Vol.2017-OS-139 No.8
2017/3/1

仮想計算機を利用した性能プロファイリングシステムの分散化

山本 昌生[1,2]　中島 耕太[2]　山内 利宏[1]　名古屋 彰[1]　谷口 秀夫[1]

概要：計算機の性能異常発生時の要因調査手段として，性能プロファイリングが広く利用されており，我々は，性能異常環境における単発実行のデータ収集と解析を VM モニタで行う性能プロファイリング手法を提案した．しかし，動作中計算機の性能異常を迅速に検出するには，収集データを定期的に格納し解析できる必要がある．また，データ収集中のプロセスの生成や終了を捉えることも重要である．そこで，本稿では，継続的なプロファイリングを行うための構成として，提案した性能プロファイリングシステムの分散化について述べる．

1. はじめに

性能プロファイリングは，計算機の性能異常発生時の要因調査手段として有用である．性能プロファイリングは，プログラムの動作情報として命令アドレスやプロセス ID（PID）などを一定周期で収集し，命令ブロック単位や関数単位，プロセス単位などで頻度集計する統計手法である．これにより，プログラム中の hotspot を容易に知ることができる．例えば，プログラムのどこで CPU 時間の多くを消費しているかや，キャッシュミスなどの性能イベントがどこで多く発生しているかについて知ることができる．性能プロファイリング手法の中でも，CPU が備える性能監視カウンタ（PMC）ベースの手法が多く提案されている．具体的には，データ収集契機として PMC のカウンタオーバフロー割込み機能を用いる手法で，低負荷のためによく利用されている．例えば，Intel VTune [1]，Oprofile [2]，Linux perf [3] がある．この様に，性能プロファイリングはソフトウェアシステムの性能異常の要因箇所の調査手段として広く利用され必要不可欠となっている．しかし，クラウド基盤などで広く普及してきた仮想計算機（VM）では，この様な性能プロファイリングが有効に機能しなくなっている．

VM に対しても，先に述べた性能プロファイリングと同様に実用的な低負荷で利用できるプロファイリングシステムを確立する必要がある．そこで，我々は，データ収集と解析を VM モニタ（VMM）で行い，1 ミリ秒周期のデータ収集中の負荷が 1%以下となる性能プロファイリング手法 [4] を提案した．しかし，この VM を利用した性能プロファイリング手法は，性能異常環境における単発実行（1 回）のデータ収集と解析しか行っていない．一方，VM による動作中の計算機では，性能異常発生が断続的な場合もある．したがって，単発実行のプロファイリング手法では動作中計算機の性能異常を迅速に検出できない．また，既手法はデータ収集中のプロセスの生成や終了について考慮していない．具体的には，データ収集中に終了したプロセスのプロセス情報を格納していないため，解析の精度が高いとは言えない．

そこで，本稿では，動作中計算機の性能異常を迅速に検出するための継続的なプロファイリングについて説明し，提案した性能プロファイリングシステムの分散化について述べる．具体的には，収集したデータを別計算機に格納し，データ収集・格納と解析を連続実行することにより，迅速な性能異常検出と要因解析を行うシステムを構築する．また，プロセス情報の格納を，データ収集中のプロセス終了時とデータ収集終了時の複数の契機で行う方式について説明する．さらに，初期的実験として，定期的なデータ格納によるオーバヘッドの評価について述べる．

2. 仮想計算機を利用した性能プロファイリングシステム [4]

2.1 構成

図 1 にプログラム構成とデータ収集の仕組みを示す．これは，一台の物理計算機（ホスト）と一台の VM（ゲスト）

[1] 岡山大学大学院 自然科学研究科
Graduate School of Natural Science and Technology, Okayama University
[2] (株) 富士通研究所 コンピュータシステム研究所
Computer Systems Laboratory, Fujitsu Laboratories Ltd.

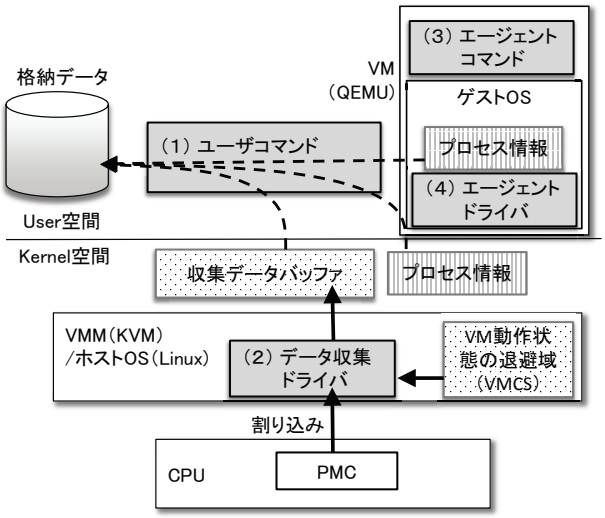


図 1 プログラム構成とデータ収集の仕組み

で構成された仮想化環境の例である．

プログラムは，以下の 4 つに分類できる．

(1) ユーザコマンド
- (a) データ収集ドライバの導入と削除
- (b) データ収集ドライバへのデータ収集実行指示
- (c) 収集データやプロセス情報などのディスクへの格納
- (d) 解析処理

(2) データ収集ドライバ
- (a) PMC や収集データバッファの初期設定
- (b) プログラム動作情報のデータ収集

(3) エージェントコマンド
- (a) ゲスト上のプロセス情報の格納

(4) エージェントドライバ
- (a) ゲスト上の CR3-PID 情報の格納

ユーザコマンドは，データ収集ドライバに対して，導入（ロード）や削除（アンロード），およびデータ収集の実行開始の指示を行う．データ収集ドライバのロード時には，収集データバッファサイズの指定も行う．また，データ収集の開始指示では，収集周期や収集時間の指定も行う．さらに，データ収集終了時に，エージェントコマンドに対して，プロセス情報やオブジェクトファイルなどの各データのディスクへの格納指示を ssh コマンドにより発行し，かつホスト上でも収集データやホスト上のプロセス情報などの各データのディスクへの格納および解析処理を行う．

データ収集ドライバは，VMM 内にあり，指定されたパラメータに従った初期設定とプログラムの動作情報のデータ収集を行う．初期設定では，指定バッファサイズに基づいた収集データバッファの確保や，収集周期トリガとなる CPU が内蔵している性能カウンタ（PMC）へのカウントベースイベントと初期カウンタ値の設定や，収集時間に基づいたタイマの設定を行う．データ収集では，データ収集トリガ発生時に動作していたプログラムの動作情報を収集する．ゲストに関する動作情報は VM 動作状態の退避域から入手する．

エージェントコマンドは，データ収集終了時に，ホスト上のユーザコマンドからの指示により，ゲスト内のプロセス情報やオブジェクトファイルなどの各データを ssh コピーコマンドによりホスト上のローカルディスクへ格納する．また，エージェントドライバに対し，CR3-PID のマッピング情報格納の指示をだす．CR3 は Intel CPU のコントロールレジスタ 3 で，ページテーブルアドレスを保持している．ページテーブルアドレスはプロセスに一意な値である．

エージェントドライバは，Linux カーネルの各プロセスのタスク構造体から CR3 と PID の値を対にして格納する．

以下に，具体的なデータ収集処理と収集データの流れを示す．

(i) PMC のカウンタオーバフロー割込み機能を利用してデータ収集のトリガとなる割込みを定期的に発生させる．

(ii) 割込み発生毎に，VMM 内のデータ収集ドライバがデータ収集を行い，メモリ上の収集データバッファに時系列で全て記録する．

(iii) データ収集終了時に，ユーザコマンドがカーネル空間の収集データバッファを Read システムコールで読み込んで時系列の記録形式のままディスクに格納する．

(iv) さらに，この時点で動作している全プロセスのプロセス情報もディスクに格納しておく．

例えば，1 ミリ秒周期（収集周期）で 60 秒間（収集時間），物理 CPU 毎に周期的にデータ収集が行われる．データ収集中は，メモリ上に収集データを時系列で全て記録保持し，収集終了時に記録形式のままディスクに書き出す．

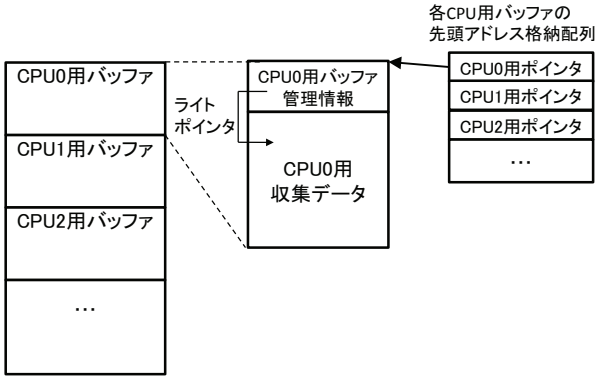


図 2 収集データバッファ構造

図 2 に収集データバッファの内部構造を示す．バッファ領域は物理 CPU 毎の専用領域に均等に分割して利用する．さらに，各物理 CPU 用バッファは，管理情報用領域と収集データを保持しておくための領域に分けて利用する．以

下に，カウンタオーバフロー割込み発生毎のこのバッファ仕様の利用手順を示す．

（1）先ず，自 CPU 番号を OS から取得する．
（2）バッファ先頭アドレス格納配列から自 CPU 用バッファポインタを取得する．
（3）自 CPU 用バッファの管理情報から，ライトポインタを取得する．（ライトポインタの初期値はバッファ先頭アドレス）
（4）ライトポインタ値にこれから書き込むデータのエントリサイズを足して，自 CPU 用のバッファ域をオーバしないことを確認する．（もしオーバしていたら，他 CPU もあわせてデータ収集を終了する）
（5）収集データを収集順でバッファに記録する．
（6）管理情報のライトポインタを更新する．

2.2 データ

データ収集の終了直後に以下のデータ（A）～（E）をプロファイリングデータとしてディスクに格納する．データ（A）はホスト上のみで，データ（B）～（E）はホスト上とゲスト上から格納する．

（A）収集データ
（B）プロセス情報
（C）オブジェクトファイル
（D）カーネルのシンボルマップ
（E）実行条件や環境情報など付帯情報

収集データは，データ収集ドライバが周期的に収集するプログラム動作情報で，命令アドレス（IP）やプロセス ID（PID）などを含む．収集データの具体的な内容を**表 1** に示す．データ収集ドライバは，これらのデータを時系列で収集データバッファに記録していく．収集データバッファは，図 2 のバッファ構造で示した様に，収集データ用の領域とバッファ管理情報用の領域とで構成されている．バッファ管理情報の具体的な内容を**表 2** に示す．

プロセス情報は，データ収集終了時に存在する各プロセスの情報で，PID やプロセス名，ロードされているプログラムのメモリマップ情報やオブジェクトファイルのファイルシステム上のパス情報を含む．また，ゲストの場合は，CR3-PID のマッピング情報もプロセス情報（B）に加える．Linux では，CR3-PID のマッピング情報はカーネルが管理している各プロセスのタスク構造体から，それ以外のプロセス情報はディレクトリの */proc/*<プロセス *ID*>*/* 配下のファイルから入手できる．

オブジェクトファイルは，プロセス情報（B）に含まれる各プロセスのオブジェクトファイルである．即ち，データ収集中に動作していたプログラムの実行バイナリファイルで，プロセス情報（B）から得たパス情報を使ってファイルシステム上からコピーしたものである．このオブジェクトファイル（C）は，ユーザプログラムの関数シンボル名を抽出するために使用する．

カーネルのシンボルマップは，動作中のカーネルのシンボルマップファイルで，カーネルの関数シンボル名とメモリマップ情報となる．Linux では */boot/System.map-*<カーネルバージョン> ファイルとして入手できる．

付帯情報は，収集周期や収集時間などの実行条件や動作環境の情報など参考情報である．

表 1 収集データ（size 単位：バイト）

名前	size	説明
Host_IP	8	ホスト上の命令アドレス
Host_Thread_ID	4	カレント・スレッド ID
Host_PID	4	カレント・プロセス ID
Host_Return_IP1	8	戻りアドレス 1
Host_Return_IP2	8	戻りアドレス 2
TSC	8	タイムスタンプ
vPROCESSOR_ID	8	CPU 管理の仮想 CPUID
Guest_IP	8	VM 上の命令アドレス
Guest_CR3	8	VM 上のページテーブルアドレス
VMEXIT_REASON	8	VM_EXIT 要因番号
VMEXIT_INTRINFO	8	VM_EXIT 割込み情報
(reserved)	16	予備

表 2 バッファ管理情報（size 単位：バイト）

名前	size	説明
Buffer ID	8	バッファフォーマット識別子
CPUID	8	実行 CPUID
IntCount	8	割込みカウンタ
Write_pointer	8	収集データの書き込み位置アドレス
Start_Tsc	8	収集開始時タイムスタンプ
End_Tsc	8	収集終了時タイムスタンプ

2.3 解析

解析処理は，2 つの処理に分けられる．一つはシンボル解決処理で，もう一つはシンボル単位での頻度集計処理である．

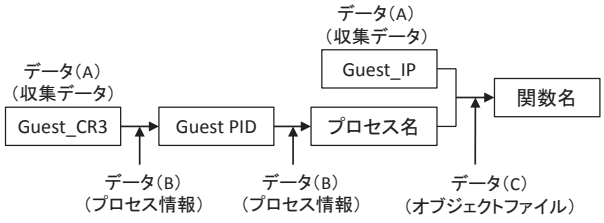


図 3 シンボル解決処理の流れ

シンボル解決処理は，収集データの PID や命令アドレスなどの数値データを，対応するプロセス名や関数名に変換する処理である．シンボル解決は，2.2 節で説明したデー

タを利用して行われる．**図 3** に，VM 上のユーザプログラムの関数シンボル名を解決する流れを示す．先ず，プロセス情報（B）に含まれる CR3-PID マッピング情報を参照しながら，収集データ（A）に含まれる Guest_CR3 の値をゲスト上での PID に変換する．次に，該当プロセスのプロセス情報から該当 PID のプロセス名やオブジェクトファイルを特定する．さらに，プロセス情報に含まれるプログラムのロードアドレス情報（ベースアドレス）とオブジェクトファイルから抽出できる関数シンボル名と関数の相対アドレス情報から，該当プログラムの関数の絶対アドレスマップが作成できる．この絶対アドレスマップを参照しながら収集データ（A）に含まれる Guest_IP を関数名に変換する．また，どのゲストか，つまりどの VM かは収集データ（A）に含まれる Host_PID と Host_IP で特定する．KVM では，一つの VM がホスト OS 上の一つのプロセスに相当する．Host_IP はゲストが動作中だったのか，VMM やエミュレーションなどホスト処理が動作中だったのかの切り分けに使う．VM 上のどの vCPU で動作していたかは収集データ（A）に含まれる vPROCESSOR_ID で特定する．これらの処理により，収集データがどの VM のどのプログラムのどの関数処理だったのかの対応付けが可能となる．

シンボル解決ができれば，シンボル毎の収集回数を集計し，頻度順に並べたリストをプロファイリング結果として出力する．**表 3** にプロファイリング結果の出力例を示す．この結果は，ゲスト上で SPEC CPU2006 [5] で利用されている libquantum 0.9.1 [6] ベンチマークプログラム実行中に，収集周期 1 ミリ秒，収集時間 30 秒で収集したデータを基に解析した結果である．収集データを関数粒度で頻度集計し，高頻度順に表示している．表示内容として各関数の収集データ数（Samples）と CPU 使用率（%CPU）と関数名（Function）とモジュール名（Module）を出力している．この例では，上位 3 つの関数で 90%を占めており CPU 時間のほとんどを消費していることがわかる．

表 3 プロファイリング結果の出力例
Total samples:29996　OS:USER:steal = 1.49%:93.47%:5.04%

Samples	%CPU	Function	Module
16984	56.62	quantum_toffoli	libquantum
6752	22.51	quantum_sigma_x	libquantum
3330	11.10	quantum_cnot	libquantum
1511	5.04	[steal]	(outside)
766	2.56	quantum_swaptheleads	libquantum
198	0.66	quantum_objcode_put	libquantum
28	0.09	apic_timer_interrupt	vmlinux
22	0.07	native_apic_mem_write	vmlinux
17	0.06	__run_hrtimer	vmlinux
17	0.06	pvclock_clocksource_read	vmlinux

3. 継続的なプロファイリング

3.1 既プロファイリングシステムの問題

既プロファイリングシステム [4] は，性能異常発生時に原因を調査する手段であるため，性能異常を検知することは想定していない．動作中計算機での性能異常の迅速な解決のためには，2 つの問題がある．

第一の問題は，迅速な性能異常検出ができないことである．既システムでは，性能異常環境において単発実行でデータ収集してから解析を行うため，動作中計算機の性能異常の迅速な検出ができない．そこで，動作中計算機の性能異常を迅速に検出するために，常時データ収集しその場ですぐに解析できることが課題となる．例えば，複数のゲスト環境が一台の物理ホストに共存している様な環境では，他のゲストからの影響による性能異常の場合がある．その様なケースでは，何も対処しなくても性能異常がなくなったり再発したりと異常発生が断続的な場合がある．そのため，性能異常を迅速に検出できる必要がある．

第二の問題は，データ収集中に終了したプロセスの情報を取得できないことである．この問題の影響は，プロセスの生成や終了の多発時に大きい．プロセス情報は，該当プロセスが生存中にしか存在しない．対して，既システムではデータ収集終了時にしかプロセス情報を格納しないため，データ収集中に終了したプロセスのプロセス情報は格納できない．データ収集中に終了したプロセスが多ければ多いほど，より多くのプロセス情報を格納できずに取りこぼすこととなる．プロセス情報が格納できなかったプロセスは，プロセス名や関数名などのシンボル解決ができずに解析不能（unknown）となる，よって，格納できなかったプロセス情報が多いと，収集データのうち unknown が占める割合が多くなり正しい解析ができなくなる．そこで，データ収集中に終了するプロセスについて，プロセス情報が破棄される前に格納することが課題となる．

また，継続的なプロファイリングには，以下のことが求められる．

（要件） データ収集や格納による性能低下が小さいこと
具体的には，1%以下のオーバヘッドに抑えることを目標とする．

3.2 継続的なプロファイリングによる対処

先ず，3.1 節の第一の問題に対し，継続的なプロファイリング手法による解決を検討する．既プロファイリングシステムは，一回のデータ収集しか行っておらず，性能異常計算機において以下の様な処理の流れとなる．

＜既プロファイリング処理＞
（1）ユーザから指定された収集周期と収集時間でメモリ上の内部バッファへのデータ収集を一回行う．
（2）収集後に収集データを含む 2.2 節で挙げたデータ

（A）～（E）をプロファイリングデータとしてディスクに格納する．

（3）格納したデータを基に解析処理を行う．

対して，継続的なプロファイリングでは，動作中計算機において，以下の様に滞りなくプロファイリング処理を連続実行する．

＜継続的なプロファイリング処理＞

（1）ユーザから指定された収集周期と収集時間（格納周期）でメモリ上の内部バッファへのデータ収集を行う．

（2）収集後にデータ（A）とデータ（B）をディスクに格納する．

以降，（1）と（2）の処理を繰り返す．

（3）格納データを基に定期的な解析処理を行う．

この継続的なプロファイリングシステムにより，性能異常の迅速な検出を可能とし，一つ目の問題を解決する．ただし，3.1 節の要件（目標 1%以下のオーバヘッド）を満たすために，異常検知対象の計算機での処理を必要最小限とする必要がある．少なくとも上で述べた継続的プロファイリングの処理（1）のデータ収集処理は異常検知対象の計算機で実行する必要がある．一方，処理（3）の解析処理はデータがあれば別計算機でも実行可能である．このため，データを別計算機に格納し，別計算機で解析処理を行うプロファイリングシステムの分散化を検討する．つまり，処理（3）の解析処理を，処理（1）のデータ収集している動作中計算機とは別計算機に分離する．解析処理の別計算機への分離にともない，処理（2）のデータ格納では，データ収集を行っている動作中計算機から別の解析処理計算機にネットワークを介してデータを格納することとする．データの格納は，プロファイリングデータにノイズ（格納処理の影響）が載ることを防ぐために，収集毎ではなくまとめて定期的に行うこととする．さらに，定期的に格納するデータは 2.2 節で挙げたデータ（A）の収集データとデータ（B）のプロセス情報のみに制限する．データ（A）とデータ（B）はプログラムの動作や状態に関する動的なデータである．対して，その他のデータ（C）～データ（E）は静的な情報であり予め入手しておけばよい．プロファイリングシステムの分散化の基本構成は，次章の 4.1 節で示す．

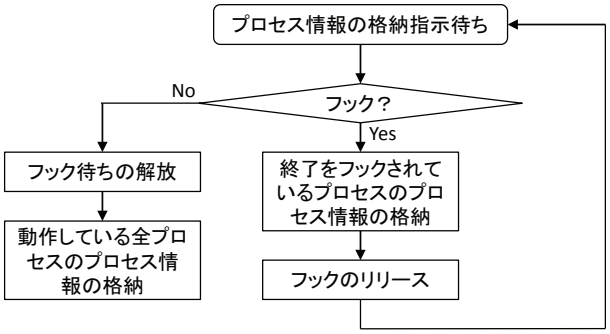


図 4　ゲスト上でのプロセス情報の格納処理の流れ

次に，3.1 節の第二の問題への対処について考える．既プロファイリングシステムではデータ収集終了時にしかプロセス情報を格納しないため，データ収集中に終了したプロセスのプロセス情報は格納できない．一方，データ収集前にプロセス情報を格納するようにしても，今度はデータ収集中に生成されたプロセスのプロセス情報が格納できなくなる．また，データ収集の前後でプロセス情報を格納するようにしても，データ収集中に生成し終了するプロセスには対処できない．結局，データ収集中のプロセスの生成または終了を捉えることが重要となる．

そこで，我々は，Linux カーネルが備えているプロセス終了処理のフック機能に着目し，データ収集後のこれまで通りのプロセス情報の格納処理に加えて，データ収集中に終了しようとするプロセスのプロセス情報の格納処理も行うことを提案する．ゲスト上でのプロセス情報の格納処理の流れを**図 4** に示す．データ収集中にプロセス情報の格納指示を待っているエージェントコマンドが，プロセス終了処理のフックで起こされた場合，フックハンドラ（エージェントドライバ）が終了プロセスの PID を採取し，エージェントコマンドへ戻り値として渡す．エージェントコマンドは戻り値を基に，終了しようとしているプロセスのプロセス情報を格納する．格納後はエージェントドライバを呼出し，フックのリリースつまり一時停止しているプロセス終了処理を再開させ，プロセス情報の格納指示待ちに戻る．一方，待ち中のエージェントコマンドが，フック以外具体的には終了指示で起こされた場合，フック待ちを解放し，その時動作している全プロセスのプロセス情報を格納する．これにより，データ収集中に生成または終了または生成終了する全てのプロセスを捉えることが可能となる．プロセス終了処理のフック機能では，フック時にコールバックされるハンドラ関数を予め登録しておくことにより，カーネルのプロセス終了処理である *do_exit()* 関数の先頭で処理が一時停止され，登録したハンドラに実行制御が移行する．つまり，ハンドラから処理が戻るまでプロセスの終了処理は待たされる．したがって，プロセス情報が削除される前に終了しようとするプロセスのプロセス情報をハンドラで格納可能となる．

4. システムの分散化

4.1 分散システムの基本構成

分散システムは，複数の計算機を通信路で結び，各計算機が通信路を介して相互に処理を進めていくシステムである [7]．**図 5** に，2 章で説明した仮想計算機を利用した性能プロファイリングシステムの分散化の構成例を示す．

先ず，2.1 節で述べた 4 つのプログラムのうち，ユーザコマンドの解析処理機能のみ，プロファイリング対象マシンから解析処理計算機に分離する．次に，2.1 節のデータ収集処理（iii）（iv）でのデータの格納先を，プロファイリ

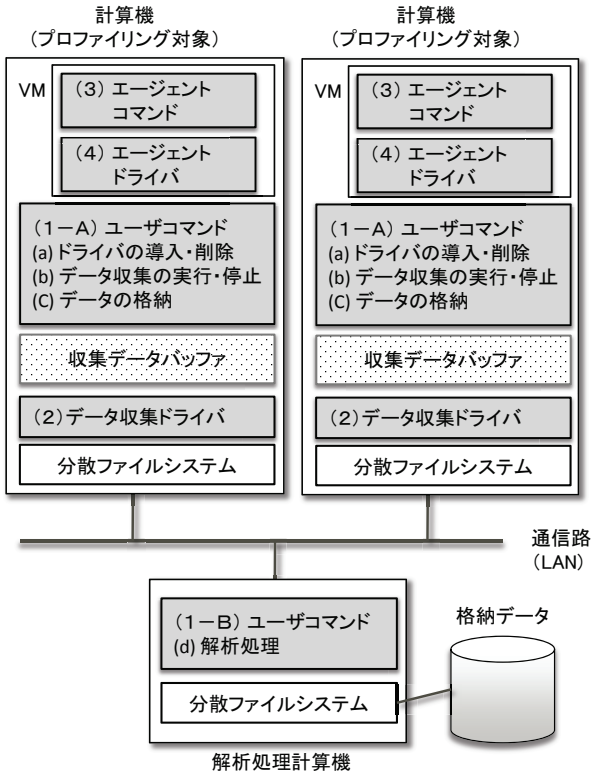


図 5 プロファイリングシステムの分散化の構成例

ング対象マシン内のローカルディスクから解析処理計算機のリモートディスクへと変更する．リモートディスクへの格納は分散ファイルシステムを基盤ソフトウェアとして用いる．また，通信路としては LAN（Local Area Network）を利用する．

4.2 データ格納を含めたプロファイリングのオーバヘッド

本節では，ユーザプログラムの処理時間の増加により，周期的なデータ格納によるオーバヘッドを評価する．一定時間内において，データ格納の周期が短い場合は格納周期が長い場合に比べて，一回あたりの格納データ量は減らせるが格納回数は増える．一方，格納周期が長いと一回あたりのデータ量は増えるが格納回数が減らせる．本評価によって，どちらの方がデータ収集環境に対する負荷影響を抑えることができるのかについての指針を示す．さらに，実用要件として我々が常に目標としている 1%以下の負荷になる条件も示す．

実験環境と測定条件を**表 4** に示す．データ格納は，**図 6** に示す通り，分散ファイルシステムを用いてネットワークを介したリモートディスクへ行う．ネットワークは，1Gbps のイーサネットを用いる．周期毎に格納するデータは 2.2 節のデータ（A）の収集データとデータ（B）のプロセス情報となる．なお，不確定要素をなくすために，Intel CPU の Hyper Thread，Turbo mode，Speed Step の各機能は無効としている．さらに仮想 CPU（vCPU）はそれぞれ重

表 4 実験環境と測定条件

ホスト環境	
CPU	Intel Xeon E5-2697v3 2.60GHz 14 コア x1
Memory	24GB
OS	kernel 3.10.0-327.28.2.el7.x86_64 （CentOS Linux release 7.2.1511 64bit）
VMM	qemu-kvm-0.12.1.2
ゲスト環境	
VM 数	1
vCPU 数	2
Memory	4 GB
OS	kernel 3.10.0-327.28.2.el7.x86_64 （CentOS Linux release 7.2.1511 64bit）
測定条件	
収集周期	1 ミリ秒
格納周期と データサイズ	20 秒（35MB），30 秒（48MB）， 60 秒（86MB），120 秒（163MB）

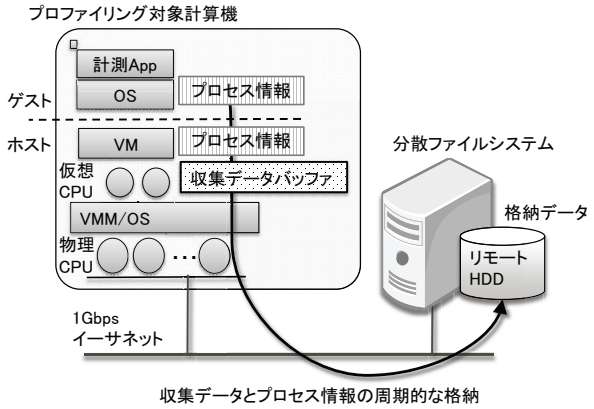
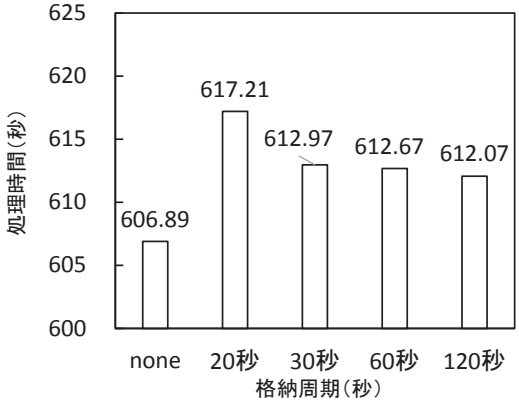


図 6 分散システムへのデータ格納評価環境の構成

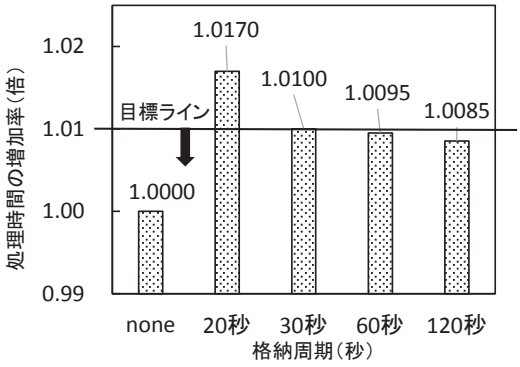
ならない特定の物理 CPU（pCPU）に固定している．

データ格納のオーバヘッドは，計測用アプリケーション（図 6 の計測 App）の処理時間の増加率から算出する．計測 App として文献 [4] と同じ姫野ベンチマーク [8] に含まれる jacobi 関数を用いる．計測 App の処理時間は以下の方法で測定する．

(1) jacobi 関数は固定回数繰り返し実行する．回数はプロファイル測定無しの時に約 10 分間かかる回数を事前測定で決めておく．
(2) 計測 App は VM 上から特定の pCPU に固定して実行する様にする．具体的には，計測 App を特定 vCPU に固定して実行投入する．vCPU は特定の物理 CPU に固定しておく．
(3) データ格納処理を計測 App とは別の特定の pCPU に固定し実行する様にする．
(4) 定期的なデータ格納を伴う継続プロファイリングを先に開始してから直後に計測 App を実行投入する．測定中には，収集時間 10 分間のプロファイリングデータの格納が発生する．



(a) 処理時間



(b) オーバヘッド

図 7 データ格納を含めたプロファイリングのオーバヘッド

(5) 計測 App で，jacobi 関数の固定回数の実行時間を計測し処理時間の測定結果として出力する．

図 7(a) に処理時間の測定結果を示す．棒グラフは左から順に，プロファイル収集なしの場合（none），格納周期 20 秒の場合（処理時間測定中の格納発生回数 30 回），格納周期 30 秒の場合（処理時間測定中の格納発生回数 20 回），格納周期 60 秒の場合（格納発生回数 10 回），格納周期 120 秒（格納発生回数 5 回）となる．この図 7(a) を見ると，jacobi 関数の一定実行回数における処理時間は，格納周期が長い方が影響が小さくなる．次に，図 7(b) にプロファイル収集なしの場合を基準にした処理時間増加率（オーバヘッド）の結果を示す．オーバヘッドはそれぞれ，格納周期 20 秒の時で 1.70%，格納周期 30 秒の時で 1.00%，格納周期 60 秒の時で 0.95%，格納周期 120 秒の時で 0.85%となっている．格納周期 30 秒以上で低負荷要件の 1%以下を満たす．

さらに，データ格納のために，データ収集が停止している時間を CPU のタイムスタンプカウンタ（TSC）で計測した結果を**図 8** に示す．これを見ると，データ格納 1 回あたりのデータ収集停止時間はそれぞれ，格納周期 20 秒の時で 1.18 秒，格納周期 30 秒の時で 1.23 秒，格納周期 60 秒の時で 1.68 秒，格納周期 120 秒の時で 2.38 秒となっている．データ収集停止時間は，格納周期が短い方が影響が

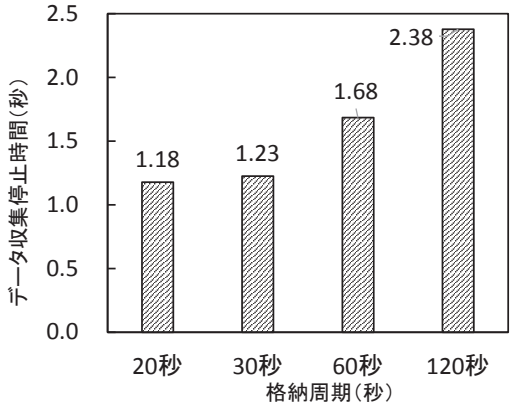


図 8 データ格納 1 回あたりのデータ収集停止時間

小さくなる．よって，オーバヘッドとデータ収集停止時間は，トレードオフの関係となる．

また，格納周期が長くなると，解析処理の開始が遅れるため性能異常の検出が遅れる．よって，オーバヘッドと性能異常検出の迅速さもトレードオフの関係となる．

これらの結果より，次のことがわかる．

(A) 格納周期は長い方が，オーバヘッドは小さくなる．
(B) 低負荷要件の 1%以下に抑えるためには，格納周期 30 秒以上が条件となる．
(C) 一方で，格納周期が短い方が，データ収集停止時間は短くなる．
(D) よって，オーバヘッドとデータ収集停止時間はトレードオフの関係となる．
(E) さらに，オーバヘッドと異常検出の迅速さもトレードオフの関係となる．

従って，格納周期は，トレードオフを理解した上で要件にあわせて決める必要がある．我々の要件の場合は，30 秒から 60 秒あたりが妥当と考える．さらに，格納周期 30 秒の場合と 60 秒の場合のデータ収集のカバレッジを，データ収集時間/(データ収集時間＋データ収集停止時間) の式で求めると，30 秒の場合が 96.1%，60 秒の場合が 97.3%となり，60 秒の方が大きい．よって，今後は，本評価環境における最適な格納周期（収集時間）として 60 秒を選択する．この様にして，異なる環境においても最適な格納周期（収集時間）を決めることができる．

5. 関連研究

プロファイリングデータの継続収集の関連研究として，DCPI （DIGITAL Continuous Profiling Infrastructure）[9] と GWP（Google-Wide Profiling）[10] がある．共通の特徴として，いずれもデータ収集を継続して行い，データベースでデータを管理し，ユーザからの要求に応じて必要な解析結果を提供するというサービスシステムとなっている．

個々の特徴として，先ず DCPI は，1990 年代に Alpha

プロセッサと DIGITAL Unix をベースとしたシステム向けに DEC 社が開発したプロファイリングシステムである．プロセッサの性能カウンタのオーバフロー割込みを用いたデータ収集を行う．基本仕様として 10 分周期でメモリ上の収集データをユーザが指定したディレクトリのデータベースへ格納する．データベースはネットワークを介して共有されるかもしれないと述べられている．プロセス情報はローダ（/sbin/loader）に手を加えて生成時に収集する仕組みを持つ．この点は，3.2 節で述べた我々の手法と比べて，データ収集中にプロセス情報を格納するという点では同じであるが，プロセス情報の格納契機がプロセスの生成時か終了時かという点で異なる．プロセスの生成時に格納できる方が，格納時に動作している全プロセスのプロセス情報格納は，定期的ではなく，最初に一度行えばよいので，継続収集時のデータ格納のオーバヘッドが低減できると推測する．Linux カーネルに動的にプロセス生成時にフックできる仕組みがないか今後調査する．ベンチマーク性能の劣化による DCPI の負荷は 1〜3%で，例えば SPECint95 [5] で約 2.0%と文献 [9] で述べられている．開発とメンテナンスは，DEC 社から COMPAQ 社，HP 社と継承されたが，Alpha プロセッサの終焉とともに 2005 年頃を最後に今はメンテナンスされていない．なお，DCPI のサブセット機能として Oprofile が派生している．DCPI は，現在多く提案されているカウンタベースのプロファイリング技術の源流といえる．Oprofile はオープンソースで開発されているが現在も HP 社が開発を支援している．

一方，GWP は，2000 年代に Google 社が DCPI を参考にして IA サーバシステム向けに開発したプロファイリングシステムである．Oprofile をベースに用いた自社データセンタ用のプロファイリングシステムで，現在も 2 万台以上のマシンを対象に実用中である [11]．

我々はデータ収集だけでなく，一歩進めて解析処理まで含めた連続実行により性能異常を迅速に検出し問題を早期に解決することに繋げたいと考えている．

6. おわりに

継続的プロファイリングを行うために，仮想計算機を利用した性能プロファイリングシステム [4] の分散化について説明した．さらに，データ格納を含めたプロファイリングシステムによるオーバヘッドとデータ格納によるデータ収集停止時間の評価を行った．その結果，オーバヘッドとデータ収集停止時間は，トレードオフの関係にあることを示した．また，1%以下の低負荷を要件とする場合は，トレードオフも考慮して 30 秒から 60 秒の格納周期が条件となることを示した．さらに，データ収集のカバレッジも考慮すると，格納周期は，60 秒が最適であることを示した．

残された課題として，解析処理も含めて滞りなく連続実行できるようにするため，解析処理時間の評価や高速化も行う．また，被測定計算機を複数台にし，規模に関する問題や設計指針を明らかにする．これらにより，仮想計算機を利用した性能プロファイリングシステムの分散化を実現する．これにより，例えば，Jubatus（ユバタス） [12] の様なオンライン機械学習型の異常検知エンジンにプロファイリング結果を入力し続け，クラウド環境での迅速な性能異常検出と解決を目指したいと考えている．

参考文献

[1] Intel: Intel VTune Amplifier, Intel Corp. (online), available from ⟨http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/⟩ (accessed 2016-12-12).
[2] Levon, J. and Elie, P.: Oprofile: A system profiler for linux, HP (online), available from ⟨http://oprofile.sourceforge.net⟩ (accessed 2016-12-12).
[3] perf: Linux profiling with performance counters, Red Hat, Inc. (online), available from ⟨https://perf.wiki.kernel.org/index.php/Main_Page⟩ (accessed 2016-12-12).
[4] Yamamoto, M., Ono, M., Nakashima, K. and Hirai, A.: Unified Performance Profiling of an Entire Virtualized Environment, *International Journal of Networking and Computing*, Vol. 6, No. 1 (2016).
[5] SPEC: SPEC CPU 2006, Standard Performance Evaluation Corporation (online), available from ⟨https://www.spec.org/cpu2006/⟩ (accessed 2016-12-12).
[6] Butscher, B. and Weimer, H.: the C library for quantum computing and quantum simulation, Libquantum (online), available from ⟨http://www.libquantum.de/⟩ (accessed 2016-12-12).
[7] 谷口秀夫：分散処理（IT Text シリーズ），オーム社 (2005).
[8] Himeno, R.: Himeno benchmark, RIKEN (online), available from ⟨http://accc.riken.jp/supercom/himenobmt/⟩ (accessed 2016-12-12).
[9] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A. and Weihl, W. E.: Continuous Profiling: Where Have All the Cycles Gone?, *ACM Trans. Comput. Syst.*, Vol. 15, No. 4, pp. 357–390 (1997).
[10] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S. and Hundt, R.: Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers (2010).
[11] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y. and Brooks, D.: Profiling a Warehouse-scale Computer, *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15 (2015).
[12] Jubatus: Distributed Online Machine Learning Framework, PFN & NTT (online), available from ⟨http://jubat.us/⟩ (accessed 2016-12-12).