プログラミング 14−20
（1997. 8. 22）

圧縮型並列ガーベッジコレクション

寺島 元章　　稲村 善平　　古賀 一生

電気通信大学大学院　情報システム学研究科

概要

本稿では CSC と呼ぶ圧縮型並列ガーベッジコレクションの設計方針と実装法および評価について述べる. CSC は Lisp 処理系である PHL を対象に設計され, オブジェクト参照での負荷が大きいリードバリアをなくし, 最小限の排他制御で, 純計算側の並列化による負担を軽減している. GC 側は, 複製作成とそれに伴うポインタ補正の反復作業による顕著な負荷増がある反面で, 圧縮法の副次的効果である世代分化による利得が並列化で増幅されることが実験結果から示されている.

Concurrent Slide-Compacting Garbage Collection

Motoaki Terashima　　Zempei Inamura
Kazuo Koga

Graduate School of Information Systems
University of Electro–Communications

1–5–1 Chofugaoka, Chofu-Shi, Tokyo 182 Japan

Abstract

The design, implementation and analysis of CSC (Concurrent Slide-Compacting Garbage Collection) that is designed as GC of PHL, a dialect of Lisp. CSC reduces List processor's overload resulting from concurrency by using no read-barrier and minimum mutual exclusion, because the read-barrier causes heavy load at data objects reference. GC processor has a time-consuming task of duplicating object(s) and adjusting pointers to them, while a gain of the generation achieved as a side effect of sliding compaction scheme increases by concurrency, that are shown by our experimental results.

1 序

ガーベッジコレクション (ごみ集めのこと, 以下単に GC と呼ぶ) とプログラムを実行するプロセス (純計算と呼ぶ) とを平行して実行できればGC の起動による純計算の停止が回避できることから, 動的データを扱う制御システムや組込み型計算機上の言語処理系の実装を目的に並列型の GC の研究開発が行われており, その成果も数多く得られている.

本稿では, 停止回収型の一種である圧縮方式 GC の並列化について, その設計方針と実装法および評価を述べる. 圧縮方式 GC は使用中オブジェクトを使用領域 (ヒープ) の一端にそれらの位置関係が変わらないように詰める.「圧縮」はこうした再配置の様子を的確に表している. オブジェクトの再配置においてその位置関係が保存されることを生成順序保存 [8] という. 圧縮型 GC が生成順序を保存することは, オブジェクトの作られた順序がヒープ上で恒久的に保存されることを意味する. そこで, オブジェクトが新たに作られる方向 (順方向) とは逆向きに圧縮が行われるならば, オブジェクトは順方向に古いものから新しいものへと並ぶ. 並列型圧縮方式 GC(Concurrent Sliding Compactor, 以下略して, CSC と呼ぶ) はこの特質を最大限利用している.

CSC の処理の流れは, その基になった圧縮方式 GC と同様, Marking(印付け), MakeOTree (ポインタ補正のための情報木の作成), Duplicating(複製の作成), Adjusting(ポインタ補正) と進行する. ただし, 後の二者は一括して処理されるのではなく, 前者の処理が細分化され, その各々にたいして後者が続行するという反復処理になる. Duplicating は, 使用済み, あるいはオブジェクトが再配置された後の空き領域に圧縮 (再配置) 対象のオブジェクトの複製を作り, Adjusting はそれらを指すポインタの補正を行う. 複製は圧縮されたオブジェクト自身であり, 元のオブジェクトはポインタ補正後に不要となる. この操作をヒープの GC 対象領域中の使用中オブジェクトすべてについて順方向に行う. 空き領域が圧縮対象オブジェクトの使用域よりも大ならば問題ないが, 小さい場合には, 全体の複製が作り終えるまでこの両者が繰り返されることになる.

Duplicating と Adjusting の反復処理は, 停止回収型が行う一括処理に比べて, Adjusting フェーズの反復に係わる分の負担増となる. しかし, 純計算側にデータの読み出し時に常にポインタを検査するというリードバリアをなくすことができる. 書込みに対して元のオブジェクトとその複製の一貫性を保つことで, その複製に対する Adjusting フェーズが終了するまではいずれのオブジェクトを参照してもよいからである. 終了時点で, 元のオブジェクトを指していたすべてのポインタは複製を指すように変わる. 一方, オブジェクトの内容を変更する書込み操作にはオーバーヘッドが生じるが, それよりも使用頻度が格段に多いとされるオブジェクトの読み出し操作に (リードバリアという) オーバーヘッドがないことは処理系全体の効率の点できわめて有利である. 結果的に, CSC は GC と純計算の負担バランスを純計算側が軽くなるように選んだことになる.

オブジェクトの生成ではその (新しい) 内容が, 書込みでは新しいオブジェクトが必要に応じて Rbox と呼ばれる記憶場所に登録される. この情報は印付けやポインタ補正で利用される. この種のアイデアは既製の並列型 GC に具現されており, 特に新規性のあることではないが, 複数の MPU をもつワークステーションなどの汎用計算機での実装と実用に耐えられるような洗練化が図られている.

さらに, CSC の設計ではオブジェクトの世代 (generation) 別管理 [5] の概念が取り入れられている. 圧縮方式自体が「古いオブジェクトはヒープの底に溜まる」という自然な世代分化を実現しているが, CSC はこれをさらに進めて, その GC サイクル内でこれをミクロ的に実現する. CSC でのオブジェクト移動の様子は「砂時計」にたとえられる. そこでは, 上方に溜まった新しいオブジェクトが時間経過とともに隘路を通り, 生き残ったオブジェクトがその底に積もっていくのである.

2 背景

GC は使用中と使用済みのオブジェクトを類別し, 後者に割り当てられた記憶領域の回収 (再利用可能にすること) を自動的に行う処理系に組み込まれた記憶管理機構である. Algol や Lisp などの処理系とともに登場した初期の GC は, それが停止回収型 GC と呼ばれるように, その他の処理 (純計算) を中断してヒープ全体に対して一括回収処理を行うものであった. 停止回収型 GC はその処理方式から, (1) 印付け回収方式 (2) 複写方式 (3) 参照計数器方式 に大別される. さらに, (1) の印付け回収方式は, (1a) 自由リスト方式 (1b) 移動方式 (1c) 圧縮方式 に細分される. GC 処理はこの順に複雑になる.

自由リスト方式はオブジェクトの再配置を行わない. 移動方式と圧縮方式はともに使用中オブジェクトの再配置を行い, まとまった回収領域をヒープの他方に作る. ただし, 後者では生成順序が保存されたまま再配置が行われる. Lisp の CONS データやシンボル, ベクトルなどの異種サイズのオブジェクトが混在する場合の GC は, その回収領域が効果的に 1 つに集約される点で圧縮方式と複写方式が有利である. 近頃, 処理時間が使用中のオブジェクトの総容量に比例するような圧縮方式の GC が考案され, それを用いればヒープの総容量が同じならば両者の総処理時間に大差はないという報告 [7] もある.

GC と純計算の 2 つのプロセスが平行して実行される環境では, 純計算側が行うオブジェクトの生成や書込みに対して GC 側がその補足に失敗するとそれで生れた実体が「宙に浮く」という致命的な事態になりかねない. そこで, 2 つのプロセスが各オブジェクトに付けられた付加的な情報を共有しながら協調動作をすることになる. これが並列型 GC の原理原則である. また, 既製の並列型 GC は停止回収型 GC のいずれかの方式に基づいている.

Dijkstra らの提案になる On-the-fly GC[3] は自由リスト方式に基づいたものとして有名である. 各オブジェクトには, 印と無印を表す「白」と「黒」の他に,「灰色」という第 3 の色が存在し, これが生成と書込みの補足を

可能する. 純計算 (mutator) 側は生れた実体 (新オブジェクト) が白ならば灰色に変えるが, この種の操作は当然, 根 (root) となるスタックやレジスタの書込み (代入) 時にも必要となる.

Yuasa によるスナップショット GC[10] は同じく自由リスト方式に基づいたものである. On-the-fly GC との違いは, オブジェクトの書込み時に (リンクの切れた) 旧オブジェクトを白から灰に変えることである. この効果として, 根は参照だけになり, スタックやレジスタの書込み操作のオーバーヘッドを解消する. その反面, GC 中に生じた「ごみ」は回収されずに, 次の GC まで残ることになる.

Steele の並列型 GC[6] は移動方式に基づいた斬新的なものである. 前者の 2 例とは異なり, 使用中オブジェクトの移動とそれに伴うポインタ補正が必要なため, GC 処理もかなり複雑である. GC の各フェーズと純計算のプリミティブ間で必要な排他制御はセマフォを利用して行われる.

Baker の実時間型 GC[1] は複写方式に基づいたものである. 複写操作そのものは副作用がないため, 並列処理に適すると言われてきた [4]. Baker の GC では, GC(複写) 処理が時系列に対して細分化されていて, それらの間隙を縫って純計算が行われるため, 実時間型と呼ばれている. 実時間型では 2 つのプロセスの排他制御の問題は生じないし, 複写方式特有のヒープの半分を自由領域に残して GC が起動される場合には, その起動中に領域不足で純計算が停止する状態 (starvation) も非常に低いとされる. Baker の GC の最大の問題点は, 行き先ポインタ (forward pointer) の確認作業がリードバリアを伴うことである.

参照計数器方式は, 特殊なハードウエアを用いない限り, 計数のオーバーヘッドが非常に大きく, しかも, 循環リストへの対処が最大課題として残されているなどの点で, 現状では効果的な実装は困難である.

3 CGC

3.1 PHL

CGC の細部は Lisp 処理系の PHL[9] を対象に設計されている. PHL は, Common LISP 準拠の仕様をもちコンパイラ指向と汎用性を備えた処理系である. 異種データオブジェクトは単一のヒープ上に作られる. 図 1 は PHL のデータ表現を示したものである. 各オブジェクトは,「フィールド」と呼ばれる対象計算機の一語を単位として構成される. 一語が 32 bits ならば, 各オブジェクトは 512 MB (29 bits) のアドレス空間に配置可能となる. フィールドの先頭の 2 ~ 3 bits は 6 つの基本型を識別するためのポインタタグである. バイトアドレス方式では使用されることのない下位の 2 bits は排他制御のために確保されている. なお, 印付け用ビット (mark bit) はデータ表現内になく, 印付け情報のための表 (Mark Bit Table, 略して MBT と呼ぶ) がヒープと別の領域に確保される. これにより, 各オブジェクトの参照は印の有無に関係なく (高速に) 行うことができる.

Data types	Field	
	tag	address/data
CONS data	000	.... xy
Symbols	001	.... xy
Blocks	010	.... xy
Short reals	011	.... xy
Short integers	10	.... xy
Characer codes	11	.... xy

Note: 'x' and 'y' denotes the mutual exclusion bit(s).

図 1 : PHL のデータ表現

ヒープ中のオブジェクトが「使用中」か「使用済みのごみ」であるかは, 根からの参照の可否による. PHL の根は次の 3 つである.

- Oblis (シンボル表)
  シンボルデータを成分とする一次元配列である. その序列はヒープ上の実体 (シンボルの本体) の出現順序と同じである. 印字名を鍵とするハッシュ表のエントリがこの配列の指標となる.
- Vstack (スタック)
  純計算側が値保持のために使用するスタック. 値渡しや束縛がこのスタックで行われる.
- Rbox (未決入れ)
  CONS, シンボル, ブロックの各オブジェクトを成分とする一次元配列である. 印付けやポインタ補正が未処理のオブジェクトが一時的に置かれる. これは, 世代別 GC で利用されている remembered set の機能に似た効果をもつ.

純計算側で GC と協調処理を行う必要のあるプリミティブは, オブジェクトの生成を行う CONS とオブジェクトの書込みを行う SET に代表される. 両者ともフィールドへの書込み操作であるが, CONS は対象フィールドが gc 対象領域外であるので, SET よりも処理は比較的単純である.

CGC はいつでも起動可能である. CGC が起動されたとき, CONS が新オブジェクトを生成する位置から逆方向に, 前回の CGC で圧縮されたオブジェクトの境界までが gcArea と呼ぶ GC 対象領域になる. 以下, CONS と SET の処理も含めて, CGC の各フェーズについて解説する. 図 2 はその間のヒープの様子を示したものである.

3.2 Marking

Marking は, Oblis, Vstack, Rbox の順にそれらの要素を走査し, それらのオブジェクトが gcArea 内にあればそれを印付けする. ポインタつながりのオブジェクトはそのポインタがたどられ, 参照できるすべてのものに印が付けられる. もし, 印付けしたオブジェクトのフィールドが (SET による書込み作業中で) lock されていれば, lock が解除されるまでその読み出しは待たされる.

CONS は gcArea 内のオブジェクトを成分に作るとき, そのフィールドを Rbox に登録する. CONS と, Marking のプリミティブ間の排他制御は不要である.

SET は, 書込みフィールドが gcArea 内であれば, 先ずそのフィールドを lock する. 次にそのフィールドに (marking の) 印が付いていれば, フィールドに書込んだ後に lock を解除し, そのフィールドを Rbox に登録する. 印がなければ, フィールドに書込んだ後に lock を解除するだけである. もし, 書込みフィールドが gcArea 外ならば, 書込んだオブジェクトが gcArea 内にある場合にのみ, そのフィールドを Rbox に登録する.

SET は, On-the-fly GC の「灰色化」に相当する処理を後で行えるように該当フィールドを Rbox という場所に一時預けるのである. これらのフィールドのオブジェクトは後で印付けされる. 排他制御はオブジェクトを確実に書込むためのものであり, これにより, 書込まれたオブジェクトは必ず印付される.

3.3 MakeOtree

MakeOTree は, gcArea 中の各クラスタ (使用中フィールドの連続した塊) の補正値を求めるための補正木 (Offset tree, 略して O-tree と呼ぶ) 作成する. O-tree はリストであり, 各節点 (node) は各クラスタの先頭に使用済みフィールドを用いて作られる (図 2 (c) 参照). 補正値は該当クラスタが再配置 (圧縮) で移動する量であり, 各節点の情報となる. それは O-tree に対する線形探索から求まる [8]. 並列型 GC の性格からその処理時間は大きな評価基準とはならない.

O-tree は gcArea を一回順方向に走査することで作成できる. O-tree が完成すると, gcArea 中の使用中フィールドの走査はその総数に比例する時間で出来ることになり, MBT は不要となる. MakeOTree が終了すると, 錨 (anchor) と呼ばれる特別なクラスタの存在の可否が確認できる.「錨」は gcArea の先端に位置し, 圧縮で動かないクラスタである. 当然, これを指すポインタは補正不要である. gcRelocArea は, gcArea からこのクラスタ部分を除いた領域のことである. gcRelocArea 中のオブジェクトは圧縮対象となる.

CONS は gcArea 内のオブジェクトを成分に作るとき, そのフィールドを Rbox に登録する.

SET は, 書込みフィールドが gcArea 外で, かつ, 書込んだオブジェクトが gcArea 内にあれば, そのフィールドを Rbox に登録する.

CONS や SET と, MakeOTree のプリミティブ間の排他制御は不要である.

3.4 Duplicating

Duplicating は圧縮 (再配置) 対象のオブジェクトの複製を作る. 対象オブジェクトが存在する領域を gcDupArea と呼ぶ. 複製作業は, その後方にある空き領域にフィールド単位で行われるが, そのフィールドが (SET による書込み作業中で) lock されていれば, lock が解除されるまでその複製作業は待たされる.

CONS は gcRelocArea 内のオブジェクトを成分に作るとき, そのフィールドを Rbox に登録する. これは後続の補正処理のためである. CONS と, Duplicating のプリミティブ間の排他制御は不要である.

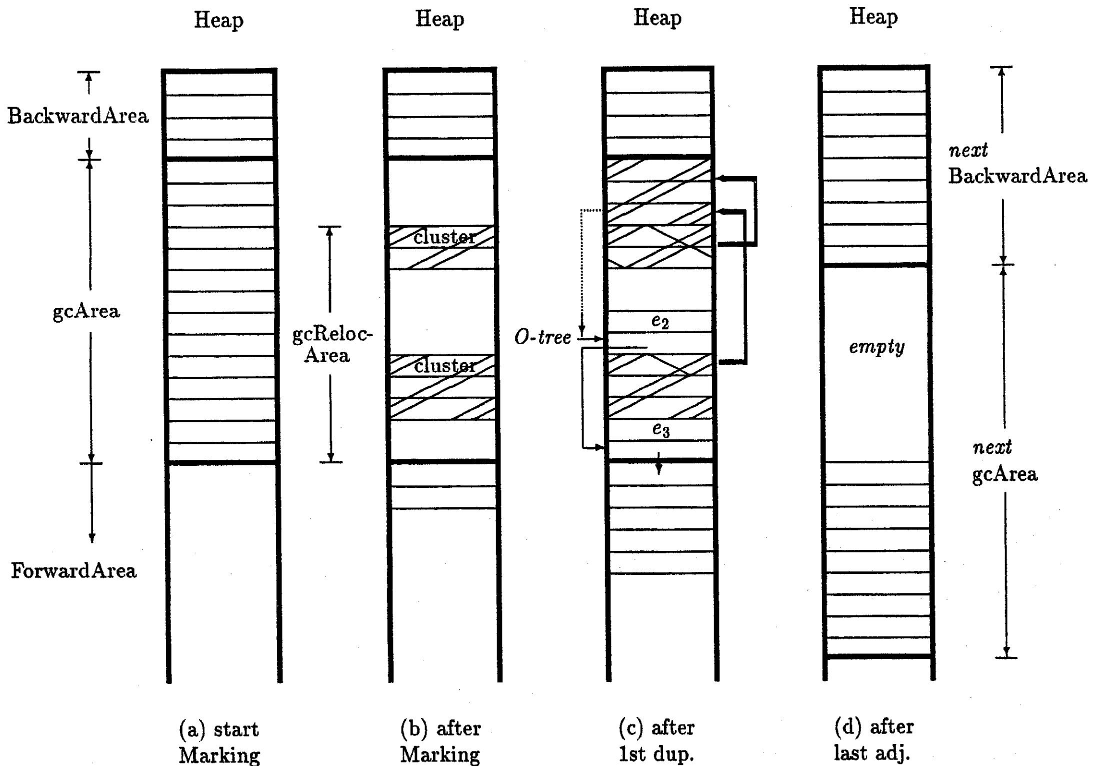
SET は, 書込みフィールドが gcDupArea 内であれば, 先ずそのフィールドを lock する. 元のフィールドと複製の作られるフィールドの双方にこの順で書込みを行い, lock を解除する. 書込みフィールドが gcArea 内ならば, 単に該当フィールドに書込むだけである. もし, gcArea 外のフィールドに書込んだオブジェクトが gcArea 内にあれば, そのフィールドを Rbox に登録する.

SET は, 元のフィールドと複製の置かれるフィールドの両者の一貫性を保つための処理の追加で複雑になる. 排他制御もこのためである. これをより確実 (安全) に行うには, 後者にも lock 機構の付加が必要となる.

3.5 Adjusting

Adjusting は複製されたオブジェクトを指すポインタの補正を行う. Adjusting は, ヒープ (BackwardArea と gcArea 中の使用中フィールド), Rbox, Vstack, Oblis の順にそれらの要素を走査し, それらのオブジェクトが gcDupAreaArea 内にあれば, それが複製を指すようにポインタを補正する. ヒープ中のポインタ補正は, そのフィールドが (SET による書込み作業中で) lock されていれば, lock が解除されるまで補正作業は待たされる.

CONS は gcDuplArea 内のオブジェクトを成分に作るとき, それが複製を指すように (ポインタ補正を) する. gcRelocArea 内のオブジェクトを成分に作るときは, そのフィールドを Rbox に登録する. CONS と, Adjusting のプリミティブ間の排他制御は, Rbox の最終要素の検出以外は不要である.



Note: e_i $(1 \leq i \leq n)$ is the offset value for i-th cluster, and e_1 and e_{n+1} nodes are made if necessary.

図 2 : CSC の動作

SET は, 書込みフィールドが ヒープ内であれば, 先ずそのフィールドを lock する. さらに, そのフィールドが gcDupArea ならば, 複製の作られるフィールドも lock する. ポインタ補正が可能ならば, 補正したポインタを必ず双方に書込み, 両者の lock を解除する. 書込みフィールドが gcArea 内である場合, ポインタ補正が可能ならば, 補正したポインタを必ず書込み, lock を解除する. もし, gcArea 外のフィールドに書込んだオブジェクトが gcArea 内にあれば, そのフィールドを Rbox に登録する. SET の排他制御は, 元のフィールドと複製の置かれるフィールドの両者の一貫性を保つためである. これをより確実 (安全) に行うには, 後者にも lock 機構の付加が必要となる.

3.6 生成順序の逆転

CGC の起動回数が増加すると, 圧縮済みオブジェクトのある BackwardArea と新たなオブジェクトが作られる ForwardArea の間に巨大な空き空間が生じるようになる. この空間の放置はオブジェクト全体の局所性を損ない, 圧縮法の理念に反する. そこで, この空間を新たなオブジェクトの作成に使う場合, それまでに作られた (Forward Area の) オブジェクトも平行して空き領域に移動することになるが, この間は一部のオブジェクト相互で生成順序が逆転する事態が生じる. ただし, この検知は容易である.

4 実装と評価

CSC は現在, 並列プログラミングでの実装化が進行中である. 表 1 は, その前段階としてのソフトウエアシミュレーションによる疑似並列の実測値である. テストプログラムの TPU[2] は CONS データを多量に消費し, その多くを残さない点では CSC に有利であるが, SETQ によるスタック値の変更が比較的多い点で CSC に不利に働く. これが L (純計算の実行時間) の差になって現れる.

ヒープサイズが大きければ, GC の起動回数も少なく, Duplicating–Adjusting の反復の負荷もそれに比例して少なくなる. また, 世代分化の効果も相乗的に現れる. そのことも GC の処理時間から読みとることができる.

表 1 : TPU プログラムの実行結果

Interpreting Program							
S	(MB)		0.10	0.20	0.30	0.40	0.80
L	(sec.)	concurrent	16.64	19.66	19.67	19.93	17.99
		non concurrent	10.23	10.38	10.42	10.44	10.49
G	(sec.)	concurrent	7.00	3.11	1.97	1.45	0.59
		non concurrent	0.30	0.15	0.11	0.11	0.07

Note : S : Heap Size
L : Lisp Time
G : GC Time

5 まとめ

本稿では, CSC と呼ぶ圧縮方式に基づく並列型 GC の設計とその評価について述べた. CSC は, リードバリアを除くために一度に限られた複製しか作れないことなどの負荷も多い. その斬新的なアイデアにもかかわらず, 現版の改良の余地は十分にあるし, 性能評価はそれを暗示している. 使用したアルゴリズムの正当性の証明も含めて, これらは今後の課題である.

参考文献

1. Baker, H. G. : List Processing in Real Time on a Serial Compter, Comm. ACM, 21 (4), 966–975, (1978).
2. Chang, C. L. : The unit proof and the input proof in theorem proving, JACM, 17 (4), 698–707 (1970).
3. Dijkstra, E. W., at el. : On-the-Fly Garbage Collection: An Exercise in Cooperation, Comm. ACM, 21 (11), (1978).
4. Fenichel, R.R. et al. : A LISP Garbage Collection for Virtual Memory Computer Systems, Comm. ACM, 12, (11), 611–612 (1969).
5. Liebeman, H. et al. : A real-time garbage collector based on the lifetimes of objects, Comm. ACM, 26 (6), 419–429 (1983).
6. Steele, G. L. Jr. : Multiprocessing Compactifying Garbage Collection, Comm. ACM 18 (9), 495–508, (1975).
7. Suzuki, M. et al. : MOA — A fast sliding compaction scheme for a large storage space, *IWMM'95*, LNCS. 986, Springer-Verlag, 197–210, (1995).
8. Terashima, M. et al. : Genetic order and compactifying garbage collectors, Inf. Proc. Letters, 7, (1), 27–32, (1978).
9. Terashima, M. et al. A New PHL Compiler (in Japanese), Preprint of WGSYM (IPSJ) 78-3, 17–24, (1995).
10. Yuasa, T. Real-Time Garbage Collection on General-Purpose Machines, Jour. of Systems and Software, 11, (1990).