

# ISM2000 報告

遠藤 敏夫

2000 年 10 月 27 日

メモリ管理 (アロケータ、GC) 分野の最高峰<sup>1</sup>のカンファレンスである ISMM 2000 (International Symposium on Memory Management) に参加した。今回は OOPSLA 2000 と一緒にミネアポリスで行なわれた。前は 98 年で、次回は 2002 年である。

論文は 39 本から 18 本が選ばれた<sup>2</sup>。参加者は 100 人弱で、数千人の OOPSLA に比べればさびしいものの、この規模のしかも分野のせまい会議にしては、さすが最高峰といえる。メンツも、Richard Jones, Hans Boehm, Eliot B. Moss, David Detlefs, Scott Nettles など、GC 業界のすごい人々が集まっていた。また Hoard を作っている Emery Berger 君がむこうから声をかけてくれたのが嬉しかった。

論文発表の全体的な印象としては、技術的にもものすごいことをやっているわけではあまりないなあと感じてしまった。しかしプレゼンとしては、きっちりした評価をし、得られた結果を明示的にかつ単刀直入に述べたものが多く、論文 / プレゼンはこうでなければいけないと思った<sup>3</sup>。

日本の会議と違い、各企業も精力的に参加しており、Microsoft 3 件、Sun 3 件、IBM 2 件 などとなっている。特に Microsoft は前回 1 件 (多分) だったことを考えると、力を入れるべきときに入れているな、という印象をもった<sup>4</sup>。

私は並列マシンの GC を実装しているのでそういう研究には特に注目しているのだが、残念ながら現在の GC コミュニティにはそういう興味を持っている人は少ないようだ。特に「スケーラビリティ」に言及している人は、MS の Steensgaard(セッション 1) と Boehm(informal presentation) くらいで、後は Sun の Detlefs がよく議論に参加していた。私も informal presentation で SGC について話した。他に明示的に並列マシンに言及していたのは IBM haifa のグループ (セッション 5) で、彼らは 1 GC スレッドとアプリスレッドを並列に走らせる。ちなみに今回分散 GC の発表は 1 つ (セッション 5) だった。

---

<sup>1</sup> というか唯一??

<sup>2</sup> 21 本のうちの一つが遠藤のものということになる。以前受け付け番号から判断して「ISM2000 は倍率高すぎ」と言っていたのは間違いだった

<sup>3</sup> やった仕事を順番にだだだしゃべるのではだめってこと

<sup>4</sup> 背景: Java, C#. ただし C# がらみの論文はまだない

## 1 Invited talk by Jon L White

寡聞にしてこの人物を知らなかったのだが、1970年からGCの開発をしていた大御所らしい。これまで関わったGC処理系の話が主だった。

1970年からMIT AI LabでNCOMPLRというLISPのためのGCを開発した。BIBOP技法により効率化を行なった。1983年にXerox PARCで、non-compactifying GCを長期間動かし続けるとフラグメンテーションのために動かなくなってしまうことを発見した。1987年にGenerational Scavenging GCに関わり失敗してしまった。ユーザに「No GC is the Best GC」という印象を植え付けてしまった。GC屋が気をつけなければいけないことの一つは「worst caseでどうなるか?」ということである。平均値だけを見ていると失敗するよ。特異点がどこにあるのか知るのが大切。

## 2 Paper Session

### 2.1 セッション1: Accuracy & Locality

**On the Type Accuracy of Garbage Collection (Martin Hirzel, Amer Diwan from Univ. of Colorado)**

Boehm GCのようなConservative GC(保守的GC)は、型情報がない状況でGCを行なう必要があるので、正確なGCに比べ無駄なオブジェクトを生き残らせてしまう可能性がある(false pointerの存在により)。

11個のCプログラムについて、

- 明示的なfreeを使った場合
- Conservative GCを使った場合
- 正確なGCを使った場合

についてのメモリ使用量を比べる。タイミング的にはGC直後のメモリ使用量を比べる。Conservative GCを使った場合、freeに比べ最悪2倍メモリを使ってしまうのが、正確なGCであれば1.2倍に抑えることができる。

正確なGC時の性能を測るために細工が必要である。この論文では同じプログラム(同じ入力)を2回走らせる。1回目はプログラムに細工がしてあり、あらゆるスタック/ヒープ/大域変数中の値がポインタとして使われるか否か記憶しておく。2回目はそのプロファイルを用いて正確なGCを行なう<sup>5</sup>。

ベンチマークはAustinベンチマーク、Bartlettベンチマーク、Spec95など。

**On the Effectiveness of GC in Java (Ran Shaham, Elliot K. Kolodner from IBM Haifa Research Laboratory, Mooly Sagiv from Tel-Aviv Univ.)**

世の中のほとんどのGC(region analysis除く)は到達可能か否かによってオブジェクトの生死を判定しているが、実際のプログラムでは到達不能になる前にオブジェクトが使い終わっている場合が多

<sup>5</sup>この方法は本番のプログラム実行には現実的とはいえず、Leak detectorに役立てたいと論文にある

い。もし、オブジェクトが使い終わった瞬間がいつも確実に分かるのなら<sup>6</sup>、どのくらいメモリ使用量が得になるか「ひたすら測りました」という話。Java で、ベンチマークは SpecJVM 利用。23% から 74% 得になるだろう、という結果を得た。

**Thread-Specific Heaps for Multi-Threaded Programs (Bjarne Sttensgaard from Microsoft Research)**

未完成ではあるが並列マシン / マルチスレッドについての仕事。スレッドローカルヒープと共有ヒープの 2 種類を持つ。スレッドローカルヒープ中のオブジェクトには、持ち主スレッドのみがアクセスできる。このために Escape analysis を用いる。この条件のもとでは、各スレッドは自分のスレッドローカルヒープを完全に独立に GC できる<sup>7</sup>。Mtrt ベンチマーク (from SpecJVM)、Volano ベンチマークなどを利用。どれくらいのオブジェクトがスレッドローカルヒープに置けるかについて調査したところ、最大の Mtrt では 99% 以上、最小 Volano クライアントでは 5%。ほかに、SpecJBB というサーバサイド Java の性能を測るベンチマークの結果も出していた。マルチプロセッサでの性能測定に使えるらしい。

現状はプロトタイプ実装だそうだ。話を聞いたところ「Shared heap の GC のために君の GC と同じ方法を使えるかもしれない」と言っていた。

**A region-based memory manager for Prolog (Henning Makhholm from DIKU, Univ. of Copenhagen)**

ちょっとこれは分かりません。Tofte, Talpin の region-based model に backtrack, cut を追加した... と Abstract に書いてあります。

## 2.2 セッション 2: Implementation

**Compact Garbage Collection Tables (David Tarditi from Microsoft Research)**

Java などの正確な GC は、スタック中のポインタのありかを知るために、コンパイラが作成する GC テーブルというのを用いる。このテーブルの大きさは実行ファイルの 15% から 20% になってしまふこともある。これを、4% 以下にする。

**Reducing Garbage Collector Cache Misses (Hans-J. Boehm from Hewlett-Packard Laboratories)**

Boehm による、Boehm GC の最適化の話。Mark&sweep GC では、キャッシュミスコストが GC 時間のかなり (1/3 など) を占める。メモリ Read 直後にその値を使うと、遅延隠蔽がまるできなく、良くない。そのような例の一つはオブジェクトの子供スキャンである。

遅延隠蔽のために、以下のようにプリフェッチを入れる (Prefetch on grey)。

- あるオブジェクトの最初の word を、mark stack からの pop 時より前の、mark stack への push 時に読んでおく。

---

<sup>6</sup> その一部を教えてくれるのが linear type というわけですね

<sup>7</sup> 私の処理系でもスレッドローカルフリーリストを用いているが、だれがどのオブジェクトをアクセスするかは混沌としている。このため GC はみんな一緒に行なう

- 大きいオブジェクトをスキャンする際には、いくつか先のキャッシュラインを読んでおく (規則的アプリでは常識)。

Zorn ベンチマーク、ghostscript などを利用。Pentium II で最大 17%, HP PA-RISC で最大 11% の速度向上を得た。

**Memory Allocation with Lazy Fits (Yoo C. Chung, Soo-Mook Moon from Seoul National Univ.)**

フリーリストを使ったアロケータにおいて、フリーリストのどの要素を利用するかという戦略は複数ある (first fit, best fit など)。Lazy fit という方式を提案する。方針は簡単で「前回に使った要素がまだ残っているならそれを使う」ということである。フリーリストの要素サイズの方がアロケータ要求サイズよりずっと大きい場合に良い効果がある。要素を食い潰してしまった場合の戦略の選択により、lazy first, lazy best... が考えられる<sup>8</sup>。

**Conservative Garbage Collection for General Memory Allocators (Gustavo Rodriguez-Rivera, Mike Spertus, Charles Fiterman from Geodesic Systems)**

Conservative GC においては、あるアドレスが生きたオブジェクトに含まれるか否か、その先頭とサイズは... という情報を高速に検索できる必要がある。その解決策の一つは Boehm GC も採用している BIBOP (Big bag of pages: あるページ中には必ず同サイズオブジェクトが含まれる) だが、これはメモリ利用効率が良くない場合がある。BIBOP 以外の (first fit でもなんでも) アロケータと conservative GC を組み合わせるために、malloc table というのを提案した。このテーブルには全ライブオブジェクトのアドレスが昇順にならんでおり、それをバイナリサーチすることによって上に述べたような情報を検索する。既存のアロケータである Doug Lea allocator に本方式を使った GC を組み合わせた<sup>9</sup>。Zorn のベンチマーク利用。Boehm GC と比べ、同じメモリ量を使ったときの GC 回数が減った (すなわちメモリ利用効率が良い)。ただし、全体の実行時間はそれほど変わらない。

## 2.3 セッション 3: Hardware Support

**Concurrent Garbage Collection Using Hardware-Assisted Profiling (Timothy H. Heil, James E. Smith from Univ. of Wisconsin-Madison)**

“On-chip multithreading” というキーワードでいきなり分からなくなりました。(プロセッサ内の?) 裏スレッドに GC と write barrier をさせる話のようです。

**Concurrent Garbage Collection Using Program Slices on Multithreaded Processors (Minoj Plakal, Charles N. Fischer from Univ. of Wisconsin-Madison)**

こちらも “On-chip multithreading”。Java ユーザプログラムをプログラム変換することにより Reference count 処理だけを行なうプログラムを抽出する。そしてそれを裏スレッドで動かす。

**Cycles to Recycles: Garbage Collection on the IA-64 (Richard L. Hudson, Sreenivas Subramoney, Weldon Washburn from Intel, J. Eliot B. Moss from Univ. of Massachusetts)**

<sup>8</sup>最近 Segregated free list などが普及しているので、どうも時代遅れではなからうか

<sup>9</sup>Doug Lea allocator がどんな方式か未調査です

Intelの新アーキテクチャであるIA-64のためのJVMのGCを実装する際の雑多な技法。世代別コピーGCで、旧世代のためにTrainアルゴリズム採用。

- IA-64にはレジスタがたくさんあるから、アロケーションポインタやストアバッファポインタ専用にレジスタを使っても大丈夫。
- プレディケートレジスタを用いたアロケーションの効率化: スレッドAのアロケーション途中でタスクスイッチが起り、他のスレッドBが先にアロケートしてしまう場合でも、Aはすぐにそれに気づくことができる(詳細は未調査)。

## 2.4 セッション4: Profiling & Object Lifetimes

**The Case for Profile-Directed Selection of Garbage Collectors (Robert Fitzgerald, David Tarditi from Microsoft Research)**

アプリの性質によって、世代別GCを使うべきか否かは異なる。SpecJVM, IMPACTなど20のベンチマークに対して、単世代のコピーGC、世代別コピーGCを使った性能を比べることにより、「全ベンチマークに対して最高の性能を出すアルゴリズムはない。一長一短」ということが分かった。トレーニング実行を通して最適なアルゴリズムを選ぶことにより、固定アルゴリズムを使う場合に比べてどのベンチマークでも15%以上の向上が見られた<sup>10</sup>。

なお、世代別GCには書き込みバリアの実装の違いにより4種類実験している。なお書き込みバリアの種類には、Sequential store buffer(書き込みがあったアドレスを何も考えずに専用バッファへ加えていく)と、Card marking 3種類( $2^n$ バイト単位の領域(カード)ごとに「書き込みあったよビット」を設ける)である。

**Efficient Object Sampling Via Weak References (Ole Agesen from VMware, Alex Garthwaite from Sun Microsystems Laboratories)**

**Dynamic adaptive pre-tenuring (Timothy L Harris from Sun Microsystems Laboratories)**

この2件を一人が続けて発表していた。世代別GCにおいて最近pre-tenuringという技法が話題になっているようだ(P. Cheng et al. PLDI98など)。世代別GCの基本は、オブジェクトを確保するときにはまず新世代ヒープに確保し、しばらく生き残ったものだけを旧世代ヒープに移す(tenure)ということである。Pre-tenuringとは、あるオブジェクトが長く生き残ることがあらかじめ分かっているならばいきなり旧世代ヒープに確保するという技法である。これにより後々無駄なコピーが省ける。

本論文では、実行時に寿命データを記録することにより、同一実行中にpre-tenureを行なう。前提として、バイトコード的に同一のnew命令によって確保されるオブジェクトは同じ性質である(同一カテゴリーに属する)、と見なす。実験では各カテゴリー中のオブジェクトの95%以上が長寿命であったら、それ以降そのカテゴリーのオブジェクトをpre-tenureする<sup>11</sup>。ところで、実行中の全オブ

<sup>10</sup> トレーニング実行の詳細は、論文を流し読みした限りでは見つからなかった。小入力サイズでトレーニングし、本番で大入力サイズを使えるようになっていないと実用に使えないと思うが...

<sup>11</sup> 95%割ったらpre-tenure取り消しか?

ジェクトについて寿命データを集めるのは重い。そのため、数回に一度の `new` 命令で確保されたオブジェクトについてのみ統計をとる (object sampling)。

SpecJVM, DeltaBlue などを実験し、GC 時間を 6% 減らすことができた。

**On Models for Object Lifetime Distributions (Darko Stefanovic from Princeton Univ., Kathryb S. McKinley, J. Eliot B. Moss from Univ. of Massachusetts)**

オブジェクト寿命の確率モデルをいくつか挙げて、58 個の実プログラムの挙動と比べた。全部のプログラムに沿うモデルはないものの、ガンマ分布が一番近い。

## 2.5 セッション 5: Concurrent & Distributed

**A Generational Mostly-concurrent Garbage Collector (Tony Printezis from Univ. of Glasgow, David Detlefs from Sun Microsystem Laboratories)**

Java のための並行で世代別な GC の実装とその性能について。GC は専用スレッドが行なうのでマルチプロセッサを利用できる。マークスイープ方式で、特徴は以下の通り。

- GC 開始時には全スレッドを止めてルートスキャンを行なう<sup>12</sup>。それ以降のマークはユーザと並行。
- スイープもユーザスレッドと並行。
- ライトバリアにはカードマーキング採用。

SpecJVM のうち 5 つで実験を行ない、そのうち 3 つでデフォルト GC よりトータル時間が短くなった。平均停止時間は 5 つとも短くなった。一方、ヒープ利用量は 5 つとも 30% から 100% 近く増えてしまった。

**Implementing an On-the-fly Garbage Collector for Java (Tamar Domani et al. from IBM Haifa Laboratory)**

マルチプロセッサ上のマルチスレッド Java プログラムのための GC について。Doligez, Leroy, Gonthier (POPL93,94) の Concurrent マークスイープ GC アルゴリズムに基づく。1 つの GC スレッドと複数のユーザスレッドが同時に動く。特徴の一つは、ユーザスレッドごとにマークバッファ (ライトバリア時に書かれたオブジェクトをためるバッファと思われる) を持つことである。

4 PowerPC プロセッサの並列マシンを使い、pBOB ベンチマーク (前述の SpecJBB を改造したものだそうだ) で実験。Stop-the-world GC (逐次だろう) に比べ、3 スレッド以上で TPM (transactions per minute、スループットのこと) は向上する。20 スレッドの場合は 10 から 18% 向上。

Future work では、プロセッサがもっと増えてアロケーションに GC が間に合わない困るから GC 自体を並列にする必要があるだろう、とあった<sup>13</sup>。

**Diffusion Tree Restructuring for Indirect Reference Counting (Dr Peter Dickman from Univ. of Glasgow)**

<sup>12</sup> これからの並行 GC は「ルートの大きさ」に対処できなければいけないと思う。八杉さんの SPA 2000, P. Cheng の PLDI98 など

<sup>13</sup> どうせならそこで俺の論文も refer して欲しかったなあ (笑) この人たちとは話そびれてしまった

分散 GC アルゴリズムの一つである Indirect Reference Counting(IRC) の改良の話。IRC においては、実体がプロセッサ A にあるオブジェクトの遠隔参照をプロセッサ B からプロセッサ C に渡したとき、B は遠隔参照についている参照カウントを一つ増やす。C は B からもらったことを永遠に覚えている。この過程では B, C 間の通信のみで充分であり、A, C 間の通信も必要な単純 RC より優れている。プログラムの実行が進むと、遠隔参照をもらった関係にそって木構造ができる (上の例の場合は C は B を知っており、B は A を知っている。A が木のルート)。この木を diffusion tree と呼ぶ。この木があまりに深くなってしまうと、プロセッサ障害によって影響を受ける可能性が高くなってしま<sup>14</sup>。

このために diffusion tree 再構築アルゴリズムを提案する。プロセッサ C の親が B であるとき、同オブジェクトの参照が D からやってきたとする。もし、D のほうが B よりルートに近かったら、C は自分の親を D に変更する。

### 3 Informal Presentations

Informal Presentations Session の長さは一時間で、前日に登録すれば論文がなくても発表できる機会があった。遠藤を含めて 9 人が発表した。

**Hardware Support for Dynamic Memory Management (? from Illinois Tech. Univ.)**

malloc, free, realloc, mark, sweep をハードウェアレベルのプリミティブにしてしまおうという話らしい。

**Towards a performance portable infrastructure for memory management systems (Ken Wakita, Yuji Uchiyama from Tokyo Institute of Technology)**

東工大の内山君の話 (SPA2000) を脇田さんが発表した。GC の実装は言語処理系の他の部分と密接に関わりすぎている。モジュール性を高めつつ性能ロスを抑えたい。手段はプリプロセッサなど (詳細を聞きそびれてしまった)。

**SGC: A Scalable Memory Management Module on Multiprocessors (Toshio Endo from Univ. of Tokyo)**

共有メモリ並列マシン上の Stop & parallel GC である、SGC の話。知名度の低さを考えて、GC 速度向上 (E10000 で 25 から 40 倍の速度向上) とアロケート速度向上の話にとどめた。SMP(E10000) ではメモリ混雑はそんなに問題はないが DSM(O2000) ではアプリによって頭打ちになってしまう、くらいのまとめをした。私の英語があまりにへばいせいもあるのか会場の反応はなかった。

「Bohem GC 6.0 alpha 1 より断然速いよ」と言って「Bohem GC も将来速くなると期待しています」とナマイキなことを言ったらあとで Boehm が「次の版では速くするよ」と言ってきた。

**Scheduling of Garbage Collection (?)**

Boehm GC を用いた、「GC or ヒープ拡張」の選択戦略の話。Boehm GC ではメモリが不足したときに、前の GC 以来「充分」メモリ確保が起っていれば GC を、そうでなければヒープ拡張をする。その閾値の決定を動的に変えてみると全体の性能が、普通と同じか少し良くなったという話。

<sup>14</sup>あと、オブジェクトが到達不能になってから実際に回収されるまでの時間も長くなると思う

### **Memory Sandboxing (David Bacon from IBM T.J. Watson)**

Java で region ベースのプログラムを書くためのライブラリらしい。ユーザプログラムがオブジェクトを作るときに、それが所属する region を指定する (?)

### **Scalable Garbage Collection (Hans-J. Boehm from Hewlett-Packard Laboratories)**

Boehm 御大による、共有メモリ並列マシン上の並列 GC について。「これはさっきの Toshio Endo がすでにやっている方法で...」などと持ち上げてくれた。感謝。

4CPU, メモリバス 750MB/s のマシンで実験しているが、メモリ混雑で悩んでいるようだった。これは先ほどの私の話と逆で、後で Boehm と話をした。E10000 のような恵まれたマシン (64CPU, 10GB/s) だけでなく、小さいマシンでも実験してみるべきだ、ということになった<sup>15</sup>。

### **High performance GC for Java with thread specific heaps (Pet Caudill from Instantiation?)**

スレッド毎にヒープを持たせて、メモリ確保時のボトルネックをなくす。スレッドローカルヒープは個別に GC することができる。

当然チェックしておくべき話だったのだが、これ以上メモとりそびれてしまった。無念。

### **Profile driven pretenuring for Java (? from Univ. of Massachusetts)**

Pretenuring についてはセッション 4 参照。メモし忘れ。

### **Improving Memory Reference (Chilimbi from Microsoft Research)**

オブジェクトグラフを見て、オブジェクトのレイアウトを並び替えるという話。

---

<sup>15</sup>でも、CPU 毎のバンド幅はたいして変わらないよなぁ？ということに後で気が付いた