オペーレーティング・システム 54－3
（１９９２．３．１３）

ジョブ間並列同期転送機能(PREST)の開発と評価

長須賀 弘文[1] 新井 利明[1] 今居 和男[2] 吉澤 康文[1]
[1]日立製作所システム開発研究所 [2]日立製作所ソフトウェア開発本部

オンライン業務の長時間化に伴い、バッチ処理業務に割当てられる時間が制限されつつある。そこで、バッチ処理業務の高速化を目的に、ジョブ間並列同期転送機能（ＰＲＥＳＴ）を提案し、開発した。ＰＲＥＳＴは、ジョブ実行の並列度を向上させ、ジョブ間でのデータ受渡しのための入出力操作を削除する機能である。ＰＲＥＳＴは、並列に実行しているジョブ間でのデータの受渡しを可能とし、データ受渡しのための入出力操作をデータ転送でシミュレーションすることで、バッチ処理の処理経過時間を短縮する。性能評価をした結果、データ入出力操作にかかるＣＰＵオーバヘッドを約１/６に削減し、処理時間を約１/２に短縮した。

THE DESIGN AND EVALUATION OF PREST

Hirofumi Nagasuka[1] Toshiaki Arai[1] Kazuo Imai[2] Yasufumi Yoshizawa[1]

[1]System Development Laboratory,Hitachi,Ltd.
(1099 Ohzenji Asao-ku Kawasaki-shi,215 Japan)
[2]Software Development Center,Hitachi,Ltd.
(5030 Totsuka-cho Totsuka-ku Yokohama-shi,244 Japan)

We proposed Parallel REference and Synchronous Transfer feature (PREST) which shortens batch exection time. PREST simulates I/O operation which is requested by a job (output job) to transfer temporary data to another job (input job). PREST allows the input job to execute in parallel with the output job. We have developed PREST, and got the following result : (1)CPU overhead for I/O operation is reduced to 1/6. (2)The execution time of a typical batch job, which sorts out and merges data, was shortened to 1/2.

１．はじめに

金融機関等では、オンライン業務のサービス時間延長に伴い、集計処理等のバッチ処理業務に割当てることができる時間帯が制限されつつある。一方で、データ量の増大、業務の複雑化にともないバッチ処理量が増加している。そのため、翌日のオンライン業務の開始までに、バッチ処理業務が終了しない危険性が大きくなりつつある。これを回避するために、バッチ処理業務の高速化を図ることが急務である。

一般的に、集計処理等のバッチジョブは、複数の処理で構成されている。各処理では、入力したデータを加工して後続の処理に渡しながら、全体の処理を進めている。大量のデータを取り扱うバッチジョブでは、ある処理とある処理でのデータの受け渡し（中間データ引継ぎ処理）のための入出力操作に時間がかかる。この傾向は、取扱うデータが大量になるにつれ、顕著である。また、中間データを引継ぐために、ジョブ実行を逐次的にさせる必要がある。

本稿で報告するジョブ間並列同期転送機能（ＰＲＥＳＴ：Parallel REference and Synchronous Transfer feature）は、ジョブ実行の並列度を向上させると共に、ジョブ間でのデータ引継ぎに伴う入出力操作を削除することで、バッチジョブの処理時間を短縮する機能である。（図１．１、図１．２参照）

ＰＲＥＳＴは、ジョブ間で保存する必要のないデータを、逐次アクセス法を利用して引継ぐ処理を高速の対象とする。データを出力するジョブ（出力ジョブ）と、その出力されたデータを入力するジョブ（入力ジョブ）を並列に実行させることで、処理時間を短縮する。データの引継ぎのための入出力操作は、全てメモリ上でのデータ転送でシミュレーションする。これにより、入出力操作を削除する。

２．ＰＲＥＳＴの特徴

ＰＲＥＳＴと同様の機能として、ＵＮＩＸのＰＩＰＥ機能がある[2]。

ＰＲＥＳＴは、汎用大型計算機用ＯＳの上で実現した。汎用大型計算機用のＯＳには、ファイルへのアクセス方法が複数ある。その中で、バッチ

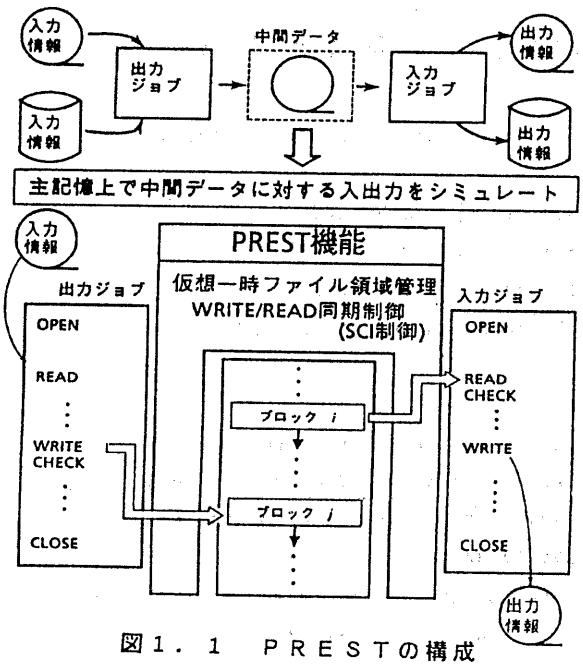

図１．１　ＰＲＥＳＴの構成

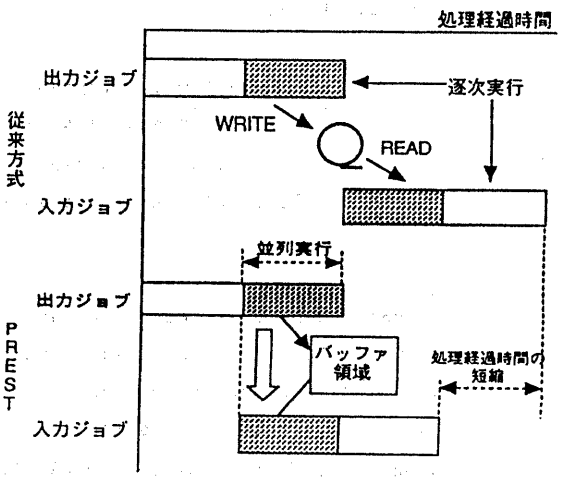

図１．２　ＰＲＥＳＴの効果

処理で頻繁に利用される逐次アクセス法に着目しこのアクセス法を用いたデータ引継ぎ処理の高速化を図ることとした。また、既存プログラムを改造することなしに、ＰＲＥＳＴの適用を可能とすることで、汎用性をもたせた。さらに、ＰＲＥＳＴの指定を、ジョブ起動用のジョブ制御文で指定するようにしたことで、柔軟性をもたせた。[1]

内部処理では、データ引継ぎのための出力ジョブと入力ジョブとの待ち合わせ処理（同期処理）回数の削減を図るＳＣＩ制御(Synchronous Control for Intermediate data)や、メモリの有効活用を図った動的バッファ容量決定方式に特徴がある。

３．ＰＲＥＳＴ

ＰＲＥＳＴの実現にあたっての課題と、その解決策について述べる。

３．１　データの転送方式

ＰＲＥＳＴでは、入出力操作をデータ転送でシミュレーションする。そこで、この転送用のマクロ命令を新設し、ユーザプログラムにマクロ命令を発行させる方式を考られるが、この方式をとると、ＰＲＥＳＴを利用するために既存のユーザプログラムを改造しなければならない。そこで、ユーザプログラムが発行した入出力命令（マクロ命令）をトラップして、データ転送に置き換える方式をとった。これにより、既存のユーザプログラムを書き換えることなしに、ＰＲＥＳＴを利用することができる。

また、出力ジョブと入力ジョブは、異なるアドレス空間上で動作している。空間間で直接データ転送を行うと、空間切り替え処理などでＣＰＵオーバヘッドが大きくなる恐れがある。そこで、出力ジョブと入力ジョブが共に参照できる領域にバッファを設け、出力ジョブが出力命令を発行するとそのバッファ領域にデータを書き込み、入力ジョブが入力命令を発行するとそのバッファ領域からデータを読み込む方式をとった。

３．２　ジョブ間の同期処理

並列に実行しているジョブ間でデータの受け渡しをする場合、例えば、入力ジョブが入力命令を発行したときに、出力ジョブがデータを出力していなければ、出力ジョブがデータを出力するまで、入力ジョブを待たせなければならない。このような待ち合わせ処理を同期処理と呼ぶ。

ＰＲＥＳＴでは、データ引継ぎのための出力ジョブと入力ジョブの同期処理を行い、データ引継ぎを保証する。

３．３　バッファ領域の管理

出力ジョブから入力ジョブへのデータ引継ぎを、バッファを介したデータ転送で実現する。そこで、割当てるバッファの容量が問題となる。最初に、取り扱うデータ量分のバッファを確保すると、大量データを取り扱う場合にメモリオーバヘッドが膨大になり、取り扱えるデータ量も制限される。

そこで、比較的小容量のバッファ領域を確保し、ラップアラウンドに使用することとした。これにより、取り扱えるデータ量にも制限がなくなる。

３．４　ＰＲＥＳＴ指定方法

ＰＲＥＳＴを利用するジョブに関する情報等を予めシステムに登録する方法も考えられるが、ＰＲＥＳＴの利用を容易とするため、ジョブ制御文で指定することとした。（図３．１参照）

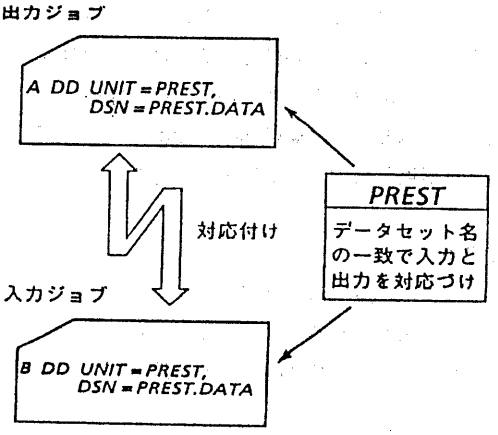

図３．１　ＰＲＥＳＴの指定方法

4．ＰＲＥＳＴの制御方式

本章では、同期処理方式とバッファ容量の管理方式について述べる。

同期処理に関しては、出力ジョブと入力ジョブが入出力命令を発行する度に同期をとると、そのオーバヘッドは膨大になるという問題点が残っている[2]。また、バッファ容量に関しては、どの程度のバッファ領域を用意すればよいのかが明確でないという問題点が残っている。

そこで、同期処理の頻度が小さく、かつ小容量のバッファを用意することで、効果的にデータ引継ぎ処理を実現できる同期処理方式とバッファ容量の管理方式を求めるために、次節に示すモデルを用いて解析した。

4．1　解析モデル

【モデル】

（1）バッファ領域は、先頭ブロックから順次使用する。

（2）出力ジョブがバッファ領域に書き込んだデータを、入力ジョブがすべて読み込んだ後、入力ジョブは、出力ジョブがmブロック分のデータを書き込むまでまつ。

（3）出力ジョブがバッファ領域のすべてのブロックにデータを書き込み、入力ジョブが、バッファ領域内のすべてのデータを読み込んでいをない場合、出力ジョブは、入力ジョブがmブロック分のデータを読み込むまで待つ。

（4）出力ジョブの書き込み要求と、入力ジョブの読み込み要求には、相互関係はない。

（5）目標とする同期処理の頻度は、α回のデータ引継ぎに対して１回未満である。

【記号】

λ：書き込み頻度（回／ｓｅｃ）

μ：読み込み頻度（回／ｓｅｃ）

ｎ：バッファ領域のブロック数（ブロック）

【検証１】書き込み頻度が読み込み頻度よりも多い場合（$\lambda > \mu$）

出力ジョブが待ち状態になり、入力ジョブがmブロックのデータを読み込んだ後、出力ジョブの待ち状態を解除したとする。待ち状態を解除した後、再び出力ジョブが待ち状態になるまでの時間ｔは、式４．１で求められる。

$$\lambda t = \mu t + m$$

$$\therefore t = m/(\lambda - \mu) \quad \cdots \text{式４．１}$$

この時間ｔが長ければ、同期処理を行う頻度が小さくなる。バッファのブロック数はｎであるので、ｔの最大値は、ｍ＝ｎの時である。

これは、一度待ち状態になった出力ジョブは、入力ジョブがバッファ内のデータをすべて読み込むまで待ち状態になっていることが、同期処理の頻度を最小にすることを意味する。

さらに、時間ｔの間に、α回以上のデータ書き込みを行うことができれば、α回のデータ書き込みに対して、１回未満のジョブジョブ間の同期処理で済ませることができる。この目標を満たす最小のバッファ容量（ブロック数）n^*は、式４．２で求められる。

$$\lambda t = \lambda n/(\lambda - \mu) > \alpha$$

$$\therefore n^* \geqq (\alpha(\lambda - \mu)/\lambda) + 1 \quad \cdots \text{式４．２}$$

【検証２】読み込み頻度が書き込み頻度よりも高い場合（$\lambda < \mu$）

入力ジョブが待ち状態になり、出力ジョブがmブロックのデータを書き込んだ後、入力ジョブの待ち状態を解除したとする。待ち状態を解除した後、再び入力ジョブが待ち状態になるまでの時間ｔは、式４．３で求められる。

$$\mu t = \lambda t + m$$

$$\therefore t = m/(\mu - \lambda) \quad \cdots \text{式４．３}$$

この時間ｔが長ければ、同期処理を行う頻度が小さくなる。バッファのブロック数はｎであるので、ｔの最大値は、ｍ＝ｎの時である。

これは、一度待ち状態になった入力ジョブは、出力ジョブがバッファ内のすべてのブロックにデータを書き込むまでまで待ち状態になっていることが、同期処理の頻度を最小にすることを意味する。

さらに、この時間ｔの間に入力ジョブが、α回

以上のデータ読み込みを行うことができれば、α回のデータ読み込みに対して、1回未満のジョブ間の同期処理で済ませることができる。この目標値を満たす最小のバッファ容量（ブロック数）n*は、式4．4で求められる。

$$\mu t = \mu n/(\mu-\lambda) > \alpha$$

$$\therefore n^* \geqq \{\alpha(\mu-\lambda)/\mu\} + 1 \quad \cdots \text{式4．4}$$

式4．2と式4．4から、書き込み要求頻度と読み込み要求頻度に応じて、式4．5に示す容量（ブロック数）のバッファ領域を確保すればよいことになる。

$$n^* \geqq \frac{\alpha|\lambda-\mu|}{max(\lambda,\ \mu)} + 1 \quad \cdots \text{式4．5}$$

以上のモデルを用いた解析をもとに、次節以降で示す同期処理の方式と、バッファ容量の管理方式を採った。ＰＲＥＳＴで用いた同期処理の方式をＳＣＩ制御と呼び、バッファ容量の管理方式を動的バッファ容量決定方式と呼ぶ。

4．1　ＳＣＩ制御

ＳＣＩ制御では、与えられたバッファ容量のもとで、同期処理の頻度が最小になるように制御する。すなわち、ジョブの待ち状態の解除を、以下に示すタイミングで行う。（図4．1参照）

（1）出力ジョブがデータ出力要求を出した時に、バッファ領域に空きがなかった場合、入力ジョブが、バッファ内のすべてのデータを入力するまで出力ジョブを待ち状態にする。

（2）出力ジョブがまだ出力していないデータに対して入力ジョブが入力要求を発行した場合、出力ジョブがバッファ内のすべての領域にデータを出力するまで、入力ジョブを待ち状態にする。

4．3　動的バッファ容量決定方式

出力ジョブと入力ジョブの入出力要求頻度が大きく異なる場合、バッファが小容量であれば、ＳＣＩ制御を施しても、ジョブ間で同期をとらなければならない機会が多くなる。そこで、動的バッファ容量決定方式では、出力ジョブと入力ジョブの入出力要求頻度の割合いに応じて、バッファ容量を決定することで、メモリの有効活用を図ると共に、同期処理にかかるオーバヘッドをある一定値以下に抑える。

すなわち、出力ジョブと入力ジョブの入出力要求頻度の割合いから、同期処理と同期処理の間に引継がれるデータ数がある設計値以上となるために必要なバッファ容量を求め、増量する。

したがって、データの引継ぎ処理を行いながら、ＰＲＥＳＴは、下記の処理を行い、メモリの有効活用を図り、同期処理の頻度を小さくする。

（1）初期割当て量として、一定容量のバッファ領域を割当てる。

（2）同期処理と同期処理との間に引継がれたデータ数をカウントする。

（3）（2）でカウントしたデータ数がある設計値未満で

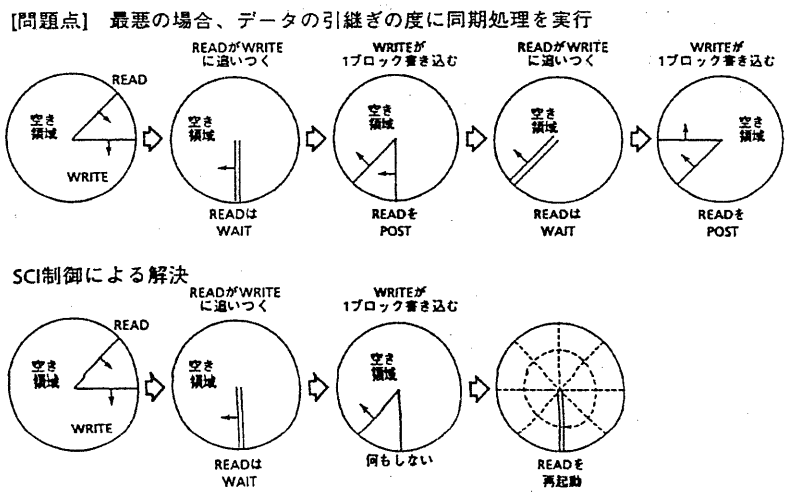

図４．１　ＳＣＩ制御

あれば、設計値を満たす最小のバッファ容量に増量する。

本開発では、最初に１０ブロックのバッファを割当て、同期処理の頻度が、５０回のデータ転送に対して１回未満となるような、最小のバッファ容量に増量するようにした。これにより、１回の入出力操作に占める同期処理のＣＰＵオーバヘッドが十分小さくなると考えたからである。

表４．１は、５０回のデータ転送に対して、同期処理の頻度を１回未満にするために必要なバッファ容量を示したものである。

表４．１ 入出力頻度とTRAD領域のﾌﾞﾛｯｸ数の関係

μ＼λ	１０	２０	３０	４０	５０	６０
１０	1	２６	３５	３９	４１	４３
２０	２６	1	１８	２６	３１	３５
３０	３５	１８	1	１４	２１	２６
４０	３９	２６	１４	1	１１	１８
５０	４１	３１	２１	１１	1	１０
６０	４３	３５	２６	１８	１０	1

ここで、λ：出力要求頻度(1/sec)

μ：入力要求頻度(1/sec)

５．評価

５．１ 単体入出力にかかるＣＰＵオーバヘッド

１ブロックのデータの入出力操作をシミュレーションするために要するＣＰＵオーバヘッド（Ｉ／Ｏ処理ステップ数）と、入出力装置へのＩ／Ｏ処理ステップ数を測定した。その結果、Ｉ／Ｏ処理ステップ数を約１／６に削減した。（図５．１参照）

図５．１において、同期処理用のＣＰＵオーバヘッドは、ＳＣＩ制御による同期処理を実行した時のオーバヘッドである。バッファ管理用のＣＰＵオーバヘッドは、データ引継ぎの実行中にバッファ領域を拡張するか否かを判断する時に、出力ジョブと入力ジョブの実行をシリアライズさせるために必要なＣＰＵオーバヘッドである。

図５．１で示した同期処理用ＣＰＵオーバヘッドとバッファ管理用ＣＰＵオーバヘッドは、１回の入出力操作当りに換算したものである。

これら、同期処理用のＣＰＵオーバヘッドとバッファ管理用のＣＰＵオーバヘッドは、ＰＲＥＳＴのＩ／Ｏ処理ステップの約３５％を占めている。

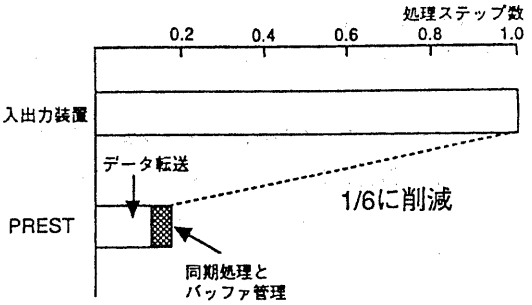

図５．１ Ｉ／Ｏ処理ステップ数の比較

５．２ SCI制御と動的ﾊﾞｯﾌｧ容量決定方式の評価

マルチプロセッサ環境下とユニプロセッサ環境下とにおいて、ＰＲＥＳＴを用いたデータ引継ぎ処理を行うバッチジョブを実行させ、同期処理の頻度と、データ引継ぎ中のバッファ容量の変化の様子を観測した。最終的なバッファ容量と、同期処理を実行した頻度を表５．２に示す。

表５．２ ﾊﾞｯﾌｧ容量の変化と同期処理頻度

	ﾕﾆﾌﾟﾛｾｯｻ	ﾏﾙﾁﾌﾟﾛｾｯｻ
バッファ容量	１０⇒５０	１０
同期処理頻度	1/４８．９	1/１７４．８

同期処理の頻度は、設計値に近い値かもしくは設計値を満たす値となっている。ユニプロセッサでは、データ引継ぎ中にバッファを５０ブロックに増量したのに対し、マルチプロセッサは、初期割当てである１０ブロックのままで、データ引継ぎを完了した。この違いは、以下のことが要因となっていると思われる。

（１）ユニプロセッサの場合

ＣＰＵが一つのため、ある瞬間に実行可能なジョブは、出力ジョブか入力ジョブのどちらか一方である。したがって、出力ジョブが、全てのバッファ

領域にデータを書き込んで待ち状態になり、次に入力ジョブがバッファ領域の全てのデータを読み込んで、出力ジョブの待ち状態を解除するといった処理を繰り返していた。このため、同期処理を行う頻度が多くなり、途中でバッファ容量を増量する処理を行った。

（2）マルチプロセッサの場合

ＣＰＵが複数あるため、出力ジョブと入力ジョブが、並列に実行することが可能である。したがって、同期処理を行う頻度が小さく、初期割当てのバッファ容量のまま、データの引継ぎ処理を完了した。

５．３　バッチジョブのモデルを用いた評価

バッチ処理モデルを用いて、従来の処理方式の場合とＰＲＥＳＴを用いた場合との処理経過時間を比較した。

（1）評価モデル

評価に用いたジョブは、店舗での商品別売上げの集計処理を想定したものである。(図５．２参照)

取り扱うデータは、２種類である。

（a）入力データ：商品毎の売り上げ数

（b）マスタデータ：商品の在庫数

処理内容は、以下の通りである。

（a）ソート処理：入力データをソートする。

（b）マージ処理：入力データとマスタデータから、現在の在庫数を求め、新しいマスタデータとして出力する。

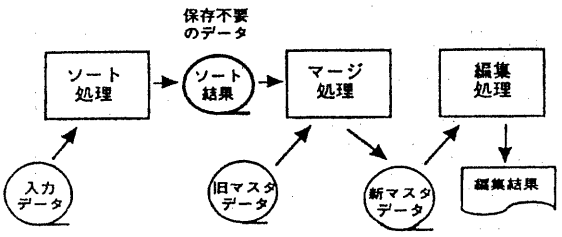

図５．２　評価モデルの構成

（c）編集処理：新しいマスタデータから、品切れ状態にある商品を求め、編集結果を出力する。

このバッチ処理において、ソート処理とマージ処理との間でのデータ引継ぎ処理にＰＲＥＳＴを適用し、処理経過時間を比較した。

（2）処理経過時間の比較

取り扱うデータ量と、データ引継ぎ用に使用する記憶媒体を変えて、各々の処理経過時間を比較した結果を図５．３に示す。

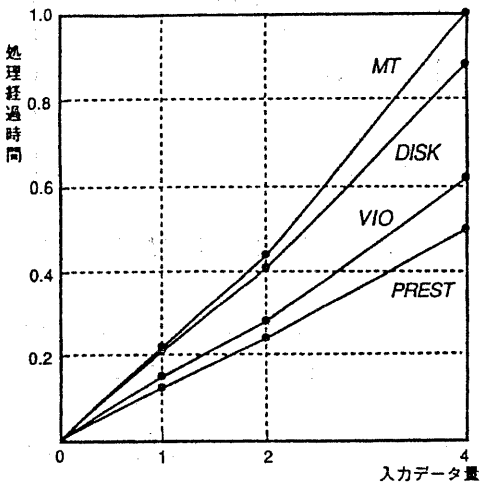

図５．３　処理経過時間の比較

図５．３において、データ量４の場合の処理経過時間の内訳を図５．４に示す。

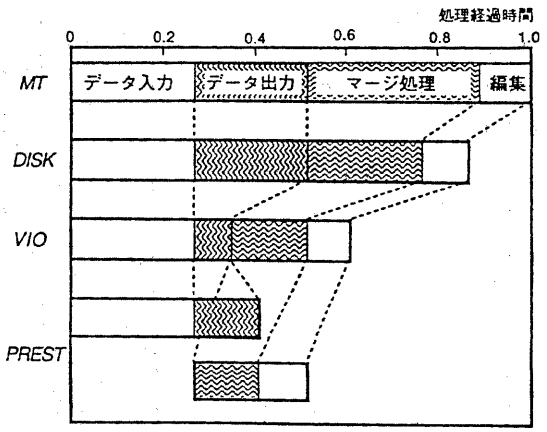

図５．４　処理経過時間の内訳

（３）ﾃﾞｰﾀ引継ぎ処理部分の処理経過時間の比較

ＰＲＥＳＴを適用したデータ引継ぎ処理部分に着目して、処理経過時間を比較する。（図５．５参照）

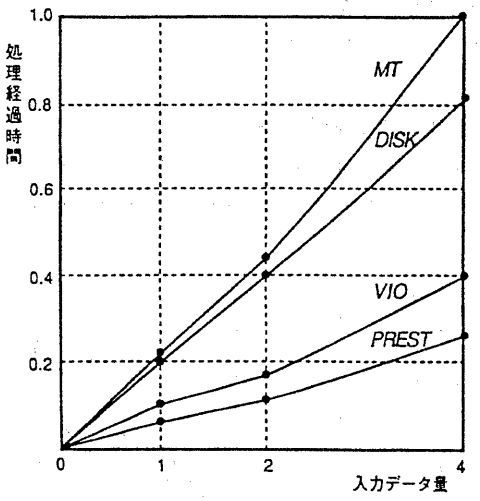

図５．５　データ引継ぎ処理の比較

ＰＲＥＳＴとＶＩＯを比較すると、約３５％の処理経過短縮程度となっている。マージ処理では、入力データの入力、マスタデータの入力、および新マスタデータの出力と、３種類のデータの入出力を一緒に行っており、マスタデータの実入出力操作のために、処理時間の短縮がこの程度となっている。

６．バッチ処理業務への適用

開発したＰＲＥＳＴを、流通系のバッチ処理業務に適用し、その効果を実測した。

ＰＲＥＳＴを適用したバッチ処理の規模は以下の通りである。

（１）ｼﾞｮﾌﾞ本数：１７

（２）ｼﾞｮﾌﾞｽﾃｯﾌﾟ数：４８７

（３）PRESTの適用箇所：７９

実測の結果、従来は約６時間２０分かかっていたバッチ処理を、約４時間１５分に短縮した。図６．１に、タイムチャートを示す。

７．おわりに

バッチ処理経過時間を大幅に短縮する機能であるＰＲＥＳＴを開発し、評価をした。その結果、ジョブ間でのデータ引継ぎに要するＣＰＵオーバヘッドを１／６に削減した。評価モデルとしたバッチ処理にＰＲＥＳＴを適用した結果、処理経過時間を約１／２（対ＭＴ比）に短縮した。

さらに、流通系のバッチ処理業務にＰＲＥＳＴを適用したところ、処理経過時間を従来の約３０％短縮した。

【参考文献】

［１］長須賀他、ｼﾞｮﾌﾞ間並列同期転送機能の開発と評価　-VOS3/AS:PREST-、第41回情報処理学会講演論文集（４）、1990年9月

［２］Maurice J.Bach,THE DESIGN OF THE UNIX OPERATING SYSTEM,1986

［３］P.J.Courtoris , Concurrent Control with "Readers" and "Writers" , Communication of the ACM , 1971

注）UNIX ｵﾍﾟﾚｰﾃｨﾝｸﾞ ｼｽﾃﾑは、UNIX ｼｽﾃﾑ ﾗﾎﾞﾗﾄﾘ社が開発し、ﾗｲｾﾝｽしています。

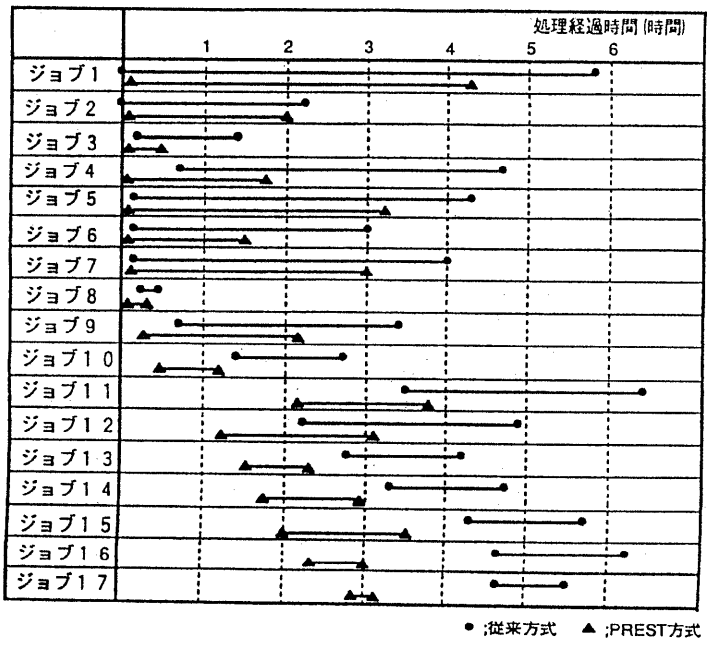

図６．１　バッチ処理業務のタイムチャート