

仮想記憶を利用した並列ガーベジコレクションの 高速記憶割り当て

小出 洋 鈴木 貢

電気通信大学情報工学科

広い仮想記憶空間をもつ共有メモリ型並列計算機のための効率的な領域確保法について提案する。本領域確保法は、コピー方式並列ガーベジコレクションなら一般的に使用可能であり、記憶領域を割り当てる位置を示すポインタは単調増加する。従って、単純な排他制御命令のみによって効率的な領域確保が可能である。また、本論では本領域確保法を使用した並列ガーベジコレクションの例を示した。

Fast Memory Allocation for Parallel Garbage Collectors based on the Virtual Memory System

Hiroshi KOIDE Mitsugu SUZUKI

Department of Computer Science, The University of Electro-Communications
Chofu-shi, Tokyo, 181, Japan
E-Mail : koide-h@cs.uec.ac.jp

We propose a new memory allocation scheme for a shared memory parallel computer which has a vast virtual memory space. We can apply the memory allocation scheme to almost all variations of the copying garbage collector. The pointer to the location for the next allocation is monotonously increased. This property makes a memory allocation so simple that we can optimize the performance. Lastly, we present a garbage collection algorithm which uses the memory allocation scheme.

1 はじめに

一般的に、ワークステーション以上の計算機はページ方式の仮想記憶を実現するためのハードウェアを備えている。最近では、その仮想記憶空間が広がってきている。一方、1 台の計算機に複数のプロセッサを内蔵して処理性能を向上させたマルチプロセッサ計算機が一般的なものになりつつある。本論文では、こうした計算機において効率的な領域確保法について提案する。また、本領域確保法が並列型や実時間型コレクタ [Bak78, AEL88, Daw82] に適用可能である例を示す。

コピー方式コレクタ [FY69] は、その計算時間が記憶領域の大きさではなく使用中オブジェクトの総容量に比例する。そのため、大容量の主記憶をもつ計算機上でよく使用される。また、コピー方式はいままでにオブジェクトの世代管理 [LH83, Ung84, App89] に多く使用されてきた。また、並列型や実時間型コレクタ [Bak78, AEL88, Daw82] にも多く使用されてきた。本領域確保法はこれらのコピー方式コレクタに有効である。

本領域確保法では、マルチプロセッサ計算機として共有メモリ型並列計算機を想定する。これは、プロセッサ数が少ない (現在では約 32 以下であることが多い) ため、各プロセッサにスヌープキャッシュを持つことができ、クロスバースイッチを使用したバス構成が可能である。そのため、記憶領域のアクセス時にプロセッサ間の競合が起こりにくい。このような並列計算機は最近普及してきており、計算機としては特殊なものではなくなりつつある。なお、シングルプロセッサの計算機において時分割で複数のプロセスを実行するシステムにおいても本領域確保法を適用することが可能である。

また、ページ方式の仮想記憶を実現するためのハードウェアを備えていて、例えば 64 ビットアドレスの広大な仮想記憶空間を利用可能であるものとする。このハードウェアは、近い将来にワークステーション以上の計算機において一般的に利用可能になるものと思われる。

本論文では、記憶領域にオブジェクトを動的に確保することを繰り返して実質的な計算を行

なうプロセスをミューテータと呼ぶ。ミューテータがつくるガベージ (ミューテータから参照できなくなったオブジェクト) の領域を回収して再利用するプロセスをコレクタと呼ぶ。また、実時間型コレクタは、コレクタの作業のためにある定数時間 (例えば、1/100 秒) より長い時間、ミューテータを停止させる必要のないコレクタをいう。並列型コレクタは、多少の排他制御を例外としてコレクタとミューテータが並列動作可能なコレクタをいう。この性質があると、同時に複数のミューテータがオブジェクトの確保を行なわないという条件のもとで、複数のミューテータが同時に動作可能になる。ミューテータが割り込み待ちなどで動作できない場合でもコレクタは動作可能なので、並列性はシングルプロセッサにおいても有意義である。

本領域確保法を利用した場合、記憶領域を割り当てる位置を示すポインタ (アロケーションポインタ) を単調増加にすることが可能になる。従って、例えば `fetch-and-add` 命令をミューテータ間の排他制御命令として使用することにより効率的な領域確保を行なうことが可能になる。また、本領域確保法ではページ方式仮想記憶ハードウェアを利用しているが、2 次記憶へのページ書き出しが起きないようにコレクタの起動条件を定める。従って、アクセスに時間を要する 2 次記憶を使用しなくても済み、効率の良いミューテータの動作が期待できる。

2 仮想記憶を利用した領域確保法

2.1 仮想記憶領域の構成

本領域確保法を使用する場合の典型的な仮想記憶領域の割り当てを図 1 に示す。各領域は使用部分に物理記憶ページが割り当てられる。このページ割り当ての作業はアルゴリズムの説明中に明示的には示さない。ディマンドページング (demand paging) の機構により自動的に行なわれるとする。不要になったページの解放はアルゴリズムの説明中に明示的に示した。本論文では、仮想記憶空間の連続領域でアドレスが p 以上で q より小さいものを $[p, q)$ と表す。

図 1 で、仮想空間全体は $[v_s, v_e)$ である。実際は

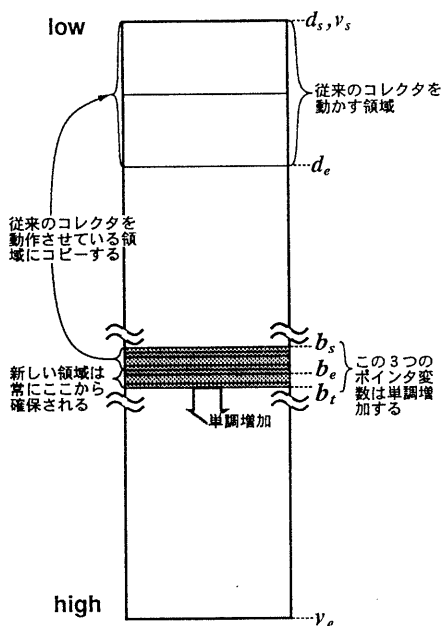


図 1: 記憶領域の構成

$[d_e, v_e]$ は $[d_s, d_e]$ や $[b_s, b_t]$ と比較して大きい。コピー方式の並列型コレクタ (例えば, [RHH85]) は, $[d_s, d_e]$ を使用して動作する。新しいオブジェクトはポインタ b_t から確保する。 $[b_s, b_t]$ 中の使用中オブジェクトは, $[d_s, d_e]$ でコレクタが動作するときに $[d_s, d_e]$ 中のオブジェクトと一緒にコピーされる。コピーが終了した部分に割り付けられたページは不要になるので解放される。

$[b_s, b_t]$ は、仮想記憶空間をアドレスの上位方向に向かって単調に進んでいく。つまり各ポインタ b_s, b_e, b_t は単調増加する。

2.2 記憶領域の確保法

ミューテータの並列動作のため、いつでも新しいオブジェクトのための記憶領域確保可能である必要がある。本方法では、いつでもアロケーションポインタ b_t から領域確保可能である (図 1)。 b_t は、仮想記憶空間 (例えば, 2^{64} 語) 上で単調増加する。仮想記憶空間は広いので b_t が境界を越えたか否かの検査は不要である。

各プロセッサがメモリの値の不可分な増加を行なう排他制御命令を使用できると仮定すると、ポインタ変数 p に大きさ $\text{sizeof}(\text{Object})$ のオブジェクトを割り当てるコードは図2のようになる。fetch-and-add 関数は、仮定したプロセッサ命令を使用して、ポインタ変数とその増加値を引数に取り、ポインタ変数の値を返すこととその値を増加することを不可分に行なう。

```
Object *p;
p = fetch-and-add(b_t, sizeof(Object));
```

図 2: 記憶領域の確保

2^{64} 語の仮想アドレス空間では、毎秒 1M 語領域を使用して、500 年以上ミューテータを動作させることが可能である。アロケーションポインタ b_t は $[d_e, v_e]$ の広い仮想記憶空間を走査する。しかし、実際には実記憶ページだけを使用する。

従来の領域確保法では、領域割り当てのための仮想記憶空間と実記憶空間が同じ大きさである。アロケーションポインタがこの領域の最上位に達すると、アロケーションポインタは再設定される。ミューテータは、この検査を領域確保の度に行なう必要がある。

1つのアロケーションポインタの変更 (増加と再設定) は、すべてのミューテータどうして排他的に行なわれる。この部分で排他的動作が必要な部分が短いことは、ミューテータの並列動作が効率的に行なわれるための必要条件である。

[RHH85] や [IT93] で使用される領域確保法は、並列コピー方式に使用可能である。[RHH85] では、記憶領域を分割して各ミューテータに局所領域を割り当て、個別のアロケーションポインタを割り当てる領域確保法を使用している。この方法では、アロケーションポインタに関する排他制御は不要である。しかし、どれかひとつ局所領域を使い果たすとガーベジコレクション終了までそのミューテータは動作できない。

また、[IT93] はオブジェクトをサイズ別に分類し、一定の大きさの連続領域別にガーベジコレクションを行なう方法である。しかし、オブジェクトのサイズ別の分類や大域変数の排他的な参照が必要である。従って、本領域確保法と比較して、記憶領域確保のための排他的動作が

必要な部分は大きい。

3 本領域確保法の適用例

本章では、本領域確保法を並列コピー方式に適用する例について述べる。例に使用したコレクタは、書き込み障壁 (write barrier) を使用した並列コピー方式である。書き込み障壁とは、ミューテータの書き込み時に変更されたりリスト構造の情報をコレクタに渡す作業である [Wil92]。

本方法の実行環境は、1章で想定した共有メモリ型並列計算機である。想定した計算機において、ひとつのコレクタと複数のミューテータが並列に動作する。

仮想記憶に関する仮定に関する仮定は、2.1節で述べた通りである。つまり、実記憶ページを仮想アドレス空間に割り当てる作業はディマンページングの機構により自動的に行なわれるものとする。これは仮想記憶を実現している計算機で標準的に備えている機構である。この機構を利用することで仮想記憶空間に効率的な実記憶ページ割り当てが可能になる。ページ割り当ての作業はアルゴリズムの説明中に明示的に示さない。不要になったページの解放はアルゴリズムの説明中に明示的に示す。

ガーベジコレクション対象領域の各オブジェクトには、ガーベジコレクション中に書換えが行なわれたことを表すビット (変更ビット) と、各プロセス (すべてのミューテータとコレクタ) が排他的なオブジェクトの読み書きを行なうためのロック用ビット (lock ビット) があるものとする。

このコレクタの動作中に以下の条件が満たされた場合、ミューテータの並列動作が可能である。

1. ミューテータがすべての使用中オブジェクトの読み書き可能
2. 新しいオブジェクトのための記憶領域確保がいつでも可能
3. 同時に2つ以上のミューテータが記憶領域確保を行なわない

3.1 記憶領域の構成

本節では、次節以降のアルゴリズムの説明に用いるための各ポインタ変数と各領域の役割について述べる。仮想記憶領域の構成は図3の通りである。

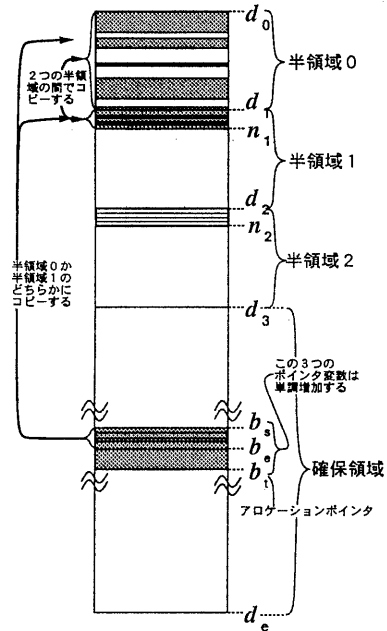


図3: 書き込み障壁並列コピー方式の記憶領域

各ポインタ変数の意味

図3の各ポインタの意味を表1に示す。

図3の各領域の役割は次の通りである。仮想記憶領域は、半領域0,1,2と確保領域に分割される。最初、オブジェクトは確保領域に割り当てられる。次に、半領域0と半領域1の間でコピーされる。

コレクタ起動時に、半領域0または半領域1の使用されていない半領域がコピー先領域である。コレクタ起動時に使用中の半領域と確保領域のガーベジコレクション対象領域がコピー元領域である。つまり、半領域0から半領域1にコピーする場合、コピー元領域は $[d_0, d_1)$ と $[b_s, b_e)$ になる。コピー先領域が半領域0の場合、コピー元領域は $[d_1, d_2)$ と $[b_s, b_e)$ になる。

表 1: 各ポインタ変数の役割

d_0	半領域 0 の開始アドレス
n_0	半領域 0 の使用部分の次のアドレス
d_1	半領域 1 の開始アドレス
n_1	半領域 1 の使用部分の次のアドレス
d_2	半領域 2 の開始アドレス
d_3	確保領域の開始アドレス
b_s	確保領域のガーベジコレクション対象領域の開始アドレス
b_e	確保領域のガーベジコレクション対象領域の終了アドレス
b_t	アロケーションポインタ
s	コピー先領域の未スキャン領域の開始アドレス
b	コピー先領域の次にコピーすべきポインタ

1. 半領域 0、半領域 1: この 2 つの領域は等しい大きさの半領域として扱う。コレクタは、使用中オブジェクトを 2 つの半領域の間でコピーする他、確保領域からコピー先半領域にコピーする。

半領域 0 または半領域 1 のうち、実際に実記憶ページが割り当てられている使用中部分は、 $[d_0, n_0)$ または $[d_1, n_1)$ である。

2. 半領域 2: この領域はコピー元領域が書き換えられてもコレクタの作業を続けられるように、書き換えるコピー元領域のポインタを保存する。この領域に必要な大きさは、一回のコレクタが起動して終了するまでにミューテータが書き換える未コピーのコピー元領域のオブジェクト中のポインタの総容量である。

半領域 2 のうち、実際に実記憶ページが割り当てられている使用中部分は、 $[d_2, n_2)$ である。

3. 確保領域: ミューテータは、アロケーションポインタ b_t から新しいオブジェクトを、コレクタの動作とは並列に割り当てることが可能である。コレクタは、起動された時点でページ 3 にある使用中オブジェクトの集まり $[b_s, b_t)$ をコピー先領域にコピーする。

確保領域のうち、実際に実記憶ページが割り当てられている使用中部分は $[b_s, b_t)$ である。

3.2 コレクタプロセス

次にコレクタが行なうガーベジコレクションの手続きを示す。コレクタは、ミューテータと並列に次のガーベジコレクションの手続きを繰り返す。アルゴリズムをより詳しく手続的に表したものを付録に示す。

1. すべてのミューテータを一時停止する。
2. コピー元半領域とコピー先半領域の役割りを切り替える。
3. ミューテータの root から直接指されているオブジェクトをコピー元領域からコピー先領域にコピーする。コピー元オブジェクトからコピー先オブジェクトに転送先ポインタを設定する。
4. ミューテータを再開する。
5. コピー先領域にコピーされたオブジェクトを走査する。

スキャンの作業はコピー先領域にコピーされたオブジェクトを走査する。 s を走査中オブジェクトへのポインタ、 b を次にコピー元オブジェクトをコピーすべき場所へのポインタとする。

ポインタ s の指すオブジェクトが未コピーのコピー元オブジェクトへのポインタだった場合、ポインタ b の指す場所にコピーする。ただし、ポインタ s の指すオブジェクトに変更ビットが立っていた場合、そのオブジェクト中のポインタは無視する。

コピー先領域中の走査が終了した後、半領域 2 中のポインタの走査を行なう。

3.3 ミューテータプロセス

最初に、ミューテータがコレクタとは並列に使用中オブジェクトを読み書きするために必要な事項について述べる。このため、ミューテータの読み書きと使用中オブジェクトの存在する領域別に分類する。

1. 読み出し

コピー元領域を参照する場合で、参照したオブジェクトがコピー先領域への転送先ポインタ

の場合、その読み出しを転送先ポインタの指すオブジェクトの読み出しに置き換える。この作業を効率的に行なうためには、そのためのハードウェアを利用する必要がある。

2. 書き込み

コピー先領域の走査済みの領域 $[d_0, s)$ または $[d_1, s)$

特別な処理を行わずに書き換えることができる。

コピー先領域の未走査の領域 $[s, b)$

書き換えるオブジェクトに変更ビットが立っていた場合、特別な処理を行わずに書き換えることができる。

書き換える前に、書き換えるオブジェクトを走査する。

コピー元領域

書き換えるオブジェクトに変更ビットが立っていた場合、特別な処理を行わずに書き換えることができる。

そうでない場合、書き換える前に書き換えるオブジェクト中のポインタを半領域2に保存し変更ビットを立てる。

ミューテータは使用中オブジェクトの読み書きの他に、新しいオブジェクトのための記憶領域を確保することができなければならない。本領域確保法を使用すれば、いつでも確保用領域のアロケーションポインタ b_i から割り当てることができる。

本方法では、オブジェクトをコピー元領域からコピー先領域にコピーして転送先ポインタを設定する部分に局所的な排他制御が必要である。このために lock ビットをコピー元領域のすべてのオブジェクトに用意する。

3.4 本適用例の正当性

本方法では、特定のオブジェクトに関して lock ビットを用意してロックすることにより排他制御を行なっている。しかし、コレクタはあるオブジェクトのロック中に他のオブジェクトをロックすることはないので、プロセス間のデッドロッ

クは起こらない。従って、1 回のガーベジコレクションは必ず終了する。

また、コレクタはガーベジコレクションの開始時点の使用済オブジェクトをすべてコピー先領域にコピーする。ガーベジコレクション中に一度ガーベジになったオブジェクトは再び使用中になることはないため、本方法では使用中オブジェクトを誤ってガーベジと判断することはない。

また、あるコレクタの作業中に発生したガーベジは、次のコレクタの作業で回収される。

一方、確保用領域で実記憶ページが割り当てられている領域は、記憶領域の割り当てとガーベジコレクションを繰り返すことより、記憶領域の上位方向に移動する。しかし、仮想アドレス空間は大きいので他の領域と重なるなどの問題は起こらない。

4 まとめ

本論文では、まず最初に広い仮想記憶空間をもつ共有メモリ型並列計算機のための効率的な領域確保法について提案を行なった。本領域確保法は、コピー方式並列ガーベジコレクションなら一般的に使用可能であり、記憶領域を割り当てる位置を示すポインタを単調増加にできる。最後に、本領域確保法を生成順序保存型コピー方式に適用した例を示した。

適用例に実時間性を導入することも含めて、本領域確保法の実際の場合での評価は今後の課題とする。

A 適用例の詳細なアルゴリズム

```
//////////////////////////////////////////
// 書き込み障壁並列コピー方式

Object *d0; // 半領域0の開始アドレス
Object *n0; // 半領域0の使用部分の次のアドレス
Object *d1; // 半領域1の開始アドレス
Object *n1; // 半領域1の使用部分の次のアドレス
Object *d2; // 半領域2の開始アドレス
Object *n2; // 半領域2の使用部分の次のアドレス
Object *d3; // 確保領域の開始アドレス
// 確保領域のgc対象領域の開始アドレス
Object *bs;
// 確保領域のgc対象領域の終了アドレス
Object *be;
```

```

Object *bt; // アロケーションポインタ
// コピー先領域の未スキャン領域の開始アドレス
Object *s;
// コピー先領域の次にコピーすべきポインタ
Object *b;
// ミューテータが直接参照するオブジェクト
Object *r[N];
Semispace sp; // 現在使用中の半領域

//-----
// ミューテータプロセス
process mutator(void) { 実質的な計算を行なう; }
// 新しいオブジェクトを割り当てる
Object *allocate(Type type)
{ return fetch_and_add(bt, size(type)); }

// 例外処理 (読み出し)
exception
Object *mutator-read-something(Object *p) {
    ミューテータがコピー元領域を参照する場合で、参照した
    オブジェクトがコピー先領域への転送先ポインタの場合、
    その読み出しを転送先ポインタの指すオブジェクトの読み
    出しに置き換える; }

// 例外処理 (コピー先領域の走査済み領域)
exception
void mutator-write-on-[d_sp,s](Object *p, val)
{ // 特別な処理は不要
    lock(p); *p = val; unlock(p); return; }

// 例外処理 (コピー先領域の未走査の領域)
exception
void mutator-wirte-on-[s,b](Object *p, val)
{ // 書き換える前に走査する
    lock(p);
    if (modifiedp(p) == FALSE)
        scan_object_body(p);
    *p = val;
    unlock(p); }

// 例外処理 (コピー元領域)
exception void
mutator-write-on-コピー元領域 (Object *p, val)
{ lock(p);
    if (modifiedp(p) == FALSE)
        scan_object_body(p);
    for (i = 0; i < size(p); i++)
        if (pointerp(p + i) == TRUE)
            *n++ = *(p + i);
    *p = val;
    set_modified_flag(p);
    unlock(p);
}

//-----
// コレクタプロセス
process collector(void) // 処理を繰り返す
{ forever { すべての mutator を一時停止する;
    flip(); update();
    mutator を再開する;
    scan(); scan_半領域 2(); } }

void flip(void) // 半領域 0,1 の役割を切替える
{ if (sp == 半領域 0) {
    // 半領域 1 をコピー先領域に設定
    sp = 半領域 1; s = b = d1;
    } else {
    // 半領域 0 をコピー先領域に設定
    sp = 半領域 0; s = b = d0;
    } }

void update(void) // スキャンの種をコピー
{ for (i = 0; i < N; i++)
    copy((Object *)*r[i]); }
void scan(void) // スキャンを行なう
{ // s が b に追い付くまで繰り返す
    for (; s < b; s++) scan_object(s); }
void scan_半領域 2(void)
{ for (p = d2; p < n2; p++)
    scan_object(p); }
void scan_object(Object *p)
{ lock(p); scan_object_body(p); unlock(p); }

void scan_object_body(Object *p)
{ if (modifiedp(p) == FALSE) {
    if (pointerp(p) == TRUE) {
        if (tospace((Object *)*(Object *)*p)
            == FALSE)
            copy((Object *)*p); // まだコピーされていない
        if (tospace(p) == FALSE)
            *p = *(Object *)*p; // 転送先ポインタの解消
    } } }

void copy(Object *p) // 実際にコピーする
{ for (i = 0; i < size((Object *)*p); i++)
    *(b + i) = *((Object *)*p + i);
    *(Object *)*p = b;
    b += size((Object *)*p); }

//-----
// その他
// 型 type のサイズを調べる
int size(Type type);
// p の指すオブジェクトのサイズを調べる
int size(Object *p);
// p の指すオブジェクトは CONS 型か?
boolean consp(Object *p);
// p の指すオブジェクトはポインタか?
boolean pointerp(Object *p);
// p の指すオブジェクトはコピー先領域にあるか?
boolean tospace(Object *p);
// 修正ビットをセットする
void set_modified_flag(Object *p);
// p の指すオブジェクトの変更ビットが立っているか?
boolean modifiedp(Object *p);
// p の値を返し、p の値を s 加算することを不可分に行なう
Object *fetch_and_add(Object *p, int s);
// lock ビットを使用して p の指すオブジェクトをロックする
void lock(Object *p);
// p の指すオブジェクトのロックを解除する
void unlock(Object *p);

```

参考文献

- [AEL88] Andrew W. Appel, John R. Ellis, and Kai Lie. Real-time concurrent collection on stock multiprocessors. *Proceedings of the SIGPLAN' 88*, pages 11–20, June 1988.
- [App89] A. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper*, 19(2):171–183, 1989.
- [Bak78] H.G. Baker. List-processing in real time on a serial computer. *Comm. of ACM*, 21(4):280–294, April 1978.
- [Daw82] Jeffrey L. Dawson. Improved effectiveness from a real-time LISP garbage collector. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 159–167, Pittsburgh, Pennsylvania, August 1982. ACM Press.
- [FY69] R. Fenichel and J. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Comm. ACM*, 12(11):611–612, November 1969.
- [IT93] Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. 4(9):1030–1040, September 1993.
- [LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.
- [RHH85] Jr. Robert H. Halstrad. Multilisp: A language for concurrent symbolic computation. *ACM Transaction on Programming Languages and Systems*, 7(4):501–538, October 1985.
- [Ung84] D. Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. *SIGPLAN Notices*, 19(5):157–167, 1984.
- [Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.