Vol.2013-OS-124 No.12
2013/3/1

Linux トレーサーを用いた，分散処理システムの I/O ボトルネック特定方式の提案と評価

松村 俊太郎 [1,a)]　関谷 博 [1,b)]　鍛治 武志 [1,c)]　中村 英児 [1,d)]

概要： 分散処理システムにおいて，アプリケーションの処理速度が低下した場合の発生箇所の絞込みは，(1) 自律動作により処理を担当するサブシステムや時系列などが異なること．(2) 処理負荷やシステム環境による競合条件などの変化によるボトルネック発生箇所の変化，により困難である．これらの課題に対して，分散処理システムにおいて I/O システムコールをトレースして性能情報や呼出し履歴を収集・分析することにより，ボトルネックの発生箇所を特定する方式を提案する．また，提案方式を実装して評価を行い，トレースによる性能低下を抑止しながら I/O ボトルネックの原因となるコード位置が特定できることを報告する．

キーワード： Linux, 分散処理システム, トレーサー, ボトルネック, システムコール, スタック・トレース

The Evaluation of a New I/O Bottleneck Detection Method for Distributed Processing System by the Tracer of Linux

MATSUMURA SHUNTARO [1,a)]　SEKIYA HIROSHI [1,b)]　KAJI TAKESHI [1,c)]　NAKAMURA EIJI [1,d)]

Abstract: In a distributed processing system, narrowing down of a generating part when the processing speed of application falls is difficult; (1) A subsystem, a time series, etc. which take charge of processing by autonomous operation are scattering; (2) A bottleneck generating part changes by the race condition by change of processing load, system environment, etc. ; As opposed to these subjects, by tracing an I/O system call in a distributed processing system, and collecting and analyzing performance information and a call history, we propose the system which pinpoints the generating part of a bottleneck. Moreover, we report that the code position leading to an I/O bottleneck can be pinpointed, evaluating by mounting a proposal system and deterring the performance decrement by trace.

Keywords: Linux, Distributed Processing System, Tracer, Bottleneck, System Call, Stack Trace

1. はじめに

筆者らは，分散処理システムの導入前に実施する性能試験において，アプリケーションの処理速度が要件を大幅に下回る事象に遭遇した．

アプリケーションの処理速度が低下した場合には，原因解明のために発生箇所の絞込みが必要であるが，分散処理システムにおいては下記の 2 つの課題により発生箇所の絞込みが困難である．

- 分散処理システムのサブシステムは自律で動作するため，同じ処理を実行する場合でも処理を担当するマシン・サブシステム・時間帯が異なり解析対象のマシンの絞込みが困難である．
- アプリケーション処理の負荷やシステム環境による競合条件などの変化により，ボトルネックの発生箇所が変化する．

これらの課題に対して，本稿では分散処理システムにおいてファイル I/O に起因するボトルネックの発生箇所を特定する方式の提案を行い，その評価結果について報告する．

2. 背景

筆者らは，大量データを蓄積・分析するための大規模分散処理基盤 CBoC タイプ 2[1]を開発・運用している．CBoC タイプ 2 は，大量データを構造化データとして管理する分散テーブル（BigTable[2]や HBase[3]に相当），大量のデータを多数のサーバーに分散格納する分散ファイル（GFS[4]や HDFS[5]に相当），これらの分散処理システムの可用性を高めるための基本機能を提供する分散ロック（Chubby[6]に相当）という 3 つの分散処理サブシステムで構成されている（図 1 参照）．

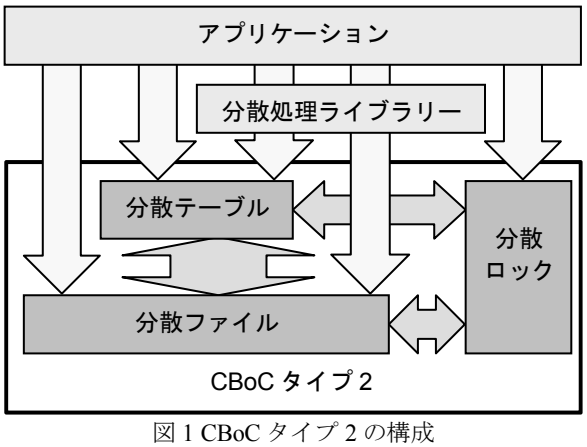

図 1 CBoC タイプ 2 の構成

1 NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center
a) matsumura.shuntaro@lab.ntt.co.jp
b) sekiya.hiroshi@lab.ntt.co.jp
c) kaji.takeshi@lab.ntt.co.jp
d) nakamura.eiji@lab.ntt.co.jp

分散テーブルは WAL（ログ先行書き込み）方式を採用している．分散テーブルへの書き込み要求を受信すると分散ファイル上のファイルに更新内容を追記し，追記が完了するとメモリー上のデータを更新し書き込み要求元へ応答を返す．

分散ファイルはファイルをチャンクと呼ばれる 64MiB の固定長サイズで分割して管理している．サーバーやスイッチの故障に備えて，各チャンクは異なるスイッチ配下にある複数のサーバー上に複製が保存される．複製作成の基本的なタイミングはデータ書き込み時であり，書き込みデータを対象とする複数のサーバーに転送し，データ転送の完了を確認後に実際の書き込みを指示する方式を採っている．

CBoC タイプ 2 の性能試験において，アプリケーションのタスクを同時に 2 つ起動すると，タスクを 1 つ起動した時に比べ性能が 1/3 未満に低下する事象が発生した．リソース使用量は，CPU・メモリー・ディスク・Network のいずれも限界値の 1/4 未満であったが，全マシン上で通常即時完了する `uname` コマンドの完了までに数秒以上を要したため，I/O システムコールにボトルネックがあることがわかったが，発生箇所を特定することは困難であった．

3. I/O ボトルネック特定方式の要件と既存方式

3.1 要件

分散処理システムにおいて，I/O システムコールのボトルネック発生によりアプリケーションの性能が低下する事象の原因を特定するためには，以下の 4 つの要件を満たす必要がある．

【要件 1】 OS・サブシステム・アプリケーションは変更できない

分散処理システムでは通信方式が確立されており，プロトコルの変更が困難なサブシステム，例えばデータベースシステムを利用する場合が多い．そのため，サブシステムなどのプログラムを変更することなくボトルネックを調査できることが必要である．

【要件 2】 トレース行う場合は，処理に影響を与えてはならない

分散処理システムでのボトルネック特定は，システム内を流れるデータをトレースし，それらを分析する方式が主流である．トレースの負荷が高いとレースコンディションなどの再現性が著しく低下してしまい，ボトルネックの特定に影響を与えることがあるため，トレースの負荷を低く抑える必要がある．

【要件 3】 I/O システムコールがクライアント要求単位で紐付けできる

分散処理システムではアプリケーションが分散処理システム上のサブシステム（例えば分散テーブル）に要求（以降「クライアント要求」と称す）を発行すると，複数のマシンで分担して処理を行うだけでなく他のサブシステム（例えば分散ファイル）に様々な要求を発行するため，1 回のクライアント要求で多数の処理が発生する．

分散処理システムは同時に多数のクライアント要求を処理するため，性能調査対象のアプリケーション群に起因するボトルネックを特定するためには，I/O システムコールをクライアント要求単位で紐付けする必要がある．

【要件 4】 ボトルネックと判定された I/O システムコールから呼出し元コード位置が特定できる

要件 3 で特定した I/O システムコールのトレースログから，処理時間が長いなどボトルネックと考えられる I/O システムコールを特定する．I/O システムコールの特定においては，I/O 対象のファイルパスのみならずボトルネックを引き起こしたコード位置を特定する必要がある．

例えば，`/etc/hosts` ファイルの読み出し操作がボトルネックになっている場合，Linux の glibc の `getaddrinfo()` などが呼出しの度に `/etc/hosts` を読み出していること，アプリケーションが同一のホスト名に対して何度も `getaddrinfo()` などを呼出していること，など様々な要因が挙げられ，発生状況により主要因が異なる．

3.2 既存方式との比較

アプリケーションの性能が低下する問題を解決するために以下の方式（A，B，C）が提案されているが，3.1 項で示した 4 つの要件を満足するものはない（表 1 参照）．

方式 A: プログラム間の遠隔呼出し毎に一意な識別子を設け，通信データに呼出し元・呼出し先の識別子を付与するように通信ライブラリを統一化し，通信の流れをトレースする方式（Dapper[7]に相当）．この方式では，全プログラムで規定の遠隔呼出し方式を用いるよう修正を行う必要がある．

方式 B: 通信レイヤーでサーバー間の通信をキャプチャーし，通信データの送信・受信アドレスを関連付けることで，データの流れをトレースする方式（Performance Debugging for Distributed Systems of Black Boxes[8]に相当）．この方式ではプログラムの修正は不要であるが，ボトルネックの特定範囲はサーバー・プロセス単位に限定され，プロセス内部のサブルーチン等の詳細までは特定できない．

方式 C: 各マシン内でシステムコール等をトレースし，ボトルネック箇所を特定する方式（Kprobes[9]，ftrace[10]，utrace[11]に相当）．この方式ではプログラムの修正は不要でありボトルネックの発生箇所も細部まで特定できるが，マシン間のデータの流れを追跡することは困難である．

表 1 既存方式と要件との比較

既存方式	要件 1	要件 2	要件 3	要件 4
方式 A	×	○	○	○
方式 B	○	○	×	×
方式 C	○	○	△※1	△※2

※1 マシンを跨ぐトレースはできない
※2 I/O システムコールのトレース中はスタック・トレースが実行できない

以上のように，方式 A，B はそれぞれアプリケーション，収集可能な情報の制約により活用できないため，方式 C に不足している要件に機能追加することで課題の解決を図る．

4. I/O ボトルネック特定方式の提案

本章では，3.1 章の要件を満足したボトルネック特定方式を提案する．各要件に対して必要な機能は表 2 に示す通り

であり，機能 1 は 4.1 章，機能 2 は 4.2 章で提案方式を説明する．機能 3 は 5.2.4 章の(1)，機能 4 は 5.2.4 章の(2)において評価結果に基づく分析手法の一例を説明する．

表 2 要件と必要機能

要件	必要な機能
要件 3	**[機能 1]** 分散処理システムを構成する各マシン上で I/O システムコールのトレースログを収集する．
	[機能 2] I/O システムコールのトレースログからクライアント要求を特定する．
要件 4	**[機能 3]** 特定された I/O システムコールのトレースログから，ボトルネックとなっている I/O システムコールを特定する．
	[機能 4] I/O システムコールのボトルネック状況と呼出し元コード位置の履歴（以降「呼出し履歴」と称す）から I/O システムコールの呼出し元コード位置を特定する．

4.1 I/O システムコールのトレースログを収集

4.1.1 Linux トレーサーの選択

アプリケーションの性能に影響する I/O ボトルネックを特定するためには，分散処理システムを構成する全てのマシン上で I/O システムコールをトレースする必要がある．Linux のトレーサーには表 3 の方式がある．ftrace は I/O システムコールの性能情報が収集できず，utrace はデフォルトで OS に組み込まれていないため要件 1 を満足しない．このため，他の方式に比べトレース時にソフトウェア割込を必要とするため負荷は高いが，他の項目を全て満足する Kprobes を採用することとした．

表 3 Linux トレーサーの比較

項目	Kprobes	ftrace	utrace
対応 OS バージョン	Linux 2.6.9～	Linux 2.6.27～	Linux 2.6.9～ *要 patch*
トレースの開始・停止処理	○	○	△
I/O システムコールの性能情報を収集	○	×	○
トレースの負荷	△	○	○

4.1.2 I/O システムコールの呼出し履歴収集方式

I/O システムコールの呼出し履歴を収集する際には，スタック・トレースの実行が必要であるが，デバッガー（gdb などが相当）のスタック・トレースとは異なり，I/O システムコールのトレース中に外部ファイルの読み出しができない制約事項がある．そのため，外部ファイルの読み出し時にはバックグラウンドタスクに依頼する方式で制約事項を解決した．

スタック・トレースの実行方式の概要を図 2 に示す．図内に記載の IP，SP，FP はそれぞれ呼出し元コード位置，スタック位置，フレーム位置を指す．まず，カーネルスタックに格納されている IP(0)，SP(0)などを取得する．その後，IP(i) {i = 0, 1, 2, ...}からプログラム・ライブラリを特定し，ライブラリ内にあるフレーム情報から位置コードを取得する．位置コードと SP(i)，FP(i)などから IP(i + 1)を取得する処理を繰り返すことで呼出し履歴を収集する．

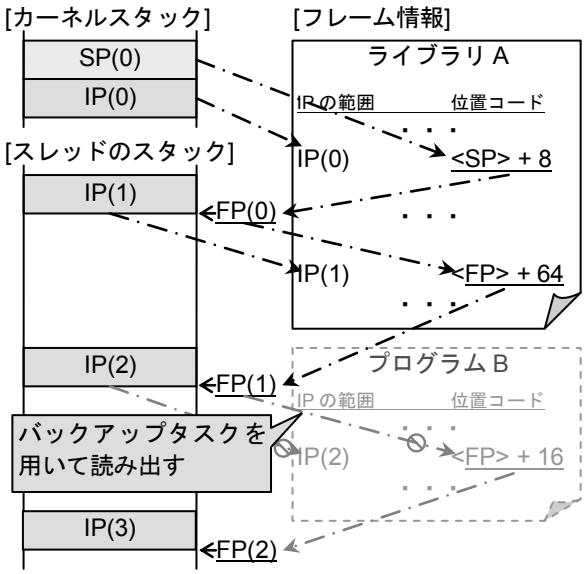

図 2 I/O システムコール上で
スタック・トレースを取得する方式の概要

図 2 の例では，IP(2)からプログラム B を特定したが，B のフレーム情報がメモリーにロードされていない場合は IP(3)以降の情報が取得できないため，本処理をバックアッププタスクに依頼し，バックアッププタスクにて B のフレーム情報を格納しているファイル（プログラム B 自身，あるいはプログラム B から切り離されたデバッグ情報[12]を格納しているファイル）からフレーム情報を読み出すことにより，IP(3)を収集する．

4.2 I/O システムコールからのクライアント要求特定方式

アプリケーションが発行した I/O システムコールのトレース情報をクライアント要求毎に紐付けし，クライアント要求毎に処理の流れがトレースできるようにする．

下記の 3 工程により I/O システムコールのトレースログをクライアント要求に紐付た．I/O システムコールのトレースログ例を表 4 に，表 4 のトレースログをクライアント要求に紐付た結果を図 3 に示す．

4.2.1 Network 受信と Network 送信の紐付け

時間に重なりがあり，かつ送信側アドレスと受信側アドレスが同一の Network 受信 I/O と Network 送信 I/O とを，同一のクライアント要求として扱う（図 3 の一点鎖線部分）．また，連続した 2 つの Network 送受信 I/O のうち，送信アドレス・受信アドレス・FD（File Descriptor）がすべて同一で，前に実行された I/O が失敗しているものは，同一のクライアント要求として扱う（図 3 の No.12, 13 部分）．

4.2.2 Network 送受信と同一スレッド上の I/O システムコールの紐付け

Network 送受信 I/O と，同一スレッドで次の Network 送受信 I/O が発行されるまでに発行された I/O 種別が送信・受信以外の I/O とを，同一のクライアント要求として扱う（図 3 の太線部分）．

4.2.3 クライアント要求の送信部分と応答部分の紐付け

4.2.1 章，4.2.2 章で生成されたクライアント要求は，クライアントからの要求送信部分(表 4 の No.1, 8, 9 を囲う▨箇所）とクライアントへの結果返信部分（No.2, 3, 10, 11 を囲う▨箇所），および複数の連続したクライアントからの要求送信部分（表 4 ではない），クライアントへの結果返信部分（No.2, 3, 10, 11 を囲う▨箇所と同 No.4, 5, 12, 13, 15 を囲う▨箇所）がそれぞれ分離されているため，これらを同一のクライアント要求として扱う（図 3 の▨箇所）．

これらの処理を行うことで，表 4 の例では図 3 の通り 1～13,15～17 と 14（および他の I/O システムコール）の 2 つのクライアント要求が特定できる．

表 4 I/O システムコールのトレースログの一例

No	開始時刻	終了時刻	ホスト名 / スレッド	FD	システムコール名	アドレス
1	.00021	.00022	vm0 / A	5	send	192.168.1.2:42857 →192.168.1.1:80
2	.00025	.00032	vm0 / A	5	recv	192.168.1.1:80 →192.168.1.2 :42857
3	.00035	.00036	vm0 / A	4	write	
4	.00038	.00043	vm0 / A	5	recv	192.168.1.1:80 →192.168.1.2:42857
5	.00046	.00047	vm0 / A	4	write	
6	.00050	.00052	vm0 / A	5	recv	192.168.1.1:80 →192.168.1.2:42857
7	.00055	.00056	vm0 / A	4	write	
8	.00023	.00024	vm1 / B	99	recv	192.168.1.2:42857 →192.168.1.1:80
9	.00027	.00028	vm1 / B	21	read	
10	.00030	.00031	vm1 / B	99	send	192.168.1.1:80 →192.168.1.2 :42857
11	.00034	.00035	vm1 / B	21	read	
12	.00037	.00038	vm1 / B	99	**send 失敗**	192.168.1.1:80 →192.168.1.2 :42857
13	.00040	.00042	vm1 / C	99	send	192.168.1.1:80 →192.168.1.2 :42857
14	.00043	.00045	vm1 / B	98	recv	192.168.1.2:57142 →192.168.1.1:80
15	.00044	.00046	vm1 / C	21	read	
16	.00050	.00051	vm1 / C	99	send	192.168.1.1:80 →192.168.1.2 :42857
17	.00054	.00055	vm1 / C	14	write	

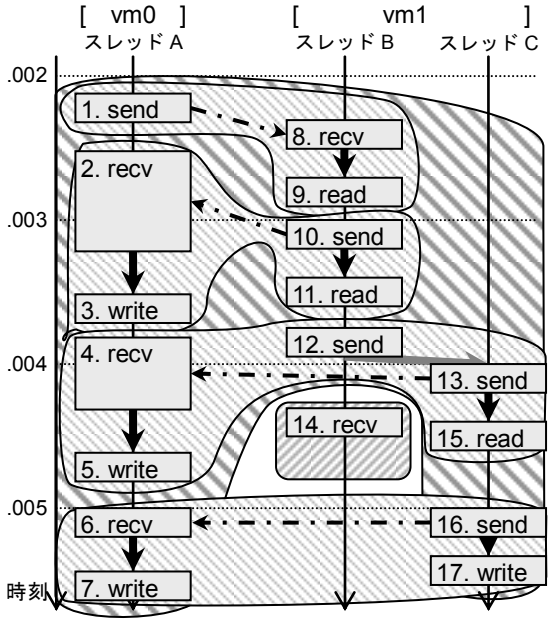

図 3 I/O システムコールから
クライアント要求を特定

5. 評価

本章では, CBoC タイプ 2 の分散ファイルにおけるデータ読出し処理に対して, 4 章で提案した方式を用いて実施したボトルネック特定の評価について報告する．

5.1 測定方法

具体的な測定方法は下記の通りである．

① 1 台の物理マシン上に 7 台の仮想マシンを起動し，表 5 の通り構成する．マシンの性能は表 6 の通りである．
② 分散ファイル上で，Client 毎に 1 GiB のファイル（ランダムデータ）を作成する．ファイルの複製数は 3 とする．
③ ディスクキャッシュの効果を抑止するために，測定直前に全マシンのディスクキャッシュをクリアする．
④ ② で作成したファイルから 2 MiB 毎に 16 KiB のデータを，全 Client より並列して読み出す．

表 5 マシン構成

ホスト名	分散ロック	分散ファイル		
		Master	Worker	Client
vm0	◎			○
vm1	○	◎		○
vm2	○	○		○
vm3			○	○
vm4			○	○
vm5			○	○
vm6			○	○

※ 分散ロック・分散ファイル Master の◎部分はプライマリーである

表 6 マシン性能

リソース名	物理マシン	仮想マシン
CPU	8 threads/4 cores, Intel Core I7-3770K@3.5GHz	2 threads

リソース名	物理マシン	仮想マシン
Memory	16GiB, DDR3-1600	1.5GiB
Disk(SSD)	233GiB, 6Gbpz	20GiB
Disk(HDD)	2,795GiB, 6Gbps, 5400rpm, 64MB buffer	320GiB
Network	1Gbps×1	1Gbps×1

5.2 評価結果

5.2.1 要件 1 の評価（OS・サブシステム・アプリケーションを変更せずにボトルネック調査が実現できること）.

本試験では，OS・サブシステム・アプリケーションのいすれも変更せずにボトルネック調査が実現できた.

5.2.2 要件 2 の評価（トレースによる性能への影響）

トレースの有無による処理時間の変動を確認し（図 4 参照），最大 2 割性能が低下に抑えることができた．なお，vm0～vm3 で処理時間が長く，vm4～vm6 で処理時間が短い理由は 5.2.3 章で説明する.

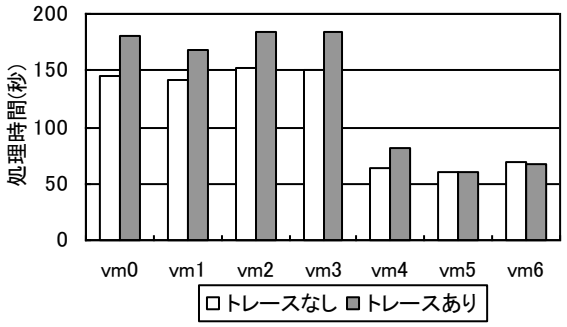

図 4 トレースの有無による，処理時間の変化

5.2.3 要件 3 の評価（I/O システムコールのトレースログからのクライアント要求特定）

分散ファイルの Client がファイルの読み出し処理時に行う処理は下記の 3 点である.

処理 1 分散ロックに対して，分散ファイルの Master のアドレスの取得を依頼

処理 2 分散ファイルの Master に対して，ファイルのオープン・クローズ，Worker 位置の取得を依頼

処理 3 分散ファイルの Worker に対して，データの取得を依頼

全マシンの I/O システムコールをトレースした情報から分散ファイルの Client が発行したクライアント要求を特定し，Client のホスト名・プロセス毎に処理時間を集計することにより，上記の処理 1～処理 3 のすべての処理を行っていることが確認できた（表 7 参照）.

表 7 より Client vm0～vm2 から処理 3 へのアクセスは vm3, vm4 にて処理していることがわかる（表 7 の**太字**部分）．分散ファイルは，Network 帯域削減のため Network 距離が短いサーバーにアクセスする特性があるためである．また，Client vm3～vm6 からの処理 3 はすべて同一マシン上で実行されていることがわかる（表 7 の***イタリック体***部分）．これは分散ファイルの機能により，ファイルの複製が作成されており，Client が動作するマシンに読出し対象のファイルが存在するためである.

図 4 で処理時間に偏りがあるのは，vm1～vm3 からの処理 3 は vm3 に集中しているためである.

表 7 Client・処理毎の合計処理時間（単位:秒）

処理 \ Client ホスト名		vm0	vm1	vm2
処理 1 (vm0)		1.1×10^{-4}	4.4×10^{-4}	9.4×10^{-4}
処理 2 (vm1)		2.2×10^{-3}	2.5×10^{-3}	1.5×10^{-3}
処理 3	vm3	**1.6×10^{2}**	**1.3×10^{2}**	**2.1×10^{2}**
	vm4	**4.3×10^{1}**	**8.1×10^{1}**	-
	vm5	-	-	-
	vm6	-	-	-

処理 \ Client ホスト名		vm3	vm4	vm5	vm6
処理 1 (vm0)		8.3×10^{-4}	6.3×10^{-5}	4.7×10^{-5}	5.7×10^{-5}
処理 2 (vm1)		2.1×10^{-3}	1.3×10^{-3}	1.2×10^{-3}	1.2×10^{-3}
処理 3	vm3	*1.7×10^{2}*	-	-	-
	vm4	-	*2.4×10^{1}*	-	-
	vm5	-	-	*2.0×10^{1}*	-
	vm6	-	-	-	*2.8×10^{1}*

5.2.4 要件 4 の評価（ボトルネック I/O システムコールからの呼出し元コード特定）

(1) ボトルネックとなった I/O システムコールの特定（機能 3 の評価）

表 7 の処理時間分布から処理 3 が他の処理よりも 1,000 倍以上の時間を要しているため，処理 3 の I/O システムコール毎に処理時間を確認する（図 5 参照）．ファイルの書き込み処理（pwrite 部分）の処理時間が長いことがわかる.

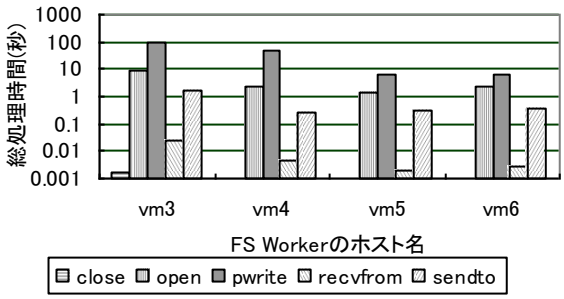

図 5 処理 3 における I/O システムコール毎の処理時間

最も総処理時間が長い書き込み処理のファイルパスはログファイルであることが確認できた．また Client 毎の書き込み回数より，512（ = 1GiB(ファイルサイズ) ÷ 2MiB(データの読み出し単位)）リクエストの処理毎に 1,032～1,088 件のログファイルへの書き出し処理行われており，クライアント要求毎に 2 回以上ログを出力していることがわかった（図 6 参照）.

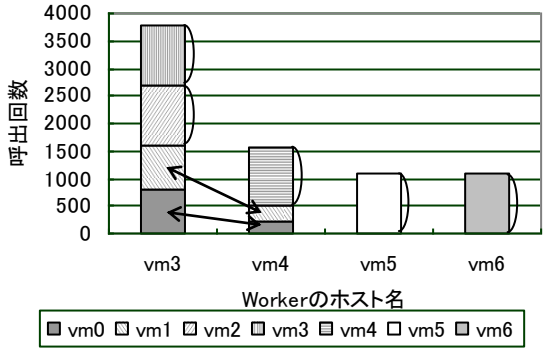

図 6 処理 3 による，Client 毎のファイル書き込み回数

(2) I/O システムコールから呼出し元コード位置の特定（機能 4 の評価）

上記(1)で特定したログファイルの書き込み処理に対して呼出し元コード位置が特定できるかを確認する．

処理 3 によるログファイルの書き込み処理に関して，I/O システムコールのトレースを分析した結果，当該ファイルへの書き込み処理に対する呼出し履歴が，表 8，表 9 に示す 2 種類であることが判明した．呼出し履歴を No 毎に比較すると，No. 1～No. 9 がすべて同一のコード位置を指しているため，ログ出力の共通フローとして考えられる．コード位置が異なる No. 10 が呼出し元コード位置として挙げられ，No. 10 のデバッグ情報を参照してソースコード位置を特定し，実際にソースコードを確認すると，両者ともにログ出力のコードが記載されていることが確認できた．

表 8 ログファイル書き出し処理の呼出し履歴(1)

No.	ファイル名	コード位置
1	glibc-2.5.so	0xc725b
2	libapr-1.so.0.2.7	0x13040
3	liblog4cxx.so.10.0.0	0x104891
4	liblog4cxx.so.10.0.0	0x17c025
5	liblog4cxx.so.10.0.0	0xd630d
6	liblog4cxx.so.10.0.0	0xd36ca
7	liblog4cxx.so.10.0.0	0x12253b
8	liblog4cxx.so.10.0.0	0x12343e
9	worker	0xfe8b6
10	**worker**	**0x86ada**
...		

表 9 ログファイル書き出し処理の呼出し履歴(2)

No.	ファイル名	コード位置
1	glibc-2.5.so	0xc725b
2	libapr-1.so.0.2.7	0x13040
3	liblog4cxx.so.10.0.0	0x104891
4	liblog4cxx.so.10.0.0	0x17c025
5	liblog4cxx.so.10.0.0	0xd630d
6	liblog4cxx.so.10.0.0	0xd36ca
7	liblog4cxx.so.10.0.0	0x12253b
8	liblog4cxx.so.10.0.0	0x12343e
9	worker	0xfe8b6
10	**worker**	**0x86ba4**
...		

5.3 解析の応用例（データ先読みの分析）

分散ファイルは，巨大なファイルを効率よく管理するためにファイルの読み出し時に予めデータを先読みする方式を採用している．本提案方式を用いた解析の応用例として，ファイル読み出し時の先読みに対して分析可能であることを示す．

分散ファイルの Client と Worker 間の Network 送信の転送サイズと呼出し回数の関係を図 7 に示す．横軸は 1 回の I/O システムコールで転送されたデータサイズ，縦軸は転送データサイズ毎の I/O の発行回数である．データの総転送サイズは，9.1×10^8 Bytes であり，想定量（5.8×10^7 Bytes = 16 KiB × (1 GiB / 2 MiB) × 7）の 16 倍であった．これは，Worker が Client へ先読みしたデータを Network 転送していることがわかる．また，Network 送信時の転送サイズは 256 KiB（図 7 での一番右側にある●部分が該当），発行回数が 3,584 = (1 GiB / 2 MiB) × 7 であり，ファイルからの読み出しに必要なサイズの 16 倍と一致するため，分散ファイルでのファイル読み出し時には，先行して 256 KiB のデータを Network 転送していることが判明した．

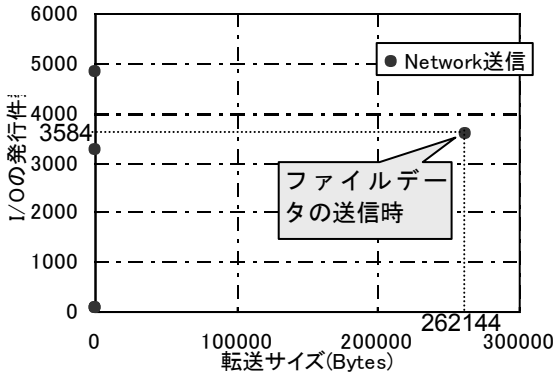

図 7 Client・Worker 間のデータ送信時のサイズと件数

6. まとめと今後の課題

分散処理システムにおいて，I/O システムコールをトレースして性能情報や呼出し履歴を収集・分析することにより，ボトルネックの発生箇所を特定する方式を提案した．また，提案した方式を実装し分散処理システムで試験を行い，トレースによる性能低下を抑止しながらボトルネックの原因となるコード位置が特定できることを確認した．

一方，分散ファイルからの読み出し試験にも係わらず，処理 3 によるファイルの読み出し処理が検出されない点が課題として残った．処理 3 では，ファイルの読み出し処理は `mmap()` でディスクキャッシュとマップし，メモリーからデータを読み出す方法で実装している．データを読み出す時にディスクキャッシュがマップされていない場合，カーネルはファイルからディスクキャッシュにデータを読み出すが，ディスクキャッシュへの読み出し処理の考慮が必要であることがわかった．

今後は，I/O システムコールのログ分析に関するノウハウを蓄積し分析処理を自動化すること，ボトルネック箇所が `mmap()` でマップされた領域を読み出すコード位置，および I/O システムコール以外のコード位置に存在するケースへの対応などについて取り組んでいきたい．

参考文献

[1] 鷲坂, 中村, 高倉, 吉田, 冨田: 大量データ分析のための大規模分散処理基盤の開発, NTT 技術ジャーナル, Vol.23, No.10, pp.22–25, (2011).
[2] Chang, Fay. Dean, Jeffrey. Ghemawat, Sanjay. Hsieh, Wilson C. Wallach, Deborah A. Burrows, Mike. Chandra, Tushar. Fikes, Andrew and Gruber, Robert E.: Bigtable: a Distributed Storage System for Structured Data, Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation, USENIX Association, (2006).
[3] The Apache Software Foundation: Apache HBase Project, http://hbase.apache.org/.
[4] Ghemawat, Sanjay. Gobioff, Howard. and Leung, Shun-Tak.: The Google File System, Proc. of the 9th ACM Symposium on Operating Systems Principles, ACM, pp.29–43, (2003).
[5] The Apache Software Foundation: Hadoop Distributed File System, http://hadoop.apache.org/hdfs/.
[6] Mike Burrows: The Chubby lock service for loosely-coupled distributed systems, Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation, USENIX Association, (2006).
[7] Benjamin H. Sigelman, Luiz Andr´e Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. In Google Technical Report dapper-2010-1, (2010).
[8] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, Athicha Muthitacharoen: Performance Debugging for Distributed Systems of Black Boxes. In the proceedings of the 19th ACM Symposium on Operating Systems Principles, (2003).
[9] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, Masami Hiramatsu: Probing the Guts of Kprobes, Ottawa Linux Symposium, Volume 2, pp.101-118 (2006).
[10] Steven Rostedt: Ftrace Linux Kernel Tracing, LinuxCon Japan, (2010).
[11] Roland McGrath: utrace: a new in-kernel API for debugging and tracing user tasks, Linux Foundation Collaboration Summit, (2009).
[12] DWARF Standards Committee: The DWARF Debugging Standard, http://dwarfstd.org/