Vol.2016-OS-136 No.15
2016/3/1

# メモリ消費のふるまいに基づくメインメモリ管理手法

中川 岳[1,a)]　川田 裕貴[1]　追川 修一[2]

**概要**：プログラムの実行単位であるプロセスは，CPU 時間，メインメモリといった種々のリソースを消費する．その中でも，利用可能なメインメモリ量はシステムの安定性に与える影響が大きい．そのため，ユーザは大量にメインメモリを使用するプロセスを把握し，必要に応じてリソース制限を行うなどして，利用可能なメインメモリを一定以上に保つことが望ましい．しかしながら，プロセスはときにユーザの意図しない，大量かつ急速なメモリ消費を行う．既存のオペレーティングシステム（OS）のリソース管理機構では，このような大量のメモリ消費によるシステムの不安定化を未然に防ぐことは難しい．システムでメモリリソースの枯渇が起こった場合に，動作中のプロセスを停止することによって，利用可能なメモリ量を確保するメモリ管理戦略も存在する．しかしながら，この戦略には，不安定化の原因ではないプロセスを誤って停止する問題もある．そこで本発表では，プロセスのメモリ消費のふるまいに基づいた，プロセスのリソース管理を提案する．この提案手法では，プロセスのメインメモリ消費の速度に着目し，問題のあるメモリ消費の予兆を検出する．問題のあるメモリ利用を行ったプロセスについては，スケジューリングから一時除外し，それ以上のシステムの不安定化を防止する．この手法を用いれば，これまで対処が難しかった，ユーザが意図しない，大量かつ急速なメモリ消費を検出することができる．また，その問題のあるメモリ消費に伴うシステムの不安定化を未然に防止することができる．本発表では，その提案手法の設計について議論する．

**キーワード**：オペレーティングシステム，メモリ管理

# A Memory Resource Management Method Based on Memory Consuming Rate

***Keywords:*** Operating System, Memory Management

## 1. はじめに

プログラムの実行単位であるプロセスは，CPU 時間，メインメモリ，ストレージといった，種々のリソースを消費する．その中でも，利用可能なメインメモリ量はシステムの安定性に大きな影響を与える．例えば，利用可能なメインメモリが不足すると，スラッシング [1] が発生し，システム性能や応答性の低下を招く．オペレーティングシステム（OS）が，ユーザプロセスを強制的に終了して利用可能なメインメモリを確保することもある．これらの例のように，利用可能なメインメモリが不足するとシステムは不安定化する．

プロセスは時に，ユーザが意図しない，大量かつ急速なメモリ消費を行いシステムを不安定化する．通常の場合，その大量のメモリ消費はあらかじめ予見されている．つまり，ユーザによって大量にメモリを消費するプログラムが把握されている．しかしながら，その予見に反して，大量のメモリを急速に消費するプログラムも存在する．例えば，メモリリークのようなメモリ管理に関する瑕疵を含んでいるプログラムのプロセスは，ユーザの意図に反して大量のメモリを急速に消費することがある．コンピュータウイルス，ワームといったマルウェアもシステム安定性を阻害す

[1] 筑波大学大学院 システム情報工学研究科 コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305–0006, Japan
[2] 筑波大学 システム情報系 情報工学域
Division of Information Engineering, Faculty of Engineering, Information and Systems
1-1-1 Tennodai, Tsukuba, Ibaraki 305–0006, Japan
[a)] gnakagaw@cs.tsukuba.ac.jp

るために，大量のメモリ消費を行う可能性がある．ここに挙げた例のように，ユーザが意図しない，大量かつ急速なメモリ消費によりシステムが不安定化する可能性がある．

既存の OS の機能では，このような問題のあるメモリ消費によるシステム安定性の低下を十分に防ぐことができない．OS に一般的に備わっているリソース管理機構を用いれば，プロセス単位，ユーザ単位，グループ単位でリソース利用の制限を行うことができる．しかしながら，この方法では，ユーザが意図しない，大量かつ急速なメモリ消費に対応することは難しい．なぜなら，この方法では制限の対象や内容を問題が発生する前に設定する必要があるためである．メモリ不足によるシステム停止を回避するために，ユーザプロセスを強制停止し，利用可能なメインメモリを確保する戦略もある．例えば，Linux に実装されている Out Of Memory Killer（OOM Killer）は，この戦略を採用している．この戦略により，問題の発生を未然に防ぐことはできずとも，メモリリソースの不足によるシステム停止を防ぐことが可能である．しかしながら，リソース不足の状況によっては，本来は動作を継続すべきプログラムを停止する問題がある．

本発表では，Linux を対象とした，ユーザ意図しない，大量かつ急速な大量のメモリ消費を防ぐための新たなメモリリソース管理手法を提案する．提案手法でメモリ管理を実施するシステムでは，すべてのプロセスは，そのメモリ利用量の増減の調査を受け，プロセスごとに，メモリ消費の速度を算出する．もし，システムにおいて急速かつ大容量のメモリ消費が発生し，システムがメモリ不足に陥る可能性がある場合には，その原因と推定されるプロセスの実行を一時凍結する．この判断には，前述したプロセスごとのメモリ消費の速度を用いる．これにより，急速なメモリ消費によってシステムが不安定化することを未然に防ぐことができる．また，実行を一時凍結された場合でも，適切な権限を持ったユーザの指示により，動作を再開することが可能である．そのため，OS による判断が誤っていた場合でも，プロセスの実行コンテキストは失われない．

本発表の構成は以下の通りである．まず 2 節では，本研究が対象とする，ユーザが意図しない，大量かつ急速なメモリ消費への対処という点から，既存のメインメモリ管理手法の問題点を整理する．3 節では，2 節での議論をふまえ，プロセスのメモリ消費のふるまいに基づいた，新しいメモリ管理方式を提案する．著者らは提案手法を Linux 3.14.0 に対して実装を試みた．4 節では，その実装について述べる．5 節では関連研究について述べる．6 節では本発表のまとめと，今後の展望について述べる．

## 2. 既存のメモリリソース管理手法の問題点

本研究が対象とする Linux は，利用者やプロセスごとにメモリ使用量を制限する機能を備えている．これらの機能を使うことで，本研究が問題としている，ユーザの意図していない，プロセスによる大量のメモリ割り当てによってシステム全体が不安定になることを防ぐことができる．しかしながら，これらの機構だけでは，利用者に柔軟なメモリを使用させながら，一方でシステムの安定化を達成することは難しい．ここでは，Linux に備わっているリソース制限機構について述べ，それぞれの課題について議論する．なおこれ以降，特段の断りがない限り，Linux における管理者権限（root 権限）を持つ利用者のことを管理者，それ以外の利用者のことを一般利用者と表す．

### 2.1 prlimit によるメモリ使用量制限

Linux には，プロセスごとに利用可能なリソースを制限する機能がある．この制限は prlimit システムコールによって設定が可能である．管理者は，内部的に prlimit システムコールを呼び出す，ulimit コマンドや prlimit コマンドを経由して，一般利用者ごとにそれぞれが起動したプロセスが利用可能なリソースを制限することが可能である．また，一般利用者も管理者が設定した範囲で，リソース制限を変更してプロセスを起動したり，自己に権限があるプロセスのリソース制限を変更することが可能である．制御できるリソースには，プロセスが利用可能な CPU 時間，同時にアクセスできるファイルの数，ユーザが起動できるプロセスの総数などがある．メモリに関する制限項目もあり，プロセスが利用可能な仮想メモリ空間の大きさ (RLIMIT_AS) や，利用可能なメインメモリの量 (RLIMIT_RSS) を設定することができる．

しかしながら，prlimit システムコールで実現できるリソース制限では，本論文が問題としている，意図しない大量のメモリ割り当てに対処するのは困難である．なぜなら，現行の Linux では，利用可能なメインメモリの量についての制限 (RLIMIT_RSS) は，特殊な場合を除いて無視されるためである [2]．そのため，RLIMIT_RSS に制限値を設けたとしてもプロセスが利用可能なメインメモリの量を制限することはできない．仮想メモリ空間の制限 (RLIMIT_AS) は有効であるが，この制限はスワップアウトされている領域や仮想アドレス空間に直接マッピングされたファイルや共有ライブラリの領域も制限するため，この制限を設けるのは現実的ではない．また，prlimit によるリソース制限だけでは，リソース消費の多いプロセスを検知して，制限を設定することはできない．prlimit で実現できるリソース制限には，リソース制限すべきプロセスを決定したり，その制限値を決定する機能は備わっていない．そのため，意図しない大量のメモリ割り当てに対処するためには，システム管理者や監視プログラムによって定期的に使用可能なリソース量を監視し，必要に応じてリソース制限を設定する必要がある．

### 2.2 cgroup によるメモリ使用量制限

cgroup は prlimit によるリソース制限に加えて新たに導入されたリソース制限機構である．cgroup では，オペレーティングシステムで動作するユーザータスクを cgroup 単位で管理する．この cgroup には，ある特定のリソースに関して，同じパラメータで管理されるタスクが登録される．この cgroup はそれぞれ管理対象のリソースごとに準備されたサブシステムに所属しており，そのサブシステムの機能により，リソースの制限が提供される．cgroup は木構造を構成することができ，ある cgroup に関して，部分的にパラメータが異なる cgroup を生成することができる．生成された cgroup は生成元となった cgroup と木構造で結ばれるため，管理が容易になる利点もある．cgroup の管理は cgroup file system を介して行われる．つまり，パラメータや所属するタスクについては，cgroup file system 上のファイルを読み書きすることで設定を行う．

cgroup には，memory subsystem が実装されている．この memory subsystem を利用することで，プロセスが利用可能なメモリに関する制限を行うことができる．具体的には，使用可能なメインメモリの量，スワップ領域を含めたプロセスが利用可能なメモリの量などが挙げられる．例えばあるプロセス A について，そのメインメモリ利用量を 512MB に制限する際には，利用可能なメインメモリを 512MB に制限した memory subsystem に所属する cgroup を生成し，この cgroup にプロセス A を参加させれば，制限をかけることができる．memory subsystem で設定可能なメモリ量の制限は，cgroup に所属するプロセスが利用しているメモリ使用量を合算した値に対して適用される．例えば，memory subsystem に所属している，cgroup$\alpha$ があるとする．そして，この cgroup$\alpha$ にはプロセス A, B が所属しているとする．cgroup$\alpha$ に関して，使用可能なメインメモリが 512MB に制限されているとき，プロセス A，B が利用可能なメインメモリの合計が 512MB に制限される．プロセス A，B にそれぞれについて独立したメモリに関する制限を行うには，それぞれ別の cgroup を構成する必要がある．

この cgroup を利用して，本論文が問題とする，意図しない大量のメモリ割り当てに対処する場合には解決すべき課題がある．それは cgroup 階層の作成と，どのプロセスがどの cgroup に所属すべきかが，静的に決定された設定に基づいて，プロセスの生成時に行われることである．cgroup が利用可能な環境では，cgroup の生成とプロセスの所属を管理するためのデーモンプログラムを動作させることができる．このデーモンは静的に設定された cgroup の階層を作成しておき，対象となるプロセスの生成を検知して，設定に基づいて cgroup にそのプロセスを所属させる．このプロセスの所属の決定は，プロセスの生成時に行われる．つまり，プロセス生成後のプロセスの挙動に対応して，cgroup の作成と制約の設定，プロセスの所属の変更を動的に行うことはできない．意図しない大量のメモリ割り当ては，プロセスの生成後に認識されるため，この制約は大きな問題である．

## 3. メモリ消費のふるまいに基づいたプロセスのリソース管理

2 節で議論したとおり，既存の OS でのメモリ管理手法では，本研究が対象とする，ユーザの意図しない，大量かつ急速なメモリ消費によりシステムの不安定化を防ぐことは難しい．そこで本節では，プロセスのリソース消費に関する特性に基づいた，プロセスにおけるメモリリソース管理の手法を提案する．

これまでの OS におけるリソース管理は，主にその利用量に着目したものであった．つまり，プロセス，ユーザなどの管理単位ごとに，「どのくらいの」メモリを消費するのかに着目したものであった．しかしながら，同じメモリの使用量でも，その状態に至るまでの，リソースの使用のありかたにははさまざまな形がある．この特性に基づくことで，本研究が対象とする，ユーザが意図しない，大量かつ急速なメモリ消費によりシステム不安定化を防ぐことが可能になる．つまり，本研究は「どのぐらい使うのか」に加えて「どのように使うのか」を考慮することで，より精度が高い，柔軟なリソース管理を提供することを目的とする．

本研究では，プロセスごとのメモリ消費の速度に着目して，急激に発生するメモリの大量消費の検出を行う．既存の手法では，ある時点で同じ量のメモリを使用しているプロセスが，次にどのようにメモリを消費するかを予測することは難しい．本手法では，システムが安定している状態でプロセスごとにメモリ消費の速度を算出し，それぞれのプロセスのメモリ消費の傾向を把握する．そのため，次にどのようにメモリを消費するか予測することができる．また，個別のプロセスについてだけではなく，システム全体のメモリ消費の傾向も把握しておく．安定した状態での，システム全体のメモリ消費の傾向がわかっていれば，急激なメモリ消費が起こった場合に，それを容易に検出することができる．また，そのときにプロセスごとのメモリアクセスの傾向を把握しておくことで，その問題のあるメモリ消費の原因となるプロセスを高い精度で特定することができる．

### 3.1 メモリ消費のふるまいの計測

提案手法を適用する OS では，動作する全てのプロセスについて，メモリ消費の速度を算出し，記録する．この速度の算出は，算出時点のメモリ使用量をそのプロセスの生存時間で除したものである．つまり，プロセスが新しいメモリを要求しつづければ，この速度は上昇し，要求が少な

ければ，この速度は小さくなる．

速度記録の処理は，プロセスがその実行権を失うタイミングで，リソース管理処理の一環として実施する．また，プロセスが新たなメモリ領域を取得する処理を検出して，その時点でも計算を行う．ただし，メモリ領域の取得処理をすべて検出，速度の計算処理を行うことは大きなオーバーヘッドを生むため，後者に関しては，検出対象のイベントのうち，一定間隔のイベントを対象とした，サンプリング方式で行う．

### 3.2 システムの全体のメモリ消費傾向

プロセスごとのメモリ消費速度と併せて，システム全体でのメモリ消費の傾向も算出，記録する．この傾向を用いれば，異常なメモリ負荷がかかった場合にも，それを検出することができる．

OS カーネルは一定周期ごとに，システム全体でのメモリ利用量を記録する．この計測は，循環バッファに一定周期ごとにデータを格納するのみで，消費傾向の算出は，必要となった時点で，はじめて行われる．その算出は，記録したデータを時系列に並べ，隣り合う記録からそれぞれのタイミングでの速度を算出する．算出した複数の速度について，移動平均を取る．

### 3.3 メモリ不足の検出

OS カーネルは，プロセスに物理メモリを割り当てるイベントをトリガーとして，システム全体の利用可能なメモリの量を調べる．もし，利用可能なメモリの量が少なくなっている場合は，3.2 節で説明した，対象システムにおけるメモリ利用の変化の記録から，直近のシステム全体でのメモリ割り当て速度を算出する．また，記録されているデータから速度を求めて，その移動平均を求める．もし，システム全体のメモリ消費の速度が求めた移動平均から乖離しているときは，OS カーネルは対処すべき，急速で大容量なメモリ消費が発生していると判断する．

もし，問題のあるメモリ割り当てを検出した場合は，システムで動作しているプロセスを走査し，もっとも大きなメモリ消費速度をもっているプロセスを特定する．特定されたプロセスは，問題のあるメモリ消費の原因とみなされ，その動作を一時的に停止される．

### 3.4 復帰可能な停止

前述の通り，システムの利用可能なメモリ量が少なくなり，またシステム全体のメモリ消費速度が大きいときは，プロセスのうち，最も大きなメモリ消費速度をもっているプロセスの実行が一時停止される．この一時停止は，プロセスの終了ではなく，タスクスケジューリングから一時的に除外することで実現される．ユーザが当該プロセスの続行を望む場合は，適切な権限をもちいることで ，その動作を再開されることができる．

## 4. 実装

3 節で述べた提案手法を Linux に実装することを試みた．実装の対象は Linux 3.14.0 である．

### 4.1 プロセスごとのメモリ消費速度の計測

プロセスごとのメモリ消費速度の算出は 2 つのタイミングで行う．1 つは，プロセスの実行権が奪われたときのコンテキストスイッチ処理時である．実行権を手放すタイミングで算出することで，その実行中のメモリ消費の変化を，正確に反映させることができる．

プロセッサの構成とプロセスの性質によっては，コンテキストスイッチが発生しにくいケースもある．そこでプロセスの実行中も，デマンドページングによるメモリの取得処理をトリガーにして，メモリ消費速度の算出を行う．ただし，すべてのデマンドページング処理をトリガとすると無視できないオーバーヘッドが生ずるため，デマンドページング処理が一定回数実施された時に，速度の算出を行うものとした．

プロセスごとに算出した消費速度の記録は，プロセス情報を管理する task_struct 構造体に，専用のデータ領域を追加することで実現した．

### 4.2 システム全体でのメモリ消費傾向の計測

当該処理を行う関数を，カーネルタイマを用いた定時タスクとして実装した．実行周期は 1 秒とした．この処理では，システム全体での利用可能なメモリの量を時系列で記録する．この利用可能なメモリの量には，通常の空きメモリ領域に加えて，ページキャッシュ，バッファキャッシュ領域も算入した．記録には循環バッファを用いており，記録から一定時間を経過した情報は新しいもので上書きされる．

カーネルタイマによる定時タスクでは，記録した情報を用いた速度の計算処理などは行わない．その処理については，必要に応じて非同期に行われる．

### 4.3 メモリ不足の検出

4.1 節で述べた，プロセスごとのメモリ消費速度と同様に，デマンドページングをトリガにして，システムがメモリ不足に陥っていないかを確認する．もしこのとき，システムの利用可能なメモリ領域が一定以下であった場合は，4.2 節で述べた処理で取得した情報を用いて，システム全体でのメモリ消費の傾向を調査する．もしこのとき，直近のメモリ消費速度が，移動平均を取る長期的な傾向から外れていた場合は，システムで急激なメモリ消費が起こっているものと判断される．

急激なメモリ消費が行われていると判断された場合は，

プロセスリストを走査して，最も大きなメモリ消費速度を持つプロセスを特定し，そのプロセスを一時停止状態にする．

### 4.4 プロセスの一時停止

プロセスの一時停止については，シグナルは用いない．なぜならば，対象プログラムがシグナルを無視するように設定されていたり，暴走していた場合には，確実に停止できないためだ．その代わりに Linux に実装されている cgroup のうち，freezer subsystem を利用する．この subsystem を用いる事で，STOP シグナルを無視するようなプロセスについても，確実に停止させることができる．

## 5. 関連研究

[3] はメインメモリが少ない，スマートフォン，タブレットにおけるメモリ管理の効率化を目的としたメモリ管理手法を提案する研究である．この研究で提案されている手法では，システムで動作するプログラムを端末の開発元が提供するものと，それ以外に分類し，それぞれに異なるメモリノードを割り当てて管理する．これにより，端末の開発元が提供しない，サードパーティのアプリケーションが，システムを構成する重要なプログラムのメモリ操作に影響を与えることを防止することが可能である．動作するプログラムを分類して，それぞれ異なるメモリ管理を行うという点では，プロセスのふるまいに基づいて，リソース管理を行う本研究と類似する．しかしながら本研究はアプリケーションの属性ではなく，実行時のプロセスの挙動を利用して，システムの安定性に影響を与えるプロセスのメモリ使用を制限する．この点で [3] の研究とは相違している．

メモリリークの検出は重要な研究対象であり，多数の先行研究がある [4–7]．Purify [4] はプログラムのコンパイル時に，メモリリークを検出する命令を実行バイナリに埋め込むことでメモリリーク検出を行う手法である．Sniper [5] は CPU のパフォーマンス監視ユニットの機能を応用したメモリアクセスのトレースをメモリリーク検出に応用したものである．SWAT [6] は対象プログラムのメモリアクセスを実行時にサンプリングし，メモリリークを検出するソフトウェアである．メモリリークによるシステム不安定化を防ぐことは，本研究の主な目的の 1 つである．本研究での提案手法はメモリリーク対策という点では，先行研究 [4–7] と類似している．しかしながら，本研究での提案手法はメモリリークによって引き起こされる大量のメモリ割り当てを防ぐことを目的としており，メモリリークそのものを検出することは行わない．この点で先行研究と本研究は相違している．

この研究では，cgrouops の機能を拡張することによってシグナルに依らないプロセスの一時停止を実現している．cgroup は Process Containers [8] を基盤としている．(Process Containers は 2007 年に cgroups に改名された [9])．その他にも，リソース分離を階層化グループで行う研究がある [10, 11]．先行研究 [10, 11] はいずれもネットワークサーバーアプリケーションを対象に細粒度かつ柔軟なリソース管理を実現することを目的としている．これらの先行研究では，関連のあるプロセスをグループ化しそのグループごとにリソース制限を設ける概念が導入された．本研究での提案手法は，このリソース管理のためのプロセスグルーピングを，先行研究とは異なる形で応用したものである．先行研究では，複数のプロセスが 1 つのグループに登録されるが，本研究での提案手法は，1 つの管理グループに 1 つのプロセスしか登録されない．この研究は，cgroups や類似した手法 [10, 11] を応用して，新しいリソース管理の形態を検討することをその目的の 1 つとしている．

## 6. まとめと今後の課題

計算機システムで動作するプログラムのリソースを制限することは，システムの安定を保つという点で重要である．特にメインメモリはシステムの安定性への影響が多い重要なリソースである．Linux には，すでに prlimit や cgroup という形でメモリリソースを制限する仕組みが備わっているが，ユーザが意図しない，急激で大量のメモリ消費に対しては十分に対抗することができない．

本論文では，その課題を解決するアプローチの 1 つとして，プロセスごとにおメモリ消費の速度に基づいたメモリ管理の手法を提案した．本手法を用いることで，プロセスの動作に基づいて，システムを不安定化する可能性のあるプロセスを検出し，予防的にその動作を一時停止する．これにより，意図しない，急激で大量のメモリ割り当てによってシステムが不安定化することを未然に防止することができる．

今後の課題としては，実装した提案手法を用いた評価実験がまず第一に挙げられる．さまざまなワークロードをを対象に，提案手法の有効性を検証する予定である．また，その実験を通して得られた知見を元に，提案手法の改良にも取り組む予定である．

### 参考文献

[1] Denning, P. J.: Thrashing: Its Causes and Prevention, *Proc. of Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), ACM, pp. 915–922 (1968).
[2] Eckhardt, D. and Kerrisk, M.: Linux Programmer's Manual GETRLIMIT(2), `http://man7.org/linux/man-pages/man2/prlimit.2.html`.
[3] Lim, G., Min, C. and Eom, Y.: Virtual memory partitioning for enhancing application performance in mobile platforms, *IEEE Transactions on Consumer Electronics*, Vol. 59, No. 4, pp. 786–794 (2013).
[4] Hastings, R. and Joyce, B.: Purify: Fast detection of memory leaks and access errors, *Proceedings of the Winter 1992 USENIX Conference*, USENIX Association,

pp. 125–138 (1991).

[5] Jung, C., Lee, S., Raman, E. and Pande, S.: Automated Memory Leak Detection for Production Use, *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM, pp. 825–836 (2014).

[6] Hauswirth, M. and Chilimbi, T. M.: Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, ACM, pp. 156–164 (2004).

[7] Clause, J. and Orso, A.: LEAKPOINT: Pinpointing the Causes of Memory Leaks, *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, ACM, pp. 515–524 (2010).

[8] Menage, P. B.: Adding generic process containers to the linux kernel, *Proceedings of the Linux Symposium*, Vol. 2, pp. 45–57 (2007).

[9] Corbet, J.: Notes from a container, `http://lwn.net/Articles/256389/` (2007).

[10] Blanquer, J., Bruno, J., Gabber, E., Mcshea, M., Ozden, B., Silberschatz, A. and Singh, A.: Resource Management for QoS in Eclipse/BSD, In Proceedings of the FreeBSD ' 99 Conference (1999).

[11] Banga, G., Druschel, P. and Mogul, J. C.: Resource Containers: A New Facility for Resource Management in Server Systems, *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, USENIX Association, pp. 45–58 (1999).