

2004 年度

オープンソースソフトウェア活用基盤整備事業

「OSS 性能・信頼性評価 / 障害解析ツール開発」

OS 層の評価

独立行政法人 情報処理推進機構

商標表記

- Linux は、Linus Torvalds の米国およびその他の国における登録商標あるいは商標です。
- Java 及びすべての Java 関連の商標及びロゴは、米国及びその他の国における米国 Sun Microsystems, Inc.の商標または登録商標です。
- Red Hat は、米国およびその他の国で Red Hat, Inc. の登録商標若しくは商標です。
- MIRACLE LINUX は、ミラクル・リナックス株式会社が使用許諾を受けている登録商標です。
- Intel, Intel Xeon, Pentium は、アメリカ合衆国およびその他の国におけるインテル コーポレーションまたはその子会社の商標または登録商標です。
- その他記載の会社名、製品名は、それぞれの会社の商号、商標もしくは登録商標です。

目次

1 概要	1-1
1.1 評価目的の概要	1-1
1.2 評価結果および分析・考察の概要	1-1
1.2.1 DBMS を模したマイクロベンチマークの結果と考察	1-1
1.2.2 OS の限界値付近での動作評価の結果と考察	1-2
2 環境定義	2-1
2.1 OS 層の DBMS を模したマイクロベンチマークの環境定義	2-1
2.1.1 diskio の動作原理	2-1
2.1.2 使用するハードウェアについて	2-7
2.1.3 使用するソフトウェアについて	2-8
2.1.4 OS のインストール	2-10
2.1.5 OS の設定	2-11
2.1.6 ツールのインストール	2-12
2.1.7 ツールの設定	2-13
2.2 OS 層の限界値ベンチマーク	2-28
2.2.1 ツールの説明	2-28
2.2.2 ハードウェア構成	2-36
2.2.3 OS のインストール	2-37
2.2.4 OS の設定	2-38
2.2.5 ツールの設定	2-40
3 評価手順	3-1
3.1 diskio による評価	3-1
3.1.1 注意事項	3-1
3.1.2 計測の実行	3-1
3.1.3 異常時の対処方法	3-3
3.1.4 ログデータの見方	3-3
3.2 Iozone による評価	3-5
3.2.1 ファイルシステムの再作成手順	3-5
3.2.2 iozone コマンド実行手順	3-6
3.2.3 OProfile 実行手順	3-10
3.2.4 LKST 実行手順	3-13

3.2.5	Iozone+OProfile 実行手順	3-15
3.2.6	Iozone+OProfile+LKST 実行手順	3-17
4	性能・信頼性評価結果と分析・考察	4-1
4.1	diskio の結果	4-1
4.1.1	はじめに	4-1
4.1.2	diskio と DBT-3 の測定結果の相関関係の確認	4-3
4.1.3	Oprofile によるカーネル内部プロファイリング	4-14
4.1.4	diskio と性能限界との関係	4-18
4.1.5	性能限界の分析と考察のまとめ	4-19
4.2	Iozone の結果	4-20
4.2.1	概要	4-20
4.2.2	CPU ビジー型	4-23
4.2.3	CPU アイドル型	4-26
4.2.4	ロック競合型	4-31
4.2.5	高負荷時の信頼性	4-39
4.2.6	評価工数についての考察	4-41
4.2.7	評価手法のまとめ	4-42
4.2.8	LKST の利点、制限事項	4-47
4.2.9	総括	4-48

1 概要

1.1 評価目的の概要

OS 層の評価では、サイジングに役立つツールの整備とツール利用方法の確立を行うことを目的として以下の 2 つの評価を実施した。

DBMS のファイル読み書きやシステムコール、カーネル空間・ユーザ空間のメモリコピーなどの挙動について近似させたマイクロ性能評価ツールを調査・作成した。今回は、マイクロベンチマークとして効果の高いと思われるパラメータに着目し、ディスク書き込みに近似するマイクロ性能評価ツール（以下、マイクロベンチ）をオープンソースソフトウェアの diskio を改良することで作成した。diskio を複数の環境で実行し、ベンチマーク評価を行うことで、Linux カーネルの性能限界を測定した。DB 層の性能・信頼性評価結果との比較によって相関関係を確認し、サイジングへの適用可能性について考察する。

OS の限界値付近での動作評価として、大規模メモリ搭載（メインメモリ 6GB）システムで I/O を高負荷状態にするベンチマーク（Iozone）とプロファイリング（OProfile）およびカーネルトレース（改良版 LKST¹）を実施した。このプロファイリングの結果から、性能限界を決めるカーネル内部コンポーネントを特定し、これによってリソース増大時の問題について報告する。また、このプロファイリング手法を紹介することで、大規模システムに限らない汎用的なカーネルのボトルネック特定方法を提供する。

1.2 評価結果および分析・考察の概要

1.2.1 DBMS を模したマイクロベンチマークの結果と考察

DBT-3 と diskio の間には相関が見られたが、相関係数を導き出すまでには至らなかった。また、OProfile を用いて diskio 使用時のカーネル内部プロファイリングを実施し、同期書き込み時のボトルネックについて探った。diskio の動作については fsync・fdatasync と osync・odsync との間に特徴的な挙動を発見した。

図 1.2-1 に示すように、diskio の fsync と DBT-3 の結果には正の相関が見られた。

¹ 改良版 LKST は、本「OSS 性能・信頼性評価 / 障害解析ツール開発」プロジェクトの成果による。

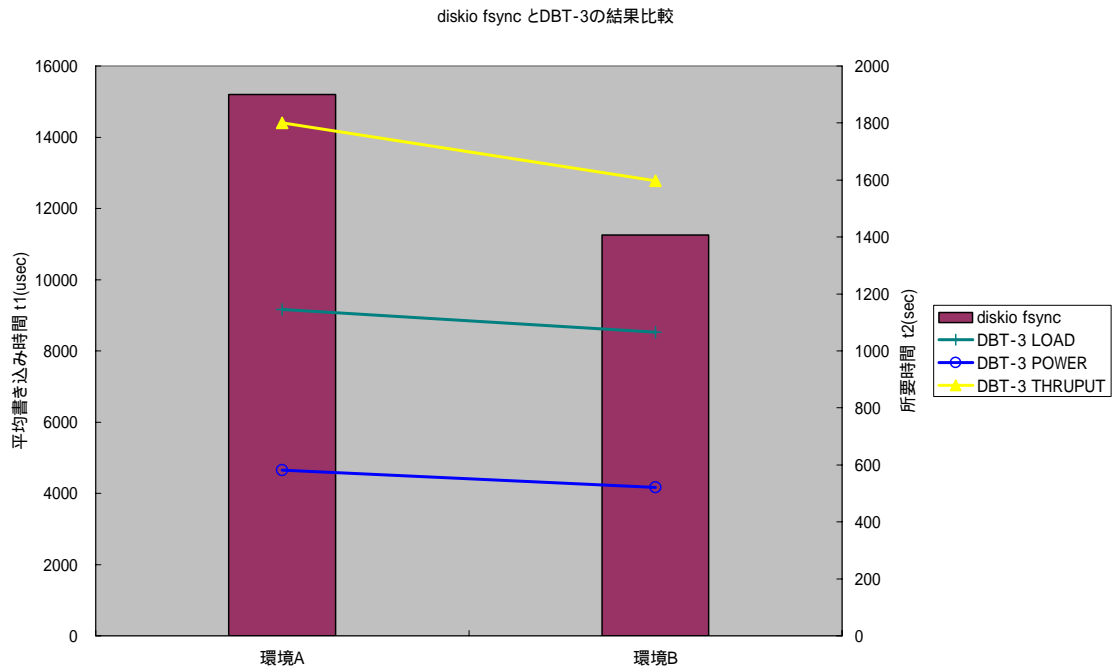


図 1.2-1 diskio fsync と DBT-3 の結果の比較

OProfile を併用した diskio 実行の結果、同期書き込み時のボトルネックをいくつか発見できた。

1.2.2 OS の限界値付近での動作評価の結果と考察

OS の限界値付近での動作評価の結果、ファイルサイズ、並列数、子プロセスまたはスレッド、iozone コマンド数を変化させた 11 種類の I/O パターンの測定を実施した。その結果、大きく分けて 3 種類に分類できることがわかった。この分類はスループット結果とプロファイリング結果とで共通だった。

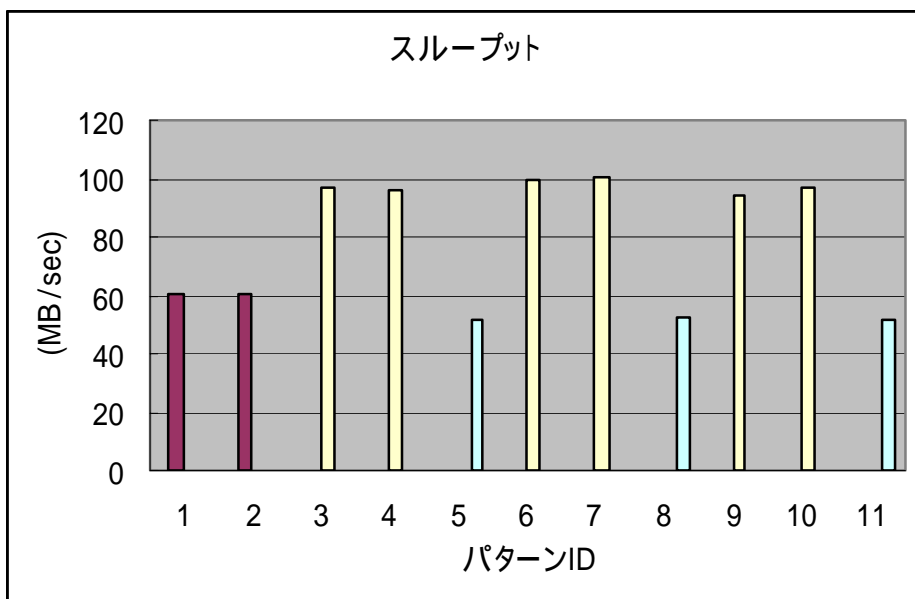


図 1.2-2 スループット結果

表 1.2-1 I/O 結果の分類

プロファイリング結果	スループット結果	I/O パターン ID
CPU ビジー型	良好	3,4,6,7,9,10
CPU アイドル型	やや悪い	1,2
ロック競合型	悪い	5,8,11

CPU ビジー型は、iozone プロセスからページキャッシュへのデータのコピー処理がボトルネックになっているケースであり、最も多く見られたパターンである。優れたスループット結果に現れている通り、非常にスムーズに I/O 処理が行われた状態にある。ボトルネックはセグメント間のメモリコピーであり、OProfile によって解析することができた。

CPU アイドル型は、8GB のファイルを単一の I/O で作成する場合に発生しており、LKST で I/O リクエスト状況を測定し、その結果を解析した。これはデバイス(ハードディスクやディスクコントローラ)がボトルネックとなり、CPU に待ち状態が発生したケースと考えられる。このような場合は、ハードディスクやコントローラの交換などによるデバイスの高速化が最も効果的なボトルネック解消方法といえる。

ロック競合型は、グローバルなカーネルロックの競合が発生したため、CPU に待ち状態が発生したケースであり、iozone コマンドを複数起動した際に観測された。具体的にカーネルのどの部分のカーネルロックが競合したかについては OProfile と LKST のデータを解析することから導き出すことができた。

2 環境定義

2.1 OS 層の DBMS を模したマイクロベンチマークの環境定義

2.1.1 diskio の動作原理

DBMS、特に PostgreSQL の動作の一部を模したマイクロベンチマークとして、diskio を利用した。diskio 利用の目的は、OS に存在するボトルネックの検出と、DBMS 性能に影響の大きい因子である HDD に対する I/O を評価することである。

本節では、PostgreSQL の動作を考察することで、同期書き込みを主たる評価ターゲットとした妥当性を示す。その上で、diskio をベンチマークとして利用するために必要となる設定パラメータの意味と指定方法を示す。

2.1.1.1 PostgreSQL の書き込み処理

本検証では DBMS 層として PostgreSQL 7.4.6 を想定している。

図 2.1-1 に PostgreSQL における書き込み処理を、書き込みの種類に注目して並べた結果を示す。

PostgreSQL はクライアントからのリクエストに応じて必要な数だけ独立したプロセスを生成する。各プロセスは共有バッファを通じてデータを共有している。共有バッファに書かれるデータは Data Volume 用データと Write Ahead Log(WAL)用データに分類される。双方とも、どのプロセスからも書き込まれる。

図左側の Data Volume 用データに関しては、check point などの同期書き込みが必要な場合を除き、非同期書き込みを行っている。図右側の WAL 用データに関しては常に同期書き込みを行っている。

一般的に、非同期書き込みはマイクロ秒台で完了し、同期書き込みはミリ秒台の処理時間を必要とする。今回、時間コストの高い同期書き込みを行う WAL 用データの書き込みに注目したベンチマークを使用することにした。そこで、ユーザ空間とカーネル空間のメモリコピーを伴う I/O を行う write システムコールや、fsync などの同期を行うシステムコールを使用するベンチマークを調査、開発する。

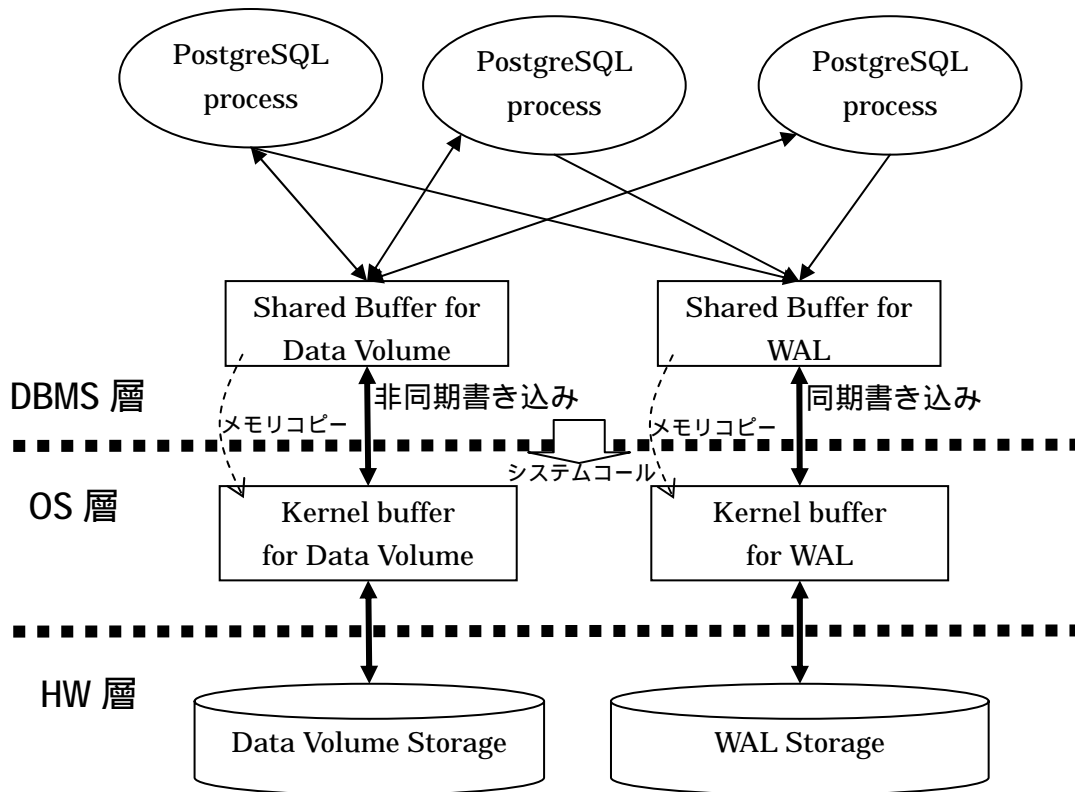


図 2.1-1 PostgreSQL における書き込み処理

2.1.1.2 diskio のマイクロベンチマークとしての利用

diskio とは、同期書き込みを繰り返し行ない、各書き込みにかかる時間を計測するプログラムである。diskio は TSC を使用しているため CPU に依存しているが、十数ナノ秒精度の正確なデータを計測することができる。DBMS で使用している同期書き込み方式をサポートし、一度にファイルに書き込むサイズが設定できるため、本検証用のマイクロベンチマークとして使用できる。

ただ、現在リリースされている diskio-1.0²では各書き込み間にランダムな長さの待ち時間を設け、ランダムなポイントへの書き込みのみが行える。しかし、これでは WAL ファイルへの書き込みをエミュレートできない。

本検証では、書き込み間の時間を最小限にし、書き込み順序の種類を増やした diskio-2.0 を開発することで、これに対応した。

² <http://developer.osdl.jp/projects/doubt/>

2.1.1.3 計測するパラメータと環境のパターン

diskio はパラメータ指定によって、動作を柔軟に変えられる性質を持っている。DBMS を模した動作をさせるために使用するパラメータは次のようになっている。

2.1.1.3.1 書き込みに関するパラメータ

diskio では書き込みを始めるポイントをアクセスポイントとして定義し、それに関連したパラメータを設定することで書き込みパターンを制御している。図 2.1-2 にアクセスポイントと関連するパラメータを示す。

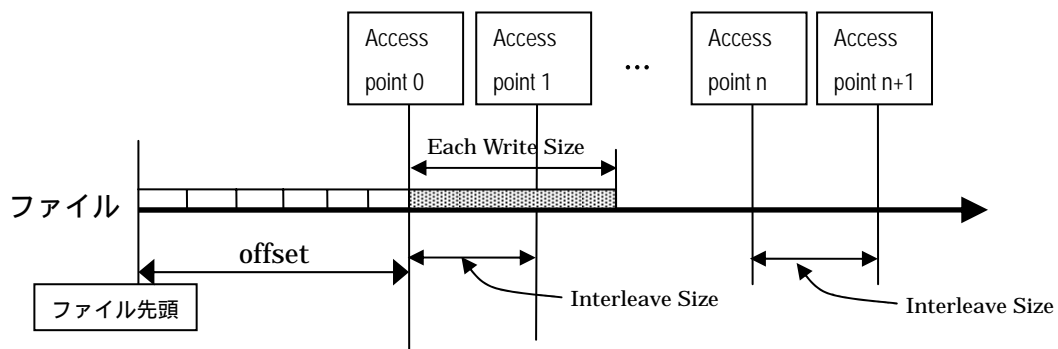


図 2.1-2 アクセスポイントと関連するパラメータ

図中の各パラメータは次のような意味を持つ。

- | | |
|-----------------|--|
| Offset | ファイル先頭から最初のアクセスポイント(アクセスポイント0)までのオフセット。単位はバイト。 |
| Interleave Size | 隣接するアクセスポイント間の距離。単位はバイト。 |
| Each Write Size | 一度の同期書き込みで書き込むサイズ。単位はバイト。 |

2.1.1.3.2 負荷生成と計測範囲に関するパラメータ

diskio が書き込みを行う回数、ならびに書き込み時間を計測する回は次のパラメータで指定する。

- | | |
|--------------------|---------------------------|
| Sampling Frequency | diskio が書き込みを行う回数。回で指定する。 |
|--------------------|---------------------------|

Start Logging	
End Logging	書き込みにかかった時間を計測する回の範囲を指定する。共に 0 から Sampling Frequency-1 までの値をとるが、Start Logging End Logging が成立しなくてはならない。最初に書き込みを行った回を第 0 回として、
Start Logging	時間計測を始める回-1 [回目]
End Logging	時間計測を終了する回+1[回目]
	のように指定する。

2.1.1.3.3 Linux の書き込み方式と計測する方式

Linux で使用できる同期書き込み方式は以下のようなものがある。³

Async open + write + fsync
 Async open + write + fdatasync
 Async open + write + sync + sync
 O_DSYNC open + write
 O_SYNC open + write
 O_SYNC open + writev
 O_DSYNC open + writev
 Async open + writev + fsync
 Async open + writev + fdatasync
 Async open + writev + sync + sync
 io_submit(2) + io_getevents(2)
 mmap(2) + msync(2)
 mmap2(2) + msync(2)
 mmap64(2) + msync(2)

今回の計測では以下の書き込み方式について計測する。

Async open + write
 *Async open + write + fsync
 *Async open + write + fdatasync
 Async open + write + sync + sync
 *O_DSYNC open + write
 *O_SYNC open + write

*がついているものは PostgreSQL が主にログ書き込みに用いている同期方式であり、PostgreSQL 起動時にどれか 1 つを指定することができる。何も指定しなければ、Async open + write + fsync に設定される。

³ 同期書き込み方式の詳細は付録 C 「Linux の同期書き込み方式」を参照のこと。

Async open + write は非同期方式であるが、各同期方式と比較のために計測する。
Async open + write + sync + sync は PostgreSQL がチェックポイント時に使用する同期方式である。

2.1.1.4 書き込み順序と Append/Overwrite

diskio では書き込みを始めるポイントをアクセスポイントとしている。書き込み順序とは、アクセスポイントを選ぶ順序を指す。diskio では Sequential⁴, Interleave, Random の 3 種類の書き込み順序をサポートする。

Sequential	アクセスポイント 0 から Each Write Size 分だけデータを書き込む。データ書き込み後は lseek(2)せず、次の書き込みを行う。Interleave Size と Each Write Size を一致させ、アクセスポイントと実際に書き込みが開始される場所を一致させる必要がある。
Interleave	アクセスポイント 0 からデータの書き込みを開始する。あるアクセスポイント i へ書き込んだ後は、アクセスポイント(i+1)へ lseek(2)し、次の書き込みを行う。
Random	アクセスポイントの中からランダム ⁵ にポイントを選びだし、そのポイントへ lseek(2)し、書き込みを行なう。

書き込み先がディスク上にあらかじめ確保されているかどうかで、Append, Overwrite の 2 種類がある。

Append	Storage 上にまだファイル領域として確保されていない領域に対し、書き込みを行う。書き込みに伴って、storage 上の領域確保も行われる。
Overwrite	Storage 上にすでにファイル領域として確保されている領域に対し、書き込みを行う。

今回の検証では Overwrite と Sequential の組み合わせについて計測する。これは、WAL 領域更新時の書き込み順序である。

⁴ スペルミスではなく、そのような名前である。

⁵ 実際には、Mersenne Twister 方式による疑似乱数である。初期値は 123456 に固定してある。この初期値自体に意味は無いが、初期値を固定する事で再現性のある、不順序な数列を得る事ができる。

2.1.1.5 今回 diskio に設定するパラメータ

本計測では PostgreSQL の動作をエミュレートするため、diskio に表 2.1-1 のようなパラメータを設定している。

表 2.1-1 diskio に設定したパラメータ

Each Write Size(Byte)	8192
Sampling Frequency	6144

Each Write Size は PostgreSQL の WAL バッファを Storage へ書き出す際の最小書き出しサイズに合わせてある。

PostgreSQL のログファイルサイズは本来 16MB である。

Sampling Frequency は、書き込み対象ファイルサイズが 48MB になるように設定してある。

diskio ではウォームアップを考慮せずに計測を行っている。

通常ベンチマークを行うときはウォームアップをして、計測するのが望ましい。

そこで、ウォームアップ分として、16MB を大きく越えるサイズにする事でウォームアップを行う。

また、ext3 ファイルシステムは indirect block access によって、ファイルサイズによって挙動が変わり、16MB はその境目になってしまうため、安定して 2 段のアクセスができる 48MB に合わせたという経緯もある。

これらのパラメータは diskio を駆動するスクリプト内で適切に設定されるようにしてあり、ユーザ側で意識する必要はない。

2.1.2 使用するハードウェアについて

2.1.2.1 今回使用したハードウェア構成

本マイクロベンチマーク(diskio) の実行環境として今回使用したハードウェア構成は表 2.1-2 の通り。

表 2.1-2 今回使用したハードウェア

モデル	Dell PowerEdge 2600 ⁶
プロセッサ	Intel Xeon 2.80GHz × 2 基
メインメモリ	4GB
ディスクドライブ	Ultra320 SCSI 36GB × 2 台

2.1.2.2 必要ハードウェア

必要なハードウェアは次の各条件を満たすものとする。

- Linux がサポートしている事
- CPU : Intel(R) Pentium(R) Processor 以降
タイムスライスカウンタ(以下 TSC と呼ぶ)を使用するため(2.1.1.1.4 ハードウェア制限事項で詳細を記述)。
- HDD
 - 計測対象 HDD : 16MB 以上の空き容量
 - ログ記録 HDD : 12MB 以上の空き容量計測対象とログ記録で別々の HDD を用いるため、2 台以上の HDD が必要
- 物理メモリ容量 : Linux の動作に必要なメモリ容量に加えて約 400Kbyte

2.1.2.3 推奨ハードウェア

以下のものを推奨ハードウェアとする。

- CPU : Intel(R) Xeon(TM) CPU 2.80GHz
- HDD
 - 計測対象 HDD : Ultra 320 SCSI HDD
48MB 以上の空き容量
 - ログ記録 HDD : 36MB 以上の空き容量
RAID、ファイバチャネルも原理的には可能であるが、今回は推奨しない。
- 物理メモリ容量 : 4GB

⁶ http://www1.us.dell.com/content/products/productdetails.aspx/pedge_2600

2.1.2.4 ハードウェア制限事項とその対処方法

2.1.2.4.1 CPU 数に関する制限事項

Intel(R) Pentium(R) Processor シリーズ以降から、TSC という 64bit 幅のレジスタが導入された。このレジスタは CPU 初期化時に 0 に設定され、以降 CPU 駆動クロックを数えつづけている。diskio は、計測始めと計測終わりの TSC の差を取ることによって時間を計測している。

シングル CPU の場合、ハイパースレッディング対応の有無に関わらず、TSC は 1 つである。従って、どのタイミングで TSC を読む命令を発行しても、読み出す TSC は常に単一である。2 つの TSC 観測点間の TSC の差分と経過時間は常に整合性が保たれている。

マルチ CPU の場合、各 CPU がそれぞれの TSC を持っており、各プロセッサはそれ自身の TSC しか参照できない。また、あるプロセスがどの CPU 上で動作しているかの情報と、TSC の値を同時に取得する術は無い。さらに TSC は初期化のタイミングがプロセッサ毎に異なるため、同一時刻における各 TSC の値は一致しない。マルチ CPU 環境下では、計測始めと計測終わりの TSC の同一性を保証する事も検出する事もできないため、再同期なしに TSC を時間計測に用いることはできない。

2.1.2.4.2 マルチ CPU システムの場合の対処方法

マルチ CPU システムの場合の対処方法は 2 通りある。TSC の同期を取る方法と取らない方法である。

TSC の同期を取る方法とは、MIRACLE LINUX V3.0 のようにカーネルの立ち上げ時に TSC の同期を取るカーネルを用いるというものである。この場合、TSC は同期されるので、diskio がどの CPU の TSC を参照しても一定の誤差範囲内に収まることが保証される。

TSC の同期を取らない方法とは、BIOS でシングル CPU マシンとして設定する、あるいはシングル CPU 用の Linux カーネルイメージを用いるというものである。この場合、シングル CPU システムと等価になるため、参照する TSC は単一になり、誤差は発生しない。

2.1.3 使用するソフトウェアについて

2.1.3.1.1 必要 OS、ソフトウェア、ライブラリ

- 対応カーネルバージョン
 - Linux kernel 2.4
- diskio 本体の使用に必要なソフトウェア、ライブラリ
 - GNU make
 - mkfs (e2fsprogs パッケージに含まれている)
 - gcc 2.9x 以降
 - glibc 2.0 以降
- 駆動用スクリプトに必要なソフトウェア、ライブラリ
 - bash-2.0 以降

- /proc/cpuinfo, /proc/version, /proc/meminfo, /proc/scsi/scsi を利用可能な環境であること

2.1.3.1.2 推奨 OS、ソフトウェア、ライブラリ

OS には MIRACLE LINUX V3.0 を推奨し、動作確認を行なった以下の rpm を推奨する。

- kernel-2.4.21-9.38AX
- kernel-smp-2.4.21-9.38AX
- kernel-debuginfo-2.4.21-9.38AXsmp
- bash-2.05b-29AX
- make-3.79.1-17
- e2fsprogs-1.32-15
- gcc-3.2.3-36
- glibc-2.3.2-95.20.1AX
- glibc-devel-2.3.2-95.20.1AX

LVM の使用も可能であるが今回は推奨しない。

2.1.3.1.3 動作確認済み OS、ソフトウェア、ライブラリ

動作確認には MIRACLE LINUX V3.0 評価版に対し、現実にシステムに用いる環境を想定して、カーネルを kernel-2.4.21-9.38AX にアップデートを行ったものを使用した。

2.1.3.1.4 ソフトウェア制限事項について

- diskio の制限事項については、2.1.2.4 ハードウェア制限事項とその対処方法を参照。
- OProfile はシングル CPU 用カーネルをサポートしていない。OProfile 使用時には smp 用カーネルを使用する必要がある。

2.1.4 OS のインストール

2.1.4.1 インストール手順について

インストールの手順は、製品付属の『インストレーションガイド』⁷に従う。インストール中のオプション選択は表 2.1-3 の通りで、これ以外はすべてデフォルト状態とする。

表 2.1-3 選択したオプション

言語	日本語
パーティション	表 2.1-4 パーティション構成 参照
ブートローダ	MBR
タイムゾーン	東京
パッケージ選択	すべて
ログインの種類	グラフィカル

2.1.4.2 パーティションの切り方について

diskio は書き込み速度計測とログ記録の 2 種類の書き込みを行う。それぞれ、異なるパーティションに書き込む必要がある。ログ記録にはルートパーティションを用いても良いが、書き込み速度計測にはルートパーティション以外のパーティションを用いなくてはならない。これは書き込み速度計測時の条件を一定にするために、毎回ファイルシステムを初期化するためである。正確な計測のためには計測対象パーティションは HDD 自体別のものであることが望ましい。

今回は表 2.1-4 のようなパーティション構成で計測を行った。

表 2.1-4 パーティション構成

パーティション	サイズ	マウント位置
/dev/sda1	1GB	/boot
/dev/sda2	1GB	スワップ領域
/dev/sda3	16GB	/
/dev/sda4	18GB	/work
/dev/sdb1	36GB	

⁷ http://www.miraclelinux.com/products/linux/ml30/pdf/installation_guide.pdf

2.1.5 OS の設定

2.1.5.1 カーネルのインストール

kernel-2.4.21-9.38AX.i686.rpm、kernel-smp-2.4.21-9.38AX.i686.rpm、
を以下の URL のうちのいずれかよりダウンロードする。

<http://ftp.miraclelinux.com/pub/Miracle/ia32/standard/3.0/updates/RPMS/>
<http://ftp.miraclelinux.com/pub/Miracle/ia32/standard/3.0/updates/RPMS/.oldrelease/>

OProfile を用いたプロファイリングには、カーネルのデバッグ情報が必要である。デバッグ用カーネル kernel-debuginfo-2.4.21-9.38AXsmp.i686.rpm を以下の URL よりダウンロードする。

<http://ftp.miraclelinux.com/pub/Miracle/ia32/standard/3.0/updates/unsupported/RPMS/>

これらのパッケージを rpm -ivh コマンドでインストールする。

```
# rpm -ivh kernel-2.4.21-9.38AX.i686.rpm kernel-smp-2.4.21-9.38AX.i686.rpm \  
kernel-debuginfo-2.4.21-9.38AXsmp.i686.rpm
```

2.1.5.2 実行に必要な権限について

diskio 本体はユーザ権限でも動作するが、計測に使用するスクリプトが mount, umount, mkfs を行なうため、実行にはルート権限が必要である。

2.1.5.3 計測対象パーティションについて

今回の計測の目的の 1 つに、計測結果を DB 層のベンチマーク評価結果と比較、相関関係を確認することがある。このため、HDD 構成は DB 層でのベンチマーク評価時と合わせる必要がある。DBMS が Write Ahead Log(WAL)保存に用いているパーティションを本ツールでの計測対象とする。

2.1.5.4 ファイルシステムとジャーナリングモードについて

diskio は複数のファイルシステムでの計測ができるようになっているが、今回は ext3 ファイルシステムのみを計測対象とする。ext3 ファイルシステムには 3 種類のジャーナリングモードがある⁸が、今回は DBT-3 による PostgreSQL の評価で用いた計測環境に合わせ、ordered モードについて計測を行う。

⁸各ジャーナルモードについては、man mount(8)のマニュアルページを参照のこと。

2.1.5.5 データ破壊について

diskio は各計測における条件を同一にするため、毎回書き込み速度計測用パーティションを mkfs でフォーマットしている。

計測実行の前にバックアップ等を取っておく事を推奨する。

2.1.5.6 計測時のランレベルについて

同期書き込み実行にかかる時間を正確に計測するためには、計測に必要な以外の I/O を、可能な限り排除する必要がある。そのため、計測時はシングルユーザモードで起動するのが望ましい。

2.1.6 ツールのインストール

2.1.6.1 ソースコードの展開

diskio-2.0.tar.gz を格納したディレクトリを\$(DIR_DISKIO)とする。

まず、tar コマンドを用いて、diskio-2.0 を展開する。

```
$ cd $(DIR_DISKIO)
$ tar zxvf diskio-2.0.tar.gz
```

今回は\$(DIR_DISKIO)は/root/diskio とし、以下のように実行した。

```
$ cd /root/diskio
$ tar zxvf diskio-2.0.tar.gz
```

2.1.6.2 ソースコードのビルド

ソースコードのディレクトリに移動し、ビルドする。

```
$ cd $(DIR_DISKIO)/diskio-2.0/src
$ make
```

今回は以下のように実行した。

```
$ cd /root/diskio/diskio-2.0/src
$ make
```

2.1.7 ツールの設定

diskio を駆動するスクリプトは OProfile を使用するものと使用しないものの 2 種類がある。手順にほとんど変更はないが、使用するカーネルが違いため、OProfile を使用する時に手順が異なる場合は、OProfile 使用時として、併記する。

2.1.7.1 計測ディレクトリの準備

駆動スクリプト、ログを置くディレクトリを作成する。

```
$ cd $(DIR_DISKIO)/diskio-2.0/  
$ mkdir run
```

書き込み用マウントポイントを作成する。

```
$ mkdir run/test
```

今回は以下のように実行した。

```
$ cd /root/diskio/diskio-2.0/  
$ mkdir run  
$ mkdir run/test
```

2.1.7.2 駆動スクリプトの編集

diskio を駆動するスクリプトは \$(DIR_DISKIO) に格納してある。

スクリプトは 2 種類あり、OProfile を使用しない "list" と OProfile を使用する "list_oprofile" がある。

駆動スクリプトを移動する。

```
$ cp $(DIR_DISKIO)/diskio-2.0/list $(DIR_DISKIO)/diskio-2.0/run
```

OProfile 使用時は以下のようにする。

```
$ cp $(DIR_DISKIO)/diskio-2.0/list_oprofile $(DIR_DISKIO)/diskio-2.0/run
```

今回は以下のように実行した。

```
$ cp /root/diskio/diskio-2.0/list /root/diskio/diskio-2.0/run
```

OProfile 使用時は以下のように実行した。

```
$ cp /root/diskio/diskio-2.0/list_oprofile /root/diskio/diskio-2.0/run
```

計測対象 HDD のデバイス名を確認する。

確認したデバイス名を \$(WRITE_DEVICE) とする。

"list" の 7 行目にある TARGET_PARTITION を
TARGET_PARTITION=\$(WRITE_DEVICE)
と設定する。

Distribution の Version 設定

計測スクリプト"list"の 22 行目にある DISTROVERSION を
DISTROVERSION="ディストリビューション名"-**"カーネルバージョン"**
のような書式で設定する。
設定した version を\$(DISTROVERSION)とする。

今回は、\$(WRITE_DEVICE)を"/dev/sdb1"とし、\$(DISTROVERSION)
を"**Miracle3.0-2.4.21-9.38AX**"としたため、list スクリプトは以下ようになった。

```
#!/bin/bash

#####
# Copyright (c) 2005 NTT DATA CORPORATION
#####

TARGET_PARTITION=/dev/sdb1
MOUNTPPOINT=./test
TESTFILE="$MOUNTPPOINT/testfile"
RUNPATH="../bin/"

#Each Write Size of PostgreSQL Transaction Log Writing
WRITESIZE=8192
#File size is 48MB(Three times of default log file size.)
SAMPLINGFREQ=6144
STARTLOGGING=0
ENDLOGGING=$SAMPLINGFREQ

INTERLEAVESIZE=8192
SEEKRANGE=6144

DISTROVERSION="Miracle3.0-2.4.21-9.38AX"
LOGFNAMEHEAD=$DISTROVERSION

TARGETFS='ext3-dj ext3-do ext3-dw'

tryfunc2 ()
{
    runfunc=$1;
    create=$2;
    fsname=$3;
```

```

echo "runfunc = $runfunc"
echo "create = $create"
echo "fsname = $fsname"

if [ -n "$LOGFNAMEHEAD" ] && [ ! -d "$LOGFNAMEHEAD" ]; then
    mkdir "$LOGFNAMEHEAD";
fi
if [ -n "$fsname" ] && [ ! -d "$LOGFNAMEHEAD/$fsname" ]; then
    mkdir "$LOGFNAMEHEAD/$fsname";
fi
runfunc2=`echo "$runfunc" | sed 's/¥///g'`
logfilename=`echo "$LOGFNAMEHEAD/$fsname/$runfunc2.log"`

echo > $logfilename;    # clear log file

case $fsname in
'ext2')
    echo mke2fs $TARGET_PARTITION >> $logfilename 2>&1;
    mke2fs $TARGET_PARTITION >> $logfilename 2>&1;
    MOUNTOPTION='';
    ;;
'ext3-dj')
    echo mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1;
    mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1;
    MOUNTOPTION='-odata=journal';
    ;;
'ext3-do')
    echo mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1;
    mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1;
    MOUNTOPTION='-odata=ordered';
    ;;
'ext3-dw')
    echo mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1;
    mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1;
    MOUNTOPTION='-odata=writeback';
    ;;

```

```

'rfs')
    echo mkreiserfs -f -f $TARGET_PARTITION >> $logfile 2>&1
    mkreiserfs -f -f $TARGET_PARTITION >> $logfile 2>&1
    MOUNTOPTION='';
    ;;
'rfs-nl')
    echo mkreiserfs -f -f $TARGET_PARTITION >> $logfile 2>&1
    mkreiserfs -f -f $TARGET_PARTITION >> $logfile 2>&1
    MOUNTOPTION='-onolog';
    ;;

'xfs')
    echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    MOUNTOPTION='';
    ;;
'xfs-nat')
    echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    MOUNTOPTION='-onoatime';
    ;;
'xfs-osids')
    echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    MOUNTOPTION='-oosyncisdsync';
    ;;
'xfs-nat-osids')
    echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    mkfs.xfs -f -q $TARGET_PARTITION >> $logfile 2>&1
    MOUNTOPTION='-onoatime,osyncisdsync';
    ;;

'jfs')
    echo 'could not find filesystem formatter for JFS' >> $logfile
2>&1

    exit -1;
    ;;
esac

if [ "$create"='create' ]; then

```

```

        echo >> $logfile;
        echo 'create file' >> $logfile

        echo mount $MOUNTOPTION $TARGET_PARTITION $MOUNTPOINT >>
logfile 2>&1
        mount $MOUNTOPTION $TARGET_PARTITION $MOUNTPOINT >> $logfile
2>&1
        if [ $PIPESTATUS != 0 ]; then
            exit -1;
        fi
        echo '' >> $logfile 2>&1

        reqfilesize=$(( WRITESIZE * SAMPLINGFREQ ));
        echo 'required file size = ' $reqfilesize >> $logfile 2>&1

        echo "head -c $reqfilesize /dev/zero" '>$TESTFILE' >> $logfile
2>&1
        head -c $reqfilesize /dev/zero > $TESTFILE

        echo umount $MOUNTPOINT >> $logfile 2>&1
        umount $MOUNTPOINT >> $logfile 2>&1
    fi

    echo >> $logfile
    echo mount $MOUNTOPTION $TARGET_PARTITION $MOUNTPOINT >> $logfile 2>&1
    mount $MOUNTOPTION $TARGET_PARTITION $MOUNTPOINT >> $logfile 2>&1
    if [ $PIPESTATUS != 0 ]; then
        exit -1;
    fi

    echo >> $logfile
    echo "/proc/cpuinfo" >> $logfile;
    cat /proc/cpuinfo >> $logfile;
    echo >> $logfile
    echo "/proc/version" >> $logfile;
    cat /proc/version >> $logfile;
    echo >> $logfile
    echo "/proc/meminfo" >> $logfile;
    cat /proc/meminfo >> $logfile;

```

```

echo >> $logfile
echo "/proc/scsi/scsi" >> $logfile;
cat /proc/scsi/scsi >> $logfile;
echo >> $logfile
echo 'TARGET_PARTITION = ' $TARGET_PARTITION >> $logfile;
echo 'FILE SYSTEM = ' $fsname >> $logfile;
echo 'TEST TYPE = ' $runfunc >> $logfile;
echo 'WRITESIZE = ' $WRITESIZE >> $logfile;
echo 'SAMPLINGFREQ = ' $SAMPLINGFREQ >> $logfile;
echo 'STARTLOGGING = ' $STARTLOGGING >> $logfile;
echo 'ENDLOGGING = ' $ENDLOGGING >> $logfile;

echo "" >> $logfile;

echo $RUNPATH/$runfunc -F $TESTFILE -w $WRITESIZE -f $SAMPLINGFREQ -s
$STARTLOGGING -e $ENDLOGGING -i $INTERLEAVESIZE -S $SEEK RANGE >> $logfile;
$RUNPATH/$runfunc -F $TESTFILE -w $WRITESIZE -f $SAMPLINGFREQ -s
$STARTLOGGING -e $ENDLOGGING -i $INTERLEAVESIZE -S $SEEK RANGE >> $logfile
2>&1;

echo umount $MOUNTPOINT >> $logfile 2>&1
umount $MOUNTPOINT
if [ $PIPESTATUS != 0 ]; then
    exit -1;
fi
}

tryfunc ()
{
    runfunc=$1;
    createfile=$2
    for fsname in $TARGETFS ; do
        tryfunc2 $runfunc $createfile $fsname ;
    done
}

if [ ! -d "$MOUNTPOINT" ]; then
    mkdir $MOUNTPOINT;

```

```
fi
```

```
for runfunc in ¥
```

```
    open-noopt-noopt-async_fdatasync-interleave¥  
    open-noopt-noopt-async_fdatasync-sequential¥  
    open-noopt-noopt-async_fdatasync-random¥  
    open-noopt-open_L-async_fdatasync-interleave¥  
    open-noopt-open_L-async_fdatasync-sequential¥  
    open-noopt-open_L-async_fdatasync-random¥  
    open-noopt-noopt-async_fsync-interleave¥  
    open-noopt-noopt-async_fsync-sequential¥  
    open-noopt-noopt-async_fsync-random¥  
    open-noopt-open_L-async_fsync-interleave¥  
    open-noopt-open_L-async_fsync-sequential¥  
    open-noopt-open_L-async_fsync-random¥  
    open-noopt-noopt-async_syncsync-interleave¥  
    open-noopt-noopt-async_syncsync-sequential¥  
    open-noopt-noopt-async_syncsync-random¥  
    open-noopt-open_L-async_syncsync-interleave¥  
    open-noopt-open_L-async_syncsync-sequential¥  
    open-noopt-open_L-async_syncsync-random¥  
    open-noopt-noopt-async_write-interleave¥  
    open-noopt-noopt-async_write-sequential¥  
    open-noopt-noopt-async_write-random¥  
    open-noopt-open_L-async_write-interleave¥  
    open-noopt-open_L-async_write-sequential¥  
    open-noopt-open_L-async_write-random¥  
    open-noopt-noopt-odsync_write-interleave¥  
    open-noopt-noopt-odsync_write-sequential¥  
    open-noopt-noopt-odsync_write-random¥  
    open-noopt-open_L-odsync_write-interleave¥  
    open-noopt-open_L-odsync_write-sequential¥  
    open-noopt-open_L-odsync_write-random¥  
    open-noopt-noopt-osync_write-interleave¥  
    open-noopt-noopt-osync_write-sequential¥  
    open-noopt-noopt-osync_write-random¥  
    open-noopt-open_L-osync_write-interleave¥  
    open-noopt-open_L-osync_write-sequential¥  
    open-noopt-open_L-osync_write-random¥
```

```
; do
```

```

    tryfunc $runfunc "create";
done

for runfunc in ¥
    open-open_C-noopt-async_fdatasync-interleave¥
    open-open_C-noopt-async_fdatasync-sequential¥
    open-open_C-open_L-async_fdatasync-interleave¥
    open-open_C-open_L-async_fdatasync-sequential¥
    open-open_C-noopt-async_fsync-interleave¥
    open-open_C-noopt-async_fsync-sequential¥
    open-open_C-open_L-async_fsync-interleave¥
    open-open_C-open_L-async_fsync-sequential¥
    open-open_C-noopt-async_syncsync-interleave¥
    open-open_C-noopt-async_syncsync-sequential¥
    open-open_C-open_L-async_syncsync-interleave¥
    open-open_C-open_L-async_syncsync-sequential¥
    open-open_C-noopt-async_write-interleave¥
    open-open_C-noopt-async_write-sequential¥
    open-open_C-open_L-async_write-interleave¥
    open-open_C-open_L-async_write-sequential¥
    open-open_C-noopt-odsync_write-interleave¥
    open-open_C-noopt-odsync_write-sequential¥
    open-open_C-open_L-odsync_write-interleave¥
    open-open_C-open_L-odsync_write-sequential¥
    open-open_C-noopt-osync_write-interleave¥
    open-open_C-noopt-osync_write-sequential¥
    open-open_C-open_L-osync_write-interleave¥
    open-open_C-open_L-osync_write-sequential¥
; do
    tryfunc $runfunc "ncreate";
done

echo "Finished."

```

OProfile 使用時は \$(WRITE_DEVICE) を "/dev/sdb1"、\$(DISTROVERSION) を "Miracle3.0-2.4.21-9.38AXsmp" としたため、"list_oprofile" は以下ようになった。

```
#!/bin/bash

#####
# Copyright (c) 2005 NTT DATA CORPORATION
#####

TARGET_PARTITION=/dev/sdb1
MOUNTPOINT=./test
TESTFILE="$MOUNTPOINT/testfile"
RUNPATH="../bin/"

#Each Write Size of PostgreSQL Transaction Log Writing
WRITESIZE=8192
#File size is 48MB(Three times of default log file size.)
SAMPLINGFREQ=6144
STARTLOGGING=0
ENDLOGGING=$SAMPLINGFREQ

INTERLEAVESIZE=8192
SEEKRANGE=6144

DISTROVERSION="Miracle3.0-2.4.21-9.38AX"
LOGFNAMEHEAD=$DISTROVERSION

TARGETFS='ext3-dj ext3-do ext3-dw'

tryfunc2 ()
{
    runfunc=$1;
    create=$2;
    fsname=$3;

    echo "runfunc = $runfunc"
    echo "create = $create"
    echo "fsname = $fsname"

    if [ -n "$LOGFNAMEHEAD" ] && [ ! -d "$LOGFNAMEHEAD" ]; then
```

```

        mkdir "$LOGNAMEHEAD";
    fi
    if [ -n "$fsname" ] && [ ! -d "$LOGNAMEHEAD/$fsname" ]; then
        mkdir "$LOGNAMEHEAD/$fsname";
    fi
    runfunc2=`echo "$runfunc" | sed 's/¥///g'`
    logfilename=`echo "$LOGNAMEHEAD/$fsname/$runfunc2.log"`

    echo > $logfilename;    # clear log file

    case $fsname in
    'ext2')
        echo mke2fs $TARGET_PARTITION >> $logfilename 2>&1;
        mke2fs $TARGET_PARTITION >> $logfilename 2>&1;
        MOUNTOPTION='';
        ;;

    'ext3-dj')
        echo mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1
        mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1
        MOUNTOPTION='-odata=journal';
        ;;

    'ext3-do')
        echo mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1
        mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1
        MOUNTOPTION='-odata=ordered';
        ;;

    'ext3-dw')
        echo mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1
        mke2fs -j $TARGET_PARTITION >> $logfilename 2>&1
        MOUNTOPTION='-odata=writeback';
        ;;

    'rfs')
        echo mkreiserfs -f -f $TARGET_PARTITION >> $logfilename 2>&1
        mkreiserfs -f -f $TARGET_PARTITION >> $logfilename 2>&1
        MOUNTOPTION='';
        ;;

    'rfs-nl')
        echo mkreiserfs -f -f $TARGET_PARTITION >> $logfilename 2>&1

```

```

mkreiserfs -f -f $TARGET_PARTITION >> $logfilename 2>&1
MOUNTOPTION='-onolog' ;
;;

' xfs' )
echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
MOUNTOPTION=' ' ;
;;

' xfs-nat' )
echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
MOUNTOPTION='-onoatime' ;
;;

' xfs-osids' )
echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
MOUNTOPTION='-oosyncisdsync' ;
;;

' xfs-nat-osids' )
echo mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
mkfs.xfs -f -q $TARGET_PARTITION >> $logfilename 2>&1
MOUNTOPTION='-onoatime,osyncisdsync' ;
;;

' jfs' )
echo 'could not find filesystem formatter for JFS' >> $logfilename
2>&1

exit -1;
;;

esac

if [ "$create"='create' ]; then
echo >> $logfilename;
echo 'create file' >> $logfilename

echo mount $MOUNTOPTION $TARGET_PARTITION $MOUNTPOINT >>
$logfilename 2>&1
mount $MUONTOPTION $TARGET_PARTITION $MOUNTPOINT >> $logfilename
2>&1

```

```

        if [ $PIPESTATUS != 0 ]; then
            exit -1;
        fi
        echo '' >> $logfile 2>&1

        reqfilesize=$(( WRITESIZE * SAMPLINGFREQ ));
        echo 'required file size = ' $reqfilesize >> $logfile 2>&1

        echo "head -c $reqfilesize /dev/zero" '$TESTFILE' >> $logfile
2>&1
        head -c $reqfilesize /dev/zero > $TESTFILE

        echo umount $MOUNTPPOINT >> $logfile 2>&1
        umount $MOUNTPPOINT >> $logfile 2>&1
    fi

    echo >> $logfile
    echo mount $MOUNTOPTION $TARGET_PARTITION $MOUNTPPOINT >> $logfile 2>&1
    mount $MOUNTOPTION $TARGET_PARTITION $MOUNTPPOINT >> $logfile 2>&1
    if [ $PIPESTATUS != 0 ]; then
        exit -1;
    fi

    echo >> $logfile
    echo "/proc/cpuinfo" >> $logfile;
    cat /proc/cpuinfo >> $logfile;
    echo >> $logfile
    echo "/proc/version" >> $logfile;
    cat /proc/version >> $logfile;
    echo >> $logfile
    echo "/proc/meminfo" >> $logfile;
    cat /proc/meminfo >> $logfile;
    echo >> $logfile
    echo "/proc/scsi/scsi" >> $logfile;
    cat /proc/scsi/scsi >> $logfile;
    echo >> $logfile
    echo 'TARGET_PARTITION = ' $TARGET_PARTITION >> $logfile;
    echo 'FILE SYSTEM = ' $fsname >> $logfile;
    echo 'TEST TYPE = ' $runfunc >> $logfile;
    echo 'WRITESIZE = ' $WRITESIZE >> $logfile;

```

```

echo ' SAMPLINGFREQ      = ' $SAMPLINGFREQ      >> $logfile;
echo ' STARTLOGGING      = ' $STARTLOGGING      >> $logfile;
echo ' ENDLOGGING        = ' $ENDLOGGING        >> $logfile;

echo "" >> $logfile;

echo "opcontrol --vmlinux=/usr/lib/debug/boot/vmlinux-`uname -r`.debug"
opcontrol --vmlinux=/usr/lib/debug/boot/vmlinux-`uname -r`.debug >>
$logfilename
echo "opcontrol --reset"
opcontrol --reset >> $logfile
echo "opcontrol --start"
opcontrol --start >> $logfile
echo "" >> $logfile

echo $RUNPATH/$runfunc -F $TESTFILE -w $WRITESIZE -f $SAMPLINGFREQ -s
$STARTLOGGING -e $ENDLOGGING -i $INTERLEAVESIZE -S $SEEK RANGE >> $logfile;
time $RUNPATH/$runfunc -F $TESTFILE -w $WRITESIZE -f $SAMPLINGFREQ -s
$STARTLOGGING -e $ENDLOGGING -i $INTERLEAVESIZE -S $SEEK RANGE >> $logfile
2>&1;

echo "opcontrol --save=diskio-$logfile"
opcontrol --save=diskio-$logfile >> $logfile
echo "opcontrol --shutdown"
opcontrol --shutdown >> $logfile

echo umount $MOUNTPOINT >> $logfile 2>&1
umount $MOUNTPOINT
if [ $PIPESTATUS != 0 ]; then
    exit -1;
fi
}

tryfunc ()
{
    runfunc=$1;
    createfile=$2
    for fsname in $TARGETFS ; do
        tryfunc2 $runfunc $createfile $fsname ;
    done
}

```

```

if [ ! -d "$MOUNTPOINT" ]; then
    mkdir $MOUNTPOINT;
fi

for runfunc in ¥
    open-noopt-noopt-async_fdatasync-interleave¥
    open-noopt-noopt-async_fdatasync-sequential¥
    open-noopt-noopt-async_fdatasync-random¥
    open-noopt-open_L-async_fdatasync-interleave¥
    open-noopt-open_L-async_fdatasync-sequential¥
    open-noopt-open_L-async_fdatasync-random¥
    open-noopt-noopt-async_fsync-interleave¥
    open-noopt-noopt-async_fsync-sequential¥
    open-noopt-noopt-async_fsync-random¥
    open-noopt-open_L-async_fsync-interleave¥
    open-noopt-open_L-async_fsync-sequential¥
    open-noopt-open_L-async_fsync-random¥
    open-noopt-noopt-async_syncsync-interleave¥
    open-noopt-noopt-async_syncsync-sequential¥
    open-noopt-noopt-async_syncsync-random¥
    open-noopt-open_L-async_syncsync-interleave¥
    open-noopt-open_L-async_syncsync-sequential¥
    open-noopt-open_L-async_syncsync-random¥
    open-noopt-noopt-async_write-interleave¥
    open-noopt-noopt-async_write-sequential¥
    open-noopt-noopt-async_write-random¥
    open-noopt-open_L-async_write-interleave¥
    open-noopt-open_L-async_write-sequential¥
    open-noopt-open_L-async_write-random¥
    open-noopt-noopt-odsync_write-interleave¥
    open-noopt-noopt-odsync_write-sequential¥
    open-noopt-noopt-odsync_write-random¥
    open-noopt-open_L-odsync_write-interleave¥
    open-noopt-open_L-odsync_write-sequential¥
    open-noopt-open_L-odsync_write-random¥
    open-noopt-noopt-osync_write-interleave¥
    open-noopt-noopt-osync_write-sequential¥
    open-noopt-noopt-osync_write-random¥
    open-noopt-open_L-osync_write-interleave¥

```

```

open-noopt-open_L-osync_write-sequential¥
open-noopt-open_L-osync_write-random¥
; do
    tryfunc $runfunc "create";
done

for runfunc in ¥
    open-open_C-noopt-async_fdatasync-interleave¥
    open-open_C-noopt-async_fdatasync-sequential¥
    open-open_C-open_L-async_fdatasync-interleave¥
    open-open_C-open_L-async_fdatasync-sequential¥
    open-open_C-noopt-async_fsync-interleave¥
    open-open_C-noopt-async_fsync-sequential¥
    open-open_C-open_L-async_fsync-interleave¥
    open-open_C-open_L-async_fsync-sequential¥
    open-open_C-noopt-async_syncsync-interleave¥
    open-open_C-noopt-async_syncsync-sequential¥
    open-open_C-open_L-async_syncsync-interleave¥
    open-open_C-open_L-async_syncsync-sequential¥
    open-open_C-noopt-async_write-interleave¥
    open-open_C-noopt-async_write-sequential¥
    open-open_C-open_L-async_write-interleave¥
    open-open_C-open_L-async_write-sequential¥
    open-open_C-noopt-odsync_write-interleave¥
    open-open_C-noopt-odsync_write-sequential¥
    open-open_C-open_L-odsync_write-interleave¥
    open-open_C-open_L-odsync_write-sequential¥
    open-open_C-noopt-osync_write-interleave¥
    open-open_C-noopt-osync_write-sequential¥
    open-open_C-open_L-osync_write-interleave¥
    open-open_C-open_L-osync_write-sequential¥
; do
    tryfunc $runfunc "ncreate";
done

echo "Finished."

```

2.2 OS 層の限界値ベンチマーク

2.2.1 ツールの説明

2.2.1.1 *iozone*

iozone は *iozone* コマンド単体で実行可能で、これにオプションを指定することで様々な種類の I/O を再現できる。キットに付属したドキュメントは tarball を展開したディレクトリ `/root/os-bench/docs/` に格納されており、また公式サイト <http://www.iozone.org/> でも参照することができる。

iozone はファイルシステムのベンチマークツールであり、ファイル操作に関する様々なパフォーマンスを測定する。また Linux だけではなく他の多くのプラットフォームに対応しているため、プラットフォーム間の I/O 性能の比較・検証にも適している。

iozone が対応している主要な機能は付属ドキュメントによると以下の通り。

- POSIX async I/O
- `mmap()` ファイル I/O
- 通常のファイル I/O
- 単一 stream 測定
- 複数 stream 測定
- POSIX スレッド
- 複数プロセス測定
- Excel フォーマット出力
- I/O レイテンシデータ
- 64 ビットシステム対応
- 巨大ファイル対応

スループットテスト時の並列処理を保証する Stonewalling 機能

- プロセッサのキャッシュサイズ設定
- `fsync`, `O_SYNC` 選択
- NFS 経由テスト対応

iozone は、レコードサイズと呼ばれるサイズ単位で一回のシステムコールを実行し、これを繰り返すことで指定されたサイズのファイルの作成や読み込みを行う。動作モードには 2 種類あり、ひとつは自動モード、もうひとつはスループットモードと呼ばれる。

自動モード (`-a` オプション) では、ファイルサイズとレコードサイズを指定された範囲で変化させながら、(ほぼ)すべてのパターンを自動的に測定する。同時に実施

するのは1個のI/O処理である。

それに対して、スループットモード(-t オプション)では、指定されたI/Oパターン(initial write や read strided など)で、指定されたファイルサイズ、指定されたレコードサイズの測定を実行し、また同時に複数のI/Oを実行することができる。同時に実行されるI/Oの数(stream数と呼ぶ)は-t オプションの引数で指定する。

始めに起動されたiozone プロセスは、実際のI/O処理を子プロセスまたはスレッドで実行させる。子プロセスとスレッドのどちらにするかは-T オプションで指定することができる。

ファイルサイズもレコードサイズ(1回のシステムコールで書き込むデータ量)もオプションで任意に設定可能である。

表 2.2-1 オプションで指定できる各サイズおよびプロセス数

	固定	可変	
		最小	最大
ファイルサイズ	-s	-n	-g
レコードサイズ	-r	-y	-q
プロセス数	-t	-l	-u

測定できる種類とiozone コマンドのパラメータそれぞれの一覧を表 2.2-2 と表 2.2-3 に示す。

表 2.2-2 測定できるI/Oの種類

タイプ	ファイル操作内容
write	新規にファイルを作成する
re-write	既存のファイルに上書きの書き込みを行う
read	既存のファイルを読み込む
re-read	最近読み込んだファイルを再び読み込む
random read	ファイルの中のランダムな位置を読み込む
random write	既存ファイルの中のランダムな位置に書き込みを行う
random mix	既存ファイルのランダムな位置に読み込みと書き込みを行う
read backwards	ファイルを逆方向(末尾から先頭への方向)に読み込む
record rewrite	既存ファイルの中の特定な位置を上書きする
read strided	ファイルの中を特定サイズの読み込みと、特定サイズのシークとを繰り返す
fwrite/frewrite	write()の代わりにfwrite()を使ったwriteおよびre-write
fread/freread	read()の代わりにfread()を使ったreadおよびre-read
pread/pwrite	pread()/pwrite()を使用したファイル操作

aio_read/aio_write	POSIX async I/O
mmap	mmap()を使用したファイル操作

表 2.2-3 iozone コマンドのオプションパラメーター一覧

-a	全自動モード。レコードサイズ：4k-16M。ファイルサイズ：64k-512M。
-A	省略無しの全自動モード（将来廃止予定）
-b filename	Excel ワークシート形式のファイルを出力
-B	mmap()を使用
-c	計測時間に close()処理も含める
-C	スループットテスト時に子プロセスごとの転送バイトを表示
-d #	スループットテスト時に子プロセスの開始時間の間隔を設定する
-D	mmap ファイルに msync(MS_ASYNC)を使用する
-e	計測時間に fsync, fflush を含める
-E	pread/pwrite を使った拡張テストを実施する
-f filename	一時ファイルとして使用するファイル名を指定する
-F filename filename ...	スループットテスト時に一時ファイルとして使用するファイル名（子プロセス/スレッド数分）を指定する。
-g #	自動モードでのファイルの最大サイズ（Kbytes 単位）を指定する
-G	mmap ファイルに msync(MS_SYNC)を使用する
-h	ヘルプを表示する
-H #	POSIX async I/O を使用する
-i #	実行するテストを指定する。0:write/read, 1:read/re-read, 2:random-read/write, 3:read-backwards, 4:re-write-record, 5:stride-read, 6:fwrite/re-fwrite, 7:freadd/re-freadd, 8:random mix, 9:write/re-pwrite, 10:pread/re-preadd, 11:pwritev/re-pwritev, 12:preadv/re-preadv
-l	すべてのファイル操作に VxFS VX_DIRECT を指定する。
-j #	Stride アクセス時の stride 幅を指定する
-J #	I/O 操作開始前に待ち時間（ミリ秒単位）を設定する
-k #	POSIX async I/O (bcopy なし)を使用する
-K	通常テストの間に攪乱目的のランダムな読み込み処理を行う
-l #	実行するプロセス数の下限を設定する
-L #	プロセッサのキャッシュラインサイズを指定する（Byte 単位）

-m	内部バッファを複数個使用する（デフォルトは1個を再利用）
-M	uname -a の実行結果を出力する
-n #	自動モードでのファイルの最小サイズ（Kbytes 単位）を設定する
-N	ファイル操作ごとにミリ秒単位で結果を出力する
-o	同期書き込み（O_SYNC）を行う
-O	ファイル操作ごとに秒単位で結果を出力する
-p	ファイル操作ごとにプロセッサのキャッシュを破棄する
-P #	指定した番号のプロセッサにプロセス/スレッドを固定する
-q #	自動モードでのレコードの最大サイズ（Kbyte 単位）を指定する
-Q	オフセットとアクセス遅延との関連データをファイルに出力する
-r #	レコードサイズ（Kbyte 単位）を指定する
-R	Excel 形式のデータを標準出力に出力する
-s #{k, K, m, M, g, G}	テストするファイルのサイズを指定する。k,K:Kbytes, m,M:Mbytes, g,G:Gbytes
-S #	プロセッサのキャッシュサイズ（Kbyte 単位）を指定する
-t #	スループットモードで実行する。子プロセスまたはスレッド。
-T	スループットモード時に子プロセスではなく POSIX pthread を使用する
-u #	実行するプロセス数の上限を設定する
-U mountpoint	テストとテストの間にアンマウント、再マウントを実施する
-v	Iozone のバージョンを表示する
-V #	データ検証（write データと read データとの照合）を実施する
-w	終了時に一時ファイルを消去せずに残す
-W	ファイルの read/write 時にロックする
-x	Stone-walling を無効にする
-X filename	write 用のテレメトリ情報を設定する
-y #	自動モードでのレコードサイズの下限（Kbyte 単位）を指定する
-Y filename	read 用のテレメトリ情報を設定する
-z	-a とともに使用する。すべてのレコードサイズをテストする
-Z	mmap I/O と通常のファイル I/O をミックスする
-+m cluster_filename	クラスタテストを有効にする
-+d	ファイル I/O 診断モード
-+u	CPU 使用率を出力する
-+x #	ファイルサイズとレコードサイズを増加させる倍率を設定する
-+p #	ミックステスト時の read の割合を設定する
-+r	同期書き込み（O_RSYNC O_SYNC）を行う

-+t	ネットワークのパフォーマンステストを実施する
-+n	Retest を実施しない (re-read, re-write 等)
-+k	ファイルサイズを子プロセス/スレッドで均等に配分する。デフォルトではそれぞれが指定ファイルサイズでテストする
-+q #	テストごとに時間間隔 (秒単位) をあける
-+l	レコード単位でロックする
-+L	一時ファイルを共有し、かつレコード単位でロックする
-+B	シーケンシャルにミックスマードを実行する。デフォルトはランダム
-+D	同期書き込み (O_DSYNC) を行う
-+A #	mmap I/O に対して madvice() を実施する

2.2.1.2 OProfile

OProfile は、Linux 2.2/2.4/2.6 システムに対応したプロファイリングツールであり、モジュールや割り込みハンドラも含んだカーネルから共有ライブラリや通常アプリケーションにいたる、システムのすべてをプロファイリングする能力を持っている。

プロファイリングは、様々なイベントの発生時点における PC (Program Counter) の値をサンプリングすることによって、その瞬間にどのバイナリ中のどの箇所が CPU によって実行中であったかという情報を記録する。この情報の蓄積を統計的に処理することによって、システム全体の中で一体どこの処理に偏っているのか、どこの処理に集中しているのかを把握し、例えばパフォーマンス向上を検討する材料となりえる。

サンプリングのトリガーとして設定できるイベントは CPU の種類によって異なるが、CPU 稼働時間のイベント (タイマーに相当する) はどの CPU でも利用できる。インテル® Pentium®III 系では CPU_CLK_UNHALTED、Pentium®4 系では GLOBAL_POWER_EVENTS と呼ばれている。今回の評価ではこのイベントを使用し、等時間間隔でのサンプリングデータを採取し、これを分析することで、システムのボトルネックと考えられる箇所の特定を試みた。

OProfile の公式ページ <http://oprofile.sourceforge.net/> にはソースコードだけでなく、ドキュメントや FAQ が掲載されているので参考にされたい。

MIRACLE LINUX V3.0 において OProfile を利用するには、oprofile-0.5.4-22.i386.rpm パッケージに含まれている各コマンドを実行する。今回の検証に使用した 3 つのコマンドを以下に紹介する。なお、OProfile がサンプリングしたデータはディレクトリ/var/lib/oprofile/samples に格納される。

/usr/bin/opcontrol

機能：

各種設定変更や起動・停止といった OProfile のコントロールを行う。

オプション：

--vmlinux=<kernel file>	デバッグ情報入りカーネルファイルを指定する
--reset	サンプリングしたデータを削除する
--ctr0-count=<number>	サンプリング頻度を指定する
--start	プロファイリングを開始する
--shutdown	プロファイリングを終了する

<code>--save=<session name></code>	サンプリングしたデータを別名で保存する
--	---------------------

使用例 :

```
# opcontrol --vmlinux=/usr/lib/debug/boot/vmlinux-$(uname -r).debug
# opcontrol --reset
# opcontrol --start
```

`/usr/bin/oprofpp`

機能 :

サンプリングしたデータから、実際に解析可能なデータを抽出する。

オプション :

<code>-i <kernel file></code>	デバッグ情報入りカーネルファイルを指定する
<code>-l</code>	シンボルごとのサンプル数を一覧表にして表示する
<code>-L</code>	シンボルを細分化した詳細な表を出力する
<code>-o</code>	すべてのサンプルに対して関数名と(アセンブリ言語での)行数を表示する
<code>-r</code>	サンプル数が多い順に表示する
<code>-t <format></code>	表の形式を指定する

使用例

```
# oprofpp -i /usr/lib/debug/boot/vmlinux-$(uname -r).debug -l -r
```

`/usr/bin/op_to_source`

機能 :

サンプリングデータが記載されたソースファイルを出力する。

オプション :

<code>-i <kernel file></code>	デバッグ情報入りカーネルファイルを指定する
<code>--output-dir <directory></code>	ソースファイルの出力ディレクトリを指定する

使用例 :

```
# op_to_source -i /usr/lib/debug/boot/vmlinux-$(uname -r).debug \
--output-dir ./data
```

2.2.1.3 LKST

拡張版 LKST については、LKST の公式サイト <http://lkst.sourceforge.net/> のドキュメントを参照のこと。今回の評価でを使用したコマンドや手順は、次章の評価手順で紹介する。

2.2.2 ハードウェア構成

大規模メモリシステム環境として今回使用したハードウェア構成は表 2.2-4 の通り。

表 2.2-4 ハードウェア構成

モデル	Dell PowerEdge 2600 ⁹						
プロセッサ	Intel® Xeon™ 2.0GHz x 2 基 (HT 機能により論理的に 4 基) [family:15, model:2, stepping:7, cache size:512KB]						
メインメモリ	6GB						
チップセット	Intel® E7500, 82870P2 P64H2, 82801CA						
ディスク コントローラ	LSI Logic 53C1030 + PERC 4/Di (ただし、RAID キーを取り外して RAID 機能を無効にした)						
ディスク ドライブ	ID	ベンダ	モデル	容量	回転数	キャッシュ	インタフェース
	0	Maxtor	ATLAS10K4	36G	10krpm	8MB	U320
	1	Seagate	ST336607LC	36G	10krpm	8MB	U320
	2	Seagate	ST336607LC	36G	10krpm	8MB	U320
	3	富士通	MAN3184MC	18G	10krpm	8MB	U160
	4	富士通	MAN3184MC	18G	10krpm	8MB	U160
	5	Seagate	ST318404LC	18G	10krpm	4MB	U160

以降で紹介する環境定義は、この CPU4 基、メモリ 6GB、ディスク 6 台のマシン構成に合わせたものである。本書の手順に従い同等の測定を再現するにあたり、もしも異なるマシン構成上にて実施する場合には、以下で紹介するファイルシステム設定や Iozone のテストパターンなどを変更する必要がある。変更方法については文中に記載している。また、OProfile は up(ユニプロセッサ)カーネルをサポートしていないため、単一 CPU 環境においても smp カーネルを使用する必要がある。

また本書においては、特に断りがない場合はすべて root ユーザでのオペレーション (コマンド実行等) とする。

⁹ http://www1.us.dell.com/content/products/productdetails.aspx/pedge_2600

2.2.3 OS のインストール

OS には MIRACLE LINUX V3.0 – Asianux Inside を使用する。また、カーネルについては 2004 年 12 月時点で最新の 2.4.21-9.38AX カーネルを使用する。

インストールの手順は、製品付属の『インストレーションガイド』¹⁰に従う。インストール中のオプション選択は表 2.2-5 の通りで、これ以外はすべてデフォルト状態とする。

表 2.2-5 インストールの選択オプション

言語	日本語
パーティション	表 2.2-6 参照
ブートローダ	MBR
タイムゾーン	東京
パッケージ選択	すべて
ログインの種類	グラフィカル

パーティション構成は表 2.2-6 の通り。今回は論理 4CPU システムで最大 4 並列の I/O 処理を実行するため、I/O 用のパーティションを 4 個作成した。これらはそれぞれ単独のディスクで構成し OS 用の 1 台とあわせて合計 5 台のディスクを使用した。よって搭載ディスクのうち 1 台は未使用で、これは予備とした。

表 2.2-6 ファイルシステム構成

パーティション	サイズ	マウント位置
/dev/sda1	100MB 以上	/boot
/dev/sda2	4GB 以上	/
/dev/sda3	2GB	スワップ
/dev/sdb1	8GB 以上	/test/f1
/dev/sdc1	8GB 以上	/test/f2
/dev/sdd1	8GB 以上	/test/f3
/dev/sde1	8GB 以上	/test/f4
/dev/sdf1	予備	未使用

¹⁰ http://www.miraclelinux.com/products/linux/ml30/pdf/installation_guide.pdf

2.2.4 OS の設定

OS のインストールの次に、今回の評価作業に必要な OS 設定を行う。サービスの設定、アップデートカーネルのインストールおよび、改良版 LKST 搭載カーネルのインストールを以下の手順で実施する。

2.2.4.1 サービスの設定

サービスは基本的にデフォルト状態のまま変更しない。

ただし、リモートログインを受け付けるために sshd の設定を以下の通りに変更して sshd サービスを再起動する。今回の測定は ssh 経由の root ユーザでリモートログインして実施した。

```
# vi /etc/ssh/sshd_config
...
PermitRootLogin yes 行のコメントを外して有効にし、
PermitRootLogin no 行をコメントアウトして無効にする。
...
# service sshd restart
```

2.2.4.2 通常カーネルのインストール

アップデートカーネル kernel-smp-2.4.21-9.38AX.i686.rpm を以下の URL のうちのいずれかよりダウンロードする。

<http://ftp.miraclelinux.com/pub/Miracle/ia32/standard/3.0/updates/RPMS/>
<http://ftp.miraclelinux.com/pub/Miracle/ia32/standard/3.0/updates/RPMS/.oldrelease/>

OProfile のプロファイリングには、カーネルのデバッグ情報が必要である。そのため、このカーネルのデバッグ用カーネル kernel-debuginfo-2.4.21-9.38AX.i686.rpm を以下の URL のうちのいずれかよりダウンロードする。

<http://ftp.miraclelinux.com/pub/Miracle/ia32/standard/3.0/updates/unsupported/RPMS/>
<http://ftp.miraclelinux.com/pub/Miracle/ia32/standard/3.0/updates/unsupported/RPMS/.oldrelease/>

両パッケージを rpm -ivh コマンドでインストールする。

```
# rpm -ivh kernel-smp-2.4.21-9.38AX.i686.rpm \  
kernel-debuginfo-2.4.21-9.38AX.i686.rpm
```

2.2.4.3 改良版 LKST カーネルと改良版ツールのインストール

改良版 LKST カーネル kernel-smp-2.4.21-9.38AX.lkst10.i686.rpm とそのデバッグ用カーネル kernel-debuginfo-2.4.21-9.38AX.lkst10.i686.rpm および改良版のツール lkstutils-2.2.1-10AX.rpm を以下の URL からダウンロードする。

<http://ftp.miraclelinux.com/pub/OSSF/2004/LKST/>

但し、これらのパッケージは改良中のため、ファイル名が今後変更される可能性がある。そのため、この URL にある最新のファイルを使用する。

ダウンロードした 3 個のパッケージを rpm コマンドでインストールする。カーネルは -ivh オプションで、ツールは -Uvh オプションを使用する。

```
# rpm -ivh kernel-smp-2.4.21-9.38AX.lkst10.i686.rpm \  
kernel-debuginfo-2.4.21-9.38AX.lkst10.i686.rpm  
# rpm -Uvh lkstutils-2.2.1-10AX.rpm
```

2.2.4.4 システムのランレベル

測定時のシステムのランレベルは 5 とする。これは実運用システムに近い状態での動作を再現する狙いがある。今回の調査を掘り下げてさらに問題を追及する場合には、ランレベルを 3 または 1 に下げて環境を整えた調査をすることも考えられるが、今回の調査範囲ではランレベル 5 の状態を使用した。

2.2.5 ツールの設定

Iozone の Web サイト (<http://www.iozone.org/>) より Stable tarball ソース `iozone3_226.tar` をダウンロードし、これをビルドする。

今回の評価では `/root/os-bench` ディレクトリにダウンロードし、同ディレクトリに環境を構築する。

```
# tar xf iozone3_226.tar
# cd src/current
# make linux
# cd ../../
```

Iozone の実行プログラムが `/root/os-bench/src/current/iozone` として作成される。これをベンチマークに使用する。

3 評価手順

3.1 diskio による評価

3.1.1 注意事項

- 1) 実行時間は HW 環境に依存するが、6 時間程度と予測される。
- 2) 問題が起きた場合は、3.1.3 異常時の対処方法を参照。
- 3) OProfile 使用時に手順が異なる箇所がある。その箇所では OProfile 使用時の手順が併記してあるため、それに従うこと。

3.1.2 計測の実行

- 1) システム設定の変更

```
$ su - root
```

として、root 権限で以下を実行する。

- 1-1) fstab の設定の確認

```
# mount
```

して、書き込みに使用する HDD が現在マウントされていないことを確かめる。
マウントされていた場合は

```
# umount $(WRITE_DEVICE)
```

として、アンマウントする。

また、/etc/fstab 内に\$(WRITE_DEVICE)のマウントに関する記述がある場合は、バックアップを取り、編集する。

```
# mv /etc/fstab /etc/fstab.bak
```

```
# cp /etc/fstab.bak /etc/fstab
```

\$(WRITE_DEVICE)の行をコメントアウトする。

今回は以下のように実行した。

```
# mount
```

```
# umount /dev/sdb1
```

```
# mv /etc/fstab /etc/fstab.bak
```

```
# cp -p /etc/fstab.bak /etc/fstab
```

1-2) inittab の設定の変更

/etc/inittab のバックアップを取り、変更する。

```
# mount
# umount /dev/sdb1
# mv /etc/inittab /etc/inittab.bak
# cp -p /etc/inittab.bak /etc/inittab
```

/etc/inittab 内の

```
id:5:initdefault:
```

を以下のように編集する。

```
id:1:initdefault:
```

2) 実行

reboot する。

```
# shutdown -r now
```

BIOS 起動後に GRUB のメニュー画面が表示されるので、**Asianux (2.4.21-9.38AX)**を選択。

OProfile 使用時は **Asianux (2.4.21-9.38AXsmp)**を選択。

スクリプトを実行する。

```
# cd $(DIR_DISKIO)/diskio-2.0/run
# ./list
```

OProfile 使用時は以下のように実行する。

```
# cd $(DIR_DISKIO)/diskio-2.0/run
# ./list_oprofile
```

今回は以下のように実行した。

```
# cd /root/diskio/diskio-2.0/run
# ./list
```

OProfile 使用時は以下のように実行した。

```
# cd /root/diskio/diskio-2.0/run
# ./list_oprofile
```

実行中は、現在の計測条件が表示される。全てが完了すると「Finished.」を表示し、プロンプトに戻る。

「Finished.」が表示されていない場合は、\$(WRITE_DEVICE)の指定が間違っている可

能性があるため、確認する。

プロンプトに戻らなければ、エラーが起きているので、3.1.3 章を参照し、対処すること。

3) システム設定の戻し

```
# mv /etc/inittab.bak /etc/inittab
```

1-1)で/etc/fstab を変更していた場合は、元に戻しておく。

```
# mv /etc/fstab.bak /etc/fstab
```

reboot する。

```
# shutdown -r now
```

3.1.3 異常時の対処方法

実行中に異常が発生した場合は、以下の作業を行なう。

1) Ctrl-Z を押し、計測を一時中断する。

```
# kill -KILL %1  
# mount
```

として、書き込み用マウントポイントにパーティションが mount されていないことを確認する。

2) マウントされている場合はアンマウントする。

```
# umount $(WRITE_DEVICE)
```

今回は、以下のように実行した。

```
# umount /dev/sdb1
```

3.1.4 ログデータの見方

ログデータは以下のディレクトリに格納される。

```
$(DIR_DISKIO)/diskio-2.0/run/$(DISTRVERSION)/ext3-do
```

今回は、以下のディレクトリに格納された。

```
/root/diskio/diskio-2.0/run/Miracle3.0-2.4.21-9.38AX/ext3-do
```

ext3-do は ext3 ファイルシステムの ordered モードでマウントしたことを表す。

各ログデータには以下のものが含まれている。

1) mkfs の結果

mkfs の結果が格納される。ファイルシステムのフォーマットに成功しているかどうかを確認できる。

2) /proc/cpuinfo,/proc/version,/proc/meminfo,/proc/scsi/scsi の内容

計測に用いるハードウェアの情報が格納される。

取得するのは以下の情報である。

- ・ CPU 情報
- ・ カーネルバージョン
- ・ メモリ情報

3) diskio 本体のログ

計測前のパラメータ

駆動スクリプトがどのような値を入力したかを確認することができる。

計測後のパラメータ

計測後にパラメータ領域に格納されている値を出力している。本来、試験前のパラメータと同じ値であるはずだが、これらの値に計測前と差異がある場合、なんらかのバグ、あるいはエラーが発生した危険性があり、計測結果は信頼するべきではない。

計測結果

計測結果は diskio 内部にて行なわれた同期書き込みごとにかかった時間である。

```
"# start      : Measurement result in ticks"
```

```
"# end        : Measurement result in ticks"
```

なる文字列に囲まれていて、スペースで区切られた 2 つのフィールドを持つ。双方共に write にかかった時間(CPU tick 数)を表わし、基本的には同じ値である。左のフィールドが 10 進数で、右のフィールドが 16 進数で表わされている。

計測結果の例:

```
8807540 866474
16890858 101bbea
16898868 101db34
~中略~
16895076 101cc64
16897422 101d58e
16811216 10084d0
16898004 101d7d4
```

3.2 Iozone による評価

Iozone による評価手順を紹介する。評価においては、Iozone ベンチマークのスコアを評価するのではなく、このベンチマークを実行することでシステムの限界状態を発生させ、この時のカーネル状態を分析・調査するのが目的である。始めに Iozone, OProfile, LKST それぞれの個別の実行手順を紹介し、最後にそれらを併せて実行する手順を示す。

3.2.1 ファイルシステムの再作成手順

ファイルシステムのコンディションに起因して I/O 処理に乱れが生じるのを防ぐため、Iozone を実施する前に毎回ファイルシステムの再作成を実施する。使用するファイルシステムそれぞれ (/test/f1 ~ /test/f4) について再作成を行う。

/test/f1 の再作成手順：

```
# umount /dev/sdb1
# mkfs -t ext3 -L /test/f1 /dev/sdb1
# mount /dev/sdb1 /test/f1
```

今回の環境での所要時間は、18GB のファイルシステムひとつあたりおよそ 15 秒程度だった。

3.2.2 iozone コマンド実行手順

問題を単純化するため、今回は Iozone の数あるパターンのうち write (すなわち initial write) のみを実施し、この時のカーネルの動作を把握する。re-write も実施しない。このためのコマンドオプションは "-i 0 -n" である。

可能な限り詳しいデータを出力するためにオプション "-C -M -+u" を指定し、また Excel 形式の出力も行うため "-R" も指定する。

I/O 用のディスクを 4 台用意したシステムにおいて、I/O を並列に 1~4 個を同時に実行する。それらのファイルサイズの合計は、メインメモリ 6GB を上回る、8GB とする。これは、ファイルへの書き込みがページキャッシュによってメモリ上に留まることなく、実際にディスクに書き込まれる状況が発生されるためである。また、レコードサイズはデフォルト値のまま (4kByte) とする。

並列の I/O は別々のファイルシステム (すなわち別々のディスク) に対して行うこととし、オプション "-f" または "-F" で指定する。

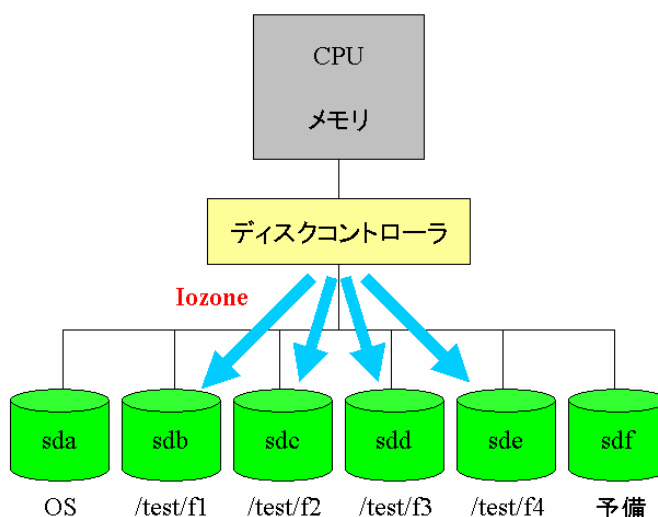


図 3.2-1 ベンチマーク構成

また Iozone 開始時のメモリ空き容量に差が出ないようにするため、測定にあたっては常にシステムを再起動した直後にベンチマークを開始する。

ファイルシステムのタイプには ext3 を使用し、直前に毎回 mkfs コマンドによってファイルシステムの再作成を行うこととする。

実施する I/O パターンの注目点は、並列の数、書き込みプロセスが子プロセスかスレッドかどうか、iozone コマンド自体を複数実行するかどうかの 3 種類とする。これは I/O 処理の競合状態に特に注目する狙いがある。

以上の条件から、Iozone の実施パターンは表 3.2-1 に示した 11 通りになる。

表 3.2-1 Iozone 実行パターン

パターン ID	オプション -s	オプション -t	オプション -T	iozone コマンド数
1	8G	1	×	1
2	8G	1		1
3	4G	2	×	1
4	4G	2		1
5	4G	1	×	2
6	2731M	3	×	1
7	2731M	3		1
8	2731M	1	×	3
9	2G	4	×	1
10	2G	4		1
11	2G	1	×	4

パターンを実行する具体的なコマンドを次に示す。パターンごとに別々のディレクトリに結果を保存するものとする。

パターン 8 の実行例 (sh 系スクリプトのループ):

```
for num in 1 2 3; do
  /root/os-bench/src/current/iozone -CMR -i 0 -+n -+u -s 2731M -t 1 -f \
  /test/f${num} > /root/os-bench/pattern8/iozone${num}.out &
done
wait
```

パターン 10 の実行例 :

```
# /root/os-bench/src/current/iozone -CMR -i 0 -+n -+u -s 2G -t 4 -T -F \
/test/f1/io /test/f2/io /test/f3/io /test/f4/io \
> /root/os-bench/pattern10/iozone.out
```

実行するシステムの性能や、パターンによって異なるが、所要時間はおよそ 3 分間前後である。

以上のオプションを指定した時の iozone コマンドの標準出力は以下ようになる。

パターン 3 の結果 :

```
Iozone: Performance Test of File I/O
```

Version \$Revision: 3.226 \$ バージョン

Compiled for 32 bit mode. ビルドに関するデータ

Build: linux

Contributors: William Norcott, Don Capps, Isom Crawford, Kirby Collins

Al Slater, Scott Rhine, Mike Wisner, Ken Goss

Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,

Randy Dunlap, Mark Montague, Dan Million,

Jean-Marc Zucconi, Jeff Blomberg,

Erik Habbinga, Kris Strecker.

Run began: Thu Jan 6 14:54:51 2005 実行開始時刻

Machine = Linux bigmem.miraclelinux.com 2.4.21-9.38AXsmp #1 SMP Wed Nov 10 21:

Excel chart generation enabled 使用したマシン名とカーネル

Excel chart generation enabled

No retest option selected

CPU utilization Resolution = 0.010 seconds.

CPU utilization Excel chart enabled

File size set to 4194304 KB ファイルサイズ

Command line used: /root/iozone3_226/src/current/iozone -CMR -i 0 --n --u -s 4G -t 2 -F
/test/f1/io /test/f2/io 指定されたコマンド

Output is in Kbytes/sec

Time Resolution = 0.000001 seconds. 計測時間単位

Processor cache size set to 1024 Kbytes.

Processor cache line size set to 32 bytes.

File stride size set to 17 * record size.

Throughput test with 2 processes

Each process writes a 4194304 Kbyte file in 4 Kbyte records ファイルサイズとレコードサイズ

Children see throughput for 2 initial writers = 75952.70 KB/sec 合計速度

Parent sees throughput for 2 initial writers = 71494.82 KB/sec

Min throughput per process = 37809.98 KB/sec 最小速度

Max throughput per process = 38142.72 KB/sec 最大速度

Avg throughput per process = 37976.35 KB/sec 平均速度

Min xfer = 4157640.00 KB 最小転送量

CPU Utilization: Wall time 113.091 CPU time 159.830 CPU utilization 141.33 %

CPU 利用率

Child[0] xfer count = 4194304.00 KB, Throughput = 38142.72 KB/sec, wall=113.091,

```
cpu=80.020, %= 70.76      子プロセス 1 の結果
      Child[1] xfer count = 4157640.00 KB, Throughput = 37809.98 KB/sec, wall=113.089,
cpu=79.810, %= 70.57      子プロセス 2 の結果

以下は Excel 用の CVS データ

"Throughput report Y-axis is type of test X-axis is number of processes"
"Record size = 4 Kbytes "
"Output is in Kbytes/sec"

" Initial write " 75952.70      スループットデータ

"      Rewrite "      0.00

"CPU utilization report Y-axis is type of test X-axis is number of processes"
"Record size = 4 Kbytes "
"Output is in CPU%"

" Initial write " 141.33      CPU 利用率データ

"      Rewrite "      0.00

iozone test complete.      出力の終了
```

3.2.3 OProfile 実行手順

OProfile は、MIRACLE LINUX V3.0 に `oprofile-0.5.4-22.i386.rpm` という RPM パッケージとして収録されており、インストール時のパッケージ選択においてこれを明示的に選択するか、または「すべて」を選んだ場合にインストールされる。

まず始めに、ハードウェアが OProfile をサポートしているかどうかを確認するため、次のコマンドを実行する。

```
# opcontrol --list-events
```

各種イベントが出力されたら OK である。

もしも以下の出力だった場合（ノート PC などで見られる）は、ハードウェアの制限によりカーネルのプロファイリングを行うことができないため、別なマシンに移行しなければならない。

```
# opcontrol --list-events
using timer interrupt
#
```

サンプリングのイベントタイプにデフォルトの `GLOBAL_POWER_EVENTS` を使用するので、経過時間に比例したサンプリングデータとなる（Pentium® III の場合は、`CPU_CLK_UNHALTED` イベントがこれに対応する）。また、カウントサイクル¹¹は、このシステムのデフォルトの 996,500 を使用した（デフォルト値はシステムによって異なる）。

OProfile を実行するコマンドは以下の通り。

準備（デバッグカーネルの指定と既存データの消去）:

```
# opcontrol --vmlinux=/usr/lib/debug/boot/vmlinux-$(uname -r).debug
# opcontrol -reset
```

開始（サンプリングの開始）:

```
# opcontrol -start
```

終了（サンプリングの終了）:

```
# opcontrol -shutdown
```

データ処理（各種統計データの抽出と、元データの保存）:

¹¹何回イベントが発生したらサンプリングを行うかの割合

```
# oprofpp -i /usr/lib/debug/boot/vmlinux-$(uname -r).debug -l -r \
-t vsqQnh > summary.out
# oprofpp -i /usr/lib/debug/boot/vmlinux-$(uname -r).debug -L -o \
-t vspqndh > detail.out
# op_to_source -i /usr/lib/debug/boot/vmlinux-$(uname -r).debug \
-output-dir ./src
# opcontrol -save=session1
```

コマンド出力をリダイレクトするファイル名(summary.out や detail.out)、ディレクトリ名(/src)、セッション名(session1) は測定ごとに変更するか、またはディレクトリを分けて保存する。

以上の手順によって、計測パターンごとに以下の 4 種類のデータが得られる。

表 3.2-2 OProfile のデータ

ファイル summary.out	カーネルの各シンボルごとのプロファイリングデータ
ファイル detail.out	シンボルを細分化した、命令単位のデータ
ディレクトリ ./src/ 下のファイル	プロファイリングデータを併記したソースコード
ディレクトリ /var/lib/oprofile/samples/sp1/	OProfile サンプル生成データ保存先。これをもとに別形式のデータを抽出可能。

summary.out のフォーマットは次の通り。占有率の高い順にシンボルがリストされる。

```
Cpu type: P4 / Xeon with 2 hyper-threads CPUの種類
Cpu speed was (MHz estimation) : 1993.57 CPUクロック
Counter 0 counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped) with a
unit mask of 0x01 (count cycles when processor is active) count 996500 イベント種別とサンプリング頻度
vma      samples  %      cum. %  symbol name
仮想アドレス  占有率      シンボル名
          サンプル数      累積占有率
c018ab58 8809   17.9545  17.9545  .text.lock.buffer
c0160be0 4852   9.88933  27.8438  do_generic_file_write
c0183540 3283   6.6914   34.5352  fs_may_remount_ro
c0184910 2511   5.11791  39.6531  remove_from_queues
c03116b0 2407   4.90594  44.559   simple_strtoull
c017d630 2144   4.36989  48.9289  alloc_bounce_page
c0109240 2142   4.36582  53.2947  default_idle
```

c0171450 1098	2.23794	55.5327	__free_pages_ok
c0186030 912	1.85883	57.3915	refile_buffer
c012de70 877	1.7875	59.179	__put_task_struct

detail.out のフォーマットは次の通り。シンボルごとに、その内訳がサンプル数や占有率と共にリストされる。

```
Cpu type: P4 / Xeon with 2 hyper-threads CPUの種類
Cpu speed was (MHz estimation) : 1993.57 CPUクロック
Counter 0 counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped) with a
unit mask of 0x01 (count cycles when processor is active) count 996500
```

vma	samples	%	%	symbol name	linenr	info
仮想アドレス	シンボル内占有率	サンプル数	全体に対する占有率	シンボル名	ソースコード:行番号	
c0109220 1	0.0020382	0.0020382	disable_hlt	/usr/.../process.c:77	シンボル	
c010922d 1	100	0.0020382		/usr/.../process.c:78	内訳	
c0109230 3	0.00611459	0.00611459	enable_hlt	/usr/.../process.c:82	シンボル	
c0109234 2	66.6667	0.00407639		/usr/.../process.c:82	内訳	
c0109236 1	33.3333	0.0020382		/usr/.../process.c:83	内訳	
c0109240 2142	4.36582	4.36582	default_idle	/usr/.../current.h:9	シンボル	
c0109248 2110	98.5061	4.30059		/usr/.../current.h:7	内訳	
c0109249 32	1.49393	0.0652223		/usr/.../current.h:7	内訳	

3.2.4 LKST 実行手順

改良版 LKST による測定には、通常のカーネルではなく改良版 LKST 搭載カーネルに変更する必要がある。これは、本 OSS 活用基盤整備事業のプロジェクトの一つ、LKST 開発グループの成果である。

「2.2.3 OS の設定」節でインストールした改良版 LKST カーネル kernel-smp-2.4.21-9.38AX.lkst10 を使ってシステムを再起動する。

LKST の詳しい使用方法については、開発チームのドキュメントを参照のこと。

この測定では 2 種類のマスクセットを使用する。ひとつは、ロックを分析するためのセットで、ロック競合傾向 (busywait) とロック取得期間 (spinlock) のみを有効にしたもの。もうひとつは、ブロック IO 処理時間 (buffer) とブロック IO リクエストキュー長 (blkqueue) のみを有効にしたものである。

開発チームのドキュメントを参照して、2 つのマスクセットファイルを作成する。今回測定に使用したファイルは lkstmask.busywait.spinlock と lkstmask.buffer.blkqueue というファイル名で用意した。

マスクセットファイルの作成には、“lkstm read -m 0 -d” の出力結果を加工するとよい。LKST が更新されて新しいイベントが追加された際に、古いマスクセットをロードして使用すると、新しいイベントはデフォルト状態のままになる。そのため、それがデフォルトで有効なイベントである場合は、ユーザが意図しないままデータを取得し、これがオーバーヘッドとなる可能性がある。これを防ぐには、LKST がアップデートされるたびに、“lkstm read -m 0 -d” からマスクセットを再作成するのが最もよい方法といえる。また、最新のバージョンでは lkst_make_mask コマンドによってマスクセットの作成とロードが一括して行えるためそちらを利用するのも便利である。

LKST の実行コマンドは以下の通り。バッファサイズは確保可能な最大値 (25MB × 4CPU=100MB) を指定しており、今回の測定ではおよそ 20 ~ 60 秒間前後 (イベントマスクや、I/O パターン依存) のデータが格納された。

準備 (サービスの起動、マスクセットの設定、バッファサイズの拡張):

```
# service lkst start
# lkstm write -f ./lkstmask.busywait.spinlock -n spinlock
# lkstbuf create -s 25M -b 1
# lkstbuf jump -b 1
# lkstbuf delete -b 0
```

開始:

```
# lkst start
# lkstm set -n spinlock
```

停止：

```
# lkst stop
```

データ処理：

```
# lkstbuf read -f ./lkst.data
# service lkst stop
# lkstlogdiv ./lkst.data
# lkstla busywait -l ./lkst.data-? > ./lkst.busywait.l
# lkstla busywait -d ./lkst.data-? > ./lkst.busywait.d
# lkstla busywait -s ./lkst.data-? > ./lkst.busywait.s
# lkstla spinlock -l ./lkst.data-? > ./lkst.spinlock.l
# lkstla spinlock -d ./lkst.data-? > ./lkst.spinlock.d
# lkstla spinlock -s ./lkst.data-? > ./lkst.spinlock.s
# rm -f ./lkst.data-?
# lkstbuf print -f ./lkst.data -r -C > ./lkst.data.print
```

以上の手順で得られるデータファイルは以下の6種類である。それぞれの見方についてはLKSTのドキュメントを参照のこと。

表 3.2-3 LKST データ

lkst.data	バイナリ形式の生データ(直接読むことはできない)
lkst.data.print	時系列データ
lkst.busywait.l	ビジーウェイト時間のログ
lkst.busywait.d	ビジーウェイト時間の対数分布値
lkst.busywait.s	ビジーウェイト時間の統計結果
lkst.spinlock.l	ロック使用時間のログ
lkst.spinlock.d	ロック使用時間の対数分布値
lkst.spinlock.s	ロック使用時間の統計結果

3.2.5 iozone+OProfile 実行手順

以上で紹介した手順を組み合わせ、以下の流れで実施する。

ステップ：

1. システム再起動
2. I/O 用ファイルシステム再作成
3. OProfile 準備
4. OProfile 開始
5. Iozone 開始
6. Iozone 終了
7. OProfile 停止
8. OProfile データ回収

手順 2～8 を一括して行うスクリプト run.sh を用意した。スクリプト冒頭のシェル変数を正しく設定した上で、引数に I/O パターンの ID を与えて実行する。

表 3.2-4 run.sh の設定変数

変数名	内容
baseDir	run.sh、マスクセットファイルを置くディレクトリ。このディレクトリの中に測定データが出力される。
IOZONE	iozone コマンドの絶対パス。
LKSTbuffersize	LKST のバッファサイズ。確保可能な限り最大の値を指定する。4CPU 6GB メモリの検証環境では 25M だった。
SIZE1	1 個の I/O(パターン id 1,2)時に作成するファイルサイズ
SIZE2	2 個の I/O(パターン id 3,4,5)時に作成するファイルサイズ。SIZE1 の 1/2 を指定する。
SIZE3	3 個の I/O(パターン id 6,7,8)時に作成するファイルサイズ。SIZE1 の 1/3 を指定する。
SIZE4	4 個の I/O(パターン id 9,10,11)時に作成するファイルサイズ。SIZE1 の 1/4 を指定する。

run.sh と一緒に提供するマスクセットの設定ファイル lkstmask.buffer.blkqueue と lkstmask.busywait.spinlock は baseDir で指定したディレクトリ内に置くこと。LKSTbuffersize についてはシステム構成依存のため、トライアンドエラーで設定するしかない。メモリに留まることなくディスクへの書き込みが発生するように、SIZE1 については搭載物理メモリを上回る値を指定する。今回の評価環境は 6GB 搭載のため、SIZE1 に 8G を指定した。

以下は I/O パターン 4 を実行した例である。

```
# ./run.sh 4
mke2fs 1.32 (09-Nov-2002)
Filesystem label=/test/f1
OS type: Linux
Block size=4096 (log=2)

(中略 : ファイルシステム再作成のログ)

This filesystem will be automatically checked every 25 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
===== Start of id 4 at Wed Feb 9 17:37:59 JST 2005 =====
Stopping profiling.
Killing daemon.
===== End of id 4 at Wed Feb 9 17:39:53 JST 2005 =====
#
```

最後に“End of id …”というメッセージが表示されたら正常終了である。
マシンの処理性能に依存するが、およそ 30 分以上経過しても終了しない場合は何らかの問題が発生した可能性がある。この場合はシステム状態を確認して、必要に応じてプログラムを強制終了するか、システムの反応がない場合はリセットスイッチを押すことにより、強制的にシステムを再起動させる。
正常終了すると、表 3.2-2 と表 3.2-3 に挙げた各データファイルが /root/os-bench/pattern<ID>-0/ ディレクトリ下に作成される。
スクリプト run.sh のより詳しい説明については付属の README ファイルを参照のこと。

3.2.6 lozone+OProfile+LKST 実行手順

ロック競合が発生するケースや、CPU にアイドルが発生するケースについては、OProfile に加えて改良版 LKST による測定、分析を行う。

改良版 LKST を使うにはこの対応を施したカーネル 2.4.21-9.38AX.lkst10.smp に切り替えてシステムを起動する。

LKST を有効にすることで若干のオーバーヘッドが懸念されるが、これに対する検証は LKST 開発チームによって報告されるのでそちらを参照されたい。この調査では改良版 LKST が実装されていない通常版カーネルと OProfile 結果の比較をすることで LKST 実装版カーネルでのデータの妥当性を判断する。

ステップ：

1. システム再起動
2. top コマンドの開始
3. I/O 用ファイルシステム再作成
4. OProfile 準備
5. LKST 準備
6. OProfile 開始
7. LKST 開始
8. Iozone 開始
9. Iozone 終了
10. LKST 停止
11. OProfile 停止
12. LKST データ回収
13. OProfile データ回収

LKST で取得したデータにプロセス id が含まれるが、今回使用したマスクセットでは、プロセスのコマンド情報が記録されないため、あらかじめ以下の通りに top コマンドを実行しておき、プロセス id とコマンドとの対応情報を記録する。

```
# top -b -C -d 10 > ~/top.log &
```

オプション `-b` はバッチモードの指定、`-d` は更新間隔（秒）である。更新間隔を短くするとプロセス情報をもれなく取得することができるが、その分 top コマンドによって発生するスピンロックの頻度が増し、LKST のスピンロック・データに占める top コマンドの割合が多くなり本来検出すべき情報が乱される。このため、可能な限り更新間隔を広げることが望ましく、今回は 10 秒間とした。

前節で紹介したスクリプト run.sh を使用することで手順 3～13 を一括して実行できる。オプションでマスクセットの種類を指定し、最後に I/O パターンを与えて実行す

る。

表 3.2-5 run.sh のコマンドラインオプション

オプション	機能
-b	マスクセットで buffer および blkqueue のみを有効にする。
-s	マスクセットで busywait および spinlock のみを有効にする。
無指定	LKST を使用しない。

I/O パターン 9 で buffer と blkqueue イベントの LKST を実行した例：

```
# ./run.sh -b 9
mke2fs 1.32 (09-Nov-2002)
Filesystem label=/test/f1
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)

(中略：ファイルシステム再作成のログ)

This filesystem will be automatically checked every 23 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
Starting Kernel State Tracer: [ OK ]
New maskset id=3 was written. (Name:buffer.blkqueue)
New buffer was created, cpu=0, id=1 size=26214400 + 4032(margin)
New buffer was created, cpu=1, id=1 size=26214400 + 4032(margin)
New buffer was created, cpu=2, id=1 size=26214400 + 4032(margin)
New buffer was created, cpu=3, id=1 size=26214400 + 4032(margin)
Currently selected buffer changed to 1 on CPU 0
Currently selected buffer changed to 1 on CPU 1
Currently selected buffer changed to 1 on CPU 2
Currently selected buffer changed to 1 on CPU 3
Buffer id=0 was deleted.
Buffer id=0 was deleted.
Buffer id=0 was deleted.
Buffer id=0 was deleted.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
Start LKST event tracing.
Currently selected maskset was changed to id=3
===== Start of id 9 at Wed Feb 9 17:23:54 JST 2005 =====
Stop LKST event tracing.
Stopping profiling.
Killing daemon.
Stopping Kernel State Tracer: [ OK ]
```

```
line:228842 timespec 1107937528.156739818(d0) less than
1107937539.544190317(d5)
line:229007 timespec 1107937527.037816664(d1) less than
1107937539.652148159(d1)
line:286292 timespec 1107937526.681834188(d0) less than
1107937537.174277083(d5)
===== End of id 9 at Wed Feb 9 17:30:19 JST 2005 =====
#
```

正常終了すると、データファイルが /root/os-bench/pattern<ID>-<指定オプション>/ ディレクトリ下に作成される。スクリプト run.sh のより詳しい説明については付属の README ファイルを参照のこと。
また、top のバックグラウンド実行も忘れずに終了する。

4 性能・信頼性評価結果と分析・考察

4.1 diskio の結果

4.1.1 はじめに

今回、我々は以下の項目について検証を行った。

1. diskio と DBT-3 の測定結果の相関関係の確認(4.1.2)
2. OProfile による内部プロファイリング(4.1.3)
3. diskio と性能限界との関係(4.1.4)

本測定で使用されている書き込み方式を表 4.1-1 のように略して記載する。

表 4.1-1 書き込み方式の略称

書き込み方式名	略称
Async open + write + fsync	fsync
Async open + write + fdatasync	fdatasync
O_SYNC open + write	osync
O_DSYNC open + write	odsync
Async open + sync + sync	syncsync
Async open + write	async

diskio と DBMS ベンチマークとの相関関係について調べるために、環境 A と環境 B の 2 つを用意した。

各測定環境の代表的なスペックは表 4.1-2 の通りである。CPU 動作クロックは /proc/cpuinfo に記されている値である。

表 4.1-2 環境 A と環境 B の代表的なスペック

	環境 A	環境 B
製品名	Dell PowerEdge 2600	HP ProLiant DL380 G3
プロセッサ	Intel(R) Xeon(TM) CPU 2.80GHz 2CPU	Intel(R) Xeon(TM) CPU 2.80GHz 2CPU
CPU 動作クロック	2791.829 MHz	2787.287 MHz
メモリ	4GByte	2.5Gbyte
ハードディスク	Ultra SCSI 320 HDD 10000rpm	Ultra SCSI 320 HDD 15000rpm

4.1.2 diskio と DBT-3 の測定結果の相関関係の確認

4.1.2.1 概要

環境 A、B で diskio と DBMS ベンチマークを実行し、結果を比較した。DBMS ベンチマークは、DBT-3 を使用した。

今回の測定では OSS 技術開発・評価コンソーシアムのドキュメント「DB 層ベンチマーク評価」の手順に従って DBT-3 を使用した。

4.1.2.2 測定結果

4.1.2.2.1 diskio の測定結果

図 4.1-1 から図 4.1-12 に diskio の測定結果のグラフを示す。c 軸は書き込み回数、t 軸は各書き込みにかかった時間(usec)を表す。折れ線は各書き込み方式での測定結果を表している。また、点線はその環境での HDD の回転にかかる時間を示しており、各書き込みを始めた位置を基準にして下から、1 回転、2 回転、3 回転するのにかかる時間になる。

環境 A の HDD は 10,000rpm なので、1 回転にかかる時間は 式 4.1-1 のようになる。

$$1 / 10,000 \times 60 \times 1,000,000 = 6000 \text{ (usec)} \quad (\text{式 4.1-1})$$

環境 B の HDD は 15000rpm なので、1 回転にかかる時間は 式 4.1-2 のようになる。

$$1 / 15,000 \times 60 \times 1,000,000 = 4000 \text{ (usec)} \quad (\text{式 4.1-2})$$

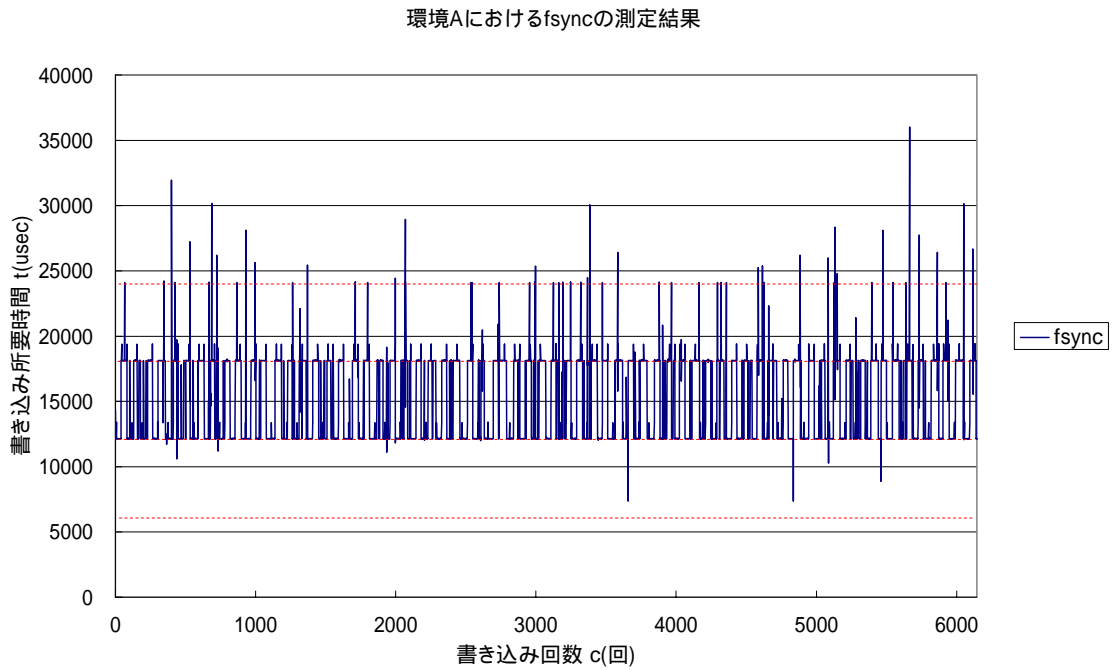


図 4.1-1 環境 A における fsync の測定結果

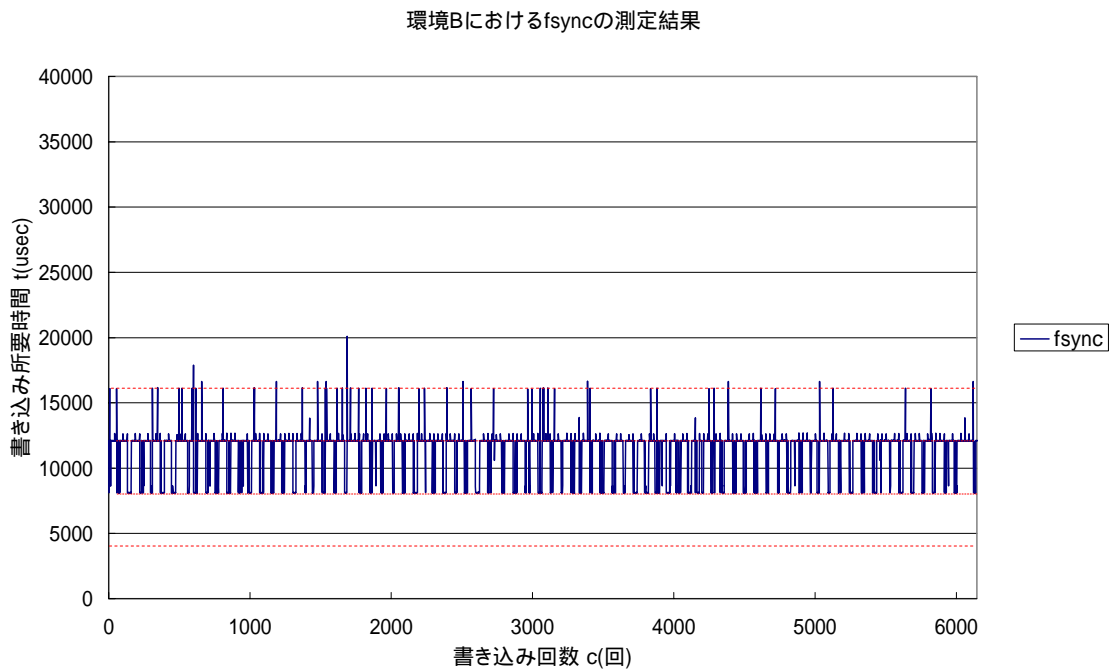


図 4.1-2 環境 B における fsync の結果

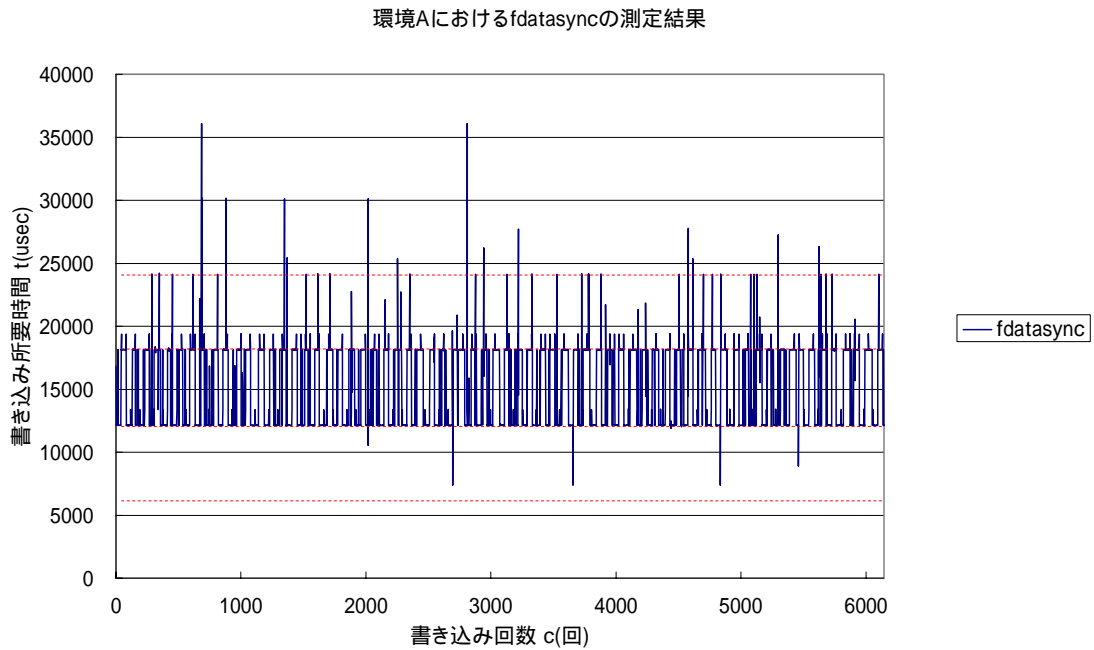


図 4.1-3 環境 A における fdatsync の測定結果

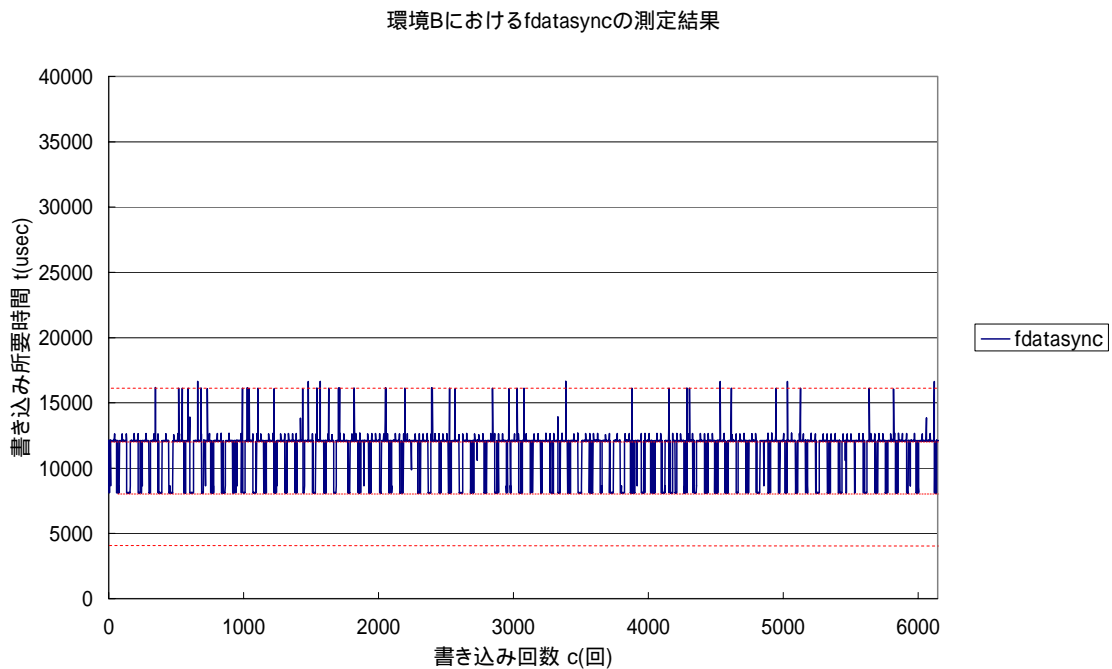


図 4.1-4 環境 B における fdatsync 測定結果

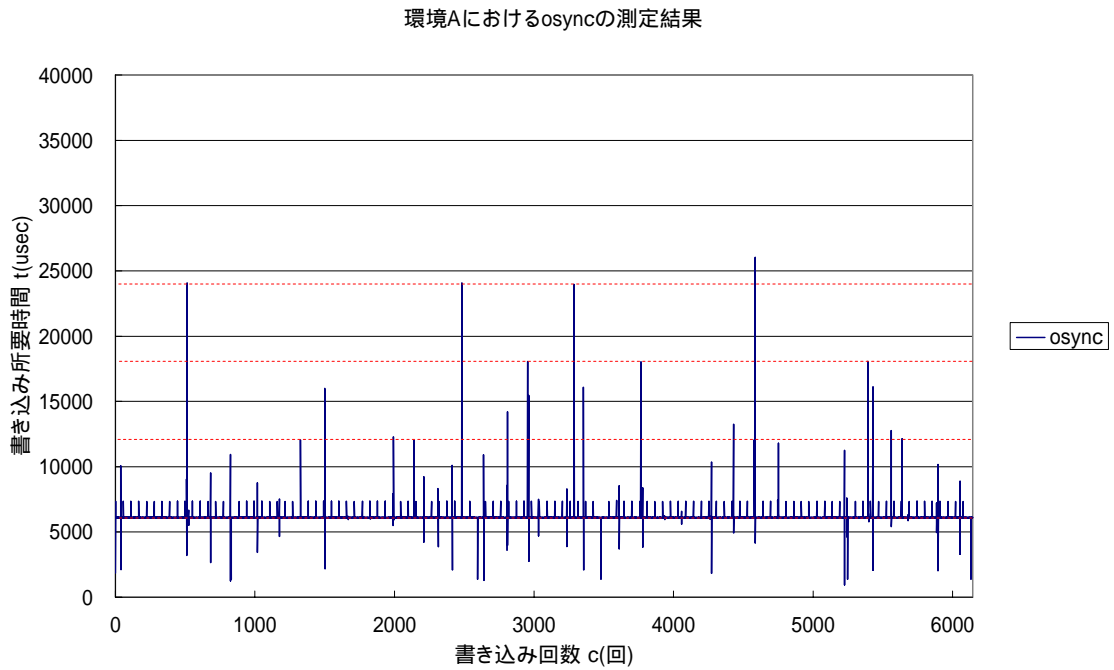


図 4.1-5 環境 A における osync の測定結果

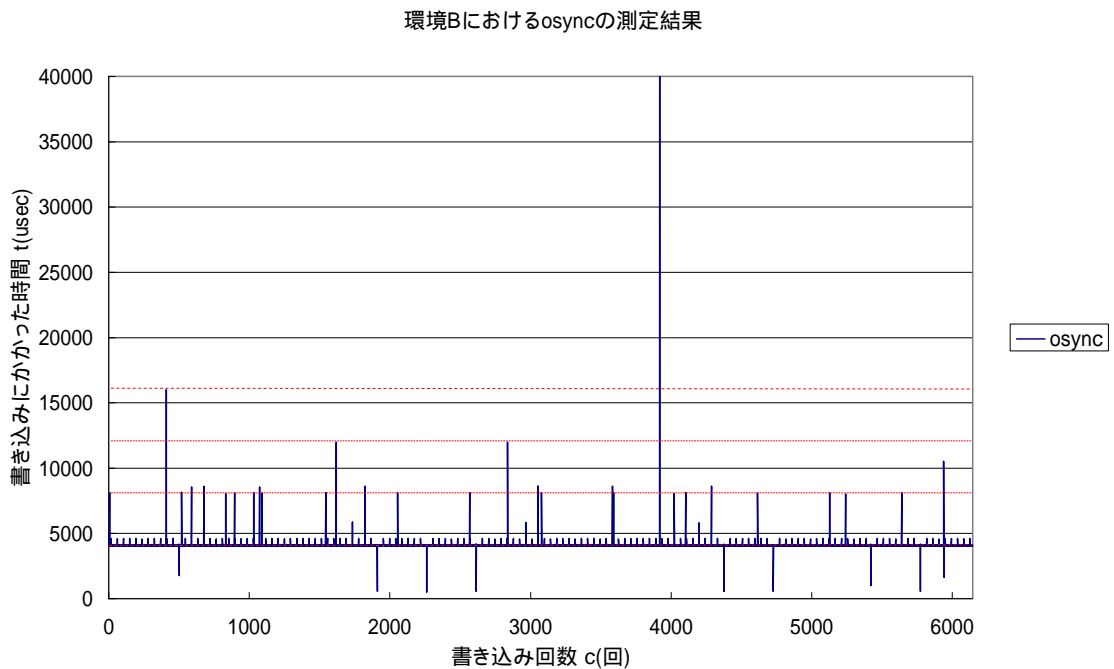


図 4.1-6 環境 B における osync の結果

環境Aにおけるodsyncの測定結果

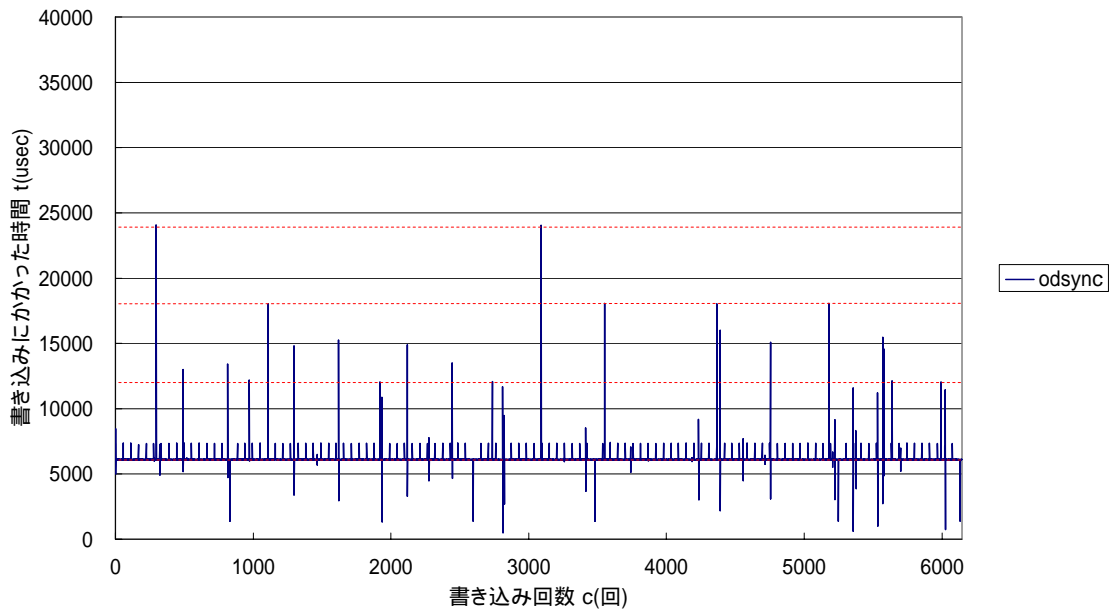


図 4.1-7 環境 A における odsync の測定結果

環境Bにおけるodsyncの測定結果

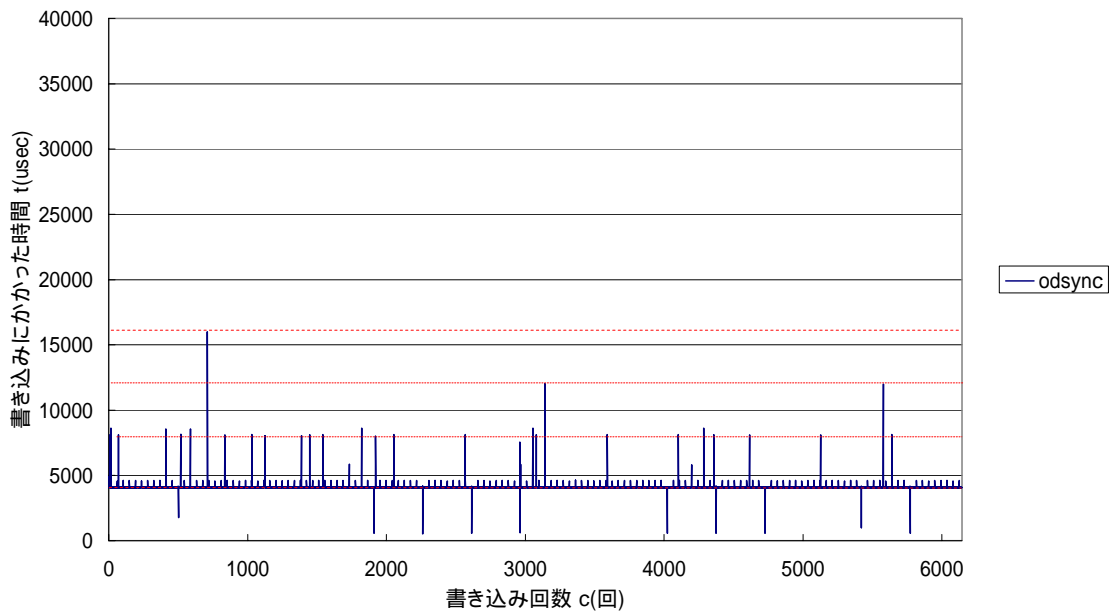


図 4.1-8 環境 B における odsync 時の測定結果

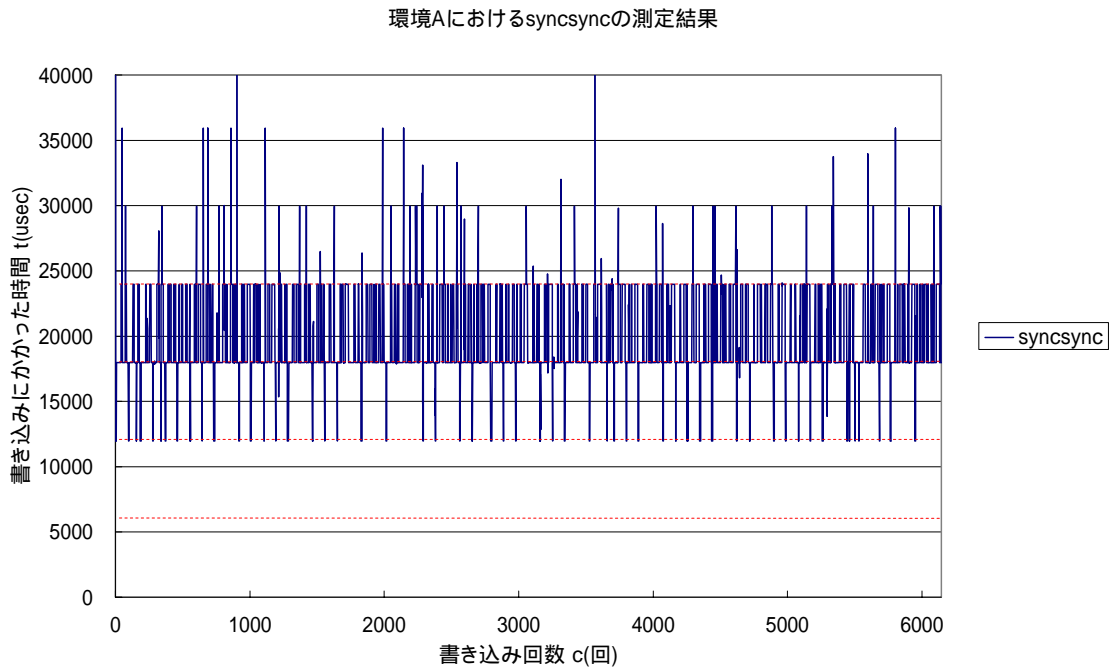


図 4.1-9 環境 A における syncsync の測定結果

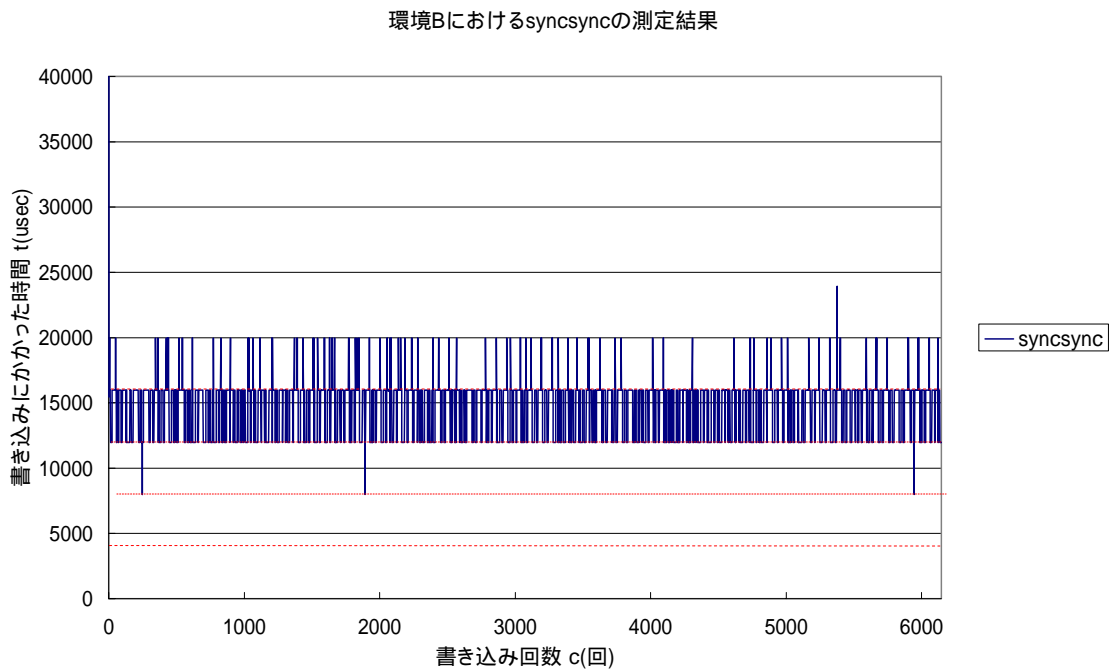


図 4.1-10 環境 B における syncsync の測定結果

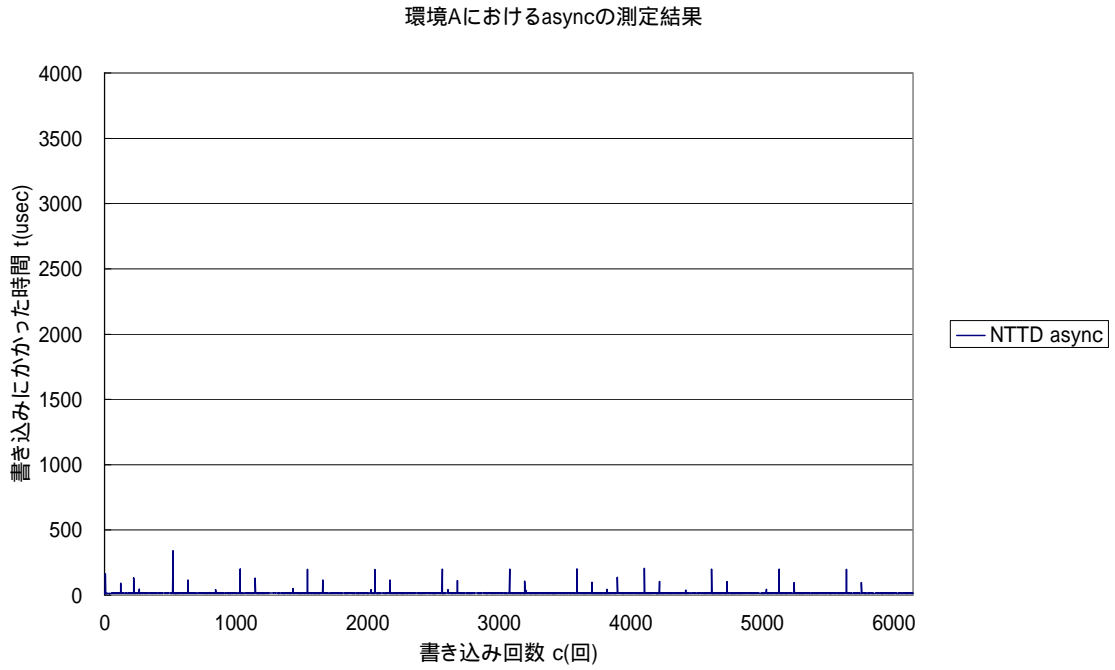


図 4.1-11 環境 A における async の測定結果

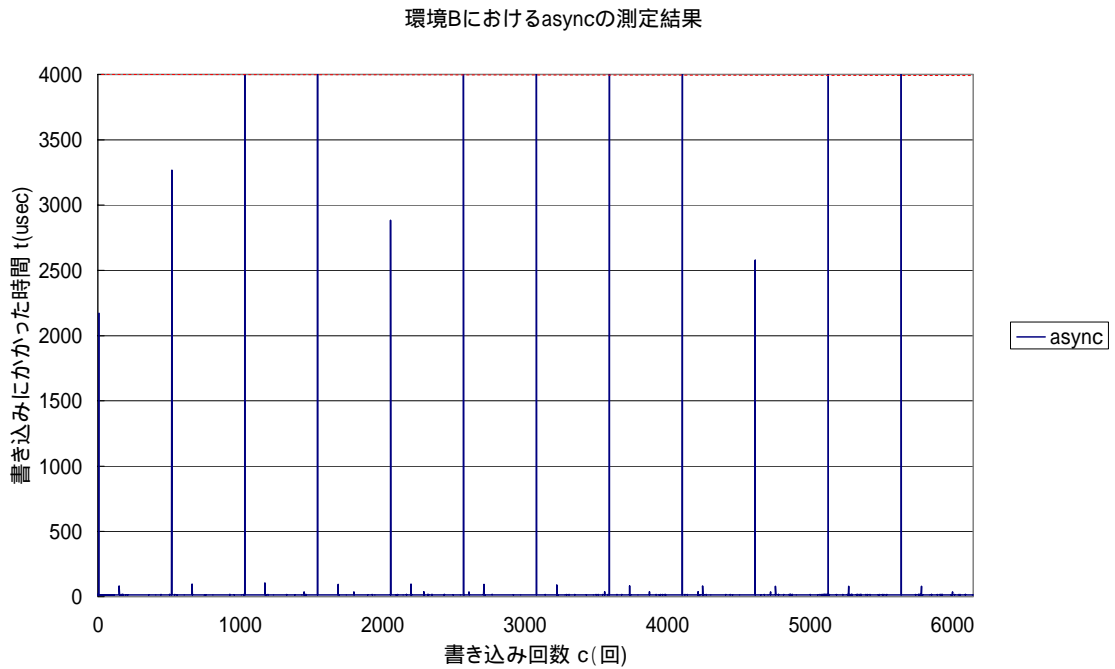


図 4.1-12 環境 B における async の測定結果

4.1.2.2.2 DBT-3 の結果

表 4.1-3 に環境 A と環境 B で測定した DBT-3 の結果を示す。

各数値はテストにかかった時間を秒に直したものである。

DBT-3 実行時の PostgreSQL のログ書き込みの同期方式は fsync で行った。

表 4.1-3 環境 A と環境 B における DBT-3 の測定結果(fsync)

DBT-3 テスト名	環境 A	環境 B
ロード時間(sec)	1146	1065
パワーテスト(sec)	581	521
スループットテスト(sec)	1801	1597

4.1.2.3 diskio の統計処理結果

表 4.1-4 に 4.1.2.2.1 項で得られた結果を統計処理したものを示す。各同期方式について、環境 A と環境 B についての平均値、最小値、最大値を usec で示してある。

表 4.1-4 diskio 測定結果の統計処理結果

書き込み方式	環境	平均値(usec)	最小値(usec)	最大値(usec)
fsync	環境 A	15202.210	7393.18	36009.40
	環境 B	11255.851	8079.48	20106.91
fdatasync	環境 A	15191.876	7393.07	36097.64
	環境 B	11257.044	8082.65	16669.46
osync	環境 A	6145.940	954.41	26024.58
	環境 B	4131.617	527.79	127788.19
odsync	環境 A	6144.879	495.46	24084.86
	環境 B	4103.449	527.87	16015.48
syncsync	環境 A	19596.666	11974.95	115236.38
	環境 B	14694.122	7981.42	92922.22
async	環境 A	16.6332	10.2299	340.2486
	環境 B	23.3006	7.5055	6181.4144

4.1.2.4 diskio の測定誤差

diskio の結果は、各書き込みにかかった CPU tick 数で出力される。1tick が何秒かは CPU ごとに異なるため、測定結果を比較するためには秒などに直す必要がある。

ここで、各環境の 1tick あたりの時間を nsec で表す。

環境 A の CPU MHz は 2791.829 MHz なので、1tick あたりの時間は 式 4.1-3 のようになる。

$$1\text{tick} = 1 \text{ (sec)} / 2791.829 \text{ (MHz)} = 0.358 \text{ (nsec)} \quad (\text{式 4.1-3})$$

環境 B の CPU MHz は 2787.287 MHz なので、1tick あたりの時間は式 4.1-4 のようになる。

$$1\text{tick} = 1 \text{ (sec)} / 2787.287 \text{ (MHz)} = 0.359 \text{ (nsec)} \quad (\text{式 4.1-4})$$

2.1.1.4 項で解説したように、diskio は TSC を使用して時間を測定している。TSC は CPU 駆動クロック数を数えているため、その精度は CPU 依存になる。

ここで生じると考えられる誤差は 2 種類ある。

最初に CPU パイプラインのストールによって生じる誤差について考える。今回使用している Xeon は Pentium 4 ベースであるため、パイプラインの段数は 20 段にもなる。仮に、全パイプライン段数分ストールしたと仮定しても、1パイプラインステージあたり 1tick しかかからないので、 $1\text{tick} = 0.36\text{nsec}$ とすると、 $0.36 \times 20 = 7.2(\text{nsec})$ となる。一方、本測定で扱う書き込み方式の中で最も短時間だったのは async 時の 7.5(μsec)であった。よって、この誤差は十分小さいと考えられる。

次に、smp カーネル使用時に発生する誤差について考える。マルチ CPU の場合、各 CPU がそれぞれの TSC を持っている。CPU の初期化タイミングが異なるため、TSC は違った値を持つことになる。本測定で用いた MIRACLE LINUX V3.0 のカーネルは起動時に CPU 間で TSC の同期をとっている。この同期の際に生じる誤差は CPU 間同期に用いられる FSB 信号 1tick 分になる。今回測定に使用した Xeon は 90nm Process の Pentium 4 ベースである。Intel の技術文書『Intel Pentium 4 Process on 90 nm Process Datasheet』によると、2.8GHz の CPU の場合、FSB の周波数は 533MHz、800MHz の 2 種類ある。533MHz の場合でも、誤差は $1 \text{ (sec)} / 533 \text{ (MHz)} = 1.88 \text{ (nsec)}$ となる。よって、十分小さいと考えられる。

4.1.2.5 diskio と DBT-3 についての考察

図 4.1-13 は、DBT-3 の結果と diskio fsync の結果をプロットしたものである。図の t1 軸が diskio fsync の平均書き込み時間、t2 軸が DBT-3 の所要時間を表している。

表 4.1-5 に示すのは、表 4.1-4 の測定結果を用いて、環境 A を 1 とした場合の環境 B の diskio、DBT-3 の測定結果を比率で表したものである。DBT-3 実行時の PostgreSQL のログ書き込みの同期方式は fsync である。そこで、diskio は fsync の結果を用いて比較を行う。また、非同期方式との違いを示すため、async の結果も示す。

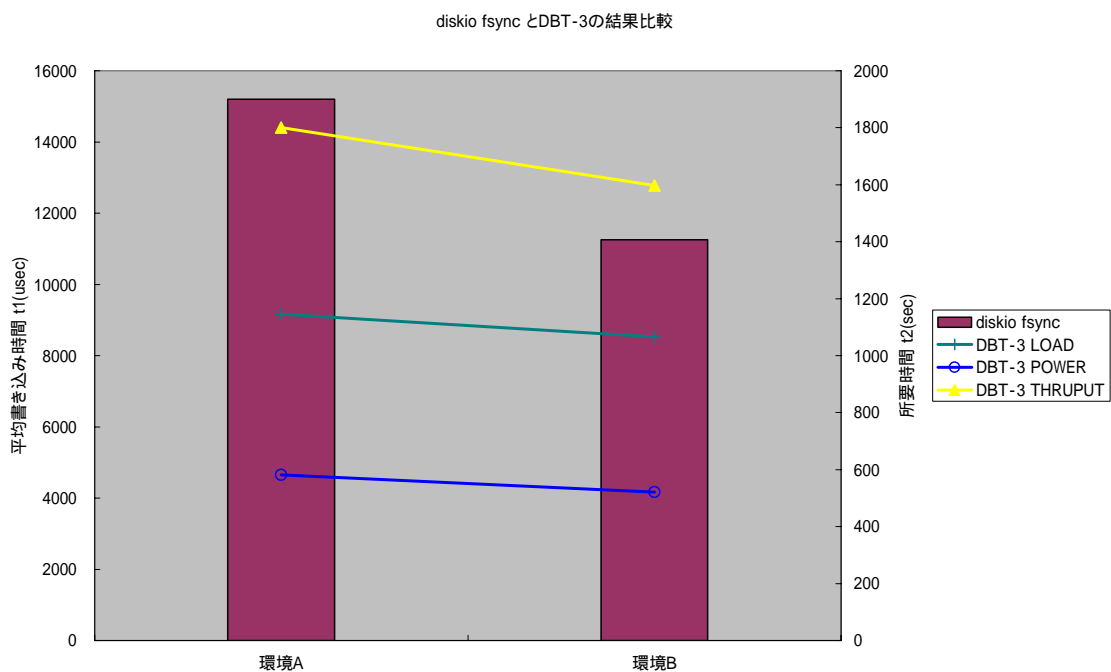


図 4.1-13 diskio fsync と DBT-3 の結果比較

表 4.1-5 環境 A を 1 としたときの環境 B の測定結果(fsync)

	環境 A を 1 としたときの比率
DBT-3 ロード時間	0.93
DBT-3 パワーテスト	0.90
DBT-3 スループットテスト	0.89
diskio fsync	0.74
diskio async	1.40

表 4.1-5 から、DBT-3 の各ベンチマーク結果と diskio fsync では共に環境 B の方が小さな値になっている。それに対し、async は環境 B の方が大きな値となっている。この事は、DBT-3 ベンチマーク負荷を DB 層にかけた場合の OS 層に対して生じる負荷の中の一定の割合が fsync 書き込みによるものであり、非同期書き込みである async 書き込みによる負荷は相対的に OS 層にとって小さい負荷であるためと考えられる。

図 4.1-14 に DBT-3 の処理時間における内部要素の寄与の割合と diskio の結果の割合との関係を示す。図中のその他には、join などの PostgreSQL プログラム中での演算などが含まれる。

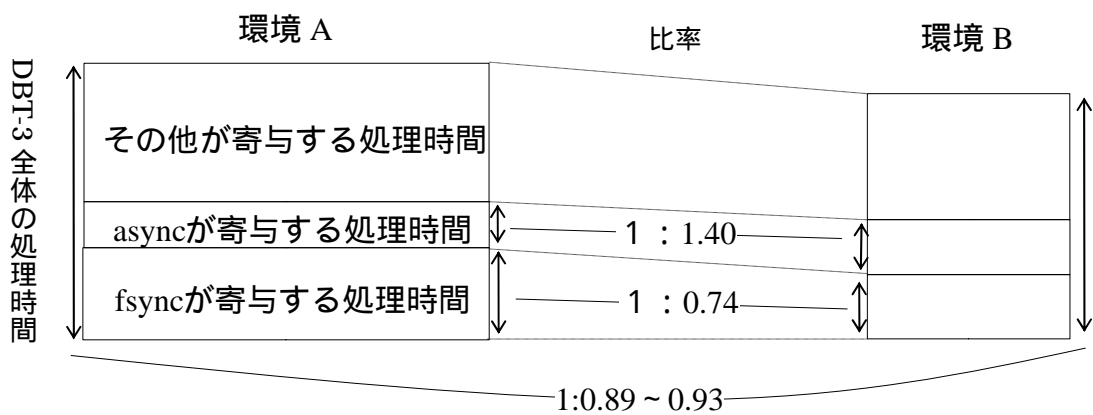


図 4.1-14 DBT-3 結果における fsync、async の寄与時間の割合

4.1.3 OProfile によるカーネル内部プロファイリング

4.1.3.1 概要

OProfile を使用して、diskio 実行時のカーネル内部での挙動を調べた。これは 2.1.4 章で述べているマイクロベンチマークとプロファイラを用いたボトルネックの検出を目的としている。

測定は環境 A のみで行い、fsync,fdatsync,osync,odsync について検証を行った。

OProfile のデフォルトサンプリング頻度は荒く、diskio のような短時間で終わってしまうベンチマークには向かない。そのため、サンプリング頻度を限界まで精度の高い 3000 に設定し、測定を行った。

4.1.3.2 OProfile の結果

表 4.1-6 から表 4.1-9 に fsync,fdatsync,osync,odsync について OProfile の実行結果を示す。サンプル数は各シンボルについて OProfile が検出した数を表す。占有率は全サンプル数に占める割合、累積占有率は、サンプル数が多いものから占有率を累積していったものである。各表はサンプル数が上位 15 個だったものについて示してあり、I/O 関係のものを で色分けした。

表 4.1-6 diskio fsync 実行時の OProfile の結果

順位	シンボル名	サンプル数	占有率(%)	累積占有率(%)
1	default_idle	1313	10.4906	10.4906
2	_wake_up	756	6.04027	16.53087
3	Schedule	686	5.48098	22.01185
4	mem_init	497	3.97092	25.98277
5	free_pages_init	452	3.61138	29.59415
6	_run_timers	446	3.56344	33.15759
7	try_to_wake_up	423	3.37967	36.53726
8	get_gendisk	357	2.85235	39.38961
9	_br_write_lock	352	2.8124	42.20201
10	_br_write_unlock	337	2.69255	44.89456
11	LKST_ETYPE_INT_HARDWARE_ENTRY_HEADER_hook	332	2.6526	47.54716
12	batch_entropy_store	298	2.38095	49.92811
13	ramdisk_size	288	2.30105	52.22916
14	ethif_probe	262	2.09332	54.32248
15	smp_apic_timer_interrupt	230	1.83765	56.16013

表 4.1-7 diskio fdatsync 実行時の OProfile の結果

順位	シンボル	サンプル数	占有率(%)	累積占有率(%)
1	default_idle	1287	10.6452	10.6452
2	_wake_up	666	5.50868	16.15388
3	Schedule	532	4.40033	20.55421
4	_run_timers	482	3.98677	24.54098
5	try_to_wake_up	426	3.52357	28.06455
6	free_pages_init	424	3.50703	31.57158
7	mem_init	410	3.39123	34.96281
8	_br_write_lock	403	3.33333	38.29614
9	get_gendisk	400	3.30852	41.60466
10	LKST_ETYPE_INT_HARDWARE_ENTRY_HEADER_hook	352	2.9115	44.51616
11	_br_write_unlock	341	2.82051	47.33667
12	batch_entropy_store	300	2.48139	49.81806
13	rebalance_tick	276	2.28288	52.10094
14	add_timer_randomness	244	2.0182	54.11914
15	ramdisk_size	235	1.94376	56.0629

表 4.1-8 diskio osync 実行時の OProfile の結果

順位	シンボル	サンプル数	占有率(%)	累積占有率(%)
1	default_idle	353	8.77019	8.77019
2	_wake_up	253	6.28571	15.0559
3	_run_timers	207	5.14286	20.19876
4	free_pages_init	170	4.2236	24.42236
5	Schedule	168	4.17391	28.59627
6	mem_init	149	3.70186	32.29813
7	_br_write_lock	147	3.65217	35.9503
8	LKST_ETYPE_INT_HARDWARE_ENTRY_HEADER_hook	136	3.37888	39.32918
9	get_gendisk	127	3.15528	42.48446
10	try_to_wake_up	125	3.10559	45.59005
11	end_buffer_io_sync	98	2.43478	48.02483
12	ramdisk_size	89	2.21118	50.23601
13	batch_entropy_store	82	2.03727	52.27328
14	_br_write_unlock	81	2.01242	54.2857
15	ethif_probe	78	1.93789	56.22359

表 4.1-9 diskio odsync 実行時の OProfile の結果

順位	シンボル	サンプル数	占有率 (%)	累積占有率 (%)
1	default_idle	356	8.60111	8.60111
2	_wake_up	227	5.48442	14.08553
3	Schedule	200	4.83209	18.91762
4	_run_timers	186	4.49384	23.41146
5	free_pages_init	185	4.46968	27.88114
6	_br_write_lock	166	4.01063	31.89177
7	mem_init	134	3.2375	35.12927
8	try_to_wake_up	125	3.02005	38.14932
9	end_buffer_io_sync	116	2.80261	40.95193
10	LKST_ETYPE_INT_HARDWARE_ENTRY_HEADER_hook	113	2.73013	43.68206
11	batch_entropy_store	107	2.58517	46.26723
12	get_gendisk	105	2.53684	48.80407
13	ramdisk_size	101	2.4402	51.24427
14	rebalance_tick	97	2.34356	53.58783
15	_br_write_unlock	92	2.22276	55.81059

4.1.3.3 diskio の OProfile 結果の解析

fsync、fdatsync と osync、odsync はそれぞれほぼ同じ傾向を示している。

上位 15 個のエントリで約 55%のサンプル数を占めている。

色づけされた I/O 関連のものにのみ着目すると、4 つの書き込み方式全てで `_br_write_lock`、`_br_write_unlock`、`get_gendisk` が表れている。

`end_buffer_io_sync` は `osync`、`odsync` のみに表れていることがわかる。

4.1.3.4 diskio の OProfile 結果の考察

4.1.3.3 で述べた 4 つの書き込み方式全てに表れている `_br_write_lock`、`_br_write_unlock` と `get_gendisk` について解析をする。

`_br_write_lock`、`_br_write_unlock` は `lib/brlock.c` に存在する。これは 'Big Reader' read-write spinlock の実装となっている。このロックは read を行うときには CPU ローカルで行われるため高速である。しかし、write を行うときには全 CPU をロックするようになっている。そのため、write を多く発行する `diskio` で浮上してきたボトルネックと推測できる。これはロックの挙動が CPU 数に依存することからもわかるように、OProfile を `smp` カーネル下で使用したために出てきた問題で、シングルユーザ用カーネル使用時には出てこないはずである。

I/O 関係でもう一つ数が多い `get_gendisk` は `drivers/block/genhd.c` に存在する。Linux カ

ーネル 2.4 では、処理中にリスト構造を使用し、それをリニアに検索している。それをたどる部分がボトルネックとなっていると考えられる。

default_idle や_wake_up などのサンプル数が非常に多い。これは、次のような挙動が原因になっていると推測される。diskio が書き込みを始めるとすぐ I/O 待ちとなり、CPU を明け渡す。HDD が書き込みを終えると、割り込みをかける。割り込みの結果、__wake_up が呼び出され、次の書き込みを行う。そして、再度 I/O 待ちとなる。このような繰り返しを行っているため、CPU は多くの場合アイドル状態となる。このように CPU をほとんど使用していなくても DBT-3 の結果と相関があることから、DBMS の性能と Storage の性能の間に非常に強い相関があることを示している。

今回、fsync、fdatasync と osync、odsync との間では書き込み時間に差はあったが、OProfile では特に異なった傾向を見出すことはできなかった。Linux カーネルが OS の処理の中で、同期書き込みにかかる時間は非常に短い。OProfile で同期書き込みを分析し、ボトルネックを詳細に検出するには、さらに長時間のベンチマークの実行が必要であることがわかった。

get_gendisk によるボトルネックであるが、Linux カーネル 2.6 では、リスト構造ではなく配列を使用してダイレクトにアクセスするようになっており、この部分のボトルネックは無くなっていると推測される。

今後の課題として、より長時間のベンチマークの実行と、Linux カーネル 2.6 で diskio を OProfile 付きで実行し、実際に get_gendisk 関連のオーバーヘッドが改善されていることを確認することがあげられる。

4.1.4 diskio と性能限界との関係

2.1.6.5 項で述べたように、PostgreSQL の WAL に対する書き込み処理は、PostgreSQL 側で排他制御されており、同時に 1 つの同期書き込み要求しか OS に対して発行されない。よって、マイクロベンチマークを用いて OS の性能限界を調査する場合には、同時に 1 つの同期書き込み要求を行う、という条件を満たしている必要がある。

diskio はこの条件を満たしつつ、可能な限り早く OS への同期書き込みを繰り返し行うために、1 つのプロセスが繰り返し同期書き込み要求を OS に対して発行するモデルを採用している。複数プロセスを排他制御しながら同期書き込みを行うと、排他制御機構に伴うオーバーヘッドが生じる。単位時間あたりに OS に発行できる同期書き込み要求量が少なくなってしまう事を考慮した結果、このモデルを利用することになった。

diskio を用いる事によって、PostgreSQL の WAL に対する書き込み処理を適切にエミュレートしつつ、単位時間あたりに発行される同期書き込み要求を最大にした場合について、各書き込み要求の応答時間を測定することができた。WAL の書き込み処理をエミュレートするだけであれば iozone などの他のベンチマークプログラムでもパラメータ設定によって可能であるが、各書き込み要求の応答速度を同時に測定できるためには diskio の開発が不可欠であった。

4.1.5 性能限界の分析と考察のまとめ

4.1.5.1 考察のまとめ

今回の測定では、マイクロベンチマーク diskio と、DBMS ベンチマーク DBT-3 との結果を比較、相関関係の有無について調べた。また、OProfile を用いて diskio 使用時のカーネル内部プロファイリングを実施し、同期書き込み時のボトルネックについて探った。diskio の動作については fsync・fdatasync と osync・odsync との間に特徴的な挙動を発見した。diskio の fsync と DBT-3 の結果には正の相関が見られた。また、diskio の async と DBT-3 の結果には負の相関が見られた。しかし、今回は測定環境が 2 つと少なく、diskio と DBT-3 の相関係数を導き出すまでに至らなかった。

OProfile を併用した diskio 実行の結果、同期書き込み時のボトルネックをいくつか発見できた。しかし、異なる同期方式におけるボトルネックの違いを検出することはできなかった。

4.1.5.2 今後の課題

今後の課題としては以下の 3 項目が挙げられる。

diskio と DBT-3 との相関係数を明らかにするには、様々な環境で diskio を実行する必要がある。その結果、ある程度簡易に DBT-3 の結果が予測できる。

OProfile を用いた検証で解明されたボトルネックには smp カーネルに由来するものがあった。シングル CPU 用カーネルでの同期書き込みのボトルネックが何なのかについては不明確なままである。これは、OProfile に対応しているカーネルが smp カーネルのみという環境上の制約である。今後、OProfile がシングル CPU 用カーネルに対応すれば、シングル CPU 用カーネルのボトルネックについても解明できる。

4.2 iozone の結果

4.2.1 概要

今回、ファイルサイズ (-s オプション)、並列数 (-t オプション)、子プロセスまたはスレッド (-T オプション)、iozone コマンド数を変化させた 11 種類の I/O パターン(表 3.2-1) の測定を実施した。その結果、大きく分けて 3 種類に分類(表 4.2-1)できることがわかった。この分類はスループット結果(図 4.2-1)とプロファイリング結果とで共通だった。

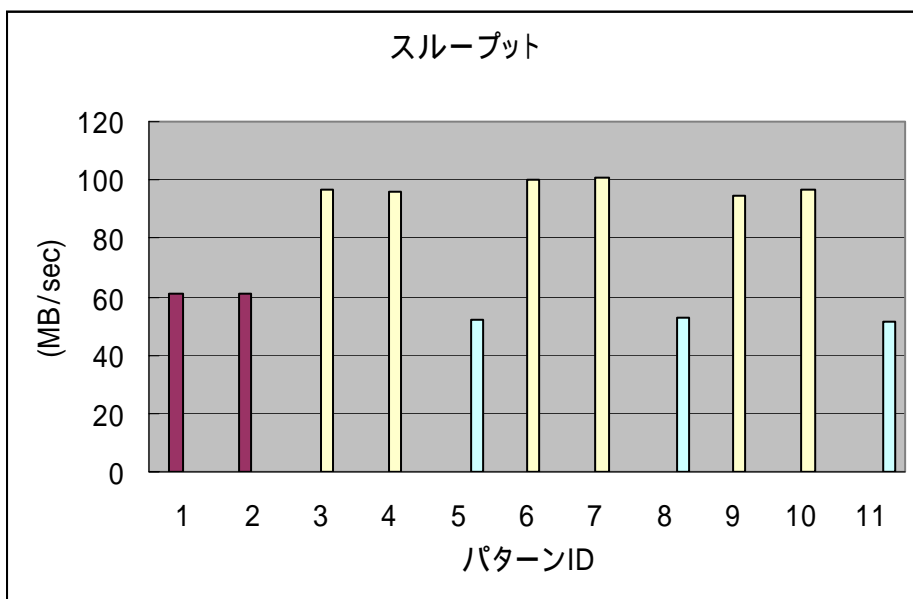


図 4.2-1 スループット結果

表 4.2-1 I/O 結果の分類

プロファイリング結果	スループット結果	I/O パターン ID
CPU ビジー型	良好	3,4,6,7,9,10
CPU アイドル型	やや悪い	1,2
ロック競合型	悪い	5,8,11

CPU ビジー型は、iozone プロセスからページキャッシュへのデータのコピー処理がボトルネックになっているケースであり、最も多く見られたパターンである。優れたスループット結果に現れている通り、非常にスムーズに I/O 処理が行われた状態にある。ボトルネックはセグメント間のメモリコピーであるが、これについては後ほど考察する。

CPU アイドル型は、8GB のファイルを単一の I/O で作成する場合に発生しており、これはデバイス(ハードディスクやディスクコントローラ)がボトルネックとなり、CPU に待ち状態が発生したケースと考えられる。この場合は、ハードディスクやコントローラの交換などによるデバイスの高速化が最も効果的なボトルネック解消方法といえる。

ロック競合型は、グローバルなカーネルロックの競合が発生したため、CPU に待ち状態が発生したケースであり、iozone コマンドを複数起動した際に観測された。具体的にカーネルのどの部分のカーネルロックが競合したかについてを OProfile と LKST のデータを解析することから導き出す方法を紹介する。

これら 3 種類のタイプについて、それぞれパターンをひとつずつピックアップして分析手法を紹介してその結果を述べ、その後今回の評価における安定性についても述べる。

なお、測定に関しては時間的な制約のため、同一パターンの測定を複数回繰り返してデータの誤差や分散を求めるなどの評価は実施していない。

子プロセスとスレッドとの比較については、プロファイリング結果・スループット結果ともに大きな差が見られないため省略する。

また、ランレベルを 5 および 3¹²それぞれに設定した状態での測定も実施したが、スループットに優位な差が現れなかったため(図 4.2-2)、この観点での分析も省略する。

$$\text{差異} = (\text{ランレベル 3 の結果} - \text{ランレベル 5 の結果}) / \text{ランレベル 5 の結果}$$

¹² ランレベル 5 は X ウィンドウが稼動している状態、ランレベル 3 は X ウィンドウが稼動していない状態であり、それ以外の条件は同一である。

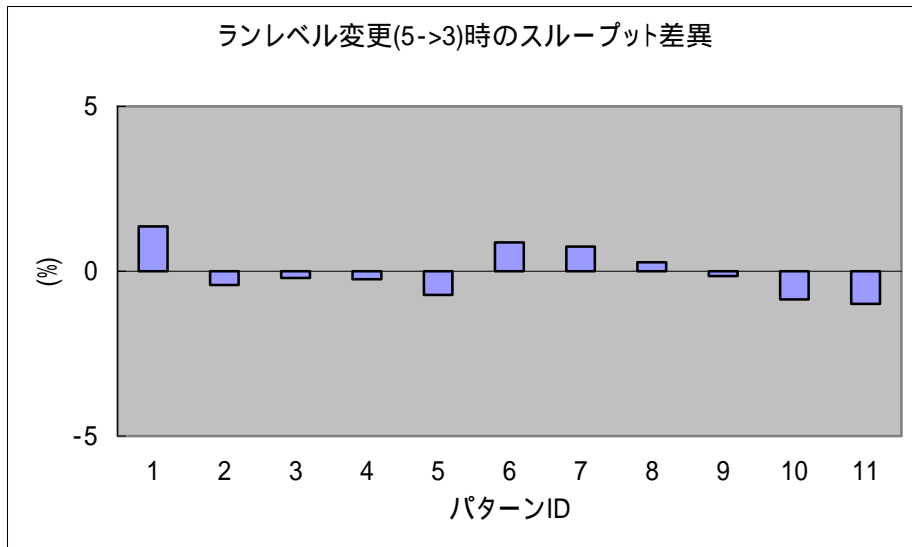


図 4.2-2 ランレベル間の比較

ただし、このデータもすべて1回ずつの測定である。

4.2.2 CPU ビジー型

CPU ビジー型については、パターン ID 9 を取り上げる。

表 4.2-2 条件

パターン ID	オプション -s	オプション -t	オプション -T	iozone コマンド数
9	2G	4	x	1

表 4.2-3 Iozone 結果

所要時間(Wall time)	89.446 秒
ファイルサイズ合計	8,388,596.00kB
スループット合計	94,740.99 kB/sec

表 4.2-4 プロファイリング全体結果

順位	シンボル	サンプル数	占有率(%)	累積占有率(%)
1	do_generic_file_write	8,462	16.16	16.16
2	get_hash_table	4,584	8.754	24.91
3	unlock_buffer	3,364	6.424	31.34
4	__br_write_lock	2,763	5.276	36.61
5	__write_lock_failed	1,989	3.798	40.41
6	__br_write_unlock	1,516	2.895	43.31
7	__free_pages_ok	1,287	2.458	45.76
8	rmqueue	1,190	2.272	48.04
9	get_unused_buffer_head	1,140	2.179	50.21
10	.text.lock.sched	1,122	2.143	52.36

do_generic_file_write を細分化した結果が表 4.2-5 の通り。

表 4.2-5 プロファイリング do_generic_file_write 詳細

アドレス	サンプル数	do_generic_file_write に対する占有率(%)	全体に対する占有率(%)	命令
0xc0160de5	1	0.011818	0.00191	shr \$0x2,%ecx
0xc0160de8	7552	89.246	14.4213	repz movsl %ds:(%esi),%es:(%edi)

0xc0160dea	358	4.23068	0.683637	mov %eax,%ecx
0xc0160dee	95	1.12267	0.181412	mov %ecx,%ebx
0xc0160df0	1	0.011818	0.00191	mov 0x10(%esp,1),%edi

命令は、objdump コマンドを使用して確認した。

```
# objdump -d /boot/vmlinux-2.4.21-9.38AXsmp --start-address=0xc0160de5 \
--stop-address=0xc0160df4
```

4.2.2.1 考察

アドレス 0xc0160de8 の命令が最も頻度が高い結果となった。

ソースコードとの突合せを行った結果、この処理は do_generic_file_write() の中で呼ばれている __copy_from_user() であり、__copy_user_zeroing() にマクロ展開される。

```
ssize_t
do_generic_file_write(struct file *file,const char *buf,size_t count, loff_t *ppos)
{
...
        status = mapping->a_ops->prepare_write(file, page, offset,
offset+bytes);
        if (status)
            goto sync_failure;
        page_fault = __copy_from_user(kaddr+offset, buf, bytes); <-ここ
        flush_dcache_page(page);
        status = mapping->a_ops->commit_write(file, page, offset,
offset+bytes);
...
}
```

```
#define __copy_user_zeroing(to,from,size) \
do { \
    int __d0, __d1; \
    __asm__ __volatile__( \
        "0:    rep; movsl\n" \
        "      movl %3,%0\n" \
        "1:    rep; movsb\n" \
... \
        <-ここ \
    ) \
}
```

`__copy_from_user()` は、ファイルに書き込もうとしているユーザ空間のデータをカーネル空間のページキャッシュに書き込む（コピーする）処理である。今回のシステムは CPU キャッシュが合計 1MB なのに対して、書き込みファイルサイズが合計 8GB のため、CPU キャッシュのミスヒットが多発していると考えられる。

このコピーは I/O の書き込み処理の中でも最も中心といえる処理であり、これがプロファイリングのトップにあるということは書き込み処理に効率よく CPU が費やされていることを表している。厳密な意味では、この `__copy_from_user()` が呼ばれる回数自体に無駄がないかどうかなど、さらなる追及の余地があるが今回の評価ではそこまでは及ばなかった。

4.2.3 CPU アイドル型

CPU アイドル型については、パターン ID 1 を取り上げる。

表 4.2-6 条件

パターン ID	オプション -s	オプション -t	オプション -T	iozone コマンド数
1	8G	1	x	1

表 4.2-7 Iozone 結果

所要時間	138.042 秒
サイズ	8,388,608.00 kB
スループット	60,768.41 kB/sec

表 4.2-8 プロファイリング全体結果

順位	シンボル	カウント	占有率	累積占有率
1	default_idle	1,993	24.37	24.37
2	try_to_free_buffers	532	6.504	30.87
3	launder_page	532	6.504	37.38
4	LKST_ETYPE_MEM_SWAPOUT_HEADER_hook	371	4.536	41.91
5	unlock_page	365	4.462	46.37
6	__remove_inode_queue	351	4.291	50.67
7	rebalance_laundry_zone	281	3.436	54.10
8	__wake_up	218	2.665	56.77
9	write_some_buffers	198	2.421	59.19
10	.text.lock.sched	192	2.347	61.54

default_idle をブレークダウンした結果は表 4.2-9 の通り。

表 4.2-9 プロファイリング default_idle 詳細

アドレス	サンプル数	default_idle に対する占有率(%)	全体に対する占有率(%)	命令
0xc0109254	3	0.1505	0.03668	je 0x0109280

0xc0109260	1	0.05018	0.01223	movl 0x14(%edx),%eax
0xc0109268	1945	97.59	23.78	hlt
0xc0109269	44	2.208	0.5380	ret

4.2.3.1 考察

プロファイリング結果より、CPU は hlt 命令による休止状態にあることが多かったことが分かる。これは、I/O リクエストが処理されるスピードよりも、CPU を使ったデータコピー処理のスピードのほうが速かったため、I/O リクエストが累積して CPU 側が待たされていた状態にあったと考えられる。

そこで、実際にどれだけの I/O リクエストの蓄積があったのかを LKST によって調査した。マスクセットとして buffer と blkqueue を使用して測定し、CPU アイドル型パターンと CPU ビジー型パターンとの比較を行った。

始めに I/O リクエスト到着から I/O 完了までの時間を測定した buffer の結果を比較する(表 4.2-10)。参考までに Write だけでなく Read も掲載する。

表 4.2-10 I/O リクエスト到着から I/O 完了までの時間 (buffer)

		カウント	平均(秒)	最大(秒)	最小(秒)
CPU アイドル型(パターン 1)	Read	19	8.094	152.5	0.002402
	Write	247,076	1.218	128.1	0.002283
CPU ビジー型(パターン 9)	Read	17	1.966	18.56	0.002406
	Write	130,531	0.9150	36.33	0.004309

これより、I/O リクエストの処理時間に大きな差があったことがわかる。特に、CPU アイドル型は最大値が大きく Write では CPU ビジー型の 3.5 倍に達する。これは、一時的に I/O が滞って長時間待たされた現象が発生したことを表している。

なお、パターン 1 とパターン 9 とでカウント数が異なっているのは 2 つの理由がある。LKST がログを格納するバッファサイズが限られているため、バッファがいっぱいになるたびにこれまでのログデータを消して新たにログの蓄積を始める。このことから、LKST 終了のタイミングによって取れるログの量が異なるのがひとつの理由である。もうひとつは、スループットが異なるパターン同士で単位時間当たりのイベントの種類とその発生頻度が一致しないため、仮に等しい時間分のデータを取得できたとしても含まれるイベント数は一致しないのが理由である。

また "lkstla buffer -d" で出力される対数時間分布は表 4.2-11 の通りである。

表 4.2-11 buffer 所要時間の対数分布

パターンID	0.001 未満	0.001 秒	0.01 秒	0.1 秒	1 秒	10 秒以上	合計
CPU アイドル型(パターン1)	0	8	1,069 (0.433%)	45,797 (18.5%)	200,138 (81.0%)	64	247,076
CPU ビジー型(パターン9)	0	1,024 (0.784%)	8,163 (6.25%)	65,325 (50.0%)	55,925 (42.8%)	94	130,531

ただし、10 秒以上がひとつの列にまとめられているが、先の最大値から明らかなように、パターン 1 の場合は 100 秒のオーダも発生しており、パターン 9 ではそれが発生していないことに注意されたい。

頻度を回数から割合に変換してグラフ化すると図 4.2-3 のようになる。CPU アイドルであるパターン 1 は CPU ビジーであるパターン 9 に比べて所用時間のオーダが 1 近く長いことが分かる。それだけ I/O リクエストの処理が相対的に遅かったことが見てとれる。

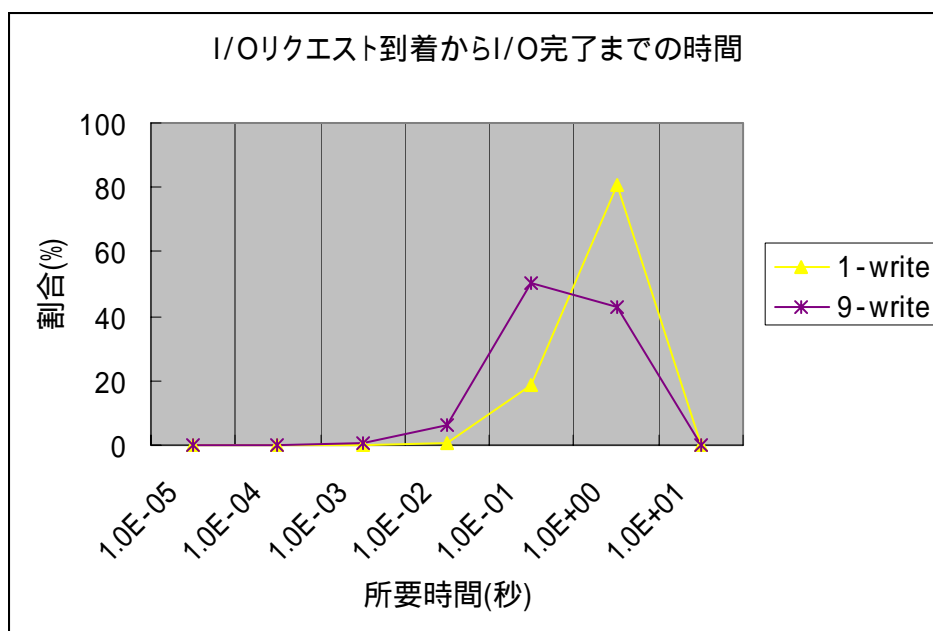


図 4.2-3 I/O リクエスト到着から I/O 完了までの時間 (buffer)

次に、I/O 要求が来た時のリクエストキューの長さ(blkqueue)についても比較する。

リクエストキューはファイルシステムごとに分かれているため、パターン 1(CPU アイドル型)は 1 個、パターン 9(CPU ビジー型)は 4 個である。なお、キューの長さの最大値は 512 である。それ以上のリクエストが来た時は、リクエストしたプロセスが Wait キューに属して休止状態に入り、I/O リクエストキューに空席ができるのを待つ。

表 4.2-12 I/O 要求が来た時のリクエストキューの長さ(blkqueue)

	カウント	平均	最大	最小
CPU アイドル型 (パターン 1)	30,435	464.8	512	1
CPU ビジー型 (パターン 9)	4,802	116.0	491	1
	6,615	158.3	504	1
	5,051	118.5	451	1
	5,616	140.4	512	1

平均値についてグラフ化すると図 4.2-4 のようになる。

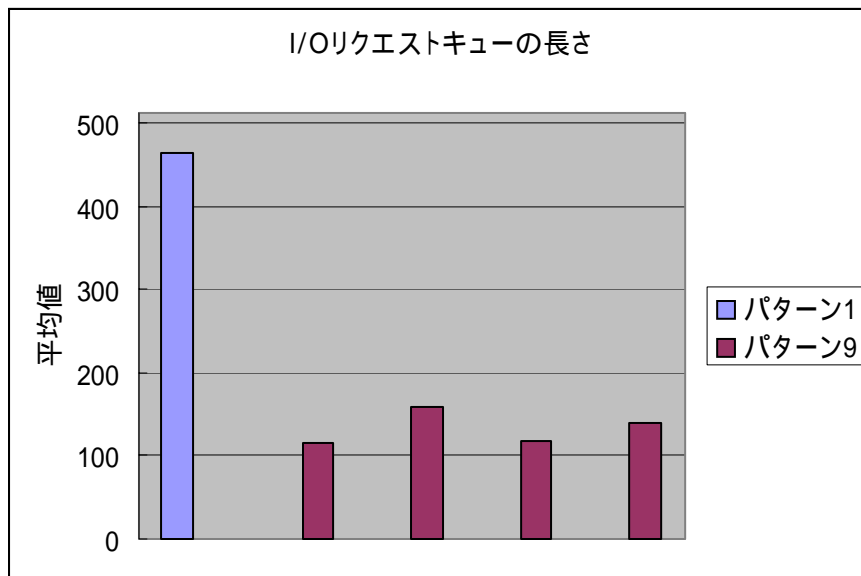


図 4.2-4 I/O 要求が来た時のリクエストキューの長さ(blkqueue)

パターン 1(CPU アイドル型)はディスクが 1 台の I/O、パターン 9(CPU ビジー型)はディスクが 4 台の I/O である。パターン 1 の平均値が非常に高く、常に I/O リクエストがほぼ上限(512)近くまで溜まっていたことが見てとれる。これはデバイスドライバのルーチンによる I/O リクエストの処理が飽和状態にあったことを表しており、

一般的にデバイス側の処理限界に突き当たっていることが多い。
この比較は、ディスクコントローラの条件が同一で、ディスクドライブの数が異なるケース同士の比較であるため、ディスクドライブの転送速度の上限に達していると推定される。また、スラッシングのようなヘッドの頻繁な動きで効率の悪い状態が発生したケースも考慮に入れなければならないが、Iozone の書き込みパターンが同一であるにも関わらずこのようなパフォーマンス差が出ていることからして、その影響は少ないと考えられる。また、エレベーターアルゴリズムについても同様であり、これについてはプロファイリング結果にキューイングする関数 (elevator_linus_merge など) のエントリが上位にないことから否定される。

4.2.4 ロック競合型

ロック競合型については、パターン ID 5 を取り上げる。

表 4.2-13 条件

パターン ID	オプション -s	オプション -t	オプション -T	iozone コマンド数
5	4G	1	x	2

表 4.2-14 Iozone 結果

	コマンド 1	コマンド 2	合計
所要時間	164.236 秒	159.397 秒	
ファイルサイズ	4,194,304.00 kB	4,194,304.00 kB	8,388,608 kB
スループット	25,538.34 kB/sec	26,313.62 kB/sec	51,851.96 kB/sec

表 4.2-15 プロファイリング全体結果 (LKST なし)

順位	シンボル	サンプル数	占有率(%)	累積占有率(%)
1	.text.lock.ioctl	12,454	26.30	26.30
2	.text.lock.buffer	8,362	17.66	43.97
3	do_generic_file_write	3,804	8.034	52.00
4	default_idle	2,027	4.281	56.28
5	unlock_buffer	2,012	4.250	60.53
6	get_hash_table	1,743	3.681	64.21
7	__br_write_lock	884	1.867	66.08
8	.text.lock.sched	679	1.434	67.51
9	rmqueue	677	1.430	68.94
10	__brelse	621	1.312	70.26

ロックについては、プロファイリング結果だけでは解析が難しいため、LKST を併用した測定を実施した。ロックについての情報を取得するため、busywait と spinlock のみを有効にしたマスクセットを使用して LKST を実行した。

表 4.2-16 プロファイリング全体結果 (LKST あり)

順位	シンボル	サンプル数	占有率(%)	累積占有率(%)
1	.text.lock.ioctl	10,310	28.74	28.74

2	.text.lock.buffer	8,257	23.02	51.76
3	default_idle	2,166	6.038	57.80
4	do_generic_file_write	2,048	5.709	63.50
5	unlock_buffer	1,057	2.947	66.45
6	get_hash_table	967	2.696	69.15
7	__br_write_lock	526	1.466	70.61
8	__make_request	502	1.399	72.01
9	rmqueue	457	1.274	73.29
10	__brelse	405	1.129	74.46

LKST を使用した際と使用しない際とを比較すると、第 3 位と第 4 位とが入れ替わっている点、および第 8 位のシンボルが異なる点以外は全く一致している。同一条件の測定においてさえ第 3 位以降は試行毎にずれた結果が得られることもある¹³ため、LKST の有無によるこの違いは測定誤差の範囲にあると考えられる。そのため、LKST による測定結果の解析結果が LKST を使用しない環境での現象説明にも適用可能であるという立場に基づいて、以下では LKST を使用した際の（プロファイリングおよび LKST）測定結果を分析する。

4.2.4.1 考察

まず始めにプロファイリング結果を解析する。`.text.lock.ioctl` をブレークダウンした結果は表 4.2-17 のようになる。

表 4.2-17 プロファイリング `.text.lock.ioctl` 詳細

アドレス	サンプル数	<code>.text.lock.ioctl</code> に対する占有率(%)	全体に対する占有率(%)	命令
0xc019f919	307	2.978	0.8558	<code>cmpb \$0x0, 0xc0405400</code>
0xc019f920	407	3.948	1.135	<code>repz nop</code>
0xc019f922	9,596	93.07	26.75	<code>jle 0xc019f919</code>

命令は、`objdump` コマンドを使用して確認した。

```
# objdump -d /boot/vmlinux-$(uname -r) --start-address=0xc019f919 \
--stop-address=0xc019f924
```

¹³ 第 1 位と第 2 位との差小さく接近したパターンにおいては、測定のたびに第 1 位が入れ替わることすらある。

```

/boot/vmlinux-2.4.21-9.38AX.lkst6smp:      file format elf32-i386

Disassembly of section .text:

c019f919 <.text+0x9f919>:
c019f919:      80 3d 00 54 40 c0 00      cmpb  $0x0,0xc0405400
c019f920:      f3 90                    repz  nop
c019f922:      7e f5                    jle   0xc019f919
Disassembly of section .entry.text:
Disassembly of section .text.init:
#

```

ソースコードとの突合せを行った結果、これはグローバルなカーネルロックの取得を待つビジーウェイトのコードであり、システムコール `ioctl()` から呼び出されたものであった。

```

asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
    struct file * filp;
    unsigned int flag;
    int on, error = -EBADF;

    filp = fget(fd);
    if (!filp)
        goto out;
    error = 0;
    lock_kernel();      <----- このロック
    switch (cmd) {

```

同様に `.text.lock.buffer` をブレークダウンした結果は表 4.2-18 の通り。

表 4.2-18 プロファイリング `.text.lock.buffer` 詳細

アドレス	サンプル数	.text.lock.buffer に対する占有率 (%)	全体に対する占有率 (%)	命令
0xc018bcae	359	4.34783	1.00075	<code>cmpb \$0x0, 0xc0405400</code>
0xc018bcb5	4,509	54.6082	12.5693	<code>repz nop</code>

0xc018bcb7	3,315	40.1478	9.24093	jle 0xc018bcae
------------	-------	---------	---------	----------------

これは、上の.text.lock.ioctlのカーネルロックと競合するビジーウェイトのコードで、sync_old_buffers()の冒頭で呼ばれている。

```
static int sync_old_buffers(void)
{
    lock_kernel();    <----- このロック
    sync_unlocked_inodes();
}
```

OProfileによるプロファイリング結果は、この2箇所のグローバルロックに費やされるCPU時間が非常に長かったことを表している。しかし、このデータは、命令の実行にのみ注目した統計情報であるため、ロックに関する情報は合計時間しか取得できず、このロックが実際にその発生時にどのような競合状態であったかの把握できない。それについてはLKSTのデータから詳しく知ることができる。

そこで、ビジーウェイトとスピンロックのマスクセットを使用したLKSTデータを解析する。

まず始めに“lkstla busywait -s”で得られる統計データから、ロックを取得するアドレス別にビジーウェイト時間が長いものをリストアップする。

表 4.2-19 ビジーウェイト時間

順位	アドレス	回数	平均(秒)	最大(秒)	合計(秒)	割合(%)
1	0xc019f83e	85	0.3777	16.14	32.1	41.1
2	0xc01977d1	1,028	0.02984	15.51	30.7	39.3
3	0xc018ad07	4	3.786	15.14	15.1	19.4
4	0xf8863ba3	61,477	1.8E-07	2.49E-05	0.0111	0.0142
5	0xf88637bb	61,476	1.79E-07	7.42E-06	0.0110	0.0141

上位3エンタリで殆どすべてのビジーウェイト時間を占めている。ちなみに、このアドレスは、LKSTのフックポイントのアドレスであるため、同一のロックのコードを表す場合でもプロファイリングデータのアドレスとは(近傍ではあるが)異なる値になる。

それぞれの対数集計結果(lkstla busywait -d)は表 4.2-20のとおり。

表 4.2-20 ビジーウェイト時間の対数分布

アドレス	未 満	10 ⁻⁷ 秒	10 ⁻⁶ 秒	10 ⁻⁵ 秒	10 ⁻⁴ 秒	10 ⁻³ 秒	10 ⁻² 秒	10 ⁻¹ 秒	10 ⁰ 秒	以 上
0xc019f83e	0	82	1	0	0	0	0	0	0	2
0xc01977d1	0	963	48	11	3	1	0	0	0	2
0xc018ad07	0	2	1	0	0	0	0	0	0	1

10 秒以上のオーダーのビジーウェイトが計 5 回発生しており、これがほぼ全体を占める。

LKST のデータを以下の手順で調査したところ、2 回の大きなロック競合が確認できた。

3.2.4 節の通りに手動実行した場合はカレントディレクトリに、スクリプト run.sh を利用した場合はそれぞれのパターンのディレクトリの下に、LKST のバッファデータ lkst.data が作成されている。このバッファデータを時系列データとして読み出した上で、注目するアドレスで grep する。

```
# lkstbuf print -f lkst.data -r -C | grep -E \  
  '0xc019f83e|0xc01977d1|0xc018ad07' > /tmp/a  
# vi /tmp/a
```

この出力の読み方は表 4.2-21 の通り。

表 4.2-21 スピンロック時系列データの読み方

第 1 列	第 2 列	第 3 列	第 4 列	第 5 列	第 6 列	第 7 列	第 8 列	第 9 列
イベント	cpu	pid	曜日	月	日	時刻	年	“call address”

第 10 列	第 11 列	第 12 列	第 13 列	第 14 列	第 15 列	第 16,17 列
call アドレス	0x0	“lock”	ロックのアドレス	0x0	“start time”	開始時刻 (16 進数)

この出力の中の第 7 列 (時刻) に注目し、不自然に長い時間間隔がある時刻のエントリ(ロック競合の発生場所と考えられる)を探し出す。

```
第 1 列      第 3 列      第 5 列      第 7 列      第 9 列      第 13 列  
            第 2 列      第 4 列 第 6 列      第 8 列      (中略)      (後略)  
"spin_lock",00,00000629,Fri,Feb,04,10:59:59.109847544,2005,"call address",...,0xc0405400,...
```

```
"spin_lock",00,00000629,Fri,Feb,04,10:59:59.109883712,2005,"call address",...,0xc0405400,...
"spin_lock",01,00000565,Fri,Feb,04,11:00:15.543853871,2005,"call address",...,0xc0405400,...
"spin_lock",00,00000629,Fri,Feb,04,11:00:15.543862258,2005,"call address",...,0xc0405400,...
"spin_lock",01,00000014,Fri,Feb,04,11:00:15.543964850,2005,"call address",...,0xc0405400,...
...
```

次に、再び lkst.data を時系列で読み出して、問題のエントリのロックアドレス(第 13 列)で grep し、問題の時刻の近傍を抽出する。

```
# lkstbuf print -f lkst.data -r -C | grep 0xc0405400 > /tmp/b
# vi /tmp/b
```

第 1 列	第 3 列	第 5 列	第 7 列	第 9 列	第 11 列
	第 2 列	第 4 列	第 6 列	第 8 列	第 10 列 (後略)
"spin_lock"	,00,00000629,	Fri,Feb,04,	10:59:59.109884679,	2005,"call address",	0xc026f829,0x00000000,...
"spin_unlock"	,00,00000629,	Fri,Feb,04,	10:59:59.109884863,	2005,"call address",	0xc019f4a3,0x00000000,...
"spin_unlock"	,03,00001305,	Fri,Feb,04,	11:00:15.543851385,	2005,"call address",	0xc0129fb8,0x00000000,...
"spin_lock"	,01,00000565,	Fri,Feb,04,	11:00:15.543853871,	2005,"call address",	0xc01977d1,0x00000000,...
"spin_unlock"	,01,00000565,	Fri,Feb,04,	11:00:15.543860220,	2005,"call address",	0xc0197778,0x00000000,...
"spin_lock"	,00,00000629,	Fri,Feb,04,	11:00:15.543862258,	2005,"call address",	0xc019f83e,0x00000000,...
"spin_unlock"	,00,00000629,	Fri,Feb,04,	11:00:15.543868558,	2005,"call address",	0xc026f897,0x00000000,...
...					

Call アドレス(第 10 列)から、どの関数においてロックを取得・解放していたのかが分かる。アドレスと関数とのつきあわせは、先に説明した objdump コマンドや lcrash コマンドから行うことができるが、アセンブリ言語の知識が若干必要とされる。

```
# lcrash /boot/System.map-$(uname -r) /dev/mem /boot/Kerntypes-$(uname -r)
...
>> dis 0xc01977d1 5
```

以上の方法を繰り返して、合計 2 箇所のロック競合を突き止めた。

一つ目のロック競合を時系列で表に整理すると表 4.2-22 のようになる。

表 4.2-22 1 回目のロック競合の時系列

時刻	cpu	pid	コマンド名	lock/unlock	関数
----	-----	-----	-------	-------------	----

10:59:59.109884679	0	629	kdm_greet	lock	
10:59:59.109884863	0	629	kdm_greet	unlock	
11:00:15.543851385	3	1305	iozone	unlock	schedule()
11:00:15.543853871	1	565	crond	lock	permission()
11:00:15.543862258	1	565	crond	unlock	
11:00:15.543862258	0	629	kdm_greet	lock	sys_ioctl()
11:00:15.543868558	0	629	kdm_greet	unlock	

ひとつめのケースでは、iozone タスクから呼び出された schedule()関数内でのロックの解放が遅かったため、その間、crond が permission()中の lock_kernel()で待ち、kdm_greet が sys_ioctl()中の lock_kernel()で待っていた。Iozone がグローバルロックを解放すると始めに crond がロックの取得・解放をおこない、次に kdm_greet が取得・解放を行った。

なお、コマンド名は、実行していた top コマンドのログを pid で検索してつきとめた。

二つ目のロック競合は表 4.2-23 の通り。

表 4.2-23 2 回目のロック競合の時系列

時刻	cpu	pid	コマンド名	lock/unlock	関数
11:00:19.099339745	1	629	kdm_greet	lock	
11:00:19.099340359	1	629	kdm_greet	unlock	
11:00:35.693422719	2	1304	iozone	unlock	ext3_delete_inode()
11:00:35.693424252	0	14	kupdated	lock	sync_old_buffers()
11:00:35.693460331	0	14	kupdated	unlock	
11:00:35.693464099	1	629	kdm_greet	lock	sys_ioctl()
11:00:35.693470122	1	629	kdm_greet	unlock	
11:00:35.693480961	3	1	init	lock	permission()
11:00:35.693489213	3	1	init	unlock	

このケースでは、iozone タスクから呼び出された ext3_delete_inode()関数内でのロックの解放が遅かったため、その間、kupdated が sync_old_buffers()冒頭の lock_kernel()で待ち、kdm_greet が sys_ioctl()中の lock_kernel()で待ち、さらに init プロセスが permission()中の lock_kernel()で待たされていた。Iozone がロックを解放した後、kupdated, kdm_greet, init の順でロックを取得・解放した。

ただし、どちらのケースも iozone がロックを取得した際のログを LKST が記録していない。これは、今回の LKST の測定方法ではログの容量が十分ではなく、数十秒ごとに上書きされてそれ以前のデータが消去されてしまうため、長い時間に渡って

ロックを保持していた `iozone` の `spin_lock` イベントの情報が失われてしまったのが原因である。現在の LKST の制限事項の一つといえる。

グローバルロック `lock_kernel()` の競合 (ビジーウェイト) が `sys_ioctl()` と `sync_old_buffers()` とで長く発生したという結果は OProfile の結果と一致する。

今回のベンチマーク構成では、並列な I/O の書き込み先はすべて別々のディスクに分かれているため、ディスク間のコンフリクトは無い。ここで観測された競合は、プロセス間だけのコンフリクトに因るものと考えられる。

このロック競合は `iozone` コマンドを複数同時実行することによって発生しているが、それに対してひとつの `iozone` コマンドから複数の子プロセスが `fork()` されたケースでは CPU ビジー型となって、ロック競合型にはならない。どちらのケースも `fork()` されたプロセスがそれぞれ 1 つずつの I/O を担当し、グローバルロック `lock_kernel()` を繰り返しているが、大きな競合の有無 (およびスループット結果) に違いが現れた。この差がなぜ起こるのかについての究明には到らなかった。

グローバルなカーネルロックは、これまでにカーネルの改良によってその数を減らしてきたが、MIRACLE LINUX V3.0 にもまだある程度残っている。2005 年 1 月現在でのコミュニティの最新のカーネル 2.6.10 を確認すると、さらに幾つかのグローバルロックが削除されている。これらについて MIRACLE LINUX でも削除を検討することが可能と考えられるため、今後の課題のひとつとしたい。

表 4.2-24 グローバルなカーネルロック

関数	MIRACLE LINUX V3.0 での位置		2.6.10 カーネルでの状態
<code>sys_ioctl()</code>	<code>fs/ioctl.c</code>	->	残留
<code>sync_old_buffers()</code>	<code>fs/buffer.c</code>	->	<code>sync_old_buffers()</code> がない
<code>permission()</code>	<code>fs/namei.c</code>	->	削除
<code>schedule()</code>	<code>kernel/sched.c</code>	->	残留
<code>ext3_delete_inode()</code>	<code>fs/ext3/inode.c</code>	->	削除

4.2.5 高負荷時の信頼性

OS の限界付近での動作評価として、Iozone ベンチマークを実施した際の性能面での評価を前節までに報告した。本節では、評価のもうひとつの観点として信頼性について報告する。

4.2.5.1 評価中の状況

Iozone 単体の実行でシステムが不安定になることは一切無かったが、Iozone と OProfile との併用においてシステムがフリーズする現象が時折確認された。これは実施した 11 種類の I/O パターンのうちの特定のパターンで発生したのではなく、また同じ条件で再実行しても必ずしも再現せず、非常に無作為な発生状況であった。このフリーズは、システムの応答が無くなるものであるが、ランレベル 5 の場合にはログイン画面が表示されたまま凍りついた状態にあった。この状態ではシステムリクエストキーが効かず、強制的にクラッシュさせることは不可能だった。また、2 度ほどクラッシュの発生もあった。ただし、この時のダンプ解析は実施していない。

短時間に多くの測定を繰り返す時ほど発生頻度が高かった。自動的にシステムリブートと測定とを繰り返すようにシステムを設定したところ、多い時で 3 回に 1 回程度の割合でシステムがフリーズした。しかし、長いインターバルを置いて測定すると（例えば、1 日に 2, 3 回程度の実行では）殆ど発生しなかった。

また、デモンストレーションのために準備したノート PC 環境では 5 回以上の測定を実施したが、フリーズ、クラッシュともに一度も発生しなかった。

4.2.5.2 分析

OProfile を併用して初めて発生したことから、原因として考えられるのは、ひとつには OProfile のモジュールおよびそれに関連するカーネルの不具合が考えられる。もうひとつは、ノート PC 上での発生がなかったことから、ハードウェア（およびそのデバイスドライバ）の問題である。

システムが使用不能になったという意味では、今回の評価で限界もしくはそれに近い状態を再現できたと言えるかもしれないが、OProfile モジュール単体の不具合を疑うと、むしろツールのバグにぶつかってしまったと考えられる。

4.2.5.3 対応方法

今後、以下のような調査によって原因を明らかにしたい。

- クラッシュダンプ解析
- 別なハードウェア(ディスク、コントローラ)上での再現確認
- OProfile をバージョンアップした上での再現確認

4.2.5.4 結び

今回の評価では、高負荷時の信頼性を判断する以前に、評価ツールの不具合が引き起こしたと推測される不安定が現れた。性能限界と信頼性限界とを同一の実行で再現することは今回の評価環境においては達成できなかった。Iozone 単体で性能限界状態を再現した場合にシステムが安定していたことから、性能限界付近での信頼性は過去の(不安定だった頃の)Linux に比較して向上していると考えてもよいだろう。

4.2.6 評価工数についての考察

今回の手法を実際の現場に適用する場合、この評価に要する工数がひとつ注目される点になる。例えどれほど平易な手法であってもそれが多大な工数を要するものであっては、それが現実的に使い物になるとは言い難い。この観点から、プロファイリングで評価する際に、得られたデータのうちのどこまでを解析すべきなのかどうかを考える。今回の例では、上位 1 または 2 エントリのみを解析を以ってボトルネックの調査としたが、現実のシステムを診断する場合には、より広範囲に渡る解析が必要になると予想とされる。

そこで OProfile 上位何エントリの把握で全体の何%を網羅できるかについて確認した。今回の 11 パターンのプロファイリング結果におけるエントリ数と累積占有率との関係をグラフにした。

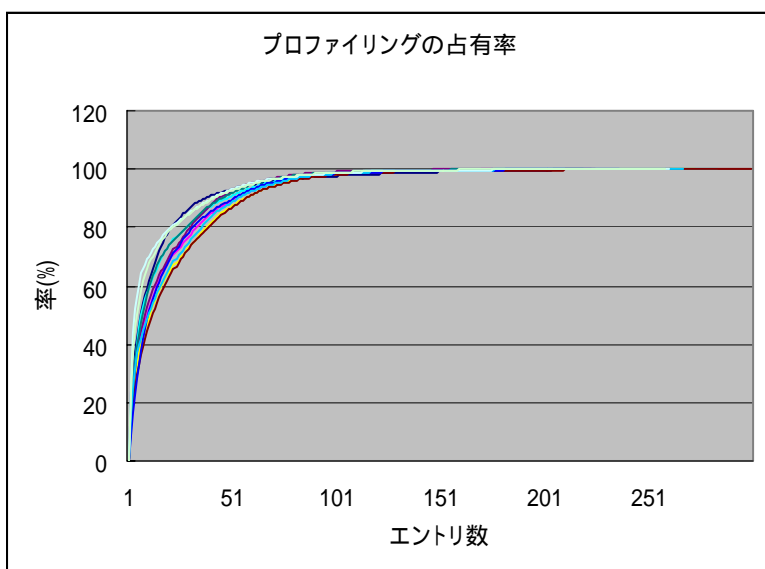


図 4.2-5 プロファイリングのエントリ数と累積占有率

それぞれ全体の 80%を把握するのに必要なエントリは 21~40 エントリで、平均は 31 である。それぞれの全エントリ数に対する割合に換算すると平均約 13%である。言い換えると、上位 13%のエントリを解析することによって、システム全体の 80%を把握できることを意味する。

以上から、プロファイリング結果の上位に絞った解析だけでも、システム全体のボトルネックの把握として十分に期待できると考えられる。

4.2.7 評価手法のまとめ

今回、3種類のタイプを実際に解析した手順を基に、評価手法についてまとめる。評価の手順は次の通りである。

1. 評価目的の設定
2. 前提条件の把握
3. 評価環境の設定
4. プロファイリング評価
5. LKSTによる追加評価
6. 総評価
7. 改善手段の考案と適用

これより個々のステップについて詳しく述べる。

4.2.7.1 評価目的の設定

始めに評価の目的を明確にする。例えば、新製品(マシン、または一部のデバイス)の性能評価であったり、または特定用途で構築することが決まっているシステムのサイジングであったり、あるいは運用中のシステムの性能向上であったりなど。目的によっては、評価の手順が異なる。

運用中のシステムであれば、その対象システムのみでの評価で十分なことが多いが、そうでない場合は、複数のシステムを用意して評価し、それらの結果の比較によってはじめて結論が下せる場合もある。新しいマシンの評価などは概ね相対評価が必要になることが多い。

4.2.7.2 前提条件の把握

評価するシステムの構成と利用方法について、詳細に把握する。この情報に基づいて具体的な評価手順が決定され、また評価結果の解釈が行われる。把握すべき情報は以下の通り。

(ア) ハードウェア構成

CPU (クロックスピード、キャッシュサイズ、個数)

メモリ (容量、スピード)

チップセット (システムバススピード、

ディスク (容量、インタフェース、回転数、キャッシュサイズ、シークスピード、転送スピード、台数)

ディスクコントローラ (種類、インタフェース、RAID、キャッシュサイズ、台数)

ネットワークコントローラ (物理接続の種類、転送速度)

他

(イ) ソフトウェア構成

カーネル

デバイスドライバ (ディスクコントローラの)

ファイルシステム (タイプ、サイズ、数)

アプリケーションとライブラリ (負荷、発行するシステムコール)

他

(ウ) 運用方法

24 時間 365 日無停止型である等

クライアントからのアクセスが多い等

他

4.2.7.3 評価環境の設定

実運用システムを評価する場合や、または導入前だとしてもその用途が明確である場合は、可能な限りその運用構成をそのままに、以降の手順で評価する。実運用での評価結果が最も信頼できる結果となるからである。しかし、開発途上にあるなどの理由で実運用環境の調達が困難であったり、この評価にあてる期間や工数の問題で用意できなかったり、あるいは「データベース性能」「Web サーバ性能」といった汎用的な評価が目的の場合は(実運用ではないという意味において)仮想的な評価環境を用意することになるだろう。また新規デバイスの評価などの場合も仮想的な構成になると考えられる。

仮想的な評価環境を設定する際、多くのケースではベンチマークソフトまたはテストツールを使用することになる。ベンチマークソフトには、広く利用されているため結果にある程度のパブリックな評価を下せるものから、その評価に使用するためだけに作成された限定的なものまであり、どれを選択するかは評価目的から判断する。

今回実施した評価では I/O 処理からシステムの限界状態の再現を目的とするため、Iozone という広く利用されている I/O のベンチマークソフトを選択した。

ベンチマークソフトが決定したら、そのソフトの設定を行う。

Iozone には以下の種類の設定がある。

- 書き込み・読み込みパターン
- 通常ファイル・mmap()
- 同期・非同期
- レコードサイズ
- ファイルサイズ
- stream 数
- iozone コマンドの個数

- 子プロセス・スレッド

今回の評価では、数ある書き込み方式のそれぞれの性能を評価するのではなく限界状態の再現が目的のため、最も基本となる initial write を選択した。同じ理由で mmap() は使用せず、通常のファイル I/O を使用し、レコードサイズもデフォルトのままとした。同期・非同期については、diskio によるマイクロベンチマークで同期処理の評価がされるため、非同期を選択した。大規模システムの動作評価のため、ファイルサイズはメモリサイズと同等の大きなサイズを指定し、また stream 数・iozone コマンドの個数は CPU 数を上限として複数パターンを選択した。子プロセス・スレッドについては両パターンとも選択した。この結果、合計 11 個のパターンが評価対象となった(表 3.2-1)。

4.2.7.4 プロファイリング評価

OProfile でプロファイリングを実施してその結果を解析する。手順は 3.2.3 節で紹介した通り。上位の傾向から今回の評価で明らかになった 3 種類に分類する。解析の詳細手順についてはそれぞれタイプの報告文章で示した。

CPU ビジー型(4.2.2 節)

結果が CPU ビジー型の場合は、上位のシンボル(関数)を上から順に解析し、具体的にどの処理に集中しているのかを把握する。今回の測定ではシンボル do_generic_file_write() が最上位で、この中で呼ばれている __copy_from_user() に処理が集中していた。

CPU アイドル型(4.2.3 節)

結果が CPU アイドル型の場合は、OProfile ではボトルネックを捉えきれないことがあるため、LKST でも測定する。今回の評価では I/O のベンチマークを実施していたので、LKST の buffer と blkqueue のデータを取得し、この結果を分析した。

ロック競合型(4.2.4 節)

結果がロック競合型の場合には、まず OProfile の詳細データからスピン時間の長いロックのコードを(マクロ的に)特定する。次に、LKST の busywait と spinlock のデータを取得して、ミクロで実際にどのような競合が起こっていたのかをつきとめる。LKST のデータを取れる時間は短いため、OProfile の結果と LKST の結果を照合することで確実性を増すことができる。

それ以外

上の 3 種類のどれにもあてはまらない場合、基本的に OProfile の詳細結果を分析して、処理が集中しているコードを特定する。続いて、このコードの処理内容に近いデータを取得する LKST 測定を実施する。

4.2.7.5 LKST による追加評価

プロファイリング結果から、必要であると判断される場合は LKST による評価を実施する。実施手順は 3.2.4 節で紹介した。どのイベントを取得するかはプロファイリング結果、また注目する観点から判断する。ロック競合が発生している場合は busywait や spinlock、I/O 処理待ちが推定される場合は buffer や blkqueue、ネットワーク負荷が高い場合は netsend や netroute、プロセススケジューリングに注目する場合は runqueue や waitqueue や waittime などを選択する。

LKST を実施する場合、通常のコアネルと異なるコアネルを使用するため、LKST の測定と同時に OProfile の測定も実施して、LKST を使用しなかった場合の結果との比較を行い、食い違いがないことを確認する。もしも大きな違いがあった場合は LKST の影響で測定環境が乱されたと考えられるため、LKST の設定を変更して乱れないように工夫するか、もしくは LKST による測定を放棄せざるを得ない。

LKST はオーバーヘッドが大きいので、有効にするマスクセットは必要最小限のものとし、また I/O の評価においては lkstlogd を使用せず、コアネル内バッファを試用するのが望ましい。コアネル内バッファを使用した場合、LKST で測定できる時間が短いため（今回の評価で、概ね数十秒～1 分未満程度）、ベンチマーク所要時間とあまりにかけ離れている場合は、LKST を有効にするタイミングを工夫する必要があるだろう。

LKST による測定結果が得られたら、まず統計情報を確認する。それに基づいて、必要に応じて時系列データから個々のイベントを抽出し、個々のイベントにおいて何が起きていたのかを詳細に把握する。今回の測定では、ロック競合のイベントを詳細に把握した。

4.2.7.6 総評価

以上の手順で得られたプロファイリング結果と LKST の結果から、システムの動作状態およびボトルネックを判断する。この時、始めに設定した評価目的に沿った観点での結論を得ること、また前提条件と矛盾しない結果でなければならない。妥当と思われる結論を得るためには、上の評価を様々な機種、様々なパターンで多大な工数を費やして実施しなくてはならないこともありえる。

4.2.7.7 改善手段の考案と適用

システムの動作が明らかになり、そのボトルネックを把握できたと判断した場合は、これを解消する手段を考案する。そのためには、単なるパラメータのチューニングから、ハードウェアの交換、コアネルの修正など幅広い対応が考えられる。基本的には複雑な手法よりも単純な手法のほうが好ましい。

改善手段を考案したらこれを適用し、再度評価を行う。4.2.7.4 節に戻って評価手順

を改めて実施する。環境が大きく変わった時は、場合によっては 4.2.7.2 節まで遡ることもあるだろう。

再評価の結果、改善があったと認められる場合は、次に現れたボトルネックをターゲットとしてこれの改善に努める。

もしも改善しなかった場合は、改善手段またはボトルネックの把握に間違いがあったことになるため、改善手段を再検討する(4.2.7.7 節)か、測定結果を再度分析しなおす(4.2.7.6 節)か、あるいは環境を見直したうえで測定自体を再度実施する(4.2.7.3 節)。

最後にフローチャートを示す。

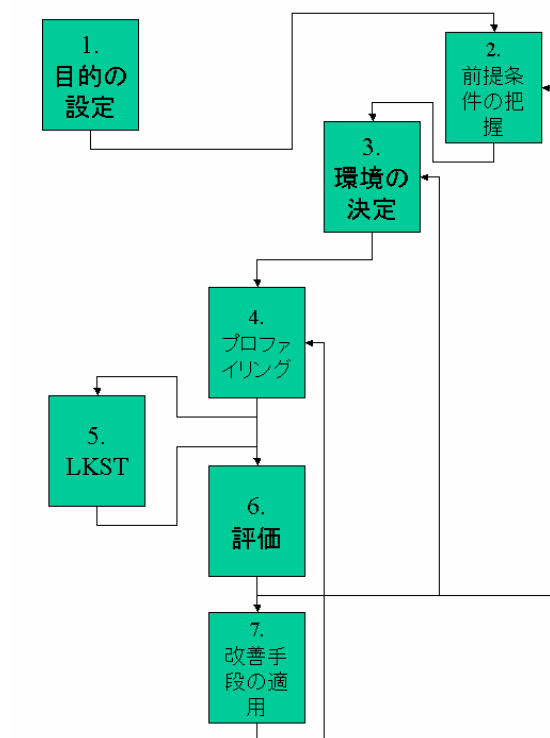


図 4.2-6 評価手順フローチャート

4.2.8 LKST の利点、制限事項

今回、OProfile では捉えきれない、個々の現象の詳細な情報を LKST によって把握することができた。現状では幾つかの制限があるものの、OProfile と組み合わせることで非常に強力な分析ツールとなることを証明した。

今回使用した、改良版 LKST によって初めて可能になった調査機能は以下の通り。

- ひとつひとつのロック取得・解除のイベントの把握
- ロックの統計データの取得
- I/O リクエストの処理時間の取得
- I/O リクエストキューの長さ（統計値ではなく瞬間値）の取得

今回使用した範囲では、以下に挙げる欠点があった。

- サンプリング(間引き)の機能がないため、データを格納する領域が不足する。CPU を複数搭載したシステムでは、CPU 数の分のデータを保持するため、相対的に測定できる時間が短くなってしまう。lkstlogd を使用すると上限がかなり広がるが、これはファイルシステムへの書き込み処理を伴うため、I/O の性能を評価する際に使うことが難しい。カーネルのバッファ領域サイズは 32bit のシステムではかなり限られたものであるため、測定時間が十分であるとは言い難い。今回の測定では、イベント内容によっては 20 秒以下しか採取することができなかったケースもあった。
- spin_lock イベントの開始時刻がエンコードされているため、ロック取得時の個々イベントのビジー時間を把握することができなかった。統計値であれば lkstla コマンドから入手できる。
- プロセス id のデータを取得できるが、今回の測定ではプロセス id とプロセス名との対応を取ることができなかったため、別に top コマンドを実施する必要があった。ただし、最新の LKST では改良されてプロセス情報を取得・抽出できるようになっている。

以上の点が改善されれば、LKST はさらに強力な分析ツールとなり、様々な現場でのシステムの解析工数を大幅に減らす手段となるだろう。今後の改良が期待される。

4.2.9 総括

これまでカーネルの評価というと、printk 文の埋め込みやパッチの適用などカーネルに直接手を入れる手法が多く、限られたごく一部のハッカーの手にしか負えない領域であったが、今回の評価によって、ユーザコマンドレベルのツールを用いて比較的容易に実施できることを示した。もちろんボトルネックを解消するためには多くの場合、カーネルを修正するなどの高度な技術が依然として必要とされるが、少なくともボトルネックの特定までは基本的な技術で到達できることが分かり、その手法を示すことができた。

大規模システムを使用した今回のパターンではCPUビジー型が最も多く観測された。これは大量のメモリコピーがボトルネックになっていることから、メモリ・レイテンシ問題の一つが浮かび上がってきた例と考えられる。メモリの高速化およびキャッシュの大容量化によって、このボトルネックの改善または解消が期待される。CPU アイドル型のようなケースではデバイスの高速化が解決手段として期待される。ロック競合型については、競合が発生する箇所を特定できたものの、これを解消する方法を導き出すには到らなかった。しかし、競合の中に、ウィンドウ環境などの常駐アプリケーションも現れていたため、これらを停止することが改善方法のひとつとして考えられるであろう。

最後に、今回実施した OProfile と LKST による評価は、それぞれのツールの機能をすべて使い切ったわけではなく、どちらも有用な機能を他にも実装している。OProfile は今回使用したバージョンが 0.5.4 であるが、2005 年 1 月現在の最新版は 0.8.1 であり、かなりの機能強化が図られている。また、LKST も今回使用した 4 個のイベントのほかに数多くのイベントがあり、より広範囲のデータを取得することができる。加えて、今回の測定において発見されて未解決となった現象も多々あり、今後追及すべき課題が以下の通りに挙げられる。

- Iozone の膨大な種類の I/O パターン(random write, mmap, 同期, 他多数)の測定
- OProfile の強化機能(統合ツール opreport, コールグラフ機能、CPU 別プロファイリング、スレッド別・プロセス別プロファイリング、HT サポート他)や他の CPU イベントによる測定
- LKST の各種トレースイベント(システムコール、スケジューラ関連、他多数)の測定
- 同一条件の測定を繰り返して得られるデータの誤差や分散等の把握
- 今回明らかになった 3 タイプのボトルネックそれぞれを改善する取り組み
 - __copy_from_user() 呼び出しに無駄がないかどうか
 - CPU キャッシュミスの調査・何らかの改善方法
 - ディスクネックと判断したパターンで、ハードウェア構成を変更しての追検証

- ロック競合発生条件の追及
- グローバルロック lock_kernel() を減らすカーネル改良
- Iozone+OProfile で時折発生したシステムフリーズの原因究明

今回は時間の制約から見送らざるを得なかったこれらの課題を、将来いずれかの機会において検証することで、さらなる解析手法のノウハウを蓄積し、それを公にして情報の共有化を進めたい。これによって信頼性を増す Linux がこれまでよりもさらに普及するための一助となれば幸いである。

以上

本書は、独立行政法人 情報処理推進機構から以下の 8 社への委託開発の成果として作成されたものです。

委託先企業：(株)日立製作所(幹事会社)

(株)SRA、(株)NTT データ、新日鉄ソリューションズ(株)

住商情報システム(株)、(株)野村総合研究所、ミラクル・リナックス(株)

ユニアデックス(株) (五十音順)