OUKA
Osaka University Knowledge Archive

Title	携帯電話のソフトウェアアドインに関する研究
Author(s)	清原, 良三
Citation	大阪大学, 2008, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/2273
rights	
Note	

Osaka University Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

Osaka University

携帯電話のソフトウェアアドインに関する研究

2008 年 7 月

清 原　良 三

携帯電話のソフトウェアアドインに関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2008 年 7 月

清 原　良 三

学位取得に係る発表論文，国際会議一覧

■ 論文誌

(1) 清原良三, 栗原まり子, 古宮章裕, 高橋清, 橘高大造: 携帯電話ソフトウェアの更新方式, 情報処理学会論文誌, Vol. 46, No. 6, pp. 1492-1500, 2005.

(2) 清原良三, 栗原まり子, 三井聡, 木野茂徳: 携帯電話ソフトウェア更新のためのバージョン間差分表現方式, 電子情報通信学会論文誌 B, Vol. J89-B, No. 4, pp. 478-487, 2006.

論文 (2) の英訳論文
Ryozo Kiyohara, Mariko Kurihara, Satoshi Mii, and Shigenori Kino: A delta representation scheme for updating between versions of mobile phone software, *Electronics and Communications in Japan*, Vol. 90, No. 7, pp. 26-37, 2007.

(3) 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, pp. 667-678, 2007.

(4) 清原良三, 三井聡, 木野茂徳: 組込みソフトウェア向けバイナリー差分抽出方式, 電子情報通信学会論文誌 D, Vol. J90-D, No. 6, pp. 1375-1382, 2007.

(5) 清原良三, 三井聡, 沼尾正行、栗原聡: 携帯端末 S/W 更新のための高速プログラム圧縮方式, 情報処理学会論文誌, Vol. 50, No. 2, 2009(投稿中）.

■ 国際発表

(1) Terumi Sunaga, Yoshiaki Terashima, Ryozo Kiyohara,Noriharu Suematsu, Tetsuro Itakura, and Yoshio Hirose: Technology and Devices for 4th Generation Mobile Communication Terminals using Software Defined Radio, *Journal of the National Institute of Information and Communications Technology*, Vol.53, No.4, pp.41 – 49, 2006.

(2) Ryozo Kiyohara, Mitsuhiro Matsumoto, Satoshi Mii, Naoki Shimizu, Masayuki Numao, and Satoshi Kurihara: Context-Aware Middleware for Mobile Phone Based on Operational Logs, *IEEE International Conference on Consumer Electronics*, 8-1-5, 2008.

(3) Mitsuhiro Matsumoto, Ryozo Kiyohara,Hidenori Fukui, Masayuki Numao, and Satoshi Kurihara: Proposition of The Context-Aware Interface for Cellular Phone Operation, *IEEE International Conference on Networked Sensing Systems*, poster-11, 2008(採択済）.

(4) Ryozo Kiyohara, Satoshi Mii, Mitsuhiro Matsumoto,Masayuki Numao, Satoshi Kurihara: A Fast Compression Method of Program Codesfor OTA Updating on Consumer Devices, *IEEE International Conference on Consumer Electronics*, (投稿中), 2009.

その他の国際発表一覧

(1) Ryozo Kiyohara, Koichi Nakao, Kumiko Wada, and Takashi Chikayama: PIMOS: A Concurrent Logic Programming Environment, *Proceedings of the ICLP 1990 Workshop on Logic Programming Environments*, Vol. 1990, pp. 63–67, 1990.

(2) Ryozo Kiyohara, and Takashi Chikayama: Parallel Inference System of the FGCS Project - status report, *IJCSLP Workshop on Concurrent and Parallel Implementations*, Vol. 1992, WorkshopB, 1992.

(3) Takashi Chikayama, and Ryozo Kiyohara: Parallel Inference System Research in the Japanese FGCS Project, *Lecture Notes in Computer Science*, Vol. 748, pp. 338–351, 1993.

(4) Katsuto Nakajima, Kazuo Seo, Ryozo Kiyohara, Atsuo Ozaki, and Kazuhiro Abe: A Space-Time Object Functions and Applications, *RWC symposium*, 1994.

(5) Masakazu Furuichi,Katsuto Nakajima, Kazuo Seo, Atsuo Ozaki Ryozo Kiyohara, and Reiko Sato: A space-time object - partial implementation and its evaluation-, *RWC Technical report TR-95001*,pp123-124,Jun 1995.

概要

近年，携帯電話にネットワーク接続機能が付加され，携帯電話で電子メールが扱えるようになってから，飛躍的に携帯電話が普及してきた．携帯電話は常時持ち歩く機器となったため，持っていると便利であると考えられる機能が続々と搭載されている．例えばゲーム機能，電子マネー機能，デジカメ機能，TV 視聴機能などである．

機能が増加してくると，機能の選択手段も選択肢が多くなるために操作性が悪くなる．またユーザによっては不要な機能もあり，自分で好きなソフトウェアを動かしたいという要求も各ユーザごとにでてくる．また開発者の立場で考えると多数の機能を搭載した携帯電話を不具合なしで出荷することは，製品開発サイクルの短さから困難であり，出荷後に不具合を修正する場合も多々ある．即ち以下に示す機能が必要である．

(1) 携帯電話の出荷後にソフトウェアの修正，バージョンアップなどによりソフトウェアを書き換える機能

(2) 好きなソフトウェアをダウンロード，インストールし実行する機能

(3) 自分の使いやすいようにソフトウェアを変更カスタマイズする機能

本論文では，上記のように出荷後にソフトウェアを変更することをソフトウェアアドインと定義する．

このようなソフトウェアアドイン機能は，通常のパソコン等では既に実装され一般に利用されている．しかしながら，携帯電話のようなコンシューマ機器上ではパソコンに搭載されている技術をそのまま利用できるわけではない．例えば，メモリの容量の制限の問題やサービスが利用できない時間を極力短くしなければ電源を急に切られるなどといった様々なケースを想定しなければならないからである．

本論文ではソフトウェアアドインを前記の 3 種類に分け，それぞれの課題に関して検討した．第 2 章で従来の研究動向と課題を整理した上で，第 3 章以降にて 3 種類の課題に対して解決するための方式提案を行い，評価し効果があることを示した．

(1) 携帯電話の出荷後に携帯電話のソフトウェアイメージを書き換えるソフトウェア更新機能で，転送するデータ量を小さくし，書き換え時間を短くすることが必要である．

    (a) 携帯電話ソフトウェア更新のための新旧版での差分データの表現方法
    更新データを如何に小さく表現するかに関しては携帯電話の特性をうまく利用した差分表現形式を利用することにより，更新データを小さく表現しデータ転送量を抑える．

(b) 携帯電話ソフトウェア更新のための書き換え量を抑えるソフトウェア構成法
単純な不具合の修正方法では，多くのコードに影響が及びソフトウェア全体の修正が必要になることを示し，提案する方式により，適切なモジュール分割を行うことにより局所の修正は局所に閉じることができ，その結果ソフトウェアの書き換え時間を短くし，かつ更新用データを小さくする．

(c) 携帯電話ソフトウェア更新のための携帯端末上での高速なソフトウェア圧縮方法
携帯端末上でプログラムデータを圧縮する際には，開発環境上での圧縮情報を活用することにより高速なデータ圧縮が可能となり，ソフトウェア書き換え時間を短くする．

(2) 携帯電話の出荷後にユーザの意思でソフトウェアをダウンロードし実行する機能

(a) 携帯電話上でのインタプリタ方式で実行する代表的言語 Java に関して携帯電話特有の条件を満たした上での高速実行方式
Java の高速化手法に関しては効率的な動的コンパイル手法を提案する．提案手法では，メモリの効率的利用という観点でコンパイル対象を静的，動的に絞りこむ手法で効率的に高速化ができる．

(3) 携帯電話の出荷後に個々のユーザに合わせた形で操作性を向上するためのカスタマイズ機能

(a) 携帯電話の位置取得機能などのコンテキスト情報を利用したユーザ操作性の向上方式
コンテキストを利用したユーザ操作性の向上に関しては，操作性の指標を定義した上で，ユーザが利便性を感じやすい部分に焦点を絞った操作性の向上方式を提案し，その有効性を示す．

目次

1　序論　1

1.1　研究の背景と目的 1

1.2　本研究の特長と成果 3

1.3　本論文の構成 5

2　ソフトウェアアドイン技術　7

2.1　携帯電話のソフトウェア更新技術 8

2.1.1　バイナリ差分抽出技術 9

2.1.2　ソフトウェア構成技術 10

2.1.3　携帯電話上でのプログラムコード高速圧縮技術 12

2.2　携帯電話アドインソフトウェアの実行技術 13

2.2.1　アプリケーション管理技術 13

2.2.2　携帯電話 Java 高速化技術 17

2.3　携帯電話ユーザによるカスタマイズ技術 19

2.3.1　携帯電話ユーザ操作予測技術 19

3　バイナリ差分抽出方式　21

3.1　バイナリ差分とは 22

3.1.1　差分表現形式 22

3.1.2　差分サイズを決める要因 24

3.1.3　携帯電話ソフトウェアの差分特性 24

3.2　表現形式の改良提案 27

3.2.1　マクロコピー方式の導入 27

3.2.2　アドレス変換方式の導入 30

3.3　提案方式の評価 32

3.3.1　マクロコピー方式の効果 32

3.3.2 アドレス変換方式の効果 34
3.3.3 ユーザ利便性評価 35
3.4 ハイブリッド方式の提案 37
3.4.1 提案二方式の課題 37
3.4.2 提案アルゴリズム 38
3.5 ハイブリッド方式評価 40
3.5.1 M32R 携帯電話ソフトウェアでの詳細評価 41
3.5.2 他 CPU アーキテクチャへの適用評価 42
3.6 本章のまとめ 44

4 ソフトウェア保守のためのソフトウェア構成法 47

4.1 モジュール分割 48
4.1.1 モジュール分割概要 48
4.1.2 モジュール分割と修正量の関係 50
4.2 評価 52
4.2.1 モジュール分割の分割数評価 52
4.2.2 空き領域評価 53
4.2.3 評価のまとめと課題 54
4.3 改良方式の提案 54
4.3.1 緩やかなアドレス固定方式 54
4.4 改良方式評価 55
4.4.1 差分データ量の評価 55
4.4.2 ベクタテーブルの効果 57
4.4.3 ブロックの効果 57
4.5 本章のまとめ 58

5 携帯電話ソフトウェア更新における高速プログラムコード圧縮方式 59

5.1 ソフトウェア更新と Byte-Pair 圧縮 60
5.1.1 携帯端末のメモリ構成 60
5.1.2 Byte-Pair 圧縮方式 60
5.1.3 圧縮データと差分更新 62
5.2 提案方式 64
5.2.1 基本的なアルゴリズム 64
5.2.2 実現方式 65

5.2.3 提案方式の性能測定と評価 66
5.3 改善提案 67
5.3.1 辞書データの圧縮 68
5.3.2 差分辞書データ 69
5.3.3 ハイブリッド方式 70
5.4 本章のまとめ 72

6 Java 実行高速化方式 75

6.1 携帯端末の特性と機能 76
6.1.1 JIT 方式による高速化の課題 76
6.1.2 通信量に対する影響 76
6.1.3 メモリ量に対する影響 78
6.1.4 ユーザ応答時間に対する影響 79
6.2 提案方式 79
6.2.1 静的な解析 80
6.2.2 動的な解析 82
6.3 評価 83
6.3.1 クラスライブラリ・メソッドの選定の有効性 83
6.3.2 静的，動的な解析による判定の有効性 85
6.3.3 試作実装を用いた評価 90
6.4 本章のまとめ 93

7 携帯電話のユーザ操作予測の有効性 97

7.1 コンテキストを利用した携帯電話の操作予測 98
7.1.1 実験環境と操作履歴 99
7.1.2 データの分類 100
7.1.3 予測処理 100
7.2 評価 102
7.2.1 予測に関する評価 102
7.2.2 操作性に関する評価 103
7.3 操作性向上のための改良方式提案 106
7.4 改良方式の評価と考察 109
7.5 本章のまとめ 109

8 結論 **111**

謝辞 **117**

研究業績一覧 **127**

図一覧

2.1 局所の修正は局所にのみ影響する構造 11
2.2 モジュール分割とジャンプベクターテーブル 12
2.3 アプリケーションマネージャ 15
2.4 状態表の例 16

3.1 新旧版の差分パターンの例 23
3.2 差分コマンドの例 23
3.3 ずれによる差分パターンの例 25
3.4 ずれによる差分コマンドの例 26
3.5 修正の影響 27
3.6 修正による全体への影響 28
3.7 マクロコピー表現導入によるコマンドの例 29
3.8 様々なずれパターン 30
3.9 アドレス変換例 31
3.10 ずれている部分で SKIP コマンドがでる例 33
3.11 アドレス変換の逆効果例 38
3.12 マクロコピー領域の発見方法 39
3.13 アドレス変換すべきかの判定方法 40
3.14 差分コマンド生成 41
3.15 差分データサイズグラフ 43

4.1 1 改修あたりの修正量とモジュール数の関係 52
4.2 ブロックを導入したモジュール分割 55

5.1 圧縮 ROM イメージ 60
5.2 Byte-Pair 圧縮方式 61
5.3 差分更新 62
5.4 提案方式 66

5.5 提案方式の実現 67
5.6 差分辞書データ方式 70
5.7 辞書中の順序情報 71
5.8 圧縮時間とトークン数 72
5.9 辞書サイズと圧縮時間 73

6.1 ゲーム操作画面 77
6.2 解析とプロファイリング 80
6.3 高速シューティングのメソッド実行時間分布 85

7.1 操作履歴の例 100
7.2 操作ログの三次元マップ 101
7.3 Web メールの予測モデル 102
7.4 航空会社の予測モデル 103
7.5 乗り換え案内の予測モデル 104
7.6 航空会社を起動したときの行動パターン 104
7.7 頻度と操作回数，操作時間の関係 105
7.8 困った時の機能推薦 108
7.9 ソフトウエア構成図 108

表一覧

3.1 携帯電話ソフトウェアの複数バージョン間での差分量 32
3.2 マクロコピー適用内の COPY コマンド数 33
3.3 アドレス変換方式における差分量 34
3.4 差分分散状況 (差分小 1 → 2) 35
3.5 差分分散状況 (差分大 1 → 2) 35
3.6 DATA コマンド数 36
3.7 実データでの差異位置 36
3.8 実データでの差分サイズ 36
3.9 差分コマンド数 42
3.10 差分データサイズ (bytes) 42
3.11 ARM 差分データサイズ (bytes) 43
3.12 SH 差分データサイズ (bytes) 44
3.13 Windows 端末アプリケーション差分データサイズ (bytes) 44

4.1 メモリ容量への影響 49
4.2 Java ベンチマークプログラムの性能劣化率 50
4.3 モジュール分割数と関数の参照数の関係 53
4.4 モジュール分割と参照数 56
4.5 モジュール分割と差分サイズ 58

5.1 ROM イメージサイズ (KByte) 64
5.2 変更ブロック数とページ数 64
5.3 基本性能 64
5.4 更新時間（秒） 65
5.5 提案方式の圧縮性能 67
5.6 提案方式更新時間（秒） 68
5.7 提案方式による更新データ増加 68

5.8 提案方式による更新データサイズ 68
5.9 辞書データ圧縮サイズ 69
5.10 トークンとペア 71
5.11 更新時間と閾値 k 72

6.1 コンパイルコードのサイズ 78
6.2 静的な解析によるネイティブコード変換 82
6.3 動的な解析に用いるプロファイリング対象 83
6.4 機能毎の実行時間の割合 86
6.5 ゲーム A,B,C: 選択されたメソッドの実行時間とサイズ 90
6.6 フレームワークによる判定 91
6.7 インタプリタに対する性能比 92
6.8 ゲーム A：各メソッドの実行時間とサイズ 94
6.9 ゲーム B：各メソッドの実行時間とサイズ 95
6.10 ゲーム C：各メソッドの実行時間とサイズ 96

7.1 取得した履歴 99
7.2 履歴取得する操作の種類 100
7.3 分類モデルの評価 101

第 1 章

序論

1.1 研究の背景と目的

近年，携帯電話にネットワーク接続機能が付加され，携帯電話で電子メールが扱えるようになってから，飛躍的に携帯電話が普及してきた．また，携帯電話は時計，眼鏡のように常時携帯する機器の一つとなり身につけて歩くといっても過言でない状況にもなっている．そのため，持っていると便利と考えられる機能が続々と携帯電話に搭載されてきている．例えばゲームの機能や，電子マネーを扱う機能，デジカメ機能，TV 視聴機能である．

機能が増加してくると，機能の選択手段一つをとっても選択肢が多くなるため，操作数や視認した上での判断時間というユーザビリティの点で操作性が悪くなる．携帯電話は駅や交差点など屋外の様々な環境で利用するため，操作性が悪くユーザを注意散漫にさせることはできる限り避けなければならない．

また，あるユーザにとっては不要な機能や，自分の所望のソフトウェアを動作させたいなど各ユーザごとに異なる要望もでてくる．端末の出荷台数は 1 つの機種でも 100 万台を超えることも多く，個々のユーザの要望に合わせてカスタムメイドの機種を作ることはコスト的には一部の例外を除いて厳しい．そこで，ユーザ自らがカスタマイズできるかまたは自動的にユーザの要望を推測するような技術が必要となる．

一方で，開発サイドから考えると多数の機能を搭載した携帯電話に不具合なしで出荷することは困難となっており，出荷後に不具合を修正する必要性もでてきている．不具合の修正を販売店などで実施するにはユーザの手間がばかにならないため，ネットワークを経由して実施できる必要もある．

本論文では，これらのニーズを解決するソフトウェアアドインに関してその課題を解決する方式を提案する．携帯電話の出荷後に携帯電話内部のソフトウェアを追加，変更することをソフトウェアアドインと呼び，ソフトウェアアドインを以下の 3 種類に分ける．

(1) ソフトウェアの不具合の修正や OS の機能拡張などで，ソフトウェア保守を目的としたソフトウェア書き換え．主にメーカやキャリアが必要に応じて実施するサービスである．

(2) ユーザが自らの意思でソフトウェアをダウンロードし，インストールすることによりソフトウェアを追加，変更すること．例えば Java[1]アプリケーションであったり，BREW[2]アプリケーションという形でキャリアからサービスされている機能のことである．

(3) ユーザが自らの意思で自動的または手動で携帯電話を使いやすいようにカスタマイズすることでソフトウェアを出荷時の状態から変更することである．例えばショートカットキーをユーザが変更する機能のことである．

このようなソフトウェアアドイン機能は，通常のパソコン等では既に実装され一般に利用されている．ソフトウェア保守の機能の例はユーザがメニューを選択すると，自動的にネットワークを利用してサーバに問い合わせを行いソフトウェアの更新が必要かどうかを問い合わせ，必要であればダウンロードしインストールする．このような機能が各アプリケーションや OS に実装されている．また，自由にソフトウェアをダウンロードし，実行することもパソコンでは広く利用されている． Java はパソコンのアーキテクチャや OS に依存しないインタプリタ実行を主とした実行環境として利用されている．また，ユーザごとにカスタマイズするようなこともパソコンでは簡単にできる．

しかしながら，携帯電話のようなコンシューマ機器上ではパソコンに搭載されている技術をそのまま利用できるわけではない．上記 3 種類のソフトウェアアドイン機能ごとの課題を以下に示す．

(1) 携帯電話のソフトウェアは，瞬時の起動の必要性などから XIP[3]というフラッシュメモリ上のコードを直接実行するモデルでソフトウェアが実装されていることが多い．フラッシュメモリは最低限の容量しか持たないため，パソコンのようにプログラム実行中にプログラムイメージを書き換えることはできない．そのため，ソフトウェアの書き換えはできる限り短時間で終えなければならないという課題がある．また，ネットワークを使うという観点からは如何にデータ転送量を減らすかという課題もある．

[1]Java および Java に関連する商標は米国 Sun Microsystem 社の商標または登録商標である

[2]BREW および BREW に関連する商標は米国 Qualcomm 社の商標または登録商標である

[3]eXecute In Place

(2) 自由にソフトウェアをダウンロード実行するモデルでもインタプリタ実行を行うようなケースではメモリと CPU の能力の観点で実行速度そのものが問題になるケースが多く，高速化が重要な課題となる．

(3) ソフトウェアのユーザごとのカスタマイズ機能においては，移動しながら片手で利用し，画面も小さい携帯電話ではパソコンと同様な手法ではユーザは不満を覚える．そのためユーザに意識させずに操作性を向上する方式が求められる．

本論文では，これら各課題に関してそれぞに有効な手法を提案・評価し，その有効性を示すことを目的としている．

1.2 本研究の特長と成果

前節に示した研究の背景と目的の元で，具体的には以下に示す成果をあげることができた．

(1) 携帯電話のソフトウェア更新方式の提案
携帯電話のソフトウェア更新方式における課題は，ネットワークを利用してデータを送信する際のデータ量の削減と携帯電話上でのデータの書き換え時間の削減の 2 点である．それぞれの課題における提案の特長と成果を以下に示す．

(a) 携帯電話のソフトウェア更新のための送信データの削減方式に関する提案
最近の携帯電話のソフトウェアの容量は全体で小さいものでも 32M バイト程度であり，大きなものでは 64M バイト程度ある．また，ソフトウェアの更新は全端末に対して短い期間でデータを配信する必要性から網への影響を考慮して，できる限り送信量を削減する必要がある．

携帯電話上で既に稼動しているソフトウェアイメージ（以下，旧版ソフトウェアイメージと呼ぶ）と工場などで動作確認を実施して，これから携帯電話のソフトウェアを置き換えようとする新しい実行イメージ（新版ソフトウェアイメージ）のバイナリ差分を抽出してその差分情報を送ることが考えられる．この差分を如何に小さく表現するかが課題である．

そこで，携帯電話のプログラムコードでは，新版と旧版の差の多くがアドレスのリファレンス関係にあるという特性を利用して，バイナリ差分が小さく表現できる方式を提案，評価し効果があることを示した．

(b) 携帯電話のソフトウェア更新のためのデータの高速書き換え手法に関する提案
携帯電話のソフトウェアの新版と旧版の差が小さくとも，フラッシュメモリ上

の位置が変わるとフラッシュメモリはすべて消去して新しく書き換えなければならない．これには最近の携帯電話のように 64M バイトものメモリを持っていると数十分かかるケースも珍しくない．ユーザの視点にたつと携帯電話のサービスが受けられない時間が数十分というのは問題である．とくにユーザが意識しないうちにソフトウェアの更新をしようとすると，ユーザにとっては理解できずにソフトウェアの書き換え中に電池を抜くなどの問題が発生するケースもあると考える．

そこで，この書き換え時間を小さくする方法として，まず開発環境上での工夫により，局所の修正は局所にのみ影響するという手法を提案し，評価の結果効果があることを示した．

(c) 最近の NAND 型のフラッシュメモリを利用したソフトウェアではいわゆる仮想記憶技術を活用するデマンドページング方式の採用も考えられ，ソフトウェアの高速展開可能な圧縮方式を利用するケースも想定される．よって，この観点にも着目した端末上での高速な圧縮方式を提案，評価し効果があることを示した．

(2) 携帯電話のアドインソフトウェアである Java のプログラム実行高速化方式の提案

Java は一般にプログラムをコンパイルし Java バイトコードに置き換える．これを携帯電話上にダウンロードし，インタプリタで逐次実行する．そのため実行速度が遅いという欠点がある．しかし，携帯端末上の各種リソースにアクセスすることもインタプリタを経由するという点で守られ，セキュアであるという点で優れているため，誰でもがプログラムを開発し携帯電話上で動作させることができるという利点がある．

一方， BREW の方式は C 言語で記述する通常のアプリケーションと同じ位置づけになるためセキュアにはならないが，他のアプリケーションと同様な高速な実行が可能である．セキュアを保証するためにキャリアなどがコードレビューをするといった認証を得たもののみがプログラムを書くことが許されるというスタイルになり，誰もが自由にプログラムを書けるわけではない．

そこで，セキュアでかつ高速に実行できる環境として， Java の実行処理を高速化する手法を検討した．一般に Java を高速に実行する手法としては JIT[4]方式が提案され，PC 上のブラウザなどで実装されている．しかし， JIT では最適にコンパイルすれ

[4]Just In Time compile は， Java のバイトコードを H/W に依存した機械語命令列にコンパイルしてから実行する

ばするほどコードの占めるメモリが大きくなるという欠点がある．携帯電話のようにメモリの余裕がないケースにはそのまま適用することができない．

また， JIT 方式ではコンパイルを実行時に行うため，コンパイルしている時間は動作できないという問題がある．起動時にすべてコンパイルすると起動に時間がかかるし，実行時に逐次コンパイルしていると携帯電話のようにインタラクティブ性の高いアプリケーションでは瞬停が多くなり操作性が悪くなる．

そこで，メモリを節約しつつ，必要な部分はコンパイルし，かつ瞬停をさせない手法を提案評価し，効果があることを示した．

(3) 携帯電話のユーザ操作のユーザビリティ向上方式の提案
出荷後のソフトウェアを変更するという意味で最も多くの機種で可能なことは，ショートカットキーの割り当てである．多くの人が共通で頻繁にアクセスすると想定されるメールやブラウザの起動はキーが既にアサインされ 1 回の操作でできる場合が多い．これに対して，個人ごとによって違う機能のをアサインすることができるのがショートカットキーであり，多くの場合は 2 タッチで所望の機能が起動できるようになる．しかし，ショートカットキーも増加すると 2 タッチというわけにはいかない．

そこで最近の携帯電話のように GPS 機能で位置が取得でき，加速度センサでユーザの状態が推定できるような場合に，ユーザのコンテキストに応じてユーザの所望する機能をユーザの過去の履歴から推測し，候補を提示する手法を提案し，その有効性を示した．

1.3 本論文の構成

本論文における第 2 章以降の各章の構成を以下に紹介する．

第 2 章 本章では，携帯電話のソフトウェアアドインを実現するための各要素技術につき，これまでの研究の流れについて簡単に述べる．その上でそれぞれの項目につき，本研究の必要性を述べるとともに，本研究の位置づけを明確にする．

第 3 章 ソフトウェア保守のための新旧バイナリ差分抽出方式に関して述べる．基本的な差分抽出方式に加え，提案する複数の差分抽出方式に関して述べた後，評価を行う．提案した複数の差分抽出方式のハイブリッド方式を提案し効果があることを示した上で，携帯電話に限らず組み込み機器一般に効果のある方式であることを示す．

第 4 章　ソフトウェア保守のためのソフトウェア構成法に関して述べる．提案方式は，ソフトウェアイメージをモジュールに分割し，参照関係をテーブル経由で行う．適切なモジュール分けを行うことにより，実行時のオーバヘッドと余分に必要となるメモリ量がわずかであることを示し，書き換え量と差分の大きさという観点から有効な手法であることを示す．

第 5 章　最近の NAND 型のフラッシュメモリを利用した携帯電話において，デマンドページング方式を採用していた場合のソフトウェア更新機能の課題である携帯端末上の高速プログラム圧縮方式に関して方式提案を行い，余分な情報を更新用の差分データにつけることなく，端末上で高速にプログラムを圧縮する方式に関して述べ，評価によってその有効性を示す．

第 6 章　携帯電話上でのアドインソフトウェアの高速な実行のための方式として，Java 動的コンパイル方式を提案する．携帯電話特有の制約条件と特長あるアプリーションに焦点をあて，評価を行いその有効性を示す．

第 7 章　携帯電話の操作性の向上のために出荷後に設定などの状態を自動的に変更するためにユーザのコンテキストに応じユーザの操作を予測し，候補を抽出する方式に関して提案する．

第 8 章　本章では，1 章から 7 章までのまとめとして，研究の流れを概説したのち，今後の課題に関しても概観する．

第 2 章

ソフトウェアアドイン技術

本章の概要

本章では携帯電話のソフトウェアアドインを実現するための各要素技術につき，これまで研究されてきた流れについて簡単に述べた上でそれぞれの項目につき，本研究の必要性を述べるとともに，本研究の位置づけを明確にする．

2.1 携帯電話のソフトウェア更新技術

近年，無線によるネットワーク接続が可能な携帯電話が普及してきた．これらにはブラウザとメール以外にもカメラ，赤外線 I/F， JavaVM まで搭載している機種もある．そのため，携帯電話上に搭載するソフトウェアの規模が急激に増大している．しかしながら，開発サイクルは変わらずに，従来と同じ期間で開発することが要求されている．即ち，十分な試験期間を設けることができず，障害がない状態で携帯電話を出荷することが困難になってきており，実際に市場に出てから不具合が多数報告されている [1][2].

そこで出荷後に，ソフトウェアの更新を簡単にできることが求められてきており，工場ではなくショップでの書換えサービスや，無線を利用したサービスも導入されている [3][4].

また，携帯電話に限らず組込機器一般に対するソフトウェアのリモートメンテナンスや，バージョンアップに関して IPA[5] やトロン協会 [6] などでも研究されている.

携帯電話に関しては，過去の障害においては多くのユーザにとっては困らないものの，万が一を考えて修正しておいた方が良い程度のものが多い．そのため，ユーザがトリガをかけてソフトウェアの更新を行うモデルが導入されている [3][4][7]．その場合，更新時間中は全く機能が使えなかったり，一部の機能しか使えないために，ユーザの利便性が悪くなる可能性が高い．そのため，時間がかかるほどユーザが余計な操作をしたり，電池が切れたりする可能性などが高くなる．故に，ユーザへのサービスを低下させないようにするためにも短時間でソフトウェアを更新することが必須となる．しかし，携帯電話のソフトウェアは 2 次記憶からメモリ上にロードして実行する形式ではなくあらかじめアドレス解決済みのコードを保持している．そのため，わずかな修正の影響が全体に及ぶ.

そこで，短時間でソフトウェアを更新するためには以下の 2 点が課題となる.

- 通信路上でのデータ転送時間を短くする．即ち，転送データ量を小さくすること
- 携帯電話上で実際に書き換える時間を短くすること，即ちデータの修正がなるべく局所に集中すること

前者を解決するためには，携帯電話上のソフトウェアの現在の版と新しい版との差分情報を送ることが考えられ，この差分を如何に小さくするが課題となる．後者を解決するためには局所の修正は局所にしか影響を及ぼさないソフトウェア構造とすることが課題となる．また最近の NAND 型フラッシュメモリを利用したデマンドページング方式などではプログラムデータを圧縮して保存していることも多く，圧縮データのソフトウェア更新方式も大きな課題である.

2.1.1 バイナリ差分抽出技術

一般的な差分表現技術

ソフトウェアの差分抽出技術は unix の diff [8][9] が代表的なアルゴリズムである． diff のアルゴリズムはテキストファイルを対象にしており，出力された差分を適用することも想定したコマンド列を出力することもできる． しかし，対象がバイナリファイルの場合はそのままでは使えない．

このような差分抽出アルゴリズムでバイナリファイルに対して適用可能にした例として bdiff[10] がある．これらは組込み機器に適用するように想定したものではなく，実行時間やメモリの利用量，差分の表現サイズなどに関して課題が残る． vdelta[11] は差分圧縮という考え方で，圧縮同様の方法で差分を作成するアプローチを提案し，効果的な方法であることを示した．旧版の続きに新版をつなげて圧縮するという考え方である．単なる圧縮は，たまたま旧版の大きさが 0 であった場合に該当するという考え方で進める．

静的な差分抽出ではないが，ネットワークで結ばれた端末間でソフトウェアの差分を動的に取得することにより，差分更新をする rsync [12] が提案されており，バージョン管理をしていなくても効果的に更新可能である． 以上は差分抽出に関する代表的なアルゴリズムであり，汎用性もあり有効な手法である．

一方，差分表現形式としては gdiff[13] が W3C[14] にて提案されている．差分を適用するためのコマンドを COPY と DATA の 2 つを主として定義して，この 2 つのコマンドで差分を表現する． COPY コマンドでは，コピーするサイズを指定する必要があるが，そのサイズや位置を指定するビット長によってコマンドを変えるというような工夫がされている．この gdiff をさらに発展させ vcdiff[15] ではコマンドテーブルを実際のデータにあったように可変にするというような工夫を行うことにより差分を小さく表現でき，さらに差分圧縮の考え方を利用して，新たに書き換えたデータをも利用するという工夫をしている．これらは汎用的な差分表現形式であり一般に利用されている．

組み込み機器向け差分抽出，表現技術

前記のいずれの方法も汎用的な差分抽出および表現方法を目指しており，様々な制約が入る組み込み機器上では差分データサイズが充分に小さくならないケースがある．例えば，組込みソフトウェアの差分抽出方式として，適用時にワークメモリがないことを想定した上で差分を計算する方式 [16] が提案されている．ワークメモリがないことを前提に，どう動作させれば制限された環境で動作するかを解いてから差分を作成する方式であり， RAM がほとんどない環境では有効な手法と考える．

特に，組込みソフトウェアの差分更新では差分の表われ方に一定の特性がある．このような組込みソフトウェアの中でも実行イメージという点に着目した研究としては文献 [17][18] がある．1 行の修正により，コンパイル時のレジスタアサインのずれが起こり，修正した点の周辺に影響を及ぼす場合がある． このようなケースには，機械語では一定のビット変化が周辺の命令（レジスタを使う命令）で起こる．レジスタアサインが変わったことによる一定の数値をオフセット値として使うことにより，差分が小さく表現できる．本方式は修正をした点の周辺に限定された差分の最適化である．

参照先の位置ずれによる影響を自動的に復元することで差分データサイズを小さくする方式 [19] も提案されている．

本論文では，アドレス部に差分が集中することを想定してデータ変換をかけ，差分の出る部分を集中させる方法（アドレス変換方式) や，全体のずれをマクロ的に見ることにより差分を小さくする手法 (マクロコピー方式) を提案する．

いずれの方式もプログラムという観点から有効な手法であるが，大規模化したソフトウェアではプログラム部とデータ部が混在しており，これらの手法が必ずしも有効とは限らない．そこで，さらにこれらの手法をソフトウェアの特性に応じて使いわけることにより差分量を抑えるハイブリッド方式を提案する．

2.1.2 ソフトウェア構成技術

携帯電話のソフトウェア構造とリンク方式

携帯電話のソフトウェアは一般に OS，各種デバイスに依存するデバイス部，携帯電話上で各種サービスをするミドルウェア部，ユーザ I/F であるアプリケーション部に分けることができる．これらのプログラム部と電源を落とした状態でも保持するべき各種設定情報やユーザデータとを不揮発性のメモリ上に配置する．

多くの場合，μ ITRON[6] などのリアルタイム OS 上に実現し，効率的な動作に主眼が置かれている．そのため，ソフトウェアの更新は想定せず， OS も含めてすべてリンク済みのコードをフラッシュメモリ上に配置する．また設定情報などのユーザのデータとなるものが不揮発性の別の領域におかれデータとして扱われることになるのが一般的である．

この場合，後からソフトウェアを修正して新しいバイナリイメージを作成するとコード全体がずれて参照関係のあるアドレスの絶対位置や相対位置がずれてしまうことになる．そのため，影響が修正によって変更された部分だけでなくソフトウェア全体に広がる可能性がある．

そこで，パソコンの OS のようにダイナミックローディング方式の採用も考えられる．しかし効率的な動作， RAM の容量などコスト的なことも考え，現状の構造や開発プロセ

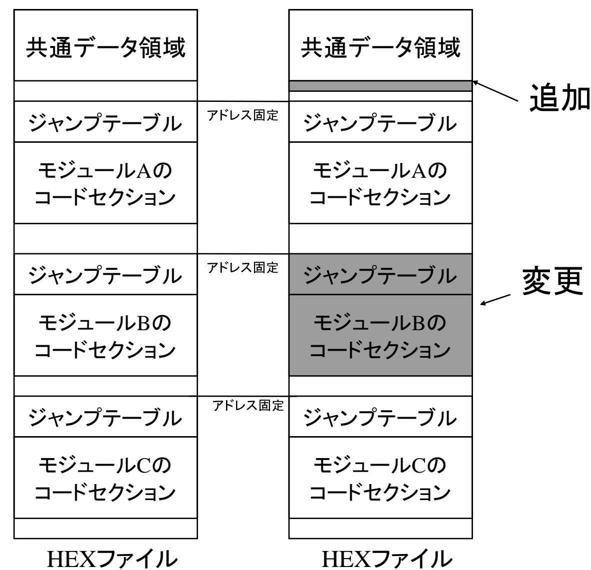

図 2.1: 局所の修正は局所にのみ影響する構造

スを変えることなくソフトウェア更新を行う方法を検討することとした.

ソフトウェア更新方式

リンク済みのコードに対して位置ずれを起こさないで不具合を修正する方法としてはバイナリパッチ方式がまずは考えられる．この方式では，不具合のある部分に対して，修正コードの配置されたエリアにジャンプするコードを入れ，修正コードの最後の部分で元に戻るジャンプコードを入れることになる.

一般に携帯電話は逐次出荷される中で，小さな不具合の改修を入れていくことが多く，出荷バージョンによってパッチを入れる部分が異なることになったり，出荷時にわざわざパッチコードを入れるなどの工夫が必要になる．これはソフトウェアの管理上も好ましくなく，複数の不具合が絡みあうようなケースにおいてはソフトウェアのバージョン管理で破綻しかねない.

そこで，一定のコードの集合をモジュールと定義し，リンク済みのコードのモジュールの間に空き領域を設けておく．不具合修正の結果，削除や挿入が入っても影響を空き領域で吸収する方法が考えられる (図 2.1)．また，モジュール間の参照はジャンプベクターテーブルを利用し，命令位置の移動が他の領域に及ばないようにする (図 2.2).

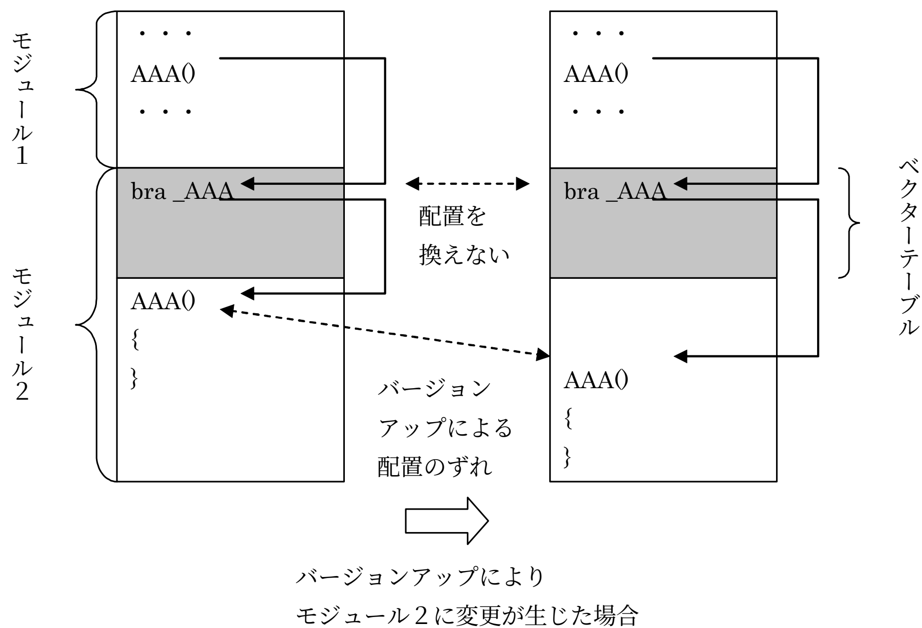

図 2.2: モジュール分割とジャンプベクターテーブル

2.1.3 携帯電話上でのプログラムコード高速圧縮技術

最近，携帯電話向けでは，NAND 型フラッシュメモリ上にプログラムコードを置き，デマンドページング方式を採用した OS も登場している [20]．効率的に NAND 型フラッシュメモリを利用するには圧縮プログラムコードに対応したソフトウェア更新技術が必要となる．

また，圧縮方式においては，古くから様々な研究がされ，実用化されているものは多い．圧縮には多少時間がかかっても，圧縮率がより高く，展開時間がそれなりに早ければ良い場合が少なくないため [21]，圧縮率の高さと高速な展開時間に関して中心に研究されている [22]．また，実行プログラムコードの圧縮という観点からも様々な方式が研究され，多くの方式が提案されている [23]．

NAND 型フラッシュメモリ上に圧縮プログラムコードを置き，デマンドページング方式をとる方式に適した圧縮方式の一つとして Byte-Pair 圧縮方式 [24] の利用が提案され，効果的であることが示されている [25]． Byte-Pair 圧縮方式では必要な部分だけの展開も可

能であり，ソフトウェアの更新を考慮しなければこの手法は非常に有効な手法である．しかし， Byte-Pair 圧縮方式は圧縮に時間がかかるという課題がある．

展開や圧縮における高速化の必要性を想定し，携帯端末で良く使われる CPU アーキテクチャに依存してはいるものの，効率の良い圧縮方式に関する研究開発も行われている [26].

文献 [27] も Byte-Pair 圧縮に対して部分展開可能であることから文字列の比較の研究を行っている．文献 [28] は圧縮データを展開せずに文字列比較をするという観点から， Byte-Pair 圧縮の圧縮処理高速化に関する研究を行っている．この研究では Byte-Pair 圧縮の速度の遅さを 1 回の走査で最も多いペアだけを発見して使っていない 1 バイトに置き換える処理そのものにあると考え，複数のペアを選択して同時に置き換えていくことで高速化を図っている．しかしながら，圧縮率が少し落ちるという問題が発生する．

そこで，本論文ではソフトウェア開発の効率化という観点からも既存の Byte-Pair 圧縮方式の仕様を変更せずに高速圧縮する手法に関して述べ，実際の携帯電話ソフトウェアイメージで評価し効果を示す．

2.2 携帯電話アドインソフトウェアの実行技術

携帯電話のアドインソフトウェアをダウンロードし，書き換えが終わったとしても，実行時にアドインでないソフトウェアと同等の性能が必要であり，かつ同等にセキュアである必要がある．本節では，セキュアに高速にアドインソフトウェアを実行する技術を概観する．

2.2.1 アプリケーション管理技術

アドインソフトウェアの代表として， Java 実行環境がある． Java 仮想マシンをサポートする携帯電話 [29] では， Java 言語で書いたアプリケーションプログラムに関してのみアドインが可能となるが，制約条件も多く実行したいことが常にできるとは限らない．また，特定のハードウエアを制御するようなドライバを記述することも Java では困難である．

そこで，ユビキタス端末として携帯電話を活用するためにネイティブコード (C/C++) で開発したアプリケーションやドライバをインストール可能にする必要がある．携帯電話のソフトウェアは組込みソフトウェアの一種であり，組込みソフトウェア特有の問題がある．

外部からアプリケーションをインストールするためには，そのアプリケーションが利用するリソースの競合をどのように処理するのかという課題がある [30]． 後から追加するア

プリケーションの開発者は，他のアプリケーションがどのようにリソースを使うのかを把握して開発しているとは限らない．そのため，競合制御が重要な課題となる．

次に，セキュリティの問題がある．セキュリティに関しては外部からプログラムを入れるという行為そのもので，様々な脅威に対する弱点を作ることにつながる． そのため，ハードウエアプラットフォームから考えて行く必要がある [31]．また，コードをチェックすることにより証明書を付与し，安全であることを示すための効率的な方式に関しても研究されている [32]．

アプリケーション管理の課題

最近の携帯電話ではマルチタスク機能がサポートされた機種が出てきてはいるが，そのほとんどは 1 画面を１つのアプリケーションが占有するタイプの携帯電話である． 例えば音を出すようなアプリケーションがあるとする． このアプリケーションは，起動中に電話がなってしまえば，たとえ先に起動されていようとも音を出してはいけない． このようにその状態によって動作を変える必要がある． このような競合管理には以下に示す課題がある．

起動制御 フォアグラウンドになれるアプリケーションは一つであり，しかもアプリケーションによってはメモリリソースを大量に消費するため，アプリケーション間での起動制御を行う必要性もある． また，ユーザの入力をトリガにしてアプリケーションが起動される場合もあれば，外部からのイベントをトリガにする場合もあり，タイミングを考慮した設計が重要となる．

各アプリケーションは，起動のときには自らアプリケーションマネージャに起動要求を出し，起動条件が満足されていれば，起動される．リソースのアクセスに関しても，アプリケーションマネージャに要求を出し，アクセス条件が満足されていれば，リソースにアクセスできる． その例を図 2.3に示す．アプリケーションの数を n とすると，単純な実装をすると， n 個のアプリケーションが自己の 2 重起動要求もも含めて考えると n 個のアプリの起動要求に対する動作を表現する必要があり，計 n^2 の動作を示す必要がある． またアプリの状態によって起動の許可，不許可をする場合もある． 起動に関連する状態数を p とすると， pn^2 の動作を示す必要がある．アプリケーションを m 個追加するとすれば，起動制御を行うために次式に示す情報を追加する必要がある．

$$kidou(n,m,p) = p(n+m)^2 - p(n^2) \quad (2.1)$$

$$= 2pnm + zm^2 \quad (2.2)$$

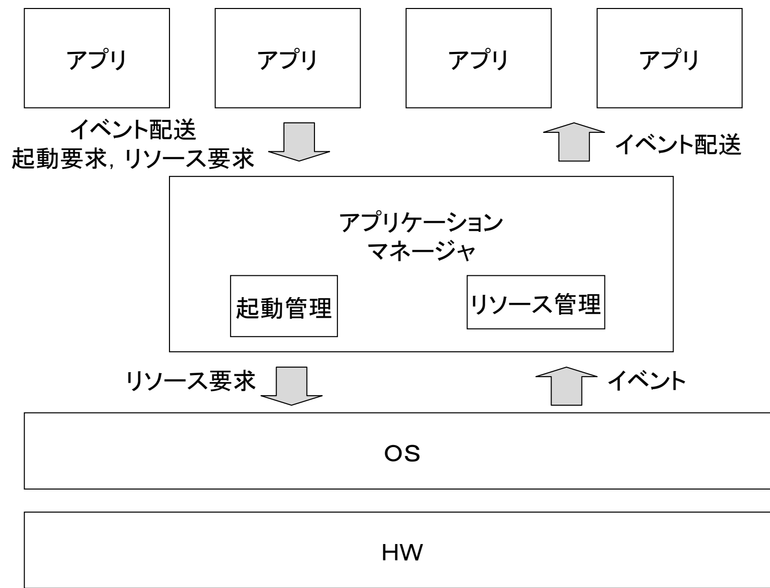

図 2.3: アプリケーションマネージャ

アプリケーションの追加想定数や，メモリの容量も考えるとたかだか一桁であり，$n >> m$ と想定できる． 従って約 $kidou(n, m, p) = 2pnm$ 個の動作に関する追加がアプリケーション自身の他に必要になる．

アプリケーションの状態ごとのイベント制御 アプリケーションは多くの場合，画面を持ち，画面ごとにキーイベントを受け付ける，受け付けないとか，外部イベントに対してどう動作するべきかなど決めなければならない． このような動作は図 2.4のような状態遷移表で示される． これをプログラムコードに変えて実行する．

一つのアプリケーションが持つ平均の状態数を s とする． また起こりうる平均のイベント数を e とすると，次式で示す状態遷移の動作を示すコードがアプリケーションのコードに含まれる必要がある．

$$jousen(s, e) = se \tag{2.3}$$

例えば，アプリケーション数が 20 個あり，起動の制御に関連する状態数の平均が 10 個であり， 2 個までアプリを追加可能とすると， 800 個の動作を記載できるようにしておく必要ある． 実際にはもっと多くの動作を管理しておかなければならない．また，アプリごと

状態	非起動時	起動要求 応答待ち	起動中	リソース獲 得要求中	. . .
イベント					
起動要求	. . . .	. . . .	. . . .	. . . .	
起動要求 応答	. . . .	. . . .	. . . .	. . . .	
リソース獲 得要求	. . . .	. . . .	. . . .	. . . .	
. . . .	. . . .	. . . .	. . . .	. . . .	
. . . .	. . . .	. . . .	. . . .	. . . .	
電源キー 長押し	. . . .	. . . .	. . . .	. . . .	

図 2.4: 状態表の例

には，状態数を 30 個程度としてもイベントが 20 あれば 600 となり，それがアプリの数あるため膨大となる．

アプリケーション間の連携　各アプリケーションの受けるイベントが別のアプリケーションからくるイベントの場合もある． このようなアプリケーション間の連携も状態遷移表で示した方が良い． しかし，実質的にはアドインのソフトウェアに関してはインストール済みのソフトウェアが知ることは困難であるため，アドインソフトウェアに関しては許可できることを想定しておく必要があり，開発時の課題と言うことができる．

関連研究

携帯電話のソフトウェアモジュールを発売後にユーザ自身の手でダウンロード，インストールする方法については，まず Java 実行環境が実現されている． Java では JavaVM そのものを JAM [1]というモジュールで制御し， Java プログラムからは JAM を制御できないようにするなど，機能を制限することによりセキュリティを確保し，アプリケーション

[1]Java Application Manager

競合に関しては予め決まった形でしか利用できなようにすることと，Java アプリケーションが同時には１つしか実行できないようにすることによりアプリケーション競合の問題を解決している [29].

しかしながら，複数の JavaVM を動作させせることなどを想定すると，予め競合の問題を解いておくことができるわけではなく，ネイティブコードのアドインに関して Java と同様の手法を採用すると，ネイティブコードであるという意味が薄れてしまう.

アプリケーションの競合問題を解く手法としては，リソースの利用方法をモデル化し，これによりリソースアクセス方法を統一して，個々のアプリケーションが利用するリソースの管理を可能とする手法が提案されている [33].

これは一種の状態遷移表モデルであるが，開発の効率化という観点から提案された手法であり，後から追加するアプリケーションにとっては，他のアプリケーションとの関係などをすべてケアせねばならず，リソースを使う複数のアプリケーションが発売後インストールされるとなると，どちらのアプリケーションも意図通りに動作させられるとは限らないという問題がある.

また，アプリケーション単位にリソースに関する記述方法を用意するという提案もされている [30].

この手法も開発効率化の観点から考えられている． 各アプリケーションが他のアプリケーションを意識することなく独立にリソースの使い方について記述できるのであれば有効な方法と言える． しかし，文献 [30] では，どのように記述すれば本課題を解決されるかが示されておらず，未解決である.

このような観点から，アドインソフトウェアとしては Java 言語で記述する方式がセキュアな実行環境が担保できると考える．そのため，本論文では Java でアドインソフトウェアを記述することを前提とし，その高速化手法に焦点を絞ることとした.

2.2.2 携帯電話 Java 高速化技術

携帯端末向けの JavaVM[34][35] は，アーキテクチャ非依存コードであるバイトコードを実行するために，インタプリタを用いる実装 [36][37] が一般的であった．しかし，インタプリタは，バイトコードを逐次解釈して実行するために実行速度が遅いという問題がある．携帯電話の Java アプリケーションでは，実行速度を重視するゲームが多く配信されており，特に，図 6.1のような高速シューティング・ゲームでは実行速度の遅さは重要な問題である.

インタプリタの実行速度の問題は，携帯端末に限らず JavaVM の一般的な課題である．実行速度を向上するために，バイトコードを CPU が直接実行できる命令 (以下，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を搭載する取り組み [38][39][40] が行われてきた．しかし，ネイティブコンパイラを用いた高速化方式では，変換したコードを保持するメモリが必要となるため，メモリ資源の限られた携帯端末に搭載する際にはそのままでは適用できず，使用するメモリ量を削減しなければならない．

ネイティブコンパイラが使用するメモリ量を削減するためには，ネイティブコードに変換するバイトコードを頻繁に実行されるホットスポットに限定し，その他のバイトコードはインタプリタを用いて実行する，バイトコードを選択して変換する方式 [41] が有効である．

ネイティブコンパイラを利用しない高速化方式として，バイトコードを直接実行する Java アクセラレータの搭載があげられる． Java アクセラレータは，ネイティブコードを格納するメモリ領域を必要としない．しかし， Java 専用ハードウェアである Java アクセラレータのコストが製品コストの増大につながる．メモリ資源は， JavaVM の実行速度の向上以外の用途にも利用することが可能であり，製品コストの低減が必要な携帯端末では，メモリ資源への投資を選択し，ネイティブコンパイラを利用すると考えられる．

また，ネイティブコンパイラを用いた高速化方式を，実行速度を重視するゲームに適用する場合には，ネイティブコンパイラの変換処理時間を考慮する必要がある．特に，高速シューティング・ゲームでは，ユーザ操作に対する応答時間の制約が厳しく，ユーザ操作に対する応答を遅延させてはならない．

関連研究

携帯端末向けのネイティブコンパイラを用いた高速化方式の研究として，ダウンロードする前にネイティブコードに変換する方式 [79] がある．しかし，携帯端末の特性から通信量が増加することは容認できず，携帯端末上でネイティブコードに変換することが必要である．

パーソナルコンピュータに代表されるコンピュータ分野のネイティブコンパイラを用いた高速化方式の研究 [38][39][40] では，高い性能を得ることに主眼を置いている．そのため，メモリ資源の制約が厳しい携帯端末に対して，そのまま利用することはできない．

性能向上を犠牲にしてメモリ消費量の低減，変換処理の時間短縮を優先するネイティブコードの変換方式の研究 [42][43][44] が行われている．これらの研究では，頻繁に実行されるホットスポットを判定してネイティブコードに変換する高速化方式については議論されていない．

検出したホットスポットをネイティブコードに変換し，コードキャッシュのメモリ量を低減する方式が，携帯端末向けの JavaVM[41] に採用されている．しかし，ユーザ操作に

対する応答時間への影響を最小限に抑えるための検討が行われていない．高速シューティング・ゲームのようなアプリケーションでは，ネイティブコンパイラの変換処理が動作した場合の瞬停の発生は致命的な問題であり，検討が必要である．

また，プロファイリングに必要なメモリ量，オーバヘッドを低減するサンプリング方式を採用した研究 [45][46] が行われている．しかし，サンプリング方式では，プロファイリングが一定した処理時間とならず，ユーザ操作に対する応答時間が一定とならない．

本論文では，メモリ量が少ない携帯端末上で，高速シューティング・ゲームを代表とするユーザ操作が伴う Java アプリケーションを，ネイティブコードに変換する時間を考慮しながらネイティブコンパイラを用いて高速化する方式を提案し，実装およびその評価を行い，効果があることを示す．

2.3 携帯電話ユーザによるカスタマイズ技術

2.3.1 携帯電話ユーザ操作予測技術

携帯電話の操作とユーザの置かれた状況に関連性があることを調査した結果が文献 [47] に示されている．文献 [47] では，実際に人の行動を人に張り付く形で観察し，コンテキストと操作に関連性があることを示している．

コンテキスト情報を各種センサで取得し，リモコン機器などを一つのボタンで実現する手法 [48] も提案，実現されている．この手法は携帯電話にも応用可能であると考えているが，携帯電話は屋外で利用されるケースが多いため，そのままでは適用できるわけではない．

文献 [49]， [50]， [51]， [52] などは携帯電話にコンテキストアウェアの考え方を検討し，携帯電話上でコンテキストアウェアアプリケーションを開発できるような環境を提案している．

また，文献 [53] では，加速度計や，照度計，マイクなどを体中につけることによって状況を把握し，必要な処理を行う研究もされている．しかし，現実に体にこのような多くのセンサをつけるのは困難であるが，携帯電話にはこれらのセンサの一部が既に搭載されており利用可能な状況にもなっている．ある決まったルールに基づいて状況を判断し処理をするモデルである．

一方，文献 [54] では，文献 [53] の研究をさらに進めて，ルールを学習して動的に変化させる方法を提案している．これらの手法は，ユーザの操作性を向上するという観点で非常に有効な手法であると考える．

また，文献 [55] では，車の中での操作は特別であり，運転中であることを判断する手法

に関して述べている．このように，停止しているのか，動いているのかなどを加速度センサを用いて判断することは非常に有効と考える．

文献 [56] では，加速度センサではなく，基地局の情報から状態を推定する手法に関して述べている．消費電力の面などを考えると有効な手法の一つと考える．

[57] や [58] ではコンテキストに応じて携帯端末を動作させる研究が進んでいる．また，[59] では，さらにコンテキスト情報をその確率の高い時だけに取得するというような手法で，コンテキスト情報取得の問題を解いている．

第 3 章

バイナリ差分抽出方式

本章の概要

本章では携帯電話のソフトウェアの差分抽出には従来の技術を利用した上で，携帯電話のソフトウェアの特性としてアドレスを示す部分の下位ビット位置に差分が発生する特性と，従来の差分表現形式の特性として差分表現のコマンド数をできる限り少なくした方が良いという特性を考慮した差分表現形式を新たに提案する． 提案した方式で， 実際の携帯電話を利用して複数のバージョン間の差分データ量に関する評価結果を示し，提案した方式が両者とも有効であることを示す．次にさらに差分量を減らすために提案した複数の方式を合わせた方式を提案し，携帯電話はもちろんのこと，他の組み込み機器にも有効であることを示す．

3.1 バイナリ差分とは

3.1.1 差分表現形式

差分表現形式は gdiff[13] 形式を基本と考え，本論文では次の 5 種類のコマンドを繰り返すことにより表現する．

(1) DATA コマンド
新版と旧版で全く内容が異なる場合は，新規にデータを表現する必要がある．DATA コマンドそのものが次に続くデータのサイズを示す方式とし，指定されたサイズ長の新規データを続ける．

(2) SKIP コマンド
新版と旧版で内容もアドレスの位置も同じ部分を SKIP としてまとめて表現する．フラッシュメモリなどを書き換える必要のない部分でありサイズを指定する．

(3) COPY コマンド
新版と旧版で位置のずれのみが生じている部分を COPY コマンドとして表現する．旧版上のデータの位置をずらすことにより新版を作成することができ，コピー元のコピー開始位置とサイズを指定する．

(4) FORCED-FLUSH コマンド
フラッシュメモリへの書き込みを指示するコマンドである．RAM 上で差分適用したデータをフラッシュメモリに書き込む．このコマンドの後に，既に書き換えたメモリを旧版として参照する COPY コマンドがないことを保証する．

(5) END コマンド
差分の最後であることを示す．

図 3.1の場合，領域 1 は旧バージョンと新バージョンで何も変わらない部分，領域 2 は旧バージョンの位置から新バージョンでは 0x30 番地だけアドレスの大きな位置にずれているが，内容は変わらない部分であり，そのズレの部分が新バージョンでの領域 5 となる．また，領域 3 はアドレスの小さな方向に 0x20 番地ずれ，領域 4 の部分は新バージョンでは削除されていることを示す．

このような場合，差分コマンドは図 3.2に示すようになる．最初の SKIP コマンドは，領域 1 を示し，次の DATA コマンドは領域 5 を示す．続く 2 つの COPY コマンドはそ

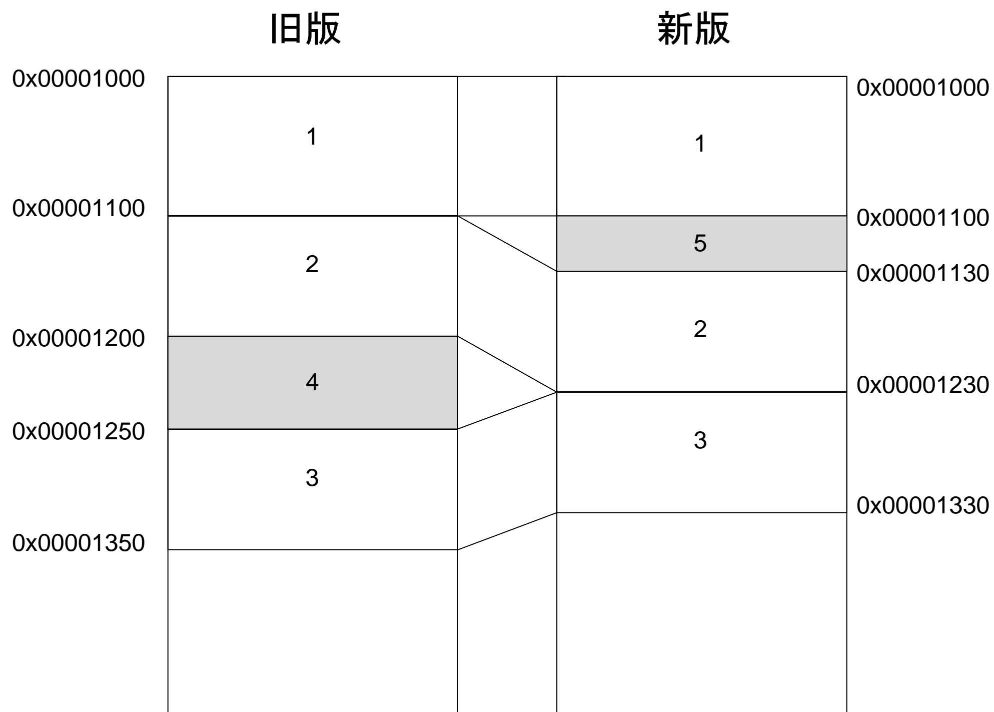

図 3.1: 新旧版の差分パターンの例

れぞれ領域 2 と領域 3 を示す．領域 4 は新版では存在しない部分であるため，差分のコマンドとしては表れない．この考え方に携帯電話の次の 2 点の特徴を考慮した拡張をする．

(1) フラッシュメモリに比べて RAM サイズが小さい

(2) コストを考慮してフラッシュメモリの二重化などが行えない

これらの特徴から携帯電話では差分の適用に際しては，新版のイメージを作成してから旧版を書き換えるのではなく，逐次差分を適用していかなければならないことがわかる．そ

```
SKIP 0x100
DATA 0x30,   Data should be changed
COPY 0x1100, 0x100
COPY 0x1250, 0x100
```

図 3.2: 差分コマンドの例

こで， FORCED-FLUSH コマンドで書換え済みのメモリを参照することがないことを保証する．なお， END コマンドは FORCED-FLUSH コマンドの意味も含む．

3.1.2 差分サイズを決める要因

差分サイズは COPY コマンド， SKIP コマンド， DATA コマンドの数によって決まる．COPY コマンドの数を c とし， DATA コマンドの数を d とし， DATA コマンドに続くデータ列の平均の長さを l とする． SKIP コマンドの数を s とし， 各コマンドは 1 バイトで表現する． DATA コマンドは gdiff のようにデータ長そのものがコマンドを意味することができる． COPY コマンドや SKIP コマンドで必要なアドレス情報は a バイトで表現できるとし，長さは n バイトで表現できるとすると，差分サイズ $dsize$ は以下の式で表すことができる．

$$dsize = (a + n + 1)c + (n + 1)s + d(l + 1) \tag{3.1}$$

ここで，新版と旧版で異なる部分（以下本質的差分と呼ぶ）は合計で $d * l$ であるから，式 (3.1) は COPY コマンド， SKIP コマンドの数を小さくすればするほど差分サイズは小さくなることを示している． また， SKIP コマンドは， COPY コマンドを利用しても同値の差分情報に置き換えられるが， $a + n + 1 > n + 1$ であるため， SKIP コマンドを利用できる部分は SKIP コマンドとした方が良い． また， 実際に COPY コマンドを使うと最低でも $a + n + 1$ バイト必要であるから， コピーする領域の大きさを DATA コマンドで表した方が小さい場合もある． これを考慮すると， $a + n$ バイト以下の COPY コマンドは DATA コマンドと表現することになる． 従って， 本質的差分には $a + n$ バイト以下で COPY コマンドで表現できる部分も含むものと考える． 即ち， $a + n$ バイト以下の COPY 部分と差分部分が交互に現れるようなケースが多いと本質的差分も多いことを示している．

3.1.3 携帯電話ソフトウェアの差分特性

携帯電話のソフトウェアは， パソコンのように実行モジュールをダイナミックにロードしてロード時にアドレス解決を行い実行するモデルではなく，アドレス解決済みのコードをフラッシュメモリ上に配置し， そのまま実行するモデルであることが多い． この場合，一部の不具合の修正で全体の位置をずらすとアドレス関係が崩れる． そのため， ジャンプ先アドレス情報はすべて差分となる． コードにずれが起きている部分では， ジャンプ命令等フラッシュメモリ上のアドレスを参照する箇所に差分があり， シーケンシャルに差分コ

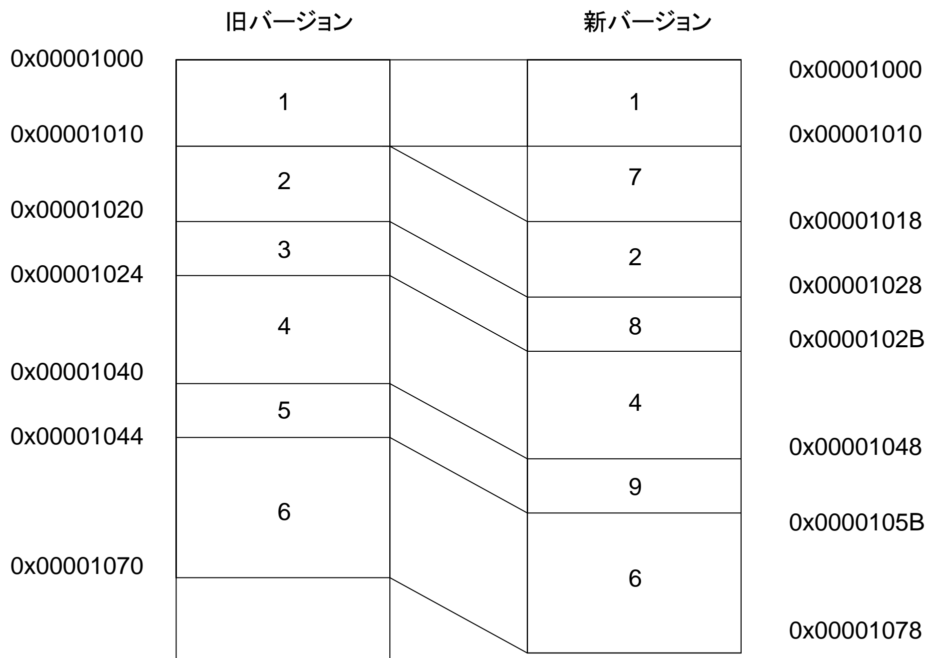

図 3.3: ずれによる差分パターンの例

マンドを生成すると DATA コマンドと COPY コマンドを繰り返すことになる．即ち，携帯電話の不具合修正においては以下の 2 点の特性があると仮定できる．

(1) 不具合の修正は全体の一部である． そのためコードの増加または削減により， 差分としてはコード全体の位置ずれに起因する部分が多い．

(2) 位置ずれの影響により参照関係のある部分のアドレス部に修正が必要となる． そのためアドレス部を中心に差分が発生する傾向が高い．

ずれによる差分

例えば， 図 3.3に示すような新版イメージと旧版イメージの場合を考える．領域 1 は新版， 旧版で全く同じ部分である． 領域 2， 4， 6 は位置ずれのみが起こっている部分である． 領域 7 は追加部分であり， 領域 8， 9 はそれぞれ， 領域 3， 5 から変更された部分である．このような場合の差分コマンドを図 3.4に示す．

この例の場合では， アドレスを示すのに 32 ビット空間で 4 バイト即ち $a = 4$ である．

```
SKIP 0x10
DATA 0x8,
COPY 0x1010, 0x10
DATA 0x4,    Data should be changed
COPY 0x1024, 0x1c
DATA 0x4,    Data should be changed
COPY 0x1044, 0x2c
```

図 3.4: ずれによる差分コマンドの例

SKIP コマンドは 1 箇所であり COPY コマンドや SKIP コマンドのサイズ n を 2 バイトで表現すると仮定すると，SKIP コマンドは 3 バイトになる．DATA コマンドは 3 箇所合計で 19 バイト，COPY コマンドが $(4+2+1)*3=21$ バイトとなる．

$$dsize = (a+n+1)c+(n+1)s+d(l+1) \quad (3.2)$$
$$= 43 \quad (3.3)$$

即ち，合計で 43 バイトの差分で表すことができる．

アドレスの参照関係による差分

図 3.3の例では 4 バイトまとめて変わったデータとしているが，携帯電話のソフトウェアの不具合を修正する場合は，1 箇所の追加修正か削除修正の影響がアドレス部に出てきて，アドレスを示す部分の情報が 1 バイト変わるようなケースが多い．即ち，DATA コマンドと COPY コマンドを繰り返すことが多い．

1 行の追加や削除で，全体の位置がずれると，図 3.5，図 3.6に示すように挿入，削除したところ以外にも影響が及ぶ．絶対アドレスで指定している部分は，そのアドレス情報にずれが生じる．相対アドレスでアクセスするような場合でも相対アドレス指定している間にずれが生じているとやはり影響を受ける．

この影響は，必ずアドレスを表現する部分に現れる．しかもずれるといっても，出荷後のソフトウェアでは，大きなずれは考えにくく，アドレス指定の下位数ビットにず

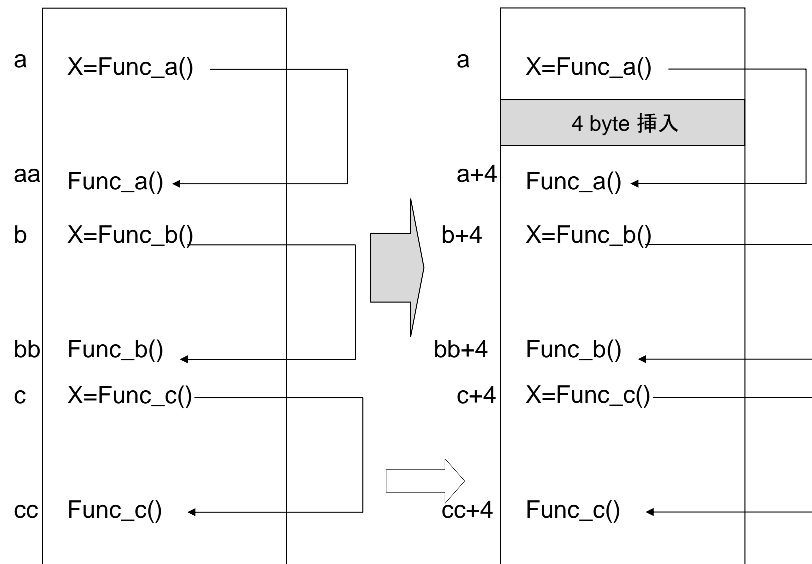

図 3.5: 修正の影響

れを生じる可能性が高い．即ち，差分の発生する部分が分散するために DATA コマンドと COPY コマンドを繰り返しやすくなり差分量を大きくする原因となっている．

3.2 表現形式の改良提案

本節では前節までに検討した課題に対して，差分を小さくする表現形式を以下の 2 点の観点から提案する．

(1) COPY コマンドを減らし SKIP コマンドをなるべく使うようにすることにより差分量を削減する．

(2) COPY コマンドや SKIP コマンドそのものを減らす方式として差分量を削減する．

3.2.1 マクロコピー方式の導入

表現形式提案

ずれによる差分は， 本質的な差分ではなく， 本来ずれがなければ SKIP コマンドで表現できるものである． ところがずれているために COPY コマンドが必要となり差分を大

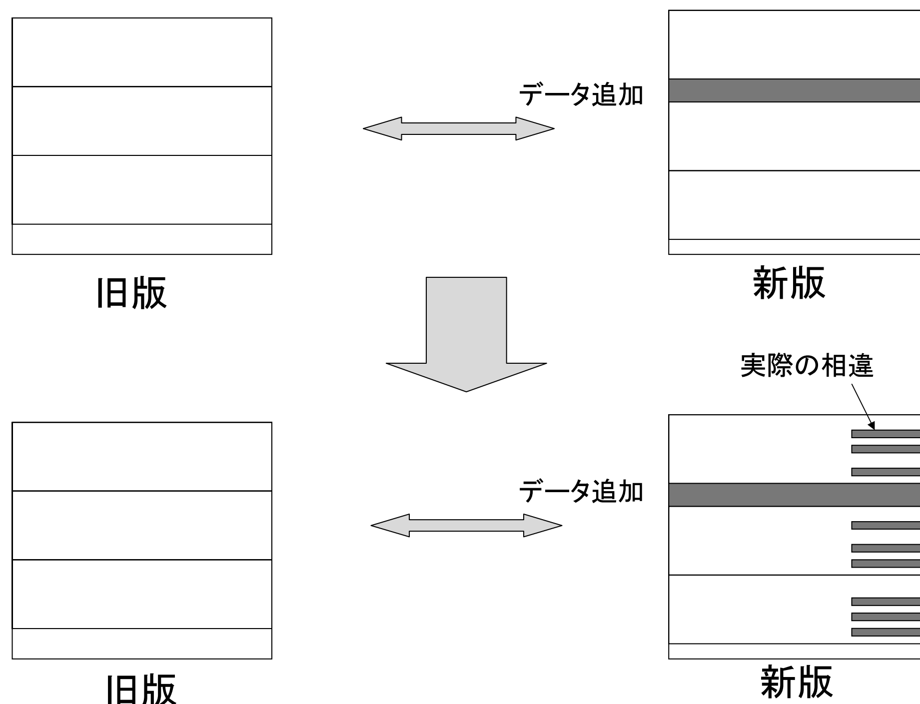

図 3.6: 修正による全体への影響

きくする要因となる．そこで従来方式に大きな単位でまとめてコピーする方法を導入する．即ち図3.3にある領域2，3，4，5，6はまとめて新版としてコピーした後に，DATAコマンドとして，領域8，9を作成するという方法である．一旦一つ前のCOPYコマンドの実行位置まで戻るコマンドとしてBACKコマンドを追加することにより，図3.7のように表現できる．

図3.7では，SKIPコマンドは4箇所あり12バイト，DATAコマンドが3箇所あり19バイト，BACKコマンド1バイト，COPYコマンド1箇所のため7バイトで合計39バイトである．改良前が式(3.3)に示すように43バイトであったため4バイト減らすことができている．SKIPコマンドに置き換わるCOPYコマンドは元のCOPYコマンドと同じで，またマクロコピー法でもCOPYコマンドは一つある．そこで差分サイズ $newdsize$ は式(3.4)になる．

$$newdsize = (a + n + 1) + (n + 1)(s + c) + d(l + 1) \tag{3.4}$$

差分量の改善は次の式で示される．

$$f(c) = dsize - newdsize \tag{3.5}$$

```
SKIP 0x10
DATA 0x8,    Data should be changed
COPY 0x1010, x60
BACK
SKIP 0x10
DATA 0x4,    Data should be changed
SKIP 0x1c
DATA 0x4,    Data should be changed
SKIP 0x2c
```

図 3.7: マクロコピー表現導入によるコマンドの例

$$= \quad a(c-1)-n-1$$

即ち， マクロコピーの対象範囲に COPY コマンドが 2 または 3 以上あって，SKIP コマンドに置き換えられるようなケースでは差分量を小さくする効果があることがわかる．

実現方式提案

マクロコピーにおいて，マクロの単位をどう発見するのかは課題となる． 図 3.8にあるような場合において，最大の範囲でマクロコピーを考えると， 領域 2 〜 7 となるが最大の領域を発見するのは計算量的に大きくなることが予想される． 最大の範囲でマクロコピーを実現してもその中で， ずれが多く発生し， SKIP コマンドが使えないのでは意味がない． そこで確実に COPY コマンドを SKIP コマンドに変える方法として以下に示す手順を提案する．

(1) マクロコピーではなく従来手法で差分を抽出する

(2) 他のコマンドを挟んで連続した COPY コマンドを探す

(3) 連続した COPY コマンドでずれ幅が同じものを探す

(4) 2 または 3 以上の連続した COPY コマンドの範囲をマクロコピーと見なす

このように実装すると， 図 3.8では， マクロコピーの対象となる COPY コマンドがない． しかし， 不具合の修正への適用と考えると全体に位置ずれをしているケースが大半と想定

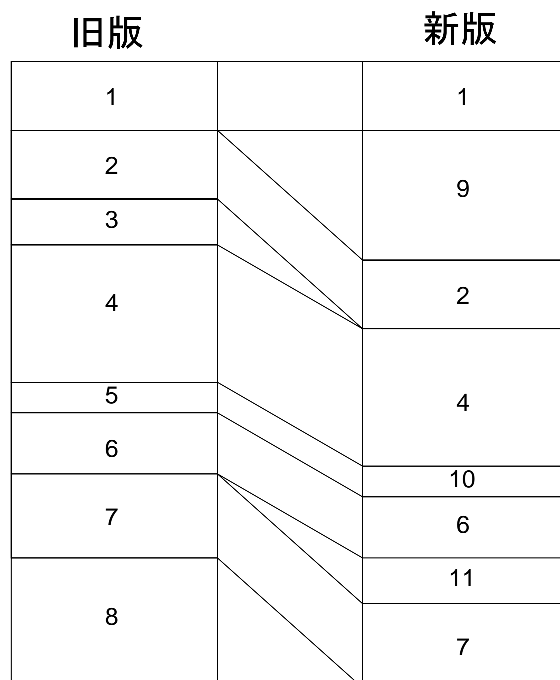

図 3.8: 様々なずれパターン

できるためこの手法で十分と考える． 他にリンク時のシンボル情報から全体のずれを推測するという方法も考えられるが， 不具合の修正という観点からすると同じ結果が得られるものと考える．

また適用時の動作であるが，従来の差分表現方式では，差分コマンドを逐次実行旧版のソフトウェアに対して実行するのみであったが，提案する方式では，逐次処理の部分に，繰り返して処理を行うことになる領域が存在することになる．フラッシュメモリの書き込みという観点からは好ましくないが，実装時には BACK コマンドを先にサーチする手法などにに より 2 度フラッシュメモリに書き込みするという操作は避けることが可能である．そのため RAM 上を 2 度書きする程度であり実行時間上は問題がないと考える．

3.2.2 アドレス変換方式の導入

アドレス空間の変換

差分を表現する上では COPY コマンドの数を減らすことがポイントである． ところが差分の出る傾向としてアドレスを表すデータ部の下位のビットの部分のみ差分が出てくる

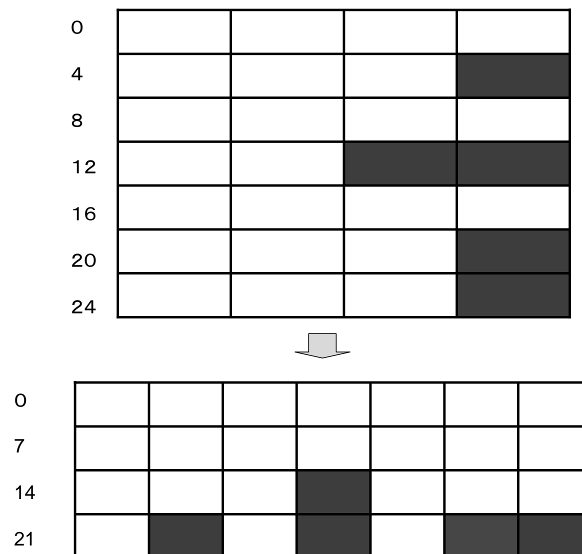

図 3.9: アドレス変換例

という特徴は， COPY コマンドと DATA コマンドが交互に出現する可能性を大きくしている． そこで， この差の出る部分の分布をできる限り集中するような変換ができれば COPY コマンドを減らして差分量を減らすことができる．

そこで， アドレス空間の変換を次式を元に行う．

$$new = \lceil old/m \rceil + \lfloor n/m \rfloor * mod(old/m) \tag{3.6}$$

各変数の意味を以下に示す．

new : アドレス変換後のアドレス

old : オリジナルの物理的アドレス

n: 全データサイズ

m: 基本的には命令語長

例えば，図 3.9に示すように全データが 28 バイトで 4 バイト命令だとすると， 0 番地は 0 番地， 1 番地は 7 番地， 4 番地は 1 番地となる． 元は 7， 14， 15， 23， 27 の各番地に差分があるが， 変換後は， 17， 22， 24， 26， 27 となる． 番地からみると後半にのみ差分が現れており差分は分散せず集中の傾向にあることがわかる． しかし， 変換前の 14 番地のように一部違う場所に差分があればその差分は離れた位置に出現する． そのため実際の分布状況に依存するのであるが， 携帯電話の不具合修正の場合， わずかの位置ずれの確率が高いため 4 バイト目に修正は集中していると予測できる．

表 3.1: 携帯電話ソフトウェアの複数バージョン間での差分量

バージョン名	従来方式 差分量 (KB)	マクロコピー方式 差分量 (KB)	削減率 (%) 削減率 (%)
1 → 2	799	571	28.4
2 → 3	1884	1382	26.7
3 → 4	210	202	3.5
4 → 5	1231	928	24.6

アドレス変換方式の実装

実装上は単純に全体をアドレス変換はできない． 物理的にはフラッシュメモリ上のアドレスであり， フラッシュメモリの消去ブロック単位で書換える必要があるので， 全アドレス空間を一度に変換していては， 書き換えて差分を適用する際にフラッシュメモリ内の書換え対象の部分をすべてを RAM 上に展開する必要が生じる． 一般に携帯電話ではフラッシュメモリより RAM は小さいため実質上一度に変換する方法は差分の適用を考えると実装不可能と考える．そこで， フラッシュメモリの消去ブロック単位に変換を掛けていくことによる実装方法を提案する． 実際には RAM の容量に依存するので， 消去ブロック複数個分をまとめても良い．

3.3 提案方式の評価

提案した方式を実際の携帯電話のソフトウェアイメージに適用し評価した． CPU として M32R[60]， μ iTRON 上で動作する携帯電話で，携帯電話の出荷直前のソフトウェアの複数のバージョンに対し差分を抽出した

3.3.1 マクロコピー方式の効果

携帯電話の出荷直前のソフトウェアの複数のバージョンに対しマクロコピー方式を適用した例を表 3.1に示す． 対象とする携帯電話のソフトウェアのイメージはほぼ 24MB 程度である．

実際に差分量が 1MB 程度の場合には 24 ～ 28 %程度の効果があることがわかる．さらに各差分にマクロコピーが適用できた範囲内の COPY コマンドの数と SKIP コマンド数

表 3.2: マクロコピー適用内の COPY コマンド数

バージョン名	COPY コマンド数	SKIP マンド数
1 → 2	57028	386
2 → 3	125740	457
3 → 4	1820	0
4 → 5	75536	204

```
   0x00000020      0x00000012
   0x00000001      0x00000000
   0x00000033      0x00000020
   0x00000002      0x00000001
   0x00000040      0x00000033
   0x00000003      0x00000002
   0x00000055      0x00000040
   0x00000004      0x00000003
   new version     old version
000000 are changed to SKIP commands
```

図 3.10: ずれている部分で SKIP コマンドがでる例

を調べた結果を表 3.2に示す.

COPY コマンド数が多いものはその効果が大きく 3 → 4 の場合は COPY コマンド数も少ないため効果も少ないと想定される. また, 本来はマクロコピーの対象範囲には少ないと想定していた SKIP コマンドがマクロコピーを適用するべき範囲内にあることがわかった. これは図 3.10に示すように関数テーブルなどでずれていてもたまたま同じアドレスに同じデータが来るようなケースに起こっていることがわかった. しかし, マクロコピーそのものに影響のあるものではないが, こういうテーブルだけでできている場合はマクロコピーの効果が薄くなると想定されるが, 全体からみるとわずかな部分にすぎない.

表 3.3: アドレス変換方式における差分量

対象 バージョン	アドレス変換前 差分サイズ	アドレス変換後 差分サイズ	削減率 (%)
1 → 2	17434	12248	30
2 → 1	17434	11697	33
1 → 3	1917130	1130463	41
3 → 1	1919958	1129068	41

3.3.2 アドレス変換方式の効果

同じ携帯電話ソフトウェアを対象にアドレス変換方式の効果を評価した．わざと不具合を入れてみる方式で評価を行った．

表 3.3に差分サイズ一覧を示す．修正部が小さい場合で，約 30% 差分サイズを削減できている．修正部が大きい場合は，約 40% 差分サイズを削減できている．

修正部が大きい場合は，連続４バイトの差異が減っていたにもかかわらず差分データサイズという観点では大きな効果が出ている． この理由は，連続 4 バイトだけに着目していたためで， COPY コマンドの数が減っている．即ちデータコマンドの数も減っているかどうかをそのままでは示していないためだと考えられる．

差分が大きい場合，小さい場合それぞれのバージョン間で，差異サイズをバイト位置単位で表したものと 4 バイト連続して差異があった量との比較を表 3.4，表 3.5に示す．差分が大きい場合も小さい場合も，変換により差分の発生するバイト位置は一定に分散している．逆に 4 バイト連続して変わっている部分との差が大きくなってる． また 4 バイト連続して差分となるバイト数が多くなっている．即ち，差分が集中していることを意味し，コマンド数の削減に繋がっていると考えられる．

表 3.6に，それぞれの場合のデータコマンド数を示す．差異部分が変わるわけではないため，データコマンド数が減ることは，データコマンドで送られる各データ長の平均長が大きくなっていることを示しており，全体の差分サイズの大きさを小さくしていることがわかる．

出荷前の状態のソフトウェアの版に対して，実際の不具合の修正を行ったもので評価を試みた． 表 3.7，表 3.8に結果を示す． 差分データサイズで 30 %強の削減があり，アドレ

表 3.4: 差分分散状況 (差分小 1 → 2)

アドレス変換前

バイト位置	1	2	3	4
a	490	490	490	2243
b	18	18	18	1771

アドレス変換後

バイト位置	1	2	3	4
a	1442	1435	1406	1433
b	430	423	394	421

a: サイズ（バイト数） b: 4 バイト連続差異

表 3.5: 差分分散状況 (差分大 1 → 2)

アドレス変換前

バイト位置	1	2	3	4
a	119,515	120,056	128,889	282,219
b	2,199	2,740	11,573	164,903

アドレス変換後

バイト位置	1	2	3	4
a	143,832	143,089	143,136	141,364
b	42,088	41,341	41,388	39,616

a: サイズ（バイト数） b: 4 バイト連続差異

ス変換の方式は実際に効果があると言うことができる．

3.3.3 ユーザ利便性評価

本来の目的である不具合修正のユーザへのサービスの観点から更新時間に着目した以下に示す 3 点に関して評価する．

(1) ソフトウェア更新準備のための時間

(2) データダウンロード時間

(3) ソフトウェア書換え時間

本 3 点の時間の合計がソフトウェア更新の時間である． (1) のソフトウェア更新準備で時間がかかるのは電池の充電である． ソフトウェア書換えのアルゴリズムによって消費電力

表 3.6: DATA コマンド数

対象 バージョン	アドレス変換前 差分サイズ	アドレス変換後 差分サイズ
1 → 2	1884	790
2 → 1	1891	791
1 → 3	172380	78383
3 → 1	171585	79124

表 3.7: 実データでの差異位置

バイト位置	1	2	3	4
差異サイズ	7472	7491	8480	28001

が上がってしまうとソフトウェア書換え中に電池不足に陥る危険性が生じる． (2) のデータダウンロード時間はデータサイズで決まる． (3) のソフトウェア書換え時間は書換えを必要とするフラッシュメモリのブロック数によって決まる．本論文で提案した差分表現方式は，差分の表現の工夫であって，不具合の修正でソフトウェアを書き換える量は変わらない． 従って，各評価項目に関して次のように考えることができる．

(1) ソフトウェア更新準備のための時間

新表現方式により消費電力への影響を検討する． 提案した方式では，通信時間は短くなり，ソフトウェアのフラッシュメモリへの書換え時間は不変である． しかし，差分データからソフトウェアの書換えイメージを作成するには，データの変換処理が必要であり， RAM 上のデータコピー処理が余分に必要になる．

一般に消費電力はフラッシュメモリの消去処理が大きく，メモリのリード処理に比較してかなり大きい．通信処理も 2CPU 構成を取ることが多く消費電力は大きくなる．それらに比べ，データの変換処理はワンパスデータを RAM 上で読み込んで RAM 上に書くだけの処理であり，ワンパス通すのみなので，同じフラッシュメモリ上の

表 3.8: 実データでの差分サイズ

	アドレス変換前	アドレス変換後
DATA コマンド数	21532	7315
差分サイズ	202792 バイト	132835 バイト

空間のメモリ消去に比べるとはるかに小さい．通信時間が短くなるためトータルで消費電力は小さくなることはあっても大きくはならない． 実際には細かく充電量を管理できるわけではなく，ソフトウェアの更新量も予測が難しいため，フル充電を要求することになるため，ソフトウェア更新準備のための時間は不変と考える．

(2) データダウンロード時間

データのダウンロード時間はダウンロードする差分データ量に依存するため，小さくなる効果がある．データ量が 30 %削減となれば， 30 %短くなる．前記評価の例で約 1M バイトであるとすると，第 3 世代携帯電話の実効ダウンロード速度を 100Kbps とすると，約 80 秒でダウンロードできる．これが約 56 秒となる．また，ダウンロード時間が短くなると，サーバ負荷が小さくなるため，同時アクセス数の増大などサーバとの通信においてユーザへサービス向上となる．

(3) ソフトウェア書換え時間

ソフトウェア書換え時間は， 20M バイト程度であれば，数分であるが，限定的な領域の書換えの場合は，書き換えるブロック数に依存する．本方式は差分の表現方法の工夫であるため書換え時間は不変のままである． 例えば，書換えとなる対象エリアが 2M バイトなら 1 分ほどである．ソフトウェアの更新において，電話のサービスの中断時間はこの書換え時間だけであるため不変であり，サービス劣化にはならない．また全体の書換え時間はこの例で 17 %程度の向上となることがわかる．

以上，システムとして評価した場合には，データ量の削減でソフトウェアの更新時間は書換え対象の量とデータ通信量およびフラッシュメモリの書き込み速度などに依存して，その度合いは変わるものの向上する． また，不具合のためのソフトウェア更新においてはサーバへのアクセスの集中が想定されるため，サーバや網側の負荷を考慮した場合には非常に有効となる．

3.4 ハイブリッド方式の提案

3.4.1 提案二方式の課題

第 3.2節のアドレス変換方式ではソフトウェア全体に対してアドレス変換をかけた後に旧版ソフトウェアと新版ソフトウェアの比較を行って差分データを生成している．アドレス変換の目的は相違箇所が集中的に現れるようにして差分データサイズを小さくすることであるが，図 3.11に示すように，アドレス変換前の状態で相違箇所が集中している箇所に対してアドレス変換を行うと却って相違箇所が分散してしまい，差分コマンド数が増えて

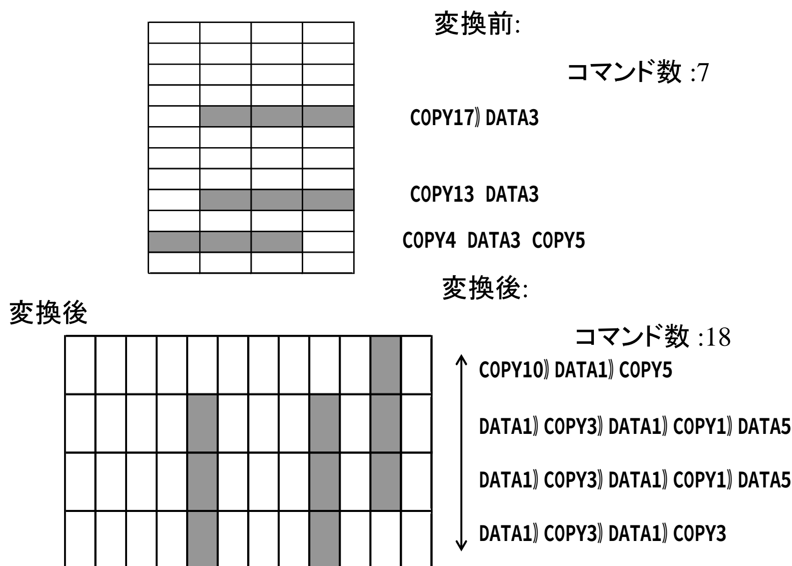

図 3.11: アドレス変換の逆効果例

差分データサイズが大きくなってしまう．実際にある組込みソフトウェアで評価した結果では第 3.2節提案方式では効果のある場合とない場合があり，その理由は上記のような相違箇所の分散が原因である．

また，前節のマクロコピー方式では，マクロコピーの効果による差分サイズ削減は可能であるものの，アドレステーブルが並んでいる場合など，アドレス変換をかけた方が差分サイズ削減に効果のある場合も少なくない．

そこで，ソフトウェア全体に対してアドレス変換をかけたり，アドレス変換をかけずにマクロコピー方式を適用するのではなく，２つの方式を組み合わせて，効果のある部分のみに対してアドレス変換をかける新方式を提案する．

3.4.2 提案アルゴリズム

提案方式ではアドレス変換の開始と終了を意味するコマンドを用い，以下の手順で差分抽出を実行する．

(1) アドレス変換をかけずに旧版ソフトウェアと新版ソフトウェアを比較し，マクロコ

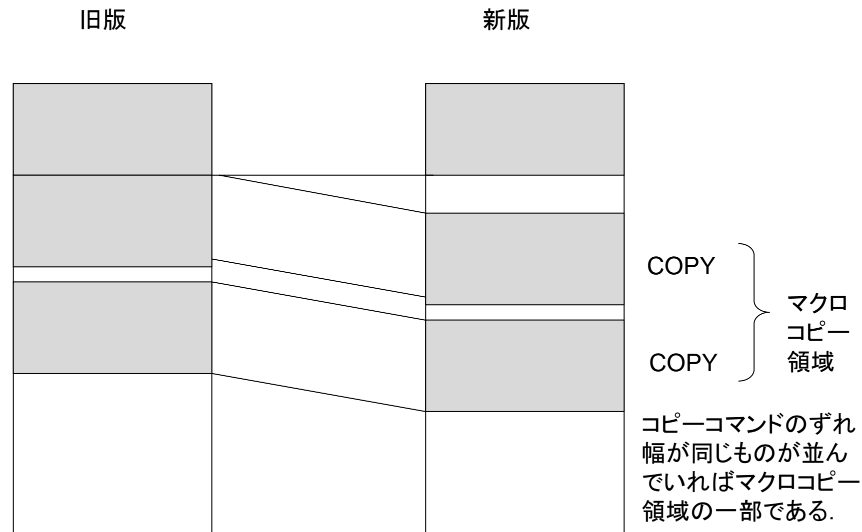

図 3.12: マクロコピー領域の発見方法

ピーで表現できる領域（マクロ一致領域）を発見する．マクロ一致領域の発見は，COPY コマンドのみに着目し，連続した 2 つ以上の COPY コマンドのずれのサイズが同じ場合は，対応する COPY 領域はマクロ一致領域とみなす方法で発見する．

(2) 各マクロ一致領域をワード単位で順次比較し，アドレス変換によってコマンドの数を減らすことのできる範囲を特定する．具体的にはマクロ一致領域ごとにアドレス変換を実施し，変換前と変換後のコマンド数を比較する．この際，アドレス変換を実施する場合は変換開始と終了の 2 コマンドが追加になることも考慮する．次にアドレス変換候補となった連続した領域の結合を行う．これは結合した方が効率が良くなる場合があるからである．この試行を行った上でアドレス変換の領域を決定する．

(3) 差分データのエンコードを行う．この際，アドレス変換範囲に対してはアドレス変換を行ってから (図 3.14) 差分コマンドを表現するとともに，その前後に変換開始，終了のコマンドを出力する．手順 (2) においては，アドレス変換を行うことによる変

- **コマンド数の計算**

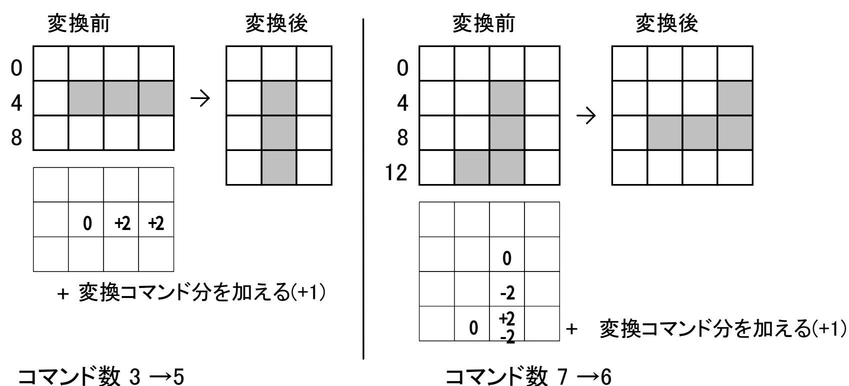

図 3.13: アドレス変換すべきかの判定方法

更開始コマンド 1 バイト，終了コマンド 1 バイトによるコマンド数の増加，即ち 2 バイト分の差分データの増加分を考慮した上で変換すべきかどうか判定しているものとする．

3.5 ハイブリッド方式評価

提案した方式を次の 4 種類のプログラムに対して適用し評価した．

(1) CPU として M32R[60] で動作する携帯電話ソフトウェアの複数のバージョンに対し差分を抽出

(2) CPU として ARM[61] で動作する携帯電話ソフトウェアの複数のバージョンに対し差分を抽出

(3) CPU として SH[62] で動作するエレベータ制御ソフトウェアの複数のバージョンに対し差分を抽出

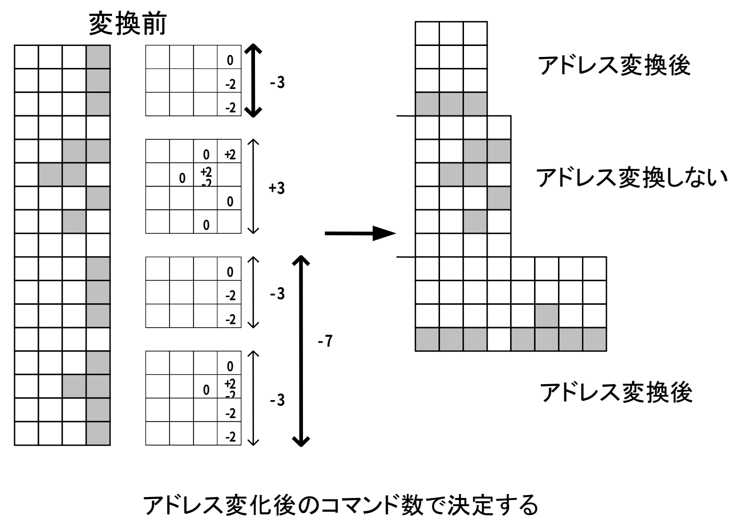

図 3.14: 差分コマンド生成

(4) Intel 系 CPU を利用する Windows 端末アプリケーションソフトウェアの複数のバージョンに対し差分を抽出

3.5.1 M32R 携帯電話ソフトウェアでの詳細評価

(1) に対する詳細評価結果を示す．ソフトウェアの対象 CPU は M32R であり，ソフトウェアのサイズは約 30Mbyte である．ソフトウェアは 3 バージョン（1， 2， 3）用意し，それぞれのバージョン間に対する差分抽出を行った．評価は " 全差分コマンド数 " と " 差分データサイズ（修正データサイズとコマンドのサイズの総合計サイズ) " の 2 つの観点から行う．評価対象とする方式は以下の 3 種類とする．

A: 3.2節提案のマクロコピー差分表現形式

B: ソフトウェア全体にアドレス変換方式で変換後マクロコピー差分表現形式とする方式

C: 提案方式

評価結果として表 3.9にコマンド数を表 3.10に差分データサイズ示す．また差分データサイズに関しては図 3.15にグラフ化してデータ量の削減の状況を示した．ソフトウェア全体にアドレス変換をかけた場合である (B) は，アドレス変換なしの場合である (A) よりコ

表 3.9: 差分コマンド数

	A	B	C
$1 \rightarrow 2$	116,114	102,226	83,841
$2 \rightarrow 3$	265,715	270,970	210,609
$1 \rightarrow 3$	171,248	195,628	147,419

表 3.10: 差分データサイズ (bytes)

	A	B	C
$1 \rightarrow 2$	403,922	367,338	334,820
$2 \rightarrow 3$	936,459	927,899	818,750
$1 \rightarrow 3$	610,033	646,493	559,162

マンド数，差分データサイズともに劣るケースがあるのに対し，提案した改良方式 (C) では全てのケースで他方式より優れている．特に差分データに関しては 8〜17% のサイズ削減効果があることが分かる．さらに (C) で各場合にアドレス変換の対象となった領域の割合を調べた． 1→2 の場合で 1.68%， 2→3 の場合で 2.92%， 1→3 の場合で 1.67% である．データサイズの削減とアドレス変換の対象領域のサイズには関連性は見られなかった．これは対象領域のサイズとコマンドの削減率が重なるためと考える．

また，サイズの削減は転送時間だけでなく， 1 台で複数台の端末を処理する場合のサーバのサービス向上に繋がる．従って，わずかな削減であっても携帯電話のソフトウエア更新のような場合には効果があると考えられる．また，エレベータの制御ソフトのように緊急性のあるソフトウエアの更新でしかもブロードバンド環境が想定できなようなケースにはわずかなデータ削減でもデータ転送時間において効果があると考えられる．

3.5.2 他 CPU アーキテクチャへの適用評価

次に CPU として表 3.11に ARM を利用した携帯電話ソフトウェア (ソフトウェアサイズ約 47Mbyte) での評価を示す．表 3.12に SH を利用したエレベータ制御プログラム (ソフトウェアサイズ約 1Mbyte) の評価を示す．表 3.13に Windows 端末の組み込みアプリケーショ

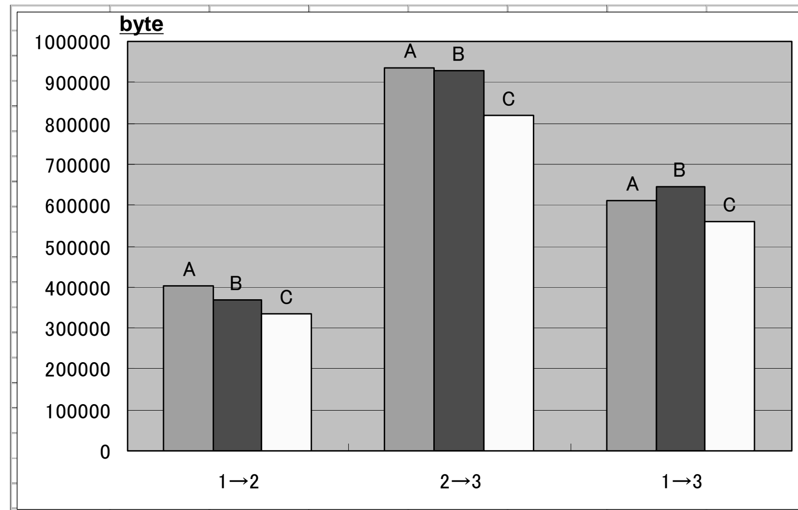

図 3.15: 差分データサイズグラフ

ン（ソフトウェアサイズ 1Mbyte) での評価を示す．

携帯電話やエレベータ制御ソフトウェアは静的にリンクされた構造の部分が多いのに対して， Windows 端末のアプリケーションは動的リンクを行う場合が多い．そのため，元々の差分そのものが比較的小さくなっている． どの場合においても提案の手法は差分が小さくなっていることがわかる． しかし，効果の割合はまちまちである．とくに Windows 端末のように， Intel 系 CPU を利用し，命令が固定長でない場合はアドレス変換そのものの効果がないため，効果が薄いことがわかる．

表 3.11: ARM 差分データサイズ (bytes)

	A	B	C
1 → 2	247,490	236,418	225,577
2 → 3	170,821	173,217	157,103
1 → 3	305,284	302,154	276,753

表 3.12: SH 差分データサイズ (bytes)

	A	B	C
$1 \rightarrow 2$	186,838	178,667	175,842
$2 \rightarrow 3$	186,500	179,474	174,625
$1 \rightarrow 3$	287,937	284,418	278,703

表 3.13: Windows 端末アプリケーション差分データサイズ (bytes)

	A	B	C
$1 \rightarrow 2$	26,393	27,198	25,545
$2 \rightarrow 3$	64,539	67,678	62,159
$1 \rightarrow 3$	22,366	22,520	21,149

結論として，提案した方式は，固定長命令をもつ CPU アーキテクチャで，静的なリンクを持つ場合にはアドレス部分の差分が多いため，アドレス変換の方式の効果が大きく，データ領域などアドレス部以外でアドレス変更の効果のない部分も含まれるケースにおいて効果があることが確認できた．このようなケースは NOR 型のフラッシュ ROM にアドレス解決済みのコードにおいて実行することが多い組込み機器で，かつソフトウエアの更新を行う際に，差分データのサイズと回線の速度の関係でどうしても差分データを小さくしたいような携帯電話やカーナビゲーションシステムあるいは緊急でもソフトウエアの更新を要するエレベータ制御プログラムなどに有効本方式は有効である．

3.6 本章のまとめ

本章では携帯電話のソフトウェア構造，CPU のアーキテクチャに着目して，ソフトウェアのバージョン間の差分を効果的に小さく表現する手法に関して提案した． 具体的にはソフトウェアの不具合修正では若干の位置ずれが大きく差分として現れることから以下の 2 点の手法を提案した．

(1) マクロコピー方式により大きなかたまりの移動を表現することにより差分量を小さくする手法．

(2) アドレス変換方式により分散された差分を集中するような変換が可能である手法．

また，実際の携帯電話のソフトウェアに適用することにより両手法とも従来方式に比べて効果があることを示した．

またさらに，マクロコピー方式とアドレス変換方式を組み合わせるとともに，アドレス変換方式の効果の高い部分にのみアドレス変換をかける方式を提案した．本方式の実現方式を示した上で，実際の組み込みソフトウェアの例として，携帯電話，エレベータ制御ソフトウェア，Windows 端末のアプリケーションプログラムを例に評価し，その有効性を示した．

今後，CPU として固定長アーキテクチャでない場合などに命令を解析した上でアドレス変換をかけるなど提案手法では効果が薄いと考えられる部分への差分量削減への可能性を追求していく予定である．

第 4 章

ソフトウェア保守のためのソフトウェア構成法

本章の概要

携帯電話のソフトウェアの更新時間は, 無線網を転送するデータ量に依存する転送時間とフラッシュメモリを更新する書き換え時間とからなる.

本章ではソフトウェア更新を短時間で実施するための方式を提案する. フラッシュメモリの該当部分だけを書き換えれば良い仕組みにする一方で開発環境の複雑化を防止する必要がある. ソフトウェアの更新を前提とした携帯電話ソフトウェアの構造に関してメモリ上に空き領域を設けることとベクターテーブルを利用することを提案する. 携帯電話のバージョン間の差分データを小さくするとともに, バージョン間でのフラッシュメモリの書き換えを小さくするにあたり, 最適な分割方法を理論上で検証した上で, 実際の携帯電話のソフトウェアを利用して評価した.

4.1 モジュール分割

4.1.1 モジュール分割概要

本章においては, フラッシュメモリのイメージ全体を複数の物理的なコードの集合とし, 物理的なコードの集まりをモジュールと呼ぶ. モジュール分割とは, フラッシュメモリ全体を複数のモジュールに分割することを示し, 次の条件に従った分割とする.

(1) モジュールは固定のアドレスから開始すること

(2) モジュールの先頭にモジュール内の関数を呼ぶためのベクターテーブルを配置する. また, ベクタテーブルに続くコード領域の開始アドレスは固定し, ベクターテーブルの大きさも固定とすること. 即ち, ベクターテーブルに対しても空き領域を設けておくこと

(3) モジュールの最後に空き領域を設けること

(4) モジュール間の関数参照はベクターテーブルを利用した参照とすること

(5) モジュール間のデータ参照は認めず, ＡＰＩ経由または共通データモジュールへのアクセスとすること

具体的にはモジュールは論理的な意味のあるプログラムの集合と考え, 携帯電話のソフトウェアを構成する OS 部,Java 関連部, アプリケーション部, カナ漢字変換部, ドライバ部, フォント部といった単位で分けることを想定する. また, 上記条件の元で次に示すアルゴリズムにより不具合の修正をする.

(1) モジュールの位置固定, ベクタテーブルも入れた状態でリンクする.

(2) リンクしたときのシンボル情報をモジュールごとに管理する.

(3) モジュール単位でアドレス解決を図る. モジュールの外部を参照する部分は該当モジュールのベクタテーブルを示すように修正したアドレスデータを利用してアドレス解決をしておく.

(4) モジュールごとにアドレス解決済みのコードを結合する.

また, 外部からの参照が増加した場合にはベクタテーブルが大きくなる. この場合の新たなシンボルはベクタテーブルの最後への追加とする. また逆にシンボル参照がなくなった場合はベクタテーブルは詰めることをせず空き領域としておく.

表 4.1: メモリ容量への影響

ソフトウェアサイズ	約 12MB
モジュール分割数	12
モジュール間参照 API 定義数	約 3,000
モジュール間参照呼出し数	約 40,000
モジュール間データ参照数	約 300
モジュール間データ参照におけるデータサイズ	約 600KB

結果としてモジュール内の修正は局所にしか影響を与えないことができる.(図 2.1, 図 2.2)

ＣＰＵとして M32R[60] の場合で，10 分割程度において, 以下の 2 点に関して評価を行った結果有効であることがわかった．その結果としてのメモリの増加量と実行時間への影響を表 4.1, 表 4.2に示す.

(1) ベクタテーブルのサイズはモジュール間参照の API 定義数で決まり, 表 4.1からモジュール分割により作成するベクターテーブルは全体のメモリ容量に比べ小さく, メモリを圧迫するほどには増えないことがわかる.

(2) 表 4.2は Java のベンチマークプログラムへの影響を測定した結果である．モジュール間を相互に渡るであろう描画やメモリ書き込みにおいても影響はわずかであり, モジュール間の参照で発生するベクターテーブル経由のアクセスによるオーバーヘッドはわずかであることがわかる.

M32R の特徴はジャンプ系命令においてプログラムカウンタ相対ジャンプが可能であり, 1 命令で任意の位置にジャンプできるようなアーキテクチャであり, 相対位置のずれに対する影響が大きい．本論文でもこのような CPU アーキテクチャで評価を実施するが, テーブルからジャンプ先アドレスをレジスタにロードするようなタイプの CPU でも大きな違いはないと考える.

また, 局所に修正箇所を限定した場合, フラッシュメモリの書換えブロック数は減らせる. 即ち書換え時間が短縮できる．そこで, 本論文ではこのような局所化と処理のオーバヘッドを考慮した最適な分割数を求める観点から検討を進めた.

表 4.2: Java ベンチマークプログラムの性能劣化率

項目	劣化率%
Calc	0.3
Loop	0.3
Method	0.5
Scratch	1.5
Panel	2.8
Graphic	0.7
Image	0.6
合計	1.2

4.1.2 モジュール分割と修正量の関係

一般にモジュール分割数を増やせば, モジュール間参照は増え, モジュール内参照は減る. モジュール間参照が増えるとベクターテーブルのメンテナンスが多く必要となる. そのため, そのバランスの取れた分割数を知る必要がある.

モジュール分割数を x とし, 全モジュール中の外部宣言シンボルの参照数を r とする.

外部宣言シンボルの数を g とする. 一つのシンボルへの参照が複数ある場合があるので, $r \geq g$ である. シンボルが全体に均一に存在し, 参照関係も均一であり, 各モジュールの大きさも同一と仮定すると, 一つのモジュール内に入る外部宣言シンボルの数は以下で表すことができる.

$$global(x) = gx^{-1} \tag{4.1}$$

一つのシンボルが参照される平均の数は rg^{-1} である. 一つのシンボルに対して, ある参照がモジュール内の参照である確率は x^{-1} である. よって, すべての参照がモジュール内に閉じる確率はそれぞれの参照が独立であるため $x^{-rg^{-1}}$ である.

一つのシンボルが少なくとも一つでもモジュール外から参照される確率は $1 - x^{-rg^{-1}}$ である. 従って, 一つのモジュール内でモジュール外から参照されるシンボル数は式 (4.1) より, 以下のように表すことができる.

$$vectorInM(x) = (1 - x^{-rg^{-1}})global(x) \tag{4.2}$$

全モジュールで参照されるシンボル数, 即ちベクタテーブルの大きさは以下で示すことができる.

$$vector(x) = x * vectorInM(x) = g - gx^{-rg^{-1}} \tag{4.3}$$

次にコードが挿入や削除という修正によって移動した場合に影響を受ける要素としてはモジュール内での参照関係がある．モジュールを分割しなければベクターテーブルは不要であるが, モジュール内の参照関係のずれが大きくなるためにコード上の修正を要する部分は大きくなる.

モジュール内での参照数は, 参照側と被参照側がともに同一のモジュールに入るかどうかが問題であり, その数は以下のように示すことができる.

$$local(x) = rx^{-2} \tag{4.4}$$

従って x 分割した場合で, 一つのモジュールでコードの位置が変わった場合にメンテナンス対象となるシンボルは以下に示すようになる.

$$f(x) = x^{-1}vector(x) + x^{-1}local(x) \tag{4.5}$$

即ち

$$f(x) = gx^{-1} - gx^{-rg^{-1}-1} + rx^{-3} \tag{4.6}$$

これを微分すると以下の式になる.

$$f'(x) = -gx^{-2} + (r+g)x^{-rg^{-1}-2} - 3rx^{-4} \tag{4.7}$$

式（4.7）は常に負となり単調減少である.

図 4.1に示すように, モジュール分割数を増やせば増やすほどコードの修正の影響が少なくなり, また漸近的に 0 に近づく．一方で分割数を増やせばベクターテーブルの大きさは大きくなりメモリの圧迫に繋がる.

ベクターテーブルの大きさは式 (4.3) で示される．これは x が大きければ最大で g に漸近的に近づく．しかしながら,1 箇所の修正あたりで増えるかもしれないメモリ量を a とし, モジュールあたりに b 個の修正が必要になると想定すると, 全体のモジュール分割のために必要余分なメモリ量は以下になる.

$$memory(x) = abx + vector(x) \tag{4.8}$$

分割数を増やせば必要なメモリがリニアに大きくなり現実的でない．そこで, 図 4.1に示すように, 分割数を増やしても改修量の効果が減るポイントが重要となる.

また, 分割数を増やすことによりベクターテーブルを経由した参照が多くなれば速度性能に及ぼす影響も深刻なものとなる．そこで携帯電話のソフトウェアとしては以下に示す 3 点の考慮した上でモジュール分割する必要があるため, 実際の携帯電話ソフトウェアの状況に応じて最適なモジュール分割数を決めていく方法を検討した.

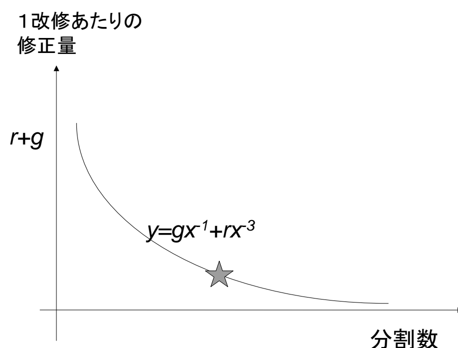

図 4.1: 1 改修あたりの修正量とモジュール数の関係

- ひとつの修正量あたりどの程度までの修正量を許容するか
- メモリの余裕はどの程度取れるか
- 決して均一な参照とは限らない

4.2 評価

前節の修正量を実際に差分表現することを考える．差分表現は前章で述べた差分方式を基本として利用した.

4.2.1 モジュール分割の分割数評価

現実の Java, カメラなどを搭載した機種のソフトウェアで検証を実施した．物理的にモジュール分割を試み, シンボルの参照関係を調べた．表 4.3にその結果を示す.

ここで, 総参照数として $r = 363247$, シンボル数は $g = 29681$ である．即ち $rg^{-1} \simeq 12$ で, 式 (4.6) よりメンテナンス対象となるシンボル数は以下に示すように近似できる.

$$f(x) = 30000(x^{-1} + 12x^{-3}) \tag{4.9}$$

即ち $f(x)$ は x^{-1} で決まることが示された.

前節で検討したように, 図 4.1で示すようなグラフで 1 改修あたりの修正量を表すことができる．そのため☆印のポイントが効果的な分割数と想定でき, 実際に分割数を変えながら分割を実施してみた.

表 4.3: モジュール分割数と関数の参照数の関係

分割数	修正を要するシンボル数	ベクターテーブル数	1モジュールあたりの修正数
1	362954	0	362954
2	349717	2336	174859
4	337722	7356	84430
9	336243	7912	37360
10	320075	9436	32007
11	304663	12996	27696
12	300968	14236	25080
13	295006	15640	22692
15	289949	15936	19329
18	286416	16600	15912

分割はある程度論理的な意味も考慮して分割したもので, 必ずしも物理的にモジュール間の参照関係が最小になるようにしたものではない.

表 4.3によれば式 (4.6) に示すとおり, 修正を要するシンボル数は単調減少である. しかし分割数 11 前後からそれ以上分割しても修正を要するシンボル数の減り方も少なく, ベクターテーブルの増加量も少ないことがわかる.

また, 1修正あたりの修正量のサイズを次のように決める. 例えば, 1修正で差分データを作るとしてその最大値を現在のキャリアのファイル送信の最大値にあわせて, 1修正あたりの修正量の目標を $100K$ バイト以下とする. 1 箇所の修正で 5 バイト（修正の情報とアドレスを示す情報）の情報が必要である場合に,1 モジュール当りの修正量は $100,000 \div 5 = 20,000$ 個所が目標となる. 実際には修正するデータにアドレス情報がすべて必要とは考えられないため,2 つに一つ程度アドレス情報が必要と考えると 3 バイト必要となり, 目標は $30,000$ 個所程度であり, 表 4.3によれば 13 分割以上が必要となる.

4.2.2 空き領域評価

次に空き領域に関してはコード全体で 16M バイトであったので, 1 %を空き領域として認めると仮定すると 160K バイトが空き領域となる. 例えば分割数を 12 とするとベクターテーブルの領域が 1 ベクターあたり 4 バイトとして 56K バイトであるため,1 モジュールあ

たりの空き領域は以下で示される.

$$space(x) = (160 - 56)/x \tag{4.10}$$

$x = 12$ の場合で $space(12) \simeq 9$ で表され,9K バイトまで可能となる. この値は過去の出荷直前の修正量などの経験値からすると十分と考える.

4.2.3 評価のまとめと課題

結論として 13 分割程度は適切な数値である. しかし, 表 4.5に示すようにこれではまだ目標とする差分量は達成できないことがわかった. しかし, これ以上の分割はベクターテーブル増加によるメモリの問題とベクタテーブル経由のアクセス増加による性能の問題から好ましくない. そこで, さらなる分割として, 緩やかなアドレス固定方式を検討した.

4.3 改良方式の提案

4.3.1 緩やかなアドレス固定方式

複数の修正が, 複数のモジュールであった場合には, 大きな差分量となる場合も想定されるため, 更に空き領域を増やすことなく, 修正の影響を少なくできないかを検討した. モジュールの分割はより細かくすることが可能であるが, 空き領域が小さくなるという問題があるため, 次のように考える. モジュールの位置は固定とし, その中を更に数分割を行いブロックと呼ぶこととする. ブロックはモジュールの中で基本的には位置固定とし, 隙間領域を分散する形とする.

図 4.2は 3 つにモジュール分割を実施した場合の例で, モジュールの中に緩やかなアドレス固定領域としてそれぞれ 3 つのブロックを定義した. 緩やかなアドレス固定領域とは基本的にアドレスを固定とするが, そのための空き領域は不足した場合に前後にずらすことを認める領域である. 各モジュールの空き領域はブロックごとに割り当てておく.

例えばモジュール A の例は 2 つ目のブロックに修正が入ったものである. ブロックの余裕領域でコードの増加分を吸収できるケースであり, コードの増大はブロック内に発生し, この影響分だけが他のブロックに影響として出る. しかし, このブロック以外の部分は固定されたままなので影響しないため, 単純に考えれば差分量は 1/3 になる. 実際には同一モジュール内の影響と他のモジュールへの影響が両方入るため, その分は大きくなる.

モジュール B の例は, モジュールの中の先頭のブロックの修正による増大分が空き領域ではカバーしきれなかったケースである. この場合は, 同一モジュール内の隣のブロック

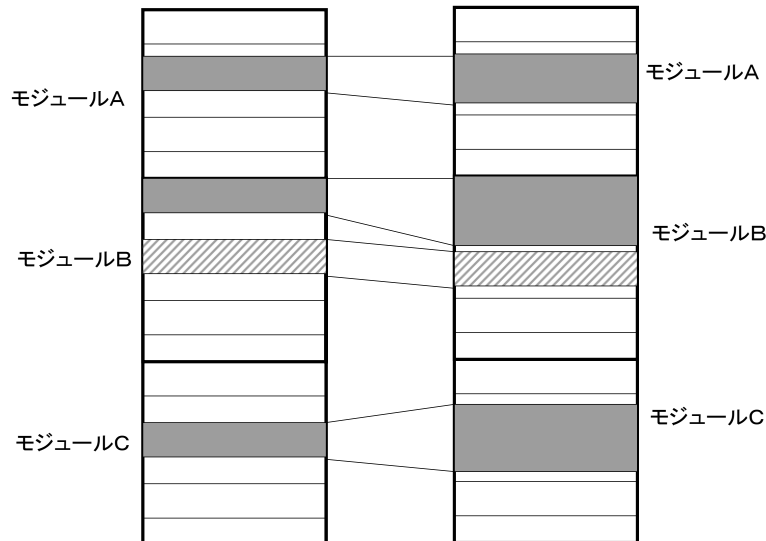

図 4.2: ブロックを導入したモジュール分割

の固定位置をずらし, 空き領域を増やすことにより対処する. これでもモジュール全体が移動するよりも差分データは少なく, 元々 2 ブロックの構成であったのと同じことになる.

さらに, モジュール C の場合は, 二つ目のブロックの修正による増加量が空き領域を超えた場合の例である. この場合は一つめのブロックの空き領域を使うことにより二つ目のブロックの位置がずれただけの形にでき, モジュール A の修正と同様の差分量に抑えることができる.

このようなブロックを設けることにより, モジュール間の差分を抑えることができるとともに, 空き領域はモジュール全体として持つことができるため, 空き領域そのものを多くとる必要がないという効果がある.

実際にこのような配置を行う効果に関しては, モジュールの先頭に 1 行の追加をするという最も影響の大きい修正を行ったときで以下に示す結果が出た. 表 4.5に示すようにブロック化しない場合で最悪値約 400KB, ブロックを導入した場合で最悪値約 80KB あった.

4.4 改良方式評価

4.4.1 差分データ量の評価

次に実際に論理的な分けやすさから 11 分割としてみた場合の参照関係を表 4.4にまとめた.

表 4.4: モジュール分割と参照数

モジュール名	モジュール内参照	モジュール間被参照
アプリ	17099	29
かな漢字変換	3972	17
ブラウザ	19179	42
文字列	3856	59
ドライバ	56798	1578
フォント	6	0
Java	28305	46
データ	171	335
ファームウェア	80467	435
通信制御	84451	49
その他	6574	610

あるモジュールを修正するとその中のモジュール内参照数とモジュール間の参照でいくつ参照されているかを示すモジュール間被参照数の部分が修正を要する可能性がある．この場合, 明らかにシンボルの参照関係には偏りがある．

また, 明らかにもっと密接に関数の呼び出し関係があるにも係らずモジュール間の参照関係が少ないものもある．この理由は関数を引数としてインダイレクトにアクセスするような形をとってる場合に発生する [?]．このような場合は, ジャンプベクターテーブル相当のものを既にコーディングのレベルで作ってしまっているケースである．しかし, 全体の中で見ればほんの一部であり, 今回は無視することとした．

その結果表 4.4からは, 例えばアプリモジュールの修正を入れると 17099 箇所とベクタテーブルの 29 を修正する可能性があることがわかる．

1 箇所あたり 1 バイトの修正があるとしても, 約 17KB の修正量が発生する．これにアドレスの情報が付与されるなどするため, アドレス情報 4 バイトとするだけでも 5 倍になり, 約 85K バイトの差分量になる．実際には修正を入れる位置により影響が決まるため, 平均的には半分の 40K バイト程度と考えられるが, 最悪ケースにおいて, 差分の表現形式によって更に情報が付加されることになるため, わずかな修正が 100K バイトを超えるデータで表現されることが容易に想像される．

そこで, 実際に各モジュール内の先頭にわざと余分なコードを入れて配置位置をずらす方法で差分をとってみた (表 4.5)．これはブロック即ち緩やかな固定領域の概念を入れな

い状態での調査結果である. ドライバと Java とデータで予想に反してより大きな差分が出てきた. これらは前節での検討結果により実際には 100KB 未満に押さえることができる. これは, 明らかに均質な参照関係ではないために起きたものと想像される.

また, Java に関しては, java 内の関数とネィティブコードで実装される関数部の参照関係を関数テーブルで構成しているため差分という観点からは大きくなる傾向にあると考えられる点も一因となる. そこで, 更に調査を行い, 次の 2 点に関して評価を実施した.

4.4.2 ベクタテーブルの効果

ベクタテーブルを導入することは, 1 箇所の修正は修正したモジュールへの影響のみに閉じることを保証することで, ソフトウェアの更新する上ではフラッシュメモリの書換え時間を考慮した場合には非常に有効であることを [?] で示している.

そこでさらにベクターテーブルによる差分データ量への効果を検証するため, ベクターテーブルを使わない運用をした場合の差分量を調査した. その結果,15 %程度差分量が増加することがわかった. 無線を利用したダウンロードを考慮した場合はベクタテーブルは, かなり有効であると考える. また [3][4] のように通常のユーザがユーザ自身によってソフトウェア更新機能を利用することも想定するとなると更に書換え時間の重要性も増すためにより有効であると考える.

4.4.3 ブロックの効果

ブロック即ち緩やかな固定領域の実際の効果を調べるために, 表 4.4に示したモジュールごとに, 先頭にわざと配置位置をずらすためのコードを挿入してみた. 先頭に入れたのは全体のコードがずれるため最も影響が大きくなるケースだからである. その結果, アプリケーション, ドライバー,Java などのデータ部分においてはとくに大きな差分量が発生した. その結果を表 4.5に示す.

予想以上の大きさの差分量に関しては関数テーブルなどが原因と考える. ここではブロックを導入した場合の効果を見る. その結果, コードがずれることに対しては最も差分の大きなモジュールに関しても 100 Kバイト以内の差分量に抑えることができた. ブロックの有効性が確認された.

また実際にはフラッシュメモリ上のイメージではあるが, RAM などのメモリにコピーされて実行されるべき部分も含んでいる. これらは [?] に示すようにそのメモリの種類ごとにベクタテーブルなどを持つため実際上はさらに細かく緩やかな固定領域を設定することも考えられる. この方法でも同じ効果が得られるものと考える.

表 4.5: モジュール分割と差分サイズ

モジュール名	差分サイズ (K バイト)
アプリ	385
かな漢字変換	2
ブラウザ	31
文字列	392
ドライバ	263
フォント	4
Java	198
データ	21
ファームウェア	81
通信制御	28
その他	59

4.5　本章のまとめ

本論文では携帯電話におけるモジュール分割の手法およびベクタテーブルの有効性を示し, 分割の数に関する考え方を整理した.

その結果, 実験に使用した携帯電話のソフトウェアにおいては 13 分割程度が適切な分割であることを示した. また, 緩やかな固定領域という概念を導入し, モジュールの単位をさらに小さいブロックとすることにより修正量の表現を小さくすることには有効であること確認した.

また, 今後のモジュール分割の課題としては次の点があると考えている.

(1) CPU のアークテクチャが異なる場合での有効性の確認

(2) モジュールおよびブロック内ソフトウェアの信頼度などの概念導入による空き領域の最適化

(3) 被参照数 1 の場合のベクタテーブルの有効性の確認

(4) 差分表現方法の工夫によるさらなる差分の最小化

(5) DLL による実装への適用検討

今後これらの課題に取り組んでいく予定である.

第 5 章

携帯電話ソフトウェア更新における高速プログラムコード圧縮方式

本章の概要

最近では，携帯端末は，コストの面から NAND 型フラッシュメモリに圧縮状態でプログラムコードを置き，デマンドページング方式で実行する方式が採用されつつある．デマンドページングに適した圧縮方式は，回答時間が高速であることと，ランダムアクセスが可能であるという条件がつく．これらの条件に適した圧縮方式である Byte-Pair 圧縮方式は圧縮に時間がかかるという欠点がある．本章では，携帯端末のソフトウェアを更新する際に，端末上で既存の Byte-Pair 圧縮方式，圧縮率を変更を伴わない高速なプログラムコード圧縮手法を提案し，実際の出荷ソフトウェアイメージを利用して評価し，効果を示す．

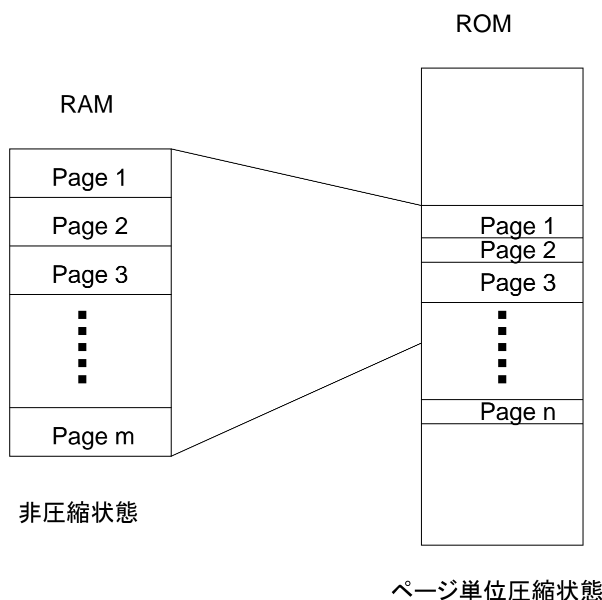

図 5.1: 圧縮 ROM イメージ

5.1 ソフトウェア更新と Byte-Pair 圧縮

5.1.1 携帯端末のメモリ構成

携帯端末上では，図 5.1に示すように，NAND 型フラッシュメモリ上にプログラムコードを RAM 上の非圧縮状態でのページに相当する単位ごとに圧縮して保持する．RAM 上で実行時にページフォルトを起こすと，該当ページの圧縮状態のプログラムコードを解凍して RAM 上にロードする．圧縮して保持するメリットは，NAND 型フラッシュメモリ領域の節約である．ローディングの性能的には圧縮してあると NAND 型フラッシュメモリを読む時間は短くなるが，展開する時間が余分に必要になる．そこで，高速に展開可能な圧縮アルゴリズムを採用する必要がある．

本章ではこのような圧縮に適したアルゴリズムである Byte-Pair 圧縮方式 [24] を対象とする．

5.1.2 Byte-Pair 圧縮方式

Byte-Pair 圧縮 [24][25] は，以下のような性質を持つことが知られている．

図 5.2: Byte-Pair 圧縮方式

- 圧縮率は ZIP，LZH より少し劣る
- 展開処理が高速で，デマンドページングなどに適している
- 圧縮処理は遅い

Byte-Pair 圧縮の基本アルゴリズムは，2 バイトデータを 1 バイトデータで置き換えることでデータを圧縮する方式である．図 5.2にその動作の例を示す．オリジナルデータ中の 2 バイトのペアで最も多いものとして，AB を選択し使っていない 1 バイト（以後，トークンと呼ぶ）即ち X に置き換えるという処理を行う．次に同様に最も多く現れるペアである XC を使っていないトークン Y に置き換える．このような処理を繰り返すという単純な方式で圧縮する．そのため，圧縮時間がかかるという欠点があるものの，展開時間はビット演算などを必要とせず高速である [27].

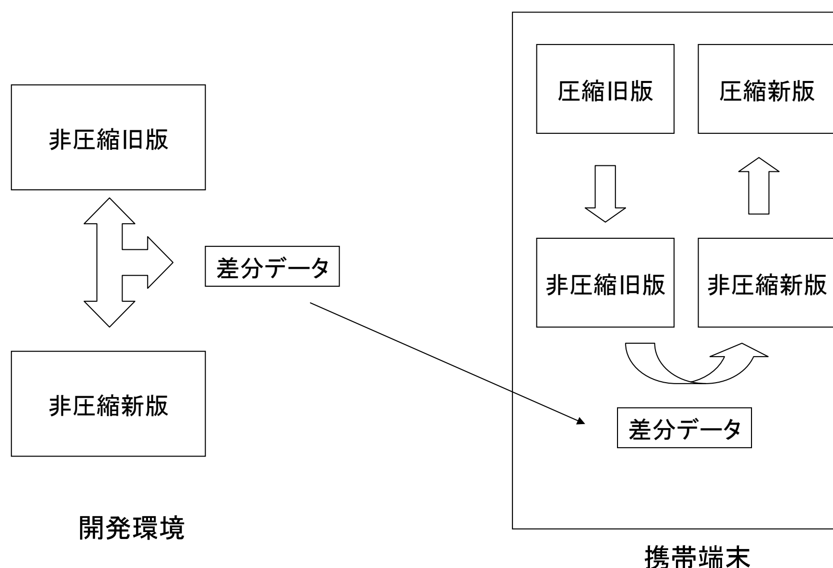

図 5.3: 差分更新

5.1.3 圧縮データと差分更新

定性的特性

わずかな不具合を修正し，非圧縮状態では旧版と新版の間に大きな差異がでない場合でも，圧縮状態になると差異が大きくなることが多い．これは Byte-Pair 圧縮に限らず一般に圧縮データに関する特徴である．そのため，圧縮状態の旧版から差分適用して新版を作成しようとすると差分データのサイズが大きくなる．

差分データサイズを小さく抑えるために，図 5.3に示すように差異の小さい非圧縮状態で差分更新を行い，端末上で圧縮状態にある旧版を RAM 上に展開し，差分更新後の非圧縮状態の新版を圧縮状態で保存する．更新時間は，データ展開時間とデータ圧縮処理の時間が余分に必要となる． Byte-Pair 圧縮では，展開時間は高速であるが，圧縮処理が遅いという欠点が大きな問題となる．

実データによる問題点抽出

実際の携帯端末のS/Wイメージを利用し，NAND型フラッシュメモリ上にByte-Pair圧縮した形でプログラムコードを置き，ソフトウェア更新機能を動作させる実験を行った．端末上でのソフトウェア更新機能は以下の手順で動作する．

(1) NAND型フラッシュメモリから旧版プログラムコードのRAMへの読み込み.

(2) RAM上での圧縮からの展開

(3) 差分データの適用による新版プログラムコードの生成.

(4) 新版プログラムコードの圧縮.

(5) 新版プログラムコードのRAMからNAND型フラッシュメモリへ変更のあった更新ブロック[1]の保存.

(6) 新版プログラムコードのベリファイ.

実験にはバージョンアップを3段階で行ったことを想定し，４つのバージョンを利用した．それぞれ$V1, V2, V3, V4$とする．それぞれのROMイメージのサイズを表5.1に示す．また，$V1 \rightarrow V2, V1 \rightarrow V3, V1 \rightarrow V4$の更新をそれぞれ，$UD1, UD2, UD3$と呼ぶこととし，表5.2にそれぞれの更新における変更のあったブロック数とページ数を示す．$UD1, UD2, UD3$の順に更新内容が大きくなり，修正量も多くなる．

また，複数のサンプルデータを適用し，Byte-Pair圧縮の速度，展開の速度，NAND型フラッシュメモリへの書き込みと読み込み，差分適用，ベリファイの各要素の基本性能を測定した．その結果を表5.3に示す．この基本性能から実データでの更新時間を計算した結果の各段階ごとの実行時間を表5.4に示す．

結果として，$UD3$の場合に，極端に圧縮時間が大きくなることがわかった．トータルで7分強の携帯端末を利用できない時間があるのに対してその中の6分が圧縮の時間であるため，更新時間が長くなるようなケースを改良するには圧縮の高速化が必須であることが確認できた．なお，$UD1$,$UD2$は修正量が小さく，圧縮時間は影響はあまりない．よって，修正が広い範囲にわたる場合が問題となる．

[1]NAND型フラッシュメモリは複数のページからなるブロック単位でデータを消去する．そのため，わずかな変更であってもブロック単位で消去し，ブロック内の全ページを書き戻す必要がある．

表 5.1: ROM イメージサイズ (KByte)

	V1	V2	V3	V4
全体	59,716	59,716	59,720	59,720
対象外	6,996	6,996	6,996	6,996
対象圧縮状態	52,720	52,720	52,724	52,724
対象非圧縮状態	85,100	85,100	85,100	85,100

表 5.2: 変更ブロック数とページ数

更新内容	変更ブロック数	変更ページ数	展開後サイズ ☆
UD1	361	13	54,008,932
UD2	324	129	54,179,307
UD3	468	4812	62,141,778

☆ 展開後サイズの単位は byte

5.2 提案方式

5.2.1 基本的なアルゴリズム

更新時間の大半を締める Byte-Pair 圧縮処理は，最大出現頻度のペアを見つける処理とペアを使っていないトークンに置き換える処理の繰り返しである．このペアとトークンの組み合わせを辞書として持ち，解凍時にはこの辞書を利用して解凍する．

圧縮時にはこのペアを見つける処理で何度もデータを走査するため最も時間がかかる．そこで，図 5.4に示すように，新版と旧版の圧縮前の差分データである更新データに圧縮用の辞書情報を付加しておき，辞書情報を利用して圧縮処理することにより繰り返し走査することを防ぎ高速化する．

表 5.3: 基本性能

項目	性能
Byte-Pair 圧縮	60.7KB/ 秒
Byte-Pair 展開	13.16MB/ 秒
NAND フラッシュ書き込み	2.13MB/ 秒
NAND フラッシュ読み込み	4.27MB/ 秒
差分適用	20.49MB/ 秒
ベリファイ計算 ☆	98.36MB/ 秒

☆ ベリファイ計算は NAND 型フラッシュメモリの読み込み部分を除外した性能

表 5.4: 更新時間（秒）

	UD1	UD2	UD3
旧版読み込み	14	14	14
旧版展開	0	0	1
差分適用	3	3	3
新版圧縮	1	9	317
新版書き込み	21	19	27
ベリファイ	14	14	14
全体	43	49	376

ただし，辞書データを付加するため更新データが大きくなるという欠点を持つ．しかし，更新データが大きくなってもバックグラウンドでのデータのダウンロードは可能であり，ダウンロード時間が長くなることによるユーザデメリットは少ない．

5.2.2 実現方式

開発環境の PC 上で新版を圧縮後，その圧縮辞書からトークンとペアの組み合わせを抽出し，図 5.5に示すように先頭からトークン数などを含む固定長の数バイトの制御情報およびトークンとペアを順番に保持する．

この辞書を利用し，携帯端末上での圧縮処理では，最も多いペアを見つけるという走査を省き，ペアをトークンに置き換えるという処理のみを実行する．ここで，圧縮の際に適用する順序は PC 上でも携帯端末上でも一致した方が良い．違う順序で適用しても圧縮は可能であるが，圧縮率が劣ってしまうことになるためである． PC 上での圧縮と同じ順にデータを置き換えていくことにより元と同じ圧縮がかかった状態とできる．

更新データに追加される辞書のサイズは，ページ単位に圧縮しているため，次式で示されるバイト数が更新のあったページ数分加わる．トークン数を n, 固定長の数バイトの制御情報を a とする．

$$DictionarySize = a + (n) * 3 \tag{5.1}$$

よって辞書のサイズはトークン数 n に比例して大きくなる．

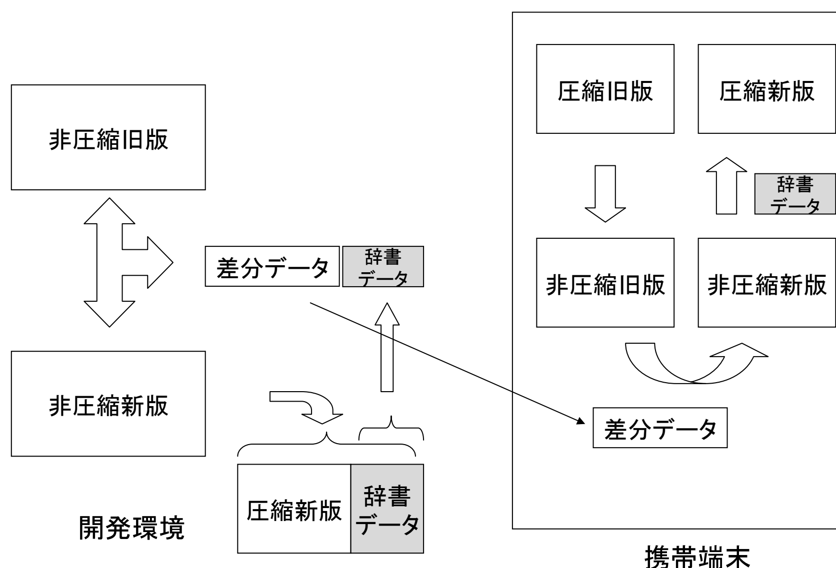

図 5.4: 提案方式

5.2.3 提案方式の性能測定と評価

表 5.5にサンプルデータを利用して実環境で評価した圧縮速度を示す．約 6 倍の高速化となることがわかった．従来の方式で問題となった $UD3$ のパターンに対する更新時間を表 5.6に示す．更新時間全体で，376 秒かかっていたのが，112 秒で更新が完了することがわかり，十分効果があることがわかった．

トレードオフとなる更新データの増加を表 5.7に示す．その結果更新データサイズが従来方式と比べどの程度大きくなったかを表 5.8に示す．対象外と記載している部分はどうしても高速化が必要で，圧縮すらしない部分である．辞書データは更新データの約 2 倍弱程度になっている．少々のデータの増加であれば問題がないが，3 倍となるとキャリア側にとってはネットワーク負荷の問題が発生し，ソフトウェアの更新費用をメーカで負担する場合にはコスト的にも問題となることがわかる．ただし，圧縮状態で差分を抽出すると，$UD3$ の場合で約 9M バイトにもなるため，差分更新を使った方が良いことは変わらない．

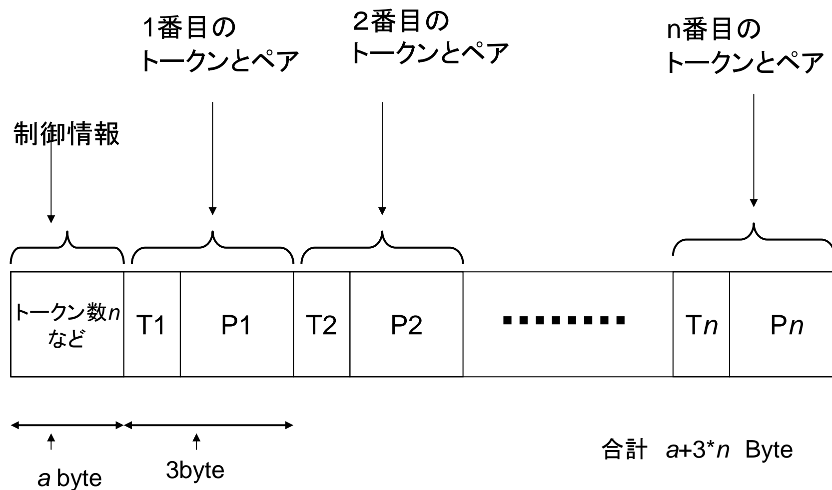

図 5.5: 提案方式の実現

表 5.5: 提案方式の圧縮性能

	従来方式	提案方式
圧縮速度 (KB/sec)	60.7	364.3

5.3 改善提案

ここまでの検証で付加する辞書データ量が多いことがわかった．このデータを削減する方法を検討する．以下の手法がまずは考えられる．

(1) 辞書データを圧縮する．

(2) 旧版辞書と新版辞書の差分をとって差分データを付加する．

(3) 圧縮にかかる時間によっては，圧縮用辞書を送らないページを指定するハイブリッド方式とする．

表 5.6: 提案方式更新時間（秒）

	UD3(従来方式)	UD3(提案方式)
旧版読み込み	14	14
旧版展開	1	1
差分適用	3	3
新版圧縮	317	53
新版書き込み	27	27
ベリファイ	14	14
全体	376	112

表 5.7: 提案方式による更新データ増加

項目	値
トークン数合計 (4812 ページ分)	458,593
1 ページあたりのトークン数 (平均)	95.3
1 ページあたりのトークン数 (最大)	205
1 ページあたりのトークン数 (最小)	3
辞書サイズ (4812 ページ分)	1,385,403 byte

5.3.1 辞書データの圧縮

辞書データの圧縮を考える．通常のテキストデータであれば 50% 程度の圧縮が期待できるかもしれないが，逆に展開時間が余分に必要となる．まずは，$UD3$ のパターンの付加辞書データを Byte-Pair 圧縮および ZIP 圧縮した結果を表 5.9に示す．この結果からほとんど圧縮の効果がないことがわかる．

Byte-Pair 圧縮では頻度の高いペアをトークンに置き換えることによって圧縮を行うが，1 ページ分の辞書データには同じペアは出現しない．また，この圧縮でトークンとして利

表 5.8: 提案方式による更新データサイズ

UD3	従来方式	提案方式
対象外	220,964	220,964
対象領域	522,245	522,245
変更ページ情報	19,248	19,248
辞書データ	0	1,385,403
更新データサイズ	743,209	2,147,860 byte

表 5.9: 辞書データ圧縮サイズ

圧縮方式	圧縮前 データサイズ	圧縮後 データサイズ	圧縮率
Byte-Pair	1,385,403byte	1,371,178 byte	99.0%
ZIP	1,385,403byte	1,335,453byte	96.0 %

用すべきバイトを既に元の圧縮のためのトークンとして利用しているため利用していないバイトが少ない．そのため，そもそも辞書データには Byte-Pair 圧縮は効果がない．

また， ZIP の圧縮にしてもページ間で考えると同じペアの出現は多いと考えられるが，トークンまで同じものを利用している確率は低く， 2 バイト程度の同じデータしか現れず，圧縮の効果が期待できない．そのため，辞書データの圧縮では，更新データに付加される辞書データを削減することはできない．

5.3.2 差分辞書データ

Byte-Pair 圧縮の基本的な考え方では，ページの内容がわずかに変わった程度では，データの出現頻度はプログラムコード上のアドレス部のみでトークンとペアの組はそれほど変化がないことが期待できる．よって辞書データには変化がないか，変化が少ないと期待できる．そこで，辞書データを旧版の辞書データとの差分情報として更新データ内に置くことにより，更新データに付加する辞書データの量の削減ができると考えた．

図 5.6に動作概要を示す．新版を圧縮した際の辞書情報を使って旧版の辞書との差分を開発環境上でとり，圧縮のための順序情報を付加して携帯端末に送る．

ここで，トークンとペアの組み合わせ順序情報は全く同じ状態に圧縮するためには必要な情報である．展開用の辞書は展開を高速に実行するためにこの順序をトークンでソーティングするなど変更していることが多い．即ち，圧縮用辞書と展開用の辞書は目的が違うために保持方法が変わるべきものである．例えばソーティングされた状態から元の状態に戻すには，図 5.7に示すような順序情報を差分データに含めなければならない．

次に，更新パターン $UD3$ を用いて辞書データ差分の有効性を検証する．表 5.10に旧辞書データと新版辞書データを比較し，一致するトークンとペアの組み合わせがどの程度になるかを示す．全く一致したページがわずか 1.97% であり，全体の一致率も 18.2% 程度であり，順序情報も送ることを考えると，この方法も効果が薄いことがわかる．

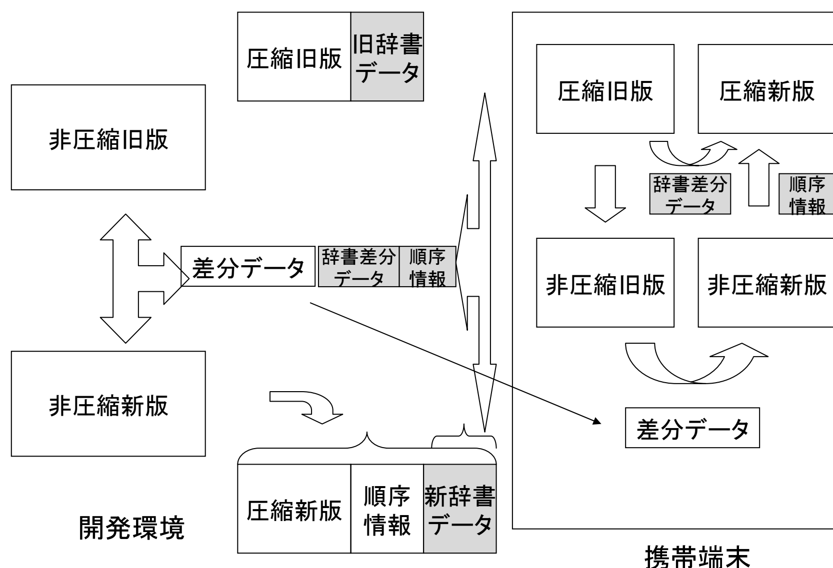

図 5.6: 差分辞書データ方式

5.3.3 ハイブリッド方式

ページごとに辞書データを用いるべきか，そうでないかを分けることによって，更新時間と転送データ量のバランスのとれた形にできない検討した．まず，圧縮時間とトークン数の関係をサンプルデータを下に調べた結果を図 5.8に示す．従来方式，提案方式に関係なくトークン数が多いと圧縮時間が増加することがわかる．これは，トークン数と変換回数が比例することによるが，多少のばらつきがある．このばらつきに着目する．

データサイズと更新時間のバランスをとるためには，辞書サイズは (5.1) 式で示されるようにトークン数に比例するので，トークン数は多いが辞書を使わなくても圧縮時間が小さいページを提案方式の適用対象からはずすと良いと考えられる．これは PC 上での圧縮処理の計算時間や，実機上で実際に測定することにより判断できる．そこでこの閾値を比例定数 k と考えると，図 5.8上の $y = kx$ より下の部分が，トークン数の割りには圧縮時間がかからないページであることがわかる．

実機上での動作時間を測定した結果に基づいて，対象からはずす閾値を変更した場合にどうなるのかを図 5.9に示す．比例定数 k を境に閾値を超えるか超えないかで判断する．この k の値に応じて，左側の軸では辞書データサイズを示しており，右側の軸で圧縮時間

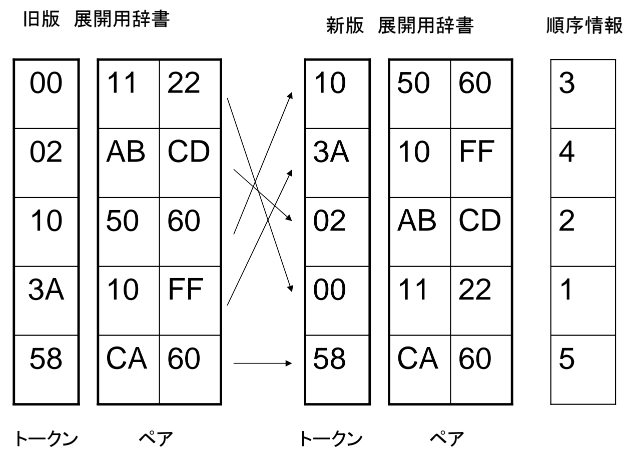

図 5.7: 辞書中の順序情報

表 5.10: トークンとペア

項目	値
トークン & ペア数合計	458,593 個
一致トークン & ペア数合計	83,506 個
比率	18.2 %
トークン & ペア完全一致ページ数	95 ページ
比率	1.97 %

を示した．例えば，十分 k を大きくとれば，すべての領域が高速化の対象からはずれるため，従来方式の値になる．逆に十分小さく k をとるとすべてが高速の対象となる．

さらにその結果更新時間がどうなるかを表 5.11に示した．この結果から，閾値である比例定数 k を適用先や，適用イメージの大きさに左右されるものの，携帯端末などでは 0.7 〜 0.8 に設定するとバランスがとれた状態になると考えられ，ユーザビリティの向上の観点から決める書き換え時間の許容範囲と，キャリア側の網への負担という観点やメーカのコスト的負担という観点から決めるデータ量の許容範囲の双方にあわせるために k の値を調整すれば良いと考える．

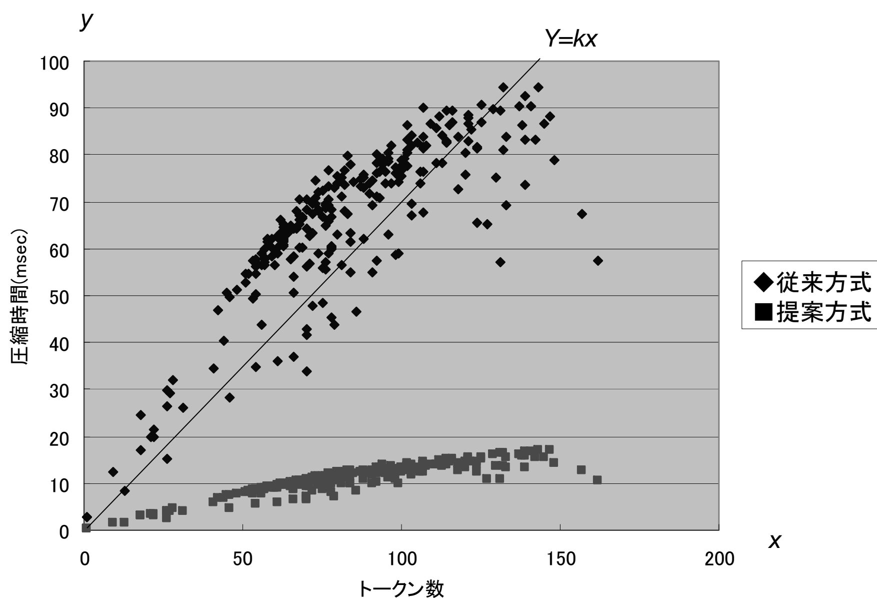

図 5.8: 圧縮時間とトークン数

5.4 本章のまとめ

今後普及してくるであろう，NAND 型フラッシュメモリを搭載し，デマンドページング方式を採用する携帯端末において，ソフトウェア更新の問題点として端末上での圧縮速度の面があることを示した．また，この課題に対して，高速化する方式を提案し評価した．提案した方式では約 6 倍の圧縮速度を実現できることがわかった．

しかし，更新用のデータが辞書データの増加により，約 3 倍弱になることがわかり，その改良方式を提案，評価した．改良方式は適用する対象によって最適な値をチューニング

表 5.11: 更新時間と閾値 k

比例定数 k	更新時間 (秒)	更新データサイズ (byte)
全高速化	112	2,147,860
0.5	115	2,124,953
0.6	131	1,984,910
0.7	167	1,770,818
0.8	239	1,371,569
従来方式	376	743,209

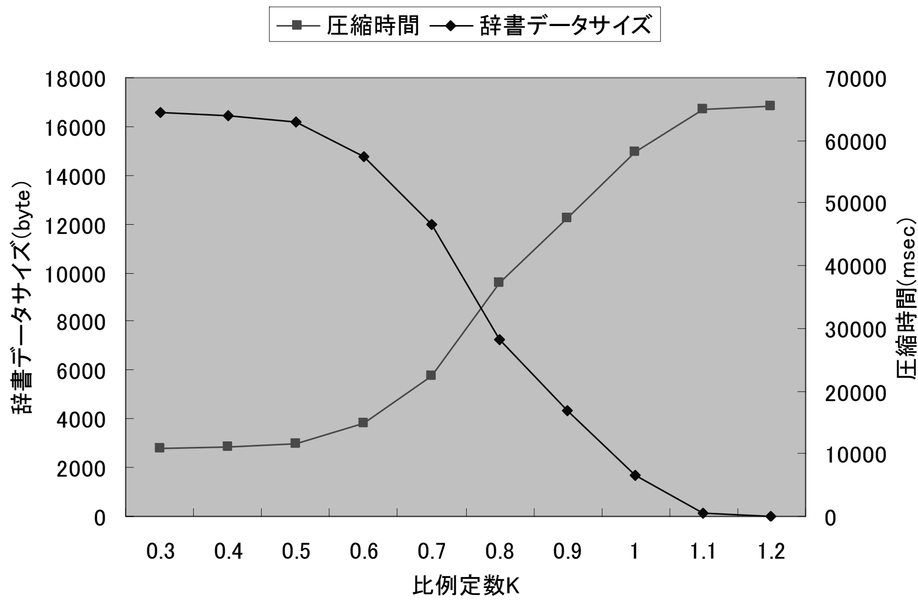

図 5.9: 辞書サイズと圧縮時間

することによりある程度は解決できることを示した．

また，最近のソフトウェア更新サービスではユーザの気づかないうちに自動的に更新するケースも増えてきている．ダウンロード中にはユーザが利用しようとしても問題なく使えるため，ダウンロード時間が長くても問題はない．しかし，書き換え中はユーザはサービスを利用できない．そのため，ユーザへのサービスという観点からは書き換え時間を短くすることがのぞましい． そういう観点では今回提案の手法は十分効果がある．

一方で，メーカやキャリアの立場ではデータ通信量が大きくなるのは好ましくない．そこで，今後さらに辞書データの特性に注目した上で，更新データに付加しなければならない辞書データの削減を検討していく必要があると考える．

第 6 章

Java 実行高速化方式

本章の概要

インターネットの発展に伴って，ポータビリティ，セキュリティを提供する Java 言語 [34] が普及し，近年は，携帯電話を代表とする多くの携帯端末に Java 仮想マシン [35](以下，Java VM) が搭載されている．携帯端末に Java VM が搭載されることで，異なるプラットフォーム上で実行できるアプリケーションを開発することが可能となり，プログラムの安全な実行とプログラム配信が可能となった．携帯端末には，ゲームや株価表示，案内等の様々なアプリケーションが提供されている．

本章では，メモリ量が少ない携帯端末上で，高速シューティング・ゲームを代表とするユーザ操作が伴う Java アプリケーションを，ネイティブコードに変換する時間を考慮しながらネイティブコンパイラを用いて高速化する方式を提案し，実装およびその評価について述べる．

6.1 携帯端末の特性と機能

6.1.1 JIT 方式による高速化の課題

携帯端末向けの Java VM は，アーキテクチャ非依存コードであるバイトコードを実行するために，インタプリタを用いる実装 [36][37] が一般的であった．しかし，インタプリタは，バイトコードを逐次解釈して実行するために実行速度が遅いという問題がある．携帯電話の Java アプリケーションでは，実行速度を重視するゲームが多く配信されており，特に，図 6.1のような高速シューティング・ゲームでは実行速度の遅さは重要な問題である．

インタプリタの実行速度の問題は，携帯端末に限らず Java VM の一般的な課題である．実行速度を向上するために，バイトコードを CPU が直接実行できる命令 (以下，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を搭載する取り組み [38][39][40] が行われてきた．しかし，ネイティブコンパイラを用いた高速化方式では，変換したコードを保持するメモリが必要となるため，メモリ資源の限られた携帯端末に搭載する際にはそのままでは適用できず，使用するメモリ量を削減しなければならない．ネイティブコンパイラが使用するメモリ量を削減するためには，ネイティブコードに変換するバイトコードを頻繁に実行されるホットスポットに限定し，その他のバイトコードはインタプリタを用いて実行する，バイトコードを選択して変換する方式 [41] が有効である．

ネイティブコンパイラを利用しない高速化方式として，バイトコードを直接実行する Java アクセラレータの搭載があげられる．Java アクセラレータは，ネイティブコードを格納するメモリ領域を必要としない．しかし，Java 専用ハードウェアである Java アクセラレータのコストが製品コストの増大につながる．メモリ資源は，Java VM の実行速度の向上以外の用途にも利用することが可能であり，製品コストの低減が必要な携帯端末では，メモリ資源への投資を選択し，ネイティブコンパイラを利用すると考えられる．

また，ネイティブコンパイラを用いた高速化方式を，実行速度を重視するゲームに適用する場合には，ネイティブコンパイラの変換処理時間を考慮する必要がある．特に，高速シューティング・ゲームでは，ユーザ操作に対する応答時間の制約が厳しく，ユーザ操作に対する応答を遅延させてはならない．

6.1.2 通信量に対する影響

ネイティブコンパイラを用いた高速化方式として，PDA における高速化 [79] のように，Java クラスファイルの全体又は一部を，携帯端末にダウンロードする前にネイティブコー

図 6.1: ゲーム操作画面

ドに変換する方式がある．しかし，変換したネイティブコードは変換前のバイトコードの数倍に増加するため，Java アプリケーションをダウンロードする際の通信量が増大する．表 6.1に，Java クラスライブラリのパッケージに含まれるバイトコードをネイティブコードに変換した際の増大率を示す．表 6.1は，バイトコードとネイティブコードの非圧縮時のサイズを示しており，変換したネイティブコードが約 7 倍から 8 倍程度に増加していることがわかる．

通信量の増大は，ユーザ，通信事業者，プログラム開発者のそれぞれに問題を発生させる．携帯端末のユーザは，課金される通信サービスを用いて Java アプリケーションをダウンロードして利用する．ダウンロード前にネイティブコードに変換した場合，ダウンロードファイルのサイズが大きくなり，ユーザの通信料の負担が増大する．通信サービスを提供する通信事業者は，通信量の増大の対策として通信網の整備という新たな投資が必要となる．

プログラム開発者は，通信事業者の通信網の設備投資を抑える処置により影響を受けることになる．通信事業者は，通信量を抑制するために，ダウンロードファイルのサイズに制限をかけることが一般的である．そのため，プログラム開発者は，クラス名やメソッド名等のシンボル情報を短縮するツール等を用いて，ダウンロードファイルに含まれるロジック，データ容量を確保しなければならない状況にある．実際に，後述の判定シミュレーションに用いた 3 つのゲームにおいても，表 6.8，表 6.9，表 6.10に示すようにクラス名，メソッド名の短縮が行われている．ネイティブコードの変換による通信量の増大は，ダウンロードファイルに格納できるプログラムや画像データ量を圧迫することになり，プログラム開発者は，プログラム，画像データの削減に迫られる．Java アプリケーションを有料コンテンツとして提供するコンテンツプロバイダにとっては，魅力あるコンテンツを提供できなくなる恐れもある．

通信量に対する影響を考えた場合，携帯端末上でネイティブコードに変換することが必

表 6.1: コンパイルコードのサイズ

パッケージ	バイトコード (KB)	ネイティブコード (KB)	増加率
java.io	3.9	34.8	8.9 倍
java.lang	1.5	11.8	7.8 倍
java.util	3.0	26.6	8.5 倍

要である．

6.1.3 メモリ量に対する影響

ネイティブコンパイラを用いた高速化方式では，変換したコードを保持するメモリ (以下，コードキャッシュ) を使用する．常にネイティブコードを実行する JIT(Just-In-Time) 方式の Java VM では，コードキャッシュのメモリ容量が不足した場合に，ネイティブコードの追い出しとバイトコードの変換処理が頻繁に発生し，ネイティブコードに変換する処理時間が増加して性能向上が得られなくなる．

プログラムの挙動に関するデータの取得 (以下，プロファイリング) を行うことで，頻繁に実行されるホットスポットを検出，ネイティブコードに変換して実行し，その他のバイトコードはインタプリタを用いて実行する HotSpot 手法では，ネイティブコードを格納するメモリ量が少ない場合でも，ホットスポットを選択してネイティブコードに変換するため，変換処理時間が増加せず，効率的に性能を向上できる．

しかし，携帯端末のようなメモリ資源が限られた環境では，変換したコードを保持するメモリ容量が非常に限られていることが考えられ，ホットスポットを検出するプロファイリングの精度を上げることが必要である．

プロファイリングの精度を上げるためには，プロファイリングに必要なメモリ量，処理時間が増大する．プロファイリングの方法には，1) 計測コード方式，2) サンプリング方式の二つの方式 [45] がある．計測コード方式では，ホットスポットを検出するために，メソッドの呼出コードにカウンタ値を増加させる等の計測コードを追加する．サンプリング方式では，実行中のスレッドのスタックからメソッド等の情報を取得するサンプリングコードを周期的に実行する．

計測コード方式は，メソッドの呼出時に計測用コードを実行させるために，サンプリング方式に比べてオーバヘッドが大きい．また，プログラム内のメソッド数が多いとプロファ

イリングに用いるメモリ容量が増大する．サンプリング方式は，周期的にプロファイリング処理を実行するため，各メソッドの呼出毎にプロファイリング処理を実行する計測コード方式に比べてオーバヘッドが少ない．しかし，サンプリング方式は，一定周期毎にプロファイルが行われるため，ホットスポットの検出の精度が下がる．

メモリ量に対する影響を考えた場合，ホットスポットの検出精度が高く，少ないメモリ量で動作するオーバヘッドの少ないプロファイリング機能を実現することが必要である．

6.1.4 ユーザ応答時間に対する影響

ネイティブコンパイラを用いて Java アプリケーションを高速化する場合，アプリケーションの動作も考慮する必要がある．高速シューティング・ゲーム等のアプリケーションは，ユーザの代理物であるキャラクタを，前後左右の移動キー及びジャンプやミサイル発射等のアクション・キーを押下することで操作し，敵や障害物，弾丸等の移動物体の位置を計算し，各移動物体の衝突を検出し，移動物体の描画処理を行う．

そのため，キー操作に対する処理，位置計算と衝突検出の処理や描画処理は，多くの実行時間を使用し，ホットスポットとして検出される．ゲーム操作中に，これらの処理を実装したバイトコードを変換する処理を行った場合には，ネイティブコードに変換する処理時間が，ユーザ操作に対する応答を遅延させて，画面が一瞬停止する問題 (以下，瞬停) が発生する．

また，プロファイリング方法としてサンプルリング方式を用いた場合，ユーザ操作に対する応答時間が一定時間とはならない．サンプリング方式では，周期的に各スレッドのメソッド等を調査するために，ユーザ操作を処理するコードとは無関係にサンプリングコードが実行されるからである．

ユーザ応答時間に対する影響を考えた場合，ネイティブコンパイラを用いて各処理の実行速度を上げると共に，プロファイリング，ネイティブコードに変換する処理の影響を削減し，ユーザ操作に対する影響を最小限に留める機能を実現することが必要である．

6.2 提案方式

メモリ資源の少ない携帯端末では，多くのネイティブコードを保持することはできない．そのため，頻繁に実行されるホットスポットをネイティブコードに変換する方式を採用する．本論文で提案するネイティブコンパイラを用いる高速化方式は，図 6.2に示すように，静的な解析による変換処理と動的な解析による変換処理を行うことでネイティブコードを生成，実行する．その他の部分はインタプリタ実装を用いてバイトコードを実行する．

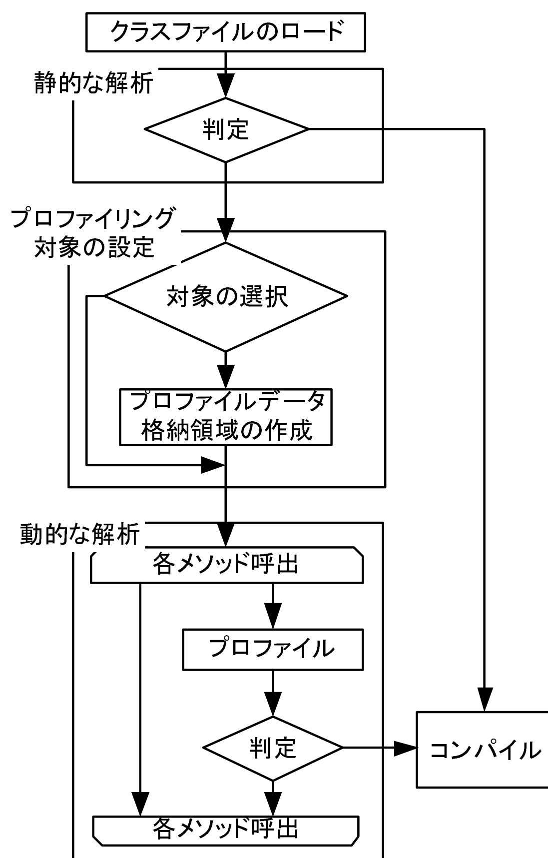

図 6.2: 解析とプロファイリング

6.2.1 静的な解析

ユーザ操作が伴う Java アプリケーションでは，瞬停が発生する恐れがあるために，アプリケーション実行中のネイティブコードに変換する処理時間に対して厳しい制約を設けなければならない．そのため，アプリケーションの構造を静的に解析してホットスポットを判定し，アプリケーションの起動前のクラスロード時にネイティブコードに変換する手法を提案する．

以下に，静的な解析に用いる判定方法を記載し，表 6.2に，ネイティブコードに変換するメソッドを示す．

メソッドサイズによる判定

頻繁に実行される部分として開発者が判断したコードは，最適化が行われる．頻繁に呼び出されるメソッドは，呼び出し時のオーバヘッドを削減するために，インライン展開が行われ，一つのメソッドが肥大化する傾向がある．

サイズの大きいメソッドは，頻繁に実行される部分として判断し，クラスロード時に一定のサイズ以上のメソッドをネイティブコードに変換する．データ初期化を行うコンスト

ラクタもサイズが大きくなる傾向にあるため，この条件に合致する．しかし，インスタンス生成に伴うデータの初期化は頻繁に実行されないため，この処理の例外としてネイティブコードに変換しない．

フレームワークによる判定

携帯端末に搭載される Java は，MIDP[63] や i アプリが定義するユーザインタフェースを実現するためのフレームワークを提供する．Java アプリケーションは，フレームワークに含まれるキー，描画イベントを処理するメソッドをオーバライドすることでキー操作，描画処理を実現する．

キー，描画に対する処理は頻繁に実行される部分であり，クラスロード時に，アプリケーション内のキー，描画イベントを処理するメソッドをネイティブコードに変換する．

呼出メソッドによる判定

携帯端末に搭載される Java VM は，CLDC 仕様 [64][65] に準拠しており，ユーザ定義のクラスロード機能を提供していないため，携帯端末上に組み込まれたシステムクラス以外のメソッドは，全てダウンロードファイルのユーザ定義クラスに含まれる．また，Java アプリケーションのサイズ制限が厳しいことから，ユーザ定義クラスでは，クラス継承やインタフェース定義を利用するような実行時にしか特定できない動的なメソッド呼出を用いておらず，各インスタンス内のメソッドを呼び出していると考えられる．また，プログラム開発者は，メソッド呼出の効率化を期待して頻繁に呼び出されるメソッドを private とする傾向がある．

以上のことから，頻繁に実行される部分から呼び出される private メソッドは頻繁に呼び出されるメソッドであると考えられる．本方式では，クラスロード時に，キー，描画イベント処理メソッドから呼び出される private メソッドをネイティブコードに変換する．

コードキャッシュの追い出し制御

本方式では，静的な解析で変換したコードをコードキャッシュから追い出さないように制御する．コードキャッシュの追い出し制御を行うことで，プロファイリングを用いた動的な解析による判定を回避し，アプリケーション動作中のネイティブコンパイラの実行を抑制することができる．

表 6.2: 静的な解析によるネイティブコード変換

<table>
<tr><th>実装箇所</th><th>メソッド
形態</th><th>メソッド
サイズ</th><th>フレームワーク
(キーイベント，描画)</th><th>呼出メソッド</th><th>アクセス
修飾子</th><th>ネイティブ
コードに変換</th></tr>
<tr><td>クラス
ライブラリ</td><td colspan="5"></td><td>×</td></tr>
<tr><td rowspan="6">アプリ
ケーション</td><td>コンストラクタ</td><td colspan="4"></td><td>×</td></tr>
<tr><td rowspan="5">メンバ</td><td>大きなメソッド
(一定サイズ以上)</td><td colspan="3"></td><td>○</td></tr>
<tr><td rowspan="4">それ以外</td><td>イベント処理
メソッド</td><td colspan="2"></td><td>○</td></tr>
<tr><td rowspan="3">それ以外</td><td rowspan="2">イベント処理メソッド
から呼び出される</td><td>private</td><td>○</td></tr>
<tr><td>それ以外</td><td>×</td></tr>
<tr><td colspan="2">それ以外</td><td>×</td></tr>
</table>

6.2.2 動的な解析

周期的にプロファイリングを行うサンプリング方式は，オーバヘッドが少ないという利点がある．しかし，サンプリング方式では，ホットスポットの検出精度が悪く，プロファイリングが一定した処理時間とならないという問題点がある．本方式では，ユーザ操作に対する応答時間の影響を考慮して，オーバヘッドが一定している計測コード方式を採用する．

また，プロファイリングに必要なメモリ量を低減するために，頻繁に実行される可能性が高い部分のみをプロファイリングの対象として選択する．

以下に，動的な解析に用いるプロファイリング対象の選定方法，プロファイリングの取得データについて記載し，表 6.3に，動的な解析に用いるプロファイリング対象のメソッドを示す．

クラスライブラリ・メソッドの選定

物体の移動，衝突判定と描画は，高速シューティング・ゲームに共通な処理であり，多くの実行時間を使用する．これらの処理の中で呼び出されるクラスライブラリのメソッドは，高速シューティング・ゲームに，共通に呼び出される使用頻度の高いメソッドと考えられる．事前に，幾つかの高速シューティング・ゲームについて，使用頻度の高いクラスライブラリ・メソッドを調査し，プロファイリングの対象メソッドの選定を行う．

表 6.3: 動的な解析に用いるプロファイリング対象

実装箇所	メソッドの事前選定	メソッドサイズ	プロファイリング対象
クラスライブラリ	選定内	小さなメソッド(一定サイズ以下)	○
		それ以外	×
	選定外	-	×
アプリケーション	-	小さなメソッド(一定サイズ以下)	○
		それ以外	×

メソッドサイズによる選定

瞬停を発生させないためには，ゲーム操作中に，ユーザに認識される変換処理を行わないことが必要である．ネイティブコンパイラの変換処理時間を推定するために，最適化処理を伴わずバイトコードをネイティブコードに対応付けて変換するのみの単純なネイティブコンパイラを採用する．本方式では，瞬停を発生させない変換処理時間となるように，一定のサイズ以下のメソッドをプロファイリング対象として選択する．

呼出回数による判定

プロファイリングの精度を上げるために，各メソッドの累積実行時間を計測することが考えられる．しかし，累積実行時間を取得するためには，メソッドが前回呼び出された時刻及び累積実行時間を保持する必要があり，多くのメモリ領域を使用する．そのため，一定時間内の呼出回数 (メソッドの呼出頻度) を計測するプロファイリングを行う．

6.3 評価

6.3.1 クラスライブラリ・メソッドの選定の有効性

クラスライブラリ・メソッドの選定が有効であることを確認するために， Java VM[36] のインタプリタにメソッドの実行時間を取得する機能を実装し， 7 個の高速シューティング・ゲームの各メソッドの実行時間を取得した．図 6.3は，アプリケーションのメソッドとクラスライブラリのメソッドのパッケージ毎の実行時間の割合を示している． 3/4 程度の

実行時間がクラスライブラリのメソッドが占められていることが分かる．クラスライブラリのメソッドをネイティブコードに変換するプロファイリング対象として設定することは，性能向上に有効である．

使用頻度の高いメソッドの処理

表6.4に示すように，実行時間が長い上位49個のメソッドで，クラスライブラリに含まれるメソッドの実行時間の50 %を占めていた．以下に，該当するメソッドを処理内容を示す．

(1) 描画処理

キャラクタ等の移動物体を描画するために，画像イメージ，線や多角形の描画処理が行われている．また，ディスプレイや描画範囲のサイズを取得するメソッドも頻繁に呼び出されている．アプリケーションが携帯端末への非依存性を保つために，常に値を取得して利用するためと考えられる．

(2) イベント処理

キー，描画イベントの取得及びイベント・キューの操作処理を行っている．ユーザインタフェースを実現するために，イベント配送，描画処理等を行う複数のJavaスレッドが実行されており，1割程度の時間がイベント処理に使用されている．

(3) 文字列操作

整数を文字列に変換する処理が頻繁に呼び出されている．点数や経過時間の表示に用いていると考えられる．

(4) 数値処理

絶対値と乱数値の取得が行われている．移動物体の位置計算に用いていると考えられる．

選定による効果

クラスライブラリ全体のメソッド数は2003個であり，50 %の実行時間を占める49個のメソッドは，クラスライブラリ全体の2.5% に過ぎない．また，49個のメソッドは，高速シューティング・ゲームに特有な処理に用いられている．それらのクラスライブラリのメソッドをプロファイリング対象として選択することで，プロファイリングによる処理及びメモリ使用量を大幅に低減することができる．

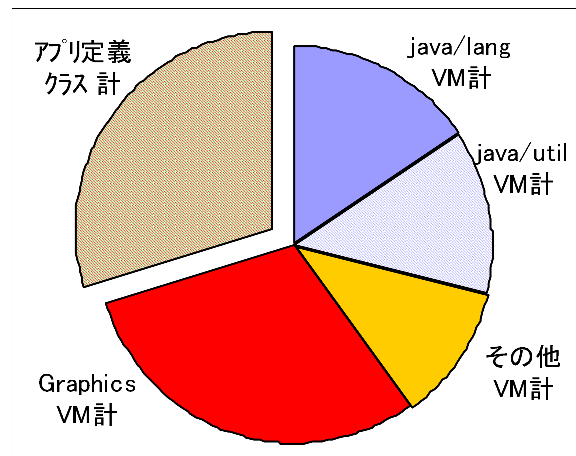

図 6.3: 高速シューティングのメソッド実行時間分布

6.3.2 静的，動的な解析による判定の有効性

静的な解析による判定，動的な解析による判定の有効性を確認するために，三つの高速シューティング・ゲームを用いて判定シミュレーションを行った．用いたゲーム A， B， C は，図 6.1にようにキー操作に伴い，敵，キャラクタ，弾丸等の移動体の位置を計算し，相互に衝突検知を行って移動物体の描画を行う．

ゲーム A 50 程度の敵が整列，移動する処理と共に，数十個の弾丸による攻撃が行われる．キャラクタは弾丸を避けながら敵を打ち落とす．移動体は単純な動作であるが移動体の数は多い．

ゲーム B 50 程度の敵が整列，移動する処理と共に，数個の弾丸，体当りによる攻撃が行われる．キャラクタは弾丸，体当りを避けながら敵を打ち落とす．敵の体当りは複雑な動作として実現されている．

ゲーム C 背景を描画することでキャラクタ自身を高速に前進させながら，正面から現れる 2， 3 の敵を攻撃する．描画する移動体は少ないが，移動速度を表現するために描画速度を必要としている．

シミュレーションでは，アプリケーション内クラスのメソッドのサイズ，呼出回数，実行時間の全体に対するメソッドの実行時間の割合を用いた．以下に，静的な解析，動的な解析による判定条件を記載し，効果について述べる．

表 6.4: 機能毎の実行時間の割合

機能	個数	実行時間 (%)
描画処理	24	26.90
イベント処理	12	9.66
文字列操作	10	9.60
数値処理	3	7.08
計	49	53.24

静的な解析による判定条件

(1) メソッドサイズによる判定

シミュレーションに用いたJava アプリケーションのダウンロードファイルは，10K バイトである．ダウンロードファイルはjar 形式であり，圧縮されている．また，ゲームに使用する画像データ，音声データも含まれている．表 6.8，表 6.9，表 6.10に含まれる全バイトコードは，それぞれ約 8K バイト，約 12K バイト，約 8K バイトである．ばらつきがあるが平均はほぼ 10K である．

圧縮と含まれる画像データ，音声データが相殺されると仮定し，アプリケーションに含まれる全バイトコードを 10K バイトとする．その 2 割である 2K バイト以上のサイズのメソッドをネイティブコードに変換する．

(2) フレームワークと呼出メソッドによる判定

描画イベントを処理するメソッド paint() とキーイベントを処理するメソッド keyEvent() 及びそのメソッドから呼び出される private メソッドをネイティブコードに変換する．

動的な解析による判定条件

(1) メソッドサイズによる選定

瞬停を発生させる時間を 1 フレームレート分の時間とする．動画表示と同様な 24 フレームレートが実現されると仮定し，1sec/24 = 42msec よりも少ない値である 10msec として設定する．変換処理時間の性能を 50μ 秒 / バイトと仮定し，10msec 以内でネイティブコードに変換処理可能なバイトコードのサイズは 200 バイトである．200 バイト以下のメソッドをプロファイリング対象として設定する．

(2) 呼出回数による判定

描画イベントを処理する paint メソッドは頻繁に呼び出されるため，プロファイリング処理の代用として，呼出回数の基準値とすることができる．判定シミュレーションでは， paint メソッドよりも呼出回数が多いメソッドをネイティブコードに変換する． 24 フレームレートであれば， 1 秒間に 24 回程度の頻度で呼ばれるメソッドをネイティブコードに変換することになる．

静的な解析による判定方法の効果

ゲーム A， B， C のそれぞれのシミュレーション結果を表 6.8，表 6.9，表 6.10に示す．変換対象となったメソッドを判定方法毎に ◯ を記載している．フレームワーク列の ⊕ を記載したメソッドは， paint メソッド， keyEvent メソッドから呼び出されているメソッドである．

(1) メソッドサイズによる判定

ゲーム A 静的な解析により変換対象となるサイズの大きなメソッド G$a::a()V が存在し，全体の実行時間の 54 %を占めている．

ゲーム B バイトコード全体の 18% を占めるメソッド M::loop()V が存在する．呼出回数は多いが全体の実行時間の 8% を占めているに過ぎない．

ゲーム C コンストラクタ以外は，サイズが 2K バイト以上のメソッドは存在しない．

メソッドサイズによる判定はゲーム B には有効性がない．メソッド数が少なく，大きなメソッドが存在するアプリケーションに対してのみ有効であると考えられる．

(2) フレームワークによる判定

ゲーム A paint メソッドは全体の実行時間の 37% を占めている． keyEvent メソッドは全体の実行時間の 0.74 %しか占めていない．

ゲーム B paint メソッドは全体の実行時間の 9% 程度を占めている． keyEvent メソッドは全体の実行時間の 1% しか占めていない．

ゲーム C paint メソッドは全体の実行時間の 7% を占めている． keyEvent メソッドは全体の実行時間の 0.04% しか占めていない．

ゲーム B， C では，描画イベントを処理する paint メソッドの全体に占める実行時間の割合はそれ程高くない．しかし，ゲーム B,C の paint メソッドは，動的な解析

による判定にてネイティブコードに変換できるサイズよりも大きく，動的な解析ではネイティブコードに変換されない．静的な解析にて描画メソッドをネイティブコードに変換することで，ユーザの応答時間に影響を与えずに実行時間の 7 %以上を占める処理を高速化できる．

キーイベントを処理する keyEvent メソッドは，アプリケーション全体の実行時間と比べると実行時間が短く，本シミュレーション結果から静的な判定による変換の効果は示せていない．

(3) 呼出メソッドによる判定

ゲーム A paint 及び keyEvent メソッドから呼び出されている private メソッドはない．

ゲーム B paint メソッドは private メソッドを呼び出さない． keyEvent メソッドから呼び出されている 2 個の private メソッドは，呼出回数が少なく，ゲーム操作に関する処理ではないと考えられる．

ゲーム C keyEvent メソッドは，クラス内のメソッドを呼び出さない． paint メソッドは，クラス内の 5 個のメソッドを呼び出している．メソッド 5 個の内の 1 個は，全体の実行時間の 9% の実行時間を占めており，呼出メソッドによる判定に効果があることを示している．その他の 4 個メソッドは， 1% 以下の実行時間を占めているのみである．

keyEvent メソッドからの呼出メソッドには，ゲーム操作中の処理とは関係しない処理が含まれており，呼出メソッドによる判定は有効に機能していない．コードキャッシュの効率的な利用を考えた場合，呼出メソッドによる判定で変換したメソッドのネイティブコードの実行頻度が少ない場合，コードキャッシュから追い出し，動的な解析に対してコードキャッシュの領域を明け渡すことが必要である．

動的な解析による判定方法の効果

ゲーム A アプリケーションはサイズの大きなメソッドから構成されており，ネイティブコードに変換可能な小さなメソッドはない． paint メソッドよりも多く呼ばれたメソッドは，静的解析による判定において変換対象とした a() だけである．

ゲーム B paint メソッドの呼出回数以上に呼び出されたメソッド 8 個の内の 5 個がネイティブコードに変換可能と判定される．

ゲーム C paint メソッド以上の呼出が行われた 8 個内の 4 個がネイティブコードに変換可能と判定されている．

ネイティブコードに変換が可能なサイズの小さなメソッドでも，実行時間の割合の高いメソッドが存在し，性能向上に効果がある．

総合的な効果

各判定方法の変換対象として設定したメソッドのサイズと実行時間に占める割合を表 6.5に示す．

ゲーム A 静的な解析によりサイズが大きいと判定したメソッドと描画イベントを処理するメソッド paint の実行時間を合わせた時間は，全体の実行時間の 91% を占めている．静的な解析による判定が有効であることがわかる．

ネイティブコードに変換した場合の高速化率を 3 倍と仮定すると，実行時間の 91% が 1/3 となるため，同一の処理を 40% 程度の時間で実行できる．

ゲーム B 動的な解析による判定において変換対象となるメソッドに含まれるバイトコードのサイズは，全体のバイトコードの 5% である．しかし，それらのメソッドの実行時間を合わせた時間は，全体の実行時間の 40% になる．動的な解析による判定が有効に機能し，実行時間が長いメソッドをネイティブコードに変換できることが分かる．静的な解析により変換されるメソッドの実行時間と合わせた時間は，全体の 61% である。

ネイティブコードに変換した場合の高速化率を 3 倍と仮定すると，実行時間の 61% が 1/3 となるため，同一の処理を 60% 程度の時間で実行できる．

ゲーム C 動的な解析による判定において変換対象となるメソッドに含まれるバイトコードのサイズは，全体のバイトコードの 5% である．しかし，それらのメソッドの実行時間をあわせた時間は，全体の実行時間の 40% になる．動的な解析による判定が有効に機能し，実行時間が長いメソッドをネイティブコードに変換できることが分かる．静的な解析により判定されたメソッドの実行時間と合わせた時間は，全体の 57% である。

ネイティブコードに変換した場合の高速化率を 3 倍と仮定すると，実行時間の 57% が 1/3 となるため，同一の処理を 60% 程度の時間で実行できる．

表 6.5: ゲーム A,B,C: 選択されたメソッドの実行時間とサイズ

<table>
<tr><th colspan="3" rowspan="2">コンパイル判定方法</th><th colspan="2">ゲーム A</th><th colspan="2">ゲーム B</th><th colspan="2">ゲーム C</th></tr>
<tr><th>サイズ%</th><th>実行時間%</th><th>サイズ%</th><th>実行時間%</th><th>サイズ%</th><th>実行時間%</th></tr>
<tr><td rowspan="6">静的解析</td><td colspan="2">サイズ大</td><td>44</td><td>54</td><td>18</td><td>8</td><td>0</td><td>0</td></tr>
<tr><td rowspan="2">フレームワーク</td><td>paint</td><td>17</td><td>37</td><td>10</td><td>9</td><td>7</td><td>7</td></tr>
<tr><td>key Event</td><td>5</td><td>1</td><td>7</td><td>1</td><td>5</td><td>0</td></tr>
<tr><td rowspan="2">呼出メソッド</td><td>paint</td><td>0</td><td>0</td><td>0</td><td>0</td><td>15</td><td>10</td></tr>
<tr><td>key Event</td><td>0</td><td>0</td><td>1</td><td>3</td><td>0</td><td>0</td></tr>
<tr><td colspan="2">計</td><td>66</td><td>91</td><td>36</td><td>21</td><td>27</td><td>17</td></tr>
<tr><td>動的解析</td><td colspan="2">サイズ小, 呼出回数</td><td>0</td><td>0</td><td>5</td><td>40</td><td>5</td><td>40</td></tr>
<tr><td colspan="3">合計</td><td>66</td><td>91</td><td>41</td><td>61</td><td>32</td><td>57</td></tr>
</table>

6.3.3 試作実装を用いた評価

ネイティブコードを常に実行する JIT 方式と提案方式を M32R 40Mhz を搭載した携帯端末上に実装した試作環境を用いて，ユーザ操作の応答時間に対する影響と速度性能の向上を評価した．試作環境は，ネイティブコンパイラを搭載した Java VM[36] に，コードキャッシュとして 32KB を設定している．

静的な解析による判定条件

判定シミュレーションの結果から一部の判定条件を修正し，試作実装を行った．

(1) メソッドサイズによる判定

判定シミュレーションにて，アプリケーションに含まれる大半のバイトコードが含まれた場合に有効性が示せた．全バイトコードがおおよそ 10K バイトであるため，4K バイト以上のメソッドをネイティブコードに変換するように実装した．

(2) フレームワークと呼出メソッドによる判定

判定シミュレーション結果から，キーイベント処理メソッドからの呼出メソッドの判定に有効性が認められなかった．そのため，キーイベント処理メソッドの呼出メソッドは静的な解析による判定の対象から除外して実装した．表 6.6に，静的な解析にてネイティブコードに変換するメソッドと追い出し抑制の設定を示す．

表 6.6: フレームワークによる判定

処理名	メソッド	判定	追出抑制
キー操作	本体	変換する	する
	呼出メソッド	変換しない	しない
描画処理	本体	変換する	する
	呼出メソッド	変換する	する

動的な解析による判定条件

(1) クラスライブラリ・メソッドの選定

第 6.3.1節において選定した実行時間の長い上位 49 個のクラスライブラリ・メソッドをプロファイリング対象とするように実装した.

(2) メソッドサイズによる選定

判定シミュレーションでは，瞬停を発生させる時間を動画表示と同様の 24 フレームレートとして設定して分析を行った．実際の高速シューティング・ゲームでは，12 フレームレートを基準として利用している．操作に対する応答時間に影響を与えるコンパイル時間を，1 フレームレートを下げる時間 (12 フレームレート:83msec) として定義した．評価に利用したネイティブコンパイラの 1 バイトあたりの変換時間 (25μsec) からコンパイル可能な小さなメソッドを 3.3K バイト以下のメソッドと設定し，プロファイリング対象となるように実装した.

(3) 呼出回数による判定

判定シミュレーションでは，paint メソッドよりも呼出回数が多いメソッドを高頻度であると設定して分析を行った．試作実装では，メソッドが一定回数呼び出された場合の経過時間を測定し，1 秒間に 10 回以上呼び出されるメソッドを頻繁な呼出であると判定するように実装した.

ユーザ操作に対する応答時間

JIT 方式を実装した Java VM では，ゲーム A の操作中に一瞬画面が停止する瞬停が発生した．JIT 方式を用いた場合，ネイティブコードに変換する処理とコードキャッシュからの追い出しが頻繁に発生していた.

本方式を実装した Java VM では瞬停は発生せず，応答時間に対する影響を排除することができている.

表 6.7: インタプリタに対する性能比

測定項目	JIT 方式	本方式
基本演算	4.95 倍	4.92 倍
ループ	4.95 倍	4.91 倍
メソッド呼出	2.44 倍	2.20 倍
記憶領域 I/O	1.35 倍	1.18 倍
パネル描画	1.23 倍	1.21 倍

速度性能の向上

携帯電話に搭載された Java において利用されるベンチマーク [66] を用いて評価を行った．表 6.7に，本方式のインタプリタ実行に対する性能向上率を示す．

本方式を用いることで，各測定項目においてインタプリタ実行と比較して速度性能が向上していることがわかる．特に，測定項目「基本演算，ループ」では，5 倍近い速度性能の向上が得られている．

提案方式の影響

表 6.7には，ベンチマーク [66] を用いて取得した JIT 方式によるインタプリ実行に対する性能向上率も合わせて示している．特定の処理のみが実行されるベンチマークプログラムでは，コードキャッシュの不足はおきず，キャッシュからの追い出しが行われ難いために，常にネイティブコードを実行する JIT 方式による測定結果が最高性能を示す．

そのため，本方式を用いた場合には，JIT 方式に比べて性能向上率が全般的に下回る．しかし，その差は最大で 0.2 倍しかなく，提案方式は速度性能に対して大きな影響を与えていない．以下に，JIT 方式と比べて性能向上率が低い理由を述べる．

(1) コンパイル判定までのインタプリタ実行時間

本方式では，ホットスポットを検出するまでのインタプリタにおける実行時間が含まれるため，JIT 方式と比べて性能低下が検出される．しかし，測定項目「基本演算，ループ」の性能差 0.02 倍が示すように僅かな差でしかない．

(2) プロファリング処理のオーバヘッド

測定項目「メソッド呼出」では，メソッド呼出が頻繁に行なわれるために，プロファイリング処理のオーバヘッドの影響を受けていると考えられる．

(3) プロファイル対象外のメソッド

測定項目「記憶領域 I/O」が頻繁に利用するストリーム処理のクラスライブラリ・メソッドをプロファイル対象と設定しておらず，インタプリタを用いた実行が行われている．

6.4 本章のまとめ

本章では，メモリ量が少ない携帯端末上で，高速シューティング・ゲームを代表とするユーザ操作を伴う Java アプリケーションを，ネイティブコンパイラを用いて高速化する方式の提案を行った．

本方式は，携帯端末の Java アプリケーションの特性であるダウンロード時の通信量，実行時のメモリ量の制約が厳しく，かつ，ユーザ操作に対する応答時間に制約がある高速シューティング・ゲームを代表とする Java アプリケーションに適用することを目的としている。

本方式では，頻繁に実行されるホットスポットを検出するために， Java アプリケーションの起動前にメソッドサイズ，フレームワーク，呼出メソッドの静的な解析を行い，アプリケーションの実行中にサイズおよび処理内容に基づいて選定したメソッドのみをプロファイリング対象とする動的な解析を行うことにより，プロファイリングに必要なメモリ量を低減し，ユーザ操作の応答時間を遅延させないネイティブコンパイラを用いた高速化を実現する．

評価では，インタプリタ実装を用いて取得したメソッドの実行時間，呼出回数を用いた判定シミュレーションを用いて，クラスライブラリ・メソッドの選定および静的，動的な判定方法の有効性の確認を行った．また，試作実装を用いた評価では，ユーザ操作に対する応答時間の影響を排除した上で， JIT 方式と同様に最大で約 5 倍の速度性能の向上を実現し，本方式の速度性能に対する影響が少ないことを確認した．

本方式の判定シミュレーション，試作実装による評価は， 10K バイト程度の Java アプリケーションを実行するメモリ量， CPU 性能を持つ携帯端末を用いて行った．現在， 100K バイト以上の Java アプリケーションが利用できる携帯電話が登場しており，評価に用いた環境と比較すると利用できるメモリ量が増加し，高速な CPU が利用できる．本方式は，メソッドサイズ，フレームワーク，呼出メソッドおよび処理内容に基づいた静的な解析，動的な解析を用いることに特徴が有る．そのため，より高速な CPU が利用可能な環境においては，ネイティブコードに変換する時間が短縮されるため，ユーザ操作に対する応答時間に影響を与えないで変換できるメソッドのサイズ制限を緩和することができる．また，

表 6.8: ゲーム A：各メソッドの実行時間とサイズ

メソッド	サイズ byte	呼出回数	静的解析		動的解析	実行時間 %
			サイズ大	フレームワーク	サイズ小	
G$a::a()V	3818	1647	○			53.68
G$a::paint(G;)V	1486	345		○		36.81
G$a::b(I)V	988	12				3.80
G$a::a([BI)I	312	10				1.60
G$a::c(I)I	518	5				1.55
G$a::run()V	156	1				1.31
G$a::keyEvent(II)V	416	31		○		0.74
G::start()V	34	3				0.32
G$a::a(I)V	256	10				0.17
G$a::<init>(LG;)V	731	1				0.02
計	8715	2065				100

より多くのメモリが利用可能な環境においては，プロファイリングに利用可能なメモリ量が増大し，プロファイリング対象として設定するメソッドの処理内容を拡大することができる．本方式では，メモリ量，CPU 性能の制約が緩和された場合にも，ユーザ操作に対する応答時間の影響を排除することでき，また，利用するメモリ量を調整することができる．

今後，試作実装による速度性能の評価において測定されたプロファイリング処理のオーバヘッドの調査とその低減を行い，サイズの大きな Java アプリケーションを実行できる，より高速な CPU，より多くのメモリが利用可能な携帯端末において，プロファイリング対象とするメソッドの選定方法を検討し，プロファイリング対象が増加した場合の影響を調査することで，本方式の完成度を高めていきたいと考えている．

表 6.9: ゲーム B：各メソッドの実行時間とサイズ

メソッド	サイズ byte	呼出回数	静的解析		動的解析	実行時間 %
			サイズ大	フレームワーク	サイズ小	
M::moveI(II)V	1312	276				13.72
M::paintI(L/G;II)V	69	322			○	13.81
M::paintB(L/G;)V	147	79			○	12.15
M::hitChk()V	952	71				11.98
M::paint(L/G;)V	1257	69		○		9.23
M::loop()V	2116	452	○			7.69
M::hitChkSh(IIII)Z	1224	6				6.77
M::rayHitChk(I)V	190	93			○	6.39
M::run()V	423	1				2.76
M::paintR(L/G;I)V	138	104			○	5.50
M::paintS(L/G;)V	109	130			○	2.24
M::goTitle()V	36	4		⊕		2.63
M::keyEvent(II)V	844	50		○		0.96
M::drawBe(L/G;I)V	170	11				1.65
M::fireR(I)V	250	11				0.82
M::drawSh(L/G;II)V	922	7				0.62
M::moveIBase()V	128	7				0.51
M::drawS(L/G;II)V	166	6				0.18
M::drawIe(L/G;II)V	35	4				0.22
I::start()V	952	1				0.05
M::drawR(L/G;)V	52	4				0.09
M::writeGData()V	56	1		⊕		0.01
M::<init>()V	530	1				0.01
計	12078	1710				100

表 6.10: ゲーム C：各メソッドの実行時間とサイズ

メソッド	サイズ byte	呼出回数	静的解析		動的解析	実行時間 %
			サイズ大	フレームワーク	サイズ小	
S::sort(I)V	332	115				18.91
S::pmiss()V	76	319			○	17.75
S::obj_bin()V	607	320				15.91
S::pDraw(L/G;)V	127	666			○	11.29
S::d_block(L/G;)V	535	310		⊕		8.82
S::chg_key(LS;LS;)V	13	310			○	7.40
S::paint(L/G;)V	555	237		○		6.90
S::shot()V	198	305			○	3.77
S::run()V	182	1				3.45
S::cmiss(I)Z	276	97				1.64
S::in_key()V	334	304				1.21
S::cshot(II)Z	544	26				0.95
S::dhot(L/G;I)V	241	6		⊕		0.87
S::getRndInt(I)I	12	473		⊕		0.65
S::clear()V	43	1				0.16
S::MkState(L/G;)V	378	58		⊕		0.12
S::dScore(L/G;LS;I)V	17	56		⊕		0.09
S::keyEvent(II)V	366	21		○		0.04
S::mAction(L/M;II)V	56	10				0.04
S::createI(ILS;)L/I;	38	4				0.02
S::terminate()V	26	1				0.01
S::score_save()V	111	1				0.01
S::ld_chk()Z	32	3				0.00
S::<init>()V	2596	2				0.00
S::score_load()V	84	1				0.00
S::init()V	37	1				0.00
計	7816	3648				100

第 7 章

携帯電話のユーザ操作予測の有効性

本章の概要

携帯電話のアプリケーションごとに異なる”時刻”や”位置”の扱い方に着目し，状況に依存したアプリケーション候補抽出方式の構成案を提案し，操作履歴の扱い方を中心に机上評価を行った．本章では，実際の操作履歴を用いて時刻と位置の分類モデルを作成することで，時刻と位置の状態空間を構成し，それぞれの状態においてどのアプリケーションが起動されたかを確率的に求めることの有効性を示す．

7.1 コンテキストを利用した携帯電話の操作予測

近年のインターネットの急速な発展も手伝い，携帯電話を始めとする携帯端末は日常生活にはなくてはならない情報管理における電子秘書としての重要な役割を担うパーソナルメディア端末となりつつある．

しかし，このような高機能化・複雑化が加速する一方，携帯し易いことも重要であり，携帯電話／端末の大きさは小型化が進んでいる．小型化が進む筐体には人が操作を行うためのボタンなどを自由に設置することはできず，限られた数と大きさのボタンなどを駆使して複雑化する機能を使いこなさなければならない．このことは，結果的に操作方法の複雑化を招き，操作方法の複雑化はユーザの負担となり，例えば目的とするアプリケーションの立ち上げに余計な時間を要する．

例えば，現在の携帯電話は GPS 機能も搭載されており，屋外にて目的地までのナビゲーションや，乗り換え案内など，いち早く調べ物をする際にもよく使われる．屋外であるが故に，できるだけ立ち止まらず，歩きながら操作できる方が望ましく，画面を見ての複雑な操作は避けたい．現在の携帯電話においても，短時間で作業できるようにショートカットなどを予めユーザが設定するなど，ユーザ独自のメニューを準備しておくことである程度は対応している．

しかし，頻繁に使用するアプリケーションについてはショートカットなどを登録することで十分かもしれないが，ユーザとしては使用頻度の低いアプリケーションに対してもメニューに登録するのは面倒であり，そのようなショートカットが増えてしまえば，結局は多くのショートカットから目的のものを選択する手間が増えることになり，問題が根本的に解決されていない．

一方，ユーザは携帯電話を ”時刻 ”，”位置 ”，”それまでの操作の状況 ”や ”日々のスケジュール ”などの様々な外的要因に依存して利用すると考えるのは自然である [47]．よって，個人の携帯電話の使用方法に特化したユーザインフタフェース (以降 UI と記す) を作ることが出来れば，おおいに有用であることが推測されるが，個人ごとに特化した UI をすべて用意することは非常に困難である．

この問題に対して，各個人における ”時刻 ”や ”位置 ”などの外的な要因に基づく携帯電話の利用の仕方を自動的に抽出できれば，時刻や位置などに基づいてユーザが利用するであろうアプリケーションを予測できる可能性が期待できる．そして，UI において利用することが予測されるアプリケーションを事前に起動することで，ユーザのアプリケーションを起動するまでに複雑な操作を軽減することが可能となる．

そこで本章では，実際に携帯端末の操作ログならびに GPS による携帯端末を操作した

表 7.1: 取得した履歴

種類名	データ量
操作日数	38 日
操作回数	5140 回
Web メール使用回数	423 回
航空会社 Web サイトアクセス回数	95 回
乗り換え案内アクセス回数	19 回

位置を一定期間記録するとともに，その操作履歴から時刻と位置で利用したアプリケーションを自動的に分類し，実際に行動パターンとアプリケーションに何らかの関係が認められるかについて分析した．

次に，利用される頻度の特に高いいくつかのアプリケーションに対する，時刻と場所との関係における決定木抽出を行った結果を上記 3 次元空間上にマッピングした結果，明らかに特徴的なパターンを抽出できていることを確認することができ，また，任意の操作形態がどの特徴的パターンに含まれるかを予測できる可能性があることも検証でき，良く使う機能に関するコンテキストに応じた操作を行うような人には操作性向上が期待できることがわかったものの，これだけでは操作性向上は充分ではないこともわかった．

そこで，利用頻度が低く，起動方法や利用方法を忘れた機能に関しても過去の操作履歴に基づき，機能推薦を行うことにより操作性の向上を図り，総合的な操作性向上する方式を提案する．

7.1.1 実験環境と操作履歴

実験環境として小型の携帯型パソコンを使用して，被験者１名に関して約２ヶ月間の操作履歴や行動履歴を取得した．抽出した情報の例を図 7.1にしめす．得られた履歴から，表 7.1に示す情報を取得し，表 7.2に示した操作の種類に関して，アプリケーションの操作時刻および操作内容を抽出した．パソコンの操作履歴と位置データの二ヶ月分の履歴の全てを図 7.2に三次元的に日付毎にデータを重ねて示す．

表 7.2: 履歴取得する操作の種類

種類名	説明
Caption	アクティブウィンドウのタイトルを表示
Type	キーボードから入力されたテキストを表示
Explore	エクスプローラで表示したディレクトリを表示
ClipBoard	クリップボードにコピーされたテキストを表示

```
14:59:08 Caption $>$ 無題 - メモ帳
14:59:15 Type $>$tesuto[SPACE].[Enter]
14:59:22 Caption $>$ マイ コンピュータ
14:59:25 Explorer $>$file:///C:/borland
14:59:28 ClipBoard$>$test
```

図 7.1: 操作履歴の例

7.1.2 データの分類

分類に際し，今回収集した操作履歴の中での代表的な 3 種類のアプリケーション (Web メール， 航空会社の Web サイト， 乗り換え案内の Web サイト) の全データを訓練データとして決定木の抽出を行い，得られた決定木に基づいて 3 つのアプリケーションの操作形態を 3 次元空間上にマップした．それぞれ図 7.3， 図 7.4， 図 7.5に示す．

それぞれの直方体は分類モデルから得られた操作内容の占める空間を表している．つまり図 7.4の直方体内においては Web メールや乗り換え案内サイトよりも航空会社サイトにアクセスしている可能性が高いことを示している．

7.1.3 予測処理

次に任意の操作履歴に対して，それが抽出された決定木のどれと合致するか，つまりは任意の操作がどの操作パターンに属するかを予測できるかどうかについての検証を行った．検証法としては，得られた操作履歴・位置情報履歴に対して一般的な 10-fold cross-validation を適用する方法を採用した．

結果を表 7.3に示す． Web メールと航空会社 Web サイトに関しては， F 値が高いことから決定木による予測が有効であることが分かる．乗り換え案内 Web サイトに関しては，

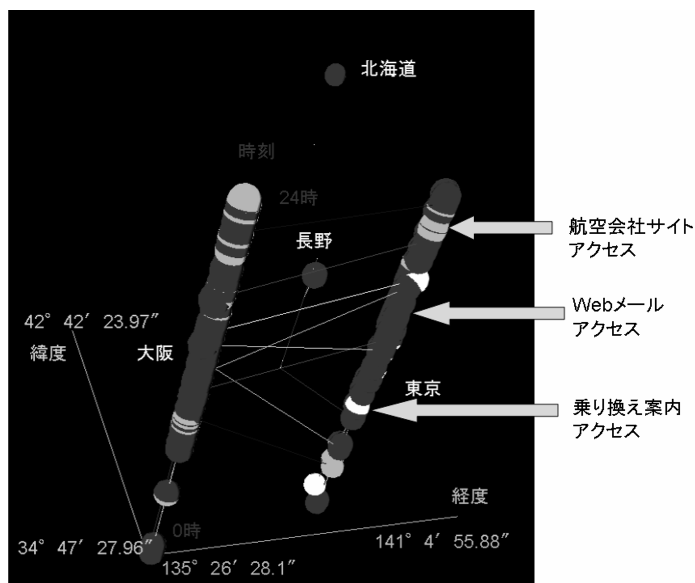

図 7.2: 操作ログの三次元マップ

表 7.3: 分類モデルの評価

属性	精度	再現率	F 値
web メール	0.927	0.96	0.943
航空会社	0.821	0.726	0.771
乗り換え案内	0.667	0.526	0.588

Web メールと航空会社 Web サイトに対して収集されたデータの絶対数が少ないため，この２つの領域に含まれてしまう傾向にある．しかし，乗り換え案内 Web サイトは Web メールと航空会社 Web サイトの領域に含まれている数より，乗り換え案内 Web サイトの領域に含まれている数の方が多く，2/3 の割合が乗り換え案内の領域に含まれている．そのため，これに関しても予測が有効であると考えられる．以上，10-fold cross-validation により構築した分類モデルの有効性について確認することが出来た．

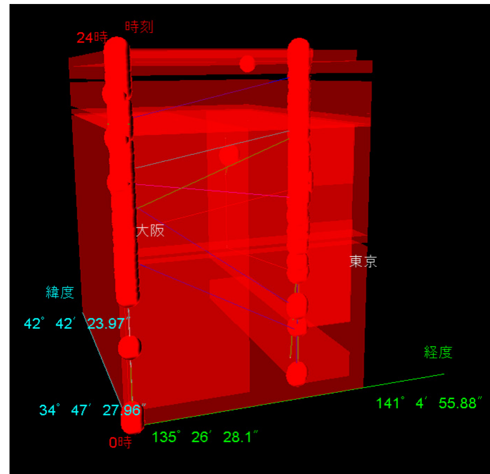

図 7.3: Web メールの予測モデル

7.2 評価

7.2.1 予測に関する評価

図 7.3，図 7.4，図 7.5から， Web メールの占める割合は残りの 2 つに比べて大きいことが分かる．これは， Web メールの使用頻度が大きく，場所と時間に関係なく使われていることを示すものである．

図 7.4からは，航空会社 Web サイトは位置ではなく，時刻に依存して利用される傾向にあることが分かる．特に 22 時以降の利用にこの傾向が見られる．これは，被験者が搭乗便の Web でのチェックインのために航空会社の Web サイトにアクセスしているためである．その航空会社の Web チェックインは出発前日の 22 時以降でなければできないため，前日の 22 時以降に必ず Web チェックインを行う必要がある．その傾向が図 7.4に表れている．一方，午前 10 時頃にも航空会社の領域が存在するが，これは出発便に関する情報を確認するためにと考えられる．

また，図 7.5より，乗り換え案内 Web サイトへのアクセスに関しては，時刻より位置に依存して利用される傾向にあることが分かる．大阪地区では乗り換え案内 Web サイトにアクセスしているケースは少ないものの，東京地区では時刻に関係なく利用されている．これは，大阪地区では電車などの公的移動手段を利用する頻度が低く，その反面東京地区では電車などを使用しての移動が多いことを示す．

これらより， 2ヶ月間での行動履歴から， 22 時以降であれば航空会社 Web サイトをメニューの上位に配置し，場所が東京地区であれば，乗り換え案内 Web サイトをメニューの上位に配置させることで，ユーザは容易に航空会社や乗り換え案内の Web サイトにアクセ

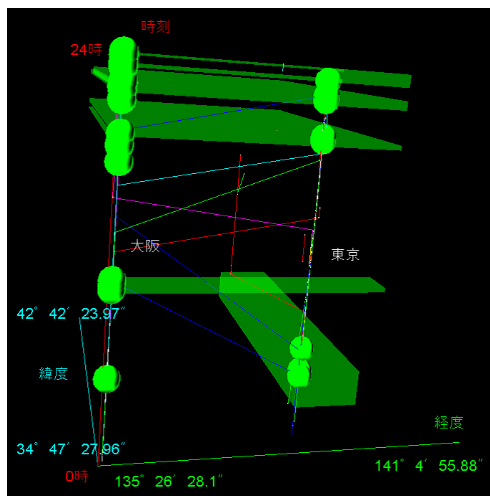

図 7.4: 航空会社の予測モデル

ス可能なことがわかる．

更に，抽出された決定木から，22 時以降に航空会社 Web サイトにアクセスがあった翌日には，遠方に移動する傾向が強いことも分かった．その例を前日分と合わせて図 7.6に示す．左から 2/8，3/20，4/4 の行動履歴である．時刻が負の部分は前日のデータである．

「ある操作をした後に，決まってある行動をとる」という情報もとても重要であると考える．なぜなら，その操作によって次の行動を予測することができ，ユーザに有用な情報を発信できるからである．航空会社 Web サイトへのアクセスの場合であれば，翌日に携帯電話が利用される場合，フライトの確認を見る Web ページに予めアクセスしておくことが可能となる．また，アプリケーション間の利用のされ方においてもこのような因果関係を抽出することができればさらに使いやすい UI を提供できる．例えば，朝，天気予報を見たあとに，ウェブにてテニスコートの予約を行う人がいる場合，天気予報から一回のクリックでテニスコートの予約画面にアクセスできるような UI を提供することで，ユーザは天気予報を閉じて，テニスコートの予約画面を探すという手間をかけなくて済む．

7.2.2 操作性に関する評価

操作性評価指標

操作性の指標は，操作の回数と操作の時間から決まるものと定義する．操作の回数は，キー押下の回数のことであり，少ないほど使い易いことを示す．しかし，実際には，何も考えずに同じキーを 3 回押す方が，1 回ではあるが，画面を見てどのキーを押すべきかを考えた上で押下するよりも使いやすく感じることがある．即ち，操作の回数だけではなく，

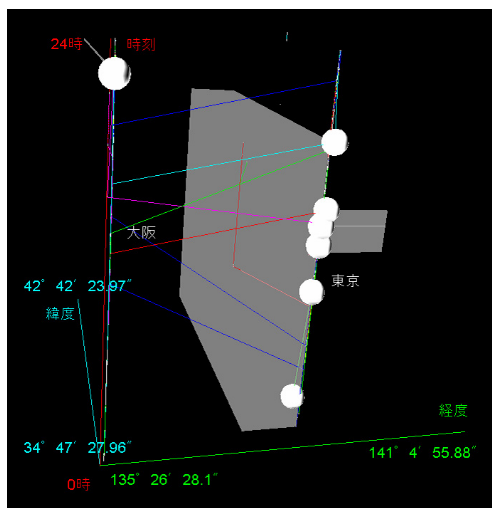

図 7.5: 乗り換え案内の予測モデル

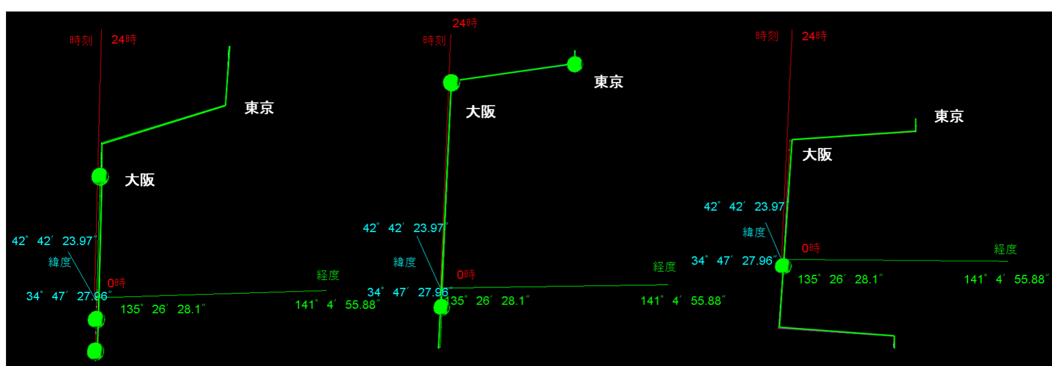

図 7.6: 航空会社を起動したときの行動パターン

操作の時間も操作性の指標に加えるべきである．

そこで，機能の数を n，機能 i に対する操作性を V_i とする． $C(i)$ を機能 i の機能を呼び出すまでの操作回数， $T(i)$ を機能 i を呼び出すまでの平均時間とすると，機能 i に関する操作性は次式で定義する．

$$V_i = C(i)T(i) \tag{7.1}$$

一般に，携帯電話には多くの機能が搭載されており，しかも機能ごとに使われる頻度は違う．機能 i を利用する頻度を $H(i)$ とすると全体としての操作性 V は次式で表される．

$$V = \sum_{i=1}^{n} C(i)T(i)H(i) \tag{7.2}$$

ここで，携帯電話の画面で表示するメニューの数を m 個とすると，階層をたぐる深さは $\log_m n$ 段になる．均等に階層化しているとし，各階層では 1 回の操作でダイレクトにメニューを

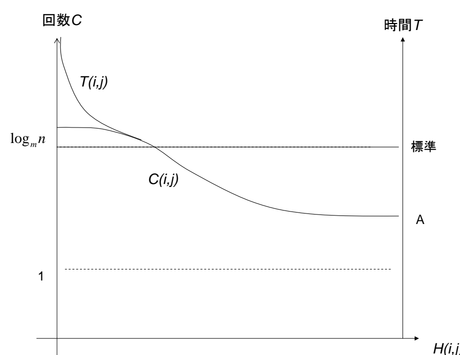

図 7.7: 頻度と操作回数，操作時間の関係

特定できるとすると，と操作数は次式で示される．

$$C(i) = \log_m n \tag{7.3}$$

同様に $T(i)$ も表示されたメニューを順に見て選ぶこととし，一つのメニューを見る時間とキー操作に伴う時間の合計を t とすると次式になる．要素が m 個あり，平均して半分の要素は見ると考えるとメニューを見て判断するために $\frac{m}{2}$ の係数がかかるため次式となる．

$$T(i) = \frac{tm \log_m n}{2} \tag{7.4}$$

以上はすべての機能の操作方法を知っている理想的なユーザが操作した場合である．

実際には頻度の高い機能を選ぶ場合には，メニューをじっくり見て判断してないであろうし，頻度の高い機能を他の機能と同様のメニュー階層に置くことはなく，ショートカットメニューなどの利用により，機能の頻度に依存してこの数値は変わると考えるべきである．ユーザによって機能の使い方は異なるし，状況に応じて使い方が異なることが [?] で報告されており，コンテキスト情報を利用することによりこの評価指標値の改善が期待できる．

そこで，コンテキスト j の元での機能 i の利用を想定した場合の評価指標 V を次式で示す．

$$V = \sum_{i=1}^{n} \int C(i,j)T(i,j)H(i,j)dj \tag{7.5}$$

$C(i,j)$ は誤った操作という部分を除くと，頻度が低くても一定の操作回数に抑えることができる．しかし，探すための時間は慣れなどに応じるため，時間に関しては頻度が低いと一定時間に抑えられない．そのため， $H(i,j)$ を横軸にとって， $C(i,j)$ および $T(i,j)$ を縦軸にとった想定される関係は図 7.7のようになると考えられる．現実には使う頻度の低い機能は誤った操作も繰り返すので，結局 $C(i,j)$ と $T(i,j)$ は共に大きくなると想定される．

操作性向上に関する評価

今回の実験結果では，操作頻度の高いものとして 3 種類のアプリケーションが見つかり，これらに関してコンテキストに応じて予測すると正しく予測できる確率は非常に高いことを示すことができた．しかし，頻繁に利用するアプリケーションは高々 3 種類であった．携帯電話を使いこなすユーザにとってはこれらをユーザカスタマイズ可能なショートカットメニューの登録や BookMark 機能を使うことにより，ワンタッチまたは 2 タッチでアクセスできる．そのため，コンテキストに依存した推薦機能を利用したからと言って操作性が向上したとは実感しないと考えられる．

操作性の評価指標としては，操作時間と操作回数であるから，これらの値が最初の登録時を除くと全く変わらないためである．では，携帯電話でコンテキストに依存して良く使う機能はどの程度あるであろうか．携帯電話ではたかだか 1 桁程度と推測できる．有効に機能したとしても，操作回数が 1 回減る程度で，ドラスティックには操作性の向上は実感できないであろう．

次に決定木を利用した操作予測に関しては， 1 種類を予測するだけでは予測がはずれた時の代償が大きい．そこで操作ログをクラスタリングした上で可能性の高い候補を複数抽出し，その中からユーザに選択させる方が予測がはずれる確率が低くなり有効と考える．そうすると， 1 段階考えるフェーズと操作のフェーズが増えるため実質上は良く使う機能に関しては操作性向上を望めなくなる．この観点から本方式のみでは操作性向上できたとは言えない．

一方，使用頻度が低い機能に関しては，”位置”や”時間”といったコンテキストでの推測はデータが少ないことが想定されるため困難である．頻度の低いものは，操作もうろ覚えが想定され，操作時間と操作回数の双方が大きくなってしまう．そのため，結局トータルの操作性評価指標値は利用頻度の高い機能の推薦程度ではあまり変わらないと推測されるため，やはり本方式のみでは操作性の向上に対しては不十分である．

7.3 操作性向上のための改良方式提案

位置や時間といったコンテキスト情報を利用した操作性向上方式では不十分であるため，その欠点を補った操作複雑性情報を利用した操作性評価指標値 V を下げる方式，とくに使用頻度の低い機能でも一定の操作回数と操作時間に抑える操作機能推薦方式を提案する．

(1) マイメニューボタンで，頻度の高い機能の推薦機能候補を表示する．基本的にはツータッチでよく使うと想定される複数の機能が選択可能となる．学習をした結果に基

づくものの，安定状態になれば，少し違う機能を選択したことがあっても頻繁にメニューの順序は変わらないと想定する．

(2) クリアキーなどのボタン一つで通常の画面に遷移できるようにする．これは頻度の低い機能を使う場合に，頻度の高い機能としては推薦されていないケースが想定できるためである．

(3) 通常画面ではトップメニューなどに”困った時”のメニューを置き，ここをクリックすると，履歴探索を行い，その階層下で誤った操作も含んで期待される機能の候補を提示する．頻度の低い機能は，履歴探索機能から推薦されることになるが，頻度が低いため，コンテキスト情報との関連が明確ではないと考えられるが，例えば圏外などというコンテキスト情報を使うと，今使えない機能はグレーアウトなどにより表示することができる．

図7.8に提案方式の実装の例を示す．この例では，左上画面例が，通常の待ちうけ画面上でマイメニューボタンを押下した後のマイメニュー画面である．推薦機能が表示されている．右上画面例は通常メニューである．右下画面例の”困った時”キーを押下すると，その下にあるような頻度は少ないがどこにあるかわからない推薦機能が表示される．左下画面例は，一階層メニューを入った後に”困った時”キーを押下した例である．いずれも，そのメニュー階層の下から選択するのではなく，そのメニュー階層からたどったことのある機能を履歴から抽出し，複雑度，前操作に応じて推薦機能を表示する．

次に提案方式のソフトウエア構成を図7.9に示す。操作履歴はアプリケーションの起動履歴をコンテキスト情報とともに取得することと、キーの操作履歴も合わせて取得する。

操作複雑性は，機能iを起動するまでの操作数$C(i)$と$T(i)$で決まるとする．PCなどでネットワークのコンフィギュレーションなど年に1回の異動の時にしか利用しない機能はいつも同じ手順で間違うことなどを経験することが多い．そこで，操作数$C(i)$は，誤った操作などを含めた操作数として過去の操作履歴より取得する．操作時間も同様に，メニューボタンを押下してから，機能を起動するまでの時間である．

履歴上の$C(i)$および$T(i)$とメニュー内を探した足跡に基づき推薦すべき機能を抽出する．これらの推薦機能は瞬時に計算され表示される必要が有る．携帯電話でこのような機能に関係するコンテキスト情報は時間であるとか電波強度も圏外か圏外かといった情報程度であると考え，予めコンテキスト，メニュー階層ごとに候補の絞りこみを行う方式とし，瞬時性を保つべきである．

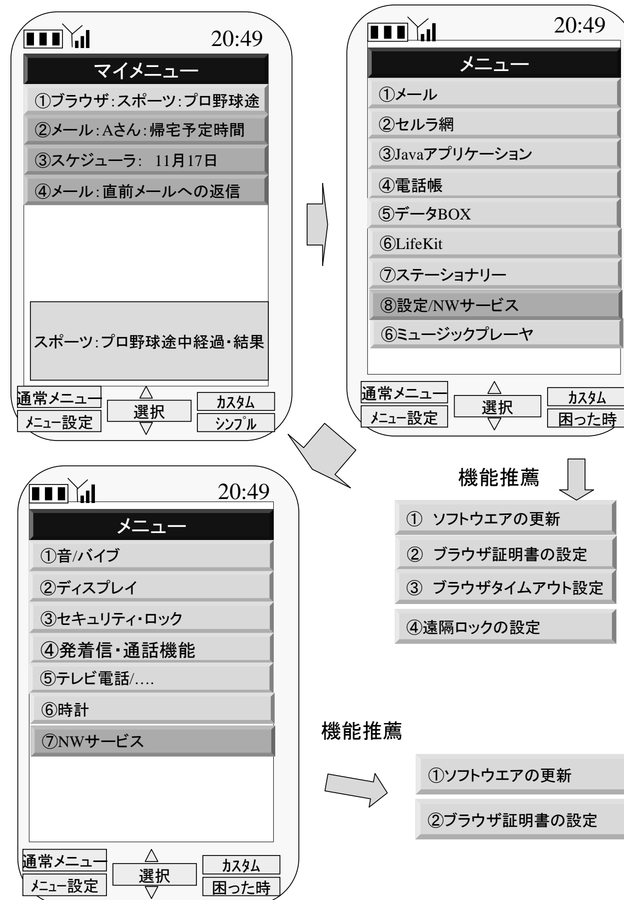

図 7.8: 困った時の機能推薦

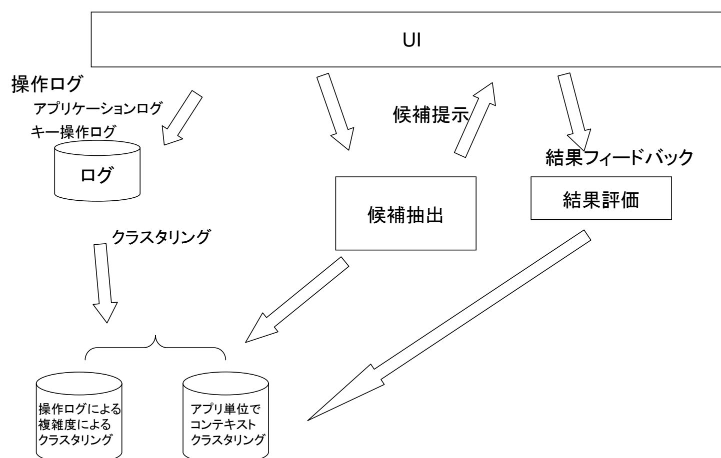

図 7.9: ソフトウエア構成図

7.4 改良方式の評価と考察

$H(i,j)$ が比較的小さい場合に関して，$C(i,j)$ および $T(i,j)$ の大きさで評価する．まず，利用したことのない機能に関しては，本提案方式でも解決していない．直感ではわかりにくい機能などを探すとこの $C(i,j)$,$T(i,j)$ は従来の方式と全く同じである．しかし，1 度でも利用した機能に関しては，探す手間が省けるため飛躍的に小さくなる．

最悪ケースとして，すべての機能を一度だけ利用した場合を想定する．機能推薦で現れるのはたかだか 1 画面程度であるため，所望の機能が機能推薦では出てこない可能性がある．各階層ごとに機能推薦をしてみては，戻るを繰り返すと確実に $C(i,j)$ は $2\log_m n$ 回大きくなると想定される．しかしながら，推薦されない機能に関してはより直感で見つけ易い機能であるため，以前と同じ思考パターンであればメニューの階層での絞り込みで発見される割合が上がると推定できる．

また，過去に一度しか使ってなくても確実にわかってる機能に関しては推薦機能を利用しないと想定されるため，$C(i,j)$ は従来方式と変わらない．一般的にはすべての機能を一度のみ利用しているケースはないと考える．一方 $T(i,j)$ は従来方式と同じ比率で効果があがるわけではない．階層の下にメニューが一定数としても見た上で判断するための時間は通常のメニューを選ぶより少し長いと想定されるからである．しかし，誤操作をして探す手間に比べるとはるかに小さい差である．

さらに，実際に同じような操作を人がしなかった場合は，同じ操作パターンの中で抽出すると絞りこみができる代わりに同じ操作をしなかった場合に目的の機能が見つからないという問題がある．本提案のように階層ごとにその下を誤った操作も含めて見る方式であれば，違う階層を見ている場合を除いて，その階層にたどりつくまでのパターンに限らず目的の機能が推薦されてくる．

しかしながら絞り込みされないため，推薦候補が多いという可能性があるが，携帯電話のように限られた機能を使う場合には候補が多いといってもたかだか片手程度と考えられるため本方式は有効である．

7.5 本章のまとめ

本章では携帯電話を例にとり，使い易さの向上を目指す一つの手法を提案した．使い易くなったことを確認するための評価指標として，操作回数と操作時間の積分を行った評価指標を定義した．この評価指標値の振る舞いは，各機能ごとに良く利用するのか，たまに利用するのかで振る舞いが違うことを示し，とくにたまに利用する場合の操作性の改善方法に関して示した．具体的には過去の操作履歴から操作回数と操作時間で決まる操作複雑

性の高いものから推薦機能として提示することにより評価指標値を下げられることがわかった．今後の課題を以下に示す．

- 過去に経験したことのない機能の選択などは従来の方式から改善されておらず他の端末との情報共有やサーバを利用した方式などの検討
- 頻度分布に偏りがあるような場合の振る舞いの解析と改善すべき場合にはその方式検討
- 人間にも学習効果があり，実際にいつも同じ誤った操作をするとは限らない．そこで，人間の学習効果も考慮したインタフェースが必要
- 携帯電話への実装による評価

第 8 章

結論

近年，携帯電話は高機能化の一途を辿ってきた．ユーザの個々の要望に答えるため，あるいは開発者側の都合から携帯電話のソフトウェアアドイン機能の必要性がでてきた．パソコン上で実現されているソフトウェアのアドイン機能は携帯電話という小さなコンシューマデバイス上では各種制限により簡単には実現できない．本論文では，これらのいくつかの点に焦点をあて，その解決策と有効性を示した．以下，ソフトウェアアドインを構成する技術を整理して各技術ごとに結論としてまとめる．

本論文で携帯電話のソフトウェアアドイン機能として検討したのは以下の点である．

(1) ソフトウェアの不具合の修正や OS の機能拡張などで，ソフトウェア保守を目的としたソフトウェア書き換え．主にメーカやキャリアが必要に応じて実施するサービスである．

(2) ユーザが自らの意思でソフトウェアをダウンロードし，インストールすることによりソフトウェアを追加，変更すること．例えば Java アプリケーションであったり，BREW アプリケーションという形でキャリアからサービスされている機能のことである．

(3) ユーザが自らの意思で自動的または手動で携帯電話を使いやすいようにカスタマイズすることでソフトウェアを出荷時の状態から変更することである．例えばショートカットキーをユーザが変更する機能のことである．

また，上記 3 種類のソフトウェアアドイン機能ごとには，以下に示す課題があった．

(1) 携帯電話のソフトウェアは，瞬時の起動の必要性などから XIP というフラッシュメモリ上のコードを直接実行するモデルでソフトウェアが実装されていることが多い．フラッシュメモリは最低限の容量しか持たないため，パソコンのようにプログラム実行中にプログラムイメージを書き換えることはできない．そのため，ソフトウェ

アの書き換えはできる限り短時間で終えなければならないという課題がある．また，ネットワークを使うという観点からは以下にデータ転送量を減らすかという課題もある．

(2) 自由にソフトウェアをダウンロード実行するモデルでもインタプリタ実行を行うようなケースではメモリと CPU の能力の観点で実行速度そのものが問題になるケースが多く，高速化が重要な課題となる．

(3) ソフトウェアのユーザごとにカスタマイズ機能においては，移動しながら片手で利用し，画面も小さい携帯電話ではパソコンと同様な手法ではユーザは不満を覚える．そのためユーザに意識させずに操作性を向上する方式が求められる．

それぞれに対して以下のような手法を提案し，その有効性を示した．

(1) 携帯電話のソフトウェア更新方式の提案
第 3 章，第 4 章，第 5 章にて，携帯電話のソフトウェア更新方式に関して述べた．携帯電話のソフトウェア更新の課題は，ネットワークを利用してデータを送信する際のデータ量の削減と携帯電話上でのデータの書き換え時間の削減の 2 点である．それぞれの課題における提案の特長と成果を以下に示す．

(a) 携帯電話のソフトウェア更新のための送信データの削減方式に関する提案
第 3 章にて，送信データ量を小さくするためのバイナリ差分抽出技術に関して述べた．最近の携帯電話のソフトウェアの容量は全体で小さいものでも 32M バイト程度であり，大きなものでは 64M バイト程度ある．また，ソフトウェアの更新は全端末に対して短い期間でデータを配信する必要性から網への影響が大きいため，できる限り送信量を削減する必要がある．

携帯電話上で既に稼動しているソフトウェアイメージ（以下，旧版ソフトウェアイメージと呼ぶ）と工場などで動作確認を実施して，これから携帯電話のソフトウェアを置き換えようとする新しい実行イメージ（新版ソフトウェアイメージ）のバイナリ差分を抽出してその差分情報を送ることが考えられる．この差分を以下に小さく表現するかが課題である．

そこで，携帯電話のプログラムコードという特性として，新版と旧版の差の多くがアドレスのリファレンス関係にあるという特性を利用して，バイナリ差分が小さく表現できる方式を提案，評価し効果があることを示した．実験によって 4 バイト命令の CPU であれば，命令コードの 4 バイト目のアドレス部分に差異が多くなることを確認した．バイナリ差分の大きさは，本質的に，差分を

適用するためのコマンド数が少なくなれば小さくなる．そのため，差異のある部分をなるべく集中させることにより差分を小さくできると考え，物理空間を論理的な 4 バイトごとの空間に変換する手法を提案し，これをモジュール単位に実装することにより効果があることを確認した．

さらにコマンドのアドレス指定部で使うバイト長を減らすことにより差分を小さくできることに着目し，命令コードのずれをマクロ的に見ることにより，細かくバイト位置を指定しなくてもすむマクロコピー方式を提案し，実際に実装することにより，差分が約 30 %削減できることを確認した．また，この 2 つの手法を組み合わせることにより，さらに差分が小さくなることを確認するとともに，各種 CPU アーキテクチャのプログラムコードに適用し，評価した結果効果があることがわかった．

これらの手法は携帯電話ソフトウェア，エレベータ制御ソフトウェア，金融端末ソフトウェアなどに実装済みであるが，今後は固定長の命令コードではない CPU 用のプログラムコードに対しても有効な差分抽出技術を開発していく予定である．

(b) 携帯電話のソフトウェア更新のためのデータの高速書き換え手法に関する提案

携帯電話のソフトウェアの新版と旧版の差が小さくとも，フラッシュメモリ上の位置が変わるとフラッシュメモリはすべて消去して新しく書き換えなければならない．これには最近の携帯電話のように 64M バイトものメモリを持っていると数十分かかるケースも珍しくない．ユーザの視点にたつと携帯電話のサービスが受けられない時間が数十分というのは問題である．とくにユーザが意識しないうちにソフトウェアの更新をしようとすると，ユーザにとっては理解できずにソフトウェアの書き換え中に電池を抜くなどの問題が発生するケースもあると考える．

そこで，第 4 章にて，この書き換え時間を小さくする方法として，まず開発環境上での工夫により，局所の修正は局所にのみ影響するという手法を提案し，評価の結果効果があることを示した．また，第 5 章にて，最近の NAND 型のフラッシュメモリを利用したソフトウェアでは所謂仮想記憶を活用するデマンドページング方式の採用も考えられ，ソフトウェアの高速展開可能な圧縮方式を利用するケースも想定されるため，この観点にも着目した方式を提案，評価し効果があることを示した．

具体的には，局所の修正は局所にのみ影響が出るようにするために，モジュール間の参照は，ジャンプベクターテーブルを経由するようにした．このトレー

ドオフで実行速度が遅くなることと，メモリが余分に必要になる．そのため，それぞれのオーバーヘッドが 3% 以内になるような適切なモジュール分割手法を提案し，評価を行いその有効性を示した．

また，圧縮方式においては，開発環境上で圧縮用の辞書を作成し，その辞書を端末へ転送し，端末上で圧縮することにより圧縮速度も高速にできることを示した．そのトレードオフとしてデータ転送量が大きくなるため，効果の高い部分のみデータ転送する手法を示し，その分岐点の求め方も示すことによりその有効性を確認した．今後はデータ転送量をさらに削減するために辞書データのフォーマットを圧縮用と展開用で兼ねる手法を検討し，データ転送量をさらに削減する方向で検討する予定である．

(2) 携帯電話のアドインソフトウェアである Java のプログラム実行高速化方式の提案

Java は一般にプログラムをコンパイルし Java バイトコードに置き換える．これを携帯電話上にダウンロードし，インタプリタで逐次実行する．そのため実行速度が遅いという欠点がある．しかし，携帯端末上の各種リソースにアクセスすることもインタプリタを経由するという点で守られ，セキュアであるという点で優れているため，誰でもがプログラムを開発し携帯電話上で動作させせることができるという利点がある．

一方， BREW の方式は C 言語で記述する通常のアプリケーションと同じ位置づけになるためセキュアにはならないが，他のアプリケーションと同様な高速な実行が可能である．セキュアを保証するためにキャリアなどがコードレビューをするといった認証を得たもののみがプログラムを書くことが許されるというスタイルになり，誰もが自由にプログラムを書けるわけではない．

そこで，第 6 章にて，セキュアでかつ高速に実行できる環境として， Java を高速化する手法を検討した．一般に Java を高速に実行する手法としては JIT[1]方式が提案され， PC 上のブラウザなどで実装されている．しかし， JIT では最適にコンパイルすればするほどコードの占めるメモリが大きくなるという欠点がある．携帯電話のようにメモリの余裕がないケースにはそのまま適用することができない．

また， JIT 方式ではコンパイルを実行時に行うため，コンパイルしている時間は動作できないという問題がある．起動時にすべてコンパイルすると起動に時間がかかるし，実行時に逐次コンパイルしていると携帯電話のようにインタラクティブ性の高いアプリケーションでは瞬停が多くなり操作性が悪くなる．

[1]Just In Time compile

そこで，メモリを節約しつつ，必要な部分はコンパイルし，かつ瞬停をさせない手法を提案評価し，効果があることを示した．具体的には，携帯電話で良く使われ，しかも応答性能が要求されるシューティングゲームに関して静的に解析した．その結果，良く使われるライブラリのメソッドなどをあげ，そのメソッドを読んでる関数をコンパイル対象とした．また，静的には良く使われていても，動的に使われなければ効果が薄いため，実行時に動的にプロファイルを取得することとした．このプロファイルもメモリに影響が出るため，良く使われるメソッドに関してのみ取得することとした．

次に，動的解析した結果で，コンパイル時間に対して実行時間の長いもの，即ち，メソッドサイズが小さくもなければ大きくもなく，ループを含んでいる処理を対象にコンパイルをすることとした．

このような工夫の結果，瞬停することもなく，実行速度の高速化を実現することができた．複数のアプリケーションでの評価の結果その有効性を示せた．今後はJavaの実行環境は速度よりむしろ利用できる機能の自由度をいかに安全にあげていくかが課題であり，ネイティブのコードとの合わせた形での実行環境の検討を進めていく予定である．また，複数のJavaのプログラムを同時に実行するといったマルチ実行環境も必要となると予測され，メモリの効率的な利用方法に関して検討を進めていく必要がある．

(3) 携帯電話のユーザ操作のユーザビリティ向上方式の提案

出荷後のソフトウェアを変更するという意味で最も多くの機種で可能なのが，ショートカットキーの割り当てである．多くの人が共通で頻繁にアクセスすると想定されるメールやブラウザの起動はキーが既にアサインされ1回の操作で見れることが多い．これに対して，個人ごとによって違う機能をアサインすることができるのがショートカットキーであり，多くの場合は2タッチで所望の機能が起動できるようになる．しかし，ショートカットキーも増加すると2タッチというわけにはいかない．

そこで最近の携帯電話のようにGPS機能で位置が取得でき，加速度センサでユーザの状態が推定できるような場合に，ユーザのコンテキストに応じてユーザの所望する機能をユーザの過去の履歴から推測し，候補を提示する手法を第7章で提案しその有効性を示した．具体的には，ユーザのコンテキストから次の動作がある程度予測可能であることを検証できた．しかし，コンテキストに依存しない動作が多かったり，サンプル数が少ないと予測不可能である．また，少々の操作数を減らしても，その有効性が疑問とされることも多い．

そこで，あまり利用されないケースの使い方に焦点を絞り，誤った操作の履歴も含めて活用することにより，同じ過ちを繰り返した場合には，正しい操作を予測するという方式を提案した．今後は，携帯電話で消費電力を考えた上で取得可能なコンテキストを検討し，その上で実装可能な操作支援に関して検討していく予定である．

謝辞

本研究をまとめに際し，ご指導，ご鞭撻を賜った大阪大学産業科学研究所・沼尾正行教授に，心から感謝いたします．また本論文の執筆に当たって懇切なご指導を頂いた大阪大学産業科学研究所・栗原聡准教授に，深く感謝の意を表します．また，公私に渡ってご指導いただいた大阪大学情報科学研究科教授石井博昭氏，同森田浩氏，同谷田純氏に心から感謝します．

本研究を推進するにあたり，ご指導，ご鞭撻いただいた三菱電機 (株）の肥塚裕至氏，西井龍五氏，勝山光太郎氏，菅隆志氏，木野茂徳氏，中川路哲男，新堂隆夫氏，小島泰三氏，橘高大造氏，松本利夫氏，清水直樹氏，斎藤正史氏，無中達司氏，に心から御礼を申し上げます．

また，本研究をまとめるにあたり，有益な討論，協力をいただいた岡田英明氏，前田慎司氏，三井聡氏，深沢司氏，山田佳邦氏，荒井兼秀氏，小阪一樹氏，竹中正憲氏，高橋清氏，小谷亮氏，高橋克英氏，中邦博氏，上江洲均氏， 神戸英利氏，則皮基宏氏，河相英典氏，小野良恭氏，古宮章裕氏，砂川陽一氏，榎秀幸氏に心から感謝いたします． また，研究室にて有意義なコメント等をいただいた大阪大学産業科学研究所沼尾研究室の松本光弘氏他諸氏に感謝いたします．

また，大学時代にご指導いただいた元大阪大学助教授・安井裕氏，斎藤年史氏，大中幸三郎氏， ICOT 出向時代にご指導いただいた故内田俊一氏，近山隆氏に心から感謝します．

三菱電機にてご指導いただいた溝口徹夫氏，太細孝氏，中島克人氏，黒田正博氏，下間芳樹氏，鈴木克志氏，に心から感謝します．

最後に，研究を推進するにあたり，暖かいご支援と多大なる御理解をいただいた妻由美，息子良文，娘知美に感謝します．

参考文献

[1] 杉山泰一: 携帯電話のバグを無線ネット経由で修復, 日経コミュニケーション, 2003 年 8 月 pp.70-72,2003.

[2] 日経エレクトロニクス: 携帯電話のバグをユーザの手元で修正へ, 2002 年 3 月,pp. 22-23,2002.

[3] Takeichi M.,Hosokawa A.,Nasu K.,Hoshi S. and Moriyama K.: Bug fix of mobile terminal software using download OTA, The Asian-Pacific Network Operations and Management Symposium,2003.

[4] Hoshi S., Ichinose A., Nose Y., Hosokawa A., Takeichi M., Yano E.: Software update system using wireless communication, NTT DoCoMo Technical Journal, Vol.5, No.4, pp.36-43,2004.

[5] 原田雅章: 組込みソフトウェアのバージョンアップ機能を持った μ ITRON 仕様 OS, 平成 14 年度 IPA 重点領域情報技術開発事業成果報告,(on line), available from http://www.ipa.go.jp/SPC/report/02fy-pro/report/1237/paper.pdf (accessed 2008-04-28),2002.

[6] トロン協会: (on line), avairable from http://www.assoc.tron.org, (accessed 2008-04-28)

[7] 日経ビジネス:ケータイのバグを無線で修正ソフトウェア更新の仕組みは?, 日経BP社(オンライン), available from http://itpro.nikkeibp.co.jp/free/TIS/keitai/20041201/153313/?P=1&ST=keitai (accessed 2008-04-28),2004.

[8] Hunt J. and Mcillroy M.: An algorithm for differential file comparison, Tech.Rep.41, AT & T Bell Laboratories,Inc. ,Murray Hill,NJ,1976.

[9] Hunt J. and Szymanski T.: A Fast Algorithm for Computing Longuest Common Subsequences, Commun. ACM,vol.20,no.5,pp.351-353,1977.

[10] Tichy W.:The string-to-string correction problem with block moves, ACM Trans.Computer systems, vol2,no.4,pp.309-321,1984.

[11] Hunt J., Vo K., and Tichy W.: Delta algorithms: an empirical analysis, ACM Trans.Software Engineering and Methodlogy,vol.7,no.2,pp.192-214,1998.

[12] Tridgell A. and Mackerras P.: The rsync algorithm, Australian National University,TR-CS-96-05,1996.

[13] W3C:Generic Diff Format Specification,(on line) available from http://www.w3.org/TR/NOTE-gdiff-19970901,1997. (accessed 2008-06-5)

[14] World Wide Web Consortium: Web Standards,(on line) available from http://www.w3.org/ (accessed 2008-06-5)

[15] Korn D. and Vo K.: Engineering a diffrencing and compression data format, Proceedings of the 2002 USENIX Annual Technical Conference,2002.

[16] Burns R., Stockmeyer L. and Long D.: In-place reconstruction of version differences, IEEE Trans.Knowledge and data engineering,vol.15,no.4,2003.

[17] Terazono K. and Okada Y.: An extended delta compression algorithm and the recovery of failed updating in embedded systems, IEEE Data Compression Conference(DCC 2004),p.570,2004.

[18] 寺薗浩平他: RISC アーキテクチャに適した差分圧縮アルゴリズム, 情報処理学会 FIT2002,A-27 pp.53–54,2002.

[19] Brenda S. Baker, Udi Manber, and Robert Muth: Compressing Difference of Executable Code, ACM Proceedings of SIGPLAN Workshop on Compiler Support for System Software WCSS'99, 1999.

[20] Symbian: Demand Paging on Symbian OS, (online), available from http://www.symbian.
com/symbianos/demandpaging/index.html
(accessed 2008-04-28),2007.

[21] 竹田正幸，篠原歩: 圧縮されたテキスト上のパターン照合 - データ圧縮とパターン照合の新展開 -，情報処理， Vol.43， No.7pp.763-769,2002.

[22] Lelewer A. D. and Hirschberg S. D.: Data Compression, ACM Computing Surveys,Vol.19. No.3 pp.261-296,1987.

[23] Beszédes Á. , Ferenc R., Gyimóthy T., Dolenc A. and Karsisto K.: Survey of code-size reduction methods, ACM Computing Surveys,Vol.35, No.3 pp.223-267,2003.

[24] Gage P.: A new algorothm for data compression, The C users Journal,Vol.12, No.2,1994.

[25] 門前 淳, 安浦 寛人: Byte Pair 符号化を用いた命令 ROM 圧縮，情報処理学会研究報告， No.2001-SLDM-101, pp.1-6,2001.

[26] AlgoTrim: Code Compression Library, (online), available from http://www.algotrim.com/index.php?page =code-compression (accessed 2008-03-21),2007.

[27] Bellaachia A. and Rassan A. I,: Efficiency of Prefix and Non-Prefix codes in String Matching over Compressed Databases on Handheld Devices, Proc. of the 2005 ACM symposium on Applied computing, pp.993-997,2005.

[28] Shibata Y., Kida T., Fukamachi S., Takeda M., Shinohara A.: Speeding up Pattern Matching by Text Compression, Lcture Notes in Computer Science, Algorithms and Complexity, Vol.1767,pp.306-315,2000.

[29] 大関江利子，山田和宏: 携帯機組み込み Java の実用化，NTT DoCoMo テクニカルジャーナル, Vol.8, no.1, pp.16-21,2001.

[30] 奥山玄, 才田好則, 臼井和敏: アプリケーション競合管理方式, 情報処理学会第 67 回全国大会論文集, PP.(1-35)-(1-36),2005.

[31] Sribaths Ravi and Anand Raghunathan: Security in Embedded Systems:Design Challenges, ACM trans. on Embedded Computing Systems, Vol.3, no.3, pp.461-491,2004.

[32] 太田賢、吉川貴、中川智尋、稲村浩: セキュア携帯機のためのアプリケーション自己監視方式, 情報処理学会, DICOMO2005, pp.725-728,2005.

[33] 朝倉義晴, 奥山玄, 中山義孝, 臼井和敏, 中本幸一: 携帯電話向けマルチアプリケーション実行環境, 情報処理学会研究報告,No.2003-OS-093, PP.135-142,2005.

[34] Gosling J., Joy B., Steele G.,Bracha G.: Java Language Specification, Second Edition, Addison Wesley ,2000.

[35] Lindholm, T. and Yellin F.: The Java Virtual Machine Specification 2nd Edition, Sun Microsystems Inc., 2000.

[36] Sun Microsytems Inc.:J2ME Building Blocks for Mobile Devices White Paper available form http://java.sun.com/products/cldc/wp/KVMwp.pdf (accessed on 2008-04-28)

[37] Kaffe: Kaffe—An Opensource Java Virtual Machine, available from http://www.kaffe.org. (accessed on 2008-04-28)

[38] Suganuma T., Ogasawara T.,Takeuchi M., Yasue T., Kawahito M., Ishizaki K., Komatsu H. and Nakatani T.: Overview of the IBM Java Just-in-Time compiler, IBM Systems J. Vol.39, 1, pp.175-193,2000.

[39] Alpern B., Attansio C. R., et al:The Jalapeño Virtual Machine. IBM Systems J. Vol39, 1, pp.211-221,2000.

[40] Sun Microsytems Inc.: The Java HotSpot Virtual Machine, White Paper, (on line), available from http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html (accessed on 2008-04-28)

[41] Sun Microsytems Inc.: The CLDC HotSpot(tm) Implementation Virtual Machine, White Paper, (on line), availble from http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf (accessed on 2008-04-28)

[42] 首藤一幸ほか:Java Just-in-Time コンパイラのためのコスト効率の良いコンパイル手法, 電子情報通信学会論文誌, Vol.J86-DI, No.4, pp.217-231,2003.

[43] 川本琢二ほか：家電向け JavaJIT コンパイラの構成法とその評価, 情報処理学会論文誌,Vol.43 No.SIG08 PRO15,pp.37-48,2002.

[44] N. Shaylor: A just-in-time compiler for memory-constrained lowpower devices, in Proc. Usenix Java Virtual Machine Research and Technology Symp. pp.119-126,2002.

[45] John Whaley: A Portable Sampling-Based Profiler for Java Virtual Machines. In ACM 2000 Java Grande Conference, pp.78-87,2000.

[46] 舩田雅史ほか: 携帯機器を対象とした Java 動的コンパイラにおけるプロファイリングシステム, 情報処理学会, 研究報告 2003-MBL-028, pp.55-62,2004.

[47] Blom J., Chipchase J. and Lehikoinen J.: Contextual and Cultural Challenges for User Mobility Research, Comm. ACM, Vol.48, No.7, pp.37-41,2005.

[48] Nakashima H.: Cyber assist project for situated human support, Proc. The Eighth International Conference on Distributed Multimedia Systems, ISBN 1-891706-11-X, Knowledge Systems Institute, pp. 3-7,2002.

[49] Widjaja I. and Balbo S.: SPHERES OF ROLE IN CONTEXT-AWARENESS, ACM Proc. of 19th Conference of the computer-human interaction SIG of Australia on Computer-human interaction, SESSION:Short papers,2005.

[50] Korpipää P., Häkkilä J., Kela J., Ronkainen S. and Känsälä I.: Utilising Context Ontology in Mobile Device Application Personalisation, ACM Proc. of MUM'04, pp.133-140,2004.

[51] Henricksen K. and Indulska J.: Developing context-aware pervasive computing applications: Models and approach, pervasive and mobile computing, Vol.2, No.1, pp.37-64,2006.

[52] 上坂大輔, 小林亜令, 横山浩之ほか：携帯端末におけるユーザ行動パターン動的モデル化手法のための評価システムの開発, 第 6 回情報科学技術フォーラム (FIT2007), pp.261-262,M-054,2007.

[53] Siewiorek D.,Smailagic A., Furukawa J., et al.: SenSay: A Context-Aware Mobile Phone, Proc. of 7th IEEE International Symposium on Wearable Computers, pp.248-249,2003.

[54] Karause A., Smailagic A.,Siewiorek P. D.: Context-Aware Mobile Computing: Learning Context-Dependent Personal Preferences from a Wearable Sensor Array, IEEE transaction on mobile computing, Vol.5, No.2, pp.113-127,2006.

[55] Zhang H., Schreiner C., Zhang K.,et al.: Naturalistic Use of Cell Phones in Driving and Context-Based User Assistance, Proc. of the 9th international conference on Human computer interaction with mobile devices & services, ID-101,2007.

[56] Sohn T., Varshavsky A., LaMarca A., et al.: Mobility Detection Using Everyday GSM Traces, Proc. of 8th International Conference on Ubiquitous Computing,2006.

[57] 遠山緑生，豊田陽一，加藤文彦，服部隆志: コンテキスト情報と操作履歴の関連付けによる操作予測システムの提案，情報処理学会研究報告 -,No.2004-UBI-006,pp.83-90,2004.

[58] 河口信夫，宮崎俊和，稲垣康善: ユビキタス情報環境における履歴を用いた機器操作支援手法，情報処理学会研究報告 -,No.2004-UBI-004,pp.57-62,2004.

[59] 内田 渉, 笠井 裕之, 倉掛 正治: コンテキスト依存型サービスのスケーラブルな実行制御方式, 電子通信学会論文誌 D,Vol.J90-D,No6,pp.1403-1416,2007.

[60] 高田浩和，近藤弘郁，清水徹: 4M バイト DRAM 内蔵 32 ビット RISC マイクロコントローラ M32Rx/D，三菱電機技報， vol.73,no.3,pp.182-185,1999.

[61] ARM: ARM Processor Overview,(online), avalilable from http://www.arm.com./products/CPUs/(accessed 2008-04-11)

[62] "SuperH プロセッサ，"CQ 出版社， TECHI,Vol.1,1999.

[63] Sun Microsystems, Inc: The Mobile Information Device Profile, (available from) http://java.sun.com/products/midp/, (accessed 2008-04-28)

[64] JCP:JSR-000030 J2ME Connected, Limited Device Configuration,(on line), available from http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html, (accessed 2008-05-02)

[65] JCP:JSR-000139 Connected Limited Device Configuration 1.1.(on line), available from http://jcp.org/aboutJava/communityprocess/final/jsr139/ (accessed 2008-05-02)

[66] KVMMark,(on line),available from http://www.seckey.net/iappli/KVMMark.html (accessed 2003-04-01)

[67] Baysun: 放電特性カーブの見方 バッテリー - ベリサン -,(on line),avairable from http://www.baysun.net/lithium/lithium10.html
(accessed 2008-04-07)

[68] 山崎 亜希子, 五味田 啓: 加速度センサ等を用いた移動状態判定方式の検討, 情報処理学会第 70 回全国大会 ,1E-3,2008.

[69] Apple:Apple-iphone,(online), avairable from
http://www.apple.com/iphone/ (accessed 2008-4-11)

[70] Parhi P.,Karlson K. A. and Bederson B. B.: Target Size Study for One-Handed Thumb Use on Small Touchscreen Devices, Proc. of the 8th international conference on Human computer interaction with mobile devices & services, pp.203-210,2006.

[71] 増井俊之: 予測 / 例示インタフェースの研究動向，コンピュータソフトウエア，Vol.14， No.1， pp.1-16(1997).

[72] Toshiyuki Masui, Itiro Siio: Real-World Graphical User Interfaces, Proc. of the International Symposium on Handheld and Ubiquitous Computing (HUC2000), pp.72-84(2000).

[73] 佐藤充子，岡田英悟，中本幸一: カスタマイズ可能な携帯端末向けユーザインタフェースの実装と評価，情報処理学会研究報告，No.2007-UBI-013，PP.195-202(2007).

[74] http://www.acrodea.co.jp/product/ui/
index.html

[75] Mika Racento, Antti Oulasvirta, Renaud Petit and Hannu Toivonen: ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications, IEEE Pervasive Computing, Vol2. pp.51-59,2005.

[76] Niels Landwehr and Luc De Raedt: r-grams: Relational Grams, IJCAI2007, pp.902-912(2007)

[77] 山田辰美，服部正嗣，平松薫，柳沢豊，岡留剛: 実世界指向コンテキストアウェアサービスの入力に着目した比較，電子通信学会論文誌 D， Vol.J90-D， No.3， pp.820-836(2007).

[78] 元田浩，津本周作，山口高平，沼尾正行: データマイニングの基礎，オーム社 (2006).

[79] 有馬 啓ほか： PDA における Java 実行の高速化の一方式，情報処理学会論文誌,Vol 42. No 6,pp.1535-1544(2001).

[80] 興津志信， 坂崎芳久， ”地上デジタルハイビジョンテレビ用ソフトウェア， ”東芝レビュー， vol.56,No.12, pp.13-16,Dec. 2003

[81] Innopath: Understanding Firmware Over The Air-FOTA,(online), available from http://www.innopath.com/pdf/fota.pdf (accessed 2008-03-21)

論文，国際会議以外の研究業績一覧

■ 研究会発表

(1) 大寺信行, 斎藤年史, 清原良三, 西開地秀和, 安井裕: EVLIS マシン上の PROLOG インタプリタとその動特性, 情報処理学会研究報告,No.1983-PRO-027, pp.1-6,1983.

(2) 中尾浩一, 和田久美子, 清原良三, 近山隆: 並列論理型言語 KL1 のデバッグ環境, 情報処理学会研究報告, NO.1992-PRO-008, pp.131-138, 1992.

(3) 尾崎敦夫, 清原良三, 中島克人, 阿部一裕, 瀬尾和男: 時空間オブジェクトモデルを用いた並列シミュレーションーミクロ交通シミュレーションへの適用検討ー, 第 13 回シミュレーション・テクノロジー・コンファレンス, Vol. 13, pp. 299–302, 1994.

(4) 清原良三, 桜田博, 高橋克英, 黒田正博, 下間芳樹: 移動体端末向けメッセージングシステム ー端末、サーバ間の通信方式ー, 情報処理学会研究報告,No2000-MBL-013, pp. 1-8, 2000.

(5) 清原良三, 栗原まり子, 高橋清, 橘高大造: 携帯電話の SW 更新に関する検討, 情報処理学会研究報告,No.2002-MBL-022, pp. 93-100, 2002.

(6) 高橋克英, 清原良三, 坂本守: 携帯端末向け Java の高速化手法の検討, 情報処理学会研究報告,No-2002-MBL-022, pp. 79-86, 2002.

(7) 清原良三, 栗原まり子, 古宮章裕, 高橋清, 橘高大造: 携帯電話の SW 更新を目的としたモジュール分割に関する検討, 情報処理学会 DICOMO2003, Vol. 2003, p. 73-77, 2003.

(8) 清原良三, 栗原まり子, 根谷崎恵理子, 橘高大造: 携帯電話 S/W の拡張方式, 情報処理学会 DICOMO2004, Vol. 2004, 3F1, 2004.

(9) 清原良三, 栗原まり子, 三井聡, 古宮章裕: 組込み SW の特性に基づいたバージョン間差分抽出方式, 情報処理学会研究報告,NO.2004-MBL-031, pp. 9-16, 2004.

(10) 前田慎司, 岡田英明, 清原良三: 携帯端末における複数 Java アプリケーション起動方法の検討, 情報処理学会研究報告,No-2004-MBL-31, pp. 17-24, 2004.

(11) 清原良三, 栗原まり子, 小野良恭, 三井聡: 携帯電話 S/W のバージョン間差分抽出方式, 情報処理学会 DICOMO2005, Vol. 2005, 8B1, 2005.

(12) Satoshi Mii, and Ryozo Kiyohara: Software Structures for Updating Mobile Phone Software, 情報処理学会 DPS ワークショップ論文集 2005, No. 19, pp. 60-64, 2005.

(13) 前田慎司, 栗原まり子, 岡田英明, 宮内直人, 寺島美昭, 宮本剛, 清原良三: シームレス通信のためのソフトウェア無線端末の管理方式, 情報処理学会研究報告, No.2005-MBL-035, pp. 23-30, 2005.

(14) 清原良三, 岡田英明, 三井聡: 携帯端末における S/W アドインのためのアプリケーション管理方式, 情報処理学会研究報告, No-2006-MBL-036, pp. 209-214, 2006.

(15) 清原良三, 深澤司, 前田慎司, 高橋克英: 携帯端末間における情報共有のための通信経路決定方式, 情報処理学会 DICOMO2006, Vol. 2006, 1A1, 2006.

(16) 清原良三, 清水直樹, 松本光弘, 栗原聡, 沼尾正行: 携帯端末におけるユーザ操作支援方式の提案, 情報処理学会研究報告,No.2006-MBL-039, pp. 89-96, 2006.

(17) 清原良三, 三井聡, 松本光弘, 沼尾正行, 栗原聡: 携帯端末のコンテキスト情報利用による操作性向上方式, 情報処理学会 DICOMO2007, Vol. 2007, pp. 1712-1719, 2007.

(18) 松本光弘, 清原良三, 福井秀徳, 沼尾正行, 栗原聡: 携帯電話におけるコンテキスト情報を用いたユーザの操作予測, 人工知能学会 第 79 回知識ベースシステム研究会資料 (SIG-KBS-A702). pp. 87-92. 2007.

(19) 岡田英明, 清原良三: 携帯端末向け Java 省メモリ実行環境の検討, 情報処理学会研究報告,No.2008-MBL-044,pp.51-58,2008.

(20) 清原良三, 三井聡, 松本光弘, 沼尾正行, 栗原聡: 携帯電話におけるコンテキスト情報としての低消費電力位置情報取得方式, 情報処理学会研究報告,No.2008-MBL-045,pp.31-36, 2008

(21) 松本光弘, 清原良三, 福井秀徳, 沼尾正行, 栗原聡: コンテキスト情報を用いた携帯電話のアプリケーションメニューの構築と評価, 情報処理学会 DICOMO2008, Vol. 2008, 1G-4, 2008(採択済）.

(22) 清原良三, 三井聡, 松本光弘, 沼尾正行, 栗原聡: 携帯端末 S/W 更新における高速プログラム圧縮方式情報処理学会 DICOMO2008, Vol. 2008, 4C-4, 2008(採択済).

■ その他雑誌

(1) 撫中達司, 清原良三: JavaOS の現状と将来, *Java World*, Vol. 1, pp. pp.56-61, 1997.

(2) 清原良三, 高橋克英, 三井聡, 橘高大造, 木野茂徳: モバイルミドルウェア技術, 三菱電機技報, Vol. 79, No. 2, pp. 27-30, 2005.

(3) 撫中達司, 清原良三, 大野次彦, 吉富洋巳, 玉田純: マルチメディアサーバ, 三菱電機技報 Vol. 71, No. 2, pp. 9-12, 1997.

■ 講演

(1) 清原良三: モーバイルネットワークコンピュータとアプリケーション, 情報処理学会連続セミナー 97 第 2 回, セッション 1, 1997

■ 大会

(1) 伍井啓恭, 清原良三, 鈴木克志, 太細孝: カタカナ異表記処理, 第 38 回情報処理学会全国大会講演論文集, Vol. 1989, No. 1, pp. 351-352, 1989.

(2) 尾崎敦夫, 清原良三, 瀬尾和男, 中島克人: 時空間オブジェクトモデルの基本設計 - ミクロ交通シミュレーションへの適用検討 -, 第 47 回情報処理学会全国大会講演論文集, Vol. 1993, No. 5, pp. 3-4, 1993.

(3) 阿部一裕, 瀬尾和男, 尾崎敦夫, 清原良三, 中島克人: ”時空間構造を導入したオブジェクト指向モデル”, 第 48 回情報処理学会全国大会講演論文集, Vol. 1994, No. 5, pp. 27-28, 1994.

(4) 清原良三, 栗原まり子, 井上淳, 斎藤謙一: マルチメディアサーバシステム (1) : システム概要, 第 53 回情報処理学会全国大会, No. 3, pp. 69-70, 1996.

(5) 栗原まり子, 井上淳, 斎藤謙一, 清原良三: マルチメディアサーバシステム (2) : 運用管理方式, 第 53 回情報処理学会全国大会講演論文集, Vol. 1996, No. 3, pp. 71-72, 1996.

(6) 川尻剛, 川村達, 粟原まり子, 斎藤謙一, 井上淳, 清原良三 マルチメディアサーバシステム (3) : データ管理, 第 53 回情報処理学会全国大会講演論文集, Vol. 1996, No. 3, pp. 73-74, 1996.

(7) 井上淳, 清原良三: マルチメディアサーバシステム (4) : セキュリティ管理, 第 53 回情報処理学会全国大会講演論文集, Vol. 1996, No. 3, pp. 75-76, 1996.

(8) 清原良三, 坂倉隆史, 井上淳, 黒田正博: 分散 IMAP4 サーバー : Nomamail(1) : 概要, 第 57 回情報処理学会全国大会, No. 3, pp. 340-341, 1998.

(9) 坂倉隆史, 井上淳, 清原良三, 黒田正博: 分散 IMAP4 サーバー : Nomamail(2) : 実現方式, 第 57 回情報処理学会全国大会講演論文集, Vol. 1998, pp. (3) 342-343, 1998.

(10) 小谷亮, 清原良三, 攝津敦, 橘高大造: 組み込み LINUX 上のソフトウエア更新方式 (1)- 全体設計 -, 第 66 回情報処理学会全国大会講演論文集, Vol. 2003, 4D-5, 2003.

(11) 河相英典, 深澤司, 小谷亮, 清原良三: 組み込み LINUX 上のソフトウエア更新方式 (2)- 圧縮データへの差分適用 -, 第 66 回情報処理学会全国大会講演論文集, Vol. 2003, 4D-6, 2003.

(12) 深澤司, 河相英典, 小谷亮, 清原良三: 組み込み LINUX 上のソフトウエア更新方式 (3)- 差分抽出方式 -, 第 66 回情報処理学会全国大会講演論文集, Vol. 2003, 4D-7, 2003.

(13) 三井聡, 清原良三, 橘高大造: 携帯電話 SW のモジュール分割方式, 第 2 回情報科学技術フォーラム, Vol. 2003, M-017, 2003.

(14) 清原良三, 栗原まり子, 根谷崎恵理子, 西村昭彦, 橘高大造: 携帯電話における機能更新方式に関する一検討, 第 2 回情報科学技術フォーラム講演論文集, Vol. 2003, M-018, 2003.

(15) 栗原まり子, 清原良三: 携帯電話における機能更新のためのダウンロード方式の提案, 第 2 回情報科学技術フォーラム, Vol. 2003, M-019, 2003.

(16) 高橋克英, 清原良三: 携帯端末 java の高速化手法の検討と評価, 第 2 回情報科学技術フォーラム, Vol. 2003, M-020, 2003.

(17) 栗原まり子, 清原良三, 渡辺拓, 古嶋寛之, 橘高大造: インターネットを利用した大規模ソフトウェアのバージョンアップ方式に関する検討, 第 3 回情報科学フォーラム講演論文集, Vol. 2004, M-010, 2004.

(18) 三井聡, 清原良三: 組込みSW 更新を目的としたオブジェクト配置方法, 第3回情報科学フォーラム講演論文集, Vol. 2004, B-030, 2004.

(19) 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, 第67回情報処理学会全国大会講演論文集, Vol. 2005, 3C-5, 2005.

(20) 高橋克英, 岡田英明, 清原良三: 携帯端末Java アプリケーションに特化した高速化手法, 第67回情報処理学会全国大会講演論文集, Vol. 2005, 3C-4, 2005.

(21) 三井聡, 清原良三: 組込みS/W 更新を目的としたオブジェクト配置順序決定方法, 第67回情報処理学会全国大会講演論文集, Vol. 2005, 3C-6, 2005.

(22) 栗原まり子, 清原良三, 西村昭彦: SDR 向けダウンロードプロトコルの検討, 電子情報通信学会2005 総合大会講演論文集, Vol. 2005, B-17-7, 2005.

(23) 深澤司, 高橋克英, 前田慎司, 清原良三: 携帯端末間同士の通信経路に関する一検討, 電子通信学会2006 総合大会講演論文集, Vol. 2006, B-7-78, 2006.

(24) 深澤司, 高橋克英, 前田慎司, 清原良三: Bluetooth 携帯端末でのアドホックネットワークモデル提案 - 携帯端末間におけるリンク確立高速化に関する検討 -, 電子情報通信学会2006 ソサイエティ大会講演論文集, 2006,B-7-75, 2006.

(25) 岡田英明, 清原良三: 携帯端末におけるセキュアアプリケーション構築環境の検討, 第68回情報処理学会全国大会講演論文集, Vol. 2006, 5A-6, 2006.

(26) 三井聡, 清原良三: 組込みSW の特性を用いたバージョン間差分抽出方式, 第5回情報科学技術フォーラム講演論文集, Vol. 2006, M-025, 2006.

(27) 三井聡, 清原良三, 栗原聡, 沼尾正行: 携帯端末UI の操作性向上方式の提案, 電子情報通信学会2–7 総合大会講演論文集, D-13-6, 2007.

(28) 清原良三, 三井聡, 松本光弘, 沼尾正行, 栗原聡: 携帯端末の操作複雑性分析とコンテキスト情報による操作支援手法の提案, 第21回人工知能学会大全国大会, Vol. 21, 1E1-4, 2007.

(29) 松本光弘, 清原良三, 福井秀徳, 沼尾正行, 栗原聡: 携帯端末による人物行動履歴の分析に関する一考察, 第21回人工知能学会大全国大会, Vol. 21, 2F3-3, 2007.

(30) 松本光弘, 清原良三, 福井秀徳, 沼尾正行, 栗原聡: 携帯端末の時空間的利用情報からの特徴的行動パターンの抽出と予測, 第6回情報科学技術フォーラム講演論文集, Vol.2007, pp.391-394. 2007

(31) 松本光弘, 清原良三, 福井秀徳, 沼尾正行, 栗原聡: ユーザのコンテキストから携帯電話の操作を予測するシステムの構築と評価, 第22回人工知能学会大全国大会, Vol. 22, 2C1-2, 2008(採択済).

■ 国内登録特許

(1) 清原良三: 特許第3247668号, 電子メール管理装置及び電子メール管理方法

(2) 坂倉隆史, 清原良三: 特許第3312886号, ファクシミリモデム装置

(3) 内藤明彦, 宮内信仁, 石川博章, 清原良三: 特許第3519653号, 通信方法及び記録媒体及び通信方式及びアダプタ

(4) 内藤明彦, 宮内信仁, 清原良三: 特許第3549837号, 経路選択装置、及び、経路選択システム

(5) 高橋克英, 橘高大造, 清原良三, 中邦博, 渡辺克也: 特許第3642772号, コンピュータ装置及びプログラム実行方法

(6) 三井聡, 清原良三, 栗原まり子, 高橋清, 橘高大造: 特許第3682050号, 組込みソフトウェア開発装置

(7) 清原良三, 高橋克英: 特許第3688286号, 情報記憶制御装置及び情報記憶制御方法及び情報記憶制御プログラム

(8) 三井聡, 清原良三: 特許第3733135号, ソフトウエア差分抽出適用システム及び差分抽出装置及び差分適用装置及びプログラム

(9) 小谷亮, 清原良三, 橘高大造: 特許第3792232号, 情報処理装置及び格納位置管理方法及びプログラム

(10) 清原良三, 特許第3798263号, 電子メールサーバ及び電子メールキャッシュ方法及び電子メールキャッシュプログラム

(11) 清原良三: 特許第3807240号, 電子メールシステム

(12) 前田慎司, 清原良三: 特許第 3900858 号, 施設案内システム

(13) 三井聡, 清原良三, 橘高大造: 特許第 4036852 号, 差分データ生成装置, 差分データ生成方法および差分データ生成プログラム

(14) 桜井鐘治, 清原良三, 高橋渉, 片岡久明：特許第 4076757 号, 撮影証明システム及び証明装置及び撮影証明方法

■ 国内審査中特許

(1) 内藤明彦, 宮内信仁, 石川博章, 清原良三: 特開 2002-368908, アダプタ, サーバ, 識別子伝達方法

(2) 栗原まり子, 清原良三: 特開 2005-086462, 携帯情報端末, 並びにソフトウエア入れ替えシステム及び方法

(3) 清原良三, 栗原まり子: 特開 2005-080175, 携帯型端末装置

(4) 清原良三：特開 2005-215841, ソフトウエア修正機能付き携帯電話端末装置

(5) 清原良三, 栗原まり子, 橘高大造：特開 2005-247819, ソフトウエア無線携帯電話端末装置

(6) 三井聡, 清原良三, 橘高大造：特開 2006-004125, 差分適用組込み機器, 差分適用組込み機器システム及び格納データ変更方法

(7) 三井聡, 清原良三, 橘高大造：特開 2008-041112, 差分データ生成装置, 差分データ生成装置の差分データ生成方法および差分データ生成プログラム

(8) 清原良三：特開 2006-209417, 更新用データ送信システム及び送信側装置及び受信側装置及び更新用データ送信方法

(9) 高橋克英, 清原良三：特開 2006-171927, プログラム実行装置及びプログラム実行方法及びデータ領域管理プログラム

(10) 栗原まり子, 清原良三, 渡部拓, 古嶋寛之：特開 2006-293512, ソフトウエア更新情報配布システム及びソフトウエア更新情報配布方法

(11) 栗原まり子, 清原良三, 三井聡, 特開 2007-066007, データ生成装置及びデータ生成方法及びデータ生成プログラム

(12) 清原良三，前田慎司，高橋克英，特開 2007-172295，共有データ情報表示装置、共有データ情報表示システム及び共有データ情報表示方法

(13) 岡田英明， 清原良三：特開 2007-166191，通信装置及び通信方法及びプログラム

(14) 三井聡， 清原良三，小浦邦和，塩崎秀樹，横田守真：特開 2007-557722，エレベータ制御プログラムの遠隔更新システム

(15) 三井聡， 清原良三：特開 2007-219768，差分生成装置及び差分適用装置及び差分生成プログラム及び差分適用プログラム

(16) 前田慎司，高橋克英， 清原良三：特開 2007-272295，データ共有システム及びデータ共有方法及びデータ配布端末

■ 国際登録特許

(1) 坂倉隆史, 清原良三: 米国特許第 6753980 号, ファクシミリモデム装置

(2) 三井聡， 清原良三，小浦邦和，塩崎秀樹，横田守真：台湾，エレベータ制御プログラムの遠隔更新システム

■ 表彰

(1) 2002 年度情報処理学会第 22 回モバイルコンピューティングとワイヤレス通信研究会優秀論文賞，携帯端末向け Java の高速化手法の検討

(2) 2003 年度情報処理学会マルチメディア，分散，協調とモバイル (DICOMO2003) シンポジウム優秀論文賞，携帯電話の SW 更新を目的としたモジュール分割に関する検討

(3) 2005 年度情報処理学会マルチメディア，分散，協調とモバイル (DICOMO2005) シンポジウム優秀論文賞，携帯電話 S/W のバージョン間差分抽出方式

(4) 平成 18 年度関東地方発明表彰発明奨励賞, 特許第 3682050 号，組込みソフトウェア更新技術

(5) 平成 19 年度関東地方発明表彰発明奨励賞，特許第 3642772 号，コンピュータプログラム高速実行技術