システムソフトウェアとオペレーティング・システム 86−16
(2001. 3. 6)

不揮発 RAM を用いた Persistent OS における カーネルメモリマネージメント

○大村 廉† 山崎 信行‡ 安西 祐一郎‡

†慶應義塾大学大学院 理工学研究科 計算機科学専攻
‡慶應義塾大学 理工学部 情報工学科
〒223-8522 横浜市港北区日吉 3-14-1
E-mail: {ren,yamasaki,anzai}@ayu.ics.keio.ac.jp

あらまし 計算機器が一般に広まるにつれ、「突発的な電源切断への対応」が必要不可欠になると考える。そこで、本研究では、近年実用化が始まっている不揮発 RAM を主記憶として用い、Persistent OS の構築を目指す。この際必要となるカーネル内の状態保存のために、(1) 主記憶が不揮発となったときの各エンティティについて復旧を可能とするための要求について考察を行い、(2) この要求を低オーバヘッドで実現するためのカーネル内のメモリ管理方式について提案を行う。サンプルプログラムを用いた評価の結果、提案する方式が十分に低オーバヘッドでカーネル内の状態の復旧を可能としていることを示す。

キーワード OS Persistent system メモリ管理 不揮発 RAM

Kernel memory management for Persistent OS using non-volatile RAM

○Ren Ohmura† Nobuyuki Yamasaki‡ Yuichiro Anzai‡

†Department of Computer Science, Faculty of Science and Technology, Keio University
‡Department of Information and Computer Science, Faculty of Science and Technology, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa 223-8522 Japan

Abstract The countermeasure against unpredictable power-failure is required when the appliances with computer become popular. We therefore aim to build a Persistent OS using non-volatile RAM which is available lately(not conventional NVRAM, but such as FeRAM or MRAM) as main memory. Using these as main memory, the persistence of kernel space is required. Consequently, this papar describes (1)the requirements for some entities to be recoverble, and (2)the method of kernel memory management satisfying these requirements, and shows achieving that with low overhead as the results of evaluation.

Keywords OS Persistent system Memory management Non-volatile RAM

1 はじめに

現在計算機機器に用いられているシステムでは、そのほとんどがブレーカの切断や停電などの「突発的な電源切断」に面したときに、その時点で稼働しているアプリケーションの実行状態は失われることとなり、電源回復後ユーザが作業を継続するのは困難であった。PC に限らず PDA やセットトップボックスなど、近年の計算機機器の一般への普及は著しく、これに伴い突発的な電源切断への対応を行ないユーザに作業の継続性を提供することが必要不可欠であると考える。

突発的な電源切断に対し、その作業の継続性を提供することを目的としたシステムは Orthogonal Persistent System として研究がなされてきた。また、これをサポートするための OS は「Persistent OS」と呼ばれ研究がなされている [5, 7, 11, 6]。しかしながら従来の Persistent OS ではユーザの作業空間をディスク装置などの主記憶より遥かにアクセス速度の劣る永続ストレージにマッピングし、システムが整合性のとれた状態で復旧可能となる時点 (recoverble consistent state) において、これを保存していた。この作業は checkpointing と呼ばれる。永続ストレージへ実行状態を保存するために生じるオーバヘッドや、これを隠蔽しようとするためのオーバヘッドにより、細粒度での checkpointing は困難であり、その実行性能を大幅に低下させることなくユーザに対して作業の継続性を提供することは困難であったといえる。

これに対し、近年 FeRAM や MRAM など、通常の RAM と同様のアクセスが可能でありかつ不揮発性をもった半導体記憶デバイスが登場し、実用化されつつある [1, 2]。以下、FeRAM や MRAM 等をまとめ「不揮発 RAM」と呼ぶ。これら不揮発 RAM を主記憶として用いることにより、従来の Persistent OS におけるオーバヘッドの大幅な削減が可能となり、ユーザに対し適切な作業の継続性を提供可能になると考える。

そこで本稿では、主記憶に不揮発 RAM を用いた場合についてプログラムの実行状態を保存するために必要となる事柄について考察を行ない、また、カーネル空間内の checkpointing 手法及びその checkpointing 手法を低オーバヘッドで可能とする、カーネル内のメモリマネージメント手法について述べる。

2 Motivation

2.1 不揮発 RAM

従来の Persistent OS では「主記憶を永続記憶へマッピングする」ということがその本質であった。永続記憶として用いられる装置やデバイスが高速化することにより、アプリケーションの実行状態保存におけるオーバヘッドは低下することとなる。しかしながら、主記憶が不揮発となることによりこの目標をそのまま成し遂げることが可能となる。

現在でも EEPROM や FlashROM など、不揮発半導体記憶デバイスは広く用いられている、これらは書き換え作業時に一度明示的にその内容を消去しなければならず、かつその消去のためにかかる時間も長い。さらに、FlashROM の場合、ブロック単位での消去となる。また、書き換え可能な回数についても主記憶として用いるには少なすぎるものであった。SRAM にバッテリを付加し、不揮発性を実現したデバイスも古くから存在するが、集積度などの問題から特定用途にしか用いられて来なかった。

これに対し、FeRAM や MRAM といった不揮発 RAM はこれらの欠点を補う特徴をもっており、主記憶として用いることが十分に可能なデバイスである (表 1)[1]。

2.2 カーネル空間の保存

しかし、主記憶に不揮発 RAM を用いたとしても CPU 内に存在する状態 (各レジスタ) や周辺デバイスの状態はその実行に伴い不揮発 RAM に保存されるものとはならない。これらの状態が失われていれば、電源切断が生じた後に電源切断時点のプログラムの実行空間が復旧されたとしてもその実行を継続することはできななる。

このため、CPU の状態およびデバイスの状態を含めシステムの状態を復旧可能とするためには、これらは明示的に保存されなければならない。つまり、従来の Persistent OS と同様に明示的な checkpointing により recoverble consistent state を保存する作業が必要となることを意味する。

不揮発 RAM を主記憶として用いた場合、アプリケーションの実行状態はそのときそのアプリケーションにマッピングされたアドレス空間において保存されることとなる。このため、復旧時、アプリケーションに対し、checkpointing 時と同じ物理メモリとそれに対応する仮想アドレス空間を割り当てなければならない。これらの情報は通常カーネル内に存在する。よって、不揮発 RAM を主記憶として用いた場合、カーネル内部の実行状態を含めその checkpointing 時の実行状態を復旧する必要が生じる。

3 目的

前章での議論をもとに本稿では、まず不揮発 RAM を主記憶として用いる際の

- システムをリカバリ可能とするための要求

について考察を行ない、その要求を低オーバヘッドで実現するための

- カーネル内のメモリマネージメント手法
- checkpointing メカニズム

を提案する。

[1]特に MRAM は数年後、現在主記憶として用いられている DRAM をほとんど全て置き換えるだろう、と注目を集めている。

表 1: 半導体メモリの性能比較 (日経エレクトロニクス 1999 年 11-15 号 pp.50 より)

	MRAM	FeRAM	DRAM	FlashROM	SRAM
不揮発性	○	○	×	○	×
書き換え時間 (ns)	10〜	100〜	50	20,000	10
読みだし時間 (ns)	10〜	100〜	50	10〜	10
セル面積 (相対値)	1 以下	1.3	1	0.8	4
書き換え可能回数	10^{15}	10^{12}	10^{15}	10^{5}	10^{15}
最大消費電力 (mW)	10〜400	2	400	100	1100

4 設計

以下、カーネル内部状態を checkpointing 時の状態へ復旧可能とするための要求事項について考察を行なう。

4.1 保存すべきエンティティ

プログラムの実行に伴い、変化するエンティティとしては以下の要素があげられる。

- CPU コンテキスト (各レジスタ)
- オブジェクト
  - 動的にアロケートされたもの
  - 静的にアロケートされたもの
- スタック

また、

- デバイスの状態

もあげられるが、本稿では扱わないこととする。デバイスの状態は適切な位置での checkpointing 及び適切な手法でのデバイスドライバの作成により、電源切断後その状態をリカバリ可能とすることができると考えるためである。

オブジェクト及びスタックはメモリに write アクセスが生じることによって実行に伴い不揮発 RAM に保存される、という点において共通である。よって、以下、実行に伴い保存が行なわれない CPU の状態と、保存が行なわれるメモリオブジェクト及びスタックの 2 種類に分け、考察を行なう。以下、オブジェクト及びスタックをまとめて以下「メモリオブジェクト」と呼ぶこととする。

4.2 CPU 状態

CPU の状態は前述したように checkpointing 時において、明示的に保存を行なう必要が生じる。しかし、保存中の電源切断 (以下、failure と呼ぶ) に対し、その保存された状態を破壊しないようするため checkpointing 時において前回とは別の領域に保存しなければならない。また、リカバリ時にどこに保存された CPU 状態を復帰すればよいか、ということを判別可能とする必要がある。

また、checkpointing 中に failure が生じた場合、CPU 以外の checkpointing 時の状態はメモリ上に存在しているため、CPU の状態さえ保存が終了していればその他の状態は復帰可能である。このため、checkpointing において CPU 状態を最初に保存を行なうべきであると考える。

4.3 メモリオブジェクト

プログラム実行に伴いその変化が保存されていくメモリオブジェクトを checkpointing 時の状態へリカバリ可能とするためには、checkpointing 時の状態が実行にともない書き潰されないようにする必要がある。このとき、以下の 2 通りの方法を考えることが可能である。以下、checkpointing 時のメモリオブジェクトの状態を明示的に残しておくことを「保存」と呼ぶこととする。

1. 静的に保存領域を確保し、checkpointing 時のメモリオブジェクトの状態を保存する
2. 動的に保存領域を確保し、checkpointing 時のメモリオブジェクトの状態を保存する

ここでいう、静的及び動的とは、メモリオブジェクトの保存領域確保に対して、保存作業時に既に確保されているものであるか、逆に保存時にその領域の確保を行なうか、ということを意味する。

1. の方式においては、メモリオブジェクトが checkpointing 後 consistent な状態から inconsistent な状態へと移り変わるとき (メモリオブジェクト write アクセスされる時点を意味する) に、フラグなどを用いてメモリオブジェクトが「アクセスされた (dirty)」という情報を保持しておき、checkpointing 時に実際の保存を行なう方法である。静的にメモリオブジェクトの保存領域を確保し、checkpointing 時のメモリオブジェクトの状態は保存領域に必ず存在するようにする。このとき、メモリオブジェクトの保存作業は checkpointing 時に集約することが可能で、CPU のアイドルタイムを利用するなどの利点を得ることができる。また、静的に保存のための領域が確保されていることにより保存作業が不必要なメモリオブジェクトに対してまでその保存のための領域が保持され、メモリ効率は悪くなるが、保存領域確保のための処理、保存領域が確保できなかった場合の対処など、実行時のオーバヘッドを削減することが可能である。

2. の方式においては、メモリオブジェクトが checkpointing 後最初に write アクセスされる時点においてそのメモリオブジェクトを保存し、checkpointing 時のメモリオブジェクトの状態を残しておくものである。この方式においては保存のために必要な分だけの保存領域を

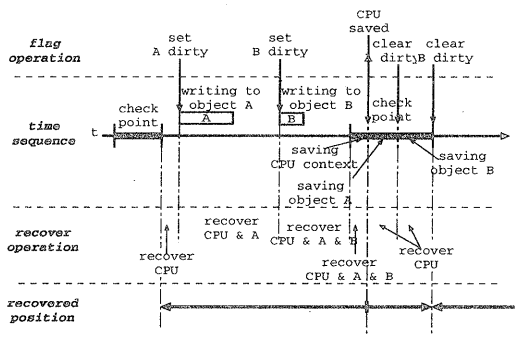

図 1: 静的に保存領域を確保した場合 (手法 1)

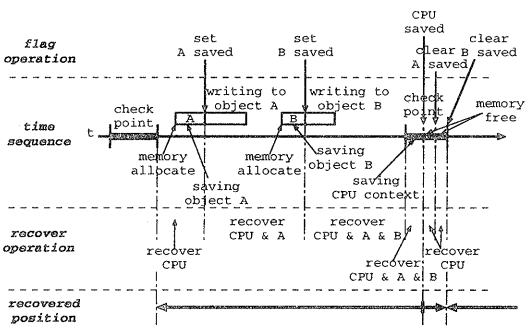

図 2: 動的に保存領域を確保した場合 (手法 2)

動的に確保するため、メモリ効率は良くなる。しかしながら、保存領域を確保しようとしたときにその領域が確保できなかった場合に対する対応をおこなわなければならない。また、checkpointing 後、2 度目以降に同一のメモリオブジェクトがアクセスされた場合、再びその保存作業を行うと、checkpointing 時の状態を書き潰す結果となる。このため 2 度目以降にメモリオブジェクトが write アクセスされた際に保存作業が行なわれないようにしなければならない。

[4] や [9] では実行に伴い、保存されている checkpointing 時の状態を書き潰さないようにするための手法が述べられている。これらの手法では checkpointing 時の状態が保存されている領域と実行に伴い保存が行なわれる領域は同列に扱われていた。しかし、主記憶に不揮発 RAM を用いることによって保存される対象も保存作業時には復旧可能となっているため、保存領域は保存領域として実行時に使用される領域とは別に扱うことが可能となる。このためポインタの切り替えや仮想アドレスのマッピングの切り替えにより、ワークスペースを切り替える必要は無くなる。

リカバリ時については、1. の手法においては「dirty である」というフラグを保持するオブジェクトを保存領域から復帰すれば良い。2. の手法においては保存領域が確保され、かつ保存が完了したものについて復帰を行えば良い。保存領域が確保されていたとしてもその保存中に failure が生じた場合、これを復帰してしまうと、保存のため確保された領域は不定であるため最終 checkpointing 時の状態へメモリオブジェクトを復帰することができなくなる。

上記議論をまとめると、それぞれ checkpointing との関係は図 1、2に示すようになる。

最上段にはメモリオブジェクトに対応するフラグの操作、2 段目には時間の流れとメモリオブジェクトへの write アクセス、3 段目にはその時 failure が生じた際にリカバリ時要求される処理、最下段にはリカバリ後実行が再開される状態を示している。

ここで注目すべきは、1.、2. とも checkpointing 中 CPU 保存後に failure が生じた際の処理であり、この時リカバリ時の処理は failure 時よりも先に進める方向に行なう。これにより、「CPU 状態さえ保存されていればその時点の checkpointing 時の状態が復旧可能」となるが、これを行なうためにはリカバリ時にオブジェクトの保存やフラグのクリア、保存用メモリ領域の開放などの checkpointing の動作を先に進める必要が生じる。

この動作を行なわない場合、手法 1. ではリカバリ後最初の checkpointing 前に再び failure が生じた場合に実際にはアクセスされていないメモリオブジェクトが「dirty である」と判別されることとなりまだ保存が完了していないメモリオブジェクトを復帰しようと試みる可能性が生じる。また、手法 2. ではリカバリ後メモリオブジェクトが最初にアクセスされる際に、もう保存されていると判断され、その状態を保存しようとしなくなる可能性が生じる。

このため、リカバリ時にシステムを過去の checkpointing 時に向かい復旧させるのか、それとも checkpointing における作業を継続しシステムを復旧させるのか、という判別が可能となるようにしなければならない。また、それを判別可能とする方法は atomic に (1 インストラクションの完了で) 行われ、曖昧な状態となることをさける必要がある。

5 実装

以下、前節における要求事項に対しこれを低オーバヘッドで実現可能とする実装方法について述べる。なお、実装に関しては linux-2.2 系列のメモリ管理を参考とした。

5.1 CPU の状態

CPU の状態は 4.2節での議論をもとに、その保存のための領域を 2 つ静的に確保しておく。また、リカバリすべき状態を特定するためのポインタを設けるが、これ

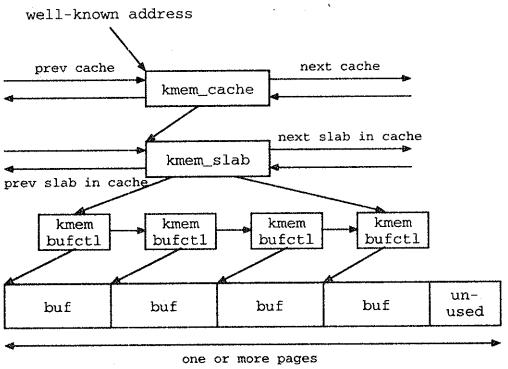

図 3: slab カーネルメモリアロケータの管理構造

をやはり 2 つ設けることとする。これはリカバリ時におこなわれる作業を atomic に切り替えることを可能にするためである。このことについては 5.4節にて詳しく述べる。

5.2 オブジェクト

4.3節での議論から、オブジェクトへの write アクセスが行われる際、そのオブジェクトを特定する機構が必要となる。MMU を用いることによりプログラムに対し透過にこれを特定することが可能であるが、この場合にはページ単位での認識となり、また、ページ単位でのコピーやページテーブル操作に関するオーバヘッドが生じる。また、オブジェクト保存において、不必要なコピーも発生し無駄が多くなる可能性が高い。

そこで、本方式ではオブジェクト単位でこれを特定するようにする。ソースコードの改変を行なう必要はあるが、対象をカーネル内部とするため、カーネル実装者の責任においてこれを行なうこととする。

4.3節で述べた要求を低オーバヘッドで実現するには、

- オブジェクトのアドレスから、オブジェクトの管理構造へのアクセスが高速である
- 動的に保存領域の確保を行う場合、オブジェクトのアロケーション及び破棄が高速である

必要がある。また、オブジェクトのアロケーション及びオブジェクトの破棄自体がシステムの consistent な状態を破壊するためこれに対応する必要がある。

5.2.1 slab カーネルメモリアロケータ

これらの要求に対し、本研究では slab カーネルメモリアロケータ [3] に着目した。

slab ではオブジェクトの種類毎に図 3に示す管理構造を用い、管理を行う。オブジェクトの作成時、ページャよりページを確保し、確保されたページをオブジェクトのサイズに区切る。オブジェクト毎に作成される kmem_bufctl によりオブジェクト単位の管理を行い、ページは kmem_slab により管理される。

オブジェクトのアロケート要求時、作成されたページ内のオブジェクトが要求元へ渡され、オブジェクトの破棄時にはページ内へオブジェクトが返される。オブジェクトの作成及び破棄は kmem_bufctl 内のポインタの操作のみ[2]となり、非常に高速なアロケーションを実現している。また、ページ管理構造体へ、対応する kmem_slab へのアドレスを保持させておくことによりオブジェクトのアドレスから対応する kmem_bufctl を高速に特定することが可能であり、また、高速な破棄を実現可能である。

また、オブジェクトの管理構造体を作成する際に各管理構造体を自身でアロケートしたオブジェクトより作成するよう実装することが可能であり、オブジェクトの作成や破棄に対して、通常のオブジェクトと同様の手法で checkpoint 時の管理構造体の状態を維持することが可能となる。

5.2.2 slab の拡張

以上のような特徴をもつ slab に対し、kmem_bufctl を拡張することにより、保存されるべきオブジェクトの特定を行うこととする。具体的には kmem_bufctl に対し、

- オブジェクト保存領域へのポインタ
- オブジェクトのフラグ

を付加する。

手法 1. においてはオブジェクトのアロケート時にオブジェクト保存領域へのポインタをセットし、実行時及び checkpointing 時にはオブジェクトのフラグの操作を行う。

手法 2. においてはオブジェクトのアロケート時にはオブジェクト保存領域へのポインタは NULL にセットし、実行時にはオブジェクト保存領域へのポインタにオブジェクトと同サイズの領域をアロケートし、実際の保存終了後、オブジェクトのフラグへ保存が完了したことを示すフラグをセットする。checkpointing 時にはポインタで示される保存領域を破棄し、フラグをクリアする。

また、checkpointing 時に処理すべきオブジェクトを全て探索する手間を軽減するため、kmem_slab へもフラグを追加し、kmem_slab にフラグが存在するページのオブジェクトのみ探索すれば良いようにする。上記のように、kmem_slab はオブジェクトのアドレスからオブジェクトを特定する際に特定されるものであるので、これは大きなオーバヘッドとはならない。

動的にアロケートされるオブジェクトについては以上の実装となるが、静的にアロケートされる領域については、slab の管理構造を用いることができない。このため、静的にアロケートされる領域はその領域へのポインタとし、初期化時、動的にアロケートするように変更するよう今回の実装では行った。これにより、静的に確保されるオブジェクトについても上記の議論に還元する。

[2]実際には、ページ内にアロケートされていないオブジェクトが存在しない場合はページャへページを要求し、再び図 3に示す管理構造を作成する

5.2.3 pager

前述したようにslabはページャへのインタフェースとなっている。このため、システム全体のconsistencyを維持するにはページ管理構造体も復帰可能としなければならない。

ページ管理構造体へは前節で述べた方法と同様に、ポインタ及び、フラグを付加する。ページのアロケート要求などによりページ管理構造体が更新されることとなった場合、ポインタへその保存のための領域を確保し、保存終了後、その処理の継続を行なう。

5.3 スタック

スタック領域に対して、始めにオブジェクトがwriteアクセスされた状態となるのはそのスレッドが最初に実行されたときである。このためスケジューラによりスレッドが最初に実行される際、に手法1.ではスレッドのフラグに対し「走行した」というフラグをセットする。手法2.ではスタック領域の保存を行ない、保存されたというフラグをセットする。

保存が行なわれる領域であるが、スレッドが最初に作成された時点のアドレスからcheckpointing時点のスタック領域までの空間を保存することとする。スタックは開始時点より現在のスタックポインタが指すまでの領域が保証されれば良い。MMUを用い保存が必要となるページを特定することも可能であるが、これはスタックサイズがページサイズより大きくなった場合のみ有効であり、今回の実装では行わなかった。

5.4 checkpointingとリカバリ

CPUの保存方法及びオブジェクトの管理手法(保存方法)について述べた。これをもとにcheckpointingを行う。この際、4章で述べた議論より、CPUの保存を最初に行う。そしてその後、オブジェクト及びスタックについて保存(保存領域の解放)及びフラグのクリアを行うが、2種類存在するリカバリ時の処理について、これをatomicに変更する必要があることについて述べた。

これに対してCPUの保存領域を示すポインタを2つ用い、ストア命令1つで状態を変更する手法を用いる。2つのポインタをそれぞれ*current*及び*next*とすると、通常時*current*と*next*は別のCPU保存領域を指しているようにする。そしてcheckpointingを以下の手順で行なう。

1. *next*が指すCPU領域へ現在のCPUの状態を保存する
2. *next*を*current*が指すアドレスと同一のものとする
3. ページ管理構造体、オブジェクト、スタックの処理
4. *current*を以前*next*が指していたアドレスに変更する

この手順により、リカバリ時には、*current*と*next*の内容の比較を行い、*current*と*next*の内容が異なった場合には、以前のcheckpointing時の状態へ向けてアクセスフラグがセットされているオブジェクトの復帰を行えば良い。また、*current*と*next*が同一の場合にはオブジェクトもしくはスタックの保存中にfailureが生じたこととなり、checkpointingの残りの作業を続け復帰すれば良いことになる[3]。いずれの場合においてもCPUの状態は*current*の状態を復帰すればよい。

表2: 実装環境

CPU	モトローラ社 MPC860 50MHz動作 I-cahce、D-cahceとも4kbyte
プログラム領域	MPC860FADS付属 FlashROM サイクルタイム 140nsec
データ領域	ラムトロン社 FeRAM FM1808-70-P×8 サイクルタイム 140nsec

6 評価

プロトタイプOS及びFeRAMを用いたメモリボードを作成し、前述した方法の実装を行った。実装環境は表2のようになっている。

なお、下記の評価はインストラクションキャッシュは有効の状態で行った。プログラムのコンパイルはgcc-2.95.2を用い、-O2オプションを用いている。

6.1 オーバヘッドとなる時間の測定

まず、実行時のオーバヘッドとなる時間の測定及び、checkpointing動作における時間の測定を行った。

図4、図5及び図6、図7はそれぞれ手法1.及び手法2.の実行中におけるフラグ操作などのオーバヘッド及びcheckpointingにかかる時間を示している。write-throughキャッシュ及びwrite-backキャッシュを用いた場合について32～4096byteのオブジェクト10個に対して1000回測定を行ったときの平均である。なお、checkpointing動作にはスタック領域の保存時間は含まれていない。

また、図5、6については同時にwrite-throughキャッシュを用い、32～4096byteのオブジェクトを10個コピーする際のオーバヘッドも同時に示した。

図5及び図6においては実際にオブジェクトの保存作業が生じる。このとき同時に示したmemcpy(オブジェクトの保存に用いている)の実行時間と比較すると、その時間のほとんどがmemcpyに費やされていることがわかる。特に図5においてはmemcpyの実行時間の誤差範囲と言って良い。このことから、提案したオブジェクトの管理手法が十分に高速に機能しているということができる。

[3]実際には上記3からの作業をやり直すこととなる

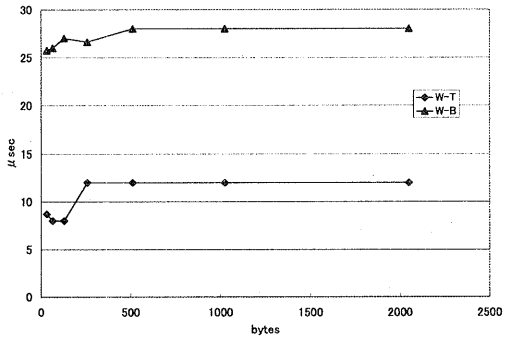

図 4: 実行時のオーバヘッド (手法 1)

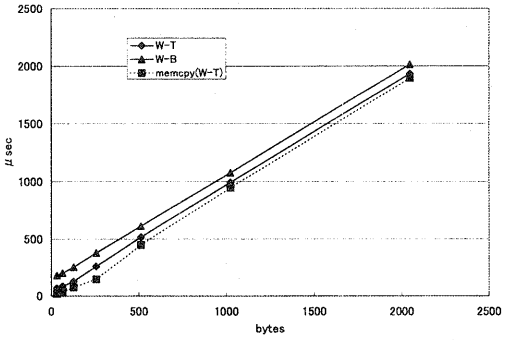

図 6: checkpointing にかかる時間 (手法 1)

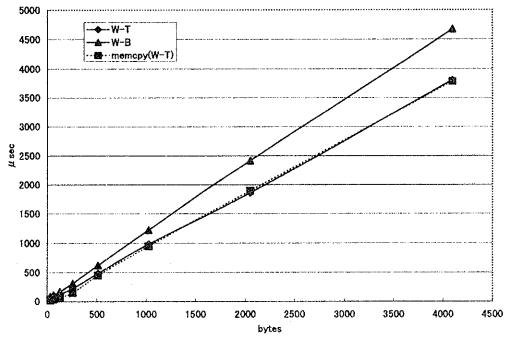

図 5: 実行時のオーバヘッド (手法 2)

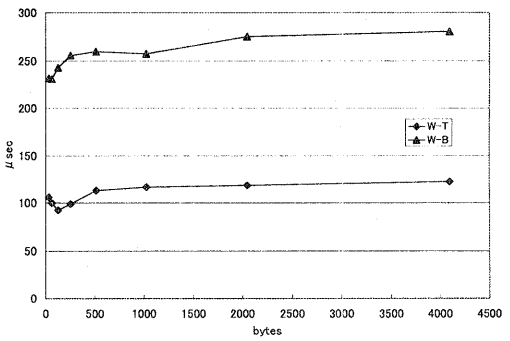

図 7: checkpointing にかかる時間 (手法 2)

全体を通し、write-back キャッシュを用いた場合の方がそのオーバヘッドが大きくなっていることがわかる。これは、メモリ上にオブジェクトが保存されていることを確実とするため、適宜キャッシュのフラッシュを行なっているためである。オブジェクトの保存にたいし、通常の memcpy ルーチンをルーチンを用いたため、この様な結果となったが、キャッシュラインサイズとメモリアクセス速度を考慮し、プリフェッチやフラッシュなどを適切に挿入した memcpy ルーチンを作成することにより、このオーバヘッドは削減可能となるものと考える。

6.2 オーバヘッドの割合の測定

前節で測定した時間が実際のプログラムの実行に対し、どの程度の割合でオーバヘッドとなるかを測定するため、カーネル内でライフゲームを実行した。実際の OS の内部ではライフゲームのような処理が行われることはないが、ある程度の処理 (ステップ数) と checkpointing の動作によるオーバヘッドの関係を見積るには適当であると考える。また、ライフゲームのプログラム自体はインストラクションキャッシュに十分納まるサイズであるため、実際のオーバヘッドは測定結果より小さくなると予想される。静的にオブジェクトアロケートした場合とし、write-through キャッシュを用いた場合と write-back キャッシュを用いた場合についてそれぞれ表 3、4に示す。checkpointing は各世代に 1 回行なっている。なお、表示などの作業は行なわず、純粋に CPU 上での処理のみの時間の測定となっている。

表 3より write-through キャッシュを用いた場合、そのオーバヘッドは 10×10 以外は約 3%程度に留まっていることがわかる。また、表 4より write-back キャッシュを用いた場合においても 10×10 以外は約 8%以下に留まっている。

前述の通り、この値は全ての処理がインストラクションキャッシュに載った状態で行なわれており、実システムにおいてはこの割合はより減少すると思われる。特に OS 内では、インタラプトハンドリングなどキャッシュが無効な状態で実行される部分が多く、さらにこの割合は減少することが予想される。

表 3: ライフゲーム (10000 世代) の実行時間 (W-T)

size	checkpointing なし (msec)	checkpointing あり (msec)	オーバヘッド
10×10	5,916	6,281	6.17%
20×20	24,029	24,771	3.09%
30×30	54,347	55,800	2.67%
40×40	96,867	99,379	2.59%
50×50	152,019	156,939	3.23%

表 4: ライフゲーム (10000 世代) の実行時間 (W-B)

size	checkpointing なし (msec)	checkpointing あり (msec)	オーバヘッド
10×10	5,794	7,026	21.26%
20×20	23,541	25,477	8.22%
30×30	53,247	56,454	6.02%
40×40	94,913	99,889	5.24%
50×50	149,646	155,893	4.17%

また、write-back キャッシュを用いた場合の 10×10 の結果であるが、このとき、データ領域とも全てキャッシュ内に納まる状態で実行されている。前述のように実システムで稼働した場合のオーバヘッドの割合の減少は他よりもより大きくなることが予想される。

10×10 の場合における処理はおおよそ、100 回の単純な代入分数個と 100 回の条件分岐数個のみの処理となる。また、関数コールは存在しない。10×10 程度の処理は通常容易に消費される計算量である。

checkpointing を行なう粒度に付いても考慮する必要はあるものの、10×10 程度の処理に対し write-back キャッシュを用いたとしても約 20%程度のオーバヘッドであること、また、キャッシュミスによるペナルティによってこの割合は大きく減少するであろうことを考慮すれば、十分に低オーバヘッドでカーネル内の状態を復元可能としている、と言うことができると考える。

7 まとめと今後の課題

本稿では、不揮発 RAM を主記憶として用いた Persistent OS の構築を目指し、このときの checkpointing 時の状態保存に対する要求について考察を行ない、また、その要求を低オーバヘッドで可能とする手法について述べた。オブジェクトについてはオブジェクト毎に保存を行ない、その管理は slab カーネルメモリアロケータを拡張し実装を行なった。評価の結果、低オーバヘッドでカーネル内の状態を復旧可能とすることができたと考える。

本研究では、デバイスの状態を復帰するためのデバイスドライバの構築方法に対する考慮を行なわなかった。システム全体の状態を復元するためにはデバイスの状態を復旧可能とすることが必要不可欠である。このためデバイスの取扱に対する研究を行なう必要があると考えている。特にネットワークに関しては、デバイスドライバに限らず、リモートの状態との consistency をいじすることを考慮したプロトコルスタックの拡張が必要になる。また、ユーザレベルの実行状態の保存についての研究を進め、突発的な電源切断に直面したとしても、ユーザが電源復帰後にそれを意識することなく作業の継続性を得ることが可能なシステムの構築を行ないたいと考えている。

参考文献

[1] 日経エレクトロニクス, 解説／DRAM 代替をねらう新メモリ「MRAM」, pp. 49–56. No. 757. 日経 BP 社, 11-15 1999.

[2] 日経エレクトロニクス, ニュースレポート IBM 社と Infineon 社 MRAM を 2004 年に量産へ, p. 25. No. 785. 日経 BP 社, 12-18 2000.

[3] Jeff Bonwich. The slab allocator:an object-caching kernel memory allocator. In *Proc. of the summer 1994 USENIX Conf.*, pp. 87–98, 1994.

[4] Michael F. Challis. Database consistency and integrity in a multi-user environment. In *DATABASE: Improving usability and responsiveness.* Academic Press, 1978.

[5] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating syste. *Computing System*, Vol. 7, No. 3, pp. 289–312, 1994.

[6] Alan Dearle and David Hulse. Operating system support for persistent systems: past, present and future. *Software – Practice-and-Experience*, Vol. 30, No. 4, pp. 295–324, 2000.

[7] Kevin Elphinstone, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Supporting persistent object systems in a single address space. In *Proc.7th POS*, 1996. http://www.cse.unsw.edu.au/ disy/papers/index.html.

[8] Charles R. Landau. The checkpoint mechanism in keykos. In *Proceedings of the IEEE Second International Workshop on Object Orientation in Operating Systems*, 1992.

[9] Raymond A. Lorie. Physical integrity in a large segmented database. *ACM Transaction on Database Systems*, Vol. 2, No. 1, pp. 91–104, March 1977.

[10] J. Rosenberg, A. Dearle, D. Hulse, A. Linderstrom, and S.Norris. Operating system support for persistent and recoverble computations. *Communications of the ACM*, Vol. 39, No. 9, Sept. 1996.

[11] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS:a fast capability system. In *17th ACM Symposium on Operating System Principles(SOSP '99)*, pp. 170–185, 1999.