2004－ARC－157　(5)
2004－HPC－ 97　(5)
2004／3／1

グリッド環境におけるVM上でのジョブ実行の検討

小倉章嗣[†1]　河野健二[†4]
松岡聡[†1,†2]　中田秀基[†3,†1]

グリッド上で利用される計算機において、ユーザの権限に応じた、適切で細粒度な各種内部リソースへのアクセス制御を行うために、プロセスの仮想化手法を用いる。近年複数提案されているが、個々の手法のコストは、対象となるアプリケーションの性質やアクセスポリシによって異なり、かつ制御を行う有効性自身も未解決である。そこで、アプリケーションの性質やポリシによって仮想化手法を選択し、常に低オーバヘッドな仮想化を実現する手法を提案し、かつそれぞれの手法の適用性を実際のグリッド上で選択されるクラスタノード上の種々のベンチマークを通じて検証する。作成したプロトタイプは Globus Toolkit2.4 を用いてユーザとポリシに基づいて仮想化手法を選択し、ジョブマネージャの一つとして仮想化環境内でジョブを起動する。NPB2.4 ベンチマークによる結果では、適切な仮想化手法を用いることでオーバーヘッドを最小にすることが可能で、かつ通信を頻繁に行なうアプリケーションではライブラリコールの仮想化が最適であり、複数プロセスの仮想化にはカーネルモジュールを用いたシステムコールの仮想化が最適であるなどの詳細な指針を得た。

Examination of the job execution on VM in Grid Environment

Shoji Ogura ,[†1] Kenji Kouno ,[†4] Satoshi Matsuoka [†2,†1]
and Hidemoto Nakada[†3,†1]

Despite recent proposals for fine-grained resource control on Grid computing nodes using virtual machine technologies, the impact of the respective virtualization schemes, as well as feasibility of actually imposing control, has not been well investigated in a comprehensive fashion. We propose a virtual machine framework for the Grid that allows selection of different virtualization schemes depending on application characteristics, and (2) perform comprehensive measurements of the impact of individual schemes, as well as when the schemes are actually used for resource control, and derive a guideline that would lead to (semi-) automated selection of virtualization schemes. The created prototype runs as a job manager in Globus 2.4, and allows selection of virtualization schemes, as well as pluggable resource control depending on the user and the intended policy. Benchmarks using NPB2.4 show that we can minimize the overhead by appropriate selection of virtualization schemes, as well as deriving several guidelines such as communication-intensive applications favor virtualization via library call interpositions, whereas virtualization of multiple process tend to favor kernel modules.

1. はじめに

グリッドとは、OS、アーキテクチャが異なり、複数の管理ドメインで管理されているリソース (計算リソース、ストレージリソース、実験装置など) を、複数の動的に構成される仮想組織 (VO) で安全に共有するための技術である[1)]。グリッドを構成するために必要な認証などの基盤技術をまとめたグリッドミドルウェアが多数開発され[2),3)]、グリッドシステムを構築することが容易になっている。

グリッドの効率的で安全な運用には、実行するジョブに対して、資源アクセスの質と量を細粒度で制御できることが望ましい。たとえば、ディスクアクセスの帯域の保障や、アクセスできるファイルシステムの制限などが必要となる。しかし、現在のグリッドミドルウェアのジョブ権限制御は、オペレーティングシステムの提供するユーザ単位の権限制御を直接用いた粗粒度のものであり、機能的に不足している。

この問題をプロセスの仮想化技術を用いてユーザレ

†1 東京工業大学
Tokyo Institute of Technology
†2 国立情報学研究所
National Institute of Information
†3 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
†4 電気通信大学情報工学科
Department of Computer Science University of Electro-Communications

ベルで解決する方法が提案されている☆。仮想化技術を用いると、競合するジョブのディスクアクセスの帯域を制限することで、他のジョブに帯域保障をしたり、ジョブに対して仮想ファイルシステムを見せることで、ファイルシステムへのアクセスを制限をすることができる。

プロセスの仮想化手法としてはいくつかの方法が提案されているが、個々の手法のコストは、対象となるアプリケーションの性質やアクセスポリシによって異なる。我々は、アプリケーションの性質やポリシによって仮想化手法を選択し、常に低オーバヘッドな仮想化を実現する手法を提案する。

このために、Globus Toolkit 2.4 を用いて ユーザ名とポリシに基づいて仮想化手法を選択し、仮想環境内でジョブを起動するシステムを試作した。さらに、アプリケーションに応じて最適な仮想化手法を選択するための基礎データとして、ptrace, mod_janus, DyninstAPI の三つの仮想化手法を用いて仮想ファイルシステムの提供と、ネットワークのバンド幅制限を実装し、いくつかのアプリケーションに対してこれらの仮想化手法のコスト評価を行なった。

2. プロセス仮想化手法

プロセスの仮想化はプロセスに対して仮想的な OS 環境を見せる技術で、サンドボックス、透過的なリモート I/O、透過的なプログラムの分散化、プロセスマイグレーションを可能にする。仮想化手法には大別して下記の 3 つの手法が存在する。

OS の機能による仮想化 OS の機能によって仮想化を行なう。例えば chroot や BSD の jail によってプロセス毎の仮想ファイルシステムを構築できる。低オーバヘッドで動作するが、単一の機能しか提供しない。

システムコールレベルの仮想化 プロセスのシステムコールをトラップし、引数の置き換え、メモリ領域の書き換えにより仮想化を行なう。ptrace や/proc ファイルシテムを利用してトラップする方法、カーネルモジュールによりトラップする方法、カーネルモジュール内部でトラップする方法がある。完全なアクセス保護を行なうことができる。

ライブラリコールレベルの仮想化 共有ライブラリの置き換え、静的なリンクによりアプリケーションの発行するライブラリコールを、仮想的なライブラリコールに置き換え、仮想化を行なう。ライブラリを使わないプログラムには適用できない。

☆ プロセスの仮想化を行なうプログラムは通常 VE(Virtual Environment) と呼ばれるが、本研究ではこれを VM(Virtual Machine) と呼ぶこととする

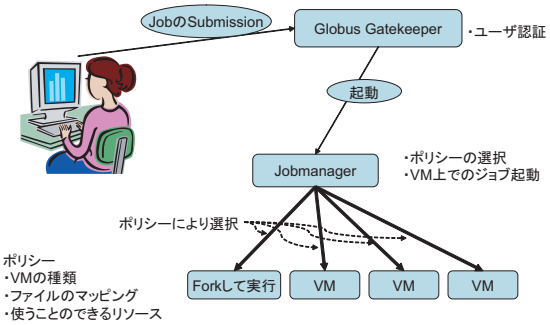

図 1 プロトタイプシステムの概要図

```
type:
1(ptrace)
file_path:
/usr/lib /usr/lib 0
/etc/passwd /etc/hosts 0
io_limit:
disk 30
net 30
etc:
trace_fork 1
```

図 2 ポリシファイルの記述例

それぞれの手法は、適用できるプログラム、用途、仮想化のコストの点で異なる。そのため、本研究では最適な仮想化手法を選択することで、低オーバヘッドでポリシに合ったジョブ実行を目指す。

3. 仮想化手法を選択するプロトタイプシステム

ユーザ、ポリシに応じて仮想化手法を選択するプロトタイプシステムを、標準的なグリッドミドルウェア Globus Toolkit 2.4[2] を用いて設計、実装した。図 1 はプロトタイプシステムの概要図である。ユーザは Gatekeeper に対してジョブを投入し、Gatekeeper はユーザ権限で JobManager を起動する。JobManager はグリッドユーザ ID を特定し、そのユーザのポリシファイルを取得する。その後、JobManager は VM を起動し、ポリシを VM に渡す。VM はジョブを起動し、ポリシに従ってアクセス制限やリソースの使用量制限を行なう。

図 2 にポリシファイルの記述例を示す。type:セクションで VM のタイプを指定する。file_path:セクションでファイルパスのマッピングを記述する。io_limit:セクションでバンド幅制限の値を記述し、etc:で VM のその他オプションを記述する。

プロトタイプシステムで仮想化手法を選択するためには、ユーザ毎のポリシファイルに記述するか、も

しくはユーザが仮想化手法を指定し、自分のジョブに合った仮想化手法を選択する。JobManager が最適な仮想化手法を選択するためには、ジョブの性質、ポリシによってどの仮想化手法が適しているのかを把握する必要がある。以下では複数の仮想化手法の実装を通してこれを探っていく。

4. 仮想化機能の実装

ptrace、mod_janus、DyninstAPI を用いて VM を実装し、仮想化コストを評価する。各手法の特徴を挙げる。

ptrace ptrace は UNIX の標準的なシステムコールであり、gdb などのデバッグシステムがプロセスをトレースするために使われる。ptrace の仮想化機能として、プロセスの発行するシステムコールの発行時/戻り時のトラップ、プロセスのレジスタの書き換え、プロセスの任意のメモリ領域の書き換え、を行なうことができる。全てのシステムコールをトラップしてしまい、また 1 ワードずつしかメモリ領域を書き換えられないため性能的に問題がある。

mod_janus mod_janus はサンドボックスシステム janus[4] で使われている Linux 用のカーネルモジュールである。mod_janus は ptrace と同じようにプロセスが他プロセスをトレースする機能を提供している。ptrace での問題点を改善するために作成された。機能的には、システムコールの発行時/戻り時の選択的なトラップ、レジスタに格納されたポインタが指すメモリ領域の書き換え、プロセスのカレントディレクトリの変更ができる。

DyninstAPI DyninstAPI[5] は、性能計測ツール Paradyn[6] で用いられている、関数単位でのバイナリ変換を行なう API、ライブラリである。DyninstAPI では、ライブラリコールの前後へのコード挿入、ライブラリコールの選択的な置き換え、などが行なえる。

4.1 仮想ファイルシステム

仮想ファイルシステムはファイルパスの置き換えによって実現する。ファイルのアクセス拒否を行なう場合には、ユーザ権限でアクセスが拒否されるファイルパスへ置き換える。ファイルパスの置き換えは、ファイル/ディレクトリアクセスを行なうシステムコール/ライブラリコールをトラップ、または置き換え、ファイルパスの書かれているメモリ領域を書き換えることで実現する。システムコール/ライブラリコールの終了時にはメモリ領域を復元する。

4.2 ディスク/ネットワークバンド幅制限

open/socket などのシステムコール/ライブラリコールをトラップしてファイルディスクリプタの情報を保持し、read/write の際にディスク/ネットワークアクセスかを判断する。データ量 (Data Size) の履歴を保存しておき、read/write 時に制限値に合うまで暫くプロセスを止める。

$$sleeptime = \frac{\sum_{T_i - T_{now} < t} DataSize_i}{LimitValue} - t$$

上記のように sleep time を算出し、システムコールの後に VM はしばらく sleep し、アプリケーションをその時間だけ再実行させないことで、バンド幅制限を実現する。

大きなバッファサイズを要求された場合には転送に偏りが出てしまうためバッファサイズを小さくする必要があるが、プロトタイプシステムでは行なっていない。

4.3 複数プロセスへの対応

Linux では fork したプロセスをすぐに ptrace でトレースすることが難しい。そのため、ptrace を用いた VM では fork システムコールを Linux 独自の clone システムコールに置き換える。トレースを行なう CLONE_PTRACE フラグを設定した状態で子プロセスを起動することで、プロセス起動時からのシステムコールトラップが可能になる。ptrace でプロセスをトレースすると、一時的に VM プロセスがトレースされるプロセスの親になる。そのため、プロセスが wait で子プロセスを待つ場合や親のプロセス ID を得る場合には、VM 側で親子関係を把握し、実際の動作をエミュレートする必要がある。

mod_janus ではプロセスが fork をした場合、子プロセスは親プロセスの fork が許可されるまで実行を開始しないことを保証する。またプロセスの親子関係を崩すこともない。そのため、ptrace のような問題は発生しない。

DyninstAPI では fork 時のコールバック関数を用意しており、それを用いて親プロセス、子プロセスの実行を再開させる。

4.4 MPI ジョブへの対応

MPI の実装として MPICH[7] を用いる。MPI ジョブに対応するためには、リモートプロセスを VM 上で起動する必要がある。リモートプロセスを VM 上で起動するには、デフォルトシェルを VM に設定する、execve システムコールの引数を検査して rsh の引数を置き換える、MPI ライブラリ自体を変更し rsh の引数を置き換えるなどの方法が考えられた。プロトタイプシステムでは実装の容易さから、MPI ライブラリの変更で MPI ジョブへの対応を行なった。

5. 仮想化コストの評価および考察

ジョブの性質、ポリシによって最適な仮想化手法を選ぶために、仮想化機能の評価及び実アプリケーションを用いた評価を通して仮想化コストを把握し、仮想

表 1　評価を行なった PC のスペック

CPU	AthlonXP 1300Mhz
メモリ	896MB
OS	Linux kernel-2.4.20
gcc	2.95.4

表 2　PrestoIII クラスタノードのスペック

CPU	AthlonMP 1900+ x2
メモリ	768MB DDR
NIC	100Base-T
OS	Linux kernel-2.4.19
gcc	2.95.4

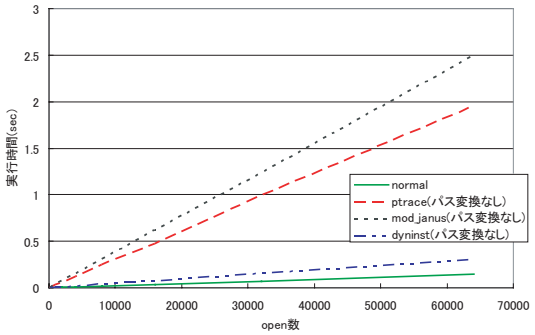

図 3　open システムコールをトラップした際のオーバヘッド

化手法を選択するための指標を同定する。

評価は表 1 に示すスペックを持った PC で行ない、NPB の評価は PrestoIII クラスタ (表 2) を用いて行なった。

5.1　仮想化機能の評価

システムコールトラップのコスト、仮想ファイルシステム実現のコスト、バンド幅制限の精度と上限、複数プロセスの仮想化コストを評価する。

5.1.1　システムコールトラップのコスト評価

図 3 はファイルの open/close を複数回繰り返し実行するプログラムを作成し、これをトラップせずに実行した際の実行時間、トラップのみ行ない引数の変更は行なわない場合の実行時間を示したものである。

これはシステムコールのトラップにかかるコストを表わしており、DyninstAPI による仮想化コストが最も低く、次に ptrace、mod_janus の順でコストが高くなっている。アプリケーションが発行するほとんどのシステムコールをトラップする必要がある場合 (ネットワーク通信を頻繁に行なう場合など) では、DyninstAPI による仮想化コストが最も低いと予想される。

5.1.2　仮想ファイルシステム実現のコスト評価

図 4 は、引数書き換えによる性能低下を見るために、open/close を 10 万回繰り返し、書き換える引数の文字数を変化させて実行時間を計測したものである。

ptrace では、1 ワードずつしかメモリ領域の書き換えができないため、書き換える文字数が多くなるとその書き換えのコストも増大している。そのため、引数

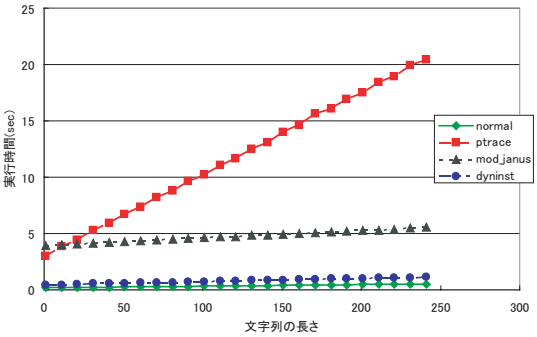

図 4　引数書き換えのオーバヘッド

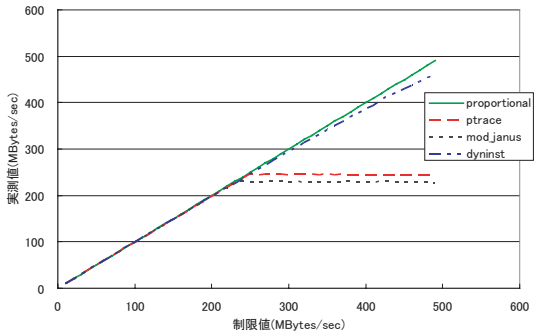

図 5　バンド幅制限の精度と上限

の書き換えが頻繁に行なわれる状況では、ptrace を用いるのは適当ではなく、DyninstAPI を用いるのが最適である。

5.1.3　バンド幅制限の精度と上限の評価

スループット計測ツール netperf[8] を用いて転送速度を計測し、バンド幅制限の精度を確認した。netperf は 16384 バイトのデータを 10 秒間にわたって送信し、スループットを算出する。異なるノードに送受信者を置き、100Mbps のイーサネットでの評価はどの仮想化手法でも 1%以内の誤差であった。低いバンド幅であれば誤差が小さく、仮想化を行なっていてもスループットの上限に違いはない。

次に同一ノードに送受信者を置いた場合に、各仮想化手法でバンド幅制限を行なったスループットを図 5 に示す。仮想化を行なわない場合のスループットは 3620.67Mbps(45250KBytes/sec) であった。評価結果から、制限値が大きくなると誤差が大きくなることがわかった。また、mod_janus では 230MBps、ptrace では 250MBps が上限であった。通信の頻度が上がるとトラップのコストが相対的に大きくなるためか、CPU の負担が大きく十分な仮想化が行なえないためだと考えられる。高速なネットワークを利用したジョブでは、システムコールの仮想化よりもライブラリコールの仮想化が有効である。

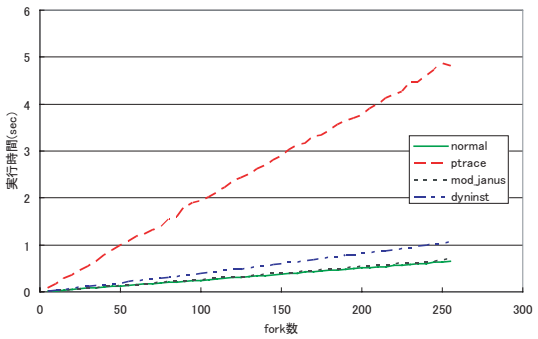

図 6 複数プロセス仮想化のコスト

	BT	CG	EP	IS	LU	MG	SP
N	539	119	26.5	4.33	810	252	297
P	532	114	26.4	4.03	649	226	293
J	537	120	26.4	4.13	811	252	297
D	539	119	26.4	4.11	809	251	297

表 3 NPB の実行性能 (Mflops)
N:normal, P:ptrace, J:mod_janus, D:dyninst

5.1.4 複数プロセスの仮想化コスト評価

複数のプロセスの同時仮想化コストを評価した。複数のプロセスを fork し、それぞれのプロセスがファイルの open/close を繰り返すプログラムを作成し、fork 数を変化させて実行時間を計測した。ptrace では、fork 仮想化のコストはほとんどかからないが、複数のプロセスを同時に仮想化した場合にオーバヘッドが大きかった。この原因としては、VM で親プロセスの wait システムコールをエミュレートするコストが高いのか、もしくは VM が wait で待つコストが考えられる。全体で考えると、ptrace が最もコストが大きく、ついで DyninstAPI、mod_janus の順であった。

5.2 実アプリケーションでの評価

並列ベンチマークである NAS Parallel Benchmarks(NPB)2.4[9] を用いて仮想化のコスト評価を行なった。評価結果を表 3 に示す。ベンチマークは BT、CG、EP、IS、LU、MG、SP で、クラス W を用いた。バンド幅制限を行なわない場合、BT、EP、IS、SP ではどの仮想化においてもほとんど性能低下が見られなかった。それらのベンチマークは実行中に通信を行わないためと考えられる。

実行中に通信を頻繁に行なう CG、LU、MG では ptrace を用いた仮想化で性能低下が見られ、LU の 4 ノード時には 20%の性能低下が見られた。これは、ptrace では全てのシステムコールをトラップしてしまうために、send/recv 毎にオーバヘッドがかかり、通信遅延が増大したためだと考えられる。一方、mod_janus、DyninstAPI では send/recv をトラップしなかったために性能低下が見られなかったと考えられる。以上の実験により通信が行なわれない CPU インテンシブなアプリケーションでは仮想化による性能低下が見られないが、ネットワークインテンシブなアプリケーションにおいては性能に影響を与えることを確認した。

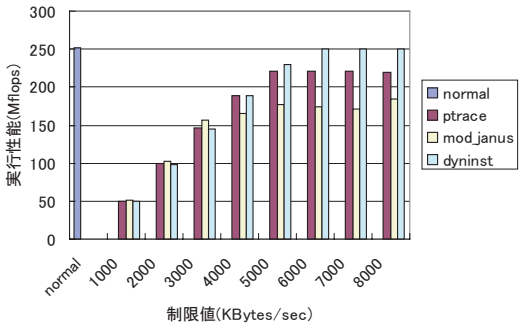

図 7 4 プロセスでバンド幅制限を行なった状態での MG の実行性能

	NRP	CV	FSI	MP
ptrace	○	○	×	×
mod_janus	×	○	△	○
DyninstAPI	○	×	○	△

表 4 各仮想化手法の指標、NRP(No Root Priviledge), CV(Complete Virtualization), FSI(Frequently Systemcalls Issued), MP(Multi Processes)

図 7 は 4 ノード 4 プロセスで NPB MG を実行し、バンド幅の制限値を変化させて計測を行なった結果である。DyninstAPI では 6000(KBps) 以上の制限値において仮想化を行なわない場合と同様の性能が出ているが、ptrace、mod_janus では実行性能に影響が出ている。これは、ptrace、mod_janus ではトラップのコストが大きいために通信に遅延が生じ、性能が落ちたものと考えられる。

5.3 仮想化手法選択のための指標

上記の考察から、グリッドジョブに対して仮想化手法を選択するために次のような指標が考えられる。表 4 に各仮想化手法の指標毎の特徴を示す。

管理者権限を必要としない (NRP) mod_janus による仮想化では、カーネルモジュールのロード時に管理者権限が必要となる。管理者権限が使えない場合には mod_janus による仮想化も行なえない。

完全な仮想化が必要 (CV) DyninstAPI ではシステムコールを直接呼び出すことで仮想化をバイパスすることができる。ユーザが信頼がおけない場合など完全な仮想化が必要な場合では、DyninstAPI を用いることは避けるべきである。

システムコールが頻繁に発行される (FSI) アプリケーションがシステムコールを頻繁に発行する場合には、DyninstAPI でオーバヘッドが小さく、ptrace、mod_janus と続く。

同時に起動するプロセス数が多い (MP) 同時に仮

想化するプロセス数が多い場合、mod_janus が最も性能がよく、DyninstAPI、ptrace と続く。

以上のポリシ、アプリケーションの性質による指標を元に、ジョブに応じた最適な仮想化手法を選択する。例えば、通信が頻繁に行なわれる並列ジョブで完全な仮想化が必要ない場合には、DyninstAPI による仮想化を行なう。

6. 関連研究

プロセス仮想化技術を利用して、グリッドに付加的な機能を提供しているシステムがある。

グリッドシステムのためのポータルを提供する PUNCH では、アクセス制限を行なうシェルと ptrace によってアクセス保護を実現し、十分に信頼できないユーザのジョブからリソースを保護するシステムを提供している[10]。ptrace によるアクセス保護だけでは性能が出ないために、機能を制限したシェルにより、シェルスクリプトで記述されたジョブは高速に実行できるとしている。このシステムはシェルスクリプトとシステムコールの仮想化を使い分けており、複数の仮想化手法を用いている点は本研究と同じである。しかし、シェルスクリプトでは適用できるアプリケーションが大きく限られる。本研究ではライブラリコールとシステムコールの仮想化手法を使い分けることで、PUNCH の研究よりも多くのアプリケーションをカバーできると考える。

Entropia は、ユーザのデスクトップなどの遊休リソースを有効に活用するデスクトップグリッドを構築するためのグリッドミドルウェアである。アプリケーション、リソースの保護、透過的なプログラムの分散化をバイナリ変換を行なうサンドボックスによって提供している[11]。デスクトップユーザのジョブを阻害しないために CPU 使用率の制限は行なうが、ディスク/ネットワークのバンド幅制限は行なわない。

7. おわりに

グリッド上で利用される計算機において、プロセスの仮想化手法を用いることで、ユーザの権限に応じた、適切で細粒度な各種内部リソースへのアクセス制御を行うことを目的とし、Globus Toolkit 2.4 を用いて ユーザとポリシに基づいて仮想化手法を選択し、仮想化環境内でジョブを起動するシステムを試作した。また、最適な仮想化手法を選択するために、仮想化手法のコスト評価を行ない、仮想化手法を選択するための指標を得た。通信を頻繁に行なうアプリケーションではライブラリコールの仮想化が最適であり、複数プロセスの仮想化にはカーネルモジュールを用いたシステムコールの仮想化が最適であった。

今後の課題としては、VM の拡張が挙げられる。仮想ファイルシステムでは fstat などで得られるファイル属性を仮想環境のものに変更し、バンド幅制限では、mmap による非明示的なディスクアクセスに対応する必要がある。また、CPU 使用率の制限、階層的なポリシ規定、動的なポリシ変更に対応する。さらに仮想化 API の共通化、他の仮想化手法の実装、ジョブマイグレーションなどの拡張を行なう。評価を重ね、仮想化手法選択のための指標を確かなものにする。

また、VM を支援する高レベルのサービスの実現も課題の一つである。サービスとして、バンド幅使用量予約を考慮したスケジューラ、VO 管理者による管理機構が挙げられる。

謝辞 本研究の一部は文部科学省「経済活性化のための重点技術開発プロジェクト」の一環として実施されている超高速コンピュータ網形成プロジェクト (NAREGI : National Research Grid Initiative) による。

参考文献

1) Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid. *International J. Supercomputer Applications*, 2001.
2) globus homepage. http://www.globus.org/.
3) Dietmar W. Erwin and David F. Snelling. Unicore - a grid computing environment. *Euro-Par 2001*, 2001.
4) T. Garfinkel and D.Wagner. Janus: A practical tool for application sandboxing. http://www.cs.berkeley.edu/~daw/janus.
5) Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. *To Appear in Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.
6) Paradyn. http://www.cs.wisc.edu/~paradyn/.
7) Mpich. http://www-unix.mcs.anl.gov/mpi/.
8) netperf. http://www.netperf.org/.
9) Nasa arc. http://www.arc.nasa.gov/.
10) Ali Raza Butt, Sumalatha Adabala, Nirav H. Kapadia, Renato Figueiredo, and Jose A. B. Fortes. Fine-grain access control for securing shared resources in computational grids. *In The Proceedings of International Parallel and Distributed Processing Symposium (IPDPS) 02*, April, 2002.
11) Andrew A. Chien, Brad Calder, and Stephen Elbert. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel Distributed Computing, 2003*, 2003.