Red Hat Enterprise Linux 6

パフォーマンスチューニングガイド

Red Hat Enterprise Linux 6 におけるサブシステムのスループットを最適化

エディッション 4.0

Last Updated: 2017-10-12

Red Hat Enterprise Linux 6 パフォーマンスチューニングガイド

Red Hat Enterprise Linux 6 におけるサブシステムのスループットを最適化
エディッション 4.0

Red Hat Subject Matter Experts

編集者

Don Domingo

Laura Bailey

法律上の通知

概要

パフォーマンスチューニングガイドは、Red Hat Enterprise Linux 6 を実行するシステムのパフォーマンス最適化方法を説明するものです。また、Red Hat Enterprise Linux 6 のパフォーマンス関連の更新も紹介しています。 本ガイドの手順は実地で試験・検証がされていますが、Red Hat ではこれらを本番環境に適用する前にテスト環境で予定するすべての設定を正確にテストすることを推奨します。また、データとチューニング前の設定すべてのバックアップも推奨します。

目次

第1章 概要

『パフォーマンスチューニングガイド』 は、Red Hat Enterprise Linux における総合的な設定および最適化のリファレンスです。このリリースには Red Hat Enterprise Linux 5 のパフォーマンス機能に関する情報も含まれていますが、本書内の指示はすべて Red Hat Enterprise Linux 6 に固有のものです。

1.1. 本書を読むにあたって

本書は、Red Hat Enterprise Linux の特定のサブシステムについて説明する章に分かれています。『パフォーマンスチューニングガイド』 は、サブシステムごとに以下の 3 つの主要テーマについてフォーカスしています。

機能

各サブシステムの章では、Red Hat Enterprise Linux 6 特有の (または別の方法で実装されている) パフォーマンス機能を説明します。また、Red Hat Enterprise Linux 5 と比べて特定のサブシステムのパフォーマンスを大幅に改善した Red Hat Enterprise Linux 6 の更新も説明します。

分析

本書では、各サブシステム特有のパフォーマンス指標も紹介します。これら指標の通常の値は特定サービスのコンテキスト内で説明されるので、実際の本番システムでの重要度の理解が容易になります。

また『パフォーマンスチューニングガイド』 は、サブシステムのパフォーマンスデータ (プロファイリング) の別の取得方法も紹介します。ここで示されているプロファイリングツールには、本書以外でより詳しく説明されているものもあります。

設定

本書で最も重要な情報は、おそらく Red Hat Enterprise Linux 6 の固有のサブシステムのパフォーマンスを調整する方法に関する指示になります。『パフォーマンスチューニングガイド』 では、特定のサービスについて Red Hat Enterprise Linux 6 のサブシステムを微調整する方法を説明しています。

特定のサブシステムのパフォーマンスを微調整すると、時には別のサブシステムのパフォーマンスに悪影響を与える場合もあることに留意してください。Red Hat Enterprise Linux 6 のデフォルト設定は、**ほとんどの** サービスが **適度な** 負荷の下で実行する際に最適となっています。

『パフォーマンスチューニングガイド』 で示されている手順は、ラボと実地の両方で Red Hat エンジニアが徹底的にテストしていますが、Red Hat ではこれらを実稼働サーバーに適用する前に安全なテスト環境で予定するすべての設定を正確にテストすることを推奨しています。また、システムのチューニング前にデータと設定情報すべてをバックアップすることも推奨しています。

1.1.1. 対象読者

本書は以下の 2 つのタイプの読者を対象としています。

システム / ビジネスアナリスト

本書では、Red Hat Enterprise Linux 6 の高いレベルのパフォーマンス機能を説明しており、(デフォルトと最適化された時の両方での) 特定の作業負荷においてサブシステムがどのように作動するかについての十分な情報を提供しています。Red Hat Enterprise Linux 6 のパフォーマンス機能は詳細なレベルで説明されているので、受け入れ可能なレベルでリソース集約型のサービスを提供する際のこのプラットフォームの適合性を潜在的な顧客やセールスエンジニアが理解できるようになっています。

『パフォーマンスチューニングガイド』 は、可能な限り各機能のより詳細なドキュメンテーションへのリンクも提供しています。この詳細なレベルでは、ユーザーはパフォーマンス機能を十分に理解して Red Hat Enterprise Linux 6 の導入および最適化における高レベルな戦略を形成することができます。これによりユーザーは、インフラストラクチャー提案を開発し**かつ** 評価することができます。

本書は機能にフォーカスしたドキュメンテーションなので、Linux サブシステムおよび企業レベルのネットワークを高度に理解できるユーザーが対象となります。

システム管理者

本書の手順は、RHCE [1] スキルレベル (もしくはそれと同等。つまり、Linux の導入および管理の経験が 3-5 年) のシステム管理者向けとなっています。『パフォーマンスチューニングガイド』 では、各設定の影響に関してできるだけの詳細を提供しており、パフォーマンスの代償の可能性もすべて説明されています。

パフォーマンスチューニングで基礎となるスキルは、サブシステムの分析やチューニング方法の知識ではありません。むしろ、パフォーマンスチューニングに精通しているシステム管理者は、**特定の目的で** Red Hat Enterprise Linux 6 システムのバランスを取り最適化する方法を理解しています。これは、特定のサブシステムのパフォーマンスを改善するための設定を実装する際に、代償やパフォーマンス低下**についても**理解していることを意味します。

1.2. リリースの概要

1.2.1. Red Hat Enterprise Linux 6 における新機能

本セクションでは、Red Hat Enterprise Linux 6 に含まれるパフォーマンス関連の変更についての簡単な概要を説明しています。

1.2.1.1. 6.6 リリースでの新機能

- **perf** がバージョン 3.12 に更新され、以下の新機能が含まれています。
  - **`perf record`** の **`-j`** および **`-b`** の各オプションで、連続するブランチの統計サンプルを行います。詳細は、**`man perf-record`** を参照してください。
  - ***`--group`*** および ***`--percent-limit`*** を含む新たな **`perf report`** パラメーターと、**`perf record -j`** および **`-b`** のオプションを有効にして収集されたデータを分類する追加オプション。詳細は、**`man perf-report`** を参照してください。
  - ロードおよび保存メモリーアクセスをプロファイリングする新たな **`perf mem`** コマンド。
  - ***`--percent-limit`*** および ***`--obj-dump`*** を含む **`perf top`** の新オプション。
  - ***`--force`*** および ***`--append`*** のオプションが **`perf record`** から削除されました。
  - **`perf stat`** の新オプション ***`--initial-delay`***。
  - **`perf trace`** の新オプション ***`--output-filename`***。
  - **`perf evlist`** の新オプション ***`--group`***。
  - **`perf top -G`** および **`perf record -g`** の各オプションが変更されました。これらはもう、**`--call-graph`** オプションの代わりにはなりません。今後の Red Hat

Enterprise Linux バージョンに **libunwind** サポートが追加されると、これらのオプションは設定済みアンワインドメソッドを有効にします。

1.2.1.2. 6.5 リリースでの新機能

- カーネルへの更新で、IRQ テーブルのサイズが 1 GB 以上の場合、I/O ロードが複数のメモリープールに拡散することが可能になることで、メモリー管理のボトルネックが取り除かれ、パフォーマンスが改善します。
- cpupowerutils パッケージが更新され、**turbostat** および **x86_energy_perf_policy** ツールが含まれるようになりました。これらのツールは、「**turbostat**」 および 「ハードウェアパフォーマンスポリシー (**x86_energy_perf_policy**)」 で説明されています。
- CIFS がより大型の rsize オプションと非同期の readpage をサポートすることで、スループットの大幅な増加が可能になりました。
- GFS2 が Orlov ブロックアロケーターを提供するようになり、ブロック割り当てが速く行われるようになりました。
- **tuned** の ***virtual-hosts*** プロファイルが調整されました。***kernel.sched_migration_cost*** の値は、カーネルのデフォルトの **500000** ナノ秒 (0.5 ミリ秒) ではなく、**5000000** ナノ秒 (5 ミリ秒) となっています。これにより、大規模な仮想化ホスト用の実行キューロックにおける競合が減ることになります。
- **tuned** の ***latency-performance*** プロファイルが調整されました。サービスの電力管理の質である **cpu_dma_latency** 要件の値が、**0** ではなく **1** になっています。
- カーネル **copy_from_user()** および **copy_to_user()** の関数にいくつかの最適化が含まれ、この両方のパフォーマンスを改善しています。
- perf ツールに、以下のものを含む多くの更新および機能強化がなされています。
  - Perf が、System z CPU 測定カウンター機能が提供するハードウェアカウンターを使用できるようになりました。利用可能なハードウェアカウンターの 4 つのセットは、基本カウンターセット、問題状態カウンターセット、暗号化アクティビティーカウンターセット、および拡張カウンターセットです。
  - 新たに **perf trace** コマンドが利用可能になりました。これにより、**perf** インフラストラクチャーを使用した **strace** のような動作で追加ターゲットの追跡が可能になります。詳細は、「Perf」 を参照してください。
  - スクリプトブラウザーが追加され、ユーザーは現行の perf データファイルに利用可能なすべてのスクリプトを閲覧できるようになりました。
  - いくつかの追加のサンプルスクリプトが利用可能になっています。
- Ext3 が更新され、ロックの競合を減らしています。これにより、マルチスレッド書き込み操作のパフォーマンスが改善しています。
- KSM が、NUMA トポロジーを認識するように更新されました。これにより、ページ結合の際に NUMA の場所を考慮することが可能になっています。これで、ページがリモートのノードに移動することに関するパフォーマンス低下が避けられます。Red Hat では、KSM の使用時にクロスノードのメモリーマージを避けることを推奨しています。

この更新では、新たに調整可能な **/sys/kernel/mm/ksm/merge_nodes** を導入してこの動作を制御しています。デフォルト値 **(1)** は、異なる NUMA ノードにまたがるページをマージします。***merge_nodes*** を **0** に設定すると、同一ノード上のページのみがマージされます。

- **hdparm** が **--fallocate**、**--offset**、および **-R** (読み-書き-検証 の有効化) を含む新たなフラグで更新されました。また、**--trim-sectors** オプションが **--trim-sector-ranges** および **--trim-sector-ranges-stdin** のオプションに置換されています。これで、複数セクターの範囲が指定できます。これらのオプションの詳細情報は、man ページを参照してください。

1.2.2. 水平方向のスケーラビリティ

Red Hat Enterprise Linux 6 のパフォーマンス改善は、**スケーラビリティ**にフォーカスが置かれています。パフォーマンスを高める機能は主に、ワークロードの範囲において異なるエリアでプラットホームのパフォーマンスに与える影響を基に評価されています。つまり、孤立した Web サーバーからサーバーファームのメインフレームにいたるまでです。

スケーラビリティにフォーカスすることで、Red Hat Enterprise Linux は異なるタイプのワークロードや目的に適した多様性を維持できます。同時に、ビジネスが拡大してワークロードが増えるのに合わせて、サーバー環境の再構築が (費用と工数の面で) 法外ではなく、より直感的なものになることを意味します。

Red Hat は、**水平的スケーラビリティ** と **垂直的スケーラビリティ** の両方で Red Hat Enterprise Linux を改善してきました。しかし、水平的スケーラビリティのユースケースの方がより広く適用可能です。水平的スケーラビリティの考えの基となっているのは、複数の**標準的コンピューター** を使って高いワークロードを分散し、パフォーマンスと信頼性を高めるというものです。

通常のサーバーファームでは、これらの標準的コンピューターは、1U ラックマウント型サーバーおよびブレードサーバーになります。各標準的コンピューターはシンプルな 2 ソケットシステムの小型のものです。ただし、サーバーファームのなかには、ソケット数の多い大型システムを使用するものもあります。エンタープライズグレードのネットワークには大型と小型のシステムを組み合わせるものもあります。その場合は、大型システムは高パフォーマンスサーバー (例えばデータベースサーバー) で、小型システムは専用のアプリケーションサーバー (例えば Web もしくはメールサーバー) になります。

このタイプのスケーラビリティは、IT インフラストラクチャーの拡大をシンプルなものにします。適当なロードの中規模ビジネスでは、ピザボックスサーバー 2 台ですべてのニーズを満たすことも可能です。従業員数が増え、オペレーションを拡大し、売上高が増えるなどすると、IT 要件はボリュームも複雑性も増すことになります。水平的スケーラビリティを用いれば、既存マシンと (ほぼ) 同一設定のマシンを新たに導入するというシンプルなことが可能になります。

要約すると、水平的スケーラビリティはシステムハードウェア管理を簡素化する抽象化レイヤーを追加します。Red Hat Enterprise Linux プラットホームを水平方向に拡張するように開発することで、IT サービスの機能およびパフォーマンス向上は新しくかつ設定が容易なマシンを追加するという簡単なものになります。

1.2.2.1. パラレルコンピューティング

ユーザーが Red Hat Enterprise Linux の水平的スケーラビリティの利益が受けられるのは、システムハードウェア管理が簡素化されるという理由だけではありません。現行ハードウェアの発達トレンドに、水平的スケーラビリティの開発指針が適していることもその理由です。

以下の点を考えてみましょう。複雑なエンタープライズアプリケーションのほとんどでは同時実行の必要がある数千ものタスクがあり、これらタスク間の連携方法は異なります。以前のコンピューターではこれらのタスクの処理にシングルコアのプロセッサーを使用していましたが、現在は利用可能なプロ

セッサーのほとんどがマルチコアです。実質的には現代のコンピューターは、マルチコアを 1 つのソケットに置いて、シングルソケットのデスクトップやラップトップをマルチプロセッサーのシステムにしています。

2010 年には、標準的な Intel および AMD のプロセッサーは 2-16 コアでした。これらのプロセッサーはピザボックスおよびブレードサーバーでは一般的になっており、今では最大 40 コアも格納可能です。これら低コストで高パフォーマンスのシステムは、大型システムの機能と特徴をメインストリームにもたらしています。

システムのパフォーマンスと使用率を最高のものにするには、各コアが常に稼働している必要があります。つまり、32 コアのブレードサーバーをフル活用するには、32 の個別のタスクを実行する必要があります。1 つのブレードシャーシにこのような 32 コアのブレードが 10 台収まると、全体の構成では最低 320 のタスクが同時処理できることになります。これらのタスクが 1 つのジョブの一部であれば、これらは連携する必要があります。

Red Hat Enterprise Linux は、ハードウェアの開発トレンドにうまく適応し、そのようなトレンドからビジネスが恩恵を十分に得られるように開発されました。「分散システム」では、 Red Hat Enterprise Linux の水平的スケーラビリティを可能にしている技術を詳細にわたって説明しています。

1.2.3. 分散システム

水平的スケーラビリティを完全に実現するために、Red Hat Enterprise Linux では多くの *分散コンピューティング* のコンポーネントを用いています。分散コンピューティングを形成する技術は、以下の 3 つの層に分けられます。

通信

水平的スケーラビリティでは、多くのタスクが (パラレルで) 同時実行される必要があります。このため、これらのタスクは、**プロセス間通信** で作業を調整する必要があります。さらに、水平的スケーラビリティのプラットホームは、複数システムにわたってタスク共有が可能である必要があります。

ストレージ

ローカルディスクによるストレージは、水平的スケーラビリティの要件に対応するには不十分です。なんらかの分散もしくは共有ストレージが必要です。単一ストレージボリュームの容量が新たなストレージハードウェアを追加することで、シームレスに拡大できるような抽象化レイヤーを備えたものが必要になります。

管理

分散コンピューティングにおける最も重要な任務は、**管理** 層になります。この管理層は、すべてのソフトウェアおよびハードウェアのコンポーネントを連携させ、通信とストレージ、共有リソースの使用を効率的に管理するものです。

以下のセクションでは、各層の技術を詳細に説明します。

1.2.3.1. 通信

通信層はデータ移動を確実にするもので、以下の 2 つで構成されています。

- ハードウェア
- ソフトウェア

複数システムが通信する最も簡潔な (かつ最速の) 方法は、**共有メモリ** です。これには、よくあるメモリの読み取り/書き込み操作が必要になります。共有メモリには、高い帯域幅、短い待ち時間、通常のメモリ読み取り/書き込み操作における低いオーバーヘッドが備わっています。

イーサネット

コンピューター間の最も一般的な通信方法は、イーサネットを使ったものです。現在ではシステム上で *Gigabit イーサネット* (GbE) がデフォルトで提供されており、ほとんどのサーバーには Gigabit イーサネットのポートが 2-4 個あります。GbE はすぐれた帯域幅と待ち時間を提供します。これは、現在使用されているほとんどの分散システムの基礎となっています。システムに高速ネットワークハードウェアがある場合でも、専用の管理インターフェースには GbE を使うのが一般的です。

10GbE

10 Gigabit イーサネット (10GbE) の使用は、ハイエンドおよびミッドレンジサーバーで急速に拡大しています。10GbE は、GbE の 10 倍の帯域幅を提供します。その主な長所の一つは、最新のマルチコアプロセッサーと使用すると、通信とコンピューティングのバランスを回復する点です。シングルコアシステムで GbE を使う場合と 8 コアシステムで 10GbE を使う場合を比較すると違いがよく分かります。この方法で使うと、総合的なシステムパフォーマンスを維持し、通信ボトルネックを回避するという点で 10GbE は特に有用です。

残念ながら 10GbE は高価なものです。10GbE NIC の価格は低下したものの、相互接続 (特に光ファイバー) の価格は高いままで、また 10GbE ネットワークスイッチは非常に高価です。長期的にはこれらの価格は下がると思われますが、10GbE は現在、サーバールームのバックボーンとパフォーマンスクリティカルなアプリケーションで最もよく使われています。

Infiniband

Infiniband は 10GbE よりもさらに高いパフォーマンスを提供します。イーサネットで使用する TCP/IP および UDP ネットワーク接続に加えて、Infiniband は共有メモリ通信もサポートします。これにより Infiniband は、*remote direct memory access* (RDMA) によるシステム間での作業が可能になります。

Infiniband は RDMA を使うことで、TCP/IP のオーバーヘッドやソケット接続なしでデータを直接システム間で移動できます。アプリケーションによっては必須である短い待ち時間が、これで可能になります。

Infiniband は、高い帯域幅や短い待ち時間、低いオーバーヘッドを必要とする *High Performance Technical Computing* (HPTC) で最もよく使われています。多くのスーパーコンピューティングアプリケーションがこの恩恵を受けており、パフォーマンス改善にはより高速のプロセッサーやより多くのメモリではなく、Infiniband に投資するのが最善の方法になっているほどです。

RoCE

RDMA over Converged Ethernet (RoCE) は、Infiniband スタイルの通信 (RDMA を含む) を 10GbE インフラストラクチャー上で実装します。増大している 10GbE 製品の価格改善傾向を考えると、RDMA と RoCE が幅広いシステムやアプリケーションでより多く使われることが見込まれます。

Red Hat は、これら通信方法の Red Hat Enterprise Linux 6 における使用を完全にサポートしています。

1.2.3.2. ストレージ

分散型コンピューティングを使用する環境では、複数インスタンスの共有ストレージが使われます。これは、以下のどちらかを意味します。

- 単一のロケーションで複数のシステムがデータを保存している。
- 複数のストレージアプライアンスで構成されるストレージユニット (例: ボリューム)

最も分かりやすいストレージの例は、システムに搭載されたローカルディスクドライブです。すべてのアプリケーションが1つのホストに格納されている場合や少数のホストに格納されている場合のITオペレーションでは、これが適切です。しかし、インフラストラクチャーが数十や数百のシステムに拡大すると、それらの数のストレージディスクを管理することは困難かつ複雑になります。

分散型ストレージは、ビジネスが拡大するにつれてストレージハードウェアの管理を容易かつ自動化する層を加えます。複数のシステムが少数のストレージインスタンスを使うようにすることで、管理者が管理するデバイス数を減らすことができます。

複数のストレージアプライアンスのストレージ機能を1つのボリュームに統合することで、ユーザーと管理者の双方に利益をもたらします。このタイプの分散型ストレージは、ストレージプールへの抽象化層を提供します。ユーザーは単一ユニットのストレージを参照することになり、管理者はさらにハードウェアを追加することでこれを容易に拡大することができます。分散型ストレージを可能にするテクノロジーのなかには、フェイルオーバーやマルチパスなどの新たな利点をもたらすものもあります。

NFS

Network File System (NFS) は、複数のサーバーやユーザーがTCPもしくはUDP経由でリモートストレージの同一インスタンスをマウントもしくは使用することを可能にします。NFSは一般的に、複数のアプリケーションが共有するデータを保持します。また大量データの一括保存にも便利です。

SAN

ストレージエリアネットワーク(SAN) は、Fibre ChannelもしくはiSCSIプロトコルを使ってストレージへのリモートアクセスを提供します。(Fibre Channelホストバスアダプターやスイッチ、ストレージアレイといった) Fibre Channelインフラストラクチャーは、高パフォーマンスと高い帯域幅、大量ストレージを組み合わせます。SANはプロセッシングからストレージを分離することで、システムデザインの柔軟性を大幅に高めます。

SANのもう一つの大きな利点は、主要なストレージハードウェア管理タスクを実行する管理環境を提供する点です。このタスクには以下のものが含まれます。

- ストレージへのアクセス制御
- 大量データの管理
- システムのプロビジョニング
- データのバックアップと複製
- スナップショットの作成
- システムフェイルオーバーのサポート
- データ整合性の確保
- データの移行

GFS2

Red Hat *Global File System 2* (GFS2) ファイルシステムは、特別機能をいくつか提供します。GFS2の基本機能は単一ファイルシステムの提供で、これには同時読み取り/書き込みアクセスが含まれ、クラスターの複数メンバー間で共有されます。つまり、このクラスターの各メンバーは、GFS2ファイルシステム内の「ディスク上」で全くの同一データを見ることになります。

GFS2では、すべてのシステムが同時に「ディスク」へアクセスできます。データの整合性を維持するために、GFS2は *Distributed Lock Manager* (DLM) を使用します。これは、一度に1つのシステムのみが特定のロケーションへの書き込みを可能にするものです。

GFS2 は特に、ストレージで高いアベイラビリティを必要とするフェイルオーバーアプリケーションに適しています。

GFS2 の詳細情報に関しては、『Global File System 2』 を参照してください。ストレージ全般に関する情報は、『ストレージ管理ガイド』 を参照してください。どちらも http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

1.2.3.3. 集中型ネットワーク

ネットワーク上の通信は通常イーサネットを使って行われ、ストレージトラフィックは専用の Fibre Channel SAN 環境を使います。システム管理に専用ネットワークやシリアルリンクがあるのは一般的で、*ハートビート*[2] がある場合もあります。その結果、単一サーバーが複数ネットワーク上にあることになります。

各サーバーに複数の接続を提供すると費用がかかり、巨大になって管理が複雑化します。このため、すべての接続を 1 つにする方法が求められました。*Fibre Channel over Ethernet* (FCoE) と *インターネット SCSI* (iSCSI) がこのニーズに応えています。

FCoE

FCoE では、標準のファイバーチャンネルコマンドとデータパケットは、単一の *集中型ネットワークアダプター* (CNA) 経由で 10GbE の物理ネットワークを使って移動します。標準 TCP/IP イーサネットトラフィックとファイバーチャンネルストレージ操作も同一リンク経由で移動できます。FCoE は 1 つの物理ネットワークインターフェース (および 1 本のケーブル) を使って複数の論理ネットワーク/ストレージ接続を行います。

FCoE には以下の利点があります。

接続数が減少

FCoE を使うと、サーバーへのネットワーク接続数が半分になります。パフォーマンスやアベイラビリティ目的で複数接続を選ぶことは依然として可能ですが、単一接続でストレージとネットワーク接続の両方が提供されます。これは特にピザボックスサーバーとブレードサーバーの場合、コンポーネントのスペースが限られているため、便利なものになります。

低コスト

接続数が減ると、直ちにケーブルやスイッチ、その他のネットワーク機器の数が減ることになります。また、イーサネットの例では規模の経済が機能します。市場におけるデバイス数が数百万から数十億に増えると、100Mb イーサネットとギガバイトのイーサネットデバイスの価格下落で見られたように、ネットワーク費用は劇的に低下します。

同様に、10GbE を採用する企業が増えるにつれて、その価格はより安価になります。また、CNA ハードウェアがシングルチップに統合されることで、その仕様拡大が市場でのボリュームを増大させ、長期的には価格が大幅に下落することになります。

iSCSI

インターネット SCSI (iSCSI) は別のタイプの集中型ネットワークプロトコルで、FCoE の代わりとなるものです。ファイバーチャンネルのように、iSCSI はネットワークを使ったブロックレベルのストレージを提供します。しかし iSCSI は完全な管理環境は提供しません。FCoE に対する iSCSI の利点は、iSCSI はファイバーチャンネルのほとんどの機能と柔軟性をより低コストで提供できるという点です。

---

[1] Red Hat 認定エンジニア。詳細は、http://www.redhat.com/training/certifications/rhce/を参照してください。

[2] **ハートビート** はシステム間でのメッセージ交換のことで、各システムが機能していることを伝えます。システムの「ハートビートがなくなる」と、失敗してシャットダウンしたと考えられ、別のシステムが引き継ぎます。

第2章 RED HAT ENTERPRISE LINUX 6 のパフォーマンス機能

2.1. 64 ビットのサポート

Red Hat Enterprise Linux 6 は 64 ビットのプロセッサーをサポートしています。これらのプロセッサーは理論的には、最大 16 **エクサバイト** のメモリーを使用できます。一般提供 (GA) の開始時点で、Red Hat Enterprise Linux 6.0 は最大 8 TB の物理メモリのサポートをテスト、検証しています。

Red Hat は、より大型のメモリブロックの使用を可能にする機能を継続して導入・改善するので、Red Hat Enterprise Linux 6 がサポートするメモリサイズはマイナーな更新を重ねるごとに拡大する予定です。(Red Hat Enterprise Linux 6 GA 時点での) 改善の例は以下の通りです。

- Huge pages および transparent huge pages
- Non-Uniform Memory Access (NUMA) の改善

これらの改善点については、後半のセクションで詳述されています。

Huge pages および transparent huge pages

Red Hat Enterprise Linux 6 に *huge pages* が実装されたことで、システムは異なるメモリーワークロードにまたがってメモリー使用を効率的に管理できるようになりました。Huge pages は標準の 4 KB ページと比較すると、動的に 2 MB ページを利用するので、アプリケーションはギガバイトやさらにはテラバイトのメモリー処理中にうまく拡張ができます。

Huge pages は手動で作成、管理、使用するには難しいものです。これに対処するため、Red Hat Enterprise 6 には *transparent huge pages* (THP) を使用する機能もあります。THP は、huge pages の使用に関わる多くの複雑性を自動的に管理します。

Huge pages および THP に関する詳細は、「Huge pages および transparent huge pages」 を参照してください。

NUMA の改善

新しいシステムの多くは *Non-Uniform Memory Access*(NUMA) をサポートしています。NUMA は、大規模システム用のハードウェアの設計および作成を簡素化します。しかし、アプリケーション開発に新たな複雑性を加えることにもなります。例えば、NUMA はローカルとリモートの両方でメモリーを実装し、リモートの方はローカルよりもアクセスに数倍の時間がかかります。この機能は、オペレーティングシステムやアプリケーションのパフォーマンスに影響を及ぼすので、慎重に設定してください。

Red Hat Enterprise Linux 6 は、NUMA システム上のアプリケーションとユーザーの役に立ついくつかの新機能によって、NUMA により最適化されています。これに含まれるのは、CPU アフィニティ、CPU ピン設定 (cpusets)、numactl および制御グループで、これらによってプロセス (affinity) もしくはアプリケーション (ピン設定) は特定の CPU または CPU セットに「バインド」することができます。

Red Hat Enterprise Linux 6 における NUMA サポートの詳細は、「CPU と NUMA トポロジー」 を参照してください。

2.2. TICKET SPINLOCKS

システム設定における要点は、あるプロセスが別のプロセスが使用しているメモリを変更しないようにすることです。メモリにおける未制御のデータ変更は、データ破損やシステムクラッシュにつながりかねません。これを防ぐには、プロセスがメモリの一部のロックし、オペレーションを実行し、その後にメモリをアンロックもしくは「自由」にすることをオペレーティングシステムが許可するようにします。

メモリロッキングのよくある実装は *spin locks* によるもので、これによりプロセスはロックが利用可能かどうかを継続してチェックできるようになり、利用可能になり次第にロックを取得します。複数のプロセスが同一ロックの取得で競合している場合は、ロックが利用可能になってから最初に要請したプロセスがこれを取得します。すべてのプロセスがメモリに同一のアクセスを持っている場合、このアプローチは「公平」で、うまく機能します。

残念ながら NUMA システム上では、すべてのプロセスがロックへ同等のアクセスを持っているわけではありません。ロックと同様の NUMA ノード上にあるプロセスには、ロック取得の不公平な優位性があります。リモート NUMA ノード上のプロセスはロック不足に陥り、パフォーマンスが低下します。

この問題に対処するため、Red Hat Enterprise Linux は *ticket spinlocks* を実装しました。この機能は、ロックへの予約キューメカニズムを追加し、**すべての** プロセスが要請した順番でロックを取得できるようにします。これにより、ロック要請におけるタイミングの問題と不公平な優位性を排除します。

Ticket spinlock のオーバーヘッドは通常の spinlock よりも多少大きくなりますが、拡張性に優れ、NUMA システム上でのすぐれたパフォーマンスを提供します。

2.3. 動的リスト構造

オペレーティングシステムは、システムの各プロセッサー上で情報セットを必要とします。Red Hat Enterprise Linux 5 では、この情報セットはメモリの固定サイズアレイに割り当てられました。各プロセッサー上の情報は、このアレイにインデックス化することで取得されました。この方法は迅速かつ容易で、比較的プロセッサーが少ないシステムには簡単なものでした。

しかし、システムのプロセッサー数が増加するにつれ、この方法は多大なオーバーヘッドを生み出します。メモリの固定サイズアレイは単一の共有リソースなので、多くのプロセッサーが同時にアクセスしようとすると、ボトルネックになってしまいます。

この問題に対処するため、Red Hat Enterprise Linux 6 ではプロセッサー情報に *動的リスト構造* を使います。これにより、プロセッサー情報用のアレイが動的に割り当てられるようになります。システムにプロセッサーが 8 つしかない場合、リストには 8 つのエントリしか作成されません。プロセッサーが 2048 個あれば、エントリは 2048 個作成されます。

動的リスト構造では、よりきめの細かいロッキングが可能になります。例えば、 プロセッサー 6、72、183、657、931、1546 で同時に情報の更新が必要な場合、より多くの並列処理で実行可能です。このような状況は当然、小型システムよりもより大型の高パフォーマンスシステムでより多く発生します。

2.4. TICKLESS カーネル

Red Hat Enterprise Linux の以前のバージョンでは、カーネルは継続的にシステム割り込みを作り出すタイマーベースのメカニズムを使用していました。各割り込み中に、システムは *ポーリング* していました。つまり、するべき作業があるかどうかをチェックしていました。

設定によっては、このシステム割り込みまたは *タイマーチック* は 1 秒間に数百回から数千回も発生する可能性があります。これはシステムのワークロードに関係なく、毎秒発生します。ロードが少ないシステムでは、プロセッサーが効果的にスリープ状態を使うことができず、**電力消費量** に影響を与えます。システムはスリープ状態にあると消費電力量が最小になります。

システムが最も省電力で稼働するには、作業を最短で終わらせ、深いスリープ状態にできるだけ長く入ることです。これを実行するために、Red Hat Enterprise Linux 6 は *tickless カーネル* を使用しています。これを使うことで、割り込みタイマーがアイドルループから取り除かれ、Red Hat Enterprise Linux 6 を完全に割り込み駆動型環境に変換します。

tickless カーネルでは、システムはアイドル時間中に深いスリープ状態に入ることができ、作業があるときはすぐに反応することができます。

詳細は、『電力管理ガイド』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

2.5. コントロールグループ

Red Hat Enterprise Linux は、パフォーマンスチューニング用の便利なオプションを数多く提供します。数百ものプロセッサーにまで拡張する大型システムにチューニングを行い、上質のパフォーマンスを提供することができます。しかし、これらのシステムのチューニングには、かなりの専門知識と明確に定義されたワークロードが必要になります。大型システムが高価で少数だった時には、これらを特別扱いしてもかまいませんでした。今ではこれらのシステムはメインストリームなので、より効果的なツールが必要になっています。

さらに複雑なことに、サービス統合により強力なシステムが今では使われています。昔は 4 から 8 台のサーバーで実行していたワークロードは、いまでは 1 台のサーバーでまかなわれています。「パラレルコンピューティング」 での説明にもあったように、現在の多くのミッドレンジシステムは、一昔前の高パフォーマンスマシンよりも多くのコアを備えています。

現在の多くのアプリケーションは並列処理向けに設計されており、パフォーマンスの改善に複数のスレッドもしくはプロセスを使用します。しかし、8 スレッド以上を効果的に使えるアプリケーションはほとんどありません。つまり、複数アプリケーションの能力を最大限活用するには、通常は 32-CPU システムにインストールする必要があります。

以下のの状況を考えてください。小規模で安価なメインストリームシステムが、今では一昔前の高価で高パフォーマンスだったマシンのパフォーマンスと変わりありません。安価でより高パフォーマンスのマシンがあることで、システムアーキテクトはより少数のマシンにより多くのサービスを統合できるようになっています。

しかし、(I/O やネットワーク通信といった) リソースは共有されており、CPU 数のようにはすぐに増えません。このため、複数アプリケーションを格納しているシステムは、1 つのアプリケーションがあるリソースを独占しすぎると、全体のパフォーマンスが低下することになります。

この問題に対処するため、Red Hat Enterprise Linux 6 は *コントロールグループ*(cgroups) をサポートします。Cgroups により、管理者は必要に応じてリソースを特定のタスクに割り当てることができます。例えば、4 つの CPU の 80% と60GB のメモリ、ディスク I/O の 40% をデータベースアプリケーションに割り当て、同一システ上で稼働している Web アプリケーションには、2 つの CPU と 2GB のメモリ、利用可能なネットワーク帯域幅の 50% を割り当てる、ということができます。

その結果、データベースアプリケーションと Web アプリケーションの両方が過剰にシステムリソースを消費することがなく、両方がすぐれたパフォーマンスを実現できるようになります。加えて、cgroups の多くの側面が **自己チューニング** なので、システムはワークロードの変化に応じで対応できます。

Cgroup には以下の 2 つの主要コンポーネントがあります。

- Cgroup に割り当てられたタスクのリスト
- これらのタスクに割り当てられたリソース

Cgroup に割り当てられたタスクは、cgroup **内で** 実行されます。そのタスクからの子タスクも、cgroup 内で実行されます。これにより、管理者はアプリケーション全体を単一ユニットとして管理できます。管理者はまた、以下のリソース配分も設定することができます。

- CPUsets
- メモリ
- I/O

- ネットワーク (帯域幅)

CPUsets 内では、cgroups は管理者による CPU 数や特定 CPU もしくはノード用のアフィニティ [3]、タスクセットが使用する CPU の時間の設定を可能にします。Cgroups を使った CPUsets の設定は、全体的なパフォーマンス向上を確保するために必須のものです。また、アプリケーションが CPU 時間不足に陥らない一方で、他のタスクを犠牲にしてあるアプリケーションがリソースを過剰に消費しないようにするためにも必須のものです。

I/O 帯域幅とネットワーク帯域幅は、他のリソースコントローラーが管理します。リソースコントローラーを使うことで、cgroup 内のタスクが消費する帯域幅を決定でき、cgroup 内のタスクがリソースを過剰消費したり、逆にリソース不足にならないようにします。

Cgroups を使うと、管理者は高レベルで様々なアプリケーションが必要とし消費するシステムリソースを定義・配分することができます。するとシステムは自動的にこれらのアプリケーションを管理してバランスを取り、すぐれた予測可能なパフォーマンスを実現し、システム全体のパフォーマンスを最適化します。

コントロールグループの使用に関する詳細情報は、『リソース管理ガイド』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

2.6. ストレージおよびファイルシステムの改善

Red Hat Enterprise Linux 6 は、ストレージとファイルシステム管理を改善する機能もいくつか備えています。このバージョンで注目すべき進歩は、ext4 と XFS のサポートです。ストレージおよびファイルシステムに関するパフォーマンス改善の総合的な説明は、*7章 ファイルシステム* を参照してください。

Ext4

Ext4 は Red Hat Enterprise Linux 6 のデフォルトのファイルシステムです。EXT ファイルシステムファミリーの第 4 世代になり、理論上では最大 1 エクサバイトのファイルシステムと 16 TB のシングルファイルをサポートします。Red Hat Enterprise Linux 6 は、最大 16 TB のファイルシステムと 16 TB のシングルファイルをサポートします。より大規模なストレージ能力に加えて ext4 の新機能には以下のものが含まれます。

- 範囲ベースのメタデータ
- 遅延割り当て
- ジャーナルチェック加算

Ext4 ファイルシステムの詳細については 「Ext4 ファイルシステム」 を参照してください。

XFS

XFS は堅牢かつ成熟した 64 ビットのジャーナリングファイルシステムで、単一ホスト上の非常に大規模なファイルおよびファイルシステムをサポートします。このファイルシステムは当初、SGI が開発したもので、非常に大型のサーバーおよびストレージアレイにおける長期間の稼働実績があります。XFS の機能には以下のものがあります。

- 遅延割り当て
- 動的配分のアイノード
- 未使用領域管理のスケーラビリティ用の B ツリーインデックス化
- オンラインデフラグおよびファイルシステム拡張

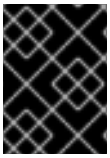
- 高度なメタデータ先読みアルゴリズム

XFS はエクサバイトまで拡張しますが、Red Hat がサポートする XFS ファイルシステムの最大サイズは 100TB です。XFS に関する詳細は「XFS ファイルシステム」を参照してください。

大規模ブートドライブ

従来の BIOS は、最大 2.2TB のディスクサイズをサポートしてきました。BIOS を使用する Red Hat Enterprise Linux 6 は、*Global Partition Table* (GPT) と呼ばれる新たなディスク構造を使うことで 2.2TB 以上のディスクをサポートします。GPT は、データディスクにのみ使用可能です。GPT は BIOS を使うブートドライブには使用できないので、ブートドライブの最大サイズは 2.2TB になります。BIOS は当初、IBM PC 用に作成されました。BIOS が最新のハードウェアに適応するように大幅に進化した一方で、*Unified Extensible Firmware Interface* (UEFI) は新たな最新ハードウェアをサポートするように設計されています。

Red Hat Enterprise Linux 6 は UEFI もサポートします。これは BIOS (まだサポート中) に取って代わることが可能です。UEFI を備え Red Hat Enterprise Linux 6 を実行するシステムでは、起動パーティションとデータパーティションの両方で GPT と 2.2TB (またはそれ以上) のパーティションを使うことができます。



重要

Red Hat Enterprise Linux 6 では 32 ビット x86 システム向け UEFI はサポートしていません。

重要

UEFI と BIOS の起動設定は、大幅に異なります。このため、インストール済みのシステムはインストール時に使用されたファームウェアと同じものを使って起動する必要があります。BIOS を使用するシステム上でオペレーティングシステムをインストールしてから、UEFI を使用するシステム上でこのインストールを起動することはできません。

Red Hat Enterprise Linux 6 では、UEFI 仕様のバージョン 2.2 をサポートしています。UEFI 仕様のバージョン 2.3 またはそれ以降に対応するハードウェアであれば、Red Hat Enterprise Linux 6 で起動および稼働しますが、これら最新の仕様で定義されている追加機能は利用できません。UEFI 仕様は http://www.uefi.org/specs/agreement/ から取得できます。

---

[3] ノードは通常、ソケット内の CPU もしくは コアのセットと定義されます。

第3章 システムパフォーマンスのモニタリングと分析

本章では、システムおよびアプリケーションのパフォーマンスをモニター・分析するためのツールを簡潔に紹介し、各ツールが最も有効に使用できる状況を提示します。最高レベルを下回るパフォーマンスの原因となっているボトルネックや他のシステム問題は、各ツールが収集するデータによって明らかになります。

3.1. PROC ファイルシステム

proc「ファイルシステム」は、Linux カーネルの現在の状況を示すファイルの階層が含まれているディレクトリーです。これにより、アプリケーションとユーザーはシステムのカーネルビューを確認することができます。

proc ディレクトリーには、システムのハードウェアおよび実行中のプロセスに関する情報が格納されています。これらファイルのほとんどは読み取り専用ですが、(主に **/proc/sys** 内の) ファイルのなかには、ユーザーやアプリケーションが操作することで、設定変更をカーネルに伝達できるものもあります。

proc ディレクトリー内のファイルの表示および編集に関する詳細情報は、『導入ガイド』を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

3.2. GNOME および KDE システムモニタ

GNOME および KDE の両デスクトップ環境には、システム動作を監視・修正する際に役立つグラフィカルツールがあります。

GNOME システムモニタ

GNOME システムモニタ は基本的なシステム情報を表示し、システムプロセスやリソース、ファイルシステムの使用量を監視することができます。GNOME システムモニタを開くには、**端末** で **gnome-system-monitor** コマンドを実行するか、**アプリケーション** メニューをクリックして **システムツール > システム・モニタ** を選択します。

GNOME システム・モニタ には 4 つのタブがあります。

システム

コンピューターのハードウェアおよびソフトウェアの基本情報を表示します。

プロセス

実行中のプロセス、それらのプロセスの関係、各プロセスの詳細情報を表示します。表示されたプロセスにフィルターをかけたり、それらのプロセスに特定のアクションを実行することもできます (スタート、ストップ、強制終了、優先順位の変更、など)。

リソース

現在の CPU 使用時間、メモリおよびスワップ領域の使用量、ネットワークの使用量を表示します。

ファイルシステム

マウントされているファイルシステムの全一覧と、ファイルシステムのタイプやマウントポイント、メモリ使用量などの各ファイルシステムの基本情報が表示されます。

GNOME システム・モニタ についての詳細はアプリケーションの**ヘルプ** メニュー、もしくは 『導入ガイド』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

KDE System Guard

KDE System Guard では、現在のシステム負荷と実行中のプロセスが確認できます。また、プロセスに対するアクションも実行できます。KDE System Guardを開くには、**端末** で **ksysguard** コマンドを実行するか、**Kickoff Application Launcher** をクリックし **アプリケーション > システム > システム・モニタ** を選択します。

KDE System Guard には 2 つのタブがあります。

Process Table (プロセステーブル)

実行中のプロセス一覧をデフォルトではアルファベット順に表示します。また、CPU 使用量や物理もしくは共有メモリ使用量、所有者、優先順位などのプロパティーでプロセスを分類することもできます。表示された結果をフィルターにかけたり、特定のプロセスを検索したり、プロセスに対して特定のアクションを実行したりすることもできます。

System Load (システム負荷)

CPU 使用量、メモリおよびスワップ領域使用量、ネットワーク使用量の履歴を表示します。グラフ上にマウスを持って行くと、詳細な分析とグラフのキーが表示されます。

KDE System Guard の詳細については、アプリケーション内の**ヘルプ** メニューを参照してください。

3.3. PERFORMANCE CO-PILOT (PCP)

Performance Co-Pilot (PCP) は、システムパフォーマンスの詳細を監視、分析し、それに対応するツールとインフラストラクチャーを提供します。PCP は完全に分散型のアーキテクチャーなので、ユーザーのニーズに応じて単一ホスト上で稼働したり、複数ホスト上で稼働することができます。また PCP はプラグインアーキテクチャーと設計されているので、幅広いサブシステムの監視とチューニングに便利なものとなっています。

PCP をインストールするには、以下のコマンドを実行します。

```
# yum install pcp
```

Red Hat では、収集データの視覚化の機能を提供する pcp-gui パッケージと、詳細な PCP ドキュメンテーションを **/usr/share/doc/pcp-doc** ディレクトリーにインストールする pcp-doc パッケージも推奨しています。

3.3.1. PCP アーキテクチャー

PCP の導入は、コレクターシステムとモニターシステムの両方で構成されています。単一ホストをコレクターとモニターの両方にしたり、またはコレクターとモニターを複数のホストにまたがって分散させることもできます。

コレクター

コレクターシステムは 1 つ以上のドメインからパフォーマンスデータを収集し、分析するために保存します。コレクターには Performance Metrics Collector Daemon (**pmcd**) があり、これはパフォーマンスデータのリクエストを適切な Performance Metrics Domain Agent および 1 つ以上の

Performance Metrics Domain Agents (PMDA) と受け渡します。これらのエージェントは、該当ドメイン (データベース、サーバー、アプリケーションなど) についてのリクエストに関する反応を担います。PMDA は、同一コレクター上で実行している **pmcd** で制御されます。

モニター

モニターシステムは、**pmie** や **pmreport** などのモニタリングツールを使って、ローカルもしくはリモートコレクターからのデータを表示、分析します。

3.3.2. PCP の設定

コレクターは、実行中の Performance Metrics Collector Daemon (**pmcd**) と 1 つ以上の Performance Metrics Domain Agents (PMDA) を必要とします。

手順3.1 コレクターの設定

1. **PCP のインストール**
   以下のコマンドを実行して、システムに PCP をインストールします。

   ```
   # yum install pcp
   ```

2. **pmcd の起動**

   ```
   # service pmcd start
   ```

3. **追加 PMDA の設定 (オプション)**
   カーネル、pmcd、per-process、memory-mapped の値、XFS、および JBD2 PMDA はデフォルトで、**/etc/pcp/pmcd/pmcd.conf** にインストールされ、ここで設定します。

   追加の PMDA を設定するには、そのディレクトリーに移動し (たとえば、**/var/lib/pcp/pmdas/*pmdaname***)、**Install** スクリプトを実行します。たとえば、**proc** 用の PMDA をインストールするには、以下を実行します。

   ```
   # cd /var/lib/pcp/pmdas/proc
   # ./Install
   ```

   そしてプロンプトにしたがい、コレクターシステムまたはモニターおよびコレクターシステム用に PMDA を設定します。

4. **リモートモニターをリッスン (オプション)**
   リモートモニターに反応するには、pmcd はポート 44321 経由でリモートモニターをリッスンできる必要があります。以下を実行して、適切なポートを開きます。

   ```
   # iptables -I INPUT -p tcp -dport 44321 -j ACCEPT
   # iptables save
   ```

   他のファイアウォールルールがこのポートへのアクセスをブロックしていないことを確認する必要があります。

モニターは、pcp がインストールされ、リモートもしくはローカルの pmcd インスタンスに少なくとも 1 つ接続できる必要があります。コレクターとモニターの両方が単一マシン上にあり、手順3.1「コレクターの設定」 の指示にしたがっている場合は、これ以上の設定は不要で、PCP が提供するモニタリングツールを使用することができます。

手順3.2 リモートモニターの設定

1. **PCP のインストール**
   以下のコマンドを実行して、リモートモニターシステムに PCP をインストールします。

   ```
   # yum install pcp
   ```

2. **リモートコレクターへの接続**
   リモートコレクターシステムに接続するには、PCP はポート **44321** 経由でリモートコレクターに連絡できる必要があります。以下を実行して、適切なポートを開きます。

   ```
   # iptables -I INPUT -p tcp -dport 44321 -j ACCEPT
   # iptables save
   ```

   他のファイアウォールルールがこのポートへのアクセスをブロックしていないことを確認する必要もあります。

これで、**-h** オプションを付けてパフォーマンスモニタリングツールを実行することで、リモートコレクターに接続できるかどうかをテストできます。このオプションは、接続先のコレクターの IP アドレスを指定します。例を示します。

```
# pminfo -h 192.168.122.30
```

3.4. IRQBALANCE

irqbalance はコマンドラインツールで、プロセッサーにハードウェア割り込みを配布して、システムパフォーマンスを向上させます。デフォルトではデーモンとして実行されますが、**--oneshot** オプションで 1 回きりの実行も可能です。

パフォーマンス改善には、以下のパラメーターが便利です。

--powerthresh

CPU が省電力モードになる前にアイドル状態になることが可能な CPU 数を設定します。しきい値を超える CPU 数が平均 **softirq** ワークロードを 1 標準偏差以上下回り、平均値から 1 標準偏差を超える CPU がなく、複数の **irq** がこれらに割り当てられている場合、CPU は省電力モードに入ります。このモードでは、CPU は **irq** バランシングの一部ではないので、不必要に再開されることがありません。

--hintpolicy

irq カーネルアフィニティーヒンティングの処理方法を決定します。有効な値は、**exact** (**irq** アフィニティーヒンティングを常に適用)、**subset** (**irq** はバランシングされるものの、割り当てオブジェクトはアフィニティーヒンティングのサブセット)、または **ignore** (**irq** アフィニティーヒンティングを完全に無視) になります。

--policyscript

各割り込みリクエストを実行するスクリプトの場所を定義します。デバイスパスおよび **irq** 番号が引数として渡され、**irqbalance** はゼロ終了コードを予想します。定義されたスクリプトでは、渡された **irq** の管理で **irqbalance** をガイドするために、ゼロもしくはそれ以上の鍵の値のペアを指定することができます。

有効な鍵の値のペアとして認識されるのは、以下のものです。

ban

有効な値は、**true** (バランシングから渡された irq を除外) もしくは **false** (この irq でバランシングを実施) です。

balance_level

渡された irq のバランシングレベルをユーザーが上書きできるようにします。デフォルトでは、バランスレベルは、irq を所有するデバイスの PCI デバイスクラスに基づきます。有効な値は、**none, package**、**cache**、または **core** になります。

numa_node

渡された irq にはローカルとみなされる NUMA ノードを、ユーザーが上書きできるようにします。ローカルノードについての情報が ACPI で指定されていない場合は、デバイスはすべてのノードから等距離とみなされます。有効な値は、特定の NUMA ノードを識別する整数 (0 から)、および irq が全ノードから等距離であるとみなすことを指定する **-1** になります。

--banirq

指定された割り込みリクエスト番号のある割り込みが禁止された割り込みリストに追加されます。

また、***IRQBALANCE_BANNED_CPUS*** 環境変数を使って、**irqbalance** によって無視される CPU のマスクを指定することもできます。

詳細は、以下の man ページを参照してください。

```
$ man irqbalance
```

3.5. ビルトインのコマンドラインモニタリングツール

グラフィカルなモニタリングツールの他に、Red Hat Enterprise Linux はコマンドラインからシステムを監視できるツールをいくつか提供しています。これらのツールの利点は、ランレベル 5 以外で使用できることです。このセクションでは各ツールについて簡潔に説明し、それぞれのツールの最適な使用方法を提案します。

top

top ツールは、実行中のシステムのプロセスを動的かつリアルタイムで表示します。システムサマリーや Linux カーネルが現在管理しているタスクなどの様々な情報の表示が可能です。また、プロセスを操作する一定の機能もあります。表示される操作と情報はどちらとも高度な設定が可能で、設定詳細は再起動後も維持するようにできます。

表示されるプロセスは、デフォルトでは CPU 使用量順となっており、多くのリソースを消費しているプロセスがひと目で分かるようになっています。

top の使用方法の詳細は、man ページ: **man top** を参照してください。

ps

ps ツールは、選択したアクティブなプロセスグループのスナップショットを撮ります。デフォルトでは、このグループは現行ユーザーが所有しているもので同一端末に関連するプロセスに限られています。

このツールはプロセスに関して **top** よりも詳細な情報を提供しますが、動的ではありません。

psの使用方法の詳細は、man ページ: **man ps** を参照してください。

vmstat

vmstat (Virtual Memory Statistics) は、システムのプロセス、メモリ、ページング、ブロック I/O、割り込み、CPU アクティビティについてのある時点でのレポートを出力します。

top のように動的ではありませんが、サンプリング間隔を指定できるのでニアリアルタイムのシステムアクティビティを監視することができます。

vmstat の使用方法の詳細は、man ページ: **man vmstat** を参照してください。

sar

sar (System Activity Reporter) は、当日のその時点までのシステムアクティビティを収集し、レポートします。デフォルトでは、その日の最初から 10 分間隔で CPU 使用量を出力します。

```
12:00:01 AM     CPU     %user     %nice   %system   %iowait    %steal
%idle
12:10:01 AM     all      0.10      0.00      0.15      2.96      0.00
96.79
12:20:01 AM     all      0.09      0.00      0.13      3.16      0.00
96.61
12:30:01 AM     all      0.09      0.00      0.14      2.11      0.00
97.66
...
```

このツールは、**top** や同様のツールによるシステムアクティビティに関する定期的なレポートの代わりとなる便利なものです。

sar の使用方法の詳細は、man ページ: **man sar** を参照してください。

3.6. TUNED および KTUNE

Tuned は、様々なシステムコンポーネントの使用量に関するデータを監視、収集するデーモンで、その情報を使って必要に応じてシステム設定を動的にチューニングします。Tuned は CPU やネットワーク使用量の変化に対応し、設定を変更することでアクティブなデバイスのパーティションを改善したり、非アクティブなデバイスの電力消費量を抑えたりすることができます。

Tuned に付随する **ktune** は **tuned-adm** ツールとともに、多くの事前設定されたチューニングプロファイルを提供します。これらは数多くの特定のユースケースでパフォーマンスを強化し、電力消費量を抑制します。これらのプロファイルを編集したり、新規プロファイルを作成したりすることで、使用中の環境に適合したパフォーマンスソリューションを作成することができます。

tuned-adm の一部として提供されるプロファイルには、以下のものがあります。

default

デフォルトの省電力プロファイルで、最も基本的な省電力プロファイルです。ディスクと CPU プラグインのみを使用可能にします。これは、**tuned-adm** をオフにすることとは違います。tuned-adm をオフにすると、**tuned** と **ktune** の両方が使用できません。

latency-performance

標準的な遅延パフォーマンスチューニング用のサーバープロファイルです。このプロファイルは、動的チューニングメカニズムと transparent hugepages を無効にします。これは、**cpuspeed** で p 状態用に **performance** ガバナーを使用し、I/O スケジューラーを **deadline** に設定します。また

Red Hat Enterprise Linux 6.5 およびそれ以降では、プロファイルは ***cpu_dma_latency*** の値 **1** を要求します。Red Hat Enterprise Linux 6.4 およびそれ以前では、**cpu_dma_latency** は **0** の値を要求していました。

throughput-performance

標準的なスループットのパフォーマンスチューニング用のサーバープロファイルです。システムにエンタープライズクラスのストレージがない場合に、このプロファイルが推奨されます。throughput-performance は省電力メカニズムを無効にし、**deadline** I/O スケジューラーを有効にします。CPU ガバナーは、**performance** に設定されます。***kernel.sched_min_granularity_ns*** (スケジューラーの最小の先取り粒度) は **10** ミリ秒に、***kernel.sched_wakeup_granularity_ns*** (スケジューラーの再開粒度) は **15** ミリ秒に、***vm.dirty_ratio*** (仮想メモリー汚染比率) は 40% に設定され、transparent huge pages が有効になります。

enterprise-storage

このプロファイルは、バッテリー駆動のコントローラーキャッシュ保護とオンディスクのキャッシュ管理などを含むエンタープライズクラスのストレージがあるエンタープライズサイズのサーバー設定に推奨されます。**throughput-performance** プロファイルと同一ですが、ファイルシステムが **barrier=0** で再マウントされています。

virtual-guest

このプロファイルは、仮想マシン用に最適化されています。***enterprise-storage*** プロファイルをベースとしていますが、***virtual-guest*** も仮想メモリーの swap を低減します。このプロファイルは Red Hat Enterprise Linux 6.3 以降で利用可能です。

virtual-host

enterprise-storage プロファイルに基づき、***virtual-host*** も仮想メモリーの swap を減らし、ダーティーページのより積極的な書き戻しを可能にします。**barrier=0** で root ファイルシステムおよび起動ファイルシステム以外のファイルシステムがマウントされます。さらに Red Hat Enterprise Linux 6.5 では、***kernel.sched_migration_cost*** パラメーターが **5** ミリ秒に設定されます。Red Hat Enterprise Linux 6.5 より前では、***kernel.sched_migration_cost*** はデフォルト値の **0.5** ミリ秒を使用していました。このプロファイルは Red Hat Enterprise Linux 6.3 およびそれ以降で利用可能になっています。

tuned および **ktune** の詳細情報については、Red Hat Enterprise Linux 6 『電力管理ガイド』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

3.7. アプリケーションプロファイラー

プロファイリングは、プログラムの実行中にそのプログラムの動作についての情報を収集するプロセスです。アプリケーションのプロファイリングは、そのプログラムの全体的なスピードを高め、メモリ使用量を減らすためにプログラムのどの分野を最適化するかを決定するために行います。アプリケーションのプロファイリングツールは、このプロセスの簡素化に役立ちます。

Red Hat Enterprise Linux 6 では、**SystemTap**、**OProfile**、**Valgrind** の 3 つのプロファイリングツールがサポートされています。これらのプロファイリングツールに関する資料は本ガイドの対象外となりますが、このセクションでは詳細情報のリンクと各プロファイラーに適したタスクの簡単な概要を提供します。

3.7.1. SystemTap

SystemTap は、トレーシング/プロービングツールです。このツールによりユーザーは、オペレーティングシステム (特にカーネル) のアクティビティの詳細な監視と分析を行うことができます。SystemTap は、**netstat**、**ps**、**top**、**iostat** などのツールの出力と同様の情報を提供します。ただし、SystemTap には、収集した情報に対する追加のフィルタリングと分析オプションが含まれます。

SystemTap は、システムのアクティビティおよびアプリケーション動作のより深く、正確な分析を提供するので、システムとアプリケーションのボトルネックの位置がより正確に分かります。

Eclipse 用の Function Callgraph プラグインは、SystemTap をバックエンドとして使用します。これにより、関数呼び出しやリターン、時間、ユーザースペースの変数などを含むプログラムのステータスの完全な監視が可能になり、これらの情報が表示されることで最適化が視覚的に容易になります。

『Red Hat Enterprise Linux 7 SystemTap Beginner's Guide』 には、パフォーマンスのプロファイリングやモニタリングに便利なスクリプトのサンプルがいくつか含まれています。これらはデフォルトで、**/usr/share/doc/systemtap-client-*version*/examples** ディレクトリーにインストールされます。

ネットワークモニタリングスクリプト (examples/network 内)

nettop.stp

5 秒ごとにプロセス (プロセス ID およびコマンド) のリストをプリントします。これには、この間隔で送受信されたパケット数やプロセスが送受信したデータ量が含まれます。

socket-trace.stp

Linux カーネルの **net/socket.c** ファイル内の各関数をインストルメント化して、追跡データをプリントします。

tcp_connections.stp

システムが受け入れた新たな受信 TCP 接続についての情報をプリントします。この情報には UID、接続を受け入れているコマンド、コマンドのプロセス ID、接続しているポート、リクエストの発信元の IP アドレスが含まれます。

dropwatch.stp

カーネル内の場所で解放されたソケットバッファー数を 5 秒ごとにプリントします。シンボリック名を見るには、**--all-modules** オプションを使用します。

ストレージモニタリングスクリプト (examples/io 内)

disktop.stp

読み込み/書き込みディスクの状態を 5 秒ごとに確認し、この期間の上位 10 位のエントリーを出力します。

iotime.stp

読み取り/書き込み操作に費やされた時間と読み取り/書き込みのバイト数をプリントします。

traceio.stp

観測された累積 I/O トラフィックに基づいて上位 10 位の実行可能ファイルを毎秒プリントします。

traceio2.stp

指定されたデバイスへの読み取りおよび書き込みが発生する際に実行可能ファイル名およびプロセス ID をプリントします。

inodewatch.stp

指定されたメジャー/マイナーデバイス上の指定された inode で読み取りおよび書き込みが発生するたびに、実行可能ファイル名とプロセス ID をプリントします。

inodewatch2.stp

指定されたメジャー/マイナーデバイス上の指定された inode で属性が変更されるたびに、実行可能ファイル名、プロセス ID、属性をプリントします。

latencytap.stp スクリプトは、異なるタイプの遅延が 1 つ以上のプロセスに及ぼす影響を記録します。30 秒ごとに遅延タイプのリストをプロセスの時間またはプロセスが待機した時間の合計の多い順で分類してプリントします。これは、ストレージとネットワークの両方の遅延の原因特定で便利なものです。Red Hat では、このスクリプトで **--all-modules** オプションを使用して遅延イベントのマッピングを有効にすることを推奨しています。デフォルトでは、このスクリプトは **/usr/share/doc/systemtap-client-*version*/examples/profiling** ディレクトリーにインストールされます。

SystemTap についての詳細は 『SystemTap Beginners Guide』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

3.7.2. OProfile

OProfile (**oprofile**) はシステム全体のパフォーマンス監視ツールです。プロセッサーにある専用のパフォーマンス監視ハードウェアを使用して、メモリの参照時期、L2 キャッシュ要求の回数、ハードウェア割り込みの受信回数など、カーネルとシステムの実行可能ファイルに関する情報を引き出します。また、プロセッサーの使用量やどのアプリケーションやサービスが最も使われているかなどの判断にも使用できます。

OProfileは、Eclipse OProfile プラグインを使って Eclipse でも使用可能です。このプラグインを使うと最も時間のかかるコードの分野を容易に見つけられ、OProfile のコマンドライン機能すべてが実行できます。また、この結果は高度に視覚化されたものになります。

ただし、OProfile には以下の制限があることに注意してください。

- パフォーマンスのモニタリングサンプルが正確でない場合があります。プロセッサーは順番通りに指示を実行しない場合があるので、サンプルは割り込みを発生させた指示ではなく、近くの指示から記録される場合があります。
- OProfile はシステム全体にわたり、プロセスが複数回にわたって開始・停止することを想定しているので、複数の実行からのサンプルが集積されます。つまり、以前の実行からのサンプルを削除する必要がある場合があります。
- これは CPU 限定のプロセスでの問題の識別にフォーカスするので、他のイベント発生をロック状態で待つ間、スリープとなっているプロセスは識別しません。

OProfile の使用に関する詳細は、『導入ガイド』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。または、システム上の **oprofile** ドキュメンテーションを参照してください。これは **/usr/share/doc/oprofile-<*version*>** にあります。

3.7.3. Valgrind

Valgrind は、アプリケーションのパフォーマンスと正確性の改善に役立つ数多くの検出およびプロファイリングツールを提供します。これらのツールは、ヒープ、スタック、アレイのオーバーランに加えて、メモリおよびスレッド関連のエラーを検出するので、アプリケーションコード内のエラーの特定と修正が容易になります。また、キャッシュやヒープ、分岐予測のプロファイリングを行い、アプリケーション速度を高め、アプリケーションのメモリ使用量を最小化できる可能性のある要素を特定することもできます。

Valgrind は、アプリケーションを統合 CPU 上で実行し、既存のアプリケーションコードを実行中にインストルメント化することでアプリケーションを分析します。その後に「コメント」を付けることで、アプリケーション実行に関わった各プロセスをユーザー指定のファイル記述子、ファイル、ネットワークソケットに対して明確に識別します。インストルメント化のレベルは、使用している Valgrind ツールとその設定によって異なりますが、インストルメント化されたコードの実行は通常のコードの 4-50 倍の時間がかかることに留意してください。

Valgrind は、再コンパイルせずにそのままアプリケーション上で使用できます。しかし、Valgrind はコード内の問題の特定にデバッグ情報を使うので、アプリケーションおよびサポートライブラリが有効なデバッグ情報でコンパイルされていない場合は、この情報を含めるように再コンパイルすることが強く推奨されます。

Red Hat Enterprise Linux 6.4 では、Valgrind は gdb (GNU Project Debugger) と統合してデバッグ効率を高めます。

Valgrind に関する詳細は 『開発者ガイド』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。または、valgrind インストール時に **man valgrind** コマンドを使用してください。付随資料は以下の場所にもあります。

- **/usr/share/doc/valgrind-<version>/valgrind_manual.pdf**
- **/usr/share/doc/valgrind-<version>/html/index.html**

Valgrind を使ってシステムメモリをプロファイリングする方法については、「Valgrind を使ったメモリ使用量のプロファイリング」 を参照してください。

3.7.4. Perf

perf ツールは多くの便利なパフォーマンスカウンターを提供し、これを使うことでユーザーはシステムに対する他のコマンドの影響を評価できます。

perf stat

このコマンドは、実行された指示や消費されたクロックサイクルなどを含む一般的なパフォーマンスイベントの全体的な統計を提供します。オプションフラグを使ってデフォルト測定イベント以外のイベントの統計を収集することができます。Red Hat Enterprise Linux 6.4 では、**perf stat** を使って 1 つ以上の指定コントロールグループ (cgroups) を基にモニタリングにフィルターをかけることができます。詳細は、man ページ: **man perf-stat** を参照してください。

perf record

このコマンドは、パフォーマンスデータを記録します。これは後で **perf report** を使って分析することができます。詳細については、man ページ: **man perf-record** を参照してください。

Red Hat Enterprise Linux 6.6 では、**-b** および **-j** のオプションが提供され、ブランチのサンプリングが可能になっています。**-b** オプションは選択されたブランチのサンプリングを行い、**-j** オプションはユーザーレベルやカーネルレベルのブランチなどの異なるタイプのブランチのサンプルを

行うように調整できます。

perf report

このコマンドは、ファイルからパフォーマンスデータを読み取り、記録されたデータを分析します。詳細については、man ページ: **man perf-report** を参照してください。

perf list

このコマンドは、特定のマシン上で利用可能なイベントを一覧表示します。これらのイベントは、パフォーマンスモニタリングハードウェアとシステムのソフトウェア設定によって異なります。詳細については man ページ: **man perf-list** を参照してください。

perf mem

Red Hat Enterprise Linux 6.6 から利用可能となっているこのコマンドは、指定されたアプリケーションまたはコマンドが実行するメモリーアクセス操作の各タイプの頻度をプロファイリングします。これによりユーザーは、プロファイルされたアプリケーションが実行する読み込みおよび保存操作の頻度を見ることができます。詳細は、man ページ: **man perf-mem** を参照してください。

perf top

このコマンドは、**top** ツールとよく似た機能を実行します。リアルタイムでパフォーマンスカウンタープロファイルを生成し、表示します。詳細については man ページ: **man perf-top** を参照してください。

perf trace

このコマンドは、**strace** ツールと同様の機能を実行します。指定されたスレッドまたはプロセスが使用するシステムコールとそのアプリケーションが受信するすべてのシグナルを監視します。追加の追跡ターゲットも使用可能です。完全なリストは、man ページを参照してください。

perf についての詳細情報は、Red Hat Enterprise Linux 『Developer Guide』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

3.8. RED HAT ENTERPRISE MRG

Red Hat Enterprise MRG の Realtime コンポーネントには、**Tuna** が含まれます。このツールを使うと、ユーザーはシステムのチューニング可能な値を調整し、その変更の結果を見ることができます。これは Realtime コンポーネントとの使用のために開発されましたが、標準の Red Hat Enterprise Linux システムのチューニングにも使用できます。

Tuna を使うと、不要なシステムアクティビティを調節したり、無効にしたりすることができます。例えば、以下のようなものです。

- 電力管理、エラー検出、システム管理割り込みに関連する BIOS パラメーター
- 割り込みコアレッシングなどのネットワーク設定および TCP の使用
- ジャーナリングファイルシステム内でのジャーナリングアクティビティ
- システムロギング
- 割り込みおよびユーザープロセスが特定の CPU もしくはいくつかの CPU で処理されたかどうか

- Swap 領域が使われたかどうか
- out-of-memory 例外の対処方法

Tuna インターフェースを使った Red Hat Enterprise MRG のチューニングに関する詳細な概念情報は、『Realtime チューニングガイド』 の「一般的なシステムチューニング」の章を参照してください。Tuna インターフェース使用に関する詳細な指示は、『Tuna User Guide』 を参照してください。両ガイドとも http://access.redhat.com/site/documentation/Red_Hat_Enterprise_MRG/ から入手できます。

第4章 CPU

central processing unit(中央処理装置) を表す CPU という言葉は、実体とかけ離れた呼称です。というのも、**central** は **single** (シングル) を意味しますが、ほとんどの最近のシステムには 2 つ以上の処理装置、コア、があるからです。物理的には、CPU は マザーボードに取り付けられているパッケージの**ソケット**内に格納されています。マザーボード上の各ソケットには、様々な接続があります。例えば、別の CPU ソケット、メモリコントローラー、割り込みコントローラー、他の周辺機器への接続といったものです。オペレーティングシステムへのソケットは CPU および関連リソースの論理グループ化です。この概念は、CPU チューニングの議論の中心となります。

Red Hat Enterprise Linux は、システム CPU イベントについての多大な統計を保持します。これらの統計は、CPU パフォーマンス改善のためのチューニング戦略を考える際に有用なものです。「CPU パフォーマンスのチューニング」 では、特に有益な統計、それらが見つかる場所、パフォーマンスチューニングのためにそれらを分析する方法について説明します。

トポロジー

古いコンピューターではシステムあたりの CPU 数が比較的少なく、*Symmetric Multi-Processor* (SMP) と呼ばれるアーキテクチャーを可能にしていました。つまり、システム内の各 CPU には利用可能なメモリに対して同様の (またはそれに釣り合った) アクセスがありました。最近では、ソケットあたりの CPU 数が非常に多くなったので、システム内の全 RAM に対して釣り合いのとれたアクセスを与えることは非常に高価なことになってしまいます。CPU 数の多いシステムのほとんどでは今日、 SMP ではなく *Non-Uniform Memory Access*(NUMA) と呼ばれるアーキテクチャーが使われています。

AMD プロセッサーは *Hyper Transport* (HT) 相互接続にこのタイプのアーキテクチャーを使っており、Intel は *Quick Path Interconnect* (QPI) 設計で NUMA の実装を開始しました。アプリケーションにリソースを配分する際にはシステムの**トポロジー**を占める必要があるので、NUMA と SMP のチューニング方法は違うものになります。

スレッド

Linux オペレーティングシステムでは、実行の単位は *スレッド* と呼ばれます。スレッドには、レジスタコンテキスト、スタック、実行可能なコードのセグメントがあり、これらは CPU 上で実行されます。利用可能な CPU でこれらのスレッドのスケジュール管理をするのがオペレーティングシステムの役割です。

OS は、利用可能なコアにまたがってスレッドの負荷を分散させることで、CPU 使用率を最大化します。OS が最も注意を払うのは CPU を稼働させることなので、アプリケーションのパフォーマンスに関しては最適な判断をしません。アプリケーションスレッドを別のソケットにある CPU に移動することで、単に現行 CPU が利用可能になるまで待機するよりもパフォーマンスが落ちる場合があります。これは、メモリアクセス操作がソケット全域で大幅に遅くなる可能性があるからです。高パフォーマンスアプリケーションの場合、通常はスレッドの格納場所については設計者が判断した方がよいとされています。「CPU のスケジューリング」 では、アプリケーションスレッドの実行における CPU とメモリの最善の配分方法を説明しています。

割り込み

アプリケーションのパフォーマンスに影響を与えるシステムイベントのうちで、あまり明確でないもの (ただし重要なもの) に *割り込み* (Linux では IRQ とも呼ぶ) があります。これらのイベントは OS が処理し、データの到着や、ネットワークを介した書き込みやタイマーイベントなどの操作の完了を知らせるために周辺機器が使用します。

アプリケーションコードを実行する OS や CPU が割り込みを扱う手法がアプリケーションの機能性に影響を与えることはありません。しかし、アプリケーションのパフォーマンスに影響を与える可能性があります。本章では、割り込みがアプリケーションのパフォーマンスにマイナスの影響を与えないようにするヒントについても説明します。

4.1. CPUトポロジー

4.1.1. CPU と NUMA トポロジー

最初のコンピュータープロセッサーは **uniprocessors**、つまりシステムには 1 つの CPU があるだけでした。単一の CPU を 1 つの実行 (プロセス) スレッドから別のスレッドに迅速にスイッチすることで、オペレーティングシステムはプロセスの並列処理という幻想を作り出していました。システムパフォーマンスを高めるために、設計者は指示を実行するためのクロック率を高めても、これが機能するのはある地点まで (通常は、現在の技術で安定的なクロック波形を作成できる限界) であることに気付きました。システムパフォーマンス全体を高めるために、設計者はもう 1 つの CPU をシステムに追加して、指示を 2 つの並列ストリームで実行できるようにしました。プロセッサーを追加するというこのトレンドが長らく続くことになります。

初期のマルチプロセッサーシステムでは、各 CPU がメモリ位置へ同一の論理パスを持っている場合がほとんどでした (通常はパラレルバス)。これにより、CPU はシステム内の他の CPU と同じ時間でメモリにアクセスすることができました。このタイプのアーキテクチャーは、Symmetric Multi-Processor (SMP) と呼ばれます。SMP は CPU の数が少ない場合はうまく機能しましたが、一定数 (8 または 16) を超えると、メモリへの同等アクセスに必要な並列トレースの数がボード面積を使い過ぎてしまい、周辺機器のスペースが不足してしまいました。

2 つの新しい概念が組み合わされて、システム内で多数の CPU が可能になりました。

1. シリアルバス
2. NUMA トポロジー

シリアルバスは、クロック率が非常に高い、単一ワイヤーパスで、データをパッケージ化されたバーストとして移動します。ハードウェア設計者は、シリアルバスを CPU 間や CPU とメモリコントローラー、他の周辺機器との間の高速の相互接続として使用し始めました。つまり、ボード上で **各** CPU からメモリサブシステムへ 32 から 64 のトレースを必要とする代わりに、**1 つの**トレースを必要とするので、ボード上での必要なスペースが大幅に削減されました。

同時に、ハードウェア設計者はダイサイズを小さくすることで、より多くのトランジスタを同一スペースにまとめました。メインボードに直接個別の CPU を載せるのではなく、マルチコアプロセッサーとして 1 つのプロセッサーパッケージに CPU をまとめ始めたのです。そして、各プロセッサーパッケージからメモリへの同じアクセスを提供するのではなく、Non-Uniform Memory Access (NUMA) 戦略という手段に訴えました。つまり、各パッケージ/ソケットの組み合わせに高速アクセスの専用メモリ領域が 1 つ以上あることになります。各ソケットには他のソケットへの相互接続もあり、遅いアクセスはこれらの他のソケットのメモリにアクセスすることになります。

簡単な NUMA の例として、ソケットが 2 つあるマザーボードを想定してみましょう。各ソケットには、クアッドコアのパッケージが設定されています。つまり、システムには合計 8 つの CPU があり、各ソケットに 4 つずつあることになります。各ソケットには、4 GB の RAM のメモリバンクもあり、システム全体のメモリは 8 GB になります。ここでは、CPU 0-3 はソケット 0 にあり、CPU 4-7 はソケット 1 にあるとします。各ソケットは、NUMA ノードにも対応することにします。

CPU 0 がバンク 0 からメモリにアクセスするには、3 クロックサイクルが必要になります。まず、メモリコントローラーにアドレスを提示するサイクル。次に、メモリの位置へのアクセスを設定するサイクル。そして、その位置への読み取り/書き込みサイクル、です。しかし、CPU 4 が同じ位置からメモリにアクセスするには、6 クロックサイクルが必要になります。別のソケット上にあることで、ソケット 1 のローカルメモリコントローラーとソケット 0 のリモートメモリコントローラーという 2 つのメモリコントローラーを通過する必要があるからです。その位置にあるメモリが争われた場合 (つまり、2 つ以上の CPU が同一位置のメモリに同時にアクセスしようとした場合) は、メモリコントローラーがメモリへのアクセスを仲裁して順番を付けることで、メモリアクセスの時間が長くかかることになります。キャッシュの一貫性を加えると (ローカル CPU キャッシュで、同一メモリ位置には同一データがあることを確認する)、プロセスはさらに複雑になります。

Intel (Xeon) および AMD (Opteron) の最新のハイエンドプロセッサーには、NUMA トポロジーがあります。AMD プロセッサーは HyperTransport™ または HT と呼ばれる相互接続を使用し、Intel は QuickPath Interconnect™ または QPI と呼ばれる相互接続を使用しています。相互接続は、他の相互接続やメモリー、周辺機器への物理的な接続方法で異なりますが、実質的には、ある接続済みデバイスから別の接続済みデバイスへの透過的なアクセスを可能にするスイッチです。このケースでは、「透過的」とは「コストゼロ」のオプションではなく、相互接続に特別なプログラミング API を必要としないことを指します。

システムアーキテクチャーは非常に多様なので、非ローカルメモリにアクセスすることで課せられるパフォーマンスペナルティーを具体的に表すことは現実的ではありません。相互接続上の各**ホップ**が、ホップにつき少なくとも比較的一定のパフォーマンスペナルティーを課している、ということは言えます。現行 CPU から 2 つの相互接続先にあるメモリ位置を参照すると少なくとも **2N + メモリサイクル時間** のユニットをアクセス時間に課すことになります。ここでの N は、ホップあたりのペナルティーになります。

このパフォーマンスペナルティーを考慮すると、パフォーマンス依存型のアプリケーションは NUMA トポロジーシステムのリモートメモリへの定期的アクセスは避けるべきです。このようなアプリケーションは、特定のノードに留まり、そのノードからメモリを割り当てるような設定にするべきです。

これを行うには、アプリケーションは以下の点を知る必要があります。

1. システムの**トポロジー**は何か?
2. アプリケーションは現在、どこで実行中か?
3. 一番近いメモリバンクはどこか?

4.1.2. CPU パフォーマンスのチューニング

このセクションでは、CPU パフォーマンスのチューニング方法とこのプロセスに役立つツールの導入方法を説明します。

NUMA は当初、単一プロセッサーを複数のメモリバンクに接続するために使われました。CPU の製造企業がプロセスの正確性を高め、ダイサイズが縮小されるにつれて、複数の CPU コアが 1 つのパッケージに収まるようになりました。この CPU コアはクラスター化され、それぞれのローカルメモリバンクへのアクセス時間が同一になり、コア間でキャッシュの共有が可能になりました。しかし、相互接続のコア、メモリ、キャッシュ間での「ホップ」には、わずかなパフォーマンスペナルティーが発生しました。

図4.1「NUMA トポロジーにおけるローカルおよびリモートのメモリアクセス」の例のシステムには、2 つの NUMA ノードがあります。各ノードには 4 つの CPU とメモリバンク、メモリコントローラーがあります。CPU は同じノード上のメモリバンクへの直接のアクセスがあります。ノード 1 の矢印にしたがって、以下のようなステップになります。

1. CPU (0-3 のいずれか) がメモリアドレスをローカルのメモリコントローラーに提示します。
2. メモリコントローラーがメモリアドレスへのアクセスを設定します。
3. そのメモリアドレス上で、CPU が読み取り/書き込み操作を実行します。

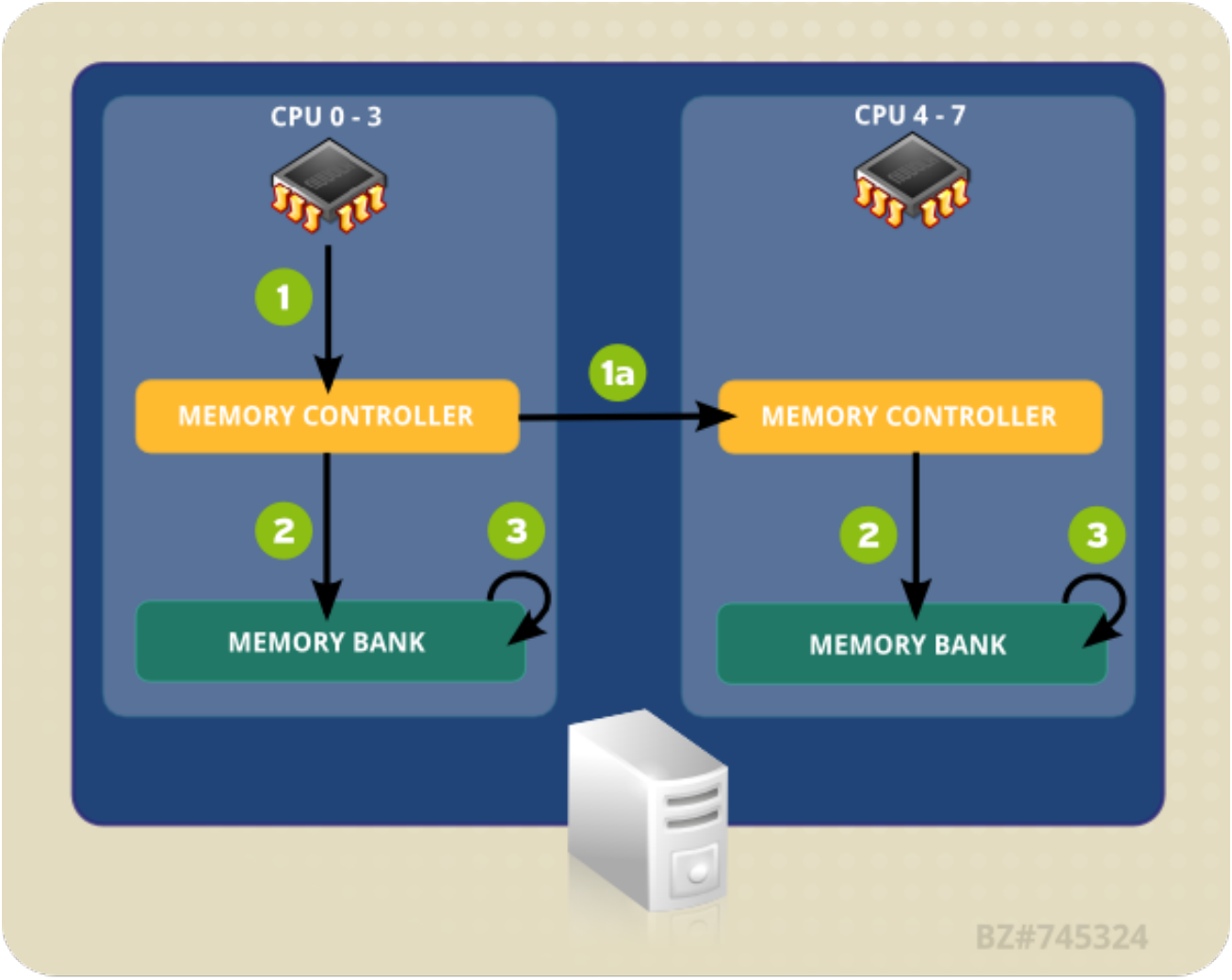


図4.1 NUMA トポロジーにおけるローカルおよびリモートのメモリアクセス

しかし、一方のノード上にある CPU が別の NUMA ノードのメモリバンク上に存在するコードにアクセスする必要がある場合、パスは直接的なものではなくなります。

1. CPU (0-3 のいずれか) がリモートのメモリアドレスをローカルのメモリコントローラーに提示します。
    1. このリモートのメモリアドレスに対する CPU の要請は、そのメモリアドレスがあるノードがローカルである、リモートのメモリコントローラーに渡されます。
2. リモートのメモリコントローラーがリモートのメモリアドレスへのアクセスを設定します。
3. そのリモートメモリアドレス上で、CPU が読み取り/書き込み操作を実行します。

リモートのメモリアドレスにアクセスしようとすると、すべてのアクションが複数のメモリコントローラーを経由しなくてはならないので、アクセスに 2 倍以上の時間がかかる可能性があります。このため、マルチコアシステムでのパフォーマンスの主な関心事は、情報が最短、最速のパスでできるだけ効率的に移動できるようにすることです。

最善の CPU パフォーマンスを発揮するようにアプリケーションを設定するには、以下の点を知る必要があります。

- システムのトポロジー (コンポーネントの接続方法)

- アプリケーションを実行するコア
- 一番近いメモリバンクの位置

Red Hat Enterprise Linux 6 は、これらの情報をみつけ、それに応じてシステムをチューニングする際に役立つ多くのツールと出荷されています。以下のセクションでは、CPU パフォーマンスのチューニングに有用なツールの概要を説明します。

4.1.2.1. taskset を使った CPU アフィニティの設定

taskset は、実行中のプロセスの CPU アフィニティを (プロセス ID で) 取得、設定します。特定の CPU アフィニティを使ってプロセスを開始することも可能で、この場合はこの指定されたプロセスは特定の CPU または CPU セットにバインドされます。しかし、**taskset** はローカルメモリの割り当てを保証するわけではありません。ローカルメモリの割り当てによるパフォーマンスの向上を必要とする場合は、**taskset** ではなく **numactl** を使用することが推奨されます。詳細については「**numactl** を使った NUMA ポリシーのコントロール」 を参照してください。

CPU アフィニティはビットマスクで表されます。順序の一番低いビットは最初の論理 CPU に対応し、順序の一番高いビットは最後の論理 CPU 対応します。これらのマスクは通常 16 進数で提供され、**0x00000001** はプロセッサー 0 を、**0x00000003** はプロセッサー 0 および 1 を表します。

実行中のプロセスの CPU アフィニティを設定するには、以下のコマンドを実行します。*mask* はプロセスを結び付けたいプロセッサーのマスクに置き換え、*pid* はアフィニティを変更するプロセスのプロセス ID に置き換えます。

```
# taskset -p mask pid
```

あるアフィニティでプロセスを開始するには、以下のコマンドを実行します。*mask* はプロセスを結び付けたいプロセッサーのマスクに置き換え、*program* は実行するプログラムおよびプログラムの引数 に置き換えます。

```
# taskset mask -- program
```

プロセスをビットマスクで指定する代わりに、**-c** オプションを使って別個のプロセッサーのコンマ区切りの一覧表やプロセッサーの範囲を提供することもできます。

```
# taskset -c 0,5,7-9 -- myprogram
```

taskset の詳細は man ページ: **man taskset** を参照してください。

4.1.2.2. numactl を使った NUMA ポリシーのコントロール

numactl は、指定されたスケジュールまたはメモリプレースメントポリシーでプロセスを実行します。選択されたポリシーは、そのプロセスとその子プロセスすべてに設定されます。**numactl** は共有メモリセグメントもしくはファイル向けに維持するポリシーも設定でき、プロセスの CPU アフィニティおよびメモリアフィニティも設定できます。 **/sys** ファイルシステムを使ってシステムトポロジーを決定します。

/sys ファイルシステムには、CPU とメモリ、周辺機器が NUMA 相互接続経由でどのように接続されているかについての情報が含まれています。特に、**/sys/devices/system/cpu** ディレクトリーには、システムの CPU がそれぞれどのように接続されているかについての情報が含まれています。**/sys/devices/system/node** ディレクトリーには、システム内の NUMA ノードとノード間の相対距離についての情報が含まれています。

NUMA システムでは、プロセッサーとメモリバンク間の距離が長ければ長いほど、プロセッサーのメモリへのアクセスは遅くなります。このため、パフォーマンス依存型のアプリケーションは、一番近いメモリバンクからのメモリを割り当てるような設定にするべきです。

また、パフォーマンス依存型のアプリケーションは、特定のコア数で実行する設定が推奨されます。マルチスレッドのアプリケーションは、特にこれが当てはまります。最初のレベルのキャッシュは通常少ないため、複数スレッドが1つのコアで実行すると、各スレッドは以前のスレッドがアクセスしたキャッシュ済みのデータを削除する可能性があります。オペレーティングシステムがこれらのスレッド間でマルチタスクを試みる際に、スレッドが互いのキャッシュ済みデータを削除し続けると、実行時間の多くがキャッシュラインの置換に費やされてしまいます。この問題は、**キャッシュスラッシング (cache thrashing)** と呼ばれています。このため、マルチスレッドのアプリケーションを単一コアではなく、ノードに結び付けることが推奨されます。これによって、スレッドは複数レベル (最初、2 番目、最終レベルのキャッシュ) でキャッシュラインの共有ができ、キャッシュを満たす操作の必要性が最小限に抑えられるからです。しかし、スレッドすべてが同一のキャッシュ済みデータにアクセスしている場合は、アプリケーションの単一コアへのバインディングは永続的なものになる場合があります。

numactl を使うとアプリケーションを特定のコアもしくは NUMA ノードにバインドできるようになり、そのアプリケーションにコアもしくはコアのセットに関連したメモリを割り当てられるようになります。numactl には以下の便利なオプションがあります。

--show

現行プロセスの NUMA ポリシー設定を表示します。このパラメーターにはさらなるパラメーターは必要なく、以下のように使用できます。**`numactl --show`**

--hardware

システム上で利用可能なノードのインベントリーを表示します。

--membind

指定のノードからのメモリのみを割り当てます。これを使用中は、このノード上のメモリが不足すると割り当ては失敗します。このパラメーターの使用方法は、**`numactl --membind=`*`nodes`* *`program`** です。ここでは、*nodes* はメモリの割り当て元となるノードのリストで、*program* はそのノードからメモリを割り当てられる必要のあるプログラムのことです。ノード番号は、コンマ区切りの一覧表か範囲、もしくはこの 2 つの組み合わせになります。詳細は、**numactl** の man ページ: **`man numactl`** を参照してください。

--cpunodebind

指定ノードに属する CPU上のコマンド (およびその子プロセス) のみを実行します。このパラメーターの使用方法は、**`numactl --cpunodebind=`*`nodes program`*** です。ここでは、*nodes* は指定プログラム (*program*) をバインドする CPU のあるノード一覧のことです。ノード番号は、コンマ区切りの一覧表か範囲、もしくはこの 2 つの組み合わせになります。詳細は、**numactl** の man ページ: **`man numactl`** を参照してください。

--physcpubind

指定ノード上のコマンド (およびその子プロセス) のみを実行します。このパラメーターの使用方法は、**`numactl --physcpubind=`*`cpu program`*** です。ここでは、*cpu* は **`/proc/cpuinfo`** のプロセッサーフィード内で表示されている物理 CPU番号のコンマ区切り一覧で、*program* はこれらの CPU 上でのみ実行するプログラムです。CPU は、現行 **cpuset** に相対的にも指定できます。詳細は、**numactl** の man ページ: **`man numactl`** を参照してください。

--localalloc

現行ノード上に常に割り当てられるメモリを指定します。

--preferred

可能な場合、メモリは指定のノードに割り当てられます。指定ノードへのメモリ割り当てができない場合、別のノードにフォールバックします。このオプションでは以下のように1つのノード番号のみを使います。**numactl --preferred=*node***。詳細は numactl の man ページ: **man numactl** を参照してください。

numactl にパッケージに含まれている **libnuma** ライブラリは、カーネルがサポートしている NUMA ポリシーへの簡潔なプログラミングインターフェースを提供します。これは、より詳細なチューニングの際に **numactl** ユーティリティーよりも便利なものです。詳細情報は man ページ: **man numa(7)** を参照してください。

4.1.3. ハードウェアパフォーマンスポリシー (x86_energy_perf_policy)

cpupowerutils パッケージには、**x86_energy_perf_policy** が含まれます。このツールを使うと、管理者はエネルギー効率に対してパフォーマンスの相対的な重要性を定義することができます。この情報はその後、この機能に対応しているプロセッサーがパフォーマンスとエネルギー効率を交換するオプションを選択する際に、そのプロセッサーに影響を及ぼすために使用できます。プロセッサーのサポートは **CPUID.06H.ECX.bit3** で示されます。

x86_energy_perf_policy は root 権限を必要とし、デフォルトですべての CPU で実行できます。

現行ポリシーを表示するには、以下のコマンドを実行します。

```
# x86_energy_perf_policy -r
```

新たなポリシーを設定するには、以下のコマンドを実行します。

```
# x86_energy_perf_policy profile_name
```

profile_name を以下のいずれかのプロファイルで置き換えます。

performance

プロセッサーは省エネルギーのためにパフォーマンスを犠牲にすることを嫌がります。これがデフォルト値です。

normal

大幅な省エネルギーの可能性がある場合、プロセッサーはマイナーなパフォーマンス低下を許可します。これは、ほとんどのデスクトップおよびサーバーで妥当な設定です。

powersave

プロセッサーは、エネルギー効率を最大化するために大幅なパフォーマンス低下の可能性を受け入れます。

このツールについての詳細情報は、man ページ: **man x86_energy_perf_policy** を参照してください。

4.1.4. turbostat

turbostat ツールは cpupowerutils パッケージの一部です。これは、Intel 64 プロセッサー上のプロセッサートポロジー、周波数、アイドル電源状態の統計数字、温度、および電力使用量をレポートします。

Turbostat を使うと管理者は必要以上に電力を消費しているサーバーや本来スリープ状態に入っているはずなのに入っていないサーバー、またはプラットホームがすぐに利用可能な場合に仮想化を検討すべき (つまり、物理サーバーを使用停止にできる) アイドル状態にあるサーバーを特定することができます。また、システム管理の割り込み (SMI) の比率や SMI を不必要にプロンプトしている、遅延を区別するアプリケーションの特定に役立ちます。Turbostat は **powertop** ユーティリティーと併せて使用することでプロセッサーがスリープ状態に入ることを妨げている可能性のあるサービスを特定することもできます。

Turbostat を実行するには **root** 権限が必要になります。また、不変タイムスタンプカウンターをサポートしているプロセッサーと、APERF および MPERF のモデル固有レジスタが必要になります。

デフォルトでは、**turbostat** はシステム全体のカウンター結果の概要と、それに続いて以下の見出しの下にカウンター結果を 5 秒ごとにプリントします。

pkg

プロセッサーのパッケージ番号。

core

プロセッサーのコア番号。

CPU

Linux CPU (論理プロセッサー) 番号。

%c0

CPU リタイヤ状態の指示の間隔のパーセント。

GHz

CPU が c0 状態にあった間の平均クロック速度。この数値が TSC の値よりも高い場合は、CPU はターボモードになります。

TSC

間隔全体にわたる平均クロック速度。この数値が TSC の値よりも低い場合は、CPU はターボモードになります。

%c1、%c3、および %c6

プロセッサーが c1、c3、または c6 の各状態だった間隔のパーセント。

%pc3 または %pc6

プロセッサーが pc3 または pc6 の各状態だった間隔のパーセント。

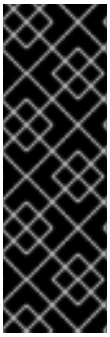
-i オプションを使ってカウンター結果の間の異なる期間を指定します。たとえば、**turbostat -i 10** を実行すると、10 秒ごとに結果がプリントされます。

注記

今後発売される Intel プロセッサーは、新たな C 状態を追加する可能性があります。Red Hat Enterprise Linux 6.5 では、turbostat は c7、c8、c9、および c10 の状態をサポートしています。

turbostat についての詳細情報は、man ページ: **man turbostat** を参照してください。

4.1.5. numastat



重要

以前の **numastat** ツールは、Andi Kleen 氏が記述した Perl スクリプトでした。Red Hat Enterprise Linux 6.4 向けには、これが大幅に書き換えられています。

デフォルトのコマンド (**numastat** でオプションやパラメーターなし) はツールのこれまでのバージョンと厳密な互換性を維持していますが、オプションやパラメーターが加えられるとこのコマンドの出力コンテンツとフォーマットが大幅に変わることに注意してください。

numastat は、プロセスとオペレーティングシステムのメモリ統計 (割り当てのヒットとミスなど) を NUMA ノードあたりで表示します。デフォルトでは、**numastat** を実行すると、各ノードの以下のイベントカテゴリーごとに占有されているメモリのページ数を表示します。

numa_miss および **numa_foreign** の値が低いと、CPU パフォーマンスが優れていることを示します。

numastat の更新バージョンも、プロセスメモリがシステム全体に拡散しているかもしくは**numactl** を使用している特定ノード上に集中しているかを表示します。

メモリが割り当てられているノードと同一ノード上でプロセススレッドが実行中かどうかを検証するために、CPU あたりの **top** 出力を使って **numastat** 出力を相互参照します。

デフォルトの追跡カテゴリー

numa_hit

当該ノードに割り当てを試みたもので成功した数。

numa_miss

別のノードに割り当てを試みたもので、当初の意図されたノードがメモリ不足だったために当該ノードに割り当てられた数。各 **numa_miss** イベントには、対応する **numa_foreign** イベントが別のノード上にあります。

numa_foreign

当初は当該ノードへの割り当てを意図したもので、別のノードに割り当てられた数。**numa_foreign** イベントには対応する **numa_miss** イベントが別のノード上にあります。

interleave_hit

当該ノードに試みたインターリーブポリシーの割り当てで成功した数。

local_node

当該ノード上のプロセスが当該ノード上へのメモリ割り当てに成功した回数。

other_node

別のノード上のプロセスが当該ノード上にメモリを割り当てた回数。

以下のオプションのいずれかを適用すると、メモリの表示単位がメガバイトに変更され (四捨五入で小数点第 2 位まで)、以下のように他の特定の **numastat** 動作に変更を加えます。

-c

表示情報の表を横方向に縮小します。これは、NUMA ノード数が多いシステムでは有用ですが、コラムの幅とコラム間の間隔はあまり予測可能ではありません。このオプションが使用されると、メモリ量は一番近いメガバイトに切り上げ/下げられます。

-m

ノードあたりでのシステム全体のメモリ使用量を表示します。**/proc/meminfo** にある情報に類似したものです。

-n

オリジナルの **numastat** コマンド (numa_hit、numa_miss、numa_foreign、interleave_hit、local_node、other_node) と同一情報を表示しますが、測定単位にメガバイトを使用した更新フォーマットが使われます。

-p ***pattern***

指定されたパターンのノードごとのメモリー情報を表示します。*pattern* の値が数字の場合は、**numastat** は数値プロセス識別子とみなされます。それ以外の場合は、**numastat** は指定されたパターンのプロセスコマンドラインを検索します。

-p オプションの値の後に入力されるコマンドライン引数は、フィルターにかける追加のパターンとみなされます。追加のパターンは、フィルターを絞り込むのではなく拡張します。

-s

表示データを降順に並び替えるので、(**total** コラムの) メモリ消費量の多いものが最初に来ます。

オプションで *node* を指定すると、表は *node* コラムにしたがって並び替えられます。このオプションの使用時には、以下のように *node* 値は **-s** オプションのすぐあとに来る必要があります。

```
numastat -s2
```

このオプションと値の間に空白スペースを入れないでください。

-v

詳細情報を表示します。つまり、複数プロセスのプロセス情報が各プロセスの詳細情報を表示します。

-V

numastat のバージョン情報を表示します。

-z

情報情報から値が 0 の行と列のみを省略します。表示目的で 0 に切り下げられている 0 に近い値は、表示出力から省略されません。

4.1.6. NUMA アフィニティ管理デーモン (numad)

numad は自動の NUMA 管理デーモンです。システム内の NUMA トポロジーとリソース使用量を監視して、動的に NUMA リソース割り当ておよび管理 (つまりシステムパフォーマンス) を改善します。

システムのワークロードによって、**numad** はベンチマークパフォーマンスを最大 50 % 改善します。このパフォーマンス改善を達成するため、**numad** は定期的に **/proc** ファイルシステムからの情報にアクセスし、ノードごとに利用可能なシステムリソースを監視します。その後はデーモンが、十分な配置メモリと最適な NUMA パフォーマンスのための CPU リソースがある NUMA ノード上に重要なプロセスを配置します。プロセス管理の最新のしきい値は 1 つの CPU の少なくとも 50 % と 300 MB のメモリです。**numad** はリソース使用量のレベルの維持を図り、割り当ての再バランス化が必要な場合は NUMA ノード間でプロセスを移動します。

numad は、様々なジョブ管理システムが質問できる配置前のアドバイスサービスも提供しており、プロセスにおける CPU とメモリリソースの初期バインディングの手助けをします。この配置前アドバイスサービスは、**numad** がシステム上でデーモンとして実行中かどうかにかかわらず、利用可能です。**-w** オプションで配置前アドバイスを使用する方法については、man ページ:**man numad** を参照してください。

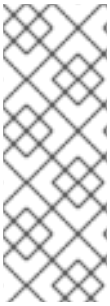
4.1.6.1. numad の利点

numad は主に、多大な量のリソースを消費する、長期間実行のシステムに利益をもたらします。これらのプロセスがトータルシステムリソースのサブセットに含まれる場合は、特にそうです。

numad は、複数の NUMA ノード分のリソースを消費するアプリケーションにも有用です。しかし、**numad** のもたらす利点は、システム上で消費されるリソースの割合が高まるにつれて低下します。

プロセスの実行時間がほんの数分であったり、多くのリソースが消費されない場合は、**numad** はパフォーマンスを改善しない可能性が高くなります。大型のメモリ内データベースなど、継続的に予想できないメモリアクセスのパターンがあるシステムも、**numad** の使用で恩恵を得る可能性は低くなります。

4.1.6.2. オペレーションモード



注記

KSM を使用する場合は、**/sys/kernel/mm/ksm/merge_nodes** の調整可能な値を **0** に変更して NUMA にまたがるページのマージを回避します。カーネルメモリーが計算した統計は、大量のクロスノードのマージ後にはそれぞれの間で相反する場合があります。そのため、KSM デーモンが大量のメモリーをマージすると、**numad** は混乱する可能性があります。システムに未使用のメモリーが大量にあると、KSM デーモンをオフにして無効にすることでパフォーマンスが高まる場合があります。

numad には 2 つの使用方法があります。

- サービスとして使用
- 実行可能ファイルとして使用

4.1.6.2.1. numad をサービスとして使用

numad サービスの実行中に、ワークロードに基づいてシステムを動的にチューニングしようとします。

サービスを開始するには、以下のコマンドを実行します。

```
# service numad start
```

リブート後もサービスを維持するには、以下のコマンドを実行します。

```
# chkconfig numad on
```

4.1.6.2.2. numad を実行可能ファイルとして使用

numad を実行可能ファイルとして使用するには、単に以下のコマンドを実行します。

```
# numad
```

numad は停止されるまで実行し続けます。実行中は、アクティビティが**/var/log/numad.log** にログ記録されます。

numad 管理を特定プロセスに限定するには、以下のオプションで開始します。

```
# numad -S 0 -p pid
```

-p pid

指定の *pid* を明示的な対象一覧に加えます。指定されたプロセスは、**numad** プロセスの重要度しきい値に達するまで管理されません。

-S mode

-S パラメーターはプロセススキャニングのタイプを指定します。例のように**0** に設定すると **numad** 管理を明示的に付加プロセスに限定します。

numad を停止するには、以下のコマンドを実行します。

```
# numad -i 0
```

numad を停止しても NUMA アフィニティの改善のためになされた変更は削除されません。システムの使用方法が大幅に変わる場合は、**numad** を再度実行することでアフィニティが調整され、新たな条件の下でパフォーマンスが改善されます。

利用可能な **numad** オプションについては、**numad** man ページ: **man numad** を参照してください。

4.2. CPU のスケジューリング

スケジューラー には、システム内の CPU を無駄なく稼働させる役割があります。Linux スケジューラーは多くの**スケジューリングポリシー** を実行し、これらのポリシーが特定の CPU コア上でスレッドがいつ、どれくらい長く実行するかを決定します。

スケジューリングポリシーは 2 つのカテゴリーに分けられます。

1. リアルタイムポリシー
   - SCHED_FIFO
   - SCHED_RR

2. 通常のポリシー
    - SCHED_OTHER
    - SCHED_BATCH
    - SCHED_IDLE

4.2.1. リアルタイムのスケジューリングポリシー

最初にリアルタイムスレッドがスケジュールされ、すべてのリアルタイムスレッドの後で通常のスレッドがスケジュールされます。

リアルタイムポリシーは、割り込みなしで完了する必要のある、時間的にクリティカルなタスクに使用されます。

SCHED_FIFO

このポリシーは **static priority scheduling** (静的優先順位スケジューリング) とも呼ばれます。これは、各スレッドの固定優先順位 (1 から 99 の間で) を定義するためです。このスケジューラーは、SCHED_FIFO スレッド一覧を優先順位でスキャンして、実行準備ができているもので最も優先順位が高いスレッドをスケジュールします。このスレッドは、ブロックまたは終了するか、実行準備ができたより優先順位の高いスレッドに取って代わられるまで実行します。

優先順位が一番低いリアルタイムスレッドでも、非リアルタイムポリシーのスレッドよりも先にスケジュールされます。リアルタイムスレッドが 1 つだけの場合は、**SCHED_FIFO** の優先順位の値は無関係になります。

SCHED_RR

SCHED_FIFO ポリシーのラウンドロビン版です。**SCHED_RR** スレッドも固定優先順位 (1 から 99 の間で) が与えられます。しかし、優先順位が同じスレッドは、特定のクォンタム、または時間枠内でラウンドロビンでスケジュールされます。**sched_rr_get_interval(2)** システムコールはこの時間枠の値を返しますが、時間枠の長さをユーザーが設定することはできません。このポリシーは、同じ優先順位で複数のスレッドを実行する必要がある場合に有用です。

リアルタイムスケジューリングポリシーの定義済みセマンティクスについての詳細情報は、システムインターフェース – リアルタイムにある 『IEEE 1003.1 POSIX standard』 を参照してください。http://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_08.html で見ることができます。

スレッド優先順位付けの定義におけるベストプラクティスは、初めは順位を低く設定し、正当な待ち時間が特定された時にだけ優先順位を高くするという方法です。リアルタイムスレッドは、通常のスレッドのように時間枠があるものではありません。**SCHED_FIFO** スレッドは、ブロックまたは終了するか、より優先順位の高いスレッドに取って代わられるまで実行します。このため、優先順位を 99 に設定することは推奨されません。これだと、プロセスをマイグレーションやウォッチドッグスレッドと同じ優先順位とすることになってしまいます。スレッドが演算ループに入ってしまったためにこれらのスレッドがブロックされると、実行不可能となってしまいます。この状況では、ユニプロセッサーシステムは最終的にはロックアップしてしまいます。

Linux カーネルでは、**SCHED_FIFO** ポリシーには帯域幅制限メカニズムが含まれます。これによりリアルタイムのアプリケーションプログラマーを、CPU を独占する可能性のあるリアルタイムタスクから保護します。このメカニズムは、以下の **/proc** ファイルシステムパラメーターで調整できます。

/proc/sys/kernel/sched_rt_period_us

CPU 帯域幅の 100 % と考えられる時間をマイクロ秒で定義します (「us」はプレーンテキストで「µs」に一番近いもの)。デフォルト値は 1000000µs もしくは 1 秒です。

/proc/sys/kernel/sched_rt_runtime_us

リアルタイムスレッドの実行に当てられる時間をマイクロ秒で定義します (「us」はプレーンテキストで「µs」に一番近いもの)。デフォルト値は 950000µs もしくは 0.95 秒です。

4.2.2. 通常のスケジューリングポリシー

通常のスケジューリングポリシーには、**SCHED_OTHER**、**SCHED_BATCH**、**SCHED_IDLE** の 3 つがあります。しかし、**SCHED_BATCH** および **SCHED_IDLE** ポリシーは優先順位が非常に低いものを対象としており、このためパフォーマンスチューニングガイドにおける関心度は低くなっています。

SCHED_OTHER または SCHED_NORMAL

デフォルトのスケジューリングポリシーです。このポリシーは、Completely Fair Scheduler (CFS) を使ってこのポリシーを使用するすべてのスレッドに対して公平なアクセス期間を提供します。CFS は、部分的に各プロセススレッドの ***niceness*** 値に基づいて動的にな優先順位リストを確立します。(このパラメーターと **/proc** ファイルシステムの詳細に関しては、『導入ガイド』 を参照してください。) これによりユーザーは、プロセスの優先順位に関して間接的なレベルのコントロールを手にしますが、動的な優先順位リストを直接変更できるのは CFS を使った場合のみです。

4.2.3. ポリシーの選択

アプリケーションのスレッドに正しいスケジューリングポリシーを選択することは、常に簡単な作業とは限りません。一般的に、リアルタイムポリシーは時間的にクリティカルまたは重要なタスクで素早いスケジュールを必要とし、長期にわたって実行されないタスクに使われるべきものです。通常のポリシーは一般的にリアルタイムポリシーよりも優れたデータスループットの結果をもたらします。これは、スケジューラーがより効率的にスレッドを実行できるようにするためです (つまり、先取りのための再スケジュールがそれほど頻繁に必要ないということです)。

大量のスレッドを管理していて、主にデータスループット (1 秒あたりのネットワークパケットやディスクへの書き込みなど) に関わっている場合は、**SCHED_OTHER** を使ってシステムが CPU 使用率を管理するようにします。

イベントの反応時間 (待ち時間) が気になる場合は、**SCHED_FIFO** を使います。スレッド数が少ない場合は、CPU ソケットを孤立させ、スレッドをソケットのコアに移動することでそのコア上で他のスレッドと時間を競合することがないようにする方法を考慮してください。

4.3. 割り込みおよび IRQ チューニング

割り込み要請 (IRQ) は、ハードウェアレベルで送信されるサービスの要請です。割り込みは情報パケット (Message Signaled Interrupt または MSI) として専用ハードウェアラインまたはハードウェアバスで送信できます。

割り込みが可能になると、IRQ の受信で割り込みコンテキストへの切り替えが促されます。カーネル割り込み発送コードが IRQ 番号と関連する登録済みの割り込みサービスルーチン (ISR) のリストを検索し、各 ISR を順番に呼び出します。ISR は割り込みを承認し、同一の IRQ からの重複割り込みを無視します。その後、保留ハンドラーをキューに入れて割り込み処理を完了し、ISR が今後の割り込みを無視しないようにします。

/proc/interrupts ファイルは、I/O デバイスあたりの CPU あたりの割り込み数を一覧表示します。表示されるのは、IRQ 番号、各 CPU コアが処理するその割り込みの番号、割り込みのタイプ、その割

り込みを受信するために登録されているドライバーのコンマ区切り一覧、です。(詳細に関しては、proc(5) の man ページ: **man 5 proc** を参照してください。)

IRQ には関連する「アフィニティ」プロパティー、***smp_affinity***、があり、これはその IRQ の ISR の実行を許可する CPU コアを定義します。このプロパティーは、割り込みアフィニティとアプリケーションのスレッドアフィニティの両方を 1 つ以上の特定の CPU コアに割り当てることでアプリケーションのパフォーマンスを改善します。これにより、指定割り込みとアプリケーションスレッド間のキャッシュライン共有が可能になります。

特定の IRQ 番号の割り込みアフィニティ値は、関連する **/proc/irq/*IRQ_NUMBER*/smp_affinity** ファイルに保存され、root ユーザーはこれを閲覧、修正できます。このファイルに保存された値は 16 進法のビットマスクで、システムの全 CPU コアを表します。

以下の例では、4 つの CPU コアがあるサーバー上のイーサネットドライバーに割り込みアフィニティを設定するために、まずイーサネットドライバーに関連する IRQ 番号を決定します。

```
# grep eth0 /proc/interrupts
32:   0     140      45       850264      PCI-MSI-edge      eth0
```

IRQ 番号を使って適切な ***smp_affinity*** ファイルの位置を特定します。

```
# cat /proc/irq/32/smp_affinity
f
```

smp_affinity のデフォルト値は **f** で、これはシステム内のどの CPU でも IRQ が実行できることを意味します。以下のようにこの値を **1** に設定すると、CPU 0 のみがこの割り込みを実行できることになります。

```
# echo 1 >/proc/irq/32/smp_affinity
# cat /proc/irq/32/smp_affinity
1
```

コンマを使って ***smp_affinity*** 値を別々の 32 ビットグループに設定することができます。32 コアを超えるシステムでは、これが必要になります。例えば、以下の例では IRQ 40 が 64 コアシステムの全コア上で実行されることを示しています。

```
# cat /proc/irq/40/smp_affinity
ffffffff,ffffffff
```

IRQ 40 を 64 コアシステムの上位 32 コアのみで実行するには、以下のようにします。

```
# echo 0xffffffff,00000000 > /proc/irq/40/smp_affinity
# cat /proc/irq/40/smp_affinity
ffffffff,00000000
```

注記

割り込みステアリングをサポートするシステムでは、IRQ の***smp_affinity*** を修正することでハードウェアを設定し、割り込みを特定の CPU で実行する決定がカーネルからの干渉なしにハードウェアレベルでできるようになります。

4.4. CPU 周波数ガバナー

CPU の周波数は、パフォーマンスと電力消費量の両方に影響を及ぼします。**cpufreq** ガバナーは常に、より高いまたは低い周波数への変更を促す動作についてのルールセットに基づいて、CPU の周波数を決定します。

Red Hat では、**cpufreq_ondemand** ガバナーがシステムに負荷がある時に高い CPU 周波数でより高いパフォーマンスを、またシステムの負荷が高くないときには低い CPU 周波数で省電力を提供することから、ほとんどの状況でこのガバナーを推奨しています。

省電力を犠牲にしてパフォーマンスを最大化する場合には、**cpufreq_performance** ガバナーを使用します。このガバナーは、できるだけ高い CPU 周波数を使用してタスクができるだけ迅速に実行されるようにします。このガバナーは、スリープやアイドルなどの省電力メカニズムを使用しないので、データセンターや類似の大型導入には推奨されません。

手順4.1 ガバナーの有効化および設定

1. cpupowerutils がインストールされていることを確認します。

```
# yum install cpupowerutils
```

2. 使用したいドライバーが利用可能であることを確認します。

```
# cpupower frequency-info --governors
```

3. ドライバーが利用可能でない場合は、**modprobe** コマンドを使用してそのドライバーをシステムに追加します。たとえば、**ondemand** ガバナーを追加するには、以下を実行します。

```
# modprobe cpufreq_ondemand
```

4. **cpupower** コマンドラインツールを使ってガバナーを一時的に設定します。たとえば、**ondemand** ガバナーを設定するには、以下を実行します。

```
# cpupower frequency-set --governor ondemand
```

tuned-adm ツールに同梱されているプロファイルも CPU 周波数ガバナーを活用します。詳細は、「Tuned および ktune」 を参照してください。

4.5. RED HAT ENTERPRISE LINUX 6 での NUMA 機能強化

Red Hat Enterprise Linux 6 には、今日の高度に拡張可能なハードウェアの潜在能力を活用するための多くの機能強化が含まれています。このセクションでは、Red Hat Enterprise Linux 6 が提供する NUMA 関連のパフォーマンス機能強化のなかで、最も重要なハイレベルの概要を説明します。

4.5.1. ベアメタルおよびスケーラビリティの最適化

4.5.1.1. トポロジー認識における機能強化

以下の機能強化により Red Hat Enterprise Linux は低レベルのハードウェアおよびアーキテクチャー詳細の検出が可能になり、システム上の処理を自動で最適化する機能が改善されます。

トポロジー検出の強化

この機能により、オペレーティングシステムは低レベルのハードウェア詳細 (論理 CPU やハイパースレッド、コア、ソケット、NUMA ノード、ノード間のアクセス時間など) を起動時に検出でき、システム上の処理を最適化できます。

Completely Fair Scheduler (CFS)

この新たなスケジューリングモードは、ランタイムが有効なプロセス間で同等に共有されるようにします。これをトポロジー検出と組み合わせることで、プロセスが同一ソケット内の CPU にスケジュールされ、不経済なリモートメモリアクセスを避けることができます。また、キャンセルコンテンツが可能な場所に確実に保存されるようにします。

`malloc`

`malloc` は、プロセスに割り当てられるメモリのリージョンがプロセスが実行されるコアに物理的にできるだけ近くなるように最適化されました。これにより、メモリのアクセス速度が高まっています。

skbuff I/O バッファ割り当て

`malloc` と同様に、デバイス割り込みなどの I/O 操作を処理する CPU に物理的に一番近いメモリを使用するように最適化されています。

デバイス割り込みアフィニティ

デバイスドライバーが記録した、どの CPU がどの割り込みを処理するかという情報を使って、同一の物理ソケット内の CPU に割り込み処理を限定できます。これにより、キャンセルアフィニティが保存され、ソケット間の高ボリュームな通信が制限されます。

4.5.1.2. マルチプロセッサー同期における機能強化

マルチプロセッサー間のタスクの調整は、並行して行われているプロセスがデータの整合性を損なわないようにするために、頻繁でかつ時間のかかる操作を必要とします。Red Hat Enterprise Linux には、このエリアでのパフォーマンスを改善する以下の機能強化が含まれています。

Read-Copy-Update (RCU) ロック

通常、ロックの 90% は読み取り専用目的で取得されます。RCU ロックは、アクセスされているデータが修正されていない場合、排他アクセスロックを取得する必要性を取り除きます。このロッキングモードは、ページキャッシュメモリの割り当てに使用されます。ロッキングが使用されるのは、割り当てもしくは解放操作のみです。

CPU 別およびソケット別のアルゴリズム

多くのアルゴリズムは、同一ソケット上の 協同 CPU のロック調整を行うために更新され、より粒度の細かいロッキングを可能にしています。多くのグローバル spinlock はソケットごとのロッキング方法に置換されています。また、アップデートされたメモリアロケーターゾーンと関連メモリのページ一覧により、操作の割り当てもしくは解放の実行時に、メモリ割り当て論理がより効率的にメモリマッピングデータ構造のサブセットを行き来できます。

4.5.2. 仮想化の最適化

KVM はカーネル機能を活用しているため、KVM ベースの仮想化ゲストはすべてのベアメタル最適化から即座に恩恵を受けます。Red Hat Enterprise Linux には、仮想化ゲストがベアメタルシステムのパフォーマンスレベルに近づくことを可能にする数多くの機能強化も含まれています。これらの機能強化

は、ストレージおよびネットワークアクセスの I/O パスにフォーカスしており、データベースやファイルサービスなどの集中的なワークロードが仮想化導入を活用できるようにしています。仮想化システムのパフォーマンスを改善する NUMA 固有の機能強化には、以下のものがあります。

CPU ピン設定

仮想ゲストを特定のソケットで実行するよう固定します。これは、ローカルキャッシュの使用を最適化し、不経済なソケット間の通信とリモートメモリのアクセスの必要性を排除するためです。

transparent hugepages (THP)

THP を有効にすると、システムは連続する大量メモリに対して自動的に NUMA 認識メモリの割り当てを実行します。これにより、ロック競合と必要なトランスレーションルックアサイドバッファ (TLB) のメモリ管理操作回数が減少するとともに、仮想化ゲストのパフォーマンスが最大 20 % 向上します。

カーネルベースの I/O 実装

仮想ゲストの I/O サブシステムがカーネルに実装されており、大量のコンテキストスイッチングと同期、通信オーバーヘッドが回避されることで、ノード間の通信およびメモリアクセスの時間が大幅に削減されます。

第5章 メモリ

本章では、Red Hat Enterprise Linux で利用可能なメモリ管理機能の概要とこれらの機能を使ってシステムでメモリ使用率を最適化する方法を説明します。

5.1. HUGE トランスレーションルックアサイドバッファ (HUGETLB)

物理メモリアドレスは、メモリ管理の一環として仮想メモリアドレスに変換されます。物理アドレスから仮想アドレスへのマッピング関係は、ページテーブルと呼ばれるデータ構造に保存されます。すべてのアドレスマッピングでページテーブルを読み取るのは時間がかかり、リソースを費やすことになるので、最近使用されたアドレスにはキャッシュがあります。このキャッシュは、トランスレーションルックアサイドバッファ (TLB) と呼ばれます。

しかし、TLB にキャッシュできるアドレスマッピングには限りがあります。要求されたアドレスマッピングが TLB にない場合、物理から仮想へのアドレスマッピングを決定するために依然としてページテーブルを読み取る必要があります。これは、「TLB ミス」と呼ばれます。メモリ要件の大きいアプリケーションは最低限のメモリ要件のアプリケーションよりも TLB ミスの影響を受ける可能性が大きくなります。これは、メモリ要件と TLB におけるキャッシュアドレスマッピングに使用されるページサイズの関係が理由です。各ミスにはページテーブルの読み取りが関わるため、可能な限りこれらのミスを避けることが重要です。

Huge トランスレーションルックアサイドバッファ (HugeTLB) は、非常に大きなセグメントでのメモリ管理を可能にすることで、一度にキャッシュ可能なアドレスマッピングを増やすことができます。こうすることで TLB ミスの確率が下がり、その結果、メモリ要件の大きいアプリケーションのパフォーマンスが改善します。

HugeTLB の設定に関する情報は、カーネルドキュメンテーション: **/usr/share/doc/kernel-doc-*version*/Documentation/vm/hugetlbpage.txt** を参照してください。

5.2. HUGE PAGES および TRANSPARENT HUGE PAGES

メモリは *pages* とよばれるブロックで管理されています。1 page は 4096 バイトです。メモリ 1MB は 256 pages と同等で、1GB は 256,000 pages と同等、ということになります。CPU にはビルトインの**メモリ管理ユニット** があり、これらのページの一覧が含まれています。また各ページは**ページテーブルエントリ** で参照されています。

システムが大量のメモリを管理できるようにするには、以下の 2 つの方法があります。

- ハードウェアのメモリ管理ユニットでページテーブルエントリの数を増やす。
- ページサイズを拡大する。

現在のプロセッサーでは、ハードウェアのメモリ管理ユニットは数百から数千のページテーブルエントリのみをサポートするので、最初の方法は費用がかさみます。さらに、数千ページ (メガバイトのメモリ) でうまく機能するハードウェアとメモリ管理のアルゴリズムは、数百万 (さらには数十億) のページではうまく実行できない可能性があります。この場合、パフォーマンス問題につながります。メモリ管理ユニットがサポートする以上のメモリページをアプリケーションが使う必要がある場合、システムはより遅い、ソフトウェアベースのメモリ管理にフォールバックするので、システム全体が遅くなってしまいます。

Red Hat Enterprise Linux 6 では、*huge pages* を使って 2 番目の方法を実行しています。

簡単に言うと、huge pages は 2MB や 1GB サイズのメモリのブロックです。2MB のページが使用するページテーブルは、複数のギガバイトのメモリ管理にふさわしく、1GB のページテーブルはテラバイトまでのメモリ拡張に適するものです。

Huge pages は起動時に割り当てられる必要があります。また、手動での管理は難しく、効果的に使用するにはコードの大幅変更が必要となる場合が多くあります。このため、Red Hat Enterprise Linux 6 は *transparent huge pages* (THP) も実装しています。THP は、huge pages の作成、管理、使用の多くを自動化する抽象化レイヤーです。

THP は、huge pages の使用における複雑性の多くをシステム管理者や開発者から取り除きます。THP の目標はパフォーマンス改善なので、(コミュニティーと Red Hat の両方の) 開発者は、幅広いシステムや設定、アプリケーション、ワークロードで THP をテスト、最適化しています。これにより、デフォルト設定の THP はほとんどのシステム設定のパフォーマンスを改善できます。ただし、THP はデータベースのワークロードには推奨されません。

THP が現在マッピング可能なのは、ヒープやスタック領域などの匿名メモリリージョンのみです。

5.3. VALGRIND を使ったメモリ使用量のプロファイリング

Valgrind は、ユーザースペースバイナリーへのインストルメンテーションを提供するフレームワークです。プログラムパフォーマンスのプロファイリングや分析に使用可能な数多くのツールと出荷されます。このセクションで紹介されるツールは、初期化されていないメモリの使用やメモリの不適切な割り当てや解放などのメモリエラーの検出に役立つ分析を提供します。これらはすべて valgrind に含まれており、以下のコマンドで実行可能です。

```
valgrind --tool=toolname program
```

toolname を使用するツール名で置き換え (メモリのプロファイリングには、**memcheck**、**massif**、**cachegrind** のいずれか)、*program* を Valgrind でプロファイリングするプログラム名に置き換えます。Valgrind のインストルメンテーションを使うと、プログラムの実行が通常よりも遅くなることに留意してください。

Valgrind の機能の概要は 「Valgrind」 で説明されています。Eclipse 用に利用可能なプラグインについての情報を含む詳細は、『開発者ガイド』 を参照してください。http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/から入手できます。付随資料は valgrind パッケージのインストール時に **man valgrind** コマンドで表示するか、以下の場所で見つけられます。

- **/usr/share/doc/valgrind-*version*/valgrind_manual.pdf**
- **/usr/share/doc/valgrind-*version*/html/index.html**

5.3.1. Memcheck を使ったメモリ使用量のプロファイリング

Memcheck はデフォルトの Valgrind ツールで、**valgrind *program*** で **--tool=memcheck** 指定せずに実行できます。本来発生すべきでないメモリアクセスや未定義もしくは初期化されていない値の使用、誤って解放されたヒープメモリ、重複するポインター、メモリリークなどの検出や診断が難しい多くのメモリエラーを検出し、それらについてレポートします。Memcheck を使うと、プログラムは通常の実行時に比べて、10-30 倍の時間がかかります。

Memcheck は、検出する問題の種類により、特定のエラーを返します。これらのエラーは、**/usr/share/doc/valgrind-*version*/valgrind_manual.pdf** に含まれている Valgrind の資料に詳細が説明されています。

Memcheck が実行できるのは、これらのエラーのレポートだけで、エラー発生を防止できるわけではないことに注意してください。通常はセグメント化障害につながる方法でプログラムがメモリにアクセスする場合、このセグメント化障害は発生し続けますが、Memcheck はこの障害のすぐ前にエラーメッセージをログ記録します。

Memcheck はチェックプロセスにフォーカスできるコマンドラインオプションを提供します。以下に例を挙げます。

`--leak-check`

これを有効にすると、Memcheck はクライアントプログラムの完了時にメモリリークを検索します。デフォルト値は **`summary`** で、これは発見されたリーク数を出力します。使用可能な他の値は **`yes`** と **`full`** で、両方とも個別リークの詳細を提供します。また、**no** を使用すると、メモリリークチェックが無効になります。

`--undef-value-errors`

これを有効にすると (**yes** に設定)、Memcheck は未定義の値が使用された際にレポートします。無効にすると (**no** に設定)、未定義の値エラーはレポートされません。デフォルトでは有効になっており、無効にすると Memcheck スピードが若干高まります。

`--ignore-ranges`

Memcheck がアドレス指定能力をチェックする際に無視する 1 つ以上の範囲をユーザーが指定できるようになります。複数の範囲は以下のようにコンマで区切ります。**`--ignore-ranges=0xPP-0xQQ,0xRR-0xSS`**

オプションの全一覧は、**`/usr/share/doc/valgrind-*version*/valgrind_manual.pdf`** にある資料を参照してください。

5.3.2. Cachegrind を使ったキャッシュ使用量のプロファイリング

Cachegrind は、プログラムとマシンのキャッシュ階層、また (オプションで) 分岐予測との対話をシミュレートします。シミュレートされた第一レベルの指示とデータキャッシュの使用を追跡し、このレベルのキャッシュで不十分なコード対話を検出します。また、最後のレベルのキャッシュ (2 番目でも 3 番目でも) でメインメモリへのアクセスを追跡します。このため、Cachegrind で実行されるプログラムは、通常の実行時に比べて 20-100 倍の時間がかかります。

Cachegrind を実行するには、以下のコマンドで *program* の部分を Cachegrind でプロファイリングするプログラムに置き換えて実行します。

```
# valgrind --tool=cachegrind program
```

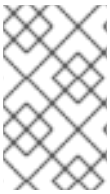
Cachegrind は、プログラム全体で、またプログラムの各機能ごとに以下の統計を収集します。

- 第 1 レベルの指示キャッシュ読み取り (または実行された指示) と読み取りミス、および最終レベルのキャッシュ指示読み取りミス
- データキャッシュ読み取り (またはメモリ読み取り)、読み取りミス、最終レベルのキャッシュデータ読み取りミス
- データキャッシュ書き込み (またはメモリ書き込み)、書き込みミス、最終レベルのキャッシュ書き込みミス
- 実行済みおよび間違って予測された条件分岐
- 実行済みおよび間違って予測された間接分岐

Cachegrind はこれらの統計のサマリー情報をコンソールに印刷し、ファイル (デフォルトで **`cachegrind.out.*pid*`**。ここでの *pid* は Cachegrind を実行するプログラムのプロセス IDになります) により詳細なプロファイリング情報を書き込みます。このファイルは以下のように、付随する

cg_annotate ツールでさらに処理可能です。

```
# cg_annotate cachegrind.out.pid
```



注記

cg_annotate は、パスの長さによって 120 文字以上の行を出力することが可能です。出力を明確にして読みやすくするには、前記のコマンドを実行する前に端末ウィンドウを少なくともこの幅に広げておくことが推奨されます。

また、Cachegrind が作成したプロファイルファイルを比較して、変更前後でのプログラムパフォーマンスをより簡単に図表にすることもできます。これを行うには、以下のように **cg_diff** コマンドを実行し、*first* を初期プロファイル出力ファイルで、*second* をその後に続くプロファイル出力ファイルで置き換えます。

```
# cg_diff first second
```

このコマンドで組み合わされた出力ファイルが作成され、その詳細は **cg_annotate** で見ることができます。

Cachegrind には出力をフォーカスする多くのオプションがあります。以下にその例を挙げます。

--I1

コンマ区切りで最初のレベルの指示キャッシュのサイズ、結合性、行のサイズを指定します: **--I1=*size*,*associativity*,*line size***.

--D1

コンマ区切りで最初のレベルのデータキャッシュのサイズ、結合性、行のサイズを指定します: **--D1=*size*,*associativity*,*line size***.

--LL

コンマ区切りで最終レベルのキャッシュのサイズ、結合性、行のサイズを指定します: **--LL=*size*,*associativity*,*line size***.

--cache-sim

キャッシュアクセスおよびミスの数の収集を有効化/無効化します。デフォルト値は **yes** (有効) です。

このオプションと **--branch-sim** の両方を無効にすると、Cachegrind には収集する情報がなくなります。

--branch-sim

分岐指示および予想を誤った数の収集を有効化/無効化します。これは Cachegrind を約 25 % 遅くするので、デフォルトは **no** (無効) に設定されています。

このオプションと **-cache-sim** の両方を無効にすると、Cachegrind には収集する情報がなくなります。

オプションの全一覧は、**/usr/share/doc/valgrind-*version*/valgrind_manual.pdf** にある資料を参照してください。

5.3.3. Massif を使ったヒープおよびスタック領域のプロファイリング

Massif は特定のプログラムが使用するヒープ領域を測定します。測定対象は、使用可能な領域とブックキーピングおよび調整目的に割り当てられている追加領域の両方です。これは、プログラムが使用するメモリ量の削減に役立ちます。これにより、プログラムスピードの向上が可能になり、プログラムを実行するマシンのスワップ領域をプログラムが使い尽くしてしまう可能性を低減します。Massif は、ヒープ領域の割り当てに関してプログラムのどの部分に責任があるかという詳細も提供します。Massif で実行されるプログラムは、通常の実行スピードよりも約 20 倍遅くなります。

プログラムのヒープ使用量をプロファイリングするには、**massif**を使用する Valgrind ツールとして指定します。

```
# valgrind --tool=massif program
```

Massif が収集したプロファイリングデータは、デフォルトで **massif.out.*pid*** と呼ばれるファイルに書き込まれます。ここでは、*pid* は指定された*program* のプロセス ID になります。

このプロファイリングデータは、以下のように **ms_print** コマンドでグラフ化することも可能です。

```
# ms_print massif.out.pid
```

これで、プログラムの実行に対するメモリ消費量を表示するグラフが作成されます。また、ピークメモリ割り当て地点を含むプログラムの様々な地点における割り当てに責任を負うサイトについての詳細情報も作成します。

Massif は、ツールの出力に指示を与えるコマンドラインオプションを数多く提供します。以下に例を挙げます。

--heap

ヒーププロファイリングを実行するかどうかを指定します。デフォルト値は **yes** です。このオプションを **no** に設定すると、ヒーププロファイリングを無効にできます。

--heap-admin

ヒーププロファイリングの有効時に管理用に使用するブロックあたりのバイト数を指定します。デフォルト値はブロックあたり **8** バイトです。

--stacks

スタックプロファイリングを実行するかどうかを指定します。デフォルト値は **no** (無効) です。スタックプロファイリングを有効にするには、このオプションを **yes** に設定します。ただし、この設定では Massif が大幅に遅くなることに留意してください。また、プロファイリングされているプログラムがコントロールするスタック部分のサイズをより分かりやすく示すために、メインスタックはスタート時にサイズがゼロであると Massif が仮定していることにも注意してください。

--time-unit

プロファイリングに使用される時間の単位を指定します。このオプションには 3 つの有効な値があります。実行済みの指示 **(i)** - これはデフォルト値で、ほとんどのケースに使えます。リアルタイム **(ms**、ミリ秒) - 特定のインスタンスでは有用です。ヒープ上に割り当て/解放されたバイトおよび/もしくはスタック **(B)** - これは異なるマシンで最も再現可能なので、非常に短期稼働のプログラムやテスト目的で有用です。このオプションは、**ms_print** で Massif 出力をグラフ化する際に有用です。

オプションの全一覧は、**/usr/share/doc/valgrind-*version*/valgrind_manual.pdf** にある資料を参照してください。

5.4. 容量のチューニング

このセクションでは、メモリやカーネル、ファイルシステムの容量、それぞれに関連するパラメーター、これらのパラメーターを調整することで発生する代償についての概要を説明します。

チューニング中にこれらの値を一時的に設定するには、echo で proc ファイルシステムの適切なファイルで必要な値を設定します。例えば、***overcommit_memory*** を一時的に **1** に設定するには、以下のように実行します。

```
# echo 1 > /proc/sys/vm/overcommit_memory
```

proc ファイルシステムのパラメーターへのパスは、変更で影響を受けるシステムによって異なることに注意してください。

これらの値を永続的なものにするには、 **sysctl** コマンドを使用する必要があります。詳細については、『導入ガイド』 を参照してくださ
い。http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

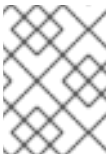
容量関連のメモリのチューニング可能なパラメーター

以下の各パラメーターは、proc ファイルシステムの**/proc/sys/vm/** にあります。

overcommit_memory

大量メモリの要求を受け入れるか拒否するかを決定する条件を定義します。このパラメーターには 3 つの値があります。

- **0** – デフォルト設定。カーネルがメモリの空き容量を概算し、明らかに無効な要求を失敗させることで、ヒューリスティックなメモリオーバーコミット処理を実行します。残念ながら、メモリは厳密なアルゴリズムではなくヒューリスティックなアルゴリズムを使用して割り当てられるため、この設定でシステム上のメモリの空き容量がオーバーロードとなる場合があります。
- **1** – カーネルはメモリオーバーコミット処理を行いません。この設定では、メモリのオーバーロードの可能性が高くなりますが、メモリ集約型のタスクのパフォーマンスも向上します。
- **2** – カーネルは、利用可能なスワップと ***overcommit_ratio*** で指定されている物理 RAM の割合の合計とメモリが同等もしくはそれよりも大きい場合の要求を拒否します。メモリのオーバーコミットのリスクを減らしたい場合は、この設定が最適です。



注記

この設定は、スワップ領域が物理メモリよりも大きいシステムにのみ推奨されます。

overcommit_ratio

overcommit_memory が **2** に設定されている場合に考慮される物理 RAM の割合を指定します。デフォルト値は **50** です。

max_map_count

プロセスが使用可能なメモリマップ領域の最大数を定義します。ほとんどのケースでは、適切なデフォルト値は **65530** です。アプリケーションがこのファイル数よりも多くマッピングする必要がある場合は、この値を増やします。

nr_hugepages

カーネルで設定される hugepages の数を定義します。デフォルト値は 0 です。システム内に物理的に連続する空白ページが十分にある場合のみ、hugepages を割り当てる (もしくは解放する) ことが可能です。このパラメーターで保持するページは、他の目的には使えません。詳細はインストール済み資料 **/usr/share/doc/kernel-doc-*kernel_version*/Documentation/vm/hugetlbpage.txt** を参照してください。

Oracle のデータベースワークロードには、システム上で実行されているデータベースすべてのシステムグローバルエリアの合計よりもわずかに大きいサイズと同等の hugepage 数を Red Hat では推奨しています。データベースのインスタンスあたり 5 つの追加 hugepage で十分です。

容量関連のカーネルの設定可能なパラメーター

以下の各パラメーターは、proc ファイルシステムの**/proc/sys/kernel/** にあります。

msgmax

メッセージキューでの単一メッセージの最大許容サイズをバイト単位で定義します。この値は、キューのサイズ (***msgmnb***) を超えることはできません。デフォルト値は**65536** です。

msgmnb

単一メッセージキューの最大サイズをバイト単位で定義します。デフォルト値は **65536** バイトです。

msgmni

メッセージキュー識別子の最大数を定義します (つまり、キューの最大数)。デフォルト値は低メモリーページの数で増大します。計算式は、全ページ数から高メモリーページ数を引き、その結果を 32 で割り、バイト数単位のページサイズでそれを掛け、最後に MSGMNB の値で割ります。

sem

プロセスとスレッドの同期を促進するセマフォは通常、データベースのワークロードを助けるように設定されます。推奨値はデータベースによって異なります。セマフォの値の詳細については、ご使用のデータベースの資料を参照してください。

このパラメーターは空白で区切られた 4 つの値を取り、それらは SEMMSL、SEMMNS、SEMOPM、および SEMMNI を表します。

shmall

一度にシステム上で使用可能な共有メモリーページの合計数を定義します。Red Hat ではデータベースのワークロードの場合、この値を ***shmmax*** を hugepage サイズで割った結果に設定することを推奨しています。ただし、Red Hat では推奨値をベンダー資料で確認することを推奨しています。

shmmax

カーネルが許可する共有メモリーセグメントの最大数をバイト数単位で定義します。Red Hat ではデータベースのワークロードの場合、この値をシステムの総メモリーサイズの 75% 以下にすること

を推奨しています。ただし、Red Hat では推奨値をベンダー資料で確認することを推奨しています。

shmmni

システム全体の共有メモリセグメントの最大数を定義します。デフォルト値は、**64** ビットと **32** ビット の両方のアーキテクチャーで **4096** です。

threads-max

システム全体でカーネルが一度に使用するスレッド (タスク) の最大数を定義します。デフォルト値は、カーネルの ***max_threads*** 値と同等です。使用されている式は以下のとおりです。

```
max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE )
```

threads-max の最低値は **20** です。

容量関連のファイルシステムのチューニング可能なパラメーター

以下の各パラメーターは、proc ファイルシステムの**/proc/sys/fs/** にあります。

aio-max-nr

すべてのアクティブな非同期 I/O コンテキスト内で許可されるイベントの最大数を定義します。デフォルト値は **65536** です。この値を変更しても、カーネルデータ構造を事前に割り当てたり、サイズの変更がされないことに留意してください。

file-max

カーネルが割り当てるファイルハンドルの最大数を一覧表示します。デフォルト値は、カーネル内の ***files_stat.max_files*** の値と一致し、これは **(mempages * (PAGE_SIZE / 1024)) / 10** もしくは **NR_FILE** (Red Hat Enterprise Linux では 8192) のどちらか大きい方に設定されます。このファイルで設定値を高くすると、使用可能なファイルハンドルの欠如で発生したエラーを解決できます。

Out of Memory (メモリ不足) 強制終了のチューニング可能な パラメーター

Out of Memory (OOM) は、スワップ領域を含むすべての利用可能なメモリが割り当てられている状態を指します。デフォルトでは、この状況はシステムのパニックを引き起こし、正常機能が停止します。しかし、***/proc/sys/vm/panic_on_oom*** パラメーターを **0** に設定すると、OOM 発生時にカーネルが **oom_killer** 機能を呼び出します。通常は、**oom_killer** は非承認のプロセスを強制終了し、システムは維持されます。

以下のパラメーターはプロセスごとの設定が可能で、**oom_killer** 機能で強制終了するプロセスの制御能力が高まります。これは、proc ファイルシステムの **/proc/*pid*/** にあり、ここでの *pid* はプロセス ID 番号になります。

oom_adj

値を **-16** から **15** の間で定義して、プロセスの***oom_score*** の決定に役立ちます。***oom_score*** の値が高いと、**oom_killer** でプロセスが強制終了される可能性が高まります。***oom_adj*** の値を **-17** に設定すると、そのプロセスの **oom_killer** は無効になります。

重要

調整されたプロセスによって生成されたプロセスは、いずれもそのプロセスの ***oom_score*** を継承します。例えば、**sshd** プロセスが **oom_killer** 機能から保護されているとすると、その SSH セッションが開始したプロセスもすべて保護されることになります。これは、OOM 発生時にシステムを救助する **oom_killer** 機能の能力に影響します。

5.5. 仮想メモリのチューニング

仮想メモリーは通常、プロセスやファイルシステムキャッシュ、カーネルが使用します。仮想メモリーの使用率は多くの要素に依存しており、それらは以下のパラメーターで影響及ぼすことができます。

swappiness

0 から 100 の値でシステムがスワップを行う程度を制御します。値が高い場合はシステムのパフォーマンスを優先し、物理メモリがアクティブでない場合にメモリからプロセスを積極的にスワップします。値が低いと対話機能を優先し、できるだけ長く物理メモリからのプロセスのスワップを回避することから、反応の待ち時間が低下します。デフォルト値は **60** です。

データベースワークロードでは、***swappiness*** の値を高く設定することは推奨されません。たとえば、Oracle データベースでは、***swappiness*** の値を **10** にすることを Red Hat では推奨しています。

```
vm.swappiness=10
```

min_free_kbytes

システム全体で空白としておく最低限のキロバイト数です。この値は各低メモリゾーンで警告を発生させる値の計算に使われます。計算後、そのサイズに比例する保存済みの空白ページ数が割り当てられます。

警告

このパラメーターを設定する際には注意してください。値が高すぎたり低すぎたりすると、損害を与える可能性があります。

min_free_kbytes の設定が低すぎると、システムによるメモリの回収が妨げられます。これによりシステムがハングし、メモリ不足によって複数のプロセスが強制終了される可能性があります。

しかしこの値を高く設定しすぎると (システムメモリ全体の 5-10%)、システムが即座にメモリ不足に陥ります。Linux は、利用可能な RAM すべてを使ってファイルシステムデータをキャッシュするように設計されています。***min_free_kbytes*** の値を高く設定すると、システムがメモリ回収に時間を費やしすぎることになります。

dirty_ratio

パーセントの値を定義します。ダーティーデータの writeout は (**pdflush** 経由で) ダーティーデータがシステムメモリ合計のこのパーセントを占める際に開始されます。デフォルト値は **20** です。

Red Hat はデータベースのワークロードに、少し低めの **15** を推奨しています。

dirty_background_ratio

パーセントの値を定義します。ダーティーデータの writeout は (**pdflush** 経由で) ダーティーデータがメモリ合計のこのパーセントを占める際にバックグラウンドで開始されます。デフォルト値は **10** です。Red Hat はデータベースのワークロードに、少し低めの **3** を推奨しています。

dirty_expire_centisecs

ダーティーデータがディスクに書き戻されるまでにページキャッシュにとどまる時間を 100 分の 1 秒単位で指定します。Red Hat では、このパラメーターの調整は推奨していません。

dirty_writeback_centisecs

カーネルフラッシャースレッドが再開して適格なデータをディスクに書き込む間隔の長さを 100 分の 1 秒単位で指定します。これを **0** に設定すると、定期書き込み動作は無効になります。Red Hat ではこのパラメーターの調整は推奨していません。

drop_caches

この値を **1**、**2**、**3** のいずれかに設定すると、カーネルはページキャッシュとスラブキャッシュの様々な組み合わせをもたらします。

1

システムはすべてのページキャッシュメモリを無効にして、解放します。

2

システムはすべての未使用スラブキャッシュメモリを解放します。

3

システムはすべてのページキャッシュとスラブキャッシュメモリを解放します。

これは非破壊的な操作です。ダーティーオブジェクトは解放できないので、このパラメーターの値を設定する前に **sync** を実行することが推奨されます。

重要

実稼働環境では、***drop_caches*** を使ったメモリの解放は推奨されません。

チューニング中にこれらの値を一時的に設定するには、proc ファイルシステム内の適切なファイルに希望する値を echo コマンドで実行します。例えば、***swappiness*** を一時的に **50** に設定するには、以下を実行します。

```
# echo 50 > /proc/sys/vm/swappiness
```

この値を永続的な設定にするには、**sysctl** コマンドを使う必要があります。詳細は 『導入ガイド』を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

第6章 入出力

6.1. 特徴

Red Hat Enterprise Linux 6 では I/O スタックで以下のようなパフォーマンス機能強化が導入されています。

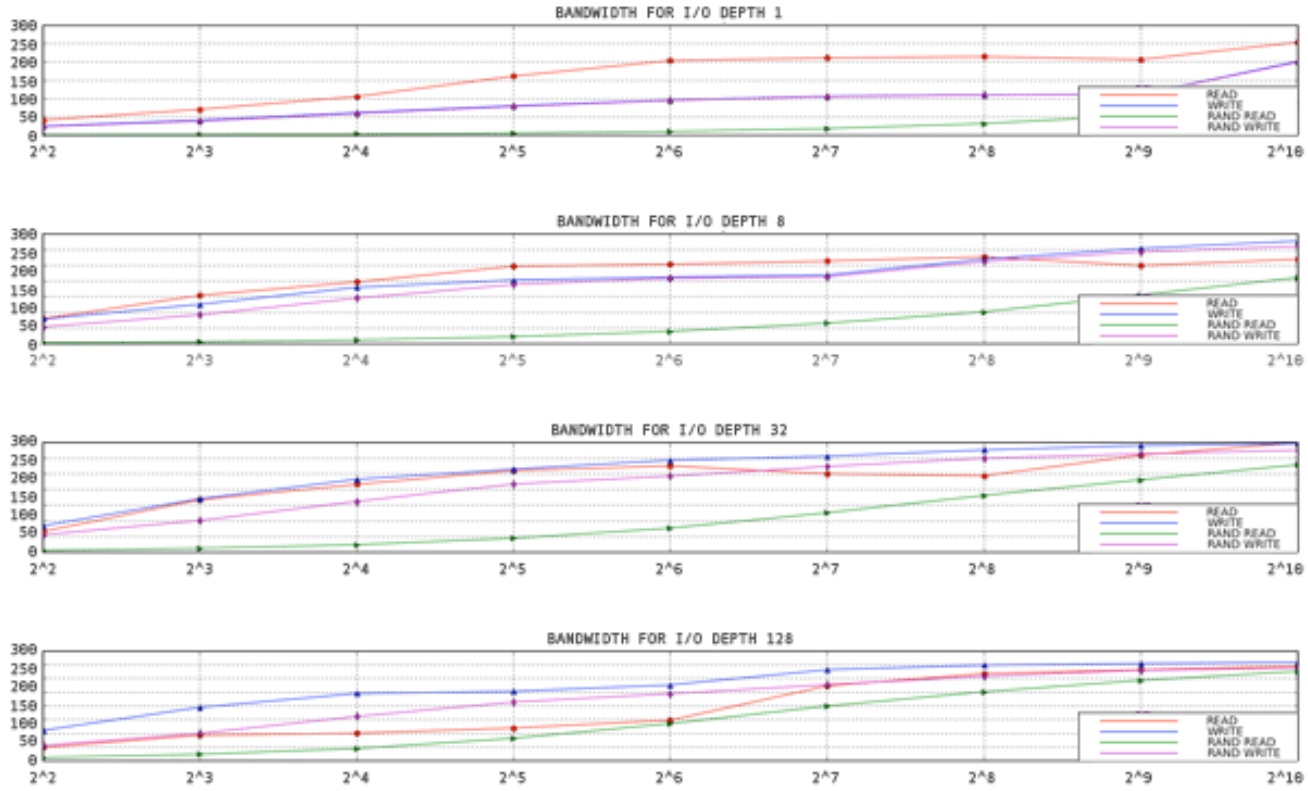
- SSD (ソリッドステートドライブ) が自動で認識され、このデバイスが実行可能な 1 秒あたりの高い I/O を活用するために I/O スケジューラーのパフォーマンスがチューニングされます。
- Discard support (破棄サポート) がカーネルに追加され、基礎となるストレージに未使用ブロックの範囲がレポートされます。これは、SSD のウェアレベリングに役立ちます。また、実際に使用中のストレージ量に関するクローズしたタブを保持することで、論理ブロックプロビジョニング (ストレージ用の仮想アドレス領域のようなもの) をサポートするストレージに役立ちます。
- Red Hat Enterprise Linux 6.1 では、ファイルシステムバリア実装が整備され、より高性能になっています。
- **pdflush** が per-backing-device flusher スレッドに置き換えられました。これは、大規模な LUN カウントの設定でシステムの拡張性を大幅に改善するものです。

6.2. 分析

ストレージスタックのパフォーマンスをうまくチューニングするには、基礎的なストレージに関する詳しい知識と様々なワークロードの下での実行方法に加えて、データがどのようにシステムで流れていくかを理解する必要があります。また、チューニングを施す実際のワークロードに関する理解も必要になってきます。

新規システムを導入する際はいつでも、ストレージのプロファイリングをボトムアップで行うことが推奨されます。raw LUN もしくはディスクから開始して、そのパフォーマンスを (カーネルのページキャッシュを迂回する) 直接 I/O を使って評価します。これは実行可能なテストのうちで最も基本的なもので、スタックで I/O パフォーマンス測定を行う際の基準となるものです。様々な I/O サイズやキューの深さにわたって連続およびランダムの読み取りや書き込みを行う (**aio-stress** などの) 基本的なワークロードジェネレーターから開始します。

以下は、**aio-stress** 実行のグラフです。それぞれで、連続書き込み、連続読み取り、ランダム書き込み、ランダム読み取り、の 4 ステージを実行しています。この例では、ツールは様々なレコードのサイズ (x 軸) とキューの深さ (グラフあたり 1 つ) にわたって実行するように設定されています。キューの深さは、ある時点で進行中の I/O 操作の数を表しています。



y 軸はメガバイト単位での帯域幅、x 軸はキロバイト単位での I/O サイズを示しています。

図6.11 スレッド、1 ファイルでの aio-stress 出力

左下から右上にかけてのスループットの線の傾きに注目してください。また、あるレコードサイズでは、進行中の I/O 数を増やすことでストレージからのスループットが増やせることにも注目してください。

このような簡単なワークロードをストレージに対して実行することで、負荷の下でのストレージパフォーマンスを理解することができます。このテストで得られたデータを保管して、より複雑なワークロードの分析時に比較してください。

デバイスマッパーもしくは md を使用する場合は、その層を次のテストに追加し、テストを繰り返してください。パフォーマンスが大幅に低下する場合、これが確実に予想されるか、説明可能になるようにします。例えば、スタックに checksumming raid 層が追加されると、パフォーマンス低下が予想されます。予想外のパフォーマンス低下は、誤って調整された I/O 操作が引き起こしている場合があります。デフォルトでは、Red Hat Enterprise Linux はパーティションとデバイスマッパーメタデータを最適に調整します。しかし、すべてのタイプのストレージが最適な調整をレポートするわけではなく、手動でのチューニングが必要となる場合もあります。

デバイスマッパーもしくは md 層を追加した後に、ブロックデバイス上にファイルシステムを追加し、直接 I/O を使ってそれに対するテストを行います。テスト結果を以前のものと比べて、不一致を間違いなく理解するようにします。Direct-write I/O は通常、事前割り当てファイル上ではよいパフォーマンスとなるので、パフォーマンステスト前にファイルを事前割り当てするようにします。

便利な総合ワークロードジェネレーターには、以下のものがあります。

- **aio-stress**
- **iozone**
- **fio**

6.3. ツール

I/O サブシステムのパフォーマンス問題の診断に役立つツールは、数多くあります。**vmstat** は、システムパフォーマンスの概要を提供します。それに続くコラムは、I/O に最も関連するものです: ***si*** (swap in)、***so*** (swap out)、***bi*** (block in)、 ***bo*** (block out)、***wa*** (I/O wait time)。 ***si*** と ***so*** は、スワップ領域がデータパーティションと同じデバイス上にある時、および全体のメモリプレッシャーの指標として便利です。***si*** と ***bi*** は読み取り操作で、***so*** と ***bo*** は書き込み操作です。各カテゴリーはキロバイトでレポートされます。***wa*** はアイドル時間で、実行キューのどの部分がブロックされて I/O の完了を待っているかを示します。

vmstat でシステムを分析することで、I/O サブシステムがパフォーマンス問題の原因であるかどうかが分かります。***free***、***buff***、***cache*** のコラムも注目に値します。***cache*** の値が ***bo*** の値とともに増えて、その後に ***cache*** が減少し ***free*** が増えていれば、システムはページキャッシュのライトバックと無効化を実行していることを示します。

vmstat がレポートする I/O 数は、全デバイスへの全 I/O の合計であることに注意してください。I/O サブシステムにパフォーマンスギャップがあるかもしれないと判断したら、**iostat** でより詳細に問題を調べることができます。これは、デバイスごとに I/O レポートティングを分類します。また、要求の平均サイズや 1 秒ごとの読み取りおよび書き込み数、進行中の I/O マージの量など、さらなる詳細情報を引き出すこともできます。

要求の平均サイズとキューの平均サイズ (**avgqu-sz**) を使うと、ストレージのパフォーマンスを特徴付ける際に生成したグラフを使って、ストレージのパフォーマンスを予想することができます。ここでは一般化が適用されます。例えば、要求の平均サイズが 4KB でキューの平均サイズが 1 の場合、スループットが非常に高性能である可能性は低くなります。

パフォーマンス数が予想するパフォーマンスにマッピングされない場合、**blktrace** を使ってより粒度の細かい分析を実行できます。**blktrace** ユーティリティは、I/O サブシステムに費やされる時間についての詳細な情報を提供します。**blktrace** の出力はバイナリー追跡ファイルで、これは **blkparse** のような別のユーティリティで後処理することができます。

blkparse は **blktrace** の仲間のユーティリティです。トレースから raw 出力を読み取り、簡略なテキストバージョンを作成します。

以下は **blktrace** 出力の例です。

```
8,64   3        1     0.000000000  4162  Q  RM 73992 + 8 [fs_mark]
8,64   3        0     0.000012707     0  m   N cfq4162S / alloced
8,64   3        2     0.000013433  4162  G  RM 73992 + 8 [fs_mark]
8,64   3        3     0.000015813  4162  P   N [fs_mark]
8,64   3        4     0.000017347  4162  I   R 73992 + 8 [fs_mark]
8,64   3        0     0.000018632     0  m   N cfq4162S / insert_request
8,64   3        0     0.000019655     0  m   N cfq4162S / add_to_rr
8,64   3        0     0.000021945     0  m   N cfq4162S / idle=0
8,64   3        5     0.000023460  4162  U   N [fs_mark] 1
8,64   3        0     0.000025761     0  m   N cfq workload slice:300
8,64   3        0     0.000027137     0  m   N cfq4162S / set_active
wl_prio:0 wl_type:2
8,64   3        0     0.000028588     0  m   N cfq4162S / fifo=(null)
8,64   3        0     0.000029468     0  m   N cfq4162S / dispatch_insert
8,64   3        0     0.000031359     0  m   N cfq4162S / dispatched a
request
8,64   3        0     0.000032306     0  m   N cfq4162S / activate rq,
drv=1
8,64   3        6     0.000032735  4162  D   R 73992 + 8 [fs_mark]
8,64   1        1     0.004276637     0  C   R 73992 + 8 [0]
```

-

このように、出力は高密度で読みづらいものです。どのプロセスがデバイスに I/O 出力を発行しているかが分かり有用ですが、**blkparse** を使うと追加情報のサマリーが理解しやすい形式で提供されます。**blkparse** のサマリー情報は、出力の一番最後に印刷されます。

```
Total (sde):
Reads Queued:          19,       76KiB  Writes Queued:     142,183,
568,732KiB
Read Dispatches:       19,       76KiB  Write Dispatches:   25,440,
568,732KiB
Reads Requeued:         0               Writes Requeued:       125
Reads Completed:       19,       76KiB  Writes Completed:   25,315,
568,732KiB
Read Merges:            0,        0KiB  Write Merges:      116,868,
467,472KiB
IO unplugs:        20,087               Timer unplugs:           0
```

サマリーでは、平均 I/O 率とマージアクティビティが示され、読み取りワークロードと書き込みワークロードが比較されています。しかしほとんどの部分で、**blkparse** の出力はそれ自体を活用するには膨大すぎるものです。幸いにも、このデータを視覚化するのに役立つツールがいくつかあります。

btt は、I/O スタックの異なるエリアで I/O が費やした時間の分析を提供します。これらのエリアは以下のとおりです。

- Q – ブロック I/O がキュー待ち
- G – 要求を取得

  新たにキュー待ちとなったブロック I/O は既存要求とマージする候補ではないので、新たなブロック層要求が割り当てられます。
- M – ブロック I/O 既存の要求とマージされます。
- I – 要求がデバイスのキューに挿入されます。
- D – 要求がデバイスに発行されます。
- C – 要求がドライバーによって完了されます。
- P – ブロックデバイスキューが接続され、要求の集合を許可します。
- U – デバイスキューが除外され、要求の集合のデバイスへの発行を許可します。

btt は、以下のようにエリア間の移動に費やされた時間に加えて、各エリアで費やされた時間も分類します。

- Q2Q – ブロック層に送信された要求間の時間
- Q2G – ブロック I/O がキュー待ちになってから要求が割り当てられるまでの時間
- G2I – 要求が割り当てられてからデバイスのキューに挿入されるまでの時間
- Q2M – ブロック I/O がキュー待ちになってから既存の要求とマージされるまでの時間
- I2D – 要求がデバイスのキューに挿入されてから実際にデバイスに発行されるまでの時間
- M2D – ブロック I/O が既存の要求とマージしてから要求がデバイスに発行されるまでの時間

- D2C – デバイスによる要求のサービス時間
- Q2C – 要求がブロック層で費やした合計時間

上記の表からワークロードについて多くのことを導き出すことができます。例えば、Q2Q が Q2C よりもかなり大きいとすると、アプリケーションが I/O を間断なく発行していないことになります。つまり、パフォーマンス問題は I/O サブシステムに全く関係していない可能性があります。D2C が非常に高い場合は、デバイスが要求を実行するのに長い時間かかっていることになります。これが意味するのは、デバイスは単にオーバーロードとなっている (共有リソースだからという理由で) か、デバイスに送信されたワークロードが最適なものではないということです。Q2G が非常に高い場合は、同時にキュー待ちとなっている要求が多くあることを意味します。つまり、ストレージが I/O 負荷に対応できないことを示しています。

最後に、**seekwatcher** は **blktrace** バイナリーデータを消費し、論理ブロックアドレス (LBA) やスループット、1 秒あたりのシーク回数、1 秒あたりの I/O (IOPS) を含むプロットのセットを作成します。

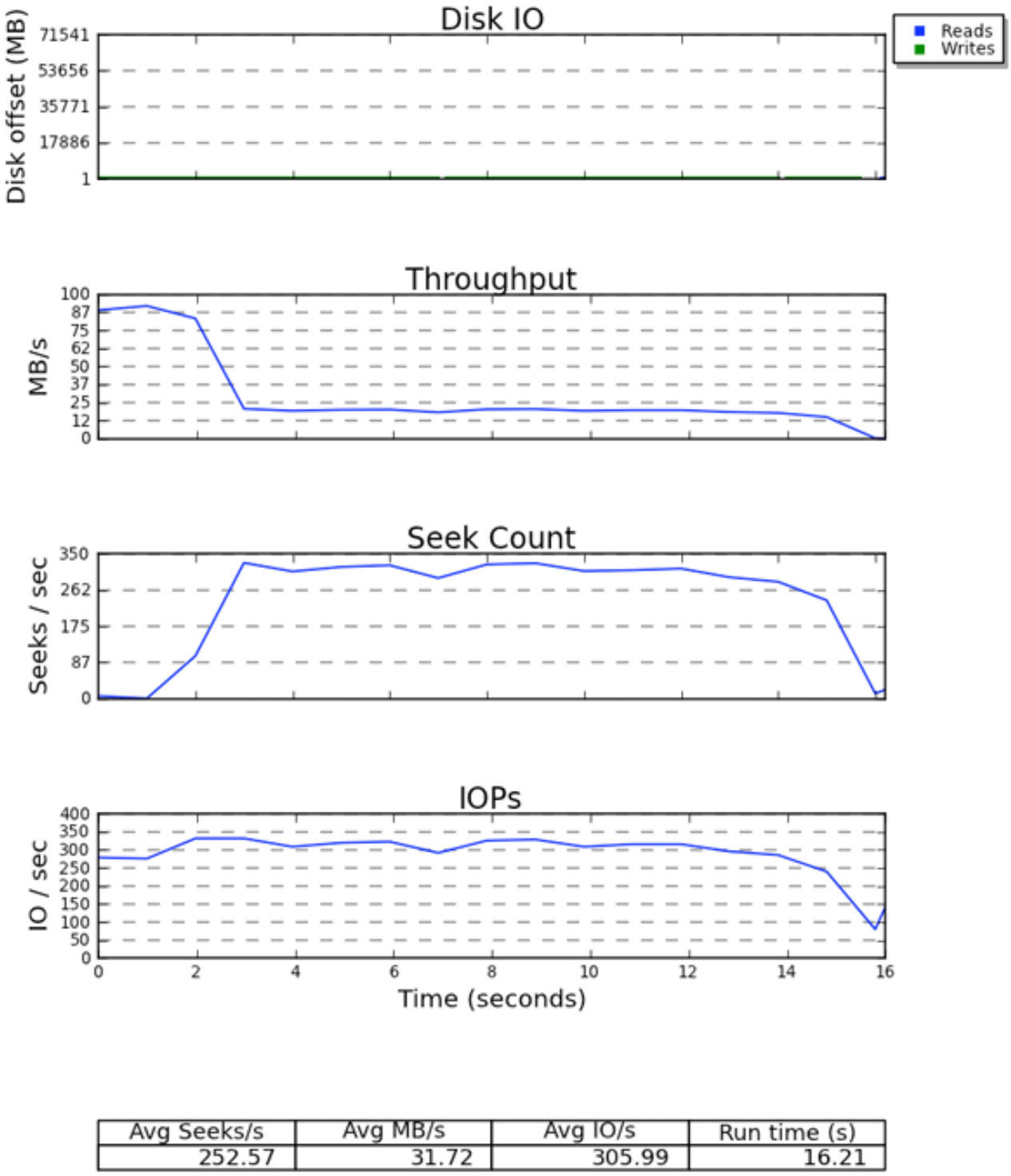


図6.2 seekwatcher の出力例

x 軸は全プロットの使用時間です。LBA プロットは、読み取りと書き込みで違う色になっています。スループットと 1 秒あたりのシーク回数のグラフの関係が興味深いものになっています。シーク回数に影響を受けやすいストレージでは、この 2 つのプロット間に逆の関係が存在します。IOPS グラフは、例えばデバイスから予想されるスループットが得られていないのに IOPS 制限に達している際などに便利なものです。

6.4. 設定

最初に必要となる決定の1つは、どの I/O スケジューラーを使用するかというものです。このセクションでは、個々のワークロードに最適なスケジューラーを決定できるように、主要スケジューラーの概要を提供します。

6.4.1. Completely Fair Queuing (CFQ)

Red Hat Enterprise Linux 6 のデフォルトの I/O スケジューラーです。CFQ は、I/O を開始したプロセスに基づいて I/O スケジューリングの決定に公平さをもたらすようにします。以下の 3 つのスケジューリングクラスが提供されています：リアルタイム (RT)、ベストエフォート (BE)、アイドル。スケジューリングのクラスは、**ionice** コマンドを使って手動で割り当てるか、**ioprio_set** システムコール経由でプログラムで割り当てることができます。デフォルトでは、プロセスはベストエフォート型のスケジューリングクラスに置かれます。リアルタイムとベストエフォートのスケジューリングクラスはさらに各クラス内で 8 つの I/O 優先順位に分けられます。このうち、0 の優先順位が一番高く、7 が一番低くなります。リアルタイムのスケジューリングクラスのプロセスは、ベストエフォートやアイドルにあるプロセスよりも積極的なスケジュールになるので、スケジュールされたリアルタイムの I/O は常にベストエフォートやアイドルの I/O よりも先に実行されます。つまり、リアルタイムの優先順位が付けられている I/O は、ベストエフォートとアイドルクラスの両方に悪影響を与える可能性があることになります。ベストエフォートスケジューリングはデフォルトのスケジューリングクラスで、このクラス内のデフォルトの優先順位は 4 になります。アイドルのスケジューリングクラス内のプロセスは、システム内に実行されていない他の I/O がない場合にのみ、実行されます。つまり、プロセスの I/O スケジューリングクラスをアイドルに設定するのは、そのプロセスからの I/O が進行に全く必要ない場合だけにします。

CFQ は、I/O 実行中の各プロセスにタイムスライスを割り当てることで公平性をもたらします。このタイムスライスの間、プロセスは (デフォルトで) 最大 8 つの要求を同時にフライト状態で保持することが可能です。スケジューラーは履歴データに基づいて、アプリケーションがさらに I/O を近いうちに発行するかどうかの予測を試みます。プロセスがさらなる I/O を発行することが予想されると、他のプロセスからの I/O の発行を待っている状態でも CFQ はアイドルになり、その I/O を待機します。

CFQ によるアイドリングのために、高速の外部ストレージアレイやソリッドステートディスクなど、大型のシークペナルティーから損害を受けないハードウェアには適さない場合が多くあります。そのようなストレージで CFQ の使用が必要な場合 (例えば、cgroup 比例加重 I/O スケジューラーも使いたい場合)、CFQ パフォーマンスの改善には設定のチューニングも必要になります。**/sys/block/*device*/queue/iosched/** にある同一名のファイルで以下のパラメーターを設定します。

```
slice_idle = 0
quantum = 64
group_idle = 1
```

group_idle が 1 に設定されても、I/O ストールの可能性は残ります (これにより、バックエンドのストレージはアイドリングのためにビジーではありません)。しかし、これらのストールの頻度は、システムの各キューでのアイドリングよりも少なくなります。

CFQ は、作業を維持しない I/O スケジューラーです。つまり、(上記で説明したように) 保留中の要求がある時でもアイドルになる場合があります。作業を維持しないスケジューラーのスタッキングは、I/O パスに大幅な待ち時間をもたらす可能性があります。その一例は、ホストベースのハードウェア RAID コントローラーに加えて CFQ を使用することです。つまり、スタックで 2 つのレベルの遅延を引き起こすことになります。作業を維持しないスケジューラーは、決定の基となるデータができるだけ多くあるときに最も優れた作動となります。アルゴリズムのスケジューリングのようなスタッキングの場合、一番下のスケジューラーは上位のスケジューラーが送信したものしか見れません。つまり、下位の層でみられる I/O パターンは、実際のワークロードとはかけはなれたものになってしまいます。

チューニング可能なパラメーター

back_seek_max

後方シークは、ヘッドの位置変更で前方シークよりも大幅な遅延を発生させる可能性があるので、パフォーマンスには通常マイナスとなります。しかし、大きくない場合は、CFQ はそれらを実行します。このパラメーターは、I/O が後方シークに許可する最大の距離を KB で制御します。デフォルト値は、**16** KB です。

back_seek_penalty

後方シークはその非効率性のために、それぞれにペナルティーが関連付けられます。ペナルティーは乗数で、例えば 1024KB のディスクヘッド位置を考えてみましょう。キューに 2 つの要求があるとします。ひとつは 1008KB にあり、もうひとつは 1040KB にあります。これら 2 つの要求は、現在のヘッド位置から等距離にあります。しかし、後方シークペナルティー (デフォルトでは 2) を適用すると、ディスク上で後ろの位置にある要求の位置は、前の位置にあるものよりも 2 倍の距離になってしまいます。このため、ヘッドは前に移動します。

fifo_expire_async

このパラメーターは、async (バッファリングされた書き込み) 要求が実行されない時間を制御します。有効時間 (ミリ秒) の後に、単一のリソース不足となっている非同期要求は、ディスパッチリストに移動されます。デフォルト値は、**250** ミリ秒です。

fifo_expire_sync

同期 (読み取りおよび O_DIRECT 書き込み) 要求では、これは fifo_expire_async と同じです。デフォルト値は、**125** ミリ秒です。

group_idle

これが設定されると、CFQ は cgroup で I/O を発行している最後のプロセスでアイドルになります。比例加重 I/O cgroup の使用時にはこれを **1** に設定し、***slice_idle*** を **0** に設定 (通常は高速ストレージで実行) することが推奨されます。

group_isolation

グループ分離 (**1** に設定) が有効な場合、これはスループットを代償にして、グループ間の分離を強化します。通常は、グループ分離が無効になっていると、公平性が提供されるのは連続するワークロードのみです。グループ分離を有効にすると、連続するワークロードランダムなワークロードの両方の公平性が提供されます。デフォルト値は **0** (無効) です。詳細は **Documentation/cgroups/blkio-controller.txt** を参照してください。

low_latency

low latency を有効にすると (**1** に設定)、CFQ はデバイス上で I/O を発行している各プロセスに最大 300 ミリ秒の待ち時間を与えようとします。これにより、スループットよりも公平性が優先されます。low latency を無効にすると (**0** に設定)、ターゲットの待ち時間が無視され、システムの各プロセスが完全なタイムスライスを取得します。low latency はデフォルトでは有効になっています。

quantum

quantum は、CFQ が一度にストレージに送信する I/O 数を制御し、実質上はデバイスキューの深さを制限します。デフォルトでは、**8** に設定されています。ストレージはより深いキューをサポートする可能性がありますが、***quantum*** を高めると待ち時間にマイナスの影響を与え、特に大量の連続する書き込みワークロードがある場合はマイナスの影響があります。

slice_async

このパラメーターは、非同期 (バッファリングされた書き込み) I/O を発行する各プロセスに割り当てられるタイムスライスを制御します。デフォルト値は **40** ミリ秒に設定されています。

slice_idle

これは、CFQ が新たな要求を待っている間にアイドルになる時間を指定します。Red Hat Enterprise Linux 6.1 およびそれ以前でのデフォルト値は、**8** ミリ秒です。Red Hat Enterprise Linux 6.2 およびそれ以降のデフォルト値は、**0** です。値が 0 だとキューおよびサービスツリーレベルのアイドリングをすべて削除するので、外部 RAID ストレージのスループットが改善されます。ただし、これは全体のシーク回数を増やすので、内部の非 RAID ストレージのスループットを低下させる場合があります。非 RAID ストレージでの ***slice_idle*** の値は、0 よりも大きくすることが推奨されます。

slice_sync

このパラメーターは、同期 (読み取りおよび直接書き込み) I/O を発行するプロセスに割り当てられるタイムスライスを規定します。デフォルト値は **100** ミリ秒です。

6.4.2. デッドライン I/O スケジューラー

デッドライン I/O スケジューラーは、要求に対して保証された待ち時間の提供を試みます。待ち時間の測定は、要求が I/O スケジューラーに到達してからしか開始されないことに留意してください (アプリケーションは要求記述子が解放されるのを待ちながらスリープになる場合があるので、これは重要な区別です)。デフォルトでは、書き込みよりも読み取りが優先されます。これは、アプリケーションが読み取り I/O をブロックする可能性の方が高いからです。

デッドラインは、I/O をバッチで送ります。バッチは、LBA で昇順となっている読み取りもしくは書き込み I/O の連続です。各バッチの処理後に、I/O スケジューラーは書き込み要求が長時間にわたってリソース不足になっていたかどうかをチェックし、読み取りもしくは書き込みの新たなバッチを開始するかどうかを決定します。要求の FIFO リストは各バッチの開始時に期限切れの要求があるかどうかだけをチェックされ、その後にそのバッチのデータ方向のみがチェックされます。このため、書き込みバッチが選択された場合に期限切れの読み取り要求があると、その読み取り要求は書き込みバッチが完了するまで実行されません。

チューニング可能なパラメーター

fifo_batch

これは、単一バッチで発行される読み取りまたは書き込み数を決定します。デフォルト値は **16** です。これを高く設定するとスループットは改善しますが、待ち時間も長くなる可能性があります。

front_merges

ワークロードが前方マージを生成しないと分かっている場合、これを **0** に設定できます。このチェックのオーバーヘッドを測定していない場合は、デフォルト設定のまま **(1)** にしておくことが推奨されます。

read_expire

このパラメーターで、読み取り要求が実行される時間をミリ秒で設定できます。デフォルト値は **500** ミリ秒 (0.5 秒) です。

write_expire

このパラメーターで、書き込み要求が実行される時間をミリ秒で設定できます。デフォルト値は **5000** ミリ秒 (5 秒) です。

writes_starved

このパラメーターは書き込みバッチの処理前に処理可能な読み取りバッチ数を制御します。この値を高く設定すればするほど、読み取りが優先されます。

6.4.3. Noop

Noop I/O スケジューラーは、シンプルな FIFO (first-in first-out) のスケジューリングアルゴリズムを実装します。要求のマージは 汎用ブロック層で発生しますが、これはシンプルな最後にヒットしたもののキャッシュです。システムが CPU にバインドされていてストレージが高速な場合、これが最良の I/O スケジューラーとなります。

ブロック層で利用可能なチューニング可能パラメーターは以下のとおりです。

/sys/block/sdX/queue のチューニング可能なパラメーター

add_random

いくつかのケースでは、**/dev/random** のエントロピープールに貢献している I/O イベントのオーバーヘッドは計測可能です。このような場合、この値を 0 に設定することが推奨されます。

max_sectors_kb

デフォルトでは、ディスクに送信される要求の最大サイズは **512** KB です。このパラメーターは、この値を上げたり下げたりできます。最小値は論理ブロックサイズに制限されます。最大値は ***max_hw_sectors_kb*** で制限されます。I/O サイズが内部の消去ブロックサイズを超えるとパフォーマンスが悪化する SSD もあります。そのような場合には、***max_hw_sectors_kb*** を消去ブロックのサイズに下げることが推奨されます。これは、**iozone** や **aio-stress** などの I/O ジェネレーターでレコードサイズを **512** バイトから **1** MB に変えるなどしてテストすることができます。

nomerges

このパラメーターは、基本的にデバッグの助けとなるものです。ほとんどのワークロードでは、要求のマージが役に立ちます (SSD などの高速ストレージでも)。しかし、マージを無効にすることがよい場合もあります。例えば、先読みを無効にしたりランダム I/O を実行することなく、ストレージバックエンドが処理できる IOPS の数を知りたい場合などです。

nr_requests

各要求キューには、読み取りもしくは書き込み I/O ごとに割り当て可能な要求記述子の合計数に制限があります。デフォルトではこれが **128** になっていて、プロセスをスリープにする前に 128 の読み取りと 128 の書き込みを一度にキュー待ちにすることができます。スリープ状態にされるプロセスは、要求の割り当てを試みる次のもので、必ずしも利用可能な要求のすべてを割り当てたプロセスではありません。

待ち時間に影響を受けやすいアプリケーションがある場合は、要求キューの ***nr_requests*** の値を引き下げ、ストレージ上のコマンドキューを (**1** にまで) 低い数値に制限することが推奨されます。こうすると、ライトバック I/O は利用可能なすべての要求記述子を割り当てることができず、書き込み I/O でデバイスキューを満たすことができません。***nr_requests*** が割り当てられると、I/O の実行を試みている他のすべての プロセスがスリープになり、要求が利用可能になるまで待機します。これだと (1 つのプロセスが連続してすべてを消費するのではなく) 要求がラウンドロビン方式で分配されるので、公平性が維持されます。デッドラインもしくは Noop スケジューラーの使用時にのみ、これが問題となることに留意してください。これは、デフォルトの CFQ 設定がこの状況に対して保護するためです。

optimal_io_size

状況によっては、基礎的なストレージが最適な I/O サイズをレポートします。これは、最適な I/O サイズがストライプサイズであるハードウェアおよびソフトウェア RAID で最も一般的です。この値がレポートされると、アプリケーションは可能な限り、最適な I/O サイズに調整され、その倍数の I/O を発行します。

read_ahead_kb

オペレーティングシステムは、アプリケーションがファイルやディスクから連続してデータを読み取っている際に、これを検出します。そのような場合、インテリジェントな先読みアルゴリズムを実行し、ユーザーが要求したデータよりも多くのデータをディスクから読み取ります。つまり、ユーザーが次のデータブロックを読み取ろうとする際には、そのデータはオペレーティングシステムのページキャッシュに既にあることになります。この動作の考えられるマイナス面は、オペレーティングシステムが必要以上のデータをディスクから読み取る可能性があるということです。この場合、メモリのプレッシャーが大きいという理由でデータが削除されるまで、このデータはページキャッシュのスペースをふさぐことになります。この状況では、複数のプロセスが間違って先読みを行うと、メモリのプレッシャーを高めることになります。

デバイスマッパーのデバイスでは、***read_ahead_kb*** の値を **8192** などの大きいものにすることが推奨されます。これは、デバイスマッパーのデバイスが複数の基礎的デバイスで構成されていることが多いためです。チューニングの手始めとしては、デフォルト値 **(128** KB**)** をマッピングするデバイス数で掛けたものにするとよいでしょう。

rotational

従来のハードディスクは回転式 **(**回転する円盤で構成**)** でしたが、SSD は違います。ほとんどの SSD は、これを適切にアドバタイズしますが、このフラグを適切にアドバタイズしないデバイスの場合は、rotational を手動で **0** に設定する必要がある場合があります。rotational が無効の場合は、非回転式メディアでのシーク操作にほとんどペナルティーがないことから、I/O エレベーターはシークを減らすための論理を使用しません。

rq_affinity

I/O の完了は、その I/O を発行した CPU とは別の CPUで処理できます。***rq_affinity*** を **1** に設定すると、I/O が発行された CPU にカーネルが完了を配信します。これにより、CPU データのキャッシング効果が高まります。

第7章 ファイルシステム

本章では、Red Hat Enterprise Linux でサポートされているファイルシステムの概要とそれらのパフォーマンスを最適化する方法を説明します。

7.1. ファイルシステムのチューニングで考慮すべき点

チューニングの際に考慮すべき点で全ファイルシステムに共通することがいくつかあります。使用中のシステムに選択するフォーマットとマウントのオプション、特定のシステムでパフォーマンスを向上させる可能性のあるアプリケーションに使用可能なアクション、などです。

7.1.1. フォーマットのオプション

ファイルシステムのブロックサイズ

ブロックサイズは、**mkfs** 実行時に選択することができます。有効なサイズの範囲はシステムによって異なります。上限はホストシステムの最大ページサイズで、下限は使用しているファイルシステムに依存します。デフォルトのブロックサイズは、ほとんどのユースケースに適しています。

デフォルトのブロックサイズよりも小さいファイルの作成を多く予定している場合は、ブロックサイズを小さく設定してディスク領域の無駄を最小限にできます。しかし、ブロックサイズを小さく設定すると、ファイルシステムの最大サイズを制限し、ファイルが選択されたブロックサイズよりも大きい場合は特に、新たなランタイムオーバーヘッドが発生する可能性があります。

ファイルシステムジオメトリ

システムで RAID5 などのストライプストレージを使用している場合、**mkfs** 実行時に基礎的ストレージジオメトリを使ってデータやメタデータを整えて、パフォーマンスを改善することができます。ソフトウェア RAID (LVM または MD) とエンタープライズハードウェアストレージのいくつかでは、この情報はクエリされ自動的に設定されますが、多くの場合、管理者がコマンドラインで **mkfs** を使って手動でこのジオメトリを指定する必要があります。

これらのファイルシステムの作成および保守についての詳細情報は 『ストレージ管理ガイド』 を参照してください。

外部ジャーナル

メタデータ集約型のワークロードは、(ext4 や XFS などの) ジャーナリングファイルシステムのログセクションが非常に頻繁に更新されることを意味します。ファイルシステムからジャーナルへのシーク回数を最小限にするには、ジャーナルを専用のストレージに配置します。しかし、主たるファイルシステムよりも遅い外部ストレージにジャーナルを配置すると、外部ストレージを使用することで得られる利点が帳消しになる場合もあることに注意してください。

警告

外部ジャーナルが信頼できるものであることを確認してください。外部のジャーナルデバイスが失われると、ファイルシステムが破損します。

外部ジャーナルは、マウント時に指定されたジャーナルデバイスで **mkfs** 実行時に作成されます。詳細は **mke2fs(8)**、**mkfs.xfs(8)**、**mount(8)** の man ページを参照してください。

7.1.2. マウントオプション

バリア

書き込みバリアは、揮発性の書き込みキャッシュが電源を喪失しても、ファイルシステムのメタデータが正しく書かれ、一貫性のあるストレージに整理されていることを確認するためのカーネルメカニズムです。書き込みバリアが有効なファイルシステムは、**fsync()** 経由で発信されたデータが停電中も確実に持続されるようにします。Red Hat Enterprise Linux はデフォルトでサポート対象の全ハードウェアに対してバリアを有効にします。

しかし、書き込みバリアを有効にすると非常に遅くなるアプリケーションもあります。**fsync()** を大量に使うアプリケーションや小さいファイルを多く作成したり削除したりするアプリケーションは特にそうです。揮発性の書き込みキャッシュがないストレージや、電源喪失後のファイルシステムの不整合やデータ損失が許される場合などの稀なケースでは、***nobarrier*** マウントオプションを使用してバリアを無効にできます。詳細は 『ストレージ管理ガイド』 を参照してください。

アクセスタイム (noatime)

従来はファイルが読み取られると、そのファイルのアクセスタイム (***atime***) は inode メタデータで更新される必要があり、このために新たな 書き込み I/O が行われました。正確な ***atime*** メタデータが不要な場合は、***noatime*** オプションでファイルシステムをマウントしてこのメタデータ更新を削除します。しかしほとんどの場合、Red Hat Enterprise Linux 6 カーネルにおけるデフォルトの相対的 atime (または ***relatime***) 動作のために、***atime*** のオーバーヘッドは大きなものではありません。***relatime*** 動作は、以前の ***atime*** が修正時間 (***mtime***) やステータス変更時間 (***ctime***) よりも古い場合にのみ、***atime*** を更新します。

注記

noatime オプションを有効にすると、***nodiratime*** 動作も有効になります。***noatime*** と ***nodiratime*** の両方を設定する必要はありません。

先読みサポートの強化

先読みは、データを先に取得してページキャッシュにロードすることでディスクからではなくメモリで先に使用可能となり、ファイルアクセスを速くします。大量の連続 I/O のストリーミングを行うワークロードなどでは、先読みの値が高いと効果が高まります。

tuned ツールと LVM ストライピングの使用は先読みの値を高めますが、ワークロードによってはこれでは不十分なものもあります。また、Red Hat Enterprise Linux は、検出可能なファイルシステムに基づく先読みの値の適切な設定が常にできるわけではありません。例えば、強力なストレージアレイが単一の強力な LUN として Red Hat Enterprise Linux に提示されたとすると、オペレーティングシステムはこれを強力な LUN アレイとして扱わず、デフォルトではストレージで利用可能な先読みの利点を完全には活かせないことになります。

先読みの値を表示、編集するには、**blockdev** コマンドを使います。特定のブロックデバイスに対する現在の先読みの値を表示するには、以下のコマンドを実行します。

```
# blockdev -getra device
```

そのブロックデバイスの先読みの値を修正するには、以下のコマンドを実行します。*N* は、512 バイトセクターの番号を表します。

```
# blockdev -setra N device
```

blockdev コマンドで選択された値は、リブート後には維持されないことに注意してください。ブート中にこの値を設定するには、ランレベル **init.d** スクリプトを作成することが推奨されます。

7.1.3. ファイルシステムのメンテナンス

未使用ブロックの破棄

バッチ破棄やオンライン破棄のオペレーションは、マウント済みファイルシステムの機能で、ファイルシステムが使用していないブロックを破棄します。ソリッドステートドライブとシンプロビジョニングのストレージの両方で役に立ちます。

バッチ破棄オペレーション は、 **fstrim** コマンドで明示的に実行されます。このコマンドは、ユーザーの条件と一致するファイルシステム内の未使用ブロックをすべて破棄します。両方のオペレーションタイプは、ファイルシステムの基盤となっているブロックデバイスが物理的な破棄オペレーションに対応している限り、Red Hat Enterprise Linux 6.2 以降の XFS および ext4 ファイルシステムでの使用が可能です。物理的な破棄オペレーションは、**/sys/block/*device*/queue/discard_max_bytes** の値がゼロでない場合にサポートされます。

オンライン破棄オペレーションは、マウント時に **-o discard** オプション (**/etc/fstab** 内、または **mount** コマンドで指定) で指定して、ユーザーの介入なしでリアルタイムで実行されます。オンライン破棄オペレーションは、使用済から空きに移行するブロックのみを破棄します。オンライン破棄オペレーションは、Red Hat Enterprise Linux 6.2 以降の ext4 ファイルシステムか、Red Hat Enterprise Linux 6.4 以降の XFS ファイルシステムで対応しています。

Red Hat は、システムのワークロードがバッチ破棄で対応できない場合、パフォーマンスの維持にオンライン破棄オペレーションが必要な場合以外は、バッチ破棄オペレーションを推奨しています。

7.1.4. アプリケーションの留意事項

事前割り当て

ext4、XFS、GFS2 の各ファイルシステムは、**fallocate(2)** glibc コール経由で効率的なスペースの事前割り当てをサポートします。書き込みパターンが理由でファイルが非常に断片化され、読み取りパフォーマンスが悪化する場合、スペースの事前割り当ては便利なテクニックになります。事前割り当てでは、データをスペースに書き込まずにそのディスクスペースがまるでファイルに割り当てられたかのようにマークします。事前割り当てのブロックに実際のデータが書き込まれるまで、読み取りオペレーションはゼロを返します。

7.2. ファイルシステムパフォーマンスのプロファイル

tuned-adm ツールを使うと、特定のユースケースのパフォーマンス強化のために設計された多くのプロファイル間でのスワップを容易にします。ストレージパフォーマンスの向上に特に便利なプロファイルを以下に挙げます。

latency-performance

標準的な遅延パフォーマンスチューニング用のサーバープロファイルです。これは、**tuned** と **ktune** の省電力メカニズムを無効にし、**cpuspeed** モードが **performance** に変更されます。各デバイスで I/O エレベーターが **deadline** に変更されます。***cpu_dma_latency*** パラメーターは **0** の値 (最低の待ち時間) で登録され、可能な範囲で Power Management Quality-of-Service が電源管理の待ち時間を制限します。

throughput-performance

標準的なスループットのパフォーマンスチューニング用のサーバープロファイルです。システムにエンタープライズクラスのストレージがない場合に、このプロファイルが推奨されます。**latency-performance** と同じですが、以下の点が異なります。

- ***kernel.sched_min_granularity_ns*** (スケジューラーの最小先取り粒度) が **10** ミリ秒に設定されます。
- ***kernel.sched_wakeup_granularity_ns*** (スケジューラーのウェイクアップ粒度) が **15** ミリ秒に設定されます。
- ***vm.dirty_ratio*** (仮想メモリーダーティー率) が 40% に設定されます。
- Transparent huge pages が使用可能になります。

enterprise-storage

このプロファイルは、バッテリー駆動のコントローラーキャッシュ保護とオンディスクのキャッシュ管理などを含むエンタープライズクラスのストレージがあるエンタープライズサイズのサーバー設定に推奨されます。**throughput-performance** プロファイルと同じですが、以下の点が異なります。

- ***readahead*** 値が **4x** に設定されます。
- root/boot ファイルシステム以外のファイルシステムが、**barrier=0** で再マウントされます。

tuned-adm についての詳細情報は man ページ (**man tuned-adm**) または 『電力管理ガイド』 を参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

7.3. ファイルシステム

7.3.1. Ext4 ファイルシステム

ext4 ファイルシステムは、Red Hat Enterprise Linux 5 で利用可能な ext3 ファイルシステムの拡張版です。ext4 は Red Hat Enterprise Linux 6 ではデフォルトのファイルシステムで、ファイルおよびファイルシステムの両方で最大 16 テラバイトのサイズまで対応します。また、ext3 ファイルシステムにある 32,000 までのサブディレクトリー制限を取り除きます。

注記

16 テラバイトを超えるファイルシステムについては、XFS などのスケーラブルで高機能のファイルシステムの使用が推奨されます。詳細については 「XFS ファイルシステム」を参照してください。

ext4 ファイルシステムのデフォルト値はほとんどのワークロードに最適化されていますが、パフォーマンス分析でファイルシステム動作がパフォーマンスに影響を及ぼしていることが分かる場合は、以下のようなチューニングオプションが利用可能です。

Inode table の初期化

大規模なファイルシステムでは、**mkfs.ext4** プロセスがファイルシステム内のすべての inode table を初期化するのに長時間かかる場合があります。このプロセスは、***-E lazy_itable_init=1*** オプションで先送りすることが可能です。これを使用すると、カーネルプ

ロセスはファイルシステムのマウント後に初期化を継続します。この初期化の発生率は、**mount** コマンドの ***-o init_itable=n*** オプションでコントロールできます。このバックグラウンド初期化に費やされる時間はおよそ 1/n で、**n** のデフォルト値は **10** です。

Auto-fsync 動作

アプリケーションの中には既存ファイルの名前変更や切り取りおよび書き換え後に **fsync()** が常に適切に機能するとは限らないものがあるので、rename 経由の replace 操作および truncate 経由の replace 操作の後には、ext4 はファイルの自動同期をデフォルト設定します。この動作は、ほぼ以前の ext3 ファイルシステムの動作と一致しています。しかし、**fsync()** 操作は時間がかかるので、この自動動作が必要ない場合は、**mount** コマンドの ***-o noauto_da_alloc*** オプションを使って無効にします。つまり、アプリケーションは明示的に **fsync()** を使ってデータの一貫性を確保する必要があります。

ジャーナル I/O の優先順位

デフォルトでは、ジャーナルコミット I/O は通常の I/O よりもわずかに高い優先順位を与えられています。この優先順位は、**mount** コマンドの ***journal_ioprio=n*** オプションで制御可能で、デフォルト値は **3** です。有効な値の範囲は 0 から 7 で、0 が一番高い優先順位となります。

他の **mkfs** およびチューニングオプションについては、**mkfs.ext4(8)** および **mount(8)** の man ページと、kernel-doc パッケージの **Documentation/filesystems/ext4.txt** ファイルを参照してください。

7.3.2. XFS ファイルシステム

XFS は堅牢かつ非常にスケーラブルな単一ホストの 64 ビットのジャーナリングファイルシステムです。完全にエクステントベースなので、非常に大規模なファイルおよびファイルシステムをサポートします。サポート対象となるファイルシステムの最大サイズは、500 TB です。XFS システムが保持できるファイル数は、ファイルシステム内の利用可能なスペースによってのみ制限されます。

XFS はクラッシュからの迅速なリカバリを容易にするメタデータジャーナリングに対応します。また、XFS ファイルシステムはマウントしてアクティブな状態でのデフラグや拡張も可能です。加えて Red Hat Enterprise Linux 6 では XFS 固有のバックアップやリストアなどのユーティリティにも対応しています。

XFS はエクステントベースの割り当てを使用し、遅延割り当てや明示的な事前割り当てなどの数多くの割り当てスキームがあります。エクステントベースの割り当てでは、ファイルシステム内の使用スペースを追跡するよりコンパクトで効率的な方法が提供されています。また、断片化とメタデータが消費するスペースを削減することで大きなファイルのパフォーマンスを改善します。遅延割り当ては、ファイルが連続するブロックグループに書き込まれる可能性を高めることで断片化を減らし、パフォーマンスを改善します。事前割り当てを使うと、アプリケーションが事前に書き込みが必要なデータ量を認識している場合に、完全に断片化を防ぐことができます。

XFS は b-tree を使ってすぐれた I/O スケーラビリティを提供し、すべてのユーザーデータとメタデータをインデックス化します。インデックス上の全オペレーションが基礎的 b-tree の対数近似スケーラビリティの特徴を継承するので、オブジェクト数は増大します。**mkfs** 実行時に XFS が提供するチューニングオプションのいくつかは b-tree の幅が異なり、これが異なるサブシステムのスケーラビリティの特徴を変更します。

7.3.2.1. XFS の基本的なチューニング

通常、デフォルトの XFS フォーマットとマウントオプションは、ほとんどのワークロードで最適なものとなっています。Red Hat では、特定の設定変更がファイルシステムのワークロードに役立つことが予想される場合以外は、デフォルト値の使用を推奨しています。ソフトウェア RAID が使用されている

場合は、**mkfs.xfs** コマンドが自動的に正しいストライプ単位と幅で設定を行ってハードウェアと調整します。ハードウェア RAID が使用されている場合は、この作業は手動で設定する必要がある場合があります。

ファイルシステムが NFS 経由でエクスポートされ、レガシーの 32 ビット NFS クライアントがそのファイルシステムへのアクセスを必要としている場合を除いて、マルチテラバイトのファイルシステムには **inode64** マウントオプションが強く推奨されます。

頻繁に修正されるファイルシステムやバースト状態のファイルシステムには、**logbsize** マウントオプションが推奨されます。デフォルト値は **MAX (32 KB、ログストライプユニット)**で、最大サイズは 256 KB です。大幅な修正が行われるファイルシステムには、256 KB の値が推奨されます。

7.3.2.2. XFS の高度なチューニング

XFS パラメーターを変更する前に、デフォルトの XFS パラメーターがパフォーマンス問題の原因となっている理解する必要があります。これには、アプリケーションのオペレーションと、それに対するファイルシステムの反応を知る必要があります。

チューニングで修正可能もしくは低減可能である目につくパフォーマンス問題は通常、ファイルシステム内のファイルの断片化やリソース競合によって発生しています。これらの問題解決にはいくつかの方法があり、場合によってはファイルシステムの設定ではなく、アプリケーションの修正が必要になることもあります。

このプロセスを以前に行ったことがない場合は、ローカルの Red Hat サポートエンジニアのアドバイスを受けることが推奨されます。

7.3.2.2.1. 多数のファイルの最適化

XFS は、ファイルシステムが保持できるファイル数に任意の制限を課します。通常この制限は高く設定され、超えることはありません。前もってデフォルトの制限値が十分でないと分かっている場合は、**mkfs.xfs** コマンドで inodes に許可されるファイルシステムのスペースの割合を増やすことができます。ファイルシステムの作成後にファイルの制限に遭遇した場合 (空白スペースが利用可能でもファイルやディレクトリーを作成しようとする際に ENOSPC エラーで示されることが多い) は、**xfs_growfs** コマンドで制限を調整できます。

7.3.2.2.2. 単一ディレクトリー内の多数のファイルの最適化

ディレクトリーブロックサイズは、ファイルシステムの使用期間中は固定されており、**mkfs** による初期フォーマット時以外は変更できません。最小のディレクトリーブロックはファイルシステムのブロックサイズで、このデフォルト値は **MAX (4 KB、ファイルシステムのブロックサイズ)** です。通常、このディレクトリーブロックサイズを縮小する必要はありません。

ディレクトリー構造は b-tree ベースなので、ブロックサイズを変更すると物理 I/O あたりで取得もしくは修正できるディレクトリー情報の量に影響が及びます。ディレクトリーが大きくなればなるほど、特定のブロックサイズでオペレーションがより多くの I/O を必要とします。

しかし、大型のディレクトリーブロックサイズが使用されると、小型のディレクトリーブロックサイズのファイルシステム上での同一オペレーションと比較して、より多くの CPU が各修正オペレーションで消費されます。つまり、ディレクトリーサイズが小さいと、大型のディレクトリーブロックサイズの修正パフォーマンスは低くなることになります。I/O がパフォーマンスを制限する要素であるサイズにディレクトリーが達すると、大型のブロックサイズディレクトリーのパフォーマンスは向上します。

バックアップを保存するファイルシステムなど、読み取りよりも書き込みがより一般的またはより重要なファイルシステム (ファイルシステムへのバックアップの書き込みがバックアップの復元よりも頻繁に発生する) の場合、以下のようになります。

- 100–200 万までのディレクトリーエントリー (エントリー名の長さは 20–40 バイト) を必要とするファイルシステムは、4 KB のファイルシステムブロックサイズおよび 4 KB のディレクトリーブロックサイズのデフォルト設定でパフォーマンスがベストになります。
- 100-1000 万のディレクトリーエントリーを必要とするファイルシステムは、より大きい 16 KB のブロックサイズでパフォーマンスが向上します。
- 1000 万以上のディレクトリーエントリーを必要とするファイルシステムは、より大きい 64 KB のブロックサイズでパフォーマンスが向上します。

ミラー化ダウンロードサーバーのファイルシステムのように、コンテンツの閲覧が修正よりも多く、書き込みよりも読み取りがより一般的またはより重要なファイルシステムの場合、ブロックサイズが適用されるエントリー数を 10 分の 1 にします。たとえば、読み取りの多いファイルシステムでは、デフォルトのシステムおよびディレクトリーブロックサイズは、1-2 万以下のディレクトリーエントリーが最適になります。

7.3.2.2.3. 同時実行の最適化

他のファイルシステムと違って、オペレーションが非共有オブジェクトで行われていれば、XFS は多くのタイプの割り当てや解放操作を同時に行うことができます。エクステントの割り当てや解放は、これらの操作が異なる割り当てグループで行われている場合は、同時に実行できます。同様に、inode の割り当てや解放も、同時操作が異なる割り当てグループに影響を与えているのであれば、同時に実行できます。

CPU 数が多いマシンでオペレーションの同時実行を試みるマルチスレッドのアプリケーションを使用する場合は、割り当てグループの数が重要になります。割り当てグループ数が 4 つのみで、これが維持される場合は、メタデータの並行操作はこれら 4 つの CPU までしか拡張されません (システムが提供する同時実行の制限)。小型のファイルシステムでは、割り当てグループ数はシステムが提供する同時実行でサポートされるようにします。大型のファイルシステム (数十テラバイト以上) では通常、デフォルトのフォーマットオプションが十分な数の割り当てグループを作成して、同時実行の制限を回避します。

XFS ファイルシステム構造に固有の並列処理を使用するには、アプリケーションは単一の競合点を認識する必要があります。ディレクトリーを同時実行で修正することはできないので、多数のファイルを作成・削除するアプリケーションは、すべてのファイルを 1 つのディレクトリーに保存すべきではありません。作成されたディレクトリーを異なる割り当てグループに置くことで、複数のサブディレクトリーにまたがるファイルのハッシュなどのテクニックで、単一の大型ディレクトリーを使用する場合に比べてよりスケーラブルなストレージパターンが提供されます。

7.3.2.2.4. 拡張属性を使用するアプリケーションの最適化

XFS は、inode にスペースがあれば直接 inode に小さい属性を保存できます。属性が inode に一致すれば、別の属性ブロックを取得する余分な I/O を必要とせずに、その属性を取得したり修正したりできます。in-line 属性と out-of-line 属性のパフォーマンスは、out-of-line 属性が 1 桁少なくなるくらい違います。

デフォルトの 256 バイトの inode サイズでは、その inode 内で保存されているデータエクステントポインターの数により、およそ 100 バイトの属性スペースが使用可能です。デフォルトの inode サイズが本当に便利なのは、少数の小さい属性を保存する場合のみです。

mkfs 実行時に inode サイズを拡大すると、属性を in-line で保存するために利用可能なスペースを増やすことができます。512 バイトの inode は属性が利用できるスペースをおよそ 350 バイトに増やします。2 KB inode にはおよそ 1900 バイトの利用可能なスペースがあります。

しかし、in-line で保存可能な個別の属性サイズには制限があります。属性名と値の両方の最大サイズは 254 バイトです (つまり、名前の長さと値の長さが 254 バイトの属性 in-line で保存されます)。これらのサイズ制限を超えると、inode にすべての属性を保存する十分なスペースがあったとしても、属性は out of line になってしまいます。

7.3.2.2.5. 維持されたメタデータ修正の最適化

維持されたメタデータ修正の達成可能レベルを決定する際の主な要素は、ログのサイズです。ログデバイスは円形なので、末尾が上書きされる前にログ内の全修正がディスク上の実際の場所に書き込まれる必要があります。これには、すべてのダーティーメタデータを書き戻すためのかなりのシーキングを伴います。デフォルト設定では、ファイルシステム全体のサイズとの関係でログサイズが拡張されるので、ほとんどの場合ではログサイズはチューニングが不要です。

小型のログデバイスだと、メタデータのライトバックが非常に頻繁に行われることになります。ログはスペースを解放するために常に末尾をプッシュし、このため修正されたメタデータが頻繁にディスクに書き込まれ、オペレーションが遅くなります。

ログサイズを大きくすると、末尾をプッシュするイベント間の期間が増大します。これによりダーティーメタデータがより効果的に統合され、メタデータのライトバックパターンが向上し、頻繁に修正されたメタデータのライトバックが少なくなります。代償としては、大きなログはメモリ内でそのままになっているすべての変更を追跡するためにより多くのメモリが必要となる、という点です。

マシンのメモリが限られている場合、大規模なログは便利ではありません。これは、大きいログの利点が活かされる前にメモリ制限によってメタデータの書き戻しが発生するためです。このようなケースでは、大きいログではなく小さいログの方がパフォーマンス向上につながります。これは、スペース不足となるログからのメタデータの書き戻しの方が、メモリ回復による書き戻しよりも効率的だからです。

ログは常に、ファイルシステムを格納しているデバイスの基礎的ストライプユニットに一致させるようにすべきです。**mkfs** はデフォルトでこれをMD および DM のデバイスで行いますが、ハードウェア RAID の場合は指定する必要があります。これを正確に設定することで、ディスクに修正を書き込む際にログ I/O が I/Oの不整列を引き起こし、読み取り・修正・書き込みの操作がその後に続く可能性を回避します。

マウントオプションを編集することでログオペレーションをさらに改善することができます。メモリ内のログバッファー (**logbsize**) を増やすと、ログに変更を書き込むスピードが高まります。デフォルトのログバッファーサイズは **MAX** (32 KB、ログストライプユニット) で、最大サイズは 256 KB です。一般的に値が大きいとパフォーマンス速度が高まります。しかし、fsync の多いワークロードでは、大型のストライプユニットの調整を使うと小さいログバッファーの方が明らかに大型バッファーよりも速くなります。

delaylog マウントオプションも、ログへの変更数を減らして維持されるメタデータの修正パフォーマンスを向上させます。これは、ログに変更を書き込む前にメモリ内の個別の変更を統合することで達成されます。頻繁に修正されるメタデータは、修正ごとにではなく、定期的にログに書き込まれます。このオプションはダーティーメタデータによるメモリ使用量を増やし、クラッシュ発生時のオペレーション損失の可能性を高めますが、メタデータ修正速度とスケーラビリティを 1 桁以上改善できます。**fsync**、**fdatasync**、**sync** のいずれかを使ってデータおよびメタデータのディスクへの書き込みを確認している場合は、このオプションの使用でデータまたはメタデータの整合性が低下することはありません。

7.4. クラスタリング

クラスター化されたストレージはクラスター内の全サーバーに一貫性のあるファイルシステムイメージ

を提供し、サーバーによる単一の共有ファイルシステムの読み取り/書き込みを可能にします。これは、アプリケーションのインストールやパッチングといったタスクを 1 つのファイルシステムに限定することで、ストレージ管理を簡素化します。クラスター全体のファイルシステムは、アプリケーションデータの冗長コピーの必要性も排除し、バックアップおよび障害回復を簡素化します。

Red Hat の High Availability アドオンは、Red Hat Global File System 2 (Resilient Storage アドオンの一部) と共に、クラスター化されたストレージを提供します。

7.4.1. Global File System 2

Global File System 2 (GFS2) はネイティブのファイルシステムで、Linux カーネルファイルシステムと直接相互作用します。また、複数コンピューター (ノード) が同時にクラスター内の同一ストレージデバイスを共有できるようにします。GFS2 ファイルシステムは、ほぼ自動チューニングですが、手動のチューニングも可能です。このセクションでは、手動でパフォーマンスチューニングを行う際のパフォーマンスの留意点を説明します。

Red Hat Enterprise Linux 6.5 では、GFS2 に Orlov ブロックアロケーターが含まれています。これにより管理者は、ディスク上のブロック割り当てを分散させ、ディレクトリーのコンテンツをディスク上のディレクトリーの近くに配置することが可能になります。これは通常、これらのディレクトリー内での書き込み速度を速めます。

GFS2 マウントポイントのトップレベルディレクトリーに作成されたディレクトリーはすべて、自動的に間隔があけられます。別のディレクトリーをトップレベルとして扱うには、そのディレクトリーに **T** 属性のマークを付けます。以下のようになります。

```
chattr +T directory
```

これで、マークが付けられたディレクトリー内で作成されたサブディレクトリーすべてがディスク上で間隔をあけられます。

Red Hat Enterprise Linux 6.4 では、GFS2 でのファイルの断片化管理が改善されています。Red Hat Enterprise Linux 6.3 およびそれ以前で作成されたファイルは、1 つ以上のプロセスで同時に複数のファイルが書き込まれた場合、ファイルの断片化が起こりがちでした。この断片化により、大きいファイルが関係するワークロードの場合は特にスピードが遅くなっていました。Red Hat Enterprise Linux 6.4 では、同時書き込みを行うとファイルの断片化が減るため、このワークロードのパフォーマンスが向上します。

Red Hat Enterprise Linux では GFS2 向けのデフラグツールはありませんが、**filefrag** ツールでファイルを識別して一時ファイルにコピーし、この一時ファイルの名前を変更することでオリジナルファイルを置き換えるという手順を踏んで、個別ファイルをデフラグすることができます (この手順は、書き込みが連続して行われている限り Red Hat Enterprise Linux 6.4 よりも前のバージョンでも可能です)。

GFS2 は、クラスターのノード間の通信が必要となるかもしれないグローバルロッキングメカニズムを使用するので、これらノード間のファイルおよびディレクトリー競合を回避するようにシステムが設計されていると、最高のパフォーマンスが達成されます。競合の回避方法の例を以下に挙げます。

- 可能な場合に **fallocate** でファイルおよびディレクトリーを事前に割り当て、割り当てプロセスを最適化するとともにソースページをロックする必要性を回避します。

- 複数ノード間で共有されるファイルシステムのエリアを最小化し、ノード間キャッシュの無効化を最小限にするとともにパフォーマンスを改善します。例えば、複数のノードが同一ファイルシステムをマウントしても、異なるサブディレクトリーにアクセスすると、あるサブディレクトリーを別のファイルシステムに移動することでパフォーマンスが向上する可能性が高まります。

- 最適なリソースグループのサイズと数を選択します。これは、典型的なファイルサイズとシス

テム上で利用可能なツリースペースによって異なります。また、複数ノードがあるリソースグループを同時に使用しようと試みる可能性に影響を与えます。リソースグループが多すぎると、割り当てスペースを探す間にブロックの割り当てが遅くなります。一方で、リソースグループが少なすぎると、解放中にロック競合を引き起こす可能性があります。通常は、複数の設定を試して、ワークロードに最適なものを決定するのがよい方法です。

ただし、GFS2 ファイルシステムのパフォーマンスに影響を与えるのは競合問題だけではありません。全体的なパフォーマンスを改善するベストプラクティスの例を以下に挙げます。

- クラスターノードから予測される I/O パターンおよびファイルシステムのパフォーマンス要件にしたがってストレージハードウェアを選択します。
- 可能な場合はソリッドステートストレージを使用してシーク時間を短縮します。
- ワークロードに適したサイズのファイルシステムを作成し、ファイルシステムが容量の 80% を超えないようにします。より小さいファイルシステムではバックアップ時間は大きさに比例して短くなり、ファイルシステムのチェックに要する時間とメモリが少なくなりますが、ワークロードに対して小さすぎると断片化が進む傾向があります。
- メタデータ集約型のワークロードの場合、もしくはジャーナル化されたデータを使用する場合に、ジャーナルサイズを大きく設定します。これはより多くのメモリを使用しますが、書き込みが必要となる前により多くのジャーナリングスペースがデータ保存に使えることから、パフォーマンスが改善します
- GFS2 ノード上のクロックが確実に同期して、ネットワーク化されたアプリケーションとの問題を回避するようにします。NTP (Network Time Protocol) の使用が推奨されます。
- アプリケーションの操作でファイルまたはディレクトリーのアクセス時間がクリティカルでないならば、ファイルシステムに **noatime** と **nodiratime** のマウントオプションをマウントします。

注記

Red Hat は GFS2 で **noatime** オプションを使用することを強く推奨します。

- クォータを使用する必要がある場合は、クォータ同期処理の頻度を下げるか、fuzzy クォータ同期を使用して継続的なクォータファイル更新から発生するパフォーマンス問題を防止します。

注記

Fuzzy quota accounting を使うと、ユーザーやグループはクォータの制限値を若干超えることが許されます。この問題を最小化するために、ユーザーもしくはグループがクォータ制限値に近づくと、GFS2 は動的に同期期間を引き下げます。

GFS2 パフォーマンスチューニングの各側面に関する詳細情報は 『Global File System 2 』 ガイドを参照してください。これは http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

第8章 ネットワーキング

Red Hat Enterprise Linux のネットワークスタックは、長期にわたって数々の自動最適化機能でアップグレードされてきました。ほとんどのワークロードで、自動設定されたネットワーク設定が最適なパフォーマンスを提供します。

ほとんどのケースでは、ネットワークのパフォーマンス問題は実際にはハードウェアの不具合やインフラストラクチャーの障害で引き起こされています。このような原因は本書の対象外となります。本章で説明されているパフォーマンス問題と解決方法は、完全に機能しているシステムの最適化において有効なものです。

ネットワークは繊細なサブシステムで、異なるパーツが敏感な接続で形成されています。オープンソースコミュニティーと Red Hat がネットワークパフォーマンスを自動で最適化する実装方法に多くの努力を注いでいるのは、このためです。それゆえ、ほとんどのワークロードでは、パフォーマンスのためにネットワークを再構成する必要すらない可能性があります。

8.1. ネットワークパフォーマンスの機能強化

Red Hat Enterprise Linux 6.1 では以下のネットワークパフォーマンス機能強化を提供していました。

Receive Packet Steering (RPS)

RPS は、単一 NIC **rx** キューが受信 **softirq** ワークロードを複数の CPU に分散することを可能にします。これを行うと、ネットワークトラフィックが単一 NIC ハードウェアキューでボトルネックとなることを防ぐのに役立ちます。

RPS を有効にするには、ターゲットの CPU 名を **/sys/class/net/*ethX*/queues/*rx-N*/rps_cpus** で特定し、***ethX*** を NIC の対応するデバイス名 (例えば **eth1**、**eth2** など) で、***rx-N*** を指定された NIC 受信キューで置き換えます。こうすることで、ファイル内で指定された CPU が ***ethX*** 上のキュー ***rx-N*** からのデータを処理できるようになります。CPU を指定する際には、キューの*キャッシュ アフィニティ* [4] を考慮します。

Receive Flow Steering (RFS)

RFS は RPS の拡張版で、アプリケーションがデータを受信し、ネットワークスタックが問い合わせる際に自動的に構成されるハッシュテーブルを管理者が設定できます。これは、どのアプリケーションがネットワークデータのどの部分を受信するかを決定します (source:destination ネットワーク情報に基づいて)。

この情報を使って、ネットワークスタックは最適な CPU をスケジュールし、各パケットを受信できます。RFS を設定するには、以下を使います。

/proc/sys/net/core/rps_sock_flow_entries

これは、カーネルが指定された CPU に誘導可能なソケット/フローの最大数を制御します。これはシステム全体での共有制限値です。

/sys/class/net/*ethX*/queues/*rx-N*/rps_flow_cnt

これは、NIC (***ethX***) 上の特定受信キュー (***rx-N***) に誘導するソケット/フローの最大数を制御します。全 NIC 上のこのファイル用のキューごとの値の合計は、**/proc/sys/net/core/rps_sock_flow_entries** と同じかそれ以下にすることに留意してください。

RPS と違って、RFS ではパケットフローの処理時に受信キューとアプリケーションの両方が同一 CPU を共有できます。場合によっては、これはパフォーマンス改善につながります。しかし、そのような改善は、キャッシュ階層やアプリケーションロードなどの要素に依存します。

TCP thin-stream の getsockopt サポート

Thin-stream は、アプリケーションがデータを非常な低率で送信し、プロトコルの再送信メカニズムが完全には飽和状態とならない送信プロトコルに使われる用語です。Thin-stream プロトコルを使用するアプリケーションは通常、TCP のような信頼性の高いプロトコル経由で送信します。ほとんどのケースでは、そのようなアプリケーションは時間の影響が大きいサービスを提供します (例: 株の取引、オンラインゲーム、管理システム)。

時間の影響を受けるサービスでは、パケットを失うとサービスの品質に多大な損害をもたらします。これを防ぐために、**getsockopt** コールが強化されて以下の 2 つのオプションをサポートするようになりました。

TCP_THIN_DUPACK

このブール値は、1 回の dupACK の後に thin stream の再送信の動的タグ付を有効にします。

TCP_THIN_LINEAR_TIMEOUTS

このブール値は、thin stream の linear timeout の動的タグ付を有効にします。

両方のオプションとも、アプリケーションが明確にアクティベートします。これらのオプションについての詳細は、**file:///usr/share/doc/kernel-doc-*version*/Documentation/networking/ip-sysctl.txt** を参照してください。thin-stream についての詳細は、**file:///usr/share/doc/kernel-doc-*version*/Documentation/networking/tcp-thin.txt** を参照してください。

透過プロキシ (TProxy) のサポート

カーネルはローカル以外でバインドされた IPv4 TCP および UDP ソケットを扱って透過プロキシをサポートするようになっています。これを有効にするには、iptable を適宜設定する必要があります。また、ポリシールーティングを適切に有効化して設定する必要があります。

透過プロキシについての詳細は **file:///usr/share/doc/kernel-doc-*version*/Documentation/networking/tproxy.txt** を参照してください。

8.2. ネットワーク設定の最適化

パフォーマンスチューニングは通常、予防的な方法で行われます。よくみられるのは、アプリケーションの実行やシステムの導入前に既知の変数を調整するというものです。この調整で効果が出ない場合は、他の変数を調整してみます。この方法の根底にある論理は、**デフォルトでは** システムは最高のパフォーマンスを発揮しないというものです。このため、システムをしかるべく調整する必要があると**考える** のです。計算し尽くされた推測の上で、このような行動に出るケースもあります。

ここまでに説明したように、ネットワークスタックはほとんど自動で最適化を行います。さらには、ネットワークに効果的なチューニングを施すには、ネットワークスタックの機能方法だけでなく、特定システムのネットワークリソース要件についても完全に理解する必要があります。ネットワークパフォーマンスの設定を間違うと、パフォーマンスは実際に低下してしまいます。

例として *bufferfloat の問題* について考えてみましょう。バッファーキューの深さを増やすと、リンクが許可するよりも広い輻輳ウィンドウのある TCP 接続になります (深いバッファリングのため)。しかし、このような接続は、フレームが多大な時間をキューで費やすため、膨大な RTT 値を持つことにも

なります。すると、輻輳を検出することができなくなるので、最適な出力が得られない結果になります。

ネットワークパフォーマンスの場合は、特定のパフォーマンス問題が明らかな場合 **以外は**、デフォルト設定を維持することが推奨されます。例えば、フレームの損失やスループットの大幅な減少などといった問題です。そのような問題でも、単にチューニング設定を上げる (バッファー/キューの長さを増やしたり、割り込み待ち時間を減らすなど) よりも、問題を細部まで調べる方がよい解決方法が見つかる場合が多くあります。

ネットワークパフォーマンス問題を適切に診断するには、以下のツールを使います。

netstat

ネットワーク接続やルーティング表、インターフェース統計、マスカレード接続、マルチキャストメンバーシップを印刷するコマンドラインユーティリティです。**/proc/net/** ファイルシステムからネットワーキングサブシステムについての情報を取得します。ファイルは以下のものです。

- **/proc/net/dev** (デバイス情報)
- **/proc/net/tcp** (TCP ソケット情報)
- **/proc/net/unix** (Unix ドメインソケット情報)

netstat および **/proc/net/** の関連ファイルについての詳細は、**netstat** の man ページ: **man netstat** を参照してください。

dropwatch

カーネルがドロップするパケットを監視する監視ユーティリティです。詳細は **dropwatch** の man ページ: **man dropwatch** を参照してください。

ip

ルートやデバイス、ポリシールーティング、トンネルを管理、監視するユーティリティです。詳細は **ip** の man ページ: **man ip** を参照してください。

ethtool

NIC 設定を表示、変更するユーティリティです。詳細は **ethtool** の man ページ: **man ethtool** を参照してください。

/proc/net/snmp

snmp エージェントで IP、ICMP、TCP、UDP 管理情報ベースに必要な ASCII データを表示するファイルです。リアルタイムの UDP-lite 統計も表示します。

『SystemTap Beginners Guide』 には、ネットワークパフォーマンスのプロファイリングと監視に使用可能なサンプルスクリプトがいくつかありま
す。http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/ から入手できます。

ネットワークパフォーマンスの問題に関連するデータを収集した後は、仮説を立てて、うまくいけば解決法を考案できるはずです。[5] 例えば、**/proc/net/snmp** で UDP 入力エラーが増えていれば、ネットワークスタックが新たなフレームをアプリケーションのソケットに加えようとする際に、1 つ以上のソケット受信キューが満杯であることを示しています。

これは、パケットが **少なくとも** 1 つのソケットキューで遅れていることを示しており、パケットのソケットキュー通過が遅すぎるか、ソケットキューに対するパケットボリュームが大きすぎることを意

味します。後者の場合は、ネットワーク集約型のアプリケーションのログを調べてデータ損失がないかどうか確認します。これを解決するには、問題となっているアプリケーションを最適化するか再構成する必要があります。

ソケット受信バッファーサイズ

ソケットの送信および受信サイズは動的に調整されているので、手動で編集する必要性はほとんどありません。SystemTap ネットワーク例 **sk_stream_wait_memory.stp** で提示される分析のようなさらなる分析でソケットキューの排出率が遅すぎると示されている場合は、アプリケーションのソケットキューの深さを高めます。これを行うには、以下のどちらかの値を設定してソケットが使用する受信バッファーのサイズを大きくします。

rmem_default

ソケットが使用する受信バッファーの **デフォルトの** サイズを制御するカーネルパラメーターです。これを設定するには、以下のコマンドを実行します。

```
sysctl -w net.core.rmem_default=N
```

N を希望する バッファーサイズのバイト数で置き換えます。このカーネルパラメーターの値を決定するには、**/proc/sys/net/core/rmem_default** を表示させます。**rmem_default** の値は **rmem_max** (**/proc/sys/net/core/rmem_max**) を超えないように注意してください。必要な場合は **rmem_max** の値を上げてください。

SO_RCVBUF

ソケットの受信バッファーの **最大** サイズをバイト数で制限するソケットオプションです。**SO_RCVBUF** の詳細については man ページ: **man 7 socket** を参照してください。

SO_RCVBUF を制限するには、**setsockopt** ユーティリティを使います。現行の **SO_RCVBUF** の値は **getsockopt** で取得できます。これらのユーティリティの詳細情報は、**setsockopt** の man ページ: **man setsockopt** を参照してください。

8.3. パケット受信の概要

ネットワークのボトルネックとパフォーマンス問題をさらに分析するには、パケット受信の機能方法を理解する必要があります。パケット受信がネットワークパフォーマンスのチューニングで重要なのは、受信パスでフレームが失われることが多いからです。受信パスでフレームが失われると、ネットワークパフォーマンスに多大なペナルティーを引き起こします。

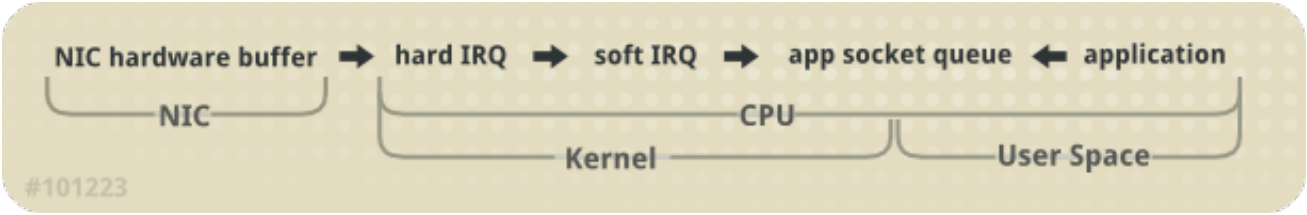


図8.1 ネットワーク受信パスの図

Linux カーネルはフレームを受信するごとに以下の 4 ステップのプロセスを行います。

1. **ハードウェア受信:** *ネットワークインターフェースカード* (NIC) が接続線上でフレームを受信します。そのドライバー設定により、NIC はフレームを内部のハードウェアバッファーメモリか指定されたリングバッファーに移動します。

2. **ハード IRQ:** NIC が CPU に割り込むことでネットフレームの存在をアサートします。これにより NIC ドライバーは割り込みを承認し、*ソフト IRQ オペレーション*をスケジュールします。

3. **ソフト IRQ:** このステージでは、実際のフレーム受信プロセスを実装し、**softirq** コンテキストで実行します。つまり、このステージは指定された CPU上で実行されている全アプリケーションよりも先に行われますが、ハード IRQ のアサートは許可します。

   (ハード IRQ と同じ CPU 上で実行することでロッキングオーバーヘッドを最小化している) このコンテキストでは、カーネルは実際に NIC ハードウェアバッファーからフレームを削除し、ネットワークスタックで処理します。ここからは、フレームはターゲットのリスニングソケットへの転送、破棄、パスのいずれかが行われます。

   フレームはソケットにパスされると、ソケットを所有するアプリケーションに追加されます。このプロセスは、NIC ハードウェアバッファーのフレームがなくなるまで、または *device weight* (**dev_weight**) まで繰り返されます。device weight についての詳細は「NIC ハードウェアバッファー」 を参照してください。

4. **アプリケーション受信:** アプリケーションがフレームを受信し、標準 POSIX コール (**read**、**recv**、**recvfrom**) で所有されているソケットからキューを外します。この時点で、ネットワークで受信されたデータはネットワークソケット上には存在しなくなります。

CPU/キャッシュアフィニティ

受信パス上で高スループットを維持するには、L2 キャッシュを **hot** に保つことが推奨されます。すでに説明したように、ネットワークバッファーは存在を知らせる IRQ と同じ CPU 上で受信されます。つまり、バッファーデータはその受信 CPU の L2 キャッシュ上にあることになります。

これを活用するには、L2 キャッシュと同じコアを共有する NIC 上でデータを最も多く受信すると予想されるアプリケーション上にプロセスアフィニティを置きます。こうすることで、キャッシュヒットの確率を最大化し、その結果パフォーマンスが改善します。

8.4. 一般的なキュー/フレーム損失問題の解決

フレーム損失の理由で圧倒的に多いのは、*queue overrun* です。カーネルがキューの長さを制限し、場合によってはキューが排出するよりも早くいっぱいになってしまいます。これが長期間発生すると、フレームのドロップが始まります。

図8.1「ネットワーク受信パスの図」にあるように、受信パスには、NIC ハードウェアバッファーとソケットキューという、2 つの主要なキューがあります。両方のキューで、queue overrun から保護するための設定が必要です。

8.4.1. NIC ハードウェアバッファー

NIC はフレームでハードウェアバッファーを満たします。すると、バッファーは **softirq** で排出を行います。これは NIC が割り込みでアサートするものです。このキューのステータスを調べるには、以下のコマンドを使います。

```
ethtool -S ethX
```

ethX を NIC の対応するデバイス名で置き換えます。こうすることで、***ethX*** 内でいくつのフレームがドロップされたかが表示されます。ドロップが発生する理由の多くは、キューがフレームを保存するバッファースペースを使い切ってしまうためです。

この問題に対処するには、以下の方法があります。

入力トラフィック

queue overrun は、入力トラフィックを遅らせることで防ぐことができます。これは、フィルタリングで統合マルチキャストグループの数を減らし、ブロードキャストトラフィックを抑えることで実現できます。

キューの長さ

別の方法では、キューの長さを伸ばすこともできます。これは、ドライバーが許可する最高値まで指定のキューでバッファー数を増やすというものです。これを行うには、以下のように ***ethX*** コマンドの **rx/tx** リングパラメーターを編集します。

```
ethtool --set-ring ethX
```

上記のコマンドに **rx** もしくは **tx** の値を追加します。詳細に関しては **man ethtool** を参照してください。

Device weight

また、キューからの排出率を高めることもできます。これを行うには、NIC の **device weight** をそれに応じて調整します。この属性は、**softirq** コンテキストが CPU を放棄して再スケジュールを行う前に NIC が受信可能なフレームの最大数を指します。これは、**/proc/sys/net/core/dev_weight** 変数で制御されます。

ほとんどの管理者は 3 番目のオプションを選ぶ傾向がありますが、このオプションの実行で影響が出ることに留意してください。1 回の反復で NIC から受信可能なフレーム数を増やすと CPU サイクルが増えることになり、この間はその CPU でアプリケーションをスケジュールできなくなります。

8.4.2. ソケットキュー

NIC ハードウェアキューのように、ソケットキューは **softirq** コンテキストからのネットワークスタックで満たされます。するとアプリケーションは **read** や **recvfrom** などのコールで対応するソケットのキューを排出します。

このキューのステータスを監視するには、**netstat** ユーティリティを使います。**Recv-Q** コラムはキューのサイズを表示します。一般的に、ソケットキューのオーバーランは NIC ハードウェアバッファーのオーバーランと同じ方法で管理されます (「NIC ハードウェアバッファー」 を参照)。

入力トラフィック

最初のオプションは、キューを満たす割合を設定することで入力トラフィックを遅らせます。これを行うには、フレームにフィルターをかけるか、前もってドロップさせます。NIC の device weight を下げることでも入力トラフィックを遅らせることができます [6]。

キューの深さ

キューの深さを高めることでも、ソケットキューのオーバーランを防ぐことができます。これを行うには、**rmem_default** カーネルパラメーターか **SO_RCVBUF** ソケットオプションのどちらかの値を増やします。これら両方に関する詳細は 「ネットワーク設定の最適化」 を参照してください。

アプリケーション呼び出しの頻度

可能な場合はいつでも、アプリケーションを最適化して呼び出し頻度を高めます。これは、ネットワークアプリケーションを修正または再構成して、POSIX 呼び出し (**recv** や **read** など) の実行頻度を高くすることで行います。こうすることで、アプリケーションによるキューの排出が速くなります。

管理者の多くにとっては、キューの深さを高めることは望ましい解決法です。これは最も簡単な解決法ですが、常に長期的に機能するとは限りません。ネットワーキング技術の速度が高まるにつれて、ソケットキューが満たされる時間は短くなり続けます。長期的には、これに対応してキューの深さを再度調整することを意味します。

最良の解決法は、アプリケーションを強化または設定してカーネルからのデータの排出を速めることです。たとえアプリケーションスペースにデータを加えることになったとしてもです。こうすることで、データを必要に応じてスワップしたりページバックしたりすることができるようになるので、データの保存がより柔軟になります。

8.5. マルチキャストにおける留意点

複数のアプリケーションがマルチキャストグループをリッスンする際は、個別のソケットのネットワークデータを故意に複製するために、マルチキャストフレームを扱うカーネルコードが必要になります。この複製は時間がかかり、**softirq** コンテキストで発生します。

このため、一つのマルチキャストグループに複数のリスナーを追加すると、**softirq** コンテキストの実行時間に直接影響します。マルチキャストグループにリスナーを加えると、カーネルがそのグループ向けに受信された各フレームに新たなコピーを作成する必要性が生まれます。

トラフィックボリュームとリスナー数が少ない場合は、この影響は限られたものです。しかし、複数のソケットがトラフィックの多いマルチキャストグループをリッスンすると、**softirq** コンテキストの実行時間が長くなり、ネットワークカードとソケットキューの両方でフレームドロップが発生します。**softirq** ランタイムが長くなると、アプリケーションが高負荷のシステムで稼働する機会が少なくなるので、高ボリュームのマルチキャストグループをリッスンするアプリケーションの数が増えればマルチキャストフレームが失われる割合が高くなります。

「ソケットキュー」 または 「NIC ハードウェアバッファー」 の説明にあるように、ソケットキューと NIC ハードウェアバッファーを最適化することで、このフレームの損失を解決できます。別の方法としては、アプリケーションのソケットの使用を最適化することもできます。これを行うには、アプリケーションが単一ソケットを制御するように設定し、受信したネットワークデータを迅速に別のユーザースペースのプロセスに配布します。

8.6. RECEIVE-SIDE SCALING (RSS)

Receive-Side Scaling (RSS) はマルチキュー受信とも呼ばれ、ネットワーク受信プロセスを複数のハードウェアベースの受信キューに配布することで、インバウンドのネットワークトラフィックを複数の CPU で処理することが可能になっています。RSS は、単一 CPU のオーバーロードで発生する受信割り込み処理におけるボトルネックを緩和し、ネットワーク遅延を低減させるために使用することができます。

ご使用のネットワークインターフェイスカードが RSS をサポートしているかを判断するには、複数の割り込みリクエストキューが **/proc/interrupts** にあるインターフェイスに関連付けられているかどうかを確認します。たとえば、**p1p1** インターフェイスを確認するには、以下を実行します。

```
# egrep 'CPU|p1p1' /proc/interrupts
      CPU0    CPU1    CPU2    CPU3    CPU4    CPU5
89:   40187       0       0       0       0       0   IR-PCI-MSI-edge
p1p1-0
90:       0     790       0       0       0       0   IR-PCI-MSI-edge
p1p1-1
91:       0       0     959       0       0       0   IR-PCI-MSI-edge
p1p1-2
92:       0       0       0    3310       0       0   IR-PCI-MSI-edge
p1p1-3
```

```
93:       0       0       0       0     622       0   IR-PCI-MSI-edge
p1p1-4
94:       0       0       0       0       0    2475   IR-PCI-MSI-edge
p1p1-5
```

上記の出力は、NIC ドライバーが 6 つの受信キューを **p1p1** インターフェイスに作成したことを示しています (**p1p1-0** から **p1p1-5**)。 また、各キューでいくつの割り込みが処理されたか、どの CPU が割り込みを処理したかも示しています。このケースでは、デフォルトでこの NIC ドライバーが CPU ごとに 1 つのキューを作成し、システムに 6 つの CPU があるので、キューが 6 つ作成されています。これは、NIC ドライバーの中ではかなり一般的なパターンです。

別の方法では、ネットワークドライバーが読み込まれた後に、**ls -1 /sys/devices/*/*/*device_pci_address*/msi_irqs** の出力をチェックします。たとえば、PCI アドレスが **0000:01:00.0** のデバイスをチェックするには、以下のコマンドを実行するとそのデバイスの割り込みリクエストキューを一覧表示できます。

```
# ls -1 /sys/devices/*/*/0000:01:00.0/msi_irqs
101
102
103
104
105
106
107
108
109
```

RSS はデフォルトで有効になっています。RSS のキューの数 (または、ネットワークアクティビティーを処理するはずの CPU) は、適切なネットワークデバイスドライバーで設定されます。**bnx2x** ドライバーの場合は、***num_queues*** で設定されます。**sfc** ドライバーは、***rss_cpus*** パラメーターで設定されます。いずれにしても、通常は **/sys/class/net/*device*/queues/*rx-queue*/** で設定されます。ここでの *device* は (**eth1** のような) ネットワークデバイス名で、*rx-queue* は適切な受信キューの名前になります。

RSS を設定する際には、Red Hat では物理 CPU コアあたりのキューの数を 1 つに制限するように推奨しています。ハイパースレッドは分析ツールではよく個別のコアとして表示されますが、ハイパースレッドのような論理コアを含むすべてのコア用にキューを設定することは、ネットワークパフォーマンスの面で有益であるとみなされていません。

RSS が有効になっていると、ネットワーク処理は各 CPU がキューに登録した処理量に基づいて利用可能な CPU 間で同等に配布されます。ただし、**ethtool *--show-rxfh-indir*** および ***--set-rxfh-indir*** パラメーターを使ってネットワークアクティビティーの配布方法を修正したり、特定のタイプのネットワークアクティビティーを他のものよりも重要なもにに加重配分することもできます。

irqbalance デーモンを RSS と併用してクロスノードのメモリー転送やキャッシュラインバウンスの可能性を減らすことができます。これは、処理ネットワークパケットの遅延を低減します。**irqbalance** と RSS の両方が使われている場合、**irqbalance** がネットワークデバイスに関連付けられている割り込みを適切な RSS キューに向けることで、遅延が最小限になります。

8.7. RECEIVE PACKET STEERING (RPS)

Receive Packet Steering (受信パケットステアリング: RPS) は、パケットを処理のために特定の CPU に向けるという点で RSS と同様のものです。しかし、RPS はソフトウェアレベルで導入され、単一ネットワークインターフェイスカードのハードウェアキューがネットワークトラフィックのボトルネックになることを防ぐ手助けとなります。

RPS にはハードウェアベースの RSS と比較して、以下のような利点があります。

- RPS はいかなるネットワークインターフェイスカードとも使用できます。
- 新たなプロトコルに対応するために RPS にソフトウェアフィルターを追加することが容易です。
- RPS はネットワークデバイスのハードウェア割り込み率を高めません。ただし、プロセッサー間の割り込みは取り込みます。

RPS は、**/sys/class/net/*device*/queues/*rx-queue*/rps_cpus** ファイル内でネットワークデバイスおよび受信キューごとに設定されます。ここでの *device* は **(eth0** などの) ネットワークデバイス名で、*rx-queue* は **(rx-0** などの) 適切な受信キューの名前です。

rps_cpus ファイルのデフォルト値はゼロです。この場合 RPS は無効になるので、ネットワーク割り込みを処理する CPU がパケットも処理します。

RPS を有効にするには、指定されたネットワークデバイスおよび受信キューからのパケットを処理する CPU で適切な **rps_cpus** ファイルを設定します。

rps_cpus ファイルは、コンマ区切りの CPU ビットマップを使用します。つまり、CPU がインターフェイス上の受信キューの割り込みを処理できるようにするには、ビットマップ上の位置の値を 1 に設定します。たとえば、CPU 0、1、2、および 3 で割り込みを処理するには、**rps_cpus** の値を **00001111** (1+2+4+8)、もしくは **f** (15 の 16進法での値) に設定します。

単一の送信キューのネットワークデバイスでは、同一メモリードメイン内の CPU を使用するように RPS を設定すると、最善のパフォーマンスが達成できます。これは、NUMA 以外のシステム上では、利用可能な CPU すべてが使用されることを意味します。ネットワーク割り込み率が非常に高い場合は、ネットワーク割り込みを処理する CPU を除外するとパフォーマンスが改善する場合もあります。

複数キューのネットワークデバイスでは、通常、RPS と RSS の両方を設定する利点はありません。これは、RSS はデフォルトで CPU を各受信キューにマップするように設定されるためです。ただし、CPU よりもハードウェアキューの方が少ない場合、および同一メモリードメインの CPU を使用するように RPS が設定されている場合は、RPS が利点ももたらす可能性があります。

8.8. RECEIVE FLOW STEERING (RFS)

Receive Flow Steering (RFS) は RPS の動作を拡張し、CPU キャッシュヒット率を高めることで、ネットワーク遅延を減らします。RPS がキューの長さのみに基づいてパケットを転送する部分で、RFS は RPS バックエンドを使用して最適な CPU を計算し、その後にパケットを消費するアプリケーションの場所に基づいてパケットを転送します。これにより、CPU キャッシュ効率が高まります。

RFS はデフォルトで無効になっています。RFS を有効にするには、以下の 2 つのファイルを編集する必要があります。

/proc/sys/net/core/rps_sock_flow_entries

このファイルの値を同時にアクティブとなる接続数で予測される最大値に設定します。適度なサーバー負荷の場合は、**32768** に設定することが推奨されます。実際には、入力された値はすべて、直近の 2 の累乗に切り上げ/下げられます。

/sys/class/net/*device*/queues/*rx-queue*/rps_flow_cnt

device を **(eth0** など) 設定するネットワークデバイス名に、*rx-queue* を **(rx-0** など) 設定する受信キューに置き換えます。

このファイルの値を **rps_sock_flow_entries** を **N** で割った値に設定します。ここでの**N** は、デ

バイス上の受信キューの数です。たとえば、**rps_flow_entries** が **32768** に設定され、設定済みの受信キューが16ある場合、**rps_flow_cnt** を **2048** に設定します。単一キューデバイスでは、**rps_flow_cnt** は **rps_sock_flow_entries** の値と等しくなります。

単一の送信者から受信したデータは、複数の CPU には送信されません。単一送信者から受信したデータ量が1つの CPU が処理できる以上のものである場合、フレームサイズを大きく設定して割り込み数を減らすことで CPU の処理作業の量も減らすことができます。別の方法としては、NIC オフロードオプション かより高速の CPU を検討してください。

RFS と併せて **numactl** または **taskset** を使用してアプリケーションを特定のコア、ソケット、または NUMA ノードに固定することを検討してみてください。これにより、パケットが間違った順番で処理されることを防ぐことができます。

8.9. アクセラレート RFS

アクセラレート RFS は、ハードウェアアシスタンスを追加することで RFS の速度を速めます。RFS のように、パケットはパケットを消費するアプリケーションの位置に基づいて転送されます。ただし、従来の RFS とは異なり、パケットはデータを消費するスレッドにローカルな CPU に直接送信されます。これは、アプリケーションを実行している CPU か、キャッシュ階層にあるその CPU にローカルな CPU のどちらかになります。

アクセラレート RFS は、以下の条件が満たされた場合にのみ、利用可能になります。

- アクセラレート RFS がネットワークインターフェイスカードでサポートされていること。アクセラレート RFS は、**ndo_rx_flow_steer()** netdevice 機能をエクスポートするカードでサポートされています。
- **ntuple** フィルタリングが有効になっていること。

これらの条件が満たされると、キューマッピングへの CPU が従来の RFS 設定に基づいて自動的に取り除かれます。つまり、各受信キューのドライバーによって設定される IRQ アフィニティーに基づいて、キューマッピングへの CPU が取り除かれます。従来の RFS 設定に関する詳細は、「Receive Flow Steering (RFS)」 を参照してください。

Red Hat では、RFSの使用が適していて、ネットワークインターフェイスカードがハードウェアアクセラレートをサポートしている場合は常に、アクセラレート RFS の使用を推奨しています。

---

[4] CPU と NIC の間のキャッシュアフィニティを確保すると、同一の L2 キャッシュを共有する設定になります。詳細は 「パケット受信の概要」 を参照してください。

[5] 「パケット受信の概要」 にはパケット送信に関する概要が記載されています。これは、ネットワークスタックでボトルネックが発生しやすいエリアの特定に役立ちます。

[6] Device weight は **/proc/sys/net/core/dev_weight** で制御されます。device weight についての詳細情報とその調節による影響は、「NIC ハードウェアバッファー」を参照してください。

付録A 改訂履歴

改訂 4.2-17.2 **Thu Jan 8 2015** **Kenzo Moriguchi**
翻訳、校閲完成

改訂 4.2-17.1 **Thu Jan 8 2015** **Chester Cheng**
翻訳ファイルを XML ソースバージョン 4.2-17 と同期

改訂 4.2-17 **Wed Oct 08 2014** **Laura Bailey**
RHEL 6.6 GA 向けにビルド

改訂 4.2-16 **Thu Sep 11 2014** **Laura Bailey**
非同期の修正を更新 (BZ 1058416)

改訂 4.2-14 **Wed Sep 10 2014** **Laura Bailey**
pmcd 使用の詳細を訂正 (BZ1130882)
msgmni パラメーターの記述を修正 (BZ1058416)

改訂 4.2-12 **Mon Aug 18 2014** **Laura Bailey**
perf mem の記述を修正 (BZ1088654)

改訂 4.2-10 **Mon Aug 11 2014** **Laura Bailey**
新機能のセクションを見つけやすくするため 1 レベル上に移動

改訂 4.2-9 **Mon Aug 11 2014** **Laura Bailey**
perf record の -b および -j オプションの新たな詳細を追加 (BZ1088653)
perf mem についての新たな詳細を追加 (BZ1088654)
RHEL 6.5 と RHEL 6.6 との間での perf への変更に言及 (BZ1088157)

改訂 4.2-7 **Fri Aug 01 2014** **Laura Bailey**
perf mem および perf record -b の詳細を追加 (BZ1088653、BZ1088654)

改訂 4.2-6 **Tue Jul 22 2014** **Laura Bailey**
vm.dirty_ratio パラメーター参照先の記述を仮想マシンから仮想メモリーに訂正 (BZ1114027)
便利な SystemTap スクリプトをツールの章に追加 (BZ988155)
shmall、shmmni、shmmax のパラメーターに基づくユニット使用の記述を訂正 (BZ1021386)

改訂 4.2-4 **Fri Jun 13 2014** **Laura Bailey**
irqbalance 付録に割り込みおよび CPU 禁止の詳細を追加 (BZ852981)
サポートの詳細を更新 (BZ977616、BZ970428)
RHEL 6.3 で仮想ホスト調整のプロファイルが利用可能であることに言及 (BZ1028907)

改訂 4.2-2 **Fri Jun 06 2014** **Laura Bailey**
irqbalance に関する詳細を追加 (BZ852981)
turbostat に関する詳細を追加 (BZ992461)

改訂 4.2-0 **Thu Jun 05 2014** **Laura Bailey**
テスト用の Performance Co-Pilot 設定セクションのドラフト完成 (BZ1092757)

改訂 4.1-5 **Wed Jun 04 2014** **Laura Bailey**
データベースチューニングのマイナーな詳細を訂正 (BZ971632)

改訂 4.1-2 **Wed May 21 2014** **Laura Bailey**
Performance Co-Pilot の詳細を訂正 (BZ1092757)

改訂 4.1-1 **Thu May 08 2014** **Laura Bailey**

データベースワークロードの詳細を調整可能な定義に追加 (BZ971632)

改訂 4.1-0 **Tue May 06 2014** **Laura Bailey**
Performance Co-Pilot のセクションをツールの章に追加 (BZ1092757)

改訂 4.0-43 **Wed Nov 13 2013** **Laura Bailey**
Red Hat Enterprise Linux 6.5 GA 用にビルド

改訂 4.0-42 **Tue Oct 15 2013** **Laura Bailey**
カスタマーフィードバックに基づく RHEL 6.5 用のマイナーな訂正 (BZ#1011676)

改訂 4.0-41 **Thu Sep 12 2013** **Laura Bailey**
SME フィードバックに基づく RHEL 6.5 用のマイナーな訂正
QE フィードバックに基づく RHEL 6.5 用のマイナーな訂正

改訂 4.0-38 **Tue Aug 13 2013** **Laura Bailey**
RHEL 6.4 と 6.5 間での perf における顧客向け変更点に言及
SME フィードバックに基づく RSS、RPS、RFS、およびアクセラレート RFS セクションへの最終的な訂正
SME フィードバックに基づく KSM 注記への最終的な訂正

改訂 4.0-35 **Fri Aug 09 2013** **Laura Bailey**
SME からのフィードバックを反映
KSM へのパフォーマンス改善に言及 (BZ#977627)
numa man ページの参照を訂正 (BZ#988240)
hdparm への更新に言及 (BZ#978096)

改訂 4.0-32 **Tue Aug 06 2013** **Laura Bailey**
ext3 マルチスレッド書き込みパフォーマンスへの更新に言及 (BZ#978099)
perf で利用可能な新 System z ハードウェアカウンターに言及 (BZ#977626)
ユーザー機能から/へのコピーの改善に言及 (BZ#977608)
「ハードウェアパフォーマンスポリシー (**x86_energy_perf_policy**)」 および **cpupowerutils** についての注記を追加 (BZ#853280)
turbostat man ページについて言及

改訂 4.0-30 **Tue Aug 06 2013** **Laura Bailey**
アクセラレート RFS に関するセクションを訂正 (BZ#976108)
スクリーンコードの例が境界をオーバーフローしないよう確認

改訂 4.0-28 **Mon Aug 05 2013** **Laura Bailey**
CIFS 更新からのスループットの改善に言及 (BZ#973504)
GFS2 における Orlov アルゴリズムの使用についての注記を追加 (BZ#973506)
調整済みプロファイルの記述を確認 (BZ#986147)
kernel.sched_migration_cost のデフォルト値変更に言及 (BZ#986149)
throughput-performance および latency-performance のプロファイルへの変更に言及 (BZ#986786)

改訂 4.0-26 **Wed Jul 31 2013** **Laura Bailey**
Intel の turbostat ツールについてのセクションを追加 (BZ#970396)

改訂 4.0-25 **Fri Jul 26 2013** **Laura Bailey**
Receive-Side Scaling、Receive Packet Steering、および Receive Flow Steering についてのセクションを訂正 (BZ#976108)
アクセラレート RFS に関するセクションを追加 (BZ#976108)

改訂 4.0-24 **Thu Jul 25 2013** **Laura Bailey**
Receive-Side Scaling、Receive Packet Steering、および Receive Flow Steering についてのセクションを追加 (BZ#976108)

改訂 4.0-23 **Mon Jul 01 2013** **Laura Bailey**

ユーザーが複数 IOMMU プールを使用可能 (BZ#977628)

改訂 4.0-22 **Tue Jun 25 2013** **Laura Bailey**
不足の文章を訂正 (BZ#964003)

改訂 4.0-21 **Tue May 28 2013** **Laura Bailey**
XFS セクションを更新し、ディレクトリーブロックサイズについての推奨事項を明確化

改訂 4.0-20 **Wed Jan 16 2013** **Laura Bailey**
一貫性に関連して若干修正 (BZ#868404)

改訂 4.0-18 **Tue Nov 27 2012** **Laura Bailey**
Red Hat Enterprise Linux 6.4 Beta の公開

改訂 4.0-17 **Mon Nov 19 2012** **Laura Bailey**
numad セクションに関する SME フィードバックを追加 (BZ#868404)

改訂 4.0-16 **Thu Nov 08 2012** **Laura Bailey**
numad についてのドラフトを追加 (BZ#868404)

改訂 4.0-15 **Wed Oct 17 2012** **Laura Bailey**
ブロックディスカードディスカッションに SME フィードバックを追加し、同セクションをマウントオプションに移動 (BZ#852990)
パフォーマンスプロファイルの記述を更新 (BZ#858220)

改訂 4.0-13 **Wed Oct 17 2012** **Laura Bailey**
パフォーマンスプロファイルの記述を更新 (BZ#858220)

改訂 4.0-12 **Tue Oct 16 2012** **Laura Bailey**
本書のナビゲーションを改善 (BZ#854082)
file-max の定義を修正 (BZ#854094)
threads-max の定義を修正 (BZ#856861)

改訂 4.0-9 **Tue Oct 9 2012** **Laura Bailey**
ファイルシステムの章に FSTRIM 推奨を追加 (BZ#852990)
カスタマーフィードバックにしたがって***threads-max*** パラメーターの説明を更新 (BZ#856861)
GFS2 断片化管理の改善についての注意点を更新BZ#857782)

改訂 4.0-6 **Thu Oct 4 2012** **Laura Bailey**
numastat ユーティリティについての新たなセクションを追加 (BZ#853274)

改訂 4.0-3 **Tue Sep 18 2012** **Laura Bailey**
新たなパフォーマンス機能強化についての注意点を追加 (BZ#854082)
file-max パラメーターの記述を修正 (BZ#854094)

改訂 4.0-2 **Mon Sep 10 2012** **Laura Bailey**
BTRFS セクションとファイルシステムの基本的な導入部を追加 (BZ#852978)
Valgrind の GDB との統合を記録 (BZ#853279)

改訂 3.0-15 **Thursday March 22 2012** **Laura Bailey**
tuned-adm プロファイルの記述を追加・更新 (BZ#803552)

改訂 3.0-10 **Friday March 02 2012** **Laura Bailey**
threads-max および file-max パラメーターの記述を更新 (BZ#752825)
slice_idle パラメーターのデフォルト値を更新 (BZ#785054)

改訂 3.0-8 **Thursday February 02 2012** **Laura Bailey**

「CPU パフォーマンスのチューニング」を再構成し、numactl によるタスクセット、バインディング CPU、メモリ割り当てを追加 (BZ#639784)
内部リンクを修正 (BZ#786099)

改訂 3.0-5 **Tuesday January 17 2012** **Laura Bailey**
「Valgrind を使ったメモリ使用量のプロファイリング」を若干修正 (BZ#639793)

改訂 3.0-3 **Wednesday January 11 2012** **Laura Bailey**
内部および外部のハイパーリンクの一貫性を確保 (BZ#752796)
「Valgrind を使ったメモリ使用量のプロファイリング」を追加 (BZ#639793)
「CPU パフォーマンスのチューニング」を追加し 4章*CPU* を再構成 (BZ#639784)

改訂 1.0-0 **Friday December 02 2011** **Laura Bailey**
Red Hat Enterprise Linux 6.2 GA 向けリリース