Vol.2015-ARC-215 No.12
Vol.2015-OS-133 No.12
2015/5/27

オペレーティングシステムに対して透過的なメモリアクセス・トレース機構

広渕 崇宏[1]　高野 了成[1]　工藤 知宏[2,1]

概要：システムソフトウェア分野において不揮発性メモリ技術に関する研究を進めるため、オペレーティングシステムのメモリアクセス傾向を把握する手法を開発した。オペレーティングシステムを改変することなく、そのメモリ読み書き速度や読み書きメモリページ番号を記録できる。ハードウェアが備えるメモリアクセス監視機能を用いて既存ハイパーバイザーを拡張することで実現した。CPU インストラクションを逐次分析する方式と比べて遙かに高速に動作する。試作を通じて提案機構の基本的な動作を確認できた。

1. はじめに

システムソフトウェア分野において、次世代の不揮発性メモリを見すえた研究に関心が高まっている。今日メインメモリとして広く用いられている DRAM と違い、不揮発性メモリはデータを維持するのみであれば電力を消費しない。計算機システムにおいて不揮発性メモリを用いることで、DRAM のみではなしえなかった大容量の搭載メモリを少ない消費電力で実現できる可能性がある。

しかし、次世代の不揮発性メモリの登場は、システムソフトウェアに対して新たな研究課題を突きつけている。例えば、次世代の不揮発性メモリの代表格である STT-MRAM は、DRAM と同等の読み書き性能を実現できる可能性があるものの、書き込み時において DRAM 以上の消費電力を要する可能性が指摘されている [1]。単純に DRAM を MRAM で置き換えただけでは消費電力削減効果が得られない可能性がある。オペレーティングシステム (OS) のメモリアクセスの挙動を分析し、その結果に照らし合わせて不揮発性メモリを使いこなす仕組みを研究しなければならない。不揮発性メモリに対する研究をシステムソフトウェア分野で遂行するためには、OS のメモリアクセス傾向を容易に把握し、定量的な分析を可能とする仕組みが不可欠であると考える。

しかし、メモリアクセスを分析可能にする既存手法は、このような研究に対して容易に用いることが難しい。アプリケーションのメモリアクセスを分析する手法として、binary instrumentation が代表的である。実行コードの CPU インストラクションを逐次解析し、メモリの読み書きを伴うインストラクションをすべて記録できる。しかし、実行に際して計測対象プログラムの大きな性能低下を伴ってしまう。一般的には、ユーザランドで動作するプログラムを対象としたものであり、OS 全体のメモリアクセスを把握する手法ではない。gem5 [2] 等の full system emulator は、OS が発行するすべての CPU インストラクションをそのレイテンシまで含めてエミュレーションする。OS 全体のすべてのメモリアクセスを記録できる。しかしハードウェアの挙動を精密に模倣できる反面、対象の OS の動作は劇的に遅くなってしまう。実環境で動作するシステムを容易に対象とすることは難しい。

そこで我々は、システムソフトウェア分野において不揮発性メモリ技術に関する研究を進めるため、仮想マシン技術を応用して、OS のメモリアクセス傾向を容易に把握する手法を開発した。OS を改変することなく、そのメモリ読み書き量や読み書きメモリページ番号を記録できる。ハードウェアが備えるメモリアクセス監視機能を用いて既存ハイパーバイザーを拡張することで実現した。CPU インストラクションを逐次分析する手法が細粒度のメモリアクセス情報を取得するために実行速度が極めて遅くなるのに対して、我々の提案手法は粗粒度のメモリアクセス情報の取得に特化することで実行速度の低下を避けている。両者は相補的な関係であり、システムソフトウェアの研究においては後者であっても一定の有用性があると考えている。

本稿では、提案機構の試作を通じてその基本的な動作を確認できたことを報告する。2 節において要求事項をまとめ、3 節で提案手法を述べる。4 節で簡単な評価結果を述

[1] 国立研究開発法人 産業技術総合研究所 情報技術研究部門
National Institute of Advanced Industrial Science and Technology (AIST)
[2] 東京大学
University of Tokyo

べ、5 節で本稿をまとめる。

2. 要求事項

システムのメモリアクセスに関する情報は様々な方法で取得できる。しかし我々は、システムソフトウェアを対象とした研究においては、以下の要件を満たす手法が新たに必要になると考えた。

第一に、OS 全体のメモリアクセスを対象としなければならない。メインメモリ上には、アプリケーションがユーザ空間で管理するデータだけではなく、OS がカーネル空間内で管理するデータも存在する。システムソフトウェアにおけるメモリ管理を研究する上では、特定のアプリケーションだけではなく、カーネルも含めたシステム全体のメモリアクセスを分析できる必要がある。

第二に、メモリアクセスを計測する機構自体のメモリアクセスを計測対象から除外できる必要がある。メモリアクセスを監視して記録する処理において、メモリの読み書きが発生してしまう。計測処理に伴うメモリ読み書きの影響を排除した上で、研究対象を評価することが求められる。計測対象の OS 上の仕組みに依存しない計測機構が必要である。

第三に、計測にともなう性能低下が軽微である必要がある。計測対象のシステムが大幅に速度低下してしまうと、定量的評価がしばしば困難になる。様々なシステムを対象に容易にメモリアクセスを分析できることが望ましく、full system emulator のように評価に要する時間が極めて長くなる手法は好ましくない。

3. 提案手法

そこで我々は、メモリアクセス情報の新たな取得手法を提案する。ハードウェアが備えるメモリアクセス監視機構を利用することで、OS 全体のメモリアクセスを対象とした情報を軽微な性能オーバヘッドで計測できる。仮想マシン技術に基づいて、計測処理が伴うメモリアクセスを計測対象の OS のメモリアクセスから完全に分離する。OS のメインメモリに対する読み書き量の計測および読み書きページ番号の記録に特化することで、大きな性能低下を伴う CPU インストラクションの分析を不要とした。

提案手法は、OS のメインメモリに対する読み込みバイト数および書き込みバイト数を時系列で記録する。さらにメインメモリをいくつかの領域に分割し、それぞれの領域ごとに読み込みバイト数および書き込みバイト数も計測できる。例えば、不揮発性メモリの消費電力は、読み込みバイト数および書き込みバイト数がわかれば 消費電力特性をもとに概ね見積もることが可能である。提案手法を利用すれば、一例としてメインメモリの一部あるいはすべてを MRAM で置き換えた場合の消費電力を議論することができる。

さらに、読み込みあるいは書き込みが発生したメモリページ番号を記録する。例えば、過去 1 秒間において書き込みがあったメモリページのページ番号を 1 秒ごとに繰り返し記録できる。OS はすべてのメインメモリを均等に利用するのではなく、実際には書き込み主体のページや読み込み主体のページが存在する。本提案手法を利用すれば、OS のメモリアクセス傾向を分析することで、例えば書き込み主体のページを MRAM に配置する手法を検討できる [3]。

3.1 メモリ読み書き量の計測

最近の Intel Xeon プロセッサはメモリサブシステムに対する Performance Monitoring Unit (PMU) [4] を備えている。メモリチャネルごとに用意された `CAS_COUNT.RD` および `CAS_COUNT.WR` レジスタを参照することで、そのメモリチャネルに対する読み込みバイト数および書き込みバイト数を計算できる。そこで、提案手法では、特定のメモリチャネルが提供する物理メモリアドレス空間を排他的にゲスト OS のメインメモリとして割り当てる仕組みを開発した。ゲスト OS のメモリ読み書きバイト数を PMU レジスタから計測できる。

提案手法では、仮想マシンモニタプログラムの Qemu/KVM [5] に対して、指定したメモリチャネルの物理メモリから仮想マシンの RAM を作成するよう拡張を施した。ホスト OS が起動する際に、対象のメモリチャネルが提供する物理メモリのアドレス空間を予約し、ホスト OS カーネルがそのメモリ領域を使用しないようにする。Qemu/KVM が仮想マシンの RAM メモリ領域を作成する処理内においては、通常は `malloc()` により仮想マシンのメインメモリを作成する。一方、我々の拡張機能においては、`/dev/mem` に対する `mmap()` を用いて、あらかじめ予約した物理アドレス空間をもとに仮想マシンのメインメモリを作成する機構を設けた。複数のメモリチャネルからそれぞれメモリを確保して仮想マシンのメインメモリを作成することも可能である。仮想マシンのメインメモリの特定の領域を異なるメモリチャネルに割り当てることができる。

計算機のメモリチャネルと物理メモリ番号のマッピングはハードウェア依存の情報である。我々が用いた計算機においては、メモリチャネルのインタリービングを BIOS で無効にすることで、一つのメモリチャネルが提供する物理メモリのアドレスが連続することを確認した。さらに、各メモリチャネルが提供する物理アドレス空間を特定する作業を半自動化するプログラムを作成した。物理アドレス空間を例えば 1GB ごとに分割し、各領域にアクセスした際にどのメモリチャネルのメモリアクセス数が増加するか記録する。このデータを分析した結果、開発に用いた計算機において、例えば CPU ソケット 0 のメモリチャネル 2 が提供する物理アドレス空間は、`0x0900000000` から

0x10fffffffであることがわかった。

3.2 メモリ読み書きオフセットの計測

従来、仮想マシンのメモリ書き込みおよび読み込みを検知する手法としては、ページテーブルエントリのアクセス可能ビットおよび書き込み可能ビットを操作する手法が一般的である。あらかじめホスト OS 側で監視対象のゲスト OS ページをアクセス不可として設定しておくと、ゲスト OS がページにアクセスした際にページフォルトが発生し、アクセス検知機構がそのページ番号を記録する。同様の仕組みをページ書き込み操作のみに対して実装したものは、仮想マシンのライブマイグレーション機構において動作するダーティページトラッキングとして知られている。しかし、この手法はページフォルトによるゲスト OS の性能低下を伴ってしまう。

そこで、提案手法では、仮想マシンモニタプログラム Qemu/KVM を拡張して、Intel CPU が備える仮想化支援機構 Extended Page Table (EPT) の Accessed Bit および Dirty Bit (AD ビット) [6] を利用した、軽量なメモリアクセス・トレース機構を開発した。EPT は、ゲスト OS から見た物理アドレスとホスト OS から見た物理アドレスの対応を、radix-tree 上に記録したものである。ハイパーバイザーが作成した EPT の情報を元に、CPU は仮想マシンのメモリアクセスを実際の物理メモリに対するアクセスに変換する。この際、CPU は変換に利用した EPT エントリに対して A ビットを立て、このメモリアクセスが書き込みであった場合はさらに D ビットも立てる。提案機構は、一定時間ごとに各 EPT エントリをスキャンし、その AD ビットの状態を保存し再びビットをクリアすることを繰り返す。読み込みがあったページ番号および書き込みがあったページ番号を継続的に記録する。

ホスト OS 上において仮想マシンを構成するプロセスとは別のプロセスが、AD ビットを操作・観察するよう設計した。仮想マシンとは別の物理 CPU で実行することで、仮想マシンに対する性能低下をほぼ抑制できる。

4. 予備評価

提案手法が意図したとおりに動作するかどうかを確認するため、Qemu/KVM (Qemu-2.2.0 および Linux Kernel 3.18.5) を対象にプロトタイプを実装し簡単な評価実験を行った。

PMU を用いたメモリ読み書き量の測定において、評価に用いた計算機は CPU として Intel Xeon E5-2650 2.00GHz (L3 キャッシュ 20MB) を 2 個、メモリ 256GB を搭載する。計 8 個のメモリチャネルのうち一つを仮想マシン用に確保し、その中から仮想マシンのメモリを割り当てた。仮想マシンは仮想 CPU を 1 個、メモリを 4GB となるよう設定した。AD ビットを用いたメモリ読み書きオフセットの計測において、評価に用いた計算機は CPU として Intel Core i5-4570 3.20GHz (L3 キャッシュ 6MB) を 1 個、メモリ 8GB を搭載する。仮想マシンの設定はメモリ読み書き量の監視を評価した実験と同一である。実験機材の都合上、異なる計算機を用いた点に注意されたい。

4.1 行列演算

簡単な行列演算を行うプログラムを作成した。コードの概要を**図 1** に示す。最初に 10 秒間は配列 A から配列 C へ各要素を代入し、その後 10 秒間は配列 A と B の和を配列 C へ代入する。各配列の要素は double 型であり、一つの配列は約 763MB のデータになる。行列演算プログラムを仮想マシン内で起動し、提案機構によりその仮想マシンのメモリアクセス傾向を計測した。

図 2 は、PMU の計測により求めた仮想マシンのメモリ読み書きスループットである。時刻 8 秒付近で行列演算プログラムを起動した。時刻 10 秒から 20 秒の間に、配列 A から配列 C への代入動作を繰り返している。配列 C への代入によって生じたメモリ書き込みが約 2.8GB/s で生じている。メモリ読み込みスループットが書き込みスループットの約 2 倍生じている。その原因として、代入動作において配列 A からメモリ読み込みが発生することに加え、配列 C への書き込みにおいても CPU がメモリからデータをキャッシュに読み込むためではないかと考える。その後の配列 A と B の和を配列 C へ代入する期間においては、新たに配列 B がデータ読み込みの対象となったために、読み込み速度は書き込み速度の 3 倍程度となった。

図 3 は、毎秒ごとに EPT をスキャンして得られた、アクセス済みページ数および書き込み済みページ数である。その 1 秒間において一度でもアクセスがあったページはアクセス済みとして記録し、一度でも書き込みがあったページは書き込み済みとして記録する。1 秒ごとにそれぞれの総数をグラフに表した。行列演算プログラムの動作中において、1 秒間に書き込みを経験したページは 200000 程度ある。これは配列 C のデータサイズに相当する。またアクセスがあったページは、配列 A から C への代入時においては 400000 程度である。これは配列 A および C のデータサイズに相当する。同様に配列 A と B の和を C に代入する際には、配列 A、B および C のデータサイズである 600000 程度のページがアクセスされた。書き込み済みのページやアクセス済みのページのオフセットについての情報も取得できている。その一例については次節のカーネルコンパイルの実験で紹介する。

以上から、メモリ読み書き速度やアクセス済みおよび書き込み済みページのオフセットは、正しく計測できていると考えられる。提案機構がもたらす仮想マシンの性能低下については、今後の研究において詳しく評価する予定である。本実験では 9GB/s 程度のメモリ読み書きスルー

```
1  static double a[100000000];    // 763MB
2  static double b[100000000];    // 763MB
3  static double c[100000000];    // 763MB
4
5  void test(void)
6  {
7    while (... 10 seconds ...)
8      for (unsigned long i = 0; i < 100000000;
           i++)
9        c[i] = a[i];
10
11   while (... 10 seconds ...)
12     for (unsigned long i = 0; i < 100000000;
           i++)
13       c[i] = a[i] + b[i];
14 }
```

図 1 行列演算の概要

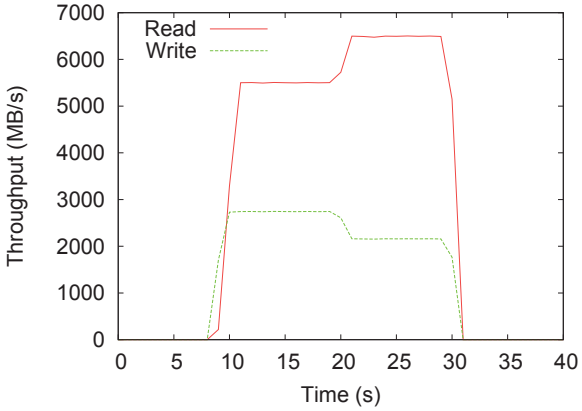


図 2 行列演算: PMU 監視によるメモリ読み書きスループット

プットが達成できており、提案機構においては、binary instrumentation や full system emulation 手法よりも遙かに高速に計測対象が動作することは明らかである。なお、PMU により計測できるのはメモリ読み込みおよび書き込みの速度であり、EPT の AD ビットが表すのはアクセス済みであることおよび書き込み済みであることである。後者においてはアクセス済みとは読み込みがあったページも書き込みがあったページも両方が該当することに注意されたい。

4.2 カーネルコンパイル

次に実際のアプリケーションとして、Linux のカーネルコンパイルを仮想マシン上で行った。最初に `tar` プログラムで圧縮ファイルからソースコードファイル群を展開し、次に `make allnoconfig` を実行してカーネルの構成を設定し、最後に `make` を実行してカーネルコンパイルを開始した。

図 4 に PMU によって得られたカーネルコンパイル中のメモリ読み書きスループットを示す。`make` 実行後は

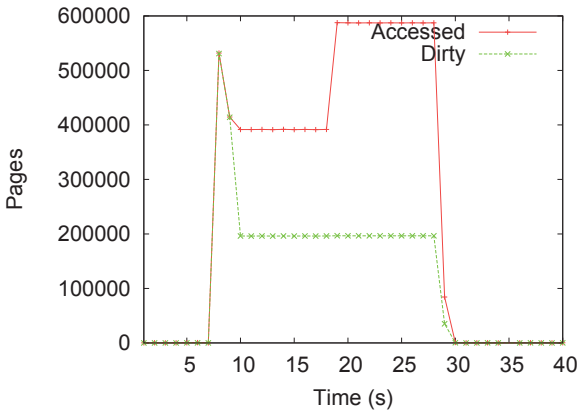


図 3 行列演算: EPT 監視による AD ビット総数

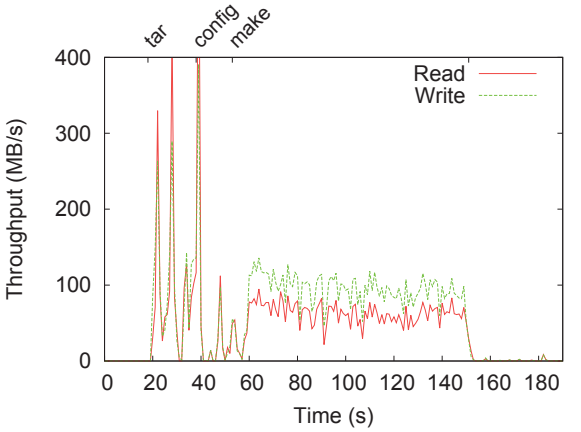


図 4 カーネルコンパイル: PMU 監視によるメモリ読み書きスループット

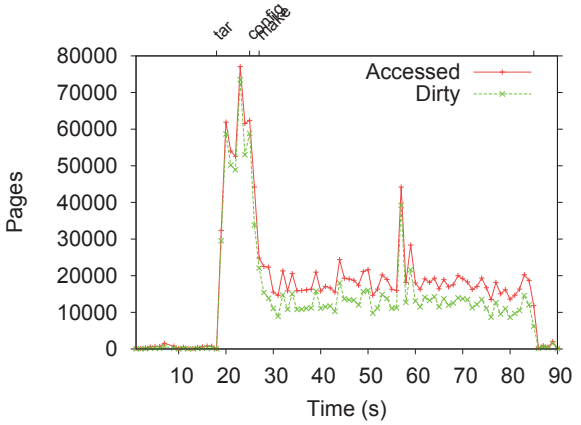
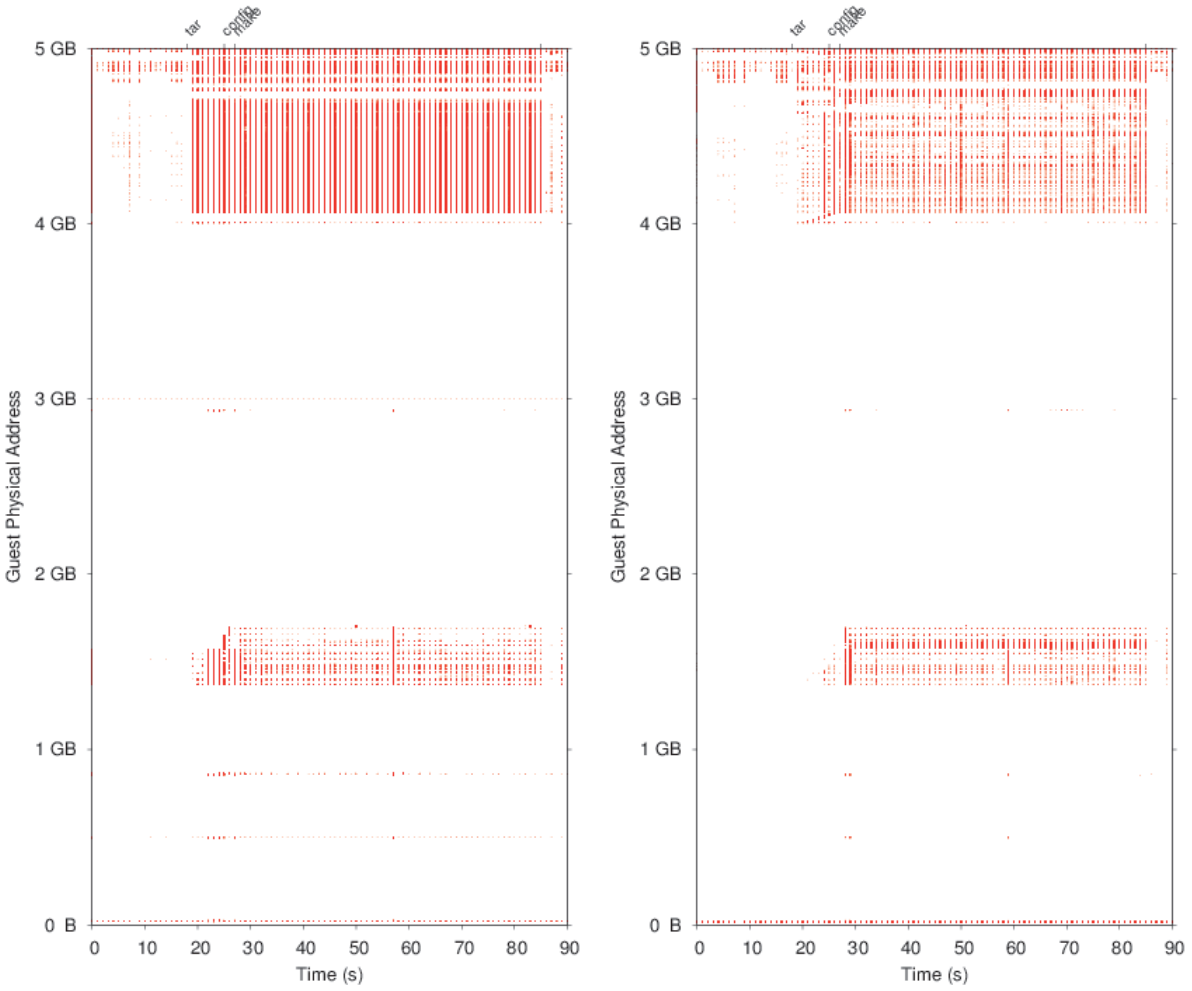


図 5 カーネルコンパイル: EPT 監視による AD ビット総数

60-70MB/s 程度のメモリ読み込みスループットを観測した。書き込みスループットは 80-100MB/s 程度であった。`make` 実行時には大半のソースコードファイルは既にページキャッシュ上にあると考えられる。コンパイラがオブジェクトファイルのデータを生成するときに、メモリ書き込みをともなっているものと想定される。**図 5** が示すように、`tar` によるソースコードファイル群の展開時に 1 秒間において 50000 ページ以上に書き込みが発生していた。そ



(a) 書き込みをともなったページ　　(b) 読み込みのみであったページ

図 6　カーネルコンパイルにおいてアクセスされたゲスト物理ページの分布

の後 make 実行時には書き込みは概ね 15000 ページ以内であった。

図 6a は書き込みが観測されたページのゲスト物理アドレスを毎秒ごとに示したものである。**図 6b** はアクセスがあったが書き込みはなかったページ（読み込みのみであったページ）のゲスト物理アドレスを示している。図 5 における Accessed と Dirty の差分に相当する。仮想マシンから見た物理アドレスにおいて 3GB から 4GB の空間においてメモリアクセスはない。これは、Qemu が仮想マシンに割り当てた 4GB のうち 3GB 分をゲスト物理アドレス 0-3GB の空間に割り当て、残りの 1GB をゲスト物理アドレス 4-5GB に割り当てているからである。実験中に使用されているメモリページは、ゲスト物理アドレスの 1.4GB から 1.7GB 付近と 4G から 5GB 付近に集中していることがわかる。図 6a と図 6b を比べると、読み込みのみであったページの分布はその密度は薄いものの、書き込みをともなったページの分布と同様に一様に散らばっていることがわかる。今後評価実験を進めることで、例えば書き込みが頻出するページを DRAM に配置し、それ以外のページを MRAM に配置した場合の省エネ効果等を議論したいと考えている。

5. まとめ

OS 全体のメモリアクセスを対象として、そのメモリ読み込みスループットおよび書き込みスループット、アクセスしたページのオフセットおよび書き込みを行ったページのオフセットを計測する手法を開発した。提案機構は計測対象の OS を改変する必要がない。仮想マシンモニタプログラムの Qemu/KVM を拡張することで、CPU が備える PMU および EPT AD ビットを用いて、ゲスト OS のメモリアクセスを計測することを可能にした。計測対象の OS の外部で動作するため、計測処理によるメモリアクセスを計測対象から除外できる。CPU インストラクションを逐次分析する手法が細粒度のメモリアクセス情報を取得するために実行速度が極めて遅くなるのに対して、我々の提案手法は粗粒度のメモリアクセス情報の取得に特化するこ

とで実行速度の低下を避けている。簡単な評価実験を通して、提案機構が正しく動作していることを確認した。今後より詳細な評価実験を行い、不揮発性メモリによる計算機の省エネ効果等を議論する予定である。

参考文献

[1] The International Technology Roadmap for Semiconductors (ITRS): International Technology Roadmap for Semiconductors 2013 Edition (2013).
[2] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D. and Wood, D. A.: The Gem5 Simulator, *SIGARCH Computer Architecture News*, Vol. 39, No. 2, pp. 1–7 (2011).
[3] 広渕崇宏, 高野了成, 工藤知宏：仮想化支援技術を利用した軽量なメモリアクセス・トレース機構の提案, *Annual Meeting on Advanced Computing System and Infrastructure (ACSI) 2015*, 情報処理学会 (2015).
[4] Intel Corporation: Intel Xeon Processor E5-2600 Product Familiy Uncore Performance Monitoring Guide (2012).
[5] Kivity, A., Kamay, Y., Laor, D. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, *Proceedings of the Linux Symposium*, The Linux Symposium, pp. 225–230 (2007).
[6] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual (2014).