ダーティビット情報を用いた世代別ごみ集めの GNU Emacs への実装

小林広和[†] 寺田実[†]

世代別ごみ集めは，ごみ集めの実行によって起こるプログラムの通常処理の停止時間を短縮できるので，対話的なアプリケーションや実時間的なアプリケーションに適している．しかし，世代別ごみ集めを実装する場合には，プログラムの通常処理で起こる古いオブジェクトへの書き換えの検出を行う必要があり，その書き換えの検出のために必要なプログラムの修正コストや，書き換えの検出によって起こる通常処理の実行時のオーバヘッドが問題となる．本論文では，テキストエディタである GNU Emacs に，古いオブジェクトの書き換えの検出にダーティビット情報を用いた世代別ごみ集めを実装し，プログラムの修正のコストやプログラム実行時のオーバヘッドの問題を回避できることを実験によって示す．

Implementation of Generational Garbage Collector Using Dirty Bit Information to GNU Emacs

Hirokazu Kobayashi[†] and Minoru Terada[†]

Generational garbage collection can reduce mutator pause times due to garbage collection, so it is appropriate for a interactive application and a real time application. But to implement a generational garbage collector, we must keep track of rewritings of old object by mutator, so it is a problem that the cost of program's modification due to tracking of rewritings and the overhead of mutator due to it. In this paper, we implement a generational garbage collector to text editor GNU Emacs which uses dirty bit information for tracking old objects rewritings, and we present the result of experiment which shows its problem is avoided.

1. はじめに

GNU Emacs は世界中で利用されているテキストエディタであり，多くの利用者が存在する．GNU Emacs は Emacs Lisp という Lisp の方言を持つ．Emacs Lisp は動的スコープを持ち，そのシンボルの値の実装には，浅い束縛を用いている．浅い束縛は，シンボルの値の束縛が変化すると，そのシンボルの値スロットを現在の値の束縛に書き換えることにより，値の参照を速くできるのが特徴である．つまり浅い束縛ではシンボルの値スロットの書き換えが頻繁に起こると考えられる．

世代別ごみ集め[7),9)]を実装する場合には，古い世代から若い世代を指すポインタを何らかの方法で検出する必要がある．このような世代間参照を検出する方法として，さまざまな手法が提案[5),11)]されている．世代間参照の検出は，オブジェクトの書き換えを検出することによって行われる．その方法は，ソフトウェアによるものとハードウェアによるものの 2 種類に分けることができる．ソフトウェアによって検出する方法は，専用のハードウェアが利用できない環境でも利用できるという利点がある．しかし，ソースコード上でオブジェクトの書き換えが行われる場所において，検出するためのコードを付け加えなければならず，時としてそのための変更が膨大な箇所にわたるのでバグの原因となりやすいという欠点がある．それに対し，ハードウェアによる方法は検出のためのハードウェアを必要とするが，オブジェクトの書き換えを検出するために必要なソースコードの変更が少ないという点で有利である．

浅い束縛を用いた実装では，オブジェクトの書き換えが頻繁に起こると考えられるので，世代間参照の検出を行うためのコストが大きいと考えられる．特にソフトウェアによる世代間参照の検出を行った場合は，オブジェクトの書き換えがあるたびにそのコードが実行されるので，書き換えを検出するために起こるプロ

† 東京大学大学院工学系研究科
School of Engineering, University of Tokyo

グラムの実行時のオーバヘッドは大きくなる．そのためハードウェアを用いた世代間参照の検出が望ましく，本論文ではダーティビット情報を用いて世代間参照の検出を行う．

現在，GNU Emacs には停止型のマークスイープ方式のごみ集めが実装されていて，世代別ごみ集めは実装されていない．そのために，ごみ集めの実行によって起こる，通常処理の停止時間が長いという欠点がある．この欠点は世代別ごみ集めを実装することによって解消することができる．しかし，GNU Emacs は世代別ごみ集めを実装することを考慮せずに実装されている．したがって，ソフトウェアによって世代間参照の検出を行う実装法は，ポインタの書き換えが起こりうるすべての場所に対してのコードの修正が必要となりソースコードの変更箇所が多いので，実装に困難がともなう．しかし，ダーティビット情報を用いた実装では，そのような必要がないので，実装は容易となる．したがって，この面においてもダーティビット情報を用いた実装が妥当であるといえる．

本論文では，ダーティビット情報を用いた世代別ごみ集めを GNU Emacs に実装し，現実的な使用状況においての性能を計測した．それによって，浅い束縛を用いた実装においても世代別ごみ集めの実装によりごみ集めの性能を改善することができることを示す．

本論文の構成は次のとおりである．2 章において，世代別ごみ集めの実装に関連のある GNU Emacs と Emacs lisp の特徴についてまとめ，それらが世代別ごみ集めの実装に対してどのような影響を与えるかを述べる．3 章において，世代別ごみ集めの実装がどのように行われたかについて述べる．4 章において，世代別ごみ集めを実装した GNU Emacs と世代別ごみ集めを実装していない従来の GNU Emacs の性能の比較を行う．5 章において，本論文に関連する研究について述べる．6 章では，本論文で得られた結果についての結論をまとめる．

2. GNU Emacs と Emacs Lisp の特徴

世代別ごみ集めの実装に関連のある GNU Emacs と Emacs Lisp の特徴を，言語仕様上の特徴と実装上の特徴に分けて説明する．また，従来の GNU Emacs に実装されているごみ集めの特徴についてもまとめる．

Emacs Lisp の言語仕様の特徴は次のようなものである．

- 多くのデータ型を持ち，その中にはベクターや文字列のような可変長のデータ型も含まれる．
- 動的スコープでクロージャーは存在しない．
- インタプリタとバイトコードインタプリタで動作する．

Emacs Lisp の実装の特徴として次のようなものがある．

- Lisp のインタプリタは直接 C でコーディングされている．
- ソースコードが約 10 万行あり巨大である．したがって，ソフトウェアによる世代間参照の検出を実装するのは困難がともなう．
- 変数の実装に浅い束縛[2)]を用いている．したがって，オブジェクトの書き換えが頻繁に起こりうる．

さらに，従来の GNU Emacs に実装されているごみ集めの方式について述べる．

- 大部分のオブジェクトはマークスイープ方式のごみ集めでごみ集めされるが，ストリング型だけはコンパクションを行っている．
- マークスイープ方式を前提としたインタプリタの実装が見られる．たとえば，ストリングはコンパクションが行われるので，ストリングのオブジェクトが代入される可能性がある変数はオブジェクトのアドレスの変化に対応している．しかし，ストリングが代入されることがない変数はオブジェクトのアドレスの変化を考慮してない実装も見受けられる．
- ごみ集めは次のどちらかの条件に当てはまるときに開始される．
  - eval や funcall の中にあるチェックポイントにおいて，前回のごみ集めの終了時から新たに確保したメモリのバイト数が，ユーザが定義できる `gc-cons-threshold` という変数の値を超えていたとき．
  - ユーザからのキー入力が一定時間以上行われなかったとき．

  したがって，これらの条件から分かるように，世代別ごみ集めを導入した場合にごみ集めの開始時に若い世代に確保されているオブジェクトの量の上限を定めることができない．

なお，本論文で用いたのは GNU Emacs バージョン 19.28 ベースの mule2.3 である．

3. 世代別ごみ集めの実装

1 章で述べたとおりに，ダーティビット情報を用いた世代別ごみ集めを GNU Emacs に実装した．

現在の GNU Emacs の実装から考えて，世代別ごみ集めを実装する場合に次のような制約を設けるのが現実的であると判断した．

- 新たなオブジェクトを確保する領域である，若い世代の大きさは可変となるようにする．若い世代の大きさを可変とすることで，GNU Emacsがごみ集めを実行するタイミングを世代別ごみ集めを実装した後も変更する必要がなくなるので，ソースの変更箇所が少なくなるという利点がある．また，アプリケーションの性質などにより若い世代の大きさを変化させることで，性能の向上がはかれる可能性があるという利点がある．
- 世代別ごみ集めを実装するときに必要な，古い世代から若い世代を指すポインタを検出する方法として，ダーティビット情報を用いることにする．ダーティビット情報を用いることの利点として，ソースコードの変更箇所を少なくすることができる点があげられる．

  ソフトウェアを用いた実装では，ポインタの書き換えが起こる場所でライトバリアのためのコードを挿入する必要がある．そのような変更箇所は，`rplaca`，`rplacd`のようにユーザがオブジェクトを明示的に書き換えるような命令のほかに，ウィンドウの表示を制御するためのコードのように一般ユーザからはオブジェクトの書き換えが分からない部分にも多数存在する．それらをすべて書き換えるためにはすべてのソースコードを調べる必要があるので大変な作業量となる．さらにコードの修正箇所の見落としのためにバグが発生する可能性もある．

  これに対し，ダーティビット情報を用いた実装ではそのような変更作業が必要ないので，大きな利点となる．また，GNU Emacsは浅い束縛で実装されていてオブジェクトの書き換えが多くなるであろうと予測されるので，ソフトウェアによってオブジェクトの書き換えを検出するのに比べ有利である．

世代別ごみ集めを実装するうえで必要な設計は次のように行われ，それに基づいて実装された．

- 若い世代と古い世代の2世代に分け，それぞれの領域を1つずつ用意することにした．
- 若い世代のごみ集めを1回生き残ったオブジェクトはすべて古い世代へと移動する．この結果，若い世代のごみ集めの直後には若い世代には生きているオブジェクトが存在しない．
- ごみ集めの開始のタイミングは，従来のGNU Emacsが備えていた機能を利用して，次のようにした．
  - evalやfuncallのチェックポイントで`gc-cons-threshold`以上のメモリを確保していたときには，若い世代のごみ集めを行うようにした．
  - ユーザからのキー入力が一定時間以上ないときに，古い世代のごみ集めを行うことにした．こうすることにより，古い世代のごみ集めによって起こる長時間の通常処理の停止に，ユーザが気づきにくくなる[10]．

    また，古い世代のごみ集めを行う前には，必ず若い世代のごみ集めを行うことにした．若い世代のごみ集めの直後には若い世代にはオブジェクトが存在しないので，このようにすることで，若い世代から古い世代を指すポインタを探す必要がなくなる．
- 古い世代では，シンボル，コンス，ストリングなどのオブジェクトについては，ブロックを確保し，1つのブロックの中には同じ種類のオブジェクトだけになるように配置する．ブロックの大きさは一般的なマシンでのページサイズである4Kバイトとし，ブロックの開始アドレスは4Kバイトの倍数のアドレスになっている．この方式は，ブロックサイズが異なること以外はもともとGNU Emacsに実装されていたごみ集めと基本的には同じ方式である．

3.1 コピー方式のごみ集めへの変更

従来のEmacs Lispのごみ集めではマークスイープ法が主に採用されていたが，世代別ごみ集めを実装するときに，若い世代から古い世代へオブジェクトのコピーを行うようにした．

この変更によってごみ集めの実行前と実行後でオブジェクトのアドレスが変化するようになるので，Emacs Lispのごみ集めのルートを拡張することが必要となった．

- ごみ集めの実行時にスタック上に存在する変数でルートとなっていない変数をルートに追加する必要が出た．
- GNU Emacsでは，アドレスが移動しないオブジェクトへのポインタが入っている大域変数で，その変数が指しているオブジェクトが他のオブジェクトから指されていると分かっているものにもルートとなっていない場合があった．これらもルートに追加する必要があった．

これらの変更はソースコードで46カ所あった．

マークスイープ方式のごみ集めからコピー方式に変更した理由は，若い世代と古い世代を別の領域とすることによりメモリ確保の局所性が向上することと，若

い世代のメモリの確保が古い世代の書き換えとして認識されないようにすることによりルート検査を行うページを少なくするためである．

3.2 ダーティビット情報を利用した世代間参照の検出

ダーティビットは OS が仮想記憶のスワップやページングを実現するために利用するものである．仮想記憶システムは，メモリ上に存在する各ページフレームにつきダーティビットと呼ばれる 1 ビットの情報を保持し，ダーティビットを維持するためのハードウェアを備えている．ダーティビットが立っていると，あるページへの書き込みがあったことを示している．そして，そのダーティビットの情報を用いて，あるページをメモリから追い出すときにディスクへの書き出しが必要であるかの決定を行う．

ダーティビット情報を用いて世代間参照を検出する原理は次のようなものである．ある古い世代のオブジェクトに対し若い世代を指すポインタが書き込まれたとすると，そのオブジェクトを含むページに書き込みが行われたことになり，そのページのダーティビットが立つ．したがって，ダーティビットがクリアされない限り，若い世代への参照が作成されたページのダーティビットは必ず立っているので，ダーティビットが立っているページのオブジェクトをすべて検査すれば，若い世代への参照の検出が行える．

これを用いて世代間参照の検出を行うには，次のようなアルゴリズムを用いればよい．

(1) 若い世代のごみ集めを行う場合に，前回の若い世代のごみ集めが完了した時点から今回のごみ集めを開始する時点までの古い世代のダーティビット情報を用意する．
(2) ダーティビットが立っているページを走査し，若い世代への参照を探す．見つかった参照もルートと見なしてごみ集めを行う．
(3) ごみ集めが終ったら，ダーティビット情報をクリアする．

しかし，従来の OS では，ダーティビット情報を利用するためには次のような問題が存在した．

- ハードウェアが提供するダーティビット自身は仮想記憶システムが独占的に利用するので，ユーザがアプリケーションを実装するのに都合の良いタイミングでダーティビット情報を利用することができない．
- そもそも，ダーティビット情報をユーザが利用するためのインタフェースを OS が備えていないので，利用するためには OS の改造が必要である．

それに対し，Solaris2 では次のような方法でダーティビット情報をユーザに対して提供している．

- /proc ファイルシステムというプロセスの情報を持つファイルシステムを OS が提供している．そのファイルシステムのページデータファイルという特定のファイルを開き，そのファイルを読むことによって，そのプロセスが利用するすべてのページに対するダーティビット情報をユーザが取得することができる☆．
- あるページデータファイルを前回読んだ時点から今回読んだ時点までのダーティビット情報が得られる．それに加え，ページの読み出しの有無の情報も得られる．つまり，ダーティビット情報は仮想記憶システムの都合とは無関係に利用することが可能である．
- OS による制限を超えない限り，1 つのプロセスに対し任意個のページデータファイルを開くことが可能である．また，それぞれのファイルは独立したダーティビット情報を提供し，あるファイルを読んでもそれは他のファイルのダーティビット情報に影響しない．

Solaris2 が提供するダーティビット情報を利用することにより，世代別ごみ集めの世代間参照を検出することができる．従来，一般的なマシンでハードウェアを利用した世代間参照の検出を行う場合は Page Marking [4] という方法が用いられていた．Page Marking の実装には，メモリをライトプロテクトし，ライトフォールトのシグナルをハンドルするという方法を用いる．このように，メモリをライトプロテクトする方法では OS の提供するシグナル機構を使用するため，ライトバリアにかなりのオーバヘッドが存在する．しかし，ダーティビット情報を利用するとシグナルハンドラの処理にかかっていた時間が節約できる．

しかし，Solaris2 での実装の場合，必要な情報，つまり古い世代の領域のダーティビット情報だけを読み出すことができず，そのプロセスが利用するすべてのページに対するダーティビット情報を読み出してしまうので，ダーティビット読み出し時の速度の低下を招いている．これについては，4.3 節で議論する．

4. 世代別ごみ集めの性能計測

ダーティビット情報を用いた世代別ごみ集めの実装を GNU Emacs に対して行ったのは，GNU Emacs

☆ solaris2.7 の場合．solaris2.6 以前のバージョンでは手順が多少異なる．

が多くのユーザに利用されているので，実際の利用状況に基づいて性能の計測を行うことができるからである．ユーザが GNU Emacs を利用する場合は対話的に利用することが多く，そのときに問題となるのはごみ集めによって起こる通常処理の停止時間であるので，停止時間の短縮の程度によって性能を評価する．

GNU Emacs のごみ集めによる処理の停止時間を計測するために，ユーザが対話的に GNU Emacs を利用している状況になるべく近くなるような方法を採用することにした．

(1) ユーザのキー操作を記録するプログラム（「打鍵記録プログラム」）を作成し，打鍵記録プログラムで GNU Emacs 利用時のユーザのキー操作を打鍵データとして記録する．このとき，ユーザのキー操作のタイミングの記録をするために，打鍵データにタイムスタンプも付けておく．

(2) 記録した打鍵データをタイミングも正しく再生するプログラム（「打鍵再生プログラム」）を作成し，打鍵再生プログラムで打鍵データを再生することによって GNU Emacs を操作し，そのときの性能を計測する．

この方法を用いることによって，キーボードからの入力によりアプリケーションが動作するという GNU Emacs の特徴をほとんど損なうことなく，同じ状況での性能の計測を何度も行うことができる．また，実際にユーザが利用している状況を記録して行っているので，現実の利用状況に即した性能の計測ができる．

以下で行った計測は SPARC Station ELC（OS Solaris2.5）を用いて行った．

実験のために，従来の GNU Emacs においてあるユーザの利用状況を記録した．そのユーザが利用したのは次のようなアプリケーションであった．

- C 言語のプログラムのエディット
- Info の利用
- Makefile の作成
- Gnus によるニュースの読み書き

このとき，打鍵数は約 2600 で，作業時間は 30 分であった．このときにごみ集めは 40 回起こった．

この打鍵記録をベンチマークプログラムとして用いて，世代別ごみ集めの実装された GNU Emacs と世代別ごみ集めの実装されていない従来の GNU Emacs の性能の比較を行った．性能の比較を行うときに 2 つの GNU Emacs の `gc-cons-threshold` の値を 100000 バイトに設定して行った．つまり 100000 バイトメモリをアロケートしたらごみ集めを開始することにした．

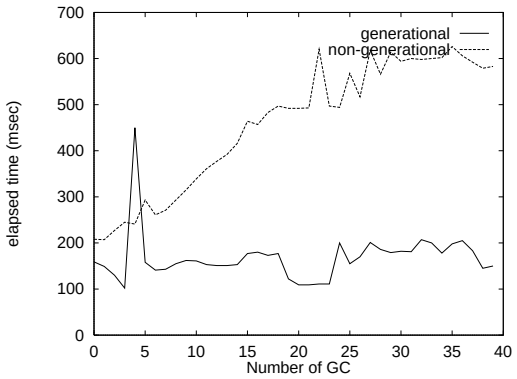

図 1 ごみ集めの停止時間の比較
Fig. 1 Comparison of pause time due to garbage collection.

4.1 停止時間の比較

図 1 は世代別ごみ集めの GNU Emacs と従来の GNU Emacs においてのごみ集めの停止時間の比較である．横軸は GNU Emacs 起動時からのごみ集めの通算の回数を表し，縦軸は各回のごみ集めにかかった時間を表す．2 つの GNU Emacs では，まったく同じ作業をし，`gc-cons-threshold` の値が同じなので，同じタイミングでごみ集めが起こっている．

このグラフにおいて，世代別ごみ集めの GNU Emacs で 4 回目のごみ集めの時間が長くなっているのは，このときにキーボードからの入力が一定時間以上なかったので古い世代のごみ集めが行われたからである．古い世代のごみ集めは，若い世代のごみ集めを行った後に，従来の GNU Emacs のごみ集めを行うのと同等のことを行っている．したがって，ごみ集めの処理時間も従来の GNU Emacs のごみ集めの時間に若い世代のごみ集めにかかる時間を足したものになっている．

2 本のグラフを比べれば分かるように，従来の GNU Emacs では起動時からの通算のごみ集めの回数が増えるにつれてごみ集めにかかる時間が増加しているが，世代別ごみ集めの GNU Emacs では，起動時からのごみ集めの通算回数が増えてもごみ集めにかかる時間はほぼ一定のままで，古い世代のごみ集めを行わない限り 200 msec を超えることはない．

4.2 使用メモリ量の比較

図 2 は使用メモリ量を比較したものであり，縦軸は Lisp オブジェクトが使用するメモリ量を表す．

図 1 と図 2 を比較すると，世代別ごみ集めの GNU Emacs では，使用メモリ量が増加してもごみ集めにかかる時間の増加がほとんどないことが分かる．

世代別ごみ集めの GNU Emacs と従来の GNU

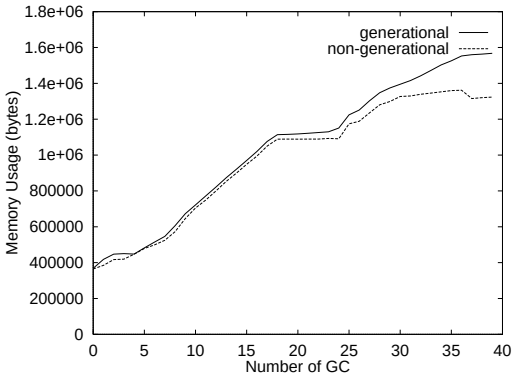

図 2 使用メモリ量の比較
Fig. 2 Comparison of memory usage.

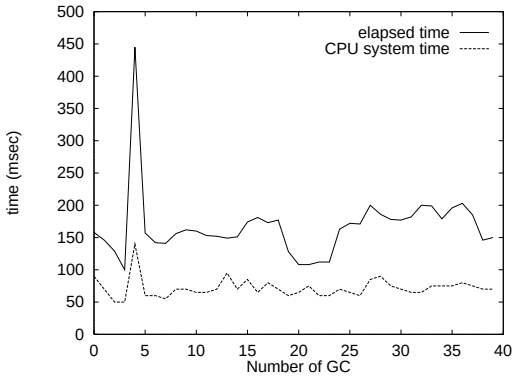

図 3 世代別ごみ集めの停止時間
Fig. 3 Pause time of generational garbage collection.

Emacs のメモリ使用量の差は，世代別ごみ集めの GNU Emacs で古い世代にたまったごみの量を表している．図 2 から古い世代のごみは，最大で生きているオブジェクトの約 20%程度であると分かる．したがって，ユーザからの入力がないときにだけ古い世代のごみ集めを実行するようにしても，ごみが古い世代に大量にたまることはないといえる．

さらに，オブジェクトの寿命を計測した結果[6]によると，25 回目のごみ集め以降で古い世代にごみが増えているのは，比較的寿命の長いオブジェクトがアプリケーションの終了とともに死亡してごみになったからである．

4.3 世代別ごみ集めの停止時間

図 3 は世代別ごみ集めを実装した GNU Emacs でごみ集めを実行するときの，CPU のシステム時間とユーザ時間を調べたものである．

計測によると，従来の GNU Emacs では CPU システム時間はほとんどかからなかったので，世代別ごみ集めの GNU Emacs で使われている CPU システム時間は，ダーティビット情報を取得するのにかかる時間であると考えられる．したがって，ごみ集めにかかる時間の半分程度がダーティビット情報の取得に使われていることが分かる．

Solaris2 が提供する機能の問題点として，ダーティビット情報の必要なページの範囲を指定できず，共有ライブラリなども含んだ，そのプロセスの利用する全ページについての情報が提供される点をあげることができる．実際にダーティビットの情報が必要なのは，古い世代のページのうちで若い世代へのポインタが存在する可能性があるページだけである．したがって，古い世代のページでも若い世代へのポインタが存在する可能性がないページや，若い世代のページやプログラムのテキスト領域のページや共有ライブラリのページなどのダーティビット情報は必要ない．

ダーティビット情報が必要なページを，多めに見積もってヒープ領域だけだと仮定してダーティビットの取得にかかる時間を試算してみる．たとえば，GNU Emacs を立ち上げた直後に使用しているメモリサイズは約 5700 K バイトでそのうちダーティビットの情報が必要なヒープ領域は 156 K バイトである．この計測の場合ではヒープの容量が最大になった場合でも約 1700 K バイトなので必要な部分だけのダーティビットの情報が得られるように実装されていれば，ダーティビットの読み出しにかかる時間は最低でも現在の約 25%程度になると推測される．実際に必要な領域は古い世代でポインタを含む領域だけであり，多くても約 1600 K バイト程度となるので，さらに高速化が可能である．

もう 1 つの Solaris の実装の問題点として，ダーティビットをクリアするという命令がない点があげられる．若い世代のごみ集めの実行中には，生存するオブジェクトを古い世代へ移動するために古い世代への書き換えが起こる．また，古い世代のごみ集めの実行時にはマークビットやフリーリストの操作のために頻繁に古い世代の書き換えが発生する．これらの書き換えは，世代間参照を検出するうえで関係のない書き換えである．したがって，ごみ集めの終了時にダーティビット情報のクリアを行う方がよい．

しかし，現在の Solaris の実装ではダーティビットをクリアするという命令がなく，ダーティビットをクリアするためには，必要のないダーティビット情報を取得する必要がある．現在の世代別ごみ集めの実装では，ダーティビット情報をクリアするために若い世代のごみ集めの終了時にダーティビット情報を取得する，ということは行っていない．なぜなら，ダーティビットをクリアすることによってルート検査のための時間

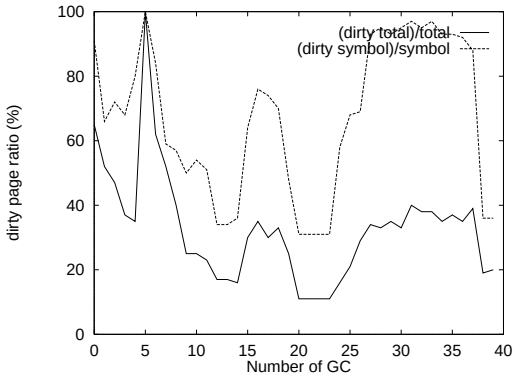

図 4 古い世代で書き換えがあったページの割合
Fig. 4 Dirty page ratio of old generation.

が短縮されるという効果よりも，若い世代のごみ集めの終了時にダーティビット情報を取得することによるオーバヘッドの方が大きいからである．したがって，ダーティビット情報のクリアは，その効果の期待できる古い世代のごみ集めの終了時にしか行っていない．しかし，高速にダーティビット情報をクリアできる命令を OS が備えていれば，毎回のごみ集めの終了時にダーティビット情報をクリアすることが可能である．ダーティビット情報のクリアのための命令はダーティビット情報を取得する命令よりも高速に動作すると期待されるので，そのような命令を提供すべきであると考える．

4.4 書き換えページ率

図 4 は古い世代に存在する全オブジェクトページに対する書き換えのあったページの割合を示す．縦軸は書き換えのあった割合である．total のグラフはオブジェクトの中身としてポインタを持つことができるオブジェクト全種類での書き換えがあった割合で，symbol のグラフは Lisp のシンボルを持っているページで書き換えのあった割合である．

図 4 を見るとシンボルの書き換えが他のオブジェクトに比べてかなり多いことが分かる．これは GNU Emacs が動的スコープを持ち，さらにそれを実装するときに浅い束縛という方法を用いているので，シンボルの値の書き換えが多くなるためと推測される．

5. 関 連 研 究

本論文の実装では，仮想記憶のハードウェアが管理するページのダーティビット情報を，OS が提供する機能を用いて利用した．仮想記憶のハードウェアを用いる方法としては，Shaw の文献 8) や，Boehm らによる文献 4) や，Appel らによる文献 1) などがある．

Shaw は OS のカーネルを変更し，新しいシステムコールを OS が提供することによりダーティビット情報を用いた世代別ごみ集めを実装できることを示した．そのために OS はダーティビットをクリアするシステムコールと，ダーティビットを返すシステムコールの 2 つのシステムコールを新たに用意している．また，この機能を実現するために OS の内部には 3 種類のダーティビットを持っている．

Boehm らは仮想記憶のハードウェアからダーティビット情報を得る方法として，ヒープ全体を書き込み禁止にし，書き込み例外のシグナルをハンドルするという方法を利用した．この方法を用いて，マークスイープ方式の並列ごみ集めを実装するためのライトバリアを実装した．この方式はシグナルをハンドルするので，Shaw の方式に比べるとオーバヘッドが存在するが，多くの OS に存在する機能だけで実装できるという利点がある．

Appel らは仮想記憶のハードウェアを用い並列コピーごみ集めを実装した．Appel らの方法は，Baker が文献 3) で示したオブジェクトへのアクセスを検出するためにリードバリアを用いて並列ごみ集めを実装するという手法を，メモリページ単位のリードバリアを用いることにより実現している．

6. 結 論

本論文では，GNU Emacs に世代別ごみ集めを実装する場合の方法および結果を示した．計測から，世代別ごみ集めにすることで使用メモリ量にかかわらずほぼ一定の中断時間になることが確認できた．また，マークスイープ法に比べ，全体のごみ集めによる停止時間が減少しているのが観測され，そのときのメモリ使用量の増加も 20%以下に抑えられることが分かった．

今後の予定として，ライトバリアとしてメモリプロテクトの方法を用いた実装との性能の比較，`gc-cons-threshold` の値を変更したときの性能の比較などを行いたいと思う．さらに，古い世代のごみ集めを開始するタイミングについても検討を行いたいと思う．

参 考 文 献

1) Appel, A.W., Ellis, J.R. and Li, K.: Real-Time Concurrent Collection on Stock Multiprocessors, *ACM SIGPLAN Notices*, Vol.23, No.7, pp.11–20 (1988).
2) Baker, H.G.: Shallow Binding in Lisp 1.5, *Communications of the ACM*, Vol.21, No.7, pp.565–569 (1978).
3) Baker, Jr., H.G.: List Processing in Real Time

on a Serial Computer, *Communications of the ACM*, Vol.21, No.4, pp.280–294 (1978).
4) Boehm, H.-J., Demers, A.J. and Shenker, S.: Mostly Parallel Garbage Collection, *ACM SIGPLAN Notices*, Vol.26, No.6, pp.157–164 (1991).
5) Jones, R.E.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
6) 小林広和，寺田　実：Emacs のごみ集め改良の試み，日本ソフトウェア科学会第 13 回大会論文集，pp.165–168 (1996).
7) Lieberman, H. and Hewitt, C.E.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM*, Vol.26, No.6, pp.419–429 (1983).
8) Shaw, R.A.: *Empirical Analysis of a Lisp System*, Ph.D. Thesis, Stanford University, Palo Alto, California (1988).
9) Ungar, D.M.: Generation Scavenging: A Non-disruptive High-Performance Storage Reclamation Algorithm, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM Press, pp.157–167 (1984).
10) Wilson, P.R.: Opportunistic Garbage Collection, *ACM SIGPLAN Notices*, Vol.23, No.12, pp.98–102 (1988).
11) Wilson, P.R.: Uniprocessor Garbage Collection Techniques, Technical Report, University of Texas (1994).

(平成 11 年 7 月 19 日受付)
(平成 12 年 5 月 11 日採録)

小林　広和（学生会員）

1972 年生．1995 年東京大学工学部機械情報工学科卒業．1997 年同大学院工学系研究科情報工学専攻修士課程修了．現在同大学院同研究科同専攻博士課程在学中．ごみ集め，記号処理言語の研究に従事．

寺田　　実（正会員）

1959 年生．1981 年東京大学工学部計数工学科卒業．1983 年同大学院工学系研究科情報工学修士課程修了．同大学計数工学科助手，電気通信大学電子情報学科助手を経て，1991 年東京大学工学部機械情報工学科講師，1992 年同助教授．工学博士．ソフトウェア科学会，ACM 各会員．