社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

NDCKPT: 不揮発性メインメモリを用いたOSによる透過的なプロセスチェックポインティングの実現

西田　　耀†　　木村　啓二†

† 早稲田大学基幹理工学部　〒105–0123 東京都新宿区大久保 3–4–1
E-mail: †hikalium@kasahara.cs.waseda.ac.jp, ††keiji@waseda.jp

あらまし　アプリケーションの耐障害性を向上させる手法の一つにチェックポインティングがある．これまでに，アプリケーションを変更することなく透過的にチェックポイントを行う手法がいくつか発表されている．また，Non-volatile DIMM (NVDIMM) を状態の保存先として利用することで，主記憶に比べて 100 倍以上遅い外部記憶へのアクセスに依存することなくチェックポイントを行う手法が提案されている．しかし，DRAM で構成された主記憶から不揮発性の記憶装置に状態をコピーするという操作は依然存在しており，これがチェックポイントのオーバーヘッドの大部分を占めている．本研究では，アプリケーションを NVDIMM 上に直接マッピングして実行することで状態のコピーを最小限に抑え，さらにページテーブルも含めたプロセスのメモリ空間を二重化して一貫性を確保しつつチェックポインティングを行う，NDCKPT という手法を提案する．Linux Kernel に NDCKPT を実装し，Optane DC Persistent Memory を用いて評価を行った結果，メモリ消費量が 1MB 程度のアプリケーションでは，100ms 程度の高頻度でチェックポイントを行っても実行時間の増加を 1% 以下に抑えられることがわかった．また，数百 MB のメモリを消費するアプリケーションにおいては，NVDIMM 上で実行を行うオーバーヘッドが支配的で実行時間比で 2 倍から 3 倍以上となる一方，チェックポイントによって加わるオーバーヘッドは 20-30 秒間隔で 10% 前後となることがわかった．
キーワード　不揮発性メモリ，チェックポインティング，オペレーティングシステム

1. はじめに

アプリケーションに耐障害性を与える手法のひとつとして，チェックポインティングが挙げられる．チェックポインティングは，ある時点におけるアプリケーションの状態そのものを保存しておき，障害発生後にその状態を復元するものである．アプリケーションの実装を変更することなく透過的にチェックポイントを行う手法 [1] がこれまでに研究されている．

突然の電源断が発生した後にプロセスの状態を復元するためには，その状態を不揮発な外部記憶などに格納しておく必要がある．しかし，プロセスの状態の大部分が置かれている主記憶に比べて，外部記憶へのアクセスは 100 倍から 1 万倍以上遅い．そのため，主記憶から外部記憶へ状態をコピーする操作が，チェックポインティングの主要なオーバーヘッドとなっていた．

そこで，より高速かつ不揮発な主記憶である，Non-volatile DIMM(NVDIMM) を用いることが考えられる．従来の DRAM で構成された主記憶に比べ，NVDIMM へのアクセスは 10 倍程度遅い．しかし，I/O バスを介さず CPU から直接アクセスができること，また CPU のキャッシュ階層の下におかれることから，NVDIMM は外部記憶より格段に高速な不揮発性ストレージとしての利用が期待されている．一方で，CPU キャッシュは揮発性であるため，NVDIMM への書き込みが不揮発になったことを保証するためには，ソフトウエアの側でキャッシュを適切にライトバックしなければならない．

NVDIMM を利用して透過的にプロセスチェックポインティングを実現する手法には先行研究 [2] が存在する．しかし，先行研究の提案した手法では，アプリケーションは DRAM 上で実行され，チェックポイントのタイミングで NVDIMM にコピーされる．

本研究が提案する NDCKPT は，アプリケーションを直接 NVDIMM 上にロードして実行することで，NVDIMM の特性をより活かしたプロセスチェックポインティングを実現する．また先行研究と同様，オペレーティングシステムにチェックポインティング機構を組み込むことで，アプリケーションからみて透過的にプロセスチェックポインティングを実現する．

NDCKPT の特徴は下記の通りである．

（1） アプリケーションを NVDIMM 上に直接マップして実行し，その領域を用いてチェックポイントを実現した

（2） NVDIMM 上にページテーブルをおくことで，チェックポイントした状態を実行時に近いデータ構造で保持した

（3） Linux Kernel 5.1.8 に実装することで，既存の OS を拡張することで実装可能であることを示した

2. Non-volatile DIMM(NVDIMM) の特性

不揮発性の主記憶デバイスである NVDIMM には，従来の主記憶に使われている DRAM と異なり，電源が失われてもデータが消失することがないという特徴がある．NVDIMM の実現方式はいくつか存在するが，本研究では，NVDIMM-P に分類さ

れる，Intel Optane DC Persistent Memory (DCPMEM) を評価に用いた．NVDIMM-P は不揮発性メモリのみを搭載しているため，DRAM DIMM よりも容量単価が安く，またリフレッシュが不要であることから消費電力を抑えることが可能であるが，アクセス速度は DRAM の 10 倍程度遅くなっている．[3]

ソフトウエアからは，NVDIMM を従来の主記憶と同様に扱うことができる．つまり，NVDIMM へは I/O バスを介さずにメモリ操作命令を用いてアクセスでき，そのアクセスはキャッシュ階層の下におかれる．

CPU のキャッシュ階層を活用できることは，メモリアクセスの高速化という恩恵を受けられる反面，NVDIMM の不揮発性という特性を生かす上で問題となる．CPU のキャッシュは従来通り揮発性であるため，NVDIMM に書き戻されていないキャッシュ上のデータは，電源喪失時に消失してしまう．これを防ぐため，ソフトウエアは適切にキャッシュのライトバックを行う必要がある．また，DCPMEM においては，8 バイト境界にアラインされた 8 バイトの書き込みのみが障害に対するアトミック性 (failure-atomicity) を保証されており [4]，電源喪失前後で一貫性を保つためには，この制約に適したデータ構造を利用する必要性がある．

3. NDCKPT の設計

本研究では，主記憶と同様に扱うことが可能という NVDIMM の特徴を活かしつつ，その不揮発性を透過的に利用できるチェックポイント手法として NDCKPT を設計した．本節では，その設計の詳細について述べる．

3.1 チェックポイントの対象

チェックポイント時に保存の対象となるデータは，チェックポイントの対象をどの範囲に定めるかによって決まる．先行研究 [2] では，複数のプロセスやスレッドからなるプロセスツリー全体を対象としていたが，本研究では実装を容易にするため，単一のプロセスを対象とした．プロセスの状態には，ユーザーモードのメモリ空間，レジスタ状態，ページテーブル構造体，カーネルモードのメモリ空間，カーネルが保有する資源の状態（ファイルハンドルやソケット等）が含まれるが，本研究ではユーザーモードのメモリ空間のうち読み書き可能な領域と，レジスタ状態，ページテーブルを保存の対象とし，カーネルモードのメモリ空間とカーネルが保有する資源の状態については対象外とした．保存されるメモリ領域には，アプリケーションのデータ領域やスタック領域，及びヒープ領域が含まれる．コード領域については実行ファイルから復元できることを考慮し，保存対象から除外した．NDCKPT を用いて，ファイル I/O が進行中の状態をチェックポイントすることはできない．これは実装を簡易にするための制約であり，先行研究 [2] の手法を用いれば実現可能と考えられる．

3.2 ページング機構を用いたメモリ空間の二重化

NDCKPT では，アプリケーションの書き込み可能なメモリ空間を NVDIMM 上の領域に直接マッピングする．これにより，アプリケーションが変更したメモリ領域を，キャッシュのライトバックを行うだけで即座に永続化することができる．これは，DRAM 上でアプリケーションを実行し，チェックポイントのタイミングでデータを不揮発性デバイスに書き出すという従来の手法とは異なり，永続化の段階でデータのコピーを必要としない．

しかし，単純にメモリ空間のマッピングを NVDIMM に変更しただけでは，アプリケーションの実行に伴ってデータが書き換わるため，チェックポイントした状態の一貫性が失われてしまう．そこで NDCKPT では，ページテーブル構造体を 2 組用意することでメモリ空間を二重化し，図 1 に示したようにこれらを切り替えつつ実行することで，保存されたメモリ空間と実行に用いられるメモリ空間とを分離した．この方式には，チェックポイントを適用した際のメモリ消費を，通常実行時の 2 倍に抑えることが可能という利点もある．

図 1 の上段は，アプリケーションが起動してから障害が発生するまで，下段は障害発生後にプロセスが再開してからの，アプリケーションの処理の進捗とチェックポイントに伴う状態の切り替えを示したものである．チェックポイントが行われた時刻をそれぞれ t0, t1, ... とおいた．これらの時刻で区分される，処理の進捗の各世代を Epoch と表現している．ctx[0], ctx[1] は NDCKPT によって二重化されたプロセスの状態を示している．Running と書かれている ctx は，その Epoch の間プロセスの仮想アドレス空間にマップされており，アプリケーションの処理が進むにつれて順次変更が加えられる．一方，State at tN と書かれている ctx は，その Epoch の間変更が加えられることはなく，時刻 tN の状態を保持している．

図 1 の上段では，Epoch2 の間に障害が発生した状況を示している．このとき，実行に用いられていた ctx[0] の内容は不定となるが，ctx[1] の内容は直前のチェックポイント時刻 t2 の状態を保持している．したがって，下段に示した通り，ctx[1] の内容をもとに時刻 t2 の状態を復元できる．

また NDCKPT では，それぞれの状態に対応するページテーブル自体も NVDIMM 上に配置することにした．これにより，チェックポインティングのためだけに新たなデータ構造を追加することなく，既存のデータ構造を活用してチェックポイントされたメモリ状態を管理できるという利点がある．

この二重化されたマッピングの作成と，チェックポイントに伴うキャッシュのライトバックは，OS に組み込まれたチェックポイント機構によって行われる．したがって，アプリケーションを変更することなく，透過的にチェックポインティングを実現できる．

3.3 NVDIMM に適した状態管理用データ構造

NDCKPT が保存の対象としているプロセスの状態には，ユーザーメモリ空間の他にも，CPU 内部のレジスタの状態や，仮想アドレス空間の割り当て状況が含まれる．これらのデータと，前述した NVDIMM 上に配置されたページテーブルへのポインタを格納した構造体が，図 1 にも示した ctx である．プロセスの実行と，チェックポイントされた状態の保存に交互に利用しながら実行を継続するため，復元可能な ctx がどちらであるかを記録しておく必要がある．DCPMEM の failure-atomicity を考慮すると，復元可能なコンテキストの切

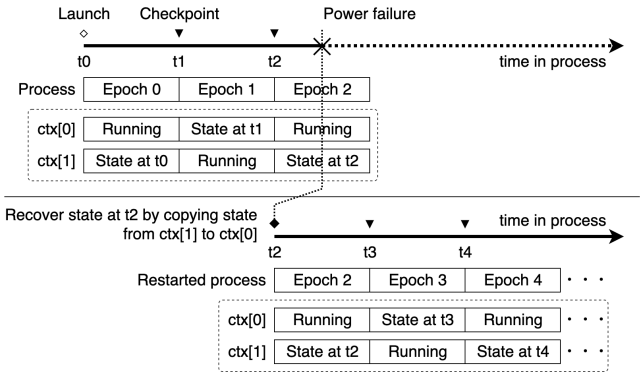

図 1　チェックポイントに伴う状態切り替えと障害発生時の復元

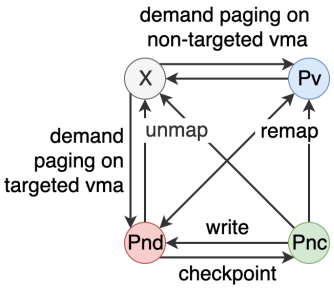

図 2　ページの状態変化

り替えは単一の整数もしくはポインタ変数の書き換えで完結する必要がある．そこで，NDCKPT では有効な ctx を指すインデックス (valid_ctx_idx) を 2 つの ctx と共に NVDIMM 上に保存しておき，これを参照することで，どの時点において障害が発生しても最後にチェックポイントされた際の整合性がとれた状態を復元できるようにしている．

NDCKPT が実現するチェックポイント機構の視点から，ページの状態とその間の遷移を示したものが図 2 である．左上がマップされていない状態 (X)，右上が通常の DRAM ページにマップされている状態 (Pv) であるが，これに加えて NVDIMM 上のページにマップされていてかつ dirty bit が 1 である状態 (Pnd)，NVDIMM 上のページにマップされているが dirty ビットが 0 である状態 (Pnc) が加わっている．

チェックポイント対象のページは，デマンドページングにより NVDIMM 上のページにマップされ，Pnd の状態に遷移する．その後，チェックポイントのタイミングでキャッシュがライトバックされると，Pnc に移行する．Pnc の状態にあるページは，NVDIMM 上でデータが永続化されているページである．このページにアプリケーションが書き込みを行うと，dirty bit が CPU によって 1 にセットされることで Pnd の状態に移行する．チェックポイント時にライトバックするページを，dirty になっている NVDIMM ページ (Pnd) のみに限定することで，フラッシュに起因するオーバーヘッドを最小限に抑えることができる．

3.4　ページテーブルの同期

図 1 に示した通り，チェックポイントを境として 2 つの ctx の役割は交換され，チェックポイント以前に実行に用いられていた ctx はチェックポイント時の状態を保存するために用いられ，もう片方の ctx はチェックポイント以降の実行のために用いられる．前者については，メモリ空間をフラッシュするだけでよいが，後者は他方の ctx が保持している，現在のメモリ空間の状態をコピーする必要がある．この際，チェックポイント前の Epoch においてマッピングの増減や変更が生じている可能性があるため，それぞれのページテーブルを突き合わせてページのマッピングを修正し，さらに変更があったページに関してはコピーを行う．これを同期 (sync) と呼ぶことにする．

3.5　チェックポイントの動作

3.4 節までで導入した概念に基づき，NDCKPT におけるプロセスの起動，チェックポイント，障害発生後の復元それぞれの動作を説明する．また，NDCKPT はデマンドページングに伴うページフォールト発生時の動作も変化させるため，これについても説明する．

3.5.1　プロセスの起動

NDCKPT の処理は，プロセスの起動時を起点として実行される．まず，OS 既存のローダにより，アプリケーションがメモリ上にロードされる．その後，ローダにより作成されたページテーブルを基にして，NVDIMM 上に配置されたページテーブルを 2 組作成する．デマンドページングにより，この段階で実際にマップされるページは最小限に留められているが，ユーザースタックのようにチェックポイント対象のページもマップされているため，これらを NVDIMM 上のページで置き換える．これらのページテーブルを含むように，2 つの ctx を初期化する．この段階で，2 つの ctx はどちらも復元可能な状態となっているため，これらを用いて復元動作を行うことで，プロセスの実行を開始する．なお，起動時に復元すべき状態がすでに指定されていた場合には，起動処理の代わりにそれらを用いて復元動作を行うことで，プロセスの実行を再開する．

3.5.2　ページフォールトのハンドル

ページフォールトハンドラの動作を下記のように修正することで，チェックポイント対象のメモリ領域に対しては NVDIMM 上のページを割り当てるようにする．

（1）　チェックポイント対象外のメモリ領域については，通常のページフォールトハンドラを呼び出して実行を再開する．

（2）　チェックポイント対象の領域に対しては，NVDIMM 上のページを確保してマッピングする．この際，ページテーブルを構成する各階層のテーブルも NVDIMM 上に存在しているようにする．これは，途中の階層のページテーブルが DRAM 上に存在すると，その配下のマッピング情報が再起動後に失われてしまうためである．

（3）　更新したページテーブルエントリをライトバックする．

（4）　アプリケーションの実行を再開する．

3.5.3　チェックポイント

図 1 の時刻 t3 におけるチェックポイントを例に説明する．この際に発生するメモリへの操作を図 3 に示した．チェックポイントはプロセスを停止させた後，カーネルモードにおいて下記の手順で行われる．なお，図中の (1), (3), (4) は，下記の手順の番号と対応している．

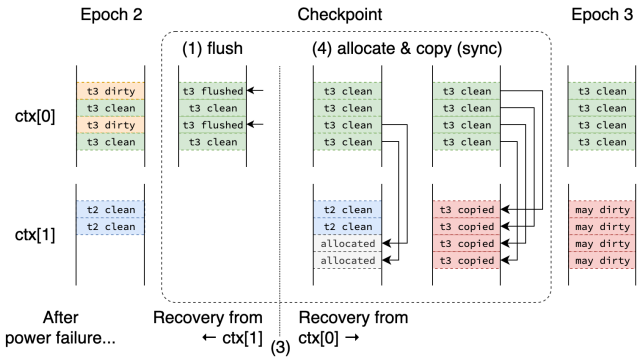

図 3　チェックポイントに伴うページのフラッシュとコピー

（1） チェックポイント直前までプロセスが実行に使っていた ctx[0] のページテーブルを走査し，dirty ページに対してライトバック命令を発行する．

（2） ユーザ空間のレジスタ状態を ctx[0] に書き込み，ライトバックする．ここで保存されるレジスタとメモリの状態は，チェックポイント処理から復帰した際のユーザ空間の状態 (t3) になっている．

（3） valid_ctx_idx を ctx[0] に切り替える．この時点で，ctx[0] は t3 の状態を復元可能な状態になっており，また ctx[1] は一つ前の時刻 t2 の状態を復元可能になっているため，この切り替え中に障害が発生しても，どちらかのコンテキストを利用して正しくプロセスの状態を復元できる．この手順が完了した後は，ctx[0] が復元対象のコンテキストになる．

ここまでの処理で，状態の保存という意味でのチェックポイントは完了している．これに加えて，その後の実行を継続するために，3.4 節で述べた手順に従って:

（4） ctx[1] に ctx[0] のマッピングを同期してから，CPU の参照するページテーブルを切り替えて実行を再開する．

これにより，上記の手順で保存した状態を破壊することなく，次の Epoch における実行を継続できる．

3.5.4　障害発生後の復元

電源断などの障害が発生した場合でも，valid_ctx_idx が指しているコンテキストは，最後にチェックポイントを行った際のデータを保持している．一方で，障害発生時に実行に利用していたコンテキストの内容は不完全なものとなっている可能性がある．そこで，復帰時に利用するコンテキストの内容を，他方のコンテキストに同期してからアプリケーションの実行を再開する．また，NVDIMM 上のページテーブル構造体が保持している，DRAM 上のページへのマッピングは不正なものとなっているため，これを消去したのち，再起動したプロセスのマッピングを参照して，DRAM 上のページへのマッピングを修正する必要もある．

図 1 の下段，時刻 t2 における動作を例に，復元時の操作を示す．復元操作は，アプリケーションがロードされた直後で，アプリケーションに制御を戻す直前のカーネルモードで行われる．

（1） 復帰時に利用するコンテキスト ctx[1] のページテーブルを走査し，DRAM ページへのマッピングを削除する．

（2） 復元を行うプロセスのページテーブルを参照して，チェックポイント対象外のメモリ領域を ctx[1] にコピーする．これらのページは DRAM にマッピングされている．

（3） 他方のコンテキスト ctx[0] の状態を ctx[1] と同期する．この作業が完了すると，両方のコンテキストが最後のチェックポイントが行われた時刻 t2 の状態を復元可能になっている．

（4） ctx[1] のページテーブルとレジスタ状態を CPU にロードし，プロセスの実行を開始する．これにより，プロセスは時刻 t2 の状態から再開される．

4. 実　　装

本研究では，提案手法である NDCKPT を Linux Kernel 5.1.8 上に実装した．(注1)

4.1　NVDIMM ページアロケータの追加

CPU からみて従来の DRAM 領域と NVDIMM 領域は共通の物理アドレス空間上にマップされており，アクセスも同等に行うことができる一方で，Linux Kernel において NVDIMM 上のページは DRAM とは異なるものとして管理されている．したがって，カーネル内部に NVDIMM 上のページを確保するアロケータは存在しない．そこで，非常にシンプルな NVDIMM ページアロケータを実装し，NDCKPT で利用するようにした．この NVDIMM ページアロケータは不揮発性を考慮して実装されているが，実装を単純にするため，ページの解放は実装していない．

4.2　syscall の追加

Linux において，新たなアプリケーションの起動は，fork() によるプロセスの生成とそれに続く exec() によるアプリケーションのロードによって行われる．NDCKPT では，アプリケーションのロード段階でプロセスの状態の二重化を開始するため，exec が実行されるよりも前にカーネルに対して該当プロセスがチェックポイントの対象であることを通知しなければならない．そのためのインターフェイスとして prctl(PR_ENABLE_NDCKPT) を追加した．この引数に復元すべき状態が格納されている NVDIMM 上の場所を指定することもできる．

チェックポイントを要求するためのインターフェイスとして ptrace(PTRACE_DO_NDCKPT) を追加した．これにより，外部のプロセスから任意のタイミングで対象プロセスのチェックポイントを要求することができる．このとき，対象プロセスは事前に ptrace によって停止されている必要がある．

4.3　データ構造の修正

task_struct および vm_area_struct には，それぞれチェックポイント対象であることを示すフラグを追加した．それ以外の既存のデータ構造は変更していない．

4.4　ページフォールトハンドラの修正

既存のページフォールトハンドラは，ページテーブルが NVDIMM 上におかれることを考慮しておらず，ページテーブルエントリの変更に伴うライトバックも行われていない．そのため，チェックポイント対象のプロセスにおいて発生したすべ

(注1)：https://github.com/hikalium/linux-ndckpt

表 1　評 価 環 境
Table 1　Evaluation platform specifications

CPU model	Intel(R) Xeon(R) Gold 5222 CPU @ 3.80GHz
Cache	L1: 256KiB, L2: 4MiB, L3: 16.5MiB(shared)
NVDIMM	2 × 128GB Intel Optane DC PMEM
DRAM	6 × 8GiB Micron DDR4 2666 MHz
Storage(SATA)	480GB Micron 5200 SSD
Base Board	Supermicro X11SPL-F v1.02
OS	Linux 5.1.8 with NDCKPT patches
	SMP Disabled, THP Disabled

てのページフォールトは NDCKPT がフックし，デマンドページングが発生した際に NVDIMM 上のページを割り当てるようにするなどの変更を加えた．

4.5　ページテーブルを操作する syscall の修正

mmap syscall は，realloc などを契機として呼び出された場合に，既存のページマッピングを変更することがある．この場合のページテーブル操作も NDCKPT がフックし，ライトバックなどを行うよう修正した．

5. 評　　価

Linux Kernel 上に実装した NDCKPT を，表 1 に示す環境で評価した．評価の結果は，3 回測定した結果の平均を示した．

5.1　評価対象アプリケーション

評価に用いたアプリケーションについて説明する．

5.1.1　円周率計算

本アプリケーションは，Spigot アルゴリズムを用いて，ある桁数まで円周率を計算することを 10 回繰り返し行うものである．メモリフットプリントが異なる場合の評価を行うため，計算桁数が 1 万 5000 桁のもの (pi_15000.bin) と，3 万桁のもの (pi_30000.bin) の 2 種類を用意した．Spigot アルゴリズムは bss 領域に確保された計算バッファに繰り返し順次アクセスする．mmap や mremap は呼び出されず，ファイル I/O も行われない．それぞれが通常実行時に消費するメモリサイズは表 3 に示した通りである．

5.1.2　xz_s (SPEC CPU 2017)

xz_s [5] は SPEC CPU 2017 [6] ベンチマークに含まれるアプリケーションの一つで，xz 形式で圧縮されたファイルの展開と圧縮を行うものである．入力データおよび実行時引数によってメモリ使用量が変動するため，表 2 に示した通りパラメータを設定して評価を行った．それぞれのケースにおける通常実行時のメモリ消費量は表 3 に示した通りである．mmap, munmap は呼び出されるが，mremap は呼び出されない．ファイル I/O はプログラムの起動直後と終了直前のみに存在するため，実行時間の大部分はチェックポイント可能である．

5.2　NVDIMM 上での実行によるオーバーヘッド

NDCKPT を適用することで生じるオーバーヘッドには，チェックポイント操作に起因するものと，アプリケーションが NVDIMM にマップされていることに起因するものが含まれている．そこでまずは，アプリケーションを DRAM 上で通常実行した際の実行時間と，チェックポイントを一度も行わずに，各アプリケーションを NVDIMM 上で実行した際の実行時間を比較することで，NVDIMM に起因するアプリケーション実行オーバーヘッドを評価した．図 4 に，各アプリケーションにおける DRAM 上と NVDIMM 上での実行時間を示した．アプリケーションのメモリ使用量がキャッシュに十分収まる程度に小さい円周率計算に関しては，NVDIMM 上で実行しても 0.2% 前後しか速度が低下しなかったものの，メモリ使用量の大きい xz_s においては，実行時間で 2 倍から 3 倍程度の速度低下が確認された．

表 2　xz_s の評価パラメータ
Table 2　Parameters of xz_s evaluations

name	xz_s_ref_cld_32	xz_s_ref_docs_128
input buf size	32	128
min compressed size	536995164	1036078272
max compressed size	539938872	1111795472
compression level	8	4

表 3　評価アプリケーションのメモリ使用量
Table 3　Maximum resident set size of evaluation targets

pi_15000.bin	0.4 MB
pi_30000.bin	0.9 MB
xz_s_ref_docs_128	659 MB
xz_s_ref_cld_32	882MB

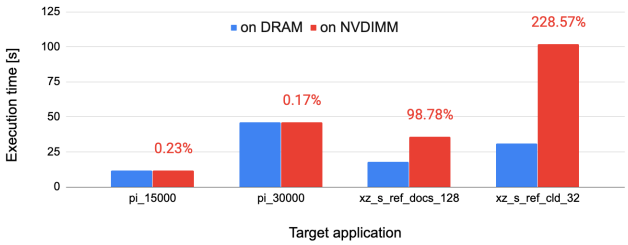

図 4　DRAM 上と NVDIMM 上での実行時間

5.3　チェックポイントの間隔とオーバーヘッドの関係

次に，チェックポイントに起因するオーバーヘッドを評価するため，チェックポイントを行う時間間隔を変化させ，チェックポイントを行わずに NVDIMM 上で実行した場合の実行時間を基準としたオーバーヘッドを測定した．図 5 に pi_15000.bin と pi_30000.bin，図 6 に xz_s の各パラメータでの測定結果を示した．メモリ使用量の小さい円周率計算に関しては，1 秒以下のような非常に高い頻度でチェックポイントを行っても，1% 以下のごく小さなオーバーヘッドでチェックポイントを実現できることが確認できた．また，メモリフットプリントの大きな xz_s の各ケースにおいても，20-30 秒間隔のチェックポイントで，チェックポイントを 1 回も行わずに NVDIMM 上で実行した場合にかかった時間と比べて 10% 以下のオーバーヘッドとなることがわかった．

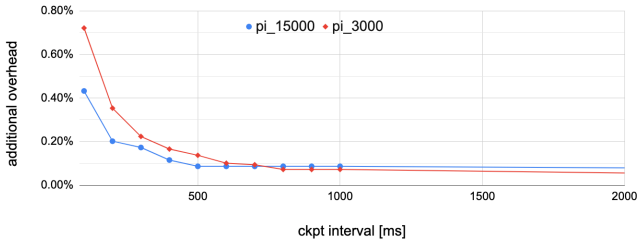

図 5　チェックポイント間隔とオーバーヘッドの関係 (pi)

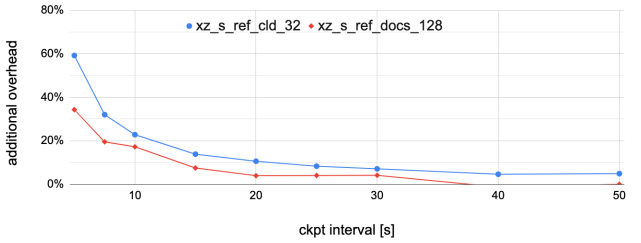

図 6　チェックポイント間隔とオーバーヘッドの関係 (xz_s)

6. 関 連 研 究

6.1 実装レイヤごとのチェックポインティング手法

チェックポインティングは，アプリケーション，ライブラリ，オペレーティングシステム，ハードウエアのどのレイヤでも実装することが可能だが，実装するレイヤによって，それぞれ利点と欠点が存在する．

アプリケーションやライブラリのレイヤでチェックポイントを実装した例としては，libckpt [1] や DMTCP [7] が挙げられる．libckpt は，ユーザ空間のライブラリによってチェックポインティングを実現する．DMTCP は，アプリケーションが保持する状態のみならず，ライブラリやカーネル内部の状態を，ライブラリ関数の呼び出しをフックしておくことにより記録し，復元する手法を提案した．これらのユーザー空間におけるチェックポイントの実装は，カーネルの変更が不要なため簡便であるという利点がある．一方で，アプリケーションの変更や再リンクが必要になるため完全に透過的ではなく，またカーネル内部の情報にアクセスできないため，原理的にサポートできない機能が存在するという点が欠点として挙げられる．

これらの欠点は，カーネルレイヤによるチェックポイントの実装で解決することが可能である．Linux-CR [8] は，チェックポインティングを Linux カーネルのレイヤで行うことで，より透過的なチェックポインティングを提供するものである．また Checkpoint and Restore In Userspace(CRIU) は，Linux カーネルに API を追加することで，チェックポイントに必要な処理の大部分をユーザ空間で実行しつつ，カーネルによるチェックポイントに近い，透過的なチェックポインティングを実現した．

6.2 NVDIMM を用いたチェックポインティング

µSnap [2] はカーネルレイヤでのチェックポインティングに加えて，その状態を NVDIMM 上に格納することにより，既存のアプリケーションが NVDIMM の不揮発性という特性を透過的に利用する手法を提案した．これにより，外部記憶よりも 10 倍から 1000 倍以上高速な NVDIMM が，チェックポインティングの状態を記憶するために利用できることが示された．µSnap は NVDIMM の不揮発性という側面をアプリケーションに対して透過的に提供する手法を確立したが，チェックポイント対象のプロセスは依然旧来の DRAM DIMM 上で動作しており，チェックポイントのタイミングで DRAM 上のページを NVDIMM にコピーするという方式を採用している．したがって，NVDIMM のバイト単位でアクセス可能であるという点，そして主記憶と同様の操作が可能という点においては拡張の余地がある．

7. ま と め

本稿では，NVDIMM 上でアプリケーションを直接実行しつつ，透過的にチェックポインティングを実現する NDCKPT という手法を提案した．評価の結果，メモリ使用量が大きなアプリケーションについては，NVDIMM 上で実行する際のオーバーヘッドが支配的だった一方，メモリ使用量 800MB 前後の大きなアプリケーションでも 10% 程度のオーバーヘッドで 30 秒間隔のチェックポイントが実現できることがわかった．

8. 謝　　辞

本研究の概念実証段階では，さくらインターネット株式会社様より研究開発をご支援いただきました．

文　　献

[1] J. S. Plank, M. Beck, G. Kingsley and K. Li: “Libckpt: Transparent checkpointing under unix”, Proceedings of the USENIX 1995 Technical Conference Proceedings, TCON’95, USA, USENIX Association, p. 18 (1995).

[2] J. H. Kim, Y. J. Moon, H. Song, J. H. Park and S. H. Noh: “µsnap: Embracing traditional programming models for persistent memory through os support”, 2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA), pp. 1–6 (2018).

[3] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao and S. Swanson: “Basic performance measurements of the intel optane dc persistent memory module” (2019).

[4] A. Rudoff: “Persistent memory programming”, ;login:, **42**, 2 (2017).

[5] “657.xz_s spec cpu®2017 benchmark description” (2019). https://www.spec.org/cpu2017/Docs/benchmarks/657.xz_s.html.

[6] J. Bucek, K.-D. Lange and J. v. Kistowski: “Spec cpu2017: Next-generation compute benchmark”, Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE ’18, New York, NY, USA, Association for Computing Machinery, p. 41–42 (2018).

[7] J. Ansel, K. Arya and G. Cooperman: “Dmtcp: Transparent checkpointing for cluster computations and the desktop”, 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–12 (2009).

[8] O. Laadan and S. E. Hallyn: “Linux-cr: Transparent application checkpoint-restart in linux”, Linux Symposium, Vol. 159Citeseer (2010).