修 士 論 文

画質調整機能を持つプロキシサーバの キャッシュ置換に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

李　　奇

2007 年 3 月

修 士 論 文

画質調整機能を持つプロキシサーバのキャッシュ置換に関する研究

指導教官　井口　寧 助教授

審査委員主査　井口　寧 助教授
審査委員　松澤　照男 教授
審査委員　田中　清史 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

510112 李　　奇

提出年月: 2007 年 2 月

概 要

本研究では画質調整機能を持つプロキシサーバにおいて画質調整された前後の画像をキャッシュ空間に置き換えする際にネットワーク転送コスト、画質調整コスト及びアクセス率などの要素を考え、画質調整された前後の画像の影響関係を考慮した。影響関係を分析するためにキャッシュされたバージョンが生成バージョンと影響バージョンに定義した。まだ生成バージョンと影響バージョンからの節約コストを記録するためにパラメータ $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ を提案した。その二つのパラメータを用い、本研究では影響関係を構築するアルゴリズムを提案した。まだそのアルゴリズムの結果によって生成バージョンの画質調整コストを算出する関数を提案した。提案したアルゴリズム及び関数を NS2 に実装し、シミュレーションした。シミュレーションの結果から見ると本研究ではキャッシュされたバージョン間の影響関係を考慮することによって、正確にキャッシュされたバージョンの節約コストの計算ができ、算出したコストに基づき、節約コストが高いバージョンがキャッシュされると従来研究の AE アルゴリズムにより、全体の節約コスト率が高くなり、全体のキャッシュヒット率も高くなった。本研究の優位性を示した。

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景

現在、インターネットの普及とともに、インターネット上でのマルチメディアコンテンツの流通形態として、WWW（World Wide Web）が最も普及し広く利用される。インターネットの発達により時代はこれまでの一定の場所から固定端末による通信からいつでもどこでもなんでもつながるユビキタスネットワーク社会に移行しつつある。ユビキタスネットワーク社会においてはモバイル環境の利用ニーズは急激に増加しモバイル通信が重要な役割を果たしている。今日、携帯電話、PDA（携帯端末)、PHS、ノードパソコンなどのモバイル端末を利用したサービスが多く利用されている。文献 [1] によると 2005 年度末における携帯電話の加入者数は 9,179 万加入、対前年度比 5.5%増加した。PHS サービスの加入者数は 469 万加入、対前年度比 4.8%を増加した。このような状況のもと、特に企業においては、新聞、雑誌、テレビなどの従来の媒体による宣伝広告よりも効果の高い宣伝媒体として WWW をとらえ、積極的に WWW による情報提供を行っている。WWW はハイパテキストの概念に基づく HTML（HyperText Markup Language）を用いてコンテンツ記述を行うが、大量な静止画像やアニメーション画像などをページに埋め込むことにより、情報提供者は視覚的効果の高い情報を提供することができる。

モバイルネットワークの転送速度は文献 [2] により、携帯電話の転送速度が 28.8kbps（PDC）〜64kbps（cdmaOne）であり、PHS の転送速度は 32kbps〜128kbps であり、IMT-2000 は 144kbps（MC-CDMA)〜2.4Mbps（EV-DO）である。モバイルネットワークは広帯域化しつつあるが、有線インタネットの 10 倍以上のデータ到達遅延時間が発生する。通信環境に応じてこの到達遅延時間は揺らぎ、大きく変動する。また、基地局との距離や加入者数が連年増えるために同一基地局へ接続するユーザの数によっては帯域の著しい変動も発生する。そして、大量な静止画像やアニメーション画像などを埋め込まれるページでは配送すべきデータ量が多くなるため、モバイルネットワークのようなネットワーク帯域幅が狭い場合にはページ全体を転送するのに長時間がかかることになり、ユーザによってそのページ内容を見る前に転送を中止されてしまうことが多い。そこで、帯域の限られた状況でも効率的に配信するために、画像の品質を調整し、転送すべきデータ量を減らすことにより短時間内での Web ページデータの配送を可能にする研究が多くなされている [3][4][5][6]。具体的には帯域の限られた環境で、Web ページに埋め込まれた JPEG ファイル、GIF ファイルなどの品質を落とすことで効率的に Web ページを表示する。

一方、モバイル端末をはじめ、Web クライアントが多様化になっている。携帯電話な

どの場合には普通のパソコンより処理能力（CPU) が低いので、大量のデータにあたり、処理時間が長くなる。 画面サイズが小さくて（携帯、PDA) またキャリア（電話会社）ごと、利用可能な画像ファイルの形式が異なっているので、WWW にアクセスするときに表示できない静止画像も予測できる。また同一キャリア内においても、複数メーカーから異なった機種のモバイル端末が提供されている。その際、機種によってはブラウザーとしての仕様が異なり、画面表示が違ったものになる場合がある。したがって、Web コンテンツ提供者は機種別に異なったコンテンツを用意しなければならない。そこでこのような環境では、モバイル端末の画面サイズ、対応できるデータフォーマットに合わせ、キャリアごと、機種ごとの制約事項をいかにうまく吸収し、Web オブジェクトの画質調整する必要があると考えられる [7]。

1.2 画質調整の三つのカテゴリ

1.2.1 概要

様々な研究 [8][9] により、帯域幅が狭いネットワークにおいて高速な応答時間を求めたり、モバイル端末の異なる状況を合わせたり、するための画質調整を行う場所として三つがある。本項ではその三つのカテゴリについて述べる。

1.2.2 クライアントによる画質調整

この方法ではクライアントが各自の端末でアクセスしたオブジェクトに対して自分の要求に合うような画質調整を行う。このアプローチの利点とは全体のシステムを変更することなくオブジェクトを更新する時におよびシステム構築するときに手間がかからない。しかしオリジナルな画像サイズが大きくて通信ネットワークに対しては負担がかかりモバイルネットワークのような帯域幅が狭いネットワークにおいてはレスポンス時間が長くなる問題が発生する。また、パソコンより本来処理能力の低いモバイル端末に対しては画質調整などの処理時間が長くなり効率がわるい点が考えられる。

1.2.3 Web サーバによる画質調整

文献 [3] は Web サーバにおいてオブジェクトに対し画質調整を行う場合には Web サーバで一つのオブジェクトを作成すると同時にオフラインで画質調整をし複数のバージョンを作成しておき、そのオブジェクトにアクセスするユーザに選択させるという方法をとっている。このアプローチの利点は事前に画質調整をしたのでクライアントからリクエストを来るときに画質調整コストを生じない、特にプロキシサーバと Web サーバの間の帯域幅が大きい場合には、ネットワーク転送コストが画質調整コストより小さくて全体発生するコストが少なくなる。しかしこの方法では提供する情報を更新する際に複数のバージョ

ンを更新しなければならないという問題や一つのオブジェクトに対して複数のバージョンが存在し保存するリソースが何倍にも増加する問題が生じ、、情報提供者側の管理コストが増大する。また、クライアントのニーズに対応する柔軟性も欠けている。文献 [5] はページに対して配送時間を指定するだけで、クライアントとの間のネットワーク帯域幅を元に提供する画像の品質とデータ量を動的に調整し、配送時間を制御することができる WWW サーバを提案した。しかし、この方法ではクライアントがキャッシュ機能をもつプロキシ経由で WWW サーバにアクセスする場合、品質調整した画像ファイルがプロキシにキャッシュされるため、その後同一プロキシを経由して同一ファイルを要求するすべてのクライアントにプロキシにキャッシュされているファイルが配送されてしまうので、プロキシにおけるキャッシュを許さないような指定を行い、プロキシキャッシュによって得られるべきサーバの負荷軽減やネットワークのトラフィックの軽減の効果が得られないことになる。また、複数クライアントが同時に同一プロキシを通して同一ページにアクセスしてきた場合に、IP アドレスでクライアントの判別を行うことができないため、適用すべき帯域幅情報に誤りが生じる。また、一つのオブジェクトに対して複数のバージョンが存在し WWW サーバによるキャッシュする時にキャッシュサイズが限定されないため保存するリソースが何倍にも増加する問題が生じ、、情報提供者側の管理コストが大きい。

1.2.4 Web プロキシサーバによる画質調整

クライアントと WWW サーバの間にあるプロキシサーバにおいて画質調整を行う。文献 [4][8][9] では幾つかの画質を落とす選択肢を提供し、各クライアントが自分のニーズに応じて選択可能である。例えば、文献 [8] の場合には一つのオブジェクトに対して、オリジナル画像の 100%,80%,60%,40%,20%の五つのバージョンの画質調整できるオプションーを用意し、クライアントに選択された通りに画質調整を行う。そうすることによってクライアントのニーズに柔軟な対応ができ、また画質調整した前後の画像をキャッシュしておき再び同じオブジェクトの同じバージョンのリクエストがくるときにすぐ応答できるメリットがある。以上の二つのアプローチに対しては Web プロキシサーバにおける画質調整が一番妥当と考えられる。本研究は画質調整機能を持つプロキシサーバにおいて画質調整した各オブジェクトおよび各バージョンについてのキャッシュ置き換えアルゴリズムに関する研究を行う。

1.3 本論文の構成

第 2 章はプロキシサーバについて紹介する。2.1 節は本研究のシステムアーキテクチャーについて述べる。2.2 節はプロキシサーバの画質調整の仕組みを語る。2.3 節は今まで既存のキャッシュ置き換えアルゴリズムを紹介しその中の問題点を洗い出す。その問題点を解決するために 2.4 節では本研究の目的を導き出す。第 3 章は本研究のアルゴリズムおよび公式化した関数について紹介する。第 4 章はネットワークシミュレータである Ns2 を用い

提案した手法を実装する。第 5 章は節約コストおよびキャッシュヒット率について従来研究と評価する。第 6 章は本研究のまとめについて述べる。

第2章　画質調整機能を持つプロキシサーバ

2.1　プロキシサーバ

2.1.1　プロキシサーバのアーキテクチャーと動作

システムアーキテクチャーとしては狭帯域ネットワークと広帯域ネットワークの間に置かれた画質調整機能を持つプロキシサーバシステムが全部適用できるが、ここで画質調整機能を持つプロキシサーバとしてはよく知られている狭帯域であるモバイルネットワークと広帯域である IP ネットワークの中間に介して存在している WAP プロキシサーバシステムを取り上げる。システムアーキテクチャーとしては以下の図 2.1 が示したようなモバイルネットワークと IP ネットワーク間の WAP プロキシサーバシステムである。

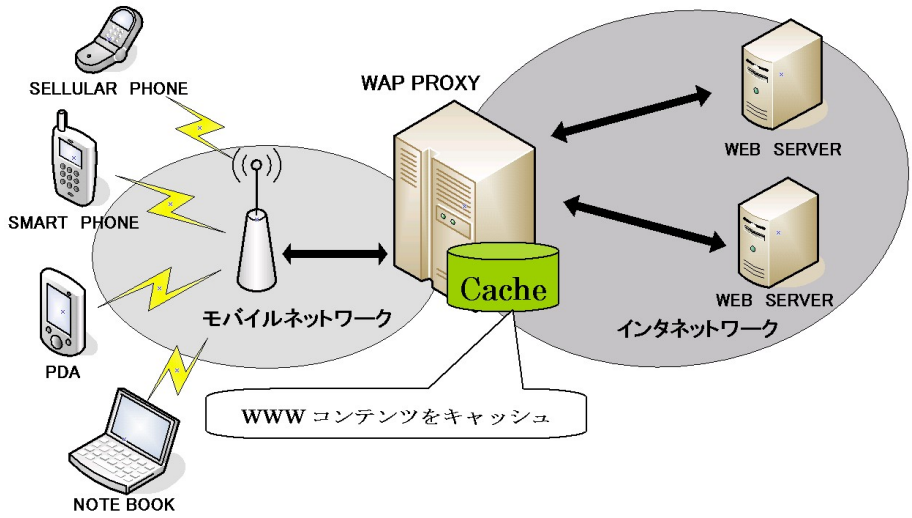

図 2.1: WAP プロキシアーキテクチャー

WAP プロキシサーバは普通のプロキシサーバと同様に一般的に Web サーバよりクライアントに近い場所に存在し、クライアントは WAP プロキシサーバに対して Web コンテンツをリクエストする。画質調整機能を持つ WAP プロキシサーバの主の動作を以下に示す。

1. クライアントからWebコンテンツのリクエストを受け取る。
2. リクエストされたコンテンツをプロキシがキャッシュに保持している場合にはそれをレスポンスとしてクライアントに送信する。
3. リクエストされたコンテンツを保持していない場合には画質調整による生成できれば、画質調整されたコンテンツをレスポンスする。
4. リクエストされたコンテンツを保持していない、しかも画質調整による生成できない場合には、Webサーバにそのコンテンツをリクエストする。
5. Webサーバからコンテンツを受け取ると、画質調整する必要があれば、画質調整してからクライアントに配送する、なければ、そのまま配送する。配送するとともに、キャッシュに保存する。

複数のクライアントがWAPプロキシサーバのキャッシュを共有することにより、クライアントの配送時間の短縮、ネットワークの負荷低減、サーバの負荷低減といったことが実現できる。また、クライアントがリクエストしたWWWコンテンツが存在するサーバがダウンしている場合でも、そのコンテンツをWAPプロキシサーバがキャッシュに保持していれば、クライアントはレスポンスを受けることができる。つまり、WWWシステムの信頼性を向上できる。

2.1.2 プロキシサーバのWebオブジェクトの画質調整

従来研究[8]は画質調整に関しては幾つかの定義を定めた。これからそれについて紹介する。

画質調整

Webオブジェクトは Web上の静止画像のことである。オブジェクトの画質調整とはモバイル端末のニーズに合わせ、画質を粗くしWebオブジェクトのデータサイズを縮小したり、データフォーマットを変換したりするための画像処理のことである。例えば，画像サイズを削減するために高解像度のJPEG画像を低解像度のJPEG画像に変換すること或いはGIFフォーマット画像からJPEGフォーマット画像に変換することが挙げられる。

Webオブジェクト

Webオブジェクトは画質調整が行われた前の画像（オリジナルな画像）と画質調整された後の各画像の総称である。他のWebオブジェクトと区別するためにオブジェクト番号がついている。

バージョン

バージョンは一つの Web オブジェクトに対して画質調整をされた前後の画像の識別子である。オリジナル画像のバージョン番号は 1 であり、画像の画質が悪くなるにつれてバージョンの番号が大きくなる。

図 2.2 は Web オブジェクト i の各バージョン間の画質調整関係を表す重み付きグラフ表現である。図 2.2 に関するパラメータが表 2.1 に示している。

表 2.1: パラメータ説明表

i	オブジェクト番号
y	バージョン番号
$o_{i,y}$	オブジェクト i の y バージョンである
$w_i(y,z)$	オブジェクト i の y バージョンから z バージョンへの画質調整コストである
d_i	Web サーバからオブジェクト i の転送コストである

$o_{i,y}$ はオブジェクト i の y バージョンである。例えば、バージョン 1 は $o_{i,1}$ であり、バージョン 2 は $o_{i,2}$ である。

節約コスト

本研究に関するコストは以下の二つの種類がある。

- 節約した画質調整時間であり、$w_i(y,z)$ を用いて表現する。$w_i(y,z)$ はオブジェクト i のバージョン y からバージョン z への画質調整コストである。例えば、図 2.2 においては $w_i(1,2)$ はオブジェクト i のバージョン 1 からバージョン 2 への画質調整コストである。

  文献 [8][9] は画質調整の節約コストを計算する際に以下の式を用いることにした。

$$transcode\_cost = \frac{version\_size}{transcode\_ratio}$$

  画質調整の節約コストはそのバージョンのデータサイズと画質調整の変換率の商である。transcode_cost は画質調整の節約コストである。version_size はバージョンのデータサイズである。transcode_ratio は画質調整の変換率であり、文献 [8][9] が画質調整の変換率を 20KB/sec にした。

- 節約した Web サーバからの応答時間と定義され、d_i を用いて表す。図 2.2 においては d_i は Web サーバからオブジェクト i の転送コストである。

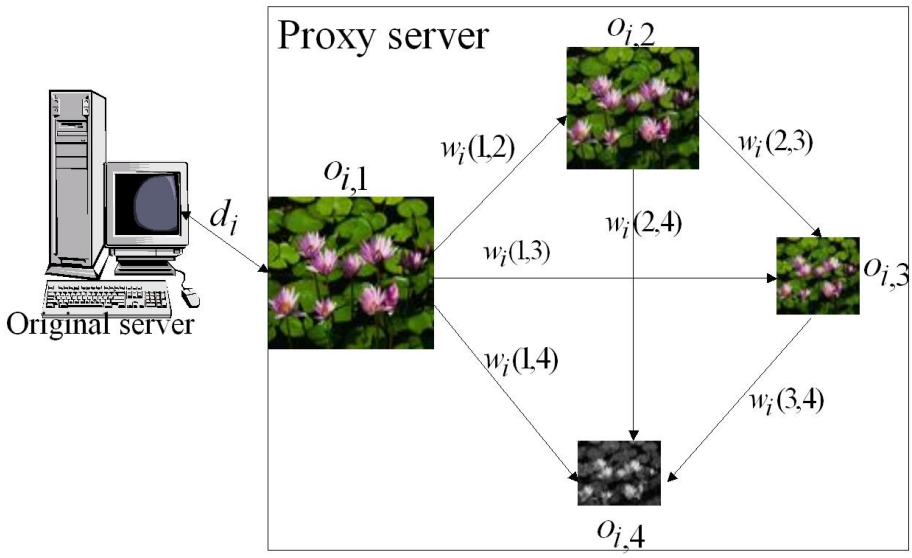

図 2.2: バージョン間の画質調整関係グラフ表現

2.2 プロキシキャッシュに関する研究

キャッシュ置き換えアルゴリズムはプロキシのパフォーマンスに大きな影響を与える。キャッシュ置き換えアルゴリズムとは有限なキャッシュ空間においてどのオブジェクトのどのバージョンをキャッシュに保持し、どのオブジェクトのどのバージョンをキャッシュから削除するかを決定する戦略であり、キャッシュ空間のサイズに大きく影響をうける。キャッシュサイズが無限であるならこのようなアルゴリズムはいらないが、実際には有限であるため必要となってくる。

2.2.1 キャッシュ置き換アルゴリズム

文献 [10][11][12][13] ではクライアントのアクセスパタンからクライアントのリクエストを予測し、それに対するレスポンスをあらかじめキャッシュしておくプリフェッチ手法を提案している。また文献 [14] では、クライアントから HTML ドキュメントをリクエストされた際に、プロキシにおいて HTML ドキュメントを解析し、リンク先の HTML ドキュメント及びそのほかのオブジェクトを Web サーバから取得するプリフェッチ手法を提案している。これらのプリフェッチ手法はネットワークトラフィックを増加させてしまうが、クライアントの配送時間の短縮、及びキャッシュのヒット率向上が実現できる。

文献 [9] は画質調整機能を持つプロキシサーバにおいて、LRU_{CV}^{MIN} と LRU_{DM}^{MIN} 二つのキャッシュ置き換えアルゴリズムを提案した。まず、画質調整を行った後の画像のキャッシュ仕方によって二つの方法が定められた。一つ目はオリジナルの画像をしかキャッシュしない coverage-based アルゴリズムで、二つ目は画質調整された後の画像をしかキャッシュ

しない demand-based アルゴリズムである。キャッシュ空間が一杯になれば、今世の中によく使われた LRU(Least Recently Used) と結合し、キャッシュの置き換えを行った。そして、coverage-based アルゴリズムと LRU を結合したのアルゴリズムは LRU_{CV}^{MIN} である。demand-based アルゴリズムと LRU を結合したのアルゴリズムは LRU_{DM}^{MIN} である。この二つのアルゴリズムはネットワークの性質及び画質調整の遅延などの要素を考慮していない、とても単純なアルゴリズムであった。また画質調整されたオブジェクトの各バージョンの関係を考慮していない。

2.2.2 従来研究の AE アルゴリズム

AE キャッシュ置き換えアルゴリズムが文献 [8] による提案したアルゴリズムである。文献 [8] は、本章 2.1 節で紹介した重み付きグラフ (図 2.2) を用い、画質調整されたオブジェクトの各バージョン間の関係を表現した。グラフ上で、プロキシと Web サーバ間のネットワークの遅延や画質調整するときの遅延などの色々なパラメータを考え出した。これらのパラメータを用い、キャッシュされたバージョンの数のもとに、節約コストの計算関数 PF（）を提案した。あるオブジェクトのバージョンが一つの場合には提案した関数式 2.1 で計算する。バージョンが複数キャッシュされた際に、各キャッシュされたバージョンのコスト和（式 2.2）を計算してから各バージョンの節約コスト（式 2.3）を計算する。

キャッシュされたバージョンが一つの場合
まず、あるオブジェクトのキャッシュされたバージョンがキャッシュに一つ存在している場合には、関数 2.1 を用いることにした。表 2.2 は式 2.1 で使われたパラメータの定義を示す。

表 2.2: 式 2.1 のパラメータ定義

$PF()$	キャッシュされたバージョンのコスト計算関数
$o_{i,j}$	オブジェクト i の j バージョン
(j,x)	バージョン j からバージョン x までの辺
$E[G_i]$	キャッシュされたバージョン i からの辺集合
$r_{i,x}$	オブジェクト i の x バージョンのアクセス率
d_i	WEB サーバからオブジェクト i の応答コスト
$w_i(1,x)$	i のバージョン 1 から x への画質調整コスト
$w_i(j,x)$	i のバージョン j から x への画質調整コスト

$$PF(o_{i,j}) = \sum_{(j,x)\in E[G_i]} r_{i,x} * (d_i + w_i(1,x) - w_i(j,x)), \tag{2.1}$$

式 2.1 の場合には、自分自身を含め、生成できる各バージョンに対して一つずつ節約コストを計算し、後にはトータルをし、キャッシュされたバージョンの節約こストを求める。

キャッシュされたバージョンが複数の場合
あるオブジェクトには複数のバージョンが同時にキャッシュされた場合にまず関数式 2.2 を用い、総合コストを算出する。表 2.3 は式 2.2 で使われたパラメータの定義を示す。

表 2.3: 式 2.2 のパラメータ定義

$PF()$	キャッシュされたバージョンのコスト計算関数
$o_{i,jk}$	オブジェクト i の jk バージョン
v	キャッシュされたバージョン
$V[G']$	キャッシュされたバージョンの集合
(v,x)	バージョン v からバージョン x までの辺
$E[G']$	キャッシュされたバージョンからの辺集合
$r_{i,x}$	オブジェクト i の x バージョンのアクセス率
d_i	WEB サーバからオブジェクト i の応答コスト
$w_i(1,x)$	i のバージョン 1 から x への画質調整コスト
$w_i(v,x)$	i のバージョン v から x への画質調整コスト

$$PF(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,jk}) = \sum_{v \in V[G']} \sum_{(v,x) \in E[G']} r_{i,x} * (d_i + w_i(1,x) - w_i(v,x)), \tag{2.2}$$

関数式 2.2 では、まず、キャッシュされたバージョンの各辺について節約コストを計算し、後には、その和を求める。

次に各キャッシュされたバージョンのコストを計算する時に式 2.3 を用い，計算する。キャッシュされたバージョンの大きい番号から、計算しているキャッシュされたバージョン番号までの和を求め、そして計算しているキャッシュされたバージョン番号の直前のキャッシュされたバージョン番号までの和を引くことにより、今計算しているキャッシュされたバージョンのコストを算出する。

$$PF(o_{i,j}|o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,jk}) = PF(o_{i,j}, o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,jk}) - PF(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,jk}) \tag{2.3}$$

文献 [8] が提案した関数を図 2.3 を用い説明する。図 2.3 には Web サーバとプロキシサーバの二つの部分からなる。Web サーバとプロキシサーバ間の応答コスト $d_i(\exists i : d_i = 20)$ が中括弧に囲まれた数字の 20 である。プロキシサーバ内の画質調整には一つのオブジェクトが五つのバージョンを持っており、バージョン 1、バージョン 2 とバージョン 3 がキャッシュされている状態である。画質調整コストが図のようにバージョン 1 から各バージョン 2，3，4，5 への画質調整コスト $(\exists i \forall x(x \in [2,5], x \in N) : w_i(1,x))$ が 10 であり、バージョン 2 からバージョン 3，4，5 への画質調整コスト $(\exists i \forall x(x \in [3,5], x \in N) : w_i(2,x))$ は 8 であり、

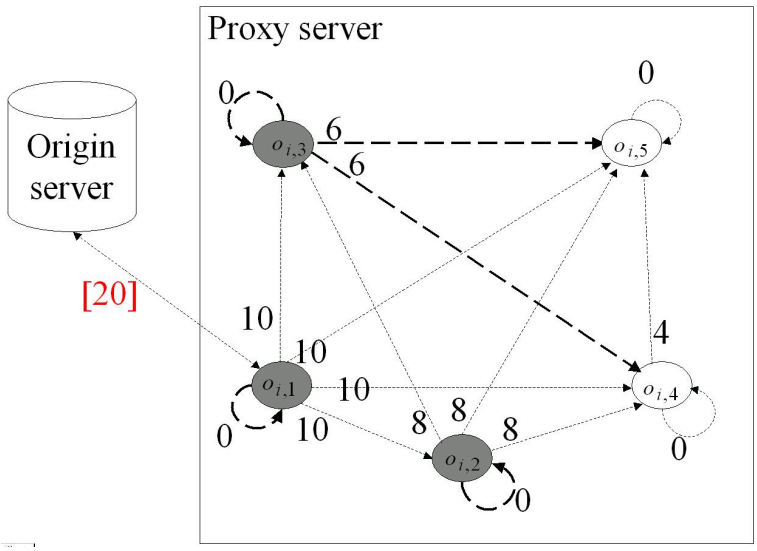

図 2.3: バージョン関係（事例）

バージョン 3 からバージョン 4，5 への画質調整コスト $(\exists i \forall x(x \in [4,5], x \in N) : w_i(3,x))$ が 6 であり、バージョン 4 からバージョン 5 への画質調整コスト $(\exists i \forall x(x = 5) : w_i(4,x))$ が 4 である。各バージョンが自分への画質調整する必要がないので、画質調整コストが 0 である。太い長破線がキャッシュされたバージョンから画質調整するバージョンをさしている。各バージョンのアクセス率 $(\exists i \forall x(x \in [1,5], x \in N) : r_{i,x})$ が 10 に仮想する。この例で、文献 [8] が提案した関数を用いキャッシュされたバージョン 1，2，3 の節約コストを計算する結果は表 2.4 に示している。

表 2.4: AE アルゴリズムの計算結果

バージョン番号	節約コスト
1	200
2	300
3	780

まず、バージョン 3 の節約コスト $PF(o_{i,3})$ は式 2.1 によって $10*(20+10-0)+10(20+10-6)+10(20+10-6)(r_{i,3}*(d_i+w_i(1,3)-w_i(3,3))+r_{i,4}*(d_i+w_i(1,4)-w_i(3,4))+r_{i,5}*(d_i+w_i(1,5)-w_i(3,5)))$ であるので、節約コストが 780 である。式 2.2 を用い、キャッシュされたバージョン 2 と 3 の総合節約コストを計算する。$PF(o_{i,2},o_{i,3})$ が $10*(20+10-0)+10*(20+10-0)+10(20+10-6)+10(20+10-6)(r_{i,2}*(d_i+w_i(1,2)-w_i(2,2))+r_{i,3}*(d_i+w_i(1,3)-w_i(3,3))+r_{i,4}*(d_i+w_i(1,4)-w_i(3,4))+r_{i,5}*(d_i+w_i(1,5)-w_i(3,5)))$ であり、バージョン 2 とバージョン 3 の総合節約コストが 1080 である。式 2.3 を用い、

バージョン 2 の節約コストは 300（1080-780）である。バージョン 1 の節約コストを計算するときに式 2.2 を用い、まず $PF(o_{i,1}, o_{i,2}, o_{i,3})$ を計算する。$PF(o_{i,1}, o_{i,2}, o_{i,3})$ は $10 * (20 - 0) + 10 * (20 + 10 - 0) + 10 * (20 + 10 - 0) + 10(20 + 10 - 6) + 10(20 + 10 - 6)(r_{i,1} * (d_i - w_i(1,1)) + r_{i,2} * (d_i + w_i(1,2) - w_i(2,2)) + r_{i,3} * (d_i + w_i(1,3) - w_i(3,3)) + r_{i,4} * (d_i + w_i(1,4) - w_i(3,4)) + r_{i,5} * (d_i + w_i(1,5) - w_i(3,5)))$ であり、バージョン 1，2，3 の総合節約コストが 1280 である。次に式 2.3 を使って計算するとバージョン 1 のコストは 200（1280-1080）である。キャッシュがいっぱいになってキャッシュ置き換えをする時にバージョン 1 の節約コストが小さくてキャッシュから削除され、残りのバージョン 2 とバージョン 3 の総合節約コストは 1080 である。

しかし、もしバージョン 2 が削除されバージョン 1 とバージョン 3 がキャッシュに残る場合には、式 2.2 を用いバージョン 1 とバージョン 3 の節約コストを計算すると $PF(o_{i,1}, o_{i,3})$ は 1180$(10*(20-0)+10*(20+10-10)+10*(20+10-0)+10*(20+10-6)+10*(20+10-6))$ であり、バージョン 1 が削除された後の残りバージョン 2 とバージョン 3 の総合節約コスト（1080）より高い。そこで、従来研究はキャッシュ置き換えのコスト高いバージョンを残す原則と矛盾する。原因は各キャッシュされたバージョンの節約コストを計算するときに他のキャッシュされたバージョンからの影響を考えてないからである。文献 [8] は式 2.2 からコストを計算する時に Web サーバとプロキシ間の遅延を表すパラメータ d_i とバージョン 1 からの画質調整遅延であるパラメータ $w_i(1, x)$ を用い計算していることによりキャッシュされたバージョンのコストを算出している時に全部 Web サーバを基準としているので、他のキャッシュされたバージョンから生成できることを考慮していない。この分析した結果によって文献 [8] は各バージョン間の影響関係を考慮してないことが分かった。

2.3 本研究の目的

本研究では以上の従来研究の問題点を解決するために、キャッシュされたバージョンの節約コストを計算するときにキャッシュされたバージョン間の影響関係を考慮することによって、節約したコストを高めることを目的としている。キャッシュされたバージョン間の相互影響を記録するために本研究は新たにパラメータを定義する。また、提案したパラメータはキャッシュにあるバージョンの状況に応じて動的に変わっていくアルゴリズムを提案する。次に、提案したパラメータを用い、バージョン間の関係を動的に表す有効な節約コスト関数を提案する。最後に、本研究で提案した関数を用い計算した結果を分析実装し、総合節約コストを高めることを示す。

第3章　相互作用を考慮したキャッシュの置き換えアルゴリズム

3.1　バージョン間の相互影響関係

本節では図 2.3 を例としてバージョン間の相互影響関係を説明する。表 3.1 はバージョン間の相互影響関係を考慮した後のキャッシュされたバージョン 1，2，3 の節約コストを示している。

表 3.1: バージョン間の相互影響関係を考慮した後の計算結果

バージョン番号	節約コスト
1	200
2	100
3	120

バージョン間の影響関係を考慮し、図 2.3 のバージョン 3 の節約コストを計算する場合にはバージョン 2 がキャッシュに存在しているので、バージョン 2 からの影響を考えなければならない。もしバージョン 3 がキャッシュされない場合には、バージョン 2 は画質調整コスト 240（8*3*10) でバージョン 3，4，5 を生成できる。バージョン 3 がキャッシュされることによってこの 240 の画質調整コストを節約することができる。一方、バージョン 3 がバージョン 4，5 を生成するためにかかる画質調整コストは 120(6*3*10) であるので、実際にバージョン 3 がキャッシュされることにより節約コストは 120（240 – 120）である。バージョン間の影響関係を考慮した場合のバージョン 3 の節約コストの計算の分析をまとめると、もしバージョン 3 がキャッシュに存在しなければ、バージョン 2 は 240 の画質調整コストがかかる。バージョン 3 の存在によって 240 の画質調整コストを節約することができるが、自分自身からかかるコストが 120 であるので、実際に節約できるコストは 120 である。同様にバージョン間の影響関係を考慮した場合のバージョン 2 の節約コストは 100（10*10）である（もしバージョン 2 がキャッシュされなかったら、バージョン 1 から 100 の画質調整コストで生成できる。バージョン 2 が存在することによって 100 の画質調整コストを節約することができた)。バージョン 1 の節約コストは 200（20*10）である。バージョン 2 の節約コストが一番小さい、言い換えるとバージョン 2 を削除したら、増やした画質調整のコストが一番小さい、なのでキャッシュ置き換えをする際に節約コストが

一番小さいバージョン 2 が削除される。第 2 章で分析した従来研究の不都合なことを生じることができないので、節約コストとキャッシュの効率が高くなることが期待できる。

本章の次の節ではバージョン間の影響関係を分析した結果をまとめ、影響関係を構築するアルゴリズムを提案していく。

3.2 本研究の提案手法

3.2.1 研究用語とパラメータの定義

本節では本研究が定義した幾つかの研究用語とパラメータ（表 3.2 に示す）について説明する。

表 3.2: 提案した用語とパラメータの定義

生成バージョン	あるバージョンを生成するコストが一番小さいキャッシュされたバージョン
影響バージョン	あるバージョンを生成するコストが二番目小さいキャッシュされたバージョン
生成関係集合	生成バージョンと生成されるバージョンから構成される集合
影響関係集合	影響バージョンと影響されるバージョンから構成される集合
$min\_cost1_{i,z}$	キャッシュされたバージョンから生成できる一番小さい画質調整コスト
$min\_cost2_{i,z}$	キャッシュされたバージョンから生成できる二番目小さい画質調整コスト

バージョンはキャッシュされたバージョン（図 2.3 の例ではバージョン 1，2，3）とキャッシュされないバージョン（図 2.3 の例ではバージョン 4，5）に分けることができる。本研究はさらにキャッシュされたバージョンを生成バージョンと影響バージョンに分類した。

生成バージョン

生成バージョンはあるバージョンを生成するコストが一番小さいキャッシュされたバージョンのことであり、このキャッシュされたバージョンはそのバージョンに対して生成バージョンである（図 2.3 の例ではバージョン 1 とバージョン 2 が自分自身に対して生成バージョンであり、バージョン 3 がバージョン 3，4，5 に対して生成バージョンである）。

生成関係集合

生成関係集合は生成バージョンと生成されるバージョンから構成される集合である（図 2.3 の例ではバージョン 1 とバージョン 2 は自分自身が生成関係集合であるが、バージョン 3 は自分自身とバージョン 4，5 から生成関係集合を構成している）、生成バージョンと生成されるバージョンの間は太い長破線である。

影響バージョン

影響バージョンは画質調整できるあるバージョンを生成するコストが二番目小さいキャッシュされたバージョンである（図 2.3 の例ではバージョン 1 がバージョン 2 に対して影響バージョンであり、バージョン 2 がバージョン 3，4，5 に対して影響バージョンである）。

影響バージョン集合

影響関係集合は影響バージョンと影響されるバージョンから構成される集合である（図 2.3 の例ではバージョン 1 がバージョン 2 に対しての影響関係集合はバージョン 1 とバージョン 2 から構成している。バージョン 2 がバージョン 3，4，5 に対しての影響関係集合はバージョン 2 とバージョン 3，4，5 から構成している）、集合の各辺は太い実線である。

キャッシュされたバージョン間の相互作用を記録するために本研究では $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ 二つのパラメータを提案した。各バージョンが $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ 二つのパラメータをもっている。

パラメータ $min\_cost1_{i,z}$

パラメータ $min\_cost1_{i,z}$ はオブジェクト i のキャッシュされたバージョンが z バージョンを生成する際にかかる画質調整コストの中に一番小さいコスト（生成バージョンからの画質調整コスト）を記録するパラメータであり、オブジェクト i の z バージョンが持っている。$min\_cost2_{i,z}$ は生成バージョンがキャッシュされない場合に影響バージョンから画質調整される時にかかるコストである。

パラメータ $min\_cost2_{i,z}$

$min\_cost2_{i,z}$ はオブジェクト i のキャッシュされたバージョンが z バージョンを生成する際にかかる画質調整コストの中に二番目小さいコスト（影響バージョンからの画質調整コスト）を記録するパラメータであり、オブジェクト i の z バージョンが持っている。$min\_cost1_{i,z}$ は生成バージョンから画質調整される時にかかるコストである。

$min\_cost2_{i,z}$ と $min\_cost1_{i,z}$ の差はバージョン $o_{i,z}$ を生成するときに影響バージョンからの影響を考慮した後の生成バージョンのコストである。$min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ の

表 3.3: 各バージョンの $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ のデフォルト値

	$o_{i,1}$	$o_{i,2}$	$o_{i,3}$	$o_{i,4}$	$o_{i,5}$
$min\_cost1_{i,z}$	20	30	30	30	30
$min\_cost2_{i,z}$	20	30	30	30	30

デフォルト値はオリジナルサーバから各バージョンを生成生成する際のコストである（図 2.3 の例では表 3.3 に示す）。

表 3.4: 提案したアルゴリズムのパラメータ説明

$o_{i,y}$	キャッシュされたバージョン y
$o_{i,z}$	キャッシュされたバージョンから画質調整できるバージョン z
$w_i(y,z)$	キャッシュされたバージョン y からバージョン z への画質調整コスト

3.2.2 影響関係を構築するアルゴリズム

本項では影響関係を形成するアルゴリズムについて説明する。その処理は *Procedure Creat_Relation* に示す。*Procedure Creat_Relation* にあるパラメータの説明が表 3.4 で示している。$o_{i,y}$ はキャッシュに入れるオブジェクト i のバージョン y であり、バージョン y はキャッシュされたバージョンを示し、キュー 1 に入れられる。$o_{i,z}$ はあるキャッシュされたバージョンから生成できるバージョンであり、キュー 2 に入れられる。$w_i(y,z)$ はキャッシュされたバージョン y からバージョン z への画質調整コストである。図 3.1 が本研究で提案したアルゴリズムのフローチャート図を示している。

Procedure Creat_Relation

1: オブジェクト i のキャッシュされたバージョンを全部キュー 1 に入れる

2: **while** キュー 1 ! = NULL **do**

3: キュー 1 から一つのバージョン $o_{i,y}$ を取り出す

4: $o_{i,y}$ が生成できるバージョンを全部キュー 2 に入れる

5: **while** キュー 2 ! = NULL **do**

6: キュー 2 から一つのバージョン $o_{i,z}$ を取り出す

7: **if** $w_i(y,z) < min\_cost1_{i,z}$ **then**

8: $o_{i,z}$ が元の影響関係集合から脱出する

9: $o_{i,z}$ の元の生成関係集合が影響関係集合に変わり、$o_{i,z}$ との間に太い実線をはる

10: $min\_cost2_{i,z} = min\_cost1_{i,z}$;

11: $o_{i,z}$ が $o_{i,y}$ の生成関係集合に入れられ、$o_{i,y}$ との間太い長破線をはる

12: $min\_cost1_{i,z} = w_i(y,z)$;

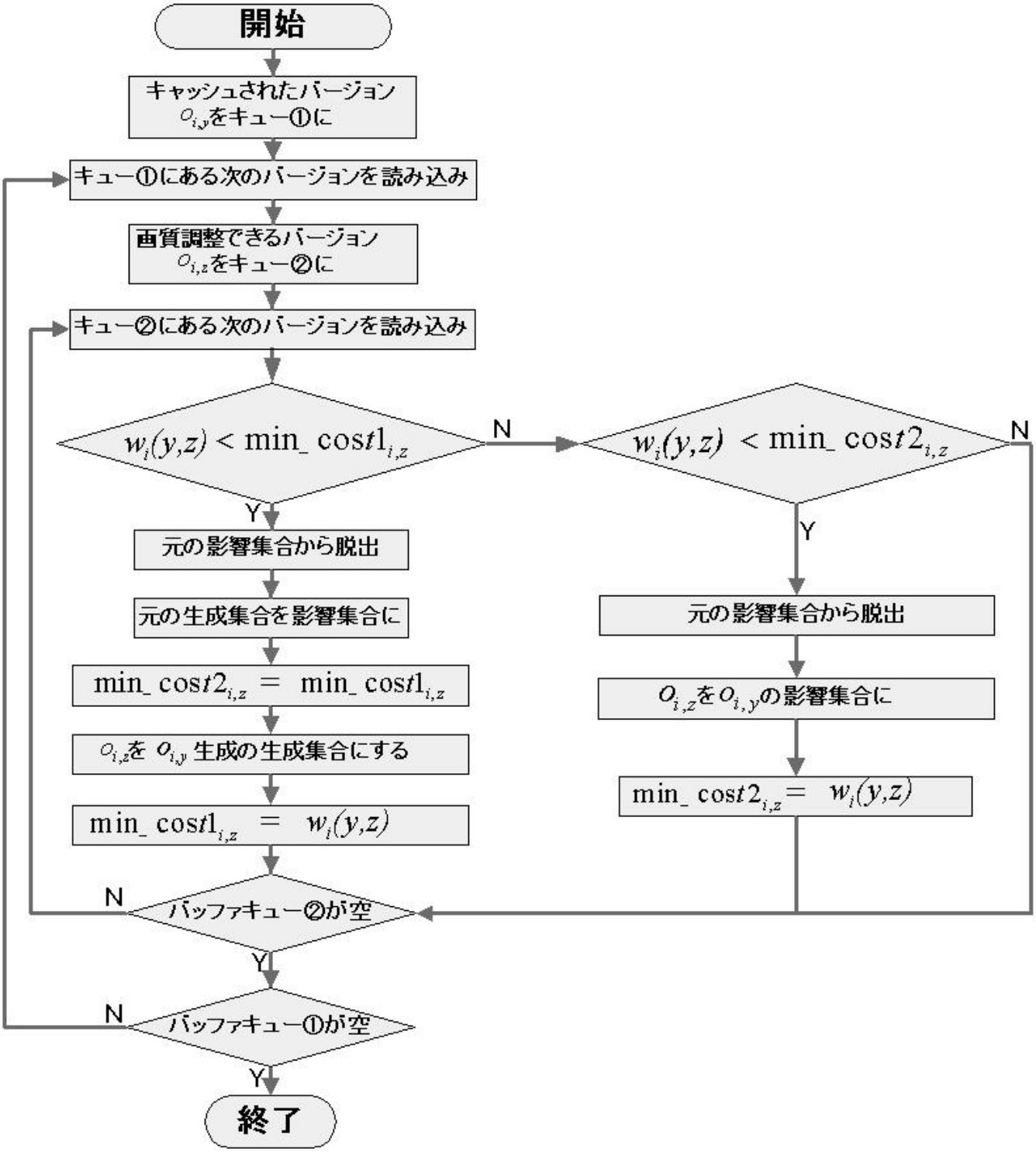

図 3.1: アルゴリズムのフローチャート図

13: **else if** $w_i(y,z) < min\_cost2_{i,z}$**then**

14: $o_{i,z}$が元の影響関係集合から脱出する

15: $o_{i,z}$が$o_{i,y}$との間に太い実線をはる

16: $min\_cost2_{i,z} = w_i(y,z)$;

17: **end if**

18: **end while**

19: **end while**

本研究が提案したアルゴリズムは前節で定めた$min_cost2_{i,z}$と$min_cost1_{i,z}$を用いバージョン間の影響関係集合を構築することができる。これから第2章図2.3の例として説明していく。まず、キャッシュされたバージョンがキュー1に入れられ、一つずつ自分の影響関係集合と生成関係集合を動的に構築していく。第2章図2.3の例ではバージョン1，2，3がキュー1に入れられる。次に今評価しているキャッシュされたバージョンが灰色にされ、自分自身を含めて画質調整できる各バージョンがキュー2に入れられ、今評価しているキャッシュされたバージョンとの関係を構築する。バージョン1の場合にはバージョン1，2，3，4，5、バージョン2の場合にはバージョン2，3，4，5、バージョン3の場合にはバージョン3，4，5がキュー2に入れられる。そしてキュー2にあるバージョンが持っている$min_cost1_{i,z}$がキャッシュされたバージョンyからバージョンzへの画質調整コスト$w_i(y,z)$と比較され、$min_cost1_{i,z}$のコストがバージョンzを生成するための一番小さいコストであるので、もし$w_i(y,z)$が$min_cost1_{i,z}$より小さければ、今の時点評価しているキャッシュされたバージョンからバージョンzに画質調整するときにバージョンyの画質調整コストが一番小さいことが示され、フローチャート図3.1の左部分が実行される。今までバージョンzに対しての生成バージョンが影響バージョンに変えられ、間のリンクを実線する。バージョンyをバージョンzの生成バージョンにし、間のリンクを長破線にする。またバージョンzが持っている影響を記録するパラメータ$min_cost1_{i,z}$と$min_cost2_{i,z}$の値も書き換えられる。本の生成バージョンが影響バージョンに変わったので、本の$min_cost1_{i,z}$が$min_cost2_{i,z}$に代入される。バージョンyが生成バージョンにされたので、$w_i(y,z)$を$min_cost1_{i,z}$に与える。もし$w_i(y,z)$が$min_cost2_{i,z}$より小さければ、今の時点ではバージョンyからバージョンzへの画質調整コストが二番目小さいことであり、フローチャート図3.1の右部分が実行される。バージョンyがバージョンzの影響バージョンに変更され、$w_i(y,z)$が$min_cost2_{i,z}$に代入される。キュー2が空ではなければ、次のバージョンについて同じ処理をされる。キュー2空であれば、キャッシュされたバージョンから構成されるキュー1が空であるかどうかを判断する。キュー1が空でなければ、次のキャッシュされたバージョンyを評価する。空である場合にはすべての

キャッシュされたバージョンに対しての処理が終わり、処理が終了する。第 2 章図 2.3 の例ではもしバージョン 1、バージョン 2、バージョン 3 の順にキュー 1 に入った場合には本研究で提案したアルゴリズムの動作様子は図 3.2、3.3 と図 3.4 に示す。$min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ 二つのパラメータの値の変化が表 3.5、3.6 と表 3.7 に示す。

表 3.5: バージョン 1 が評価されるた後 $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$

	$o_{i,1}$	$o_{i,2}$	$o_{i,3}$	$o_{i,4}$	$o_{i,5}$
$min\_cost1_{i,z}$	0	10	10	10	10
$min\_cost2_{i,z}$	20	30	30	30	30

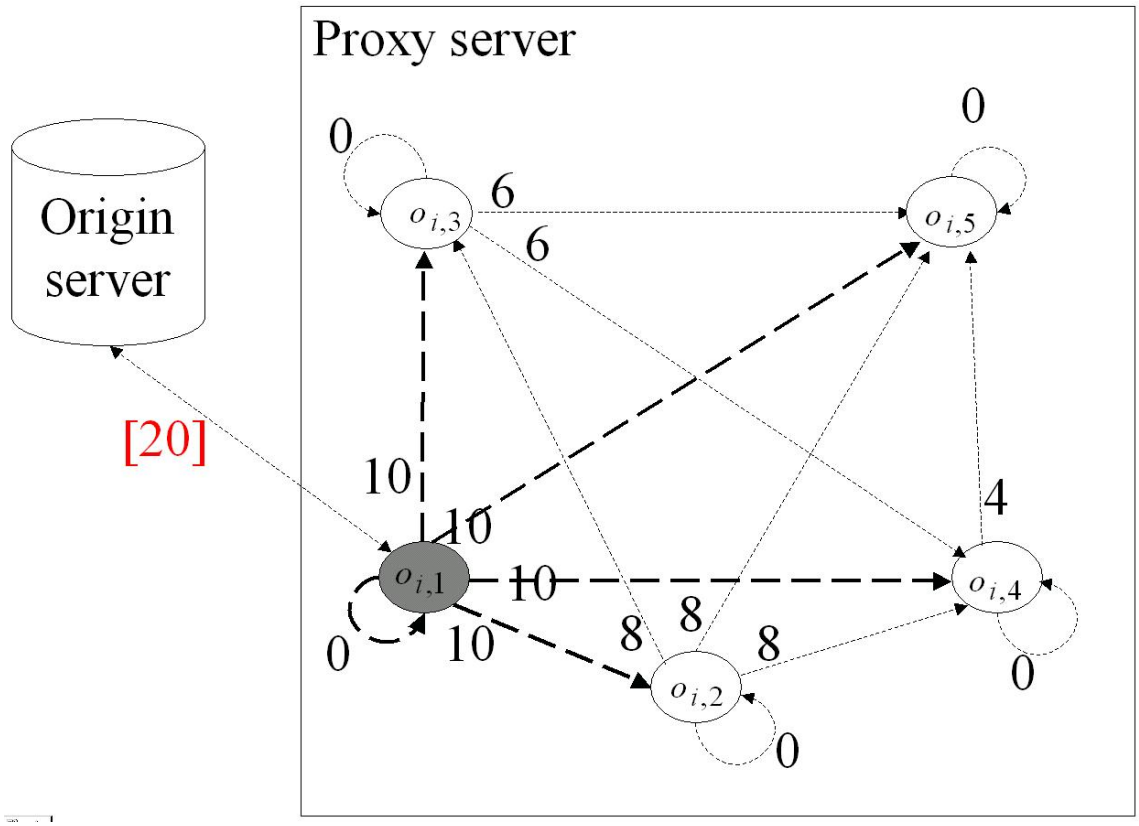

図 3.2: バージョン 1 を評価した後の関係形成図

表 3.6: バージョン 2 が評価された後の $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$

	$o_{i,1}$	$o_{i,2}$	$o_{i,3}$	$o_{i,4}$	$o_{i,5}$
$min\_cost1_{i,z}$	0	0	8	8	8
$min\_cost2_{i,z}$	20	10	10	10	10

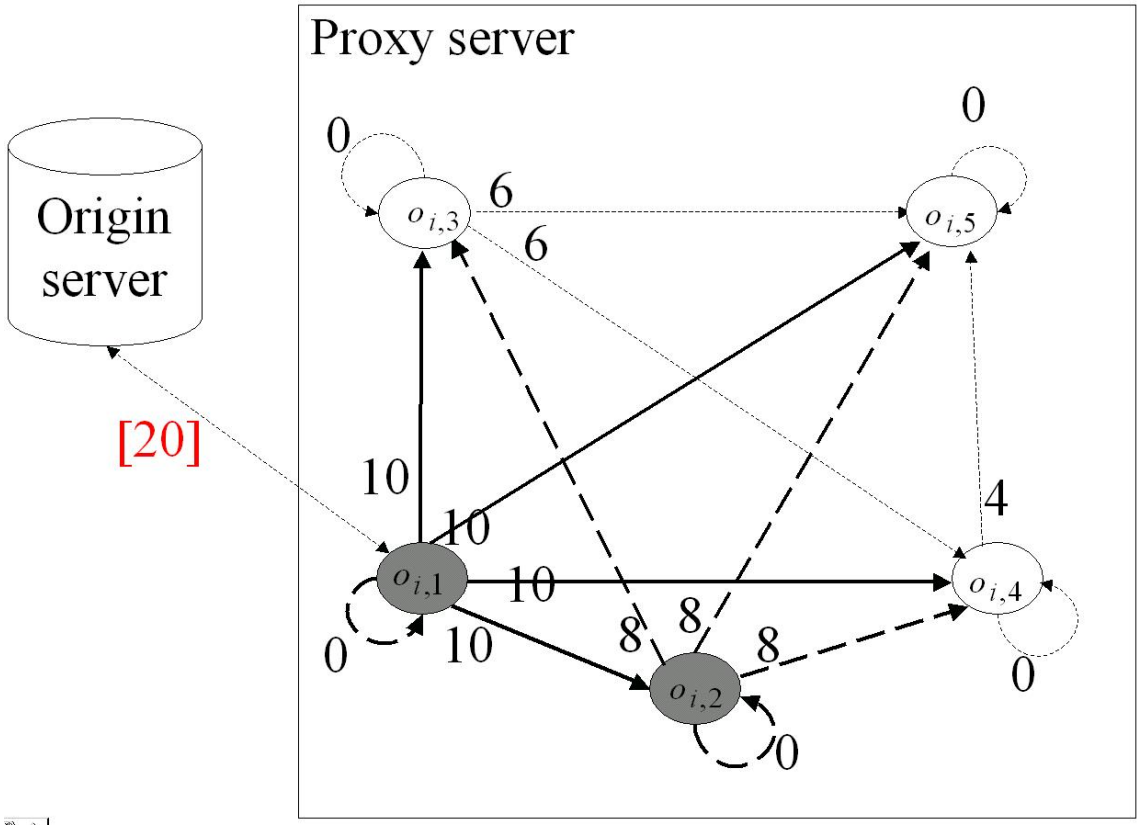

図 3.3: バージョン 2 を評価した後の関係形成図

表 3.7: バージョン 3 が評価された後の $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$

	$o_{i,1}$	$o_{i,2}$	$o_{i,3}$	$o_{i,4}$	$o_{i,5}$
$min\_cost1_{i,z}$	0	0	0	6	6
$min\_cost2_{i,z}$	20	10	8	8	8

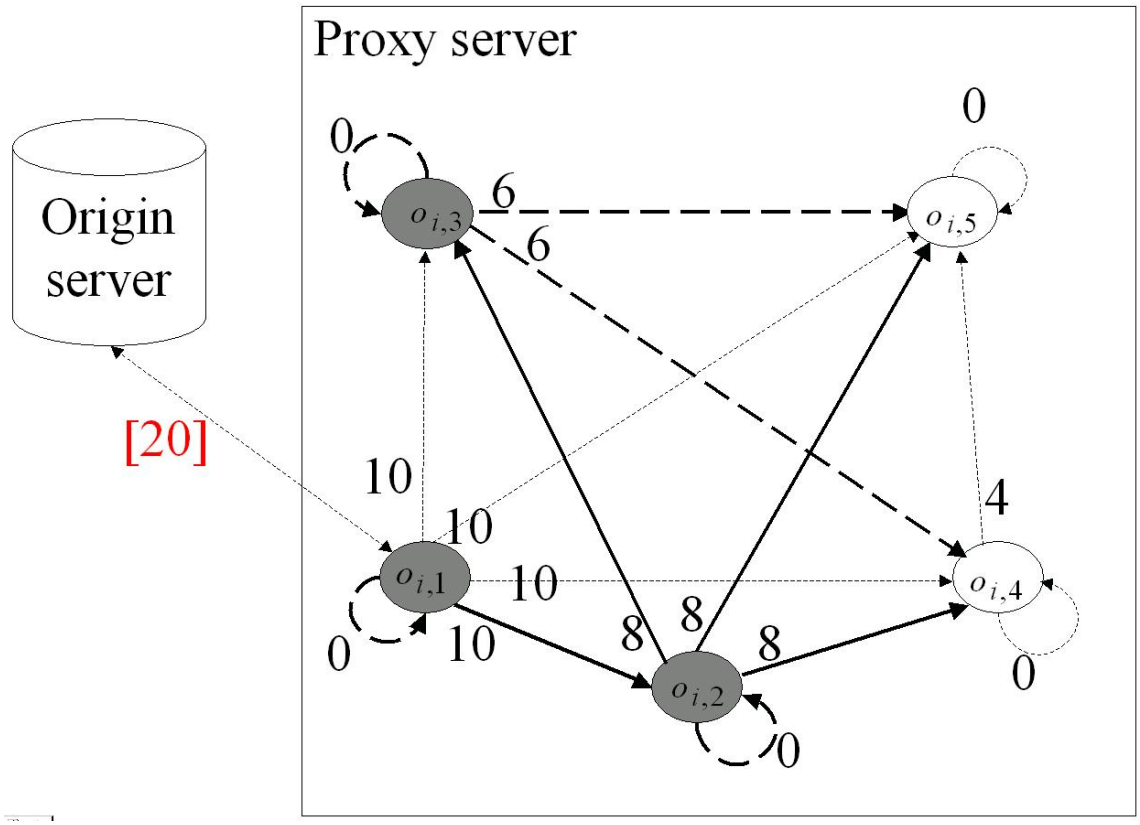

図 3.4: バージョン 3 を評価した後の関係形成図

3.3 提案関数

前節で提案したアルゴリズムは各バージョンが持っている $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ の値及び影響関係を構築した。本節ではその結果を利用し、キャッシュされたバージョンの節約コストを計算する関数を提案する。

各バージョンが $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ を持っている。$min\_cost2_{i,z}$ はキャッシュされたバージョンからバージョン z を生成するための画質調整コストが二番目小さいコストであり、もしバージョン z の生成バージョンが削除されれば、影響バージョンからの画質調整コストである。$min\_cost1_{i,z}$ はキャッシュされたバージョンからバージョン z を生成するための画質調整コストが一番小さいコストであり、バージョン z の生成バージョンからの画質調整コストである。$min\_cost2_{i,z}$ と $min\_cost1_{i,z}$ の差はバージョン z に対して一回のリクエストが来るときにバージョン z の影響バージョンからの影響関係を考慮した後のバージョン z の生成バージョンが節約したコストである。$r_{i,z}$ はオブジェクト i の z バージョンのアクセス率であり、$r_{i,z}$ が $min\_cost2_{i,z}$ と $min\_cost1_{i,z}$ の差をかけて得た値は影響バージョンからの影響を考慮した後のバージョン z に関しての節約コストである。キャッシュされたバージョンの節約コストを計算する際に、このキャッシュされたバージョンが頂点として構成された生成関係集合の各バージョンの影響関係を考慮した後の節約コストの和はこのキャッシュされたバージョンの節約コストである。この考えを基にして本研究のキャッシュされたバージョンの節約コストを計算する関数を公式化した（式 3.1 に示す)。式中のパラメータの説明には表 3.8 に示す。

表 3.8: 式 3.1 にあるパラメータの説明

$REC(o_{i,y})$	キャッシュされたバージョン y の節約コストを計算する関数
$G_{i,y}$	キャッシュされたバージョンの生成関係集合
z	バージョン z
$r_{i,z}$	オブジェクト i のバージョン z のアクセス率

$$REC(o_{i,y}) = \sum_{z \in G_{i,y}} r_{i,z} * (min\_cost2_{i,z} - min\_cost1_{i,z}). \tag{3.1}$$

$REC(o_{i,y})$ は本研究提案したキャッシュされたバージョン y の節約コストを計算する関数である。$G_{i,y}$ はキャッシュされたバージョンの生成関係集合である。z はバージョン z である。

式 3.1 にあるバージョンのアクセス率 $r_{i,z}$ が静的なものではなく、時間と共に変わっていく。例えば、あるバージョンが一回アクセスしたが、二時間が経ってもアクセス率がまだ 1 ではなく、時間が経つことと伴って、変化していく。本研究では文献 [15][16] が提案した *sliding average* 関数の概念を使うことにした。*sliding average* 関数が式 3.2 に示さ

れる。

$$r_{i,z} = \frac{k}{t_{i,z} - t^k_{i,z}} \tag{3.2}$$

sliding average 関数にあるパラメータの説明には表 3.9 に示す。

表 3.9: 式 3.2 にあるパラメータの説明

$t_{i,z}$	バージョン z の一番新しいリクエストが到着時間
k	アクセス回数
$t^k_{i,z}$	K 番目のリクエストの到着時間

時間 $t_{i,z}$ はバージョン z の一番新しいリクエストが到着時間である。k は新しいリクエストを含めてのアクセス回数である。時間 $t^k_{i,z}$ は一番新しいリクエストから K 番目のリクエストの到着時間である。アクセス率 $r_{i,z}$ を計算する時に k の値を多めに取れば取るほど、もっと正確のアクセス率を計算することができる。

3.4 提案関数の性能分析

本研究では生成バージョンの節約コストを計算する際に生成バージョンからコスト $min\_cost1_{i,z}$ がかかって生成関係集合にあるバージョン z が生成されることができる。もし生成バージョンがキャッシュから削除される場合には影響バージョンからコスト $min\_cost2_{i,z}$ がかかることによって同じことができる。そして生成バージョンがキャッシュされることによって $min\_cost2_{i,z}$ と $min\_cost1_{i,z}$ の差の分のコストを節約したこともすでに説明した。こういう考え方は常に生成バージョンがキャッシュに存在していない時の状況を考え、影響バージョンからの影響を考慮し、適切な生成バージョンの節約コストを計算することができ、最小な節約コストの生成バージョンが削除されることによってキャッシュに最大な節約コストが保たれ、第 2 章で紹介した従来研究 [8] のような不都合なことが生じることができなくなり、総合節約コストが上がると考えられる。

キャッシュヒット（HIT) は EHIT と UHIT の二つの状況にある。表 3.10 で示している。EHIT はクライアントからリクエストされたオブジェクトのバージョンがキャッシュに存在す

表 3.10: 本研究のキャッシュ判断状況の説明

$EHIT$	オブジェクトのバージョンが直接にキャッシュにある
$UHIT$	オブジェクトのバージョンが画質調整による生成できる
HIT	EHIT と UHIT の二つの状況の総合
キャッシュミス	EHIT と UHIT の以外の場合

る時に発生したキャッシュヒットである。UHIT はクライアントからリクエストされたオブジェクトのバージョンがキャッシュに存在しないが、生成バージョンがキャッシュされたので画質調整を通して生成できる時に発生したキャッシュヒットである。以上のキャッシュヒット以外の場合にはキャッシュミスである。本研究で提案した影響関係を構成するアルゴリズムを用い、構成した関係図は図3.4 に示す。算出した $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ の値は表3.7 に示す。もし各バージョンのアクセス率が全部 10 の場合には前節で提案した関数式3.1 で各生成バージョンの節約コストを計算するとバージョン 1 の節約コスト $REC(o_{i,1})$ は 10*(20-0) $(r_{i,1}*(min\_cost2_{i,1}-min\_cost1_{i,1}))$ であり、3.1 節の分析と同じに 200 である。バージョン 2 の節約コスト $REC(o_{i,2})$ は 10*(10-0)$(r_{i,2}*(min\_cost2_{i,2}-min\_cost1_{i,2}))$ であり、3.1 節に示したように 100 である。バージョン 3 の節約コスト $REC(o_{i,3})$ は 10*(8-0)+10*(8-6)+10*(8-6)$(r_{i,3}*(min\_cost2_{i,3}-min\_cost1_{i,3})+r_{i,4}*(min\_cost2_{i,4}-min\_cost1_{i,4})+r_{i,5}*(min\_cost2_{i,5}-min\_cost1_{i,5}))$ であり、3.1 節の計算結果と同じように 120 である。バージョン番号が大きいなバージョン（例えばバージョン 4、バージョン 5）に対しての節約コストを計算する時に影響バージョンからのコスト $min\_cost2_{i,z}$ と生成バージョンからのコスト $min\_cost1_{i,z}$ の差が小さくて、アクセス率がちょっと大きくなっても、バージョン番号が大きいな生成バージョンの節約コストがあがらないことが分かった。この分析によって本研究の全体ヒット率が高くなるが、キャッシュヒットの二つの状況である UHIT と EHIT では UHIT のヒット率が高く、EHIT のヒット率が低くなることが考えられる。

3.5 まとめ

本章ではまず、第 2 章の事例を用い、バージョン間の影響関係を分析した。分析した結果のもとに 3.2.1 節に本研究は生成バージョン、影響バージョン及び影響を記録するパラメータ $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ などの研究用語とパラメータを定義した。3.2.3 節では定義した研究用語と二つのパラメータ $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ を用い、本研究の影響関係を構成するアルゴリズムを提案した。また、提案したアルゴリズムで計算した $min\_cost1_{i,z}$ と $min\_cost2_{i,z}$ の値と構成した関係を用い、3.3 節で本研究が生成バージョンの節約コストを計算する関数を公式化した。3.4 節では提案手法の性能を分析し、総合節約コストが高くなることと UHIT のヒット率が高くなり、EHIT 率が低くなることが分かった。

第4章　Ns2による実装

Ns2 は，カリフォルニア大学バークレイ校で開発され、C++と OTcl で書かれたオブジェクト指向のイベントドリブン・ネットワークシミュレータである。広域ネットワークをシミュレートするのに役立つ。本研究は WAP プロキシネットワークシステムをシミュレーションするために Ns2 を用いシミュレーションを行う。

4.1　キャッシュ判断の実装

本項では、Ns2 のシステムにもともと入っているキャッシュ仕組みと本研究の違いを紹介し、そして本研究の目的にそって画質調整機能のシミュレーション部分の実装について述べる。

4.1.1　NS2 のキャッシュ仕組み

シミュレータ Ns2 の中にプロキシサーバをシミュレーションすることができる。プロキシサーバはクライアントからリクエストを来るたびに図 4.1 のようにキャッシュを保存するハッシュテーブルに問い合わせをしキャッシュの存在を判断する。

図 4.1 の仕組みは Ns2 の中で OTcl で書かれている。このキャッシュ仕組みはまず、クライアントのリクエストを受け取り、リクエストされたオブジェクトがキャッシュに存在しているかどうかをキャッシュに尋ね、存在している場合にはキャッシュヒットであり、キャッシュからオブジェクトを取り出した後にクライアントにレスポンスをする。リクエストされたオブジェクトがキャッシュに存在していない場合にはキャッシュミスであり、クライアントの代わりにオリジナルサーバにリクエストを出し、オリジナルサーバからオブジェクトが返されてきた後にクライアントにレスポンスをする。

本研究の場合にはキャッシュヒット（HIT) は EHIT と UHIT の二つの状況にある。表 4.1 で示している。EHIT はクライアントからリクエストされたオブジェクトのバージョンがキャッシュに存在する時に発生したキャッシュヒットである。UHIT はクライアントからリクエストされたオブジェクトのバージョンがキャッシュに存在しないが、生成バージョンがキャッシュされたので画質調整を通して生成できる時に発生したキャッシュヒットである。以上のキャッシュヒット以外の場合にはキャッシュミスである。Ns2 のキャッシュ判断仕組みと比べると異なるので、本研究の目的に沿って実装する必要がある。次節では Ns2 の中の本研究のキャッシュ判断仕組みの実装について紹介する。

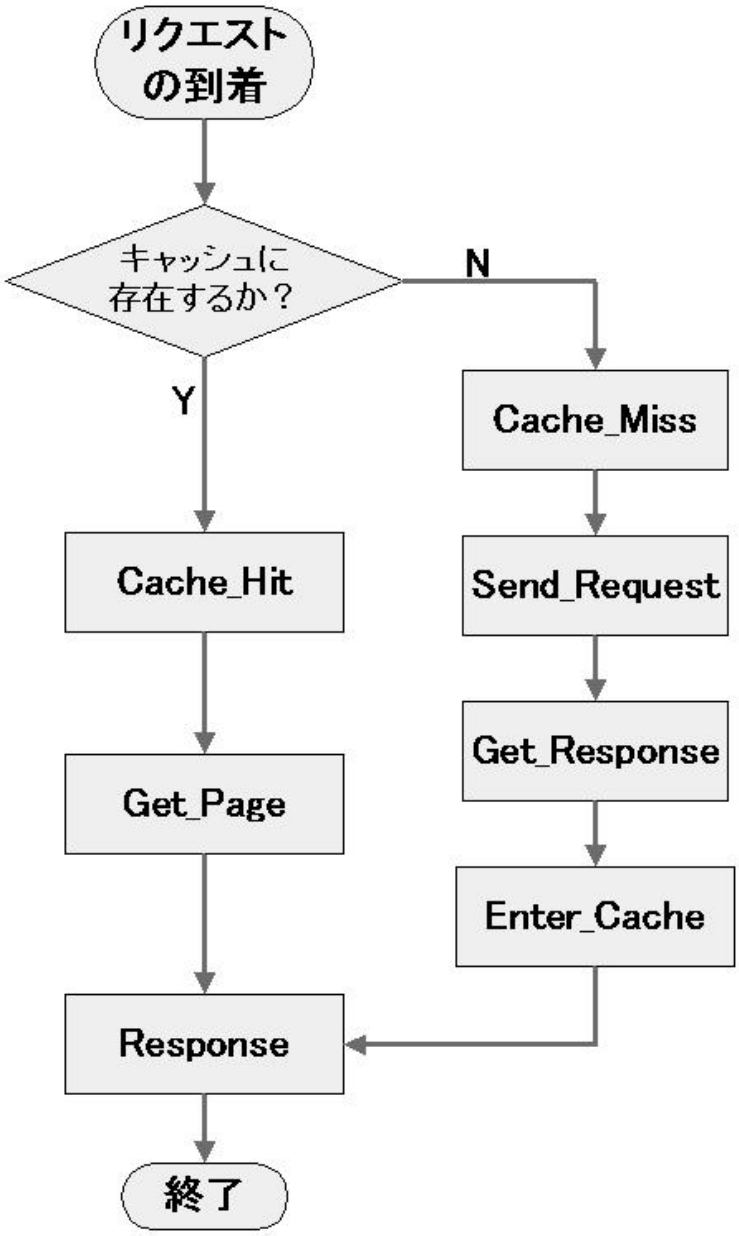

図 4.1: Ns2 のキャッシュ判断仕組み図

表 4.1: 本研究のキャッシュ判断状況の説明

EHIT	オブジェクトのバージョンが直接にキャッシュにある
UHIT	オブジェクトのバージョンが画質調整による生成できる
HIT	EHIT と UHIT の二つの状況の総合
キャッシュミス	EHIT と UHIT の以外の場合

4.1.2 キャッシュ判断の実装

本研究はキャッシュヒットが二つの状況に分かれたので、Ns2 を用いシミュレーションをする場合にはフローチャート図 4.2 を示したようにキャッシュ判断の実装をした。

本研究はキャッシュに問い合わせする際に、三つの状況に分けて考え、判断する。その処理は *Procedure Cache_Exist* に示す。

Procedure Cache_Exist(char *name,int ver)

1: リクエストされたオブジェクトの *name* をキーとしてキャッシュに問い合わせる
2: **if** リクエストされたオブジェクトがキャッシュに存在しない **then**
3: return 0;
4: **end if**
5: キャッシュからオブジェクトを取り出す
6: **if** リクエストされたバージョンがキャッシュにある **then**
7: リクエスト回数を一回増やす
8: return 1;
9: **end if**
10: **if** リクエストされたバージョンの生成バージョンがキャッシュにある **then**
11: return2;
12: **end if**
13: return 0;

Procedure Cache_Exist は二つの引数を持っている。一つはオブジェクトの識別子 *name* であり、実ネットワークの *URL* に相当する。二つ目では *ver* であり、オブジェクトのバージョン番号を表している。キャッシュされたオブジェクトへのポインタがハッシュテーブルに保存される。まず、オブジェクト *name* をキーとしてハッシュに問い合わせる。もしオブジェクトへのポインタがハッシュに存在しない場合には、キャッシュミスであり、0 をリターンし、処理が終了する。存在している場合には、ハッシュテーブルからオブジェクトへのポインタを取り出す。次には、リクエストされたバージョンがオブジェクトに存在しているかを判断する。存在している場合には前節で述べたように *EHIT* であり、リクエスト回数が一つカウントされ、1 をリターンした後に、処理が終了する。存在しない場

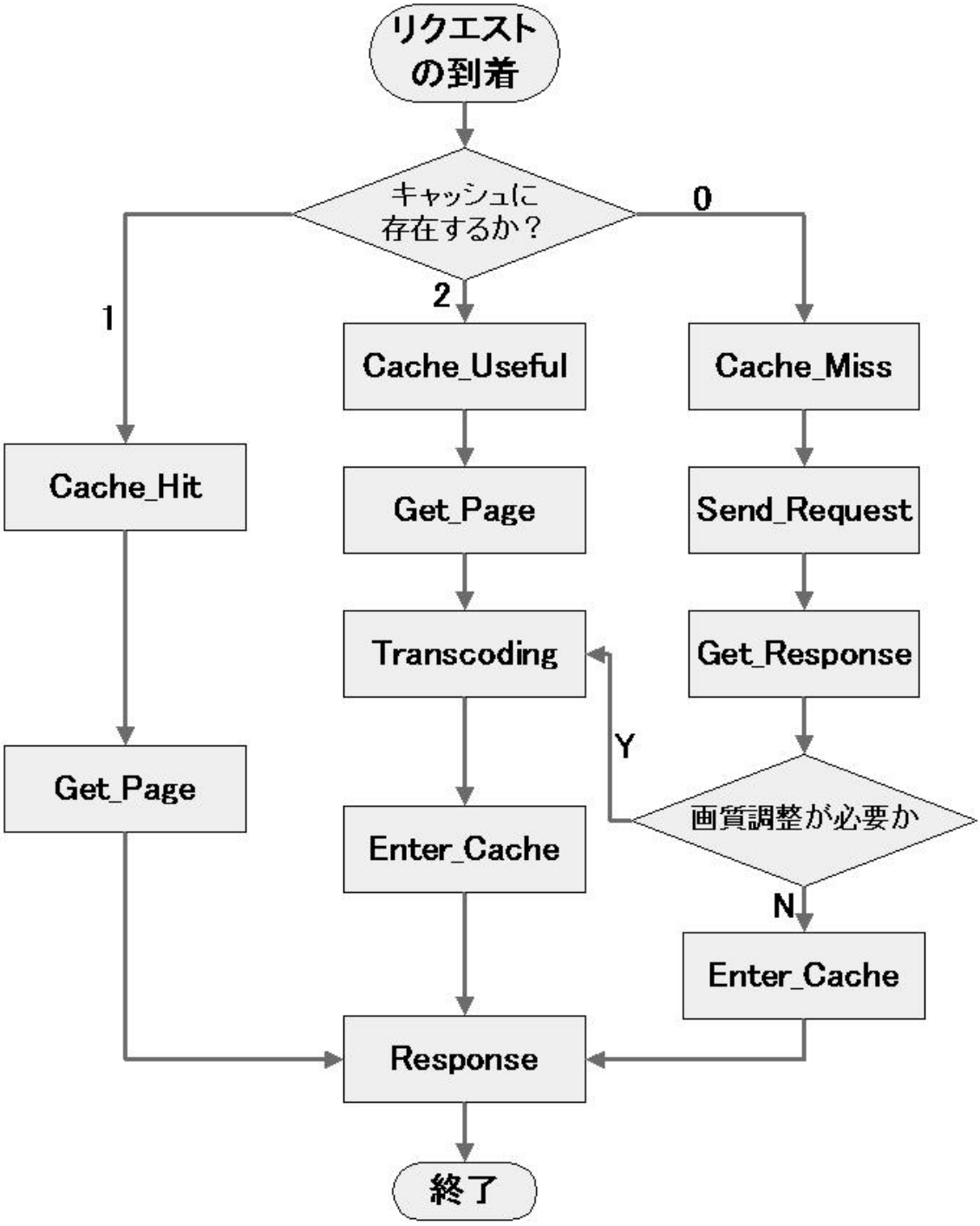

図 4.2: 本研究のキャッシュ判断仕組みの実装図

合にはこのバージョンの生成バージョンが存在してるかを判断する。生成バージョンが存在している場合には前節紹介した*UHIT*であり、2をリターンし、処理が終了する。生成バージョンも存在しなければ、キャッシュミスであり、0をリターンし、処理が終了する。

*Procedure Cache_Exist*の戻り値にのもとに処理が三つの状況に分かれている。一つ目にはEHITであり、Ns2の元処理のキャッシュヒットと同じように判断している。二つ目にはリクエストされたオブジェクトのバージョンの生成バージョンが存在している、この場合にはUHITであり、生成バージョンをキャッシュから取り出し、画質調整を行うことによってリクエストされたバージョンが生成され、クライアントに応答されると同時にキャッシュに保持する。三つ目の場合にはリクエストされたオブジェクトのバージョンの生成バージョンも存在していない場合にはキャッシュミスであり、クライアントの代わりにオリジナルサーバにリクエストを送り出す。オリジナルサーバにはオリジナルな画像しかないので、ここでのリクエストはオリジナル画像に対するリクエストである。オリジナルサーバからオリジナルな画像を送ってきた後に画質調整をする必要があるかを判断する。オリジナル画像（バージョン1）がリクエストされる場合にはそのままクライアントに応答する。画質調整をする必要がある場合には画質調整を行った後にクライアントに応答を返し、キャッシュに保持する。

4.2 置き換えアルゴリズム部分の実装

4.2.1 キャッシュオブジェクトの実装

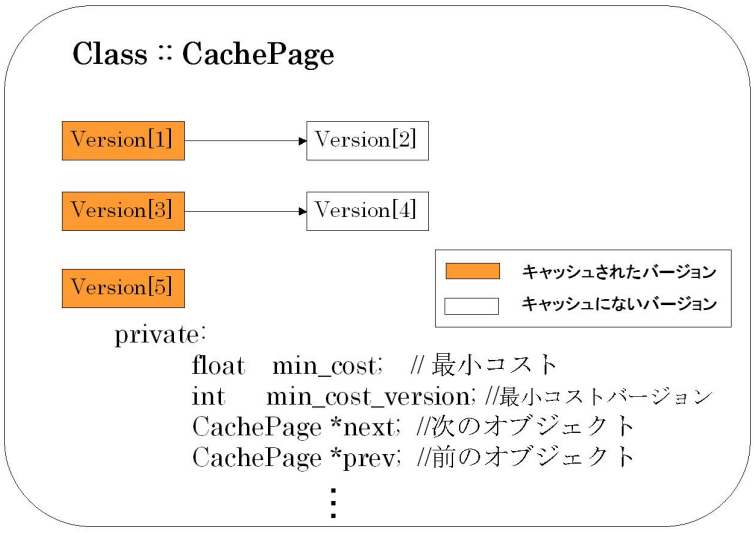

図 4.3: キャッシュされた各オブジェクト及び各バージョンのデータ構造

Ns2はC++言語を用いプロキシサーバのオブジェクトのキャッシュ仕組みを実装した。

Ns2 の中にはもともとキャッシュ空間のサイズに対しての制限がもうけられないので、リクエストされたオブジェクトを全部キャッシュに入れられる。また Ns2 のプロキシサーバのオブジェクトのキャッシュ仕組みはバージョンが存在しないない状況を前提として実装されたが、本研究は一つのオブジェクトに対して画質調整された前後に複数のバージョンが存在する。そこで、本研究は独自のオブジェクトと各バージョンのデータ構造を記述するクラス CachePage を図 4.3 のように実装した。

表 4.2: 各バージョンのデータ構造

$exist\_$	キャッシュに存在しているか
$size\_$	バージョンのデータサイズ
$cost\_$	節約コスト
$ri\_$	アクセス回数
$r\_$	アクセス率
$min\_cost1\_$	キャッシュされたバージョンからの最小画質調整コスト
$min\_cost2\_$	キャッシュされたバージョンからの二番目最小画質調整コスト
$tran\_version\_$	画質調整し自分を生成できるキャッシュされたバージョン番号
$time\_p\_$	キャッシュされた時の時刻
$next\_$ と $prev\_$	生成関係を動的に構成するための構造体へのポインタ

図 4.3 にある Version は各バージョンの情報を記述する構造体である。各バージョンが持っているデータ構造は表 4.2 に示している。$exist\_$ は int 型のパラメータであり、このバージョンがキャッシュに存在しているかのことを示す。キャッシュされる場合には 1 であり、キャッシュされない場合には 0 である。このパラメータはキャッシュ判断をする際に使われる。$size\_$ は $float$ 型のパラメータであり、バージョンのデータサイズを記録している。$cost\_$ は $float$ 型のパラメータであり、このバージョンがキャッシュされる場合にはキャッシュされることによって節約したコストである。$ri\_$ は int 型のパラメータであり、このバージョンがアクセスされた回数である。初期値は 0 である。$r\_$ は $float$ 型のパラメータであり、バージョンのアクセス率を記録している。$min\_cost1\_$ と $min\_cost2\_$ は $float$ 型のパラメータであり、本研究によって定義された相互作用を記述するパーらメタである。$tran\_version\_$ は int 型のパラメータであり、もしこのバージョンが存在していなければ、画質調整によってこのバージョンを生成できるキャッシュされたバージョンの番号であり、キャッシュ判断をする時に使われる。$time\_p\_$ は $float$ 型のパラメータであり、バージョンをキャッシュに入れる時のシステムの時刻であり、キャッシュ率を算出する際に用いられる。$next\_$ と $prev\_$ は構造体へのポインタであり、各バージョン間の生成関係を動的に構成するときに用いられる。本研究はバージョン間の生成関係を双方向リストを用いて表現した。

クラス CachePage の一つのインスタンスは一つのオブジェクトである。CachePage クラスは Ns2 本来のオブジェクト情報の他に Version 構造体の配列を持っている。図 4.3 に

は構造体配列は五つ（Version[0] が使われていない）の構造体から構成されている時の例であることで一つのオブジェクトは五つのバージョンを持っていることを示している。この例から見ると Version[1]、Version[3] と Version[5] がキャッシュされ、画質調整によって Version[1] から Version[2]、Version[3] から Version[4] が生成される関係を表している。また本研究にはクラス CachePage に Ns2 が本来に持っているオブジェクト情報の他に幾つかのクラス変数を持たせている。表 4.3 を用い説明していく。min_cost は $float$ 型のパラ

表 4.3: CachePage クラス変数の説明

min_cost	節約コストが最小であるキャッシュされたバージョンのコスト
$min\_cost\_version$	節約コストが最小キャッシュされたバージョンの番号
$next\_$ と $prev\_$	CachePage クラスへのポインタ

メータであり、一つのオブジェクトの各キャッシュされたバージョンの中に節約コストが最小であるキャッシュされたバージョンのコストである。$min\_cost\_version$ は int 型のパラメータであり、一つのオブジェクトの中に最小節約コストを持っているキャッシュされたバージョンの番号である。この二つのパラメータはキャッシュ空間のサイズが今キャッシュされた各バージョンのサイズの総和より小さい場合にはキャッシュ置き換えをし、節約コストが最小であるバージョンをキャッシュから削除するとき使われる。$next\_$ と $prev\_$ は CachePage クラスへのポインタであり、キャッシュされたオブジェクトの双方向リスト構造を作る際に使われる。

4.2.2 提案手法の実装

シミュレータ Ns2 の中にキャッシュの置き換えアルゴリズムが実装されないので、本研究は C++言語を用い、提案した手法を Ns2 に実装した。実装する全体の像は図 4.4 に示す。図 4.4 にあるパラメータの説明が表 4.4 に示す。$o_{i,y}$ は今キャッシュに入れるバージョ

表 4.4: 図 4.4 にあるパラメータの説明

$o_{i,y}$	キャッシュに入れるバージョン y
$s_{i,y}$	バージョン y のデータサイズ
$Cached\_size$	キャッシュにあるバージョンの総合サイズ
$Cache\_size$	キャッシュ空間のサイズ
$s_{j,x}$	キャッシュから削除されたオブジェクト j のバージョン x のデータサイズ
$o_{j,y}$	キャッシュされたオブジェクト j のバージョン x

ン y のことである。$s_{i,y}$ はバージョン y のサイズである。$Cached\_size$ はキャッシュにされ

た全部バージョンのサイズ和である。バージョンにキャッシュを入れるときに$Cached_size$の値が増える。$Cache_size$はキャッシュ空間のサイズである。$o_{j,x}$はキャッシュされたオブジェクトのjのバージョンxである。$s_{j,x}$はキャッシュから削除されたオブジェクトjのバージョンxのデータサイズである。

図4.4に示すようにキャッシュにあるオブジェクトのバージョンyをキャッシュに入れるときに、まずキャッシュに入れるバージョンyのサイズ$s_{i,y}$を$Cached_size$にプラスする。次に本研究で提案した影響関係を構築するアルゴリズムを実行され、バージョン間の関係及び定義したパラメータ$min_cost2_{i,z}$と$min_cost1_{i,z}$の値を計算する。この部分が$Creat_Relation$関数による実装された。具体的なアルゴリズム構成には第3章のフローチャート図3.1及び$Procedure\ Creat_Relation$に示される。そして、キャッシュされたバージョンの総合サイズ$Cached_size$がキャッシュ空間のサイズ$Cache_size$と比較する。もしキャッシュされたバージョンの総合サイズ$Cached_size$がキャッシュ空間のサイズ$Cache_size$より小さい場合にはキャッシュ空間は余裕があり、キャッシュの置き換えを行う必要がなく、処理がそのまま終了する。もしキャッシュされたバージョンの総合サイズ$Cached_size$がキャッシュ空間のサイズ$Cache_size$より大きい場合にはキャッシュに入れないことが示され、キャッシュの置き換えを行う必要がる。この場合には、本研究で第3章3.3節に提案した関数$REC()$を用い、キャッシュされた各バージョンの節約コストを計算する。この部分が$Procedure\ Rec_Function$による実装された。$Procedure\ Rec_Function$にあるパラメータの説明には表4.5に示す。

表 4.5: CachePage クラス変数の説明

min_cost	オブジェクトの節約コストが最小であるキャッシュされたバージョンのコスト
$min\_cost\_version$	あるオブジェクト内に節約コストが最小キャッシュされたバージョンの番号

Procedure Rec_Function

1: オブジェクトiのキャッシュされたバージョンを全部キュー1に入れる

2: **while** キュー1 ! = NULL **do**

3: キュー1から一つのバージョン$o_{i,y}$を取り出す

4: $o_{i,y}$の生成集合に属しているバージョンを全部キュー2に入れる

5: **while** キュー2 ! = NULL **do**

6: キュー2から一つのバージョン$o_{i,z}$を取り出す

7: $o_{i,y}$の$cost+ = o_{i,z}$の$r_{i,z} \times (min_cost2_{i,z} - min_cost1_{i,z})$

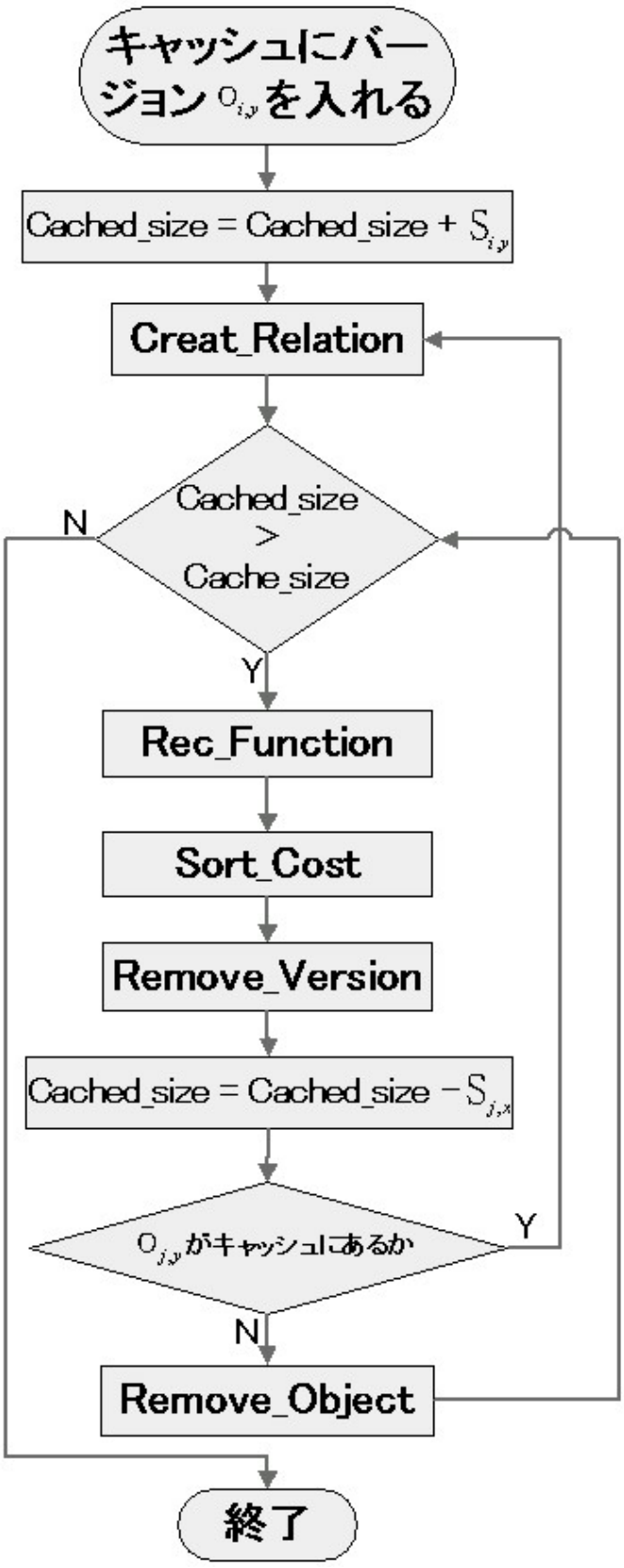

図 4.4: キャッシュ置き換えアルゴリズムの実装

8: **end while**

9: **if** $min\_cost == -1$ **then**

10: $min_cost = o_{i,y}$の $cost$;

11: $min\_cost\_version = o_{i,y}$

12: **else if** $min_cost > o_{i,y}$の $cost$; **then**

13: $min_cost = o_{i,y}$の $cost$;

14: **end if**

そして、$Sort\_Cost$ 関数を実装し、各オブジェクトが持っている min_cost のもとに節約コストをソートし、節約コストが一番小さいバージョンを取り出す。節約コストが一番小さいバージョンが $Procedure\ Remove\_Version$ による削除される。$Procedure\ Remove\_Version$ は一番小さいバージョン $(min\_cost\_version)$ をキャッシュから削除した後に、このオブジェクトが持っているすべてのバージョンがキャッシュに保持されているかを判断する。まだキャッシュにある場合 $(exist\_ == 1)$ には $Creat\_Relation$ 関数を呼ぶだし、関係を再構築し、0 が戻り値として返される。キャッシュにない場合には -1 が返される。

Procedure Remove_Version

1: バージョン $min\_cost\_version$ の $exist\_$ に 0 を代入する

2: **for all** version **then**

3: **if** version の $exist\_ == 1$ **then**

4: $Creat\_Relation$ 関数を呼ぶだし、関係を再構築する

5: return 0;

6: **end if**

7: **end for**

8: return -1;

$Cached\_size$ から削除されたバージョンのバージョンサイズ $s_{j,x}$ を引く。削除されたバージョンが最初にキャッシュに入れたバージョンと同じオブジェクトであることが分からないので、ここで削除されたバージョンのオブジェクトが j にした。次に、$Procedure\ Remove\_Version$ の戻り値に基づき、処理が進む。戻り値が 0 の場合には $Create\_Relation$

関数を呼び出し、影響関係とパラメータ $min\_cost2_{i,z}$ と $min\_cost1_{i,z}$ の値を計算しなおす。戻り値が 0 でなければ $Remove\_Object$ 関数が呼び出され、オブジェクト j を削除される。その後にまたキャッシュされたバージョンの総合サイズ $Cached\_size$ がキャッシュ空間のサイズ $Cache\_size$ と比較する。こういうようにキャッシュされたバージョンの総合サイズ $Cached\_size$ がキャッシュ空間のサイズ $Cache\_size$ より小さくなるまで繰り返し処理をする。

4.3 まとめ

本章ではまず、NS2 のキャッシュ判断仕組みについて分析し、本研究との異なりがあることが分かった。そして、4.1.2 節で EHIT と UHIT について説明し、それを判断できるキャッシュ判断仕組み $ProcedureCache\_Exist$ を実装した。4.2.1 節ではキャッシュオブジェクトを表すクラスを実装した、一つのオブジェクトクラスが持っているクラス変数及び五つのバージョン構造体について説明した。このクラスを用い 4.2.2 節で本研究の提案手法を幾つかの関数にわたって実装した。

第5章　実験と結果

5.1　概要

本章では、第2章に例として挙げた多様なクライアントがWAPプロキシサーバを経由しWebサーバにアクセスするシステムをシミュレーションし、色々な状況を考慮した実験を行った。

本研究の比較する対象となる従来研究アルゴリズムは表5.1に示している。AEアルゴリズムは第2章で紹介された文献[8]が提案したアルゴリズムである。RECは本研究による提案されたアルゴリズムである。

表 5.1: 評価の対象となるアルゴリズム一覧

Algorithm	Description
AE	文献[8]のアルゴリズム
REC	本研究が提案したアルゴリズム

評価する項目としては節約したコスト比率をあらわすDSR（Delay Saving Ratio)とキャッシュヒット率（HR)を用いる。DSRは節約した総合コストと発生した全部のコストの比率であり、キャッシュが存在していない状態で計算される。キャッシュヒット率の場合にはUHR（Useful Hit Ratio)、EHR（Exact Hit Ratio)とHR（Hit Ratio)の三つの状況に分けて分析していく。EHRはリクエストされたバージョンが直接キャッシュにある場合のキャッシュヒット率である。UHRはリクエストされたバージョンが直接キャッシュにないが、画質調整を行うことによってこのバージョンを生成できるバージョンがキャッシュにある場合のキャッシュヒット率である。HRは以上の二つのキャッシュヒット率の総和である。

5.2　ネットワークトポロジー

シミュレーションを行ったネットワークトポロジーのネットワーク図は5.1に示す。ノードSはオリジナル画像を保有するWebサーバを意味している。ノードEはクライアントとWebサーバの間に存在するWAPプロキシサーバを示している。WAPプロキシサーバとWebサーバの帯域幅は10Mbpsにした。ノードU1からノードU5まではモバイル端末

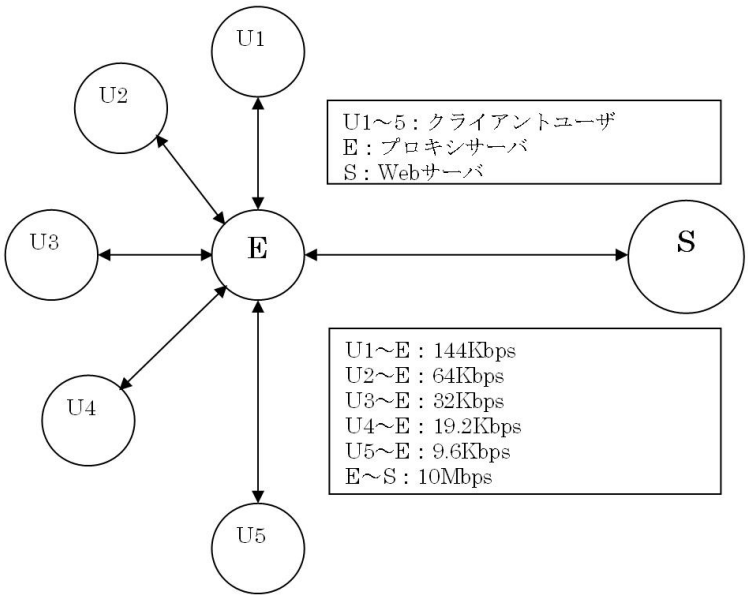

図 5.1: システムネットワーク図

を表すクライアント側である。WAP プロキシとアクセスしてくるクライアント間の帯域幅の多様化をシミュレーションするために文献 [2] に紹介された移動系ネットワークの種類に従ってクライアントが五つのクラスに分けられた。各クラスについての記述は表 5.2 を用い紹介する。

表 5.2: クライアントのタイプ

$ClassNO$	$Service$	Bandwidth
1	$IMT-2000$	144Kbp
2	PHS	64Kbps
3	$cdmaOne, PHS$	32Kbps
4	PDC	19.2Kbps
5	$GSM, CDPD$	9.6Kbps

クラス 1 は ITU（世界電気通信連合）に標準化が進められた IMT（international mobile telecommunications)-2000 技術（第 3 世代移動通信システム）である。この技術が目指しているのは、1, 世界中で同じ端末が利用できること、2, 固定通信網に相当する通信品質の高さ、3, 最高 2Mbps のデータ通信速度などである。現在主な IMT-200 のサービスとして転送速度が 144Kbps である MC-CDMA、転送速度が 384Kbps である DS-CDMA と転送速度最大 2.4Mbps の EV-DO などがある。PHS（Personal Handyphone System）はピッチとも呼び、移動先で使用できる、携帯可能な小型電話機。また、同電話機による移動体通信サービスの事を言う。この規格のサービスが色々提供され、転送速度もそれぞれであるが、文献 [2] によると転送速度は 32Kbps〜128Kbps である。本研究にはクラス 2 は

PHS サービスの転送速度が 64Kbps にした。クラス３は転送速度の 32Kbps である PHS と携帯電話の cdmaOne にした。クラス４は携帯電話の PDC（Personal Digital Cellular）である。PDC (Personal Digital Cellular) は、日本で開発され、日本国内で利用される FDD-TDMA の第二世代携帯電話の通信方式の一つである。クラス５はまだ機能をしている過去の転送速度が低いシステム（GSM、CDPD）などをシミュレーションするために設定した。

5.3 画質調整ポリシー

シミュレーションしたネットワークトポロジーに合わせて文献 [8][9] のように一つのオブジェクトに対して細かく五つのバージョンに分けれて画質調整が行われた。五つのバージョンのデータサイズは元の画像の 100%,80%,60%,40%,20%にした [8]。各バージョン間の画質調整の関係は図 5.2 に示す。バージョン１はオリジナル画像でもあり、各バージョンを生成することができる。バージョン２はバージョン１に戻れないが、自分より番号が大きいバージョンを生成することができる。バージョン３、バージョン４はバージョン２と同じである。バージョン５は他のバージョンから生成されることができるが、他のバージョンを生成することができない。実験ではクライアントがオブジェクトをアクセスするときに各バージョンの選択がランダムで行われる。

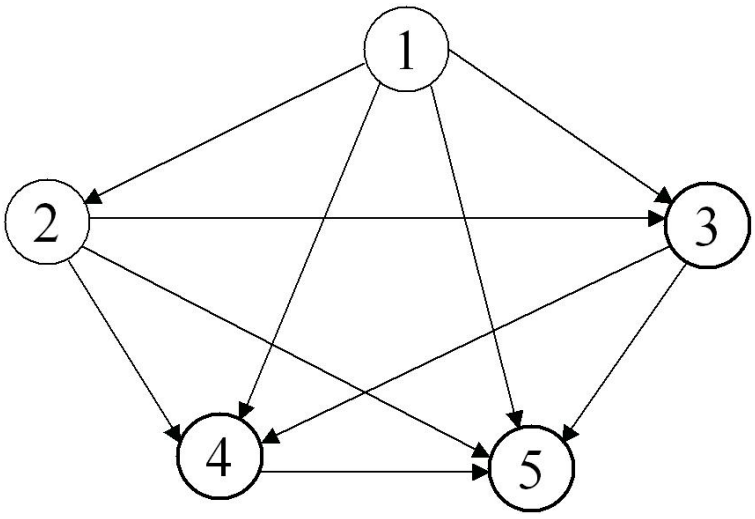

図 5.2: バージョン間の関係

5.4 各パラメータの定義

各パラメータの定義とデフォルト値は表 5.3 のようになっている。

表 5.3: 各パラメータ設定表

Parameter	Distribution\Default value
object_size	Pareto dist. ($\mu = 20KB$)
Number of Web object	N = 100
アクセス回数	6000 回
Object popularity(reference rate)	Zipf-like dist.(s = 0.7)
Cache_capacity	$0.05 * \sum_{k=1}^{100} object\_size$
transcode_ratio	20KB/sec
d_i	Exp dist. ($\mu = 2sec$)

オブジェクトの合計サイズは生成された各オブジェクトサイズの合計である。実験では 100 個のオブジェクトを生成し、シミュレーションを行った。各オブジェクトのデータサイズ（object_size）は文献 [8] のようにパレート分布に従い、平均サイズを 20KB にした。本研究のオブジェクトの合計サイズが 2MB(100*20KB) である。

キャッシュ空間サイズ（cache_capacity）はキャッシュにキャッシュできるバージョンの能力を示すパラメータである。節約コスト率とキャッシュヒット率を大きく左右している。本研究のキャッシュ空間サイズのデフォールト値は式 5.1 に示すように 100 個のオブジェクトの合計サイズの 5%であり、100KB にした。k はオブジェクトの順位番号である。

$$cache\_capacity = 0.05 * \sum_{k=1}^{100} object\_size \tag{5.1}$$

画質調整の節約コストを計算する際に式 5.2 を用いることにした [6][8][9][13]。

$$transcode\_cost = \frac{version\_size}{transcode\_ratio} \tag{5.2}$$

画質調整の節約コストはそのバージョンのデータサイズと画質調整の変換率の商である。transcode_cost は画質調整の節約コストである。version_size はバージョンのデータサイズである。transcode_ratio は画質調整の変換率であり、文献 [8][9] のように画質調整の変換率を 20KB/sec にした。例えば、バージョン 1 のデータサイズが 20KB であれば、式 5.2 によってバージョン 1 の画質調整の節約コストは 1sec(20/20) である。

ジップの法則はウェブページへのアクセス頻度に適用できることがよく知られている [8][15][17][18][19]。数学的一般のジップの法則は式 5.3 である。

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s} \tag{5.3}$$

f(k;s,N) は k 番目のオブジェクトの人気度合である。k はオブジェクトの順位番号である。N はオブジェクトの合計の数であり、本研究に対しては 100（100 個のオブジェクト）で

ある。ｓはオブジェクト間の人気度合いの差を決めるパラメータであり、ジップ分布のグラフの形を左右する。1 の場合には元来のジップ法則であり、実験では文献 [8] の経験値 0.7 にした。この式によると順位一番のオブジェクト 1 の人気度合（アクセス回数）が一番高く、s が 0.7 の場合には、オブジェクト 1 のアクセス回数は全体アクセスの 9.5%である。本研究は 6000 のアクセスを生成したので、オブジェクト 1 のアクセス回数は 570 である。順位番号が高くなるにつれて人気度合が低くなる。各オブジェクトのアクセス回数の分布が図 5.3 に示す。

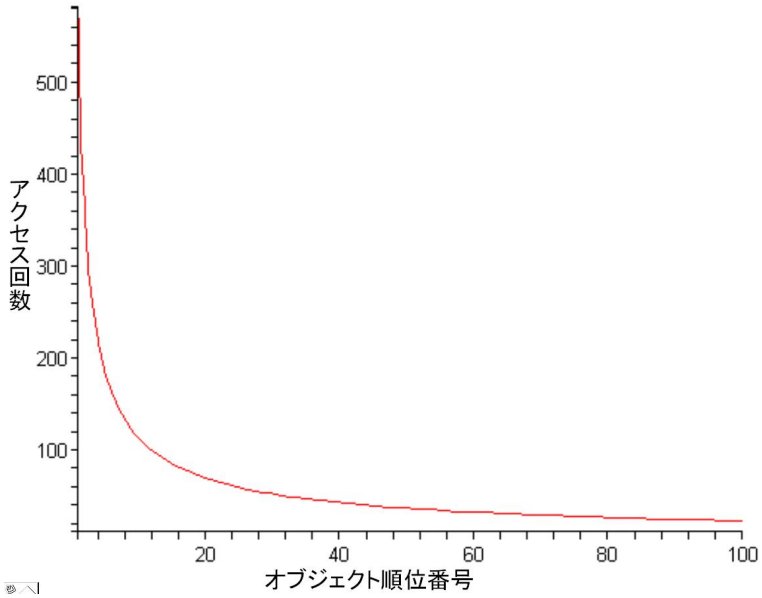

図 5.3: 各オブジェクトのアクセス回数分布

時間間隔などが指数分布 に従うので、本研究ではリクエストがプロキシサーバに到着時間間隔を指数分布にした。指数分布は平均と分散の二つの種類があるが、本研究はプロキシ到着時間間隔を一定の期待値にしたいので、平均の方にした（式 5.4)。

$$g(x) = \frac{1}{\mu} e(-x/\mu) \tag{5.4}$$

$g()$ は平均到着間隔時間が μ であるときに到着間隔時間 x の比率である。μ は時間間隔の平均値であり、分布の曲線の形を決めるパラメータである。x は到着間隔時間である。e は自然対数である。μ_1 がアクセス率が一番高いオブジェクト（オブジェクト 1）の到着間隔時間の平均値である。アクセス率が低くなるに伴って到着間隔時間期待値が長くなる。μ_k が k 番目のオブジェクトのプロキシサーバの到着間隔時間である。アクセス率と到着間隔時間が正比例にし、式 5.5 を用い μ_k を計算することにした。

$$\mu_k = \frac{f(1;s,N) * \mu_1}{f(k;s,N)} = \frac{\frac{1}{\sum_{n=1}^{N} 1/n^s} * \mu_1}{\frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}} = \mu_1 * k^s \tag{5.5}$$

本研究は μ_1 を 3 秒にした。そして、オブジェクト 1 のプロキシサーバに到着間隔時間が $3 * 1^{0.7}$ であり、3 秒である。オブジェクト 2 のプロキシサーバに到着間隔時間が $3 * 2^{0.7}$ であり、4.86 秒である。このようにオブジェクト 100 までのプロキシサーバに到着間隔時間を計算した。

5.5 節約コストによる評価

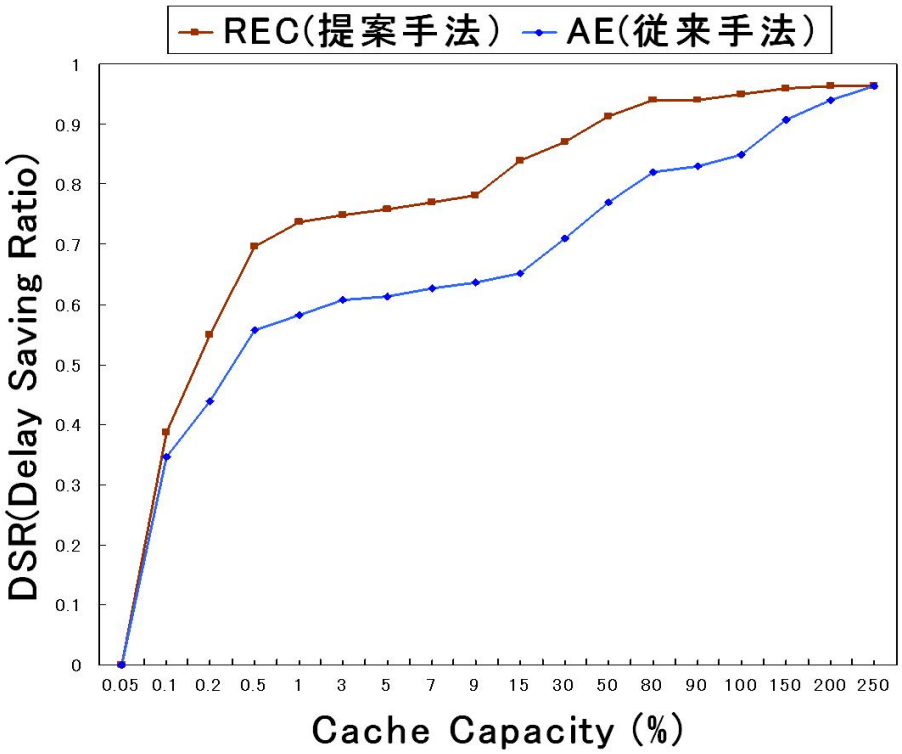

図 5.4: キャッシュサイズに変化による節約したコストの考察

実験ではまずキャッシュ空間サイズが大きくするたびに全体の DSR の変化状況について分析評価した。シミュレーション結果は図 5.4 に示している。このグラフの横軸ではキャッシュ空間領域を表している、本研究ではキャッシュ空間サイズがオブジェクト合計サイズの 0.05%から 250%までの間の DSR の変化について実験した。キャッシュ空間サイズがオブジェクト合計サイズの 0.05%の以下の場合には、オブジェクトの一番大きい番号のバージョンのサイズより小さく、何もキャッシュができいない状況になっているので、節約コスト率 (DSR) が 0 である。オブジェクトが五つのバージョンを持っているので、オブジェクトの合計サイズの 100%になっても全部のオブジェクトの各バージョンがキャッシュできることがなく、本研究の実験の状況に対してはオブジェクトの合計サイズの 250%になってから、従来研究と本研究の (DSR) が一緒になり、固定値になった。このことにより、本

実験に対してはキャッシュ空間サイズがオブジェクトの合計サイズの 250%の場合には、キャッシュ空間サイズが制限なく全部キャッシュすることになった。縦軸では DSR を表している。結果によると、本研究にて提案したアルゴリズム（REC) が節約したコストの比率（DSR）は各キャッシュ空間サイズがオブジェクト合計サイズの 0.05%から 250%までの間に従来研究と比べると高いことが分かった。

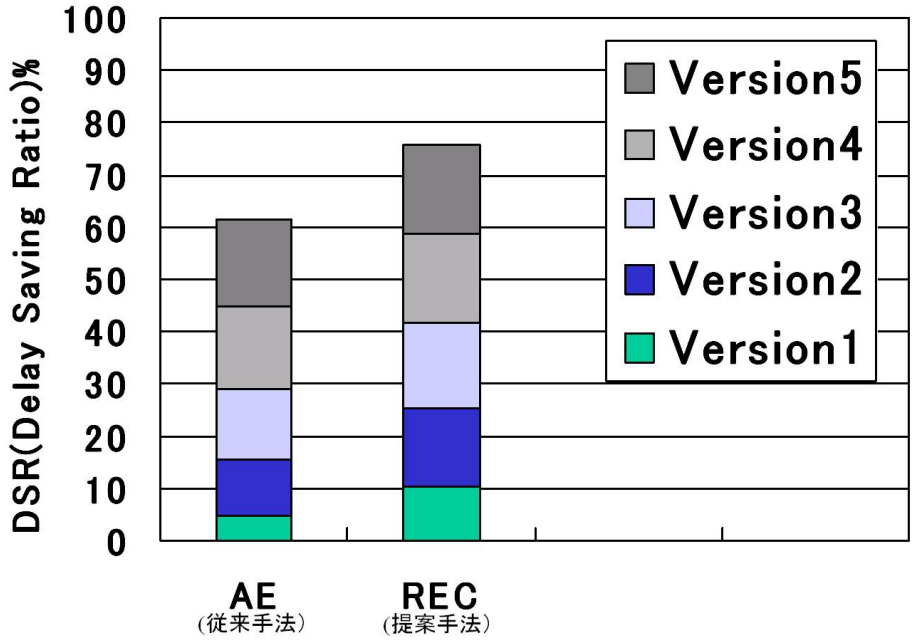

図 5.5: バージョン毎の節約コストの状況

表 5.4 ではキャッシュ空間サイズが大きくなるにつれて、提案手法と従来手法の節約コスト率を示した。

図 5.5 ではキャッシュ空間サイズがデフォルト（オブジェクトの合計サイズの 5%）である場合に各アルゴリズムが節約したコスト比率をバージョンごとに分割して分析した結果を示している。各アルゴリズムは低いバージョン番号から高いバージョン番号順にバージョンごとに節約したコストの比率が高くなる傾向であるが、本研究提案した REC アルゴリズムが各バージョンごとに節約したコストが従来研究より高いことが分かった。本研究の各バージョンごとの節約コスト率がバージョン 1 からバージョン 5 の順で、10.1%、15.2%、16.1%、17.2%、17.0%である、従来研究の各バージョンごとの節約コスト率がバージョン 1 からバージョン 5 の順で 4.7%、10.5%、13.5%、15.9%、16.6%である。

以上の結果は第 3 章の性能分析したように、本研究では常に生成バージョンがキャッシュに存在していない時の状況を考え、影響バージョンからの影響を考慮し、適切な生成バージョンの節約コストを計算することができ、最小な節約コストの生成バージョンが削除されることによってキャッシュに最大な節約コストが保たれ、第 2 章で紹介した従来研究 [8] のような不都合なことが生じることがなくなり、総合節約コスト率 DSR が高くなった。

表 5.4: 節約コスト率の実験結果

	提案手法	従来手法
0.1%	38.6%	34.7%
0.2%	55%	44.1%
0.5%	69.7%	55.7%
1%	73.7%	58.3%
3%	74.9%	60.8%
5%	75.9%	61.3%
7%	76.9%	62.6%
9%	78.1%	63.6%
15%	81.4%	65.1%
30%	87.0%	71%
50%	91.2%	77%
80%	94.0%	82%
90%	94.6%	83%
100%	94.9%	85%
150%	95.9%	90.7%
200%	96.3%	94%
250%	96.4%	96.4%

5.6　キャッシュヒット率による評価

全体のキャッシュヒット率（HR)、画質調整がいらないキャッシュヒット率 EHR（Exact Hit Ratio)、画質調整による UHR（Useful Hit Ratio）の三つの状況で細かくキャッシュヒット率を分析し、実験を行った。本項ではアルゴリズム毎に画質調整ポリシー毎にキャッシュヒット率およびバージョンごとのキャッシュヒット率についてのシミュレーション結果を紹介する。図 5.6 はキャッシュ空間サイズがデフォルト（オブジェクトの合計サイズ

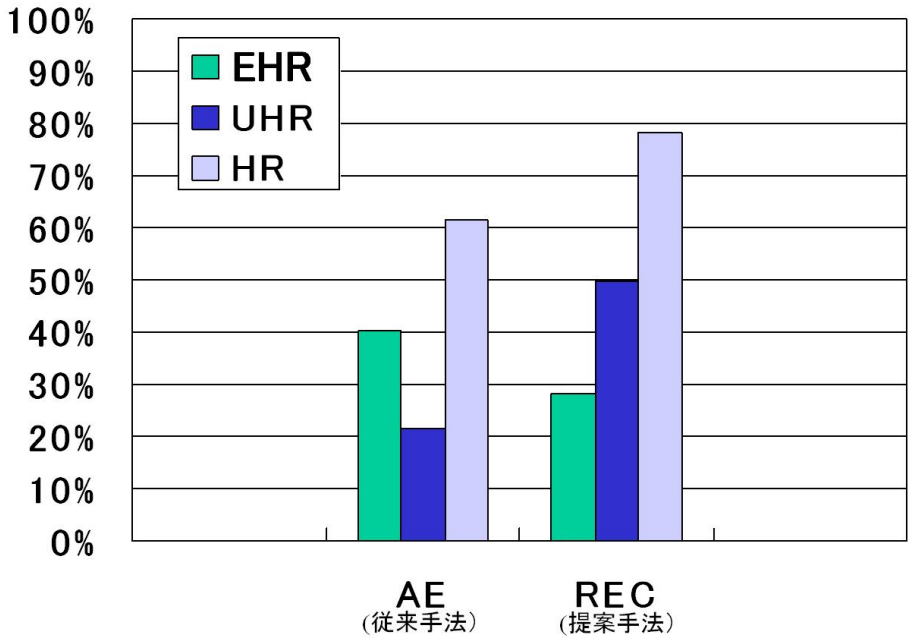

図 5.6: キャッシュヒット率

の 5%）である場合で画質調整ポリシー毎のキャッシュヒット率の実験結果を示す。本研究の REC アルゴリズムの HR は 78.1%であり、従来研究の AE アルゴリズムの 61.6%より高い、特に UHR を見ると、本研究の UHR が AE より高い。本研究の REC アルゴリズムの UHR が AE アルゴリズムより高い原因は第 3 章に分析したように複数のバージョンが同時にキャッシュされる場合には本研究ではキャッシュヒット率が同じな時にバージョン番号が小さいバージョンのコストが高く、キャッシュされると対して、AE アルゴリズムはバージョン番号が大きいバージョンのコストが高くキャッシュされるためである。本研究バージョン番号が小さいバージョンがキャッシュされるので、高いバージョン番号のバージョンがリクエストがされるときに、UHIT を発生し UHR が高くなる。本研究の場合にはバージョン間の相互関係を配慮し、各バージョンの節約したコストを適切に計算し、計算結果に基づき、そのバランスを取れた大きいバージョン番号のバージョンと小さいバージョン番号のバージョンをキャッシュすることによってキャッシュヒット率が高くなる原因である。これについてまた図 5.7、図 5.8 を用いバージョン毎の EHR および UHR を分

析し、実験の結果を紹介する。図 5.6 の EHR の部分を見ると従来研究の AE アルゴリズムが本研究の REC アルゴリズムより高い。その原因は本研究の REC アルゴリズムは影響関係を考慮しているので、影響バージョンからの節約コストと生成バージョンからの節約コストの差が小さくて、番号が大きいバージョンのアクセス率がちょっと大きくなっても、節約コストがそんな上がらないためにキャッシュに保持できなくなるからである。しかし、その分で UHR が高く、キャッシュミスが少なくなり、全体の節約コスト率が高くなる。

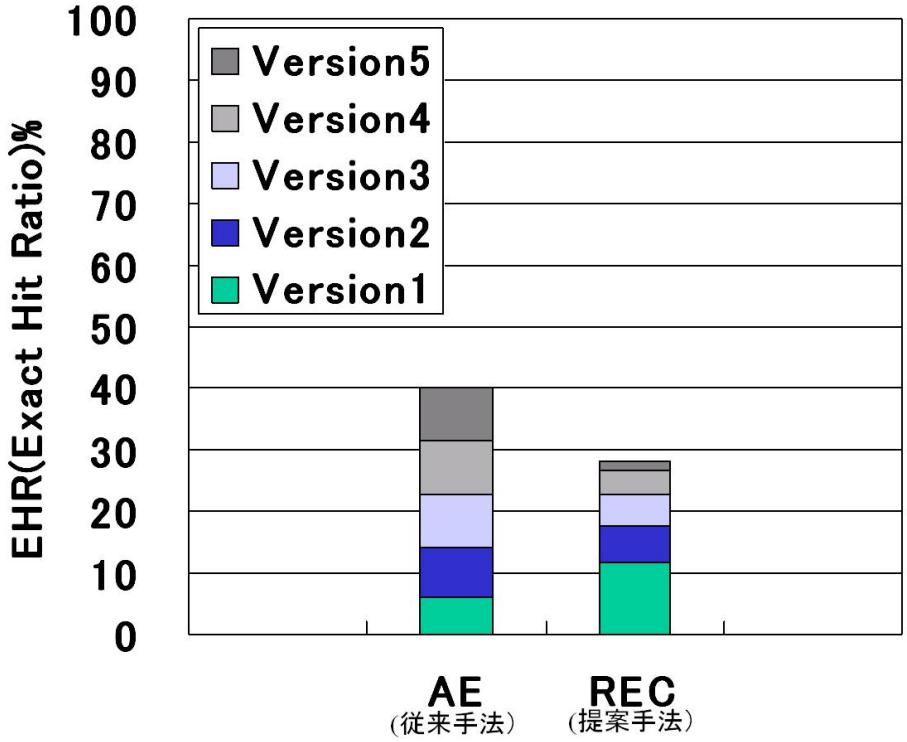

図 5.7: バージョン毎の EHR の状況

図 5.7 ではバージョン毎の EHR を示す。この図から見ると従来研究 AE アルゴリズムの EHR が前に説明したように本研究の REC アルゴリズムより高いことが分かった。AE アルゴリズムはバージョン 1 からバージョン 5 の EHR が 5.6%、8.1%、8.7%、8.7%、8.6%である。本研究の REC アルゴリズムではバージョン番号が大きくなるにつれて EHR が低くなる、バージョン 1 からバージョン 5 の EHR が 11.8%、6%、5.1%、3.8%、1.3%である。本研究の REC アルゴリズムには番号が小さいバージョンがよくキャッシュされることが分かった。

図 5.8 ではバージョン毎の UHR を示す。この図から見ると本研究の REC アルゴリズムの EHR が従来研究 AE アルゴリズムよりずいぶん高いことが分かった。それは本研究の REC アルゴリズムは番号が小さいバージョンがよくキャッシュされるためである。第 3 章で分析したように従来研究では番号が大きいバージョンをキャッシュするのでバージョ

ン番号が小さいバージョンがリクエストされるときにキャッシュミスになり、キャッシュヒット率が低くなる原因である。本研究の REC アルゴリズムは、バージョン 2 からバージョン 5 の UHR が 9.1%、11%、13.5%、16.2%である。AE アルゴリズムはバージョン 2 からバージョン 5 の UHR が 2.2%、4.5%、6.9%、7.7%である。

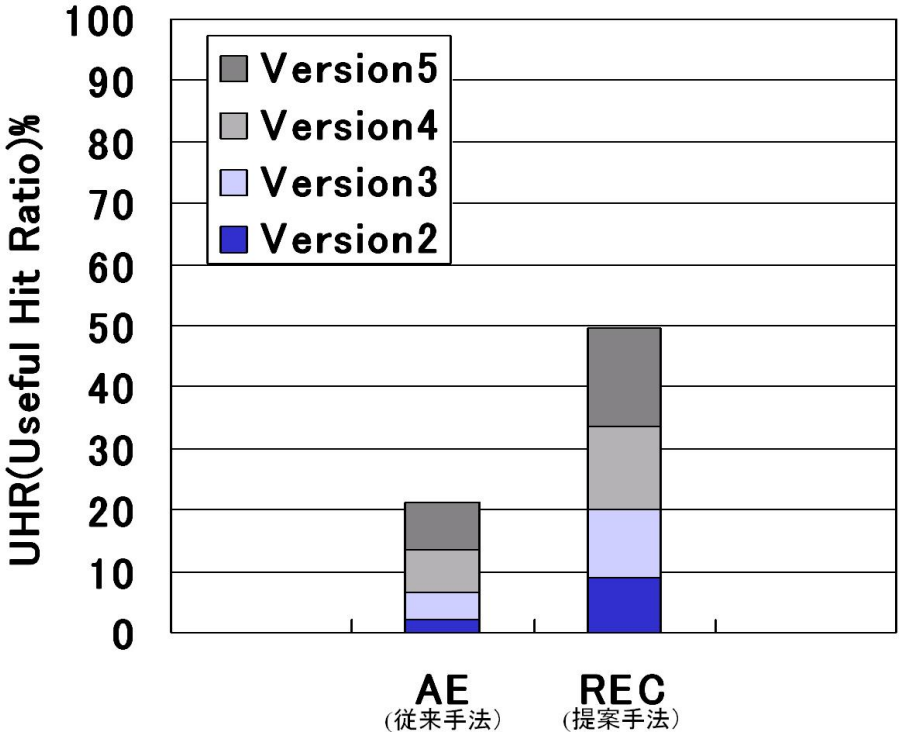

図 5.8: バージョン毎の UHR の状況

本研究は全体のキャッシュヒット率 HR が従来研究より高く、本研究の優位性を示した。また、本研究ではキャッシュされたバージョン間の影響関係を考慮していることによって、前分析したように番号が大きいバージョンのコストを計算する際、影響バージョンと生成バージョンの節約コストの差が小さく、番号が大きいバージョンのアクセス率がちょっと大きくなっても、節約コストの値が大きく上がらないので、画質調整いらないキャッシュヒット率 EHR に関しては従来研究より低い、しかし、番号が小さいバージョンがキャッシュされ、その分の UHR が高くなり、キャッシュミスが減少し、全体のキャッシュヒット率が高くなった。

5.7 サーバ台数の増加による実験

以上の実験では Web サーバが一台の状況であった。本節では前のデフォルト設定と同じアクセス状況のときに、Web サーバの台数が増えることにより、節約コスト率の変化具合を実験した。実験結果が図 5.9 に示す。

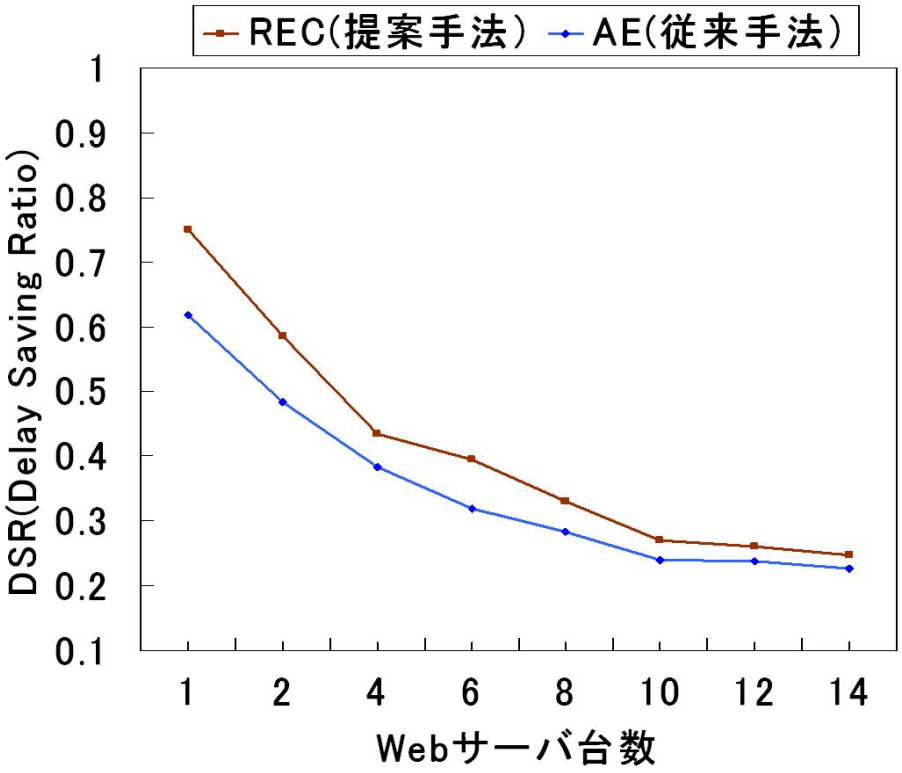

図 5.9: サーバの台数の増加に従って節約コスト率の変化

表 5.5: サーバの台数の増加による節約コスト率の実験結果

台数	提案手法	従来手法
1	75.9%	61.9%
2	58.5%	48.3%
4	43.5%	38.3%
6	39.5%	31.9%
8	33.1%	28.4%
10	27.0%	24.0%
12	26.1%	23.8%
14	24.7%	22.6%

前のデフォルト設定と同じアクセス状況のときに図 5.9 によるとサーバの台数が増えるにつれて節約コスト率が低くなることが分かった。本研究の REC アルゴリズムではサーバ 1 台のときから二台ずつ増やして 14 台までの節約コスト率が表 5.5 に示す。

サーバの台数が増えると、アクセス回数が同じな場合にはアクセスが各サーバ毎に分散され、アクセスされたオブジェクトが頻繁に変わり、キャッシュに対して、毎回に違うオブジェクトがアクセスされ、キャッシュミスが多くなり、全体の節約コスト率が低くなった。

5.8 人気度合い指数による実験

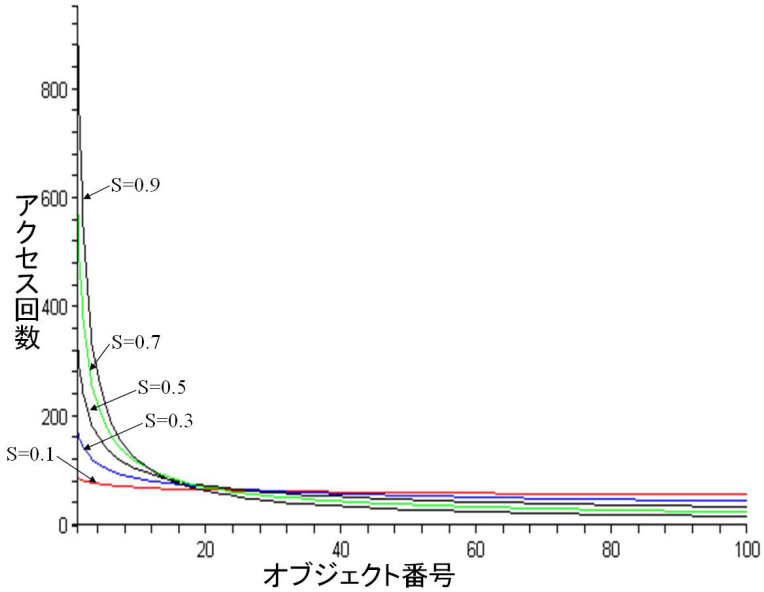

図 5.10: s の値毎の各オブジェクトのアクセス回数

本研究では各オブジェクトの人気度合がジップ分布に従って設定されたが、デフォルトとして分布の曲線状態を決めパラメータ s が文献 [8] の経験値である 0.7 に設定された。s が 1 の場合には本来のジップ法則になり、s の値が小さくなればなるほど、各オブジェクトの人気度合いの差がだんだん小さくなる。図 5.10 は総合アクセス回数 6000 に対して s の値が 0.1 から 0.9 までの各状況に各オブジェクト毎のアクセス率分布である。s が 0.9 のときではオブジェクト間の人気度合いの差が大きく、アクセス回数のさも大きい、s の値が小さくなるに従って人気度合いの差が小さくなり、アクセス回数の差も小さくなった。特に s が 0.1 の場合には分布が直線と近い曲線になった。それは各オブジェクトのアクセス率が殆ど同じになることを示している。

本研究では s の値が変わり、節約コストの変化をはかるために以下の実験をした。図 5.11 では s の値が 0.1 から 0.9 に変わるときの節約コスト率を示している。本研究の REC

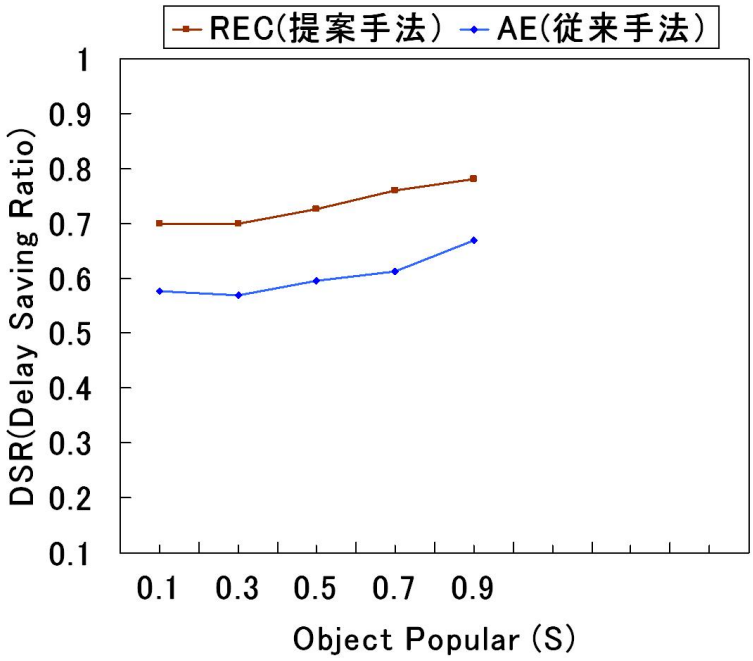

図 5.11: 人気度合よる節約コストの変化

表 5.6: 人気度合いの差による節約コスト率の実験結果

s	提案手法	従来手法
0.1	69.9%	57.7%
0.3	70.0%	56.8%
0.5	72.5%	59.5%
0.7	75.9%	61.3%
0.9	78.0%	67.0%

アルゴリズムではsの値が0.1から0.9に変化するときに節約コスト率が表5.6に示す。従来研究のAEアルゴリズムのsの値が0.1から0.3に変わるときに節約コスト率が小さくなった以外には、本研究と従来研究両方もsの値が大きくなるにつれて節約コスト率が大きくなる。

本項では人気度合いがジップ分布にしたときに、sの値が小さくなるたびに、人気度合いの差が小さくなり、毎回に違うオブジェクトがアクセスされる可能性が高くなり、節約コスト率が低くなることを確認した。

5.9 他の実験

本研究では五つのバージョン間の関係について、実験を行った。バージョン間の関係が図5.12に示す。このらの関係に基づき、実験の結果が表5.7に示す。

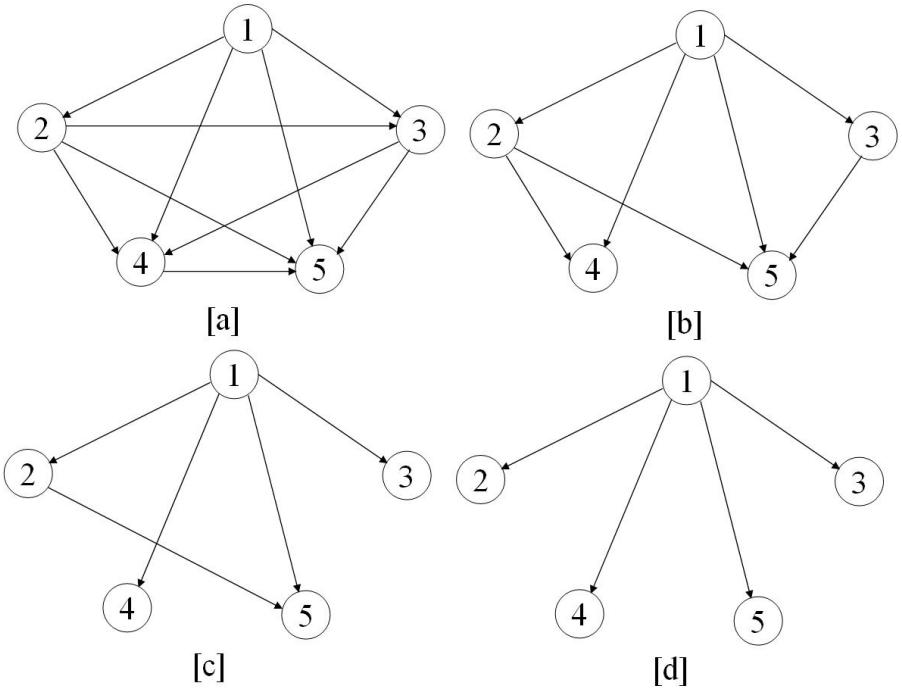

図 5.12: バージョン間の関係

この結果による、バージョン間の関係が異なるときに、節約コスト率に対して影響が与えられる。

また、本研究ではバージョンの個数が変わるときに、節約コストの変化についても実験した。バージョンの個数と関係は図5.13に示す。

実験が五個、七個、十個のバージョンに分けて行った。結果が表5.8に示す。この結果による、バージョン個数が異なるときに、節約コスト率に対して影響が与えられる。

表 5.7: バージョン間の関係による節約コスト率の実験結果

関係	提案手法	従来手法
a	75.9%	61.3%
b	72.8%	57.5%
c	71.9%	57.0%
d	71.0%	58.8%

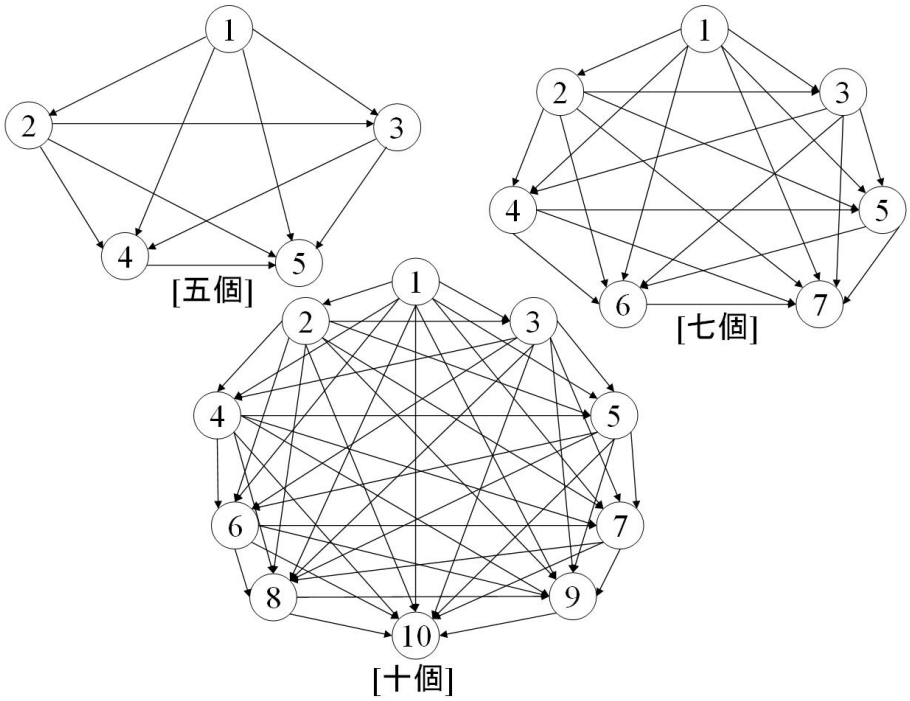

図 5.13: バージョンの個数と関係

表 5.8: バージョンの個数による節約コスト率の実験結果

個数	提案手法	従来手法
五個	75.9%	61.3%
七個	79.1%	65.9%
十個	74.5%	61.8%

5.10　まとめ

実験のシナリオを説明し、実験の結果を紹介した。また節約コスト率に影響するネットワークトポロジーやパラメータの設定等を考え、色々な状況を想定し、追加実験もした。それらの実験結果によると本研究が提案したアルゴリズムの有効性を示した。

第6章　まとめ

6.1　まとめ

本研究では画質調整機能を持つプロキシサーバにおいて画質調整された前後の画像をキャッシュ空間に置き換えする際にネットワーク転送コスト、画質調整コスト及びアクセス率などの要素を考え、画質調整された前後の画像の影響関係を考慮した。影響関係を分析するためにキャッシュされたバージョンが生成バージョンと影響バージョンに定義した。また、本研究では影響関係を構築するアルゴリズムを提案した。またそのアルゴリズムの結果によって生成バージョンの画質調整コストを算出する関数を提案した。提案したアルゴリズム及び関数を NS2 に実装し、シミュレーションした。シミュレーションの結果から見ると本研究ではキャッシュされたバージョン間の影響関係を考慮することによって、正確にキャッシュされたバージョンの節約コストの計算ができ、算出したコストに基づき、節約コストが高いバージョンがキャッシュされると従来研究の AE アルゴリズムにより、全体の節約コスト率が高くなり、全体のキャッシュヒット率も高くなった。

この結果により、本研究が Web サーバのアクセス回数を減少し、Web サーバの負荷を低減した。ネットワークのトラフィックの削減にも役に立った。クライアントへの迅速な応答を実現した。

謝辞

本研究を行うにあたり、多くの御助言、御指導を賜りました情報科学センター井口　寧助教授に深く感謝するとともに、ここに御礼申し上げます。

適切な御意見、御助言を頂きました本学の松澤照男教授、田中清史助教授に深く御礼申し上げます。

貴重な御意見、討論を頂いた井口研究室のみんなさんに様々なアドバイスを頂きました。

本研究に関する学会発表

1. 画質調整機能を持つプロキシサーバにおけるキャッシュリプレースメントに関する研究，李　奇，井口　寧，2006 年度　電気関連学会　北陸支部大会，金沢工業大学.

参考文献

[1] 総務省. **ユビキタス エコノミー（平成 *18* 年版情報通信白書）**. 総務省, 2006.

[2] 総務省. **「*u-Japan*」の胎動 *: 2010* 年の「*u-Japan*」実現に向けて（平成 *17* 年版情報通信白書）**. 総務省, 2005.

[3] J.R.Smith R.Mohan and C.-S.Li. *Adapting Multimedia Internet Content for Universal Access.* IEEE Trans.Multimedia, vol.1, no.1,pp.104-114, 1999.

[4] R.Han and P.Bhagwat. *Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing.* IEEE Personal Comm.Magazine, pp.9-17,Dec, 1998.

[5] 中野　賢、春もと　要 、下條 真司、西尾　章治郎. **ページ配送時間を考慮した画質調整機能を持つ *WWW* サーバ**. 電子情報通信学会論文誌, D-１ Vol.J83-D-１ No. １, 194 頁〜202 頁, 2000.

[6] H Shen Keqiu Li and Keishi Tajima. *An Effective Cache Replacement in Transcoding-Enabled Proxies.* Journal of Supercomputing,Vol.35,No.2, February, 2006.

[7] Chi-Hung Chi Palit H.N. and Lin Liu. *Proxy-Based Pervasive Multimedia Content Delivery.* IEEE Computer Society, Vol.1 P.255 - 264, 2006.

[8] Y.-W.Huang V.Cardellini, P.-S.Yu. *Collaborative Proxy System for Distributed Web Content Transcoding.* Proc. ACM Int 'l Conf, Information and Knowledge Management, pp. 520-527, 2000.

[9] C.chang and M.Chen. *On Exploring Aggregate Effect for Efficent Cache Replacement in Transcoding Proxies.* IEEE Trans.on Parallel and Distributed Systems, vol.14,No.6,PP.611-624, 2003.

[10] A.Bestavros and C.Cunha. *APrefetching Protcol Using Client Speculation for the WWW.* in Rech.Rep.TR-95-011, BostonUniv., 1995.

[11] T. Loon and V. Bharghavan. *Alleviting the Latency and Bandwidth Problems in WWW Browsing.* Proc. USENIX Symp. Internet Tech. and Sys., P.219 - 230, 1997.

[12] T. Kroeger D.Long and J. Mogul. *Exploring the Bounds of Web Latency Reduction from Caching and Prefeching.* Proc. USENIX Symp.Internet Tech. and Sys., P.13 - 22, 1997.

[13] Ming-Syan Chen Wei-Guang Teng, Cheng-Yue Chang. *Integrating Web Caching and Web Prefetching in Client-Side Proxies.* IEEE Trans. Parallel Distrib. Syst., 444-455, 2005.

[14] K. Chinen and S. Yamaguchi. *An Interactive Prefetching Proxy Server for Improvement of WWW Latency.* Proc. INET'97Conf., 1997.

[15] P. Scheuermann J. Shim and R Vingralek. *Proxy Cache Algorithms: Design, Implementation, and Performance.* IEEE Transactions on Knowledge and Data Engineering, v.11 n.4, p.549-562, July, 1999.

[16] D.-L.Lee J.xu, Q.Hu and W.-C.Lee. *SAIU:An Efficient Cache Replacement Policy for Wireless On-Demand Broadcasts,.* Proc.ACM Int'l Conf.Information and Knowledge Management,, pp.46-53,, 2000.

[17] L. Breslau P. Cao L. Fan G. Phillips and S. Shenker. *Web Caching and Zipf-Like Distributions: Evidence and Implications.* Proc. IEEE INFOCOM, Mar, 1999.

[18] Joel L. Wolf Charu Aggarwal and Philip S. Yu. *Caching on the World Wide Web.* IEEE Transactions on Knowledge and Data Engineering, v.11 n.1, p.94-107, January, 1999.

[19] Venkata N. Padmanabhan and Lili Qiu. *The content and access dynamics of a busy Web site: findings and implications.* Proc.ACM SIGCOMM, p.111-123, 2000.

[20] Steve Mann and Scott Sbihli. *WAP*実践と導入リファレンス. 株式会社アスキー, 東京渋谷区代々木 4 丁目 33 番 10 号, 2001.