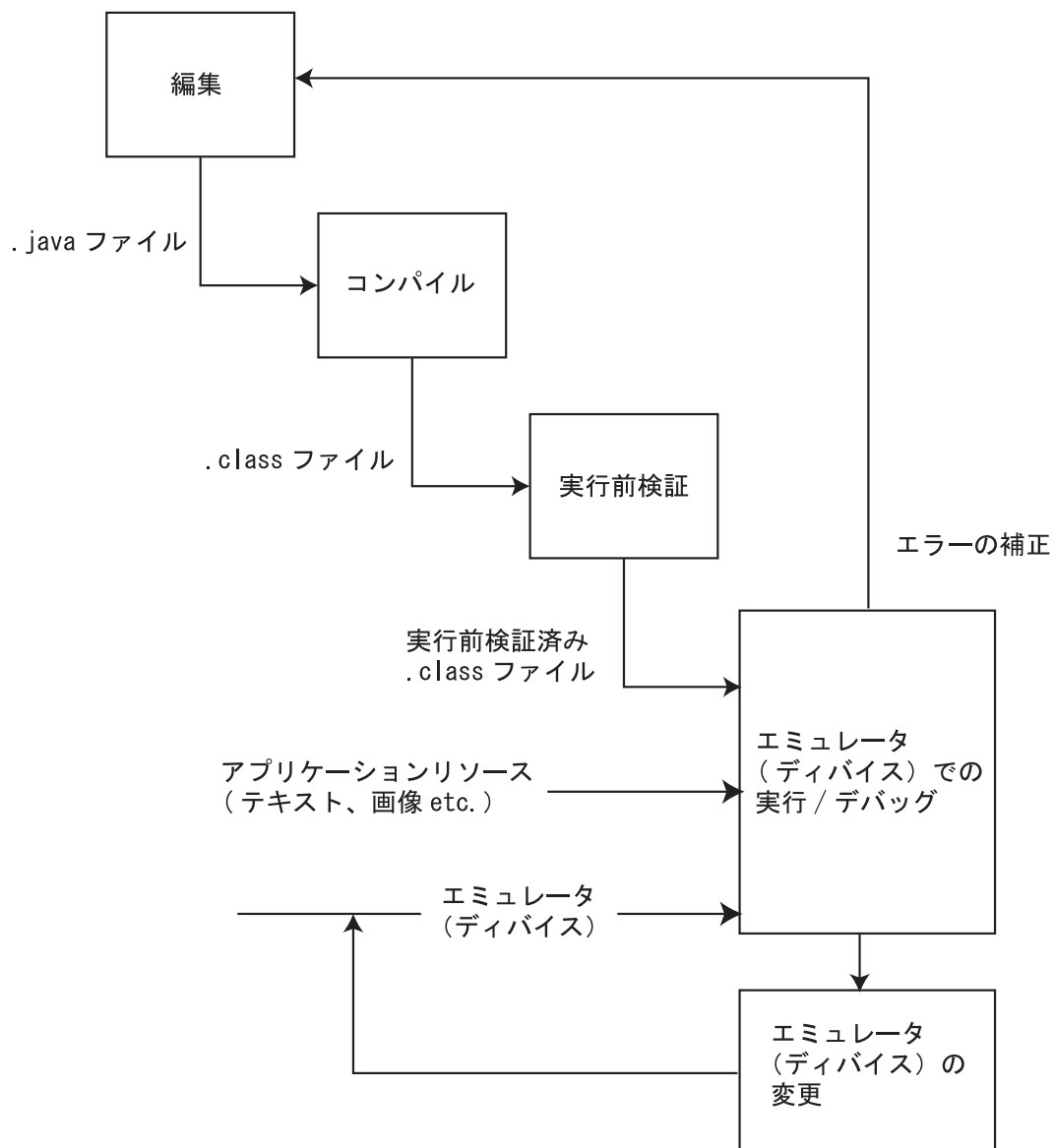


図 4.3: アプリケーションの開発とテスト

## 4.4 エミュレータの使用法

Java Wireless Toolkit のエミュレータは、デスクトップ上で MIDP デバイスをシミュレートする。エミュレータを使用すると、アプリケーションが MIDP 環境でどのように動作するかを確認することができる。また、エミュレータは環境設定を行うことができ、さまざまな実行環境を想定した上でアプリケーションをテストすることができる。エミュレータの外見を図 4.4 に示す。

### 4.4.1 エミュレータの環境設定

エミュレータの環境設定には、以下の項目がある。

- ネットワークプロキシ
  - エミュレータでは、デスクトップコンピュータのネットワーク接続が使用されている。この項目では、HTTP のバージョンの設定、プロキシサーバの使用の有無などを設定できる。
- ヒープサイズ
  - アプリケーションのオブジェクトが保存されるメモリ領域であるヒープ領域のサイズを変更することができる。最小のヒープサイズは、32K バイト、最大のヒープサイズは、65536K バイトである。
- ストレージと消去
  - ストレージファイルの保存場所を変更したり、ストレージのサイズを制限したりできる。
- エミュレータのパフォーマンス調節
  - 実際のデバイスの動作に近づけるため、Graphics クラスの描画メソッドの応答時間、画面表示の更新方法、VM スピードの変更、ネットワークスピードの変更などを調節できる。

### 4.4.2 アプリケーションの監視

Java Wireless Toolkit には、アプリケーションの動作を監視するために以下のようなツールが用意されている。これらのツールは、コードのデバッグや最適化を行うときに役立つ。

- プロファイラ
  - アプリケーション内の各メソッドについて、使用頻度と実行時間が表示される
- メモリーモニター
  - アプリケーション実行時のメモリ使用率が表示される



図 4.4: エミュレータの外見

- ネットワークモニター

- アプリケーションによって送受信されたネットワークデータが表示される。HTTP、HTTPS、SMS、CBS などの多数のネットワークプロトコルがサポートされている。

- トレース

- 低レベルの情報が KToolbar のコンソールに表示される。

ここでは、今回の実験で使用したプロファイラとメモリーモニターについて更に説明する。

## プロファイラ

プロファイラでは、アプリケーション内の各メソッドが追跡される(図 4.5)。アプリケーション 1 回の実行について、各メソッドに費やされた時間と全体に対する割合、各メソッドが呼び出された回数が計算される。メソッドプロファイラには更に、各メソッドの呼び出し関係が階層リストで表示され、プロセッサ時間と呼び出された回数は、そのメソッドとそれに呼び出されたメソッドも含めたものを表示することができる。

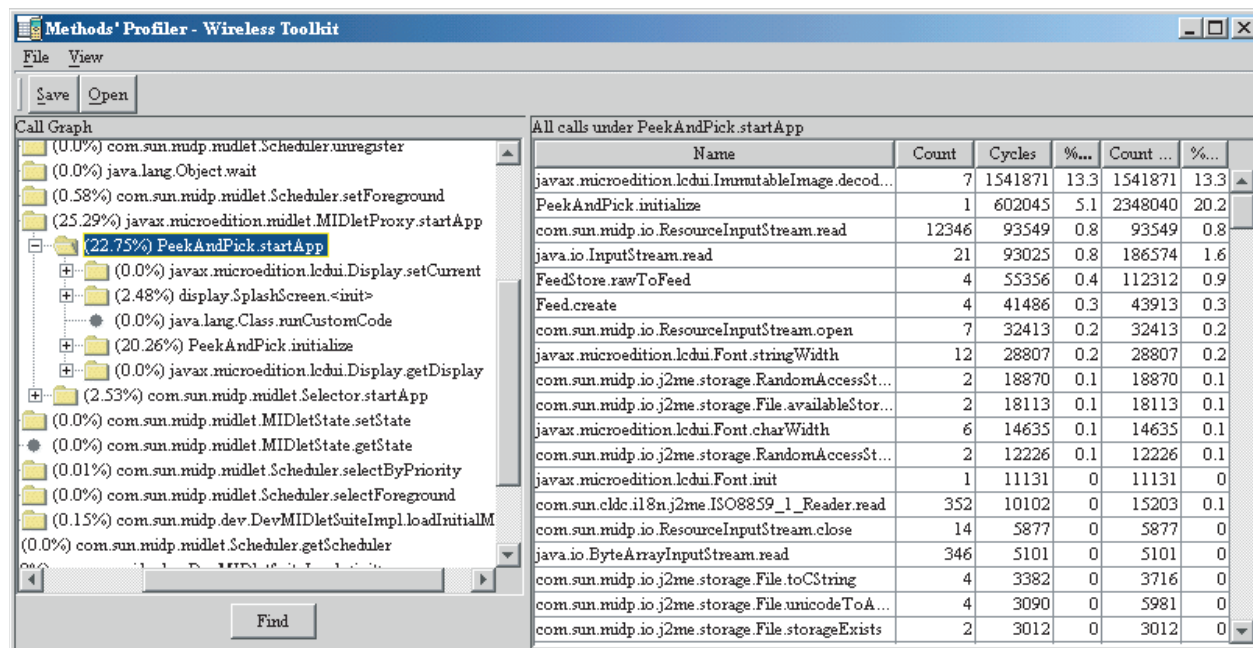


図 4.5: プロファイラ

## メモリーモニター

メモリーモニターでは、アプリケーションで使用しているメモリ量と、各オブジェクトごとのメモリ使用率を表示することができる(図 4.6)。メモリーモニターのグラフには、

- 現在アプリケーションで使用中のメモリ量

- プログラム実行開始から現在までで使用したメモリの最大量
- ヒープ内のオブジェクト数
- 使用されたメモリの量
- 未使用で、使用できる状態のメモリー量
- 起動時に使用できる状態のメモリー量

が表示される。

また、メモリーモニターのオブジェクトのウィンドウでは、

- インスタンス数
- アプリケーションの開始から現在までに作成されたオブジェクトの総数
- オブジェクトで使用されるメモリーの合計量
- オブジェクトの平均サイズ(アクティブなインスタンスの数で合計サイズを割って計算される)

が表示される。

メモリーモニターを使用しているときは、作成される全てのオブジェクトが記録されるため、アプリケーションの動作は遅くなる。

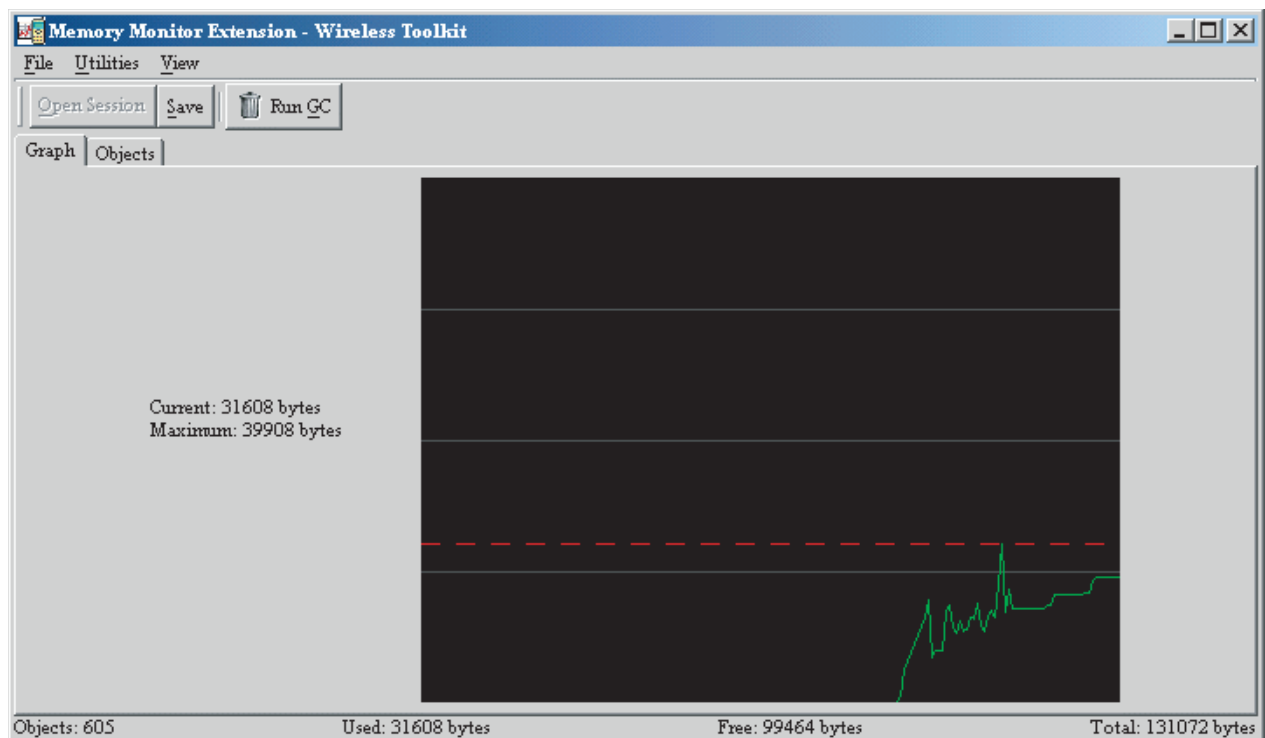


図 4.6: メモリーモニター

## 第5章 実験

### 5.1 アプリケーションの高速化

アプリケーションの高速化の方法としては、以下のような項目を実験した。。

- for 文は、デクリメント計算を使用して、毎回の比較を 0 と行うようにする
- ループを展開する
- メソッド呼び出しをできるだけ取り除く
- $2^n$  の数字による乗除算にはビット演算と使用する

これらの項目を J2ME Wireless Toolkit のデモアプリケーションである、Life3D に適用した。Life3D とは、ライフゲーム (Conway's Game of Life) の 3D 版である。実験で対象とした主なメソッドについて簡単に説明する。

- `startApp()`
  - MIDlet を実行した際に、一番始めに呼び出されるメソッド。Java でいう main メソッド。
- `paint(Graphic g)`
  - Life3D の一番主要なループのメソッドである。Life3D のルールに従ってセルを画面に表示する。startApp に呼び出されている。
- `updateNeighbours(int i)`
  - 生きているセルがあった場合に、その情報を周りの 26 のセルに与える。paint に呼び出されている。

それぞれの実験についてこれ以降の節で詳しく述べていく。

## 5.2 for文の変更

Java では for 文を使用する際に、条件文の中にインクリメントを用いるよりも、デクリメントを用いて毎回の比較を 0 と行うようにした方がプログラムの実行速度は速くなる。すなわち、

```
for(int i = 0; i < 10; i++)
```

と記述するよりも、

```
for(int i = 9; i >= 0; i--)
```

と記述した方が、プログラムの実行速度が速くなるということである。

### 5.2.1 原理

この理由は、Java のアセンブリ言語の中に、0 との比較による命令が存在していることによる。この命令により、上の 2 つの for 文をアセンブリ言語に変換した際にループ内の命令数に差が生じるのである。図 5.1 にソースコードの一部、図 5.2 図 5.3 に図 5.1 で示した部分のクラスファイルのコードを示す。ちなみに、本論文では携帯用のアプリケーションを対象としているので、プログラムは MIDlet のものを使用する。このプログラムはエミュレーター上で動かしてみて、実際に 2 つの for 文の実行速度に差があるのかどうか確認をした。

図 5.2 と図 5.3 のクラスファイルを比較してみると、35 番地のコードまでは同じであることが分かる。35 番地はプログラムの `start = System.currentTimeMillis();` の部分で、32 番地で取ってきた `System.currentTimeMillis()` の long の値をレジスタにしまっている。

そのあと `for(int i = 0; i < 1000000; i++)` の方は、36 番地で `i` のはじめの値である 0 をレジスタにしまい、37 番地でその値をレジスタから読み込んでいる（ちなみにこれは冗長なコードである）。ループは 30 番地から 49 番地の 5 行である。`for(int i = 999999; i >= 0; i--)` の方では、36 番地で `i` のはじめの値である 999999 をレジスタにしまい、38 番地で再度それを読み込んでいる。ループは 40 番地から 48 番地の 4 行である。それぞれのループについて詳しく説明する。まず、`for(int i = 0; i < 1000000; i++)` の方は

```
39:  iload    13 //スタックフレームの 13 番地の値 (現在の i の値) を演算スタックに読み込む
```

```
41:  ldc     #8; //1000000 をスタックに読み込む
```

```
43:  if_icmpge    52 //value1 >= value2 ならば 52 番地へ飛ぶ。
```

ただし、value1 はスタックトップの次の値、value2 はスタックトップの値である

```
46:  iinc    13, 1 //スタックフレームの 13 番地の値 (現在の i の値) に 1 を加える
```

```
49:  goto   39 //39 番地に飛ぶ
```

`for(int i = 999999; i >= 0; i--)` の方は、

### **i-- のソースコード**

```
public class Test extends MIDlet implements CommandListener {
    public void startApp() {
        ...
        start = System.currentTimeMillis();
        for(int j = 0; j < 1000000; j++) {}
        finish = System.currentTimeMillis();
        result = finish - start;
        ...
    }
    ...
}
```

### **i++ のソースコード**

```
public class Test extends MIDlet implements CommandListener {
    public void startApp() {
        ...
        start = System.currentTimeMillis();
        for(int j = 999999; j >= 0; j--) {}
        finish = System.currentTimeMillis();
        result = finish - start;
        ...
    }
    ...
}
```

図 5.1: ソースコード

```

public void startApp():
Code:
...
...
32: invokestatic    #7; //Method java/lang/System.currentTimeMillis:()J
35: lstore_3
36: iconst_0
37: istore 13
39: iload 13
41: ldc    #8;
43: if_icmpge      52
46: iinc 13, 1
49: goto 39
52: invokestatic    #7; //Method java/lang/System.currentTimeMillis:()J
55: lstore 5
...
...

```

図 5.2: i++のクラスファイル

```

public void startApp():
Code:
...
...
32: invokestatic    #7; //Method java/lang/System.currentTimeMillis:()J
35: lstore_3
36: ldc    #8; //Push int 999999
38: istore 13
40: iload 13
42: iflt 51
45: iinc 13, -1
48: goto 40
51: invokestatic    #7; //Method java/lang/System.currentTimeMillis:()J
54: lstore 5
...
...

```

図 5.3: i-のクラスファイル

```
40:  iload   13 //レジスタ 13 の値 (現在の i の値) をスタックに読み込む
42:  iflt   51 //スタックトップの値が 0 未満であれば 51 番に飛ぶ
45:  iinc   13, -1 //レジスタ 13 の値 (現在の i の値) に-1 を加える
48:  goto   40 //40 番地に飛ぶ
```

となっている。ここで 42 番地の `iflt 51` は 0 との比較による条件分岐命令である。この命令の存在により、`for(int i = 0; i < 1000000; i++)` では `i` と 1000000 との比較のために毎回 1000000 をスタックに読み込んでいたのに対し、`for(int i = 999999; i >= 0; i--)` では 0 の読み込み無しに比較が行える。すなわち、後者の方がループ内の命令数が 1 つ少なく済み、計算速度が速くなるのである。

## 5.2.2 実験 - テストプログラムー

実際に J2ME Wireless Toolkit のエミュレータ上で、プログラムを動かしてみた。その結果を図 5.4 と図 5.5 に示す。それぞれの数字を説明すると、「回数」は `for` 文をまわした回数である。「時間」は、プログラムの 2 つの `System.currentTimeMillis()` の差であり、すなわち 2 つ目の `for` 文にかかった時間である。単位はミリ秒であり、10 回測った時間の平均をとっている。「メモリ」は、2 つ目の `for` 文で計算されたメモリ量であり、プログラム全体で計算されたメモリ量から、2 番目の `for` 文を消したプログラムの計算にかかったメモリ量を引いたものを 10 で割って計算をしている。

この結果から、エミュレータ上では `for(int i = 0; i < n; i++)` よりも `for(int i = n - 1; i >= 0; i--)` のプログラムの方が時間的にもメモリの的にも速く少なく計算されていることが分かる。この結果は上で述べた原理から、予想通りといえる。

回数	時間	メモリ
10	0	55
10 <sup>2</sup>	0	505
10 <sup>3</sup>	0	5005
10 <sup>4</sup>	3	50035
10 <sup>5</sup>	32	500170
10 <sup>6</sup>	332	5000567.3
10 <sup>7</sup>	3323	50000648
10 <sup>8</sup>	33303	---

図 5.4: `for(int i = 0; i < n; i++)` の結果

回数	時間	メモリ
10	0	44
10 <sup>2</sup>	0	404
10 <sup>3</sup>	0	4004
10 <sup>4</sup>	3	40034
10 <sup>5</sup>	26	400169
10 <sup>6</sup>	265	4000563.4
10 <sup>7</sup>	2634	40000647
10 <sup>8</sup>	26326	---

図 5.5: `for(int i = n - 1; i >= 0; i--)` の結果

### 5.2.3 実験 - Life3D に適用した場合 -

次に、この理論を Life3D の startApp と、ループメソッドである paint に適用した。

#### 結果

結果を図 5.6 ~ 図 5.8 に示す。図のサイクルはそれぞれのメソッドの実行 1 回に費やされたプロセッサ時間、子を含めたサイクルはメソッドとそれに呼び出された子のメソッドも含めたプロセッサ時間（親メソッド 1 回の実行）である。タイムは、プログラムの中でそれぞれのメソッドの始めと終わりを System.currentTimeMillis() で挟み、その差を取った値 (milli seconds) である。バイト数は、プログラム全体で実行された延べのバイトコード数である。数値は、Life3D が始まって画面の描写を 1000 回行った時点でゲームを終了させて取ったものである。実験は 5 回行い、図の数値は 5 回の相加平均である。(以下同じ)

適用前	サイクル	子を含めたサイクル	タイム(ms)
paint	4453810	30673833	16681
startApp	12363800	412042614	219
バイトコード数	36699924		

図 5.6: 変更前のデータ

適用後	サイクル	子を含めたサイクル	タイム(ms)
paint	4165391	31219129	16878
startApp	12241746	406313865	219
バイトコード数	35668718		

図 5.7: 変更後のデータ

メソッド	サイクル	子を含めたサイクル	タイム
paint	93.52	101.78	101.18
startApp	99.01	98.61	100
バイトコード数	97.19		

図 5.8: 変更前に対する変更後の割合 (%)

#### 考察

まず paint についての数値を見てみると、paint のみのサイクルは減少しているのに対し、子を含めたサイクルおよびタイムはわずかに増加していることが分かる。paint の子を含めたサイクルがわずかに増加した理由としては、

- paint の子のメソッドは変更をしなかったため、それらを加えたサイクルを測った時、paint のサイクルの減少の効果が見られなくなった
- データの誤差（5 回の実験で paint の子を含めたサイクルの、平均との誤差は最大で 3.95% であった）

の 2 つが考えられる。タイムの値の増加も、この 2 つが原因と考えられる。

startApp についての数値は、サイクル、子を含めたサイクル共にわずかに減少していた。しかし startApp で変更した for 文のループ数はわずか 512 回であり、startApp 自体はプログラム 1 回の実行につき 1 回しか呼ばれないためあまり効果は期待できず、この数値はデータの誤差である可能性が高い。タイムについては、変更の効果が小さすぎたために変化が見られなかったと考えられる。

バイト数については、減少していた。バイト数は 5 回の実験値の平均との誤差は最大で 0.03% であったので、この減少は誤差でないと考えられる。

## 5.3 ループの展開

プログラム中のループ文は展開した方が計算時間が速くなる。

### 5.3.1 原理

プログラムの中にループ文があると、プログラムの流れを制御するフロー制御のコードができる。したがってループ文を展開することによって、フロー制御のコードによるオーバーヘッドを減らすことができる。

### 5.3.2 実験－Life3Dに適用した場合－

Life3Dのループメソッドであるpaintに適用した。paintは1回実行されるとその中で64回×2のループを行っていたので、その部分を展開した。

#### 結果

結果を図5.9～図5.11に示す。

適用前	サイクル	子を含めたサイクル	タイム
paint	817912	32137085	12837
バイトコード数	8656444		

図 5.9: 変更前のデータ

適用後	サイクル	子を含めたサイクル	タイム
paint	754121	31945904	12775
バイトコード数	8135800		

図 5.10: 変更後のデータ

メソッド	サイクル	子を含めたサイクル	タイム
paint	92.201	99.405	99.514
バイトコード数	93.985		

図 5.11: 変更前に対する変更後の割合 (%)

## 考察

paint についてのデータを見てみると、サイクルに減少が見られ、子を含めたサイクルおよびタイムにはわずかな減少が見られた。これは予想通りの結果である。ただ、子を含めたサイクルでは効果が薄れており、タイムの減少もあまり見られなかった。

バイト数は減少していた。5回の実験値の揺れを見る限り、この減少率は誤差でないと考えられる。

## 5.4 メソッド呼び出しの展開

プログラム中のメソッド呼び出しはできるだけ減らした方が、計算速度は速くなる。特にループ内のメソッド呼び出しは減らした方が良い。ただし、これによりプログラムのソースコードが読みにくくなる可能性は高い。

### 5.4.1 原理

メソッド呼び出しを無くすことで、メソッド呼び出しによるオーバーヘッドを無くすことができる。

### 5.4.2 実験－Life3D に適用した場合－

Life3D のループ文で呼び出されている `paint` に適用した。 `paint` に呼び出されている、 `updateState` と `updateNeighbours` のメソッドを展開した。それぞれのメソッドについて簡単に説明する。

- `updateState(int i)`
  - 自分の周りの生きているセルの数のデータを受け取り、ゲームのルールに従って次にどのセルが生きているかのデータを返す
- `updateNeighbours(int i)`
  - 自分が現在生きていれば、その情報を自分の周りの 26 のセルに与える

この2つのメソッドは、 `paint` 1 回の実行につき、それぞれ 512 回ずつ呼ばれていた。

### 結果

結果を図 5.12～図 5.14 に示す。

適用前	サイクル	子を含めたサイクル	タイム(ms)
<code>paint</code>	4470695	30408032	16475
<code>updateState</code>	14579	14633	---
<code>updateNeighbours</code>	5586	5586	---
バイトコード数	36697998		

図 5.12: 変更前のデータ

適用後	サイクル	子を含めたサイクル	タイム(ms)
paint	10490273	33566114	14238
バイトコード数	32598383		

図 5.13: 変更後のデータ

メソッド	展開したメソッドを含めたサイクル	子を含めたサイクル	タイム
paint	87.28	110.39	86.42
バイト数	88.83		

図 5.14: 変更前に対する変更後の割合 (%)

## 考察

実験結果は、タイム、バイトコード数は予想通り減少したのに対し、子を含めたサイクルは増加していた。

タイムは、実際に画面にセルが描写されているのを観察して描写の速度が速くなっていることが確認できたので、この減少は事実と一致しているといえる。バイトコード数の減少は、5回の実験データより誤差の範囲でないと考えられる。

paintの子を含めたサイクルが増加している点については、理論に反しており、またタイムが減少したという実験結果にも矛盾している。この原因を解明するために、プロファイラの他のメソッドの情報にも注目した。まず、インライン展開した後の paint のみのサイクルについては、従来の paint に updateState のみのサイクルと updateNeighbours のみのサイクルがそれぞれ 512 回足されたものなので、インライン展開前の、paint のみのサイクル + updateState のみのサイクル × 512 + updateNeighbours のみのサイクル × 512、の値を計算し、インライン展開後の paint のみのサイクルと比較した。その結果、インライン展開後の方が数値が小さいことが分かった。そしてこの数値の減少率は、タイムの減少率とも近くなっている。よって、paintの子を含めたサイクルの増加の原因は、paint、updateState、updateNeighbours 以外のメソッドにあると考えられる。

そこでプロファイラの他のメソッドについて、インライン展開前と後でデータに変化が見られるものを探したところ、paintの子のメソッドである bindTargetGraphics と drawString の値に増加が見られた。図 5.15 にそのデータを示す。bindTargetGraphics は、Graphic の

	適用前のサイクル	適用後のサイクル	適用後 ÷ 適用前(%)
bindTargetGraphics	8362347	15236731	182.21
drawString	2159796	2456403	113.73

図 5.15: bindTargetGraphics, drawString のデータ

描写の準備としてプログラムから与えられた Graphic3D の情報をまとめるメソッドである。このメソッドのサイクルは大幅に増加していた。これはプロファイラのデータの取り方が原因なのか、bindTargetGraphics の性質が原因なのか不明である。drawString のサイクル

はやや増加していた。このメソッドは5回の実験での最大誤差が19.6%であり、サイクル数の揺れの大きいメソッドであるので5回の実験のみのデータで比較したことが増加の原因である可能性がある。

## 5.5 ビット演算の使用

プログラムの中に  $2^n$  の数字による乗除算があった場合は、普通に計算するよりもビット演算を使用した方が計算速度は速くなる。

### 5.5.1 原理

説明するまでもないが、普通の乗除演算よりもビット演算の方が JVM (エミュレータの場合は KVM) の命令が単純であるので、コンピュータは速く計算できる。

### 5.5.2 実験－ Life3D に適用した場合－

Life3D のプログラムの中には、`updateNeighbours(int i)` というビット演算を使用しているメソッドがある。このメソッドは、今自分の周りの生きているセルの数をカウントする `nextState[i]` を更新するメソッドである。今自分が生きていれば、自分の周りのセルそれぞれの `nextState[i]` のカウントを 1 上げるという計算をしている。この計算を、元のプログラムではビット演算を用いて行っていたが、その部分を普通の演算に直してプログラムの挙動を観察し、ビット演算の効果を調べた。

#### 結果

変更前のプログラムコードを図 5.16 に、変更後のプログラムコードを図 5.17 に、結果を図 5.18 ~ 図 5.20 に示す。タイムは `paint` のタイムである。( `updateNeighbours` は `paint` に呼ばれている )

#### 考察

実験結果より、変更後は `updateNeighbours` のサイクル、子を含めたサイクル共に増加しているのが分かる。 `paint` のタイムはわずかに増加しているが、ほとんど変わっていない。バイトコード数は増加している。5 回の実験の最大誤差は 0.01% であったので、これは誤差でないと考えられる。

予想通りの結果といえるが、問題は `updateNeighbours` のビット演算を普通の演算に変更した際に、アルゴリズムの関係で本来使われていなかった条件文を 26 箇所を使用した。よってこの実験結果の増加がそのままビット演算の効果を表しているとは言えない。この結果から言えることは、`updateNeighbours` はアルゴリズム的にもビット演算を使用するのがもっとも効率のよい方法であるということである。

```

public void updateNeighbours(int i)
{
    if(currentState[i]!=0)
    {
        int ix0 = (i-STEPX)&MASKX;
        int iy0 = (i-STEPLY)&MASKY;
        int iz0 = (i-STEPZ)&MASKZ;

        int ix1 = (i)&MASKX;
        int iy1 = (i)&MASKY;
        int iz1 = (i)&MASKZ;

        int ix2 = (i+STEPX)&MASKX;
        int iy2 = (i+STEPLY)&MASKY;
        int iz2 = (i+STEPZ)&MASKZ;

        ++nextState[ix0|iy0|iz0];
        ++nextState[ix0|iy0|iz1];
        ++nextState[ix0|iy0|iz2];
        ++nextState[ix0|iy1|iz0];
        ++nextState[ix0|iy1|iz1];
        ++nextState[ix0|iy1|iz2];
        ++nextState[ix0|iy2|iz0];
        ++nextState[ix0|iy2|iz1];
        ++nextState[ix0|iy2|iz2];

        ++nextState[ix1|iy0|iz0];
        ++nextState[ix1|iy0|iz1];
        ++nextState[ix1|iy0|iz2];
        ++nextState[ix1|iy1|iz0];

        //!      ++nextState[ix1|iy1|iz1];

        ++nextState[ix1|iy1|iz2];
        ++nextState[ix1|iy2|iz0];
        ++nextState[ix1|iy2|iz1];
        ++nextState[ix1|iy2|iz2];

        ++nextState[ix2|iy0|iz0];
        ++nextState[ix2|iy0|iz1];
        ++nextState[ix2|iy0|iz2];
        ++nextState[ix2|iy1|iz0];
        ++nextState[ix2|iy1|iz1];
        ++nextState[ix2|iy1|iz2];
        ++nextState[ix2|iy2|iz0];
        ++nextState[ix2|iy2|iz1];
        ++nextState[ix2|iy2|iz2];
    }
}

```

図 5.16: 変更前の updateNeighbours

```

public void updateNeighbours2(int i){
    if(currentState[i]!=0)
    {
        int next;
        next = i - STEPX - STEPY - STEPZ;
        if( next >= 0){
            ++nextState[next];
        }else{
            ++nextState[next + NUMCELLS]
        }
        next = i - STEPX - STEPY;
        if( next >= 0){
            ++nextState[next];
        }else{
            ++nextState[next + NUMCELLS]
        }
        next = i - STEPX - STEPY + STEPZ;
        if( next >= 0){
            ++nextState[next];
        }else{
            ++nextState[next + NUMCELLS]
        }
        next = i - STEPX          - STEPZ;
        if( next >= 0){
            ++nextState[next];
        }else{
            ++nextState[next + NUMCELLS]
        }
        next = i - STEPX;
        if( next >= 0){
            ++nextState[next];
        }else{
            ++nextState[next + NUMCELLS]
        }
        next = i - STEPX          + STEPZ;
        if( next >= 0){
            ++nextState[next];
        }else{
            ++nextState[next + NUMCELLS]
        }
        ...
        ...
        ...
        ...
    }
}

```

図 5.17: 変更後の updateNeighbours

変更前	サイクル	子を含めたサイクル	タイム(Paint)
updateNtighbours	16475	5586	16475
バイトコード数	36697998		

図 5.18: 変更前のデータ

変更後	サイクル	子を含めたサイクル	タイム(Paint)
updateNtighbours	6174	6174	16594
バイトコード数	37722400		

図 5.19: 変更後のデータ

メソッド	サイクル	子を含めたサイクル
updateNeighbours	110.53	110.53
タイム(Paint)	100.72	
バイト数	102.79	

図 5.20: 変更前に対する変更後の割合 (%)

## 5.6 実験のまとめ

実験結果より、Paint のみのサイクルはどれも理論通り数値を減らすことができた。しかし、プログラム全体として大きな効果が見られたのはインライン展開のみであった（Paint 子を含めたサイクルが増加するという不明な値が出たが）。

今後の課題としては、インライン展開で理論と反するデータがでることの原因を探求することがあげられる。

## 第6章 関連研究

### 6.1 携帯端末向けの Java 高速化手法とその評価

Java プログラムを高速化する手法として、バイトコードをネイティブコードに変換するネイティブコンパイラを用いた高速化方式である JIT 方式や HotSpot 手法が利用されている。しかし、限られたメモリ資源の携帯端末にネイティブコンパイラを適用すると、1) コンパイル処理およびプロファイリングのためのメモリが足りない、2) コンパイル処理時間がユーザー操作の応答時間に悪影響を与える、という問題が発生する。一方、ホストコンピュータであらかじめ Ahead Of Time Compiler により、マシンコードを生成しておき、マシンコードを携帯端末に送る方法も考えられる。しかしこの方法では、クラスファイルを送るといった慣例に反しており、さら Java アプリケーションをダウンロードする際の通信料が増大してしまう。

そこで、参考文献 [3] では、ユーザー操作の応答時間を保ちながらバイトコードをネイティブコードに変換する手法として、クラスロード時にアプリケーションを解析し、頻繁に使用されると判定されたメソッドをネイティブコードに変換する手法が考えられる。更に本手法では、短時間に変換可能なメソッドをプロファイリング対象として設定し、頻繁に利用されるメソッドをネイティブコードに変換する。本方式を用いることで、コンパイラに利用できるメモリが少ない場合でもユーザー操作に対する応答時間を保ち、JIT 方式と同等の性能が提供できると期待される。

### 6.2 家電向け Java JIT コンパイラの構成方法とその評価

携帯電話などの家電機器への Java 普及が進みつつある中で、Java プログラムの高速化が求められ、Java JIT コンパイラが要望されている。家電機器向けの JIT コンパイラの実装では、実行性能追求よりも家電機器の持つ2つの特性「メモリ資源の制約」「機種展開時の CPU 変更に対応する移植性」を考慮する必要がある。PC 用 JIT コンパイラは最適化による高速化に力を入れているが、家電機器では最適化を実装するとコンパイラのメモリ資源が増え、更に CPU の移植性が低下してしまう。その対策案として、参考文献 [4] では、1) 最適化を実装しないことでコンパイラサイズを縮小し、2) GCC の利用とインタプリタ C ソースコードからコンパイルによる自動生成による移植性の向上、を実現させた。

## 第7章 まとめと今後の課題

本論文では、携帯用アプリケーションの高速化を目的として、携帯用エミュレーターである Java Wireless Toolkit 上でアプリケーションを擬似的に実行し、その挙動を観察した。実験結果からは、本論文で実験した手法はおおむねアプリケーションの高速化に成功したが、一部の手法では高速化適用の副作用により周りのメソッドの計算速度が落ちるという現象も見られた。このことから、理論上ではプログラムを高速化できると考えられるものでも、実際のプログラムに適用するまではその効果は保障できないということが分かった。

今後の課題は、身近なものとしては、本論文では議論できなかった J2ME Game Optimization Secret の他のプログラム高速化手法を実験してみることなどがあげられる。発展課題としては、Life3D 以外のさまざまな携帯用アプリケーションを対象とし、(高速化以外にも)プログラムのチューニングの方法を Java Wireless Toolkit を用いて検討することである。

# 謝辞

本研究を進めるにあたり多大なる御指導ご鞭撻を頂いた、東京工業大学 数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します。

また、佐々研究室の皆様にはさまざまな面で助力を頂きました。あらためまして、ここに深くお礼申し上げます。

## 参考文献

- [1] Sun Microsystems, Inc. 「ユーザーズガイド J2ME Wireless Toolkit 2.2」 2004年10月
- [2] Mike Shivas 「J2ME Game Optimization Secrets」 URL: <http://www.microjava.com/>
- [3] 高橋克英 清原良三 「携帯端末向け 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
- [4] 川本琢二 春名修介 金丸智一 「家電向け Java JIT コンパイラの構成方法とその評価」 情報処理学会論文誌 Vol. 43 No. SIG8 2002年9月