オペレーティングシステムの
カーネル外から資源管理ポリシーを
変更する手法に関する研究

山田 浩史

慶應義塾大学大学院
理工学研究科開放環境科学専攻
博士 (工学) の学位申請論文

2009年3月

オペレーティングシステムのカーネル外から資源管理ポリシーを変更する手法に関する研究

山田浩史

論文要旨

オペレーティングシステム (OS) における資源管理ポリシーはアプリケーションの種類や実行環境に応じて適切なものを選択する必要がある．アプリケーションやその実行環境の多様化に伴い，新たな資源管理ポリシーが今もなお提案され続けている．これまでに有用な資源管理ポリシーが数多く提案されてきたが，これらのポリシーを実用に供することは容易ではない．既存 OS の資源管理ポリシーを変更するには，肥大化・複雑化した OS カーネルのソースコードを修正する必要があり，これらを解析・修正することには多大な労力を要する．また，ソースコードが入手しにくい商用 OS はカーネルの修正さえ難しい．結果として，資源管理ポリシーが提案されてから OS に組み込まれるまでの期間は長く，ユーザはその間新しいポリシーの恩恵を享受することが難しい．そこで，所望のポリシーが OS に組み込まれるまでの間に擬似的にそのポリシーを使用する手法として，カーネルを修正することなく，ユーザレベル上で資源管理ポリシーを変更する手法が提案されてきた．しかし，これまでの手法は，システムコールから資源使用量を把握できるネットワーク帯域や， OS によってユーザレベルへ公開している内部状態から資源使用量を取得できる CPU 時間などが対象であった．そのため，ユーザレベルから使用量を把握できないディスク帯域や，OS がユーザレベルへ十分に内部状態を提供しない TCP の輻輳制御に関するポリシーをカーネルの修正なしに変更することは困難である．

本論文では，OS の動作をカーネル外から推測することにより，変更可能なポリシーの範囲を拡大する手法を提案する．まず，ユーザレベルでプロセスのディスク帯域割り当てポリシーを変更する機構，*DiscNice* を提案する．ディスク I/O は，OS の介在により，プロセスによる read() や write() といったファイル I/O とは連動しない．そこで，DiscNice ではディスクアクセスに関する OS の動作に着目する．ファイル先読みといった OS の動作を考慮しながらファイル I/O に伴うディスク I/O サイズを推測することで，ディスク帯域使用量をユーザレベルで把握可能にする．制限ポリシーに従って，ファイル I/O の発行を調整することで，間接的に

ディスク帯域使用量を制限する．プロトタイプを Linux 2.4.27 上に実装し，監視下のアプリケーションのディスク帯域使用量を制限するポリシーが擬似的に導入できることを示す．

次に，ユーザレベルで扱えない情報を利用しながら OS の動作を推測することで，資源管理ポリシーを擬似的に導入する手法として，仮想マシンモニタ (VMM) を利用する手法，*FoxyTechnique* を提案する．VMM 上では，OS は VMM の提供する仮想デバイス上で稼働する．FoxyTechnique は，仮想デバイスに対する OS の動作を考慮しながら，ユーザレベルでは行えないデータレジスタ値の改変や OS からのリクエストの遅延など行うことで，OS に実環境とは異なった環境を提供し，OS に所望の動作をさせる．この手法を用いて，TCP 輻輳制御ポリシーの 1 つである TCP Vegas の動作をネットワークインターフェースに届くデータの値を書き換えることで擬似的に作り出すことができる．さらに，ディスクスケジューリングポリシーの 1 つである IdleTime スケジューリングの動作もディスクコントローラに届く OS からのリクエストを遅延させることで擬似的に作り出すことができる．実際に，Xen 3.0.2-2 上にプロトタイプを実装し，上述のポリシーの恩恵を享受できることを示す．

A Study on Changing Resource Management Policies from outside Operating System Kernels

Hiroshi Yamada

Abstract

Developing resource management policies of the operating system (OS) is still an ongoing process. Because an appropriate resource management policy depends largely on the type of applications and their computing environments, OS researchers must continue to develop innovative resource management policies to satisfy the needs of emerging applications and everchanging computing environments. Despite the numerous innovative policies being proposed, it is quite difficult to widely deploy a single innovation. The traditional way to change resource management policies is to modify OS kernel source code. Modifying a kernel is a difficult, costly and often impractical endeavor. Modern OSes consist of large and complex bodies of code, and it is almost impossible to modify proprietary and closed source OSes. As a result, users cannot enjoy a new resource management policy until the integration of the policy in an OS, which is a time-consuming task, has been completed. To obtain effects similar to a new policy until an OS supports it, some research projects have explored the way to change resource management policies at the user-level without any modification of kernels. However, their approaches focus only on the policies that manage the resources whose state can be obtained at the user-level. Specifically, they just target the resources such as network bandwidth and CPU time, whose state is explicitly exposed by an OS. The resources such as disk bandwidth whose state cannot be observed at the user-level have been out of scope of the existing techniques. In addition, the policies whose execution is blind at the user-level, such as TCP congestion control policies, are still difficult to change.

In this dissertation, we propose techniques to expand the scope of policies feasible without modifying kernels. Our key insight is inference of OS behavior from outside the kernel. The dissertation first presents *DiscNice*, a user-level mechanism which enables us to control disk bandwidth usage. Disk I/O is not always accompanied by file I/O because the OS caches and reads disk blocks in advance. DiscNice predicts disk I/O size from the file I/O, making an inference of the OS disk access behavior. We implemented

the prototype on Linux 2.4.27 and demonstrate that we can roughly carry out a policy under which the disk bandwidth of user-specified processes is throttled to a given one.

The dissertation next describes *FoxyTechnique*, a technique which brings effects similar to a desired policy by inferring OS behavior and using information unavailable at the user-level. FoxyTechnique uses virtual machine monitors (VMMs), which emulate the underlying devices for OSes. FoxyTechnique controls virtual devices (e.g., modifies the value of data registers and delays requests from the OS), inferring the OS reaction against the transformed devices. By using this technique, we can make OS behavior similar to a desired policy. In fact, we can forge TCP Vegas, one of TCP congestion control policies, by modifying packets inside the network interface card. The technique also allows us to forge IdleTime scheduling, one of disk scheduling policies, by delaying OS disk requests inside the disk controller. We implemented these prototypes on Xen 3.0.2-2 and show that we can obtain benefits of the targeted policies.

目 次

第 1 章　はじめに **1**
1.1　背景 1
1.2　本研究の動機 3
1.3　本研究の目的 5
1.3.1　ユーザレベルでのディスク帯域割り当てポリシーの変更 . . 6
1.3.2　仮想マシンモニタレベルでの資源管理ポリシー変更手法 . . 7
1.4　本研究の貢献 7
1.5　本論文の構成 9

第 2 章　関連研究 **10**
2.1　カーネルによる資源管理ポリシー変更のサポート 10
2.2　ユーザレベルでの資源管理ポリシー変更手法 11
2.2.1　OS による内部情報提供の利用 12
2.2.2　OS が提供する内部情報を用いたアプローチ 13
2.2.3　GrayBox 手法 15
2.2.4　GrayBox 手法を用いたアプローチ 16
2.3　仮想化技術を利用した手法 17
2.3.1　仮想化技術 17
2.3.2　VMM を活用したアプローチ 19
2.4　まとめ 21

第 3 章　カーネル外による OS の動作推測 **23**
3.1　ユーザレベルでの OS の動作推測 23
3.2　VMM レベルでの OS の動作推測 24
3.3　議論 25
3.4　まとめ 25

第 4 章　ユーザレベルでのディスク帯域割り当て変更　27
4.1　ディスク帯域制限の必要性と困難さ 28
4.1.1　必要性 28
4.1.2　ディスク帯域割り当て変更の困難さ 33
4.2　ユーザレベルでのディスク帯域割り当ての変更 34
4.2.1　設計 35
4.2.2　Cache Detector 37
4.2.3　Read-Size Predictor 38
4.2.4　Write-Size Predictor 39
4.3　実装 40
4.3.1　libDiscNice 41
4.3.2　write-adjuster 43
4.3.3　policy_d と現在のポリシー 44
4.3.4　Read-Size Predictor 46
4.3.5　議論 46
4.4　実験 47
4.4.1　マイクロベンチマーク 47
4.4.2　DiscNice モジュールの有効性 49
4.4.3　マクロベンチマーク 52
4.4.4　オーバヘッド 53
4.4.5　ケーススタディ 56
4.5　まとめ 57

第 5 章　VMM レベルでの資源管理ポリシー変更手法　59
5.1　FoxyTechnique 61
5.1.1　Foxy VMM 63
5.1.2　議論 64
5.2　実装 65
5.2.1　Xen 66
5.3　Case Study: FoxyVegas 68
5.3.1　TCP Vegas 68
5.3.2　設計 68
5.3.3　実装 70

5.3.4 実験 70
5.4 Case Study: FoxyIdle 75
5.4.1 アイドル時間スケジューリング 75
5.4.2 設計 76
5.4.3 実装 78
5.4.4 実験 79
5.5 まとめ 86

第 6 章 おわりに **90**
6.1 まとめ 90
6.2 今後の展望 91
6.2.1 他の資源管理ポリシーの擬似的な導入 91
6.2.2 他の OS への適用 91
6.2.3 OS プロファイリングツールの利用 92

謝辞 **93**

論文目録 **95**

参考文献 **96**

図目次

1.1 The contribution of this dissertation. 9

4.1 Disk bandwidth usage. 31
4.2 CPU usage. 32
4.3 Memory usage. 33
4.4 Difference between file I/O and disk I/O. 34
4.5 Control flow in DiscNice. 36
4.6 An example of cache detection. 37
4.7 OS behavior in file creation and updates. 40
4.8 Write-size predictor behavior in file creation and updates. 41
4.9 DiscNice architecture 42
4.10 Rate-windows. 45
4.11 Results of throttling micro-benchmark programs. 49
4.12 Results of throttling 4-phases benchmark. 51
4.13 Results of throttling macro-benchmark programs. 54
4.14 Average disk I/O rate of `Tar` and `Make` with DiscNice. 55

5.1 The difference between conventional VMM and Foxy VMM. 62
5.2 Implementation. 67
5.3 FoxyVegas 69
5.4 Comparison of the window size. 72
5.5 Comparison of RTT fluctuations 74
5.6 Idletime scheduling. 76
5.7 FoxyIdle. 77
5.8 Difference of the execution time of foreground `Sequential` with background `Sequential` on native Idle, guest Idle and FoxyIdle. . . . 81

5.9 Difference of the execution time of background `Sequential` with foreground `Sequential` on native Idle, guest Idle and FoxyIdle. . . . 82
5.10 Difference of the execution time of foreground `Random` with background `Sequential` on native Idle, guest Idle and FoxyIdle. 83
5.11 Difference of the execution time of background `Sequential` with foreground `Random` on native Idle, guest Idle and FoxyIdle. 84
5.12 Timer Errors of the idletime schedulers. 86
5.13 Difference between the concurrent execution and the FoxyIdle execution. 88

表目次

4.1 Performance degradation of `Sequential` caused by disk I/O contention with `Rsync`. 29
4.2 Performance degradation of `Tar` caused by disk I/O contention with `Rsync`. 30
4.3 Experimental environment 48
4.4 Accuracy of disk I/O prediction (micro-benchmark). 48
4.5 DiscNice configuration. 50
4.6 Accuracy of disk I/O prediction (macro-benchmark). 53
4.7 Overhead incurred by DiscNice. 56
4.8 Comparison of execution times of `Sequential` 57
4.9 Comparison of execution times of `Tar` 57

5.1 Comparison of throughput for various configurations of TCP Vegas. . . 73
5.2 Comparison of throughput in network congestion. 75
5.3 Comparison of execution times of `Grep` and `Updatedb`. 87

第1章　はじめに

1.1　背景

オペレーティングシステム (OS) とは，計算機資源とユーザとの間に介在し，計算機資源を管理するソフトウェアである．OS の目的は，アプリケーションを適切に効率よく実行する環境をユーザに与えることである．CPU やデバイスの高速化やネットワークの発達に伴って，汎用 OS に求められる機能は時間とともに変化している．従来は CPU やハードウェアの使用率を最大限に高めることが第一目標であった．しかし，現在は，デスクトップ PC 上で音楽プレーヤや 3D グラフィックスを描画するゲームといったマルチメディアアプリケーション，ワードプロセッサや表計算ソフトといったビジネスアプリケーションも稼働するようになり，ユーザの利便性や応答性を重視するようになっている．また，高性能化が進んだデスクトップ PC を用いて Web サービスを運用する例も少なくなく [1]，Web サーバのパフォーマンスを最大限に高める機能も要求されている．

アプリケーションを適切に動作させるための重大な要素のひとつとして，資源管理ポリシーが挙げられる．OS における資源管理ポリシーとは， CPU 時間やメモリ量，ディスク帯域やネットワーク帯域といった計算機資源のプロセスへの割り当て方針を指す．狭義には，プロセスへの CPU の割り当て方を決定する CPU スケジューリングポリシーやプロセスへのメモリの割り当て方を決定するメモリ割り当てポリシー，そして I/O デバイスへのリクエストの処理順番を決定する I/O スケジューリングポリシーが挙げられる．また間接的に計算機資源の割り当てに関わるポリシー，たとえば，メモリのスワッピングポリシーや TCP の輻輳制御ポリシーなども広義には資源管理ポリシーである．資源管理ポリシーによって計算機資源の利用方針が決定するため，アプリケーションのパフォーマンスは資源管理ポリシーに大きく依存する．

OS における資源管理ポリシーはアプリケーションの種類や実行環境に応じて適切なものを選択すべきであることが知られている．もし資源管理ポリシーを適切

に選ばなければ，アプリケーションのパフォーマンスを著しく低下させたり，本来実現すべき実行環境を構築することができなくなってしまう．たとえば，ウィルスキャンやバックアップユーティリティ，PC グリッドアプリケーション [2, 3] といったタスク (バックグラウンドタスク) は，ユーザが現在使用しているタスク (フォアグランドタスク) の動作を妨げることなく動作することが望ましい．ここで，もしすべてのプロセスに公平な量の資源を与えるフェアシェアスケジューリングポリシーを採用していると，バックグラウンドタスクが動作することでフォアグラウンドタスクの使用できる資源量が減ってしまう．結果として，フォアグラウンドタスクのスループットや応答時間が低下する．この際は，フォアグランドタスクの資源使用量が減少しないようバックグラウンドタスクに資源を与えるというポリシーを試行することが求められる [4, 5, 6, 7, 8, 9]．また，1 台の物理マシン上で複数のホストを稼働させる仮想ホスティング環境では，サービス提供者とユーザとの契約内容に応じたサービスが提供できるよう，各ホストを動作させることが望ましい．もし，同様にフェアシェアスケジューリングを採用していると，契約内容とは関係なく各ホストが同じ品質でサービスを提供してしまう．この場合には，各ホストへ割り当てる資源をあらかじめ指定可能な資源管理ポリシーを試行することが求められる [10, 11, 12, 13, 14].

アプリケーションやコンピュータの利用形態の多様化に伴い，OS に求められる資源管理ポリシーも多様化し続けている．そのため，現在もなお新たな資源管理ポリシーの提案が盛んに行われている [15, 16, 17, 18, 19, 20, 21].

これまで数多くの資源管理ポリシーが提案されてきたが，これらのポリシーを実用に供することは容易ではない．OS の資源管理ポリシーは OS カーネル内で実現されており，ポリシーを変更するにはカーネルを修正しなければならない．現在，広く利用されている OS カーネルには様々な機能が組み込まれており，そのソースコードは長い年月をかけて肥大化，複雑化し続けてきた．その結果，カーネルのソースコードを解析，修正することには多大な労力を要してしまう．そのため，文献 [22] では，カーネルのことを “hardware masquerading as software” と比喩している．仮に，ある OS カーネル上に所望のポリシーを実装したとしても，そのポリシーの保守は難しい．なぜなら，セキュリティホールやバグ修正に伴うカーネルのバージョンアップが頻繁に行われているためである．バージョンアップによってカーネルの内部構造が変わることも多い．そのため，ポリシーを保守するためには，バージョンアップの度に再びカーネルの内部構造を解析し，適宜修正を加えなければならない．さらに，Windows Vista や Mac OS X といった商用 OS

やソースコードが入手しにくい OS はカーネルの修正そのものが非常に困難である．結果として，多くの新たな資源管理ポリシーはその有用性にもかかわらず，実用に至るまでに長い時間を要してしまう．

1.2 本研究の動機

新たな資源管理ポリシーの導入を円滑に行うために，これまで OS 自身の構成を再構成可能にする手法が提案されてきた [23]．マイクロカーネル [24, 25, 26, 27, 28] では，OS カーネルで提供する機能をアドレス空間の管理やプロセス間通信といったものに限定し，ファイルシステムやプロトコルスタックといったサービスをユーザレベル上でサーバとして実現する．これにより，新たな資源管理ポリシーを導入する際は，カーネルを修正することなく，新たなサーバを実装し，既存のものと差し替えるだけでよい．また，拡張可能 OS [29, 30, 31, 32] では，カーネルへの様々な機能追加を可能にする機能を提供する．ユーザは拡張可能 OS に対応したモジュールを作成することで，アプリケーションを適切に動作させる機能や資源管理ポリシーの導入が可能になる．他のアプローチとして，OS が自身の内部状態をユーザレベルに提供することで，ユーザレベルで資源管理ポリシー変更機構の実装を促進するインフォカーネル (Infokernel) 方式もある [33, 34]．しかしながら，ユーザがこれらの恩恵を受けるためには，上述の拡張機能が組み込まれた特別なカーネルを使用しなければならず，使い慣れていない OS を使うことを強いられる．もしくは，拡張機能をサポートするように，自身の使っている OS カーネルに大幅な修正を施さなければならない．OS ベンダからすると，これまで開発したカーネルソースコードの大部分を破棄し，新たにソースコードを開発することになるため，そのコストは高い [35, 33]．そのため，実際に，Linux や Windows といった広く利用されている OS にこれらの機能は組み込まれておらず，依然として資源管理ポリシーの変更には時間を要する．

そこで，所望の資源管理ポリシーが OS に導入されるまでの間，そのポリシーの効果を得る手法として，カーネル外から資源管理ポリシーを変更する手法が提案されている．この手法では，OS カーネルに修正を加えることなく，ユーザレベル上でプロセスを制御することで，カーネル内に所望の資源管理ポリシーを実現したような効果を得る．つまり，新たな資源管理ポリシーの擬似的な導入を可能にする．カーネルを変更することなく資源管理ポリシーを導入することには以下

のようなメリットがある．カーネルの修正を必要としないので，多くの有用な資源管理ポリシーの恩恵を擬似的にではあるが早期に享受することができる．また，資源管理ポリシー変更機構はユーザレベルで動作するため，カーネル内に実装するよりも，他の OS への移植も容易であると考えられる．さらに，OS ベンダがサポートするべき資源管理ポリシーの選択に要するコストを下げることも期待できる．従来はカーネルを修正しなければ，ポリシーの有用性を実際に確かめることができなかった．ユーザレベルでの手法を用いることで，多くのポリシーの有用性を実稼働する OS 上で擬似的に確かめることができるためである．カーネルを修正せずに資源管理ポリシーを導入することには以上のようなメリットがあり，その研究成果が期待されている．現在，どのような資源管理ポリシーがカーネル外から導入可能であるかを明確にするために，様々な資源管理ポリシーを導入している [35, 36, 37, 38, 39, 40, 41, 42].

ユーザレベルでプロセスを適切に制御するためには，OS の内部挙動に関する情報を取得する必要がある．取得した情報から OS の内部挙動を把握して，プロセスの挙動を調整することで，所望の資源管理ポリシーを導入したときに近い効果を得る．具体的には，システムコールやその引数，プロセスファイルシステムといった OS が自身の内部状態を提供する機構を利用することで OS の内部挙動を把握する．たとえば，UPS [41] では，ネットワーク帯域を制限するために，`send()` といったプロセスのネットワークの送信に関するシステムコールをフックしている．その引数を監視することで，実際にネットワークに送信されるメッセージ量を算出して，閾値を超えると，システムコールの発行を遅延させる．また，Newhouse らは，CPU プロポーショナルシェアスケジューリングポリシーを擬似的に導入するために，各プロセスの CPU 使用時間を OS が提供している情報から取得している．ここから各プロセスの CPU 使用率を計算し，閾値を超えたプロセスをシグナルによって停止する [37, 36]．ユーザレベルで取得できる情報を利用することで，これまでに CPU 使用率やメモリ使用量の制御，ネットワーク帯域幅の制限といったポリシーの導入手法が示されている [36, 37, 38, 39, 40, 41]．さらにユーザレベルで取得できる内部挙動を増やすために，Graybox 手法 [35] と呼ばれる手法が提案されている．GrayBox 手法は，ユーザレベルのある操作に対する OS の反応を観察することで OS の内部状態を取得する手法である．この手法を用いることで，OS がユーザレベルに明示的に提供していない内部状態を把握することができる．この手法を応用することで，ユーザレベル上でバッファキャッシュの内容を考慮したファイルプロセッシングやファイルブロックのディスク上のレイアウト操作と

いったことが可能となる [35, 42].

カーネル外から導入できる資源管理ポリシーの種類は，カーネル外から把握できる OS の内部挙動に依存している．OS の内部挙動を十分に把握できない場合，プロセスの挙動を適切に制御することができないからである．たとえば，ユーザレベルで各プロセスの CPU 使用時間が把握できない場合，CPU プロポーショナルシェアスケジューリングポリシーの導入は難しい．各プロセスの CPU 使用率が計算できないため，停止するべきプロセスを判断することができないためである．このように，カーネル外から把握可能な OS 内部挙動が少ないと，所望の資源管理ポリシーを導入することは困難である．実際，現状では，システムコールと資源使用量が連動しない資源に対するポリシーや OS の内部状態が取得できないポリシーの変更は困難である．たとえば，これまでディスク帯域の割り当てポリシーをユーザレベルで変更する手法は提案されていない．ディスク帯域の競合はアプリケーションへのパフォーマンスへの影響が大きい．ディスク帯域の割り当てポリシー変更の重要性にもかかわらず，ユーザレベル上でディスク帯域を制御する手法が提案されていない．また，OS がユーザレベルへ十分に内部状態を提供しない TCP の輻輳制御ポリシーの変更も困難である．適した TCP の輻輳制御ポリシーはワークロードや環境によって変わるため，用途に応じて迅速に変更する必要がある．実際に，これまでに数多くの TCP 輻輳制御ポリシーが提案されている．しかし，ユーザレベルで TCP の輻輳制御ポリシーを変更するためには特殊な内部情報が必要であるため [33, 34]，Linux や Windows といった OS 上での変更は困難である．

1.3 本研究の目的

本研究では，OS カーネルを修正することなく，擬似的に導入可能な資源管理ポリシーの範囲を拡大する手法を提案する．提案手法では，カーネル外から OS の動作を推測することに着目する．OS カーネルのソースコードやドキュメントを利用することで，OS がどのように動作しているかを知ることができる．たとえば，OS が採用している CPU スケジューリングポリシーやバッファキャッシュ管理ポリシーといった知識を得ることができる．この知識を利用することで，ある操作に対する OS の反応や，その操作後の動作を推測することができる．これにより，カーネル外から把握できる OS の内部挙動をより多くする．

OS の動作の推測に加え，カーネル外から取得できる情報と併用することで，これまで導入が困難であったポリシーを擬似的に導入する．まず，ユーザレベルでプロセスのディスク帯域の割り当てを変更する機構を提案する．OS は各プロセスの振る舞いによって生じるディスク I/O サイズをユーザレベルに提供しない．そこで，ディスクアクセスに関する OS の動作を推測することで，プロセスの発行するファイル I/O からディスク I/O サイズを推測する．推測したサイズを元にディスク帯域使用量を把握し，ファイル I/O の発行を制御する．さらに，OS の動作を推測することで仮想化技術 [43] を資源管理ポリシーの導入手法に応用することができる．本研究では，仮想化環境で動作する OS (ゲスト OS) の資源管理ポリシーを変更する手法を提案する．仮想マシンモニタ (VMM) では，デバイスが扱うディスクブロックやネットワークパケットといったユーザレベルとは異なるデータを取得，操作することができる．そこで，提案手法では，デバイスに対する OS の動作を推測する．デバイスの振る舞いを意図的に変更することで，ゲスト OS の動作を制御する．これにより，資源管理ポリシーを導入したときに近い効果を得る．

1.3.1 ユーザレベルでのディスク帯域割り当てポリシーの変更

ディスク I/O は CPU 速度やメモリへのアクセス速度と比べて格段に低速である．そのため，4 章でも述べるように，ディスク帯域の競合によるアプリケーションへの影響は小さくない．ディスク帯域制御の必要性 [9, 13] にもかかわらず，汎用 OS にはプロセスごとにディスク帯域割り当てを変更する資源管理ポリシーが導入されていない．また，これまでユーザレベルでディスク帯域割り当てを変更する機構は提案されていない．提案機構ではディスクアクセスに対する OS の動作に着目する．OS の動作を推測し，プロセスのファイル I/O を監視することで，ディスク帯域幅の使用量を把握する．

本論文では，ユーザレベルで動作するディスク帯域制限機構である *DiscNice* の設計，実装ならびに実験結果について述べる．Linux 2.4.27 上に DiscNice のプロトタイプを実装し，実験を行う．マイクロベンチマーク，マクロベンチマークを用意し，DiscNice のディスク I/O サイズの推測誤差を測定したところ，最大でも精度誤差は 4.4% であることを報告する．また，オーバーヘッドも 12% 以下である．さらに，DiscNice を用いて，優先度の低いプロセスのディスク帯域を制限することで，ディスク帯域競合によるフォアグラウンドタスクのスループット劣化を緩和可能かを確認する実験についても述べる．実験により，DiscNice で制限するディ

スク帯域使用量に従って，フォアグランドタスクのスループット劣化を緩和可能であることを示す．

1.3.2 仮想マシンモニタレベルでの資源管理ポリシー変更手法

今日，仮想化技術はサーバやデスクトップ PC 上に広く普及している．仮想化技術は，1 台のマシン上に複数の OS を稼働させる環境を提供する．仮想化環境では，仮想マシンモニタ (VMM) と呼ばれるソフトウェアが CPU やメモリ，I/O デバイスといったすべての計算機資源を管理し，仮想デバイスを提供する．ゲスト OS は仮想デバイス上で稼働することになる．提案手法では，デバイスの振る舞いに対する OS の動作に着目する．OS の動作を推測し，仮想デバイスの振る舞いを変更することで，ゲスト OS に所望の動作をさせる．これにより，資源管理ポリシーを導入したような効果を得る．

本論文では，ゲスト OS の資源管理ポリシーを変更する手法，*FoxyTechnique* について述べる．さらに，FoxyTechnique を用いた資源管理ポリシー変更機構である *FoxyVegas*，*FoxyIdle* の設計，実装ならびに実験結果について述べる．FoxyVegas は TCP 輻輳制御ポリシーの 1 つである TCP Vegas [44] の動作を，FoxyIdle はディスクスケジューリングポリシーの 1 つであるアイドル時間スケジューリング [9] の動作を作り出す機構である．これらを Xen 3.0.2-2 上に実装し，ゲスト OS として Linux 2.6.16 を稼働させ実験を行う．FoxyVegas に関しては，Linux が搭載している 9 つのすべての TCP 輻輳制御ポリシーに対して FoxyVegas を適用し，その有用性について報告する．具体的には，FoxyVegas は 8 つのポリシーの振る舞いを適切に変更し，TCP Vegas の効果をもたらすことを示す．また，FoxyIdle に関しても，Linux が搭載している 4 つのすべてのディスクスケジューリングポリシーに対して FoxyIdle を適用し，その結果について述べる．実験結果は，FoxyIdle は 4 つすべてのポリシーの振る舞いを適切に変更し，アイドル時間スケジューリングの効果をもたらすことを示している．

1.4 本研究の貢献

カーネル外から資源管理ポリシーを変更するためには，OS の内部挙動に関する情報を取得する必要がある．これまでは，CPU 使用率の制御やネットワーク帯域幅の制限といった，システムコールと資源使用量がほぼ連動する資源に対するポ

リシーや，ポリシーを変更するために必要な OS の内部情報が明確に把握できるものが対象であった．そのため，ディスク帯域割り当てポリシーや TCP 輻輳制御ポリシーといった，その動作時の挙動や情報が OS によって明示的に提供されていない，もしくは既存の GrayBox 手法では取得できないポリシーを変更することは困難である．本研究では，OS の動作を推測することで，OS の内部挙動をカーネル外からより多く把握し，擬似的に導入可能な資源管理ポリシーの範囲の拡大を行う．

本研究では，ユーザレベルでプロセスのファイル I/O に伴うディスク I/O の量が推測可能であることを示す．さらに，その推測機構の設計，実装についても示す．これを利用することでディスク帯域の割り当てポリシーをユーザレベルで変更可能であることを示す．

さらに，本研究では VMM レベルでゲスト OS の資源管理ポリシーを変更する手法について示す．VMM では，OS とデバイスとのやりとりを完全に監視することができるため，ユーザレベルとは異なるデータの取得，操作が行える．本研究では，FoxyTechnique を用いて TCP 輻輳制御ポリシーの 1 つである TCP Vegas，ディスクスケジューリングポリシーの 1 つであるアイドル時間スケジューリングの擬似的な導入を可能にする機構の設計，実装についても示す．

また，本研究で提案した機構の実装は実際の OS や VMM 上で行っている．実験に関しても，実際のアプリケーションを用いて行い，その有用性を検証している．

本研究を図 1.1 のように位置づけることができる．所望の資源管理ポリシーを導入するためには，従来はカーネルを修正する必要がある．直接 OS カーネルの修正することで，様々な資源管理ポリシーを導入することができる反面，カーネルの修正という多大な労力を強いられる．そこで，資源管理ポリシーを円滑に導入するために，マイクロカーネルや拡張可能 OS，インフォカーネルといった OS の構成法が提案されてきた，マイクロカーネルはカーネル内で実装する機能によって，変更可能なポリシー数は変動する．拡張可能 OS は，OS の様々なコンポーネントを拡張できるので，様々な資源管理ポリシーを導入することができる．インフォカーネルは前述の 2 つの OS 構成法に比べると OS の書き換え量は少なくすむが，導入できる資源管理ポリシーはその分少ない．これらの手法はカーネルを修正することなく資源管理ポリシーを導入できる．しかし，これらの恩恵を受けるには，これらの機能が組み込まれた特別なカーネルを使用しなければならない．もしくは，これらの機能をサポートするよう，現在利用している OS のカーネルを修正する必要がある．そこで，カーネルを修正することなく，カーネル外から資源管理

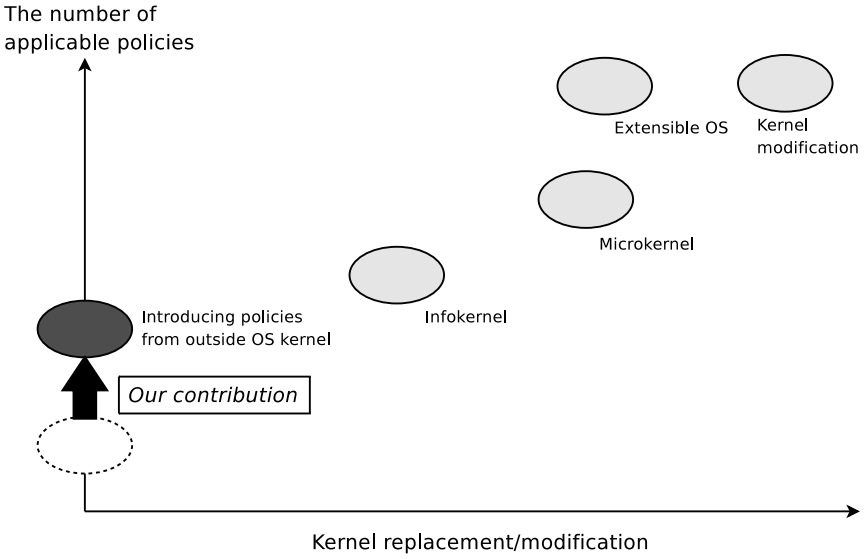


図 1.1: **The contribution of this dissertation.**

ポリシーの導入を擬似的に行う手法が注目されている．しかしながら，その適用範囲は他のアプローチと比べると狭い．現在，カーネル外から導入可能な資源管理ポリシーの種類を明確にするために，様々な種類の資源管理ポリシーの導入が活発に行われているのが現状である．本研究では，OS の推測に着目し，DiscNice, FoxyTechnique の提案を行うことで，カーネル外から導入可能な資源管理ポリシーの範囲を拡大したと言える．

1.5 本論文の構成

本論文の構成は以下の通りである．2 章では関連研究について述べる．具体的には，カーネルレベルとユーザレベル，VMM レベルと大別して，既存の手法について示す．3 章では，カーネル外からの OS の動作推測について述べる．まずユーザレベルにおける OS の動作推測に述べ，次に VMM レベルにおける OS の動作推測について述べる．4 章では，DiscNice について述べる．まず，ディスク帯域制御の必要性について述べ，ユーザレベルでプロセスのディスク帯域使用量を把握するために考慮すべき OS の動作について考察する．そして，DiscNice 設計，実装について述べ，実験と実験結果を示す．5 章では，FoxyTechnique について述べる．まず，FoxyTechnique の概要について述べる．次に，実装について述べ，FoxyTechnique を用いたポリシー変更機構である FoxyVegas と FoxyIdle の実装ならびに実験，実験結果について示す．最後に 6 章で本論文をまとめる．

第2章　関連研究

本研究では，カーネル外から擬似的に導入できる資源管理ポリシーの範囲の拡大を行う．資源管理ポリシーを擬似的に導入するためには，カーネル外から OS の内部挙動を知る必要がある．提案手法では，OS から取得できる情報に加え，OS の動作を推測することで 従来より OS の内部挙動をより多く把握する．これらの情報を利用しながら所望の資源管理ポリシーの導入を行う．本章では，本研究の関連研究について述べる．最初に，資源管理ポリシーの導入を容易にするためのカーネル構成法に関する研究について述べる．そして，カーネル外から資源管理ポリシーを擬似的に導入する既存手法について述べる．まずユーザレベルでの資源管理ポリシーの擬似的な導入手法について述べる．具体的には，ユーザレベルから OS の内部挙動を把握する手法について述べ，既存手法について示す．続いて，仮想マシンモニタを用いた手法について述べる．まず，仮想化技術の概要について述べ，仮想マシンモニタ内で資源を効率よく管理することで，ゲスト OS に恩恵をもたらす手法について述べる．

2.1　カーネルによる資源管理ポリシー変更のサポート

用途に応じた資源管理ポリシーを円滑に導入するために，カーネルの構成を再構成可能にする研究が行われてきた [23]．カーネルを再構成可能にすることで資源管理ポリシーの導入が円滑になる反面，これらの機能が組み込まれた特別なカーネルを使用することが課せられる．また，既存の OS 上でこれらの機能を利用するには，カーネルを再構成可能な構造にするためにカーネルコードを大幅に修正する必要がある．

マイクロカーネル [24, 25, 26, 27, 28] では，カーネルはアドレス空間の管理やプロセス間通信といった機能のみを提供し，その他のシステムサービスをユーザレベルで動作するサーバとして実装する．具体的には，デバイスドライバやファイルシステム，プロトコルスタックといった機能をユーザレベルのサーバとして動

作させる．これにより，所望の資源管理ポリシーをサーバとして実装し，既存のサーバと差し替えることで，そのポリシーを試行することができる．

拡張可能 OS [29, 30, 31, 32] は，用途に応じた機能や資源管理ポリシーを導入できる機能を提供する．たとえば，SPIN [29] では，専用のカーネルモジュールを SPIN カーネル内に動的に導入することができる．カーネルモジュールを Moduler-3 と呼ばれる型安全な言語で記述することで，カーネルモジュールの安全な導入を実現している．また，Exokernel [31, 32] では，ハードウェアのインターフェースに近いインターフェースを提供している．これを利用して所望の資源管理ポリシーを搭載したライブラリのような OS (libOS) を実装し，アプリケーションとリンクさせ動作させる．これにより，アプリケーションにとって適切な資源管理ポリシーを試行しながら，アプリケーションを動作させることができる．

カーネルの修正が困難であることを考慮し，カーネルの修正をできるだけ少なくし，資源管理ポリシーの導入をサポートする手法がある．インフォカーネル [33] では，ユーザレベルによる資源管理ポリシー導入を支援するために，OS の内部情報の詳細をユーザレベルに提供する．具体的には，ディスクスケジューリングポリシー (C-look など) やページングポリシー (LRU など) といった OS が採用している資源管理ポリシーや犠牲ページリストといった OS の詳細な内部情報を提供する．これらを利用することで，ユーザレベル上での資源管理ポリシー変更機構の実装を促す．また，icTCP [34] では，TCP の輻輳制御ポリシーの変更をユーザレベルで行うために，輻輳ウィンドウサイズといった情報をユーザレベルに提供する．マイクロカーネルや拡張可能 OS といったアプローチと比べると，カーネルの修正ははるかに少量であるが，近年の OS はバージョンアップが頻繁に起こるため，OS ベンダのサポートなしでこれらの技術を利用することが困難である．また，カーネルの修正を要するので，カーネルのソースコードが入手できない OS への適用は困難である．

2.2 ユーザレベルでの資源管理ポリシー変更手法

ユーザレベルで OS の内部挙動を把握するために，OS が明示的に提供する内部情報や統計情報，システムコールの返り値やその引数を利用することができる．これらの情報を用いることで，資源管理ポリシーを変更する手法がある．また，ユーザレベルで様々なサービスを構築するために，OS の内部状態をより多く取得する

ための手法，GrayBox 手法が提案されている [35]．これは，OS の知識を利用して，ある操作に対する OS の反応を見てその内部状態を知る手法である．この手法を用いたポリシーの導入手法もある．

2.2.1 OS による内部情報提供の利用

OS が自身の内部情報をユーザレベルに提供することで，資源管理ポリシーの導入以外にも，様々なサービスがユーザレベル上で実現可能であることが知られている [45, 46]．これにより，近年の OS は自身の内部情報をユーザレベルに提供しているものが少なくなく，多くの汎用 OS 上で OS の内部情報が取得できるようになっている [47, 48, 49, 50, 51]．

Linux では，プロセスファイルシステムと呼ばれる，システムの状態を示したファイルを保有するファイルシステムを提供している．プロセスファイルシステムでは，各プロセスの詳細な情報やシステムの統計情報を取得することができる．たとえば，現在稼働しているプロセスの名前や権限，使用した CPU 時間，仮想メモリサイズ，稼働時間等を取得することができる．また，システムの統計情報として，メモリの使用量やスワップされたメモリ量，マウント情報等を取得できる．こうした情報を定期的に取得することで，OS がどのように動作してきたかを把握することができる．たとえば，各プロセスの CPU 使用時間を監視することで，どのプロセスが OS によってスケジュールされたかを知ることができる．各プロセスの CPU 使用時間を記録し，差をとることで，ある期間に実行されたプロセスを把握することができる．さらには，ある期間に使用した CPU 時間を知ることができる．

また，システムコールの引数や返り値を監視することで，プロセスがどのように資源を使用するかを知ることができる．プロセスの計算機資源へのアクセスは，OS が提供するシステムコールを経由して行われるからである．たとえば，Linux 上では，`send()` や `recv()` といったネットワーク I/O に関するシステムコールの引数や返り値を監視することで，プロセスのネットワーク帯域使用量を把握することができる．ネットワークに送信されるメッセージサイズは `send()` で指定するバッファサイズとほぼ連動しており，この値を基に，ネットワークメッセージの送信量を計算できる．また，メッセージの受信量は `recv()` の返り値から計算できるので，両者を足し合わせることによって，ネットワーク帯域使用量を把握することができる．

本研究においても OS が提供する情報を利用することで，資源管理ポリシーの導入を行っている．実際，4 章で述べるように，ディスク帯域を制限する際にも，`read()` や `write()` といったシステムコールの引数や，OS がディスクにデータブロックを書き込んだ総量といった OS が提供する内部情報を利用している．

2.2.2 OS が提供する内部情報を用いたアプローチ

ユーザレベルにおいて，OS が明示的に提供する内部状態を利用して，資源管理ポリシーの擬似的な導入を行っている研究がある．

rate-windows [52] は監視下のプロセスが発行する I/O を制限する機構である．提案手法は，ローダブルカーネルモジュールとして実現されているが，システムコールをフックすることでユーザレベル上で容易に実装することができる．rate-windows では，プロセスの発行したディスク I/O サイズを把握するために，システム全体の統計情報を用いる．具体的には，システムが稼働してから発行した全ディスク I/O サイズと全ファイル I/O サイズとの比率を計算する．これらの情報はユーザレベルで取得可能であり，この比率を用いてプロセスのファイル I/O サイズから発行され得るディスク I/O サイズを計算する．そのため，各プロセスが発行したディスク I/O サイズを取得することは実際には困難である．たとえば，ディスク I/O の発行率が極端に異なる 2 つのプロセスが稼働すると，算出される各プロセスのディスク I/O サイズは実際とは異なってしまう．そのため，監視下のプロセスが発行するディスク I/O の制限を行うことは難しい．

Newhouse らは，カーネルを修正することなく CPU プロポーショナルスケジューラを疑似的に導入する手法を提案している [36, 37]．OS が提供する情報から，監視下のプロセスの CPU 使用率を取得する．そして，あらかじめ設定した使用率に従って，プロセスの停止，再開を行う．提案機構は FreeBSD 上で実装されており，プロセスファイルシステムから CPU 使用率を取得し，シグナルを用いてプロセスの停止，再開を実現している．

MS Manners は，フォアグラウンドタスクのスループットへの影響が少なくなるように，バックグラウンドタスクを動作させるスケジューリングポリシーを擬似的に導入する機構である [38]．MS Manners はユーザレベル上で動作し，バックグラウンドタスクの進捗に基づいてスケジューリングを行う．あらかじめバックグラウンドタスクの動作を把握しておき，バックグラウンドタスクは自身の進捗を MS Manners に通知させる．もし進捗が低下していると，MS Manners はバックグ

ランドタスクがフォアグランドタスクと資源を競合しているとみなし，バックグランドタスクを停止する．これにより，資源競合によるフォアグラウンドタスクのスループット劣化を緩和することができる．MS Manners はバックグランドタスクを実行するか停止するかのいずれかをし，バックグラウンドタスクに任意の資源量を与えることはしない．

Abe らは MS Manners と同様，バックグラウンドタスクがもたらすフォアグラウンドタスクのスループットへの影響を少なくするスケジューリングポリシーを擬似的に導入する機構を提案している [39]．提案機構では，バックグラウンドタスクとフォアグランドタスクとが引き起こす資源競合をユーザレベルで検出し，バックグラウンドタスクをユーザレベルで制御する．OS のプロファイリングツール [50] を利用して，CPU 時間やディスクブロック読み込み量，ネットワークメッセージの送信量を監視し，バックグラウンドタスクの資源使用率を算出する．資源使用率が閾値を超えるとフォアグランドタスクと資源競合をしていると見なし，バックグラウンドタスクを停止する．この機構は，MS Manners と同様，任意の資源使用量をバックグラウンドタスクに与えることはしない．

Chang らはユーザレベルで動作するサンドボックスを Windows 上で実現している [40]．サンドボックスとは資源使用が制限された実行環境であり，サンドボックス上で動作するアプリケーションは，あらかじめ設定された量の資源しか使用することができない．提案手法では，監視下のアプリケーションの CPU 時間，メモリ使用量，ネットワーク帯域の制限を行う．システム API を用いて監視下のアプリケーションの CPU 時間、メモリ使用量を取得する．ネットワーク帯域使用量は socket API をフックすることで取得する．あらかじめ与えた閾値を超えると，アプリケーションの実行を停止する．

Entropia [53] はユーザレベルでプロセスの資源使用を制限する Windows 上で稼働するシステムである．Entropia は CPU 時間やメモリ使用量の制限を行っている．また，監視下のプロセスが使用可能なディスク容量の制限もしている．これらの制限の実現方法について詳細は述べられていないが，ディスク帯域の制限は行っていない．

OS が提供するロウパケットを送信するシステムコールを利用して，ユーザレベル上で TCP のプロトコルスタックを実現する手法がある [54, 55, 56]．これを用いることで，OS の内部情報を用いることなく TCP の輻輳制御ポリシーといった資源管理ポリシーをユーザレベル上で導入することができる．

これらの手法は，システムコールや OS によって提供される情報を利用すること

で実現可能な手法である．そのため，これらの手法の適用範囲は，ユーザレベルで明示的に取得できる情報に依存しており，資源管理ポリシーを変更するために十分な情報が取得できない場合，所望のポリシーの導入は難しい．そのため，たとえば，これらの手法ではディスク帯域割り当てポリシーの変更は困難である．なぜなら，ディスク帯域の使用量はユーザレベルで取得できない情報であるためである．また，ユーザレベルで動作するため，実現した機構がシステムコールといったOS 固有のインターフェースに依存してしまう．そのため，VMM 上で動作するすべての OS に原理上適用可能な FoxyTechnique と比べると移植性が低いといえる．

2.2.3 GrayBox 手法

GrayBox 手法は Arpaci-Dusseau らによって提案された手法である [35]．GrayBox 手法は，OS がユーザレベルに提供していない内部状態をユーザレベルから取得可能にする手法である．この手法では，OS のソースコードや内部動作が記されたドキュメント，ならびに LRU といったよく知られた資源管理の実装方法に関する OS の知識を利用する．これにより，OS を完全なブラックボックスとしてではなくグレーボックスとして扱う．この知識を利用しながら，ある操作に対する OS の反応を観察することで，OS の内部状態がどのようになっているかを知ることができる．Linux といったオープンソースの OS は，ソースコードを入手できるので，グレーボックスとして扱いやすい．

Arpaci-Dusseau らは，GrayBox 手法の有用性を示すために，バッファキャッシュの状態を取得する機構を実装している．この機構はあるファイルブロックがバッファキャッシュ内に存在するか否かを判断する．具体的には，ファイルを 1 バイト読み込み，その速度を観察する．読み込み速度が速ければ，ファイルはメモリ上にあると見なし，バッファキャッシュ内に存在すると判定する．遅ければ，ファイルのデータはディスクからメモリにロードされたと見なして，バッファキャッシュ内にはないと判断する．この機構を利用することで，`grep` のパフォーマンスを向上させている．複数のファイルを処理するとき，改良された `grep` はまず各ファイルがバッファキャッシュに存在するか否かを判断する．そして，バッファキャッシュ内に存在するファイルから処理を行う．こうすることで，`grep` の実行時間を約 3 倍短縮することが示されている．他にも GrayBox 手法の有用性は示されており，i-node 番号を利用してファイルのディスクレイアウトを検出する機構 [42] や，OS の採用しているバッファキャッシュ管理アルゴリズムを自動で検出する機構な

どが提案されている [57].

DiscNice もまた，ディスク帯域を制限するために，読み込むファイルがバッファキャッシュ内に存在するか否かを上述の手法を応用して判定している．

2.2.4 GrayBox 手法を用いたアプローチ

前節で述べたように，GrayBox 手法を用いることで OS が明示的に提供していない内部状態を取得することができる．この手法を用いることで，ユーザレベルで取得できる OS の内部状態が増えることで，擬似的な導入が困難であったポリシーをユーザレベルで導入することができる．

Arpaci-Dusseau らは，バッファキャッシュの状態の取得の他に，ディスク上にファイルがどのように配置されているかを検出する手法を提案している [35]．Berkely Fast File System (FFS) では，ファイルの i ノード番号とそのファイルのディスク上での位置に相関がある．これを利用して，同じディレクトリ内においてサイズの小さなファイルを大量にスキャンする場合，i ノード番号の小さなファイルから読み込むことで，ディスクへ効率よくアクセスできることを示している．これにより，ウィルススキャンといった，多くのファイルにアクセスするアプリケーションのパフォーマンスを向上することができる．

Dust [57] はユーザレベルから OS の採用しているバッファキャッシュ管理ポリシーを検出する機構である．サイズの大きなファイルに対して，シーケンシャルな読み込みや同じ箇所を何度も読み込んだりといった様々なファイル読み込みを行う．その際に，ファイルの読み込み時間を計測しておく．即座に読み込みが完了すると，読み込んだ部分はバッファキャッシュ内に存在するとみなす．その逆は，データがディスクにあるとみなす．これにより，OS がどのようにバッファキャッシュを管理しているかを把握することができる．また，著者らは Dust を用いて検出した OS のバッファキャッシュ管理ポリシーを考慮した Web サーバを実装した．バッファキャッシュ内に存在するデータが必要なリクエストから処理を行うことで，平均応答時間と平均スループットを向上できることを示している．

PLACE [42] はファイルやディレクトリのディスク上の配置をユーザレベルで操作する機構である．FFS やその後継である ext2 では，同一のディレクトリ内のファイルデータは同じシリンダグループ (ext2 ではブロックグループ) として，ディスク上において近い領域にデータを配置する．これを利用して，あるディレクトリ下でファイルを生成して `rename()` を行えば，様々なディレクトリ間にまたがる

ファイルを同じシリンダグループに所属させることができる．これにより，あるアプリケーションのアクセスするファイルを同じシリンダグループに所属させることで，アプリケーションはディスクへ効率よくアクセスすることができる．実際に PLACE を Web サーバに組み込むことで，Web サーバのスループットが上がることを示されている．

GrayBox 手法を活用することで，バッファキャッシュの状態を考慮した処理やディスク上のレイアウト操作が可能となる．しかしながら，GrayBox 手法はその有用性にも関わらず，その適用範囲は限定的である．そのため，ディスク帯域の割り当てポリシーや TCP の輻輳制御ポリシーを変更することは依然として困難である．なぜなら，既存の GrayBox 手法を用いても，ディスク帯域割り当てや TCP の輻輳制御ポリシーを変更するために必要な内部挙動を取得することが難しいためである．たとえば，既存手法ではファイルブロックの先読みサイズを取得することができない．ディスクアクセスに関する OS の挙動を十分に把握できないと，ディスク帯域を適切に制御することができない (4.4.2 節).

2.3 仮想化技術を利用した手法

仮想化環境では，仮想マシンモニタ (VMM) と呼ばれるソフトウェアが OS と物理ハードウェアとの間に介在する．VMM が物理ハードウェアや OS の実行を制御するので，VMM を活用することで，その上で動作する OS に対して新たな機能の恩恵をもたらす手法が提案されている．

2.3.1 仮想化技術

今日，仮想化技術はサーバやデスクトップ PC 上に広く普及している [58, 59, 60, 61]．仮想化技術は，1 台のマシン上で複数の OS (ゲスト OS) を同時に稼働させる環境を提供する．仮想化技術の発達に伴って，仮想化技術を提供するソフトウェア [62, 63, 64] も充実し，データセンタにおけるサービス可用性の向上 [65, 66, 67] や動的な資源の割り当て [68, 69]，セキュリティモニタ [70, 71, 72, 73, 74, 75]，フォールトトレランス [76]，大規模分散テストベッドの構築 [77]，OS の動的なアップデート [78] などを実現することができる．また，CPU によって仮想化支援機能 [58, 60] も提供され，これらを用いた VMM の開発も積極的に行われており [79]，仮想化技術はさらなる広がりを見せている．

仮想化環境で稼働しているゲスト OS は仮想マシン (VM) 内で動作する．各仮想マシンには CPU やメモリ，ディスクといった計算機資源が与えれている．これらを p 使用してゲスト OS は実ハードウェア上で動作するかのように稼働する．VM に与えられる仮想的な CPU やデバイスをそれぞれ仮想 CPU，仮想デバイスと呼ぶ．これらをまとめて仮想ハードウェアとも呼ぶ．また，各仮想マシンはそれぞれが明確に分離しており，ある VM 内で OS がクラッシュしたとしても，他の VM へ影響を及ぼすことはない．VM への計算機資源の提供や各 VM の管理は仮想マシンモニタ (VMM) と呼ばれるソフトウェアが行っている．VMM は実ハードウェア上で動作し，実計算機資源を時間的，空間的に分割して仮想マシンに与える．つまり，VMM が実計算機資源管理を行っている．

仮想化環境では，VMM は特権モードで，ゲスト OS は非特権モードで動作する．ゲスト OS が特権命令を発行すると，制御が VMM に移り，発行した特権命令を適切にエミュレートする．これにより，あるゲスト OS が実 CPU 上で特権命令を実行することがなく，他のゲスト OS の動作を阻害することはない．また，ゲスト OS がメモリやデバイスといった資源へアクセスした際も，制御が VMM に移り，各ゲスト OS に割り当てた資源へのアクセスを VMM が行う．このように，ゲスト OS の計算機資源へのアクセスは VMM を仲介するので，ゲスト OS が仮想デバイスや仮想 CPU に要求したリクエストをすべて取得することができる．

VMM 上にセキュリティ機能や効率のよい資源管理機構といった従来 OS が提供してきたサービスを実現すると，次のようなメリットが生じる [80, 81, 82]．まず，VMM 上で動作する様々なゲスト OS に対して，VMM 内で実現したサービスを適用できる点である．VMM 上で 1 度だけ実装を行うだけで，その上で動作するゲスト OS はそのサービスの恩恵を受けることができる．また，カーネルの修正ができない OS に対しても，新たなサービスを導入したときに近い効果を得ることができる．

また，VMM は OS と物理ハードウェアとの間に介在するソフトウェアなので，ユーザレベルとは異なる情報を取得することができる．具体的には，OS によって抽象化されていない情報を取得できる．たとえば， ユーザレベルではディスクのデータブロックはファイルという形で抽象化されているが，VMM ではそのデータブロックの読み書きを直接監視することができる．また，ユーザレベルではネットワーク上のデータパケットはメッセージとして抽象化されているが，VMM ではデータパケットの送受信を直接監視することができる．データブロックやデータパケットの詳細な情報の取得はユーザレベルでは困難である．

VMM ではユーザレベルでは扱えない情報を取得できる反面，OS の抽象化を理解できないという点もある．たとえば，VMM からは“プロセス”や“ファイル”といった OS がユーザレベルに提供する抽象化を取得することができない．VMM と OS との間の情報のギャップはセマンティックギャップ [82] と呼ばれ，広く知られている．資源管理ポリシーによってはこれらの抽象化を利用するものもある．擬似的にポリシーを導入する際は，適宜利用するソフトウェアレイヤを選択する必要がある．また，最近はセマンティックギャップを埋める手法も提案されてきており，これらを利用するもの有効な手段である [80, 81].

2.3.2 VMM を活用したアプローチ

仮想化技術の発展に伴い，VMM 上で効率のよい資源管理機構を実現することで，ゲスト OS にその恩恵を与える手法が提案されている．

Jones らは，OS-VMM 間のセマンティックギャップを埋める手法として Antfarm と Geiger をいう機構を提案している [80, 81]．これらは，OS と仮想ハードウェアとのやりとりを VMM 内で監視し，それぞれ“プロセス”，“バッファキャッシュ”といった OS レベルの抽象化を VMM 内で認識可能にする．Antfarm では，CPU 上のページテーブルの切り替えを VMM 上で監視することで，ゲスト OS 内のプロセスの生成，終了，コンテキストスイッチを把握する．たとえば，プロセスのコンテキストスイッチをゲスト OS が行うときには，プロセスの仮想アドレス空間を変更するため，使用するページテーブルを切り替える．ページテーブルの切り替え要求は特権命令であるため，VMM からページテーブルの切り替えは全て監視可能である．ページテーブルの切り替えをゲスト OS によるコンテキストスイッチとみなすことで，VMM からゲスト OS がコンテキストスイッチを行ったことを把握できる．Geiger では，ディスクの読み込みリクエストがゲスト OS から発行されると，VMM はそのリクエストを処理した後，要求されたデータブロックはゲスト OS のバッファキャッシュ内に存在するとみなす．また，ゲスト OS がそのデータを読み込むために使ったページが以前に使っていたものであるとすると，そのページにあったデータブロックはバッファキャッシュから破棄されたとみなす．Jones らは提案手法を用いて，I/O スケジューラやバッファキャッシュ管理機構，プロセス監視機構 [83] を VMM 内に実装し，その有用性を示している．これらの手法は仮想デバイスの振る舞いは変更しておらず，FoxyTechnique では，OS-VMM のやりとりの監視とともに，OS の動作を推測しながら仮想デバイスの振る舞いを

変更することで，VMM 上で資源管理ポリシーを導入している．また，Antfarm と Geiger は FoxyTechnique と併用可能である．

Lu らは，ゲスト OS のバッファキャッシュ管理を VMM と協調させることでシステム全体のスループットを向上させる手法を提案している [84]．Geiger と同じ手法で，ゲスト OS によってキャッシュされているファイルブロックを把握し，ゲスト OS が破棄したバッファキャッシュの内容を VMM 内でいくつか保持すること [85] で，ディスクアクセスの回数を減らしている．提案手法は，システム全体のスループットの向上を目的としており，ゲスト OS の資源管理ポリシーの変更は行わない．

Balloon driver [86] では，ゲスト OS に割り当てるメモリ量を動的に変更するカーネルモジュールである．メモリ量を調整するために，Balloon driver は， VMM と通信を行い，指定されたメモリ量を確保する．この際，確保するメモリ領域はスワップアウトされないよう設定を行うことで，ゲスト OS は実際に使用可能なメモリ領域が小さくなったかのように振る舞う．Balloon process driver [87] は，複数の仮想 CPU を持つゲスト OS の CPU スケジューラの振る舞いを変更するカーネルモジュールである．各仮想 CPU に割り当てられる CPU 時間を考慮し，ゲスト OS の CPU スケジューラキューの長さを制御する．これらの手法の目的はシステム全体のスループットを向上させることであり，本研究の目的とは異なる．また，これらの手法を利用する際には，各 OS ごとにカーネルモジュールを作成する必要がある．

ソースコードを変更することなく OS の動作を変更する手法として IntroVirt [88] がある．IntroVirt はカーネルのソースコードを変更せずにセキュリティパッチを適用する手法を提案している．カーネルのソースコードにブレークポイントを埋め込み，VMM に制御を移し，VMM が OS の代わりにパッチを適用したときと同等の振る舞いを行う．そのため，IntroVirt を使用するには，予めソースコードにブレークポイントを挿入する必要がある．そのため，商用 OS やソースコードの入手が困難な OS への適用は難しい．

システム全体のスループットを上げるために，ゲスト OS と連動することなく計算機資源を効率よく管理する手法がある．Disco [89] や Potemkin [68] では，メモリを効率よく利用するために，Copy-on-write 技術を用いて複数のゲスト OS 間で同一の内容のページを共有する．Difference Engine [90] では，ページ内容が同一のものだけではなく，ページ内容が酷似しているページも共有する手法を提案している．これらはシステム全体のスループットを向上させることを目的として

おり，ゲスト OS の資源管理ポリシーの変更は行わない．

仮想デバイスの振る舞いを調整することで OS の動作を制御する手法は，資源管理ポリシーの変更だけでなく，他の目的にも利用されている．Time dilating [91] や DieCast [92] では VM に通知するタイマ割り込みのペースを遅くすることで，ゲスト OS の時間の流れを遅くする．これにより，高性能な NIC に必要な CPU 処理速度の見積もりや分散システムの動作検証を可能にする．Crandall らは，タイマの振る舞いを変更することで，悪意のあるプログラムの時間に依存した振る舞いを解析する手法を提案している [93]．この手法では，ゲスト OS の時刻を任意に設定することで，OS 内に存在する悪意あるプログラムがいつどのように動作するかの情報を収集する．

2.4 まとめ

本章では，本研究の関連研究について述べた．資源管理ポリシーの導入を円滑に行えるようにカーネルの構成そのものを変更するアプローチがある．マイクロカーネルや拡張可能 OS，インフォカーネルでは，カーネルを修正することなく資源管理ポリシーを導入できるが，ユーザは専用カーネルの使用を課せられる．もしくは，ユーザの使用している OS カーネルの構成そのものを修正する必要がある．そこで，OS を修正，変更することなく，カーネル外から所望の資源管理ポリシーを擬似的に導入する手法が提案されている．ユーザレベルから資源管理ポリシーを擬似的に導入する手法では，OS がユーザレベルに提供する自身の内部情報やシステムコールやその引数を活用することで，プロセスの挙動を制御し，カーネル内にポリシーを導入したときに近い効果を得る．これらの情報を利用することで，CPU 時間の使用率やメモリ使用量，ネットワーク帯域幅の制御といったポリシーを擬似的に導入する手法が提案されている．しかし，従来手法では，これらのユーザレベルで取得できる情報に依存しており，システムコールと資源使用量が連動しない資源に対するポリシーや OS が明示的に内部情報を提供しないポリシーの変更は困難である．GrayBox 手法を用いることで，ユーザレベルから取得できる OS の内部状態は増え，ユーザレベル上で導入可能なポリシーの種類を増やすことができる．GrayBox 手法は，ユーザレベルのある操作に対する OS の反応を観察することで OS の内部状態を取得する手法である．この手法を用いることで，バッファキャッシュの内容を考慮したファイルプロセッシングやファイルブ

ロックのディスク上のレイアウト変更といったことが可能となる．しかしながら，GrayBox 手法の適用範囲は限定的であり，既存技術を用いても OS の内部状態を十分に取得できない場合がある．これにより，導入可能なポリシーは依然として限定的である．最後に，仮想化技術を利用した研究について述べた．従来の研究では，VMM が効率の良い資源管理を行うことで，ゲスト OS にその恩恵をもたらす手法が数多く提案されているが，ゲスト OS の資源管理ポリシーの変更を行う手法は提案されていない．次章では，本研究で着目した OS の推測について述べる．これにより，ユーザレベル，VMM レベルで扱える OS の内部情報をさらに増やすことができ，これまで導入が困難であったディスク帯域割り当てポリシーや TCP の輻輳制御ポリシーの変更を可能にする．

第3章　カーネル外によるOSの動作推測

カーネル外から擬似的に導入できる資源管理ポリシーの範囲の拡大を行うために，GrayBox 手法の概念を活用し，提案手法では OS の動作を推測することに着目する．OS の動作を推測することで，これまで把握が困難であった OS の内部挙動をカーネル外から把握可能にする．本章では，カーネル外から OS の内部挙動を推測する手法について述べる．まず，ユーザレベルにおける OS の推測に基づいた内部挙動の把握手法について述べる．次に，VMM レベルにおける OS の推測に基づいた内部挙動の把握について述べる．

3.1　ユーザレベルでの OS の動作推測

本研究では，前章で述べた，OS から取得できる情報や GrayBox 手法によって取得できる情報に加えて，OS の動作の推測をユーザレベルで行う．これは，GrayBox 手法と同様，OS に関する知識を利用する．具体的には，あるユーザレベルの操作に対して OS がどのように動作するかに関する知識を用いる．知識を用いることで，ある操作に対して， OS の反応を観察することなく，OS の内部動作を把握することができる．これにより，ユーザレベルで明示的に反応を観察することができない挙動も，ユーザレベルで把握可能となる．

たとえば，ユーザレベルで OS の動作を推測することで，OS のファイル先読みの挙動を把握することができる．ファイル先読みとは，OS がプロセスに要求されたファイルブロックをディスクからロードする際に，要求されたデータブロックだけではなく，その先にプロセスが読むであろうデータブロックも同時にロードする OS の機能である．これにより，ディスクのシーク時間を減らし，システムのスループットを向上することができる．これまで，OS によるファイル先読みの動作はユーザレベルから把握することは困難であった．なぜなら，ファイル先読みの動作に関する情報をユーザレベルで取得することができないからである．そ

こで，OS のドキュメントやソースコードから OS が採用しているファイル先読みアルゴリズムに関する知識を十分に得る．そして，ユーザレベル上で，ファイル読み込みを監視しながらファイル先読みアルゴリズムをエミュレートすることで，OS が行うファイル先読みの挙動を把握することができる．

実際，提案機構である DiscNice は OS によるファイル先読みの動作を把握している．上述のように，OS の採用している先読みアルゴリズムをエミュレートし，監視下にあるプロセスが発行したファイル読み込みに対して，OS がどれだけデータブロックを先読みするかを計算し，実際に生じるディスク I/O サイズを推測している．

3.2 VMM レベルでの OS の動作推測

VMM で資源管理ポリシーの擬似的な導入を行うために，VMM レベルでゲスト OS の動作を推測する．3.1 節で述べた手法と同様，OS の知識を用いる．具体的には，あるデバイスの振る舞いに対する OS の反応に関する知識を用いる．この知識を用いることで，デバイスがデータや割り込みを通知した直後の OS の動作を推測することが可能になる．さらに，もしデバイスの挙動を変化させると，OS がどのような動作をするかまで推測することができる．これにより，VMM によるデバイス操作直後の OS の内部挙動を把握することができる．

たとえば，OS のタイマ割り込みに関する知識を利用することで，プロセスのタイムスライスの挙動を推測することができる．タイマは VMM によって仮想化されているデバイスであるため，VMM がゲスト OS にタイマ割り込みを通知する．ここで，OS はタイマ割り込みをカウントすることによってプロセスのタイムスライスを計測しているという知識を利用する．もし VMM がゲスト OS に通知したタイマ割り込みを通知すると，VMM は現在実行しているプロセスのタイムスライスの残り時間が減少すると推測することができる．さらに，もしタイマ割り込みを通知しないと，プロセスのタイムスライスは減少しないためそのプロセスは動作し続ける．また，タイマ割り込みをより多く通知すると，タイムスライスの減少は早くなると推測することができる．

また，データブロックの読み込み完了通知に対する OS の知識を利用することで，データブロックを要求したプロセスの挙動を推測することができる．OS は要求されたデータブロックの読み込みが完了するまで，その要求を発行したプロセ

スをブロックする．もし VMM がデータブロックの読み込み完了通知をゲスト OS に送信すると，要求を発行して停止していたプロセスがレディ状態となる．さらに，もし読み込み完了通知を送信しないと，プロセスは停止し続けると推測することができる．

提案機構である FoxyIdle はデータブロックの読み込み完了通知に対する OS の知識を利用している．この詳細については 5 章で述べる．

3.3 議論

ここで，明示的に取得できる情報に加えて，OS の内部挙動を推測することで導入可能となる資源管理ポリシーの種類について議論をする．擬似的に導入可能となる資源管理ポリシーの種類を明示することは難しい．なぜなら，どのような OS の挙動をどのように推測するかの方法は多々あると考えられるからである．これによって適用可能なポリシーの種類は変化する．

また，OS が持つ元々の資源管理ポリシーと新たに導入するポリシーとが競合する可能性がある．たとえば，ゲスト OS がアイドル時間スケジューリングを行っている環境に，VMM を用いてデッドラインスケジューリングを導入することは難しい．なぜなら，バックグラウンドプロセスの要求に VMM 上でデッドラインを与えても，ゲスト OS がいつまでもその要求を仮想ディスクに発行しない可能性があるからである．そのため，デッドラインを守って対象の要求を発行することができない．

以上の理由より，OS を推測することによって導入可能となる資源管理ポリシーの種類を明言することは難しい．本研究では，資源管理ポリシーの適用範囲の明確化に向けての最初のステップとして，DiscNice，FoxyVegas，FoxyIdle という 3 つの機構を用いた資源管理ポリシーの擬似的な導入を示す．

3.4 まとめ

本章では，本研究で着目したカーネル外からの OS の動作推測について述べた．GrayBox 手法の概念を活用し，OS の知識を利用することで，ある操作に対する OS の動作を推測することができる．これにより，ある時点での OS の内部状態を推測することができ，従来よりもより多くの OS の内部挙動が把握可能となる．本

研究では，ユーザレベル，そして VMM レベルという 2 種類のソフトウェアレイヤから OS の動作推測を行っている．ユーザレベルでは，プロセスのある操作に対しての OS の動作を推測することで，これまで取得が困難であったファイルブロックの先読みサイズといった内部挙動が把握可能となる．一方，VMM レベルでは，あるデバイスの操作に対する OS の動作をあらかじめ推測する．これにより，デバイスによるデータの通知や割り込み通知後の OS の内部挙動を VMM 上で把握することができる．次章より，これらの OS の内部情報の取得方法を用いた資源管理ポリシーの変更について述べていく．具体的には，ユーザレベルで動作するディスク帯域割り当てポリシーの変更機構，さらに VMM レベルでの資源管理ポリシーの変更手法について具体的に述べていく．

第4章　ユーザレベルでのディスク帯域割り当て変更

本章では，ユーザレベルでプロセスのディスク帯域を制限する機構，*DiscNice* について述べる．ディスク I/O は CPU 速度やメモリのアクセス速度と比べて格段に低速である．そのため，プロセスのディスク帯域の割り当て方を用途に応じて制御しなければ，アプリケーションのスループットは著しく低下してしまう．各プロセスのディスク帯域割り当て変更の必要性にもかかわらず，Windows や Linux といった汎用 OS にはプロセスごとにディスク帯域を制限する資源管理ポリシーが導入されていない．また，これまでユーザレベルでディスク帯域の割り当てを変更する機構は提案されていない．この理由はユーザレベルからプロセスのファイル I/O に伴うディスク I/O を観測することができないためであると考えられる．たとえプロセスが OS に対して発行するファイル I/O を制限しても，ファイル I/O はディスク I/O と連動しないため，ディスク帯域使用量を適切に制限することができない．そこで，DiscNice ではディスクアクセスに関する OS の動作に着目する．ファイル I/O が発行された際，OS の動作を推測することでそのファイル I/O に伴って発行されるディスク I/O サイズを推測する．推測したサイズから，各プロセスのディスク帯域使用量の把握を行う．そして，DiscNice は搭載されたポリシーに従ってファイル I/O の発行率を調整することで，ディスク帯域使用量を制限する．これにより，各プロセスのディスク帯域を制限するようカーネルを修正したときに近い効果を得る．DiscNice は以下の特徴をもつ．

- **OS カーネルの変更が不要:** DiscNice は既存の OS カーネルに修正を加えずに導入することが可能である．DiscNice の各モジュールは，カーネルの修正やカーネルモジュールのロードを必要とせず，すべてユーザレベルで動作する．カーネルの修正を要さないため，商用 OS の Windows であってもディスク帯域の制限が原理的に可能である．

- **ディスク帯域制御機構の移植性向上:** DiscNice のすべてのモジュールをユー

ザレベル上で実装しており，また OS 固有の機能を用いていない．そのため，システムコール等のインターフェースの変更は要するものの，カーネルレベルのディスク帯域制御機構と比べると，他の OS への移植が容易である．

- **ポリシーと機構との明確な分離:** DiscNice では，プロセスのディスク I/O サイズを推測する方法 (機構) と，ディスク I/O を制限するために行うファイル I/O の発行制限の仕方 (ポリシー) を明確に分離している．これにより，ポリシーを変更する際はポリシーのみの実装を変更すればよく，ディスク帯域使用量の制限の仕方を用途に応じて設定できる．

プロトタイプを Linux 2.4.27 上に実装した．そして，実際のアプリケーションを含むベンチマークを用意して実験したところ，DiscNice はディスク帯域使用量を制限可能な精度でディスク I/O サイズを推測できることがわかった．また，各ベンチマークディスク帯域使用量をポリシーに従い制限し，バックグラウンドタスクが起こすディスク競合を緩和することで，フォアグランドタスクのスループット劣化を抑えられることがわかった．加えて，DiscNice によるオーバヘッドを計測するために，実際のアプリケーションを用いて，DiscNice を稼働させたときとそうでないときとの実行時間を比較した．すると， 実行時間の増加は 12% 以下にとどまった．

本章では，まずディスク帯域制限の必要性について示し，ユーザレベル上でのディスク帯域制限の困難さについて述べる．次に考慮すべき OS の動作について述べ，DiscNice の設計，実装および実験について述べる．そして，最後に本章をまとめる．

4.1 ディスク帯域制限の必要性と困難さ

4.1.1 必要性

ディスクへのアクセス時間は CPU 速度やメモリへのアクセス速度に比べて格段に低速である．そのため，ディスクアクセスを適切に制御しなければ，アプリケーションのスループットは著しく低下してしまう．例えば，バックグランドタスクがフォアグランドタスクと同時にディスクアクセスを行うと，フォアグランドタスクのスループットは著しく低下する．また，仮想ホスティング環境上で複数ホ

表 4.1: **Performance degradation of `Sequential` caused by disk I/O contention with `Rsync`.** This table reports the execution times of `Sequential` without `Rsync`, with `Rsync`, and with low priority `Rsync`.

Benchmark	Execution time [sec]	Increasing rate
w/o `Rsync` [sec]	4.18	—
w/- `Rsync` [sec]	17.49	418%
w/- low priority `Rsync`	17.26	413%

ストが同時にディスクアクセスを行うと，ユーザとの契約に応じたサービスを提供できなくなってしまう可能性がある．

ディスク帯域の競合がアプリケーションへ及ぼすの影響が大きいことを示すために，バックグラウンドタスクとフォアグランドタスクとを同時に動作させる実験を行った．実験で用いたマシンは Pentium 4 2.8 GHz の CPU，1024 MB のメモリ，そして SCSI で接続されたハードディスクを備え，Linux 2.4.27 が稼働している．バックグランドタスクとしてファイルのバックアップを行うアプリケーション `Rsync` を用いる．`Rsync` では，600 MB のファイルのバックアップを別ホストに保存する．そして以下の 2 つのアプリケーションをフォアグランドタスクとして用いる．

- `Sequential` : 200 MB のファイルをシーケンシャルを読み込む．
- `Tar`: 50 MB のアーカイブファイルを展開する

実験ではフォアグラウンドタスクとバックグラウンドタスクの実行時間，ならびにそれぞれの資源使用量を計測した．計測は次の 3 つの条件下で行った．まず，バックグラウンドタスクを動作させることなく，フォアグラウンドタスクのみをそれぞれ動作させた．次に，フォアグラウンドタスクとバックグラウンドタスクとを同時に動作させた．この状況下ではディスク帯域の競合が起きている．最後に，CPU 優先度を下げたバックグラウンドタスクをフォアグラウンドタスクと同時に動作させた．具体的には，`Rsync` の nice 値を 19 に設定する．

フォアグラウンドタスクとバックグラウンドタスクの実行時間を表 4.1.1，4.1.1 に示す．`Rsync` と同時に動作させると，`Sequential` の実行時間は約 4.2 倍長く

表 4.2: **Performance degradation of `Tar` caused by disk I/O contention with `Rsync`.** This table reports the execution times of `Tar` without `Rsync`, with `Rsync`, and with low prirority `Rsync`.

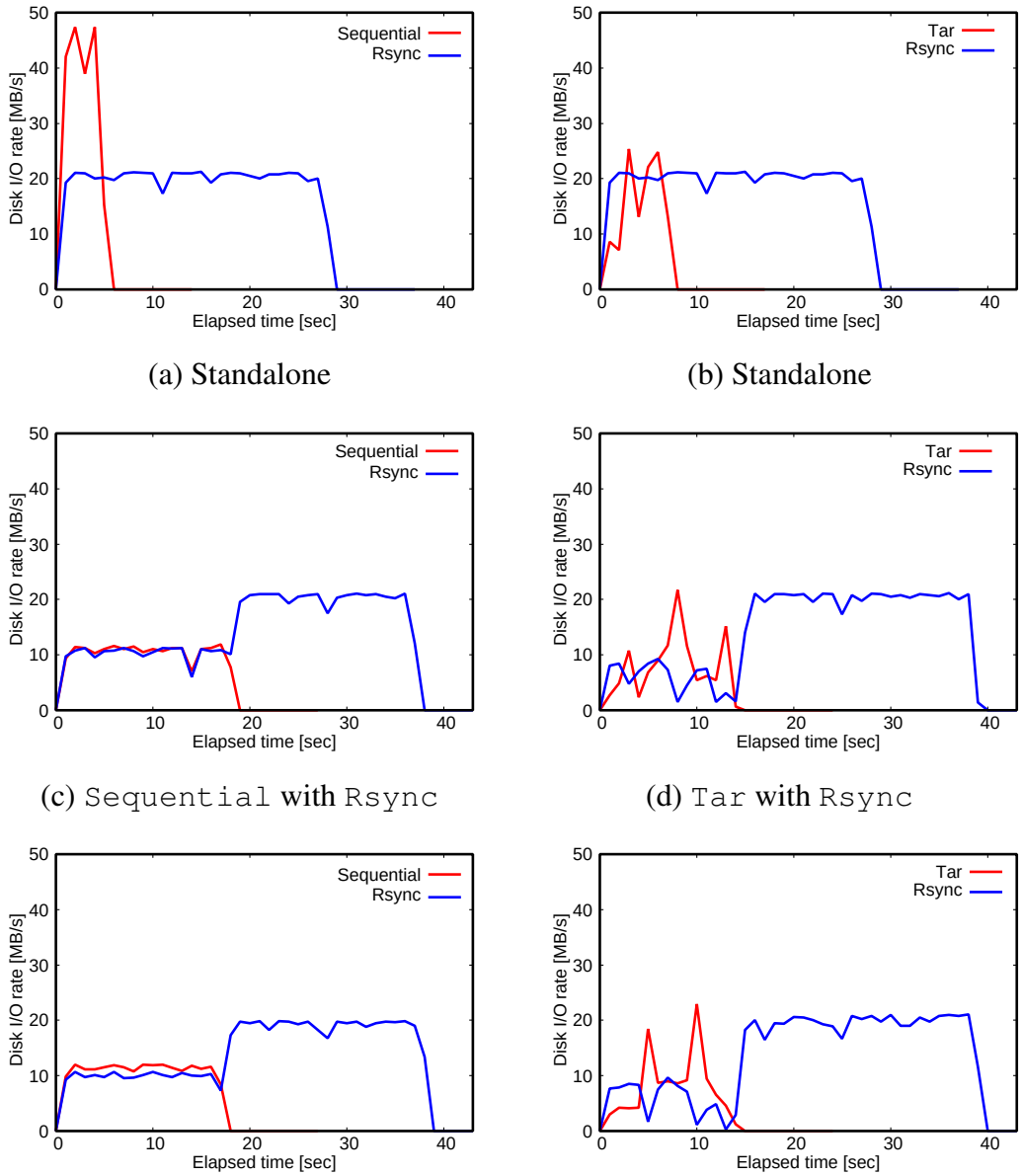
Benchmark	Execution time [sec]	Increasing rate
w/o `Rsync` [sec]	4.47	—
w/- `Rsync` [sec]	13.10	293%
w/- low priority `Rsync`	12.82	287%

なっている．また `Tar` も，`Rsync` と同時に動作させると約 2.9 倍の実行時間を要してしまう．`Rsync` の優先度を下げても，フォアグラウンドタスクのスループットは低下している．

各アプリケーションのディスク帯域幅の使用量を図 4.1 に示す．文献 [52] と同様，1 秒間の平均ディスク帯域使用量をプロットしている．一見，`Tar` の実行時間が表 4.1.1 と図 4.1 とで異なっているように見える．これは OS による遅延書き込みが原因である．遅延書き込みとはシステムのスループットを上げるために OS が行う最適化である．これについては 4.1.2 節で述べる．遅延書き込みによって，`Tar` の実行が終了した後も，`Tar` が発行したディスク書き込み要求が観測される．

図 4.1 より，`Sequential` と `Tar` が `Rsync` と同時に実行させると，ディスク帯域の競合が起きることがわかる．`Sequential` と `Tar` が単独で動作したときのディスク帯域使用量はそれぞれ約 47 MB/s，20 MB/s であるが，`Rsync` と同時実行したときには，ディスク帯域使用量が 11 MB/s，10 MB/s にまで落ち込む．フォアグラウンドタスクの実行が終了すると，`Rsync` のディスク帯域使用量が単独で動作したときの値である約 20 MB/s にもどる．図 4.1(c)，(f) より，たとえ `Rsync` の CPU の優先度を低く設定しても，ディスク帯域の競合は依然として起きていることがわかる．

CPU がボトルネックとなってディスク帯域使用量の低下が起きていないことを示すために，実験中の各アプリケーションとシステム全体の CPU 使用率を図 4.2 に示す．図より，実験中には CPU の使用率が 100% になっていないことがわかる．フォアグラウンドタスクとバックグラウンドタスクを同時に実行したときでさえ，高くても 51% の CPU 使用率である．また，`Rsync` の CPU 使用率がフォアグラウンドタスクと同時に動作させると著しく低下していることがわかる．これは，



(a) Standalone

(b) Standalone

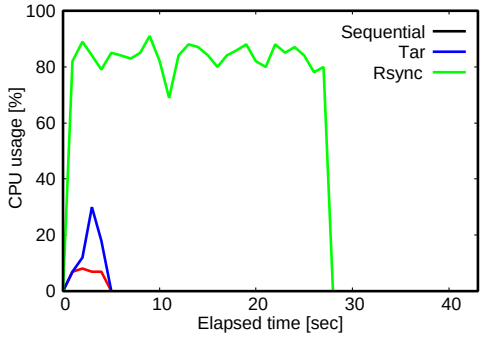
(c) `Sequential` with `Rsync`

(d) `Tar` with `Rsync`

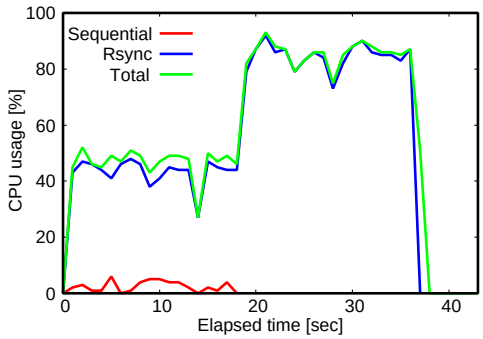
(e) `Sequential` with low priority `Rsync`

(f) `Tar` with low priority `Rsync`

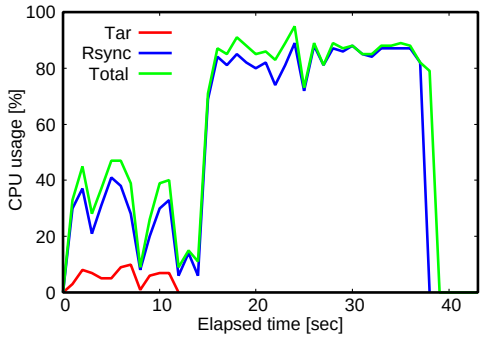
図 4.1: **Disk bandwidth usage.** These graphs show the disk bandwidth usage of `Sequential`, `Tar` (foregrounds) and `Rsync` (background) on three situations. The x-axis is the elapsed time and the y-axis is the disk I/O throughput over 1 second. The first situation is that each application ran standalone. The second is that the foregrounds were executed with the background. The last situation is that we executed the foregrounds with the background whose nice value was set to 19.



(a) Standalone



(b) `Sequential` with `Rsync`



(c) `Tar` with `Rsync`

図 4.2: **CPU usage.** The three graphs show the CPU usage of `Sequential`, `Tar` (foregrounds) and `Rsync` (background). The x-axis is the elapsed time and the y-axis is the CPU usage over 1 seconds. For comparison, we show the CPU usage of the three applications when they ran standalone.

`Sequential` や `Tar` とディスク帯域を競合するため，ディスク内のデータを取得することに時間がかかるからである．これにより，`Rsync` は単独で動作したときのような処理速度で動作できない．

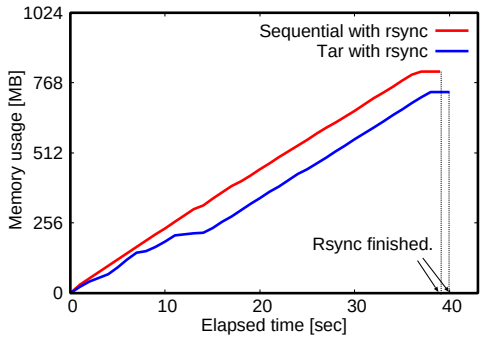


図 4.3: **Memory usage.** This figure shows the system memory usage on two situations. The x-axis is the elapsed time and the y-axis is the amount of system memory usage. The first situation is that we ran `Sequential` and `Rsync`. The other is that we ran `Tar` and `Rsync`.

図 4.3 に実験中のシステム全体のメモリ使用量を示す．図より，実験中では，メモリ使用量は全体で 1 GB 以下であることがわかる．フォアグランドタスクとバックグラウンドタスクが同時に動作したとしても，使用したメモリ量は利用可能メモリ量以下である．

4.1.2 ディスク帯域割り当て変更の困難さ

前節で述べたように，ディスク帯域競合が生じると，アプリケーションのスループットは著しく低下する．そのため，ディスク帯域割り当てを用途に応じて行う必要がある．本研究では，ユーザレベルでディスク帯域割り当てポリシーを変更する機構を提案する．ユーザレベルでディスク I/O 帯域を制御するためには，ファイル I/O とディスク I/O とを明確に区別する必要がある．

ファイル I/O とディスク I/O の違いを図示すると図 4.4 となる．ファイル I/O はプロセスによって発行されるファイルへの読み書き要求である．ここで，ファイル I/O はシステムコールの横取り等を行うことでユーザレベルで完全に監視可能である．一方，ディスク I/O は OS がディスクに対して行うデータブロックのロード，ストア操作である．ディスクアクセスを制御するためには，ファイル I/O ではなくディスク I/O を制御する必要がある．なぜなら，ファイル I/O の発行は必ずしもディスク I/O の発行を伴わないためである．たとえば，あるプロセスがファイル読み込みを行っても，ディスク読み込みが生じないことがある．また，実際に

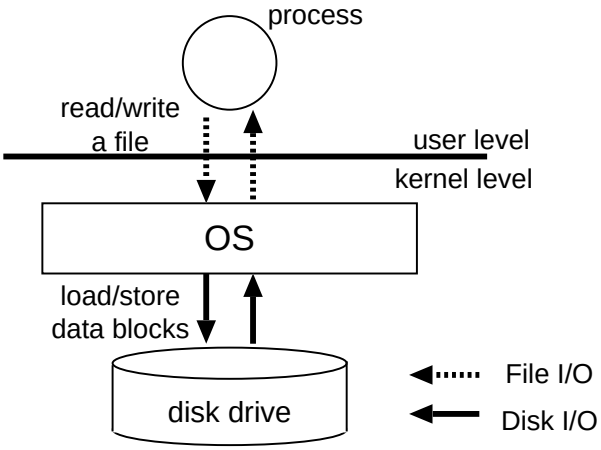


図 4.4: **Difference between file I/O and disk I/O.** This figure exhibits the difference between file I/O and disk I/O. File I/O is read/write requests processes issue to files. Disk I/O is load/store operations the OS issues to disk drives.

データブロックをロードしても，ファイル I/O サイズと同サイズのディスク I/O が生じるとは限らない．これらは以下に列挙する OS による振る舞いが原因である．

- **バッファキャッシュ:** バッファキャッシュ上に存在するファイルに対する読み込みはディスク I/O を伴わない．
- **ブロックサイズアクセス:** ディスクアクセスはデータブロック単位で行われる．そのため，ディスク I/O サイズはファイル I/O サイズと等しいとは限らない．
- **ファイル先読み:** システム全体のスループットを上げるために，OS 要求されたデータブロックと同時にいくつかのデータブロックをロードする．そのため，ディスク I/O サイズがファイル I/O サイズより大きくなることがある．
- **遅延書き込み:** OS は汚れバッファがある一定値を超えるまで，もしくは一定時間経過するまで更新のあったデータブロックをディスクに書き込まない．そのため，ファイル書き込みとディスク書き込みは非同期に行われる．

4.2 ユーザレベルでのディスク帯域割り当ての変更

ディスク I/O を制御するために，本研究ではユーザレベルでプロセスのディスク帯域使用量を制御する機構，*DiscNice* を提案する．DiscNice では，アプリケー

ションの発行するファイル I/O から実際に発生するディスク I/O サイズを推測する．もしディスク I/O が生じなかったら，ディスク I/O サイズは 0 となる．推測したディスク I/O サイズを基に，ファイル I/O を制限することで，間接的にディスク帯域使用量の調整を行う．DiscNice では，4.1.2 節で述べた OS の挙動を以下のように扱う．

- **バッファキャッシュ:** バッファキャッシュ内に存在するファイルブロックへの読み書きを識別するために，DiscNice はファイルブロックがバッファキャッシュ内に存在するか否かを判断する．
- **ブロックサイズアクセス:** ディスク I/O サイズを推測するために，DiscNice はファイル I/O サイズをデータブロックのサイズで丸め上げる．
- **ファイル先読み:** DiscNice は OS が採用しているファイル先読みアルゴリズムをエミュレートすることで，ファイル I/O から実際に起こるディスク I/O サイズを推測する．
- **遅延書き込み:** ディスク書き込みはファイル I/O とは非同期に行われる．そのため，ディスク書き込みが起きるまでに生じた同じ領域へのファイル書き込みはディスク I/O サイズに加算しない．また，ディスク書き込みが起きるまでに生成，削除されたファイルへの書き込みもディスク I/O サイズに加算しない．これらに関しては 4.2.4 節で詳しく述べる．

4.2.1 設計

ディスク帯域割り当てポリシーを変更するために，DiscNice では，1) アプリケーションのファイル I/O からディスク I/O サイズを推測し，2) ポリシーに従ってファイル I/O の発行を調整することで間接的にディスク I/O 帯域使用量の調整を行うようにする．また，用途に応じてディスク帯域割り当てポリシーの変更を容易にするために，ディスク I/O サイズを推測するメカニズムとディスク帯域使用量の制限ポリシーとを分離するようにする．これにより，ディスク I/O サイズを推測するメカニズムに手を加えることなく，制限ポリシーを用途に応じて変更可能となる．

上述の用件を満たすために，DiscNice を**プレディクタモジュール**，**ポリシーモジュール**，そして**スロットリングモジュール**の 3 つのモジュールからなるよう設計する．プレディクタモジュールは監視下のアプリケーションが発行したファイル

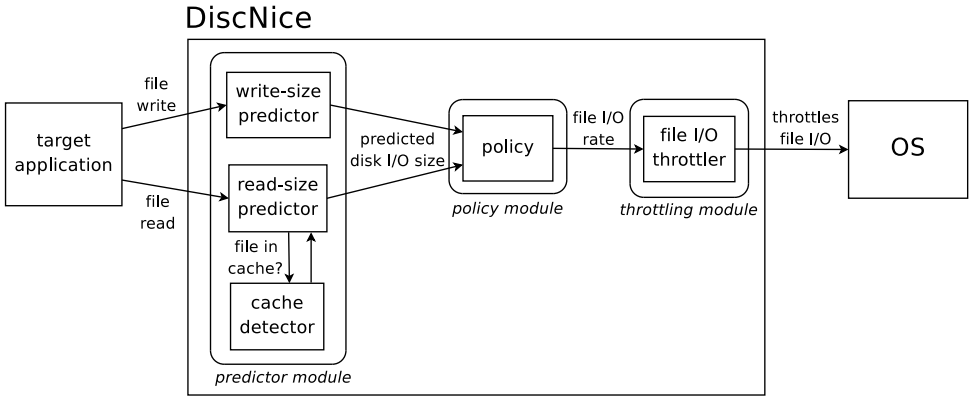


図 4.5: **Control flow in DiscNice.** This diagram shows the overview of how DiscNice behaves. DiscNice consists of three modules; *predictor module*, *policy module* and *throttling module*. Predictor module infers disk I/O size the target application requests. Predictor module also sends the disk I/O size to policy module, which decides how much file I/O should be throttled. And then, throttling module throttles file I/O.

I/O からディスク I/O サイズを推測する．プレディクタモジュールでは OS のディスクアクセスに関する振る舞いを考慮するサブモジュールである *Cache Detector*, *Read-Size Predictor*，そして *Write-Size Predictor* からなる．ポリシーモジュールは推測したディスク I/O サイズからファイル I/O の発行を制限するか否かを決定する．スロットリングモジュールはポリシーモジュールに従って実際にファイル I/O の発行を制限する．制限ポリシーを変更するには，ポリシーモジュールを差し替えるだけでよい．

DiscNice の動作の流れを図 4.5 に示す．監視下のアプリケーションがファイル I/O を発行すると，プレディクタモジュールがシステム内で発生するディスク I/O サイズを推測する．ポリシーモジュールは，プレディクタモジュールの推測したディスク I/O サイズを基に，組み込まれたポリシーに従って，アプリケーションのファイル I/O の発行を制限するか否かを決める．ポリシーモジュールはディスク I/O を制限するために，アプリケーションが発行してもよいファイル I/O サイズを計算する．計算結果に従い，スロットリングモジュールがアプリケーションのファイル I/O を制限する．

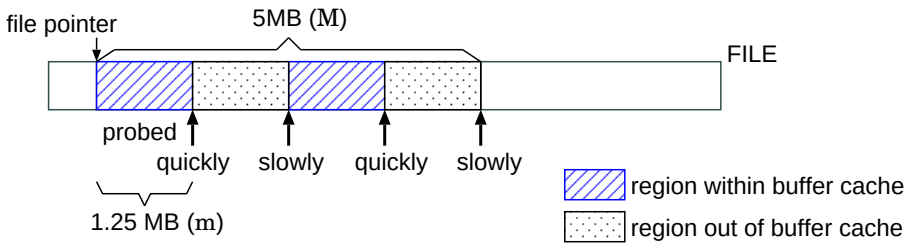


図 4.6: **An example of cache detection.** This diagram exemplifies how Cache Detector behaves, where M is 5 MB and m is 1.25 MB. In this case, Cache Detector's probes quickly return at 1.25 MB and 3.75 MB. This result indicates that the two regions, $(fp, fp + 1.25\ \text{MB}]$ and $(fp + 2.5\ \text{MB}, fp + 3.75\ \text{MB}]$, are in buffer cache.

4.2.2 Cache Detector

Cache Detector はあるファイルがバッファキャッシュ内に存在するか否かを判定する．Cache Detector で用いているアルゴリズムは Arpaci-Dusseau らによって提案された手法 [35] をチューニングしたものである．Cache Detector はファイルを 1 バイト読み込む時間を計測することでプローブを行う．メモリへのアクセスとディスクへのアクセスとを比べると，メモリへのアクセスは格段に短くすむ．本プローブはこの現象を利用する．もしプローブ時間が短ければ，そのファイルの領域はバッファキャッシュに存在すると判定する．もし長ければ，アクセスしたファイルの領域はディスク内に存在すると見なす．

監視下のアプリケーションがファイル読み込みを発行したとき，読み込みを開始する地点 (ファイルポインタ，fp) から M バイトの領域についての判定を行う．まず，M バイトの領域を m バイトの領域に分割する．そして，M バイトの領域中の $\lfloor M/m \rfloor$ 箇所でプローブを行う．具体的には，$fp+m, fp+2m, \ldots, fp+\lfloor M/m \rfloor \cdot m$ の地点でプローブを行う．もし，$fp+k \cdot m$ 地点のプローブ時間が短ければ，$fp+(k-1) \cdot m$ から始まる m バイトの領域はバッファキャッシュ内に存在すると判定する．もしプローブ時間が長ければ，バッファキャッシュ内にデータは存在しないと見なす．DiscNice はアプリケーションがファイル読み込みを発行する度に Cache Detector を呼び出すわけではない．もしアプリケーションのファイルポインタがすでにプローブを行った M バイトの領域を指している場合，Cache Detector は呼び出さずに前回のプローブ結果を用いる．そうでなければ，Cache Detector を呼び出し，プローブを行う．

Cache Detector の動作例を図 4.6 に図示する．今，監視下のアプリケーションが

`read(fd, buf, 4096)` を発行したとし，M と m の値がそれぞれ 5 MB，1.25 MB である場合を考える．Cache Detector がプローブを行ったところ，1.25 MB 目と 3.75 MB 目のプローブ時間が短かった．このとき，Cache Detector では，*fp* から *fp* + 1.25 MB 目まで，*fp* + 2.5 MB 目から *fp* + 3.75 MB 目までの領域がバッファキャッシュ内に存在すると判定する．その後，アプリケーションが `read(fd, buf, 8192)` を発行しても，ファイルポインタが前回プローブした M バイトの領域に含まれるため，DiscNice は Cache Detector を呼び出さない．

ここで，M や m の値は注意深く設定する必要がある．もし M が大きすぎると，時間とともに M バイトの領域の状況が変化してしまい，プローブ結果の精度が低下してしまう．また，M が小さすぎると，OS によるファイル先読みやブロックサイズアクセスによって，常にファイルのデータがバッファキャッシュに存在すると判定してしまう．m に関しても同様である．予備実験を行い，現在のプロトタイプでは M を 5 MB，m を 1.25 MB としている．元々の手法とはこのパラメータが異なっている．

4.2.3 Read-Size Predictor

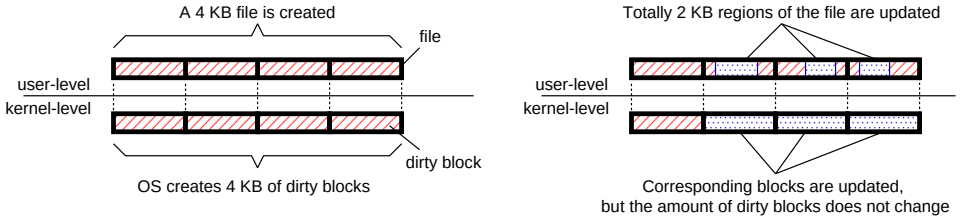
Read-Size Predictor はファイル読み込みに伴うディスク I/O サイズを推測する．監視下のアプリケーションが `read(fd, buf, n)` を発行すると，Read-Size Predictor は Cache Detector を呼び出す．もし読み込むファイルがバッファキャッシュ内にあれば，ディスク I/O サイズを 0 と推測する．もしバッファキャッシュ内になければ，Read-Size Predictor は `n` をブロックサイズで丸め上げる．そして，ファイル先読みで生じるディスク I/O サイズを丸め上げた `n` の値に加えることで，生じるディスク I/O サイズとする．

ファイル先読みによって生じるディスク I/O サイズは，OS が採用しているファイル先読みアルゴリズムをユーザレベル上でエミュレートすることで算出する．ファイル先読みによって生じるディスク I/O サイズはユーザレベルで取得することが難しい．OS がファイル先読みの挙動をユーザレベルに提供しないからである．そこで，OS が採用しているファイル先読みアルゴリズムをソースコードやドキュメント等を通して知る．そして，そのアルゴリズムをエミュレートすることで，ファイル先読みによって生じるディスク I/O サイズをユーザレベルで把握する．

4.2.4 Write-Size Predictor

ディスク書き込みサイズを推測するためには，OS によるブロックサイズアクセスだけでなく遅延書き込みも考慮する必要がある．遅延書き込みは，更新のあったデータブロック数がある一定値を超えたら，もしくは一定時間経過したらデータブロックをディスクに書き込む OS の振る舞いである．そのため，ファイル上の同じ領域への更新は統合されてしまうので，ディスク書き込みサイズとファイル書き込みサイズが必ずしも同じになるとは限らない．たとえば，あるプロセスが新たなファイルを作成し，4 KB のデータを書き込み，続いて 4 KB の領域中の 2 KB の領域にデータを書き込んだとする．まず，プロセスが 4 KB のデータを書き込むと，OS は 4 キロの汚れブロックをメモリ中に作成する (図 4.7(a))．その後，プロセスが 2 KB のデータを書き込むと，OS は先ほど作成した汚れブロックの中から，書き込みのあった領域に該当するブロックの内容を更新する (図 4.7(b))．この場合，ファイル I/O サイズは $6\ (= 4 + 2)$ KB であるが，実際に生じるディスク I/O サイズは 4 KB である．これは，OS が新たな汚れブロックを作成しないことに起因する．また，上述の例において，2 KB の上書きの代わりに，ファイルを削除した場合，ファイル I/O サイズは 4 KB となるが，実際に生じるディスク I/O サイズは 0 となる．なぜならプロセスがファイル削除を行ったとき，OS は余計なディスクアクセスを避けるために，メモリ中の汚れブロックを削除するからである．

遅延書き込みによって生じるファイル I/O サイズとディスク I/O サイズの違いを考慮するために，Write-Size Predictor は *Update Region Table* というデータ構造を保持する．Update Region Table は汚れブロックがディスクに書き込まれるまで，データの書き込みがあったファイルの領域を保持する．監視下のアプリケーションが `write(fd, buf, n)` を発行したとき，Write-Size Predictor は書き込みがあった領域を調べる．具体的には，b をデータブロックサイズとすると，$[\lfloor \mathtt{fp}/b \rfloor \cdot b, \lceil (\mathtt{fp} + \mathtt{n})/b \rceil \cdot b)$ の領域に更新があったとみなす．計算後，Write-Size Predictor は Update Region Table を参照する．もし算出した領域が Update Region Table に登録されていたら，Write-Size Predictor は今回の `write()` はディスクアクセスを伴わないと判断する．もし登録されていなければ，その領域を登録する．もしファイルが削除されたら，そのファイルに関する領域の登録を削除する．OS が汚れブロックをディスクへ書き出すとき，Write-Size Predictor は Update Region Table を参照しながら，登録されている領域の大きさの合計をディスク書き込みサイズとする．そして，Write-Size Predictor はディスク書き込みサイズをポリシーモ



(a) When a file is created　　(b) When the file is updated

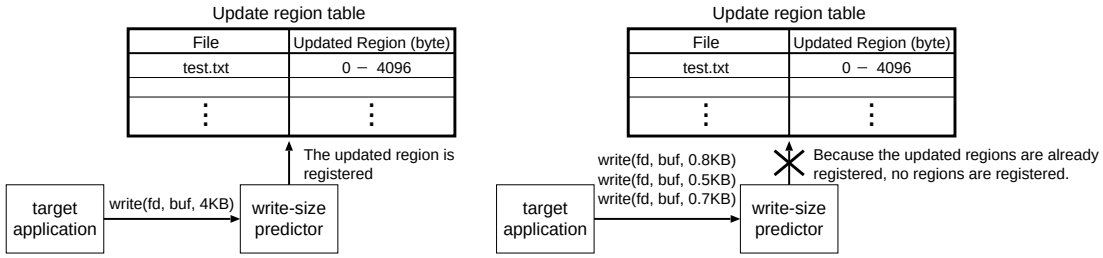
図 4.7: **OS behavior in file creation and updates.** The two figures show OS behavior in file creation and updates. When we open a file and write 4 KB data on it, OS creates dirty blocks. After that, when we updates a 2 KB region of it, OS writes the updates on the dirty blocks.

ジュールに通知し，Update Region Table の登録内容を破棄する．

上述の例 (図 4.7) では，ファイルが生成されて書き込みが生じると，Write-Size Predictor は 4 KB の領域を Update Region Table に登録する (図 4.8(a))．次にプロセスがファイルへ 2 KB のデータが書き込むが，Write-Size Predictor は Update Region Table の内容を更新しない．これは，最初の 4 KB の書き込みの際に，書き込む領域がすでに Update Region Table に登録されているからである (図 4.8(b))．これにより，OS が汚れブロックをディスクに書き出すとき，Write-Size Predictor は Update Region Table の登録内容に従って 4 KB のディスク書き込みが生じると推測し，その値をポリシーモジュールに通知する．通知後，Write-Size Predictor は Update Region Table の登録内容をクリアする．

4.3 実装

DiscNice のプロトタイプを Linux 2.4.27 上に実装を行った．プロトタイプは `libDiscNice` ライブラリ，`write-adjuster` プロセス，そして `policyd` プロセスからなる．DiscNice のアーキテクチャを図 4.9 に示す．`libDiscNice` は監視するアプリケーションにリンクさせるための動的リンクライブラリである．`libDiscNice` は，Cache Detector と Read-Size Predictor，スロットリングモジュールを含み，リンクしたアプリケーションのファイル I/O を監視する．アプリケーションがファイル読み込みを行うと，`libDiscNice` は Cache Detector と Read-Size Predictor を呼び出しディスク読み込みサイズを推測する．アプリケーション



(a) When a file is created　　(b) When the file is updated

図 4.8: **Write-size predictor behavior in file creation and updates.** The two figures show how Write-Size Predictor manages *update region table* in file creation and updates. When targeted applications write data to a file, Write-Size Predictor registers the updated region to update region table. When the same region is updated, update region table is not changed and thus we can avoid accounting the file write size to disk I/O size.

がファイル書き込み，もしくはファイルの削除を行うと，`libDiscNice` はそれを `write-adjuster` に通知する．`write-adjuster` は Write-Size Predictor を含むプロセスである．Write-Size Predictor は， Update Region Table を使ってディスク書き込みサイズを推測するため，プロセスのファイル書き込みとファイル削除を把握する必要がある．ここで，Update Region Table の管理を容易にするために，Write-Size Predictor と `libDiscNice` とを分離させている．`write-adjuster` がファイル書き込み，ファイル削除の通知を `libDiscNice` から受けると，Write-Size Predictor を呼び出しディスク書き込みサイズを推測する．`libDiscNice` と `write-adjuster` はそれぞれ推測したディスク I/O サイズを `policyd` プロセスに通知する．`policyd` は予め与えられるポリシーに従って，ファイル I/O の発行率を決定するプロセスである．`policyd` はファイル I/O の発行率を `libDiscNice` のスロットリングモジュールに通知し，アプリケーションの発行するファイル I/O を制御する．

4.3.1 libDiscNice

`libDiscNice` はファイル I/O リクエストをフックする．具体的には，`libc` 内のファイル I/O に関する関数をオーバーライドしている．現在のプロトタイプでは，`read()` と `write()`，`unlink()` をオーバーライドしている．これらの関

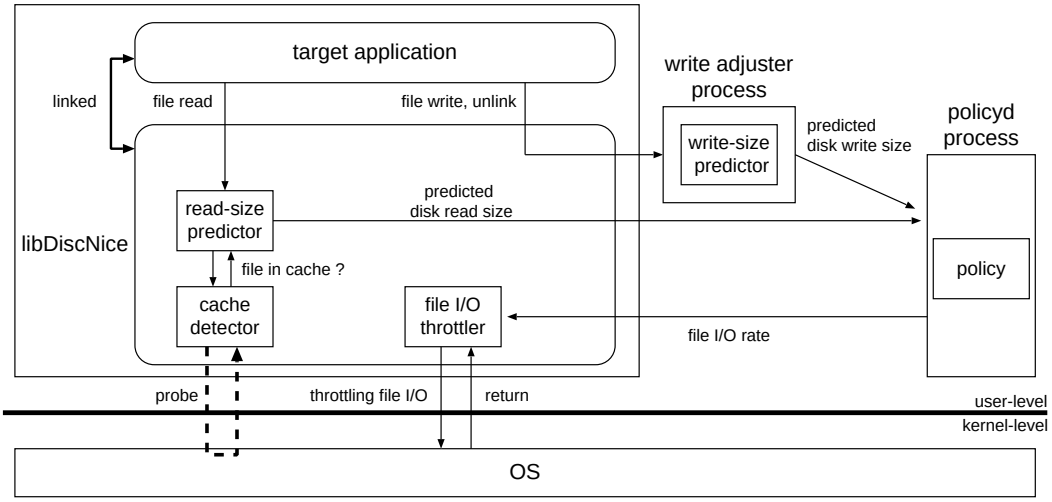


図 4.9: **DiscNice architecture.** DiscNice is composed of `libDiscNice` library, the `write-adjuster` process, and the `policyd` process. `libDiscNice` contains the cache detector, the read-size predictor, and the throttling module. The `write-adjuster` process implements the write-size predictor. The `policyd` process determines the file I/O rate based on a given policy.

数をオーバーライドするために，標準ライブラリとリンクする前に特定のライブラリとのリンクを行うことができるプリロード機能を用いる．Linux では環境変数 `LD_PRELOAD` を用いることで，特定のライブラリを強制的にリンクすることができる．

ファイル読み込みをフックすると，`libDiscNice` は推測したディスク I/O サイズを `policyd` に通知する．ファイル書き込みやファイル削除をフックすると，`libDiscNice` はそのリクエストを `write-adjuster` に通知する．`libDiscNice` は `policyd` や `write-adjuster` とは TCP/IP コネクションを通してメッセージのやりとりを行う．もし推測したディスク I/O サイズが 0 であれば，不必要なプロセス間通信を避けるために，サイズの通知を行わない．

また，監視下のアプリケーションが子プロセスを生成しても，その子プロセスは `libDiscNice` へ自動的にリンクされるため，DiscNice によってディスク I/O が制御される．現在のプロトタイプでは，子プロセスのディスク I/O 発行量は親プロセスに加算されるようになっている．

4.3.2 write-adjuster

監視下のアプリケーションが `write(fd, buf, n)` を発行すると，`libDiscNice` はファイルポインタの現在地と `n` とを `write-adjuster` に送信する．そして，`write-adjuster` は Write-Size Predictor を呼び出す．Write-Size Predictor はファイル書き込みのあった領域が Update Region Table に登録されているかどうかを確認する．もし登録がされていなかったら，書き込む領域を Update Region Table に登録する．

Update Region Table は `wpred_st` 構造体のハッシュテーブルである．キーはファイルの i ノード番号である．`wrep_st` 構造体はファイルの i ノード番号，そして更新された領域を保持する配列からなる．

```
struct wrepd_st {
        /* file's inode-number */
        ino_t   ino;

        /* updated region */
        boolean wpos[MSIZE];
};
```

`wpos` は 4 KB 単位でファイルの更新があった領域を保持する．4 KB で保持する理由は，Linux 2.4.27 がディスク上のファイルのデータを 4 KB 単位でアクセスするからである．もし `wpos[i]` の値が true であれば，$[i \cdot 4KB, (i+1) \cdot 4KB)$ の領域を更新のあった領域とみなす．たとえば，監視下のアプリケーションがファイルを新たに作成し，100 KB のデータをファイルの先頭から書き込んだとき，Write-Size Predictor は Update Region Table に新たなエントリを作成し，`wpos` 配列の 0 から 24 $(= 100KB/4KB - 1)$ の値を true とする．もしファイルが削除されたら，Write-Size Predictor は該当するエントリを削除する．ディスク書き込みサイズを推測するために，Write-Size Predictor は Update Region Table 中の true の個数を数える．

OS が汚れブロックをディスクに書き込んだとき，Write-Size Predictor は `policyd` に推測したディスク書き込みサイズを通知し，Update Region Table をクリアする．Write-Size Predictor は遅延書き込みの発生を知るために，プロセスファイルシステムの監視を行う．プロセスファイルシステムでは，システム全体で生じたディスク

書き込み量を取得することができる．この値の変化を監視することで，Write-Size Predictor は汚れブロックがいつディスクに書き出されたかを検出する．システム全体の状態を監視する手法は Linux 以外の OS にも適用可能であると考えられる．なぜなら，最近の OS は自身の内部状態をユーザレベル上に提供するからである．実際に，Windows では System Information Function [47] が，Solaris には Dtrace [50] という OS の内部状態を取得するためのインターフェースが存在する．

プロセスファイルシステムへのアクセスによるオーバヘッドを減らすために，アクセスする回数を削減した．監視下のアプリケーションが `write()` を呼び出す度に，プロセスファイルシステムにアクセスするのはコストが高い．そこで現在のプロトタイプでは，500 ms ごとにプロセスファイルシステム上の値を取得している．4.4 節で述べるように，このオーバヘッドの削減手法はディスク I/O サイズの推測への影響は少ない．

4.3.3 policy_d と現在のポリシー

`policyd` プロセスはディスク I/O サイズを取得し，`policy()` 関数を呼び出す．`policy()` 関数はファイル I/O をどのように制限するかを決める．`policy()` の定義は policy(pid_t pid, size_t predicted) である．`policy()` はプロセス ID，そして推測したディスク I/O サイズを引数として受け取る．返り値として，`pid` のプロセス ID を持つプロセスが発行すべきファイル I/O の発行率を返す．ここで，発行率とはプロセスが要求したファイル I/O サイズから算出される，発行してもよいファイル I/O の割り合いを指す．

プロトタイプ上に実装した制限ポリシーは rate-windows [52] を基にしている．rate-windows はある閾値以下にディスク帯域の使用量を制限するポリシーである．ディスク I/O の発行量を制限するために，ディスク I/O リクエストを記録するスライディングウィンドウを用いる．スライディングウィンドウを用いることで，監視下のアプリケーションのディスク帯域使用量を算出することができる．アプリケーションのディスク帯域使用量が閾値を超えると，`sleep` をアプリケーションに挿入してディスク I/O の発行を遅延させる．

スライディングウィンドウの図を図 4.10 に示す．監視下のアプリケーションがディスク I/O を発行すると，プロトタイプに実装したポリシーでは 2 種類の情報を保存する．1 つめは推測したディスク I/O サイズ，2 つめは最近のディスク I/O リクエストからの経過時間である．実装したポリシーは直前の N 秒間でのプロセ

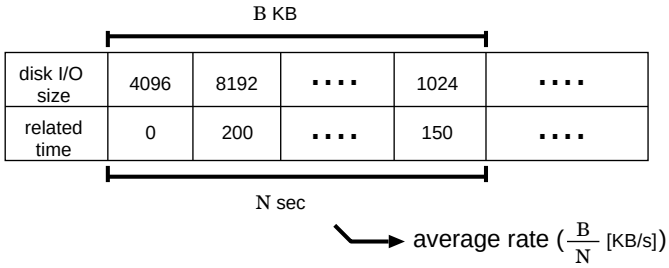


図 4.10: **Rate-windows.** This figure shows the data structure of rate-windows. We record the predicted disk I/O size and the time elapsed from the last disk I/O. We can caluculate the disk I/O rate during the last N seconds from these pieces of information.

スのディスク帯域使用量をスライディングウィンドウを用いて算出する．閾値を R KB/s，直前の N 秒間に発行されたディスク I/O サイズを B KB とする．もし $B/N \leq R$ であれば，ディスク I/O を発行しても帯域使用量は R 以下であるとみなす．そのため，`policy()` はスロットリングモジュールにファイル I/O を実行するよう通知する．もし $B/N > R$ であれば，ディスク帯域使用量が閾値を超えるので，`policy()` は $B/R - N$ だけ `sleep` するようスロットリングモジュールに通知する．`sleep` 後，スロットリングモジュールはファイル I/O を実行する．現在のプロトタイプでは，文献 [52] を参考にし，N を 1 秒としている．

汚れブロックがディスクに書き出されるときに，B の値が大きくなることがある．なぜなら，遅延書き込みによって，OS が汚れブロックをまとめてディスクに書き出すからである．結果として，遅延書き込みが起きた直後に，B/N が R を超えてしまうことがある．プロトタイプに実装したポリシーでは，B/N が R より低くなるまで，ファイル I/O を遅延させている．

`libDiscNice` と `policyd` の通信によるオーバヘッドを削減するために，通信回数を削減した．OS は汚れブロックをまとめてディスクに書き出すため，アプリケーションが `write()` や `unlink()` を呼び出す度に `libDiscNice` が `policyd` からファイル I/O の遅延時間を取得することは無駄が多い．そこでプロトタイプでは，`libDiscNice` は 300 ms ごとに遅延時間の取得を行っている．4.4 節で述べるように，このオーバヘッド削減手法はディスク I/O の推測への影響は少ない．

4.3.4 Read-Size Predictor

Linux 2.4.27 上では，ディスクブロックサイズは 1 KB であるが，ディスク上のファイルにはページサイズ単位 (4 KB) でアクセスする．そのため，プロトタイプではブロックサイズを 4 KB として扱っている．

ファイル先読みアルゴリズムをエミュレートするために，Linux 2.4.27 のソースコードを調査した．Linux 2.4.27 では，ファイルの先読みを効率的に行うために，2 つのモードを用意している．1 つめは synchronous モード，2 つめは asynchronous モードである．synchronous モードでは，ファイルの先読みサイズは 4 KB で固定である．一方，asynchronous モードでは，ファイルの読み込みがシーケンシャルであればファイルの先読みサイズが大きくなっていく．まず，Linux は synchronous モードでファイルを読み込む．そして，次のファイルの読み込みが前回の読み込みと連続していれば，asynchronous モードに移る．ファイルがシーケンシャルで読み込まれる限り，Linux は asynchronous モードのままである．もし，ファイル読み込みがシーケンシャルでなくなると，synchronous モードとなり，ファイル先読みサイズが 4 KB に戻る．

プロトタイプでは，synchronous モードのみをエミュレートしている．asynchronous モードのエミュレーションはディスク I/O サイズの推測精度を上げることができると考えられる．しかし，DiscNice を用いると，asynchronous モードでのファイル先読みサイズの増加はあまり行われない．なぜなら，Linux が Cache Detector のプローブをランダム読み込みであると判定し，定期的に synchronous モードに戻り，ファイル先読みサイズが 4 KB となる．そのため，asynchronous モードのエミュレーションを行わなくても，推測精度を大きく損なわずにディスク I/O サイズの推測を行うことができる．実際，4.4 節で述べるように，ディスク I/O サイズの推測精度誤差は最大でも 4.4% であり，DiscNice を用いることによる実行時間の増加はシーケンシャルな読み込みを行う実際のアプリケーションを用いても 3.05% 以下である．

4.3.5 議論

Linux 上では，ファイル I/O を伴わずディスク I/O を発行することがある．`mmap()` システムコールは，ファイルをメモリへマップする．マップされたファイルにアクセスをすると，Linux がディスクからファイルの内容を読み込み，メモリにロード

する．そのため，現在の DiscNice では mmap() によって生じるディスク I/O を検出することができない．このディスク I/O を検出するためには，DiscNice を拡張する必要がある．具体的には，libDiscNice が mmap() によって生じるディスク I/O を検出できるように mprotect() システムコールを用いてファイルをマップしたメモリ領域を保護する．保護したメモリ領域にアプリケーションがアクセスしたときに，ページフォールトが 1 回生じるだけなので，オーバヘッドは低いと考えられる．

また，open() 後に stat() を行うだけのプログラムは，ファイル I/O を伴わず，i ノードだけを読み込むことによってディスク I/O を発行する．このプログラムが大量のファイルを処理すると，ディスク I/O が頻繁に発行されることが考えられる．これらのディスク I/O の検出は，open() の時間を計測することで行うことができる．i ノードキャッシュ内に存在する i ノードを読み込んだときとディスク内の i ノードを読み込んだ際には，open() の実行時間に差が生じることが知られており [94]，この手法を DiscNice に組み込むことで，これらのディスク I/O を検出することができると考えられる．

悪意のあるアプリケーションの制御は本論文の対象外とする．現在の DiscNice 実装は，信頼できるアプリケーションを制御することを仮定している．そのため，悪意のあるアプリケーションは DiscNice による監視をかいくぐることができる．たとえば，悪意のあるアプリケーションは libDiscNice によるフックを抜け，read() や write()，unlink() を直接呼び出すことができる．そのため，悪意のあるアプリケーションへの対応は今後の課題とする．

4.4 実験

DiscNice の有用性を確認するために実験を行った．実験環境は表 4.4 の通りである．

4.4.1 マイクロベンチマーク

DiscNice のディスク I/O サイズの推測精度を確認するために，実際に生じたディスク I/O サイズと DiscNice が推測したディスク I/O サイズとを比較した．実際に生じたディスク I/O サイズを取得するために Linux 2.4.27 を拡張した．具体的には，ディスクアクセスを発行する際に，どのプロセスがどれだけのディスクアクセ

表 4.3: **Experimental environment**

CPU	Pentium 4 2.8 GHz
Memory	1024 MB
HDD	7200 rpm, SCSI
OS	Linux 2.4.27

表 4.4: **Accuracy of disk I/O prediction (micro-benchmark).** This table reports the actual disk I/O size of micro benchmarks and disk I/O size predicted by DiscNice.

Benchmark	`Sequential`	`Stride`	`Random`	`Create`
Actual disk I/O [byte]	$615,022,592$	$411,250,688$	$491,073,536$	$409,612,288$
Predicted disk I/O [byte]	$615,006,208$	$410,173,440$	$492,625,920$	$409,600,000$
Error margin	-0.00%	-0.26%	0.32%	-0.03%

スを行ったかを保持するよう拡張した．ディスクアクセスリクエストを管理する構造体にプロセス ID を付加し，ディスクアクセスを行う関数で ID をたどる．そして，その ID に該当するプロセス構造体があれば，ディスク I/O サイズを記録する．記録したディスク I/O サイズはプロセスファイルシステムから取得可能である．この拡張を使いながら，以下のマイクロベンチマークプログラムを実行した．

- `Sequential` ：600 MB のファイルをシーケンシャルに読み込む
- `Random` ：600 MB のファイルを 10 回ランダムに読み込む．
- `Stride` ：600 MB のファイルを 12 KB ごとに 1 バイト読み込む．
- `Create` ：ファイルを open し，文字 'a' を 400 MB 分書き込む．

実験結果を表 4.4.1 に示す．すべての場合において，精度誤差は 1% 以下である．

DiscNice を用いてディスク帯域割り当てを変更可能かを確認するために，先ほど用いたマイクロベンチマークと DiscNice を動作させ，ディスク帯域使用量を計測した．閾値を 5 MB/s と設定した．比較として，まず DiscNice を用いないとき

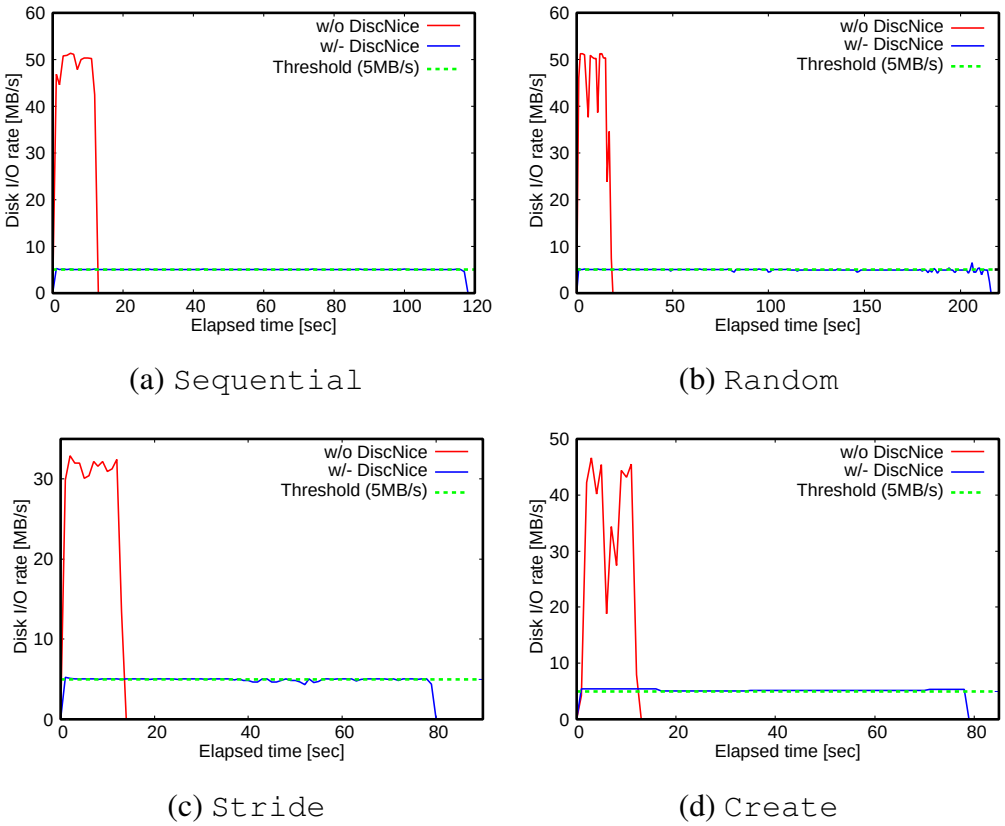


図 4.11: **Results of throttling micro-benchmark programs.** These graphs show disk I/O rate of micro-benchmarks with DiscNice. The x-axis is the elapsed time and the y-axis is disk I/O rate.

のディスク帯域使用量も測定した．結果を図 4.11 に示す．x 軸は経過時間，y 軸はディスク帯域の使用量である．実線が DiscNice で制限を行ったときのディスク帯域使用量，破線が DiscNice を用いないときのディスク帯域使用量である．`Create` のグラフでは，汚れブロックがディスクに書き出された間隔でのディスク帯域使用量である．図 4.11 より，DiscNice が閾値以下にディスク帯域使用量を制限していることがわかる．

4.4.2 DiscNice モジュールの有効性

ディスク帯域を制限する際に DiscNice の各モジュールがどのように貢献しているかを調べるために，以下の 4 つのフェーズからなるベンチマークを用意した．

1. Random-Read フェーズ: 200 MB のファイルを 10 回ランダムに読み込む．

表 4.5: **DiscNice configuration.** This table shows the five configurations of DiscNice we used in the experiment to confirm the contribution of each module.

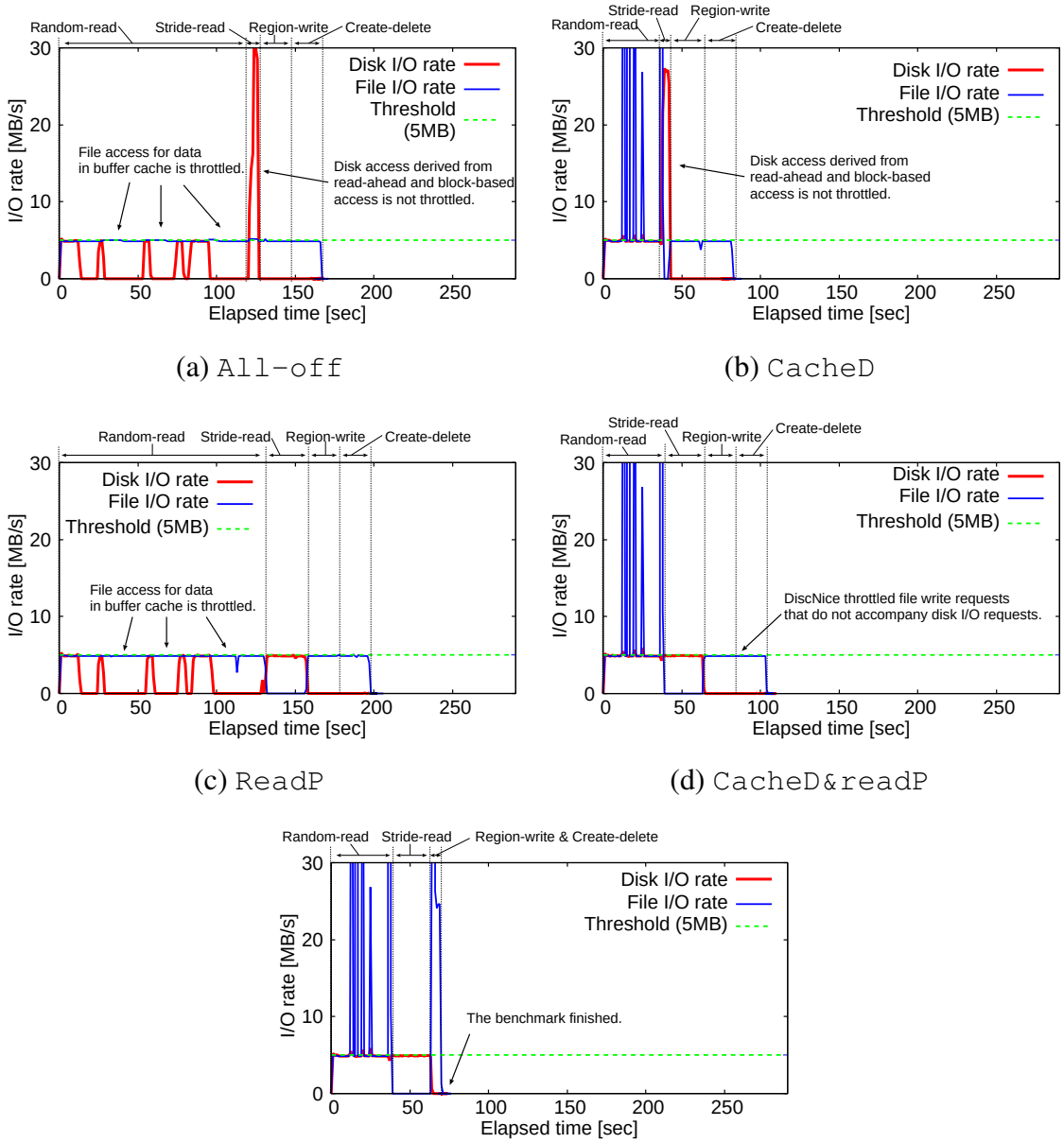
Configration Name	Cache detector	Read-size predictor	Write-size predictor
`All-off`	off	off	off
`CacheD`	on	off	off
`ReadP`	off	on	off
`CacheD&readP`	on	on	off
`All-on`	on	on	on

2. Stride-Read フェーズ: 200 MB のファイルを 12 KB おきに 1 バイト読み込む.
3. Region-Write フェーズ: 1 つのファイルに 10 KB のデータを 10,000 回同じ領域に書き込む.
4. Create-Delete フェーズ: 10 KB のファイルの生成，削除を 10,000 回繰り返す.

5 種類の設定の異なる DiscNice を用いて，このベンチマークのディスク帯域を制限した．5 種類の設定は表 4.4.2 のとおりである．5 種類の設定は DiscNice のモジュール (Cache Detector, Read-Size Predictor, Write-Size Predictor) の有無が異なる．ディスク帯域の閾値は 5 MB/s とした．

結果を図 4.12 に示す．`All-off` では，Random-Read フェーズにおいて，ディスク I/O を過剰に制限していることがわかる (図 4.12(a))．`All-off` はファイル I/O サイズをディスク I/O サイズと考え制限しているため，バッファキャッシュ 内のデータを読み込んでいるときもファイル I/O を制限している．また，`All-off` ではファイル先読みやブロックサイズでのアクセスによるファイル I/O サイズとディスク I/O サイズとのずれを扱うことができない．結果として，ディスク帯域使用量が Stride-read フェーズでは閾値を超えている．

Cache Detector を組み込むと，DiscNice はバッファキャッシュ内に存在するファイルに対しての読み書きを制限しなくなる (図 4.12(b))．しかし，ファイル先読みやブロックサイズでのアクセスによって生じるディスク I/O を制限することはできない．そのため，Stride-Read フェーズで，ディスク帯域使用量が閾値を超えている．



(a) All-off

(b) CacheD

(c) ReadP

(d) CacheD&readP

(e)All-on

図 4.12: **Results of throttling 4-phases benchmark.** These graphs show disk I/O rate of 4-phases benchmark. We throttled the benchmark with DiscNice, turning on and off DiscNice's modules. The x-axis is the elapsed time and the y-axis is disk I/O rate.

Read-Size Predictor を組み込むと，DiscNice はファイル先読みやブロックサイズでのアクセスによって生じるディスク I/O を考慮し，ファイル I/O の発行を制限する．しかし，バッファキャッシュ内に存在するファイルに対しての読み書きを制限

しないので，Random-Read フェーズにおけるディスク I/O を過剰に制限してしまう (Fig. 4.12(c)).

Cache Detector と Read-Size Predictor とを組み込むと，DiscNice は，Random-Read フェーズと Stride フェーズにおいて，過剰な制限なくディスク帯域使用量を閾値以下に抑えることができる．しかし，Region-Write，Create-Delete フェーズではディスク I/O が発行されないにもかかわらず，DiscNice はファイル I/O を制限している (Fig. 4.12(d)).

Fig. 4.12 (e) より，`All-on` がすべてのフェーズにおいて過剰な制限なくディスク帯域使用量を閾値以下に抑えていることがわかる．`All-on` では，Random-Read, Region-Write, Create-Delete フェーズ上の，ディスク I/O を伴わないファイル I/O の制限をしていない．そのため，`All-on` の設定の DiscNice を用いると，ベンチマークの実行時間が最も短くなっている．

4.4.3 マクロベンチマーク

DiscNice が実際のアプリケーションのディスク帯域を制限可能かを確認するために，以下のベンチマークプログラムを用意した．

- `Tar`: 約 200 MB のアーカイブファイルを展開する．展開後のファイル数，ディレクトリ数はそれぞれ 666 個，25,860 個である．
- `Make`: Apache [95] 2.0.52 をコンパイルする．ソースコードのサイズは約 8 MB である．
- `Grep`: Linux カーネル 2.4.27 のソースとドキュメントから ‘epoch’ という文字列を検索する．ファイルの合計サイズは約 200 MB である．ファイル数は 25,860 個である．

表 4.4.3 に実際のディスク I/O サイズと推測したディスク I/O サイズを示す．マクロベンチマークでの精度誤差は．最大でも 4.4% である．誤差は `Grep` を動作させたときに顕著である．この誤差はディレクトリを読み込んだときに生じる誤差である．現在のプロトタイプでは，ディレクトリの読み込みに関するプローブを行っていないためである．

DiscNice が実際のアプリケーションのディスク帯域を制限できるかを確認するために，上記の 3 つのベンチマークを使ってディスク帯域使用量の制限を行った．

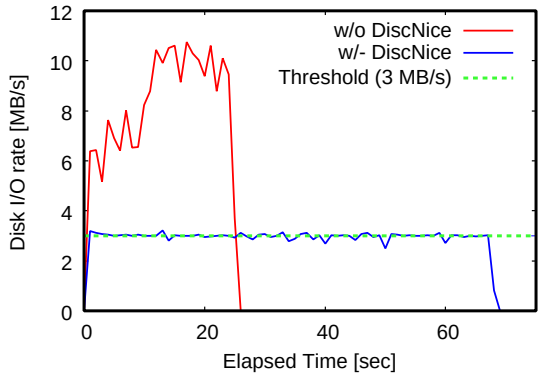
表 4.6: **Accuracy of disk I/O prediction (macro-benchmark).** This table shows the actual disk I/O size of macro benchmarks and disk I/O size predicted by DiscNice.

Benchmark	`Tar`	`Make`	`Grep`
Actual disk I/O [byte]	$435,482,624$	$48,271,360$	$215,965,696$
Predicted disk I/O [byte]	$430,954,752$	$46,735,360$	$206,376,960$
Error margin	-1.0%	-3.2%	-4.4%

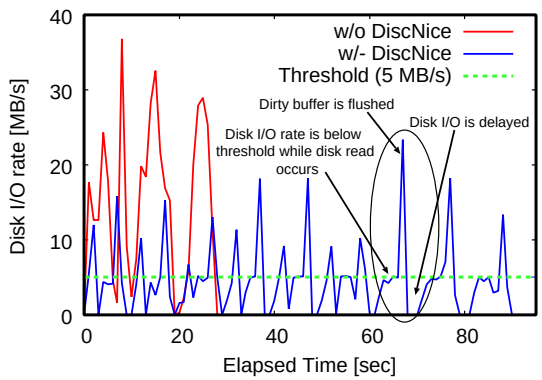
閾値はそれぞれ，`Tar` に対しては 5 MB/s，`Make` には 500 KB/s，そして `Grep` には 3 MB/s という閾値を与えた．図 4.13 に結果を示す．x 軸は経過時間，y 軸はディスク帯域使用量を示している．実線は DiscNice でディスク帯域を制限したときのディスク帯域使用量，破線は DiscNice を用いなかったときのディスク帯域使用量である．図 4.13 より，`Grep` に関してはディスク帯域を閾値以下に抑えていることがわかる．これは `Grep` が，データを書き込むことがなく，読み込みのみを行うベンチマークであることに起因する．データ書き込みを行う他のベンチマークでは，閾値を超える点がいくつかある．この点は OS が汚れバッファをディスクに書き出すときに生じる．たとえば，図 4.13(b) の円で囲んだ部分に注目する．ディスク帯域使用量は，しばらくの間，閾値以下となっている．そして，OS が汚れブロックをディスクに書き出すとき，ディスク帯域使用量は一時的に閾値を超える．その後，ディスク I/O はしばらく発行されない．これは，DiscNice 上に実装したポリシーが平均ディスク帯域使用量を閾値以下に抑えようとするために，ファイル I/O を遅延しているからである．図 4.14 に，遅延書き込みが生じた間隔で計算した平均ディスク帯域使用量を示す．図 4.14(a) の円は図 4.13(b) の円と対応している．図 4.14 より，DiscNice は搭載したポリシーに従って，ディスク帯域使用量を制限していることがわかる．

4.4.4 オーバヘッド

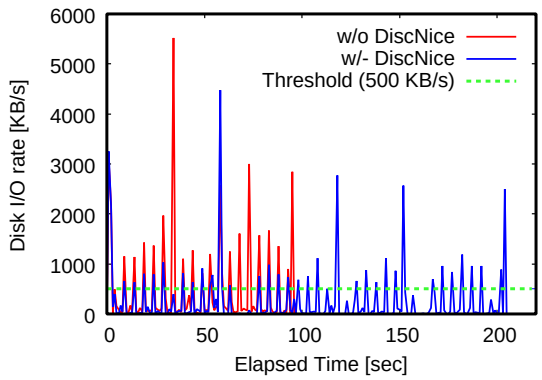
DiscNice によるオーバヘッドを計測するために，4.4.3 節で用意したベンチマークを使って，DiscNice を用いたときと用いなかったときの実行時間を計測した．ここで，オーバヘッドは DiscNice を用いたときの実行時間の増加を指す．設定した



(a) `Grep` (Disk I/O threshold is set to 3 MB/s)

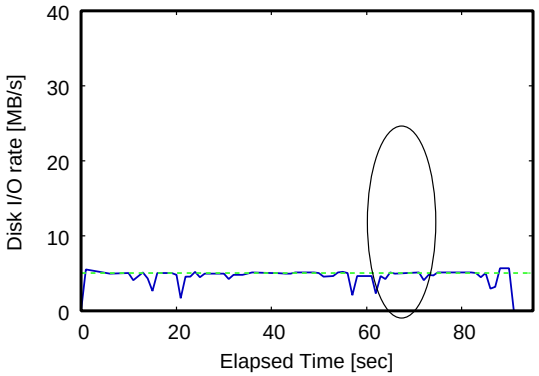


(b) `Tar` (Disk I/O threshold is set to 5 MB/s)

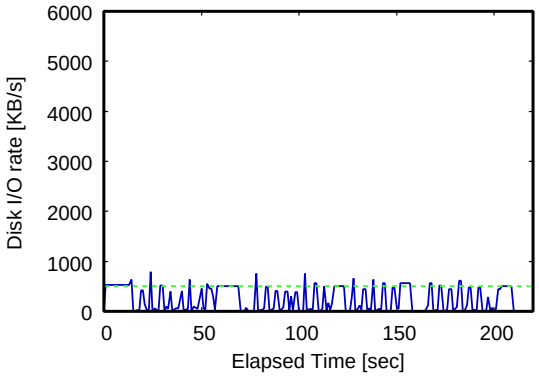


(c) `Make` (Disk I/O threshold is set to 500 KB/s)

図 4.13: **Results of throttling macro-benchmark programs.** These graphs show disk I/O rate of micro-benchmarks with DiscNice. The x-axis is the elapsed time and the y-axis is disk I/O rate. The circled region in (b) corresponds to the one in Fig. 4.14.



(a) Tar



(b) Make

図 4.14: **Average disk I/O rate of `Tar` and `Make` with DiscNice.** The two graphs show the average disk I/O rate over intervals between the dirty blocks flush and the following disk I/O. The circled region in (a) corresponds to the one in Fig. 4.13.

閾値は無限である．

表 4.4.4 に結果を示す．オーバヘッドは Tar と Grep に関しては 4% 以下にとどまっている．また，表 4.4.4 より，Make のオーバヘッドが他のベンチマークと比べて大きいことがわかる．Make はディスク I/O を行う時間より CPU を使用する時間の方が長いため，DiscNice のパフォーマンスへの影響は大きい．DiscNice によって生じるオーバヘッドは 12% 以下である．

ここで，これらのオーバヘッドには Cache Detector によるサイドエフェクトが含まれていることに注意されたい．4.3.4 節で述べたように，Cache Detector によるプローブは， Linux のファイル先読みアルゴリズムを阻害するため，アプリケーションのスループットに影響することが考えられる．本実験では，Tar がワース

表 4.7: **Overhead incurred by DiscNice.** This table reports how overhead DiscNice imposes for real applications.

Benchmark	Tar	Make	Grep
w/o DiscNice [sec]	29.87	97.13	27.87
w/- DiscNice [sec]	30.78	108.55	27.98
Overhead	3.05%	11.76%	0.40%

トケースである．なぜなら，`Tar` は 200 MB のアーカイブファイルをシーケンシャルに読み込むからである．しかし，`Tar` のオーバヘッドは 4% 以下にとどまっている．

4.4.5 ケーススタディ

DiscNice を用いたケーススタディとして，バックグラウンドタスクのディスク帯域を制限し，ディスク帯域競合によるフォアグランドタスクのスループット劣化を軽減できるかを確認する．実験として，4.1.1 節で行った実験を DiscNice を用いて行った．DiscNice によって制御されている `Rsync` と同時に動作する `Sequential` と `Tar` の実行時間をそれぞれ計測した．閾値を 1 MB/s，3 MB/s，そして 5 MB/s と変化させていき，それぞれの場合での実行時間を取得した．

表 4.4.5 に `Sequential` の実行時間を示す．表から，ディスク帯域幅を制限されていない `Rsync` と同時に動作すると，実行時間は約 4 倍となることがわかる．しかし，DiscNice で `Rsync` のディスク帯域使用量を制限すると，`Sequential` の実行時間の増加は軽減されている．閾値を 1 MB/s とすると，実行時間の増加は約 6% に抑えることができる．また，閾値を 5 MB/s とすると，実行時間は約 101% の増加に抑えることができる．

表 4.4.5 に `Tar` の実行時間を示す．`Sequential` の場合と同様に，DiscNice でディスク帯域使用量を制限することで，`Tar` のスループット劣化を軽減することができる．`Rsync` のディスク帯域を制限しないと，`Tar` の実行時間は約 3 倍となる．しかし，DiscNice を用いて閾値を 1 MB/s とした場合，`Tar` の実行時間の増加は約 12% となる．閾値を 5 MB/s とすると，実行時間の増加を約 105% までに抑えることができる．実験より，DiscNice を用いることで，バックグラウンドタスクのディスク帯域使用によるフォアグランドタスクのスループット劣化を軽減す

表 4.8: **Comparison of execution times of `Sequential`** This table shows how DiscNice alleviates disk I/O contention for foreground `Sequential`. DiscNice changed its own threshold; 1MB, 3MB and 5 MB.

Benchmark	Execution time [sec]	Increasing rate
w/o `Rsync`	4.18	—
w/- controlled `Rsync` (1 MB)	4.81	106%
w/- controlled `Rsync` (3 MB)	5.91	143%
w/- controlled `Rsync` (5 MB)	8.27	201%
w/- low priority `Rsync`	17.26	413%
w/- `Rsync` [sec]	17.49	418%

表 4.9: **Comparison of execution times of `Tar`** This table shows how DiscNice alleviates disk I/O contention for foreground `Sequential`. DiscNice changed its own threshold; 1MB, 3MB and 5 MB.

Benchmark	Execution time [sec]	Increasing rate
w/o `Rsync`	4.47	—
w/- controlled `Rsync` (1 MB)	5.02	112%
w/- controlled `Rsync` (3 MB)	6.73	151%
w/- controlled `Rsync` (5 MB)	9.15	205%
w/- low priority `Rsync`	12.82	287%
w/- `Rsync` [sec]	13.10	293%

ることができた．

4.5 まとめ

本章では，ユーザレベルでプロセスのディスク帯域を制御する機構， DiscNice について述べた．ユーザレベルではプロセスの発行するディスク I/O サイズを知ることは困難である．DiscNice は，OS のディスクアクセスに関する振る舞いに着

目する．OS のファイル先読みや遅延書き込みといった動作を推測することで，プロセスのファイル I/O に伴うディスク I/O の発行量を推測する．推測したディスク I/O サイズを基に，プロセスのディスク帯域使用量を把握し，制御ポリシーに従って，ファイル I/O の発行を制限する．ファイル I/O の発行を制限することで，間接的にプロセスのディスク帯域を制限する．

DiscNice は次の特徴をもつ．はじめに，DiscNice は OS カーネルを修正することなく導入可能である．DiscNice のすべてのモジュールはユーザレベルで実装されている．また，カーネルレベルでのディスク帯域制御機構と比べると移植性が高い．DiscNice は OS 固有の機能を利用していないので，システムコールやライブラリのインターフェースを変更することで他の OS へ移植可能である．最後に，ディスク帯域推測機構と制御ポリシーとが明確に分離しているため，ポリシーを用途に応じて設定することができる．

ファイル I/O の制御ポリシーを rate-windows として，プロトタイプを Linux 2.4.27 上に実装した．実際のアプリケーションを含む実験を行ったところ，ディスク I/O サイズの推測誤差は最大でも 4.4%，オーバヘッドも 12% 以下にとどまった．また，DiscNice を用いてバックグラウンドタスクを制御したところ，ディスク帯域の競合によって生じるフォアグランドタスクのスループット劣化を緩和することができた．

第5章　VMM レベルでの資源管理ポリシー変更手法

本章では，仮想化環境で稼働している OS (ゲスト OS) の資源管理ポリシーを変更する手法，*FoxyTechnique* について述べる．FoxyTechnique は仮想化環境を構築しているソフトウェアである仮想マシンモニタ (VMM) を利用する．仮想化環境では，OS は VMM が提供する仮想デバイス上で稼働する．FoxyTechnique では，仮想デバイスの動作に対する OS の振る舞いに着目する．OS の知識を利用して OS の振る舞いを推測しながら仮想デバイスの動作を意図的に変更することで，実際とは異なる状況をゲスト OS に見せる．これにより，OS に所望の動作をさせて，新たな資源管理ポリシーを導入したときと近い効果を得る．たとえば，タイマ割り込みをカウントすることで OS がタイムスライスを計測しているという知識を用いる．タイマ割り込みの配送率を変更することで，ゲスト OS 内の時間の流れを変える．この現象を利用し，タイマ割り込みの配送後のタイムスライスの長さを推測することで，CPU スケジューリングポリシーを変更することができる[1]．

FoxyTechnique は次の特徴を持つ．

- **OS カーネルの変更が不要:** FoxyTechnique では，既存の OS カーネルを変更せずに新たな資源管理ポリシーの疑似的な導入が可能である．仮想デバイスの振る舞いを制御するモジュールは VMM で動作するため，ゲスト OS とは完全に独立している．そのため，カーネルを変更せずに新たな資源管理ポリシーを導入することができ，Windows 等の商用 OS に対しても原理上適用可能である．これにより，OS ベンダが正式にサポートを行う前に，所望の資源管理ポリシーを擬似的に導入することができる．以下，仮想デバイスの振る舞いを制御するモジュールを **Foxy モジュール**と呼ぶ．

[1] これは FoxyTechnique の概念を説明するための例である．実際に FoxyTechnique を使って汎用 OS の CPU スケジューリングポリシーを変更するときには，タイマ割り込みの配送率を変更するだけでなく，他のことも考慮せねばならない可能性がある．

- **ポリシーの移植性の向上:** 特定の OS に依存する知識を使わずに Foxy モジュールを実装することで，1 つの資源管理ポリシーを多くのゲスト OS に適用することが可能となる．ユーザレベルの資源管理ポリシー変更手法と比べても，システムコール等のインターフェースの変更が不要であるため，移植性が高いと考えられる．本論文では，実際に単一の Foxy モジュールが様々な資源管理ポリシーの動作を変更することを示す．たとえば，4 つの異なるディスクスケジューリングポリシーの動作を単一の Foxy モジュールが所望のものに変更していることを示す．しかし，ある状況下では，FoxyTechnique を用いても所望の資源管理ポリシーを導入できない場合がある．これに関しては 5.1 節で述べる．

- **パフォーマンス情報の正確さ:** 資源管理ポリシーの中にはパフォーマンス情報を収集するものがある．パフォーマンス情報の正確さは，そのポリシーの動作に影響する．たとえば，TCP Vegas はパケットのラウンドトリップタイム (RTT) 情報を収集して輻輳ウィンドウサイズを決定する．もし，RTT が正しくなければ，TCP Vegas は正しく動作しない．Foxy モジュールは VMM 内で動作するので，仮想化によるオーバーヘッドが生じるゲスト OS よりも正確にパフォーマンス情報を取得できる．そのため，パフォーマンス情報を用いる資源管理ポリシーは，ゲスト OS へ直接導入するよりも，FoxyTechnique で導入する方がよい場合がある．実際，TCP Vegas 用の Foxy モジュールはゲスト OS 上で動作する TCP Vegas と FoxyTechnique で導入した TCP Vegas とを比べると，FoxyTechnique で導入した TCP Vegas の方が高いスループットをもたらす．

FoxyTechnique の有効性を示すために，2 つの Foxy モジュールを Xen 3.0.2-2 上に実装した．1 つめは *FoxyVegas* である．FoxyVegas は TCP Vegas [44] を作り出す Foxy モジュールである．TCP Vegas は輻輳ウィンドウを決める TCP 輻輳制御ポリシーの一種である．広く知られている TCP Vegas を Case Study とすることで，FoxyTechnique の有用性を示す．実際，資源管理ポリシーの導入に関する既存研究のいくつかは，その有用性を示すために TCP Vegas を Case Study として使用している [33, 34]．FoxyVegas では，TCP はフロー制御と輻輳制御に基づいてパケットのウィンドウサイズを決めているという知識を利用している．そこで，FoxyVegas は仮想ネットワークインターフェースカード (NIC) の振る舞いを変更する．仮想 NIC に届けられる ACK パケットの内容を改ざんすることで，データの受信ホスト

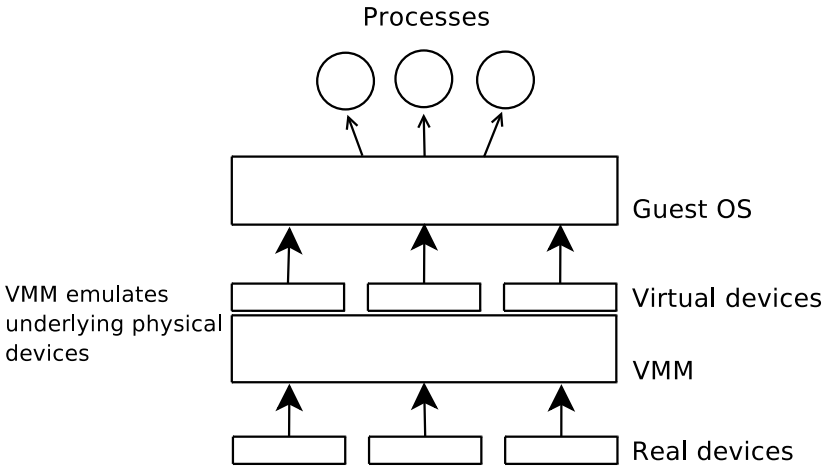
が高負荷であるという状況をゲスト OS に見せる．これによりゲスト OS にウィンドウサイズを調整させることで，TCP Vegas の振る舞いを作り出す．実際に，Linux 上に搭載されている 8 つの TCP 輻輳制御ポリシーに FoxyVegas を適用すると，振る舞いが TCP Vegas に近くなり，その効果をもたらす．また，FoxyVegas は RTT をゲスト OS 上の TCP Vegas より正確に計測できるので，ゲスト OS 上の TCP Vegas よりも実機上で動作する TCP Vegas に近い振る舞いをする．

2 つめは *FoxyIdle* である．FoxyIdle はアイドル時間 (Idletime) スケジューリング [9] を作り出す Foxy モジュールである．FoxyIdle はディスク読み込み要求に対してアイドル時間スケジューリングを行う．アイドル時間スケジューリングでは，バックグラウンドタスクによるフォアグラウンドタスクのスループット劣化を防ぐために，バックグラウンドタスクのディスク読み込み要求がフォアグラウンドタスクのそれと競合した場合，バックグラウンドタスクの要求を止めることで，ディスク帯域の競合を避ける．FoxyIdle では，ディスク読み込み要求を発行したプロセスは OS がデータを届けるまでブロックされるという OS の基本的な振る舞いの知識を利用する．そこで，FoxyIdle は仮想ディスクコントローラの振る舞いを変更し，ディスクがいつまでもデータをシークしていると偽った状況をゲスト OS に見せる．このシーク時間を調整することで，アイドル時間スケジューリングの動作を作り出している．実際に Linux 上に搭載されている 4 つすべてのディスクスケジューリングポリシーに FoxyIdle を適用すると，その振る舞いがアイドル時間スケジューリングに近いものになり，バックグラウンドプロセスによるフォアグラウンドプロセスのスループット劣化を避けている．

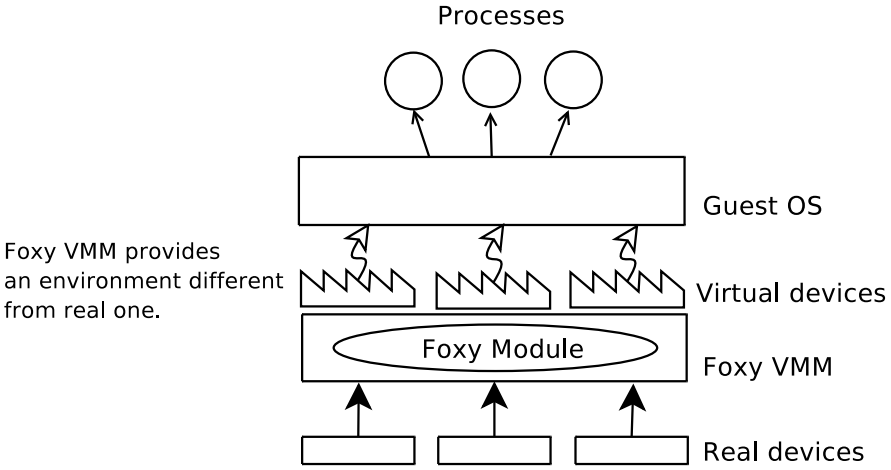
本章では，まず FoxyTechnique の概要について述べ，FoxyTechnique の実装について述べる．次に，ケーススタディである FoxyVegas と FoxyIdle の設計，実装および実験について述べ，本章をまとめる．

5.1 FoxyTechnique

カーネルを修正することなく資源管理ポリシーを擬似的に導入するために，FoxyTechnique では仮想マシンモニタ (VMM) を利用する．図 5.1 に従来の VMM と FoxyTechnique を用いた VMM との違いを示す．以下，FoxyTechnique を用いている VMM を **Foxy VMM** と呼ぶ．両者の違いは VMM とゲスト OS とのやりとりにある．従来の VMM では，ゲスト OS に提供する仮想デバイスは物理デバイスのエミュレー



(a) Conventional virtual machine monitor



(b) Foxy-enabled virtual machine monitor

図 5.1: **The difference between conventional VMM and Foxy VMM.** These figures show virtualization of a conventional virtual machine monitor (VMM) and Foxy-enabled VMM. The conventional VMM emulates underlying physical hardware. In contrast, the Foxy VMM virtualizes physical hardware so that a guest OS is forced to change its own behavior differently from the real one.

ションのみを行う．Foxy VMM では，仮想デバイスは物理デバイスとは異なる振る舞いをする．これにより，ゲスト OS は従来の VMM 上での動作とは異なる動作をする．ゲスト OS の動作を所望のものにするために，Foxy VMM はゲスト OS の動作を推測しながら仮想デバイスを制御する．

5.1.1 Foxy VMM

仮想化環境では，3 章で述べたように，VMM は物理ハードウェアを制御するために特権モードで，ゲスト OS は非特権モードでそれぞれ動作する．ゲスト OS がメモリ管理ユニット (MMU) や I/O デバイスにアクセスするために特権命令を実行すると制御は VMM に移る．その後，VMM によってその要求が適宜処理される．そのため，VMM はゲスト OS の資源アクセスをすべて捕捉することができ，その処理も VMM が行うので，資源管理を調整することが可能である．

Foxy VMM は物理デバイスを単にエミュレートするだけでなく，その振る舞いを調整することで，ゲスト OS の動作を制御する．ゲスト OS の動作を所望のものにするために，Foxy VMM は OS の動作を推測しながら仮想デバイスの振る舞いを調整する．具体的には，3 章で述べたように，仮想デバイスの振る舞いに対するゲスト OS の動作についての知識を用いる．たとえば，ゲスト OS がタイマ割り込みをカウントすることでタイムスライスを計測しているという知識を用いて，タイマ割り込みの配送率を調整する．これにより，タイムスライスの長さを制御することができる．このように，実際とは異なる状況をゲスト OS に提供することで，所望の動作を作り出す．たとえば，以下のように仮想ハードウェアの振る舞いを変更することで実際とは異なる状況を作り出すことができる．

- **割り込み配送率の調節:** タイマといった割り込みのゲスト OS への配送を制御する．たとえば，タイマ割り込みを実際よりも多くゲスト OS へ送ると，ゲスト OS のクロックは実際よりも早く進む．逆に，タイマ割り込みを少くすると，クロックは実際よりも遅く進む．

- **割り込みの遅延，破棄:** 物理ハードウェアの起こした割り込みをゲスト OS へすぐに配送しない．その割り込みを遅めに配送，もしくは破棄をすることで，ゲスト OS の動作を変えることができる．たとえば，ネットワークパケット到着の OS への通知を遅延，破棄することで，TCP においてネットワーク帯域の制御が可能となる．到着の通知の遅延，破棄をゲスト OS はネットワークの輻輳として検出し，TCP の輻輳制御を開始する．結果として，パケットのウィンドウサイズは小さくなり，利用できる帯域は狭まる．

- **データレジスタ値の変更:** ゲスト OS や物理ハードウェアがデータレジスタに書き込んだ値を変更することで，ゲスト OS の動作を変更できる．たとえば，TCP において，パケットの送信時に使われるウィンドウサイズを制御で

きる．Foxy モジュール は，パケットに含まれるサーバの受信可能バッファサイズを変更した後で，ゲスト OS にパケットを配送する．ゲスト OS は，そのバッファサイズに合わせて，自身のウィンドウサイズを調整し始める．

新たな資源管理ポリシーを導入する際，Foxy モジュールはプロセスやファイルといった OS レベルの抽象化に関する情報を要することがある．たとえば，優先度に基づくディスクスケジューリングを導入する場合，Foxy モジュールはプロセスの優先度によってディスク I/O 要求を処理しなければならないので，“プロセス”という抽象化を理解する必要がある．しかし，3 章で述べたように，VMM レベルではこうした情報は不足している．これをセマンティックギャップ [82] と呼ぶ．

FoxyTechnique では，このセマンティックギャップを埋めるために既存の手法を用いる．Antfarm [80] ではプロセスを，Geiger [81] ではバッファキャッシュの状態を VMM で推測することができる．これら以外の抽象化情報はユーザレベルのプロセスを用いて Foxy モジュールに伝える．ここで，FoxyTechnique では，ユーザレベルのプロセスは不正な情報を伝えることはないと仮定している．もし，不正な情報が Foxy モジュールに通知された場合，他の仮想マシンに影響を及ぼすおそれがある．こうした問題は本論文の対象外とする．

5.1.2 議論

VMM を用いて資源管理ポリシーを導入するアプローチは必ずしも成功するとは限らない．まず第一に，3 章で述べたときと同様に，ゲスト OS が持つ元々の資源管理ポリシーと新たに導入するポリシーとが競合し得るからである．たとえば，ゲスト OS がアイドル時間スケジューリングを行っているとする．このゲスト OS に対して，デッドラインスケジューリングを導入することは難しい．なぜなら，バックグラウンドプロセスのディスクアクセス要求に Foxy モジュールがデッドラインを設定しても，ゲスト OS がいつまでもその要求を仮想ディスクドライブに発行しない可能性があるからである．これにより，Foxy モジュールはデッドラインを守って要求を発行することができないため，デッドラインスケジューリングの振る舞いを作り出すことができない．

また，特定の OS に依存する Foxy モジュールは様々な OS に適用することが難しいと考えられる．たとえば，Linux のカーネルデータを読み書きする Foxy モジュールを実装したとする．この Foxy モジュールは Linux に対しては所望の動作

をより高い精度で作り出すことができる．しかし，Windows といった他の OS に対しては適用することができない．FoxyTechnique では，導入したいポリシーに近い効果を得ることができる動作精度を保ちつつ，できるだけポリシーの移植性を高めることを目標としている．実際，5.3 章，5.4 章で示す Foxy モジュールは，所望のポリシーの効果をもたらしつつ，様々な資源管理ポリシーの動作を変更している．

ここで，FoxyTechnique で導入することができる資源管理ポリシーの範囲についても議論する．3 章で述べたのと同様，一般的にその範囲を明示することは難しい．なぜなら，導入できるポリシーはどのような OS のデバイスに対する動作を推測するか，またどのデバイスに対する OS の動作を用いるかに依存しているからである．適用範囲を明確にするために，様々な資源管理ポリシーを FoxyTechnique を用いて今後導入していく必要があると考えられる．本研究ではその最初のステップとして 2 種類の Foxy モジュールについて述べる．

さらに，Foxy モジュールの設計についても議論を行う．FoxyTechnique を用いて資源管理ポリシーを擬似的に導入するときには，まずポリシーの動作を詳細まで把握して，デバイスの振る舞いを変更することで導入可能か否かを判断する．この際，対象となる OS の知識を十分に有している必要があり，デバイスの振る舞いに対して OS がどのように動作するかを知っている必要がある．さらに，デバイスの振る舞いを変更したことによる OS の副作用も考慮する必要がある．たとえば，タイマ割り込みの配送率を変更することで，タイムスライスの長さを調整することが可能であるが，OS が保持するシステム時間が乱れると予想される．こうした OS への副作用も十分に考慮してデバイスの動作を変更する必要がある．また，移植性とポリシーの動作精度のトレードオフについても考慮しなければならない．上述の通り，もし対象の OS 固有の知識を用いて Foxy モジュールを実装すると，その Foxy モジュールは他の OS への適用は困難である．

5.2 実装

FoxyTechnique の有効性を示すために，2 つの Foxy モジュールを実装した．1 つめは FoxyVegas，2 つめは FoxyIdle である．それぞれ異なる資源管理ポリシーを対象としており，FoxyVegas では TCP 輻輳制御ポリシーを，FoxyIdle ではディスクスケジューリングポリシーを対象としている．これらの詳細は 5.3 章，5.4 章で述

べる．

5.2.1 Xen

実装を行う VMM として，Xen [62] を用いた．Xen は x86 アーキテクチャ上で動作する VMM である．Xen では，仮想化のオーバーヘッドを削減するために準仮想化 (paravirtualization) [96] と呼ばれる仮想化方式を採用している．準仮想化方式では，VMM は通常のハードウェアとは異なるインターフェースをゲスト OS に提供する．具体的には，複雑なエミュレーションを要するゲスト OS の処理をハイパーコール (hypercall) と呼ばれるシステムコールのようなインターフェースに修正する．これにより，仮想化のオーバーヘッドを削減するだけでなく，VMM の構造が複雑になることを防いでいる．しかし，通常のハードウェアとは異なるインターフェースを利用するため，準仮想化機能を利用するためには，ゲスト OS がハイパーコールを呼び出すようにカーネルコードを修正する必要がある．

また，Xen の特徴の 1 つとして，スプリットドライバ方式 [97] を採用していることが挙げられる．従来，ゲスト OS がデバイスにアクセスすると，そのリクエストは VMM に送信され，VMM がデバイスにアクセスしていた．そのため，VMM がデバイスドライバを保持する必要があり，様々なデバイスドライバを VMM 用に再開発する必要があった．スプリットドライバ方式では，ドライバドメインと呼ばれる特権 VM が物理デバイスへのアクセスを直接行う．ここで，物理デバイスに直接アクセスするデバイスドライバをバックエンドドライバと呼ぶ．他の VM (ゲストドメイン) がデバイスにアクセスすると，そのリクエストは VMM によってドライバドメインに送られ，ドライバドメインによってアクセスが行われる．ゲストドメインで動作しているデバイスドライバをフロントエンドドライバと呼ぶ．スプリットドライバ方式により，特権 VM 上で動作する OS の元々保持しているデバイスドライバを利用することができ，VMM 上に新たなデバイスドライバを開発する必要はない．また，ゲストドメインで稼働している OS によるデバイスアクセスを全て特権 VM で監視できるため，ファイアウォール等のセキュリティ機能を特権 VM 上で動作させることで，一括でアクセス管理をすることができる．また，VMM 上でデバイスドライバを保持する必要がないので，VMM の構造の複雑化を避けることができる．しかし，一旦，VMM がドライバドメインにリクエストを送信するのでオーバーヘッドが生じてしまう．そのため，現在このオーバーヘッドを削減する研究が活発に行われている [98, 99, 100, 101, 102].

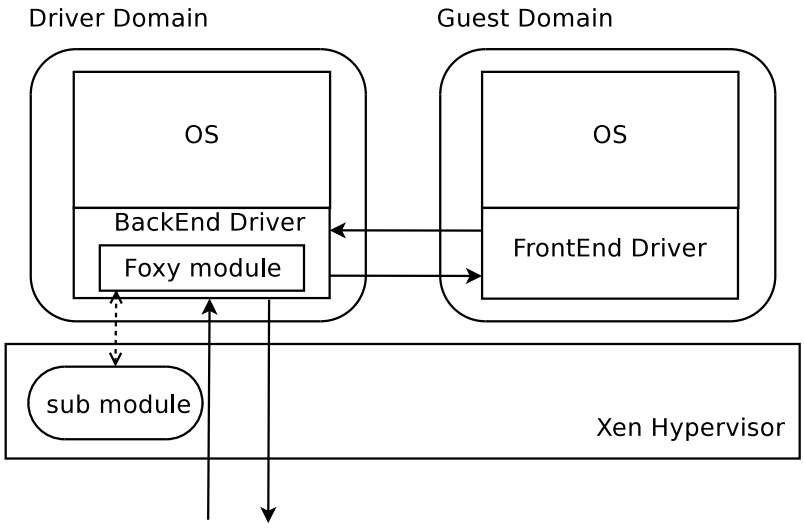


図 5.2: **Implementation.** This figure shows our implementation. A foxy module runs inside the back-end driver. It monitors requests from front-end drivers, and controls the response to the guest OS to change its behavior. It also communicates with a sub module running on Xen hypervisor, which bridges the semantic gap between guests and VMM.

FoxyVegas，ならびに FoxyIdle を Xen 3.0.2-2 上に実装した．これらの Foxy モジュールはバックエンドドライバ上で動作する．フロントエンドドライバからのリクエストやそれに通知するイベントを Foxy モジュールが変更することで，ゲストドメインで稼働する OS の動作を変更する．また，OS-VMM 間のセマンティックギャップを埋めるために，必要であれば Xen ハイパーバイザ内で動作するサブモジュールとハイパーコールを経由して通信を行い，OS に関する情報を取得する．ここで，Xen ハイパーバイザはメモリや CPU の仮想化を行うソフトウェアレイヤであり，Xen 上で稼働する VM すべてを管理する．サブモジュールは Xen ハイパーバイザ内で動作する Foxy モジュールと通信するモジュールである．

ここで，Foxy モジュールは準仮想化を活用することなく実現可能である．準仮想化方式を採用した Xen 上で実装を行ったが，実際に実現した Foxy モジュールは準仮想化に関する API を用いることなく実装されている．そのため，FoxyTechnique は VMware [86, 63] といった他の VMM にも原理上適用可能である．

5.3 Case Study: FoxyVegas

FoxyVegas は TCP 輻輳制御ポリシーの 1 つである TCP Vegas [44] の振る舞いを作り出す Foxy モジュールである．TCP 輻輳制御ポリシーは，ネットワークの輻輳時に使用される輻輳ウィンドウ (`cwnd`) の大きさを調整する．現在までに，様々な輻輳制御ポリシーが提案されている．TCP Vegas はより敏感にネットワークの輻輳を検出し，輻輳時においても高いスループットを保つことができるという特徴をもつ．

5.3.1 TCP Vegas

TCP Vegas はパケットのラウンドトリップタイム (RTT) を用いて `cwnd` を決定する．計測した RTT 値が大きくなると，TCP Vegas は輻輳が起きているとみなして `cwnd` を小さくする．一方，RTT 値が小さくなると，`cwnd` を大きくする．RTT 値が前回計測した値と差があまりなければ，`cwnd` を変更しない．

具体的には，輻輳回避フェーズでは以下の式によって `cwnd` が計算される．

$$cwnd(t+t_A) = \begin{cases} cwnd(t)+1 & (diff < \frac{\alpha}{base_rtt}) \\ cwnd(t) & (\frac{\alpha}{base_rtt} < diff < \frac{\beta}{base_rtt}) \\ cwnd(t)-1 & (\frac{\beta}{base_rtt} < diff) \end{cases}$$

ただし，

$$diff = \frac{cwnd}{base_rtt} - \frac{cwnd}{rtt}$$

ここで，rtt はラウンドトリップタイム，$base_rtt$ は最小ラウンドトリップタイム，α，β は定数である．このウィンドウサイズ制御が理想的に動作すると，ウィンドウサイズは一定値に固定される．これにより，パケットロスは発生せずに安定したスループットを得ることが期待できる．

5.3.2 設計

FoxyVegas は TCP Vegas の振る舞いを作り出すために，仮想 NIC (Network Interface Card) の動作を変更する．ここで，`cwnd` はカーネルで保持されるデータ構造なので，直接 VMM で制御することはできない．そこで，FoxyVegas はゲスト

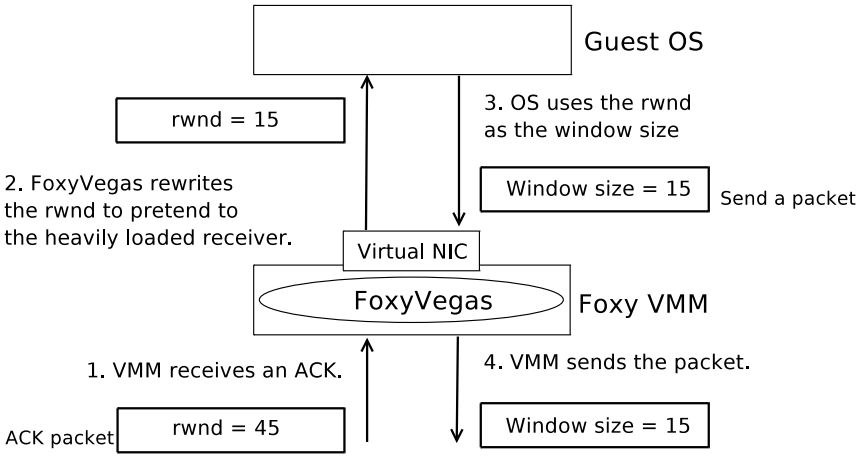


図 5.3: **FoxyVegas.** When a receiver's acknowledgment packet (ACK) arrives, FoxyVegas rewrites the advertised window size (rwnd) included in the ACK. The guest OS considers the receiver is loaded heavily, and changes the window size conforming with the receiver ACK.

OS のウィンドウサイズの調整を行うことで cwnd を変更したときと同様の効果を得る．そのために，FoxyVegas はゲスト OS に対して，データの受信側が高負荷であるという状況を見せる．これによって，ゲスト OS は自身のウィンドウサイズを調整し始める．この現象を利用して，FoxyVegas はゲスト OS のウィンドウサイズを TCP Vegas のアルゴリズムで計算される cwnd にする．

受信側を高負荷に見せかけるために，FoxyVegas では TCP に関する知識を利用する．TCP はウィンドウサイズを調整することでパケットの送信率を制御しており，2 種類の値を用いてウィンドウサイズを決めている．1 つは cwnd で，もう 1 つは受信側が送る広告ウィンドウ (rwnd) である．TCP はこの 2 つの値の小さい方をウィンドウサイズとして用いる [103]．上述のように，cwnd はカーネル内のデータ構造であるため，VMM から直接制御はできない．しかし，rwnd サイズ は ACK パケットに含まれているため，VMM で制御することができる．

FoxyVegas は仮想 NIC を制御して，rwnd の書き換える (図 5.3)．仮想 NIC が ACK パケットを受信したとき，FoxyVegas の計算した cwnd サイズを rwnd エントリに上書きする．ただし，実際の rwnd が FoxyVegas の計算した cwnd よりも大きいときのみ上書きする．FoxyVegas も TCP のウィンドウサイズの決め方 [103] に従って，rwnd が小さいときはゲスト OS のウィンドウサイズを rwnd にする．rwnd エントリの上書き後，ACK パケットをゲスト OS に配送する．すると，ゲスト OS は受信側が高負荷であると認識し，ウィンドウサイズを rwnd に設定す

る．これによって，ゲスト OS は FoxyVegas の計算した cwnd にウィンドウサイズを定め，パケットの送信を行う．

5.3.3 実装

FoxyVegas のプロトタイプをネットバックドライバ上に実装した．ネットバックドライバとはドライバドメインで動作する NIC 用のバックエンドドライバである．ネットバックドライバはゲスト OS によるネットワークの送受信を仲介する．そのため，ネットバックドライバでは，ゲスト OS が他ホストに送信するメッセージと他ホストからゲスト OS に送信されたメッセージをすべて把握することができる．具体的には，ゲスト OS のパケット送信時には，ゲスト OS 側でイーサパケットを作成し，そのパケットをネットバックドライバに送信する．ゲスト OS 宛のパケットを物理 NIC が受信したときには，ネットバックドライバは届いたパケットからイーサパケットを作り，ゲスト OS に送信する．

TCP Vegas のアルゴリズムをネットバックドライバ内で動作させるために，プロトタイプでは TCP パケットの RTT をネットバックドライバ内で計測する必要がある．TCP Vegas では cwnd を決定するときに TCP パケットの RTT を利用するためである．しかし，ネットバックドライバ内では IP レベルのパケットしか扱えない．そこで，プロトタイプでは送信されるイーサパケットを解析し，TCP ヘッダを取り出す．そして，TCP ヘッダに含まれるシーケンス番号を記録する．また，シーケンス番号と一緒にパケットの送信時刻を記録する．そして，ネットバックドライバ上で受信したイーサパケットから ACK パケットを探し出す．ACK パケットに含まれる番号に従って，受理されたパケットの番号と現在時刻から RTT を計算する．

TCP Vegas のアルゴリズムは Linux 2.6.16 上の実装を元にしている．α や β の値も同じ値を使用している．

5.3.4 実験

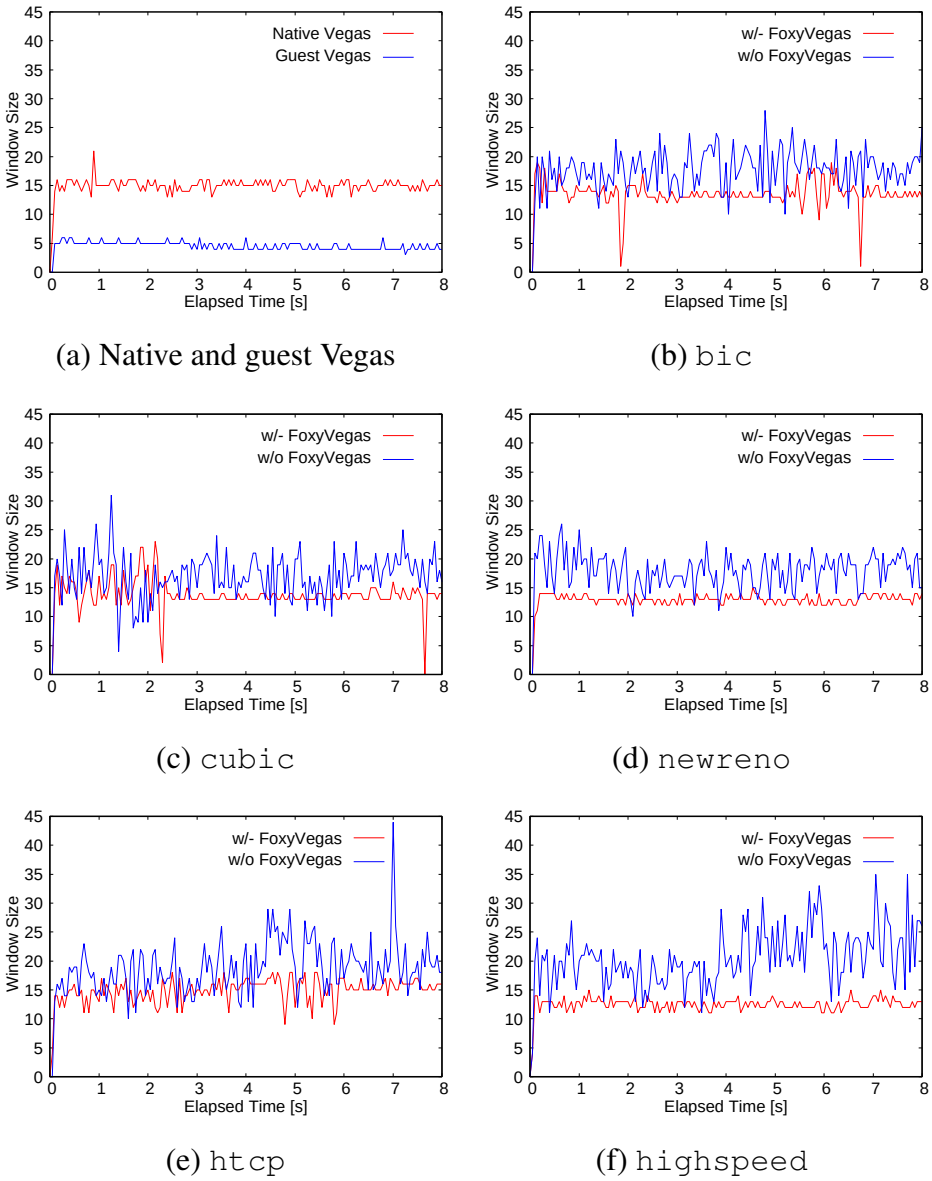
FoxyVegas の有効性を検証するために，3 台の PC を用いて実験を行った．用いた PC は Pentium 4 2.4 GHz の CPU，512 MB のメモリ，IDE のハードディスクを備えている．3 台の PC はそれぞれ送信者，受信者，そしてルータとして動作し，送信者と受信者はルータにギガビットイーサネットで接続されている．FoxyVegas

は送信者上で動作し，Xen の各ドメインでは Linux 2.6.16 が稼働している．制御ドメインには 256 MB のメモリ量を，ゲストドメインには 128 MB のメモリ量を割り当てている．ルータと 受信者上では Linux 2.6.16 が稼働している．

FoxyVegas がゲスト OS の動作を TCP Vegas に変更できることを検証するために，Linux 上に実装されている 9 つすべての輻輳制御ポリシーに対して FoxyVegas を適用した．用いたポリシーは，1) Binary Increase Congestion control (`bic`)， 2)CUBIC TCP (`cubic`)，3) NewReno (`newreno`)，4) H-TCP (`htcp`)，5) High Speed TCP (`highspeed`)，6) TCP-Hybla (`hybla`)，7) TCP Westwood+ (`westwood`)，8) Scalable TCP (`scalable`)，そして 9) TCP Vegas (`vegas`) である．本論文では，Foxy モジュールの移植性の高さを示すために，様々なポリシーに対する FoxyVegas の有効性を示す．送信者と受信者は Discard 通信 [104] を行う．Discard 通信とは，ただデータを送信するだけの通信であり，受信側では届いたデータに対して特別な処理はしない．また，ルータは token bucket filter [105] を用いて，帯域幅 2000 KB/s，遅延 10 ms，キューサイズ 6000 KB というボトルネックをエミュレートした．

実験結果を図 5.4 に示す．横軸に経過時間，縦軸にパケットのウィンドウサイズをとっている．比較として，実機で稼働した Linux の TCP Vegas (以下，*Native Vegas*) のウィンドウサイズと，ゲストドメイン上で稼働した Linux の TCP Vegas (以下，*Guest Vegas*) のウィンドウサイズを図 5.4(a) に示す．図より，`vegas` を除くすべての TCP 輻輳制御ポリシーの振る舞いが TCP Vegas の振る舞いに近づいていることがわかる．FoxyVegas は TCP Vegas を採用しているゲスト OS の振る舞いを変更することができない．これは FoxyVegas が算出するウィンドウサイズよりゲスト OS が保持する `cwnd` の値の方が小さいため，ウィンドウサイズが `cwnd` になってしまう．

ここで，FoxyVegas は Guest Vegas よりも Native Vegas に近い動作を作り出していることに注目する．FoxyVegas は物理デバイス上で動作するため，FoxyVegas は Native Vegas と同様に正確に RTT を計測することができる．図 5.5 に Native Vegas, Guest Vegas そして FoxyVegas が観測した RTT の分布を示す．仮想化によるオーバーヘッドが原因で，Guest Vegas の観測した RTT 値は他の 2 つと比べるとやや長い．そのため，Guest Vegas はこの仮想化のオーバーヘッドをネットワークの輻輳とみなし，`cwnd` の値を減らす．一方，FoxyVegas は仮想化によるオーバーヘッドをあまり被ることなく RTT を計測できるので，Guest Vegas よりも大きな `cwnd` サイズを設定しようとする．そのため，送信者と受信者間のスループットを計測したところ，Guest Vegas よりも FoxyVegas の方がスループットが高い (表 5.1).



(a) Native and guest Vegas (b) `bic`

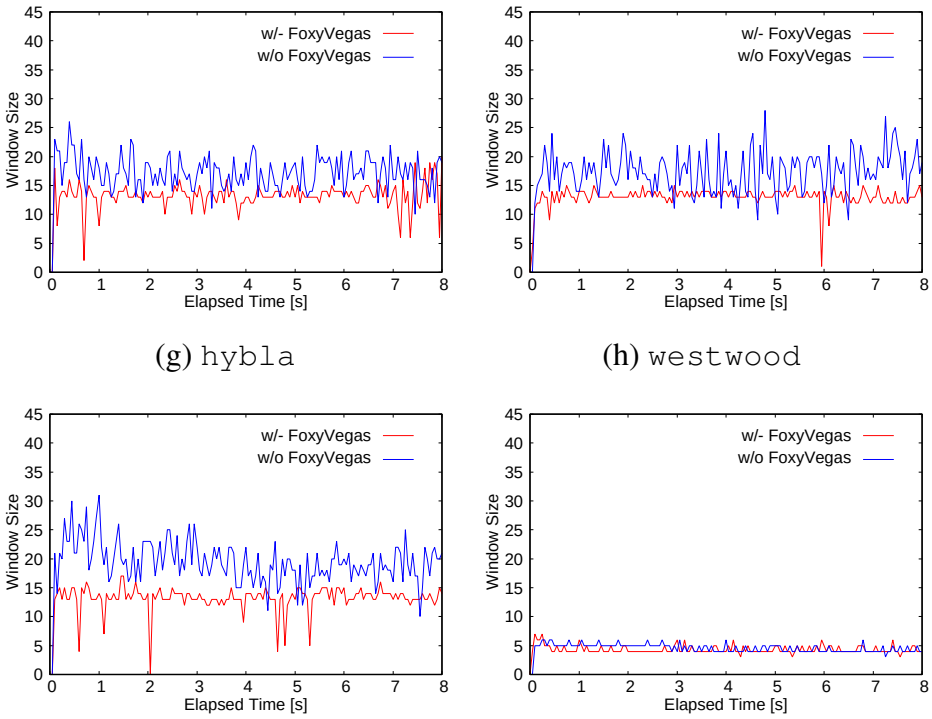
(c) `cubic` (d) `newreno`

(e) `htcp` (f) `highspeed`

図 5.4: **Comparison of the window size.** The figure shows the aggregate window size for various TCP congestion control algorithms whether FoxyVegas is used or not. The experiment uses nine congestion control algorithms.

FoxyVegas が TCP Vegas の効果をもたらすことを示すために，ネットワーク輻輳時のスループットを計測した．送信者は 400 MB のデータを受信者に送信する．ルータは上述と同様の輻輳をエミュレートしている．

結果を表 5.3.4 に示す．表より，TCP Vegas を除くポリシーに対しては，FoxyVegas によってネットワーク輻輳下において高いスループットを達成していることがわ



(g) hybla (h) westwood

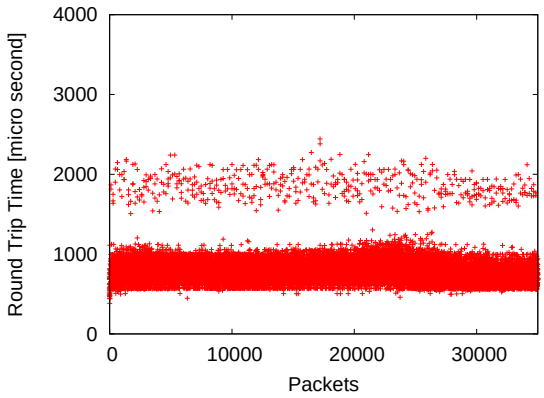
(i) scalable (j) vegas

図 5.4: Comparison of the window size.

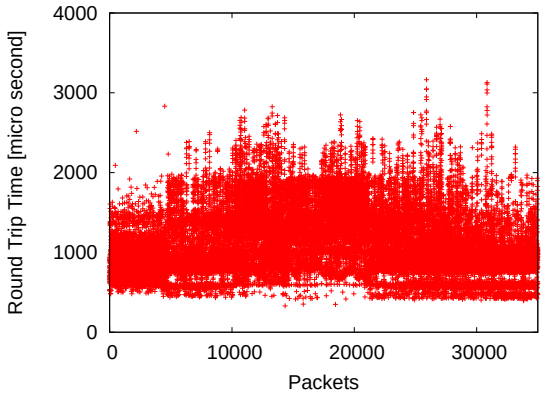
表 5.1: **Comparison of throughput for various configurations of TCP Vegas.** The table reports the aggregated throughput for various configurations of TCP Vegas. The experiments use three Vegas. The first is Linux TCP Vegas running on the physical machine (Native Vegas). The second is Linux TCP Vegas running on the Xen guest domain (Guest Vegas). The last is FoxyVegas which triked Linux BIC. BIC is the Linux default congestion control algorithm.

Benchmark	Throughput [MB/s]	Decreasing rate
Native Vegas	42.442	—
Guest Vegas	23.831	43.85%
FoxyVegas	33.692	20.62%

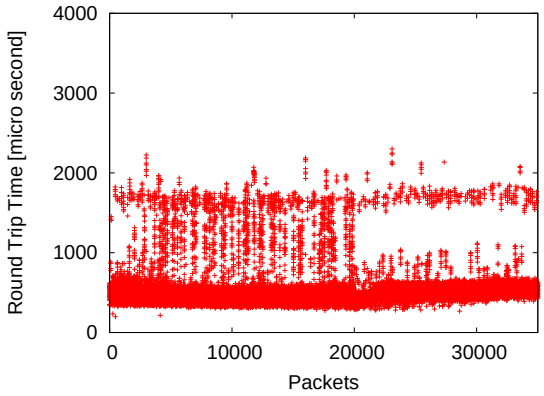
かる．FoxyVegas が TCP Vegas に対して動作するとスループットが下がる理由は FoxyVegas と TCP Vegas が競合するためである．この理由は上述の通りである．



(a) Native Vegas



(b) Guest Vegas



(c) FoxyVegas

図 5.5: **Comparison of RTT fluctuations.** These graphs show the distribution of RTTs observed by native Vegas, guest Vegas and FoxyVegas. The tendency of the distribution of FoxyVegas is similar to one of native Vegas. The RTTs of guest Vegas is larger than native Vegas and FoxyVegas.

表 5.2: **Comparison of throughput in network congestion.** The table reports the improvement rate of throughput when we used FoxyVegas. In almost cases, FoxyVegas makes guest Linux achieve higher throughput under network congestion (a bottleneck of 2000 KB/s, a delay of 10 ms and queue size of 6000 KB).

Tricked Algorithm	bic	cubic	newreno	htcp	highspeed	hybla	westwood	scalable	vegas
w/o FoxyVegas [KB/s]	789	921	830	1063	1030	1058	902	732	1369
w/- FoxyVegas [KB/s]	1281	1270	1275	1367	1291	1317	1326	1328	1235
Ratio	62.3%	37.8%	53.6%	28.6%	25.3%	24.5%	46.9%	81.5%	−9.8%

5.4 Case Study: FoxyIdle

FoxyIdle はディスク I/O 要求に対してアイドル時間スケジューリング [9] の振る舞いを作り出す Foxy モジュールである．アイドル時間スケジューリングは遊休資源を利用して動作すべきウィルススキャンや SETI@home などのバックグラウンドタスクを制御する．アイドル時間スケジューリングは，ユーザのフォアグラウンドタスクのスループットを劣化させないように，バックグラウンドタスクの要求を実行する．

5.4.1 アイドル時間スケジューリング

バックグラウンド要求を実行するために，アイドル時間スケジューリングでは**横取り遅延期間** (preemption interval) と呼ばれる遅延を用いる (図 5.6)．横取り遅延期間はフォアグラウンド要求の処理が終了した後に始まる期間である．その期間中，フォアグラウンド要求は処理されるが，バックグラウンド要求の処理は遅延される．横取り遅延期間が終了すると，はじめてバックグラウンド要求が処理される．

横取り遅延期間の長さは，遊休資源をより積極的に使うか，それともフォアグラウンドタスクのスループットを保つかというトレードオフによって決める．も

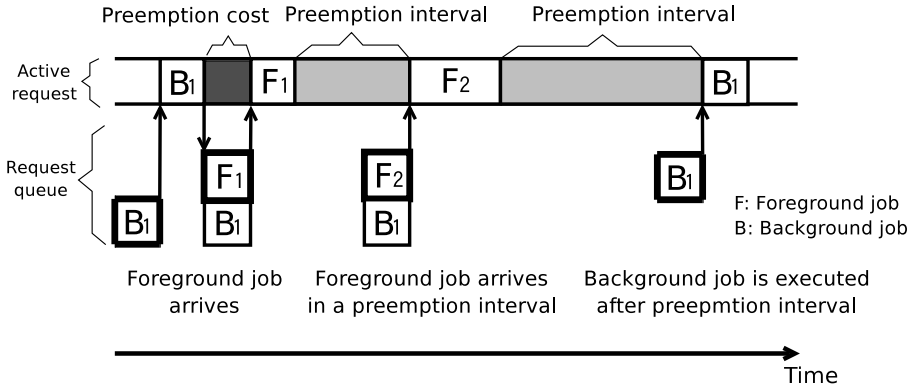


図 5.6: **Idletime scheduling.** The figure shows the behavior of the idletime scheduling. A preemption interval starts whenever a foreground request finishes. While the preemption interval is active, the background requests are not executed. The background jobs are started only when a preemption interval expires.

し横取り遅延期間が長ければ，フォアグラウンドタスクのスループットは保たれるが，資源が遊休する時間は長くなる．一方，横取り遅延期間が短かければ，その逆が起こる．

5.4.2 設計

アイドル時間スケジューリングの振る舞いを作り出すために，FoxyIdle は仮想ディスクコントローラを制御する．バックグラウンド要求を止めるために，FoxyIdle は仮想ディスクが要求されたデータをいつまでもシークしているという状況をゲスト OS に見せる．このシーク時間の間，ゲスト OS はバックグラウンドプロセスをブロックする．ここで FoxyIdle は，ディスクから割り込みが送られるまで OS はデータを待つプロセスを待機状態にしておくという知識を利用している．シーク時間を調節することで，横取り遅延期間の間，バックグラウンドプロセスを止めておくことができる．

データをシークする振りをするために，FoxyIdle は物理ハードディスクへのリクエスト発行を遅らせる．バックグラウンドプロセスの読み込み要求の処理を遅らせることで，データ読み込み完了の通知を遅らせる．これにより，読み込み要求を発行したプロセスの停止時間を長くする．FoxyIdle の振る舞いを図 5.7 に示す．バックグラウンドプロセスがディスク読み込み要求を発行すると，FoxyIdle は横取り遅延期間が有効である間，その要求を止めておく．その間，ゲスト OS はディ

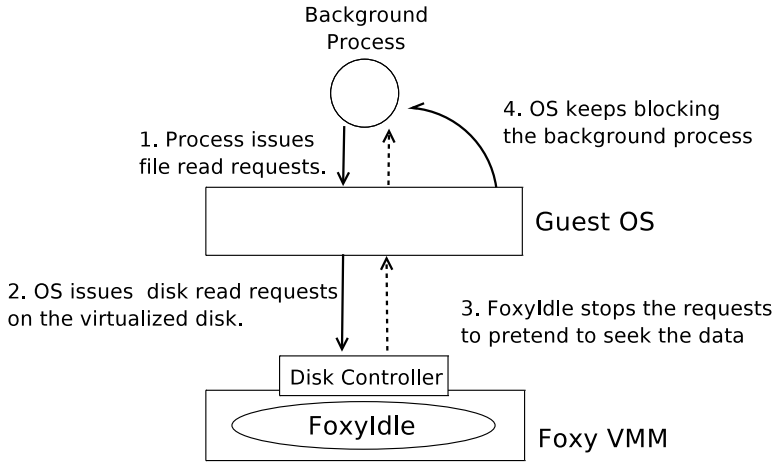


図 5.7: **FoxyIdle.** FoxyIdle controls background disk read requests. When a background process issues disk read requests via the guest OS, FoxyIdle delays responses for the requests. The guest OS considers the data is sought by the virtual disk drive, and keeps blocking the background process.

スクが要求したデータをシークしていると思い込み，バックグラウンドプロセスをブロックし続ける．FoxyIdle は横取り遅延期間が終了すると，止めていた要求をディスクに発行する．すると，ゲスト OS はディスクから割り込みを受け，バックグラウンドプロセスのブロックを解除する．なお，FoxyIdle は，フォアグラウンドプロセスの要求を受け取ると，そのまま要求を発行して横取り遅延期間を開始する．

バックグラウンド要求を遅延させるために，FoxyIdle は 1) ディスク読み込み要求を発行したプロセスはどれか，2) そのプロセスはバックグラウンド属性か否かを識別する必要がある．これらは OS レベルの抽象化情報であり，VMM 内で認識することは難しい．

バックグラウンド要求の識別を実現するために，FoxyIdle では，VMM 内でプロセスを認識する手法を用いる．さらにそれを活用して，上述の 1 と 2 の認識を可能にする手法をそれぞれ用いる．まず，VMM 内でプロセスを認識するために，FoxyIdle は Antfarm [80] を用いる．Antfarm は VMM 内で “プロセス” の認識を可能にする技術である．Antfarm では，VMM 内でプロセスの仮想アドレス空間を追跡する．Intel x86 上では，制御レジスタの 1 つである CR 3 にページテーブルの実アドレスを格納する．そのため，ゲスト OS は仮想アドレス空間を切替える際に CR 3 の値を書き換える．この書き換えは特権命令なので，VMM は CR 3 の値の更新を知ることができ，CR 3 値をチェックすることで，VMM は仮想アドレスを

追跡することができる．FoxyIdle ではこの CR 3 の値をプロセス ID に相当するものとみなす．

次に，ディスク読み込み要求とそれを発行したプロセスとを結びつけるために，FoxyIdle では context association [80] を用いる．この方式では，ディスク読み込み要求を VMM が受け取ると， その時点の CR 3 値に結び付けられたプロセスがその要求を発行したとみなす．しかし，この方式は結びつけを誤る場合がある．たとえば，あるプロセスがディスク読み込み要求を発行したとする．その要求が仮想ディスクに届くまえに別プロセスにコンテキストスイッチが行われると，この方式では誤った結びつけをする．しかし，FoxyIdle を構築するには十分な精度である．実験を行ったところ，FoxyIdle はアイドル時間スケジューリングの効果をもたらしている．

最後に，プロセスとバックグラウンド属性との結びつけを行うために，バックグラウンドプロセスは FoxyIdle に UDP メッセージを送信する．FoxyIdle はそのメッセージを受け取ると，現在の CR3 値とバックグラウンド属性とを結びつける．

5.4.3 実装

FoxyIdle のプロトタイプをディスクバックドライバ，ネットバックドライバ，Xen ハイパーバイザ上に実装した．ディスクバックドライバとはドライバドメインで動作するハードディスク用のバックエンドドライバである．ディスクバックドライバはゲスト OS によるディスクアクセスを仲介する．そのため，ゲスト OS のすべてのディスクアクセスはディスクバックドライバ内で把握可能である．具体的には，ゲスト OS がハードディスクからデータを読み込むときには，ゲスト OS 側でディスク読み込み要求を作成し，それをディスクバックドライバに送信する．ディスクバックドライバが読み込み要求を受け取ると，要求されたデータをハードディスク内から取得して，そのデータと共に読み込み完了をゲスト OS に通知する．ゲスト OS がディスク書き込みを行うときには，ゲスト OS が汚れブロックを作成し，それをディスクバックドライバに送信する．ディスクバックドライバが汚れブロックを受信すると，該当する箇所にデータを書き込む．

バックグラウンド要求を遅延させるために，ディスクバックドライバ内でバックグラウンド要求を格納するキューを実装した．横取り遅延期間が有効である間にバックグラウンド要求がディスクバックドライバ内に到着すると，FoxyIdle はそのリクエストをキューに格納する．フォアグランド要求が到着すると，そのま

ま物理ハードディスクに要求を発行する．横取り遅延期間が終了すると，FoxyIdle はキューからバックグラウンド要求を取り出し物理ハードディスクに要求を発行する．

ディスクバックドライバ内でバックグラウンド要求を認識するために，まず Antfarm を Xen ハイパーバイザ内にサブモジュールとして実装した．さらに，context association を実現するために，ハイパーコールを追加し，ディスクバックドライバ内でゲスト OS が現在実行しているプロセスの CR3 の値を取得できるようにした．ディスク読み込み要求がディスクバックドライバに届くと，FoxyIdle はハイパーコールを呼び出し CR 3 の値を取得する．取得した CR 3 の値がバックグラウンド属性であれば，横取り遅延期間が有効であるかを確認する．有効であれば，届いたディスク読み込み要求をキューに格納する．

バックグラウンド属性通知に関する UPD メッセージの送受信に関する実装を Xen 内ならびにゲスト OS のユーザレベル上にて行った．バックグラウンド属性を通知する UDP メッセージを受信するために，FoxyIdle はネットバックドライバ上で UDP パケットを監視している．特定のポートに UDP パケットが到着すると，上述したハイパーコールを用いてゲスト OS が実行している CR 3 の値を取得し，その値をバックグラウンド属性と結びつける．また，バックグラウンドプロセスが UDP メッセージを送信できるように，現在の実装では，そのプロセスの `libc_main_start()` を上書きを行っている．Linux では，環境変数 `LD_PRELOAD` を用いることでライブラリ関数を上書きすることができる．この機能を用いることで，バックグラウンドプロセスは自身のタスクを実行する前に UDP メッセージを FoxyIdle に送信することができる．これによりアプリケーションのソースコードを改変することなく FoxyIdle に UDP メッセージを送信することができる．Linux のライブラリのプリロード機構により，アプリケーションが子プロセスを生成したときも，その子プロセスの `libc_main_start()` を上書きし，UDP メッセージを FoxyIdle に送信する．

5.4.4 実験

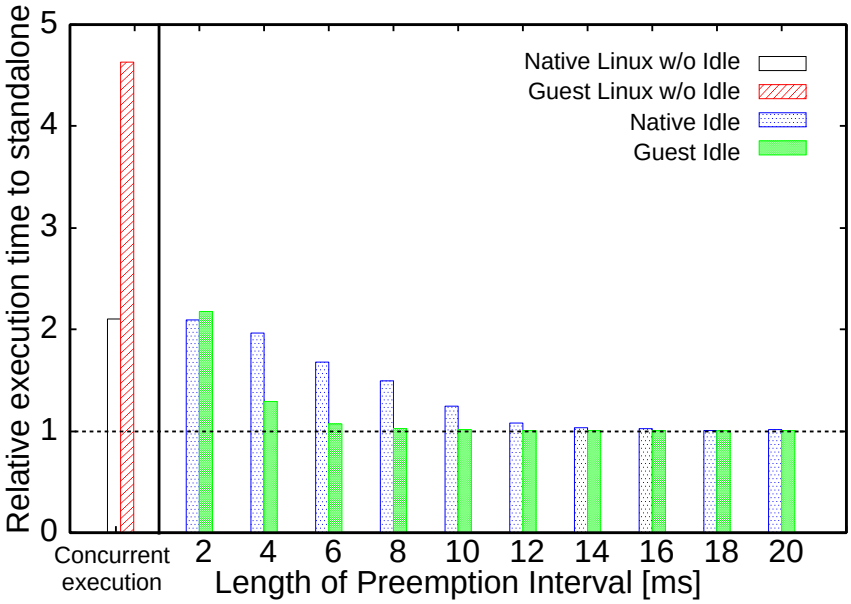
FoxyIdle の有効性を検証するために実験を行った．実験で用いた PC は，Pentium D 3.0 GHz の CPU，1 GB のメモリ，SATA で接続されたハードディスクを備え，Xen 3.0.2-2 が稼働している．各ドメインでは Linux 2.6.16 が動作しており，特権ドメインに 512 MB のメモリ量を，ゲストドメインに 128 MB のメモリ量を割り

当てている．

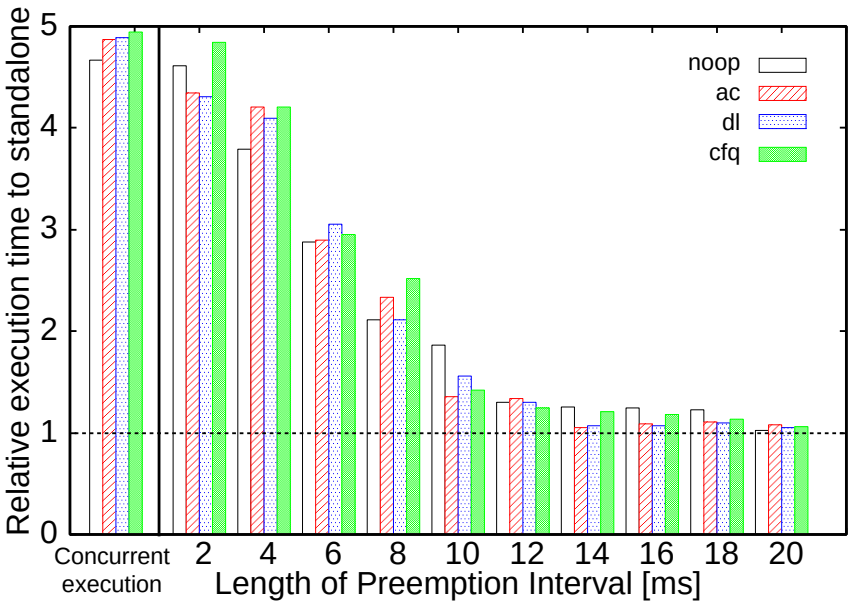
FoxyIdle がゲスト OS の動作をアイドル時間スケジューリングに変更できることを検証するために，Linux で設定可能な 4 つのディスクスケジューリングポリシーすべてに対して FoxyIdle を適用した．用いたポリシーは 1) no operation (`noop`), 2)anticipatory (`ac`), 3) deadline (`dl`), 4) complete fairness queuing (`cfq`) である．FoxyVegas と同様に，Foxy モジュールの移植性の高さを示すために，様々なポリシーを FoxyIdle が適切に変更できるかどうかを検証する．また，Linux 2.6.16 上にアイドル時間スケジューリングを実装し，FoxyIdle の振る舞いと比較した，本実験では，2 種類のベンチマークを用意した．1 つめは `Sequential`，2 つめは `Random` である．`Sequential` は 1 GB のファイルをシーケンシャルに読み込む．一方，`Random` は 1 GB のファイルをランダムに 10 回読み込む．それぞれのベンチマークをフォアグランドタスク，バックグラウンドタスクとして同時に動作させ，横取り遅延期間を変えながら，実行時間を計測した．

結果を図 5.8，5.9，5.10，5.11 に示す．横軸に横取り遅延期間の長さ，縦軸にはそれぞれのタスクを単独で動作させたときの実行時間で正規化した値をとっている．それぞれの図において，(a) はアイドル時間スケジューラを搭載した Linux が実機上で動作している (*Native Idle*) ときと，Xen のゲストドメイン上で動作している (*Guest Idle*) ときの結果である．そして，(b) は FoxyIdle を動作させたときの結果である．グラフの最左のプロットは FoxyIdle を動作させずにフォアグラウンドタスクとバックグラウンドタスクを同時に動作させたときの実行時間であり，*Concurrent execution* とラベル付けをしている．図 5.8，5.9 は両タスクとも `Sequential`，そして図 5.10，5.11 はフォアグラウンドタスクが `Random`，バックグラウンドタスクが `Sequential` のときの結果である．

これらの図から 2 つのことがわかる．1 つ目は FoxyIdle がアイドル時間スケジューリングに近い振る舞いを作り出しているという点である．具体的には，横取り遅延期間 が長くなるにつれて，FoxyIdle がフォアグランドタスクのパフォーマンス劣化を抑えている．2 点目は FoxyIdle がすべてのポリシーに対してアイドル時間スケジューリングの動作を作り出すことに成功している点である．アイドル時間スケジューリングでは，横取り遅延期間が長くなると，フォアグラウンドタスクの実行時間は短くなっていき，単独で実行させたときの実行時間に近づいていく．実際，図 5.8(a) と図 5.10(a) から，Native Idle と Guest Idle を用いて 横取り遅延期間 を 14 msec 以上に設定すると，フォアグラウンドタスクの実行時間を正規化した値は 1 に近づく．一方 FoxyIdle では，横取り遅延期間 を長く設定すれば

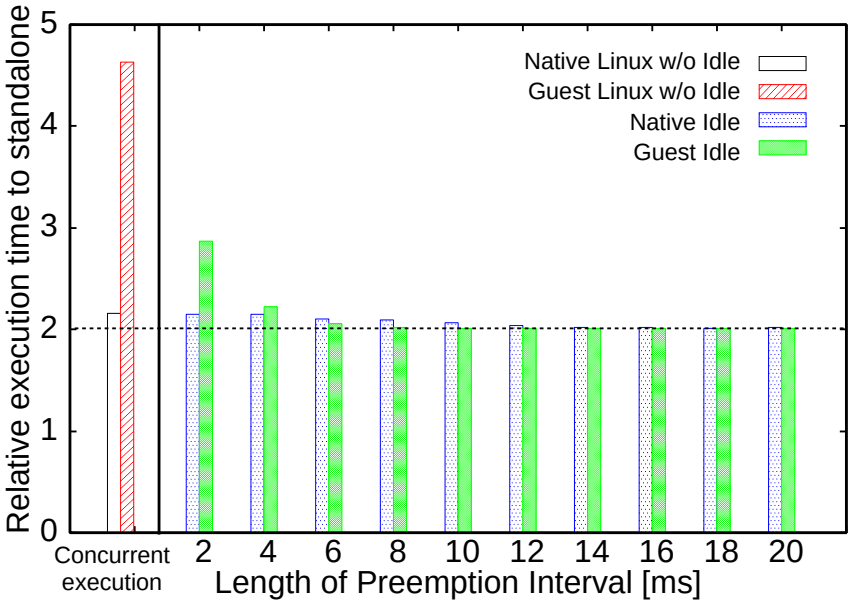


(a) Execution time of foreground `Sequential` on native Idle and guest Idle

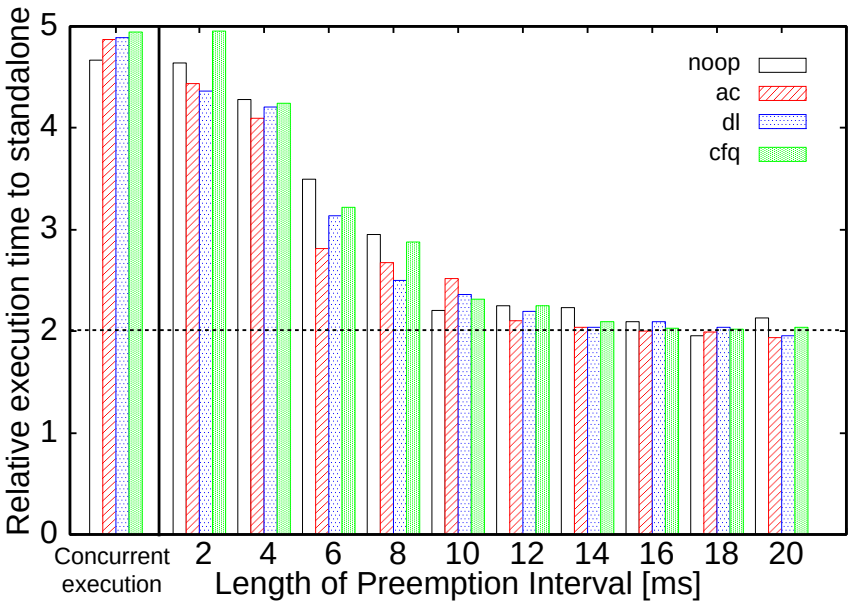


(b) Execution time of foreground `Sequential` on FoxyIdle

図 5.8: **Difference of the execution time of foreground `Sequential` with background `Sequential` on native Idle, guest Idle and FoxyIdle.** In each graph, the leftmost bars (labeled concurrent execution) plot the execution times when the idletime scheduler is disabled.

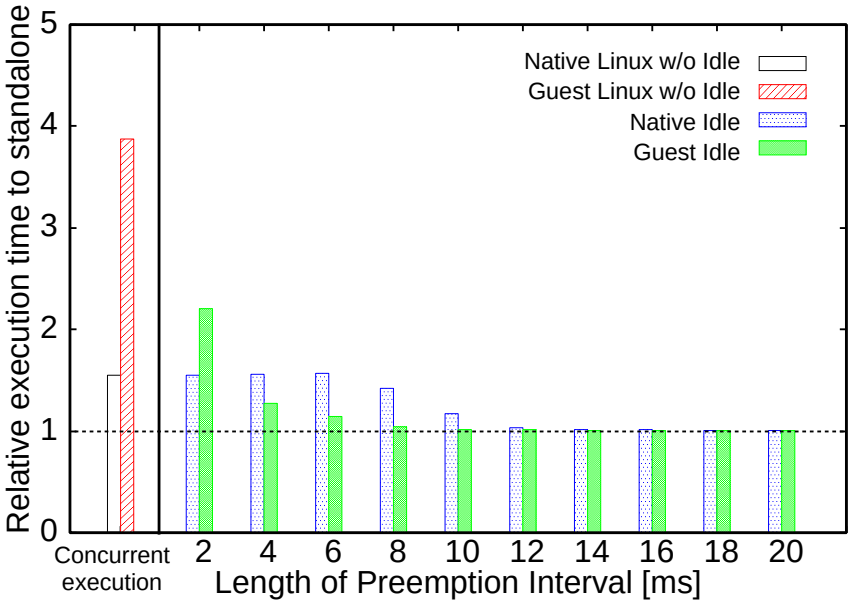


(a) Execution time of background `Sequential` on native Idle and guest Idle

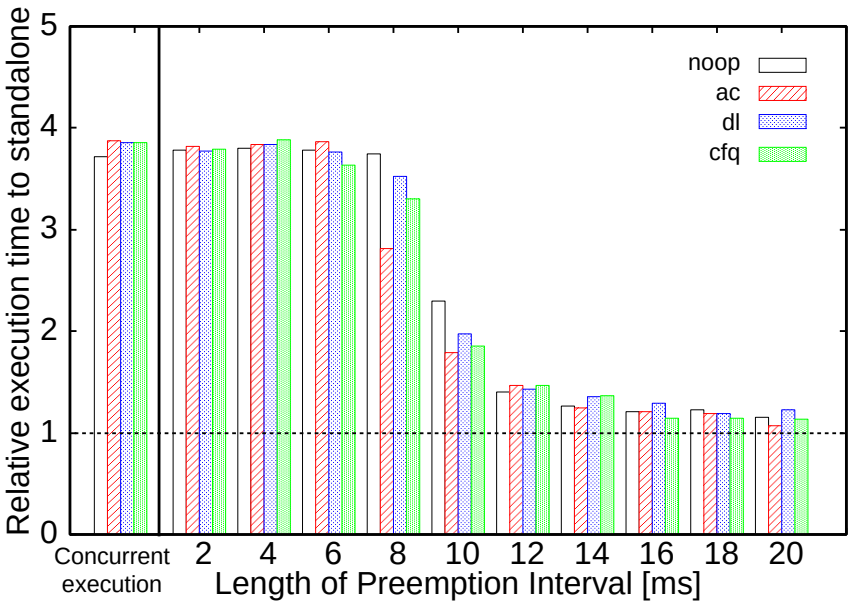


(b) Execution time of background `Sequential` on FoxyIdle

図 5.9: **Difference of the execution time of background `Sequential` with foreground `Sequential` on native Idle, guest Idle and FoxyIdle.** In each graph, the leftmost bars (labeled concurrent execution) plot the execution times when the idletime scheduler is disabled.

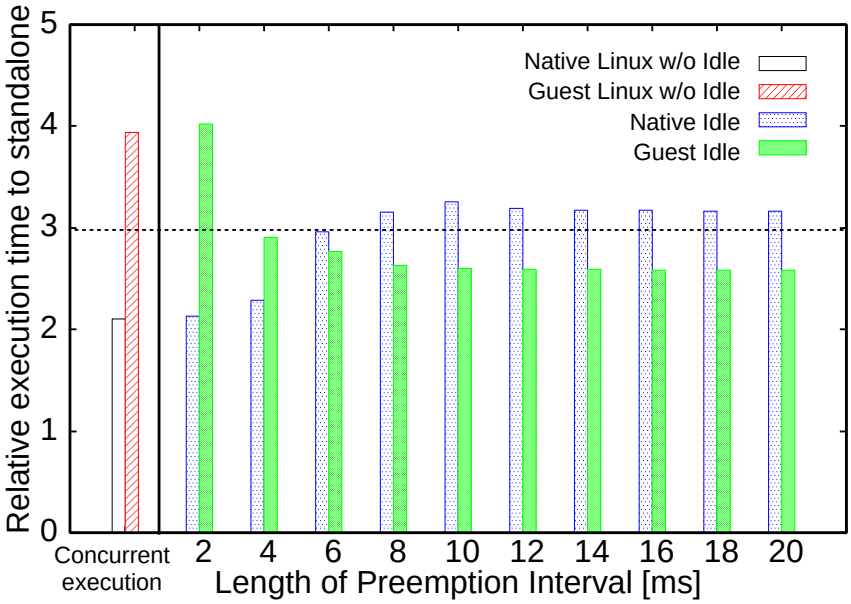


(a) Execution time of foreground `Random` on native Idle and guest Idle

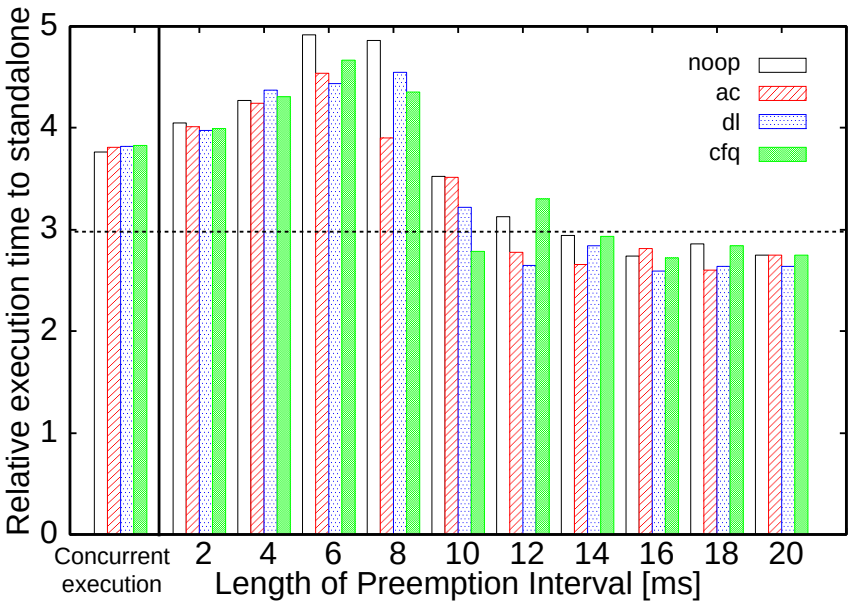


(b) Execution time of foreground `Random` on FoxyIdle

図 5.10: **Difference of the execution time of foreground `Random` with background `Sequential` on native Idle, guest Idle and FoxyIdle.** In each graph, the leftmost bars (labeled concurrent execution) plot the execution times when the idletime scheduler is disabled.



(a) Execution time of background `Sequential` on native Idle and guest Idle



(b) Execution time of background `Sequential` on FoxyIdle

図 5.11: **Difference of the execution time of background `Sequential` with foreground `Random` on native Idle, guest Idle and FoxyIdle.** In each graph, the leftmost bars (labeled concurrent execution) plot the execution times when the idletime scheduler is disabled.

するほど，フォアグランドタスクの値は 1 に近づく (図 5.8 (b)，5.10(b))．図 5.9(a) より，Native Idle と Guest Idle を用いて 横取り遅延期間 を長く設定すると，バックグラウンドタスクの実行時間は 2 倍になることがわかる．さらに，同様の傾向が FoxyIdle にも見られる (図 5.9(b))．フォアグラウンドタスクが `Random` であると，Native Idle を用いて 横取り遅延期間 を長くしていくと，バックグラウンドタスクの実行時間を正規化した値は 3 に近づいていく．一方，FoxyIdle や Guest Idle では，バックグラウンドタスクの実行時間は 3 以下にとどまっている．

またこの実験から，同じ長さの横取り遅延期間を用いたとき，Guest Idle の方が Native Idle よりフォアグラウンドタスクのパフォーマンス劣化を抑制していることがわかる．たとえば，図 5.8(a) より，横取り遅延期間 が 4 ms のとき，Native Idle 上のフォアグラウンドタスクの実行時間の増加が 1.96 倍であるのに対し，Guest Idle 上では 1.29 倍である．これは Guest Idle がバックグラウンド要求をブロックする際に用いるタイマーの精度が仮想化によるオーバーヘッドが原因で低下しているためである．実際にタイマの精度を計測するために，アイドル時間スケジューリングや FoxyIdle が算出したバックグラウンド要求をブロックする時間と実際にブロックした時間との差を計測した．横取り遅延期間 が 4 ms のときに計測を行い，結果を図 5.12 に示す．Linux のタイマ機能を使ったとしても，Native Idle でさえ，差の中央値は 1.16 ms である．Guest Idle では，差の中央値は Native Idle と FoxyIdle と比べると高く，6.83 ms である．結果として，Guest Idle はバックグラウンド要求をより長い間ブロックしてしまい，フォアグラウンドタスクのパフォーマンス劣化を防止している．FoxyIdle の差も Native Idle よりは高いが，Guest Idle と比べるとタイマの精度は高いので，Guest Idle ほどは過剰にバックグラウンド要求をブロックしない．

次に FoxyIdle が実際のアプリケーションに対して有用であるかを確認するために，新たに 2 種類のベンチマークを用意した．1 つ目が `Grep`，2 つ目が `Updatedb` である．`Grep` は Linux 2.6.16 のカーネルのソースコード，ならびにドキュメントから “submit_bio” という文字列を検索する．`Updatedb` は同じバージョンの Linux のソースコードとドキュメントのインデックス化を行う．ファイルの合計サイズは 260 MB である．この実験ではフォアグラウンドタスクを `Grep`，バックグラウンドタスクを `Updatedb` とした．両タスクを同時に実行し，横取り遅延期間を変更しながら実行時間を計測した．ゲストドメインで稼働している Linux が使用しているディスクスケジューリングポリシーはデフォルトの Anticipatory スケジューリングである．また，アイドル時間スケジューリングの効果をより際立たせるため

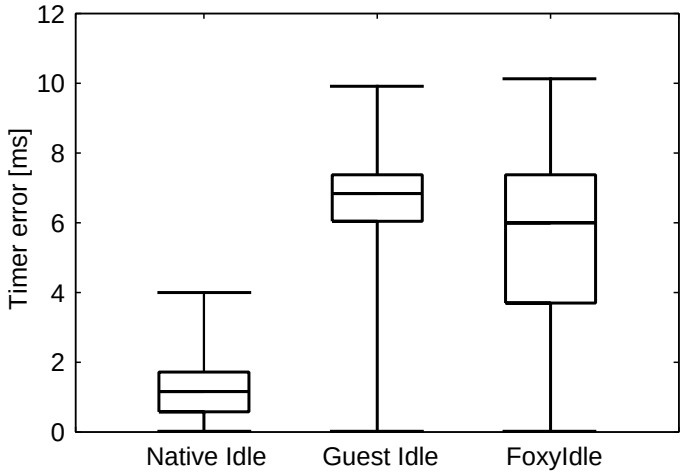


図 5.12: **Timer Errors of the idletime schedulers.** The figure shows the absolute values of the errors between the calculated interval and the actual interval. The calculated interval is an interval which the idletime scheduler or FoxyIdle calculated for stopping background requests. The actual interval is an interval for which background requests were actually stopped.

に，両タスクはそれぞれ異なる Linux カーネルのソースツリーの走査を行う．このようにすることで，それぞれのタスクがバッファキャッシュを共有することなく，ディスク読み込み要求を発行する．

表 5.3 に結果を示す．比較として，FoxyIdle やアイドル時間スケジューリングを用いないときの実行時間 (Concurrent execution) とタスクを単独で動作させたときの実行時間 (Standalone) も示す．結果より，Concurrent execution と比べると，FoxyIdle はすべての場合において両タスクの実行時間を短縮していることがわかる．Native Idle や Guest Idle もほぼ同じ結果である．実行時間を短縮している理由は FoxyIdle を用いることで，Concurrent execution 時よりもディスク競合を減少させているからである (図 5.13)．実験結果より，FoxyIdle がアイドル時間スケジューリングを動作を作り出し，その恩恵をもたらすことがわかった．

5.5 まとめ

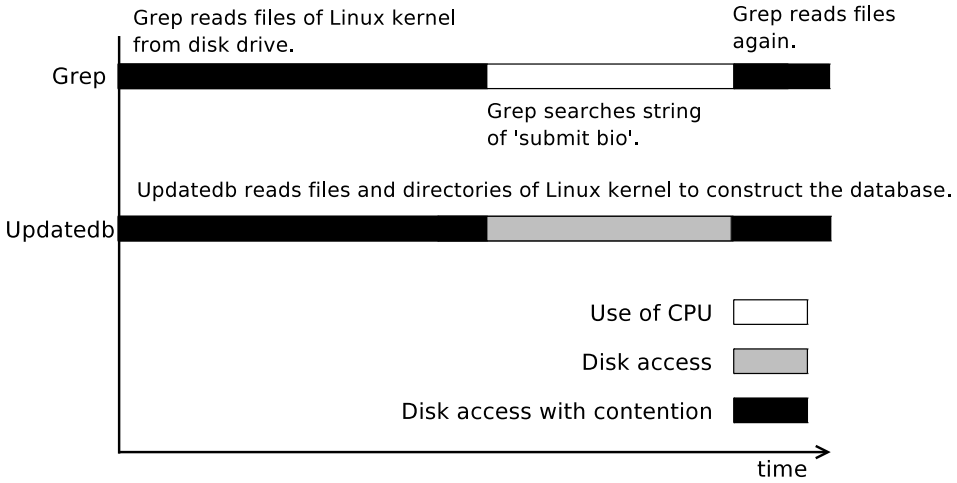
本章では，仮想化環境において，資源管理ポリシーを擬似的に導入する手法 FoxyTechnique を提案した．仮想化環境では，OS は VMM によって提供される仮想デバイス上で稼働する．従来，VMM は物理デバイスのエミュレートを行う

表 5.3: **Comparison of execution times of `Grep` and `Updatedb`.** The table reports execution times of `Grep` (foreground) and `Updatedb` (background) with FoxyIdle. Both jobs access totally about 260 MB of files and directories.

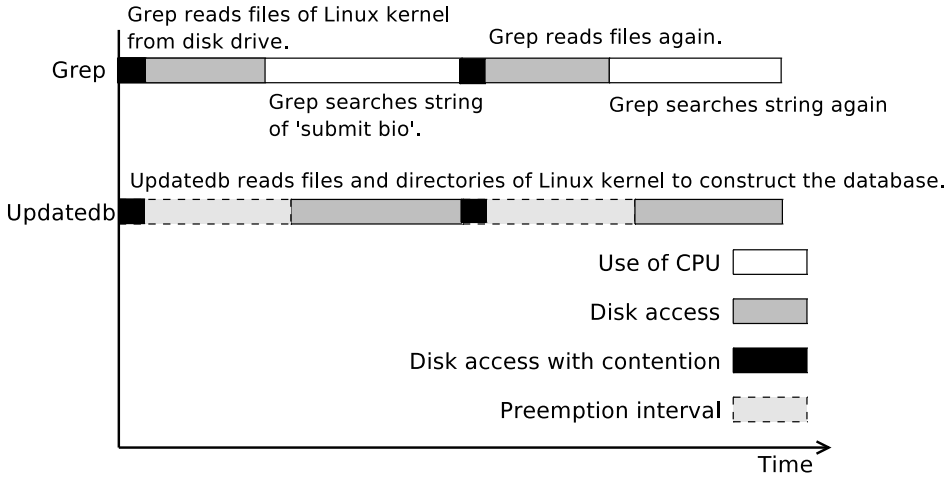
	Native Idle		Guest Idle		FoxyIdle	
Preemption intervals	`Grep` (sec)	`Updatedb` (sec)	`Grep` (sec)	`Updatedb` (sec)	`Grep` (sec)	`Updatedb` (sec)
5 ms	9.72	101.13	24.34	113.44	24.92	113.74
10 ms	9.86	101.91	17.50	109.07	16.40	107.94
15 ms	9.25	101.17	15.40	107.59	15.50	107.60
20 ms	8.72	103.46	14.63	107.30	15.41	107.46
Concurrent execution	10.14	102.02	82.94	123.31	82.94	123.31
Standalone	8.67	92.04	14.34	87.62	14.34	87.62

ことによって OS に仮想デバイスを提供していた．FoxyTechnique では，デバイスの動作に対する OS の振る舞いを推測する．OS の動作を推測しながら，仮想デバイスの動作を意図的に変更することで，OS に所望の動作をさせる．これを利用して，カーネルを修正して資源管理ポリシーを導入したときに近い動作を作り出す．

FoxyTechnique は次の特徴をもつ．まず，OS カーネルを修正することなく資源管理ポリシーを擬似的に導入可能である．Foxy モジュールを VMM 内に実装しているので，VMM 上で稼働する OS に対して特別な修正を加えることなく，Foxy モジュールを適用することができる．次に，FoxyTechnique を用いて実現したポリシーは様々な OS に適用可能である．Foxy モジュールを VMM 内に 1 回実装することで，VMM 上で稼働するすべての OS に対して擬似的に資源管理ポリシー



(a) Concurrent Execution



(b) Execution with FoxyIdle

図 5.13: **Difference between the concurrent execution and the FoxyIdle execution.** The figure shows `Grep` and `Updatedb` behavior with and without FoxyIdle. In concurrent execution, both jobs make progress with disk contention. In contrast, FoxyIdle allows both benchmarks to run with less disk contention.

を導入することができる．3 つめの特徴は，FoxyTechnique を用いて実現したポリシーはゲスト OS で稼働するポリシーよりも正確に動作することがある点である．VMM 内で Foxy モジュールが動作するため，仮想化によるオーバーヘッドを被ることなくパフォーマンス測定が可能である．そのため，RTT やディスクシーク時間を利用する資源管理ポリシーは，FoxyTechnique を用いて導入した方がよい場合

がある．

実際に FoxyTechnique を用いて，2 つの Foxy モジュールを Xen 3.0.2-2 上に実装した．1 つめは FoxyVegas である．FoxyVegas は TCP Vegas [44] を作り出す Foxy モジュールである．TCP Vegas は輻輳ウィンドウを決める TCP 輻輳制御アルゴリズムの一種である．FoxyVegas では，TCP はフロー制御と輻輳制御に基づいてパケットのウィンドウサイズを決めているという知識を用いる．FoxyVegas は，データの受信側が高負荷であると偽った状況をゲスト OS に見せることで，ゲスト OS のウィンドウサイズを調整し，TCP Vegas の振る舞いを作り出している．また，FoxyVegas は RTT をゲスト OS 上の TCP Vegas より正確に計測できるので，ゲスト OS 上の TCP Vegas よりもより正しく振る舞うことができた．

2 つめは FoxyIdle である．FoxyIdle はアイドル時間 (idletime) スケジューリング [9] を作り出す Foxy モジュールである．FoxyIdle はディスク読み込み要求に対してアイドル時間スケジューリングを行う．アイドル時間スケジューリングでは，バックグラウンドプロセスによるフォアグラウンドプロセスのパフォーマンス劣化を防ぐために，バックグラウンドプロセスのディスク読み込み要求がフォアグラウンドプロセスのそれと競合した場合，バックグラウンドプロセスのリクエストを止める．FoxyIdle では，ディスクがいつまでもデータをシークしていると偽った状況をゲスト OS に提供する．FoxyIdle はこのシーク時間を調整することで，アイドル時間スケジューリングの動作を作り出している．実際，FoxyIdle は，4 つのディスクスケジューリングポリシーをアイドル時間スケジューリングに近い動作にすることができた．

第6章　おわりに

6.1　まとめ

アプリケーションや実行環境の多様化に伴って，新たな資源管理ポリシーの提案が多数なされているが，これらの資源管理ポリシーを実用に供することは容易ではない．OS における資源管理ポリシーの選択はアプリケーションの種類やその実行環境によって適切に行うべきであるが，資源管理ポリシーを導入する際には OS カーネルの修正を伴うため，多大な労力を必要とする．そのため，所望の資源管理ポリシーの恩恵を享受するまでには長い時間を要してしまう．

そこで，新たな資源管理ポリシーを擬似的に導入する手法として，ユーザレベルから資源管理ポリシーを変更する手法がある．この手法は，カーネルを修正することなく，ユーザレベルでプロセスを管理することで，資源管理ポリシーを導入したときに近い効果を得ることができる．そのため，OS ベンダによって所望の資源管理ポリシーがサポートされるまでの間の代替手法として期待できる．ユーザレベルで資源管理ポリシーを導入するときには，OS の内部挙動の把握が必要不可欠であり，内部状態を知った上でプロセスを制御する必要がある．そのため，変更するために十分な内部情報を取得できない資源管理ポリシーに関しては導入が困難である．

本研究では，カーネルを修正することなく，カーネル外から変更できる資源管理ポリシーの適用範囲を拡大する手法を提案した．提案手法では，OS が明示的に提供する内部状態や既存の GrayBox 手法で取得できる内部状態だけでなく，ある操作に対する OS の動作をあらかじめ推測することでより多くの OS の挙動を把握する．本論文では，ユーザレベルでディスク帯域の割り当てポリシーを変更する機構，DiscNice を提案した．DiscNice ではディスクアクセスに関する OS の動作を推測し，プロセスが発行するファイル I/O に伴うディスク I/O サイズを推測する．推測したディスク I/O サイズを元に，ファイル I/O を制限することでディスク帯域の使用量を調整する．プロトタイプの実装を行い，設定したポリシーに従ってディ

スク帯域を制限できることを確認した．さらに，本研究では，仮想化技術を利用して資源管理ポリシーを変更する手法，FoxyTechnique を提案した．FoxyTechnique は仮想デバイスに対する OS の動作を推測し，仮想デバイスの振る舞いを変更することで，OS の動作を制御する．この手法を用いることで，TCP 輻輳制御ポリシーの 1 つである TCP Vegas を，ディスクスケジューリングポリシーの 1 つである アイドル時間スケジューリングを擬似的に導入する機構 FoxyVegas，FoxyIdle を実装し，それぞれのポリシーの恩恵を享受できることを確認した．

6.2 今後の展望

新たな資源管理ポリシーの提案は今もなお積極的に行われている．そのため，擬似的な資源管理ポリシーの導入手法は，今後も盛んに研究が行われるトピックの 1 つであると考えられる．本節では本研究の今後の展望について述べる．

6.2.1 他の資源管理ポリシーの擬似的な導入

今後の課題の 1 つとして，カーネル外から導入できる資源管理ポリシーの範囲を明確にする必要があると考えられる．本研究では，ユーザレベルから 1 つの資源管理ポリシー，VMM レベルから 2 つの資源管理ポリシーの擬似的な導入を具体的に示した．導入可能なポリシーを範囲を明示するためには，提案手法を用いて様々な資源管理ポリシーの導入を行う必要がある．実際に，資源管理ポリシーの導入を引き続き行うことで，導入可能，不可能なポリシーを規則性を発見できる可能性がある．

6.2.2 他の OS への適用

提案手法の有用性を検証するために，Windows や Mac OS X といった様々な OS を用いて実験をする必要があると考えられる．本研究で提案した機構の有用性を確かめる実験は，いずれも Linux を用いている．そのため，他の OS へ提案手法を適用すると不具合が生じる可能性がある．これらを検証することで，提案手法の有用性をより明確にすることができる．

6.2.3 OS プロファイリングツールの利用

発達した OS のプロファイリングツールを用いることで，OS の詳細な内部状態をユーザレベル上で取得できる．これらのプロファイリングツールは本来 OS のデバッグやボトルネック検出のために利用されるツールである．しかし，近年のツールはオーバーヘッドも少なく，動的にプロファイルコードを差し込むことができる．こうしたプロファイリングツールが現在実用化に至っている [50, 49].

これらのプロファイリングツールを用いることで，ユーザレベルから導入できる資源管理ポリシーがさらに増えることが期待される．プロファイリングツールを利用することで，これまでユーザレベルで得られなかった詳細な内部情報を取得できる．これにより OS の挙動を細部まで把握することができる．ユーザレベルで取得できる内部情報が増加するため，これまで導入が困難であったポリシーの導入が期待できる．

謝辞

本論文をまとめるまでに，多くの方々からご協力とご指導を賜りました．お世話になったすべての方々に心より感謝いたします．

まず，指導教員をしていただいた河野健二准教授に心から深く感謝いたします．河野健二准教授には，筆者が電気通信大学学部 4 年時から本論文執筆に至るまでの 6 年間，あらゆる面で暖かで親身なご指導をいただきました．日頃からのご指導だけでなく，様々な研究者との出会いの場を提供していただきました．なにより，システムソフトウェアの研究の醍醐味を教えていただき，深く感謝いたします．

筆者が電気通信大学に所属していた学部 4 年生から博士前期課程修了までの 3 年間，ご指導いただいた岩崎英哉教授には，特に論文の執筆やプレゼンテーションについての細やかなご指導をいただきました．また，プログラミングの精緻さ，奥深さについて教えていただきました。深く感謝いたします．

筆者が電気通信大学大学院博士前期課程所属時に指導教員をしていただいた岩田茂樹教授には，日々の相談に乗っていただいたり，研究活動をあたたかく見守っていただきました．深く感謝いたします．

筆者が電気通信大学学部 4 年時に指導教員をしていただいた益田隆司教授には，研究を進めていく上で不自由のない研究環境を提供していただきました。深く感謝いたします．

また，寺岡文男教授，山崎信行准教授，矢向高弘准教授には，貴重な時間を割いて本論文を査読していただき，ご指導とご助言をいただきました．深く感謝いたします．

国際会議や研究会，プロジェクトなど大学外で出会った方々からは，研究意欲をより一層高める刺激をいただきました．また多くのご助言をいただきました．ここに感謝いたします．

河野研究室や電気通信大学情報工学科の益田研究室，岩崎研究室の方々のおかげで，刺激的な研究生活を送ることができました．特に，杉木章義氏や浅原理人氏，阿部芳久氏，吉田哲也氏，小林卓嗣氏とは研究について活発な議論を行うこ

とができました．かけがえのないメンバーと出会えたことを幸運に思い，心から感謝の意を表します．

最後に，経済的支援を惜しまず暖かく見守ってくださった両親に心から深く感謝いたします．

論文目録

定期刊行誌掲載論文

- Hiroshi Yamada and Kenji Kono, “DiscNice: User-level Regulation of Disk Bandwidth”, 情報処理学会論文誌コンピュータシステム, Vol.48, No.SIG 18 (ACS 20), pp.83–98, December 2007. (IPSJ Digital Courier 船井若手奨励賞)
- Hiroshi Yamada and Kenji Kono, “Introducing New Resource Management Policies using a Virtual Machine Monitor”, 情報処理学会論文誌コンピュータシステム, Vol.1, No.1, pp.144–159, June 2008.

国際会議論文

- Hiroshi Yamada and Kenji Kono, “User-level disk-bandwidth control for resource-borrowing network applications”, In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS ’06) (CD-ROM)*, April 2006.
- Hiroshi Yamada and Kenji Kono, “FoxyTechnique: Tricking Operating System Policies with a Virtual Machine Monitor”, In *Proceedings of the 3rd ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE ’07)*, pp.55–64, June 2007.

研究会報告

- 山田 浩史, 河野 健二, “ユーザレベルでのディスク帯域制御機構”, 第 8 回プログラミングおよび応用のシステムに関するワークショップ (SPA 2005), pp.30–36, March 2005.
- 山田 浩史, 河野 健二, “FoxyTechnique: 仮想機械モニタによる OS の資源管理ポリシーの変更”, 情報処理学会システムソフトウェアとオペレーティングシステム研究報告, Vol.2007, No.36, pp.79–86, April 2007. (IPSJ 山下記念研究賞, IPSJ SIGOS 最優秀学生発表賞)

参考文献

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Oct. 2003.

[2] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@HOME-Massively Distributed Computing for SETI. *IEEE Computing in Science and Engineering*, 3(1):78–83, 2001.

[3] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.

[4] Kyung D. Ryu, Jeffery K. Hollingsworth, and Peter J. Keleher. Exploiting Fine Grained Idle Periods in Networks of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 11:683–698, 2000.

[5] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, and David F. Nagle. Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI '00)*, pages 87–102, Oct. 2000.

[6] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 10–22, Jan. 2002.

[7] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation (OSDI '02)*, pages 329–344, Dec. 2002.

[8] Ashish Gupta, Bin Lin, and Peter A. Dinda. Measuring and Understanding User Comfort With Resource Borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC '04)*, pages 214–224, 2004.

[9] Lars Eggert and Joseph D. Touch. Idletime Scheduling with Preemption Intervals. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 249–262, Oct. 2005.

[10] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI '94)*, pages 1–12, Nov. 1994.

[11] David Petrou, John W. Milford, and Garth A. Gibson. Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, pages 1–14, Jun. 1999.

[12] Cart A. Waldpurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical report, MIT/LCS/TM-528, Massachusetts Institute of Technology, 1995.

[13] David G. Sullivan and Margo I. Saltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 337–350, Jun. 2000.

[14] Sun Microsystems. *Consolidating Applications with Solaris Containers (White Paper)*. http://www.sun.com/datacenter/consolidation/solaris10_whitepaper.pdf, 2004.

[15] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 73–86, Dec. 2008.

[16] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina Popovici, Aditya Akella, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Suman Banerjee. Avoiding File System Micromanagement with Range Writes. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 161–176, Dec. 2008.

[17] Andreas Merkel and Frank Bellosa. Task Activity Vectors: A New Metric for Temperature-Aware Scheduling. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys '08)*, pages 1–12, Apr. 2008.

[18] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient Guaranteed Disk Request Scheduling with Fahrrad. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys '08)*, pages 13–25, Apr. 2008.

[19] Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. Competitive Prefetching for Concurrent Sequential I/O. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys '07)*, pages 189–202, Mar. 2007.

[20] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys '07)*, pages 47–58, Mar. 2007.

[21] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the USENIX Annual Technical Conference (USENIX '07)*, pages 261–274, Jun. 2007.

[22] Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 78. IEEE Computer Society, May 1995.

[23] G. Denys, F. Piessens, and F. Matthijs. A Survey of Customizability in Operating Systems Research. *ACM Computer Survey*, 34(4):450–468, Dec. 2002.

[24] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems (ICDCS '92)*, pages 512–520, Jun. 1992.

[25] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX Development. In *Proceedings of Summer USENIX Conference*, pages 93–112, Jun. 1986.

[26] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 64–75, Dec. 1981.

[27] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 237–250, Dec. 1995.

[28] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys '06)*, pages 133–145, Apr. 2006.

[29] Braian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–283, Dec. 1995.

[30] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–227, Oct. 1996.

[31] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management.

In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 251–266, Dec. 1995.

[32] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Oct. 1997.

[33] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Oct. 2003.

[34] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deploying Safe User-Level Network Services with icTCP. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, Dec. 2004.

[35] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Oct. 2001.

[36] Travis Newhouse and Joseph Pasquale. ALPS: An Application-Level Proportional-Share Scheduler. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC '06)*, pages 279–290, Jun. 2006.

[37] Travis Newhouse and Joseph Pasquale. A User-Level Framework for Scheduling within Service Execution Environments. In *Proceedings of IEEE International Conference on Service Computing (SCC '04)*, pages 311–318, Sep. 2004.

[38] John R. Douceur and William J. Bolosky. Progress-based regulation of Low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 247–260, Dec. 1999.

[39] Yoshihisa Abe, Hiroshi Yamada, and Kenji Kono. Enforcing Appropriate Process Execution for Exploiting Idle Resources from Outside Operating Systems. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys '08)*, pages 55–64, Apr. 2008.

[40] Fangzhe Chang, Ayal Itzkovitz, and Vijay Karamcheti. User-level Resource-constrained Sandboxing. In *Proceedings of the 4th USENIX Windows System Symposium (WSS 2000)*, pages 25–36, Aug. 2000.

[41] S. Ghias and S. Zeadally. Design and Implementaion of a User-level Prioritization Service. In *Proceedings of the 8th IEEE Symposium on Computers and Communications*, pages 30–35, Jun. 2003.

[42] James Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 311–324, Jun. 2003.

[43] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[44] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the ACM Conference on Communications architechtures, protocols and applications (SIGCOMM '94)*, pages 24–35, Aug. 1994.

[45] Yaoping Ruan and Vivek Pai. Making the "Box" Transparent: System Call Performance as a First-class Result. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 1–14, Jun. 2004.

[46] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual Memory Support for Garbage-Collected Applications. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 103–116, Dec. 2006.

[47] Microsoft. *System Information Functions (Windows)*, 2008. `http://msdn.microsoft.com/en-us/library/ms724953(VS.85).aspx`.

[48] Linux Technology Center : Welcome. Kprobe. http://sourceware.org/systemtap/kprobes/.

[49] Apple - Mac OS X Leopard - Developer Tools - Instruments. Apple - Mac OS X Leopard - Developer Tools - Instruments. http://www.apple.com/macosx/developertools/instruments.html.

[50] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 15–28, Jun. 2004.

[51] Richard J. Moore. A Universal Dynamic Trace for Linux and other Operating Systems. In *Proceedings of the FREENIX Track: the USENIX Annual Technical Conference (FREENIX '01)*, pages 297–308, Jun. 2001.

[52] Kyung D. Ryu, Jeffery K. Hollingsworth, and Peter J. Keleher. Efficient Network and I/O Throttling for Fine-Grain Cycle Stealing. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '01) (CDROM)*, pages 3–3, Nov. 2001.

[53] Andrew Chien, Brad Calder Stephen Elbert, and Karan Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63:597–610, 2003.

[54] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, and Erich Nahum. Daytona: A User-Level TCP Stack, 2008. http://nms.lcs.mit.edu/~kandula/data/daytona.pdf.

[55] Aled Edwards and Steve Muir. Experiences implementing a high performance TCP in user-space. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '95)*, pages 196–205, Aug. 1995.

[56] David Ely, Stefan Savage, and David Wetherall. Apline: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and System (USITS '01)*, pages 171–184, Mar. 2001.

[57] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Jun. 2002.

[58] AMD. *AMD64 Programmer's Manual Volume2: System Programming*, Sep. 2007. `http://www.amd.com/us-en/assets/content_type/white_papers_andtech_docs/24593.pdf`.

[59] W. J. Armstrong, R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, and R. C. Swanberg. Advanced Virtualization Capabilities of POWER5 Systems. *IBM Journal of Research and Development*, 49(4/5):523–532, Jul. 2005.

[60] Intel Corporation. *Intel64 and IA-32 Architechtures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, Nov. 2008. `http://download.intel.com/design/processor/manuals/253669.pdf`.

[61] Sun Microsystems. Sun Consolidation and Virtualization, 2007. `http://www.sun.com/virtualization`.

[62] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Oct. 2003.

[63] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 1–14, Jun. 2001.

[64] Sun Microsystems. Virtualbox, 2008. `http://www.virualbox.org`.

[65] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual

Machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 273–286, May 2005.

[66] Timothy Wood, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 229–242, Apr. 2007.

[67] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and implementation (NSDI '08)*, pages 161–174, Apr. 2008.

[68] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex Snoeren, Geoffrey Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 148–162, Dec. 2005.

[69] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys'07)*, pages 289–302, Mar. 2007.

[70] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual-Machine Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 193–206, Oct. 2003.

[71] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 223–236, Oct. 2003.

[72] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Over-

shadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Mar. 2008.

[73] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection Through VMM-based "Out-of-the-Box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 128–138, Oct. 2007.

[74] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 335–350, Oct. 2007.

[75] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 1–14, Jun. 2008.

[76] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-Based Fault Tolerance. *ACM Transaction Computer System*, 14(1):80–107, Feb. 1996.

[77] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys '07)*, pages 275–287, Mar. 2007.

[78] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2nd ACM International Conference on Virtual Execution Environments (VEE '06)*, pages 35–44, Jun. 2006.

[79] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '07)*, pages 2–13, Mar. 2006.

[80] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, Jun. 2006.

[81] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, Oct. 2006.

[82] Peter M. Chen and Brian D. Noble. When Virtual is Better than Real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 133–138, Jun. 2001.

[83] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. In *Proceedings of the 4th ACM International Conference on Virtual Execution Environments (VEE '08)*, pages 91–100, Mar. 2008.

[84] Pin Lu and Kai Shen. Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache. In *Proceedings of the USENIX Annual Technical Conference (USENIX '07)*, pages 29–43, Jun. 2007.

[85] Zhifeng Chen, Yuanyuan Zho, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 269–282, Jun. 2003.

[86] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation (OSDI '02)*, pages 181–194, Dec. 2002.

[87] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium (VM '04)*, pages 43–56, May 2004.

[88] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Pro-*

ceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05), pages 91–104, Oct. 2005.

[89] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Dec. 1997.

[90] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 309–322, Dec. 2008.

[91] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, pages 87–100, May 2006.

[92] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI '08)*, pages 407–421, Apr. 2008.

[93] Jedidiah R. Crandall, Gary Wassermann, Daniela A. S. de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. In *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 25–36, Oct. 2006.

[94] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 117–130, Feb. 1999.

[95] The Apache Software Foundation. Apache HTTP server, 1995. `http://www.apache.org/`.

[96] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation (OSDI '02)*, pages 195–209, Dec. 2002.

[97] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Oct. 2004.

[98] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, pages 15–28, Jun. 2006.

[99] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High Performance VMM-Bypass I/O in Virtual Machine. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, pages 29–42, Jun. 2006.

[100] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling I/O in Virtual Machine Monitors. In *Proceedings of the 4th ACM International Conference on Virtual Execution Environments (VEE '08)*, pages 1–10, Mar. 2008.

[101] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. VMM-Independent Graphics Acceleration. In *Proceedings of the 3rd ACM International Conference on Virtual Execution Environments (VEE '07)*, pages 33–43, Jun. 2007.

[102] Jose Renato Santos, Yoshio Turner, G.(John) Janakiraman, and Ian Pratt. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 29–42, Jun. 2008.

[103] M. Allman, V. Paxson, and W. Stevens. RFC2581: TCP Congestion Control, 1999. http://www.rfc-editor.org/rfc/rfc2581.txt.

[104] J. Postel. RFC863: Discard Protocol, 1983. http://www.rfc-editor.org/rfc/rfc863.txt.

[105] S. Shenker and J. Wroclawski. RFC2216: Network Element Service Specification Template, 1997. http://www.rfc-editor.org/rfc/rfc2216.txt.