Vol.2015-DPS-165 No.4
2015/12/10

車載 ECU 向けソフト更新のためのデータ圧縮方式

野澤 優尚[†1]　小沼 寛[†1]　清原 良三[†1]

概要：自動車がネットワークに接続されるいわゆる Connected Car が話題になって久しい．また，最近では自動運転の話題がつきずに，公道での実証実験もされるようになりつつある．一方，車がネットワークに接続されることによる脅威もある．盗難車防止のための遠隔制御の機能がハッキングされれば，遠隔で自動車を制御されるがゆえに自由に攻撃者から攻撃される可能性もある．多くの場合，システムの欠点をつく攻撃がされ，この欠点に対応するソフトウェアが配布されることになる．このようなソフトウェアの配布と修正を現在はリコールという形で実現しているが，自動運転などが実用化されてくると，緊急でソフトウェアの修正を必要とする場合も想定される．カーナビの地図の更新のように販売店や自宅で更新する必要がある．そこで本論文では，効率的な更新方式に関して検討した上で，従来手法である bsdiff を改良し，短時間でソフトウェア更新可能な手法を提案する．

キーワード：車載機器，バイナリ差分，ソフトウェア更新，ECU, CAN

1. はじめに

近年，自動車の ADAS（アドバンスド・ドライバー・アシスタンス・システム）や ABS（アンチブレーキロックシステム）に見られるように制御システムの電子化が進み，システムを制御するコントロールユニット，ECU（Electronic Control Unit）が増加している．最近の自動車は ECU を 50 近く[1]も搭載しており，複雑な処理を行うためにそのソフトウェアの規模が増大し，様々なタイミングでのイベント発生の可能性があるなど，ソフトウェアの出荷後の不具合が増加している．出荷前の欠陥の除去はもちろんのこと，出荷後の不具合の修正も重要となっている．

出荷後の不具合の例として，フィアットクライスラーの大規模リコール[2]があげられる．リコールの原因もソフトウェアのバグとされている．また，フォルクスワーゲンでのディーゼル車排出ガス不正問題がニュースとして話題になっている．違法ソフトが原因で大規模なリコールを余儀なくされている．

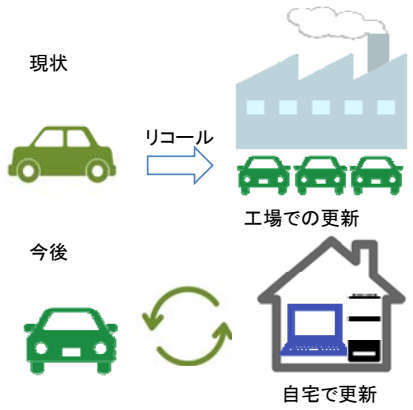

図 1 将来のソフトウェア不具合の修正

上記の例は２つともリコールであるが，例えば自動運転システムにおける障害など，緊急を要する場合は，図１に示すようにカーナビゲーションシステムにおける地図更新と同様に所有者の自宅や販売店でのソフトウェア更新が必要となる．

ソフトウェアの更新は，最近の一般ユーザでは携帯電話のソフトウェア更新や，PC のソフトウェア更新に接する機会も多くなじみのある機能であり，ECU のソフトウェアの規模は大きいといっても，携帯電話や PC のソフトウェア規模に比較すればはるかに小さい．また，ECU が 50 個近くあるとはいえ，トータルで見た場合も携帯電話などに比較すると小さい．

しかしながら，販売店や自宅での ECU のソフトウェアの更新には車載ネットワークの利用が不可欠であり，この速度が速くても 1Mbps であるため[3]，ブロードバンドを利用できるケースに比べ格段に遅く，効率的な配信を検討する必要がある．とくに自宅でのソフトウェア更新時にはエンジンをかけた状態で，駐車場での更新が想定されるためできるかぎり短い時間での更新が期待される．

そこで，本論文では効率的なソフトウェア更新方法を検討し，提案する．　以下，2 章ではソフトウェア更新の注意すべき点と関連研究を示す．3 章ではベースとなる技術であり，改良対象とする bsdiff に関して述べる．4 章で提案手法を述べ，5 章で考察し，まとめる．

2. ソフトウェア更新と関連研究

ソフトウェア更新システムを考える上で重要なのは対象とする製品のサービス停止時間を如何に短くできるかである．携帯電話や PC のソフトウェア更新時の動作を考えると，データのダウンロードはバックグランドで実施することが多くあまり気にならない．しかし，PC 上で起動や終了時に行う構成変更などの時間は，おそらく非常に気になるのではないか，このような点を如何に適切にするかが重要となる．以下にソフトウェア更新時に考慮すべき点を整理

†1 神奈川工科大学
Kanagawa Institute of Technology

列挙する．

(1) ソフトウェア構成
(2) ネットワークダウロード時間
(3) ソフトウェア書き換え時間

この 3 点が重要である．以下にそれぞれの関連研究とその課題を示す．

2.1 ソフトウェア構成

ダウンロード時間や書換え時間を考慮すると，ソフトウェアの不具合の修正には一定の制約を書けざるを得ないが，開発者がソフトウェア開発を行う際や，出荷後の不具合を改修する時に，ソフトウェア構成を気にしなくてはならないようではバグの温床になりかねない．そこで開発が気にすることはなく，構成管理において不具合の修正における影響を局所にとどめ，ソフトウェア全体におよぼす影響を抑える研究がある[4]．図 2 に示すようにアドレスの参照関係が変わるだけで何も修正をしていない場所が差分として現れる．そのため全体をモジュール分割することによりこの参照関係の問題を解決している．この研究の対象は従来型の携帯電話であり，大規模なソフトウェア開発で有効な手法である．本研究で扱うようなソフトウェア構成では ECU のフラッシュメモリのサイズは小さく，また，出荷後のソフトウェアは車載機では多くの場合部品供給メーカが異なるこの研究にあるようなソフトウェアのコンパイル時のシンボルを活用するような手法は使えない．

2.2 ネットワークダウンロード

ネットワークのダウンロード時間は最近の携帯電話や PC では全く気にする必要がない．その理由は，図 3 に示すように優先度を低くしてダウンロードすることと，ファイルシステムを構成する要素がハードディスクや，NAND 型のフラッシュメモリであるため，実行しているコードに影響を与えないからである．

一方，車載機器の ECU は NOR 型フラッシュメモリを採用していることが多い．起動が早く，それほど CPU の処理速度を要求しない場合に適している．

また，車載ネットワークの速度は遅いことと，異なった ECU には異なったデータを必要とする．そのため，データ量を削減する必要がある．そのための技術としては，携帯電話などに利用されている差分ダウンロード技術が有効である[5][6]．これらの研究では，新旧版の差分が出る原因を整理し，修正した部分以外の差分の多くの要因に配置が変わることにより参照アドレスの違いが大きいことを示している．そのため，それらの点を如何に小さな差分表現で表すかを工夫し，各種 CPU のコードで効果のある手法を示している．

また，同様に参照アドレスの違いが大きいことに着目するとともに，差分抽出した後に，データを圧縮することを想定し，圧縮に効果のあるような差分を抽出する方式も提案，実装されている[7]．この手法は非常に有効な手法であり，CPU として i386 を対象に開発され評価されている．

本論文ではデータの削減に基本手法として文献[7]に示された手法を活用し，さらにその圧縮効率を上げることを目指すこととした．

2.3 ソフトウェア書き換え時間

ソフトウェアの書き換え時間はサービスを停止する大きな要因となる．ソフトウェア更新でサービスを停止する時間 T とした場合，ネットワークの速度を V として書換える必要のある領域の大きさを S，差分データ量を D とし，書換え時間を R とすると以下の式で示すことができる．

$$T(D,S) = \frac{D}{V} + R(S) \qquad (1)$$

ここで，転送時間は差分のデータ量 D とネットワークの速度 V で決まる．書換え時間は書換え対象のサイズ S で決まるのでトータルのサービス停止時間 T は，書換え対象と差分データのサイズで決まるが，圧縮率が高ければ高いほどデータの転送時間を短くでき，ソフトウェアの構成などの

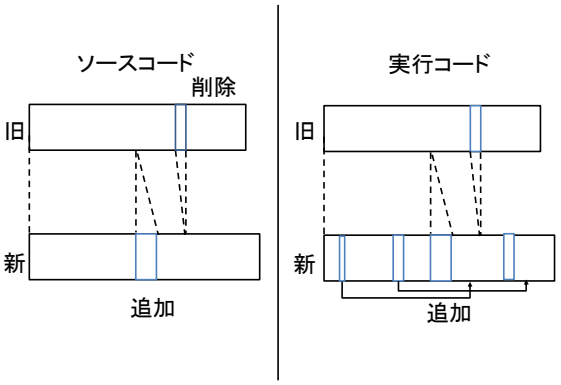

図 2　ソフトウェア差分の主要因

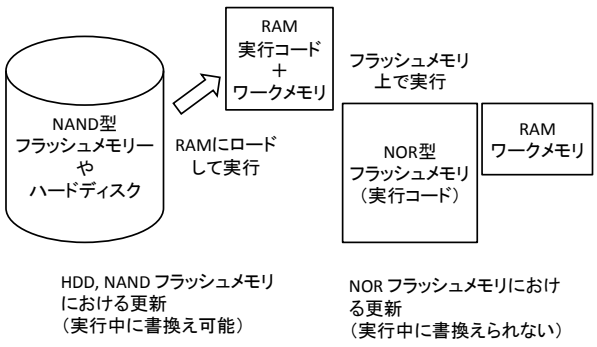

図 3　車載 ECU ソフトウェア更新の特徴

工夫により書換え領域を小さくできるほど R を小さくできる．

携帯電話や PC ではバックグラウンドでダウンロードするため単に書き換え領域のデータの大きさだけでサービス停止時間が決まるのに対して, 車載 ECU では対象となるプログラムデータが小さく, 複数の ECU があっても並行処理可能であるため，並行処理ができる形でのダウンロードプロトコルとネットワーク速度が遅いために効いてくるデータの圧縮率とが重要となる．

これらの点に着目した上で, データの参照関係に着目し，ソフトウェアの構成上の課題に着目した研究もある[8]．しかし，この研究は書換え中の処理制約条件を bsdiff への適用上の課題として示したものであり，圧縮率をあげるものではない．

3. bsdiff とその課題

3.1 bsdiff の処理方式

bsdiff は新バイナリファイルと旧バイナリファイルを比較し，差分を抽出するツールである．bspatch は bsdiff によって抽出された差分を使い，旧バイナリファイルを新バイナリファイルに更新する．

bsdiff は新旧のバイナリファイルを比較し，差分ファイルを生成する．図 4 に示すように新版は旧版を比較すると次の 3 種類の領域からなることがわかる．

(1) 全く同じ領域
(2) 基本的に同じであるが, 参照アドレスなどの直値部分とオペコード中レジスタアサインのみが違う領域
(3) 全く新しいコードが追加または削除されている領域

(1)の全く同じ領域のソースコードは変わっておらず，参照先なども全く不変な領域である．(2)の部分は，ソースコードを変更していなくとも図 5 に示すように参照先の配置アドレスが変われば変更されてしまう部分である．さらに同じ関数内などでもソースコードを一部修正することによりレジスタの使用方法が変わると同じ関数内で修正をしてない部分にまで影響がでる場合もある．こういった部分が該当する．(3)は追加や削除されたコードそのものである．

このような３つの領域に分けるにあたり，bsdiff ではまず完全一致の領域を探す. 最大一致する領域(LCS)を探すアルゴリズム[9]などでこの機能は実装可能である．

完全一致する領域以外が, (2)および(3)に該当する領域となる．(2)の領域の場合で，参照アドレスのみが変わる場合は，図６に示すように，命令コードの部分が一致し，その他の部分が一致しないことになる．バイナリコードレベルではこの違いを正確に判別するのは難しい．そこで，bsdiff ではこの違いをコードの中の 50% 以下の違いであればアドレス部が違う可能性が高いと判断し，50%以上であれば

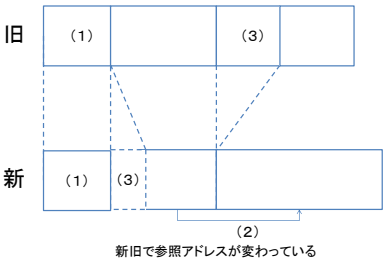

図 4　bsdiff における一致領域の探索

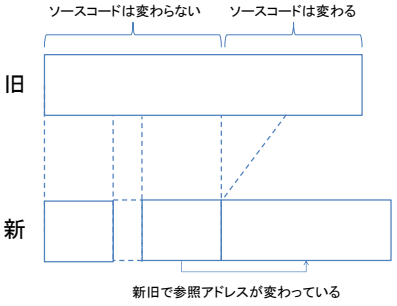

図 5　ソースコードの修正とバイナコードの関係

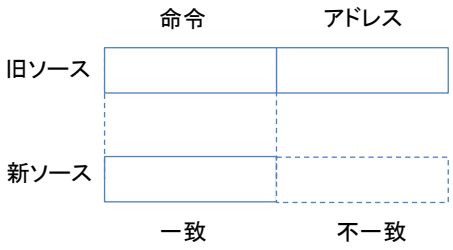

図 6 参照アドレスのみが変わる場合

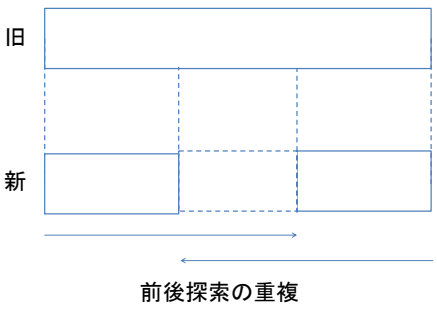

図 7　bsdiff の重複の探索

新規のコードであると判断する．

(2)のような領域は図 7 に示すように，通常完全一致する領域と隣接するため，その前後を探せば良い．従って図 7 のように完全一致する領域の間で以下に示すステップにて探す．50%をキーにして探す．

1. コードを前からの探索
2. コードを後からの探索
3. 前後探索により重なった部分の調整

このようにして，新旧のバイナリファイルを比較した場合，新旧のバイナリファイルは，3 つのバイト列に分割することができることになる．

最終的には，新版と旧版の対応する部分を引き算する．一致する領域部分は 0 になり，アドレス部分に差があればほぼ同じ値がでてくることになる．

これは非常に圧縮しやすいデータであり，このような値を汎用の圧縮方式である bzip2 で圧縮する．また，完全に違うコードとみなした部分に関しては引き算をとることなく，同じ圧縮方式 bzip2 で圧縮する．このようにしてできるコードの圧縮は関連研究[10][11]と同様の問題点を考慮した上での非常に効率的手法ということができる．

また bsdiff では LCS を探す上でも効率的な接尾辞配列を応用したアルゴリズムを採用しており，全体に高速に動作するため，有用なツールということができる．

3.2 bsdiff の課題

bsdiff は CPU として i386 を向けのアルゴリズムである．i386 は可変長の命令コードであるため，最初から逆アセンブルしないと全体の中でどこがアドレス部分かなどが判断できない．そこで，50% というしきい値を設けている．しかし，車載 ECU として搭載される組み込み機器向け CPU は固定長命令のアーキテクチャを持つケースが多い．また，コード全体がそれほど大きくないため，どこが直値かということも十分判定可能であると考える．

このように，bsdiff は固定長命令でかつ規模が小さめのソフトウェアのバイナリ差分を抽出するにはさらに効率の向上ができる余地が十分にある．

また，本来差分が出なくて良いはずのレジスタアサインのずれに関しては対処できておらず，ソースコードが変わってないにも関わらず，全体を書き換えたという扱いになる部分が少なからずあると考えられる．

4. 提案手法

3 章において bsdiff の課題として以下の 2 点を挙げた．

1. 固定長命令のアーキテクチャを対象としていない
2. レジスタアサインのずれで余計な差分が出る

上記の 2 つの課題を解決するために以下の手法を提案する．

4.1 固定長命令用改良

bsdiff は可変長の命令を対象とする．ソースの変更をしていないにも関わらずバイナリコードに新旧版の違いがあるのは参照先アドレスが変わることによる．しかし，不具合の修正が軽微であれば参照先のずれは一定の幅に決まっていることが多い．bsdiff はこの性質を利用して新版と旧版の引き算をすることにより同じパターンが出現してきて圧縮の効果を生んでいる．このアドレスの変化のみがある部分と，完全にソースを書き換えている部分を判別するにあたり 50%程度の違いおよび連続して違う情報が 8 バイト以上あるかどうかという 2 点から判断している．

図 8 に可変長命令における参照先アドレスのみが変わる

旧

アドレス	オペコード	アセンブラ
80484b4:	8b 45 f0	mov −0x10(%ebp),%eax
80484b7:	89 04 24	mov %eax,(%esp)
80484ba:	e8 a1 ff ff ff	call 8048460 <kei>
80484bf:	89 44 24 04	mov %eax,0x4(%esp)
80484c3:	c7 04 24 19 85 04 08	movl $0x8048519,(%esp)

新

アドレス	オペコード	アセンブラ
80484f0:	8b 45 f0	mov −0x10(%ebp),%eax
80484f3:	89 04 24	mov %eax,(%esp)
80484f6:	e8 85 ff ff ff	call 8048480 <kei>
80484fb:	89 44 24 04	mov %eax,0x4(%esp)
80484ff:	c7 04 24 89 85 04 08	movl $0x8048589,(%esp)

旧

アドレス	オペコード	アセンブラ
8048490:	e8 67 fe ff ff	call 80482fc <printf@plt>
8048495:	b8 00 00 00 00	mov $0x0,%eax

新

アドレス	オペコード	アセンブラ
80484a5:	e8 52 fe ff ff	call 80482fc <printf@plt>
80484aa:	8b 45 f0	mov −0x10(%ebp),%eax
80484ad:	03 45 ec	add −0x14(%ebp),%eax
80484b0:	89 45 e8	mov %eax,−0x18(%ebp)
80484b3:	b8 00 00 00 00	mov $0x0,%eax

図 8 可変長命令のアドレスのみが変わるケース（左），別のコードが入るケース（右）

旧

アドレス	オペコード	アセンブラ
8434:	e51b1014	Ldr r1, [fp, #-20]
8438:	e51b2018	Ldr r2, [fp, #-24]
843c:	ebffffe2	Bl 83cc <kei>
8440:	e1a03000	Mov r3, r0
8444:	e1a00004	Mov r0, r4

新

アドレス	オペコード	アセンブラ
8470:	e51b1014	ldr r1, [fp, #-20]
8474:	e51b2018	Ldr r2, [fp, #-24]
8478:	ebffffe2	Bl 8408 <kei>
847c:	e1a03000	Mov r3, r0
8480:	e1a00004	Mov r0, r4

旧

アドレス	オペコード	アセンブラ
8400:	ebffffba	Bl 82f0 <printf@plt>
8404:	e3a03000	Mov r3, #0

新

アドレス	オペコード	アセンブラ
8418:	ebffffb4	Bl 82f0 <printf@plt>
841c:	e51b200c	Ldr r2, [fp, #-12]
8420:	e51b3010	Ldr r3, [fp, #-16]
8424:	e0823003	Add r3, r2, r3
8428:	e50b3008	Str r3, [fp, #-8]
842c:	e3a03000	Mov r3, #0

図 9　固定長命令のアドレスのみが変わるケース（左），別のコードが入るケース（右）

場合と別のコードが入った場合の例を示す．

可変長命令であるため 8 バイト命令中の 4 バイトの参照アドレスが変わっていることがある．また，次の命令では 1 バイト命令, 3 バイト命令とばらばらになる可能性があるため 50%というしきい値で判断をしている．また，別のコードが 8 バイト追加される場合もあり，違う情報が 8 バイト以上あるかという判断している．なお，bsdiff は比較的サイズの大きなプログラムを対象としているため，参照先アドレスの変化も大きな場合があることも想定している．

しかし， 車載 ECU を対象とした場合，32bit で固定の CPU が使用されている場合がほとんどである．32bit で固定の CPU の場合は, 図 9 に示すように固定長命令であるためどのバイト数も全て固定されている．また，別のコードが追加されたとしてもバイト数は変わらない．そこで，可変長命令とは違い固定長命令であれば命令はバイト数が固定であるので，50%というしきい値，連続して違う情報が 8 バイト以上での判断では固定長命令には適していないのではと考える．そこで，50%というしきい値は 25%程度で十分であること．連続して違う情報 4 バイト程度以上のバイト数を変更することを提案する．

また，固定長命令であれば直値の判定がしやすいため，25% というしきい値を用いる探索ではなく直値を察する方法で行えるのではないかと考える．そこで，bsdiff のコード中の 25%の違いで判断し，探索するコードをオペコードからアドレスを判定するもので効率を上げることはできないかと考える．

4.2 レジスタアサインのずれの対処

レジスタアサインがずれる場合がある．図 10 に示すように新バイナリコードと旧バイナリコードとでソースは変わっていないが新しくソースに変数を扱うような行が追加されることにより，旧バイナリファイルで使っていたレジスタとは違うレジスタアサインになることがある．そのため，新バイナリファイルにおいてレジスタアサインのずれが発生し，差分として探索されてしまう．これはアセンブラレベルで言えば，命令は同じであるがレジスタが変わることを示す．レジスタのアサインの変化は，アドレス参照の変化のように，一定のオフセットを考えればいいわけではない．

そのため，bsdiff の方法で圧縮の効果のあるような差分にはならないケースが多いと考えられる．結果として差分

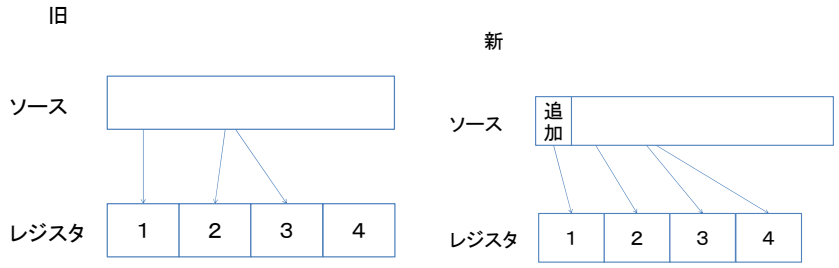

図 10　新旧におけるレジスタアサインのずれ

表 1　レジスタの置換え

命令セット	
変更前	変更後
命令　レジスタ1　レジスタ2	命令　レジスタ1　レジスタ2
命令　レジスタ2　レジスタ3	命令　レジスタ1　レジスタ2
命令　レジスタ3　レジスタ4	命令　レジスタ1　レジスタ2

データ量が増えてしまう．そこでレジスタアサインのずれに対処するために新バイナリファイルと旧バイナリファイルを同一のレジスタとなるように置換えることで効率を上げられるのではないかと考える．

表 1 に示すようにレジスタアサインのずれが一定であればそのずれに合わせてレジスタを元のレジスタに置き換える．実際のコードを置き換えるのではなく，置き換えた状態で差分を作成し，差分適用時に元に戻す．そのための情報は辞書として生成するため，トレードオフがあると考える．

5. 考察とまとめ

本論文では，ソースコード上で変更がないにも関わらず，バイナリコード上で差分が作成される原因を示し，これに対して効果的な手法として bsdiff を紹介した．しかし，bsdiff は i386 という可変長の命令コードを扱う CPU を対象しており，しかも比較的大規模なソフトウェアで評価を行い効果があることがわかったものである．そこで，差分にはなるが，本来ソースを変更していない部分も探索手法に関して，複数の手法を適用することを提案した．

固定長であり，比較的小規模なソフトウェアを対象とするため命令コード中の直値を対象にその一部のアドレスコードのみが変わっていることを想定した探索手法，およびレジスタのアサインの影響を考慮した，命令群としての差分の生成と元に戻す手法の提案を実施した．

効果を評価するために,実際の ECU の命令コードで今後その効果を示す．また，差分生成時には，繰り返し差分抽出処理を行うことにより，より詳細な分析を行い，より小さな差分が生成できるのではないかと考える．

また，32 ビット CPU でのロングジャンプなどにおいては直値でジャンプすることができないため，飛び先をメモリに配置する間接ジャンプを行うことも多い. この場合は，飛び先アドレスばかりが固まって配置されるケースも多いと考える．このような場合は，一定のずれのあるデータの塊となることが多くなるため，元の bsdiff の方が効率の良い可能性が高い．そこで，差分生成時に，このようなアドレステーブルを探す処理を前処理として挿入することも想定した実装を行う評価する予定である．

参考文献

1)　“2014 車載 ECU 関連市場の現状と将来展望,” https://www.fcr.co.jp/pr/15010.htm<accessed 2015/11/08>
2)　“Chrysler，車の遠隔操作問題で 140 万台のリコール発表,” http://www.itmedia.co.jp/enterprise/articles/1507/27/news038.html <accessed 2015/11/08>
3) 佐藤道夫, ”車載ネットワーク・システム徹底開設,” CQ 出版社, ISBN978-4-7898-3721-7
4) 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
5) 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
6) 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
7) Colin Percival, “Matching with Mismatches and Assorted Applications ,” doctoral thesis, Wadham College University of Oxford http://www.daemonology.net/papers/thesis.pdf <accessed 2015/11/08>
8) 施欣漢， 中西恒夫， 久住憲嗣， 福田晃,”放送による車載情報機器向けソフトウェア差分更新方式，“ 情報処理学会研究報告組込みシステム（EMB） 2011-EMB-20(23), 1-6, 2011
9) J.W. Hunt and T.G. Szymanski, ”A fast algorithm for computing longuest common subsequences,” Commun. ACM, vol.20, no.5, pp.351-353,1977
10)　Interface 編集部（編），“ ＡＲＭプロセッサ入門，” TECHI, Vol8, CQ 出版社,2003
11)　Interface 編集部（編），“ superH プロセッサ，”TECHI，Vol, CQ 出版社,1999