

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
～LKST 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	はじめに	1-1
1.1	開発と評価の背景	1-1
1.2	開発および評価の概要	1-1
1.2.1	EM64T対応版LKST (LKSTLogTools) の開発	1-1
1.2.2	LKSTLogToolsの機能拡張	1-2
1.2.3	LKSTを用いた性能評価	1-2
1.2.4	LKST性能評価機能のオーバヘッド評価	1-3
1.3	本ツールの想定利用者について	1-3
2	開発した機能	2-1
2.1	EM64T対応版LKST (LKSTLogTools) の開発	2-1
2.1.1	LKST (LKSTLogTools) の概要	2-1
2.1.2	EM64Tへの移植作業	2-2
2.1.3	EM64Tでの有効性の確認	2-2
2.2	LKSTLogToolsの機能拡張	2-7
2.2.1	拡張した機能の概要	2-7
2.2.2	評価環境	2-8
2.2.3	時系列解析結果にプロセス名を併記する機能	2-9
2.2.4	ランキューにつながれてから実行するまでの時間を取得する機能	2-11
2.2.5	プロセス名によるフィルタリング機能	2-13
2.2.6	複数のキーワードで情報を収集し、1つのグラフで表示する機能	2-14
2.3	最新版LKSTのMIRACLE LINUX V3.0 への適用	2-16
2.3.1	移植における課題と対策	2-16
2.3.2	移植の成果	2-17
3	LKSTを用いた性能評価に関する考察 – DB実行時のカーネル評価 –	3-1
3.1	評価目的	3-1
3.2	評価概要	3-1
3.2.1	システム環境	3-2
3.2.2	DBT-1 の概要	3-3
3.2.3	LKSTによるログの採取手法	3-7
3.3	32ビット/64ビット環境での実行状況評価	3-14
3.3.1	測定目的	3-14
3.3.2	測定内容	3-14
3.3.3	測定結果	3-14
3.3.4	評価と考察	3-16
3.4	DBT-1 のBuyConfirmインタラクションの実行状況評価	3-17
3.4.1	DBT-1 のランプアップ時の状況	3-17
3.4.2	DBT-1 平常時の状況	3-24
4	LKST 性能評価機能のオーバヘッド評価	4-1

4.1	評価環境	4-1
4.1.1	システム構成	4-1
4.1.2	ベンチマーク	4-1
4.1.3	評価パターン	4-1
4.1.4	LKSTのセットアップ	4-2
4.2	評価結果	4-3
4.2.1	dbench	4-3
4.2.2	LMbench	4-4
4.2.3	tiobench	4-7
4.3	考察	4-10
5	総括	5-1
5.1	LKSTの有効性	5-1
5.2	性能評価ツールの開発規模	5-2
5.3	今後の課題	5-2
5.3.1	LKST本体の課題	5-2
5.3.2	性能評価ツールの課題	5-3
6	付録	6-1
6.1	LKST適用OSの構築とインストール方法	6-1
6.1.1	LKST適用カーネルをソースコードから構築しインストール	6-1
6.1.2	LKSTを適用したOSの再起動	6-4
6.1.3	LKSTコマンドと解析ツール	6-5
6.1.4	lkstモジュールのロードと実行	6-6
6.2	LKSTLogToolsで採取したデータ	6-6
6.2.1	システムコールの開始から終了までの時間（抜粋）	6-6
6.3	ベンチマーク測定データ	6-12
6.3.1	dbench	6-12
6.3.2	LMbench	6-12
6.3.3	tiobench	6-14

1 はじめに

1.1 開発と評価の背景

サーバ分野において、Linux を適用したシステムが普及・拡大している。最近では、アプリケーションサーバ、DB サーバから構成されるエンタープライズシステムへ適用される事例も出てきている。

エンタープライズシステムでは障害や性能遅延が発生した場合、迅速な対応を求められるが、Linuxにはダンプやトレースといった解析のための標準的なツールはまだ組み込まれていない状況にある。これに対応する手段として、LKST (LKSTLogToolsを含む、2.1.1項参照) の開発を推進している。しかし、ハードウェア/ソフトウェアのすべての種類や技術に対応できていないのが現状である。

例えば、サーバのアプリケーションが大規模なデータを扱うようになるにつれ、大容量メモリを搭載したサーバを望む声が多くなっている。現在ではメモリの単価が安くなったこともあり、大容量メモリを搭載したサーバも登場し、Linux も 64 ビット化をサポートしてこのアーキテクチャに対応している。64 ビットアーキテクチャにはいくつか種類があるが、特に EM64T は、大容量メモリを扱える安価な 64 ビットサーバであること、既存の 32 ビットプログラムも実行可能であること、などから、一般的なニーズが高い。しかし、64 ビット環境は、32 ビットとは OS のチューニングの仕方も障害の発生状況も異なるため、障害発生時に 32 ビット環境下での経験が活かせないことが多い。この対応のために障害解析用にトレース機能が必須であるが、64 ビット版にもまだ、標準的なツールは組み込まれていない。

また、アプリケーションが多様化しているのに対応して、カーネルでの処理は複雑化しており、障害解析を効率よく行うためには、色々な絞込みを行う必要がある。しかし、現在の LKST はこの絞込み機能が不足しており、手作業で補っている面が多々ある。また、解析ノウハウが不足している障害では、解析の試行錯誤の時間が多くなる。この解析作業の効率を向上させるためには、ノウハウを貯め、機能を充実させていく必要がある。

1.2 開発および評価の概要

今回、前述の課題に対し、具体的に以下の開発と評価を行った。

1.2.1 EM64T対応版LKST (LKSTLogTools) の開発

IA-32 から EM64T への移植は以下のポイントに着目し、修正が必要となる箇所を検出し、修正を行った。

- (1) アドレス値の扱い
- (2) LKST が記録するイベントの引数の扱い

移植後に、2004 年度に実施したIA-32 版評価と同じ項目を実施したところ、IA-32 版と同様にデータの採取・可視化が行えることを確認できた。さらに、IA-32 と同じ手法でカーネルの性能解析ができることを確認できた。詳細は2.1節参照のこと。

1.2.2 LKSTLogTools の機能拡張

従来の LKSTLogTools では、収集した性能情報を 1 つのキーワードで選択、もしくは全部を可視化という形のものであったが、実際に使っていると、何らかの情報の組み合わせで解析を行うことの有効性が高いことがわかった。そこで今回、解析で必要となった以下の機能を開発した。

- (1) 時系列解析結果にプロセス名を併記する機能
- (2) ランキューにつながれてから実行するまでの時間を収集する機能
- (3) プロセス名によるフィルタリング機能
- (4) 複数のキーワードで情報を収集し、1 つのグラフで表示する機能

これらの機能を実現したことで、カーネル処理をプロセスで絞り込む、特定キーワードのデータだけを選択・可視化して比較する、などが容易に実現できるようになった。このため、データを得るための手作業を削減することができるようになった。詳細は2.2節を参照のこと。

1.2.3 LKST を用いた性能評価

以下に示す、DB サーバ稼動時のカーネル性能評価を行なった。

(1) 32 ビット／64 ビット環境での実行状況評価

今回の開発で LKST の EM64T 対応を実現し、32 ビット／64 ビットの両方のモードでの実行が可能になった。そこで、MaxDB 上で DBT-1 を稼動させ、32 ビット／64 ビットの双方でシステムコール性能を測定した。

この結果、BT値(DBT-1 が出力する性能指標)だけでは知ることができない、システムコール毎における性能の変動を知ることができた。このような評価は、プログラムとアーキテクチャの相性などを判断する時などに有効に活用できると考える。詳細は3.3 節を参照のこと。

(2) DBT-1 の BuyConfirm インタラクションの実行状況評価

2004 年度に実施した DB 層の評価結果により、DBT-1 の「BuyConfirm」インタラクションの性能が、他のインタラクションと異なっていることがわかっている。そこで、LKST を利用して以下の 2 点での解析を行った。

- (a) DBT-1 ランプアップ時
- (b) DBT-1 平常時

この結果、(a)のランプアップ時には、DBのデータボリュームの先頭約 1GBをメモリ上に読み込んでおり、何らかの理由でBuyConfirmはこの処理の完了を待っていると予測できるまでの情報を得ることができた。また(b)では、BuyConfirm処理時にセマフォによる長時間の待ちが多発し、全体的に処理が滞ると予測できるまでの情報を得ることができた。これらの情報は、DB内の処理を分析する際に有効活用できると考える。詳細は3.4節を参照のこと。

1.2.4 LKST 性能評価機能のオーバーヘッド評価

今回いくつかの新しい機能を拡張開発したので、2004 年度に実施したオーバーヘッド評価の項目のうち dbench、Lmbench、tiobench の 3 つのベンチマークについて評価を行った。

この結果、機能拡張分だけ多少のオーバーヘッドの増加が見受けられた。詳細については4章を参照のこと。

1.3 本ツールの想定利用者について

本ツールの使用に当たっては、OS についての基本的な知識を持っていることを前提としている。本ツールの利用者としては、カーネル開発者、システムエンジニア、ミドルウェア開発者などを想定している。

カーネル開発者：

本ツールを使うことで、現行カーネルのボトルネック解析、新しいアルゴリズムや実装のミクロな評価などを行うことが出来る。

システムエンジニア：

本ツールを使うことで、システムのスローダウンの原因が OS にある場合にその根源を探るために利用することが出来る。

ミドルウェア開発者：

本ツールを使うことで、アプリケーションを Linux 上で動かした場合のシステムコールから先の振る舞いを検証することが出来る。

2 開発した機能

2.1 EM64T 対応版 LKST (LKSTLogTools)の開発

2.1.1 LKST (LKSTLogTools) の概要

今回拡張開発を行う LKST は2つの機能を持つ。

LKSTのソースコードと使用説明書はすべて、SourceForge (<http://lkst.sourceforge.net>, <http://lkst.sourceforge.jp>) にて公開しているので、詳細はそちらを参照して欲しい。

2.1.1.1 トレーサ機能

LKST はクラッシュするような障害を解析するために、カーネル内の処理の遷移情報を得ることを目的に開発を進めていた(*)。そのため、カーネル内部で発生したイベントの情報を発生順に記録し保持する機能を有している。記録したイベント情報に関しては、解析者が自由に加工しやすいように、テキスト形式に変換する機能を提供している。LKST はカーネル内のイベント情報を記録するための、柔軟で拡張性のある、以下の特徴を持つ。

- (1) 取得するイベントの種類を容易に追加可能な構造
- (2) 取得するイベントを動的に選択する機能
- (3) イベント取得時に、記録する情報を加工する機能
- (4) 情報を記録するバッファのサイズを動的に変更する機能
- (5) 例外発生時でもイベントの記録が可能な構造

*)日立、IBM、富士通、NEC の4社協業「Enterprise Linux の信頼性の強化の推進」の一環として開発プロジェクトを開始。

2.1.1.2 性能評価機能

Linux カーネルの性能を評価するためには、Linux カーネル内の制御の要となる部分の性能情報が必要である。LKST ではこの性能評価に必要な情報の収集機能・解析ツール (lkstla) を提供している。この機能とツールの総称が LKSTLogTools である(**)。最新版 LKST は、LKSTLogTools 機能をマージして提供している。この機能を利用することにより、障害要因となる性能の問題点を検出する、もしくは原因を究明する、などの重要なヒントを得ることができる。

**この一部は「2004年度 オープンソースソフトウェア活用基盤整備事業」で開発。

2.1.2 EM64T への移植作業

LKST の IA-32 版から EM64T 版への移植は以下のポイントに着目し、修正が必要となる箇所を調査した。

(1) アドレス値の扱い

IA-32 と EM64T の大きな違いは、指定可能な仮想アドレスの範囲である。IA-32 では 32 ビットで表現可能な 0x00000000~0xffffffff に対し、EM64T では 64 ビットで表現可能な 0x0000000000000000~0xffffffffffffffff となる。また C 言語上の違いもある。Linux 標準のコンパイラ gcc では、32 ビットアーキテクチャの場合 int 型、long 型は共に 32 ビット長であるが、64 ビットアーキテクチャでは int 型は 32 ビット長、long 型は 64 ビット長となる。したがってアドレス値を変数に格納する場合、両アーキテクチャで共に使用できる型は long 型以上のサイズの整数型でなければならない。

(2) LKST が記録するイベントの引数の扱い

LKST におけるカーネル内イベントの記録では、1 イベントあたり最大 4 つの引数を記録でき、1 つの引数を記録する領域は 64 ビット長である。これは IA-32 と EM64T 共通の仕様となっている。よって 1 つのイベントで 5 つ以上の引数を記録したい場合、64 ビットの領域を 2 つの領域に分け、32 ビットの値を 2 つ記録する方法を用いてきた。このとき、記録する値にアドレス値や long 型の値が含まれていると問題となる。すなわち 32 ビットアーキテクチャでは 32 ビット長だった値が、64 ビットアーキテクチャでは 64 ビット長となり、上記方法で 5 つ以上の引数を記録することはできない。

上記の課題に対し、次の 3 点について修正を行った。

- (a) アドレス値や long 型の値を int 型で扱っている箇所があれば、long 型に修正する
- (b) lkstla コマンドによる解析結果にアドレス値を含む場合、表示桁数を調整する
- (c) 1 つのイベントにおいて 5 つ以上の引数を記録している場合、2 つのイベントとして分割して記録するよう変更する

2.1.3 EM64T での有効性の確認

移植後の動作確認も兼ねて、EM64T で LKST を実行し、以下の 2 点を確認した。

- ・ IA-32 と同じ手順を実施し、同じように採取できること
- ・ 上記のうちの 1 つについて、IA-32 と同じ、同じ手法で解析できること

2.1.3.1 評価環境

- (1) 評価を行ったシステム環境を表 2.1-1 に示す。

表 2.1-1 評価を行うシステム環境

項番	項目	評価環境
1	ハードウェア	CPU
2		メモリ
3		ハードディスク
4	ソフトウェア	カーネル
5		OS
6		解析ツール
7		負荷ツール
8		ファイルシステム

(2) LKST 実行条件

評価を行う際の、LKST の実行条件は以下の通り。

- (a) 採取するイベントは LKST 性能評価用のイベントのみとする
- (b) 使用するハンドラはデフォルトハンドラおよび lksteh_procstat

なお、LKSTを適用したカーネルの再構築方法は、6.1節を参照のこと。

(3) イベント情報の採取

LKST によるカーネル内のイベント情報の取得は lkstlogtools の再構築実行中に行う。以下の処理を実施する。

- (a) lkst モジュールおよびイベントハンドラをカーネルに組み込む
- (b) 解析用のマスクセットを作成し、有効にする
- (c) バッファを作成する
- (d) キャッシュを空にする
- (e) (c)で作成したバッファを使用可能にして、一度内容をクリアする
- (f) 負荷を生成する
- (g) イベントの記録を停止する
- (h) バッファの内容をファイルへ保存する
- (i) ファイルを CPU ごとのファイルに分割する (時系列保証のため)

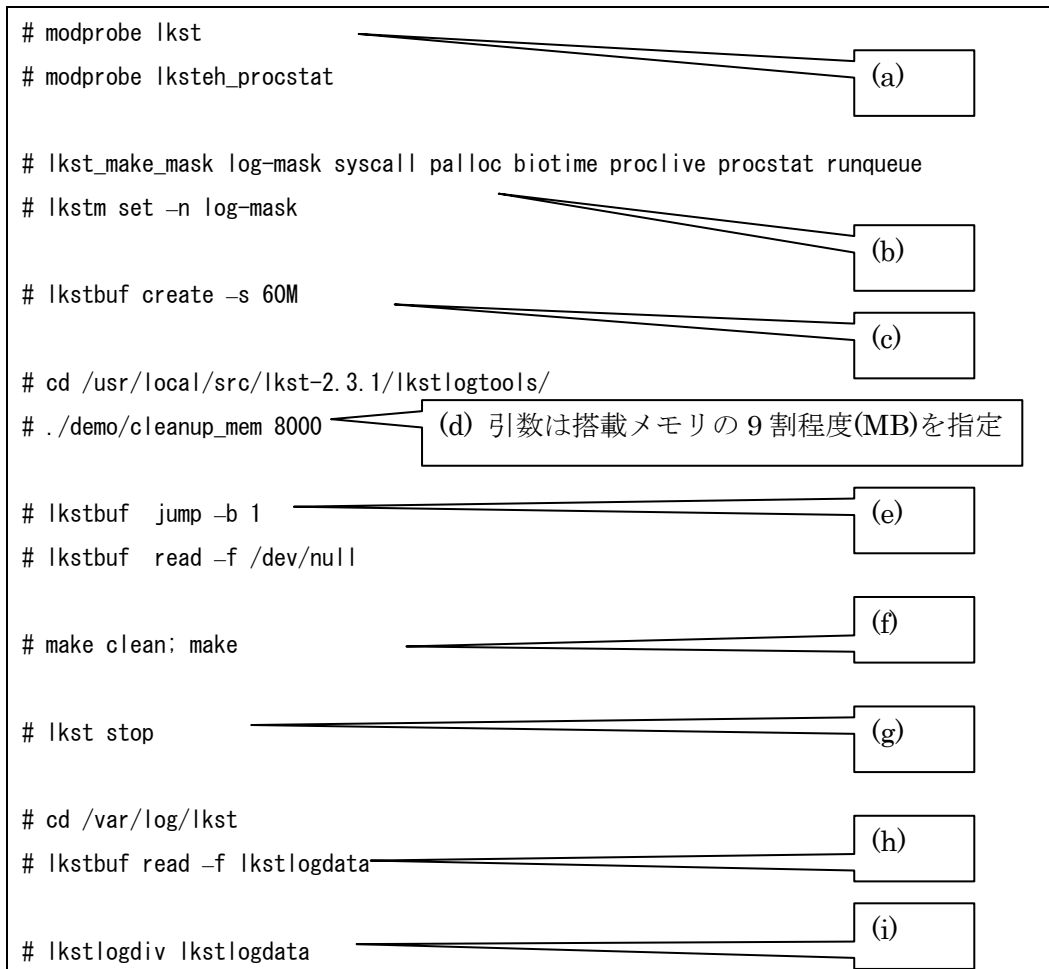


図 2.1-1 データ採取の手順

2.1.3.2 データ採取の確認

(1) 評価対象

IA-32 版で検証した表 2.1-2の項目について、以下の3種類のデータを採取し、IA-32 版と同じように採取できるかを調査した。

- (a) 時系列データ：使用オプションは-l
- (b) 統計データ：使用オプションは-s
- (c) 分布データ：使用オプションは-d

表 2.1-2 評価する LKSTLogTools の機能

項番	評価項目	使用するアナライザ
1	システムコールの開始から終了までの時間の計測	syscall
2	メモリ確保に要する時間、実行回数の計測	palloc
3	プロセスの状態の変化と各状態での時間の計測	procstat
4	I/O 要求から終了までの時間の計測	biotime
5	ユーザプロセスの生成から終了までの時間の計測	proclive
6	プロセスのランキュー情報を取得	runqueue

(2) 測定結果

表 2.1-2の全データについて、IA-32 版と同じ品質のデータが取得できていることを確認した。採取したデータに関しては、項番 1 のみ6.2節に添付している。

2.1.3.3 システムコールの呼び出しの解析の確認

まず、lkstlogtools 再構築中に採取したログから、各システムコールの呼び出し状況の統計情報を次のコマンドで取得した。

```
# lkstla syscall -s lkstlogdata-*.stat > syscall.stat
```

取得した統計情報を、次のコマンドによりグラフに変換したものを図 2.1-2に示す。

```
# lkst_plot_stat -a syscall.stat
```

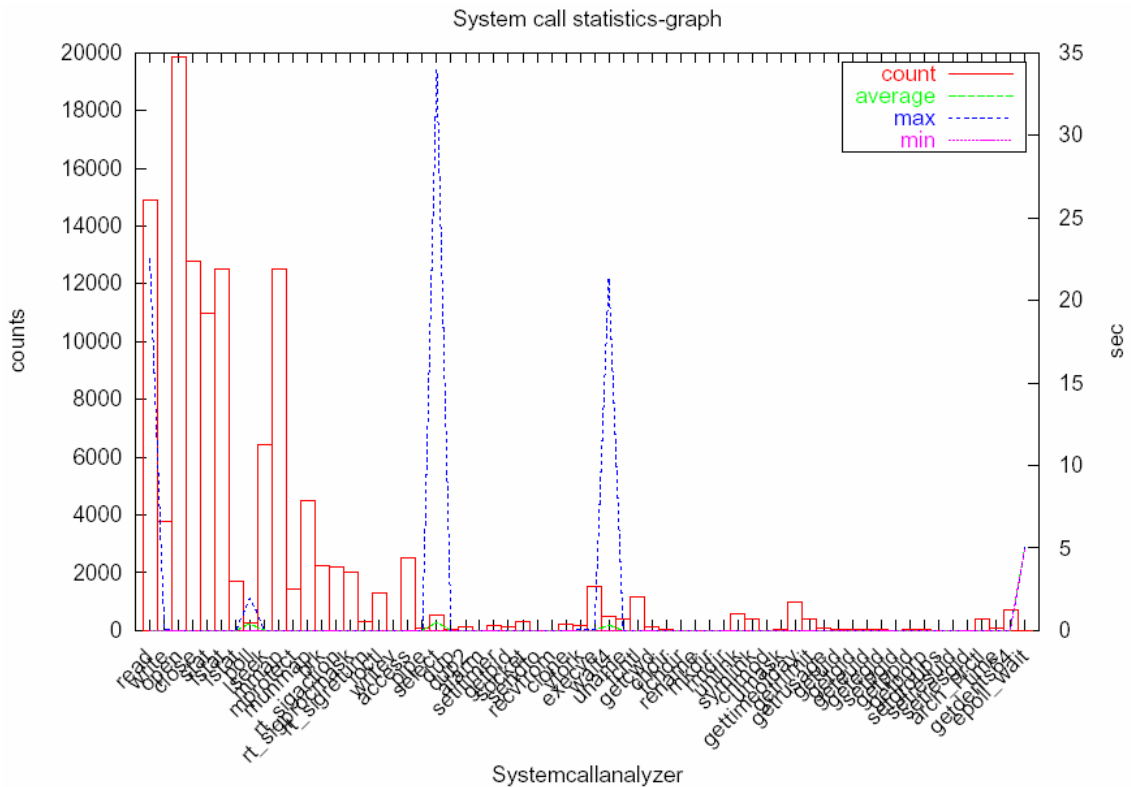


図 2.1-2 システムコールの呼び出し状況の統計グラフ

図 2.1-2は、横軸がシステムコールの名称、棒グラフで表される縦軸(左側)がそのシステムコールが記録された回数、折れ線で表される縦軸(右側)がそれぞれのシステムコールの平均実行時間、最大実行時間、最小実行時間を表す。

この統計情報を、プロセスの実行時間に対する占有時間(total値)の多い順にソートし、上位 10 を抜粋したものを表 2.1-3に示す。この表から、どのシステムコールがプロセスの実行時間に大きな影響を与えているか容易に分かる。

表 2.1-3 lkstlogtools 構築時にプロセスの実行時間の多くを占める上位 10 システムコール

sysno	syscall_name	count	average	max	min	total	percent
23	select	539	0.501837166	33.989652492	0.000002972	270.490232718	44.459
61	wait4	465	0.328774243	21.377246264	0.000000457	152.880023046	25.128
7	poll	240	0.415976533	1.984573387	0.000000995	99.834367975	16.409
232	epoll_wait	94	4.998941763	4.999608297	4.998600911	44.990475867	7.395
0	read	14912	0.002427569	22.520000078	0.000000250	36.199902752	5.950
2	open	19839	0.000065267	0.017791240	0.000001065	1.294827931	0.213
4	stat	10987	0.000113425	0.020459669	0.000000805	1.246196885	0.205
59	execve	1513	0.000279161	0.046829803	0.000001300	0.422371224	0.069

217	getdents64	693	0.000545757	0.013170680	0.000000410	0.378209748	0.062
58	vfork	160	0.001095924	0.046872893	0.000066178	0.175347878	0.029

このように、EM64T でも IA-32 版と同様に、LKST 性能評価機能を利用してカーネルのシステムコール実行時間を解析する情報を取得できた。

参考までに 2004 年度「OSS性能・信頼性評価／障害解析ツール開発」で報告したグラフを図 2.1-3に示す。採取した状況が異なるためグラフの値は異なるが、同じ品質のデータが採取できていることがわかる。

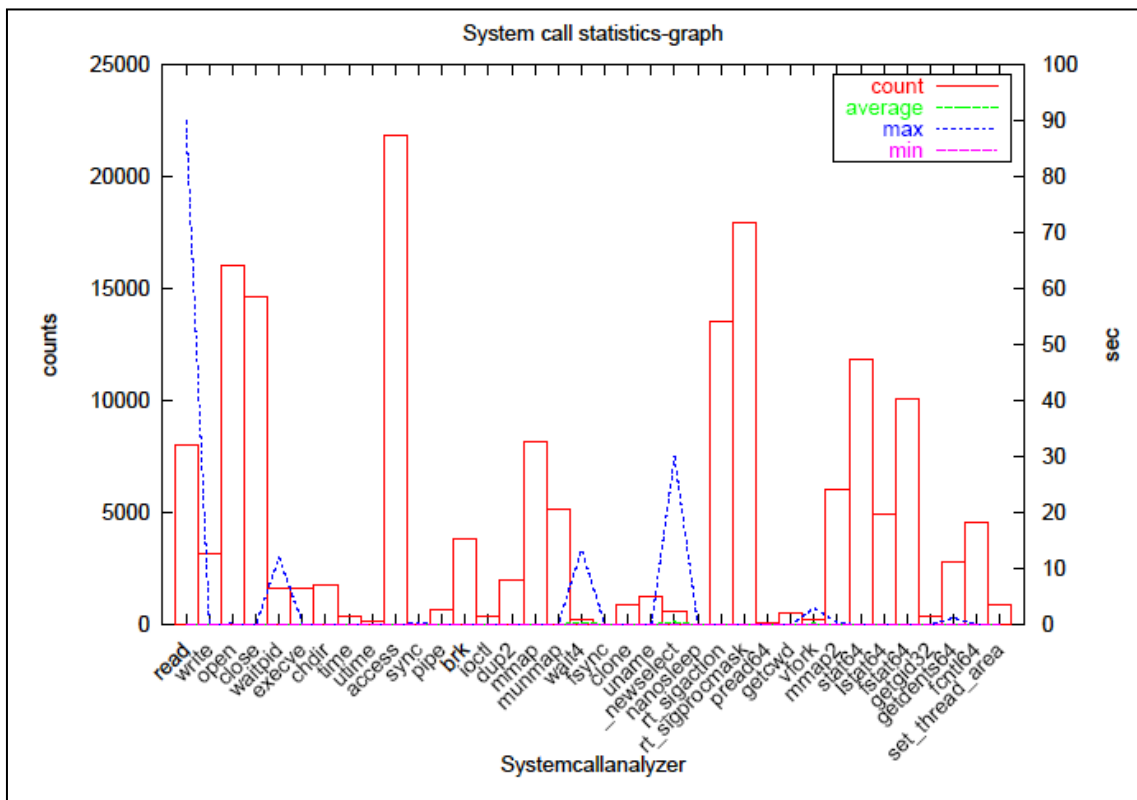


図 2.1-3 2004 年度で得たグラフ

2.2 LKSTLogTools の機能拡張

2.2.1 拡張した機能の概要

従来の LKSTLogTools では、収集した性能情報を 1 つのキーワードで選択、もしくは全部を可視化という形のものであった。これは開発開始時に、全データを可視化し、その中から特徴を見つけ出すことを目的としたためである。

しかし LKSTLogTools を開発し、使い始めてみると、何らかの情報の組み合わせで絞り込

んで解析を行うことが多いことに気がついた。そこで今回、解析で必要となった表 2.2-1に
 挙げる機能を開発した。これらの内容について、2.2.2項以降で詳細に説明する。

表 2.2-1 拡張した機能

項番	拡張範囲	概要
1	解析機能	時系列解析結果にプロセス名を併記する機能
2		ランキューにつながれてから実行するまでの時間を収集する機能
3		プロセス名によるフィルタリング機能
4	プロット ツール機能	複数のキーワードで情報を収集し、1つのグラフで表示する機能

2.2.2 評価環境

(1) 評価を行なったシステム環境を表 2.2-2に示す。

表 2.2-2 評価を行なうシステム環境

項番	項目	評価環境	
1	ハードウェア	CPU	Intel Xeon 3.20GHz×2
2		メモリ	8GB
3		ハードディスク	SCSI 140GB
4	ソフトウェア	カーネル	UpstreamKernel-2.6.9 (64 ビット) + 新 LSKT
5		OS	Fedora Core 3
6		解析ツール	lkstla
7		負荷ツール	lkstlogtools
8		ファイルシステム	ext3

(2) LKST 実行条件

評価を行なう際の LKST の実行条件は以下の通り。

- (a) 採取するイベントは LKST 性能評価用のイベントのみとする
- (b) 使用するハンドラはデフォルトハンドラ、lksteh_procstat および lksteh_procname

(3) イベント情報の採取

今回開発した機能の有効性の確認を行なうため、以下の手順でイベント情報の採取を採取する（記号は図 2.2-1に対応）。

- (a) lkst モジュールおよびプロセス名取得用のイベントハンドラをカーネルに組み込む。
このイベントハンドラではプロセス名情報を通常の記録バッファとは別の、特定バッファに書き込む
- (b) 解析用のマスクセットを作成し、有効にする

- (c) バッファを作成する
- (d) キャッシュを空にする
- (e) (c)で作成したバッファを使用可能にして、一度内容をクリアする
- (f) 負荷を生成する
- (g) イベントの記録を停止する
- (h) 通常のバッファとプロセス名記録用のバッファの内容をそれぞれファイルへ保存する
- (i) ファイルを CPU ごとのファイルに分割する (時系列保証のため)

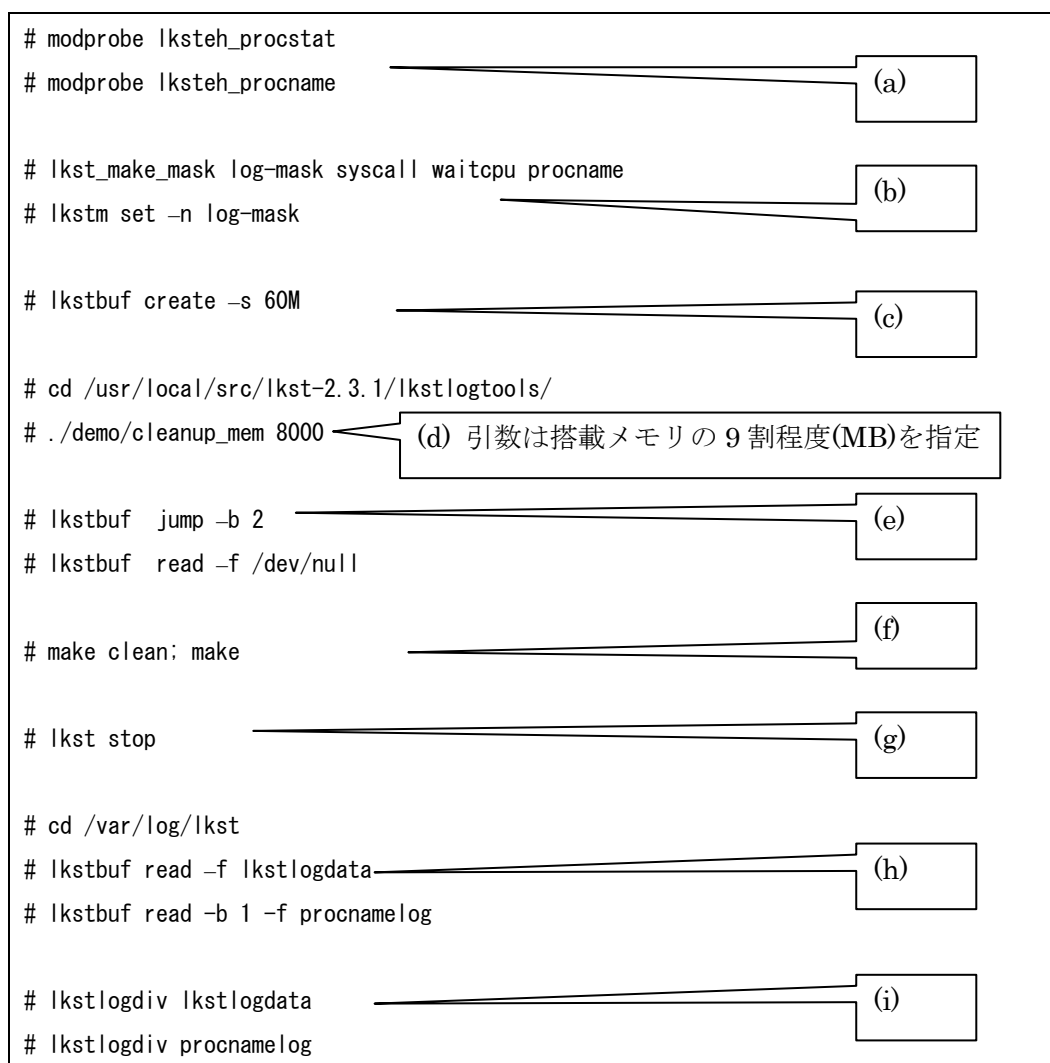


図 2.2-1 イベント情報の採取の手順

2.2.3 時系列解析結果にプロセス名を併記する機能

従来の lkstla では、procstat や proclive など一部のアナライザのみ、時系列解析結果にイベントを起こしたプロセス名が含まれていた。ユーザプロセスの振舞いや、プロセス間

の関係などを解析する場合、そのイベントが何のプロセスによって起こされたかという情報はとても有効である。そこで、全てのアナライザの時系列解析結果の各レコードに、そのイベントを起こしたプロセスの名前を併記する機能を開発した。

procstat や proclive アナライザは、その中で使用している PROCESS_SCHED_ENTER イベントからプロセス名を取得している。しかしこのイベントは発生頻度が高く、そのイベントを採取する必要が無い時にプロセス名を取得するためだけに採取するには、オーバーヘッドが大きい。またバッファの浪費にもつながる。そこで、本機能では子プロセスの生成 (fork) イベントとプログラムの実行開始 (execve) イベントからプロセス名を取得することとした。これらのイベントであれば PROCESS_SCHED_ENTER イベントに比べ発生頻度が低く、かつ必要なプロセス名を取得することができる。

以下のコマンドを実行し、プロセス名付きの時系列解析結果を得る。

```
# lkstla syscall -IN lkstlogdata-* procnamelog-* > syscall-withname.log
```

結果を図 2.2-2に示す。

System call analyzer sysno	syscall_name	comm	プロセス名	start[sec]	processing-time
2	open	sh	...	1127479014.320207175	0.000042430
72	fcntl	sh	...	1127479014.320250282	0.000000768
72	fcntl	sh	...	1127479014.320251475	0.000000970
72	fcntl	sh	...	1127479014.320252740	0.000000262
72	fcntl	sh	...	1127479014.320254172	0.000000377
33	dup2	sh	...	1127479014.320255529	0.000000810
3	close	sh	...	1127479014.320256817	0.000000245
1	write	sh	...	1127479014.320270308	0.000016339
33	dup2	sh	...	1127479014.320289057	0.000001025
72	fcntl	sh	...	1127479014.320290417	0.000000360
3	close	sh	...	1127479014.320291249	0.000000293
11	munmap	sh	...	1127479014.320323617	0.000005254
61	wait4	make	...	1127479014.316293973	0.004152029
15	rt_sigreturn	make	...	1127479014.320459514	0.000002255
14	rt_sigprocmask	make	...	1127479014.320465931	0.000000405
14	rt_sigprocmask	make	...	1127479014.320485172	0.000000275
4	stat	make	...	1127479014.320489077	0.000007124
4	stat	make	...	1127479014.320498406	0.000016389
4	stat	make	...	1127479014.320516940	0.000006269
4	stat	make	...	1127479014.320524547	0.000003752
4	stat	make	...	1127479014.320531973	0.000002853

図 2.2-2 プロセス名付きシステムコール処理時間の時系列解析結果 (抜粋)

本機能を用いることで、どのプロセスがどのようなシステムコールを発行しているか、そのプロセスが何をしているのか、などの情報が判断し易くなる。例えば図 2.2-2 の下の方で stat システムコールが連続して発生しているのが見てとれるが、それを実行しているのが

makeコマンドであることが分かれば、ソースファイルの更新の有無を調べているところである、という推測が簡単にできる。

2.2.4 ランキューにつながれてから実行するまでの時間を取得する機能

プロセスは実行可能な状態になるとランキューにつながれ、CPU上で実行されるのを待つ。この待ち時間が極端に長いと、プロセススケジューラに問題ある可能性がある。そこで、この情報を得るためのアナライザを開発した。

以下のコマンドを実行し、ランキュー上での待ち時間について、時系列解析結果を得る。なお、アイドルプロセス(PID 0)に関しては、本例ではPIDによるフィルタリング機能(-p オプション)を用いて除外する。

```
# lkstla waitcpu -p ¥!0 -s lkstlogdata-* > waitcpu.stat
```

時系列解析結果を表 2.2-3に示す。

表 2.2-3 CPU 待ち時間の統計 (ソート済み)

①

pid	task_name	count	average	max	min	total	percent
16740	as	26	0.000254522	0.004728336	0.000001738	0.006617575	0.358
16682	as	45	0.000116747	0.002498291	0.000001712	0.005253622	0.284
16681	cc1	40	0.000083416	0.002477056	0.000001528	0.003336622	0.18
16697	make	1106	0.000019219	0.002342061	0.000002207	0.021255714	1.149
16619	sh	18	0.000231012	0.002315668	0.000002000	0.004158220	0.225
16621	cc1	191	0.000037936	0.002249111	0.000002605	0.007245797	0.392
16731	as	28	0.000077728	0.002108016	0.000001690	0.002176381	0.118
16406	sh	7	0.000351637	0.002103442	0.000002990	0.002461457	0.133
16615	make	24	0.000603103	0.002067067	0.000002540	0.014474480	0.782
16730	cc1	14	0.000280999	0.002029842	0.000002457	0.003933988	0.213
16712	cc1	18	0.000149944	0.001997402	0.000002508	0.002698995	0.146
16722	as	27	0.000087516	0.001849996	0.000001738	0.002362923	0.128
16626	gensyms	45	0.000244762	0.001771266	0.000002523	0.011014276	0.595
16703	cc1	20	0.000111335	0.001760378	0.000002668	0.002226710	0.12
16403	make	6	0.000295367	0.001752310	0.000002358	0.001772203	0.096
16649	sh	18	0.000221899	0.001639914	0.000001632	0.003994177	0.216
16667	as	60	0.000108036	0.001603830	0.000001740	0.006482148	0.35
16664	sh	20	0.000202332	0.001559293	0.000001745	0.004046649	0.219
16598	make	359	0.000019938	0.001525998	0.000002055	0.007157827	0.387

表の各項目は左から順に次の意味を表している。

pid : プロセス ID (PID)
task_name : プロセス名
count : CPU 待ちが起こった回数
average : CPU 待ち時間の平均
max : CPU 待ち時間の最大値
min : CPU 待ち時間の最小値
total : CPU 待ち時間の総計
percent : CPU 待ち時間の総計の占める割合

CPU待ち時間の最大値が最も大きいプロセスを調べてみるとPID 16740 のasコマンドで0.004728336 秒であった(①)。そこでPID 16740 のプロセスに注目し、以下のコマンドを実行して時系列解析結果をグラフ化した。結果を図 2.2-3に示す。

```
# lkstla waitcpu -p 16740 -l lkstlogdata-* > waitcpu-16740.log  
# lkst_plot_log -l points waitcpu-16740.log
```

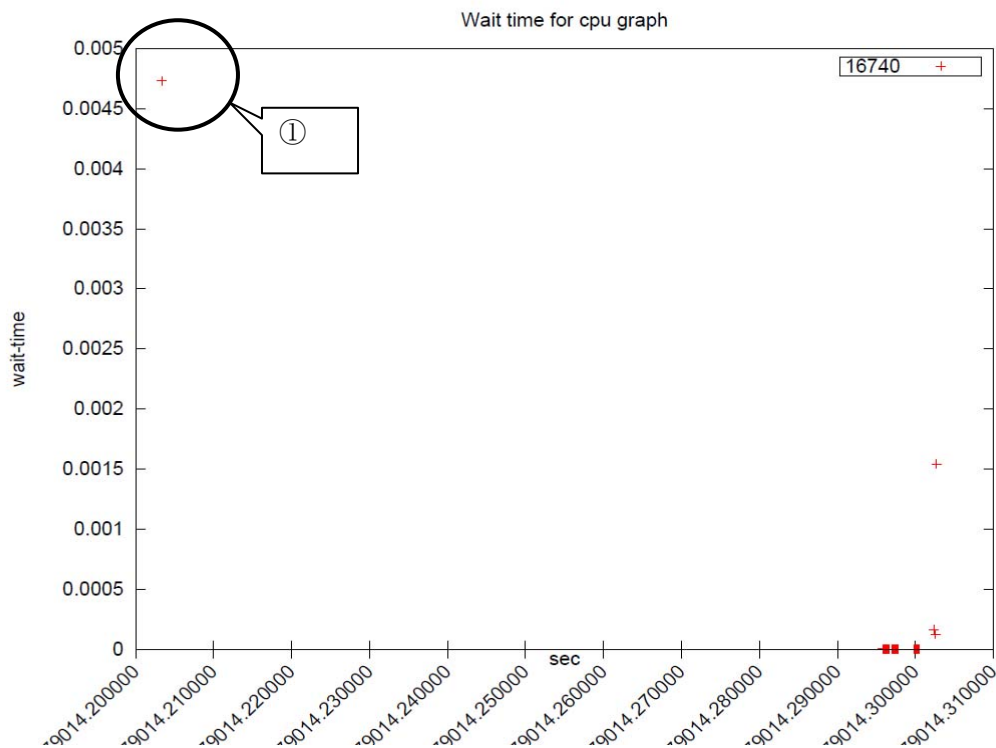


図 2.2-3 PID 16740 のプロセスの CPU 待ち時間

図 2.2-3を見ると、最初に長いCPUの待ちが発生している (①)。恐らくプロセス生成時と考えられる。それ以外の箇所ではそれほど長い待ち時間は発生していない。グラフの最

後の方に1点あるが、最初の時に比べると大きいものではない。したがって、このプロセスに関して言えば、プロセス生成時に何かの要因で待たされているだけで、稼働中のスケジューリングには問題はないと考えられる。

このようにこの機能は、プロセスのスケジューリング状態を確認する時などに使うことができる考える。

2.2.5 プロセス名によるフィルタリング機能

従来の `lkstla` では、特定のプロセスに注目して解析したい場合、対象とするプロセスの PID を `-p` オプションの引数として羅列する必要があった。例えばサーバプロセスが多数のスレッドで動作していた場合、それらのスレッドに注目して解析するには、まずどの PID がそのサーバプロセスのスレッドなのかを調べ、その PID を一つひとつ指定しなければならなかった。これに対して今回、「プロセス名」を指定して、特定プロセスの解析結果を得る機能を開発した。

特定プロセス名に関する解析データを得る手順は以下の通り。今回の例では `make` プロセスのシステムコールの統計情報を得る。

```
# lkst_proc_list procname log > procname.txt
# lkstla syscalls -p `lkst_proc_filter procname.txt make` -s lkstlogdata-* > syscalls-make.stat
```

得られた統計情報を表 2.2-4に示す。

表 2.2-4 `make` が発行するシステムコールの統計情報

sysno	syscall_name	count	Average	max	min	total	percent
0	read	2291	0.001335557	0.185766572	0.000000255	3.059761148	4.046
1	write	79	0.000014193	0.000027156	0.000002837	0.001121273	0.001
2	open	1161	0.000450364	0.011732329	0.000001738	0.522872362	0.691
3	close	1309	0.000002507	0.000032018	0.000000233	0.003282202	0.004
4	stat	7013	0.000146479	0.020459669	0.000001225	1.027254436	1.358
5	fstat	1171	0.000000549	0.000001618	0.000000305	0.000642764	0.001
8	lseek	13	0.000000586	0.000000878	0.000000293	0.000007624	0
9	mmap	940	0.000001782	0.000010037	0.000000775	0.001675427	0.002
10	mprotect	39	0.00000306	0.000007979	0.000002102	0.000119335	0
11	munmap	862	0.00000539	0.000019026	0.000003067	0.00464653	0.006
12	brk	282	0.000002943	0.000008614	0.000000313	0.000829964	0.001
13	rt_sigaction	199	0.000000356	0.000001102	0.000000205	0.000070889	0
14	rt_sigprocmask	356	0.000000488	0.000011229	0.000000227	0.000173711	0
15	rt_sigreturn	160	0.000002309	0.000003495	0.000000418	0.000369501	0
16	ioctl	2	0.00000132	0.000001697	0.000000942	0.000002639	0

21	access	21	0.000002773	0.000004258	0.000001482	0.000058233	0
22	pipe	82	0.000008925	0.000013647	0.000002932	0.000731831	0.001
58	vfork	160	0.001095924	0.046872893	0.000066178	0.175347878	0.232
59	execve	34	0.00080391	0.025355291	0.000001623	0.027332941	0.036
61	wait4	160	0.440183867	21.37282974	0.000003437	70.42941876	93.134
63	uname	13	0.000000742	0.000001018	0.000000055	0.000009651	0
72	fcntl	309	0.000000039	0.000001157	0.000000297	0.000120629	0
79	getcwd	22	0.00000244	0.000003422	0.00000016	0.000053683	0
80	chdir	22	0.000005422	0.000012709	0.000000317	0.000119278	0
96	gettimeofday	14	0.000000997	0.000001268	0.00000008	0.000013957	0
109	setpgid	2	0.000001154	0.000001285	0.000001022	0.000002307	0
158	arch_prctl	13	0.000001211	0.000001594	0.000000072	0.000015746	0
217	getdents64	661	0.000552594	0.01317068	0.000000041	0.365264679	0.483

この表から **make** コマンドは **stat** システムコールの発行回数が多いことが確認できる。**stat** システムコールはファイルの属性情報を取得するものであり、**make** するか否かを決める更新日付などの情報を取得していると考えられる。また、多くの子プロセスを生成するためか、**wait4** による待ち時間の合計が大きいことも分かる。

一連の処理の中で複数のプロセス（スレッド）が大量に生成される状況において、本機能があれば、特定プロセスの振舞いを容易に調査可能である。

2.2.6 複数のキーワードで情報を収集し、1つのグラフで表示する機能

2.1.3.3では、全システムコールの統計情報を取って可視化していたが、システムコールの種類が多いと煩雑になり評価しづらい。今回、特定の「キー」に注目し、それらのキーに関する情報のみを解析し、結果を表示する機能を開発した。例えば、システムコール処理時間の解析結果をグラフ化する場合、キーとしてシステムコール名をいくつか指定すれば、指定したシステムコールについてのみ描画することができる。

複数のキー情報の解析結果を1つのグラフで表示する得る手順は以下の通り。今回の例では、以下の10のシステムコールについてのみ、統計情報を可視化する。

select,wait4,poll,epoll_wait,read,open,stat,execve,getdents64,vfork

以下のコマンドを実行し、統計解析結果を得る

```
# lkst_plot_stat --log -a -k select,wait4,poll,epoll_wait,read,open,stat,execve,\
getdents64,vfork syscall.stat
```

作成したグラフを図 2.2-4に示す。

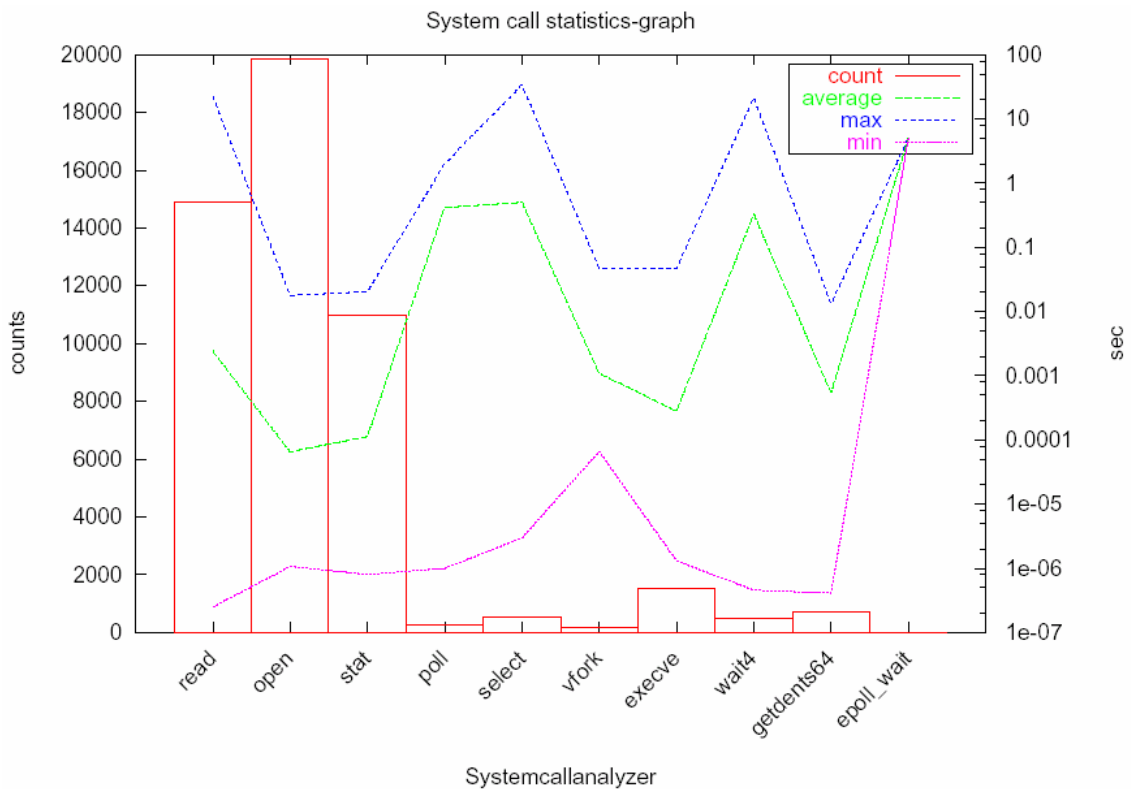


図 2.2-4 lkstlogtools 構築時にプロセスの実行時間の多くを占めるシステムコールの統計

図 2.1-2と比較すると見やすさが見て取れる。図 2.1-2は複数のデータから傾向や特出すべき特徴を探すには良いグラフであるが、特定の処理に関連した一連のシステムコールに注目したい場合には、図 2.2-4のグラフの方が有効である。

以上のように、今回拡張した機能を用いると、情報の絞込みが容易にでき、比較も行い易くなる。

2.3 最新版 LKST の MIRACLE LINUX V3.0 への適用

前回まで開発した LKST は MIRACLE LINUX にも対応している。そこで、前回対応した MIRACLE LINUX V3.0(*)へ最新版 LKST を適用 (移植) し、EM64T サーバ、および IA-32 サーバでの動作確認を行った。

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

2.3.1 移植における課題と対策

今回の開発で検出した課題とその対策について説明する。

2.3.1.1 kernel に対する修正

カーネルへ LKST のフックポイントなどを適用するために、以下の作業を行った。

- (1) 2.4 カーネルと 2.6 カーネルではブロック I/O のコードが大きく違うため、ブロック I/O 関係のフックポイント(BIO_MAKE_REQ2、ELV_NEXT_REQ)は近似箇所を探して適用した。
- (2) EM64Tに対応する際に、1つのイベントを2つに分割する作業を行った (2.1.2項参照)。具体的にはSYSTEM_INFOおよびMEM_VM_INFOイベントであるが、仕様の統一を図るため、この分割を 2.4 カーネルにも適用した。
- (3) 今回開発した lksteh_procname イベントハンドラの追加と、同イベントハンドラが使用する PROCESS_EXEC イベント用のフックポイントを追加した。
- (4) また、移植に際に BUFFER_SUBMIT_BH、BUFFER_END_BH、BIO_END_IO については、従来のフックポイントでは不十分であることが分かったため、追加のフックポイントを挿入した。このフックポイントにより Direct I/O を用いた I/O と、ジャーナルへのコミットに関する I/O イベントが採取可能となる。

上記以外はフックポイントの挿入箇所には 2.4 カーネルと 2.6 カーネルの差はほとんどなかった。

2.3.1.2 lkstla の修正

2.4 カーネルと 2.6 カーネルではブロック I/O の処理が異なり、LKST が捉えるイベントの発生順序も異なる。そのため、デバイスドライバから見た I/O 処理時間を解析する bdevtime アナライザでは、処理内容を修正する必要がある。2.4 カーネルと 2.6 カーネルの双方をサポートするように、2.4 カーネル用 bdevtime アナライザを追加開発した。

2.3.2 移植の成果

最新版 LKST を MIRACLE LINUX V3.0 へ適用する作業を行うことにより、以下の成果が得られた。

- (1) MIRACLE LINUX V3.0 で最新版 LKST、特に評価機能を利用することが可能になった。
- (2) LKST を MIRACLE LINUX V3.0 製品へ移植する際の注意点と課題の洗い出しができた。また今回の移植により対策案の検討が済んでいるので、今後の移植は短期間で実現することができる。
- (3) (1) は IA-32/EM64T の双方の環境で稼動することができる。

(3) の結果を利用すれば、32 ビット／64 ビット環境でのカーネルの動作比較を行うことができる。実際にDBサーバ動作時の 32 ビット／64 ビット環境でのカーネルの動作比較を行っているので、3.3節を参照して頂きたい。

3 LKSTを用いた性能評価に関する考察 –DB 実行時のカーネル評価–

3.1 評価目的

Linux をベースとするシステムの1つにDBサーバがある。このDBサーバには性能を測定するベンチマークも多くあり、DBサーバ自身を評価することができる。しかし、そこで何らかの現象が起きた時に、その事象がカーネルにどのように関連しているのかを判断する手段は今まで無かった。もし性能が悪くなった場合に、カーネルの中で時間がかかる処理を実行しているのか、ロックで待たされているのか、カーネルでの処理時間は短いのか、などが容易にわかると、DBサーバの処理と照らし合わせることで性能に関する考察が深まり、改善などのアイデアも生まれると考える。

今回、このカーネルの情報を収集する手段として、性能評価機能を備えたLinuxカーネルのトレーサLKSTを用いて、DBサーバ実行時のカーネルの評価を行った。実際にはDBサーバとしてMaxDBを、DBサーバのベンチマークツールとしてOSDL DBT-1を使用し、3.2節に示す評価を行なった。

3.2 評価概要

(1) 32ビット/64ビット環境での実行状況評価

今回の開発でLKSTおよびlkstlaのEM64T対応を行った。これにより32ビット/64ビットの両方のモードでの実行が可能になった。そこでDBサーバとしてMaxDBを、ベンチマークとしてDBT-1を使用し、32ビット/64ビットにおいて、システムコールの処理時間を見ることで、どのような性能状態にあるかを評価した。その結果、性能の変化を確認することができた。

(2) BuyConfirm インタラクショナル実行時のカーネルの実行状況評価

2004年度に実施したDB層の評価では、OSDL DBT-1を用いてオープンソースDBMSの評価を行っている。この中で、DBT-1のインタラクショナル毎の実行状況が使用するDBMSによって異なっていることが見て取れた。特に図3.2-1で示すように、DBT-1 MaxDB (ストアド) 版の「BuyConfirm」インタラクショナルのレスポンスが、他のインタラクショナルやPostgreSQLの実行結果に比べて時間がかかっていた。

また、今年度のDB層評価では、DBT-1のロジック部分をストアド (MaxDBのストアドをベースとしている) からC言語に移行し、MySQL及びPostgreSQLにポーティングを行った。これらを利用したDBT-1のインタラクショナル実行結果では、図3.2-1中のPostgreSQLに近いものとなっており、そのことからDBT-1 MaxDB版でのBuyConfirmに何らかの問題が含まれているのではないかと推測している。

DBTによる評価では、このBuyConfirmインタラクショナルに性能問題があるところまでは捕捉出来るが、それ以上の詳細な挙動に関しては捕らえられない。そこでLKSTを利用した解析を行った。

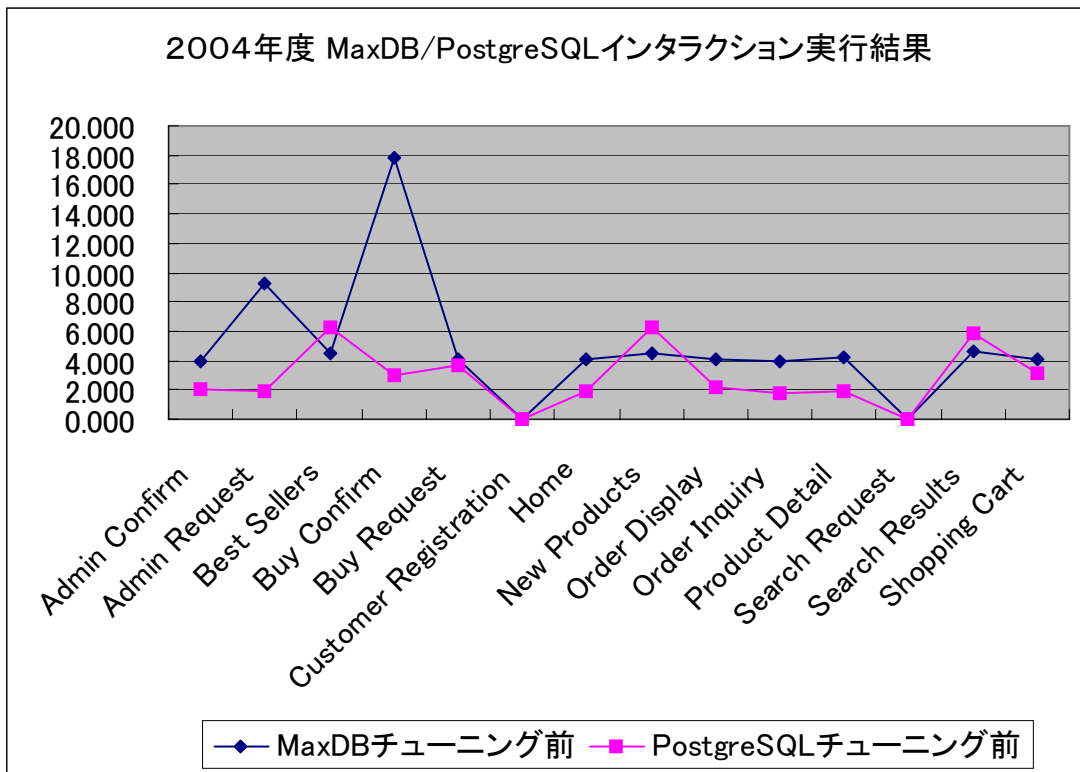


図 3.2-1 インタラクション分布による比較 (MaxDB、PostgreSQL)

BuyConfirm インタラクション実行時のカーネルの実行状況評価は、次の 2 つの視点で行なった。

(a) DBT-1 ランプアップ時

DBT-1 ランプアップ時の状況を観測したところ、極端に長い処理時間を要する BuyConfirm インタラクションが発生する異常現象が確認された。そこで LKST による詳細分析を行なったところ、ランプアップ時に DB のデータボリュームの先頭約 1GB をメモリ上に読み込んでいる様子が確認できた。

(b) DBT-1 平常時

DBT-1 平常時にも、ランプアップ時ほどではないが、やはり BuyConfirm で長い処理時間を要することがある。LKST で詳細な調査を行なってみると、BuyConfirm 処理時にセマフォによる長時間の待ちが多発し、全体的に処理が遅くなることがわかった。

3.2.1 システム環境

評価を行ったシステム環境を表 3.2-1に示す。

表 3.2-1 評価を行うシステム環境

項番	項目	評価環境	
		32bit	64bit(EM64T)
1	ハードウェア	CPU	Intel Xeon 3.20GHz× 2
2		メモリ	8GB
3		ハードディスク	SCSI 140GB
4	ソフトウェア	OS	MIRACLE LINUX V3.0
5		カーネル	2.4.21-19.38AX + LKST 2.4.21-20.19AX + LKST
6		解析ツール	LKST と lkstla
7		負荷ツール	OSDL DBT-1
8		DB	MaxDB-7.5.00.19(*)
9		ファイルシステム	ext3

*) MaxDB

ライセンスは GPL。パラメータや設定事項が少なく、GUI が豊富で、管理が容易である。

3.2.2 DBT-1 の概要

3.2.2.1 概要

OSDL DataBase Test1 プロジェクト(OSDL DBT-1)は、LinuxOS やオープンソースソフトウェアのためのトランザクションベースの負荷テストツールとして、簡便に利用できるように開発された。このツールは TPC-W をベースに単純化したものである。

TPC-W ワークロードは、Web ベースのオンライン書店から商品を購入するユーザの活動をシミュレートする。OSDL DBT-1 テストは、TPC-W のワークロード特性を使用して、実行時のボトルネックを検出するため、また相対的なパフォーマンス向上のための指標として利用できる。

3.2.2.2 基本的な構造

DBT-1 は表 3.2-2 ようにモジュールに分かれて構成されている。また各モジュール間の関係を示す構成図を図 3.2-2 に示す。

表 3.2-2 DBT-1 構成表

名称	機能概要
dbdriver	トランザクションを起動する仮想ユーザの動作をシミュレートする
appServer	トランザクション管理サーバ
appCache	アプリケーションサーバのキャッシュをシミュレートするキャッシュサーバ
DBT-1 LibODBC	DBT-1 の ODBC インターフェース。このライブラリを経由してデータベースにアクセスする
MaxDB ODBC	MaxDB の ODBC ドライバ

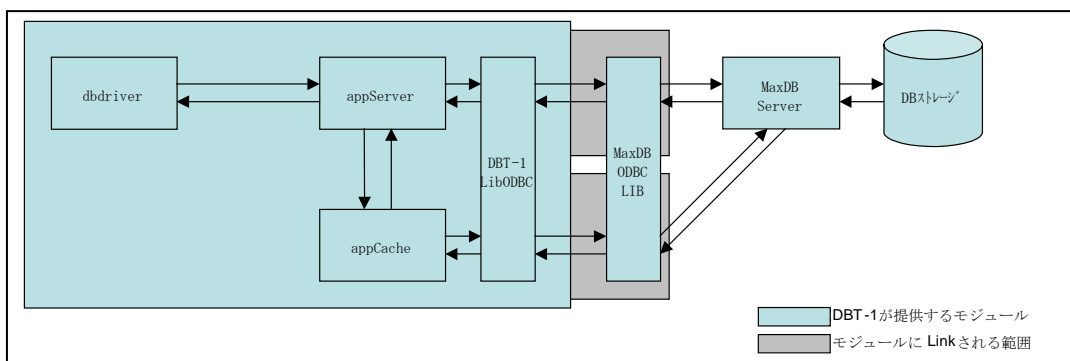


図 3.2-2 DBT-1 構成概要

dbdriver は想定された仮想ユーザ数のスレッドを立ち上げ、個々のユーザ操作を開始する。仮想ユーザの活動をシミュレートする各スレッドから構成され、マルチスレッドで動作する。

appServer は Web 3 階層モデルにおける、アプリケーション層にあたるソフトウェアである。dbdriver から要求されたトランザクションを受け取り、データベースにクエリを送り、得られた結果を dbdriver に返す。また、多くの個々の仮想ユーザを扱うために、指定された接続数でデータベース接続を行う。

3.2.2.3 DBT-1 のトランザクション概要

DBT-1 では表 3.2-3に示すTPC-Wに準拠したインタラクションをトランザクションパターンとして利用している。

表 3.2-3 TPC-W インタラクション一覧

番号	インタラクション	内容
1	ホーム (Home)	仮想店舗のホームページ。製品リスト (新製品、ベストセラー製品) へのリンクを含むウェブページの情報 を返す。
2	ショッピングカート (Shopping Cart)	関連する顧客のカート情報(日付、追加した新しいアイテム、既存のアイテム情報)を更新し、最新のカート情報を返す。ショッピング・セッションが未定義の場合、新しいセッションを作成する。
3※	顧客登録 (Customer Registration)	既存顧客の場合、登録されている必要な情報を返し、新規顧客の場合には登録を促すページを返す。
4	購入リクエスト (Buy Request)	登録済みの新規顧客または認識されている顧客かを確認し、顧客についての情報を表示する。また、カートに関するサマリ情報と、クレジットカード情報やショッピング・オプションの入力フィールドを含めたページを表示する。
5	購入確認(Buy Confirm)	既存顧客の関連するカートの内容を新しく注文として作成し、支払い認証を実行する。新しく作られた注文の明細を含むページを返す。
6	注文照会(Order Inquiry)	再訪した顧客の本人認証のためのページを返す。以降の処理のエントリページとなる。
7	注文表示(Order Display)	顧客によって出された最終の注文のステータスを返す。
8※	検索リクエスト (Search Request)	指定した商品を検索する条件を入力するページを返す。
9	検索結果表示 (Search Results)	与えられた条件に適合する商品リストのページを返す。
1 0	新商品(New Products)	最近リリースされた商品一覧のページを返す。
1 1	ベストセラー (Best Sellers)	ベストセラーの商品一覧のページを返す。
1 2	商品詳細 (Product Detail)	選択した商品の詳細情報のページを返す。
1 3	管理者リクエスト (Admin Request)	顧客に商品の最新版へのアクセスを許可するページを返す。
1 4	管理者確認 (Admin Confirm)	商品を更新し、更新した商品の詳細を確認するページを返す。

※ 項番3に関しては現在実装されていない。

※ 項番8に関しては実装されているが、利用されていない。

3.2.2.4 DBT-1 のコンフィギュレーション

DBT-1 のコンフィギュレーションはDBT-1 インストールディレクトリ以下にある `scripts/stats/dbt1.config` 編集することで行なう。今回使用した `dbt1.config` の内容を図 3.2-3 に示す。3.3節の測定では、仮想ユーザ数を示す `eu` の値 (図 3.2-3の(1)) を 200、800 または 1600 に変えて、DBT-1 を実行した。

```
[database]
#hostname instance username password
localhost:DBT1:dbt:dbt

[cache]
#hostname port dbconnections items appCache_executable_directory
localhost:9999:5:10000:/home/sapdb/dbt1-v2.1/cache

[appServer]
#hostname
localhost
#port
9992
#dbconnection
100
#transaction_queue_size
2000
#transaction_array_size
2000
#items
10000
#appServer executable directory
/home/sapdb/dbt1-v2.1/appServer

[dbdriver]
#hostname
localhost
#items
10000
#customers
2880000
#eu
200
#eu/min
100
#mean think_time
7.2
#run_duration in seconds
4200
#dbdriver executable directory
/home/sapdb/dbt1-v2.1/dbdriver
```

図 3.2-3 dbt1.config の内容

3.2.2.5 負荷パターン

図 3.2-4の通り、DBT-1 の負荷パターンは設定した仮想ユーザの接続が全て接続した後で、最初の仮想ユーザから順次処理を終えるまでの間に負荷のピークが来るように設計されている。従って、計測時には負荷のピークのタイミングを想定する必要がある。

ランプアップ時間は、全ての仮想ユーザが接続し、アクションを開始するまでの時間であり、(eu)/(eu/min)により計算できる。euおよびeu/minは図 3.2-3のdbt1.configに記述されているパラメータ ((1), (2)) である。

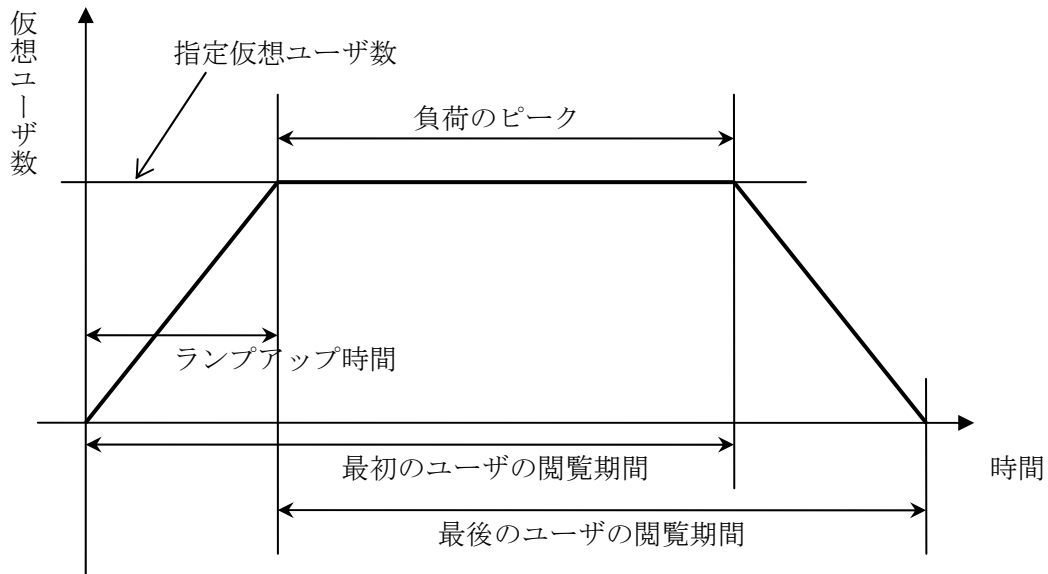


図 3.2-4 負荷のかかり方

3.2.3 LKSTによるログの採取手法

通常 LKST では、解析に必要となる全てのイベントをトレースするよう構成したマスクセットを用いて、ログを採取する。しかしトレース対象のイベントを多くすると、LKST のイベントバッファが埋まるペースも速くなり、結果として短期間のイベントしか記録できない問題が起こる。特に IA-32 用のカーネルでは、最大でも 100MB 強のイベントバッファしか確保できないため、何らかの対応が必要である。

DBT-1 の実行においては、ランプアップ終了後は処理内容が定常状態になると予想できる。そのため少数のイベントをトレースする複数のマスクセットを、一定時間間隔で切り替えながらログを採取することにした。以後、この一連のマスクセットを「マスクセットシリーズ」と呼ぶことにする。この方法により、多数の種類イベントを比較的長時間、採取することができる。各マスクセットは、例えばシステムコールの解析用、I/O の解析用、スケジューラの解析用というようなものを用意し、それぞれ解析に必要な最低限のイベントのみ採取するよう記述した。

3.2.3.1 マスクセットシリーズ

図 3.2-5はマスクセット 1～nから成るマスクセットシリーズを使用した状況を示しており、60 秒おきにマスクセットを切り替えている。あるマスクセットにおけるログの採取時間は 30 秒とし、ログをファイルに出力した後、30 秒の間隔を空ける。間隔を空ける理由は、LKSTのログ出力処理の影響を次の測定に残さないようにするためである。

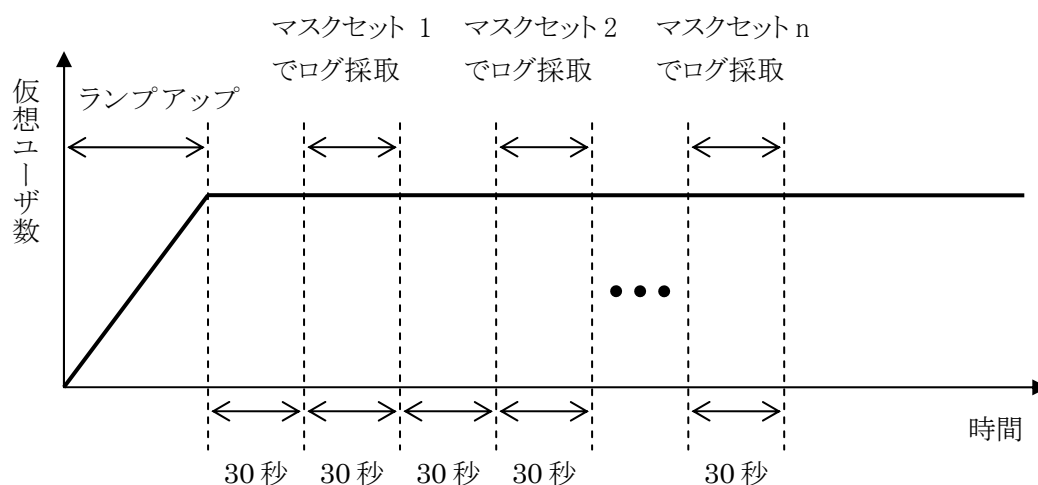


図 3.2-5 マスクセットシリーズを用いたマスクセットの切り替え

今回の測定では、目的に応じ、使用するマスクセットの組（図 3.2-5のマスクセット 1～nの組）を変更して、LKSTによるログの採取を行なった。今回の評価で使用したマスクセットを表 3.2-4に、拡張イベントハンドラを表 3.2-5に示す。表中の○印はデフォルトのイベントハンドラでイベントを採取することを意味し、空白のセルはイベントの採取を行わないことを意味する。アルファベット 2 文字が記載されている部分は拡張イベントハンドラを用いていることを示し、表 3.2-5の略号に対応する。各マスクセットの用途を表 3.2-6に示す。

表 3.2-4 マスクセットシリーズに使用したマスクセット

イベント	マスクセット											
	1	2	3	4	5	6	7	8	9	10	11	12
	exception	io	iposyscall	Netsyscall	schedule	spinlock	syscall	waitcpu	waitqueue	rampup	wqiposyscall	wqsyscall
PROCESS_CONTEXTSWITCH					○			PS		PS		
PROCESS_WAKEUP					○			PS		PS		
PROCESS_ADD_WQ									WC	WC	WC	WC
PROCESS_REM_WQ									○	○	○	○
PROCESS_SCHED_ENTER					○			○				
PROCESS_SCHED_EXIT					○							
PROCESS_FORK	PN	PN	PN	PN	PN	PN	PN	PN	PN	PN	PN	PN
PROCESS_EXIT												
PROCESS_EXEC	PN	PN	PN	PN	PN	PN	PN	PN	PN	PN	PN	PN
EXCEPTION_ENTRY	○											
EXCEPTION_EXIT	○											
SYSCALL_ENTRY			○	NT			NT			○	○	○
SYSCALL_EXIT			○	NT			NT			○	○	○
NET_PKTSEND				○								
NET_PKTSENDI												
NET_PKTRECV												
NET_PKTRECVI												
NET_SOCKETIF				○								
SYSV_IPC_SEMOP			○								○	
SYSV_IPC_SEMGET			○								○	
SYSV_IPC_SEMCTL			○								○	
SYSV_IPC_MSGSEND			○								○	
SYSV_IPC_MSGRCV			○								○	
SYSV_IPC_MSGGET			○								○	
SYSV_IPC_MSGCTL			○								○	
SYSV_IPC_SHMAT			○								○	
SYSV_IPC_SHMDT			○								○	
SYSV_IPC_SHMGET			○								○	
SYSV_IPC_SHMCTL			○								○	
LK_SPINLOCK						○						
LK_SPINTRYLOCK						○						
LK_SPINUNLOCK						○						
BUFFER_SUBMIT_BH		○								○		
BUFFER_ENDIO_BH		○								○		
BIO_END_IO		○								○		
BLK_PUT_REQ		○								○		
ELV_NEXT_REQ		○								○		
BIO_MAKE_REQ2		○								○		
NET_START_XMIT				○								

表 3.2-5 使用した拡張イベントハンドラ

イベントハンドラ名	略号	説明
procname	PN	プロセス名と PID の対応関係、プロセスの親子関係を記録する。
wqcounter	WC	ウェイトキューに関する解析を行なう場合に必要な情報を記録する。
nosystime	NT	システムコール関係のイベントを記録する際、time システムコールを記録しないようにする。これは IA-32 上で DBT-1 を実行すると、time システムコールが非常に高頻度で発行され、他のイベントを記録するスペースがなくなってしまう問題を回避するために利用する。
procstat	PS	プロセスの状態遷移情報を記録する。

表 3.2-6 マスクセットと用途

項番	マスクセット名	用途
1	exception	例外の発生頻度や処理時間に異常がないか調査する。
2	io	ブロック I/O に関する分析を行なう。I/O の頻度や処理時間をレイヤ別、read/write 別、デバイス別、セクタ別などで取得する。
3	ipcsyscall	IPC 関係のシステムコールイベントを記録する。特に IA-32 で IPC 関係のシステムコールの処理時間の解析向け。
4	netsyscall	ソケット関係のシステムコールイベントを記録する。特に IA-32 でソケット関係のシステムコールの処理時間の解析向け。
5	schedule	プロセススケジューラに性能上の問題がないか調査する。
6	spinlock	性能のボトルネックになっているスピロックがないか調査する。
7	syscall	システムコールイベントを記録する。3,4,11 より長時間ログを採取したい場合はこのマスクセットを使用する。
8	waitcpu	プロセススケジューラが公平なスケジューリングを行なっているか調査する。
9	waitqueue	ウェイトキューによる待ち時間を解析し、ボトルネックを探す。
10	rampup	DBT-1 におけるランプアップ時の様子を分析する。I/O 関係の調査、システムコールの調査、プロセスの状態遷移の調査を行なう。
11	wqipcsyscall	IPC 関係のシステムコールと、ウェイトキューによる待ち時間の関連性を調査する。
12	wqsyscall	システムコールと、ウェイトキューによる待ち時間の関連性を調査する。

3.2.3.2 マスクセットの策定と利用

表 3.2-4に挙げるマスクセットの策定は以下の手順で行なった。

- (1) 事前調査を行なう。
 - (a) 全てのイベントの採取を有効にして、測定対象実行時のログを採取する
 - (b) `lkstla event -s <ログファイル>` を実行し、各イベントの発生頻度を調べる
 - (c) `lkstla event -l <ログファイル>` を実行し、最初と最後のイベントの発生時刻から、何秒間ログを採取できたか調べる
 - (d) (b)と(c)から各イベントが、単位時間あたりどれだけ発生するか予測値を求める
- (2) 解析に使用するアナライザが必要とするイベント、そのほか解析に利用できそうなイベントをリストアップする。
- (3) (2)でリストアップしたイベントを、関連性のあるイベント同士でグループ化し、それを1つのマスクセットとする。例えばネットワーク関連のイベント(イベント名が `NET_` で始まるもの)全てを採取するように構成したマスクセットを作る、など。
- (4) (1)で算出した各イベントの発生頻度から、(3)で作成したマスクセットで期待した時間のログを採取できるか検討する。期待した時間、採取できそうになれば、別のマスクセットに分ける。例えば待ち時間関連のイベント `PROCESS_ADD_WQ`、`PROCESS_REM_WQ`、`LK_SPINLOCK` から成るマスクセットを作成しても、`LK_SPINLOCK` の発生頻度は非常に高いため、残り2つのイベント情報が少量しかログに残らなくなる。この場合、`LK_SPINLOCK` は別マスクセットに分けるべきである。

解析対象の問題点が明確でない場合、まず表 3.2-4のマスクセット 1~9 から成るマスクセットシリーズを使用してログを採取し、様々な観点から分析すると良い。ある程度明確になったら、目的に応じてマスクセットおよびマスクセットシリーズを構成し直す。例えばウェイトキューによる大きな待ちが発生していることが分かれば、11 や 12 のようなマスクセットを使用する。

3.2.3.3 ログの採取箇所

ログの採取は図 3.2-6に示す(a)~(c)の 3 箇所で行なう。

- (a) ランプアップ直後：
全ての仮想ユーザがアクションを開始した状態であるが、まだ安定状態になっていない可能性がある。
- (b) ランプアップ後 15 分後：
安定稼動中と思われる。
- (c) ランプアップ後 30 分後：
(b)と比較しあまり差がなければ、ランプアップ後 15 分以降は安定稼動中であるといえる。

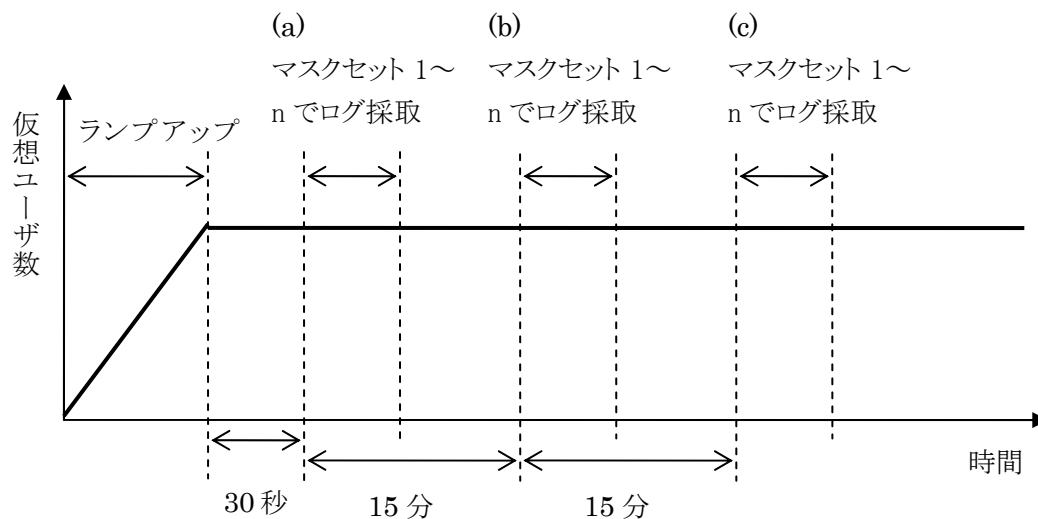


図 3.2-6 ログの採取箇所

3.2.3.4 ログの採取手順

ログの採取のステップを以下に示す。

- (1) ランプアップ終了後、LKST のモジュールやイベントバッファのセットアップを行う

```
# modprobe lkst
# modprobe lksteh_procname
# modprobe lksteh_wqcounter
# modprobe lksteh_procstat
# lkstbuf create -s 30M
New buffer was created, cpu=0, id=1 size=31457280 + 4032(margin)
New buffer was created, cpu=1, id=1 size=31457280 + 4032(margin)
# lkst stop
Stop LKST event tracing.
# lkstbuf jump -b 1
Currently selected buffer changed to 1 on CPU 0
# lkstbuf del -b 0
Buffer id=0 was deleted.
```

- (2) 30 秒後、最初のマスクセット(ここでは maskset1)にてログの採取を開始する

```
lkstm write -S -f maskset1
```

- (3) さらに 30 秒後、ログの採取を停止し、イベントバッファの内容をファイル(ここでは log-maskset1)に書き出す

```
lkst stop
lkstbuf read -f log-maskset1
```

- (4) 30 秒後、次のマスクセット(ここでは maskset2)にてログの採取を開始する

```
lkstm write -S -f maskset2
```

- (5) 以下同様、マスクセット n でのログの採取が完了するまで繰り返す
(6) マスクセット 1 でログの採取を開始してから 15 分後、再びマスクセット 1 でログの採取を開始する(2 周目)
(7) 以下同様、3 周目完了まで繰り返す

3.3 32ビット／64ビット環境での実行状況評価

3.3.1 測定目的

DBサーバとしてMaxDBを、ベンチマークとしてDBT-1を使用し、32ビット／64ビットで性能がどれくらい変化するかの評価を行った。

DBT-1にはDBT-1が算出する性能値としてBT値がある。この値は単位時間当たり処理可能な擬似トランザクション数を意味し、全体性能の変化を判断するには有用な値である。しかし、この情報だけでは、カーネルの処理性能が均一に向上しているのか、それとも特定処理の性能だけが向上しているのかの判断はできない。カーネルの性能情報がわかると64ビット環境でDBT-1の性能向上の検討時にヒントとなると考え、LKSTで評価を行うこととした。

3.3.2 測定内容

(1) 測定内容とコンフィギュレーション

MaxDBにおいて、仮想ユーザ数 200/800/1600 の3種類でDBT-1を実行し、LKSTによるログの採取を行なった。またDBT-1のコンフィギュレーションに関しては図 3.2-3を参照のこと。

(2) LKSTにおける測定項目

DBT-1ではシステムコールの発行が多いため、そのシステムコールの性能を見ると特徴を捉えることができるのではないかと考え、測定を行った。発行される全システムコールを測定したが、ここではその中でも発行回数の多かったread/write、mmap/munmapを示す。

- read：ファイルデータの読み取り処理
- write：ファイルデータの書き込み処理
- mmap：ファイルをメモリにマップする処理
- munmap：ファイルをメモリからアンマップする処理

3.3.3 測定結果

(1) BT値

DBT-1が算出したBT値を表 3.3-1に示す。

表 3.3-1 各プロセス数における32ビット／64ビットでのBT値

	32ビット時	64ビット時	比率
仮想ユーザ数：200	27.2	27.1	0.9963
仮想ユーザ数：800	100.1	101.2	1.011
仮想ユーザ数：1600	134.2	143.6	1.07

この結果から、仮想ユーザ数が多いほど、64 ビット環境の方が性能が良いことが分かる。すなわち高負荷時において 64 ビット環境は性能向上していることが分かる。

(2) システムコール処理性能

(1)と同じ条件において LKST で性能を測定した、32 ビット/64 ビット環境における各システムコールの性能のグラフを図 3.3-1～図 3.3-8 以下に示す。

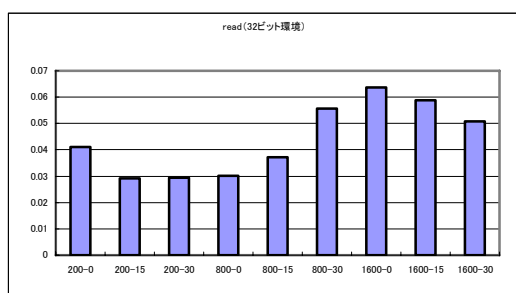


図 3.3-1 read(32 ビット環境)

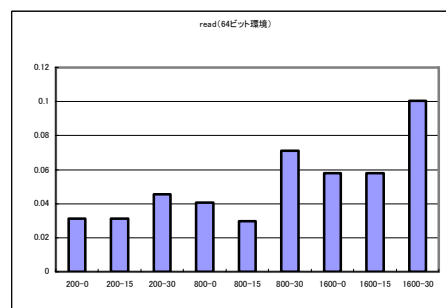


図 3.3-2 read(64 ビット環境)

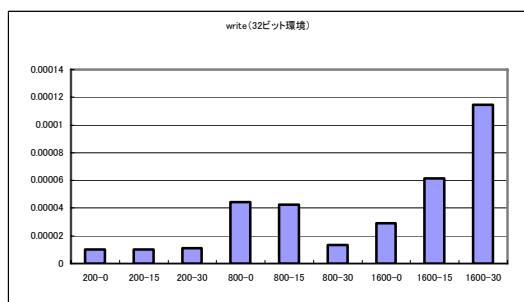


図 3.3-3 write(32 ビット環境)

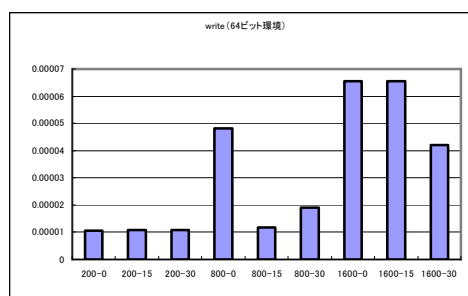


図 3.3-4 write(64 ビット環境)

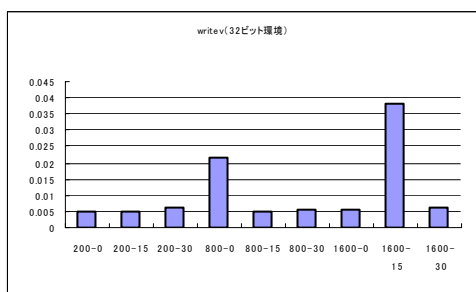


図 3.3-5 mmap(32 ビット環境)

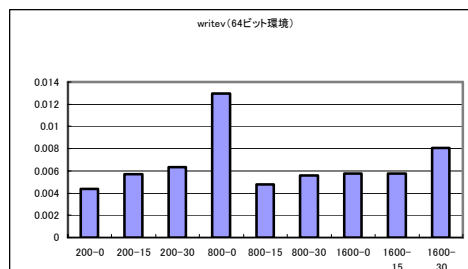


図 3.3-6 mmap(64 ビット環境)

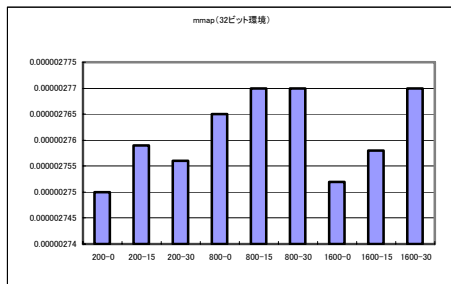


図 3.3-7 munmap(32 ビット環境)

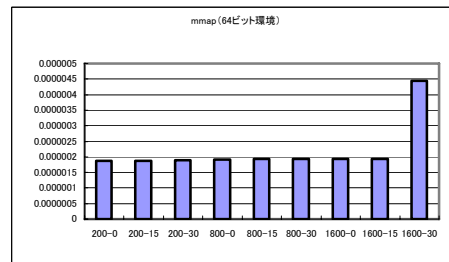


図 3.3-8 munmap(64 ビット環境)

3.3.4 評価と考察

LKST を用いて測定した結果、各システムコールにおいて、200/800/1600 で性能の変化を確認することができた。BT 値だけでは知りえない情報、すなわち性能は一律に向上しているわけではなく、大きく向上／低下しているもあれば、小さな差しかない状況もある、という変化があることを LKST で知ることができた。

性能の差は、その時点での負荷にも依存するため一概には言えない。しかし、LKST で何度か測定し、可視化した結果を見比べれば、性能に一定のパターンがあるか、安定性はあるかなどを知ることができ、そのアプリケーションとプラットフォームとの相性を判断する材料にも使えると考える。

3.4 DBT-1 の BuyConfirm インタラクションの実行状況評価

DBT-1 の BuyConfirm インタラクションの実行状況について、LKST を用いて評価を行った。本評価は 64 ビット環境で実施した。

3.4.1 DBT-1 のランプアップ時の状況

DBT-1 のプログラムappServerはインタラクションの処理時間に関するログをtime.logに書き出している。このログを見ると、ランプアップ中に処理されたBuyConfirmインタラクションの中で、極端に処理時間を要するものがいくつか存在することが分かった。さらに様々なコンフィギュレーションでDBT-1 を実行したが、どの場合もこの現象は現れた。図 3.4-1はtime.logの情報をもとに、BuyConfirmの処理時間をグラフ化したものの一例である。緑のライン(破線)はDBでの処理時間を表し、赤のライン(実線)はappServerにおける、DBを含めた処理時間である。また、横軸はランプアップ開始からの経過時間、縦軸はBuyConfirmインタラクションの処理時間である。このグラフからDBT-1 開始 120 秒くらいの時に 100 秒を越える処理時間を要したBuyConfirmが存在している様子が見てとれる。

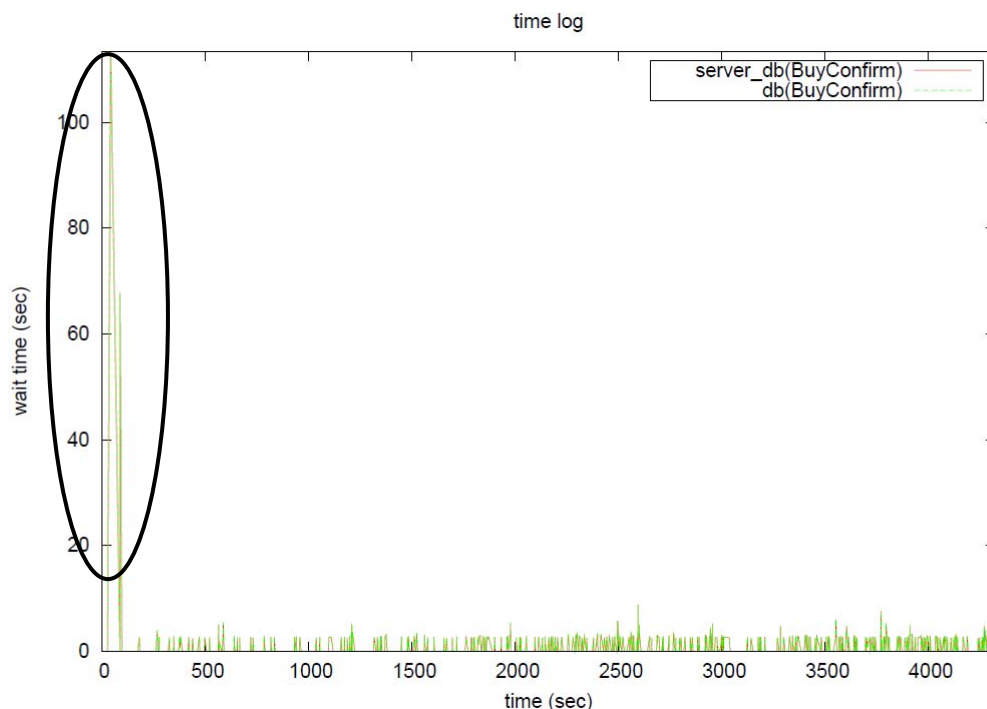


図 3.4-1 BuyConfirm インタラクションの処理時間

上記の測定時には、DBT-1 を実行する時には過去の実行の影響を受けないように、必ずシステムを再起動して実行していた。そこで、再起動せずに再度DBT-1 を実行したところ、上記現象は現れないことが分かった。その時のtime.logを元に作成したBuyConfirmの処理

時間のグラフを図 3.4-2に示す。

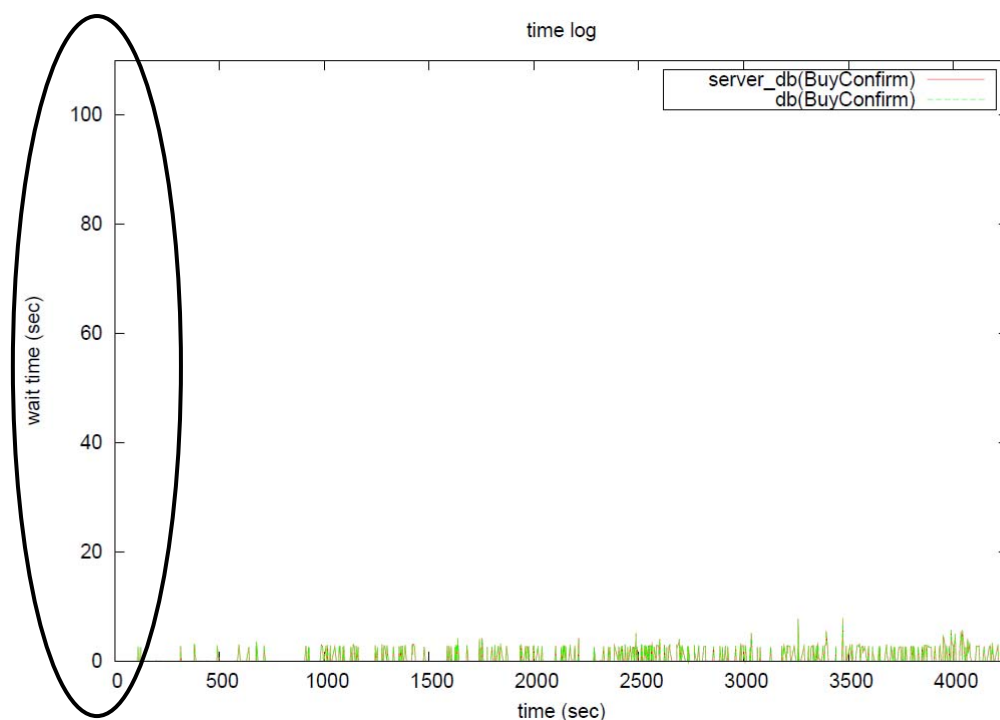


図 3.4-2 DBT-1 再実行時の BuyConfirm の処理時間

DBT-1 の 1 回目と 2 回目の実行時の状態に大きな違いがあるとすれば、DB のデータのキャッシング状態ではないかと考えた。つまり 2 回連続して実行した時には、必要となる DB のデータが既にメモリ上にあるため I/O 時間が削減され、ランプアップ時の異常現象が現れなかったと考えられる。

そこで、ランプアップ時の異常現象の発生はキャッシング処理が影響しているのではないかと予想し、さらなる詳細調査を LKST を用いて行なった。

3.4.1.1 LKST によるログの採取方法

LKSTによるログの採取はrampupマスクセット（表 3.2-4参照）のみからなるマスクセットシリーズにて行なった。採取箇所はランプアップ開始から 240 秒間である（図 3.4-3）。

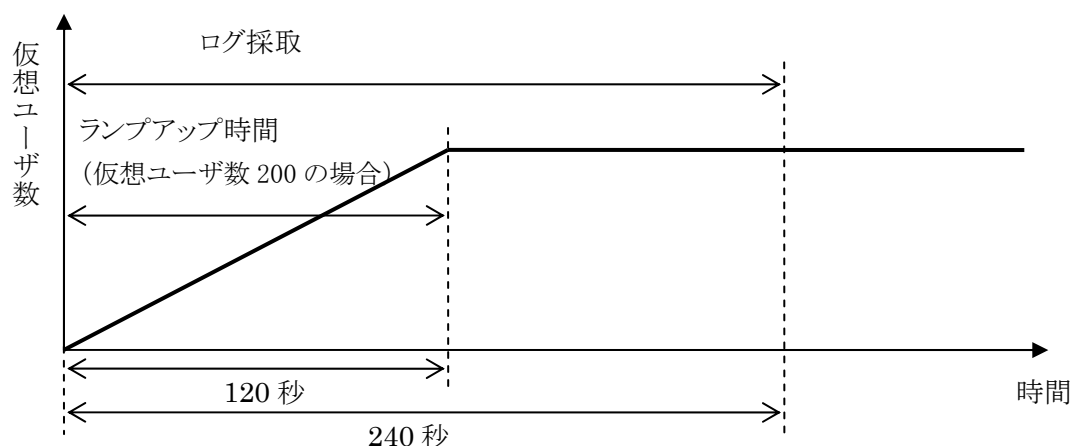


図 3.4-3 ランプアップ時の分析における LKST のログの採取箇所

また今回の測定では、1つのマスクセットで複数種類のイベントを長時間採取する必要があるため、サイズの大きいイベントバッファが必要である。そこで 60MB¹のイベントバッファを 4 つ連結し、合計 240MB²のバッファを作成した。以下にバッファの作成手順を示す。

- (1) 60MB のバッファ (ID 1) を作成し、次バッファに ID 2 を指定する

```
lkstbuf create -s 60M -b 1 -n 2
```

- (2) 60MB のバッファ (ID 2) を作成し、次バッファに ID 3 を指定する

```
lkstbuf create -s 60M -b 2 -n 3
```

- (3) 60MB のバッファ (ID 3) を作成し、次バッファに ID 4 を指定する

```
lkstbuf create -s 60M -b 3 -n 4
```

- (4) 60MB のバッファ (ID 4) を作成し、次バッファに ID 1 を指定する

```
lkstbuf create -s 60M -b 4 -n 1
```

- (5) 最初に使用するバッファを ID 1 のバッファにする

```
lkstbuf jump -b 1
```

¹ カーネルの制約により、一つのイベントバッファのサイズは 64MB が上限である。

² イベントバッファは vmalloc 領域と呼ばれるメモリ領域から確保しているが、i386 アーキテクチャでは vmalloc 領域は 100MB 強しかないため、240MB のバッファを作成することはできないので注意のこと。

3.4.1.2 time.log より得られる BuyConfirm の処理時間

time.logの内容から、非常に多くの処理時間がかかっているBuyConfirmがあることがわかっている。そこでまず、time.logを用いて、ランプアップ開始からの240秒間において、BuyConfirmの処理時間をグラフ化したものを図3.4-4に示す。80秒付近と105秒付近で発生したBuyConfirmは極端に長い処理時間を要していることがわかる。またこの2つのBuyConfirmは共に190秒付近で完了していることも読み取れる。

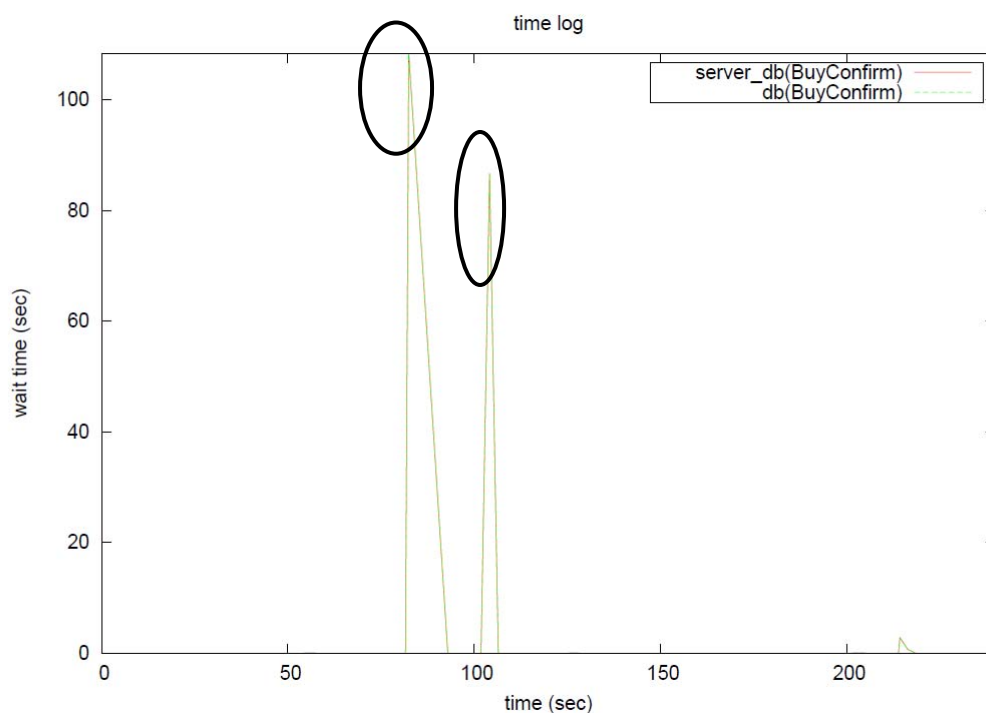


図 3.4-4 ランプアップ開始 240 秒間の BuyConfirm の処理時間

3.4.1.3 ディスクドライブ別の I/O 要求処理の分析

BuyConfirmはその処理の特性からI/Oを多く発行していると考えられる。そこでLKSTのbdevtimeアナライザを用いて、3.4.1.2項と同く、ランプアップ開始からの240秒間のI/O要求処理の状況を解析することにした。

bdevtimeアナライザは、デバイスドライバがI/O要求をキューから取り出してから、そのI/Oが終了するまでの時間を、I/O要求キュー別に得ることができる。Linux 2.4カーネルではI/O要求キューはディスクドライブごとに存在するため、ディスクドライブ別のI/O処理として見ることもできる。

bdevtimeアナライザによる解析結果を図3.4-5に示す。横軸はランプアップ開始からの経過時間、縦軸はI/O要求の処理時間である。また図中の+印、×印、*印はそれぞれ別のディスクドライブへのアクセスを意味する。+はDBのデータボリュームのみ置いているドライブ、×はOSやMaxDB、DBT-1などを置いているドライブ、*はDBのログボリュームのみを置いているドライブである。

グラフより、80秒から190秒くらいにかけて、DBのデータボリュームに対して、非常に頻繁にI/Oが行なわれていることがわかる。

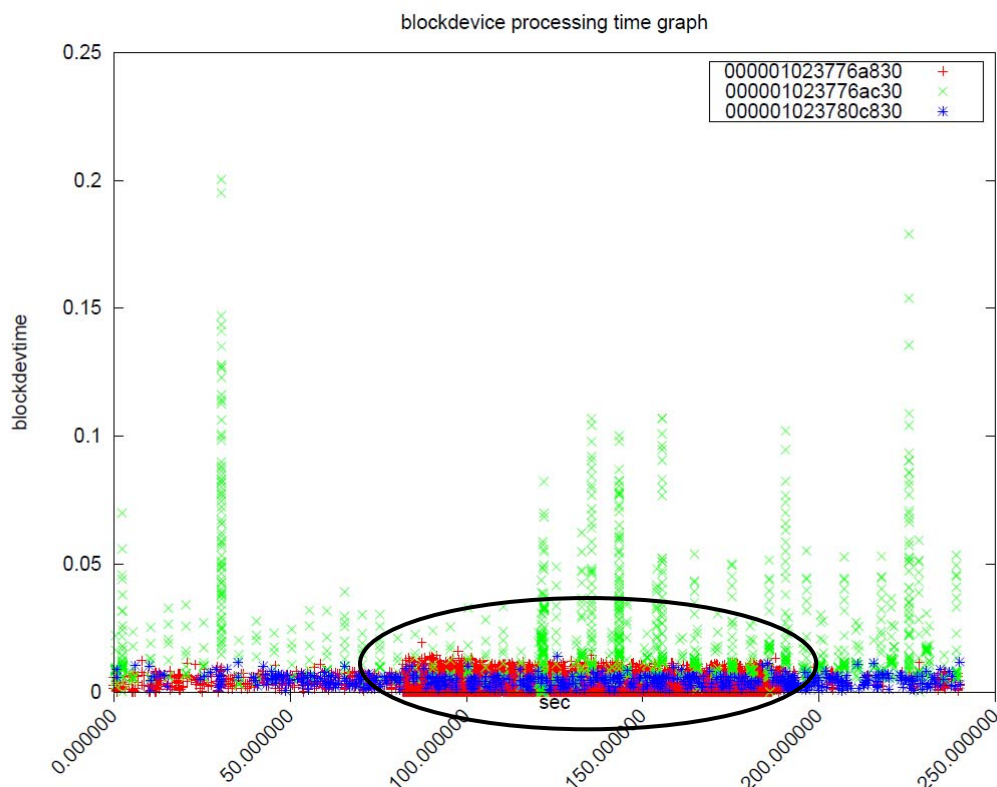


図 3.4-5 ランプアップ開始 240 秒間のデバイスドライバでの処理時間

3.4.1.4 read/write 別の I/O 処理の分析

LKST の I/O 処理のアナライザには、上記の bdevtime の他、もっと上層部からの処理性能を測定する buffer アナライザがある。buffer アナライザはバッファヘッドが生成されてから、そのバッファヘッドが指すバッファに対する I/O が完了するまでの時間を、read/write 別に得ることができる。なお、バッファヘッドとは、I/O を行なう際に使用するバッファを管理するためのカーネル内のデータ構造である。

buffer アナライザによる解析結果を図 3.4-6 に示す。図の横軸はランプアップ開始からの経過時間 (秒)、縦軸はバッファヘッドの I/O 時間 (秒) である。グラフより、80 秒から 190 秒にかけて、頻繁に読み込み用のバッファヘッドが作成されていることがわかる。この頻繁に読み込みを行なっている区間は bdevtime の解析結果 (図 3.4-5) と一致するため、データに不整合はないと言える。この状況から、この時間帯に大量の読み込み処理を行っており、その目的はキャッシングのためと推測できる。

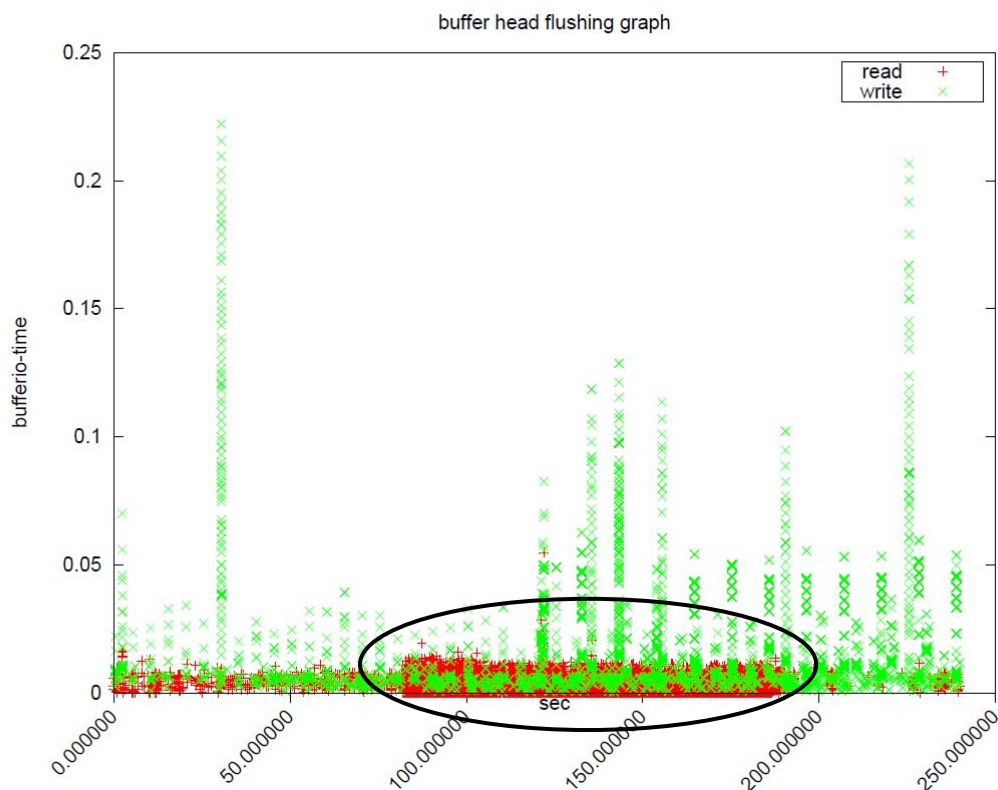


図 3.4-6 ランプアップ開始 240 秒間のバッファヘッドの処理時間

3.4.1.5 パーティション内のアクセス位置の分析

iosector アナライザは何番のセクタに対し I/O を行なったか、すなわちパーティション内のどの位置に対し I/O を行なったか解析することができる。前のアクセス位置との距離が離れているとシーク時間がかかり、処理性能に影響する。そこで、対象時間帯の I/O がどのようなセクタ位置にアクセスしたかを、iosector アナライザを用いて調査した。

iosector アナライザによる解析結果を図 3.4-7 に示す。図の横軸はランプアップ開始からの経過時間 (秒)、縦軸はセクタ番号である。また、ここでは lkstla のフィルタリング機能を用い、DB のボリュームが置かれたディスクドライブに対する I/O のみ抽出した。

図 3.4-7 を見ると、80 秒から 190 秒にかけて順次アクセスを行なっている様子が良くわかる。この時間帯は図 3.4-6 における read が大量発生している時間帯と一致するため、この順次アクセスは read 処理であると推測できる。またこの read はセクタ 0 から 2000000 の辺りに対して行なわれているが、1 セクタは 512B であるので、パーティションの先頭部分の約 1GB に対するものだといえる (1GB=2000000 セクタ*512B)。DB のデータボリュームはパーティションの先頭に配置しているため、この時は DB のデータボリュームの先頭約 1GB に対し順次読み込みを行なっていると考えられる。

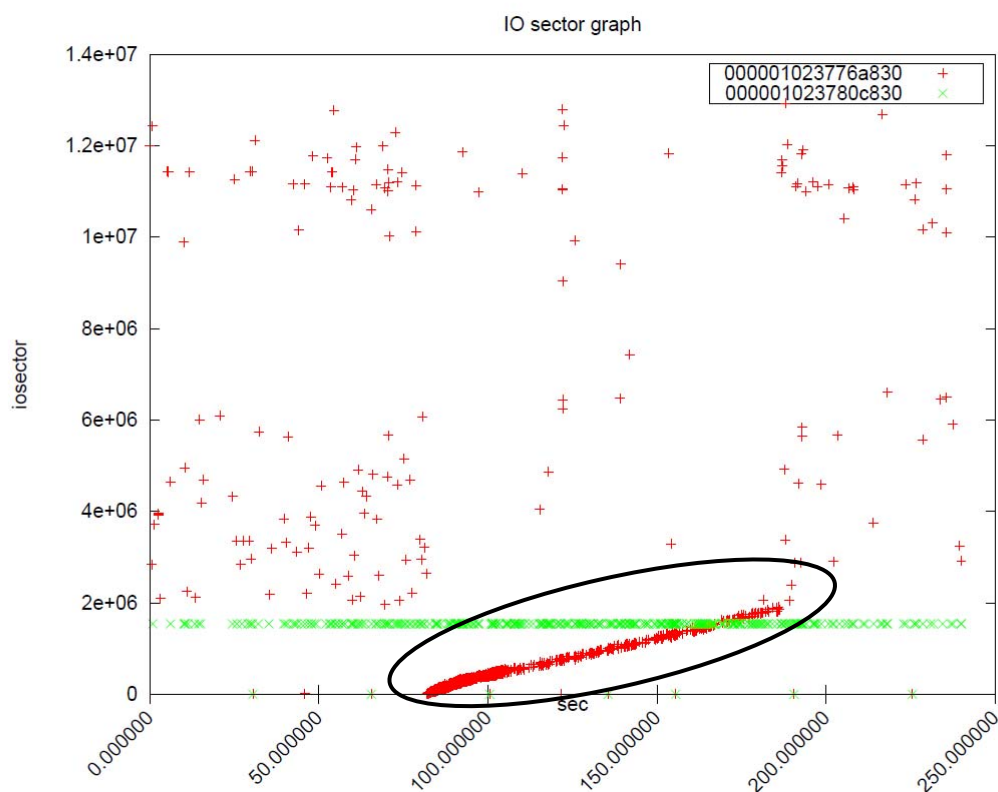


図 3.4-7 ランプアップ開始 240 秒にアクセスしたセクタ

3.4.1.6 評価と考察

極端に処理時間が長い 2 つの BuyConfirm がランプアップ開始 80 秒から 190 秒の区間に生じている、という現象に関して、LKST で解析を行った結果、以下のことが判明した。

- (1) BuyConfirm の処理時間帯に I/O 処理が多発している
- (2) I/O 処理内容は、DB のボリュームの先頭 1GB に対して大量の順次読み込みを行うものである
- (3) 順次読み込みの完了は BuyConfirm の処理完了とほぼ一致する

これらの情報から、今回の状況を以下のように推察できる。

- (a) DB はランプアップ期間中に、データボリューム中の何らかのデータの一括読み込みを開始する。
- (b) (a) は連続したデータをアクセスしている。このため、データベース処理に必要なデータをキャッシュに載せようとしているのではないかと考えられる。
- (c) さらに、同時刻に発行される BuyConfirm は、(a) の処理の完了を待つため処理完了までに長時間かかることがある。
- (d) 上記が DB や OS の、キャッシング処理の問題なのか、スケジューラの問題なのか、排他制御の問題なのか、それ以外の要因なのかは現在までのデータでは解析できなかった。先頭の 1GB にどのようなデータがあるのかがわかれば、もう少し状況を推測できると考える。

上記の評価から、LKST を用いることにより、DB サーバ実行時に現れる現象を、より詳細に解析できることがわかった。この結果はシステムの性能改善を行う上での有用なヒントとして使うことができると考える。

3.4.2 DBT-1 平常時の状況

LKST を利用して、DBT-1 の平常時の動作解析を行った。なお、本解析も 64 ビット環境で行った。

waittime アナライザは、プロセスがウェイトキュー上で待っている時間を解析できる。BuyConfirm を処理する中で、何らかの大きな待ちが発生している可能性があるため、waittime アナライザによる分析を試みた。ここでは、今回開発したプロセス名によるフィルタリング機能を用いて、プロセス毎の待ち時間情報を取得する。対象とするプロセスは DBT-1 のプロセス dbdriver および appServer、MaxDB のプロセス vserver および kernel の 4 種のプロセスである。

3.4.2.1 dbdriver における待ち時間

図 3.4-8は仮想ユーザ数 200、使用マスクセットはwaitqueue、ランプアップ後 0 分の時のdbdriverの待ち時間の様子である。図の横軸はランプアップ開始からの経過時間（秒）、縦軸はウェイトキュー上での待ち時間（秒）である。dbdriverは仮想ユーザ数のスレッドからなっているが、それら全てのスレッドにおける待ち時間がグラフ上にプロットされている。

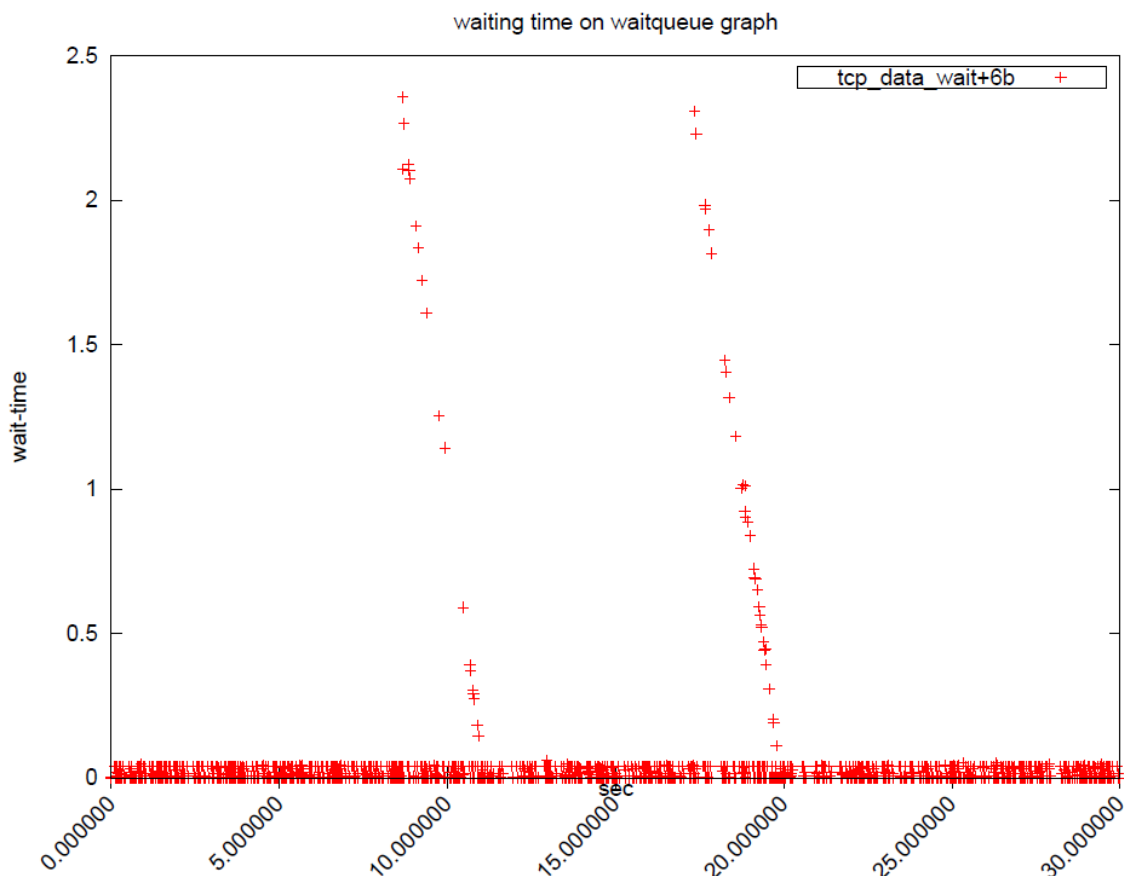


図 3.4-8 仮想ユーザ数 200 時の dbdriver における待ち時間

まず dbdriver はカーネル内の tcp_data_wait 関数内で頻繁に待ち状態になっていることが分かる。これは TCP パケットの到着を待っている時間を意味している。dbdriver が TCP パケットの到着を待つとすれば、その相手は appServer であり、インタラクションの応答待ちを行っていると推測できる。

仮想ユーザ数 200 の時など比較的負荷が小さい状況下では、ほとんどの場合においてパケットの到着待ちは短時間で終了する。しかし図 3.4-8を見ると 10 秒付近、20 秒付近で、極端に大きな待ち時間が発生しているのが分かる。この大きな待ち時間は間隔が約 10 秒なので、10 秒間隔で実行しているデータ採集コマンド(sar, top, iostat, x_consなど)が原因ではないかと予想した。これらデータ採集用のコマンドはDBT-1 のcollect_data.shスクリプトで起動をかけている。そこでcollect_data.shを実行せずにDBT-1 を動作させ、再度LKST

によるログの採取を行なった。

3.4.2.2 データ採集無しの場合の dbdriver における待ち時間

collect_data.shによるデータ採集を実行せずにDBT-1 を実行し、LKSTによりログの採取を行なった。マスクセットはwaitqueueマスクセットを使用し、このマスクセットによるログの採取期間を先ほどの30秒から90秒へと延長した。ランプアップ後15分後の、dbdriverにおける待ち時間グラフを図3.4-9に示す。このグラフを見ると、時折発生する大きな待ち時間は依然として発生しており、collect_data.shはその原因ではないことが分かった。

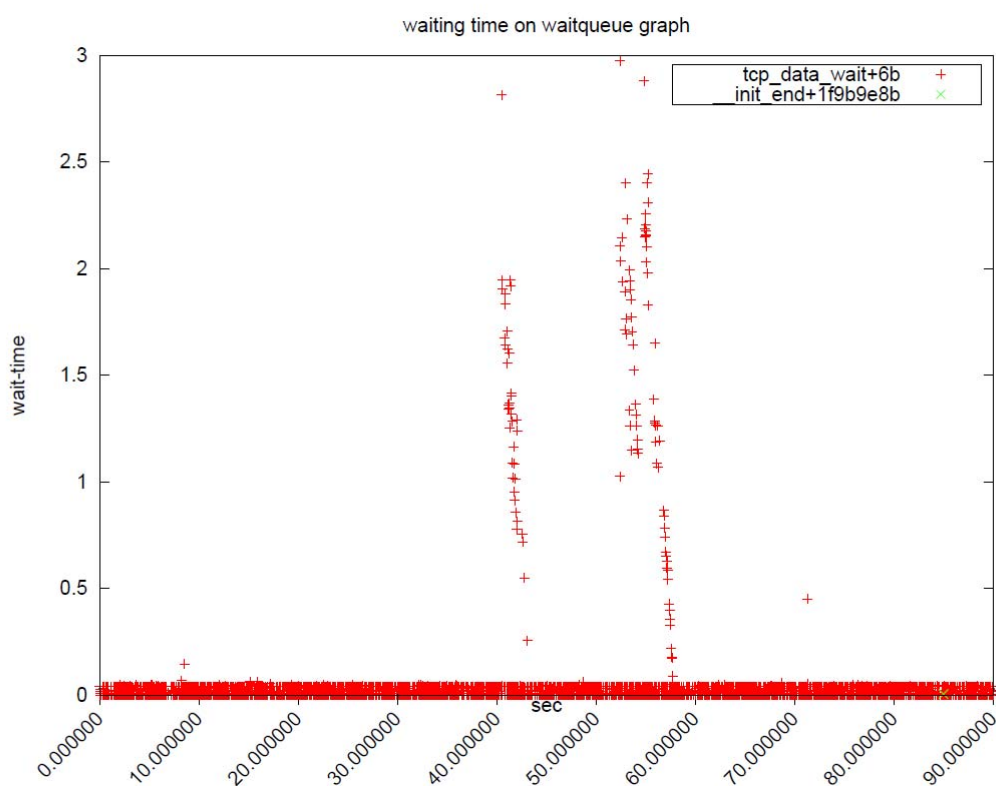


図 3.4-9 データ採集無しの時の dbdriver における待ち時間

次に同じ時間帯のvserverにおける待ち時間グラフを図3.4-10に示す。vserverもdbdriverと同様、TCPパケットの到着を待っている。vserverの通信あいてはappServerであるため、このグラフの点はappServerからのトランザクションの到着を待っている時間であると考えられる。横軸上にプロットされた点に注目すると、3箇所ほど1秒程度の空白時間があることに気付いた。これは図3.4-9における大きな待ち時間の発生箇所にはほぼ一致する。この空白は、その区間、DBは新たなトランザクションを受け付けようとしていない、ということの意味する。この状況はDBの性能に影響を与えている可能性も高く、少なくとも一時的な応答性低下の発生につながっていると考えられる。

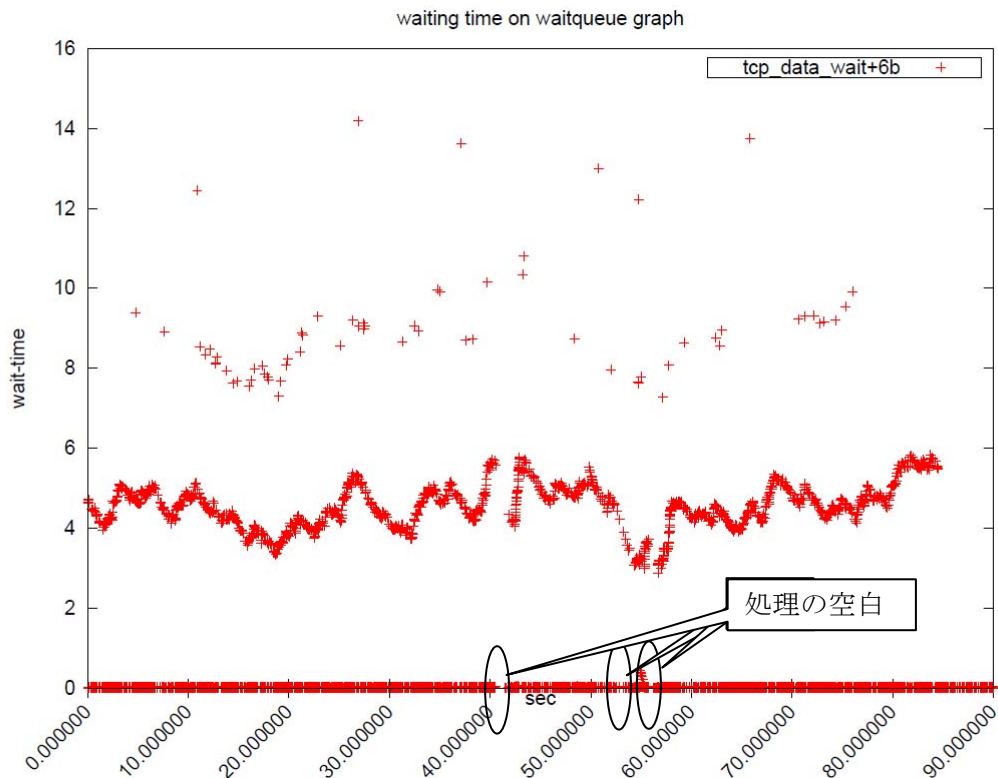


図 3.4-10 データ収集無しの時の vservers における待ち時間

3.4.2.3 DBT-1 のログとのつき合わせ

DBT-1 のプログラム appServer は time.log というログを出力している。time.log には各インタラクションについて、DB での処理時間および DB での処理時間を含めた appServer の処理時間が記録されている。このログ出力機能に対し、インタラクションの発生時刻も併せて出力するようプログラムを修正し、LKST のログとの対応付けをできるようにした。図 3.4-9 における大きな待ち時間が発生している時間帯でどんなインタラクションが発生しているか、time.log より調べた。図 3.4-9 の 41 秒付近に相当する time.log の内容を図 3.4-11 に示す。各項目の意味は以下の通り。

- リストの左端の 2 文字の文字列：インタラクションの略号
- start：インタラクション発生時刻
- total：dbdriver からリクエストを受け取ってからレスポンスを返すまでの時間
- db：DB にトランザクションを発行してから結果を受け取るまでの時間
- server_db：dbdriver からのリクエストをトランザクションキューにエンキューしてからレスポンスを返すまでの時間

図 3.4-11 を見ると、大きな処理遅延が生じる最初のインタラクションは BuyConfirm であ

ることが分かる。さらに他の遅延発生箇所についてtime.logを調査すると、どれもBuyConfirmインタラクシオンが遅延のきっかけになっていることが分かった。また後続のインタラクシオンもBuyConfirmの遅延に引きずられ、大きく処理時間がかかっている。

H0, start 1127392022.170000, total 0.000000, db 0.000000, server_db_time 0.000000	
PD, start 1127392022.170000, total 0.000000, db 0.000000, server_db_time 0.000000	
SU, start 1127392022.170000, total 0.010000, db 0.010000, server_db_time 0.010000	
BR, start 1127392022.500000, total 0.040000, db 0.040000, server_db_time 0.040000	
BC, start 1127392022.510000, total 2.810000, db 2.810000, server_db_time 2.810000	図 3.4-9 の 41 秒付近
SC, start 1127392022.530000, total 1.940000, db 1.940000, server_db_time 1.940000	
SC, start 1127392022.580000, total 1.900000, db 1.900000, server_db_time 1.900000	
H0, start 1127392022.820000, total 1.670000, db 1.670000, server_db_time 1.670000	
BS, start 1127392022.850000, total 1.880000, db 1.880000, server_db_time 1.880000	
PD, start 1127392022.860000, total 1.640000, db 1.640000, server_db_time 1.640000	
NP, start 1127392022.870000, total 1.830000, db 1.830000, server_db_time 1.830000	
PD, start 1127392023.030000, total 1.550000, db 1.550000, server_db_time 1.550000	
NP, start 1127392023.080000, total 1.700000, db 1.700000, server_db_time 1.700000	
SU, start 1127392023.120000, total 1.620000, db 1.620000, server_db_time 1.620000	
H0, start 1127392023.140000, total 1.360000, db 1.360000, server_db_time 1.360000	
H0, start 1127392023.140000, total 1.360000, db 1.360000, server_db_time 1.360000	

図 3.4-11 大きな遅延の発生箇所の time.log の内容

3.4.2.4 遅延発生時に発行されたシステムコールの分析

次に BuyConfirm 実行時にどのようなカーネルの処理が行われているのか、その処理が BuyConfirm の遅延に影響を与えるものなのかを、LKST を用いて調査した。評価には wqsyscall マスクセットを用いた。wqsyscall マスクセットは待ち時間解析用のイベントとシステムコール解析用イベントの両方を記録するものであり、ウェイトキューによる待ちとシステムコールの発行の関連性を調査することができる。

まずdbdriverの待ち時間グラフを図 3.4-12に示す。23 秒付近で大きな処理時間がかかっているのが分かる。さらに同じ時間帯にvserverが実行しているシステムコールの処理時間のグラフを図 3.4-13に示す。図の横軸はランプアップ開始からの経過時間 (秒)、縦軸はシステムコール処理時間 (秒) である。図 3.4-13の 23 秒付近では複数のsemopシステムコールで大きな処理時間がかかっており、何らかの大きなロック (セマフォ) の解放を待っていると判断できる。

ちなみにセマフォによる待ちは、ウェイトキューではなくセマフォ独自のキューで待つことになるため、waittimeアナライザの結果にはその待ちの様子が現れない。この情報から、図 3.4-10の 41 秒付近に現れた空白の時間帯は、セマフォによる待ちを行なっているといえる。

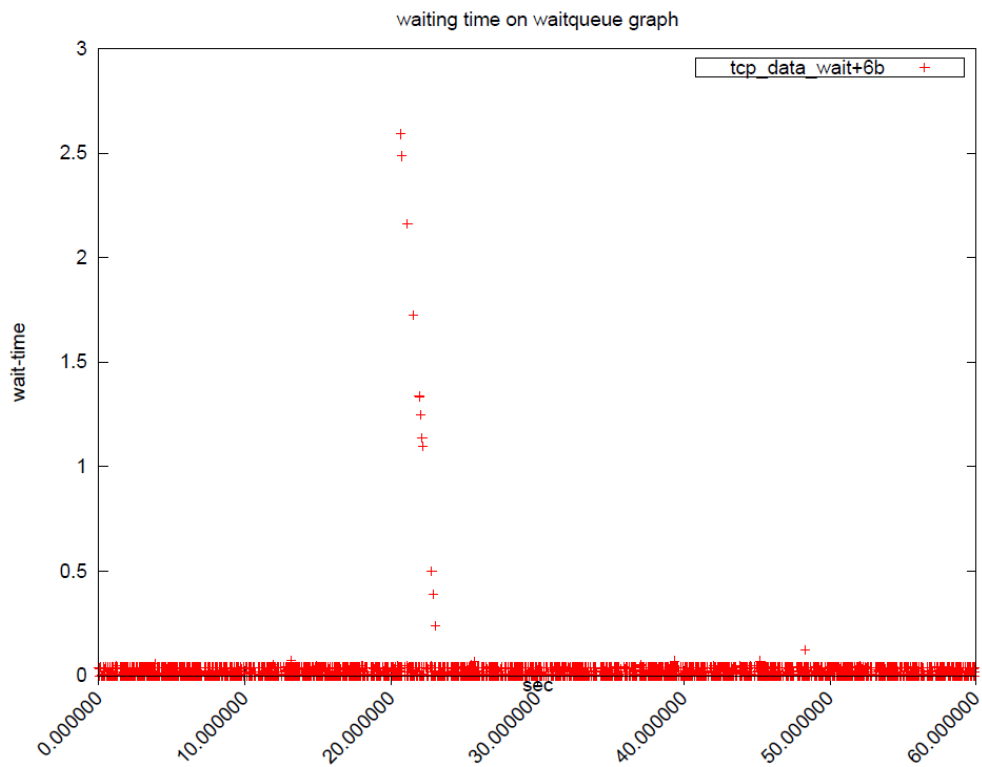


図 3.4-12 dbdriver における待ち時間

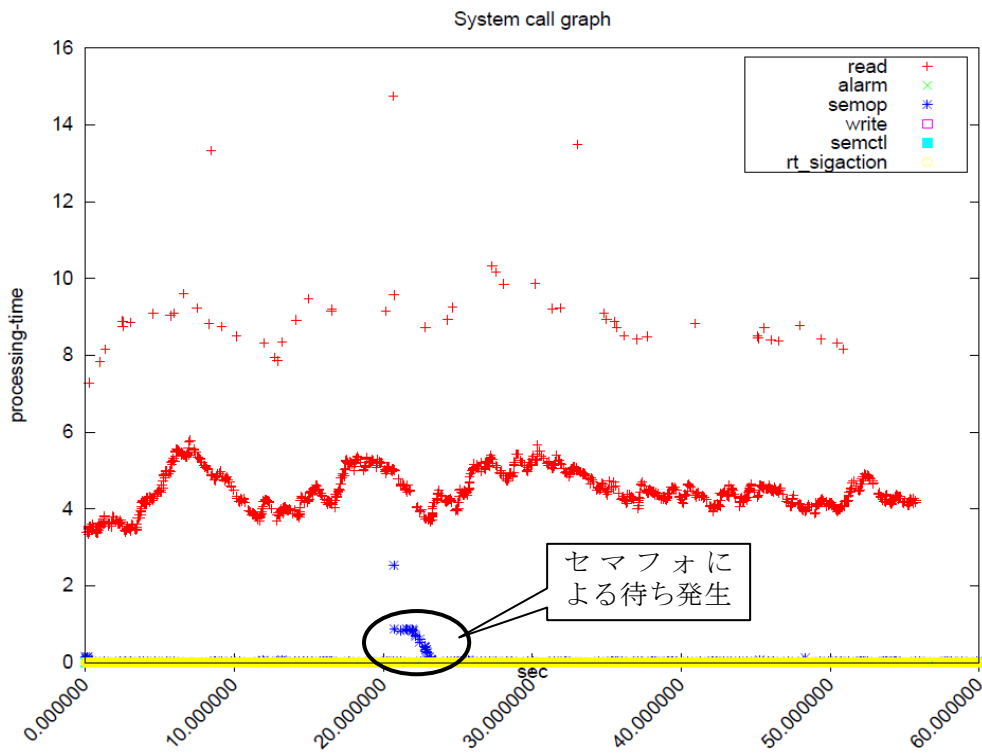


図 3.4-13 vsrver におけるシステムコール処理時間

3.4.2.5 評価と考察

これまでの分析により以下の点が分かった。

- (1) dbdriver の各スレッドは appServer からの応答を待っているが、appServer の応答性が極端に悪くなり、長時間またされる時間帯が存在する
- (2) 上記時間帯において、vserver 内ではセマフォによる大きな待ち時間が連続して発生している
- (3) 上記時間帯では BuyConfirm の処理が滞っており、それに引きずられて他のインタラクションの処理も滞る

これらの情報から、今回の状況を以下のように推察できる。

- (1) appServer の応答性が悪い時にカーネル内でセマフォによる待ちが発生している。このために appServer の処理が遅れ、結果として dbdriver の各スレッドの処理も遅れる。
- (2) BuyConfirm の処理時間は上記に大きく影響を受けていると推測される。
- (3) BuyConfirm の処理遅延の影響を受けて、遅延するインタラクションがある。
- (4) 上記が関連する一連の処理であるために待ちが発生しているのか、DB や OS の排他制御の問題なのか、それ以外の要因なのかは不明。セマフォの原因を解明できれば、もう少し状況を推測できると考える。

この例では LKST を用いることにより、DB サーバ実行時に現れる現象そのものだけでなく、その現象に影響を与えていると思われるイベントを知ることができ、より詳細に解析できることがわかった。今回の結果は DBT-1 の仕様を再検討する時などに、有用なヒントとして使うことができる。また、さらに評価を行う時に今回の評価ノウハウは役立つと考える。

4 LKST 性能評価機能のオーバヘッド評価

4.1 評価環境

4.1.1 システム構成

評価を行ったシステム環境を表 4.1-1に示す。

表 4.1-1 評価を行うシステム環境

項番	項目	評価環境	
1	ハードウェア	CPU	Intel Xeon 3.20GHz× 2
2		メモリ	8GB
3		ハードディスク	SCSI 140GB
4	ソフトウェア	カーネル	UpstreamKernel-2.6.9(64ビット) + 新LKST
5		OS	Fedora Core 3
6		解析ツール	Lkstla
7		負荷ツール	dbench, lmbench, tiobench
8		ファイルシステム	ext3

4.1.2 ベンチマーク

LKST 性能評価機能を組み込んだことによるカーネルの性能に対する影響を調査するため、以下のベンチマークを利用してオーバヘッドの計測を実施した。

(a) dbench-3.03

<ftp://ftp.samba.org/pub/tridge/dbench/dbench-3.03.tar.gz>

(b) lmbench-3.0-a4

<http://www.bitmover.com/lmbench/>

(c) tiobench-0.3.3

<http://sourceforge.net/projects/tiobench/>

4.1.3 評価パターン

それぞれのベンチマークにおいて、表 4.1-2の 5 パターンの設定を行って計測を実施した。それぞれのベンチマークは各 4 回ずつ実行し、平均値を求めて比較を行なった。

表 4.1-2 OS のパターン

項番	名称	概要
1	pure	LKST の機能が組み込まれていないカーネルを利用して計測
2	LKST norec	LKST の機能が組み込まれたカーネルを利用し、全てのフックポイントを無効にした状態で計測
3	LKST IPA	LKST 性能評価機能で利用されるフックポイントのみを有効にした状態で計測
4	LKST nolock	LKST 性能評価機能で利用されるフックポイントのうち、スピンロック関連のフックポイントを無効にし、その他のフックポイントを有効にした状態で計測
5	LKST all	LKST で記録可能な全てのフックポイントを有効にした状態で計測

4.1.4 LKST のセットアップ

表 4.1-2の項番 2～5 では次の手順でLKSTのセットアップを行なった。

- (1) lkst モジュールをカーネルにインストールする

```
# modprobe lkst
```

- (2) LKST 制御用のデバイスファイル/dev/lkst を作成する(2.6 カーネルの場合)

```
# MAKEDEV lkst
```

- (3) 10MB のバッファを作成する

```
# lkstbuf create -s 10M
```

- (4) 作成したバッファを使用するよう設定する

```
# lkstbuf jump -b 1
```

- (5) デフォルトのバッファを削除する

```
# lkstbuf del -b 0
```

- (6) マスクセットの設定を行なう

- (a) 項番 2 の場合

```
# lkst stop
```

- (b) 項番 3～5 の場合 (maskset はマスクセットの定義ファイル)

```
# lkstm write -S < maskset
```

なおイベントの記録には、デフォルトのイベントハンドラを使用する。

4.2 評価結果

各ベンチマークによる性能測定結果をグラフにまとめた。それぞれのグラフは、LKST 機能を含まないカーネルによる測定結果(pure)を基準値として1とし、その他の項目は pure に対する相対値で表している。

4.2.1 dbench

dbench は、Samba サーバが発行する I/O 要求を模してローカルファイルシステムのスループットを計測することができるベンチマークである。また、複数のクライアントからの同時接続を想定した負荷を発生させることができる。

4.2.1.1 測定条件

以下のコマンドラインオプションで実行した。

```
dbench -D /mnt/bench 1 -c ./client.txt
```

/mnt/bench は作業用のディレクトリであり、ext3 のクリーンなパーティションをマウントしている。client.txt は dbench パッケージに含まれる負荷パターンである。

4.2.1.2 測定結果

dbench-3.03 によるベンチマークの計測結果を図 4.2-1に示す。計測結果はスループットを表し、数値が大きいほど性能が高いことを表す。

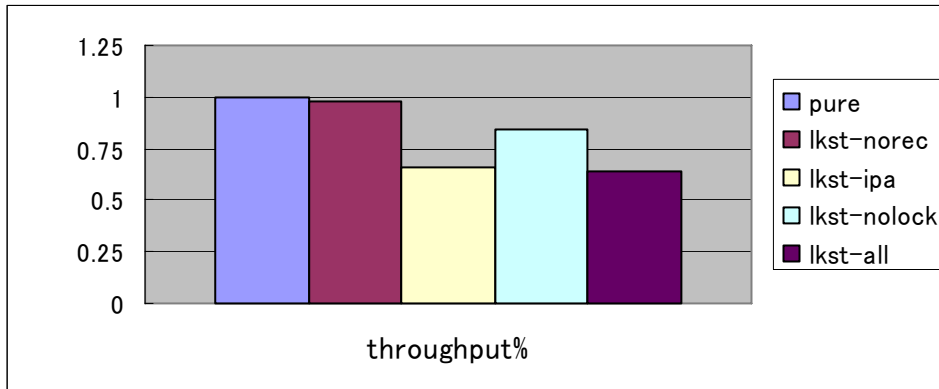


図 4.2-1 dbench の測定結果の比較（比率）

前回同様、何もイベントを採取しない場合でもわずかなオーバーヘッドが存在することがわかる。また、スピンロック関係のイベント採取のオーバーヘッドが大きいこともわかる。

4.2.2 Lmbench

Lmbench は、UNIX の基本機能の性能をはかるためのマイクロ・ベンチマークである。Lmbench を利用することで、次のような機能のベンチマークを行うことができる。

- (1) メモリ管理
- (2) プロセス管理
- (3) ネットワーク
- (4) シグナル処理
- (5) ファイル操作
- (6) システムコール

4.2.2.1 測定条件

Lmbenchは初回実行時、ベンチマークのコンフィギュレーションをインタラクティブに行なう。コンフィギュレーション実行時の問いと入力内容を図 4.2-2に示す。太字の部分が入力内容である。

```

MULTIPLE COPIES [default 1] 1
MB [default 5581] 2000
SUBSET (ALL|HARWARE|OS|DEVELOPMENT) [default all] OS
SLOWFS [default no] no
REMOTE [default none] (改行のみ)
Processor mhz [default 3191 MHz, 0.3134 nanosec clock] (改行のみ)
FSDIR [default /usr/tmp] /mnt/bench
Status output file [default /dev/tty] /dev/null
Mail results [default yes] no

```

図 4.2-2 LMBench のコンフィギュレーション

ここで、/mnt/bench は作業用のディレクトリであり、ext3 用のパーティションをマウントしている。

4.2.2.2 測定結果

lmbench-3.0-a4 を利用したベンチマークの測定結果を図 4.2-3から図 4.2-7 に示す。値はpureの測定値を 1 とした比率であり、値が大きい程、オーバーヘッドが大きいことを示している。

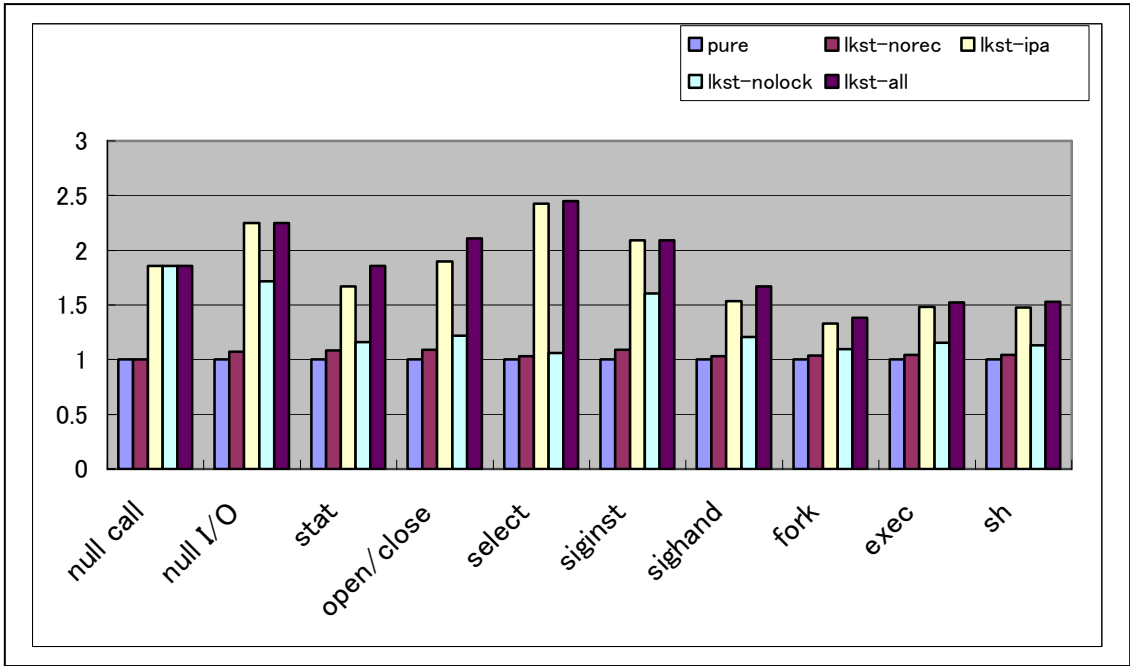


図 4.2-3 LMBench の測定結果(システムコール、シグナル等)の比較 (比率)

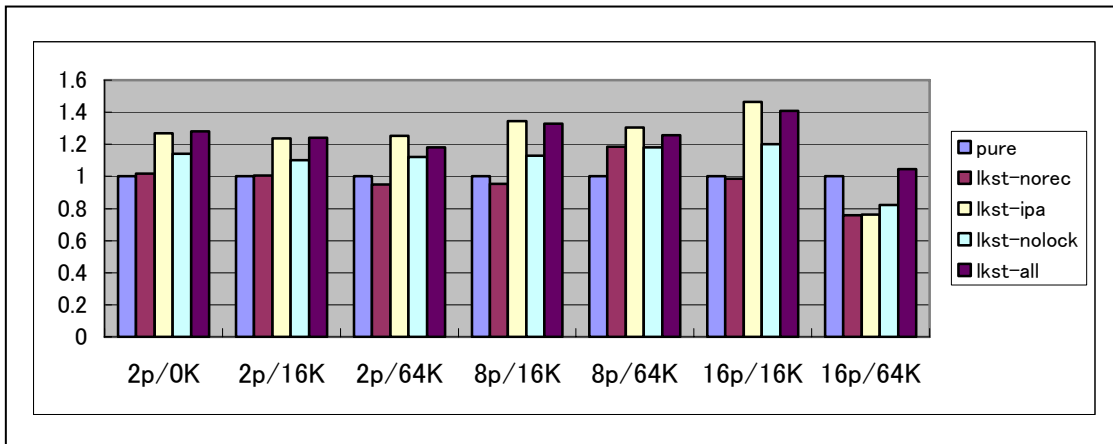


図 4.2-4 LMBench の測定結果(コンテキストスイッチ)の比較 (比率)

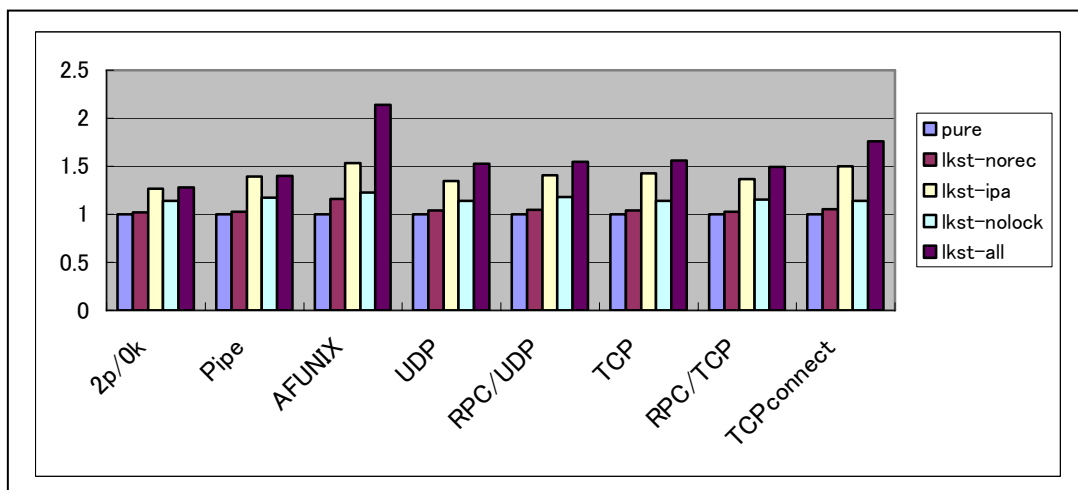


図 4.2-5 LMBench の測定結果(ネットワーク等)の比較 (比率)

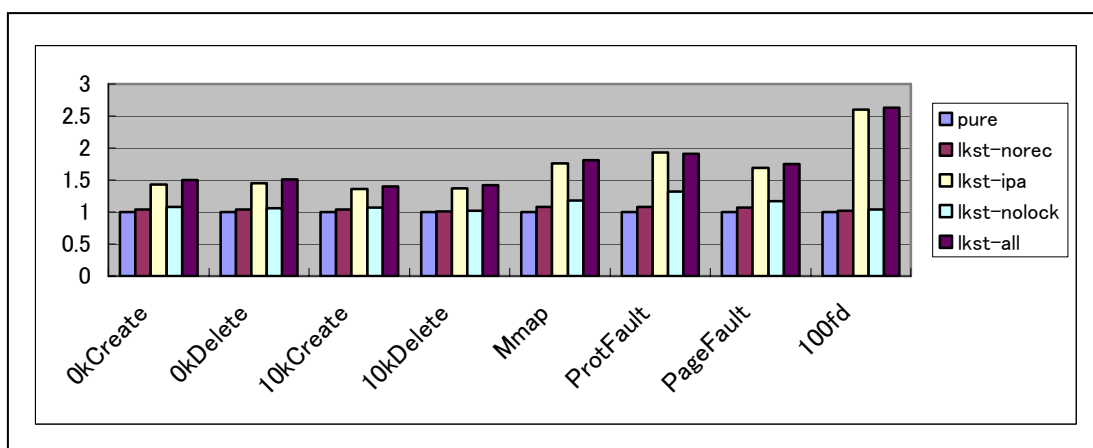


図 4.2-6 LMBench の測定結果(ファイル操作、VM 性能等)の比較 (比率)

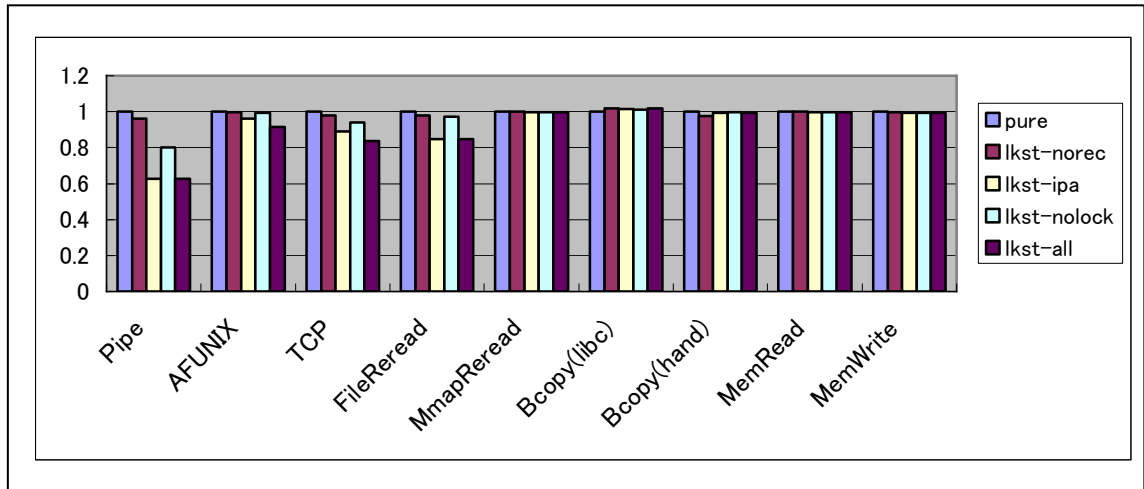


図 4.2-7 LMBench の測定結果(ローカルコミュニケーション)の比較 (比率)

オーバーヘッドの傾向はdbenchの時と同様である。また図 4.2-3のnull call、null I/Oのような短時間で終わる処理では、オーバーヘッドの大きさが目立つ。図 4.2-6の 100fdではlkst-ipaおよびlkst-allのオーバーヘッドが 2.4 倍以上と大きい。lkst-nolockのオーバーヘッドが小さいことから、スピンロックによるオーバーヘッドが大部分を占めているものと推測できる。

4.2.3 tiobench

tiobench は同時 I/O 性能を計測するためのベンチマークである。Sequential Reads、Random Reads、Sequential Writes、Random Writes の 4 パターンの同時 I/O 性能を測定することができ、同時に I/O を発行するスレッドをパラメータによって増減することができる。

4.2.3.1 測定条件

以下のコマンドラインオプションを指定して実行した。

```
tiotest -t 4 -f 256 -r 32768 -d /mnt/bench
```

このオプションでは四つのスレッドにより同時 I/O を発行する。Sequential Reads と Sequential Writes では、各スレッドは 256MB のファイルを順次アクセスする。また Random Reads と Random Writes では 32768 回のアクセスを 256MB のファイルに対して行なう。

合計ファイルサイズ(今回は 256MB×4 = 1GB)は物理メモリ量より十分大きな値になるように設定しないと、キャッシュの効果により Sequential Reads と Random Reads が一瞬で完了してしまう。ベンチマークが短時間で完了すると結果が不正確になる既知の問題があるため、注意が必要である。今回の測定ではカーネル起動パラメータに mem=512M を

指定し、利用可能な物理メモリサイズを 512MB に制限してベンチマークを実行した。

4.2.3.2 測定結果

tiobench-0.3.3 を利用したベンチマークの測定結果を図 4.2-8から図 4.2-11に示す。値はpureの測定値を1とした比率である。throughputおよびCPU effectは値が大きいほど、性能が良いことを意味している。CPU%は値が高いほど、オーバーヘッドが大きいことを意味している。

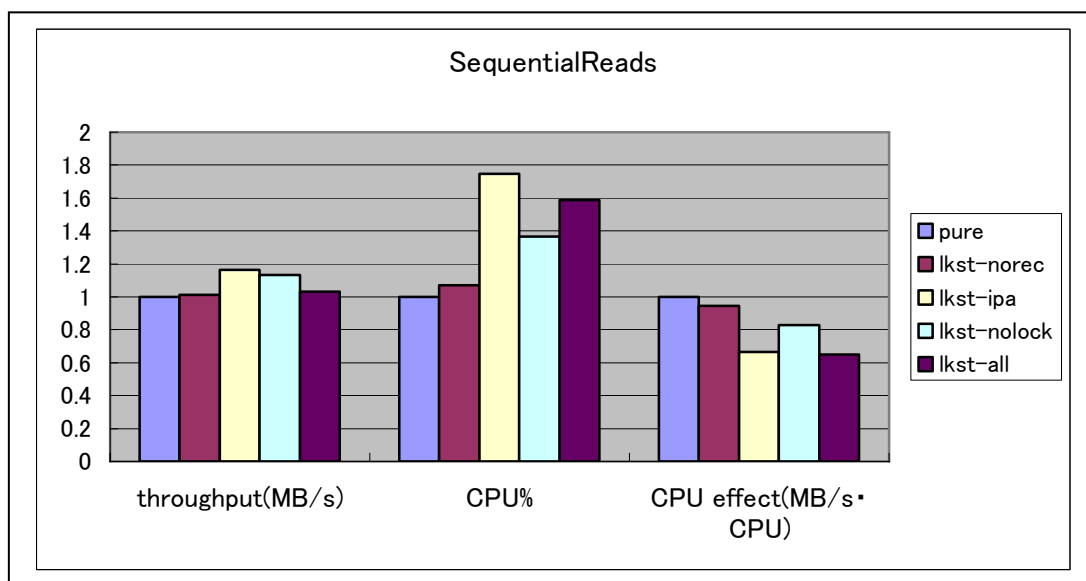


図 4.2-8 tiobench の Sequential Read の測定結果の比較 (比率)

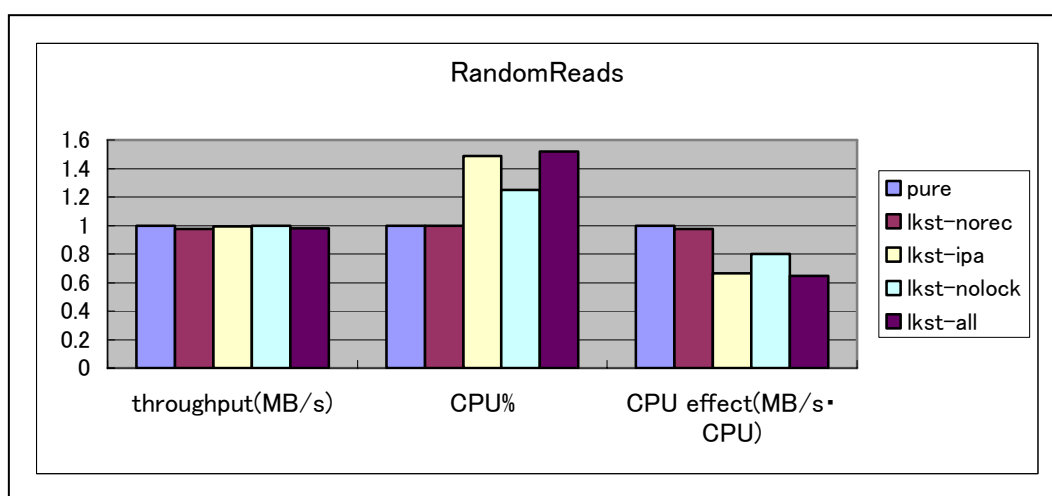


図 4.2-9 tiobench の Random Reads の測定結果の比較 (比率)

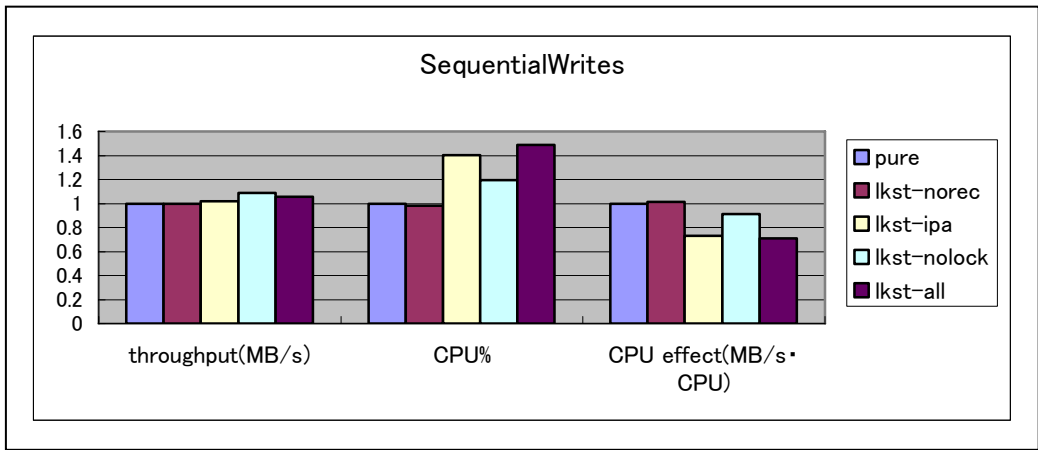


図 4.2-10 tiobench の Sequential Writes の測定結果の比較 (比率)

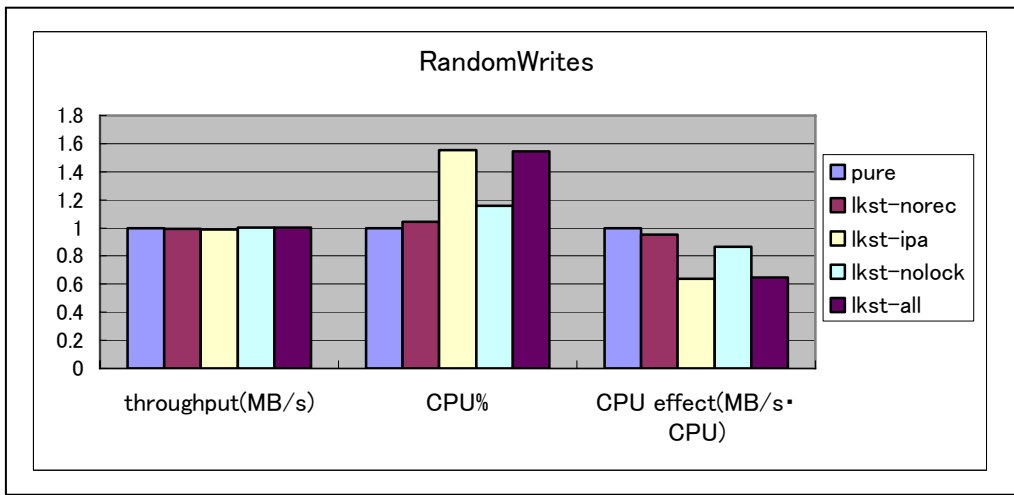


図 4.2-11 tiobench の Random Writes の測定結果の比較 (比率)

図 4.2-8と図 4.2-10 の計測結果で、スループットがpure の結果を上回る良い結果が計測されている。これはLKST のオーバーヘッド以外に、計測結果を大きく左右する要因があるためと考えられるが、今回の測定の範囲では要因の特定には至らなかった。

4.3 考察

今回のベンチマーク結果から、次のことがわかった。

- (1) LKST 性能評価機能を組み込むことで、オーバーヘッドが生じる。
- (2) LKST を組み込むことで、イベントを全く記録しない場合でも、わずかであるがオーバーヘッドが生じる。
- (3) 記録するイベントの種類を増やすことによって、オーバーヘッドは大きくなるが、ベンチマークの種類によって、オーバーヘッドは異なる。

今回実施したベンチマークは、2004 年度の評価で用いたものと同じマイクロ・ベンチマークである。これらは特定の処理やシステムコールを繰り返し、カーネル内部の同じ処理を頻繁に繰り返す種類のものである。そのためその場所にあるフックポイントに関しては、そのオーバーヘッドを明確に捕らえることができている。

そのオーバーヘッドをみると、機能が増えた分のオーバーヘッドはあるものの、2004 年度の評価 (IA-32 版) とほぼ同じような結果になっている。

今回も性能測定は5つのパターンで行った。これらには以下のような、記録するフックポイント数にバリエーションを設けるようにしている。

- ・ フックポイントの数が多しパターン (lkst_all)
- ・ 出現頻度が多いフックポイントを除いたパターン (lkst_nolock)
- ・ トレースを採らないパターン (lkst_norec)

これらの結果からも、EM64T 版でも、取得が必要なデータを絞り込み、有効にするフックポイントの数を減らすことで、オーバーヘッドを削減できることが確認できた。

5 総括

5.1 LKST の有効性

(1) 32/64 ビットにおける比較：

今回の EM64T 対応の実現により、64 ビット環境でも 32 ビット環境と同等の LKST を使ったカーネル評価を行えるようになった。特に両方の環境で動くアプリケーションがある場合には、双方で動作させた時のカーネル内性能の比較評価を行うことができ、総合性能だけではわからない負荷や処理の違いを解析することができる。例えば、今回は MaxDB 上での DBT-1 実行時の 32/64 ビット環境の性能の変化をカーネルレベルで行なうことができた。得られた結果は、そのアプリケーションとアーキテクチャとの相性を予想するための材料として使うことができる。すなわち、顧客の希望する仕様（機能・性能・負荷など）を、どのようなシステムとして構築したら良いかを判断する材料を得る手段として、LKST は有効であると考ええる。

(2) プロセスに着目した性能解析：

LKSTLogTools は、性能遅延を見つけた時に、その後に行なう原因究明に対して、調査すべき時間帯、関連するプロセスなどを絞り込むことに有効に活用できる。今回開発したプロセス名の併記機能や、特定プロセスに着目した性能情報の抽出機能を使うことにより、プロセス間の関係や役割を意識した解析を、より容易に行うことができるようになった。

(3) プロセスに着目した動作解析：

DBT-1 を実行中のカーネルの性能を評価する際にも、DBT-1 や MaxDB のプロセスとカーネルのプロセスを明確に結びつける手段がない現在、関連するプロセスの動きを追う事で状況を判断することになる。DB サーバ稼働時のカーネル評価でも、任意の時刻で DB 層のプロセスとカーネル処理を対応させ、そのカーネル処理を要求しているプロセスが他に何をしているかを解析することで状況が推測できた。この作業は試行錯誤を繰り返すため膨大な時間が必要であるが、今回開発した絞込み機能によりその時間を短縮できるようになったことは、有用であると考ええる。

(4) 選択による可視化：

今まで採取したデータの可視化を行う場合に、1 つもしくは全部のキーについて可視化するという選択肢しかなかった。今回開発した複数のキーデータの同時可視化機能を使えば、指定した複数のデータを 1 つのグラフにまとめて描画できる。これにより、いろいろな視点での比較を容易に、しかも視覚的に行うことができるため、解析の絞り込み時間が大幅に短縮できるようになった。

(5) カーネルバージョン間の比較：

開発した機能はカーネル 2.4 でもカーネル 2.6 でも動作できるようにしており、カーネル・バージョンの違いによるアプリケーション実行時のカーネル内性能比較評価を行うことができる。

5.2 性能評価ツールの開発規模

今回の開発規模を表 5.2-1に示す。

表 5.2-1 開発規模

#	項目		開発言語	ステップ数
1	母体分	LKST	C	26588
2			アセンブラ	971
3		LKSTLogTools	C	6709
4	改造分	LKST	C	9
5		LKSTLogTools	C	2423
6	新規作成分	LKST	C	0
7		LKSTLogTools	C	1486

5.3 今後の課題

5.3.1 LKST 本体の課題

(1) イベントバッファからのデータの読み取り方法の改善：

イベントバッファからデータを読み込む際に、読み取れるデータ量が実際に採取したデータより極端に少なくなることがある。この現象は、イベントバッファの先頭から最後に書き込みを行なった位置までのデータを読み取る仕様のため、丁度バッファを一周したタイミングで読み取りを行なうと発生する。したがってバッファを一周以上した場合を考慮したデータ読み取りの仕様を検討する必要がある。

(2) 読み出しオーバーヘッド低減：

イベントの情報をユーザプログラムに渡す部分で、何らかの方法でメモリコピーのオーバーヘッドを減らす必要があると考える。また、記録した情報をディスクに保存する場合、ディスク I/O が頻発している状態であると、LKST 自身の記録が待たされてしまう。これに対応するため、ネットワークを使った転送や、イベントの圧縮、あるいはカーネルデーモンを使ったディスクへの保存などを検討する必要がある。

(3) サンプルング機能：

LKST の取得するイベントによっては、詳細にすべてを記録するよりも、いくつかのサンプルングを取ったほうがデータ数を抑制しやすいものがある。イベントハンドラの拡張やフックポイントの改造などを行い、特定のイベントをサンプルングしてデータ数を抑制することが望まれる。今回、要望はあったもののまだ十分に検討ができていない。引き続き検討を進めていく必要がある。

(4) パッチ量の削減：

LKSTはカーネルにパッチを適用する必要があるが、表 5.2-1を見てもわかるようにパッチサイズが大きい。処理の見直しを図り、パッチ量の削減を推進する必要がある。

5.3.2 性能評価ツールの課題

(1) GUI ツールの開発：

LKST の設定やコマンドの引数などは数が多いため、簡単になるように工夫をしてきているが、使い慣れないとわかりにくい。インタラクティブな GUI ツールがあれば、敷居を低くすることができる。また、結果表示や結果レポートの作成機能も GUI で行うことを検討したい。

(2) イベントフィルタ・アナライザの拡充：

今回解析ツールの充実を図り、アナライザやイベントフィルタの有効性を実感した。しかし、いろいろな解析を行うにはまだ不足している。より多くのイベントフィルタリングの機能を拡充する必要があると考える。

(3) 中間データファイル生成による効率化：

現在の解析ツールは、解析結果の表示形式やフィルタを変えた場合でも、再度データの解析を実行するようになっている。データファイルが大きくなると、一度の解析に時間がかかるため、解析データの絞込みを繰り返し行うことが難しい。中間データファイルを作ることで、解析の効率化が図れると考える。

(4) アラーム機能：

現在の解析ツールは、アプリケーション実行後にデータを取得し、解析を行っているが、いくつかの解析ルーチンについては、LKST の拡張イベントハンドラとしてカーネル内部で実行することが可能である。この仕組みを利用し、アプリケーション実行中に解析を行い、何らかの警告ログを出すことも可能と考える。

(5) 解析対象時間帯の選択：

イベントバッファは CPU ごと存在しており、ある CPU 上で発生したイベントはその CPU が管理するイベントバッファに記録される。バッファの埋まる速度は CPU ごとに異なるため、それぞれのバッファに記録されているイベントの時間帯に食い違いが生じ

る。例えば CPU 1 のバッファには 1:00~1:10 に発生したイベントが記録され、CPU 2 のバッファには 1:01~1:10 のイベントが記録される、ということが起こる。この時、1:00~1:01 は CPU 1 で発生したイベント情報しかないため、正しい評価ができない。よって全ての CPU について同じ時間帯のイベント情報のみを取り出し、解析するための手段が必要である。

最後に、従来、今回の開発成果、将来的な目標について、機能面での比較を行ったものを表 5.3-1に示す。

表 5.3-1 機能面での比較

	前回の開発結果	今回の開発成果	将来的な目標
機能	LKST + lkstlogtools	LKST + lkstlogtools	強化版 LKST + 強化版 lkstlogtools + GUI
カーネルの内部情報の記録	◎	◎	◎
性能評価情報の記録	◎	◎	◎
記録した情報の解析	○	○+	◎
複合的な情報解析	△	○	◎
情報の可視化	○	○+	◎

◎：十分に行うことが出来る

○+：○の課題のうち対応済み件数が追加された

○：一部を手動で行う必要がある

△：手動で行う必要がある

×：出来ない

5.3.3 コミュニティへの貢献

LKSTに関して、以下の活動を行っている。

- ・ sourceforgeにおけるソースコード公開
- ・ カーネル開発者やディストリビュータに対するプレゼンテーション
- ・ Linuxのユーザ会における勉強会の開催
- ・ 日経Linuxへの記事の執筆
- ・ LinuxWorld2005などのカンファレンスにおける講演

LKSTはミラクル・リナックス社製品に採用されている。

カーネル開発者やメンテナへのプレゼンテーションでは、トレーサおよび解析結果の有効性に関しては理解して頂いたが、Linuxカーネルへの採用には至っていない。カーネル2.6系のメンテナであるAndrew Morton氏と話をしたが、パッチサイズをもっと小さくすることが採用の条件の1つであるとの助言を頂いた。今後はパッチサイズの削減にも取り組んでいく。

6 付録:

6.1 LKST 適用 OS の構築とインストール方法

6.1.1 LKST適用カーネルをソースコードから構築しインストール

プログラムの構築とインストール手順は以下の通りである。

(1) ソースコードの入手

以下のソースコードを用意する。

(a) カーネルソースコード

- linux-2.6.9.tar.bz2

URL: <http://kernel.org>

(b) LKST のソースコード:

- lkst-2.3.1.tar.gz

URL: <http://sourceforge.net/projects/lkst/>

(c) LKST の構築環境:

- Fedora Core 3

(2) 下記のコマンドを実行し、スーパーユーザでログインする。

```
# su -
Password:
```

スーパーユーザのパスワードを要求されるので、入力する。

(3) LKST のソースパッケージを展開する。ソースアーカイブは圧縮されているので、コマンドラインから下記のコマンドを実行し、ソースファイルを展開する。

```
# cd /usr/local/src
# tar xzvf lkst-2.3.1.tar.gz
```

(プロンプト # が表示されたら終了)

(4) LKST を適用したカーネルの構築を行う。

- (a) LKST のパッチを Linux upstream カーネル 2.6.9 に当てる。パッチが複数あるので、一括して実行するスクリプトを提供している。(2)で作成したディレクトリ (lkst-2.3.1/)に移動し、そのスクリプトを実行する。

```
# cd lkst-2.3.1
# ./scripts/enpatch.sh /usr/src/linux-2.6.9
```

(プロンプト # が表示されたら終了)

実行すると、以下の確認を聞いてくる(質問の後ろに記載した[]の中がデフォルトの指定)ので、文末の強調部分の文字を入力する。

```
Apply preassigned page table patch (if not, apply vmsync patch automatically) (Y)es/(N)o? [Y] Y
Apply performance analyzing patch(recommend) (Y)es/(N)o? [Y] Y
Apply early boot-time tracing patch (optional) (Y)es/(N)o? [N] N
Apply extra event set patch (optional) (Y)es/(N)o? [N] N
Apply ignore hooks placed in inline-functions patch (optional) (Y)es/(N)o? [N] N
```

(5) Linux カーネルのディレクトリへ移動し、Configuration を設定する。

```
# cd /usr/src/linux-2.6.9
# make menuconfig
```

この中で、LKST を有効にするために、文末の強調部分の値で設定を行なう。(カーネルの他の設定項目は使用機種に依存するため、ここでは説明を省略する)

```
"Loadable module support"を選択し、
"Module unloading" Y
"Kernel hacking"を選択し、
"Kernel debugging" Y
"Magic SysRq key" Y
"Kernel Hook Support" Y
"Kernel State Tracing (LKST)" m
```

Save して終了する。

(6) LKST を適用した Linux upstream カーネルを再構築する。

```
# make modules bzImage &> make.log
```

(プロンプト # が表示されたら終了)

再構築でエラーが表示されないことを確認するため、以下のコマンドを実行して何も表示されないことを確認する。

```
# grep Error make.log
```

(7) LKST を適用した Linux upstream カーネルとモジュールを、所定のディレクトリにインストールします。

```
# make modules_install
# make install
```

(プロンプト # が表示されたら終了です)

モジュールがインストールされていることを確認するため、次のコマンドを実行する。

```
# ls /lib/modules/2.6.9-lkst231
Build/                modules.isapnpmap
kernel/               modules.parpportmap
modules.dep           modules.pcimap
modules.generic_string modules.pnpbiosmap
modules.ieee1394map   modules.usbmap
```

また、Linux upstream カーネルがインストールされていることを確認するため、次のコマンドを実行する。

```
# ls /boot
Kerntypes
Kerntypes-2.6.9-lkst231
System.map
System.map-2.6.9-lkst231
config-2.4.20-8
config-2.4.20-8smp
grub/
initrd-2.6.9-lkst231.img
kernel.h
message
message.ja
module-info
module-info-2.4.20-8
module-info-2.4.20-8smp
vmlinuz-2.6.9-lkst231
(その他は省略)
```

(1)

(2)

(3)

出力画面に(1)(2)(3)があることを確認する。

- (8) 起動 OS のリストに、今インストールした LKST を適用した Linux upstream カーネルが追加されていることを確認する。

```
# cd /boot/grub/  
# cat grub.confx
```

以下の画面が出てくることを確認する。

```
# grub.conf generated by anaconda  
#  
(途中省略)  
default=9  
timeout=10  
splashimage=(hd0,1)/boot/grub/splash.xpm.gz  
title Original Linux (2.6.9-lkst231)  
    root (hd0,1)  
    kernel /boot/vmlinuz-2.6.9-lkst231 ro root=LABEL=/ hdc=ide-scsi  
    initrd /boot/initrd-2.6.9-lkst231.img  
(以後省略)
```

図 6.1-1 grub.conf 内確認画面

6.1.2 LKSTを適用したOSの再起動

- (1) LKST を適用した Linux upstream カーネルでシステムを再起動する。
- (2) 再起動後、OS の確認をする

バージョン文字列が 2.6.9-lkst231 であること、さらに EM64T 対応のカーネルであることを示す x86_64 の文字列があることを確認する。

```
# uname -r  
2.6.9-lkst231 x86_64
```

6.1.3 LKST コマンドと解析ツール

6.1.3.1 LKST コマンドと解析ツールをソースコードから構築し、インストール

- (1) LKST 本体のディレクトリの直下に移動し、LKST コマンドの構築のための、カーネルソースコード（ヘッダファイル）へのパス設定を行う。

```
# cd /usr/local/src/lkst-2.3.1/  
# make config
```

(プロンプト # が表示されたら終了)

- (2) LKST コマンドの構築とインストールを行う。

```
# make  
# make install
```

(プロンプト # が表示されたら終了)

6.1.3.2 LKST プログラムの確認

- (1) LKST 実行に必要なソフトウェア環境を確認する。

```
# ls /lib/modules/2.6.9-lkst231/kernel/drivers/lkst/
```

- (2) LKST モジュールが表示されることを確認する。

```
# ls /lib/modules/2.6.9-lkst231/kernel/drivers/lkst  
/lib/modules/2.6.9-lkst231/kernel/drivers/lkst/lkst.ko
```

- (3) LKST コマンド群が表示されることを確認する。

```
#ls /usr/sbin/lkst*  
  
/usr/sbin/lkst                    /usr/sbin/lksteh  
  
/usr/sbin/lkst_event_count        /usr/sbin/lkstlogd  
  
/usr/sbin/lkst_make_mask          /usr/sbin/lkstlogger  
  
/usr/sbin/lkstbuf                 /usr/sbin/lkstm
```

- (4) LKST 解析ツール群が表示されることを確認する。

```
#ls /usr/bin/lkst*  
  
/usr/bin/lkst_plot_dist    /usr/bin/lkst_proc_list  
  
/usr/bin/lkst_plot_log    /usr/bin/lkstdogkeeper  
  
/usr/bin/lkst_plot_stat    /usr/bin/lkstla
```

- (5) イベントハンドラ群が表示されることを確認する。

```
# ls /lib/modules/2.6.9-lkst231/kernel/drivers/lkst/lksteh_*  
/lib/modules/2.6.9-lkst231/kernel/drivers/lkst/lksteh_procname.ko  
/lib/modules/2.6.9-lkst231/kernel/drivers/lkst/lksteh_procstat.ko  
/lib/modules/2.6.9-lkst231/kernel/drivers/lkst/lksteh_sysinfo.ko  
/lib/modules/2.6.9-lkst231/kernel/drivers/lkst/lksteh_vminfo.ko  
/lib/modules/2.6.9-lkst231/kernel/drivers/lkst/lksteh_wqcounter.ko
```

6.1.4 lkstモジュールのロードと実行

- (1) lkst モジュールをロードし、トレースの実行を開始する。

```
# modprobe lkst
```

- (2) LKST 制御用のデバイスファイル/dev/lkst を作成する。

```
# /usr/local/src/lkst-2.3.1/scripts/mkdev.sh
```

- (3) lkst が実行していることを確認する。

以下の画面が表示されればLKSTは稼動中である。

```
#lkst status  
<Current status>  
version of LKST      : 2.3.1  
number of cpus      : 2  
number of masksets  : 3  
number of event-handlers: 3
```

6.2 LKSTLogTools で採取したデータ

今回測定した表 2.1-2の項目のうち、項番1「システムコールの開始から終了までの時間の計測」に関するデータを掲載する。

6.2.1 システムコールの開始から終了までの時間（抜粋）

6.2.1.1 時系列データ

System call analyzer

sysno	syscall_name	start[sec]	processing-time
5	fstat	1127478987.892992668	0.000000812
9	mmap	1127478987.892995380	0.000001630
1	write	1127478987.893001774	0.000008842
14	rt_sigprocmask	1127478987.893024798	0.000000797
14	rt_sigprocmask	1127478987.893026025	0.000000273
0	read	1127478987.893028075	0.000004152
23	select	1127478987.893066389	0.000004398
14	rt_sigprocmask	1127478987.893071242	0.000000402
14	rt_sigprocmask	1127478987.893071937	0.000000235
1	write	1127478987.893072962	0.000020695
3	close	1127478987.893102057	0.000001687
11	munmap	1127478987.893111386	0.000004955
14	rt_sigprocmask	1127478987.893207401	0.000000603
16	ioctl	1127478987.893208559	0.000002165
14	rt_sigprocmask	1127478987.893211026	0.000000245
16	ioctl	1127478987.893211991	0.000001675
16	ioctl	1127478987.893214258	0.000000463
14	rt_sigprocmask	1127478987.893239479	0.000000312
61	wait4	1127478987.893242771	0.000000850
15	rt_sigreturn	1127478987.893244126	0.000000835
14	rt_sigprocmask	1127478987.893255893	0.000000227
13	rt_sigaction	1127478987.893257050	0.000000960
201	time	1127478987.893268217	0.000000665
4	stat	1127478987.893276109	0.006718084
201	time	1127478987.900002305	0.000000747
14	rt_sigprocmask	1127478987.900032370	0.000000840
16	ioctl	1127478987.900033562	0.000001885
14	rt_sigprocmask	1127478987.900035757	0.000000295
13	rt_sigaction	1127478987.900036502	0.000000573
14	rt_sigprocmask	1127478987.900046621	0.000000293
16	ioctl	1127478987.900047661	0.000000470
16	ioctl	1127478987.900048444	0.000000390
16	ioctl	1127478987.900049296	0.000001700
16	ioctl	1127478987.900054318	0.000003932
14	rt_sigprocmask	1127478987.900062220	0.000000368
13	rt_sigaction	1127478987.900063480	0.000000345
13	rt_sigaction	1127478987.900064470	0.000000310
13	rt_sigaction	1127478987.900065212	0.000000673
13	rt_sigaction	1127478987.900066260	0.000000235
13	rt_sigaction	1127478987.900066902	0.000000428
13	rt_sigaction	1127478987.900067672	0.000000225
13	rt_sigaction	1127478987.900068377	0.000000283
13	rt_sigaction	1127478987.900069057	0.000000415
13	rt_sigaction	1127478987.900069837	0.000000208
13	rt_sigaction	1127478987.900070445	0.000000412
13	rt_sigaction	1127478987.900071219	0.000000210
13	rt_sigaction	1127478987.900071827	0.000000407
13	rt_sigaction	1127478987.900072637	0.000000282
1	write	1127478987.900081879	0.000006582

6.2.1.2 統計データ

System call analyzer

sysno	syscall_name	count	average	max	min	total	percent
0	read	14912	0.002427569	22.520000078	0.000000250	36.199902752	5.950
1	write	3799	0.000032547	0.036980192	0.000000270	0.123646275	0.020
2	open	19839	0.000065267	0.017791240	0.000001065	1.294827931	0.213
3	close	12785	0.000002204	0.000668399	0.000000162	0.028173292	0.005
4	stat	10987	0.000113425	0.020459669	0.000000805	1.246196885	0.205
5	fstat	12493	0.000000519	0.000009927	0.000000274	0.006477940	0.001
6	lstat	1688	0.000003823	0.000013679	0.000000708	0.006453995	0.001
7	poll	240	0.415976533	1.984573387	0.000000995	99.834367975	16.409
8	lseek	6421	0.000000245	0.000009532	0.000000170	0.001574439	0.000
9	mmap	12535	0.000002180	0.000034750	0.000000518	0.027332375	0.004
10	mprotect	1410	0.000003044	0.000010194	0.000001692	0.004291531	0.001
11	munmap	4504	0.000004522	0.000130353	0.000001745	0.020364959	0.003
12	brk	2243	0.000001660	0.000011221	0.000000295	0.003723334	0.001
13	rt_sigaction	2214	0.000000453	0.000014706	0.000000202	0.001002576	0.000
14	rt_sigprocmask	2006	0.000000426	0.000011229	0.000000182	0.000854338	0.000
15	rt_sigreturn	330	0.000001875	0.000003495	0.000000337	0.000618603	0.000
16	ioctl	1316	0.000063469	0.003179158	0.000000273	0.083524672	0.014
20	writew	3	0.000004630	0.000006422	0.000003615	0.000013891	0.000
21	access	2499	0.000021590	0.013296776	0.000001180	0.053953862	0.009
22	pipe	106	0.000008869	0.000013647	0.000002932	0.000940094	0.000
23	select	539	0.501837166	33.989652492	0.000002972	270.490232718	44.459
32	dup	28	0.000000803	0.000001360	0.000000307	0.000022489	0.000
33	dup2	150	0.000001065	0.000006047	0.000000240	0.000159719	0.000
37	alarm	1	0.000001269	0.000001269	0.000001269	0.000001269	0.000
38	setitimer	170	0.000001033	0.000002140	0.000000593	0.000175651	0.000
39	getpid	146	0.000000256	0.000000485	0.000000117	0.000037312	0.000
41	socket	306	0.000002898	0.000010455	0.000001640	0.000886711	0.000
44	sendto	2	0.000016972	0.000021730	0.000012214	0.000033944	0.000
45	recvfrom	2	0.000003687	0.000004773	0.000002602	0.000007375	0.000
56	clone	222	0.000084493	0.000160803	0.000058938	0.018757537	0.003
58	vfork	160	0.001095924	0.046872893	0.000066178	0.175347878	0.029

59	execve	1513	0.000279161	0.046829803	0.000001300	0.422371224	0.069
61	wait4	465	0.328774243	21.377246264	0.000000457	152.880023046	25.128
63	uname	418	0.000000728	0.000001430	0.000000442	0.000304109	0.000
72	fcntl	1173	0.000000729	0.000006327	0.000000262	0.000854801	0.000
79	getcwd	108	0.000002850	0.000007242	0.000001600	0.000307825	0.000
80	chdir	30	0.000004378	0.000012709	0.000000995	0.000131328	0.000
81	fchdir	2	0.000001162	0.000001552	0.000000772	0.000002324	0.000
82	rename	11	0.000032210	0.000039317	0.000017749	0.000354310	0.000
83	mkdir	7	0.000006952	0.000041220	0.000000970	0.000048661	0.000
84	rmdir	1	0.000175569	0.000175569	0.000175569	0.000175569	0.000
87	unlink	587	0.000164564	0.013039413	0.000002370	0.096598931	0.016
88	symlink	397	0.000024018	0.000109524	0.000011447	0.009535065	0.002
90	chmod	5	0.000006719	0.000009767	0.000005882	0.000033593	0.000
95	umask	14	0.000000217	0.000000375	0.000000117	0.000003040	0.000
96	gettimeofday	997	0.000000649	0.000007219	0.000000345	0.000646985	0.000
97	getrlimit	378	0.000000183	0.000000642	0.000000097	0.000069310	0.000
98	getrusage	70	0.000001637	0.000002085	0.000000985	0.000114615	0.000
102	getuid	33	0.000000358	0.000000565	0.000000135	0.000011822	0.000
104	getgid	33	0.000000114	0.000000147	0.000000100	0.000003764	0.000
107	geteuid	53	0.000000174	0.000000483	0.000000090	0.000009203	0.000

6.2.1.3 分布データ

System call analyzer

sysno	syscall_name	less-than-min	0.00000100	0.00001000	0.000010000	0.000100000	0.001000000	0.010000000	0.100000000	1.000000000	more-than-max
0	read	0	2012	8780	1747	1294	1025	26	26	1	1
1	write	0	122	2509	1092	57	18	1	0	0	0
2	open	0	0	17494	2133	55	116	41	0	0	0
3	close	0	8677	4024	58	26	0	0	0	0	0
4	stat	0	7	9765	925	71	205	14	0	0	0
5	fstat	0	11956	537	0	0	0	0	0	0	0
6	lstat	0	59	1628	1	0	0	0	0	0	0
7	poll	0	1	130	0	0	0	0	84	25	0
8	lseek	0	6409	12	0	0	0	0	0	0	0
9	mmap	0	893	11588	54	0	0	0	0	0	0
10	mprotect	0	0	1409	1	0	0	0	0	0	0
11	munmap	0	0	4364	139	1	0	0	0	0	0
12	brk	0	849	1392	2	0	0	0	0	0	0
13	rt_sigaction	0	2064	149	1	0	0	0	0	0	0
14	rt_sigprocmask	0	1944	61	1	0	0	0	0	0	0
15	rt_sigreturn	0	69	261	0	0	0	0	0	0	0
16	ioctl	0	737	552	1	0	26	0	0	0	0
20	writev	0	0	3	0	0	0	0	0	0	0
21	access	0	0	2481	11	2	4	1	0	0	0
22	pipe	0	0	63	43	0	0	0	0	0	0
23	select	0	0	153	34	10	13	138	168	21	2
32	dup	0	17	11	0	0	0	0	0	0	0
33	dup2	0	61	89	0	0	0	0	0	0	0
37	alarm	0	0	1	0	0	0	0	0	0	0
38	setitimer	0	104	66	0	0	0	0	0	0	0
39	getpid	0	146	0	0	0	0	0	0	0	0
41	socket	0	0	304	2	0	0	0	0	0	0
44	sendto	0	0	0	2	0	0	0	0	0	0
45	recvfrom	0	0	2	0	0	0	0	0	0	0
56	clone	0	0	0	163	59	0	0	0	0	0

58	vfork	0	0	0	50	76	32	2	0	0	0
59	execve	0	0	1116	128	228	28	13	0	0	0
61	wait4	0	75	73	19	43	82	118	46	3	6
63	uname	0	402	16	0	0	0	0	0	0	0
72	fcntl	0	1152	21	0	0	0	0	0	0	0
79	getcwd	0	0	108	0	0	0	0	0	0	0
80	chdir	0	1	27	2	0	0	0	0	0	0
81	fchdir	0	1	1	0	0	0	0	0	0	0
82	rename	0	0	0	11	0	0	0	0	0	0
83	mkdir	0	1	5	1	0	0	0	0	0	0
84	rmdir	0	0	0	0	1	0	0	0	0	0
87	unlink	0	0	16	536	19	14	2	0	0	0
88	symlink	0	0	0	395	2	0	0	0	0	0
90	chmod	0	0	5	0	0	0	0	0	0	0
95	umask	0	14	0	0	0	0	0	0	0	0
96	gettimeofday	0	893	104	0	0	0	0	0	0	0
97	getrlimit	183	195	0	0	0	0	0	0	0	0
98	getrusage	0	1	69	0	0	0	0	0	0	0
102	getuid	0	33	0	0	0	0	0	0	0	0
104	getgid	0	33	0	0	0	0	0	0	0	0
107	geteuid	1	52	0	0	0	0	0	0	0	0
108	getegid	1	40	0	0	0	0	0	0	0	0
109	setpgid	0	3	3	0	0	0	0	0	0	0
110	getppid	0	33	0	0	0	0	0	0	0	0
111	getpgrp	0	33	0	0	0	0	0	0	0	0
116	setgroups	0	1	3	0	0	0	0	0	0	0
117	setresuid	0	1	3	0	0	0	0	0	0	0
119	setresgid	0	4	0	0	0	0	0	0	0	0
158	arch_prctl	0	107	276	1	0	0	0	0	0	0
201	time	0	90	2	0	0	0	0	0	0	0
217	getdents64	0	289	62	219	56	62	5	0	0	0
232	epoll_wait	0	0	0	0	0	0	0	0	9	0

6.3 ベンチマーク測定データ

参考のために、今回測定したベンチマークの測定データを掲載する。

6.3.1 dbench

表 6.3-1 dbench の測定データ

	throughput
pure	256.244
lkst-norec	249.7443
lkst-ipa	168.6268
lkst-nolock	215.2423
lkst-all	164.389

6.3.2 Lmbench

表 6.3-2 lmbench の測定データ(システムコール、シグナル等)

	null call	null I/O	stat	open/ close	select	siginst	sigband	fork	exec	sh
pure	0.21	0.28	1.855	2.5575	10.4	0.33	3.1175	167.5	544.5	1788
lkst-norec	0.21	0.3	2.0075	2.7925	10.725	0.36	3.2225	174	568	1864.75
lkst-ipa	0.39	0.63	3.1	4.8575	25.225	0.69	4.7925	223.25	807.5	2635.5
lkst-nolock	0.39	0.48	2.1575	3.1125	11	0.53	3.755	183.75	629.25	2025.5
lkst-all	0.39	0.63	3.445	5.3925	25.45	0.69	5.215	231.75	831	2730.25

表 6.3-3 lmbench の測定データ(コンテキストスイッチ)

	2p/0K	2p/16K	2p/64K	8p/16K	8p/64K	16p/16K	16p/64K
pure	4.41	4.7375	4.865	4.2525	5.2525	4.05	13.975
lkst-norec	4.4875	4.7625	4.62	4.0475	6.23	3.9925	10.5775
lkst-ipa	5.59	5.8625	6.0875	5.715	6.845	5.93	10.6675
lkst-nolock	5.035	5.2225	5.46	4.8075	6.2125	4.8575	11.4925
lkst-all	5.6475	5.8725	5.7525	5.6575	6.5975	5.7075	14.6025

表 6.3-4 lmbench の測定データ(ネットワーク等)

	2p/0k	Pipe	AFUNIX	UDP	RPC/UDP	TCP	RPC/TCP	TCP connect
pure	4.41	11.2	13.65	18.05	23.6	21.175	28.525	32
lkst-norec	4.4875	11.5	15.8	18.725	24.675	22	29.375	33.75
lkst-ipa	5.59	15.575	20.975	24.35	33.175	30.15	39.025	48
lkst-nolock	5.035	13.175	16.775	20.6	27.8	24.2	32.875	36.5
lkst-all	5.6475	15.65	29.175	27.575	36.575	33.1	42.65	56.25

表 6.3-5 lmbench の測定データ(ファイル操作、VM 性能等)

	0kCreate	0kDelete	10kCreate	10kDelete	Mmap	ProtFault	PageFault	100fd
pure	14.725	12.75	47.7	28.375	12.25	0.70325	1.787875	9.22775
lkst-norec	15.35	13.2	49.375	28.675	13.2	0.7595	1.90685	9.4025
lkst-ipa	21.05	18.5	64.725	38.875	21.55	1.3555	3.0229	24.025
lkst-nolock	15.85	13.5	50.975	28.975	14.425	0.9315	2.0949	9.5825
lkst-all	22.025	19.25	66.95	40.425	22.125	1.3435	3.120475	24.275

表 6.3-6 lmbench の測定データ(ローカルコミュニケーション)

	Pipe	AFUNIX	TCP	FileReread	MmapReread	Bcopy(libc)	Bcopy(hand)	MemRead	MemWrite
pure	645.5	1287	774	2528.075	3910.25	1213.925	1208.375	3668	1655.75
lkst-norec	619.5	1283	758.5	2477.8	3910.475	1235.775	1180.05	3668.5	1650.5
lkst-ipa	404.75	1236	690	2141.875	3895.7	1230.825	1201.225	3653.5	1644.75
lkst-nolock	518	1276.5	726.25	2458.875	3899.475	1228.15	1204.225	3658.25	1645
lkst-all	404.75	1176.5	647	2141.6	3892.725	1234.375	1200.85	3651.75	1643.75

6.3.3 tiobench

表 6.3-7 tiobench の測定データ(Sequential Read)

Sequential Reads	throughput	CPU%	avg latency	max latency	lat >2s %	lat >10s %	CPU effect
pure	4.511304314	2.397	3.223745	1575.5105	0	0	188.2293254
lkst-norec	4.568233867	2.563	3.3014475	1500.004	0	0	178.2148063
lkst-ipa	5.255370999	4.186	2.7977075	1651.0715	0	0	125.5322709
lkst-nolock	5.110377861	3.277	2.8364075	1639.14175	0	0	155.9666774
lkst-all	4.658074952	3.803	3.0632025	1616.2505	0	0	122.4773801

表 6.3-8 tiobench の測定データ(Random Reads)

Random Reads	throughput	CPU%	avg latency	max latency	lat >2s %	lat >10s %	CPU effect
pure	1.962065062	1.803	6.958705	1055.273	0	0	108.8427854
lkst-norec	1.914127582	1.799	6.96151	1046.9105	0	0	106.3891683
lkst-ipa	1.947826886	2.685	6.957675	1046.19575	0	0	72.55539168
lkst-nolock	1.956107526	2.252	6.98629	979.49875	0	0	86.87017532
lkst-all	1.926060639	2.738	6.898235	1037.9165	0	0	70.35008442

表 6.3-9 tiobench の測定データ(Sequential Writes)

Sequential Writes	throughput	CPU%	avg latency	max latency	lat >2s %	lat >10s %	CPU effect
pure	3.569501772	6.025	3.7708825	34915.2913	0.02232	0.012495	59.24511654
lkst-norec	3.554564937	5.916	3.664885	35528.1585	0.02165	0.01154	60.08450851
lkst-ipa	3.645554518	8.445	3.65902	32832.805	0.01745	0.012208	43.16777711
lkst-nolock	3.889989132	7.201	3.39178	31809.7995	0.01822	0.011253	54.01736336
lkst-all	3.766661777	8.978	3.16039	32686.899	0.01698	0.01087	41.95210004

表 6.3-10 tiobench の測定データ(Random Writes)

Random Writes	throughput	CPU%	avg latency	max latency	lat >2s %	lat >10s %	CPU effect
pure	1.890675877	2.111	5.8592475	20631.0445	0.02422	0.003813	89.57888738
lkst-norec	1.881928452	2.204	6.4475025	28895.3173	0.02098	0.00343	85.37833483
lkst-ipa	1.873102706	3.278	6.1775725	42704.7935	0.02174	0.00305	57.14199619
lkst-nolock	1.891456025	2.445	6.160855	28128.9438	0.02098	0.00343	77.36189881
lkst-all	1.892066512	3.264	5.9374075	17883.8275	0.01812	0.003623	57.97654498