推薦論文

携帯電話ソフトウェアの更新方式

清原良三[†] 栗原まり子[†] 古宮章裕[††]
高橋清[†††] 橘高大造[†††]

近年の i-mode をはじめとする携帯電話のネットワーク接続サービスの開始により，携帯電話上に搭載するソフトウェアの規模は急激に拡大している．そのため障害のない状態で携帯電話を市場に出すことが困難な状態になりつつある．そこで，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．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．データ転送量を小さくするために，新版と旧版の差分データの転送量を小さくする必要がある．またフラッシュメモリの該当部分だけを書き換えればよ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．本論文では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うえで，実際の携帯電話のソフトウェアを利用して評価した．

A Method of Software Update on a Mobile Phone

Ryozo Kiyohara,[†] Mariko Kurihara,[†] Akihiro Komiya,[††]
Kiyoshi Takahashi[†††] and Taizo Kittaka[†††]

Due to increasing services installed in cellular phones (e. g. i-mode), it is difficult to release bug-free cellular phones. We have proposed a technology for updating cellular phones' software in a shorter period. The time required for software update consists of time for data transmission and time for rewriting data in Flash ROM. Data size should be small. In order to reduce the data size, differences between old version and new one should be as few as possible. It is required to update only the needed area as well as to keep the development environment simple. In this paper, we investigated module fragment method to minimize the updating time and evaluated our updating software in optimized method.

1. はじめに

最近，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．すなわち，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き換えサービスや，無線を利用したサービスも検討されつつある[3),4)]．また，携帯電話に限らず組み込み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は問題がないが修正しておいた方が良い程度

† 三菱電機株式会社情報技術総合研究所
INFORMATION TECHNOLOGY R & D CENTER, MITSUBISHI ELECTRIC CORPOLATION
†† 三菱電機インフォメーションシステムズ株式会社
MITSUBISHI ELECTRIC INFORMATION SYSTEMS
††† 三菱電機株式会社モバイルターミナル製作所
MOBILE TERMINAL CENTER, MITSUBISHI ELECTRIC CORPOLATION

本論文の内容は平成 15 年 6 月のマルチメディア，分散，協調とモバイル（DICOMO2003）シンポジウムにて報告され，DICOMO2003 プログラム委員会委員長により情報処理学会論文誌への掲載が推薦された論文である．

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検討されている[3),4)]．その場合，更新時間中はまったく機能が使えなかったり，一部の機能しか使えなかったりするために，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．ゆえに，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．すなわち，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，すなわ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，この差分をいか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．

すでに筆者らはソフトウェアの構成方法としてモジュール分割とベクターテーブルを利用することを提案し，局所的なソフトウェアの修正が局所にしか影響しない特徴を持つ方法[7)]を示している．また，差分の表現方法においても差分のデータが小さくなる手法を示している[8)]．

本論文では，実際のモジュール分割に際し，具体的な分割方法の指標を示すとともに，実際に市場に出ている携帯電話を利用した差分情報に関するデータを用いた評価結果を示す．

2. 携帯電話のソフトウェア更新における課題

2.1 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 ROM 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．

2.2 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たりす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 1**）．また，モジュール間の参照は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**）．

3. モジュール分割

3.1 モジュール分割概要

本論文においては，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

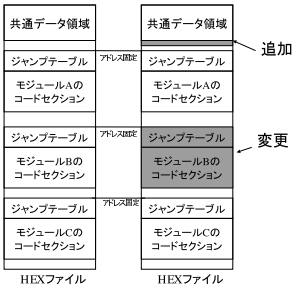

図 1 局所の修正は局所にのみ影響する構造
Fig. 1 An illustration of a proposed software structure.

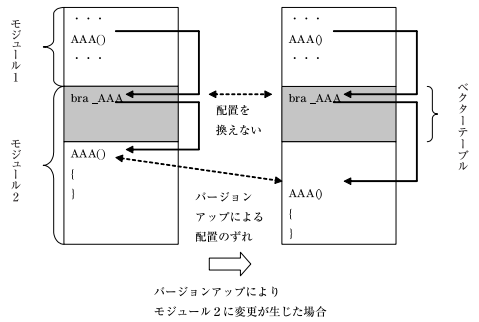

図 2 モジュール分割とベクターテーブル
Fig. 2 A module partition and a jump vector table.

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，ベクター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．すなわち，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，ベクター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ー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ー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ー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ーテーブルは詰めることをせず空き領域としておく．

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 1，図 2）．

我々は CPU として M32R[10] のような場合にはその有効性をすでに示した．[7] 10 分割程度において，以下の 2 点に関してもすでに評価を行った[7]．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 1**，**表 2** に示す．

(1) ベクター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 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分かる．
(2) 表 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．モジュール間相互にわたるであろう描画やメモリ書き込みにおいて

表 1 メモリ容量への影響
Table 1 Some numbers on the target software and its module partition.

ソフトウェアサイズ	約 12 MB
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 600 KB

表 2 Java ベンチマークプログラムの性能劣化率
Table 2 Performance decline ratios of the module-partitioned software on some Java benchmark programs.

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
平均	1.2

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ヘッドはわずかであることが分かる．

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き換えブロック数は減らせる．すなわち書き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．

3.2 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，そのバランスのと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．1 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，1 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．

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

1 つのシンボルが参照される平均の数は rg^{-1} である．1 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．

1 つのシンボルが少なくとも 1 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．したがって，1 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 (1) より，以下のように表すことができる．

$$vectorInM(x) = (1 - x^{-rg^{-1}})global(x) \tag{2}$$

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

$$vector(x) = x * vectorInM(x) = g - gx^{-rg^{-1}} \tag{3}$$

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．

$$local(x) = rx^{-2} \tag{4}$$

したがって x 分割した場合で，1 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．

$$f(x) = x^{-1}vector(x) + x^{-1}local(x) \tag{5}$$

すなわち

$$f(x) = gx^{-1} - gx^{-rg^{-1}-1} + rx^{-3} \tag{6}$$

これを微分すると以下の式になる．

$$f'(x) = -gx^{-2} + (r+g)x^{-rg^{-1}-2} - 3rx^{-4} \tag{7}$$

式 (7) はつねに負となり単調減少である．

図 3 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つながる．

ベクターテーブルの大きさは式 (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 カ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

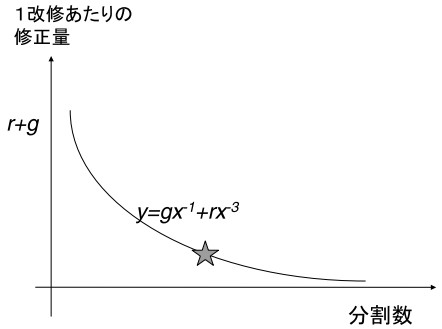

図 3 1 改修あたりの修正量とモジュール数の関係
Fig. 3 A correlation between the number of modules and the size of changes.

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なメモリ量は以下になる．

$$memory(x) = abx + vector(x) \quad (8)$$

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 3 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 点の考慮したうえ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．

- 1 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とれるか．
- 決して均一な参照とは限らない．

3.3 差分表現と差分量

前節までの修正量を実際に差分表現することを考える．差分は diff[12] や rsync[11] などのアルゴリズムに基づき作成する．またその表現形式としては gdiff[13] に基づく方法とした．

基本的には差分を以下に示す 3 つの情報で表す．

(1) 新版と旧版で内容も位置も変わってないもの
(2) 新版と旧版で内容は同じで位置のみずれているもの
(3) 新版にはあるが，旧版にはなかった情報

たとえば，次の (1)，(2) を比較してみる．

(1) abcdefghijklmnopqrstu
(2) abcdeffghijkbabcdefstu

この場合の差分は最初の文字を 0 番目として以下のように表現する．

skip 6（最初から 6 文字は内容も位置も不変）
copy 6 5（次の 6 文字は旧版の 5 番目から順にコピー）
data 1 b（次の 1 文字は b）
copy 6 0（次の 6 文字は旧版の 0 番目から順にコピー）
skip 3

このような表現の場合，フラッシュメモリのアドレス情報がさらに必要になる．4 バイトのアドレス空間があるとすると skip を表現するには skip を示すコマンドと 4 バイト分のサイズ情報から 5 バイト必要である．copy はサイズ情報を 4 バイトで表現するとすれば 9 バイトの情報が必要となる．

表 3 差分表現
Table 3 Delta commands.

コマンド種別	コマンド値	パラメータ 1	パラメータ 2
EOF	0	なし	なし
DATA	1-249	更新データ（コマンド値分）	なし
FLUSH	249	なし	なし
COPY	250	コピー元位置	コピー長（1 バイト）
COPY	251	コピー元位置	コピー長（2 バイト）
COPY	252	コピー元位置	コピー長（3 バイト）
SKIP	253	スキップ長（1 バイト）	
SKIP	254	スキップ長（2 バイト）	
SKIP	255	スキップ長（3 バイト）	

そこで skip と copy を示すコマンドを複数用意し，それぞれに必要なサイズ情報を 1 バイトから 3 バイトとすることにより小さな単位の skip，copy を小さなコマンドで表すことができる．具体的には **表 3** に示すようなコマンドとして表現することとした．

また，携帯電話では搭載しているフラッシュメモリのサイズに比べて RAM の容量は小さい場合が多い．そこで差分を適用する動作は，フラッシュメモリの消去ブロック単位でコードイメージを RAM 上に展開してはフラッシュメモリに書き込む操作の繰返しとなる．

もしコピーの発生する範囲を限定しなければ，書き換えを実際に行う際にコピーを意味するコマンドを実行するときに元のデータがすでに書き換えられてしまうことにより存在しない場合が起こりうる．安全のためにフラッシュメモリを二重化できる場合にはこのような問題は発生しないが，携帯電話においては多くの場合コストの問題で許されない．

フラッシュメモリの二重化ができない場合は限られた範囲内でコピーを行うことを前提として差分抽出を行って，一定の RAM サイズでの動作を保証する必要がある．そのため，フラッシュメモリへの書き込みのタイミングを指示するコマンド（FLUSH）も追加する必要がある．

4. 評　　価

4.1 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** にその結果を示す．

表 4 モジュール分割数と関数の参照数の関係
Table 4 Influences of the module partition.

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	362,954	0	362,954
2	349,717	2,336	174,859
4	337,722	7,356	84,430
9	336,243	7,912	37,360
10	320,075	9,436	32,007
11	304,663	12,996	27,696
12	300,968	14,236	25,080
13	295,006	15,640	22,692
15	289,949	15,936	19,329
18	286,416	16,600	15,912

ここで，総参照数として $r = 363{,}247$，シンボル数は $g = 29{,}681$ である．すなわち $rg^{-1} \simeq 12$ で，式 (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．

$$f(x) = 30{,}000(x^{-1} + 12x^{-3}) \quad (9)$$

すなわち $f(x)$ は x^{-1} で決まることが示された．

前章で検討したように，図 3 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．

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 によれば，式 (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分かる．

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．たとえば，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 100 K バイト以下とする．1 カ所の修正で 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 モジュールあたりの修正量は 100,000÷5 = 20,000 カ所が目標となる．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 1 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 カ所程度であり，表 4 によれば 13 分割以上が必要となる．

次に空き領域に関してはコード全体で 16 M 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 160 K バイトが空き領域となる．たとえば分割数を 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 56 K 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．

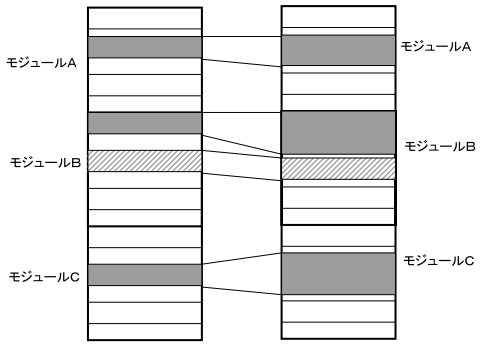

図 4 ブロックを導入したモジュール分割
Fig. 4 An illustration of the module partition introducing the idea of “blocks”.

$$space(x) = (160 - 56)/x \quad (10)$$

$x = 12$ の場合で $space(12) \simeq 9$ で表され，9 K 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．

結論として 13 分割程度は適切な数値である．しかし，表 6 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分かった．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ー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．

4.2 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，さら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，その中をさら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 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．

たとえばモジュール 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

表 5 モジュール分割と参照数
Table 5 The number of symbol references.

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	17,099	29
かな漢字変換	3,972	17
ブラウザ	19,179	42
文字列	3,856	59
ドライバ	56,798	1,578
フォント	6	0
Java	28,305	46
データ	171	335
ファームウェア	80,467	435
通信制御	84,451	49
その他	6,574	610

表 6 モジュール分割と差分サイズ
Table 6 The size of delta for each module.

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

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，2 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 1 つめのブロックの空き領域を使うことにより 2 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 6 に示すようにブロック化しない場合で最悪値約 400 KB，ブロックを導入した場合で最悪値約 80 KB あった．

4.3 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 5** にまとめた．

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かかわ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いる場合に発生する[8]．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すで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 5 からは，たとえばアプリモジュールの修正を入れると 17,099 カ所とベクターテーブルの 29 を修正する可能性があることが分かる．

1 カ所あたり 1 バイトの修正があるとしても，約 17 KB 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 85 K 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 40 K 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さらに情報が付加されることになるため，わずかな修正が 100 K 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 6**）．これはブロックすなわ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 100 KB 未満にお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，さらに調査を行い，次の 2 点に関して評価を実施した．

4.3.1 ベクターテーブルの効果

ベクターテーブルを導入することは，1 カ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うえではフラッシュメモリの書き換え時間を考慮した場合には非常に有効であることを文献 7) 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分かった．無線を利用したダウンロードを考慮した場合はベクターテーブルは，かなり有効であると考える．また文献 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さらに書き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．

4.3.2 ブロックの効果

ブロックすなわち緩やかな固定領域の実際の効果を調べるために，表 5 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，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 6 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おさえることができた．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．これらは文献 7) に示すようにそのメモリの種類ごとにベクター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．

5. おわりに

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ー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) MPU 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 の場合のベクター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

このほか携帯電話のソフトウェアの更新という観点からは次の点があると考えている．

(1) フラッシュメモリ書き換え中の電池切れなどの場合に備えた安全な書き換え手法
(2) 論理的機能ごとのバージョンアップ
(3) ホットスタートによる書き換え

今後これらの課題に取り組んでいく予定である．

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 (2003).
3) Takeichi, M., et al.: Bug Fix of Mobile Terminal Software using Download OTA, *The Asian-Pacific Network Operations and Management Symposium* (2003).
4) 星 誠司ほか：無線通信を利用した「ソフトウェア更新システム」，NTT DoCoMo テクニカルジャーナル，pp.36–41 (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.
6) トロン協会．http://www.assoc.tron.org/
7) 清原良三ほか：携帯電話の SW 更新に関する検討，情報処理学会 MBL 研究会，074, pp.1512–1526.
8) 栗原まり子ほか：携帯電話 SW のバージョン間差分データに関する検討，情報処理学会 DICOMO2003, 074, pp.301–304 (2003).

9) 清原良三ほか：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, 073, pp.297–300 (2003).
10) 高田浩和ほか：4 M 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).
11) Tridgell, A. and Mackerras, P.: *The rsync algorithm*, Australian National University, TR-CS-96-05 (1996).
12) Hunt, J.W., et al.: A Fast Algorithm for Computing Longest Common Subsequences, CACM, Vol.20. pp.350–353 (1977).
13) W3C: Generic Diff Format Specification (1997).
14) 寺薗浩平ほか：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).

(平成 16 年 4 月 1 日受付)
(平成 17 年 4 月 1 日採録)

推　薦　文

本論文は，携帯電話におけるソフトウェアの更新を局所化するために，ソフトウェア構成をモジュール分割する際の種々の工夫について提案および評価を行っている．本提案手法は実際に携帯電話に搭載されているソフトウェアをもとに評価されており，その有効性に関する信憑性も高い．本研究結果は，今後ますます加速する携帯電話のソフトウェア開発・改善を迅速に行ううえで，実用性の高いものといえる．
（DICOMO2003 プログラム委員会委員長　高橋　修）

清原　良三（正会員）

昭和 35 年生．昭和 60 年大阪大学大学院工学研究科応用物理学専攻博士前期課程修了．同年三菱電機（株）入社．昭和 63 年より（財）新世代コンピュータ技術開発機構に出向，平成 4 年三菱電機（株）に復職．携帯電話のソフトウェア更新，JavaVM 実装に関する研究開発に従事．ACM 会員．

栗原まり子（正会員）

平成 2 年図書館情報大学図書館情報学部図書館情報学科卒業．同年三菱電機（株）入社．携帯電話のソフトウェア更新技術に関する研究開発に従事．

古宮　章裕（正会員）

昭和 46 年生．平成 7 年東洋大学工学部情報工学科卒業．同年三菱電機東部コンピュータシステム（現：三菱電機インフォメーションシステムズ）（株）入社．携帯電話ソフトウェアの開発に従事．

高橋　　清（正会員）

昭和 37 年生．昭和 61 年早稲田大学理工学部電子通信学科卒業．同年三菱電機（株）入社．携帯電話ソフトウェアの開発に従事．

橘高　大造（正会員）

昭和 33 年生．昭和 59 年大阪大学大学院工学研究科機械工学専攻博士前期課程修了．同年三菱電機（株）入社．携帯電話のソフトウェア更新技術，JavaVM 実装に関する研究開発に従事．