社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－OS－103 (12)
2006／7／31

Linuxにおける共有メモリを保持するプロセスのマイグレーション機構

藤田 肇† 松葉 浩也† 石川 裕†

クラスタはハイエンドコンピューティングの分野においてますます広く用いられるようになってきている．そうしたシステムにおいて，クラスタの性能と可用性を向上させる重要な技術の1つがプロセスマイグレーションである。本研究では，既に我々が提案したクラスタにおける単一IPアドレスプロトコルスタックと分散共有メモリとを統合した透過的なプロセスマイグレーション機構を提案する．この機構のもとでは，共有メモリ領域を通じて他のプロセスと通信を行うプロセスや，ソケットを通じて別のホストと通信を行っているプロセスを移送し，異なるホストで実行を継続することができる．予備実装によるシステムをApache HTTPサーバによる動的コンテンツの配信を用いて評価した結果，ワーカプロセスを4台のクラスタ上に分散させることにより，使用するノード数に比例した性能向上が得られることが示された．

Process Migration System with Shared Memory Region on Linux

HAJIME FUJITA,† HIROYA MATSUBA† and YUTAKA ISHIKAWA†

Cluster systems are becoming more popular in high-end computing. In such systems, process migration is one of the most important techniques for improving performance and availability. In this research, we propose a transparent process migration mechanism that is integrated with distributed shared memory and single IP address cluster which we introduced before. By using this system, processes communicate with other process by means of shared memory or socket are able to keep running at another host. Experimental results with a preliminary implementation show that our system enables Apache HTTP Server to be distributed among cluster nodes and the system improves Apache's performance in dynamic content creation. The performance is linearly proportional to the number of nodes at least up to four nodes.

1. はじめに

今日，既にインターネットは我々の生活に欠かせないインフラとなっている．インターネットユーザ数の増加に伴って，サーバに求められる処理能力もまた増加する一方である．

個々のプロセッサの処理能力の向上に限界が見えてきた現在，さらなる処理性能向上のためには並列処理が不可欠であり，中でもクラスタ[6]は比較的安価に中～大規模の並列処理環境を実現する手段として，数値計算分野だけでなくサーバの分野でも広く使われるようになっている．

クラスタは基本的に分散メモリ型並列計算機であるため，その並列性能を利用するためには明示的な通信処理を記述したプログラミングを行う必要があり，煩雑であることが多い．またクラスタは安価なPCサーバ等を利用して構築されることが多く，個々のノードの故障という可能性が常に存在する．

プロセスマイグレーション[5]を用いると，各ノードの負荷が均等になるようにプロセスを分散させたり，障害の発生したノードからプロセスを別のノードに避難させたりすることができ，クラスタシステム全体としての性能や可用性が向上する．しかしながらアプリケーションの中には共有メモリ領域を通じて他のプロセスと通信している場合や，ソケットを通じて他の計算機と通信している場合もある．そのようなプロセスは単純に他のホストに移したのでは正しく実行を継続することができない．また，特にサーバ分野においては，ユーザが自らプログラムを書いて直接走らせるのではなく，既存のサーバアプリケーションを配置して運用することも多い．優秀なサーバソフトウェアの多くがオープンソースプロダクトとして入手可能であるとはいえ，全ての製品のソースコードが入手可能なわけではなく，またパッケージ管理や更新の手間を考慮するとたとえオープンソースプロダクトであってもバイナリパッケージを変更なしに運用したいという要望は大きい．

本研究では，既存のLinuxアプリケーションとのバイナリ互換性を維持しつつプロセスマイグレーションの機能を提供する機構をLinuxカーネルに実装す

† 東京大学情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

表 1 追加したシステムコール

システムコール	説明
`int migrate(int pid, unsigned long addr)`	`pid` で指定されたプロセスを `addr` で指定されたノードへ移送する.
`int restart(int fd)`	移送元からの要求を受け取ってプロセスの実行を再開する. `fd` はリモートからの接続を受け付けたソケット.

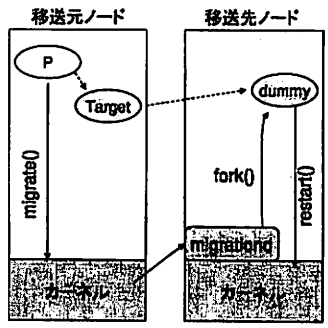

図 1 プロセスマイグレーションの基本設計

る. この機構は分散共有メモリおよび SAPS(Single IP Address Protocol Stack)[13] と統合され, プロセス間共有メモリやソケットを用いて通信を行っているプロセスをバイナリ互換性を維持したまま移送することができる. さらにこの機構により, Web サーバーによる動的コンテンツ生成の負荷を分散し, ノード数に比例する性能向上を得ることができる.

2. 設　　計

2.1 プロセスマイグレーションの設計

プロセスマイグレーション機能を追加するにあたり, 次のような環境を前提とした. まず, プロセスの移動元及び移動先の 2 台のコンピュータは同じ CPU アーキテクチャを採用し, 同じカーネルが動作しているものとする. また, 2 台のコンピュータは同じファイルシステムを共有しているものとする.

マイグレーション機能を追加するために, Linux カーネルにシステムコールを追加した. 追加したシステムコールを表 1 に示す. マイグレーションの基本的な設計は図 1 のようになっている. マイグレーション機能を実装したカーネルに加え, ユーザーモードデーモンである `migrationd` を移送先ノードに用意しておく. 移送元ノードにおいて, あるプロセス (図 1 中'P') が対象プロセス (図 1 中'Target') を別のノードに移送したいとする. このとき, プロセス P は `migrate` システムコールを発行し, カーネルに対象プロセスと移送先を指定する. `migrate` システムコールが発行されると, カーネルは対象として指定されたプロセスの実行を一旦中断し, 移送先の `migrationd` に接続する. `migrationd` は接続を受け付けると, 自分自身を `fork` して子プロセスを作り, 子プロセスが `restart` システムコールを発行する. 移送先のカーネルは `restart` システムコールの中で移送先からプロセス情報を受け取り, `restart` システムコールを発行したプロセスを上書きする形で移送前のプロセスの状態を復元する.

プロセスマイグレーション機構はカーネルによって提供されており, マイグレーション対象のプログラムは何らマイグレーションを前提とした書き方をする必要はなく, また特別なライブラリ等とのリンクも必要ない.

2.2 分散共有メモリの設計

プロセス間でメモリ領域を共有しているプロセス群のうち一部のプロセスが他のノードへ移った場合, 共有メモリ領域の変更がノードを越えて伝わらなくなってしまう. そこで, 共有メモリ領域を持つプロセスを移送する際には, その領域のページを分散共有メモリの管理下に移す.

分散共有メモリのプロトコルとして, IVY[9] が用いているようなページ単位の Write-Invalidation プロトコルを用いる. このプロトコルでは, あるページへの書き込みが許される計算機は高々 1 つであり, ある計算機が書き込み権限を取得している間, 他の計算機はそのページを参照することはできない. また, あるページに関して書き込み権限を持っている計算機が 1 つも存在しない場合, そのページを複数の計算機で同時に読むことができる. これはハードウェアキャッシュのコヒーレンス制御に用いられる MSI プロトコルと同じである. このプロトコルでは, 各ページは変更可能 (M), 共有 (S), 無効 (I) のどれかの状態を取る. 変更可能なページには読み書きが可能である. 共有状態のページには読み込みのみが許される. 無効なページには一切の読み書きができない (図 2). 書き込み禁止または無効化されたページにプロセスがアクセスするとページフォルトが発生し, カーネルのハンドラへと処理が移る. カーネルはこのページフォルトハンドラの中で, それが分散共有メモリの領域へのアクセスであればページ所有権を持つノードに対してページ要求を出す.

分散共有メモリの性能を向上させるために一貫性を緩和したメモリモデルや実装が多数提案されている[2] が, それらの緩和されたモデルは共有領域へのアクセスには明示的なロック操作が伴うことを前提としており, 既存のアプリケーションをそのままでは動作させられない可能性がある. SMP 環境で動作することを前提に書かれたサーバアプリケーションの場合, 厳密なデータの一貫性が保たれなくてもかまわないような

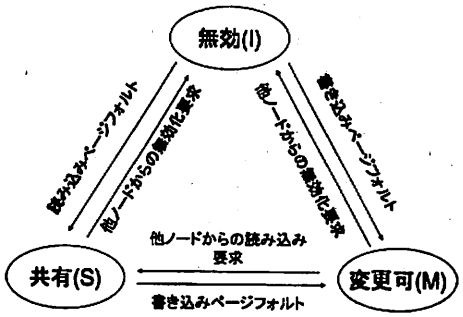

図 2 ページの状態遷移

データに関しては明示的なロックを取得せずにメモリアクセスを行うことがある．例えば Apache HTTP サーバ[3] は各ワーカープロセスの統計情報の記録と参照のために共有メモリを用いるが，その領域へのアクセスに際して特にプロセス間での同期操作は行っていない．このような，明示的なロックを記述しないで行われる共有メモリ領域へのアクセスにおいても，あるノードにおけるデータの変更を別のノードに正しく反映させる必要がある．

2.3 ソケットのマイグレーション

移送されるプロセスがソケットを通じて他のホストと通信を行っている場合，プロセス移送後もプロセスが正しく動作するためには，何らかの手段で移動先のホストにおいて通信を継続できるようにしなければならない．個々のノードが別々の IP アドレスを持っている場合，プロセスの移動によってプロセスが送受信に利用する IP アドレスが変わることになるため，通信相手がマイグレーションを考慮して設計されていない限り通信を継続することはできない．また，特に TCP は複雑なステートやバッファを管理しながら通信を行っているため，ある時点の通信状態を別のホストに移すことは簡単ではない．

我々はプロセスマイグレーション機構を SAPS[13] と統合することにより，TCP ソケットのマイグレーションを実現する．SAPS ではクラスタ内のノードはユーザーから見える IP アドレスを持たず，すべての IP 通信を I/O サーバを経由させることによってクラスタ全体を 1 つの IP アドレスに見せる．この場合，プロセスは TCP コネクションを維持したまま，アプリケーションノードの間を自由に移動できる．

SAPS を利用した場合の TCP コネクションのマイグレーションは以下のように行われる．

(1) 移動元アプリケーションノードのカーネルは I/O サーバに対しマイグレーションの開始を通知する．

(2) I/O サーバは当該プロセスのソケットに関するデータの転送を一時的に中止し，マイグレーション開始の許可を返す．これ以降，通信の再開までに外部ホストから受け取ったデータについては I/O サーバが保管する．

(3) 許可を受け取ったら，移動元のカーネルはプロセスのソケットに関する情報と，プロセスのソケットバッファに残っているデータがあればそれらを併せて移動先ノードへ転送する．

(4) 移動先アプリケーションノードのカーネルは必要なデータを受け取った後，I/O サーバに対して通信の再開を要求する．これ以降，I/O サーバは移動先のアプリケーションノードに対してそのコネクションに対するデータを転送するようになる．

3. 実　　装

我々はこのプロセスマイグレーション機構の予備実装を i386 アーキテクチャ上の Linux 2.6.14 に対して行った．

3.1 プロセスマイグレーションの実装

3.1.1 マイグレーションの開始

プロセスのマイグレーションは，ユーザーや各種デーモン，あるいはカーネル自身の判断によって任意のタイミングで強制的に行えることが望ましい．またプログラムのバイナリ互換性という観点から考えると，プロセスの外部からマイグレーションを指示する機構が必要とされる．

我々はこの目的を達成するため，シグナルを用いたマイグレーションの開始機構を実装した．`migrate` システムコールが発行されると，まずカーネルはシステムコールを発行したプロセスがマイグレーション対象自身であるかどうかを調べ，自身であった場合にはそのシステムコールを発行したプロセスのコンテキストを用いてマイグレーション処理を開始する．そうでない場合では，`migrate` システムコールは `kill` システムコールのように振る舞い，カーネル内部で対象プロセスに対して `SIGSTOP` シグナルを送る．この際，対象プロセスのプロセス構造体にマイグレーション要求を示すフラグを立てておく．次に対象プロセスがスケジューラによって選択された際にはシグナルハンドラの起動へと処理が移るが，カーネル内で `SIGSTOP` を処理する部分にコードを加え，マイグレーション要求が来ているかどうかをチェックする．マイグレーション要求が来ていたならそのシグナル処理のコンテキストを用いてマイグレーションを開始する．

システムコールの実行中にマイグレーションを要求された場合には，システムコールがシグナルによって割り込まれたように見える．システムコールの中には自動的にそのシステムコールの再実行を要求するものがあり，その場合にはプロセスの実行再開後にシステムコールを再開できるよう，システムコールを発行したソフトウェア割り込み命令の位置まで命令ポインタ

を戻しておく．

3.1.2 情報の収集と転送

プロセスの実行状態を別のホストに移すためには，現在稼働中のプロセスの情報を収集し転送しなければならない．必要な情報は以下のようなものである．

- ユーザーモードにおけるレジスタの値
- 使用しているメモリマッピングの範囲と属性
- 開いているファイル名と現在のファイルポインタのオフセット，ファイルディスクリプタの値
- 開いているソケット
- 登録されているシグナルハンドラとシグナルのマスク状態
- プロセスのメモリ空間にマップされているページの内容

このうち，レジスタの値の取り出しと復元，シグナルに関する情報の収集と復元には Linux 向けチェックポイントライブラリである BLCR[7] のコードを使用している．レジスタを操作する部分以外のコードは CPU アーキテクチャに非依存である．

現在のプロセスマイグレーションの実装は，カーネル間の通信に TCP/IP を使用している．このためマイグレーションの開始にあたって，移送元のカーネルは移送先で待機している待ち受けデーモン `migrationd` に対して接続要求を出す．一旦接続が確立されると，それ以降はその接続を利用して必要なデータを送信する．

3.1.3 プロセスの再開

プロセスの再開にあたっては，再開されるプロセスの雛形となるダミーのプロセスが必要である．現在の実装では，`migrationd` が自らを `fork` して新しいプロセスを作り，その上に送られてきたプロセスコンテキストを復元する形をとっている．

新しいプロセスによって `restart` システムコールが発行されると，カーネルはプロセスの再開処理を開始する．`restart` システムコールは `migrationd` から移送元ホストとの接続を引き継ぎ，移送元ホストのカーネルからプロセスの情報を受け取る．このシステムコールは UNIX の `execve` システムコールのように振る舞い，成功した場合現在のプロセスコンテキストを再開されるプロセスのコンテキストで上書きする．すなわちこのシステムコールはユーザーモードのレジスタの値を書き換え，現在のメモリマッピングを全て解除し，転送されてきた情報をもとに移送前のプロセスのメモリ空間を復元する．`restart` システムコールからユーザーモードに返ると，`migrate` システムコールによる `SIGSTOP` を受け取った直後の状態からプロセスの実行が再開される．

3.1.4 ページ転送に関する最適化

プロセスのメモリ空間にマップされているメモリ領域のうち，存在しないページや変更が加えられていないページついては移動先に転送しなくてもよい．転送されなかったページについては，移動先でプロセスの実行が再開された後にページフォルトが発生することになる．このとき，カーネルは新しいページを用意しなければならない．プロセスがアクセスしたメモリ領域が無名領域，すなわちファイルにマップされていない領域であればゼロクリアされたページを用意し，ファイルにマップされた領域であればメモリアドレスに対応するファイルの内容を読み込んだページを用意する必要がある．

2.1 節で述べたように，マイグレーションの実装は移動元と移動先が同じファイルを共有していることを前提としている．このため，ファイルの内容はどのホスト上でみても同じであることが期待できる．よってファイルにマップされた領域であっても変更されていないページはマイグレーション時に転送する必要がない．一般的にファイルのメモリへのマッピングは実行ファイルや共有ライブラリのロードのために行われることが多く，こうした共有ライブラリはすでにメモリ上にロードされている可能性が高いと考えられる．従って不要なページの転送を削減することで，プロセスマイグレーションに関わるオーバーヘッドを削減することができる．

あるページが変更されていないかどうかを判定する際には注意が必要である．まず，メモリ領域の属性が書き込み禁止になっているだけでは不十分である．これは，`mprotect` システムコールによってメモリ領域の内容を変更した後に書き込み禁止とすることができるためである．また，プロセスのページテーブルエントリの変更ビットを調べるだけでも不十分で，そのページをマップしている全てのプロセスのページテーブルエントリを調べる必要がある．

3.2 分散共有メモリの実装

3.2.1 Linux における共有メモリ

Linux において複数プロセス間で共有されるメモリ領域には次のような種類がある．

- ファイル共有マッピング
- System V 共有メモリ
- POSIX 共有メモリ
- `/dev/zero` に対する共有マッピング
- 無名共有マッピング

Linux では，これらの領域は全て他のプロセスと共有されることを示すフラグがつき，かつ何らかのファイルをマッピングしているように見える*．これは，Linux のプロセス間共有メモリがファイルごとのページキャッシュを基に実現されているためである．複数のプロセスが 1 つのファイルのページキャッシュを共有することで，メモリの共有を実現している．

* 無名共有マッピングにおいても，実際には mmap システムコールの内部で `/dev/zero` を開いてマップしている．

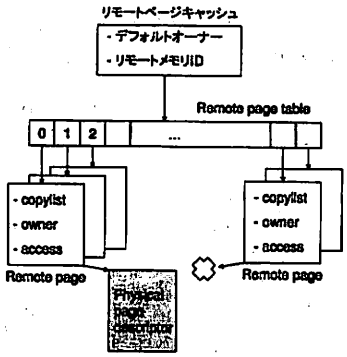

図 3 リモートページキャッシュ

3.2.2 リモートページキャッシュ

分散共有メモリを実現するため，共有領域ごとにページキャッシュに類似したカーネルオブジェクトを作成した (図 3)．このオブジェクトはファイルに対応して各ノードに存在し，共有領域のページがどの状態にあるか，ページの所有者といった情報をページごとに保持する．

このリモートページキャッシュはグローバルなハッシュ表とファイルのページキャッシュの両方から辿れるようになっており，ページフォルトハンドラはページキャッシュからリモートページキャッシュへのポインタを得て分散共有メモリに関するページフォルトを処理する．一方リモートページキャッシュにはクラスタ内で一意な ID が振られている．他のノードにページを要求する際には，この ID と，領域先頭からのオフセットを組にしてページを指定する．

3.2.3 分散共有メモリの有効化

プロセスの移送を行う際メモリ領域に共有であることを示すフラグが立っていれば，その領域はプロセス間で共有されているので，分散共有メモリを用いる必要がある．その領域に対して初めて分散共有メモリが必要とされた際に必要なデータ構造が準備され，ページキャッシュと関連づけられる．このとき，その領域を表す ID が新しく生成され，プロセスの移送先に通知される．もしその領域が既にマイグレーションを経験しており，ID が振られている場合には，その ID が通知される．通常のメモリ領域と違い，共有領域のページはマイグレーション時点では送信されない．移送されたプロセスが実行再開した時点では，共有領域の全ページは無効化された状態になっており，以後その領域にアクセスがあると分散共有メモリの機能を利用してページを元のノードに要求することになる．

このように現状の実装では，もともと同じノードに存在しておりプロセスマイグレーションを経て分散したプロセス同士の間でしかメモリ領域を共有できないが，この問題はクラスタ全体に渡るグローバルな名前解決の仕組みを用意することで解決できる．すなわち，

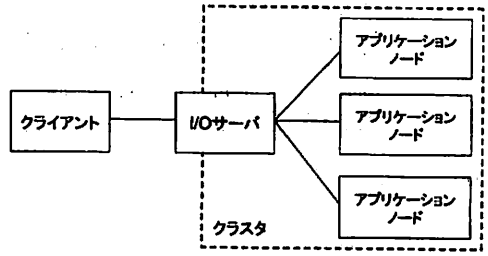

図 4 実験環境の概略

ファイル名等何らかのグローバルな識別子に対して，それに対応するメモリ領域を指す ID を返すような機構が必要である．ID がわかれば，あとはそれを用いて別のノードにページ要求を出すことができる．

4. 評　　価

予備実装に対する性能評価を以下のように行った．

4.1 実験環境

実験は図 4 に示すような環境で行った．クラスタ内の 1 台のノードが I/O サーバーと呼ばれ，クラスタ内部と外部の間の通信を行う．クラスタ内の他のノードはアプリケーションノードと呼ばれ，アプリケーションノード上のプロセスがソケットを開くと，SAPS によってそのソケットを通じた通信は自動的に I/O サーバーを経由して行われる．よってクラスタ外からは I/O サーバーのみが通信を行っているように見え，実際に通信を行っている相手のプロセスがどのアプリケーションノードで動作しているかを気にすることなく通信を行うことができる．

I/O サーバの背後に複数台のアプリケーションノードを配置し，クラスタ外部にあるクライアントから I/O サーバに対して要求を出す．それぞれの計算機の構成を表 2 に示す．クラスタ内のノード間の通信に，SAPS は Myrinet を用い，プロセスマイグレーションと分散共有メモリは Ethernet 上の TCP/IP を用いる．

4.2 基本性能の評価

まずプロセスマイグレーションに関する基本的な評価として，ソケットや共有メモリを用いないプログラムを移送するのに要するコストを測定した．用いたプログラムは以下の 2 つである．1 つは一定時間ごとに整数値を表示するプログラムである (図 5)．このプログラムは C 言語等で記述された一般的なプログラムの最小のサイズに近く，プロセスマイグレーションに要する最小のコストを見積もることができると考えられる．もう 1 つは約 100MB の書き込み済みヒープ領域を持つプログラムである (図 6)．このプログラムでは，使用メモリサイズとマイグレーションのコストの関係を見る．これらのプログラムをアプリケーション

表 2 本実験に使用した計算機

役割	外部クライアント	I/O サーバ	アプリケーションノード
CPU	Intel Xeon 3.0GHz	AMD Opteron 248 (2.2GHz)× 2	Intel Xeon 2.8GHz × 2
メモリ	3GB	6GB	2GB
Ethernet カード	Intel Pro/1000 Server	Broadcom BCM5703X GE	Intel Pro/1000 Server
Myrinet カード	-	Myrinet XP	Myrinet XP

```
int i = 0;
for (;;) {
  printf(''%d\n'', i++);
  sleep(1);
}
```

図 5 テストプログラム 1: 数値を表示するプログラム

```
char *buff = malloc(100*1024*1024);
for (;;) {
  for (i=0; i<100*1024*1024; i++)
    buff[i]++;
}
```

図 6 テストプログラム 2: 100MB のメモリを使用するプログラム

ノードとなっている 2 台の計算機間で移動させ，要した時間を計測する．

マイグレーションの各部分に要する時間を細かく測定するために，図 7 に示すように時間を区切って測定を行った．

- t_0: SIGSTOP のハンドラの開始時点から移送先ホストに対する TCP コネクションの確立までに要する時間．移送元で測定する．
- t_1: TCP コネクションが確立した後，移送先ホストから確認応答があるまでの時間．移送元で測定する．
- t_2: 確認応答パケットの往復時間．移送先で測定する．
- t_3: レジスタ，プロセス情報，メモリを含む，全てのプロセスイメージを転送するのに要する時間．

このうち，確認応答パケットの片道分の時間は往復時間の半分であると見なして，合計 $t_0+t_1+t_2/2+t_3$ をプロセスマイグレーションに要する合計時間とした．この期間が，プロセスがマイグレーションによって応答不能となっている期間である．

2 つのテストプログラムの移送に要した時間を表 3 に示す．この結果からマイグレーションにかかる時間を決定しているのは t_3 であることがわかる．レジスタや開いているファイルといった情報は小さいため，所要時間を決定しているのは転送すべきページの数である．

4.3 Apache による評価

分散共有メモリを含めた総合的な評価を行うため，Apache HTTP サーバを用いた実験を行った．Apache は各ワーカープロセスの状態を収集するため，スコア

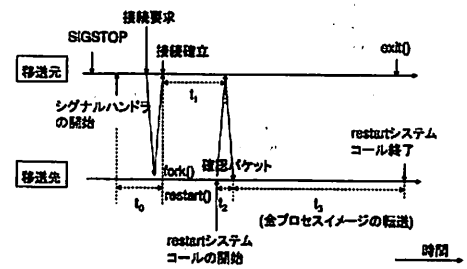

図 7 プロセスマイグレーションに要する時間測定区間

ボードと呼ばれる共有メモリ領域を使用している．このため，分散共有メモリなしで一部のワーカプロセスを他のノードに移した場合，そのプロセスの情報が取得できなくなってしまう．

この実験では，あるアプリケーションノードで起動した Apache のワーカプロセスを別のアプリケーションノードに移送し，分散して配置する．その上で Apache に付属するベンチマークプログラム ApacheBench を用いて外部のクライアントから I/O サーバにリクエストを出す．実験に用いた Apache は Apache 2.0.54 をそのままビルドしたものである．ワーカプロセス数は合計 20 程度で一定となるように設定した．この環境に対して，静的コンテンツおよび動的コンテンツを用いたベンチマークを行った．静的コンテンツを用いたベンチマークでは，Apache に付属する `index.html.en`(1465bytes) を要求する．一方動的コンテンツを用いたベンチマークでは，FreeStyleWiki[1] 3.6.2 を用い，標準添付されているヘルプのページを要求するということを行った．これは Perl で書かれた CGI であり，ディスク上のファイルを基に動的なコンテンツを生成する．これらのテストに対し，同時接続数 10，合計リクエスト数 1000 でベンチマークを行った．

実験の結果を図 8 に示す．図 8 は，単位時間あたりのリクエスト処理数を，1 ノードの場合の値を 1 としてグラフ化したものである．これからわかるように，静的コンテンツを用いたベンチマークでは，分散配置した Apache はノード数が増加するほど性能が悪化している．これは分散共有メモリの一貫性維持のためのオーバーヘッドによるものと考えられる．Apache の各ワーカプロセスは自分の状態が変化するたびにスコアボードに書き込みを行う．このため，並行して Apache 自体に高い負荷をかけるとページへの書き込み要求が各ノードで多発し，性能が大きく低下する．

表 3 マイグレーションに要した時間

プログラム	転送ページサイズ (KB)	$t_0(\mu s)$	$t_1(\mu s)$	$t_2(\mu s)$	$t_3(\mu s)$	合計 $(t_0+t_1+t_2/2+t_3)(\mu s)$
1	72	220.89	537.0	199.67	1983.56	2841.3
2	102476	271.44	568.67	217.44	910630.33	911579

表 4 各ノードにおけるページ要求回数と遅延

ノード番号	要求	回数	最小 (μs)	最大 (μs)	平均 (μs)
0	読み込み	174	221	15819	1993
	書き込み	166	225	16709	1215
	無効化	194	100	397	240
1	読み込み	196	159	12293	1780
	書き込み	182	224	20861	1136
	無効化	207	98	822	248
2	読み込み	306	169	32003	1080
	書き込み	165	247	9547	1317
	無効化	162	95	458	286
3	読み込み	319	212	11226	846
	書き込み	158	315	7504	1011
	無効化	167	101	436	261

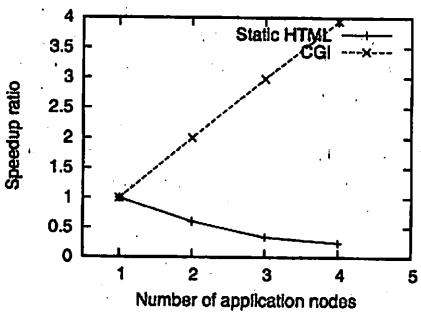

図 8 単位時間あたりリクエスト処理数の変化

また全ノード中で 1 つのページに書き込めるノードは同時にただ 1 つであるため，ノード数が増えるに従って性能が低下しているものと考えられる．

一方動的コンテンツを用いたベンチマークでは，ノード数の増加に対してほぼ比例する形で性能が向上している．これは，動的コンテンツを生成する際の子プロセスの生成や Perl インタプリタの実行にかかるコストが大きく，負荷を各ノードに分散することによる性能向上が分散共有メモリによるオーバーヘッドを上回るためであると考えられる．

分散共有メモリによるページ一貫性制御自体のオーバーヘッドを調べるため，1 回のベンチマークあたりに発生した共有メモリ領域におけるページフォルト回数と，ページ要求を出してから要求が満たされるまでの時間を計測した (表 4)．これは，4 ノードを使用した静的コンテンツに対するベンチマークにおける計測値である．表 4 から，ページの読み書きには平均して 1ms 強の時間がかかっている．これらは通信のためのオーバーヘッドとページを要求した先のノードにおける同期のコストとを加えた値である．たとえば，ページを要求したノードにもページがなく，そのノードがさらに別のノードにページを要求中だった場合，最初にページを要求したノードは長時間にわたり待たされることになる．一方ページの無効化はページに対する同期をせずに行われるため，ページの無効化要求に要している時間は純粋な通信のオーバーヘッドに近い．

5. 関連研究

プロセスマイグレーションの機構を UNIX 上に実装したものとしては，MOSIX[4] がある．MOSIX では，プロセスが移動した際，代理プロセス (deputy process) という特殊なプロセスが元のホストに残される．移動後のプロセスが移動先のホストでシステムコールを発行すると，そのシステムコールはカーネルによって移動元のホストへと転送される．転送されたシステムコールは待機している代理プロセスによって実行され，実行結果のステータス値がシステムコールを発行したホストへと返される．この機構によってカーネルの変更を少なく抑えつつ移送対象アプリケーションに対する制約を少なくできるが，一方でプロセスの移動後はシステムコールの発行のたびにリモートのホストに対する通信が発生するため，アプリケーションの種類によっては大きく性能を落とすことになる．また MOSIX は分散共有メモリを備えておらず，他プロセスとメモリを共有しているプロセスを移動することはできない．

プロセスマイグレーションは単一システムイメージを提供するシステムの一要素として実装されることが多い．近年開発が進められているシステムとしては，openMosix[11]，OpenSSI[12]，Kerrighed[10] がある．

openMosix は MOSIX の機能を Linux 上に実現したものであり，現在はオープンソースプロジェクトと

して開発が進められている．openMosix は MOSIX と同じ特徴を備えており，システムコール転送によるオーバーヘッドや，分散共有メモリを備えていないといった問題がある．

OpenSSI はオープンソースの要素技術を統合する形で開発が進められているプロジェクトであり，Linux クラスタ上にシングルシステムイメージを実現する．OpenSSI もまた分散共有メモリを持っておらず，共有メモリ領域をもつプロセスをノードをまたいで移動させることはできない．

Kerrighed はフランスの INRIA で開発が進められているシングルシステム Linux クラスタである．Kerrighed は分散共有メモリを持ち，プロセスの個々のスレッドが別々のノードで実行されることを可能にしているが，そうした機能を利用するためには専用のライブラリとリンクした実行ファイルを用意しなければならず，完全なバイナリ互換性を提供しているわけではない．

仮想化という観点から単一システムイメージを実現しているものとして，Virtual Multiprocessor[8] がある．Virtual Multiprocessor は分散共有メモリを用いて複数の計算機にまたがる仮想マシンを構築し，その上で動作する OS やアプリケーションにとってはあたかも SMP マシンの上で動作しているように見える．Virtual Multiprocessor はアプリケーションの互換性を保ちつつ分散環境で実行できるが，仮想化が全てユーザーモード上で実装されており，オーバーヘッドが大きい．

6. おわりに

本研究では、Linux 上にプロセスマイグレーション機構を実装し、クラスタ上でソケットを用いて通信を行っているプロセスや，他のプロセスと共有メモリを通して通信を行っているプロセスがノード間を自由に移動することを可能にした．予備実装による評価の結果，Apache のような実用的かつ大規模なプログラムをバイナリレベルの互換性を保ったまま移送でき，Apache による動的コンテンツの配信において，少なくとも 4 台までのクラスタ環境においてノード数に対し線形に性能を向上できることが確かめられた．

今後の課題としては，まずバイナリレベルの互換性を保ちつつ分散共有メモリのパフォーマンスを向上させるための工夫が必要である．また，クラスタ内でのカーネル間通信を前提とした高速かつ低オーバーヘッドな通信方式の利用が必要である．現在の実装では TCP/IP を利用してカーネル間通信を実現しているが，クラスタ内に限った通信では TCP/IP のような複雑なルーティングを考慮したプロトコルは不必要であり，余計なオーバーヘッドとなっていることが考えられる．また，クラスタ内では Ethernet に限らず Myrinet や Infiniband のような通信デバイスも用いられることがある．こうした様々なデバイスへの対応を考慮すると，通信デバイスに依存しないカーネル間通信のインターフェースを設計・実装する必要がある．

参考文献

1) FreeStyleWiki. http://fswiki.poi.jp/.
2) S.V. Adve and K.Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, Vol. 29, No. 12, pp. 66–76, 1996.
3) The Apache HTTP Server. http://httpd.apache.org/.
4) A. Barak and R. Wheeler. MOSIX: An Integrated Multiprocessor UNIX. In *Proceedings of the USENIX Winter*, 1989.
5) Milojicic D., Douglis F., Paindaveine Y., Wheeler R., and Zhou S. Process migration. *ACM Computing Surveys*, Vol. 32, No. 3, pp. 241–299, 2000.
6) William Gropp, Ewing Lusk, and Thomas Sterling. *Beowulf Cluster Computing With Linux*. MIT Press, 2003. ISBN 0262692929.
7) Duell J., Hargrove P., and Roman E. The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, 2003.
8) Kenji Kaneda, Yoshihiro Oyama, and Akinori Yonezawa. A virtual machine monitor for providing a single system image. In *Proceedings of the 17th IPSJ Computer System Symposium (ComSys2005)*, pp. 3–12, November 2005.
9) Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–359, 1989.
10) Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallee. Towards an efficient single system image cluster operating system. *Future Generation Computer System*, Vol. 20, No. 4, pp. 505–521, 2004.
11) openMosix. http://openmosix.sourceforge.net/.
12) OpenSSI. http://openssi.org/.
13) 松葉浩也，石川裕．シングル IP アドレスクラスタの設計．情報処理学会研究報告、2005-OS-100(SWoPP05), pp. 41–48, 2005.