情報処理学会論文誌　コンシューマ・デバイス & システム　Vol.2　No.1　94–103 (Mar. 2012)

研究論文

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,a)]　田中 功一[1]　寺島 美昭[1]

受付日 2011年5月20日, 採録日 2011年12月2日

概要：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，動作確認の繰返しになる．本論文ではこの繰返しの中で，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．

キーワード：コンシューマデバイス，ソフトウェア更新，rsync，実行イメージ同期

A Method of Software Updating for Consumer Devices

Ryozo Kiyohara[1,a)]　Koichi Tanaka[1]　Yoshiaki Terashima[1]

Received: May 20, 2011, Accepted: December 2, 2011

Abstract: 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.

Keywords: consumer devices, software updating, rsync, code synchronization

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ことも多い．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，たとえば文献 [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，これらの手法で障害を完全に排除できるわ

[1] 三菱電機株式会社情報技術総合研究所
Information Technology R&D Center, Mitsubishi Electric Corporation, Kamakura, Kanagawa 247–8501, Japan
[a)] Kiyohara.Ryozo@ah.MitsubishiElectric.co.jp

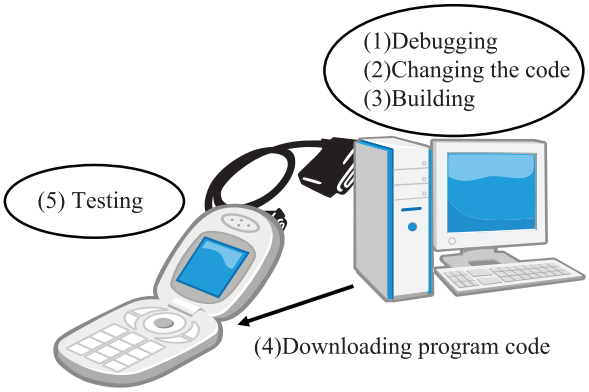

図 1　デバッグ，試験，確認サイクル
Fig. 1　Iterative steps of bug fixing just before shipment.

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，つねに効果がでるわけではないことが分かった．

そこで，本論文では，既存の手法を複数の角度から検証し，改良点を絞ったうえで，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．

ソフトウェアのバージョンは一定の動作確認を得ているものを構成管理ツールに登録して管理する．開発者が修正をして動作確認ができてないコードを構成管理ツールに登録して管理することは手間でもある．また，開発者が並行して作業するため開発者ごとの固有のバージョンまで発生することになり間違いの元にもなる．

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．rsync 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とる技術である．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，実際にどうパラメー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うえで，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

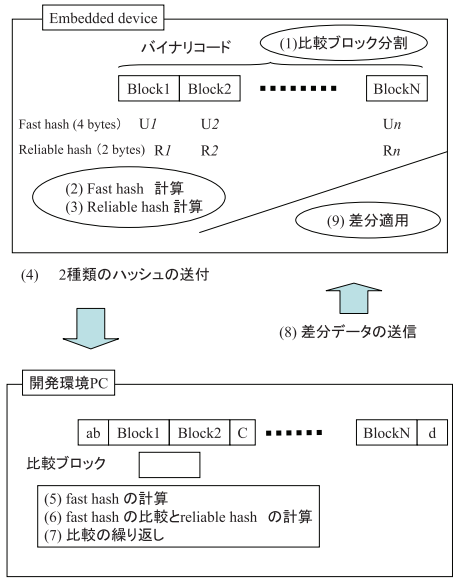

図 2　差分抽出アルゴリズム
Fig. 2　Algorithm of delta calculation.

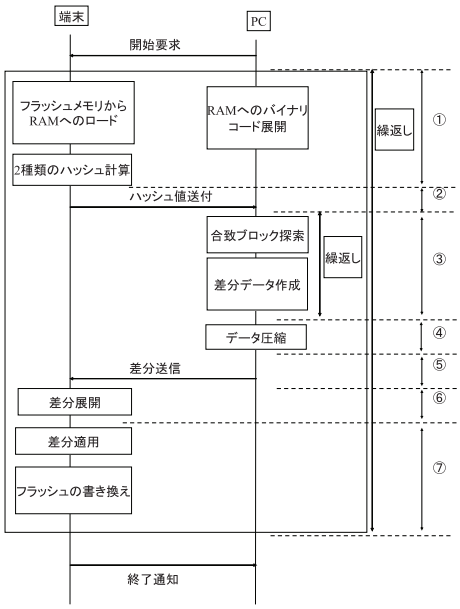

図 3　差分抽出処理シーケンス
Fig. 3　Flow of calculation of delta.

体を分割する．

(2) ブロックの同一性を確認するためにハッシュ値を計算する．このハッシュを FastHash と呼ぶ．

(3) 次に FastHash が一致した場合に別のハッシュ関数で確認するために，別のハッシュ関数を用いてハッシュ値を計算する．この確認のためのハッシュを ReliableHash と呼ぶ．

(4) 計算したハッシュ値のペアをすべてサーバ側に送る．

(5) 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．始点，終点を少しづつずらしなら計算する際に，何度も同じデータを読んではハッシュを計算するため，FastHash は高速性が必要になるため FastHash と呼ぶ．フラッシュメモリは，**図 4** に示すように，いったんフラッシュメモリのセクタ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すで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．したがって，書き換えを実施してないはずの範囲に限って行う．

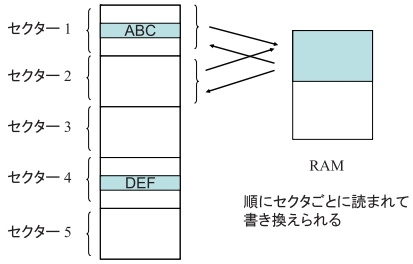

図 4　フラッシュメモリの書き換え
Fig. 4　How to rewrite flash memories.

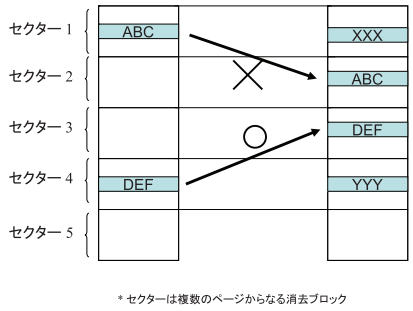

図 5　書き換え制約
Fig. 5　Constraint of rewriting flash memories.

(6) 前記探索で，携帯端末上のブロックと同じ FastHash となった部分に関しては ReliableHash も計算し，同じかどうかを判定する．ここで同じでないと判断できた部分に関して携帯端末側に送るデータの候補とする．

(7) 前 (5)，(6) の工程を繰り返して，全体として送付の必要なデータの候補を抽出する．

(8) 携帯端末側に送信する差分データを圧縮，送信する．

(9) 携帯端末側に差分データを解凍し，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 [11] で一般論として検証し，文献 [4] でも議論しているように，ハッシュ衝突の可能性は，ブロックサイズとハッシュに利用するビット数に依存する．すなわち，もし $1/2^d$ よりも衝突確率を小さくしたいならば，ビット数とブロック数の関係を次式のようにすることにより実現できる．

$$Bits = log_2(n) + log_2(n/b) + d \qquad (1)$$

表 1　評価環境の性能データ
Table 1　Time of total downloading.

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

ここで，n がデータのサイズで，b がブロックのサイズとなる．この関係は，文献 [4] で分析され，ほぼ 50 ビット近くあれば，0.1%以下に抑えられる．本論文での実験では，64 ビットで実装しているため実質上は競合が起きないと考える．しかし偶然競合が起きるケースもあるため，本方式は開発環境の中だけで利用すべき方式ということができる．

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ータを調整すれば最適なのかがこれだけでは不明である．

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ータが決まる．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 の所要時間を T_i とする．ここで，各区間ごとの合計すなわち区間 7 までの全ダウンロード時間は以下で示される．

$$DownloadTime = \sum_{i=1}^{7} T_i \qquad (2)$$

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．

表 2　評価対象データの特性
Table 2　Time of each component.

パターン名	更新消去ブロック数
A	14
B	409

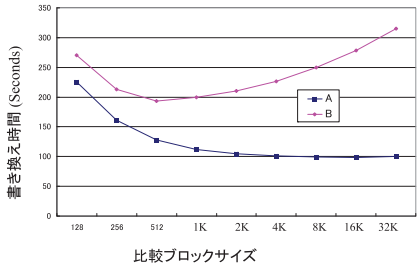

図 6　トータルダウンロード時間
Fig. 6　Total downloading time.

表 3　性能データトレードオフ
Table 3　Trade off.

		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

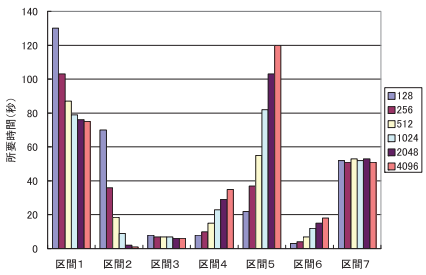

図 7　区間ごとの所要時間
Fig. 7　Time of each component.

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

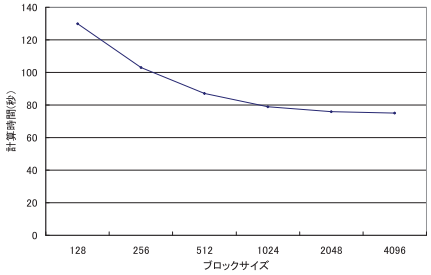

図 8　ハッシュ値計算性能
Fig. 8　Performance of hash calculation.

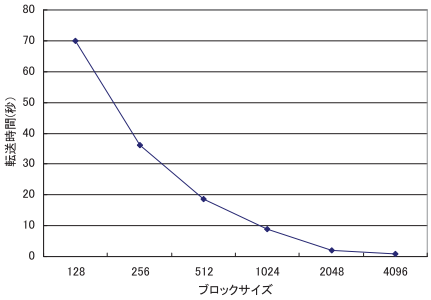

図 9　ハッシュ値転送時間
Fig. 9　Time of sending the hash value.

が大きいと想定される．したが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 をブロックサイズとする．その区間ごとのデバイス側所要時間を $D_i(b)$，サーバ側所要時間を S_i とする．

$$T_1 = \max(D_1(b), S_1) \tag{3}$$
$$= D_1(b) \tag{4}$$

ハッシュ計算は比較ブロックの大きさ，すなわち，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分かる．

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．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ーン 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すむことが分かる．

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よい．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よい．

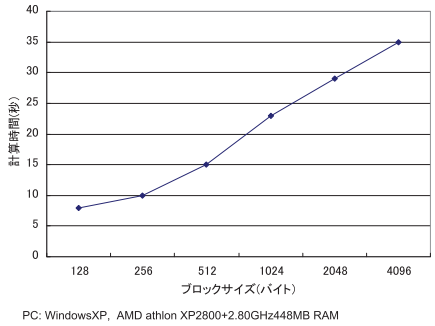

図 10　圧縮時間
Fig. 10　Time of compression.

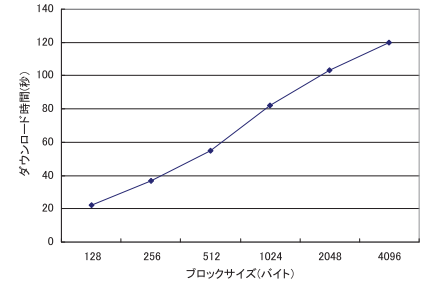

図 11　ダウンロード時間
Fig. 11　Time of downloading.

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いわれる byte-pair（BPE）圧縮 [12], [13] 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見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よい．

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．したがって，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とが分かる．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 1 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図 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お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，どんな場合でも 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フラッシュメモリイメージ内に保持すればよい．こうするとハッシュ計算が不要になりハッシュ値をフラッシュメモリ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，フラッシュメモリイメージが約 1 M 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，デメリットはフラッシュメモリ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機器に依存してあらかじめサイズが決まるため，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．

$$T_1 = \max(D_1(b), S_1) \tag{5}$$

$$= S_1 \tag{6}$$

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．

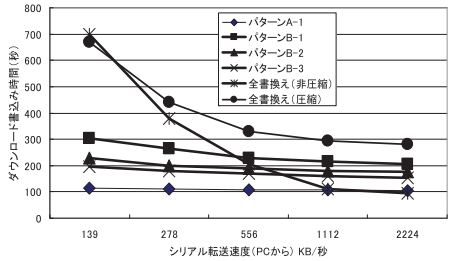

図 12　シリアルの通信速度による書き込み時間の変化

Fig. 12　Rewriting time according to serial communication speed.

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ほかに全書き換えを前提に，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速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速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速くなく，実測値で 256 kbps の機種もあるのが実情であり，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分かる．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き換えないかで性能が決まる．具体的には消去の手間が大きいため，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，次章で検討する．

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．たとえば，単純にフラッシュメモリ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

表 4　更新パターン A
Table 4　Upgrade pattern 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
Table 5　Upgrade pattern 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 1 K 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 64 K 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

フラッシュメモリイメージサイズはどの場合も約 90 Mbyte

表 6　更新パターンの更新ブロック数
Table 6　Number of upgrade blocks for each upgrade pattern.

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 128 K バイト

してもメモリ上の書き換えは発生する．

そのため，局所の修正は局所にとどめるようなソフトウェア構成法が必要となる．文献 [14] 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フラッシュメモリ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，一方ではフラッシュメモリ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**，**表 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．

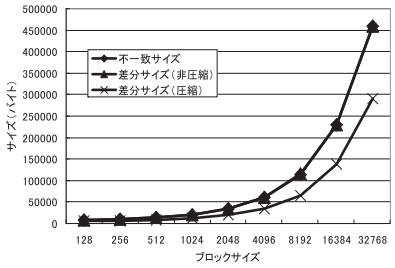

図 13　更新パターン A-1 のデータサイズ
Fig. 13　Data size of upgrade pattern A-1.

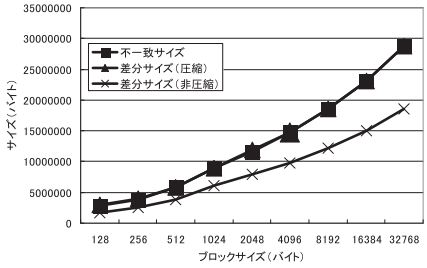

図 14　更新パターン A-2 のデータサイズ
Fig. 14　Data size of upgrade pattern A-2.

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 16 Mbyte 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 32 K バイトを利用した．**図 13**，**図 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分かる．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，図 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分かる．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，逆にいえば，わずかなずれを防ぐフラッシュメモリ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分かる．すなわち障害の修正がわずかな場合にはフラッシュメモリのアラインメント機能が有効である．しかし，繰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き換えが発生することはあっても，書き換え対象が，

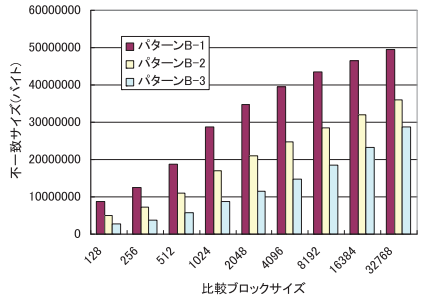

図 15　位置ずれ発生位置の影響（差分サイズ）
Fig. 15　Impact of posision sliding for data size.

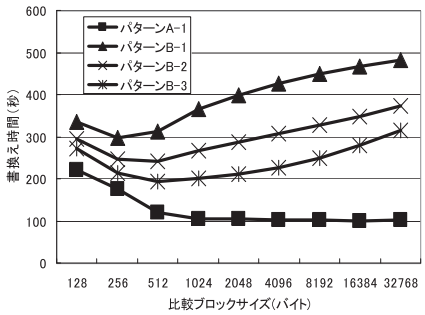

図 16　位置ずれ発生位置の影響（書き込み時間）
Fig. 16　Impact of posion sliding for time of rewriting.

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分かる．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．すなわち，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分かる．

したがって，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フラッシュメモリ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分かる．

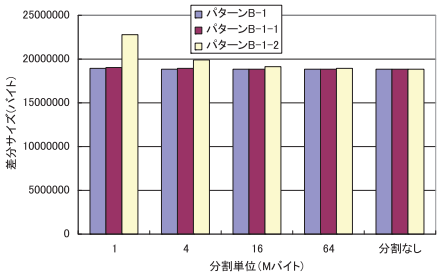

図 17　分割単位の差分サイズへの影響
Fig. 17　Impact for delta size according to unit size.

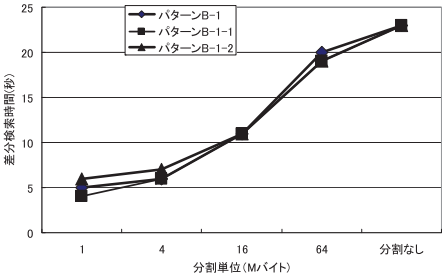

図 18　分割単位の差分検索時間への影響
Fig. 18　Impact for search time accoding to unit size.

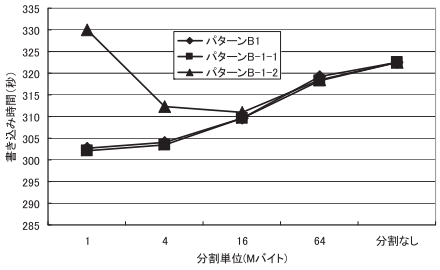

図 19　分割単位の書き込み時間への影響
Fig. 19　Impact of rewriting time according to unit size.

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 に示した更新パターン *B*-1，*B*-1-1，*B*-1-2 を利用して調べた．

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よいと考える．

一方，フラッシュメモリの書き換え時間を**図 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 1 M 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 16 M 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 16 M 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フラッシュメモリ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に決めることは難しい．したがって，あらかじ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，多くの場合は速いが，最悪ケースは遅くなってもよいのであれば，なるべく比較ブロックは大きくし，16 K 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 1 K バイト程度が妥当であると考える．

6.5 開発工程全体への影響

携帯端末などソフトウェアの開発では開発環境は使いやすいシミュレータ環境におき，実機において動作確認することが多い．多くのソフトウェアバグはシミュレータ環境で取り除くことができる．しかし，多くの状態を持ちイベントドリブンで動作することの多い携帯端末などではタイミングによって動作が異なるケースが多々あり，実機での動作検証が必須となる．この最後の工程においては必ず実機へのダウンロードという工程が入る．多くの場合，プロジェクト全体でソフトウェアのバージョン管理を行う．なんらかの構成管理ツールを使い，ソースコードと対応してバージョンを付与し管理する．

ここで，最終工程では毎日のようにこの構成管理ツールを利用して集積したモジュールをバージョン管理しながら試験者や開発者に対してリリースする．試験の結果，不具合が報告されると修正が要求される．修正して動作確認ができたら再度モジュールを構成管理ツールに投入して，その修正による他の機能への影響などの確認を行う．このときの自己のモジュールにおける確認では，構成管理ツールなどを独自に使うことは少ないと考える．また，1 行程度の修正で，まだ正しいかどうかを確認する前のものを使うということも考えにくい．

そこで，独自にコンパイルしてはチェックするフェーズが必ず存在する．たとえば，このような操作を 1 回の不具合に対して，5 回程度行うと仮定しても，前章の評価結果からすると，400 秒程度であり，全体をダウンロード時間が 1/3 以下に縮められることにより，開発の待機時間は，1 時間以上も減らすことに寄与することになる．

これが全開発者が対象となり，最終試験でのリリースの回数にも依存すると，最後のフェーズの開発者全体への待機時間をかなり抑えるという効果があることが分かる．特に，最近の携帯端末ではソフトウェアの規模も膨大になっており，そのダウンロード時間は増加の一途をたどっているためこの手法の採用による開発効率化への寄与は大きいと考える．

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に対して，ハッシュ値などをあらかじめ計算すること，フラッシュメモリアラインメント機能などを利用すること，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が向上することを示した．

今後，提案手法に加えて，さらに書き換え時間の短縮をする手法の検討と，そのコスト対効果を検討する予定である．

参考文献

[1] Cusumano, M., MacCormack, A., Chris, K.F., 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., 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 Trans. 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, available from ⟨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* (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] 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).
[12] Gage, P.: A New Algorithm for Data Compression, *The C Users Journal*, Vol.12, No.2, (1994), 23.38.
[13] 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.
[14] 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).

清原 良三　（正会員）

昭和 60 年大阪大学大学院工学研究科応用物理学専攻前期課程修了．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．組込み機器のソフトウェア更新，JavaVM 実装方式，組込み機器の信頼性，保守性向上に関する研究開発に従事．平成 20 年大阪大学大学院情報科学研究科博士後期課程修了．博士（情報科学）．ACM，IEEE 各会員

田中 功一　（正会員）

昭和 62 年金沢工業大学大学院工学研究科修了．同年三菱電機（株）入社．現在，同社情報技術総合研究所に勤務．情報ネットワークシステム構築と運用，ネットワーク応用ソフトウェアにおけるセキュリティ対策に関する研究・開発に従事．修士（工学）．

寺島 美昭　（正会員）

昭和 59 年埼玉大学工学部電子工学科卒業．同年三菱電機（株）入社．現在，同社情報技術総合研究所に勤務．分散システム構築と適合性試験，センサアドホックネットワーク，ネットワーク評価に関する研究・開発に従事．博士（工学）．電子情報通信学会会員．