# コンシューマ機器向けソフトウェア高速書き換え方式

清　原　良　三[†1]　　田　中　功　一[†1]　　寺　島　美　昭[†1]

携帯端末などのコンシューマ向けの機器のソフトウェア規模が巨大化し続けている．パソコンなどと違い，不具合があればソフトウェアの更新をすぐに要するなどコストがかかる．そのためなるべく不具合を少なくして出荷する必要性がある．製品の開発においてはクロス環境上のシミュレータを利用することにより効率良くデバッグ試験などを行うこともできるが，各種ネットワークを利用した場合の複合処理などタイミングに依存して起こる障害などもあり，出荷直前には実機での検証は不可欠である．出荷直前にはデバッグ，コード修正，ビルド，ダウンロード，動作確認の繰り返しになる．本論文ではこの繰り返しの中で，既存のダウンロード高速化手法を改良，評価し，効果があることを確認した．

# Reprogramming Method for Mobile Devices

Ryozo Kiyohara,[†1] Koichi Tanaka[†1]
and Yoshiaki Terashima[†1]

This paper proposes a method for fast downloading the binary code to mobile devices during the testing and debugging phases in the development of mobile devices. The increasing number of features in mobile devices has made it difficult to release bug-free devices. Software for mobile devices has to be developed and tested using a cross-platform simulator. However, many cases have to be considered while using the target devices during testing, making several debugging phases and software update releases inevitable. These processes have to be performed iteratively. Therefore, the time required for the binary code to download to the target devices should be small. In this paper, we improve a previous study for fast downloading for mobile devices and evaluate it.

†1 三菱電機（株）情報技術総合研究所
MITSUBISHI ELECTRIC CORPORATION INFORMATION TECHNOLOGY R & D CENTER

## 1. はじめに

携帯端末やカーナビゲーションシステムのソフトウェアの規模が増大し，組込みソフトウェア開発の危機が叫ばれて久しい．技術者のスキルアップなど組込み技術者が養成される一方で，クロス環境上でのシミュレータの充実などデバッグの効率化なども図られている．しかしながら，携帯端末などで，複数の通信機能を搭載し，複雑な動作を状態遷移表で管理するようなケースではシミュレータで動作確認しただけでは不十分なケースも多い．そのため，出荷直前になると，多数の開発者や試験者が図1に示すように，携帯端末の実機に最新のソフトウェアをダウンロードしては動作試験を繰り返すということが多い[1)]．開発の最終フェーズでの効率化を目指すには，以下に示すようなアプローチがそれぞれ重要であると考える．

- 障害の発生を抑えること．
- シミュレータなどクロス環境上での開発をなるべく多くすること
- ソフトウェアのバージョンアップを高速にすること

障害の発生を抑えるためのソフトウェア開発技術は，組込み向けにも，例えば文献 2),3) などの数多くの研究があり，開発手法からツール類まで製品ベースでも存在している．しかし，これ

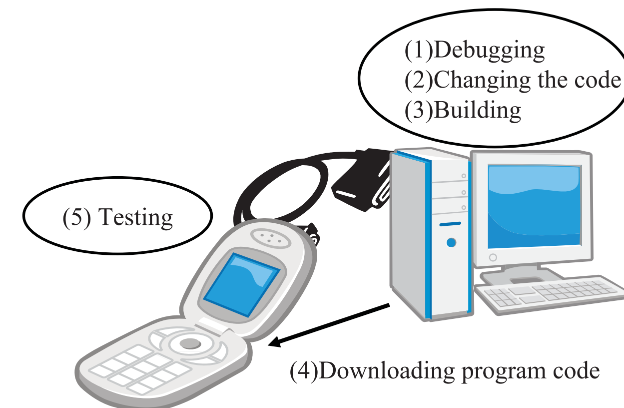



**図 1**　デバッグ，試験，確認サイクル

らの手法で障害を完全に排除できるわけではない．

また，シミュレータなどの環境も充実してきており，動作確認も十分可能ではあるものの，通信機能などを利用した場合のタイミング障害などの動作確認はシミュレータでは十分なわけではなく，出荷時には必ず実機で動作確認する必要があり，やはりダウンロードしての動作確認を省略できるわけではない．

ソフトウェアの規模が大きければ，ダウンロードするデータ量は増大し，ダウンロード時間は増す．また，フラッシュメモリへの書き込みデータ量も多くなるため，書き換えに時間がかかる．バイナリパッチで一部だけ書き換えるという手法もあるが，そもそも動作のタイミングを気にしなければならないケースに実機検証を行うのであり，その後，正式なソフトウェアをビルドした際の動作とは変わる恐れもある．また，パッチのままでリリースすると，将来のバグの元でもあるため避けるべきである．

そこで，このように何度もソフトウェアを繰り返しダウンロードする場合の高速化手法として，一定の効果があることを示した手法[4)] に関して検証した結果，携帯端末に付属するシリアルの通信速度や，ソフトウェアの構成方法に依存する要素も多く，既存の手法を単純に実装するだけでは，常に効果がでるわけではないことがわかった．

そこで，本論文では，既存の手法を複数の確度から検証し，改良点を絞った上で，改良方式を提案し，有効であることを示す．

## 2. 関 連 研 究

携帯電話のソフトウェア更新技術[5)6)] を適用するとダウンロードするデータ量を削減でき，書き換え時間も短くできる．しかし，ダウンロード先のプログラムデータと同じものをサーバ上にも保持し，新しい版との差分をサーバ上で計算してから最小化した差分データを送るため，携帯端末上のプログラムデータのバージョン管理が必須である．出荷直前で複数の人がデバッグしている環境では多くの場合は，一定のタイミングで全員が修正したファイルを管理サーバ上にコミットし，バージョン管理を行うことが多い．

個々の開発者が端末実機上にダウンロードしているソフトウェアのバージョンを管理をすることは困難であり，ミスを起こし易くなると想定される．そのため，バージョン管理が必須となる手法は，個々の開発者が修正しては試してみるというフェーズではこのままでは適用できない．

そのため，バージョン管理をせずにしかも高速にダウンロードできる手法が必要となる．このようなこと実現する方式としては， rsync[7)8)] がある． rsyc はバージョン管理せずにネットワークを経由して複数のファイルの間での差分を転送して同期を取る技術である．ファイルを一定のブロックに分割し，ブロック単位で複数のハッシュ値を計算し，複数のハッシュ値が一致すれば同一の内容であると判断し転送しない．このようにして差分を計算し情報を転送することにより実現するが，ファイルが対象であり，そのままでは携帯電話などの組込みソフトのプログラムには適用できない．

組込みソフトのプログラム更新に rsync を適用した例として，センサーネットのノードのプログラム更新に適用する研究がある[9)10)]．これらの研究では，センサノードのマルチホップ通信を利用してプログラムの更新を行う際に rsync の適用を行う．しかし， rsync ではハッシュ計算を必要とし，センサノードのような CPU リソースの小さい環境ではそのまま適用できないため，結局バージョン管理を行い，サーバ側で rsync のハッシュ計算を行うことにより解決している．そのため，この方式もそのまま適用はできない．

しかし，サーバから携帯端末にダウンロードする方式ではサーバ上でのハッシュ計算時間は気にする必要がほとんどないと想定されるため，オリジナルの rsync を携帯電話に組み込むための手法を提案した文献 4) の方式が有効であると考える．この方式の既存の rsync との違いは変更ブロックの探し方や比較ブロックの大きさなどを環境に合わせる必要がある点などであるが，一般的に効果がありそうだということのみで，実際にどうパラメタを設定すれば効果的なのかまでは示せていない．

そこで，本論文では，既存の方式[4)] を細かく分析し，改良するべきポイントを洗い出した上で，改良方式を提案する．

## 3. 既存携帯端末ダウンロード高速化手法

### 3.1 アルゴリズム

文献 4) に示された既存手法を説明する．この手法は，図 2，に示したアルゴリズム，図 3に示した処理シーケンスで動作する．その手順を以下に示す．

(1) 携帯端末上のバイナリコードを一定の分割サイズに全体を分割し， 2 種類のハッシュ値を一定サイズの比較ブロックごとに計算する

(2) 計算したハッシュ値のペアをすべてサーバ側に送る

(3) ハッシュ値が同じとなるブロックをサーバ側でブロックの始点をずらしながら検出する．ここで，フラッシュメモリは，図 4に示すように，一旦フラッシュメモリのセクターと呼ばれる複数のページからなる消去ブロック単位に RAM に読み込んでからページ単位に書き換える．そのため，図 5に示すように実際にコピーを試みる際に当該データが既に書き換えられているケースを考慮しなければならない．そのため，書き換えを実施してな

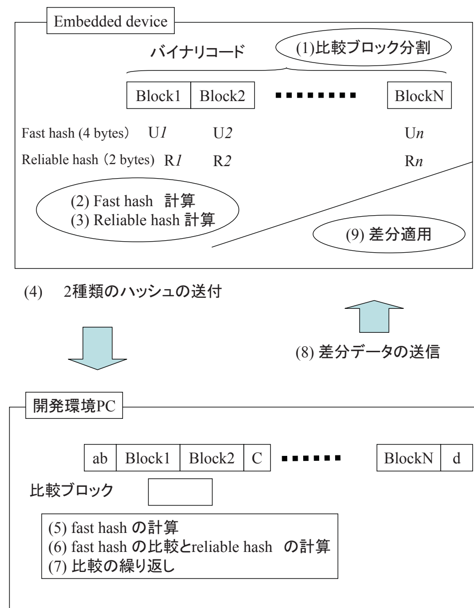



**図 2** 差分抽出アルゴリズム

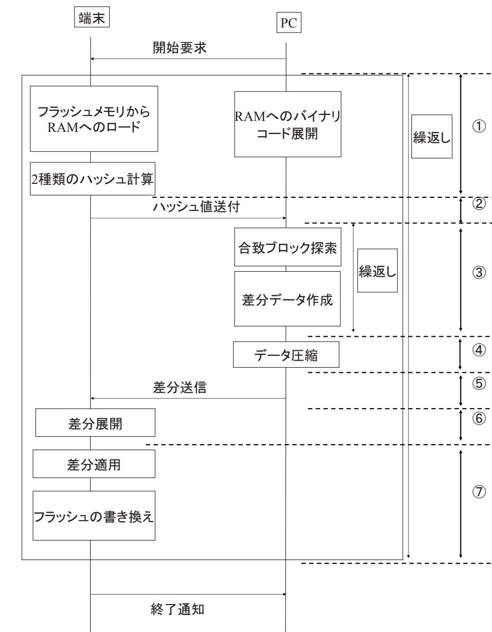



**図 3** 差分抽出処理シーケンス

いはずの範囲に限って行う．

(4) 前記探索で，携帯端末上のブロックと同じであると判断できる部分と異なると判断できる部分に区別し，異なると判断できる部分を携帯端末側に送るデータの候補とする．

(5) 携帯端末側に送信する差分データを圧縮する

(6) 携帯端末側に差分データを送信する

(7) 携帯端末側で，差分データを解凍する

(8) 解凍した差分データを携帯端末上のソフトウェアイメージに適用しフラッシュメモリを書き換える．

このようにして差分を送る．しかし，複数のハッシュを使えば必ず同じイメージである保証はない．この点に関しては文献 12) で一般論として検証し，文献 4) でも議論しているように，試験環境として利用するこの方式で十分な確率で同じデータになるため問題にならないと考える．

## 4. 性 能 分 析

### 4.1 既存方式の性能

既存方式は表 1に示す性能の評価環境で，表 2に示す特性を持つ $A, B$ の種類の評価データを利用することにより，比較ブロックサイズに応じて図6に示すような特性を持つ．パターン A では書き換え量が少ないため，比較ブロックサイズを大きくしても書き換え量は一定になる．パターン B では比較ブロックを大きくすればするほど無駄な書き換えも多くなり書き換え時間が大きくなる．しかし，これらの時間はソフトウェア構成にも依存し，ダウンロードサイズやシリアルの通信速度とも互いに影響し，実際にはどうパラメタを調整すれば最適なのかがこれだけで

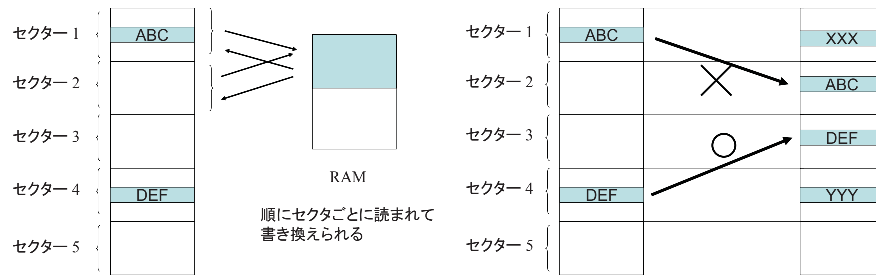



**図 4**　フラッシュメモリの書き換え

**図 5**　書き換え制約

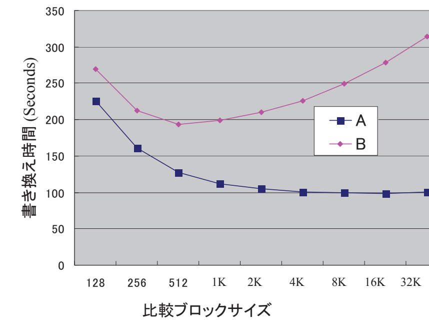



**図 6**　トータルダウンロード時間

**表 1**　評価環境の性能データ

| 項目 | 性能 | 備考 |
|---|---|---|
| 転送 サーバ→携帯端末 | 139KB/秒 | |
| 転送 携帯端末→サーバ | 70KB/秒 | |
| 圧縮 (サーバ) | 705KB/秒 | 4Kbyte 単位圧縮 |
| 展開 (携帯端末) | 1.41MB/秒 | 4KByte 単位圧縮データの展開 |
| フラッシュ書込 | 2.13MB/秒 | |
| フラッシュ読込 | 4.27MB/秒 | |
| チェックサム計算 | 204.08MB/秒 | |

**表 2**　評価対象データの特性

| パターン名 | 更新消去ブロック数 |
|---|---|
| A | 14 |
| B | 409 |

は不明である．

ダウンロード性能は，表3に示すような性能トレードオフがあり，シリアルの通信時間や，ソフトウェア構成手法による工夫などと関係して適切なパラメタが決まる．図3に示した区間ごとに性能を検討していくため，更新パターン $B$ を利用して詳細な時間を計測した．ここで，区間 $i$

**表 3**　性能データトレードオフ

| | | ブロックサイズ大 | ブロックサイズ小 |
|---|---|---|---|
| 転送データサイズ | 差分情報サイズ | 大 | 小 |
| | ハッシュ転送サイズ | 小 | 大 |
| ハッシュ衝突確率 | | 低 | 高 |
| ハッシュ計算時間 | | 小 | 大 |

の所要時間を $T_i$ とし，その区間のデバイス側所要時間を $D_i$，サーバ側所要時間を $S_i$ とする．全ダウンロード時間は以下で示される．

$$DownloadTime = \sum_{i=1}^{7} T_i \tag{1}$$

ほとんどの区間の実行速度はシリアルの通信や CPU 速度などの H/W の性能と比較ブロックのサイズに依存する．まずは，シリアルの通信や CPU 速度を固定し，比較ブロックのサイズを変えながら実行時間を測定した．図 7に全体の処理時間を区間ごとに整理した．

### 4.2　各区間の意味と実行時間測定結果

#### 4.2.1　区　間　1

区間 1 は携帯端末上でのハッシュ値の計算とサーバ上でのプログラムイメージのメモリへのロードが並行して動作する． CPU パワーの貧弱な携帯端末上での計算時間の方が大きいと想定

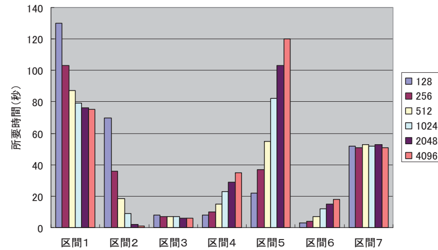



**図 7**　区間ごとの所要時間

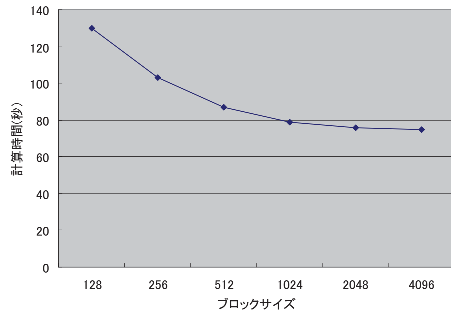



**図 8**　ハッシュ値計算性能

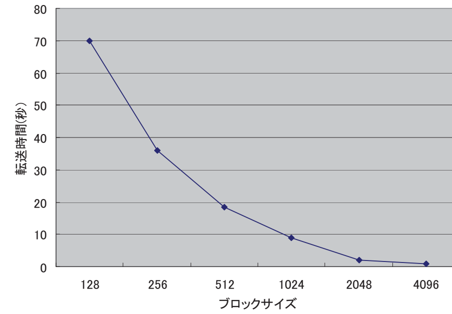



**図 9**　ハッシュ値転送時間

される．従って次式で示されるようにハッシュ計算の時間で決まる． $b$ をブロックサイズ， $D_i$ を区間 $i$ での携帯端末上での計算時間， $S_i$ を区間 $i$ でのサーバ上での計算時間とする．

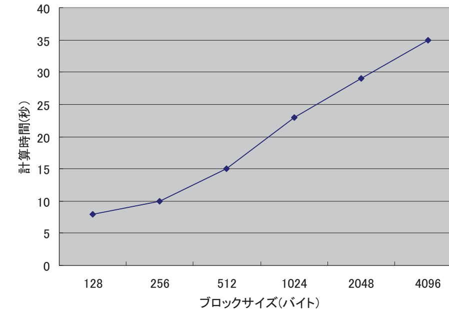
$$T_1 = max(D_1(b), S_1) \quad (2)$$

$$= D_1(b) \quad (3)$$

ハッシュ計算は比較ブロックの大きさ即ち，ハッシュ値の数とメモリの大きさで決まる．図 8にブロックサイズごとのとハッシュ計算時間の実測値をグラフで示した．ある程度以上のブロックサイズにするとメモリを読む量に依存する要素が大きくなるため，一定のブロック数以上では速度に差がでないものの，比較ブロックが大きければ速度は速いことがわかる．

#### 4.2.2　区　間　2

区間 2 は，携帯端末上のハッシュ値をサーバに送付する．よって，シリアルの通信速度とハッシュ値ペアの数に依存して次式で示される．ハッシュ値ペアのデータ量サイズは比較ブロックのサイズから決まる．図 9にブロックサイズごとの更新パタン $B$ でのダウンロード時間比較を示す．比較ブロックが大きいほどハッシュ値転送時間が少なくて済むことがわかる．




PC: WindowsXP, AMD athlon XP2800+2.80GHz448MB RAM

**図 10**　圧縮時間

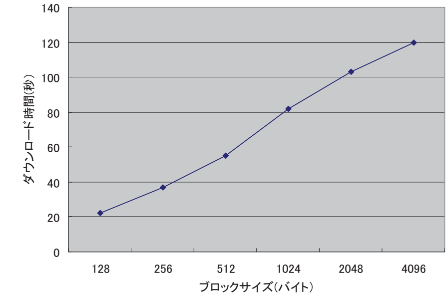



**図 11**　ダウンロード時間

#### 4.2.3　区　間　3

区間 3 は一致ブロック検索，および差分情報作成である．この動作はサーバ上で動作させるため十分高速と考えて良い．実際に時間を測定しても 5 秒程度であり，区間 1, 区間 2 に比べて時間が短く，この区間を分析し高速化を図っても全体に対する影響は小さいため，高速検討の対象外として良い．

#### 4.2.4　区　間　4

区間 4 は圧縮時間である．貧弱なリソースしか持たない携帯端末上での解凍時間を考慮し，解凍時間が高速であると言われる byte-pair 圧縮[13)14)] を利用して評価した．ただし， BPE は圧縮に時間がかかるため圧縮時間と展開時間のバランスを考える必要があるため，実際にサーバ上で計測した結果を図 10に示す．ブロックサイズは小さい方が良いのは，ブロックが大きいほど一部でも異なると違うものとみなされるのに対して，ブロックが小さいほど一致する領域が増えるため，差分サイズが小さくなるからである．実行時間はサーバの能力にもよるが，全体から図7にも示されるように，それほど時間がかからないため高速化検討の対象外として良い．

#### 4.2.5　区　間　5

区間 5 はダウンロード時間であり，差分サイズに依存する．従って，比較ブロックが小さいほど良い．実行時間を図 11に示す．差分サイズの小さくなる比較ブロックが小さい場合が良いこ

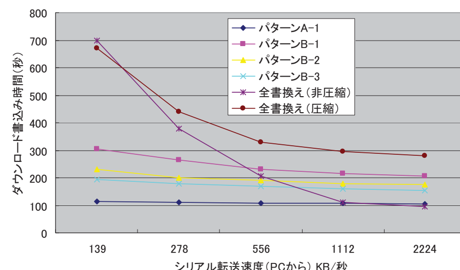



**図 12** シリアルの通信速度による書き込み時間の変化

とがわかる．また全体の実行時間に占める割合も大きい．ここは工夫により小さくするべきである．ただし，ダウンロード時間はシリアルの通信速度に依存する．

#### 4.2.6 区 間 6

区間 6 は差分適用処理である．メモリ上のみを一度走査しながら適用していく操作であり，実測値では悪い場合でも 7に示されるように 20 秒以下で全体から見ると工夫しても影響が少ない処理である．

#### 4.2.7 区 間 7

区間 7 は書き換え処理である．本質的には書き換え処理は比較ブロックサイズには影響を受けない．比較ブロックサイズよりむしろ更新パターンとソフトウェア構成法の影響を受ける．時間も更新パターン $B$ で 50 秒程度はかかるため，なるべくソフトウェアの書き換えの発生する部分を小さく押さえることが重要となる．

### 4.3 処理全体からの分析

処理時間を比較するため各区間ごとの所要時間をまとめた図 7より，区間 1, 区間 5, 区間 7 が，どんあ場合でも 20 秒以上要しているため，重点的に時間削減することが処理全体の速度向上に効果的である考えられる．また，区間 2 も削減のための比較ブロックのサイズによっては問題となる．

## 5. 性能改善検討

### 5.1 区間 1 の高速化

区間 1 はハッシュ計算であり，この処理時間を短縮するためにはハッシュ値を ROM イメージ内に保持すれば良い．こうするとハッシュ計算が不要になりハッシュ値を ROM に読み込むだけですみ，高速化が十分可能と考える．デメリットとしては毎回バイナリイメージ生成時に同時にハッシュ値を計算をして書き込む必要があることと，ROM イメージが約 1M バイト程度大きくなることである．この処理はサーバ上での計算となり，実際には 5 秒もかからなかった．そのため，デメリットは ROM イメージの大きさのみである．これは製品に依存する．メモリは調達する際には一定の大きさでしか調達できないため，その程度の余裕がある場合は多いと考える．

また， 5 秒程度で計算できるとはいえ，通常の開発環境でのリンクの後にツールを利用してハッシュ計算をした値を実行イメージに入れる必要がある．ツールの作り方にも寄るが開発者の操作ミスや手間となっては問題である． 2 分程度の処理を省くことが目的で，そのために操作ミスの可能性が大きくなってはならないため，単純に操作できるツールであるべきで，処理が単純なため，十分実装可能な範囲と考える．

このように開発環境上で，ハッシュ計算をすることにより，区間 1 の実行時間は次式となる．

$$T_1 = max(D_1(b), S_1) \quad (4)$$

$$= S_1 \quad (5)$$

ここで，サーバ上での動作はほとんど 0 に近いため無視できる程度となる．

### 5.2 区間 5 の高速化

区間 5 はダウンロード時間であり，差分データの大きさに依存する．ソフトウェア構成の手法によって差分がなるべく発生しないような工夫である程度小さくできると考える．一方，シリアルの通信速度に依存する面も大きい．そこで，シリアルの通信速度が変わった場合の挙動をシミュレーションにより調べた結果を図 12に示す．

更新パターンの差分ファイルの他に全書換えを前提に，全ファイルを圧縮してダウンロードする場合，と非圧縮でダウンロードする場合とを比較のために加えた．シリアルの通信速度が速くなると圧縮せずに全ファイルを転送する方が早くなることも考えられるためである．転送速度が速くなることにより，メモリ上での展開時間の方がより問題となるためである．

結果として図 12によればある程度より早くなると圧縮せずに全情報をダウンロードするほうが良くなる．つまり，本方式のような差分による転送は不要となり，メモリへの書き込み時間で抑えられる．しかし，携帯端末では実効速度としてそれほどシリアルの通信は早くないのが実情であり，我々の提案方式は将来的には不要になるかもしれないが，現状では有効であることがわかる．ソフトウェア構成法に関して後述する．

### 5.3 区間 7 の高速化

区間 7 はフラッシュメモリへの書き込み時間である．これは本質的に変わった部分がどの程度あるか，また変わってない部分をいかに書換えないかで性能が決まる．具体的には消去の手間が

表 4　更新パターン A

| パターン名 | 版の内容 |
|---|---|
| A-1 | 従来の A と同じでずれはない |
| A-2 | 従来の A の修正点の直後に 16 バイトダミー追加 |

表 5　更新パターン B

| パターン名 | 版の内容 |
|---|---|
| B-1 | 従来の B にアドレスの小さいところに 16 バイトダミー追加 |
| B-1-1 | 従来の B にアドレスの小さいところに 1K バイトダミー追加 |
| B-1-2 | 従来の B にアドレスの小さいところに 64K バイトダミー追加 |
| B-2 | 従来の B のアドレス上中央部に 16 バイトダミー追加 |
| B-3 | 従来の B のアドレス上後方部に 16 バイトダミー追加 |

ROM イメージサイズはどの場合も約 90Mbyte

表 6　更新パターンの更新ブロック数

| パターン名 | 更新ブロック数 |
|---|---|
| A-1 | 14 |
| A-2 | 409 |
| B-1,B-1-1,B-1-2 | 699 |
| B-2 | 503 |
| B-3 | 409 |

ブロックサイズ 128K バイト

大きいため，消去ブロックがどの程度あるかで決まる．大きくは，ソフトウェア構成手法に依存するため，区間 5 の高速化とあわせて，次節で検討する．

## 6. ソフトウェア構成による影響

ソフトウェアの構成法によっては一部の修正でバイナリイメージ全体がずれることがある．例えば，単純に ROM 上の配置を上から詰める形では修正によりコードが途中で追加されてしまうと残りの部分は単純にずれてしまう．たとえ，リンク情報がない画像イメージだとしてもメモリ上の書換えは発生する．

そのため，局所の修正は局所に留めるようなソフトウェア構成法が必要となる．文献[11] に示されるように，一定の余裕領域をプログラム中に埋め込むことにより，この問題は解決可能である．しかしながら，こういった工夫はツールを利用して出荷間際にフラッシュメモリの余裕の状態を見極めつつソフトウェアの配置を決めればできるものであり，デバッグ，試験といった繰り返しのフェーズで開発者に考えさせることはできるものではない．

そこで，リンカなどに付属する ROM のアラインメント機能を活用することでもちょっとした修正には有効であると考える．この手法は，一方では ROM イメージを大きくするというマイナス要素もあるがわずかであると考える．そこで，前記の更新パターン $A, B$ にさらに手を加えて，表 4,5に示す位置ずれの有無や位置ずれの発生場所の違う更新パターンを作成し差分サイズや不一致となるデータのサイズや実行時間の評価を行った．表 6にデータの特性を示す．

### 6.1 位置ずれ有無による影響

位置ずれの有無による影響を調べるため，更新パターン $A-1$ （位置ずれなし）と $A-2$(位置ずれあり）を利用して不一致と認識したサイズ，差分サイズの圧縮前と圧縮後および区間 3 の探索時間を測定した．測定環境としては全体を 16Mbyte 単位に分けてその間での比較を繰り返した．ブロックサイズは 128 バイトから 32K バイトを利用した．図 13,14にその結果を示す．

本質的に不一致のサイズと計算してできた差分（非圧縮）にはほとんど差がなく，差分を表現するためのコマンド情報などは誤差の範囲であることがわかる．比較のブロックサイズを大きくすると差分情報が大きくなりこの観点からは比較ブロックは小さい方が良い．位置ずれの有無による差は図 13,14から桁が違うため，非常に大きい．なお，区間 3 における計算時間は，位置ずれのない場合で 2 秒以下． 位置ずれのある場合でも 5 秒以下であり，計算時間は気にする必要はない．

差分サイズはわずかでもバイナリイメージに位置ずれがあると大きく影響することがわかる．前述のとおりツールなどで位置調整するのは煩わしい．しかし， 16 バイト程度のずれでも大きい差分となっており，逆に言えば，わずかなずれを防ぐ ROM のアラインメント機能などの活用でも十分効果があることがわかる．即ち障害の修正がわずかな場合には ROM のアラインメント機能が有効である．しかし，繰り返しの修正の場合では，急激に差分サイズが増加する場合も考えられる．小さいな不具合を修正して検証することが目的であり，何度も同じ部分の書換えが発生することはあっても，書換え対象が，徐々に大きくなっていくことは考えにくい．想定するモデルは一定間隔で開発者全員が自分の不具合を修正し，バージョン管理を行うサーバにコミットし，そこから最新版をダウンロードし，そのイメージに対して不具合修正を行うことであるため，次々に修正が溜まることはないと考えられるため，有効であると考える．

### 6.2 位置ずれ発生箇所による影響

次に位置ずれの発生箇所の違いによる影響がどの程度か調べた．図 15に，更新パターン $B-1$(修正箇所がアドレス空間の前方にある)， $B-2$(修正箇所がアドレス空間の中央にある)， $B-3$(修正箇所がアドレス空間の後方にある) における不一致サイズをブロックサイズに応じて示した．これらの場合の差分サイズはパターン $A$ での比率とほぼ同じ傾向を示し，区間 3 の探索時間は長くても 12 秒であり無視できる範囲であった．障害の修正位置の影響は大きく差分サイズに影響することがわかる．また，図 16にフラッシュメモリの書き換え時間を比較ブロックサイズごとに示す．アドレス空間後方の修正のため，位置ずれの起きない場合は影響が小さい．即ち，位置ずれが発生する場所の影響は大きく，メモリの前方の修正は避けるべきであることがわかる．

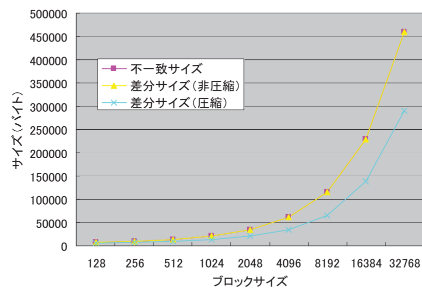



**図 13** 更新パターン A-1 のデータサイズ

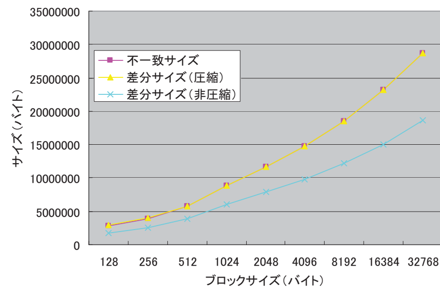



**図 14** 更新パターン A-2 のデータサイズ

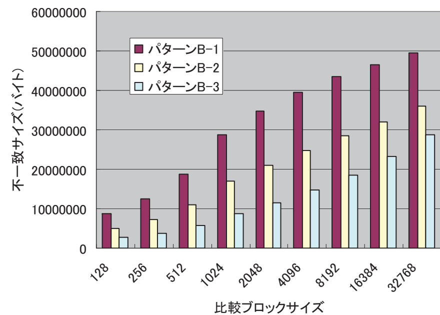



**図 15** 位置ずれ発生位置の影響 (差分サイズ)

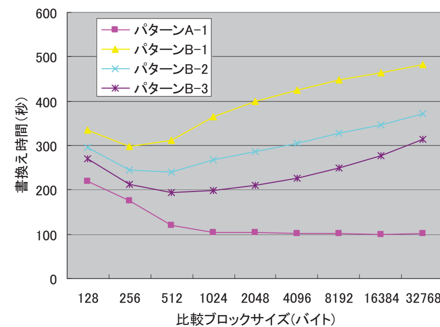



**図 16** 位置ずれ発生位置の影響 (書込み時間)

従って，試験において配置位置に制約がない場合は，開発者は自分用のリンカーでは配置をなるべく ROM のアドレス空間の後方に配置することにより影響を抑えられることがわかる．

### 6.3 分割単位による影響

本方式の適用にあたり，バイナリプログラム全体をファイルと考えるのは RAM の関係からも妥当でなく，適当な大きさのブロックに区切って比較の単位とするべきである．この分割の単位の影響を調べた．分割する単位による影響がどの程度異なるかを示すために，図 17に差分サイズを，図 18に区間 3 である差分検索時間を示した．位置ずれ幅の大きさで比較するために表5に示した更新パタン $B-1, B-1-1, B-1-2$ を利用して調べた．

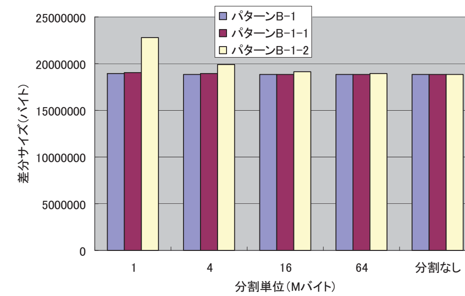



**図 17** 分割単位の差分サイズへの影響

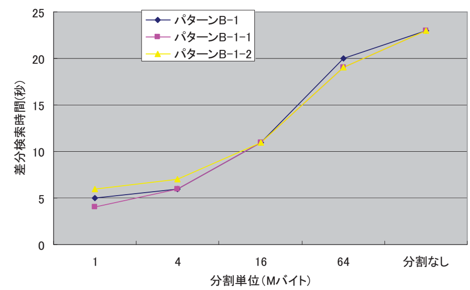



**図 18** 分割単位の差分検索時間への影響

結果からは位置ずれ幅が大きく，分割単位が小さい場合に差分が大きくなるものの，対象とする試験フェーズ程度の位置ずれでは分割単位の影響は差分サイズにはほとんどないと考えられる．

逆に不一致箇所の検索時間には大きく影響するため，ある程度分割単位は小さくても良いと考える．

一方，フラッシュメモリの書き換え時間を表 19に示す．ここで，位置ずれ幅の大きい場合のみ分割単位を 1M バイトとした場合に書き込み時間が大きくなっているが，これは差分サイズが大きくなっている分の影響を受けているためと考えられ，この場合に限り 16M バイトの分割が良くなるが，ここで想定するような試験サイクルでは分割単位は 16M バイト以下であれば書き込み時間と差分の検索時間を加えてもそれほど時間に差がでないと考える．

### 6.4 性能改善のまとめ

性能改善のためのポイントを以下にまとめる．

(1) 位置ずれが発生しないようにソフトウェアを構成すること．しかし，位置ずれの発生を予測することは無理である．そのため，位置ずれが発生したとしても影響範囲が小さくなるように自分の担当モジュールなどは ROM 上の後方に配置しておくこと．また，リンカなどによるアラインメント機能は活用し，最低限の位置固定の努力はしておくことが望ましい．

(2) 比較ブロックサイズを適切に決めること．比較ブロックサイズを開発者がダイナミック

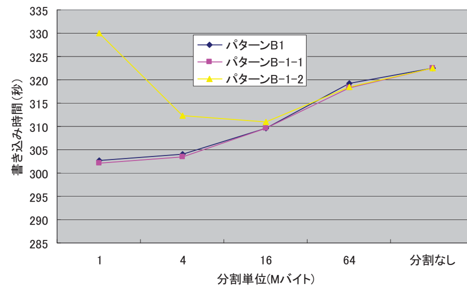



**図 19** 分割単位の書き込み時間への影響

に決めることは難しい．従って予め適切な値に固定することが望ましいが，場合によってその適切な値は変わる．そこで，ここで利用したＣＰＵなどの環境では，多くの場合は早いが，最悪ケースは遅くなっても良いのであれば，なるべく比較ブロックは大きくし，16K バイト程度が良い．しかし，最悪ケースでもそれなり速度を期待するのであれば，256 バイト程度の小さな値にすると良い．バランス的なものを考慮すると 512 バイトから 1K バイト程度が妥当であると考える．

## 7. おわりに

携帯端末の開発フェーズ終盤での実機ハードウェアへのソフトウエアダウンロードの高速化手法に関して提案，評価した．提案手法は，携帯端末などにおけるファイル同期化手法の中で有効な手法である rsync のアルゴリズムをベースにした既存手法と，ハッシュ値などを予め計算すること，ROM アラインメント機能などを利用するとと，修正のターゲットとする自己のモジュールをアドレス空間の後方に配置するという手法を適用することで性能向上があることを示した．

サーバ側にハッシュ計算などのツールを使うことがあるのであれば，修正内容を調査し，影響範囲まで調べるツールの導入も考えられる．即ちバージョンは管理しないが想定したバージョンが携帯端末に入っていれば高速に書き換えられるが，そうでなくとも正しく書き換えられるという方法の導入も考えられる．

今後，このような手法の導入と実際にそれで短くなる時間との効果を検討しながら開発を進めるべきと考える．

## 参 考 文 献


1) Cusumano M., MacCormack A., Kemerer F. Chris and et al.: Software Development Worldwide: The State of the Practice, IEEE Software, Vol.20,No.6,pp.28-34(2003)
2) Mellor J. S. and Balcer M., "Executable UML: A Foundation for Model- Driven Architecture," Addison-Wesley, 2002.
3) Mellor J. S. and Clark N. A., and Futagami T., "Model-Driven Development," IEEE Software, Vol.20, No.5, pp.14-18, 2003.
4) Kiyohara R., Mii S., Tanaka K., et al., "Study on Binary Code Synchronization in Consumer Devices," IEEE on Transs.CE, Vol.56, Issue 1, pp.254-260(2010)
5) Kiyohara R., Kurihara M., Mii S., et al., "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.
6) Terazono K. and Okada Y.: An Extended Delta Compression Algorithm and the Recovery of Failed Updating in Embedded Systems, Proc. IEEE Data Compression Conference 2004,p.571(2004)
7) Tridgell A. and MacKerras P., "The rsync algorithm," Technical Report TR-CS-96-05, Australian National University, 1996.
8) Rsync: http://rsync.samba.org
9) Pamta K. R., Bagchi S., and Midkiff P. S., "Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation," Usenix 2009
10) Pamta K. R. and Bagchi S., "Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks, " Proc.IEEE INFOCOM 2009, pp.639-647, 2009
11) 清原良三，栗原まり子，古宮章裕ほか: 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).
12) Irmak U., Mihaylov S. and Suel T., "Improved Single-Round Protocols for Remote File Synchronization," IEEE Infocom Conference 2005, Vol.3, pp.1665-1676, 2005
13) Gage P., " A New Algorithm for Data Compression, " The C Users Journal, Vol.12, No.2, (1994), 23.38.
14) Shibata Y., Matsumoto T., Takeda M., et al., " A Boyer-Moore type algorithm for compressed pattern matching, " Proc. 11th annual Symposium on Combinatorial Pattern Matching, Vol.1848 of Lecture Notes in Computer Science, (2000), 181.194.