マルチスレッド環境における実時間ごみ集めのための スレッド再開バリア*

Thread Resumption Barrier for Real-Time Garbage Collection in Multi-Threaded Environment

鵜川 始陽† 花井 亮† 八杉 昌宏† 湯淺 太一†

Tomoharu Ugawa, Ryo Hanai, Masahiro Yasugi, Taiichi Yuasa

† 京都大学大学院情報学研究科

Graduate School of Informatics, Kyoto University

{foosen, hanai, yasugi, yuasa}@kuis.kyoto-u.ac.jp

マルチスレッド環境において，ごみ集め開始時にプログラムが長時間停止することを避けるために，スレッド再開バリアを提案する．スレッドの実行再開時点にバリアを張ることにより，ごみ集め開始時の全スレッドの同期を回避し，スレッド単位のスタックスキャンを可能にする．これにより，多数のスレッド間で同期をとる必要がない実時間ごみ集めが実現できる．

1 はじめに

マルチスレッドやごみ集めはもはや特別先進的な機能ではなくなっている．最近の言語はこれらの機能を言語としてサポートしており，多くのプログラマはこれらの機能を使ってプログラミングをしている．特に Java はユーザインタフェースを持つシステムや，ゲーム，組込みシステムのように実時間応答が必要な場面にも使われている．そのため，ごみ集め全体を小さい不可分処理に分割してアプリケーションプログラム（以後**ミューテータ**と呼ぶ）と並行して少しずつ行う実時間ごみ集めの重要性が増している．

一般にマルチスレッドのアプリケーションを設計する際，実時間制約の厳しい仕事をするスレッドには高い優先度を与えて，優先的に CPU リソースを割り当てる．通常，高優先度のスレッドは多くの時間「待ち」状態にあり，タイマや外部イベントにより時々実行状態になる．一度実行状態になると，優先的に CPU リソースを使って決められた量の仕事をし，再び待ち状態になる．しかし，いくら優先度を高くしても，ごみ集めが起動するとごみ集めの 1 回の不可分処理の間は CPU リソースが利用できなくなり，緊急を要する仕事の完了はそれだけ遅れてしまう．そのため，実時間ごみ集めでは 1 回の不可分な処理の時間をどれだけ短かくできるかが重要な指標のひとつと言える．これをミューテータの停止時間と呼ぶ．

マーク・アンド・スイープ GC やコピー GC はポインタをトレースすることで，ミューテータがアクセスする可能性の残っているオブジェクトを発見し，その補集合をごみとして回収する．このタイプのごみ集めでは，**ルート集合**と呼ばれる，ミューテータが直接参照できる領域を基点としてポインタをトレースする．ルート集合は，大域変数のように，どのスレッドからもアクセスできる**共有ルート**と，特定のスレッドからしかアクセスされない**ローカルルート**に分けられる．ローカルルートには，レジスタやスタック，スレッドローカルストレージなどが含まれる．

既存の多くの実時間ごみ集めは，1）全スレッドのスタックをルート集合として一括してスキャンしたり，2）ごみ集めの開始時か終了時に全スレッドの同期作業が発生したりするため，スレッド数に比例した時間ごみ集めを不可分に実行しなければならず，ミューテータの停止時間の上限を与えることができない．例えばスナップショット GC はごみ集めを細切れにし，メモリ割当てごとに少しずつ進めるが，ごみ集め開始時には全スレッドを止めて共有ルートと各スレッドのローカルルートを一括してスキャンする．また，JIT などによりコンパイルしてネイティブスレッドで実行する場合，1）実行中でないスレッドのレジスタが OS により保護された領域に保存されていて，そのスレッドを実行させなければスキャ

*この研究は文部科学省リーディングプロジェクト「e-society 基盤ソフトウェアの総合開発」の「高信頼組込みソフトウェア構築技術」の一環として行われた．

ンできない，2）ヒープへの書込みバリアを一斉に有効にしなければならないという理由で，全スレッドが同時に同期してローカルルートをスキャンする．

本論文では，スレッド数によらない停止時間の上限を与える実時間ごみ集めを提案する．これは次のようなシステムを対象としている．

- 多数のスレッドが同時に実行されており，それぞれのスレッドには実時間制約を反映した優先度が設定されている．高優先度のスレッドの仕事はできるだけ早く完了させたい．つまり，高優先度のスレッド実行中のごみ集めによる停止時間を短かくし，上限を与えることが目的である．

- それぞれのスレッドの振舞いは分かっていない．各スレッドがどの程度メモリを要求するかおよび，どれだけの時間待ち状態で，どれだけの時間実行されるかはあらかじめ分かっていない．最悪の場合，多くのメモリを要求するスレッドを起動する外部イベントが何回か連続して発生することも考えられる．この場合，高優先度のスレッド実行中にもごみ集めを進めなければ，**飢餓状態**（割当て可能なメモリがなくなった状態）に陥いるかもしれない．

提案するごみ集めはスナップショット GC[1] を改良した方式で，ごみ集め開始時の処理を改良している．その基本的なアイデアはごみ集め開始時にはカレントスレッドのローカルルートのみを一括してスキャンし，他のスレッドのローカルルートはそのスレッドの実行を再開しようとした時か，次にごみ集めを進める時にスキャンするというものである．これにより，ローカルルートのスキャンをスレッド単位で分割することができる．また，ローカルルートにはそのスレッドが実行されない限りアクセスされることはない．そのため，ごみ集め開始時の正確なスナップショットを基にポインタをトレースできる．このごみ集めを**スレッド再開バリア GC** と呼ぶことにした．

スレッド再開バリア GC を実装するにあたっては，正しくごみ集めを行うために，スレッドの実行を再開する時に特別な処理をするための**スレッド再開バリア**が必要になる．また，高優先度のスレッド実行中でもごみ集めを進めるために，カレントスレッド以外のスレッドのローカルルートをスキャンする手段も必要になる．

我々は，ユーザモードスレッド上で実現する方法，OS の機能を拡張してネイティブスレッドで効率よく実現する方法，および，ポーリングを使って OS を拡張せずにネイティブスレッドで実現する方法を考えた．これらのうち，ポーリングによる方法を実際に実装し，マイクロベンチマークを作って性能を測定した．その結果，スレッド数を 500 スレッドまで増やしても停止時間はほとんど増えず，Windows 上での実験ではスレッド数が 500 スレッドの時スナップショット GC に比べ 0.003%の 67μs の停止で収まった．また，プログラム全体の実行時間について，全スレッドが同期してルート集合をスキャンする方式と比較しても，ほとんどオーバヘッドは見られず，Windows 上の実験では，最もオーバヘッドが大きくなっているスレッド数が 500 スレッドの時で 5%程度であった．

以降本論文では，2 章で関連研究に触れ，3 章で提案するアルゴリズムを，ユーザモードスレッドを仮定して述べる．4 章ではネイティブスレッド環境で実現する方法を示し，5 章で議論する．6 章では提案するアルゴリズムを試験実装して得られた実験結果を示す．最後に 7 章で今後の課題を挙げ，8 章でまとめる．

なお，3 章と 4 章ではユニプロセッサのシステムを仮定して説明する．また，インクリメンタル（メモリ割当て毎にごみ集めを進める）方式として説明する．そのため，ごみ集めをする**コレクタ**は特定のスレッドで動作するのではなく，ミューテータのスレッドが必要に応じてコレクタとして動作する．マルチプロセッサのシステムやごみ集めスレッドを用いる方式については 5 章で議論する．

また，以降のごみ集めアルゴリズムの説明では，オブジェクトの状態を白，灰色，黒の 3 色で抽象化するモデル [2] を使う．簡単に説明すると，オブジェクトはまだマークされていない状態では白で，マークされると灰色になる．灰色のオブジェクトをスキャンして，そこから指されているオブジェクトを全てマークすると，そのオブジェクトは黒になりそれ以降はスキャンされない．

2 関連研究

様々な実時間ごみ集めのアルゴリズムが提案されている．スナップショット GC[1] は，マーク・アンド・スイープ GC[2] をインクリメンタルにしたごみ集めである．スナップショット GC は書込みバリアを使って，ごみ集め開始時にコレクタが到達可能であ

るオブジェクトが，ごみ集め中にミューテータが動作することによって到達不可能にならないことを保証する．ごみ集めの最初にごみ集めルートを一括してスキャンし，そこから直接指されるオブジェクトを灰色にする．その後，メモリ割当ての度にマークフェーズ，スイープフェーズの順にごみ集めを少しずつ進める．マークフェーズ中にミューテータによりポインタが上書きされる時は書込みバリアが働いて，上書きされて消えようとしているポインタで指されるオブジェクトを灰色にする．オブジェクトへのポインタはごみ集めルート集合と，ヒープにあるオブジェクトのフィールドに格納されている．スナップショット GC では，ルート集合については一括してスキャンし，ヒープについては書込みバリアを使うことで，仮想的にごみ集め開始時の正確なスナップショットを作り，それに基いてごみ集めをする．

リターンバリア [3] は，スナップショット GC を改良した方式で，スタックのスキャンを細分化する．スタックはプログラムの実行に伴って伸び縮みし，大きさの上限を与えることができない．そのため，スタックを一括してスキャンするごみ集めでは停止時間の上限を与えることができない．リターンバリアは，ミューテータがスタックトップにあるカレントフレームしかアクセスしないことに注目し，コレクタはスタックのトップからボトムに向かってミューテータがリターンするより速くスタックをスキャンする．ミューテータがコレクタを追い越しそうになると，バリアが働いてコレクタを少し先まで進める．リターンバリアは，スキャン済みのフレームの中で最も底にあるフレームに保存されたリターンアドレスを，コレクタを進めるコードのアドレスに置き換えることで効率よく実現できる．これにより，カレントフレームが常にスキャン済みの状態であることを保証している．スレッド再開バリア GC とリターンバリアとの併用については 5 章で触れる．

スレッド再開バリア GC と同様のメカニズムは，データ駆動並列計算機上の並列オブジェクト指向言語処理系 ABCL/EM-4 のごみ集め [4] でも見られる．ハードウェアでスケジューリングを行い処理が細分化されてしまうデータ駆動計算機では，並行ごみ集めを開発する必要があった．文献 [4] では，並行ごみ集めにリードバリアや書込みバリアを用いるのではなく，並列オブジェクトがアクティブになる時にそのオブジェクトの内部を一括してスキャンするアクティブバリアを用いる方法を提案している．我々の提案は，アクティブバリアと同様のメカニズムを一般的なマルチスレッド環境で実現し，積極的にごみ集めを細分化することで，ミューテータの停止時間を小さくすることである．

スライディングビューを使ったスナップショット GC[5] は，複数のスレッドが並列に実行している時に，全てのスレッドを同時に止めずにごみ集めを開始する．スライディングビューとは，異なる時刻にスキャンしたそれぞれのスレッドのローカルルートの状態を基にしたスナップショットのことである．あるスレッドのローカルルートをスキャンしている間も他のスレッドは実行を続けるため，スライディングビューは正確なスナップショットにはならない．しかし，書込みバリアを巧妙に使って，全てのスレッドのローカルルートのスキャンが完了した時刻 t において，ルート集合から直接指されるオブジェクトはマークされて灰色になっているか，または，灰色のオブジェクトから到達可能になっていることを保証している．灰色のオブジェクトから到達できればいずれは灰色になるため，時刻 t においてルート集合から直接指されているオブジェクトは，時刻 t かそれ以降で灰色になる．したがって，時刻 t でルート集合から間接的に到達可能なオブジェクトは，いずれマークされる．スライディングビューを使った GC では，まずスレッド間でハンドシェイクをして，全てのスレッドの書込みバリアを有効にする．その後，やはりハンドシェイクをすることにより，1 スレッドずつローカルルートをスキャンする．ローカルルートのスキャンが始まった時には書込みバリアは有効になっている．マークフェーズが終わるまでの間，共有ルートかヒープへの書込み

$$*a = o;$$

があると，上書きされるポインタ $*a$ により指されるオブジェクトをマークして灰色にする．さらに，カレントスレッドのローカルルートがまだスキャンされていない時の書込みでは，書込むポインタ o をスヌーブバッファと呼ばれるスレッドローカルな領域に記録し，ローカルルートをスキャンする時にローカルルートとして扱う．これによりマーク漏れを防いでいる．ただし，スヌープバッファに記録されたオブジェクトはローカルルートのスキャン前に到達できなくなったとしても，**浮動ごみ**（実際にはごみであるにもかかわらず，回収されない）として残ってしまう．

マルチスレッド環境で効率よくごみ集めをするアルゴリズムには，スレッドローカルヒープを使う方法もある．この方式のごみ集めには，世代別コピーGC の新世代を各スレッドごとに持たせるものが多い [6, 7, 8]．プログラム解析により，スレッドにローカルなオブジェクト，つまり生成したスレッドしか使わないオブジェクトはスレッドローカルヒープに割りつけるようにする．こうすることで，他のスレッドと独立してマイナー GC をすることができ，メジャー GC をしない限り，スレッド数によらない停止時間でごみ集めを完了することができる．これらのアルゴリズムは，メジャー GC が起こらない状況での効率を追及しており，全てのスレッドが関係するメジャー GC のアルゴリズムとは直交している．

スレッド再開バリアに使える OS の機能は，不可分処理の実現をサポートする機能として Brain らによって提案されている [9]．この機能は Daiv らによりマルチプロセッサに拡張され，Soralis 用のカーネルドライバとして実装されている [10]．Brain らはユニプロセッサ上で複数のスレッドが動作する環境で，データを読み出して変更し書き込む操作をロックなしで不可分に実行できる restartable critical section を提案した．ユニプロセッサでは，クリティカルセクションで競合が起きるのは，クリティカルセクション中でプリエンプトされた時だけであることに注目し，スレッドの実行がクリティカルセクションの途中から再開しようとする時はクリティカルセクションの先頭からやり直す．Brain らは，プリエンプトされたスレッドが実行を再開する時はいつでもリカバリコードと呼ばれる特定のアドレスから実行する OS の機能を拡張してこれを実現した．リカバリコードにはプリエンプトされた時のプログラムカウンタの値が渡され，実際に実行を再開するアドレスをリカバリコード中で決定し制御を移す．このリカバリコードはスレッド再開バリアそのものであり，スレッド再開バリア GC で利用できる．

3 提案アルゴリズム

この章では，スレッド再開バリア GC のアルゴリズムを提案する．ここでは簡単のためユーザモードスレッドを仮定し，書込み操作の途中やごみ集めの不可分な処理中にプリエンプトされることはないとする．

```
gcStart(){
  scanSharedRoot();
  for each t in allThreads
    scanLocalRoot(t);
  gcPhase = MarkPhase;
}
```

図 1: スナップショット GC の開始

```
update(Address a, Object o){
  if(gcPhase == MarkPhase)
    shade(*a);
  *a = o;
}
```

図 2: スナップショット GC の書込みバリア

3.1 スナップショット GC

スナップショット GC は，ルート集合全体をごみ集め開始時に一括してスキャンする．ヒープにはごみ集め開始時に書込みバリアを張って，書き込もうとした時に上書きされるポインタで指されるオブジェクトをマークし灰色にすることで，ごみ集め開始時のスナップショットに基づいてごみ集めをする．

スナップショット GC 開始の処理を記述した擬似コードを図 1 に示す．ミューテータがメモリを割当てようとした時，割当て可能なメモリの残量が閾値を下回ると，ごみ集めを開始するために *gcStart* が呼ばれる．*gcStart* は

1. まず共有ルートをスキャンし，
2. 各スレッドのローカルルートをスキャンし，
3. *gcPhase* をマークフェーズに設定することで書込みバリアを有効にする．

gcStart は不可分処理であり，実行している間はミューテータの実行を停止する．したがって，スレッド数が増えると 2. の手順で時間がかかり，停止時間の上限を与えることができない．

書込みバリアの擬似コードは図 2 に示すように単純である．マークフェーズであれば書込みバリアが有効になり，*shade* により，上書きされるポインタが指すオブジェクトがまだマークされていなければ，マークして灰色にする．

マークフェーズ中のメモリ割当てでは，新しいオブジェクトはマークした状態で作られる．新しいオブジェクトにメモリを割当てた時点では，まだそのオブジェクトのフィールドには有効なポインタが入っていない．そのため，新しいオブジェクトは黒とし，そのオブジェクトを基点としたポインタのトレース

はしない．

3.2 基本的なアイデア

スレッド再開バリア GC は，大雑把に言うと，共有ルートとごみ集めを開始したスレッドのローカルルートのみを一括してスキャンし，全てのルート集合を一括してスキャンはしない．その代わりに，ヒープに加えてまだスキャンしていないローカルルートにも書込みバリアを張る．ただし，スタックやレジスタに書込みバリアを張るのはオーバヘッドが大きいため，実際にはスレッドの実行再開にバリアを張る．つまり，まだローカルルートをスキャンしていないスレッドの実行を再開しようとすると，それに割り込んでローカルルートをスキャンする．そのうえで，メモリ割当ての時にまだスキャンしていないローカルルートを少しずつスキャンする．

ローカルルートには，そのスレッドしか書込まない．スレッド再開バリア GC では，スレッド再開バリアを使って，ローカルルートのスキャンがそのスレッドの実行が再開されるまでに終わることを保証し，ごみ集め開始時の正確なスナップショットに基づくごみ集めを可能にしている．

スレッド再開バリア GC では，ローカルルートのスキャンが終わったスレッドは自由に実行してもよい．そのため高優先度のスレッドは，自分のローカルルートだけをスキャンし，他のスレッドのローカルルートがスキャンされるのを待たずに実行を再開できる．

図 3 にスレッド再開バリア GC のごみ集め開始からルートスキャン完了までの処理を記述した擬似コードを示す．ごみ集めは $gcStart$ により開始される．$gcPhase$ がルートスキャンの間，$needScan$ はローカルルートがまだスキャンされていないスレッドを保持し，スキャンが終わったスレッドは $scanned$ に移される．ごみ集め開始前は，全てのスレッドが $scanned$ に入っているように，スレッドの生成時と消滅時にメンテナンスしておく．補助関数 $processOneThread$ は指定されたスレッドのローカルルートをスキャンし，$needScan$ と $scanned$ を更新する．$needScan$ が空になると，全てのローカルルートがスキャンし終わったことを示しているので，マークフェーズに進む．

ルートスキャンフェーズ中にメモリを割当てようとすると，ごみ集めを進めるために $rootScan$ が呼び出され，どれかのスレッドのローカルルートをスキャンする．

```
processOneThread(Thread t){
  scanLocalRoot(t);
  scanned = scanned ∪ {t};
  needScan = needScan\{t};
  if(needScan == ∅)
    gcPhase = MarkPhase;
}
gcStart(){
  gcPhase = RootScanPhase;
  scanSharedRoot();
  needScan = scanned;
  processOneThread(currentThread);
}
rootScan(){
  Thread ∀t ∈ needScan;
  processOneThread(t);
}
onThreadResume(Thread nextThread){
  if(nextThread ∈ needScan)
    processOneThread(nextThread);
}
```

図 3: スレッド再開バリア GC のルートスキャン

スレッドの切替え時にはスレッド再開バリアである $onThreadResume$ が実行される．この時，切替え先，つまり，これから実行を再開しようとしているスレッド $nextThread$ のローカルルートがまだスキャンされていなければ，ここでスキャンする．$needScan$ はルートスキャンフェーズ以外では空になっているため，$gcPhase$ はテストしていない．

3.3 ミューテータの協力

スレッド再開バリア GC でもスナップショット GC と同様に，ごみ集め中にミューテータが書込みとメモリ割当てを行う時に特別な処理をする．特別な処理の内容はスナップショット GC と同じだが，マークフェーズに加えルートスキャンフェーズでも書込みバリアを有効にし，また，新しく割当てるオブジェクトをマークし黒オブジェクトとする．スナップショット GC では一括してルートをスキャンしていたため，ルートスキャンフェーズは存在しなかった．

3.4 スレッドの生成と消滅

非同期にスレッドを生成している最中にごみ集めが開始されると，新しく生成されるスレッドに渡されるオブジェクトがマークされない恐れがある．一般にスレッドを生成する時は，生成されるスレッドに実行すべきコードのアドレスとスレッドが使うデータを渡す．データにはポインタが含まれる．スレッド

が非同期に生成される，つまり，スレッドの生成が完了する前にスレッドを生成した側のスレッドの実行が再開すると，新しいスレッドに渡したオブジェクト o へのポインタを上書きするかもしれない．この時ごみ集めが開始されると，コレクタは新しいスレッドの存在を知らないため，o にはコレクタから到達できず，ごみとして回収されてしまう．この問題は，スレッドを生成する最初のステップで，新しいスレッドに渡すデータを一時的な共有ルートにコピーし，スレッドの生成が完了した時にルートから外すことで解決できる．

スレッドの消滅については，消滅したスレッドのローカルルートには二度とアクセスされないため，特別な処理は必要ない．

最後に，スレッドの生成がごみ集め開始のタイミングに影響を与えることを注意しておく．これについては 3.5 節で議論する．

3.5 ごみ集め開始の閾値

スナップショット GC ではスイープフェーズまでごみは回収されず，割当て可能なメモリは増えない．したがって，ルートのスキャンを分割しているスレッド再開バリア GC では，スナップショット GC よりも早くごみ集めを開始しなければ，飢餓状態に陥る危険がある．ここでは，単純化して全てのオブジェクトが同じサイズと仮定し，スナップショット GC に比べどの程度早くごみ集めを開始すればよいかを見積る．

ある時刻 a にごみ集めを始めたとする．その時システムに存在するスレッド数を $N(a)$，1 回のメモリ割当てで K スレッドのローカルルートをスキャンするとすると，ルートスキャンが終わるまでに最大で $N(a)/K$ 回のメモリ割当てが起る．したがって，スナップショット GC の見積もりより $N(a)/K$ 回早くごみ集めを開始すれば飢餓状態に陥いらない．

ここで，ごみ集め開始の閾値は $N(a)$ に依存して決まる，つまり，新しいスレッドの生成の際に，ごみ集め開始の閾値が変化することに注意しなければならない．新しいスレッドが生成されると，ルートスキャン中に行われるメモリ割当ての回数が増加するため，スレッドを生成する際に新しい閾値を計算し，割当て可能なメモリと比較して必要であればごみ集めを開始する．

4 ネイティブスレッド環境での実装

実行効率を追及した Java 処理系では，バイトコードをネイティブコードにコンパイルし，OS のネイティブスレッドを使って実行する [11]．ネイティブスレッドには次のような特徴があり，スレッド再開バリア GC を実装する妨げとなる．

- スレッドの実行再開は OS が決める．通常の OS では，アプリケーションプログラムはスレッドの実行が中断されたり再開されることには気がつかないため，スレッドの実行再開時に特別な処理をすることが難しい．
- 実行状態にないスレッドのレジスタは OS で保護された領域に保存される．そのため，ごみ集めをしているスレッドが他のスレッドのローカルルートの一部をスキャンできない．なお，スタックは共有しているメモリ空間内にあるため，他のスレッドからでもアクセスできると仮定する．
- 任意のタイミングでプリエンプトされる．図 2 に示したスナップショット GC の書込みバリアでは，$gcPhase$ をテストして書込みバリアが有効になっているか調べてから実際の書込みをするまでの間に書込みバリアが有効になっても，書込みバリアは働かない．そのため，このままではマーク漏れの恐れがある．

この章では，これらの問題への対応を中心にネイティブスレッド環境でスレッド再開バリア GC を実装する 2 通りの方法を述べる．

4.1 OS の機能拡張による実装

まず，OS の機能を拡張してユーザモードスレッドと同様の操作をできるようにする方法が考えられる．スレッド再開バリアを実現するためには，既存の一般的な OS として 2.6 系列の Linux を考えると，スレッドの実行再開を通知する機能と，実行状態にないスレッドのレジスタの値を得る機能を追加すればよい．

4.1.1 スレッドの実行再開を通知する機能

通常スレッドの実行を再開する時は，スレッドがプリエンプトされたアドレスから実行が再開される．これを，あらかじめ設定しておいたハンドラから実行再開するようにすれはよい．この時，プログラム

カウンタとスタックポインタ以外のレジスタはプリエンプトされた時の状態に復元しておく．さらに，プリエンプトされた時のプログラムカウンタをスタックのトップに積み，スタックポインタがその 1 ワード分増えた状態で OS を抜けユーザモードのハンドラに制御を移す．この機能は，文献 [10] のカーネルドライバが実現するものと同じ機能である．

ハンドラ内では，１）作業用のレジスタをスタックに退避し，２）ごみ集めがルートスキャンフェーズにあり，かつ，カレントスレッドのローカルルートがまだスキャンされていなければスキャンした上で，３）スタックに退避したレジスタを戻して，４）スタックトップに積まれたアドレスに制御を移してミューテータの実行を再開すればよい．

ただし，次の 2 点に注意する．ハンドラの実行中にプリエンプトされると，ハンドラが二重に実行される．そこで，ローカルルートのスキャン前にプリエンプトされた時のプログラムカウンタ（これはスタックに積まれている）を調べて，もしハンドラが二重に実行されていればローカルルートをスキャンせずにハンドラを抜ける．

また，バリアによるルートスキャンとコレクタによるルートスキャンが競合する恐れがある．コレクタがスレッド T のローカルルートをスキャンしている最中にプリエンプトされて，スレッド T が実行を再開しようとすると，ローカルルートを二重にスキャンしてしまう．そこで，ローカルルートのスキャンをクリティカルセクションとして排他制御する．このクリティカルセクションは実際にはほとんど競合しないし，もし競合してもコレクタへのタスクスイッチ往復分の時間が余計にかかるだけですむ．さらに，排他制御に restartable critical section[9, 10] を用いれば，もし競合が起きた時はコレクタによるルートスキャンを中止することで，タスクスイッチする必要もなくなる．Restartable critical section は本質的にはスレッド再開バリアと同じもので，簡単に併用できる．

4.1.2 実行状態にないスレッドのレジスタの値を得る機能

実行状態にないスレッドのレジスタの値は，タスク構造体と呼ばれる OS の構造の中に保存される．この値をユーザ空間のメモリにコピーするシステムコールを追加すればよい．他のスレッドのレジスタを読み出す機能を悪用すれば不正なアクセスができてしまうため，読み出し権限は同じメモリ空間を共有するスレッドに限定するなど，慎重に設定しなければならない．

別の方法として，ごみ集めをしているスレッドが他のスレッドのローカルルートをスキャンしようとした時は，そのスレッドの実行を再開させることが考えられる．これによりスレッド再開バリアが働き，ローカルルートがスキャンできる．この方法はスレッドの優先度を操作することで，OS の機能を拡張することなく実現できるが，スレッドを切り替えるオーバヘッドがかかる．

IO やタイマイベントなどを待っているスレッドは長時間ブロックする．このようなスレッドは，シグナルを送ることで強制的にブロックを解除して実行を再開させることができる．しかしメモリ空間を共有しているので，長時間ブロックする場合はブロック前にレジスタの内容をスタックか特定のメモリ領域に書き出してコレクタがアクセスできるようにしておくことで，簡単にローカルルートをスキャンできる．

4.1.3 クリティカルセクション

図 2 に示したスナップショット GC の書込みバリアでは，スレッド S が書込み操作

$$update(\&o_1.f, o_2);$$

を行なっている最中にプリエンプトされて，スレッド T がごみ集めを開始すると問題が起こることがある．

1. スレッド S が図 2 の 2 行目で書込みバリアが有効になっているかどうかをテストした直後にプリエンプトされる．
2. スレッド T がメモリを割当てようとして，ごみ集めを開始する．
3. スレッド U のローカルルートをスキャンする．
4. スレッド U が $o_1.f$ の値を読み込む．
5. スレッド S が図 2 の 4 行目で $o_1.f$ を上書きする．

最初に $o_1.f$ から指されていたオブジェクト o_3 が $o_1.f$ 以外から指されていなければ，ステップ 5. 以降では o_3 ポインタがスレッド U のローカルルートにしか存在しなくなる．しかし，スレッド U のローカルルートはステップ 3. でスキャンし終わっているため，o_3 がマークされずごみとして回収されてしまう．

書込み操作とごみ集め開始のコードを排他制御すれば問題は解決できるが，書込みのたびにロックを

獲得するとオーバヘッドが大きい．軽い実装には，書込み操作のプログラムが決まった形をしていることを利用する．ローカルルートをスキャンする時，プログラムカウンタが指すアドレス周辺のコードを調べて，書込み操作の最中であることが分かれば，書込み先のアドレス（これもプログラムとレジスタの内容から分かる）から指されるオブジェクトをマークし灰色にする．

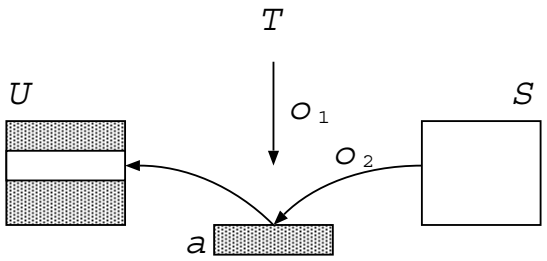

図 4: 競合の例

4.2 ポーリングによる実装

スレッドの実行再開後，ポーリングによりごみ集めの開始に気付いてローカルルートをスキャンする方法も考えられる．この方法では，ごみ集め開始後，スレッドの実行が再開してからポーリングするまでの間，実行が進みローカルルートが書換えられるため，正確なスナップショットは得られない．しかし，その実行の間に共有ルートやヒープへの書込みが起こらなければマーク漏れは生じない．なぜなら，ローカルルートからしか参照されていないオブジェクトへのポインタが上書きされるとマークの機会が失われるが，ミューテータもそのオブジェクトにがアクセスすることはできないためである．

4.2.1 競合

ポーリングまでの間に起こる共有ルートやヒープへの書込みでは，図 2 に示したスナップショット GC の書込みバリアを使って，消えようとしているポインタで指されるオブジェクトを灰色にすればマーク漏れは生じない．ただし，書込み操作の途中でごみ集めが開始されると，4.1.3 節で述べた問題が起こる．

これは次のようにして 2 段階で解決する．4.1.3 節で述べた問題は書込み操作開始時と実際に書き込む時の間に書込みバリアが有効になることに原因がある．そこで，ルートスキャンフェーズの前に同期フェーズを設けて，全てのスレッドとハンドシェイクをしてごみ集めの開始を伝える．全てのスレッドで書込みバリアが有効になってはじめてローカルルートをスキャンするルートスキャンフェーズに進めば，ルートスキャンフェーズでの書込み操作開始時には必ず書込みバリアが有効になっている．

しかし，まだマーク漏れの原因となるもう 1 つの実行パターンがある．こののパターンは，図 2 の 3 行目の *shade* と 4 行目の実際の書込みの間に割り込まれる可能性があることに原因がある．実際に書き込む時には，書込み先のアドレス a に 4 行目でマークしたオブジェクトとは別のオブジェクトへのポインタが書かれているかもしれない．もし，図 4 で示すように，スレッド T によるアドレス a への o_1 の書込みが割込まれて，まだスキャンが終わっていないスレッド S のローカルルートから，アドレス a を経由して，スキャンが終わったスレッド U のローカルルートに白オブジェクト o_2 のポインタが渡されれば，o_2 はマーク漏れになるかもしれない．そこで，文献 [12] やスライディングビューによる方式 [5] と同様に，同期フェーズのハンドシェイクを 2 回に分ける．最初のハンドシェイクからルートスキャンが終わるまでの間は，書込みの後に書き込んだポインタが指すオブジェクト（例では o_2）も灰色にする．

最後に，コレクタがプリエンプトされてポーリングが起こると，同じスレッドのローカルルートのスキャンで競合する恐れがあることに注意する．そのため，コレクタとポーリングによるローカルルートのスキャンは排他制御する．

4.2.2 優先度の制御

ごみ集めを進めるには，低優先度のスレッドのローカルルートもスキャンしなければならない．ネイティブスレッドでは，カレントスレッド以外のスレッドのレジスタをスキャンできないため，高優先度のスレッドが低優先度のスレッドのローカルルートをスキャンしてごみ集めを進めることはできない．そこで，高優先度のスレッドがごみ集めを進める時は，スキャンしたいローカルルートを持つスレッドの優先度を一時的に高くし，そのスレッドがローカルルートをスキャンするのを待つ．優先度を上げられたスレッドは，ポーリングによりごみ集め開始に気付きローカルルートをスキャンすると，自分で優先度を戻す．同期フェーズのハンドシェイクを進めるため

にも，これと同じ優先度の操作をする．

4.2.3 コード例

以上をまとめたコード例を図 5 と図 6 に示す．メモリ割当ての時に，割当て可能なメモリが一定量以下であることが分かれば，gc が呼び出されごみ集めが開始される．最初は $gcPhase$ が $Inactive$ に設定されており，$scanned$ には前回のごみ集めにより全てのスレッドが入れられている．これを $needSync_1$ に移し同期フェーズに進む．続くごみ集めでは，$needSync_1$ から 1 スレッドとり出してはそのスレッドの優先度を上げ実行を再開させる．優先度を上げられたスレッドはポーリングまで実行し，自身を $needSync_1$ から $needSync_2$ に移して優先度を元に戻す．全てのスレッドが $needSync_2$ に移ると，$Sync_2Phase$ に進み同様の手順で $needSync_2$ から $needScan$ に 1 スレッドずつ移す．スレッドが属する集合は，どの書込みバリアが働くかに影響している．$needSync_2$ か $needScan$ に属する時に通常のスナップショット GC にはない書込みバリア（4.2.1 節）が働く．全てのスレッドが $needScan$ に移ると，共有ルートをスキャンしてルートスキャンフェーズに進む．

なおコード例では省略したが，gc の中で自分のスレッドのハンドシェイクやルートスキャンがまだ終わっていなければ，他のスレッドの優先度を上げるのではなく $polling$ 相当の処理をして，自分のスレッドのハンドシェイクやルートスキャンをするようにしている．

4.2.4 ごみ集め開始の閾値の変更

ポーリングによる実装では 3.5 節の見積りよりも早くごみ集めを開始しなければならない．以下にルートスキャンが完了するまでに起こるメモリ割当てが増える要因を挙げる．ここで，同期フェーズ開始の時刻 a におけるスレッド数を $N(a)$ とする．また，1 回のメモリ割当ての時にハンドシェイクするスレッドは 1 スレッドとする．

1. メモリ割当の途中に割り込んだスレッドがごみ集めを開始する可能性がある．これによるごみ集め回数の増加は最大で $N(a)-1$ 回である．
2. 2 回の同期フェーズを追加した．そのため，ルートスキャンが完了するまでに，$3N(a)$ 回のポーリングが必要になる．
3. 高優先度のスレッドが低優先度のスレッドの優

```
gc(){
  synchronized{
    switch(gcPhase){
    case Inactive :
      gcPhase = Sync1Phase;
      needSync1 = scanned;
      break;
    case SynciPhase(i = 1, 2) :
      Thread ∀t ∈ needSynci;
      setHighPriority(t);
      wait;
      break;
    case RootScanPhase :
      Thread ∀t ∈ needScan;
      setHighPriority(t);
      wait;
      break;
    ...
    }
  }
}
polling(){
  Thread ct = currentThread;
  switch(gcPhase){
  case Sync1Phase :
    if(ct ∈ needSync)
      synchronized{
        needSync2 = needSync2 ∪ {ct};
        needSync1 = needSync1\{ct};
        if(needSync1 == ∅)
          gcPhase = Sync2Phase;
        notify;
      }
      restorePriority();
    break;
  case Sync2Phase :
    if(ct ∈ needSync)
      synchronized{
        needScan = needScan ∪ {ct};
        needSync2 = needSync2\{ct};
        if(needSync2 == ∅){
          scanSharedRoot();
          gcPhase = RootScanPhase;
        }
        notify;
      }
      restorePriority();
    break;
  case RootScanPhase :
    if(ct ∈ needScan)
      synchronized{
        scanLocalRoot(ct);
        scanned = scanned ∪ {ct};
        needScan = needScan\{ct};
        if(needScan == ∅)
          gcPhase = MarkPhase;
        notify;
      }
      restorePriority();
    break;
  }
}
```

図 5: ポーリングによるスレッド再開バリア GC の実装

```
update(Address a, Object o){
  if(gcPhase ∈ {Sync_iPhase(i = 1, 2),
        RootScanPhase, MarkPhase})
    shade(*a);
  *a = o;
  if(currentThread ∈ needSync_2||
    currentThread ∈ needScan)
    shade(o);
}
```

図 6: ポーリングによるスレッド再開バリア GC の実装（書込みバリア）

先度を上げてポーリングさせる場合，高優先度のスレッドだけでなく低優先度のスレッドもメモリを割り当てる可能性がある．そのため，各フェーズの最初のスレッドによるポーリング以外では，1 回のポーリングに対して 2 回メモリが割り当てられる可能性がある．2. の要因と合わせて考えると，最大で $3(1 + 2(N(a) - 1))$ 回メモリが割り当てられる可能性がある．

4. 同期フェーズ中のスレッド数の増加は，新しいスレッドをルートスキャン完了の状態にして生成することで見積りに影響しないようにできる．

以上より，最大で $7N(a) - 4$ 回のメモリ割当ての可能性がある．

4.2.5 Azatchi らのごみ集めとの比較

スレッド再開バリア GC のポーリングによる実装は，Azatchi らのスライディングビューを使ったごみ集め [5] と非常によく似ている．実際，ポーリングによる実装ではスナップショットではなくスライディングビューを使っている．

両者の違いは，スレッド再開バリア GC はポーリングでごみ集め開始に気付くと，直ちにハンドシェイクをしたりローカルルートをスキャンしたりしてごみ集めを進めるるのに対して，Azatchi らのごみ集めではあくまでごみ集めスレッドがルート集合をスキャンする．さらに，スレッド再開バリア GC では高優先度のスレッドがメモリを割り当てる時でも，低優先度のスレッドのローカルルートがスキャンされていなければ，そのスレッドの優先度を一時的に高くすることでごみ集めが確実に進むようにしている．そのため，スレッド再開バリア GC ではごみ集め中に割り当てられるメモリ量が見積りやすい．一方で，スレッド再開バリア GC では短い時間で連続してスレッドが切り替わると，連続してローカルルートをスキャンすることになりごみ集めの負荷が増える．高優先度のスレッドが少なければ，高優先度のスレッド間でスレッドが切り替わることはほとんどないが，高優先度のスレッドが多いと高優先度のスレッド間でのスレッド切替えが頻発する恐れがある．

5 議論

5.1 最大停止時間の保証

ローカルルートの大部分を占めるスタックは，プログラムの実行により動的に伸び縮みし，非常に大きくなることもある．スレッド単位でローカルルートのスキャンを分割しても，大きなスタックを含むローカルルートを一括してスキャンすると停止時間が問題となる．

リターンバリア GC[3] はスタックを関数フレーム単位に分割してスキャンする．これを併用して，スレッド再開バリア GC でスレッド単位に分割したローカルルートを，さらに関数フレーム単位に分割してスキャンすると，ルートの大きさによらずに最大停止時間を保証できる．

5.2 マルチプロセッサ

マルチプロセッサ環境では 2 つ以上のスレッドが同時に実行されるため，スレッドの実行再開にバリアを張るだけでは，ごみ集めが開始した時に実行しているスレッドのスナップショットが得られない．そこで，まず全てのプロセッサで同期した上で，スレッド再開バリアを張りごみ集めを始める．

ごみ集めを開始する時は，ほかのスレッドにごみ集め開始を伝えるためにごみ集めフェーズをルートスキャンフェーズに進めるが，他のプロセッサで実行されているスレッドがこれに気付くまで，そのプロセッサの実行を止める．これは，スレッドの優先度を高くしてビジーウエイトすることで実現できる．それぞれのスレッドはごみ集めの開始をポーリングしている．ごみ集め開始に気付くと，そのスレッドを実行しているプロセッサの実行を止めて全てのプロセッサが同期するのを待つ．最後のブロセッサは同期完了を他のプロセッサに伝えて，ごみ集めを開始する．

この方法では，プロセッサ間の同期が完了するまでの時間は無駄になるが，この時間は最大でもポーリングの間隔 1 回分を実行する時間にしかならない．

```
int fib(int n) {
  if (n > 15)
    gc_alloc(10);
  else if (n > 12)
    POLL();

  if (n <= 1)
    return 1;
  else
    return fib(n-1)+fib(n-2);
}
void* start_routine(int id) {
  sync(id);
  while (count[id]-- > 0)
    fib(20);
}
```

図 7: ベンチマークプログラム

5.3 ごみ集め専用スレッド

これまでは，メモリ割当ての時に少しずつごみ集めを進めるインクリメンタル GC としてスレッド再開バリア GC を説明してきたが，ごみ集め専用スレッドを用いる方式でも実現することができる．しかし，ごみ集め専用スレッドを用いる方式では，飢餓状態になる前にごみ集めが確実に終わるかどうかは，スケジューラに依存する．ごみ集めスレッドの優先度を使ってある程度スケジューラを制御できるが，ごみ集めにかかる時間を見積るのは難しい．そのため，ごみ集めにかかる時間を余分に見積もって，必要以上に早くごみ集めを開始しなければならない．

スレッド再開バリア GC では，スライディングビューを使ったごみ集めに比べルートスキャンにかかる時間が見積りやすい．そのため，ごみ集めスレッドを使う場合でも，インクリメンタルな処理と併用して [13] ごみ集め開始を遅らせるのがよい．

6 性能評価

4.2 節で示した方法で，スレッド再開バリア GC をポーリングを使って試験実装した．C 言語用のライブラリとして保守的ごみ集めを実装し，ポーリングや書込みバリアはミューテータのプログラムに明示的に挿入した．なお，マークフェーズとスイープフェーズは今回の提案と関係ないため一括して行い，その時間は最大停止時間には含めていない．[1]

実験には，高優先度のスレッドの最大停止時間を計測するために作った人工的なプログラムを用いた．ベンチマークプログラムは，図 7 の start_routine を実行するスレッドを複数同時に実行する．そのうち，1 スレッドだけを高優先度に設定し，残りは通常の優先度で実行する．start_routine の先頭の sync は，全てのスレッドが起動して優先度が設定されるのを待つために同期しているもので，ごみ集めとは関係ない．それぞれのスレッドは決められた回数だけ再帰呼出しを使ったフィボナッチ数の計算を繰り返す．このとき，再帰の浅い呼び出しでは，POLL によりポーリングする．さらに再帰の浅い呼出しでは gc_alloc(10) によりメモリの割当てを行う．ただし割り当てたメモリはすぐにごみにしてしまう．また，このベンチマークプログラムでは，書込みバリアは使っていない．なお，スレッド数によらず，プログラム全体でフィボナッチ数を計算する回数は同じになるようにループ回数を調節した．

通常のプログラムでは，高優先度のスレッドはイベントに反応して短時間実行し，多くの時間はイベント待ち状態になっている．そのため，高優先度のスレッドの実行中にごみ集めが起こることはごく稀にしかない．このベンチマークでは，高優先度のスレッドが待ち状態になることなく仕事を続けるため，このような稀な状況での停止時間を測定できる．

ベンチマークプログラムは，Windows と Linux にそれぞれ用意した．Windows 用は Win32 API により実装を試みたが，同じミューテックスやイベントなどで複数のスレッドが待っている時，実行が再開される順序がスレッドの優先度通りにならず，そのために高優先度のスレッドが長時間待たされることがあった．そこで，直接 Win32 API を使うのではなく，独自に優先度通り実行が再開されるような排他制御のルーチンを作成し，スレッド再開バリア GC を実装した．この排他制御ルーチンは内部で Win32 API を使っているが，スレッドごとに異なるイベントを待つようにしており，ユーザレベルで優先度に従って実行を再開させるスレッドを選択する．ただし，内部で使っている Win32 API のミューテックスで稀に少数のスレッドが競合することがあり，その時は高優先度のスレッドが低優先度のスレッドに追い越されてしまうことがある．Windows では，通常のプロセスのスレッド優先度は動的に変化する．これを避けるため，ベンチマークプログラムは実時間領域（REALTIME_PROCESS_CLASS）で実行した．

Linux 用は POSIX Thread ライブラリである Linux Threads ライブラリを使った．スレッドの優先度を設定するために，高優先度のスレッドはラウン

[1] マークフェーズとスイープフェーズの時間はヒープサイズ 512KB で，ごみ集め開始の閾値が 256KB の時，Windows を使った実験で 80μs〜260μs 程度だった．

ドロビンスケジューラに設定した．また，ごみ集めで一時的に優先度を上げる時は FIFO スケジューラに設定した．なお，実験の途中で Linux Threads ではスレッドの優先度を下げる際にスレッドライブラリの中で優先度逆転が起こっていることに気づいた．

Windows と Linux での，高優先度のスレッドの最大停止時間と，プログラム全体の実行時間を図 8〜図 13 に示す．TRB-GC がスレッド再開バリア GC の結果を，SS-GC がスナップショット GC の結果を表わす．実験環境は，以下の通りである．

- Windows

  OS: Windows XP SP2
  CPU: Intel Celeron M 1.4GHz
  Cache: 1MB
  コンパイラ: Visual C (VS.Net 2005)
  最適化オプション: /O2 /Ob2

- Linux

  OS: Linux 2.6.8 (debian)
  CPU: Intel Xeon 3.2GHz
  Cache: 512KB (+1MB L3)
  コンパイラ: gcc version 3.2
  最適化オプション: -O2

また，ヒープサイズは 512KB としごみ集め開始の閾値は，空きメモリ 256KB とした．

Windows での実験結果ではスレッド数によらず，スレッド再開バリア GC が高優先度のスレッドの最大停止時間を抑えるのに役立っているのが分かる．図 8 の 300 スレッドと 350 スレッドでスレッド再開バリア GC の長い停止時間が記録されている．Windows のベンチマークプログラムでは，スレッド数によらず時々このような長い停止時間が見られたため詳しく調べたところ，Win32 API のミューテックス待ちから実行再開する順序に原因があることが分かった．これを除くと，スレッド再開バリア GC での停止時間は 1 スレッドで 33μs に対して，500 スレッドで 67μs と 2 倍程度にしか増えていない．2 倍に増えた原因は今後検証する必要がある．

スレッド数が少ない時の停止時間では，1 スレッドの時にスナップショット GC が 24μs に対してスレッド再開バリア GC は 33μs と，やや長く停止しているが，2 スレッドになるとスナップショット GC で 40μs と倍近くになっているのに対して，スレッド再開バリア GC では 32μs と 1 スレッドの時と変わっていない．4 スレッド以上でも同じ傾向が続いている．

プログラムの総実行時間は図 10 の傾きより，スレッド数に比例してかかるオーバヘッドがスレッド再開バリア GC の方がスナップショット GC よりも大きいことが分かる．1 スレッド当たりのオーバヘッドを計算すると，スナップショット GC が 302μs に対してスレッド再開バリア GC では 645μs だった．これらは，ハンドシェイクのための優先度の操作や，スレッドの切替えによるオーバヘッドと考えられる．

Linux での実験結果は，イレギュラなデータが多い．スレッドが少ない時の結果が乱れているのは優先度逆転によるものと考えられる．スレッドが多い時に，超線型に停止時間や総実行時間が長くなっているのは，スレッドライブラリがスレッドを線型リストで管理しており，これをスキャンする時間と回数の両方が増えることが原因ではないかと考えられる．それを除けば，Windows での結果と同じ傾向が見られる．

次にスレッド数を 500 に固定して，ごみ集め開始の閾値となる割当て可能なメモリ量を 128KB と 64KB に変えて実験した．ベンチマークプログラムでは，1 回のメモリ割当てでメモリブロックのヘッダを含めて 28 バイト割り当てる．したがって，4.2.4 節の議論より，500 スレッドでごみ集め開始までにおよそ 98KB のメモリが割り当てられる可能性がある．実験の結果，128KB では飢餓状態に陥らなかったが，64KB では飢餓状態に陥ることがあった．なお，優先度の制御を行わなければ 256KB を閾値としても飢餓状態に陥った．

以上よりスレッド再開バリア GC では，

- スレッド数が増えても比較的小さなオーバヘッドで 1 スレッドの時とほとんど変わらない最大停止時間を実現でき，
- ごみ集め開始の閾値も見積りやすい

ことが確かめられた．

7 今後の課題

今回はポーリングによる実装のみの実験しか行えなかった．OS の機能拡張による本物のスレッド再開バリアを実現し，それを使ったごみ集めの性能を測定することが今後の課題である．また，今回は汎用 OS を使って測定したため，スレッドライブラリに起

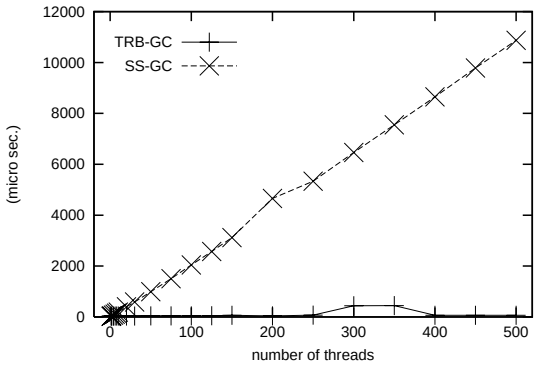

図 8: 最大停止時間（Windows）

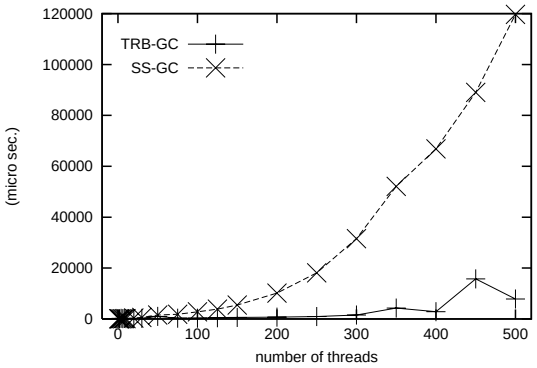

図 11: 最大停止時間（Linux）

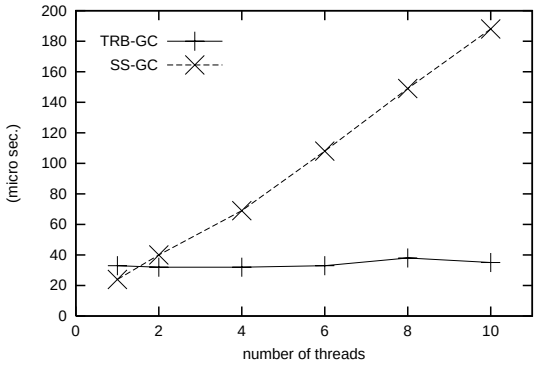

図 9: 最大停止時間（Windows，スレッド少数）

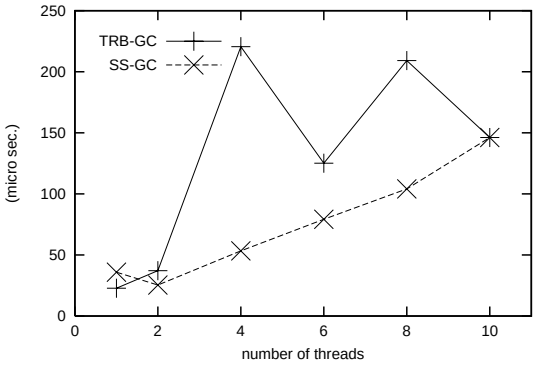

図 12: 最大停止時間（Linux，スレッド少数）

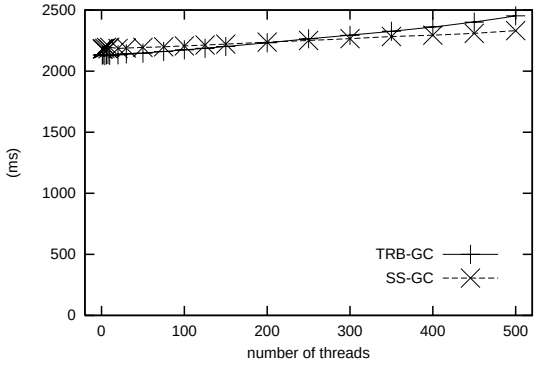

図 10: 総実行時間（Windows）

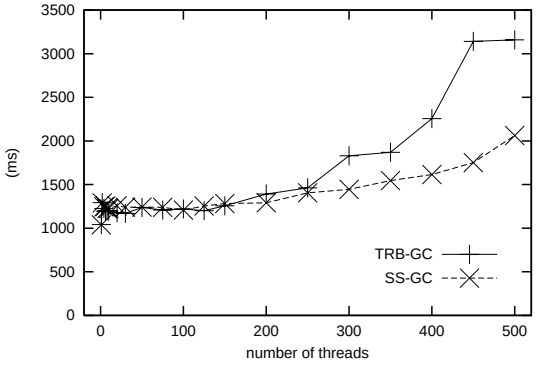

図 13: 総実行時間（Linux）

因するイレギュラな結果が多く見られた．実時間 OS 上で実験して，どのような結果が得られるかも興味がある．

ベンチマークプログラムについても，今回は人工的なベンチマークプログラムを使った性能測定しか行っていないため，Jikes RVM[14] のような本格的な言語処理系に実装し実用的なプログラムでの振舞いも調べる必要がある．

最後に，リターンバリアとの併用（5.1 節）やマルチプロセッサでの実装（5.2 節）と実験も今後の課題である．

8 まとめ

本論文では，スレッドの実行再開にバリアを張るスレッド再開バリアと，それを使った実時間ごみ集めを示した．スレッド再開バリア GC の実装方法は，ユーザモードスレッドでの実装方法に加えて，ネイティブスレッド上で実現する方法を 2 種類示した．そのうち，ポーリングを使う方法を実際に実装し，性能を測定した．その結果，スレッドライブラリによる性能低下が比較的小さいと思われる Windows において，実行中のスレッドが 1 スレッドと 500 スレッドの時で最大停止時間が約 2 倍しか変わらなかった．また，スレッド増加に共なうオーバヘッドの増加もスナップショット GC に比べて 2 倍程度と小さかった．また，オーバヘッドの実行時間全体に占める割合も 500 スレッドの環境で 5%と実用に耐えるものであった．さらに，ごみ集め開始の安全な閾値も容易に見積ることができた．

参考文献

[1] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, Vol. 11, No. 3, pp. 181–198, 1990.

[2] Richard Jones and Rafael Lins. *Garbage Collection*. John Wiley & Sons, 1996.

[3] 湯淺太一, 中川雄一郎, 小宮常康, 八杉昌宏. リターン・バリア. 情報処理学会論文誌：プログラミング, Vol. 41, No. (PRO 8), pp. 87–99, 2000.

[4] 八杉昌宏. 並列オブジェクト指向言語のためのガーベジコレクタ. 情報処理学会論文誌, Vol. 39, pp. 1691–1699, 1998.

[5] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'03)*, 2003.

[6] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the 2nd international symposium on Memory management (ISMM'00)*, pp. 18–24, 2000.

[7] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for java. In *Proceedings of the 3rd international symposium on Memory management (ISMM'02)*, pp. 76–87, 2002.

[8] 千葉雄司. スレッド別ごみ集めにおけるライトバリアの高速化. 情報処理学会論文誌：プログラミング, Vol. 45, No. SIG 5(PRO 21), pp. 53–61, 2004.

[9] Brain N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth international conference on Architecture support for programming languages and operating systems (ASPLOS)*, pp. 223–233, 1992.

[10] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd international symposium on Memory management (ISMM'02)*, 2002.

[11] Sun Microsystems, Inc. The java hotspot performance engine architecture, 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[12] Damine Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 70–83, 1994.

[13] 花井亮, 岡田慧, 湯淺太一, 稲葉雅幸. ロボット行動ソフトウェア環境に適した実時間ごみ集め. コンピュータソフトウェア, Vol. 22, No. 3, pp. 173–178, 2005.

[14] B. Alpern, S. Augart, S. M. Blackburn, M. Burrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM System Journal*, Vol. 44, No. 2, pp. 399–417, 2005.