2005 年度上期
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
～CPU スケーラビリティ評価編～

作成
OSS 技術開発・評価コンソーシアム

商標表記

- Alicia は、ユニアデックス株式会社の登録商標です。
- Asianux は、ミラクル・リナックス株式会社の日本における登録商標です。
- Intel、Itanium および Intel　Xeon 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。
- Intel は、Intel Corporation の会社名です。
- 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。
- 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。
- Pentium は、Intel Corporation のアメリカ合衆国及びその他の国における登録商標です。
- 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。
- Solaris は、米国 Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。
- SUSE は、米国 Novell, Inc.の一部門である SUSE LINUX AG.の登録商標です。
- Turbolinux は、ターボリナックス株式会社の商標または登録商標です。
- UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。
- Windows は、米国およびその他の国における米国 Microsoft Corp.の登録商標です。
- 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 評価目的の概要

OS 層の評価では、2004 年度、性能限界を決めるカーネル内部コンポーネントの特定方法を確立したが、今年度はそれをうけて CPU スケーラビリティの調査と、そのカーネルボトルネックの発見を行った。

まず 2004 年度の成果を利用して、大規模メモリ搭載（メインメモリ 4GB以上）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（LKST[1]）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た。特定したカーネルボトルネックを解決するパッチを作成しその効果を検証確認した。また同時に作成したパッチはLKML (Linux Kernel Mailing List)で公開し、Linuxコミュニティからのフィードバックを受け、Andrew Morton氏の-mm treeにマージされた。メインストリーム(Linus' treeと呼ばれている)にマージされる予定である。

今回のプロジェクトの主な成果は、

1. 2004 年度に開発した性能限界を決めるカーネル内部コンポーネントの特定方法が有効であることを実証した。
2. 同方法により性質の悪い CPU キャッシュミス(cache pollution と呼ばれる)を容易に発見できることを示した。
3. 性質の悪い CPU キャッシュミス(cache pollution)が多発した場合の解決策（カーネルパッチ）を開発し、その有効性を実証した。（I/O 負荷(write)が高い場合 10%ほど実行時間が減少し、L3 キャッシュミスも 70%ほど減少する）
4. カーネルパッチの開発はその初期の段階において Linux Kernel コミュニティに公開し、様々なコメント、フィードバックを受けた。バザールモデル的な開発手法によって有用なパッチが作成できることも実証した。

今回実施したベンチマークは I/O が高負荷のものであったが異なる特性を持つベンチマークによって発見されるであろう CPU キャッシュミスの問題も、今回開発したフレームワークで容易に解決できると予想される。従来の研究では OS カーネルのキャッシュミスの動的特性およびその性能向上というものが十分なされていなかったことを考えるとこの成果には実用性のみならず新規性もあると考えられる。

[1]2004 年度日本OSS推進フォーラム開発基盤WG　OSS性能・信頼性評価 / 障害解析ツール開発、LKST

1.2 評価方法の概要

2004 年度の作業手順を利用して、Unisys ES7000(16CPU)と日立 HA8000(4CPU)のマシンで予備的な調査を行った。2004 年度の成果より、ロック競合に CPU スケーラビリティの問題が存在するという仮説のもと調査を行った。

まず、CPU クロックのサンプリングを行い、最も CPU クロックを消費しているコンポーネントを特定した。上位から当該ソースコードを分析しアルゴリズム的に問題がないか調査分析を行った。

次に LKST を利用し、ロック待ちの部位を特定した。同様にソースコードを分析し問題分析を行った。2004 年度ではカーネルロック(カーネル 2.4)の待ちが発生している部分が特定されたが、今回のカーネル 2.6 では特に性質の悪いカーネルロックは発見できなかった。

そこで CPU キャッシュの観点から調査を行ったところ、性質の悪いキャッシュミスを多発している部位を発見した。その問題を解決するパッチを作成し評価したところ、その有効性が確認できた。

1.3 分析･考察の概要

OProfile の CPU クロックのサンプリングから__copy_from_user_ll()というユーザ空間からカーネル空間へコピーする関数が最も実行時間がかかっていることが特定できた。write(2)システムコールの場合、この作業は避けようがないし、アルゴリズム的に工夫の余地も少ないと思われた。そこで視点を変え、CPU キャッシュに注目し、L3 キャッシュミス、メモリトラフィックのサンプリングを行ったところ、性質の悪い L3 キャッシュミスを発見した。この性能ボトルネックが cache pollution と呼ばれる性質のものであるということを特定し、それを解決するカーネルパッチを作成した。

その有効性を再度ベンチマークの実施をして確認した。

1.3.1 キャッシュミス削減の重要性

CPU は通常ハードウェアキャッシュと呼ばれる､メモリアクセスを高速化するハードウェアを持つ。データにアクセスするときキャッシュにそれが存在している時、キャッシュヒットと呼び、L1 キャッシュ（一次キャッシュ）の場合そのコストは 2 クロック程度である。一方でデータがキャッシュにない場合、キャッシュミスを呼ぶがメインメモリまでアクセスすると 200～300 クロック程度かかり、キャッシュアクセスに比べ 100 倍以上コストがかかる。メモリアクセス時にキャッシュミスによって CPU に他にするべき仕事がないと 200～300 クロック遊んでしまうことになる。そのため性能向上のためにはキャッシュミスを減らすことが重要になる。

表 1.3-1 アクセスコスト

デバイス	アクセスコスト

L1(一次)キャッシュ	2 クロック
L2(二次)キャッシュ	10 クロック
L3(三次)キャッシュ	50 クロック程度
メインメモリ	200～300 クロック前後

Cache line fill（CPU がキャッシュにデータを置くこと）するとき CPU はそのキャッシュラインを古いデータから新しいデータに置き換えるが、置き換えられた古いデータがすぐに必要になる場合もある。必要なデータを置き換えてしまったとき cache conflict あるいは cache pollution が発生したという。cache conflict あるいは cache pollution はキャッシュミスを引き起こし、メモリアクセスコストを著しく上げるため好ましくない。

1.3.2 Cache pollution aware patch

Pentium 4/Xeon の CPU レジスタからキャッシュ（L1/L2/L3）を経由せず直接メインメモリにデータを移動する命令を利用して cache pollution の防止を試みるパッチを作成した。このパッチを cache pollution aware patch とよぶ。その効果を測定した。その結果、実行性能で 10%程度、キャッシュミスにおいては 70%前後、当該関数においてはキャッシュミスをほぼ 0 に削減することに成功した。また cache pollution によるキャッシュミスについての評価もおこなった。

1.3.3 Cache pollution の発見方法

OProfile を利用し L3 キャッシュミスを測定すれば、cache pollution が発生している箇所を容易に特定できることをしめした。

1.3.4 メモリプロファイリング

従来のプロファイリングが主にタイマイベントのサンプリングであった。今回提案した手法は、タイマイベントによるサンプリングではなくメモリアクセスイベントに注目したサンプリングである。L3 キャッシュミスに注目しキャッシュミスが多発しているところをチューニングすると言うアプローチであり、このようなメモリアクセスに注目したアプローチは今後ますます重要になってくると考えられる。今回の調査ではメモリプロファイリングの重要性を指摘し、その効果、有効性を定量的に実証した意義は大きい。

1.3.5 CPU スケーラビリティ

2004 年度の成果を利用してロック競合の箇所が CPU スケーラビリティの問題を発生させているのではという仮説のもと調査を行ったが、2.6 カーネルでは特にスケーラビリティ上大きな問題となるような箇所は発見できなかった。cache pollution の削減が CPU スケーラビリティにどのような影響があるのかという点についての調査は行えなかった。

1.4 評価技法のまとめ

CPU ビジー型の性能向上手法として、キャッシュに注目した性能向上手法（メモリプロ

ファイリング）を提案した。

1.5 今後の課題

今後の課題として下記をあげた。

1. CPU スケーラビリティと cache pollution の防止の相関
2. Iozone 以外のベンチマーク実施
3. メモリプロファイリングによるカーネルボトルネックの計測および性能改善
4. OProfile の制限事項の解消
5. Iozone+OProfile で時折発生したカーネルパニックの原因究明

1.6 付録：Linux Kernel Mailing List(LKML)での議論

今回開発したパッチについては早い段階から LKML に公開しコミュニティの peer review を受けた。その結果、様々なコメント、助言を戴き、それをもとに改良を加えた。オープンソースソフトウェアの開発におけるコミュニティとのアライアンスの事例として付録にその経緯を記した。読者の参考としていただきたい。

2 環境定義

2.1 CPU スケーラビリティの調査

2004 年度の成果に従って今年度も OS の性能調査、特に CPU スケーラビリティの観点から調査を行った。環境定義等は 2004 年度の成果に従った。

2.1.1 ツールの説明

2.1.1.1 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 2.1-1 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 2.1-2と表 2.1-3に示す。

表 2.1-2 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 2.1-3 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:fread/re-fread, 8:random mix, 9:write/re-pwrite, 10:pread/re-pread, 11:pwritev/re-pwritev, 12:preadv/re-preadv
-I	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	通常テストの間に撹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相関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モードを実行する。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

2.1.1.2 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 稼動時間のイベント（タイマに相当する）はどの CPU でも利用できる。 PentiumIII 系では CPU_CLK_UNHALTED、Pentium4 系では GLOBAL_POWER_EVENTS と呼ばれている。今回の評価ではまずこのイベントを使用し、システムのボトルネックの箇所を特定した。LKST と組み合わせて、グローバルロックの箇所などを特定した。その後、CPU キャッシュミスに注目し、Linux カーネル内でのたちの悪いキャッシュミスを発見し、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 V4.0 において OProfile を利用するには、oprofile-0.8.2-1AX.i386.rpm パッケージに含まれている各コマンドを実行する。OProfile はデフォルトでインストールされているので、特に準備は必要ない。今回の検証に使用した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。

表 2.1-4OProfile コマンドとオプション

コマンド	説明
opcontrol [options]	各種設定をする。oprofile の起動停止
opreport [options] [profile specification]	レポートを作成する
opstack [options] [profile specification]	コールグラフを作成する
opannotate [options] [profile specification]	ソースコードとプロファイルを出力する
oparchive [options] [profile specification]	oprofile データをアーカイブする
opgprof [options] [profile specification]	gprof 形式のプロファイルデータを出力

表 2.1-5 オプション

session:sessionlist	セッションリストを指定。セッションリストの指定がない場合は、current セッションとみなす
session-exclude:sessionlist	取り除くセッションのリスト
image:imagelist	プロファイルするイメージのリスト
image-exclude:imagelist	取り除くイメージのリスト
lib-image:imagelist	イメージリストと同様
lib-image-exclude:imagelist	取り除く lib-image のリスト
event:eventname	イベント名
count:eventcount	カウント数。イベントがこの回数発生したらデータをサンプリングする
unit-mask:maskvalue	ユニットマスク
cpu:cpulist	プロファイルする CPU リスト（0 から始まる）
tgid:pidlist	プロファイルするタスクグループを指定する
tid:tidlist	プロファイルするスレッドを指定する。最近のスレッドライブラリを利用すると、プロセスのすべてのスレッドは同じタスクグループ ID を共有するが異なるスレッド ID を持つ

コマンドの解説：

/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。

表 2.1-6opcontrol 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
--no-vmlinux	カーネルファイルを持っていなく、カーネルプロファイリングを取りたくない時指定する
--help	ヘルプを表示する
--version	バージョンを表示する
--init	oprofile のモジュールをロードする
--setup	プロファイリングオプションの設定をし、/root/.oprofile/daemonrc に格納する
--start-daemon	oprofile デーモンを起動する。プロファイリングそのものはまだ開始しない。2.2/2.4 カーネルでは利用できない
--reset	サンプリングしたデータを削除する
--dump	プロファイリングしたデータをデーモンへフラッシュする
--start	プロファイリングを開始する
--stop	プロファイリングを終了する。デーモンは起動したままである。2.2/2.4 カーネルでは利用できない
--shutdown	プロファイリングを終了する。デーモンも kill する
--deinit	デーモンをシャットダウンし、oprofile モジュールと oprofilefs をアンロードする
--buffer-size	サンプルを格納するカーネルバッファサイズ
--cpu-buffer-size	CPU ごとのカーネルバッファサイズ.(2.6 のみ)
--event=[event|" default"]	イベントないしは"default"を指定する
--separate=[none,lib,kernel,thread,cpu,all]	セパレータによりサンプリングを分離する
--callgraph=#depth	コールグラフのサンプリングを有効にする。#depth が 0 の場合、コールグラフサンプリングを行わない。2.6 カーネル(x86)以降で利用可能である
--image=[name,name...|"all"]	
--verbose	デーモンのログに詳細を出力。オーバヘッドが大きい
--kernel-range=start,end	16 進でカーネルの VMA アドレスを指定する
--save=<session name>	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/lib/modules/`uname -r`/vmlinux
# opcontrol --reset
# opcontrol -start
```

/usr/bin/opreport

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。

表 2.1-7opreport オプション：

--accumulated / -c	シンボルリストのサンプル数、パーセンテージを積算する
--debug-info / -g	各シンボルのソースファイルと行を表示する
--demangle=none|smart|normal	
--details / -d	命令毎の情報を表示
--exclude-dependent / -x	アプリケーション依存のライブラリ、モジュール、カーネル等イメージを含めない。--separate を利用したときに意味がある
--exclude-symbols / -e [symbols]	シンボルを除外する
--global-percent	全体に対する比率を表示する
--help / -? / --usage	ヘルプを表示する
--image-path / -p [paths]	バイナリの追加検索パスを指定する。2.6 カーネル以降、モジュールを見つけるために必要である
--include-symbols / -i [symbols]	symbols だけを含める
--long-filenames / -l	フルパスを出力する
--merge / -m [lib, cpu, tid, tgid, unitmask, all]	--separate セッションで分離したプロファイルデータをマージする
--no-header	プロファイリングパラメータの詳細情報を含めない
--output-file / -o [file]	stdout ではなく file へ出力する
--reverse-sort / -r	デフォルトとは逆順にソートする
--show-address / -w	各シンボルの VMA アドレスを表示する
--sort / -s [vma, sample, symbol, debug, image]	[vma, sample, symbol, debug, image]でソートする
--symbols / -l	シンボルごとにリストする
--threshold / -t [percentage]	percentage 以上のデータのシンボルだけを出力する
--verbose / -V [options]	詳細なデバッグ情報を与える
--version / -v	バージョンを示す。

使用例

```
# opreport -l -p /lib/modules/$(uname -r)  > ${saveDir}/summary.out 2>&1
```

/usr/bin/opannotate

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。

表 2.1-8opannotate オプション：

--assembly / -a	アセンブリを出力する。--source と一緒の場合、ソース、アセンブリ混在で出力する
--output-dir <directory>	ソースファイルの出力ディレクトリを指定する
--demangle=none|smart| normal	
--exclude-dependent / -x	アプリケーション依存のライブラリ、モジュール、カーネル等イメージを含めない。--separate を利用したときに意味がある
--exclude-symbols / -e [symbols]	シンボルを除外する
--help / -? / --usage	ヘルプを表示する
--image-path / -p [paths]	バイナリの追加検索パスを指定する。2.6 カーネル以降、モジュールを見つけるために必要である
--include-file [files]	files だけを含める
--include-symbols / -i [symbols]	symbols だけを含める
--objdump-params [params]	objdump を呼ぶときの追加のパラメータ
--merge / -m [lib, cpu, tid, tgid, unitmask, all]	--separate セッションで分離したプロファイルデータをマージする
--output-dir / -o [dir]	出力ディレクトリ。opannotate 出力をソースファイルごとに行う
--search-dirs / -d [paths]	ソースファイルを検索するパス。イメージに対するデバッグ情報が早退パスを含むとき、このオプションを利用する必要がある
--base-dirs / -b [paths]	デバッグソースファイルから取り除くパス。--search-dirs の base-dirs を探す前に行う
--source / -s	annotate ソースを出力する。これにはバイナリのデバッグ情報が必要である
--threshold / -t [percentage]	percentage 以上のデータのシンボルだけを出力する
--verbose / -V [options]	詳細なデバッグ情報を与える
--version / -v	バージョンを示す

使用例：

```
# opannotate -s -o ${saveDir}/src 2> ${saveDir}/opannotate.log
```

そのほか、opstack/opgprof/oparchive などのコマンドがある。

利用可能な性能イベントは、x86info コマンドで下記のよう得られる。

```
Family: 15 Model: 2 Stepping: 6 Type: 0 Brand: 12
CPU Model: Pentium 4 (Northwood) Original OEM
Processor name string: Intel(R) Xeon(TM) MP CPU 2.20GHz

Feature flags:
 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflsh dtes acpi mmx fxsr sse sse2 selfsnoop ht acc pbe
Extended feature flags:
 cntx-id
Instruction trace cache:
 Size: 12K uOps 8-way associative.
L1 Data cache:
 Size: 8KB      Sectored, 4-way associative.
 line size=64 bytes.
L2 unified cache:
 Size: 512KB    Sectored, 8-way associative.
 line size=64 bytes.
L3 unified cache:
 Size: 2MB      8-way associative.
 line size=64 bytes.
Instruction TLB: 4K, 2MB or 4MB pages, fully associative, 128 entries.
Data TLB: 4KB or 4MB pages, fully associative, 64 entries.
The physical package supports 1 logical processors
```

図 2.1-1 利用可能な性能イベント

このマシンの利用可能な性能評価イベントは、opcontrol –list-events で得られる。性能評価イベントは CPU アーキテクチャに依存するので注意が必要である。例えば Pentium III と Pentium 4/Intel Xeon 系では取得できるイベントはまったく異なる。

各イベントの詳細については、IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, Appendix A を参照のこと。

表 2.1-9Pentium4/Xeon の性能評価イベント

oprofile: available events for CPU type "P4 / Xeon" See Intel Architecture Developer's Manual Volume 3, Appendix A and Intel Architecture Optimization Reference Manual (730795-001)
イベント名およびユニットマスク
GLOBAL_POWER_EVENTS: (counter: 0, 4) time during which processor is not stopped (min count: 3000) Unit masks (default 0x1) ---------- 0x01: mandatory
BRANCH_RETIRED: (counter: 3, 7) retired branches (min count: 3000) Unit masks (default 0xc) ---------- 0x01: branch not-taken predicted 0x02: branch not-taken mispredicted 0x04: branch taken predicted 0x08: branch taken mispredicted
MISPRED_BRANCH_RETIRED: (counter: 3, 7) retired mispredicted branches (min count: 3000) Unit masks (default 0x1) ---------- 0x01: retired instruction is non-bogus
BPU_FETCH_REQUEST: (counter: 0, 4) instruction fetch requests from the branch predict unit (min count: 3000) Unit masks (default 0x1) ---------- 0x01: trace cache lookup miss
ITLB_REFERENCE: (counter: 0, 4) translations using the instruction translation lookaside buffer (min count: 3000) Unit masks (default 0x7) ---------- 0x01: ITLB hit 0x02: ITLB miss 0x04: uncacheable ITLB hit
MEMORY_CANCEL: (counter: 2, 6) cancelled requesets in data cache address control unit (min count: 3000)

Unit masks (default 0x8) ---------- 0x04: replayed because no store request buffer available 0x08: conflicts due to 64k aliasing
MEMORY_COMPLETE: (counter: 2, 6) completed split (min count: 3000) Unit masks (default 0x3) ---------- 0x01: load split completed, excluding UC/WC loads 0x02: any split stores completed 0x04: uncacheable load split completed 0x08: uncacheable store split complete
LOAD_PORT_REPLAY: (counter: 2, 6) replayed events at the load port (min count: 3000) Unit masks (default 0x2) ---------- 0x02: split load
STORE_PORT_REPLAY: (counter: 2, 6) replayed events at the store port (min count: 3000) Unit masks (default 0x2) ---------- 0x02: split store
MOB_LOAD_REPLAY: (counter: 0, 4) replayed loads from the memory order buffer (min count: 3000) Unit masks (default 0x3a) ---------- 0x02: replay cause: unknown store address 0x08: replay cause: unknown store data 0x10: replay cause: partial overlap between load and store 0x20: replay cause: mismatched low 4 bits between load and store addr
BSQ_CACHE_REFERENCE: (counter: 0, 4) cache references seen by the bus unit (min count: 3000) Unit masks (default 0x73f) ---------- 0x01: read 2nd level cache hit shared 0x02: read 2nd level cache hit exclusive 0x04: read 2nd level cache hit modified 0x08: read 3rd level cache hit shared 0x10: read 3rd level cache hit exclusive

```
        0x20: read 3rd level cache hit modified
        0x100: read 2nd level cache miss
        0x200: read 3rd level cache miss
        0x400: writeback lookup from DAC misses 2nd level cache
IOQ_ALLOCATION: (counter: 0)
        bus transactions (min count: 3000)
        Unit masks (default 0xefe1)
        ----------
        0x01: bus request type bit 0
        0x02: bus request type bit 1
        0x04: bus request type bit 2
        0x08: bus request type bit 3
        0x10: bus request type bit 4
        0x20: count read entries
        0x40: count write entries
        0x80: count UC memory access entries
        0x100: count WC memory access entries
        0x200: count write-through memory access entries
        0x400: count write-protected memory access entries
        0x800: count WB memory access entries
        0x2000: count own store requests
        0x4000: count other / DMA store requests
        0x8000: count HW/SW prefetch requests
IOQ_ACTIVE_ENTRIES: (counter: 4)
        number of entries in the IOQ which are active (min count: 3000)
        Unit masks (default 0xefe1)
        ----------
        0x01: bus request type bit 0
        0x02: bus request type bit 1
        0x04: bus request type bit 2
        0x08: bus request type bit 3
        0x10: bus request type bit 4
        0x20: count read entries
        0x40: count write entries
        0x80: count UC memory access entries
        0x100: count WC memory access entries
        0x200: count write-through memory access entries
        0x400: count write-protected memory access entries
        0x800: count WB memory access entries
```

```
        0x2000: count own store requests
        0x4000: count other / DMA store requests
        0x8000: count HW/SW prefetch requests
BSQ_ALLOCATION: (counter: 0)
        allocations in the bus sequence unit (min count: 3000)
        Unit masks (default 0x21)
        ----------
        0x01: (r)eq (t)ype (e)ncoding, bit 0: see next bit
        0x02: rte bit 1: 00=read, 01=read invalidate, 10=write, 11=writeback
        0x04: req len bit 0
        0x08: req len bit 1
        0x20: request type is input (0=output)
        0x40: request type is bus lock
        0x80: request type is cacheable
        0x100: request type is 8-byte chunk split across 8-byte boundary
        0x200: request type is demand (0=prefetch)
        0x400: request type is ordered
        0x800: (m)emory (t)ype (e)ncoding, bit 0: see next bits
        0x1000: mte bit 1: see next bits
        0x2000: mte bit 2: 000=UC, 001=USWC, 100=WT, 101=WP, 110=WB
X87_ASSIST: (counter: 3, 7)
        retired x87 instructions which required special handling (min count: 3000)
        Unit masks (default 0x1f)
        ----------
        0x01: handle FP stack underflow
        0x02: handle FP stack overflow
        0x04: handle x87 output overflow
        0x08: handle x87 output underflow
        0x10: handle x87 input assist
MACHINE_CLEAR: (counter: 3, 7)
        cycles with entire machine pipeline cleared (min count: 3000)
        Unit masks (default 0x1)
        ----------
        0x01: count a portion of cycles the machine is cleared for any cause
        0x04: count each time the machine is cleared due to memory ordering issues
        0x40: count each time the machine is cleared due to self modifying code
TC_MS_XFER: (counter: 1, 5)
        number of times uops deliver changed from TC to MS ROM (min count: 3000)
        Unit masks (default 0x1)
```

0x01: count TC to MS transfers
UOP_QUEUE_WRITES: (counter: 1, 5)
number of valid uops written to the uop queue (min count: 3000)
Unit masks (default 0x7)

0x01: count uops written to queue from TC build mode
0x02: count uops written to queue from TC deliver mode
0x04: count uops written to queue from microcode ROM
INSTR_RETIRED: (counter: 3, 7)
retired instructions (min count: 3000)
Unit masks (default 0x1)

0x01: count non-bogus instructions which are not tagged
0x02: count non-bogus instructions which are tagged
0x04: count bogus instructions which are not tagged
0x08: count bogus instructions which are tagged
UOPS_RETIRED: (counter: 3, 7)
retired uops (min count: 3000)
Unit masks (default 0x1)

0x01: count marked uops which are non-bogus
0x02: count marked uops which are bogus
UOP_TYPE: (counter: 3, 7)
type of uop tagged by front-end tagging (min count: 3000)
Unit masks (default 0x2)

0x02: count uops which are load operations
0x04: count uops which are store operations
RETIRED_MISPRED_BRANCH_TYPE: (counter: 1, 5)
retired mispredicted branched, selected by type (min count: 3000)
Unit masks (default 0x1f)

0x01: count unconditional jumps
0x02: count conditional jumps
0x04: count call branches
0x08: count return branches
0x10: count indirect jumps
RETIRED_BRANCH_TYPE: (counter: 1, 5)

retired branches, selected by type (min count: 3000)
Unit masks (default 0x1f)

0x01: count unconditional jumps
0x02: count conditional jumps
0x04: count call branches
0x08: count return branches
0x10: count indirect jumps

TC_DELIVER_MODE: (counter: 1, 5)
duration (in clock cycles) in the trace cache and decode engine (min count: 3000)
Unit masks (default 0x4)

0x04: processor is in deliver mode
0x20: processor is in build mode

PAGE_WALK_TYPE: (counter: 0, 4)
page walks by the page miss handler (min count: 3000)
Unit masks (default 0x1)

0x01: page walk for data TLB miss
0x02: page walk for instruction TLB miss

FSB_DATA_ACTIVITY: (counter: 0, 4)
DRDY or DBSY events on the front side bus (min count: 3000)
Unit masks (default 0x3f)

0x01: count when this processor drives data onto bus
0x02: count when this processor reads data from bus
0x04: count when data is on bus but not sampled by this processor
0x08: count when this processor reserves bus for driving
0x10: count when other reserves bus and this processor will sample
0x20: count when other reserves bus and this processor will not sample

BSQ_ACTIVE_ENTRIES: (counter: 4)
number of entries in the bus sequence unit which are active (min count: 3000)
Unit masks (default 0x21)

0x01: (r)eq (t)ype (e)ncoding, bit 0: see next bit
0x02: rte bit 1: 00=read, 01=read invalidate, 10=write, 11=writeback
0x04: req len bit 0
0x08: req len bit 1

0x20: request type is input (0=output)
0x40: request type is bus lock
0x80: request type is cacheable
0x100: request type is 8-byte chunk split across 8-byte boundary
0x200: request type is demand (0=prefetch)
0x400: request type is ordered
0x800: (m)emory (t)ype (e)ncoding, bit 0: see next bits
0x1000: mte bit 1: see next bits
0x2000: mte bit 2: 000=UC, 001=USWC, 100=WT, 101=WP, 110=WB

SSE_INPUT_ASSIST: (counter: 2, 6)
input assists requested for SSE or SSE2 operands (min count: 3000)
Unit masks (default 0x8000)

0x8000: count all uops of this type

PACKED_SP_UOP: (counter: 2, 6)
packed single precision uops (min count: 3000)
Unit masks (default 0x8000)

0x8000: count all uops of this type

PACKED_DP_UOP: (counter: 2, 6)
packed double precision uops (min count: 3000)
Unit masks (default 0x8000)

0x8000: count all uops of this type

SCALAR_SP_UOP: (counter: 2, 6)
scalar single precision uops (min count: 3000)
Unit masks (default 0x8000)

0x8000: count all uops of this type

SCALAR_DP_UOP: (counter: 2, 6)
scalar double presision uops (min count: 3000)
Unit masks (default 0x8000)

0x8000: count all uops of this type

64BIT_MMX_UOP: (counter: 2, 6)
64 bit integer SIMD MMX uops (min count: 3000)
Unit masks (default 0x8000)

0x8000: count all uops of this type

128BIT_MMX_UOP: (counter: 2, 6) 128 bit integer SIMD SSE2 uops (min count: 3000) Unit masks (default 0x8000) ---------- 0x8000: count all uops of this type
X87_FP_UOP: (counter: 2, 6) x87 floating point uops (min count: 3000) Unit masks (default 0x8000) ---------- 0x8000: count all uops of this type
X87_SIMD_MOVES_UOP: (counter: 2, 6) x87 FPU, MMX, SSE, or SSE2 loads, stores and reg-to-reg moves (min count: 3000) Unit masks (default 0x18) ---------- 0x08: count all x87 SIMD store/move uops 0x10: count all x87 SIMD load uops

2.1.1.3 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。

2.1.2 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、以下のとおりである。

1. HA8000/270
2. ES7000/520

表 2.1-10 ハードウェア構成の通り。

表 2.1-10 ハードウェア構成

モデル	日立 HA8000/270
プロセッサ	Intel Xeon MP2.0GHz x 4 [family:15, model:2, stepping:7, cache size:512KB] L1 キャッシュ 8KB、L2 キャッシュ 512KB、L3 キャッシュ 2MB
メインメモリ	5GB

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）
ハードディスク	34GB x 4

表 2.1-11 ハードウェア構成

モデル	Unisys ES7000/520
プロセッサ	Intel　Xeon　MP2.0GHz x 16 [family:15, model:2, stepping:7, cache size:512KB] L1 キャッシュ 8KB、L2 キャッシュ 512KB、L3 キャッシュ 2MB
メインメモリ	16GB
ディスク コントローラ	Emulex LP9802 x 4 (SANARENA1570)
ハードディスク	72GB x 14 (13 LU/RAID0)

以降で紹介する環境定義は、この CPU4 基、メモリ 5GB、ディスク 4 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。

2.1.3 OS のインストール

OS には MIRACLE LINUX V4.0 – Asianux Inside(ベータ版)を使用した。また、カーネルについては 2.6.9 ないしアップストリームカーネルの最新版(2.6.12.4, 2.6.13)を使用した。

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』[2]に従う。インストール中のオプション選択は表 2.1-12の通りで、これ以外はすべてデフォルト状態とした。

表 2.1-12 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 2.1-13参照

[2] http://www.miraclelinux.com/products/linux/ml30/pdf/installation_guide.pdf

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	テキスト

パーティション構成は表 2.1-13の通り。今回は 2004 年度の構成に従って最大 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。

表 2.1-13 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/sda7	8GB 以上	/test/f4

2.1.4 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。サービスの設定をする。

2.1.4.1 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
```

2.1.4.2 通常カーネルのインストール

Asianux 2.0 入手し、インストールする。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-2.6.9-5.25AX.i686.rpm を以下の URL よりダウンロードする。ただしカ

ーネルはベータ版であるので適宜最新版に置き換え実行する。今回の評価は Asianux 2.0 ベータ版にて行ったが、報告書執筆時点では Asianux 2.0 は既に出荷されているので、それを利用するのが望ましい。
http://ftp.miraclelinux.com/pub/OSSF/2005/OS/
http://www.asianux.com/download.php

パッケージを rpm -ivh コマンドでインストールする。

```
# rpm -ivh kernel-debuginfo-2.6.9-5.25AX.i686.rpm
```

2.1.4.3 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 イベントハンドラ lksteh_ax2b3.tgz を以下の URL からダウンロードする。
http://ftp.miraclelinux.com/pub/OSSF/2005/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。

2.1.5 ツールの設定

Iozone の Web サイト (http://www.iozone.org/) より Stable tarball ソース iozone3_239.tar をダウンロードし、これをビルドする。
今回の評価では /usr/src 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_239.tar
# cd src/current
# make linux
# cd ../..
```

Iozone の実行プログラムが /usr/src/iozone3_239/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

2004 年度の成果に基づいて、評価を行った。

3.1 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。

3.1.1 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。

3.1.2 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。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 5 GB 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を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。

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。

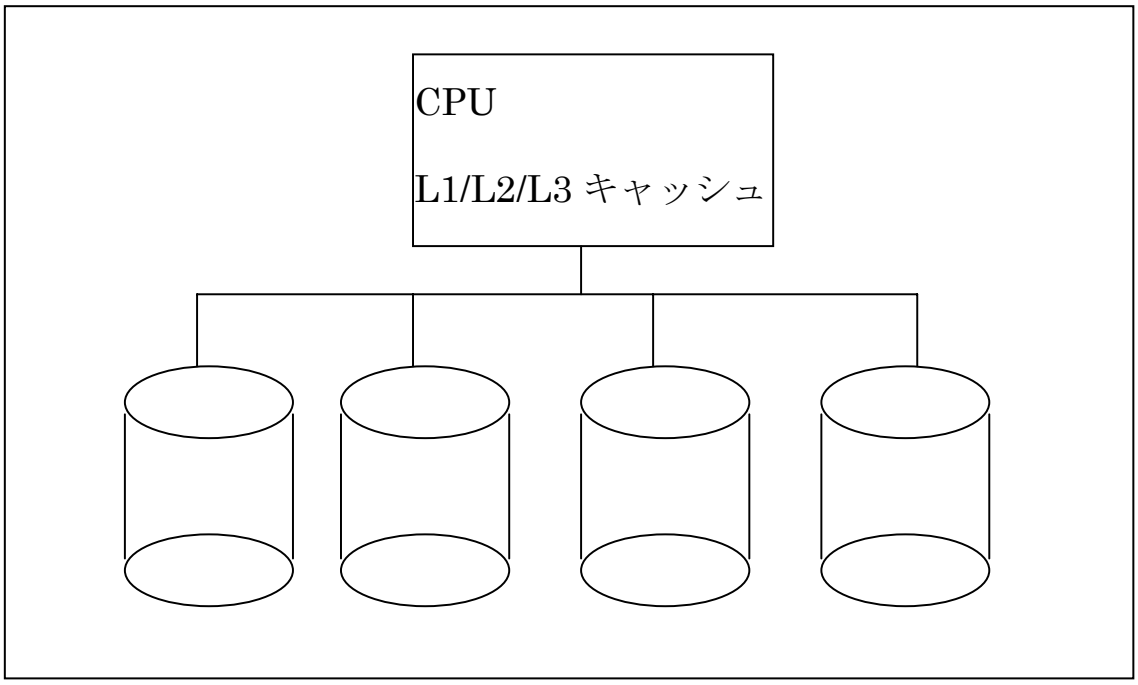


図 3.1-1 ベンチマーク構成

以上の条件から、Iozoneの実施パターンは表-3.1-1に示した 11 通りになる。

表-3.1-1 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。

ディレクトリ名は＜パターン名>＋＜LKST のフラグ＞＋＜CPU 数＞＋＜oprofile のフラグ＞＋＜日付＞というフォーマットになっている。

LKST のフラグの意味

1. b: buffer/blkqueue
2. s:busywait/spinlock
3. v:vmscan
4. 0:LKST なしで測定

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
 /usr/src/iozone3_239/src/current/iozone -CMR -i 0 -+n -+u -s 2666M -t 1 -f¥
/test/f${num} > /os-bench/pattern8-0-cpu4-0-09081425/iozone${num}.out &
done
wait
```

パターン 10 の実行例：

```
# /usr/src/iozone3_239/src/current/iozone -CMR -i 0 -+n -+u -s 2000M -t 4 -T -F ¥
/test/f1/io /test/f2/io /test/f3/io /test/f4/io ¥
> /os-bench/pattern10-0-cpu4-0-09081436/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のようになる。

パターン 9 の結果：

```
                Iozone: Performance Test of File I/O
                        Version $Revision: 3.239 $        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: Mon Sep   5 11:52:16 2005  実行開始時刻

            Machine = Linux hitachi01 2.6.13 #1 SMP Sat Sep 3 16:16:09 JST 2005 i686 i686    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 = 0.004 seconds.
            CPU utilization Excel chart enabled
            File size set to 2048000 KB  ファイルサイズ
            Command line used: /usr/src/iozone3_239/src/current/iozone -CMR -i 0 -+n -+u -s 2000M -t 4 -F
/test/f1/io /test/f2/io /test/f3/io /test/f4/io  実行したコマンド
            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 4 processes
            Each process writes a 2048000 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   4 initial writers   =   198133.89 KB/sec  合計速度
            Parent sees throughput for   4 initial writers    =   140604.09 KB/sec
            Min throughput per process                        =    41429.47 KB/sec  最小速度
            Max throughput per process                        =    53011.06 KB/sec  最大速度
            Avg throughput per process                        =    49533.47 KB/sec  平均速度
            Min xfer                                          = 1572952.00 KB  最小転送量
            CPU Utilization: Wall time    46.518      CPU time    59.212      CPU utilization 127.29 %
                                                                  CPU 利用率
            Child[0] xfer count = 1959908.00 KB, Throughput =    51849.09 KB/sec, wall=41.150, cpu=14.961, %= 36.36
子プロセス0の結果
            Child[1] xfer count = 2016036.00 KB, Throughput =    53011.06 KB/sec, wall=39.966, cpu=14.933, %= 37.36
子プロセス1の結果
            Child[2] xfer count = 2048000.00 KB, Throughput =    51844.27 KB/sec, wall=39.503, cpu=14.921, %= 37.77
子プロセス2の結果
            Child[3] xfer count = 1572952.00 KB, Throughput =    41429.47 KB/sec, wall=46.518, cpu=14.397, %= 30.95
子プロセス3の結果

以下は Excel 用 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"  スループットデータ
"    Initial write "    198133.89
"          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 "       127.29  CPU 利用率データ
"          Rewrite "          0.00
iozone test complete.  実行終了
```

図 3.1-2 パターン９の実行例

3.1.3 OProfile 実行手順

OProfile は、MIRACLE LINUX V 4 .0 (Asianux 2.0)に oprofile-0.8.2-1AX.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[3]は、このシステムのデフォルトの 100,000 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=${kernel}
# opcontrol --reset
```

ここで**${kernel}**は/usr/lib/debug/lib/modules/$(uname -r)/vmlinux である。

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）:

[3]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

```
# opreport -l -p /lib/modules/$(uname -r) > ${saveDir}summary.out 2>&1
# opreport -l -p /lib/modules/$(uname -r) > ${saveDir}detail.out 2>&1
# opannotate -s -d /usr/src/linux-$(uname -r) ¥
-o ${saveDir}/src > ${saveDir}/opannotate.log 2>&1
# opcontrol -save=${session}
```

データ処理用コマンドは oprofile 0.5.4（MIRACLE LINUX V3.0 に同梱）では **oprofpp** であったが、oprofile 0.8.2 (MIRACLE LINUX V4.0/Asianux 2.0)では、**opreport** と変更になっているので注意が必要である。また **op_to_source** は **opannotate** となった。

コマンド出力を保存するディレクトリは先に記したとおり＜パターン名>＋＜LKST のフラグ＞＋＜CPU 数＞＋＜oprofile のフラグ＞＋＜日付＞となっているので毎回分けて保存する。

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-3.1-2 OProfile のデータ

ファイル summary.out	カーネルの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/session/	OProfile サンプリング生データ保存先。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。
pattern9-0-cpu2-0-09081635/summary.out

CPU: P4 / Xeon with 2 hyper-threads, speed 2993.03 MHz (estimated) CPUの種類とクロック

Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped) with a unit mask of 0x01 (mandatory) count 100000 測定したイベントとサンプリング頻度

samples	%	image name	app name	symbol name
サンプル数	占有率	イメージ名	アプリケーション名	シンボル名
215795	5.6329	vmlinux	vmlinux	
__copy_user_zeroing_intel_nocache				
104172	2.7192	jbd.ko	jbd	journal_stop
102359	2.6719	jbd.ko	jbd	journal_add_journal_head
93709	2.4461	jbd.ko	jbd	do_get_write_access
89073	2.3251	vmlinux	vmlinux	__might_sleep
74740	1.9509	jbd.ko	jbd	journal_put_journal_head
73707	1.9240	jbd.ko	jbd	journal_dirty_metadata
68674	1.7926	vmlinux	vmlinux	__find_get_block
63702	1.6628	jbd.ko	jbd	journal_cancel_revoke
63660	1.6617	vmlinux	vmlinux	__brelse
62624	1.6347	jbd.ko	jbd	journal_dirty_data
57284	1.4953	vmlinux	vmlinux	mark_offset_tsc
55985	1.4614	jbd.ko	jbd	start_this_handle
54099	1.4121	vmlinux	vmlinux	__cond_resched
47993	1.2528	vmlinux	vmlinux	__mark_inode_dirty
47878	1.2498	vmlinux	vmlinux	unlock_buffer
42326	1.1048	ext3.ko	ext3	ext3_new_block
39625	1.0343	vmlinux	vmlinux	__block_write_full_page

図 3.1-3summary.out 出力

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: P4 / Xeon with 2 hyper-threads, speed 2993.03 MHz (estimated)  CPU の種類とクロック
Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped) with a unit mask of
0x01 (mandatory) count 100000
vma       samples  %        image name               app name                 symbol name
c01c3be1 215795    5.6329   vmlinux                  vmlinux
__copy_user_zeroing_intel_nocache
仮想アドレス      全体に対する占有率                  アプリケーション名          シンボル名
        サンプル数    当該関数内での占有率
  c01c3be1 86         0.0399
  c01c3be2 74         0.0343
  c01c3be4 35         0.0162
  c01c3be5 40         0.0185
  c01c3be7 10         0.0046
  c01c3be8 30         0.0139
  c01c3bea 117        0.0542
  c01c3bed 3404       1.5774
  c01c3bf0 157        0.0728
  c01c3bf2 2231       1.0339
  c01c3bf5 9581       4.4399
  c01c3bf6 4308       1.9963
  c01c3bf9 2554       1.1835
  c01c3bfc 295        0.1367
  c01c3c00 5485       2.5418
  c01c3c04 4328       2.0056
  c01c3c07 2171       1.0060
  c01c3c0a 325        0.1506
  c01c3c0e 4915       2.2776
```

図 3.1-4detail.out の出力例

3.1.4 LKST 実行手順

Asianux 2.0 のカーネルには標準で LKST が搭載されているので特にカーネルのインストール等は必要ない。これは、2004 年度 OSS 活用基盤整備事業プロジェクトの LKST 開発グループの成果である。

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 I/O 処理時間（buffer）とブロック I/O 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。LKST のバッファサイズは CPU 数に依存して設定した。バッファサイズは確保可能な最大値（60MB）を CPU 数で割ることによって指定しており、今回の測定ではおよそ数秒～20 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。CPU 数が多くなると 1 CPU あたりのバッファサイズが小さくなるので測定可能な時間は短くなる。

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 3.1-3 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

3.1.5 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-3.1-4 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。LKSTbufersize 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 5GB 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.
略

#
```

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-3.1-2と表 3.1-3 LKSTデータに挙げた各データファイルが /os-bench/pattern<ID>-0-<CPU数>-0-<date>/ 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。

3.1.6 Iozone+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 LKST による測定、分析を行う。

LKST は、Asianux 2.0 にデフォルトで実装されているので、特にインストール等の作業は必要ない。

LKST を有効にすることで若干のオーバ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。。

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 &
```

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-3.1-5 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
umount /dev/sdb1 and mkfs /test/f1
mke2fs 1.35 (28-Feb-2004)
umount /dev/sdc1 and mkfs /test/f2
mke2fs 1.35 (28-Feb-2004)
中略
opcontrol --vmlinux=/usr/lib/debug/lib/modules/2.6.9-11.11AXsmp/vmlinux
Starting Kernel State Tracer:                                [  OK  ]
lkstm write -f /os-bench/lkstmask.busywait.spinlock -n busywait.spinlock
New maskset id=3 was written. (Name:busywait.spinlock)
lkstbuf create -s 15M -b 1
New buffer was created, cpu=0, id=1 size=15728640 + 4032(margin)
New buffer was created, cpu=1, id=1 size=15728640 + 4032(margin)
New buffer was created, cpu=2, id=1 size=15728640 + 4032(margin)
New buffer was created, cpu=3, id=1 size=15728640 + 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 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
Start LKST event tracing.
Currently selected maskset was changed to id=3
Linux hitachi01 2.6.9-11.11AXsmp #1 SMP Thu Jul 14 05:15:42 EDT 2005 i686 i686
i386 GNU/Linuxmke2fs 1.32 (09-Nov-2002)
========== Start of id 9 s cpu 4 at Sun Sep 18 14:42:58 JST 2005 ============
   65.05s real     0.94s user    93.03s system
略
New buffer was created, cpu=3, id=0 size=102400 + 4032(margin)
Currently selected buffer changed to 0 on CPU 0
Currently selected buffer changed to 0 on CPU 1
Currently selected buffer changed to 0 on CPU 2
Currently selected buffer changed to 0 on CPU 3
Buffer id=1 was deleted.
Buffer id=1 was deleted.
Buffer id=1 was deleted.
Buffer id=1 was deleted.
/os-bench/pattern9-s-cpu4-0-09181442
============ End of id 9 s at Sun Sep 18 14:44:25 JST 2005 ==============#
```

図 3.1-5run.sh パターン９、-b 実行例

正常終了すると、データファイルが /os-bench/pattern<ID>-<**指定オプション**>/ ディレクトリ下に作成される。(上の例だと/os-bench/pattern9-s-cpu4-0-09181442)

3.2 テストスクリプトについて

ベンチマークを自動化するために run.sh を作成した。

```
#!/bin/ksh
#
# Copyright (C) 2005 MIRACLE LINUX CORPORATION. All rights reserved.
#
################################################################################
#
# Exection program of Iozone + OProfile + LKST
#
# Usage:
#        run.sh [-b|-s|-v|-o] <Pattern ID>
#
#                -b   : LKST buffer and blkqueue
#                -s   : LKST busywait and spinlock
#                -v   : LKST vmscan
#                -o   : without oprofile
#            No option: No LKST
#
# Pattern IDs of iozone:
#      1)  -s ${SIZE1} -t 1
#      2)  -s ${SIZE1} -t 1 -T
#      3)  -s ${SIZE2} -t 2
#      4)  -s ${SIZE2} -t 2 -T
#      5) (-s ${SIZE2} -t 1) x 2
#      6)  -s ${SIZE3} -t 3
#      7)  -s ${SIZE3} -t 3 -T
#      8) (-s ${SIZE3} -t 1) x 3
#      9)  -s ${SIZE4} -t 4
#     10)  -s ${SIZE4} -t 4 -T
#     11) (-s ${SIZE4t 1) x 4
#
# Prerequisite:
#     + Iozone (compiled)
#        Specified by IOZONE variable.
#     + RPM packages:
#        - for LKST
#           lkstutils  (Enhanced version)
#           kernel-smp (Enhanced LKST within)
#        - for OProfile
#           kernel-smp
#           kernel-debuginfo
#     + Filesystems:
#         Each one should be capable of the size specified by SIZE1 variable.
#           /test/f1
#           /test/f2
#           /test/f3
#           /test/f4
```

```
#
# Output:
#   Create pattern<id>-<option> directory
#    and output following data files and directories in it:
#     + lozone
#         iozone.out
#     + OProfile
#         summary.out
#         detail.out
#         src/
#         samples/
#     + LKST
#         lkst.data
#         lkst.data.print
#         lkst.data.<analyzer>.{d,l,s}
#
################################################################################
#
# Settings

CPUS=`grep processor /proc/cpuinfo|wc -l`
baseDir=/os-bench
IOZONE=/usr/src/iozone3_239/src/current/iozone
case $CPUS in
  1) LKSTbuffersize=60M;;
  2) LKSTbuffersize=30M;;
  4) LKSTbuffersize=15M;;
  8) LKSTbuffersize=8M;;
 16) LKSTbuffersize=4M;;
esac
#SIZE1=9500M
SIZE1=8000M
#SIZE2=4750M
SIZE2=4000M
#SIZE3=3166M
SIZE3=2666M
SIZE4=2000M
SIZE13=615M

################################################################################
#
# Defines

dummyFiles2='/test/f1/io /test/f2/io'
dummyFiles3='/test/f1/io /test/f2/io /test/f3/io'
dummyFiles4='/test/f1/io /test/f2/io /test/f3/io /test/f4/io'
dummyFiles13='/test/f1/io /test/f2/io /test/f3/io /test/f4/io /test/f5/io  ¥
/test/f6/io test/f7/io   /test/f8/io /test/f9/io /test/f10/io /test/f11/io ¥
/test/f11/io /test/f12/io /test/f13/io'
defaultOptions='-CMR -i 0 -+n -+u'
kernel=/usr/lib/debug/lib/modules/$(uname -r)/vmlinux
oprofileDir=/var/lib/oprofile
sampleDir=${oprofileDir}/samples
withLKST=0
noprofile=0
OPevent=
```

```
LANG=C
time=$(date '+%m%d%H%M')

print_usage () {
  print usage: $(basename $0) '[-b|-s] <pattern id>'
  exit 1
}

checkEnv () {
  if [[ ! -x ${IOZONE} ]]; then
    print Cannot find Iozone at ${IOZONE}
    exit 1
  fi
  if [[ ! -x /usr/bin/opcontrol ]]; then
    print Cannot find OProfile.
    exit 1
  fi
  if [[ ${withLKST} != 0 ]]; then
    if [[ ! -x /usr/sbin/lkst ]]; then
      print Cannot find LKST.
      exit 1
    fi
  fi
}

datadir () {
# $1: <serial number for data stock>
  outDir=${baseDir}/pattern${1}-${2}-cpu$CPUS-${3}-${time}
  rm -rf ${outDir}
  mkdir -p ${outDir}
}

searchDevice () {
  drives=''
#  for i in 1 2 3 4 5 6 7 8 9 10 11 12 13; do
  for i in 1 2 3 4 ; do
    drives="${drives} $(df | grep /test/f${i}¥$ | cut -d' ' -f1)"
  done
}
cleanfs () {
# $1: <number of filesystems to use>
  let max=$1
  let i=1
  for device in ${drives}; do
    (( i > max )) && break
    mountPoint=/test/f${i}
    print umount ${device} and mkfs ${mountPoint}
    umount ${device} && mkfs -t ext3 -L ${mountPoint} ${device} > /dev/null && mount ${device}
${mountPoint}
#    umount ${device} && mount ${device} ${mountPoint}
    let i=i+1
  done
}

logging_tasks () {
  top -b -d 10 > ${outDir}/top.log &
```

```
}

prepare_lkst () {
  if [[ ! -f ${baseDir}/lkstmask.${lkstMaskset} ]]; then
    print Cannot find Maskset file ${baseDir}/lkstmask.${lkstMaskset}
    exit 1
  fi
  service lkst start
  print lkstm write -f ${baseDir}/lkstmask.${lkstMaskset} -n ${lkstMaskset}
  lkstm write -f ${baseDir}/lkstmask.${lkstMaskset} -n ${lkstMaskset}
  print lkstbuf create -s ${LKSTbuffersize} -b 1
  lkstbuf create -s ${LKSTbuffersize} -b 1
  lkstbuf jump -b 1
  lkstbuf delete -b 0
}

start_lkst () {
  lkst start
  lkstm set -n ${lkstMaskset}
  modprobe lksteh_sysinfo
}

stop_lkst () {
  lkst stop
}

post_lkst () {
  lkstLog=${outDir}/lkst.data
  lkstbuf read -f ${lkstLog}
  service lkst stop
  lkstlogdiv ${lkstLog}
  for analyzer in ${lkstAnalyzers}; do
    for format in l d s; do
      lkstla ${analyzer} -${format} ${lkstLog}-? > ${lkstLog}.${analyzer}.${format}
    done
  done
  rm -f ${lkstLog}-?
  lkstbuf print -f ${lkstLog} -r -C > ${lkstLog}.print
  lkstbuf create -s 100K -b 0
  lkstbuf jump -b 0
  lkstbuf del -b 1
}

prepare_op () {
#  opcontrol --ctr0-event=GLOBAL_POWER_EVENTS --ctr0-count=996500
  if [[ -f ${kernel} ]]; then
    opcontrol --vmlinux=${kernel} ${OPevent}
    print opcontrol --vmlinux=${kernel} ${OPevent}
  else
#    opcontrol --no-vmlinux
    print Cannot find ${kernel}
    exit 1
  fi
  opcontrol --reset
}
```

```
start_op () {
  opcontrol --start
}

stop_op () {
  saveDir=${outDir}
  session=aaa$$bbb
  opcontrol --shutdown
  print opcontrol shutdown...
  opreport -l -p /lib/modules/$(uname -r)  > ${saveDir}/summary.out 2>&1
  print opreport summary...
  opreport -d -p /lib/modules/$(uname -r)  > ${saveDir}/detail.out 2>&1
  print opreport detail...
  opannotate -s -o ${saveDir}/src 2> ${saveDir}/opannotate.log
  print opannotate...
  opstack -f -p /lib/modules/$(uname -r) > ${saveDir}/callgraph.out 2>&1
  print opstack
  if [[ -d ${sampleDir}/${session} ]]; then
    mv ${sampleDir}/${session} ${sampleDir}/${session}.backup.$$
  fi
  opcontrol --save=${session}
  print opcontrol save...
  if [[ -f ${oprofileDir}/oprofiled.log ]]; then
    mv ${oprofileDir}/oprofiled.log ${saveDir}/oprofiled.log
  fi
  cat /proc/cpuinfo > ${saveDir}/cpuinfo
#  mv ${sampleDir}/${session} ${saveDir}/samples
}

################################################################################
#
# Main flow

while getopts "bsve:o" opt; do
  case ${opt} in
    b) withLKST=${opt}
       lkstAnalyzers='buffer blkqueue'
       lkstMaskset=buffer.blkqueue
       ;;
    s) withLKST=${opt}
       lkstAnalyzers='busywait spinlock'
       lkstMaskset=busywait.spinlock
       ;;
    v) withLKST=${opt}
       lkstAnalyzers='vmscan palloc'
       lkstMaskset=vmscan.palloc
       ;;
    e) OPevent="${OPevent} -e=$OPTARG"
       print OPevent ${OPevent}
       ;;
    o) noprofile=1
       OPevent="No profile";;
    ?) print opt ${opt}
       print_usage
  esac
done
```

```
shift $(($OPTIND - 1))
if [[ $# != 1 ]]; then
  print_usage
fi
pattern=$1

checkEnv
datadir ${pattern} ${withLKST} ${noprofile}
searchDevice

case ${pattern} in
   1) cleanfs 1 ;;
   2) cleanfs 1 ;;
   3) cleanfs 2 ;;
   4) cleanfs 2 ;;
   5) cleanfs 2 ;;
   6) cleanfs 3 ;;
   7) cleanfs 3 ;;
   8) cleanfs 3 ;;
   9) cleanfs 4 ;;
  10) cleanfs 4 ;;
  11) cleanfs 4 ;;
  12) cleanfs 2 ;;
   *) print Invalid pattern id: ${pattern}
      print 'Pattern id should be between 1 and 11.'
      exit 1
esac

[[ ${noprofile} = 1 ]] || prepare_op ${OPevent}
[[ ${withLKST} = 0 ]]  || prepare_lkst
[[ ${noprofile} = 1 ]] || start_op
[[ ${withLKST} = 0 ]]  || start_lkst
(vmstat 1 | ./chrononome.pl > ${outDir}/vmstat.log) &

uname -a> ${outDir}/uname
print =========== Start of id ${pattern} ${withLKST} cpu $CPUS ${OPevent} at $(date) ============

case ${pattern} in
   1) time ${IOZONE} ${defaultOptions} -s ${SIZE1} -t 1    -F /test/f1/io > ${outDir}/iozone.out ;;
   2) time ${IOZONE} ${defaultOptions} -s ${SIZE1} -t 1 -T -F /test/f1/io > ${outDir}/iozone.out ;;
   3) time ${IOZONE} ${defaultOptions} -s ${SIZE2} -t 2    -F ${dummyFiles2} ¥
 > ${outDir}/iozone.out ;;
   4) time ${IOZONE} ${defaultOptions} -s ${SIZE2} -t 2 -T -F ${dummyFiles2} ¥
 > ${outDir}/iozone.out ;;
   5)
    for num in 1 2; do
      ${IOZONE} ${defaultOptions} -s ${SIZE2} -t 1    -F /test/f${num}/io ¥
 > ${outDir}/iozone${num}.out &
    done ;;
   6) time ${IOZONE} ${defaultOptions} -s ${SIZE3} -t 3    -F ${dummyFiles3} ¥
 > ${outDir}/iozone.out ;;
   7) time ${IOZONE} ${defaultOptions} -s ${SIZE3} -t 3 -T -F ${dummyFiles3} ¥
 > ${outDir}/iozone.out ;;
   8)
    for num in 1 2 3; do
      ${IOZONE} ${defaultOptions} -s ${SIZE3} -t 1 -F /test/f${num}/io ¥
```

```
> ${outDir}/iozone${num}.out &
    done ;;
   9) time ${IOZONE} ${defaultOptions} -s ${SIZE4} -t 4  -F ${dummyFiles4} ¥
 > ${outDir}/iozone.out ;;
  10) time ${IOZONE} ${defaultOptions} -s ${SIZE4} -t 4 -T -F ${dummyFiles4} ¥
 > ${outDir}/iozone.out ;;
  11)
    for num in 1 2 3 4 ; do
#    for num in 1 2 3 4 5 6 7 8 9 10 11 12 13; do
      ${IOZONE} ${defaultOptions} -s ${SIZE4} -t 1    -F /test/f${num}/io ¥
 > ${outDir}/iozone${num}.out &
    done ;;
  12)
    cd /test/f1
    time tar xjf ${baseDir}/src/linux-2.6.12.4.tar.bz2
    cd linux-2.6.12.4
    cp -p ${baseDir}/src/config .config
    make oldconfig > /dev/null
    time make -j $CPUS > /dev/null 2>1
    cd ${baseDir}
    ;;
esac

while (ps ax|grep ${IOZONE##*/}|grep -v grep > /dev/null)
do
  sleep 5
done
echo IOZONE  ${IOZONE##*/}

killall vmstat
[[ ${withLKST} = 0 ]]  || stop_lkst
[[ ${noprofile} = 1 ]] || stop_op
[[ ${withLKST} = 0 ]]  || post_lkst

while (lsof |grep '/test/f'|grep -v grep > /dev/null)
do
  sleep 5
done

print ${outDir}
print ============= End of id ${pattern} ${withLKST} at $(date) ==============

exit 0
################################################################################
```

図 3.2-1run.sh スクリプト

また自動的に再起動、テスト実行を繰り返すテストスクリプトも同時に作成した。

実行するパターンの番号を OSBENCH_PATTERN に、LKST のマスクを LKSTMASK に定義しておくことによって、それぞれを自動実行する。またベンチマーク用のマシンの最大 CPU 数を MAXCPUS に定義する。実行の詳細は下記テストスクリプトを参照してほ

しい。

```
#!/bin/bash -x
#
##########################
# setting
OSBENCH_PATTERN=/var/local/osbench
LKSTMASK=/var/local/osbench.lkst
MAXCPUS=4
########################
#
# Main

if [[ ! -f $OSBENCH_PATTERN ]]; then
  echo Can not find os-bench pattern
  exit 1
fi
if [[ ! -f $LKSTMASK ]]; then
   echo Can not find LKST mask parameter for os-bench
   echo The parameter should be one of ¥'b¥', ¥'s¥', or ¥'v¥'
   exit 1
fi

echo OSBENCH_PATTERN $OSBENCH_PATTERN
read patternid < $OSBENCH_PATTERN
echo $patternid

case ${patternid} in
  1 | 2) pattern=1;
         next=9;;
  3 | 4 | 6 | 7 | 9 | 10)
         pattern=9;
         next=11;;
  5 | 8 | 11) pattern=11
         next=12;;
  12)    pattern=12;
         next=1;;
  *) echo pattern shuld be from 1 to 12
     exit 1;;
esac

CPUS=`grep processor /proc/cpuinfo|wc -l`
#if [[ $CPUS -ge $MAXCPUS ]]; then
#echo $next > $OSBENCH_PATTERN
#fi

uname -a

echo run.sh $pattern
# ./run.sh -o $pattern
# ./run.sh -o $pattern
# ./run.sh -o $pattern
# ./run.sh -o $pattern
# ./run.sh -o $pattern
```

```
./run.sh -e "BSQ_CACHE_REFERENCE:3000:0x200:1:1" $pattern
./run.sh -e "BSQ_CACHE_REFERENCE:3000:0x3f:1:1" $pattern
./run.sh -e "GLOBAL_POWER_EVENTS:100000:" $pattern
./run.sh -o  $pattern

if [[ $pattern = 1 ]]; then
 ./stress.sh $lkstmask 1
 ./stress.sh $lkstmask 2
fi
```

図 3.2-2test.sh スクリプト

下記の部分のコメントを外すと最大 CPU 数になったとき次の OSBENCH_PATTERN を実行する。CPU アイドル型（パターン 1,2)、CPU ビジー型(パターン 3,4,6,7,9,10)、ロック競合型(パターン 5,8,10)、カーネルソースビルド（パターン 12）の順で実行する

```
#if [[ $CPUS -ge $MAXCPUS ]]; then
#echo $next > $OSBENCH_PATTERN
#fi
```

図 3.2-3 パターンの切り替え

再起動したとき、テストを起動する処理は/etc/rc.local に埋め込み実際のテスト等の定義は、/etc/rebootbench.conf におく。ベンチマークマシンの最大 CPU 数を MAXCPUS に定義する。テスト用ディレクトリ、テストスクリプト名、テスト用カーネルの定義、正常系カーネルの定義、テストフェーズ、最大実行回数を/etc/rebootbench.conf に定義しておく。

表 3.2-1/etc/rebootbench.conf の定義

変数	意味
TESTPATH	テストディレクトリ
TESTPROG	テストスクリプト名
TESTKERN	テスト用カーネル grub.conf での番号
NORMKERN	通常カーネル grub.conf での番号
PHASE	-1,テストをしない。1 以上テストの実行回数
MAX_PHASE	最大実行回数

最大実行回数までリブート、テストを繰り返す。

```
TESTPATH=/os-bench
TESTPROG=./test.sh
TESTKERN=0
NORMKERN=1
PHASE=-1
MAX_PHASE=3
```

図 3.2-4/etc/rebootbench.conf の例

下記が/etc/rc.local の定義である。/etc/rebootbench.conf の定義に従って、リブート、テ

ストを繰り返す。

```
#!/bin/sh -x
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

 touch /var/lock/subsys/local

#reboot benchmark
MAXCPUS=4
. /etc/rebootbench.conf
CPUS=`grep processor /proc/cpuinfo|wc -l`
echo $CPUS > /var/log/osbench.cpus
OSBENCHLOG=/var/log/osbench.log
if [ $PHASE -ge 0 ]; then
  echo "Phase " $PHASE ": benchmark started"
  cd $TESTPATH
  echo "############################################" >> $OSBENCHLOG
  echo `date` >> $OSBENCHLOG
  $TESTPROG >> $OSBENCHLOG
  echo "benchmark finished ... rebooting"
  if [ $PHASE -ge $MAX_PHASE ] ; then
    echo "All benchmarks finished. reboot to normal"
    PHASE=-1
    /sbin/grubby --set-default=$NORMKERN
  else
   PHASE=`expr 1 + $PHASE`
   CPUS=`expr  $CPUS + $CPUS`
    if [ $CPUS -gt $MAXCPUS ] ; then
      CPUS=1
    fi
    /sbin/grubby --set-default=$TESTKERN  --update-kernel=$TESTKERN --args=maxcpus=$CPUS
  fi
  cat > /etc/rebootbench.conf << FEOF
TESTPATH=$TESTPATH
TESTPROG=$TESTPROG
TESTKERN=$TESTKERN
NORMKERN=$NORMKERN
PHASE=$PHASE
MAX_PHASE=$MAX_PHASE
FEOF
reboot
fi
```

図 3.2-5/etc/rc,local

4　性能･信頼性評価結果と分析･考察

4.1　Iozone の結果

4.1.1　概要

2004 年度は、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 4.1-1)できることがわかった。

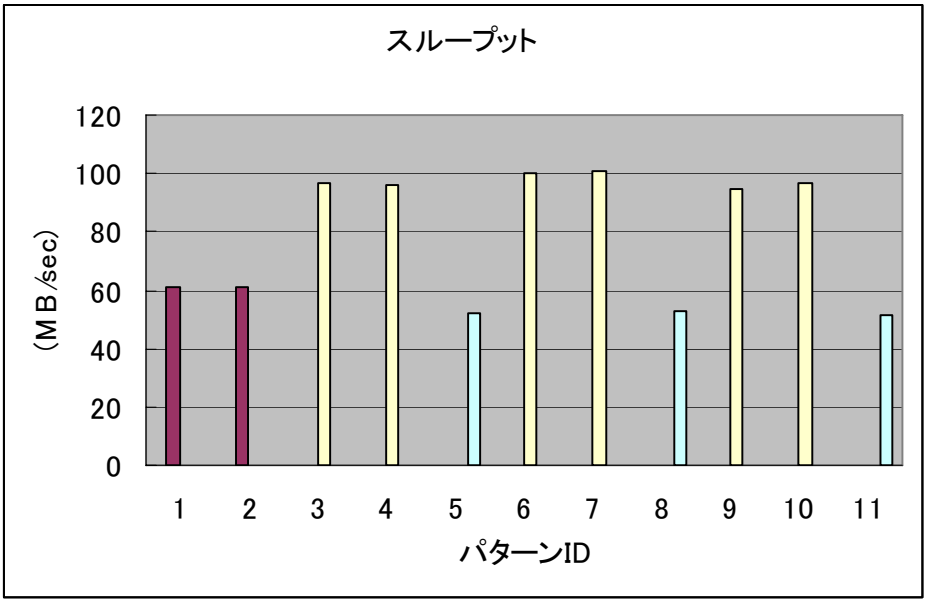


図 4.1-1 実行パターンとスループット

表 4.1-1I/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、I/O 待ちもロック競合による待ちも無く CPU をほぼ 100%使い切ったパターンで、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。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は２００４年度に手順化した。

スループットの絶対値だけでは、CPU アイドル型とロック競合型を明確に区別することは難しいが、OProfile および LKST で分析すれば一目瞭然にその差を区別することができる。

CPU アイドル型の場合、性能向上はソフトウェア的な解決策ではなく、I/O デバイスのバンド幅の向上(高速化)によってはかられると考えた。そこで今年度は CPU ビジー型とロック競合型に注目して、性能限界を特定し、性能向上を試みた。

4.1.2 CPU ビジー型ベンチマーク結果

CPU ビジー型については、パターン ID 9 を取り上げる。

表 4.1-2 パターン ID9 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	×	1

表 4.1-3 Iozone 結果/pattern9-0-cpu16-08031605

所要時間(Wall time)	27.256 秒
ファイルサイズ合計	8,186,880.00kB
スループット合計	364,908.53 kB/sec

表 4.1-4 プロファイリング全体結果/ pattern9-0-cpu16-08031605

順位	シンボル	サンプル数	占有率(%)	累積占有率(%)
1	__copy_from_user_ll	3201368	28.54	28.54

2	_spin_lock_irq	1870823	16.68	45.22
3	no symbols/jbd	1071441	9.55	54.77
4	_spin_lock	769409	6.86	61.63
5	no symbols/ext3	484137	4.32	65.95

__copy_from_user_llを細分化した結果が表 4.1-5の通り。

表 4.1-5 プロファイリング__copy_from_user_ll 詳細/2.6.9-11.5AX

アドレス	サンプル数	__copy_from_user_ll に対する占有率(%)		命令
c01c03d4	3036318	92.71		repz movsl %ds:(%esi),%es:(%edi)
c01c03d6	96922	2.96		mov %eax,%ecx
c01c03d8	6999	0.21		repz movsb %ds:(%esi),%es:(%edi)
c01c03da	110331	3.37		jmp c01c03e5
c01c03e6	23495	0.71		mov %ecx,%eax

命令は、objdump コマンドを使用して確認した。

```
# objdump -S /usr/lib/debug/lib/modules/2.6.9-11.5AXsmp/vmlinux
```

```
#define __copy_user_zeroing(to,from,size)                                ¥
do {                                                                     ¥
        int __d0, __d1, __d2;                                            ¥
        __asm__ __volatile__(                                            ¥
                "       cmp  $7,%0¥n"                                    ¥
                "       jbe  1f¥n"                                       ¥
                "       movl %1,%0¥n"                                    ¥
                "       negl %0¥n"                                       ¥
                "       andl $7,%0¥n"                                    ¥
                "       subl %0,%3¥n"                                    ¥
                "4:     rep; movsb¥n"                                    ¥
                "       movl %3,%0¥n"                                    ¥
                "       shrl $2,%0¥n"                                    ¥
                "       andl $3,%3¥n"                                    ¥
                "       .align 2,0x90¥n"                                 ¥
                "0:     rep; movsl¥n"                                    ¥
                "       movl %3,%0¥n"                                    ¥
                "1:     rep; movsb¥n"                                    ¥
                "2:¥n"                                                   ¥
                ".section .fixup,¥"ax¥"¥n"                               ¥
                "5:     addl %3,%0¥n"                                    ¥
                "       jmp 6f¥n"                                        ¥
                "3:     lea 0(%3,%0,4),%0¥n"                             ¥
                "6:     pushl %0¥n"                                      ¥
                "       pushl %%eax¥n"                                   ¥
                "       xorl %%eax,%%eax¥n"                              ¥
                "       rep; stosb¥n"                                    ¥
                "       popl %%eax¥n"                                    ¥
                "       popl %0¥n"                                       ¥
                "       jmp 2b¥n"                                        ¥
                ".previous¥n"                                            ¥
                ".section __ex_table,¥"a¥"¥n"                            ¥
                "       .align 4¥n"                                      ¥
                "       .long 4b,5b¥n"                                   ¥
                "       .long 0b,3b¥n"                                   ¥
                "       .long 1b,6b¥n"                                   ¥
                ".previous"                                              ¥
                : "=&c"(size), "=&D" (__d0), "=&S" (__d1), "=r"(__d2)    ¥
                : "3"(size), "0"(size), "1"(to), "2"(from)               ¥
                : "memory");                                             ¥
} while (0)
```

__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。。

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。

4.1.3 ロック競合型ベンチマーク結果

ロック競合型については、パターン ID 11 を取り上げる。

表 4.1-6 条件/pattern11-0-cpu4-08031840

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
11	650M	1	×	13

表 4.1-7 Iozone 結果/pattern11-0-cpu4-08031840

	合計
所要時間	43.89 秒(平均)
ファイルサイズ	8,186,880kB
スループット	186,582.62 kB/sec

表 4.1-8 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	__copy_from_user_ll	2307055	47.66	47.66
2	no symbols/jbd	604496	12.49	60.15
3	no symbols/ext3	221320	4.57	64.72
4	kunmap_atomic	200826	4.15	68.87
5	_spin_lock	131181	2.71	71.58
6	_spin_lock_irq	71774	1.48	73.06
7	_spin_unlock	57036	1.18	74.24
8	LKST_ETYPE_BUFFER_SUBMIT_BH_HEADER_hook	46978	0.97	75.21
9	_spin_lock_irqsave	43678	0.9	76.11
10	release_pages	42796	0.88	77

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。

4.2 分析

4.2.1 ロック競合型分析

まず始めにプロファイリング結果を解析する。__spin_lock()をブレークダウンした結果は表 4.2-1のようになる。

表 4.2-1 プロファイリング 詳細

アドレス	サンプル数	_spin_lock に対する占有率 (%)	命令
c02d580f	478148	19.5423	test %al,%al
c02d581c	1096486	44.8143	cmpb $0x0,(%ebx)
c02d581f	678143	27.7162	jle c02d581a <_spin_lock+0x3d>

命令は、objdump コマンドを使用して確認した。

```
# objdump -d /usr/lib/debug/lib/modules/2.6.9-11.5AXsmp/vmlinux
```

ソースコードとの突合せを行った結果、

```
c02d5807:       89 c6                   mov    %eax,%esi
c02d5809:       89 d7                   mov    %edx,%edi
c02d580b:       31 c0                   xor    %eax,%eax
c02d580d:       86 03                   xchg   %al,(%ebx)
c02d580f:       84 c0                   test   %al,%al
c02d5811:       7f 10                   jg     c02d5823 <_spin_lock+0x46>
c02d5813:       eb 05                   jmp    c02d581a <_spin_lock+0x3d>
c02d5815:       f0 fe 0b                lock decb (%ebx)
c02d5818:       79 09                   jns    c02d5823 <_spin_lock+0x46>
c02d581a:       f3 90                   repz nop
c02d581c:       80 3b 00                cmpb   $0x0,(%ebx)
c02d581f:       7e f9                   jle    c02d581a <_spin_lock+0x3d>
c02d5821:       eb f2                   jmp    c02d5815 <_spin_lock+0x38>
c02d5823:       83 3d a4 f0 34 c0 00    cmpl   $0x0,0xc034f0a4
c02d582a:       79 25                   jns    c02d5851 <_spin_lock+0x74>
```

同様にブレークダウンした結果は表 4.2-2の通り。

表 4.2-2 OProfile による実行時間/pattern11-s-cpu8-08032103/summary.out

サンプル数	全体に対する占有率(%)	関数名
2446735	15.9751	_spin_lock
2406729	15.7139	__copy_from_user_ll
1695754	11.0718	_spin_unlock

そこで、LKST によって busywait を測定してみた。当該アドレス、呼び出し場所、回数、平均待ち時間、最大待ち時間、最小待ち時間、合計待ち時間となっている。合計待ち時間が大きければ、spinlock によってあるプロセスが長く待たされているということになる。

```
# head pattern11-s-cpu8-08031323/lkst.data.busywait.s
busywait time distribution analyzer
address   calling_point     count         average             max             min           total
c0104a15  sys_execve+45          7     0.000000092     0.000000132     0.000000084     0.000000645
c0104fa3    __down_trylock+11    5     0.000000084     0.000000084     0.000000083     0.000000418
c0105107 sys_rt_sigsuspend+8f    9     0.000000085     0.000000097     0.000000082     0.000000761
c0105445     sys_sigreturn+60   10     0.000000087     0.000000121     0.000000080     0.000000874
c0105b4d     handle_signal+93   10     0.000000405     0.000003063     0.000000082     0.000004050
```

```
c01074b0  show_interrupts+68    239      0.000000202      0.000001273      0.000000082      0.000048329
c0107b0d         do_IRQ+6       6359     0.000000098      0.000002994      0.000000080      0.000623796
c0107bb7 LKST_ETYPE_INT_HARDWARE+6b        6359      0.000000094      0.000002925      0.000000080
0.000600680
```

図 4.2-1LKST による busywait/ pattern11-s-cpu8-08031323/lkst.data.busywait.s

待ち時間合計でソートしてみると下記のとおりになる。

```
# sort -nr -k 7 pattern11-s-cpu8-08031323/lkst.data.busywait.s|head
c0142233 find_get_page+e      97924     0.000002473      0.000247097      0.000000080      0.242189948
c0121e70 scheduler_tick+12b   39703     0.000000512      0.000009898      0.000000080      0.020333137
c0113984 mark_offset_tsc+57    6235     0.000001608      0.000005250      0.000000875      0.010023539
c012de5a run_timer_softirq+32 40315      0.000000164      0.000005101      0.000000080      0.006605918
c016297e      put_super+d      1606     0.000003573      0.000115192      0.000000080      0.005738664
c0179ba6 get_super_to_sync+c   1143     0.000003625      0.000150842      0.000000080      0.004143312
c01219c1    load_balance+17   30107     0.000000105      0.000003688      0.000000080      0.003167752
c0113944 mark_offset_tsc+17    6235     0.000000340      0.000003889      0.000000080      0.002122548
c012d6d8   __mod_timer+98      2578     0.000000400      0.000004588      0.000000080      0.001030253
c0162e6c sync_filesystems+76    552     0.000001489      0.000153222      0.000000080      0.000821915
```

図 4.2-2LKST による待ち時間合計の大きい順

最大の find_get_page での待ち時間合計でも 0.24 秒である。実行時間が 50 秒前後のベンチマークのうちの 0.24 秒なので、性能に大きな影響を与えているとは考えられない。

もしここで、LKST によって長い待ち時間が発生しているようなスピンロックを発見できれば、その部分のソースコードを解析し、なんらかの対策を講じることによって性能向上をはかることを試みることができる。しかし今回はそのようなたちの悪いスピンロックを発見できなかった。

2004 年度の調査では、10 秒を超えるビジーウェイトが発生していたが今回の調査ではそのようなものは一切なかった。

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、2004 年度調査によっても発見できたカーネル 2.4（MIRACLE LINUX V3.0）にはまだある程度残っていた。今回調査したカーネル 2.6(MIRACLE LINUX V4.0 ベータ)では特に問題となるようなグローバルなカーネルロックは発見できなかった。

4.2.2 CPU ビジー型およびロック競合型のスケーラビリティ

CPU ビジー型(パターン 9)およびロック競合型（パターン 11）のスケーラビリティについて CPU 数を変化させスループット(MB/秒)で比較してみた。どちらもほぼ同じスループ

ットを持ち、スケーラビリティ上の問題は特に発見できなかった。CPU 数が 4 を超えると I/O 待ちが発生し CPU アイドル型になってしまった。十分な I/O バンド幅を用意できればさらにスケールアップしたと考えられる。

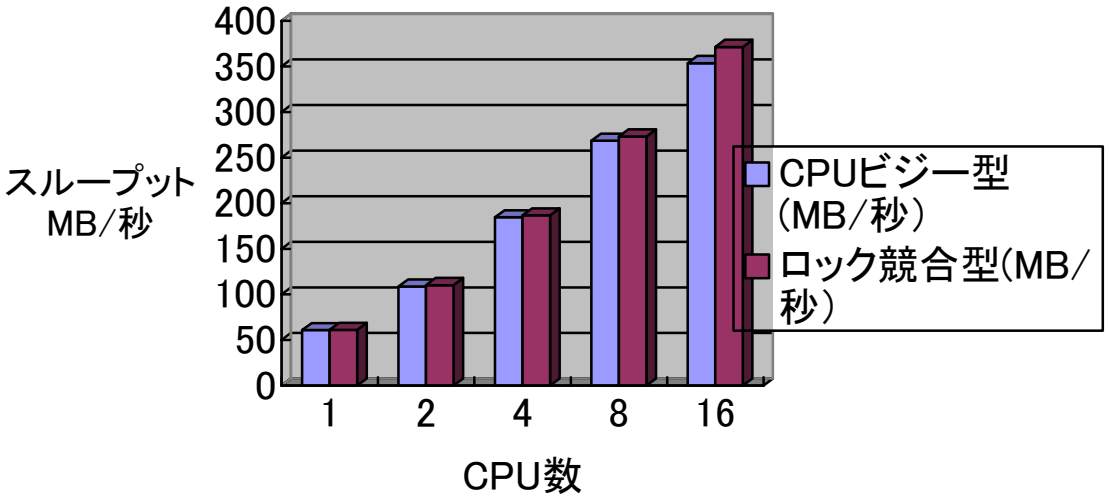


図 4.2-3CPU スケーラビリティ

4.2.3 キャッシュミスに注目したベンチマーク

今回実施した CPU ビジー型およびロック競合型のベンチマーク結果よりどちらもスケーラビリティを阻害するような重大な問題は発見できなかった。カーネルが 2.4 から 2.6 へ進化する過程で多くのスケーラビリティ上の改良が積み重ねられた結果だといえる。

OProfile と LKST によって各ベンチマークの特性について検討したが、

1. アルゴリズム的に無駄な部分は発見できなかった。
2. たちの悪いロック待ちは発見できなかった。

そこで、もう一度、OProfile において CPU ビジー型の問題点について分析してみることにした。当該ソースコードを分析すると、__copy_from_user_ll()の
repz movsl %ds:(%esi),%es:(%edi) というところであった。表 4.1-5を参照)

これは%ESI レジスタで示されるアドレスから%EDI レジスタで示されるアドレスへのデータのコピー命令である。これ自体はユーザ空間からカーネル空間へデータをコピーするのに必要で、特にアルゴリズム的に問題とは考えられない。むしろ効率的に I/O 処理が行われているといえる。

ここでは大量のコピーをするため、cache pollution の発生を疑った。cache pollution というのは、キャッシュが時間的にすぐに利用しないデータによってキャッシュが埋められてしまい結果的に時間的にすぐに利用されるデータをキャッシュから追い出されることをいう。そのために性能上の問題が発生する。

4.2.3.1 L3 キャッシュミスの測定

そこで、L3 キャッシュミスを測定してみる。

```
# opcontrol –e=BSQ_CACHE_REFERENCE:3000:0x200:1:1
```

下記のとおり、__copy_from_user_ll()でキャッシュミスが多発しているのが分かる。約63%のキャッシュミスが、当該ルーチンで発生していて、2 位以下の寄与率は 2%以下である。この関数は非常にたちの悪い L3 キャッシュミスを起こしているというのがよくわかる。

```
CPU: P4 / Xeon, speed 2201.02 MHz (estimated)
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a
unit mask of 0x200 (read 3rd level cache miss) count 3000
samples  %        app name                 symbol name
37260    63.1772  vmlinux                  __copy_from_user_ll
1039      1.7617  vmlinux                  _spin_lock_irqsave
935       1.5854  vmlinux                  _spin_lock
925       1.5684  vmlinux                  blk_rq_map_sg
910       1.5430  vmlinux                  journal_add_journal_head
```

図 4.2-4OProfile による L3 キャッシュミス測定 pattern9-0-cpu4-0-08141508/summary.out

一方で、L2/L3 アクセスは以下のとおりである。__copy_from_user_ll()のアクセスは全体の 7.3%程度しかないのに比べてキャッシュミスは 63.1%とその多さが際立っている。

```
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit)
with a unit mask of 0x3f (multiple flags) count 3000
samples  %        app name                 symbol name
118812    7.3122  vmlinux                  __copy_from_user_ll
82947     5.1049  vmlinux                  _spin_lock
65983     4.0609  vmlinux                  journal_add_journal_head
61903     3.8098  vmlinux                  __find_get_block
59457     3.6593  vmlinux                  journal_dirty_metadata
```

図 4.2-5OProfile による L2/L3 キャッシュアクセス pattern9-0-cpu4-0-08141510/summary.out

4.2.3.2 Cache pollution aware patch

write(2)は、ユーザ空間からカーネル空間へデータを移動するのがその主な機能である。実際にデータをハードディスクへ書き出すのは非同期で別のカーネルプロセスが執り行う。ということは、カーネル空間（ページキャッシュ）へ転記されたデータはすぐにアクセス

されるとは限らない。この性質に注目して、カーネル空間へデータを移動するときに、キャッシュをバイパスし、カーネル空間に書き込めば、本来キャッシュに残るべきデータがキャッシュから追い出されること（これを cache pollution と呼ぶ）を防げるのではないだろうかと考えた。

そして IA-32 のキャッシュをバイパスする命令を利用してレジスタからメモリへデータをコピーするパッチを作成し cache pollution aware patch と名づけた。具体的には下記の部分である。

```
+__copy_user_zeroing_intel_nocache(void *to, const void __user *from, unsigned long
size)
+{
+        int d0, d1;
+
+       __asm__ __volatile__(
+                       ″        .align 2,0x90¥n″
+                       ″0:      movl 32(%4), %%eax¥n″
+                       ″        cmpl $67, %0¥n″
+                       ″        jbe 2f¥n″
+                       ″1:      movl 64(%4), %%eax¥n″
+                       ″        .align 2,0x90¥n″
+                       ″2:      movl 0(%4), %%eax¥n″
+                       ″21:     movl 4(%4), %%edx¥n″
+                       ″        movnti %%eax, 0(%3)¥n″
+                       ″        movnti %%edx, 4(%3)¥n″
+                       ″3:      movl 8(%4), %%eax¥n″
+                       ″31:     movl 12(%4),%%edx¥n″
+                       ″        movnti %%eax, 8(%3)¥n″
+                       ″        movnti %%edx, 12(%3)¥n″
以下略
```

図 4.2-6Non-temporal move 命令

movl 0(%4), %%eax および movl 4(%4), %%edx でユーザ空間アドレスのデータを EAX レジスタおよび EDX レジスタへコピーし、movnti %%eax, 0(%3)、および movnti %%edx, 4(%3) で EAX レジスタおよび EDX レジスタにコピーされたデータをキャッシュをバイパスしてカーネル空間にコピーする。movnti 命令がキャッシュをバイパスする命令である。この 2 組の命令で 8 バイト転送できる。これを 8 セット組み合わせることによって、一回のループで 64 バイト転送している。ループアンローリングを手動で行ったわけだが、データ依存性がまったくないので、ハードウェア的には並列実行されているようである。Pentium 4 系のプロセッサは、深いパイプライン、アウトオブオーダー実行、レジスタリネーミングな

どの機能により８セットの転送がほぼストールすることなく並列に行われているようである。（ソフトウェア的には確認できないが実行性能上そのように推測される。）
またキャッシュをバイパスする命令を使うことによって、cache pollution が減少することが期待できる。

4.2.4 Cache pollution aware patch の評価

CPU ビジー型のパターン９で評価した。

オリジナルと改良版のiozoneの実行時間をすると(表 4.2-3 オリジナル版と改良版 iozone.out)CPU時間でおよそ 11.9%(100-88.1)ほど減少しているのが見て取れる。

表 4.2-3 オリジナル版と改良版 iozone.out

	pattern9-0-cpu4-0-08141700 オリジナル	pattern9-0-cpu4-0-08141627 改良版	性能比
スループット	181090.50KB/sec	186029.54KB/sec	1.027
CPU wall time	46.263	45.385	98.1%
CPU time	67.397	59.400	88.1%

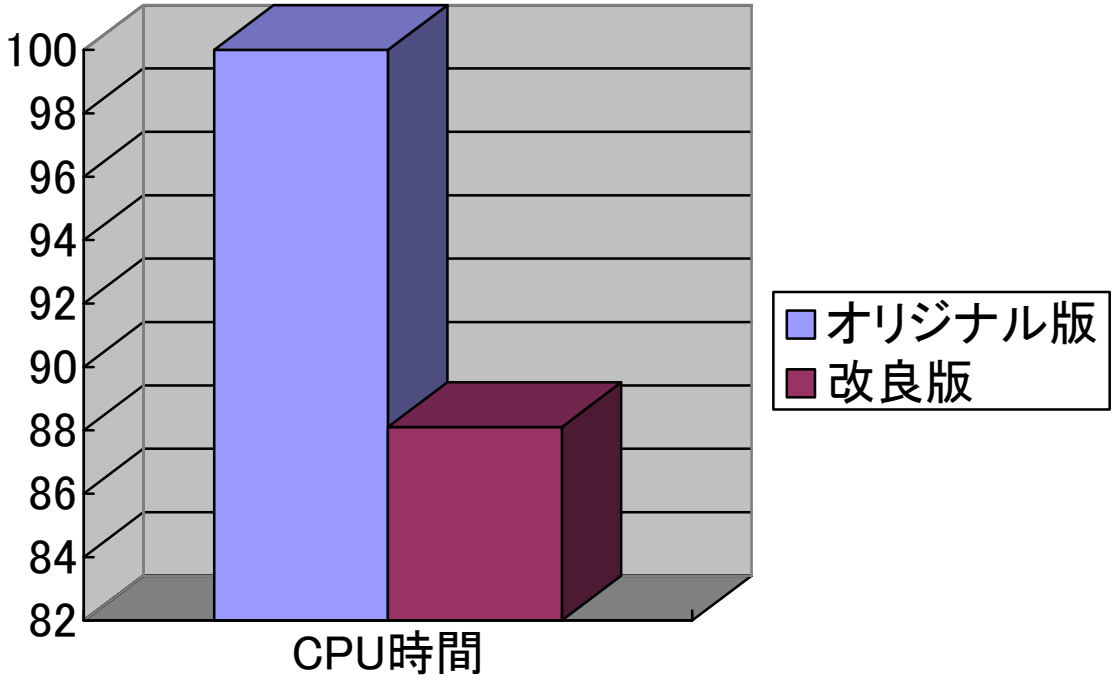


図 4.2-7 オリジナル版と改良版の CPU 時間の比率

キャッシュミスについては下記のとおりである。

表 4.2-4 オリジナル版と改良版の L3 キャッシュミスイベント数

	pattern9-0-cpu4-0-08141508 オリジナル oprofiled.log	pattern9-0-cpu4-0-08141455 改良版	比率
L3 cache miss	58978	15112	25.6%

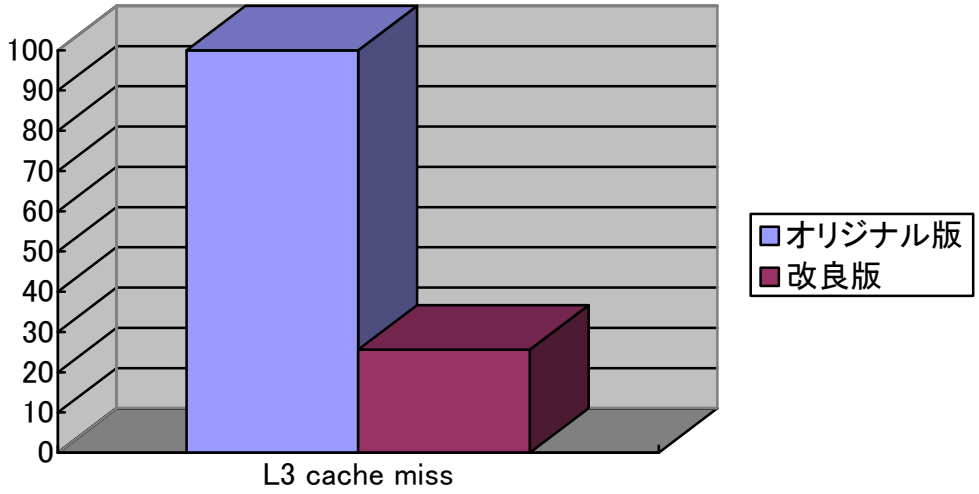


図 4.2-8L3 キャッシュミス

改良版の L3 キャッシュミスはオリジナル版のおよそ 25.6%である。8 月 14 日時点で、__copy_from_user_ll()の改良版関数として_mmx_memcpy_nt()というパッチを作った。そのキャッシュミスをオリジナルのものと比較すると、下記のようにほぼ 0 になっている。

表 4.2-5 当該関数の L3 キャッシュミス

	pattern9-0-cpu4-0-08141508 __copy_from_user_ll	pattern9-0-cpu4-0-08141455 _mmx_memcpy_nt	比率
L3cache miss	37260	37	0.099%

以上のようにキャッシュをバイパスする命令（IA-32 の Non-Temporal Move 命令）を利用することによって効果的に L3 キャッシュミスを防止することができ CPU 時間においても 10%前後の性能向上達成することができた。

4.2.5 Cache pollution aware patch の改良

上記のようなパッチを Linux Kernel Mailing List (LKML)に投稿したところ、次のようなコメントを得た。

1. 常にキャッシュをバイパスするように__copy_from_user_ll()を変更するのではなく、キャッシュをバイパスする関数と従来の関数を分けるべきだ。
2. 分けた上でキャッシュをバイパスする関数の有効性が確認できるところのみにそれを利用するべきだ。

そこで__copy_from_user_ll_nocache()という関数を新規に作り、write(2)から呼ばれている箇所でのみ新規に作成した関数を利用することにした。しかし、カーネルのソースコードの中にはいたるところに copy_from_user()が利用されていてどこを変更するのが効果的なのか簡単には判定できそうになかった。

```
# find -name '*.[chS]' | xargs egrep copy_from_user | wc –l
2848
```

図 4.2-9 カーネルソース内での copy_from_user 利用数

カーネルソースで copy_from_user の出現する回数を数えたところ 2848 箇所発見できた。これを逐一調査することは量的にも簡単ではない。

そこで、OProfile のコールグラフ機能を利用してどこから copy_from_user が呼ばれているか調査した。

```
# opcontrol --callgraph=#depth
```

図 4.2-10call graph の設定例

opcontrol で#depth に測定すべき深さを指定する。コールグラフの計測はオーバヘッドがかかるので注意が必要である。

計測後 opstack でコールグラフのレポートを作成し分析する。

```
#   opstack -f -p /lib/modules/$(uname -r) > ${saveDir}/callgraph.out 2>&1
```

図 4.2-11call graph のレポート作成

以下に実行例を示す

```
  self      %          child     %          app name                    symbol name
  8537      30.9671    27596     3.8719     vmlinux generic_file_aio_write
  4157      15.0791    325670    45.6934    vmlinux __generic_file_aio_write_nolock
  14874     53.9539    359462    50.4347    vmlinux generic_file_buffered_write
14874       0.8457     359462    20.4370    vmlinux generic_file_buffered_write  <ここ
  14874     3.0504     359462    59.7334    vmlinux generic_file_buffered_write
  2521      0.5170     31132     5.1733     vmlinux __ext3_journal_stop
  11626     2.3843     26401     4.3872     vmlinux walk_page_buffers
```

図 4.2-12 コールグラフレポート

ここでは、generic_file_buffered_write()という関数に注目した場合、上側がそれを呼んでいる関数、下側が generic_file_buffered_write()から呼んでいる関数になる。ここでは 14874 回サンプリングされている。generic_file_buffered_write()が呼んでいる関数（子の関数）のサンプリング総数は 359462 回である。さてここで__copy_from_user_ll()の場合は下記のとおりである。

```
  0           0   6        8.8e-04   vmlinux copy_from_user
```

```
  4157     21.8433  325670  47.5335   vmlinux __generic_file_aio_write_nolock
  14874    78.1567  359462  52.4656   vmlinux generic_file_buffered_write
241263   13.7168  1        5.7e-05  vmlinux __copy_from_user_ll
  1        100.000  0              0  vmlinux __copy_user_zeroing_intel
```

図 4.2-13 コールグラフ(__copy_from_user_ll)

copy_from_user()/__generic_file_aio_write_nolock()/generic_file_buffered_write()という関数から呼ばれていることが分かる。

そこで generic_file_buffered_write()のソースコードを確認し、当該関数にパッチをあて、新規に作成したキャッシュをバイパスする関数__copy_from_user_inatomic_nocache()を呼び出すようにした。

```
# diff -u filemap.c.orig filemap.c|less
--- filemap.c.orig      2005-08-05 16:04:37.000000000 +0900
+++ filemap.c   2005-08-16 10:16:06.000000000 +0900
@@ -1727,13 +1727,13 @@
        int left;

        kaddr = kmap_atomic(page, KM_USER0);
-       left = __copy_from_user_inatomic(kaddr + offset, buf, bytes);
+       left = __copy_from_user_inatomic_nocache(kaddr + offset, buf, bytes);
        kunmap_atomic(kaddr, KM_USER0);

        if (left != 0) {
                /* Do it the slow way */
                kaddr = kmap(page);
-               left = __copy_from_user(kaddr + offset, buf, bytes);
+               left = __copy_from_user_nocache(kaddr + offset, buf, bytes);
                kunmap(page);
        }
        return bytes - left;
@@ -1750,7 +1750,7 @@
                int copy = min(bytes, iov->iov_len - base);

                base = 0;
-               left = __copy_from_user_inatomic(vaddr, buf, copy);
+               left = __copy_from_user_inatomic_nocache(vaddr, buf, copy);
                copied += copy;
                bytes -= copy;
```

```
                vaddr += copy;
```

図 4.2-14filemap.c のパッチ

上記パッチをあてたバージョンに関しても同様に評価をした。

表 4.2-6 オリジナル版と改良版 iozone.out

	pattern9-0-cpu4-0-08141700 オリジナル	pattern9-0-cpu4-0-08190838 改良版	性能比
スループット	181090.50KB/sec	187870.50KB/sec	1.037
CPU wall time	46.263	44.242	95.6%
CPU time	67.397	63.905	94.8%

キャッシュミスについては下記のとおりである。

表 4.2-7 オリジナル版と改良版の L3 キャッシュミスイベント数

	pattern9-0-cpu4-0-08141508 オリジナル oprofiled.log	pattern9-0-cpu4-0-08190859 改良版	比率
L3 cache miss	58978	17398	29.5%

キャッシュミスを劇的に減少させている。

表 4.2-8 当該関数の L3 キャッシュミス

	pattern9-0-cpu4-0-08141508 __copy_from_user_ll	pattern9-0-cpu4-0-08141455 改良版	比率
L3cache miss	37260	51	0.13%

改良版の関数は__copy_user_zeroing_inatomic_nocache()である。

OProfile のコールグラフ機能を利用することによって効率的にある関数を呼んでいる関数を発見することができた。非常に多くの箇所から呼ばれている関数だと、その関数を不用意に変更すると予期しない副作用が発生する場合がある。特に OS カーネルの場合その影響が広範囲にわたるため汎用的な関数の変更は注意深くやる必要がある。

__copy_from_user_ll()はソースコード上 2848 箇所で呼ばれていたが、コールグラフ機能によって、効率的に今回のベンチマークで実行時間がかかっている関数を発見でき、3 箇所を変更するだけで、上記の効果を得た。3 箇所だけ新規に開発したキャッシュをバイパスする関数を呼ぶようにしたので、残りに関しては一切変更していないため、副作用は一切ないといえる。

4.2.6 Cache pollution の影響

データ参照には下記の性質がある。

表 4.2-9 データ参照の性質

名前	性質
時間的局所性(Temporal)	データは時間的にすぐに利用される
空間的局所性(Spatial)	データは空間的にそばのものが利用される
時間的非局所性(Non-temporal)	時間的にすぐには利用されない

例えば時間的非局所性の場合、あるデータは一度利用された後、すぐ利用されない。一方時間的局所性がある場合は、すぐに利用されるので、キャッシュに残しておけばアクセスコストを削減することができる。

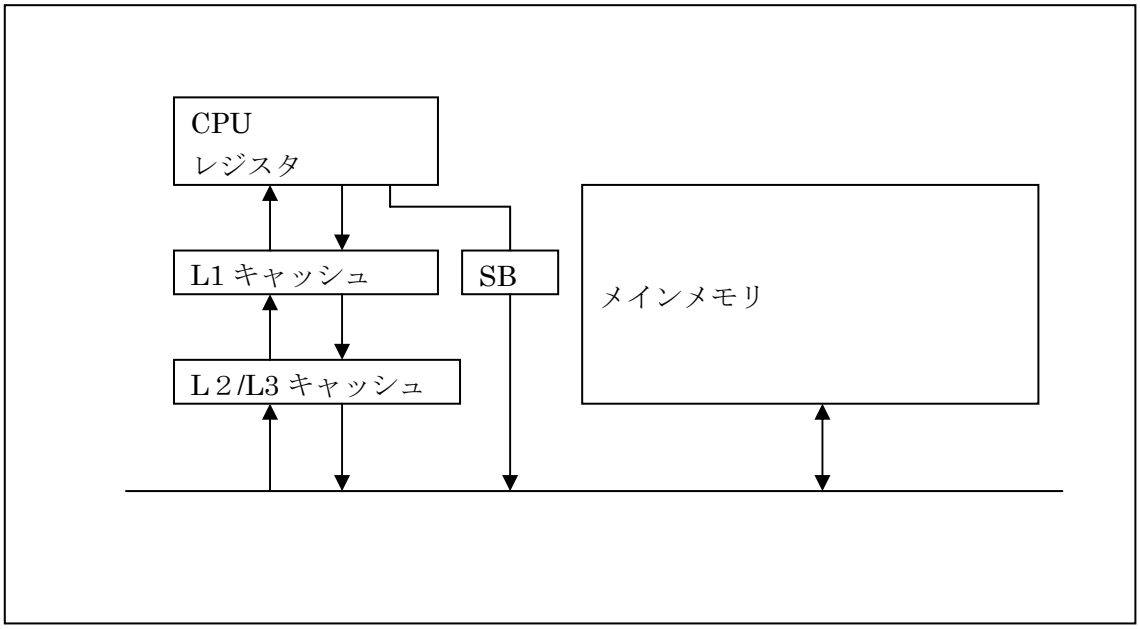


図 4.2-15CPU、メモリ、キャッシュ構成図

メインメモリからデータをアクセスする場合、そのメモリがキャッシュ可能なら、適当なキャッシュにデータを置く。これを cache line fill とよぶ。次回データを読むとき、同じデータがキャッシュにあれば、CPU はメインメモリからではなくキャッシュからデータを読むことができる。これをキャッシュヒットと呼ぶ。Pentium 4 の場合、キャッシュラインは 64 バイトなので、たとえば 4 バイトのアクセスだとしてもいっきに 64 バイトキャッシュに読み込む。隣の 4 バイトアクセスする場合、すでにキャッシュに載っているのでキャッシュヒットをするため、アクセスが高速化される。

CPUがキャッシュ可能なメモリへ書き込む場合、最初キャッシュにそのメモリのキャッシュラインが存在するかチェックする。有効なキャッシュラインが存在した場合、CPUは（書き込みポリシーに依存するのだが)、メインメモリではなくキャッシュに書き込む。これをライトヒット(write hit)と呼ぶ。もしキャッシュをライトミスしたら（有効なキャッシュライン存在しない)、CPUはcache line fill, write allocationを行う。データをキャッ

シュラインに書き、そしてメインメモリへ書く。もしデータがメモリへ書かれるなら、それは最初にストアバッファ(Store Buffer)へ書き込まれ、システムバスが利用できるようになったら、ストアバッファ[4]からメモリへ書き込まれる

書き込みの場合、すぐにそのデータを読み込まないのであれば（時間的局所性がない場合)、キャッシュにそのデータを置いてもアクセスを高速化することにはならないのに注意したい。

CPU がキャッシュにデータを置くとき CPU はそのラインを新しいデータに置き換えるが、置き換えられたデータがすぐに必要になる場合もある。必要なデータを置き換えてしまったとき cache conflict あるいは cache pollution が発生したという。

つまり、時間的局所性のないデータをキャッシュに置くことは、cache pollution 等の副作用を考えると好ましくない。

ユーザ空間のデータをカーネル空間へコピーするときの CPU の動作はおおむね下記のようになる。

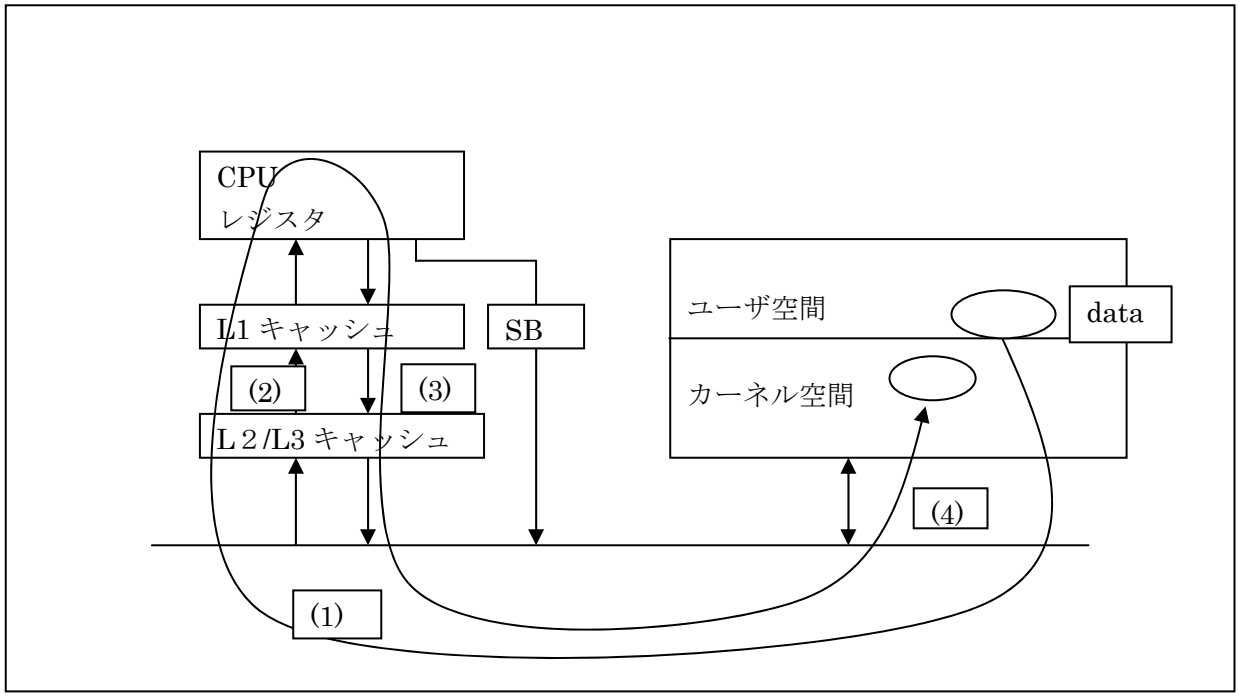
（1） ユーザ空間からレジスタへコピー

（2） キャッシュミスが発生した場合 cache fill が発生

（3） レジスタからカーネル空間へコピー。cache line fill, write allocation が発生

（4） ストアバッファからカーネル空間へ write back

というような動作になる。



ここで（3）では、読み込むアドレスと書き込むアドレスが同一ならば、キャッシュにデータが乗っているのでキャッシュヒットする(ライトヒットする)。今回の場合は読み込むアドレス（ユーザ空間）と書き込むアドレス（カーネル空間）が異なるので、常にキャ

[4] Pentium 4 およびIntel Xeon プロセッサは 24 エントリのストアバッファを持つ。

ッシュミスをする。キャッシュミスをしたためキャッシュラインを確保するために有効なキャッシュのどれかを捨てることになる。時間的な局所性のあるデータを捨てると、すぐにデータがアクセスされるので、そのときはキャッシュミスを発生させる。これは cache pollution によるキャッシュミスである。書き込み時にキャッシュをバイパスするとライトミスは 0 になるとともに、時間的局所性のあるデータを捨てないので、cache pollution によるキャッシュミスが減少する。

キャッシュミス＝ライトミス＋cache pollution によるキャッシュミス

ここでライトミスは copy_from_user_ll()での L３キャッシュミスとして近似すると cache pollution によるキャッシュミスが求められる。

表 4.2-10cache pollution によるキャッシュミス

	L3 キャッシュミス	ライトミス copy_from_user_ll	cache pollution によるキャッシュミス
pattern9-cpu4 現行	58466	37125	21341(1)
pattern9-cpu4 改良	17583	0	17583(2)
現行―改良の差	40883	37125	3758(3)

オリジナル版のライトミス以外のキャッシュミスは上記の表の(1)である。改良版はライトミスが 0 で、キャッシュミスは(2)なので、cache pollution により(3)の 3758 回のキャッシュミスを発生させていたことになる。今回のパッチによって L3 キャッシュミスのうち、cache pollution によるキャッシュミスを削減できたことになる。

今回開発した cache pollution aware patch の元となったアイデアの cache pollution という性質それ自身は昔から知られていた。しかしながら cache pollution に着目したパッチが Linux カーネルに取り込まれることは今までなかった。これは、効果的に cache pollution を発生している場所を見つけることがツールなしでは困難だということ、そして OProfile 等のツールの整備が近年やっとなされてきたことなどが主な原因かと思われる。cache pollution という観点からカーネルの性能向上に着目した研究は著者の調査の範囲では発見できなかった。

cache pollution aware patch は IA-32 の Non-temporal 命令を利用することによって、メモリへの書き込みでキャッシュをバイパスする。そのため、cache pollution を発生しない。時間的局所性のあるデータをキャッシュから追い出さないので cache pollution による L3 キャッシュミスが減少した。

4.3 考察

4.3.1 キャッシュミス削減の重要性

CPU の高速化のペースは年率 50%前後であるが、メモリアクセススピードの高速化のペースは年率 7%程度といわれている。その結果、CPU とメモリアクセスの性能差は年々拡大していっている。

そこでメモリアクセスコストを削減するために CPU は通常ハードウェアキャッシュと

呼ばれるハードウェアを持つ。通常メモリアクセスは遅いのでキャッシュにデータを載せることを CPU は試みる。データにアクセスするときキャッシュにそれが存在している場合、キャッシュヒットと呼び、L1 キャッシュ（一次キャッシュ）ヒットの場合そのコストは 2 クロック程度である。一方でデータがキャッシュにない場合、キャッシュミスと呼ぶがメインメモリまでアクセスすると 200～300 クロック程度かかり、キャッシュアクセスに比べ大変コストがかかる。そのためキャッシュミスを減らすことが重要になる。

表 4.3-1 アクセスコスト

デバイス	アクセスコスト
L1(一次)キャッシュ	2 クロック
L2(二次)キャッシュ	10 クロック
L3(三次)キャッシュ	50 クロック程度
メインメモリ	200～300 クロック前後

メモリアクセスコストは下記の式で求められる。

メモリアクセスコスト＝L1 ヒット＋キャッシュミス率＊メモリアクセス

例えばキャッシュミス率が 10%(キャッシュヒット率は 100%-10%=90%)で、L1 ヒットが 2 クロック、メモリアクセスが 200 クロックとして、簡単のために L2/L3 はないものと考えて、メモリアクセスコストを求めると

メモリアクセスコスト＝2+0.1*200=2+20=22 となる。

キャッシュミス率を 1%に削減すると

メモリアクセスコスト＝2+0.01*200=2+2=4

となる。キャッシュミス率を削減するとアクセスコストが 22 クロックから 4 クロックへと約 5 倍の性能向上をする。

4.3.2 Cache pollution aware patch

Cache pollution aware patch により cache pollution によって発生していたキャッシュミスを削減することに成功した。今回の評価により、従来は計測する方法があきらかでなかった、cache pollution によるキャッシュミス回数なども測定できるようになった。

4.3.3 Cache pollution の発見方法

Cache pollution を発生させる時間的非局所性(Non-temporal)なコピーは OProfile を利用し L3 キャッシュミスの回数を測定し当該部分のソースコードを分析すれば容易に発見できることを示した。cache pollution を発生している関数を呼んでいる Linux カーネルの場所も OProfile のコールグラフ機能を利用し特定した。また cache pollution を防止するパッチを作成しその効果も定量的に評価し第三者に確認可能な形で示せた。

OProfile を利用すれば容易にキャッシュミスの場所を特定でき、キャッシュミスを多発しているところのソースコードを分析すれば、容易に cache pollution を発見できる。これは従来よく知られていなかったことであり、今回の調査の成果の一部である。

4.3.4 メモリプロファイリング

OProfile での従来のプロファイリングは主にタイマイベントのサンプリングであった、プログラムのどこで最も時間がかかっているかという点に着目してプロファイリングし、そのデータを元に性能向上をはかるというものである。

サンプリングデータは「どこ」という情報を提供するが「なぜ」という情報は提供しない。そのため、ソースコードを分析し、場合によっては LKST などを利用してより詳細な分析をこころみた。

今回提案した手法はタイマイベントによるサンプリングではなく、メモリのアクセスに注目したサンプリングである。L3（ないし L2）キャッシュミスに注目し、キャッシュミスが多発しているところをチューニングするというアプローチである。このようなメモリアクセスに注目したプロファイリングは、今後ますます重要になってくると考えられる。

CPU の性能向上（年率 50%前後）に比べメモリアクセスの性能向上のスピード（年率 7%程度）は遅く、その性能比は年々広がっているといわれている。メモリアクセスのペナルティは現時点でも 200～300 クロックあるが、今後そのペナルティが増加する傾向にあり、ソフトウェアの観点からキャッシュを意識したプログラミングがますます重要になってくる。

しかしながらキャッシュを意識したプログラミングというのは非常に難しく従来のプログラミングテクニックではなかなか解決できなかった。そのために、メモリプロファイリングツールなどが必要になってくる。この手のツールがない限り、効率的にプログラムのチューニングをすることはきわめて難しいといっても過言ではない。

今回の調査ではメモリプロファイリングの重要性を指摘し、その効果を定量的に実証した意義は大きい。

従来はそれほど注目されていなかった技術ではあるが、OProfile 等のツールを利用すれば容易にキャッシュ上の問題点を発見でき、cache pollution に関しては IA32 の Non-temporal move 命令を利用することで性能向上をはかれることを示した。

4.4 評価手法のまとめ

CPU ビジー型の場合のチューニング技法として

- アルゴリズムに注目した性能向上
- キャッシュに注目した性能向上（メモリプロファイリング）

のうち、キャッシュに注目した性能向上手法を今回提案した。

1. OProfile で L2/L3 キャッシュミスのトップ 5 を求める
2. キャッシュミスが多発している部分のソースコードを検討し、分析する。
   もし、cache pollution が発生していたら、キャッシュをバイパスするようなコードを開発し、評価してみる。

4.5 今後の課題

4.5.1 CPU スケーラビリティ

今回は cache pollution に注目して性能向上をはかったが、cache pollution の防止と CPU スケーラビリティについての相関についての検証は行っていない。

cache pollution を防止することによって、メモリバストラフィックは減少する。メモリバストラフィックが減少することによって SMP 環境においてはメモリアクセスのスループットが増加することが期待できるが今回は検証できていない。

cache pollution が減ったので全体としてキャッシュミスも減少している。それが SMP 環境下においては CPU スケーラビリティを向上させるという検証は行っていないので今後の課題としたい。

4.5.2 Iozone 以外のベンチマークの実施

今回は Iozone によってカーネルのボトルネックを発見したが、Iozone は何か特定のアプリケーションを想定しているわけではないので、現実的な問題への適用性という意味での説得力が弱い。より幅広いベンチマークを実施し、今回の手法の有効性および限界を検証しなければならない。

4.5.3 メモリプロファイリングによるカーネルボトルネックの計測および性能改善

今回開発したメモリプロファイリング手法の有効性および限界について、カーネルボトルネックを計測し、性能改善をはかることによって実証していくべきだと考える。

4.5.4 OProfile の制限事項

今回、OProfile を駆使し様々なハードウェアイベントの情報を入手できた。そして OProfile が非常に強力な分析ツールであることを証明した。OProfile は Linux カーネルに組み込まれているため、特別の準備なく簡単に利用できた。

しかし今回使用した範囲では、以下に挙げる制限事項があった。ここではその制限について議論する。

4.5.4.1 L1 キャッシュミスの測定

OProfile のイベントでは L1 キャッシュミスを測定できない。今回は L3 キャッシュミスから cache pollution を発見したが、より細かいキャッシュ上の問題を発見するためには L1 キャッシュの実行についても詳細に把握する必要がある。

4.5.4.2 PEBS (Precise Event Based Sampling)の機能

今回メモリアクセスに注目したイベントサンプリングを行い、キャッシュミスを発生させているプログラムの場所は特定できた。しかしながら、キャッシュミスを発生させてい

る場所で、メモリのどこにアクセスしているかの情報は OProfile では入手できない。今回下記の命令でキャッシュミスを多発しているということが分かったが、ESI レジスタおよび EDI レジスタの値を記録していないので、メモリ中どこにアクセスして L3 キャッシュミスを起こしたのか分析できない。

```
repz movsl %ds:(%esi),%es:(%edi)
```

図 4.5-1L3 キャッシュミスを多発したコード

Pentium 4/Intel Xeon プロセッサでは上記の問題に対し、PEBS (Precise Event Based Sampling) と呼ばれる機能によって、以前のプロセッサでは不可能だった、より精密なプロセッサの状態のサンプリングが可能になった。具体的には、あるイベントが発生したときの EIP（プログラムカウンタ）レジスタの値をサンプリングするだけではなく、汎用レジスタ、EFLAGS レジスタを自動的にサンプリングする。この機能を利用すれば、キャッシュミスが発生したとき、どのメモリにアクセスしていたのでキャッシュミスが発生したなどという情報が入手できるようになり、より詳細な分析ができるようになる。

しかしながら現在の OProfile では PEBS の機能は利用できないため、この機能拡張が望まれる。

以上の点が改善されれば、OProfile はさらに強力な分析ツールとなり、従来解析不可能だった cache conflict の発見などが容易に行えるようになる。

4.5.5 Iozone+OProfile で時折発生したカーネルパニックの原因究明

2.6.13 カーネルで発生したカーネルパニックに関しては原因究明までいたらなかった。クラッシュダンプの取得などにより原因究明する必要がある。

4.6 総括

2004 年度に開発したカーネルの評価手順を利用し、ユーザコマンドレベルのツールを用いてカーネルボトルネックの発見をおこなった。2004 年度にみられたロック競合型の問題を今年度は発見できなかった。2.6 カーネルになって、グローバルロックの問題が改良されたためだと思われる。

CPU ビジー型の問題についてはボトルネックの発見だけではなく、それを一歩進めてカーネルのボトルネックの解消を試みた。CPU ビジー型について詳細な分析を行った。2004 年度では CPU ビジー型の性能向上まではいたらなかったが今年度は cache pollution を防止するパッチを開発し、性能向上に成功した。

今回の測定において発見されて未解決となった現象もいくつかあり、今後追及すべき課題が以下の通りに挙げられる。

- cache pollution の減少と CPU スケーラビリティの相関。
- Iozone 以外のベンチマークによる計測およびカーネルボトルネックの発見。

- メモリプロファイリングによるカーネルボトルネックの計測および性能改善。
- 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。
- Iozone+OProfile で時折発生したカーネルパニックの原因究明

以上

5 付録

5.1 LKML での議論について

Linux に対するパッチは LKML(Linux Kernel Mailing List)での議論によってコンセンサスが得られないと Linus が保守管理するメインストリームに取り込まれない。そこでここでは今回のパッチ作成を一つのケーススタディとしてどのような議論をどのようにして行ったかを明らかにすることによって、これからパッチをサブミットすることを考えている人たちへの参考にすることを試みる。一つのケースで過度な一般化を行うことは危険ではあるがあくまで参考程度に受け取って頂ければ幸いである。

8/8: 開発基盤 WG、OS 層のミーティングで初めて cache pollution aware patch のアイデアを話す。この時点では MOVNTI 命令を使うというアイデアだけで実装はなかった。

8/9:カーネル読書会（YLUG--横浜 Linux Users Group の有志が主催しているインフォーマルな会合）で cache pollution aware patch のアイデアを話す。MMX レジスタを利用すると page fault 時など例外発生時にレジスタの退避、回復をやっていないので問題が発生するという指摘をうける。

8/10:_mmx_memcpy()という MMX レジスタを利用して memcpy()する関数を参考に_mmx_memcpy_nt()という関数を作成し評価した。OProfile を利用してクロック数、L2/L3 キャッシュミス数などを測定した。

8/14:ある程度実装にめどが立ったので[RFC] [PATCH] cache pollution aware __copy_from_user_ll()というタイトルでLKMLに初めて投稿した。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112401103800730
すると、すぐに Arjan van de Ven より、cache を利用しないことでの副作用があるのではというコメントを頂く。
それに対し、OProfileのデータを示し反論。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112401498215115&w=2
Arjan はキャッシュを利用しないことの利点があることは分かるが多くの場合キャッシュは有用なのでキャッシュを利用しない別関数を作ることを提案される。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112401594215553&w=2
Christoph Hellwing も 2 つの関数に分けることを支持。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112401638906857&w=2
Ian Kumlien は確証もてないがわたしのアイデアが面白いとコメント。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112405473028938&w=2
ベンチマークを iozone ではなく、カーネルビルドに変えて結果をポスト。

http://marc.theaimsgroup.com/?l=linux-kernel&m=112408829819450&w=2

8/15: Arjan から反論。データをすぐに利用するケースではキャッシュをバイパスするのはコストが高い。データをすぐに利用しないケースではわたしの copy 関数はよい。そこで、2 つのケースに対応するように API を 2 つにわけたらばどうか。キャッシュを使わない特殊化した API として copy_from_user_nocache()を作り、データをすぐに再利用しない所でそれを使う。通常の場合は昔の API を使う。多くの場合は通常の API を使うのでわたしの関数を利用するのはカーネルのごく一部であろう。そこを変更するだけで性能は向上するはずだ。

> My suggestion is to realize there are basically 2 different use cases, and that in the code the first one is very common, while in your profiles the second one is very common. Based on that I suggest to make a special copy_from_user_nocache() API for the cases where the kernel will not use the data (and ignore software raid5 here) and use your excellent version for that API, while leaving the code for the cases where the kernel WILL use the data alone. Code wise the "will use" case is the vast majority, so only changing the few places that know they don't use the data will be very efficient, and will give immediate big improvement in your profile data, since those few places tend to get used a lot in the cases you benchmark.

http://marc.theaimsgroup.com/?l=linux-kernel&m=112409029932039&w=2
Arjan から Ian へのコメント。ケースによっては非常に効果があるだろうと認めている。

> It is. It's good proof that you can make a big gain already by converting a few key places to his excellent code. And neither me nor hristoph are suggesting to ditch his effort! Instead we suggest that what he is doing is useful for some cases and harmful for others, and that it is quite easy to identify those cases and separate them from eachother, and that thus as a result it is more optimal to have 2 apis, one for each of the cases.
>
> (his excellent code というコメントがちょっと嬉しい)

http://marc.theaimsgroup.com/?l=linux-kernel&m=112409063832647&w=2
Arjan への返信。2 つ API を作ることは合意。問題はどこでそれを使うべきかわたしがよく知らないことだ。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112409558522631&w=2
linux@horizon.comから、write()以外で効果があるかどうか質問がでる。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112410823032265&w=2
zerocopy, sendfile, sendmsg あたりに最適化の余地があるか。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112411883030150&w=2

8/16: __copy_from_user_ll_nocache()と__copy_from_user_inatomic_nocache()を公開した。そして filemap_copy_from_user()から__copy_from_user_inatomic_nocache()を呼ぶように変更した。_nocache を取ったのが昔からある API 名である。L3 キャッシュミスが激減しているデータを添付した。

http://marc.theaimsgroup.com/?l=linux-kernel&m=112416341010746&w=2
高橋さんより、page fault 時の処理はどうしているかコメントを受ける。例外処理時に FPU レジスタが汚染されるという指摘。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112416631515090&w=2
いいアイデアがないのでそのまま質問する。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112416841224244&w=2
Arjan から感動したというコメント。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112417146313181&w=2
Andi Kleen より preemption を禁止しなくてはいけなくて、それは low latency な人がうれしくないだろう。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112419826104913&w=2
MMX の退避回復をきれいにやる方法が求められているが自分の理解が全然足りないので理解を深める意味で現時点での自分の理解をまとめる。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112425460127067&w=2
塚本さんがコメントをくれる。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112429211420667&w=2

8/18:MMX の状態をスタックへ退避回復する low latency 版を公開がリグレッションしていたのですぐに取り下げる。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112440786129276&w=2

8/22: http://marc.theaimsgroup.com/?l=linux-kernel&m=112475380008182&w=2

8/24: MMXレジスタではなく一般レジスタを利用した版を公開する。クロック数で 12%、L3 キャッシュミス 70%削減した。__copy_from_user_ll()のキャッシュミスはほぼ 0 になった。http://marc.theaimsgroup.com/?l=linux-kernel&m=112489315002172&w=2
Andi が CPU によっては movnti をサポートしていないものがあると指摘。
高橋さんが、sfance/mfance を利用する必要があると指摘。

9/1: ほぼ最終版を公開。1）sfance 命令を利用し store 命令のシリアライズをおこなった。2） CPU が XMM2 拡張を持つかチェックした。(movnti 命令が使えるかどうかの確認)
この版はメインラインへのマージを検討するのに値すると初めて言及した。また、Linus と Andrew にも CC をいれた。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112556596103063&w=2
Andi からは合意を得る。同様なことを x86_64 でやるかもしれない。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112556757925932&w=2

9/2: 次にやることはなにか？
http://marc.theaimsgroup.com/?l=linux-kernel&m=112562585113515&w=2
誰かが反対しない限り Andrew が面倒を見てくれるだろう。

http://marc.theaimsgroup.com/?l=linux-kernel&m=112562695120000&w=2
Andrew からの初めてのコメントは、-mm tree の待ち行列にいれるけど、効果には疑問があるというもの。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112562714013328&w=2
Andi からすぐに反論。ロスよりゲインの方が大きいだろう。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112562752526958&w=2
Andrew から OProfile のデータではなく、実測時間のデータを出してほしいと要請される。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112562836720370&w=2
iozone の結果を添付する。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112563297405092&w=2
Andrew: 2.6.12.4 ではなくて最新版に対するパッチを送れ。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112563553123814&w=2

9/3: 2.6.13 用のパッチを公開した。
http://marc.theaimsgroup.com/?l=linux-kernel&m=112574915427327&w=2

9/4: Andrew よりからの-mm にマージしたとメールを受ける。(私信)

思いついてから約一月でここまでたどり着いた。LKML では 60 通を超えるやり取りがあった。また YLUG(横浜 Linux Users Group)のカーネル読書会のメンバーからも多くのやり取りがあった。ここに記して感謝したい。
コミュニティからの指摘に一つ一つ答えていくことによってコードがどんどんシンプルになり一般性を持っていくプロセスを体感した。ソフトウェアバザールモデルの強さを実感した瞬間であった。ネットワークの向こう側にいる Linux Kernel Hackers による peer review が機能しているということを感じた。メールに対するすばやい反応、特に一日に何通もやりとりするというのは片手間ではできない。コードをすばやく修正して頻繁にリリース。それを繰り返すことがバザールモデルの本質だということを実感した。
今回のやり取りの中で思ったことは、LKML でのコメントを一つ一つ丁寧に回答していくことによって相手との信頼性が生まれるということである。問題点の指摘には真摯にコードの修正によって答えていった。MMX レジスタの退避回復の問題は最後まで解決できなかったが、逆転の発想で、最終的には MMX レジスタを利用しないことによって回避した。そのために preemption を禁止することもなく、従って low latency の人たちにとって迷惑をかけることもなくなった。性能的な観点から MMX レジスタを使わなければいけないという思い込みをなぜか持っていたのだが、一般レジスタでも十分効果があることを確認できた。
あえてまとめるならば、コードを完璧にするために何週間もかかるなら、不完全でも一つの問題を修正したパッチを頻繁にリリースして、多くの人にレビューしてもらってさらに改良を重ねるスタイルのほうが最終的にはよりよいものが出来上がる可能性が高く、そのオープンなプロセスによってコミュニティにも評価され、受け入れられるということかと思う。