静的プロファイルを用いたファイル・プリフェッチ手法の提案

安江俊明[†] 小松秀昭[†] 中谷登志男[†]

近年の大規模アプリケーションの起動時間の増加は，アプリケーションの開発や運用において大きな問題となっている．特に起動中のファイル読み込みにより発生する I/O 待ち時間は起動時間を増加させる要因の 1 つである．本論文では，この I/O 待ち時間を短縮する方法として，対象アプリケーションにおける I/O 処理の静的プロファイルを用いてプリフェッチ処理を実行する手法（Scenario-Based Prefetching）を提案する．従来よりプリフェッチ処理を用いたファイル読み込み処理の最適化手法が多数研究されてきている．しかしその多くは汎用的な手法である一方で，大規模アプリケーションの起動時のように，ファイル読み込み処理が多数のコンポーネントに分散し，多数のファイルをランダムアクセス的に読み込むような状況下ではあまり効果を得ることができなかった．これに対して，我々は大規模アプリケーションの起動時の I/O 処理がほとんど同じ順序で行われている点に着目し，(1) 事前に収集された静的プロファイルを用いてアプリケーションの挙動に合わせてプリフェッチ処理をスケジューリングしたシナリオ生成しておき，(2) このシナリオを用いて実行時にプリフェッチ処理を実施することで，アプリケーションを変更することなしに起動時間を大きく削減することを可能とした．本手法をいくつかのアプリケーションで評価した結果，起動時間を約 30%削減できることを示した．

Scenario-Based Prefetching

Toshiaki Yasue,[†] Hideaki Komatsu[†] and Toshio Nakatani[†]

The increase of the startup time of the recent large applications causes the degradation of the system utilization and the turn around time of the development. One reason is that much I/O wait occurs during the startup, because those applications need to read much data from many files to setup the applications. In this paper, we propose a new prefetching technique called Scenario-Based Prefetching, which performs prefetching by using the information generated from static I/O profiles. Although many conventional techniques have been proposed to reduce the I/O wait time by using the prefetching, most techniques did not reduce the time enough for the start up. Our approach utilizes the characteristics that the sequence of the I/O operations is almost same for every startup. In our technique, first we collect I/O traces for the application in a training execution and generate a scenario, a sequence of the prefetching, by analyzing the traces and scheduling the prefetches for the application, and then we perform prefetching on a private thread by using the scenario. With this technique, we can perform prefetching for the target application without any modifications. The experimental results shows that our technique can be reduce the start up of the application by about 30% in some applications.

1. はじめに

近年の Web アプリケーションやアプリケーションサーバなどに代表される大規模アプリケーションは，Java 言語の普及などもあいまって広く構築されるようになってきた．しかしアプリケーションの規模が増大し複雑化するに従って，その起動時間の増加が問題となってきている．起動時間の増加は，システムの運用時間の低下だけでなくプロビジョニングなどのオン・デマンドな運用に対しても障害となる．さらに，アプリケーションの開発時にはより高頻度に起動と終了が繰り返されるために，開発効率の低下を引き起こしている．特にアプリケーションサーバは，目的とするアプリケーションを実行するためのフレームワークであり，その起動時間が多くのアプリケーションに対して影響を与えてしまう点で問題が大きい．したがって，このようなアプリケーションの起動時間を高速化することは非常に重要な課題である．

アプリケーションの起動時間を増加させる要因は様々あるが，その 1 つの要因としてファイル読み込みにともなう I/O 待ち時間があげられる．特にアプリ

† 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Tokyo Research Laboratory, IBM Japan Ltd.

ケーションの起動時は実行ファイルや設定ファイルなど多くのファイル読み込みが発生する一方で，処理が逐次的に進行するために，I/O 処理が他の計算処理とオーバラップされない場合が多く，I/O 処理待ち時間がそのまま起動時間に現れてしまう場合が多い．

この I/O 待ち時間は，HDD 上のファイルからデータを読み込む速度と CPU の処理速度の差により生じる．HDD からのデータ読み込み処理は，大きく分けると HDD のヘッドを目的のデータがある領域に移動させる処理（シーク処理）と，HDD から目的のデータを読み取るための処理（データ転送処理）からなる．シーク処理時間はヘッドが移動する回数や距離に比例して増加する．他方，データ転送処理時間は HDD から読み出されるデータの量に比例して増加する．したがって，同じ量のデータを読み出す限り必ず同じ時間を必要とする．

I/O 待ち時間を減らすために解決すべき課題は大きく 2 つある．第 1 点としては，アプリケーションの起動時は多数のデータが読み込まれるので，読み込み処理を起動処理とオーバラップさせながら I/O 資源の利用効率を上げることである．第 2 点としては，特に Java 言語で構築されたシステムでは，JAR ファイル中のクラスファイル読み込みにともない多数のランダムアクセス読み込みが発生する．ランダムアクセス読み込みは，読み込みのたびにシーク処理を実施するため，このオーバヘッドが I/O 待ち時間の増加の要因の 1 つとなっている．このため，何らかの方法でシーク処理時間を削減する方法が必要である．

I/O 待ち時間を削減する従来手法は大きく 3 つに分類できる．1 つめは非同期 I/O 処理を用いてアプリケーション自体を書き直す手法，2 つめは何らかの方法で読み込み処理に先行してプリフェッチ処理を発行してデータをファイルキャッシュに読み込ませる方法，3 つめは HDD 上の読み出し時にヘッドの移動する距離が最小になるようにしてシーク処理時間を削減する方法である．このうち非同期 I/O を用いる方法は，プログラマがプログラムの知識を用いて記述するために原理的には最も効果の高い方法であるが，現実的には大規模なシステムでのプログラムの書き換えは困難である．

プリフェッチ処理を用いる従来手法は，さらに，コンパイラなどで解析的にプログラムにプリフェッチ処理を挿入する方法[1),2)]，元のプログラムからプリフェッチ処理用のプログラムを生成して投機的に実行させる方法[3),4)]，動的プロファイル情報を用いて実行時にファイルのアクセスパターンを検出してプリフェッチ処理を実行する方法[5),6)]，アプリケーションにディレクティブなどを挿入してプリフェッチ処理を行う手法[7)~9)]，静的プロファイルを用いてプリフェッチ処理を行う方法[10)~12)] に分類できる．このうち静的プロファイルを用いる方法以外の方法は，汎用的な手法である反面，プリフェッチ処理と実際の read 処理の時間的間隔をあまり大きくとれないことが多く，連続して多くのファイル読み込みが発生するような場合には効果が少ない．他方，静的プロファイルを用いる手法は，ファイル読み込みに再現性のあるプログラムにしか効果がないが，アプリケーションが読み込むファイルの情報を使って十分に早い時期からプリフェッチ処理を実施できるため，再現性のあるプログラムに対しては大きな効果を得ることができる．しかし，従来手法では，プロファイルされた状態と実際の実行が一致しているかどうかの検出に重点が置かれており，プロファイルを用いていかに高速化するかという観点では議論されていなかった．

シーク処理時間を削減する方法としては，OS の I/O 要求キュー中にある要求の実行順序をヘッドの移動距離が最小になるように並べ替える手法[13),14)] や，静的プロファイルを用いて HDD 上のファイルの配置を並べ替える手法[15)~17)] などが提案されている．前者の手法は汎用的ではあるが，同時に I/O 要求キューに存在する要求の間でのみシーク処理時間の最適化が行われるため，逐次的に読み込みを行うプログラムにおけるシーク処理時間を短縮することはできない．また後者の方法は OS などシステムレベルでの実装が必要となるため，既存のシステムで容易に利用できる方法ではない．

本論文では，静的プロファイルから作成したシナリオを用いてプリフェッチ処理を実行する手法を提案する．提案手法は，まず事前に収集した I/O 処理の静的プロファイル情報を解析してプリフェッチ処理を実施するためのシナリオを作成する．シナリオ作成では，シーク処理オーバヘッドの軽減を実現するために，HDD 上の連続領域に対するプリフェッチ処理を連続して行うようにプリフェッチ処理の順序の並べ替えを行う．実行時には，このシナリオを用いてアプリケーションとは別の専用スレッド上でシナリオに従ってプリフェッチ処理を実施する．従来の静的プロファイルを用いる手法ではプロファイルされた順にプリフェッチ処理が実施されていたのに対して，本手法ではこの並べ替えにより起動時に発生する I/O 待ち時間自体を短縮することができる．また本手法は，OS のファイルキャッシュを用いてプリフェッチされたデータを

保持させるので，実装に特殊な仕組みを必要とせず，既存のほとんどのシステム上で実装可能である利点もある．

RedHat9 Linux 上で IBM WebSphere Application Server 5.1.1 の起動時間による提案手法の評価を行ったところ，本手法を用いてプリフェッチ処理を実施することで起動時間を 20〜33%短縮できることが分かった．またプリフェッチ処理を実施する順序を並べ替えることで，約 10%の速度向上が得られることが分かった．

以下では，まず 2 章において，アプリケーション起動時の I/O 待ち時間の問題について述べる．続いて 3 章において我々が提案する Scenario-Based Prefetching 手法について説明し，4 章において提案手法を用いた評価結果を示す．最後に 5 章で関連研究について説明する．

2. アプリケーション起動時の I/O 待ち時間の問題

本章では，まず I/O 待ち時間について説明した後，アプリケーション起動時におけるファイル読み込みにともなう I/O 待ち時間の影響について示す．その後，従来手法を示しながら，アプリケーション起動時間の短縮のために解決すべき課題について述べる．

2.1 I/O 待ち時間

現在の計算機では，CPU が HDD からデータを読み出す速度はメモリなどからデータを読み出す速度に比べて遙かに遅いので，この速度差から I/O 待ち時間が生じる．本論文では，I/O 待ち時間を CPU が HDD 上のデータを読みにいく際に生じる CPU のアイドル時間と定義する．HDD からデータを読み出す時間は，シーク処理時間とデータ転送処理時間に分けられる．シーク処理時間は HDD のヘッドを読み出すデータが置かれている位置に移動する処理にかかる時間であり，データ転送処理時間は HDD から目的のデータを実際に読み出す際にかかる時間である．シーク処理時間はヘッドが移動するたびに発生するオーバヘッドであるため，プログラムのデータのアクセスの仕方によって変化する．たとえば逐次アクセスで HDD 上の連続領域を読み出す場合はシーク処理コストが最小となり，ランダムアクセスで毎回異なる位置からデータを読むと増大する．一方，データ転送処理時間は読み出すデータ量にのみ比例してかかる時間でり，同じデータを読み出す場合は読み出す順番によらず同じ時間を必要とする．

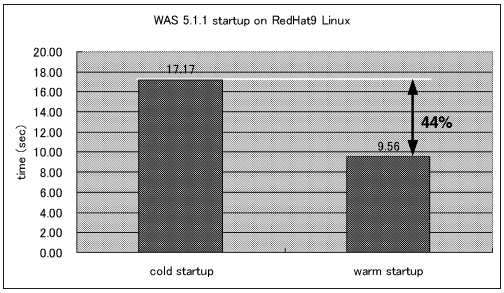

図 1 RedHat9 Linux での WAS 5.1.1 の起動時間の評価結果
Fig. 1 WAS 5.1.1 startup time on RedHat9 Linux.

2.2 アプリケーション起動時における I/O 待ち時間

実際の I/O 待ち時間がアプリケーションの起動にあたえる影響として，Redhat9 Linux において IBM WebSphere Application Server（WAS）5.1.1 の起動処理時間を測定した結果を**図 1** に示す．図 1 では，WAS の起動時間を，計算機の起動直後に測定した場合（cold startup）と，WAS を 1 回起動し終了することで起動時に読み込まれるすべてのファイルがファイルキャッシュ上に保持されている状態で測定した場合（warm startup）の結果を示している．この 2 つの実行の差は，HDD からのデータ読み込み処理の有無であることから，WAS の起動においては，HDD からのファイル読み込みによる I/O 待ち時間に起因するオーバヘッドが起動時間の 44%を占めていることが分かる．つまりアプリケーションの起動時には多くのファイル読み込みが発生し，その I/O 待ち時間が起動時間に大きく影響を与えているといえる．また，システム自体を再起動しない場合でも，アプリケーション自体の実行の結果としてファイルキャッシュから起動時に読み込まれるファイルが追い出されてしまう場合は，その起動処理に対してつねに cold startup の場合と同様の I/O 待ち時間を被ってしまう．

2.3 解決すべき課題

大規模アプリケーションの起動時はファイルの読み込み量が多い．前述の結果では，WAS の起動時間の約 44%が I/O 待ち時間であり，この値はアプリケーションによっては 70%近くに達する場合もある．このことから，I/O 待ち時間を最小にするためには，ファイルからのデータ読み込みと計算処理とをオーバラップさせることで，起動処理中の I/O 資源の使用率を上げることが重要な課題となる．

従来手法においては，静的プロファイル手法を除くとプリフェッチを起動直後から連続的に実施することは原理上不可能であった．静的プロファイルを用いる

手法は，汎用性は低いがアプリケーションの起動処理などの再現性の高いアプリケーションに対しては有効な方法である．しかし，従来の静的プロファイルを用いる手法では，その論点は主にプリフェッチ処理に使用するプロファイルが実際のアプリケーションの実行と一致しているかどうかの検出に重点が置かれており，収集されたプロファイルを用いていかにアプリケーションを高速化するかという観点では論じられていなかった．たとえば，Lei らの手法[10] や Griffoen らの手法[11] では，プリフェッチ処理はファイル単位であるため，ファイルの一部しかアクセスしないような場合を考慮していない．また Grimsrud らの方法[12] は，HDD 上の各セクタごとに次にアクセスされやすいセクタの情報を保持しておくことで効率的なプリフェッチ処理を実現しようとしている．しかし HDD の読み出し順序はアプリケーションのアクセス順序と同じであるため，ランダムアクセス読み込みにより発生するシーク処理のオーバヘッドの影響をそのまま被ってしまう．特に Java 言語で構築されているシステムでは，JAR ファイルからのクラスの読み出しがランダムアクセス読み込みとなり，多数のクラスファイルの読み込みによって生じるシーク処理のオーバヘッドの増大が実行速度を低下させる原因の 1 つとなっている．したがって，このシーク処理のオーバヘッドを削減することもアプリケーションの起動時間短縮のためには重要な課題となる．

3. Scenario Based Prefetching

前章で述べた課題を解決する方法として，我々は Scenario-Based Prefetching 手法を提案する．本手法では，静的プロファイルを解析して生成したシナリオを用いてプリフェッチ処理を実施する．アプリケーションの実行とファイルからのデータ読み込みを最大限にオーバラップさせることで I/O 待ち時間を減少させるとともに，プリフェッチ処理の順序を最適化することでシーク処理オーバヘッドの削減も実現する．本手法の処理は，事前実行による I/O プロファイルの収集，シナリオの生成，プリフェッチの実施，という 3 つのフェーズから構成される．以下で，これらの各フェーズについて順に説明する．

3.1 プロファイルの収集

まず，対象アプリケーションの I/O 処理に関するプロファイルを収集する．収集されるプロファイルの情報は，対象プログラムにより実行される I/O 処理（open，close，read，write，lseek など）の引数と返り値，thread ID，時刻，実行に要する時間などである．今回の実験では，Linux で用いられている weak alias による関数解決の仕組みを利用して，LD_PRELOAD 環境変数でフック関数のライブラリを指定し，I/O 処理用の各システムコールをフックすることでプロファイルを収集した．たとえば，read 関数の場合は，**図 2** に示すように，システム関数である read と同じ名前の関数を定義した関数を用意する．フック関数中では，必要なプロファイルを収集するとともに，read 関数の実体の名前である __libc_read 関数を呼ぶ．なお，フック関数中で使用されている関数 SBP_GetPerformanceCounter() は，時刻に相当する値を返す関数，関数 SBP_dumpProfile() は，渡された引数をプロファイルとしてログファイルに書き出す関数である．

```
 1. ssize_t read(int  fd, void  *buff, size_t  n) {
 2.     ssize_t result;
 3.     long long t_start, t_end;
 4.     int tid;
 5.     totalReadSize += n;
 6.     t_start = SBP_GetPerformanceCounter();
 7.     result = __libc_read (fd, buff, n);
 8.     t_end = SBP_GetPerformanceCounter();
 9.     SBP_dumpProfile (fd, buff, n, result,
10.         t_start, (t_end - t_start));
11.     return result;
12. }
```

図 2 read 処理に対してプロファイル情報を収集するための処理
Fig. 2 Hook mechanism for collecting read profiles.

このようにして収集した I/O プロファイルの例を **図 3** に示す．図 3 (a) はプロファイルを収集する対象のプログラムであり，図 3 (b) はこのプログラムを実行した場合のタイム・チャートを示している．これに対して収集された I/O プロファイルが図 3 (c) に示されている．

3.2 シナリオの生成

次に，収集したプロファイルを解析し，実行環境での I/O 資源を考慮しながらプリフェッチ処理の実行時期と実行順序を決定し，シナリオを生成する．以下に，シナリオの生成処理の処理手順を示す．

3.2.1 I/O 処理リストの作成

I/O プロファイルを順に走査し，ファイルデスクリプタの値を用いてファイルごとに時系列順の I/O 処理リストを生成する．同じファイルが複数回オープンされている場合は，1 つの I/O 処理リストにまとめる．

3.2.2 プリフェッチ処理リストの作成

ファイルごとに read 処理のうちで実際に I/O 要求を発行する read 処理を検出する．既存の OS では，read 処理で読まれるデータがファイルキャッシュ上に

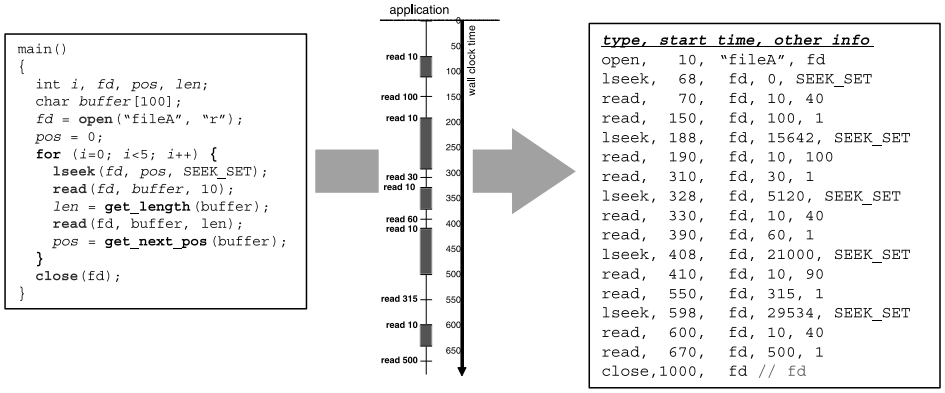

図 3 I/O プロファイルの例
Fig. 3 An example of I/O profiles.

```
1. Tr_total = 0;
2. T_total = 0;
3. Tclr = 0;
4. for (P = プリフェッチ処理リスト中の先頭要素から順に繰り返す) {
5.     Tc = P に対応する read 処理のプロファイル上の開始時刻 - Tclr;
6.     Tclr = P に対応する read 処理のプロファイル上の完了時刻;
7.     Tr_total += P に対応する read 処理の処理時間;
8.     if (T_total + Tc > Tr_total) {
9.         T_total += Tc;
10.    } else {
11.        T_total = Tr_total;
12.    }
13.    P に対応する read 処理の開始時刻= T_total;
14. }
```

図 4 I/O 処理をオーバラップさせた場合の各 Read 処理の開始処理を求めるアルゴリズム
Fig. 4 Algorithm to calculate the estimated start time for each read.

存在しない場合，OS により規定された I/O 要求のサイズに正規化されてた I/O 要求を生成して HDD からデータの読み込みが実行される．たとえば実験に使用した Linux では 4 KB 単位で HDD からデータが読み出される．一度 HDD から読み込まれたデータはファイルキャッシュ上に保持されるので，同じキャッシュブロック上に存在するデータを読み出す read 処理に対して I/O 要求は生成されない．ファイルごとに時系列順の read 処理リストを走査し，各キャッシュブロックのデータを最初に読み出す read 処理を検出し，その read 処理に対応させてプリフェッチ処理のリストを生成する．

3.2.3 各 read 処理の完了時間の計算

HDD からのデータの読み込みが計算処理と完全に並列に実行できた場合の各 read 処理の完了時刻を計算する．ここでは，アプリケーションの処理が逐次処理されたと仮定して，プリフェッチ処理を計算処理と並列に実施した場合における各 read 処理の完了時刻 Ti を計算する（**図 4**）．

3.2.4 各プリフェッチ処理の最終開始時間の計算

すべての read 処理の完了時刻を遅らせない範囲で，各プリフェッチ処理の最も遅い開始時刻（最終開始時刻）を計算する（**図 5**）．

3.2.5 プリフェッチ処理の並べ替え

プリフェッチ処理の並べ替えは，各プリフェッチ処理の最終開始時刻を遅らせないことを条件として隣接する領域へのプリフェッチ処理を連続に行えるかを調べながら実施する．ここでは，プリフェッチ処理リストを先頭から順に走査しながら並べ替えが可能かどうかを調べていく（**図 6**）．アルゴリズム中の関数 isLowerSelected()，isUpperSelected() は，現在処理中のプリフェッチ処理である P がアクセスする領域に隣接する

```
1. P0 = (プリフェッチ処理リストの最後のプリフェッチ処理)
2. (P0 の最終開始時刻) = (P0 に対応する read 処理の開始時刻)
3.    - (P0 に対応する read 処理の処理時間)
4. T_deadline = (P0 の最終開始時刻)
5. for (P = P0 の直前の要素から順に，リストを逆順にたどりながら繰り返す) {
6.    if (T_deadline < (P に対応する read 処理の開始時刻)) {
7.       (P の最終開始時刻) = T_deadline - (P に対応する read 処理の処理時間)
8.    } else {
9.       (P の最終開始時刻) = (P に対応する read 処理の開始時刻)
10.         - (P に対応する read 処理の処理時間)
11.   }
12.   T_deadline = (P の最終開始時刻)
13. }
```

図 5 各プリフェッチ処理の最終開始時刻を求めるアルゴリズム
Fig. 5 Algorithm to calculate the estimated deadline time for each prefetch.

```
1. P1 = (プリフェッチ処理リストの先頭要素)
2. (P1 の開始時刻) = 0;
3. P2 = (プリフェッチ処理リストで P1 の次の要素)
4. while (P2 が NULL でない間繰り返す) {
5.    F1 = (P2 の最終開始時刻) - (P1 の開始時刻);
6.    F2 = (P に対応する read 処理の開始時刻) - (P1 の開始時刻);
7.    free_time = min (F1, F2)
8.    if (free_time > (P の処理にかかる時間)) {
9.       for (;;) {
10.         lower = (P の領域の直前の領域をプリフェッチする処理)
11.         upper = (P の領域の直後の領域をプリフェッチする処理)
12.         if (isLowerSelected (P, lower, upper)) {
13.            concatinateLowerPrefetch (P);
14.         } else if (isUpperSelected (P, lower, upper)) {
15.            concatinateUpperPrefetch (P);
16.         } else {
17.            break; // end of for
18.         }
19.      }
20.   }
21.   (P2 の開始時刻) = (P1 の開始時刻) + (P1 の処理にかかる時間)
22.   P1 = P2;
23.   P2 = (プリフェッチ処理リストで P1 の次の要素)
24. }
```

図 6 各プリフェッチ処理の最終開始時刻を遅らさない範囲で隣接するプリフェッチ処理を融合していく処理のアルゴリズム
Fig. 6 Algorithm to merge prefetches.

領域のプリフェッチ処理が存在する場合に，直前の領域と直後の領域のどちらを融合するほうが効果的かを計算する評価関数である．また，関数 concatinateLowerPrefetch()，concatinateUpperPrefetch() は，隣接領域のプリフェッチ処理を融合する関数である．この処理中にプリフェッチ処理される領域とプリフェッチ処理にかかる時間が更新される．

3.2.6 同期点の決定

本手法では，最終的にシナリオを用いてアプリケーションと独立のスレッド上でプリフェッチ処理を実施する．プリフェッチ処理がアプリケーションの実行と独立に実行される場合，両者の間に何らかの同期処理が必要となる．たとえば，プリフェッチ処理が先行しすぎてしまうと，必要以上に多くのファイルをプリフェッチしてしまい，まだ読み込みが終わっていないデータをファイルキャッシュから追い出してしまう状況が起こる．これはプリフェッチ処理の効果をなくすだけでなく同じデータを複数回読み込むことになり起動時間の増加を引き起こしてしまう．この問題を解決するために，本手法では，プリフェッチ処理リストを走査しながら，ファイルキャッシュが保持しているアクティブなデータのサイズを計算し，この量が閾値を超えた

```
1. Tr_total = Tp_current = Tc_offset = Tc_base = profidx
2.    = n_states = total_cache_size = 0
3. for (P = プリフェッチ処理リストの各要素を順にたどりながら繰り返す) {
4.    current_cache_size = 0;
5.    start_pref = P;
6.    while (P が NULL でない間繰り返す) {
7.       Tp_current += (P の処理にかかる時間)
8.       total_cache_size += (P によりキャッシュに追加されるデータサイズ)
9.       total_cache_size -= (P の処理が完了する時刻までのアプリケーションの
10.         実行で使用済みとなるキャッシュサイズ)
11.      T_target = (次のプリフェッチ処理要素の最終開始時刻) - (同期に要する時間)
12.      T_nextRead = (P の処理が完了した時刻以降で最初に実行される
13.         read 処理の時刻)
14.      if (total_cache_size > (保持するキャッシュサイズの閾値) &&
15.         T_target > Tp_current && T_target > T_nextRead) {
16.            break;
17.      }
18.      P = (プリフェッチ処理リストにおける P の次の要素)
19.   }
20.   targetRead = (T_target の直前に実行される Read 処理)
21.   makeNode(start_pref, P, targetRead)
22. }
```

図 7 アプリケーションとの同期点を検出するアルゴリズム
Fig. 7 Algorithm to detect synchronization points.

場合に，各プリフェッチ処理の最終開始時刻を遅らせない範囲でプリフェッチスレッドを停止させる期間を検出する（**図 7**）．アルゴリズム中の makeNode() 関数は，指定された範囲のプリフェッチ処理を実行した後，targetRead で指される read 処理で同期するための情報を生成する関数である．

3.2.7 シナリオの生成

以上で生成された情報からプリフェッチ処理シナリオを生成する．生成されるシナリオの表現形式は，実行時のプログラムとの同期方法により決定される．

3.3 プリフェッチの実施

最後に，生成されたシナリオを用いて，対象プログラムの実行に同期してプリフェッチ処理を実行する．プリフェッチ処理の実行は，アプリケーションと独立のプリフェッチ処理スレッド上で実施される．アプリケーションの実行進度を得るために，read 処理をフックする関数を用意し，この関数中でファイルから読み出されるデータの合計値をもとにプリフェッチ処理スレッドに対して同期のためのシグナルを送る．

プリフェッチ処理の実施例を**図 8** に示す．図 8 (a) はアプリケーションの実行に従って，read 処理でデータサイズ a，b，c ···，g のデータを順に読み込む様子を示している．図 8 (b) はプリフェッチ処理のシナリオで，いくつかのノードを直線状に並べたグラフとして表現されている．各ノードは一連のプリフェッチ処理を保持し，各エッジはその入力先のノードの開始

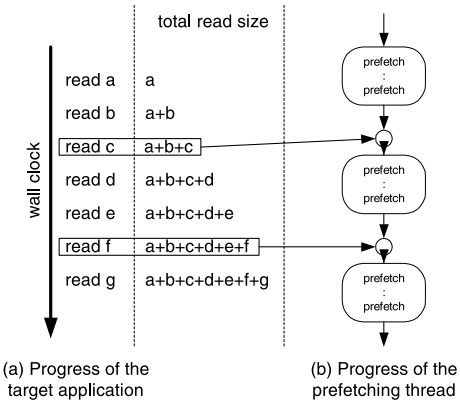

図 8 Scenario-based prefetching の実行モデル
Fig. 8 Execution model of Scenario-based prefetching.

条件としての読み込みデータ総量の値を保持する．あるノードのプリフェッチが完了すると，プリフェッチ処理スレッドはスリープ状態となる．アプリケーションが進行し読み込み総量が条件の値以上となった時点でフック関数からシグナルが送られ，プリフェッチ処理スレッドが次のノードの処理を開始する．以降は同期処理を繰り返しながら，終端ノードに到達した時点でプリフェッチ処理が終了する．

4. 評　　価

4.1 評価環境

評価は Intel Xeon 2.8 GHz × 2，2 GB memory の

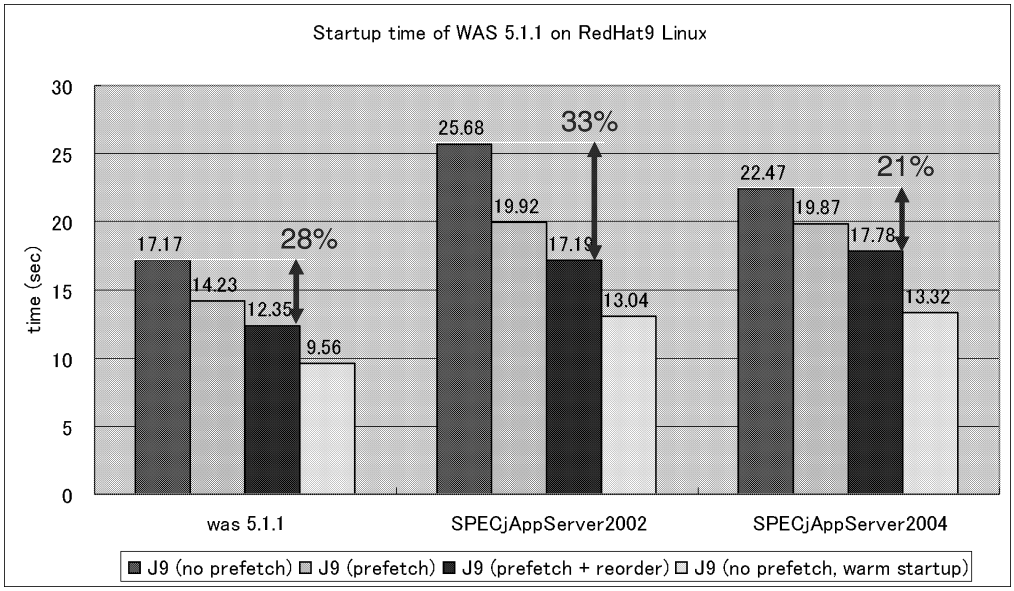

図 9 評価結果
Fig. 9 Evaluation results.

IntelliStation MPro 上で実施した．使用した HDD は，HITACHI Deskstar DK250 120 GB ATA100 モデルである．使用した Java VM は，IBM Java SDK 1.4.2 J9 prototype build である．評価は，WAS 単体の場合の起動時間，WAS に SPECjAppServer2002 ベンチマークをインストールした状態での WAS の起動時間，WAS に SPECjAppServer2004 ベンチマークをインストールした状態での起動時間を測定した．測定項目は，cold startup でプリフェッチ処理を実施しない場合の起動時間（cold startup），cold startup でプロファイルの順にプリフェッチ処理を実施した場合の起動時間（cold+prefetch），cold startup で Scenario-based prefetching 手法を用いてプリフェッチ処理を並べ替えた場合の起動時間（SBP），warm startup でプリフェッチ処理を行わない場合の起動時間（warm startup）の 4 点である．

4.2 評価結果

図 9 に，本手法の評価として，WebSphere Application Server 5.1.1 の起動時間を評価した結果を示す．評価結果より，Scenario-based prefetching 手法を用いた場合，WAS 単体で 28%，SPECjAppServer2002 で 33%，SPECjAppServer2004 で 21%の速度向上があったことが分かる．また，いずれの場合でも，単純にプロファイルの順にプリフェッチ処理を実施した場合に比較して，プリフェッチ処理の並べ替えにより約 10%の速度向上が得られていることが分かる．一般に，本手法による効果は，対象とするアプリケーションにおいてファイル読み込み処理と他の計算処理がどの程度並列に実行できるかに依存する．今回の実験から，アプリケーションや Java VM にまったく変更を加えない状態で平均 30%の速度向上を得ることができる点は，その実装のポータビリティとともに，本手法の有効性を示しているといえる．

5. 関連研究

従来手法は，大きく 7 つの方法が提案されている．以下にそれらについて説明する．

5.1 プログラムの実行時のファイルアクセスパターンを用いてプリフェッチ処理を行う方法

この方法は，実行時にファイルのアクセスパターンを検出し，そのアクセスパターンに従って次にアクセスされる領域を予測してプリフェッチ処理を行う方法である．従来手法[5),6)] では，特に UNIX におけるファイルのアクセスが，ファイル全体に対して逐次読み出しする場合がほとんどであるという特性を利用して，逐次読み出しに対してプリフェッチ処理を実施する方法である．しかしこの方法はファイルアクセスが一定のパターンで行われる場合のみ有効であるので，JAR ファイルからのクラス読み込みのように，ファイルを不規則にアクセスする場合には効果がない．

5.2 プログラムの静的プロファイルを用いてプリフェッチ処理を行う方法

この手法は，静的プロファイルを用いる点で本手法と類似する．Lei らの手法[10)] では，各実行ファイル

に対して，その実行ファイルがアクセスするファイルとその順序をあらかじめ access tree として記録しておき，それらと実行時のファイルのオープン順序を比較して類似する access tree を検出し，それに基づいて次に open されるであろうファイルをプリフェッチ処理する方法である．また，Griffion らの方法[11] は，あるファイルとそのファイルが open されたときに次に open される可能性のあるファイルとの関連を，各 edge に頻度を持たせたグラフ（probability graph）で表しておき，プリフェッチ処理するファイルを予想する方法である．しかし，いずれの方法もプリフェッチ処理の対象がファイル単位であるので，JAR ファイルのようにファイルの一部しか読み込まないファイルに対しては，不要データ読み込みによる遅延が生じてしまう問題がある．他方，Grimsrud らの方法[12] は，HDD の各クラスタに対して，そのクラスタがアクセスされたときに，次にアクセスされやすいクラスタの情報をその重みとともに保持してプリフェッチ処理を実施する方法である．この方法は，ランダムアクセスファイルに対しても有効である点が利点である．ただし，情報が HDD 上のセクタに関連付けられているために同じファイルを異なるアプリケーションから異なるパターンでアクセスする場合には，かえって性能低下を引き起こす問題もある．

5.3 アプリケーションを書き換えてプリフェッチ処理を行う方法

この方法は，アプリケーションを書き換えることで I/O wait 時間を削減する方法である．この方法として最も単純な方法は，アプリケーション作者がそのアルゴリズムを非同期 I/O 処理を用いて書き換える方法である．しかし非同期 I/O 処理を用いたアプリケーションの書き換えは，アプリケーションの規模が増大するに従って時間的，および作業量的に困難となる．アプリケーションのアルゴリズム自体は書き換えずに，I/O システムに対してアプリケーションの I/O アクセスに関する情報を伝える方法として，Win32 API におけるファイルアクセスのヒント[7] や，Patterson らによる I/O システムに disclosure と呼ぶ I/O アクセスのヒント情報を伝える方法[8] などが存在する．どちらの場合も，提供された情報を用いて I/O システムがプリフェッチ処理を生成して実施する．Win32 API の場合，ファイルに対して逐次アクセス属性とランダムアクセス属性を設定できるが，ランダムアクセス属性に対しては過去 2 回分の履歴しか保持しておらず，複雑なパターンには対応できない．また Patterson らの方法では，アプリケーション中に ioctrl 命令を使って I/O システムにファイルのアクセスパターンなどを伝えるコードを挿入し，I/O システムはその情報をもとにプリフェッチ処理を実施する．この際，I/O システムは評価関数を用いてプリフェッチ処理によりメモリ上に保持されるデータ量とプリフェッチ処理による効果をを考慮して適切なプリフェッチ処理を生成する．しかしながら，この方法も結局は特定のパターンを持ったファイルアクセスにしか対応できていない．

5.4 コンパイラによりプログラムを解析することでプリフェッチ処理を行う方法

この方法[1),2)] は，コンパイラによりプログラムを解析することでファイルのアクセスパターンを検出し，そのアクセスパターンに従ってプリフェッチ処理コードをプログラム中に挿入する．しかしながら，現状では手続き間解析の能力の問題で，ループ内などの限られた範囲でしかプリフェッチ処理コードを生成することができていない．

5.5 プログラムを多重化して生成したプリフェッチ処理プロセスを用いる方法

この方法[3),4)] は，対象とするプログラムから不要処理や副作用命令などを除外することでプリフェッチ処理プロセスを生成し，対象プログラムに対して投機的に実行することで，プリフェッチ処理を実施する方法である．プリフェッチ処理プロセスは基本的にもとのプログラムの実行を縮退させて生成するので，どのようなアクセスパターンに対しても対応可能であるが，反面プログラムを縮退させてしまうためにプリフェッチ処理プロセスの実行状態が元のプログラムの実行と必ずしも一致せず，誤ったデータをプリフェッチ処理してしまう状況を防げない問題が存在する．また，プリフェッチ処理プロセス自体のオーバヘッドにより，本来の実行を遅延させてしまう点も問題となる．Fraser ら[3)] は，OS を修正し I/O 待ち時間が発生した場合にのみこのプロセスを実行させるように制御することで，元のプログラムを遅くしないことを保証する方法を提案しているが，プリフェッチ処理プロセスの実行時間を限定するためにプリフェッチ処理の効果が減少してしまう欠点もある．

5.6 ハードディスクのシーク時間を最小化する方法

この方法[13),14)] は，シークタイム最適化，あるいはエレベータシーキングとして知られている方法である．I/O システムの I/O 要求キューにある I/O 要求を，ハードディスクのシーク処理コストを最小限にするように並べ替える方法である．この方法は同時に I/O 要求キューに入っている I/O 要求に対してのみ適用されるので，処理と処理の間に依存関係があるよ

うな読み出し処理など同時に要求キューに入らない場合には効果がない．

5.7 ハードディスク上のファイルの配置を並べ替える方法

この方法[15)~17)]は，静的プロファイル情報をもとにHDD上のファイルの配置を実行時に連続アクセスとなるように並べ替えることで，シーク処理時間を最小化する手法である．アプリケーションの変更を必要としない点は我々の手法と同じであるが，OS内に実装しないとならない点から，すべてのシステムで利用できる方法ではない．

6. おわりに

本論文では，Scenario-based prefeting手法という，静的プロファイルを用いてアプリケーションに対してプリフェッチ処理を実施する方法を提案した．この方法では，静的プロファイルを解析して，シーク処理にともなうオーバヘッドを削減するようにプリフェッチする順序を並べ替えて生成したシナリオを用いてプリフェッチ処理を実施する．本手法をRedhat9 Linux上でIBM WebSphere Application Server 5.1.1を用いて評価した結果，平均約30%の速度向上を得ることができた．

本手法はアプリケーションや実行環境の変更を必要としない点で，多くの計算機上で利用できるポータブルな手法である．大規模な既存のアプリケーションに対してもプログラムの書き換えを行うことなしにプリフェッチ処理による高速化を実施できる点は非常に有用である．特に，本論文で評価に用いたアプリケーションサーバは，その上で多くのWebアプリケーションが動作するフレームワークであり，この起動時間は多くのWebアプリケーションの運用や開発に影響を与える．本手法によるアプリケーションサーバの起動処理時間の短縮は，プロビジョニングなどのオンデマンドな運用における反応時間の向上や，Webアプリケーション開発時における頻繁な起動と終了の繰返しにおいての開発効率の向上を実現するうえで非常に有効な手法といえる．

謝辞 本研究を行うに際して貴重なご助言をいただいた日本アイ・ビー・エム（株）東京基礎研究所システムズグループ諸氏に感謝いたします．

参考文献

1) Brown, A.D. and Mowry, T.C.: Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently, *Proc. 4th Symposium on Operating Systems Design and Implementation* (*OSDI 2000*), pp.31–44 (2000).
2) Mowry, T.C., Demke, A.K. and Krieger, O.: Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Application, *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp.297–306 (1993).
3) Fraser, K. and Chang, F.: Operating System I/O Speculation: How two invocations are faster than one, *Proc. USENIX 2003 Annual Technical Conference*, pp.325–338 (2003).
4) Yang, C.K., Mitra, T. and Chiueh, T.C.: A Decoupled Architecture for Application-Specific File Prefetching, *Proc. USENIX Annual Technical Conference*, pp.157–170 (2002).
5) Feiertang, R.J. and Organisk, E.I.: The Multics Input/Output System, *Proc. 3rd Symposium on Operating System Principles*, pp.35–41 (1971).
6) McKusick, M.K., Joy, W.J., Leffler, S.J. and Fabry, R.S.: A Fast File System for Unix, *ACM Trans. Computer Systems*, Vol.2, No.3, pp.181–197 (1984).
7) Richter, J.: *Advanced Windows—The Developer's Guide to the Win32(R) API for Windows NT(TM) 3.5 and Windows 95*, Microsoft Press (1995).
8) Patterson, R.H., et al.: Informed Prefetching and Caching, *Proc. 15th Symposium on Operating System Principles*, pp.79–95 (1995).
9) Cao, P., Felten, E.W., Karlin, A. and Li, K.: Implementation and Performance of Application-Controlled File Caching, *Proc. 1st USENIX Symposium on Operating Systems Design and Implementation*, pp.165–178 (1994).
10) Lei, H. and Duchamp, D.: An Analytical Approach to File Prefetching, *Proc. USENIX Annual Technical Conference*, pp.275–288 (1997).
11) Griffioen, J. and Appleton, R.: Reducing File System Latency using a Predictive Approach, *Proc. USENIX Summer 1994 Technical Conference*, pp.197–208 (1994).
12) Grimsrud, K.S., Archibald, J.K. and Nelson, B.E.: Multiple Prefetch Adaptive Disk Caching, *IEEE Trans. Knowledge and Data Engineering*, Vol.5, No.1, pp.88–103 (1993).
13) Gotlieb, C.C. and MacEwen, G.H.: Performance of Movable-Head Disk Strage Devices, *J. ACM*, Vol.20, No.4, pp.604–623 (1973).
14) Geist, R. and Daniel, S.: A Continuum of Disk Scheduling Algorithms, *ACM Trans. Computer Systems*, Vol.5, No.1, pp.77–92 (1987).

15) Akyurek, S. and Salem, K.: Adaptive block rearrangement, *ACM Trans. Computer Systems* (*TOCS*), Vol.13, No.2, pp.89–121 (1995).
16) Vongsathorn, P. and Carson, S.D.: A system for adaptive disk rearrangement, *Software Practice and Experience*, Vol.20, No.3, pp.225–242 (1990).
17) Sivathanu, M., Prabhakaran, V., Popovici, F.I., Denehy, T.E., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: Semantically-Smart Disk Systems, *Proc. 2nd USENIX Conference on File and Storage Technologies*, pp.73–89 (2003).

(平成 17 年 2 月 21 日受付)
(平成 17 年 6 月 27 日採録)

安江 俊明（正会員）

1991 年早稲田大学大学院理工学研究科電気工学専攻修了．1995 年同大学院博士後期課程退学後，日本アイ・ビー・エム東京基礎研究所入社．最適化コンパイラ，並列処理の研究に従事．

小松 秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM 東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

中谷登志男（正会員）

1975 年早稲田大学理工学部数学科卒業．同年日本アイ・ビー・エム（株）入社．1985 年米国プリンストン大学大学院コンピュータ・サイエンス学科より M.S.E. および M.A., 1987 年，Ph.D. を各取得．同年より同社東京基礎研究所においてプログラム最適化技術の設計・実装・評価の研究に従事．2000 年よりディスティングイッシュト・エンジニア．現在同研究所システムズ担当．基礎研究部門におけるコンパイラ・ファームウエア技術のストラテジストを兼任．MICRO-22（1989）および MICRO-23（1990）にて，Best Paper Award を連続受賞．ACM，IEEE 各会員．