Incremental Garbage Collection II

BAKKALAUREATSARBEIT
(*Seminar aus Softwareentwicklung: Garbage Collection*)

zur Erlangung des akademischen Grades

Bakkalaureus/Bakkalaurea der technischen Wissenschaften

in der Studienrichtung

INFORMATIK

Eingereicht von:
SCHATZ Roland, 0355521

Angefertigt am:
Institut für Systemsoftware

Betreuung:
Prof. Dr. Hanspeter Mössenböck

Naarn, Jänner 2006

Incremental Garbage Collection II

Roland Schatz

***Abstract*— In the past, garbage collection was often dismissed for performance critical applications as not being able to provide adequate performance, especially for real-time applications.**

But incremental garbage collection algorithms are quite capable of providing real time performance at acceptable latencies. There is also the possibility of providing hardware support to further reduce latency.

There are also approaches for using garbage collection on embedded systems, where it is important not to waste too much memory on fragmentation.

This paper is the second part of a three paper series intending to give an introduction to different approaches to real-time memory management through incremental garbage collection algorithms.

Building on the basic algorithms presented in the first part of this series, this paper introduces some advanced algorithms to improve the performance, make the algorithms more appropriate for uncooperative environments and reduce memory usage, making real-time garbage collection feasible for embedded systems.

There is also a short chapter about hardware supported garbage collection.

***Index Terms*— Garbage collection, real-time, embedded systems, Treadmill, operating systems**

I. Introduction

THIS paper is intended to give an overview over real-time incremental and concurrent garbage collection. While in the past garbage collection was often thought of not being capable of giving convincing real-time performance, Baker's copying garbage collection algorithm [1] proved that it is possible to implement a real-time garbage collector.

This paper is the second part of a three paper series.

In the first part, Christian Wirth describes the basic ideas behind incremental garbage collection and gives a general introduction into the area [2].

He then presents some algorithms, most important Baker's incremental copying collector.

This paper assumes that the reader has read Wirth's paper and is familiar with the basic idea behind Baker's algorithm.

First the Appel-Ellis-Li garbage collector is presented. It builds on top of Baker's algorithm and improves it by utilizing page level protection hardware. This reduces the cost of the read barrier sacrificing real-time performance. Unlike Baker's algorithm, it requires no support from the compiler, but it requires a minor modification to the operating system kernel.

Then a short introduction about replicating collectors is given. They keep the original object around and let the mutator access the original object instead of the copy, synchronizing changes to the object later. This has the advantage of needing only a write barrier instead of a more expensive read barrier.

After that a new approach for synchronizing between multiple concurrent threads is given. Instead of using locking mechanisms, the Doligez-Leroy-Gonthier collectors try to seperate the heap into seperate regions for each thread, avoiding synchronization completely when accessing these regions.

Then a non-copying garbage collector, Baker's Treadmill, is introduced. It has most characteristics of Baker's original copying collector, but it operates without moving objects. Because of this it is possible to relax some restrictions without loosing consistency, for example using a write barrier instead of a read barrier.

Basically Baker's Treadmill has good performance, but produces much fragmentation. The next algorithm presented takes the Treadmill algorithm as base and improves its memory usage and also its performance by reducing fragmentation, making the algorithm feasible for operating system kernels, as demonstrated with the SPIN operating system, and also for embedded systems, where memory usage is still an important factor.

Finally there is a short chapter introducing some ideas how incremental garbage collection algorithms might be supported by specialized hardware while still staying compatible to common general purpose architectures.

In the last part of this series, Thomas Würthinger presents the Train algorithm, which combines the concepts of generational garbage collection with real-time incremental garbage collection [3].

II. The Appel-Ellis-Li Collector

Baker's copying collector [1], [2] is real-time, but it is not concurrent. When the collector does some work, the mutators must be stopped. It also requires special hardware support to implement the read trap with acceptable performance.

Appel, Ellis and Li improve this algorithm [4]. Their garbage collector is copying and concurrent, and it requires neither special hardware nor any modification to the compiler.

Their collector uses the same memory layout as Baker's collector (see Fig. 1):

- Black objects are already scanned. They can only contain pointers to tospace.
- Gray objects are copied but not scanned. They may still contain pointers to fromspace.
- White objects are still in fromspace. At the end of a collection cycle they are garbage and their storage is reclaimed.

The collector must make sure that the mutator only sees tospace pointers. Baker's algorithm simply intercepts every pointer read operation and makes sure the mutator never sees a white pointer.

The Appel-Ellis-Li collector uses a slightly stricter constraint: The mutator is allowed to see black objects only. Black objects can only contain tospace pointers, so this automatically

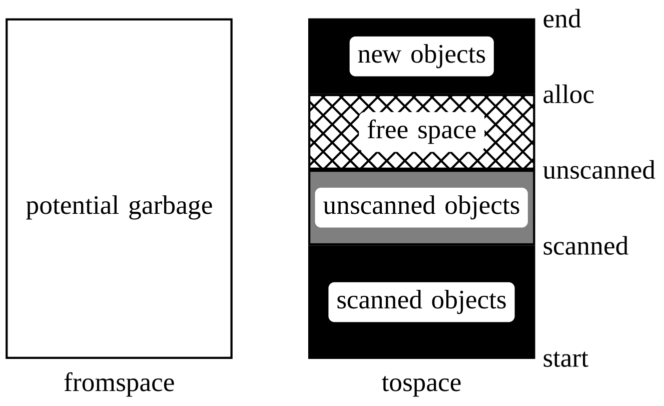

Fig. 1. Memory layout of Appel-Ellis-Li collector.

maintains the constraint that the mutator should never see a fromspace pointer.

Because the mutator can never fetch a fromspace pointer into its registers, newly allocated objects can never contain fromspace pointers, so they can be treated as black without the need to scan them.

A. Implementation

The black-only constraint is implemented using the virtual page protection mechanism of the operating system.

The mutator can never get a pointer to fromspace, so white objects can't be seen anyway. So we can maintain the black-only constraint by setting all tospace pages containing gray objects (gray pages) to no access. When the mutator tries to access a gray page, it triggers a page trap and the collector can scan the page.

The collector uses two threads to scan pages. The trap thread waits for mutator threads that are caught in a page trap, the scanner thread scans gray pages continuously:

```
void TrapThread() {
  for(;;) {
    WaitForTrap(&thread, &page);
    synchronized(lock) {
      ScanPage(page);
    }
    ResumeThread(thread);
  }
}

void ScanThread() {
  for(;;) {
    synchronized(lock) {
      while(scanned >= unscanned) {
        signal(scanFinished);
        wait(lock, unscannedPages);
      }
      ScanPage(scanned);
      scanned += PageSize;
    }
  }
}
```

ScanPage scans a gray page for pointers. All white objects encountered are copied to tospace (“grayed”) and the pointers replaced. Finally the access protection is removed from the page. Now the page contains only black objects.

```
void ScanPage(page) {
  if(!protected(page)) {
    return;
  }
  for(object on page) {
    ScanObject(object);
  }
  unprotect(page);
}

void ScanObject(object) {
  for(pointer in object) {
    pointer = MoveObject(pointer);
  }
}

void *MoveObject(object) {
  if(inTospace(object))
    return object;
  if(object->forwardPtr)
    return object->forwardPtr;

  ret = unscanned;
  unscanned += sizeof(object);
  copy(object, ret);
  object->forwardPtr = ret;
  signal(unscannedPages);
  return ret;
}
```

A problem with this approach is that the collector threads have to be able to access protected pages somehow. A serial implementation could unprotect the page before scanning it, but for a concurrent implementation the page must be protected until every fromspace reference is evacuated.

Most architectures support two execution modes, user- and kernel-mode. If the collector runs in kernel mode and page access is restricted only for user mode code, the page can be scanned while still being protected. This requires only minor modifications to the operating system kernel.

B. Allocation

Moved objects are stored at the bottom of the free space, new objects are allocated at the top of the free space:

```
void *allocate(int size) {
  synchronized(lock) {
    unused = alloc - unscanned;
    if(unused < size || unused < FlipTH) {
      Flip();
```

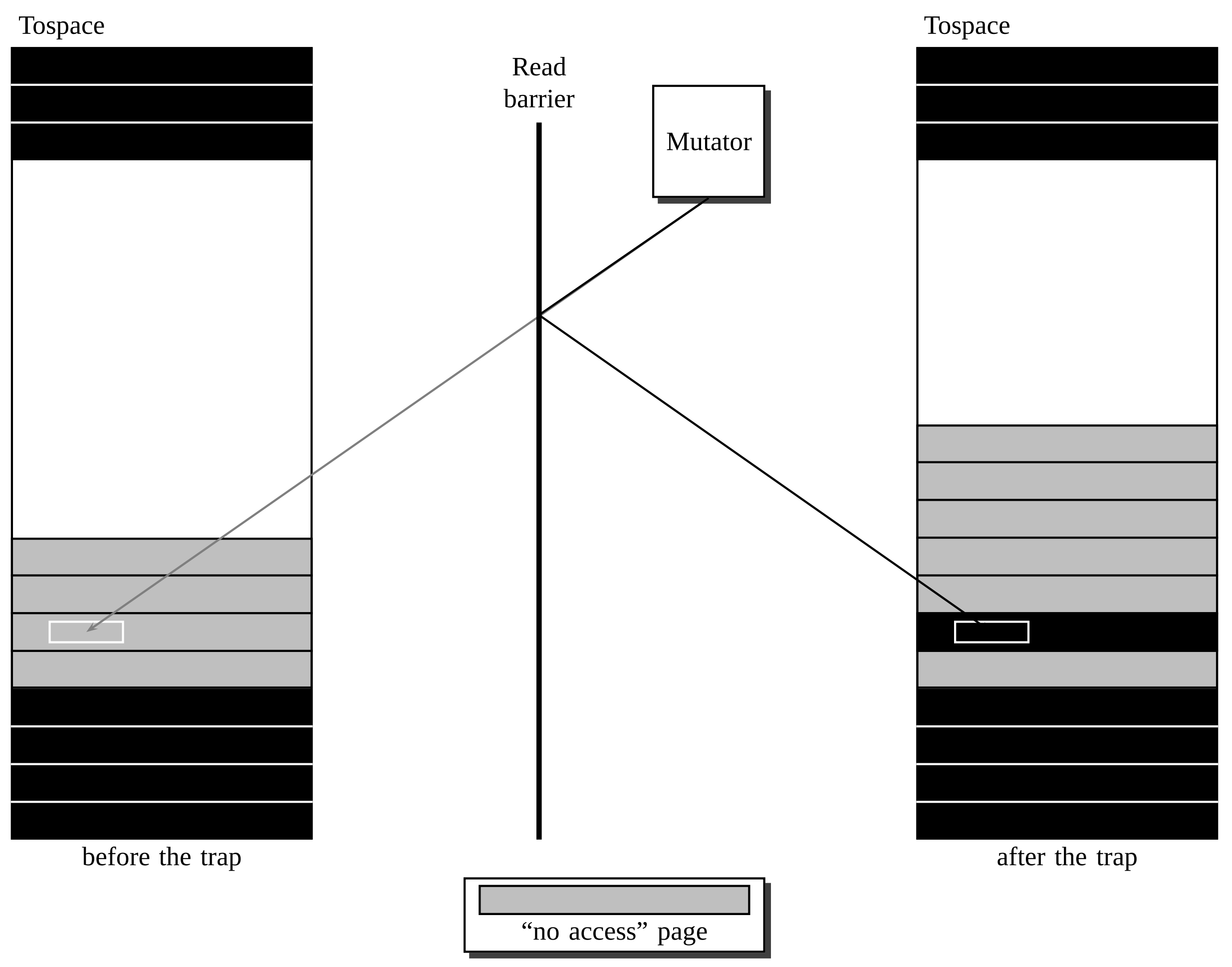

Fig. 2. Appel-Ellis-Li “black-only” read barrier. [5]

```
      }
      alloc -= size;
      return alloc;
    }
}
```

If the free space is too small to satisfy the allocation or below a treshold, a “flip” is triggered. The treshold must be chosen large enough to leave enough space to finish scanning.

One problem with this allocation function is that it needs the same lock as both collector threads, creating a bottleneck. The function can be improved by a two-stage allocator. One stage allocates blocks of memory using the normal allocation function, the other stage allocates objects from these blocks and calls the first stage only when it runs out of space.

C. Flipping

When a flip is triggered, all mutator threads are stopped and the collector waits for the scanner thread to finish scanning all remaining gray objects. At this point, all reachable objects are black and all remaining white objects are garbage.

The collector flips the roles of tospace and fromspace, implicitly reclaiming all white objects. Then all objects directly reachable from roots (registers, stacks, globals) are grayed.

Finally the mutator threads are resumed and the scanner thread is signaled that there are new gray pages to be scanned.

```
void Flip() {
  stopMutators();
  synchronized(lock) {
    while(scanned < unscanned) {
      wait(lock, scanFinished);
    }

    swap(tospace, fromspace);
    scanned = tospace->start;
    unscanned = tospace->start;
    alloc = tospace->end;

    for(pointer in roots) {
      pointer = MoveObject(pointer);
    }

    signal(unscannedPages);
  }
  resumeMutators();
}
```

Because the flip operation has to scan all roots it can have a

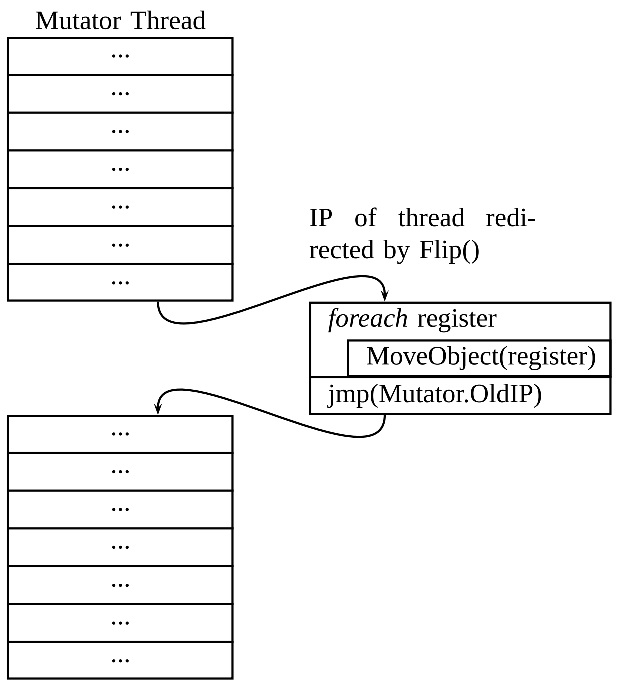

Fig. 3. Threads scanning their own registers.

high latency. For the stacks this can be solved by treating the stacks themselves as gray objects, setting their page protection to no access.

Also scanning all registers can introduce a high latency when there are many threads. But the registers need not be scanned if the current instruction pointer of each thread is redirected to a special routine that scans the registers and then jumps to the old location, so the thread automatically scans its own registers when it is first scheduled (see Fig. 3).

D. Large Objects

Because the algorithm scans pages and not objects, there are some problems when objects cross page boundaries.

If the scanner can easily determine for an arbitrary word if it is a pointer or not, it can just ignore object boundaries entirely. No harm is done when the scanner scans only half an object and resumes with the rest later:

```
void ScanThread() {
  for(;;) {
    synchronized(lock) {
      while(scanned >= unscanned) {
        signal(scanFinished);
        wait(lock, unscannedPages);
      }

      if(isPointer(scanned)) {
        *scanned = MoveObject(*scanned);
      }
      scanned += WordSize;
    }
  }
}
```

But if the scanner needs information from the object header to determine where the pointers are located it can not start scanning in the middle of an object. For small objects the collector could move objects to the start of the next page if they don't fit into the current one.

When objects are too large this wastes too much space, and when they are larger than a page it is impossible to avoid crossing a page boundary. To solve this, the Appel-Ellis-Li collector uses a crossing map.

Crossing[p] is true if an object crosses the boundary between the pages p-1 and p. When a page with crossing=true should be scanned, the scanner has to skip back to the first page with crossing=false. This page starts with the start of an object. It then scans all pages until another page with crossing=false is encountered.

```
void ScanPage(page) {
  if(!protected(page)) {
    return;
  }

  Page *p = page, *end = page;
  while(Crossing[p]) {
    p--;
  }
  while(Crossing[end]) {
    end++;
  }

  for(object on pages[p..end-1]) {
    ScanObject(object);
  }
  while(p < end) {
    unprotect(p);
    p++;
  }
}
```

III. Replicating Copying Collectors

A disadvantage of using a read barrier is that it places a small overhead on every pointer operation. The following collectors try to avoid this.

A. Nettles' replicating collector

Nettles and O'Toole developed a copying garbage collector that does not use an expensive read barrier [6].

Their algorithm leaves the original objects in fromspace and allows the mutator to access them (see Fig. 4). This requires an additional word in the object header for the forwarding pointer, because the object itself must not be destroyed.

Every change that happens to an object that was already copied to tospace has to be stored in a mutation log and applied to the tospace object as well before the flip. When flipping fromspace and tospace, the mutation log is applied, then the roots are changed to point to the tospace objects and the collection cycle is complete.

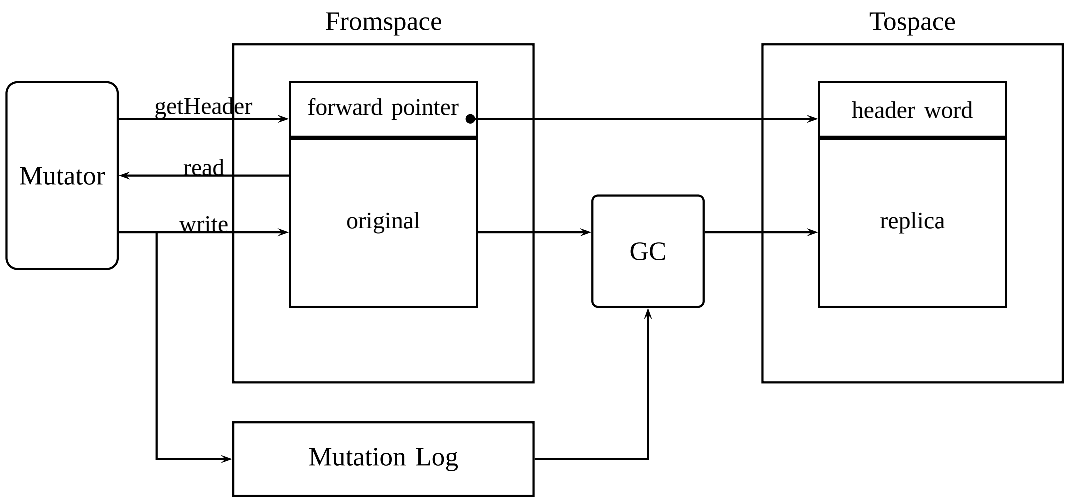

Fig. 4. Replicating garbage collection. [5]

The mutation log can be maintained with a write barrier. The cost of this depends on the language used. For functional languages destructive write operations are rare, so the cost is very low.

It's important to note that the write barrier of this algorithm must catch all writes, while the read barrier of the previous algorithms only need to catch pointer reads.

B. The Huelsbergen and Larus collector

When a language distinguishes mutable and immutable data, the collector can be further improved.

Huelsbergen and Laurus exploit the fact that most data in the ML language is immutable [7].

For immutable data, the mutator is allowed to access either the fromspace or the tospace copy. Obviously no mutation log is neccesary.

For mutable data that has already been copied, only the tospace version may be used. The mutator has to follow the forwarding pointer when accessing a fromspace object. When the object is copied while the mutator is accessing it, all changes have to be reapplied to the tospace copy.

IV. The Doligez-Leroy-Gonthier Collectors

A disadvantage of a write barrier is that it still places some overhead on all writing mutator operations. Also all previous algorithms make heavy use of synchronisation between threads.

Doligez and Leroy [8] divide the heap into two generations (see Fig. 5).

The older generation is called major heap. Mutable objects are always allocated in the major heap. In addition to that, every thread has its own minor heap that is only accessible from inside the thread.

The thread stack and the registers may hold references to the minor heap and to the major heap. Words in the minor heap may also hold references to both heaps, but words on the major heap and global variables may only hold references to the major heap.

If an attempt is made to write a reference to the minor heap into a mutable object (always located on the major heap), the referenced object and all objects referenced by it have to be replicated on the major heap. Since only immutable objects are stored on the minor heaps, no consistency problems arise.

The minor collections use copying collection on a single minor heap, moving all live objects to the major heap. This stops only the thread whose minor heap is currently collected, no synchronisation is neccesary.

A special thread collects the major heap using a simple mark-and-sweep algorithm.

The advantage of the Doligez-Leroy collector is that it places no overhead on pointer operations accessing data that is local to the thread.

A. Synchronising major collections

The algorithm was further improved by Doligez and Gonthier [9]. They try to minimize the amount of synchronization required on major collections.

Instead of using locks, the synchronisation is done cooperatively with a complex phase protocol.

Each thread has a phase variable, initially set to *Async*. The mutator threads have to cooperate by periodically monitoring the phase of the collector thread.

When the collector starts a cycle, it advances to *Sync1*, signaling the mutators that it is about to start garbage collection. The mutators acknowledge this by moving to *Sync1*. When all mutators have done this, the collector advances to *Sync2*.

Now the mutators have to make sure all update operations that are in progress are complete, and then advance to *Sync2*. Once this is complete, the collector can go back to *Async* again, signaling the mutator threads they should shade their roots before going into *Async* themselves.

During the *Sync* phases the write barrier is very conservative, both the old and the new value of each written pointer

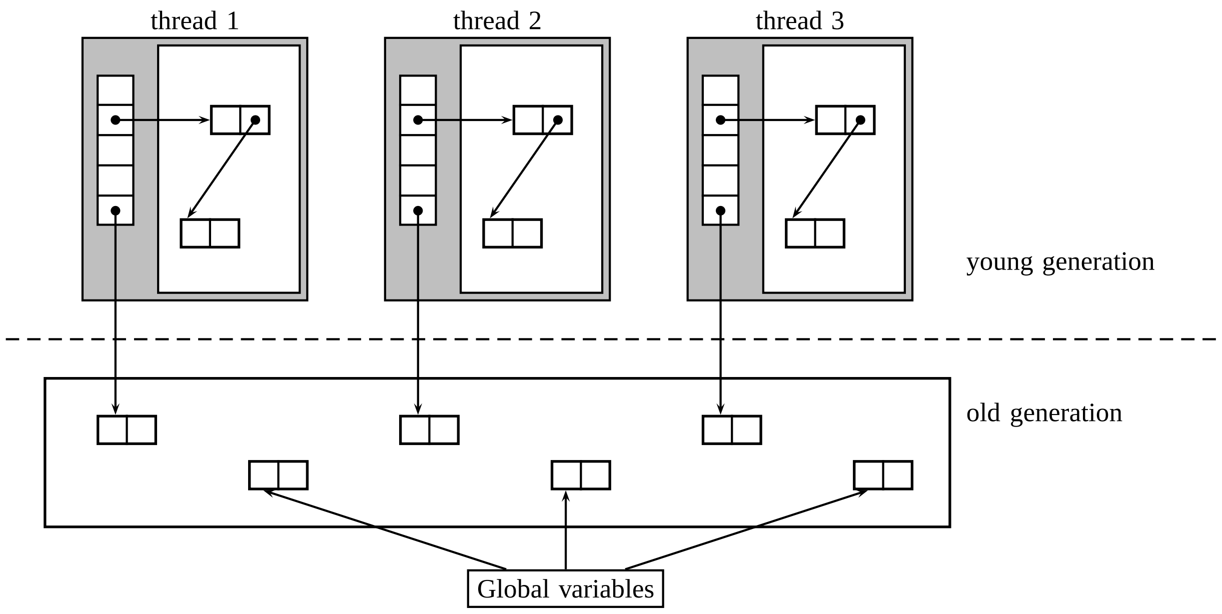

Fig. 5. Thread local heaps. [5]

have to be shaded. During the *Async* phase only a simple barrier that shades the new value is needed.

V. In-place Garbage Collection revisited

Most algorithms presented so far are based on copying collectors, they copy live objects from one part of the heap (fromspace) into another part (tospace), and then reclaim the whole fromspace atomically.

This has the advantage of improving the locality of reference and compacting the memory, effectively eliminating external memory fragmentation. It also is relatively simple, memory allocation is just a pointer increment (or decrement) and all free space can be reclaimed implicitly in constant time. It is also implicitly incremental.

But inplace collectors are better suited to uncooperative environments where we have no means of reliably identifying pointers. In some environments we might get false positives when trying to determine wheather a value is a pointer or an integer. With copying collectors this error can be fatal, while with in-place collectors it makes the collector just a little bit more conservative.

It's also easier to ensure consistency when not moving objects, since the mutator does not need to be protected from changes made by the collector, and there are never two copies of a particular object around that could run out of sync.

A. Basic Garbage Collection

Lets first revisit how a basic garbage collector works. Memory on the heap can be seperated in four sets:

- scanned objects (black)
- visited but unscanned objects (gray)
- not yet visited objects (white)
- free space

Generally speaking, a collection cycle starts with all memory that is not free as white and the roots gray. It then repeatedly scans a gray object, making referred white objects gray and the scanned object black. When there are no more gray objects, the cycle is complete and the remaining white objects may be freed.

Obviously a simple stop-the-world mark-and-sweep collector uses exactly this strategy. But the two semispaces used by the original Baker's algorithm [1], [2] (and most other copying collectors) may be seen just as another way to implement these four sets, as already seen on Fig. 1 in Section II.

B. Requirements for Efficient Collection

There are only a few requirements that a data structure must satisfy in order to efficiently implement this tri-color-marking algorithm on top of it [10]:

1) it is easy to enumerate free cells (fast allocation)
2) it is easy to enumerate gray cells (efficient marking)
3) it is easy to determine the color of a cell
4) it is easy to change the color of a cell
5) it is easy to interchange the interpretation of white, black and free (fast memory reclamation)

Obviously the semispace organisation of Fig. 1 fulfils everything except perhaps point 4. Changing from white to gray requires a copy operation. This can still be seen as a constant time operation and suitable for real-time when objects are reasonably small, but it may be arbirary expensive when there is no upper bound to the object size.

C. A simple Data Structure

There is another simple data structure that satisfies all these points: Double-linked lists [11].

The most simple implementation consists of just three double-linked lists, one for each color. Allocation is simply moving an object from the free-list to the white-list. When the free list is empty, all objects are on the white-list.

All mutators are stopped and the collection cycle runs, moving the roots to the gray-list. Then all gray objects are scanned, moving them to the black list. When the gray list is empty, the marking terminates. This leaves all objects either in the white list (unreachable) or the black list (reachable). Then the white and black lists are exchanged, and the black list is reinterpreted as free list, implicitly reclaiming all unreachable objects.

```
void collect() {
  stopMutators();

  for(root in rootset) {
    white->remove(root);
    gray->insert(root);
  }

  for(object in gray) {
    gray->remove(object);
    for(pointer in object) {
      if(white->contains(*pointer)) {
        white->remove(*pointer);
        gray->insert(*pointer);
      }
    }
    black->insert(object);
  }

  swap(black, white);
  resumeMutators();
}

Object allocate() {
  // black is interpreted as free
  Object ret = black->removeFirst();
  white->insert(ret);
  return ret;
}
```

All basic operations of this algorithm consist only of moving an object from one list to another and the exchanging of two lists. These are all constant time operations, but of course this algorithm is neither incremental nor concurrent.

D. An Incremental Algorithm

This algorithm can be implemented as realtime algorithm by interleaving marking and mutator operations the same way as in the old Baker's algorithm [1], [2].

For this to work we have to introduce a fourth color, called *dead-white,* distinguishing free objects from not yet marked (white) objects[1].

When the mutator wants to access a white object, it first has to move it to the gray list, indicating it should be scanned by the collector at some later time. This can be implemented using a simple read barrier. This way consistency is ensured.

```
void doSomeWork() {
  if(gray->empty())
    return;

  Object object = gray->removeFirst();
  for(pointer in object) {
    if(white->contains(*pointer)) {
      white->remove(*pointer);
      gray->insert(*pointer);
    }
  }
  black->insert(object);
}

Object allocate() {
  if(deadwhite->empty()) {
    Flip();
  }

  Object ret = deadwhite->removeFirst();
  black->insert(ret);
  return ret;
}

void Flip() {
  stopMutators();

  while(!gray->empty()) {
    doSomeWork();
  }

  deadwhite = white;
  white = black;
  black = empty;

  for(root in rootset) {
    white->remove(root);
    gray->insert(root);
  }

  resumeMutators();
}

void readBarrier(object) {
  if(white->contains(object)) {
    white->remove(object);
    gray->insert(object);
  }
}
```

[1]In [10], Baker called the free-list *white* and the not yet marked objects *off-white.* I have changed the colors to be consistent with earlier usage, because in Section II *white* refers to not yet marked fromspace objects.

The algorithm uses only constant time operations, so it should be obvious that this algorithm is realtime if and only if the original copying collector it is based on is realtime.

E. Concurrency

Although the previous pseudocode sample is for an incremental algorithm analogous to Baker's copying collector, with doSomeWork being called a few times at each allocation, it should be no problem to make the algorithm concurrent, analogous to the Appel-Ellis-Li collector presented in Section II.

The same argument is equally valid for all following pseudocode samples.

Because the following algorithms are all based on the simple four-list non-moving collector, they can all be implemented either incremental and concurrent. The pseudocode samples are given for the incremental variant because they are simpler and easier to read.

VI. Baker's Treadmill Collector

Baker's Treadmill is a data structure for garbage collection that satisfies all of the requirements mentioned in the previous section. The Treadmill collector is a real-time non-moving collector that still retains some of the advantages and the simplicity provided by copying collectors.

A. General Structure

The algorithm presented in the previous section can be further optimized by linking all four lists together into a single cyclic double-linked list (called Treadmill) [10]. The original lists are in consecutive segments, seperated by four pointers, in the following order:

- *bottom*
- white
- *top*
- gray
- *scan*
- black
- *free*
- dead-white
- *bottom*

These pointers are the same as in Baker's incremental copying collector, except that the white objects are neither adjacent to top nor to bottom, but in a seperate region of the heap.

Fig. 6 illustrates how the Treadmill might look like in the middle of a garbage collection cycle.

B. Allocation

Allocation is as simple as pushing the *free* pointer one object counter-clockwise, making the newly allocated object black.

```
Object allocate() {
  if(free == bottom) {
    Flip();
  }
  Object ret = free;
  free = free->cw;
  return ret;
}
```

This is a constant time operation, and obviously a lot cheaper than having to manipulate a free object list.

C. Marking

The marker always scans the object pointed to by the *scan* pointer and then pushes the *scan* pointer one object clockwise, making the scanned object black.

When it encounters a pointer to a white object while scanning, the object is unlinked and inserted into the gray segment. This is the only relink operation neccesary. For this we need to know weather or not an object is white, so we need a single color bit for each object that tells us wheather or not the object is white.

```
void doSomeWork() {
  if(scan == top)
    return;

  for(pointer in scan) {
    if(pointer->isWhite()) {
      unlink(*pointer);
      pointer->clearWhite();
      // insert object cw after top
      insert(*pointer, top);
    }
  }

  scan = scan->ccw;
}
```

D. Scanning Order

The object can be inserted either at the *top* pointer or at the *scan* pointer. When inserting at the *top* pointer, the scanning order is breath-first. This is identical to a normal copying collector. When inserting at the *scan* pointer, the scanning order is depth-first.

Depth-first scanning seems to cause fewer page faults and cache misses [5].

Note that there is no marking stack neccesary to scan in depth-first order. One could argue that the links in the Treadmill are used as a stack, and occupy the space permanently instead of only during the collections. But as we will later see, the Treadmill links don't use any more space than copying collectors.

E. Flipping

When the *scan* pointer meets the *top* pointer, there are no more gray objects and the garbage collection cycle is complete. When the *free* pointer meets the *bottom* pointer, the free list is empty and we have to flip.

Before flipping, we first have to make sure scanning is already complete.

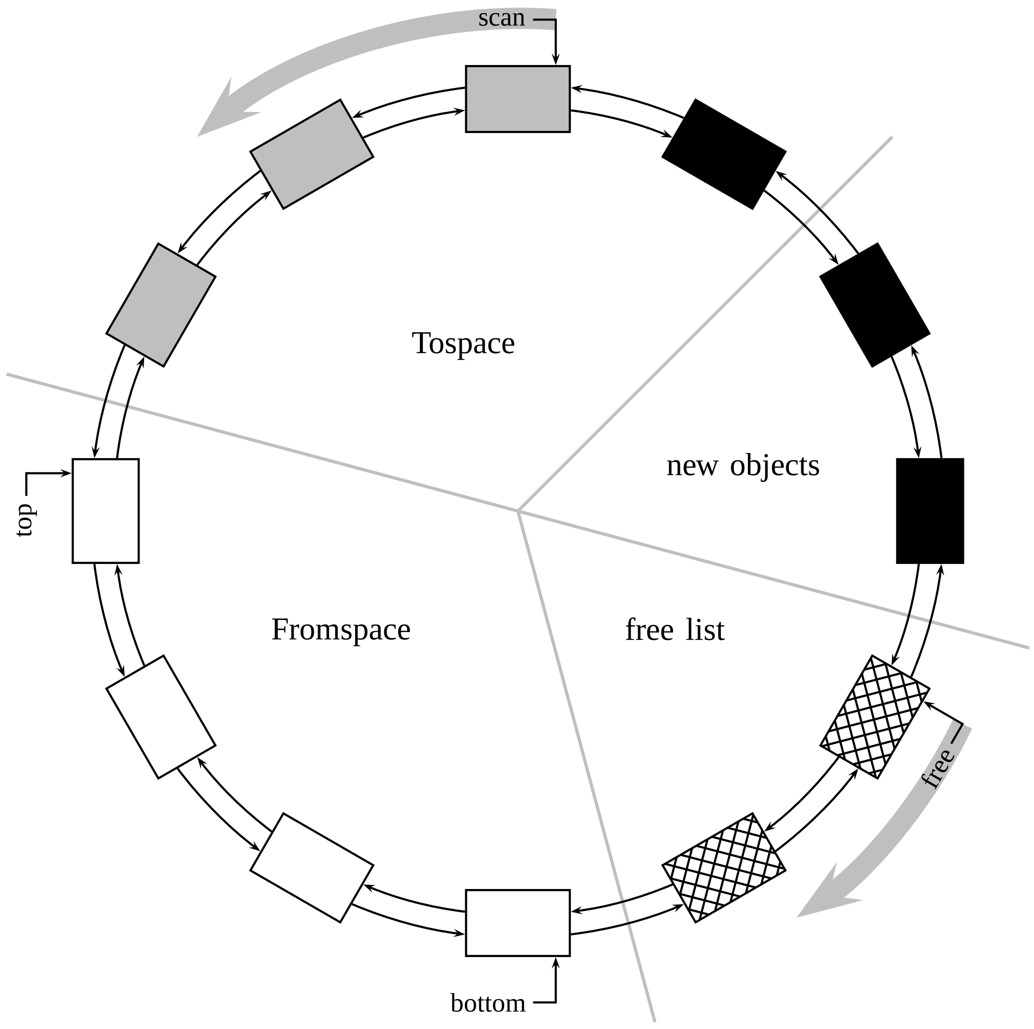

Fig. 6. Baker's Treadmill. [5]

At this point there are only black and white objects in the Treadmill. Now we have to reinterprete the black objects as white and the white objects as dead-white. This is simply archieved by advancing the segments of the Treadmill, that is, swapping the *top* and *bottom* pointers.

Now we have just white and dead-white objects. Finally we set the *scan* pointer to the top pointer. Then we can begin with scanning the root pointers, inserting the objects pointed to by them between *top* and *scan*, making them gray. The next garbage collection cycle can begin by blackening these gray objects.

```
void Flip() {
  while(scan != top)
    doSomeWork();

  bottom = top->cw;
  top = free->ccw;

  for(root in rootset) {
    unlink(root);
    insert(root, top);
  }

  scan = free->ccw;
}
```

Note that we normally would have to change the white bit of every object in the Treadmill. This can be avoided by implizitly exchanging the interpretation of set and clear white bits after every flip.

F. Performance Comparisons

Compared to simple double-linked lists, the Treadmill has the advantage that the color changes from dead-white to black and from gray to black are simply done by moving a pointer forward in a list, instead of relinking an object. That is a single pointer write compared to about six pointer writes.

We still have to relink an object when changing its color from white to gray.

Compared to the incremental copying algorithm [1], [2], the Treadmill requires 2 additional pointers per object for the links in the double-linked list. This is offset by the fact that it does not require a seperate tospace. A CONS pair in Lisp is exactly 2 pointers large, so the space requirements are equal, but for larger objects the Treadmill requires less space.

Also the cost of copying a CONS pair to tospace is lower than relinking an object from the white segment to the gray segment, but thats only because a CONS pair is always only two words large. As the object size gets larger, the cost of copying it increases linearly, making it a non-constant operation for unbounded object sizes, while relinking of an object is independant of its size.

The cost of allocation is also higher than simply incrementing a pointer, but it is still less than unlinking something from a linked list, scanning a free memory bitmap or searching a free list.

The same reasoning holds for making a scanned object black and advancing to the next gray object, but this advantage is lost when the copying collector tries to implement depth-first scanning.

G. Using a Write Barrier

In [10], Baker assumes that the synchronisation between the collector and the mutator is done with a read barrier, just like in his copying collector.

The reason why Baker's copying collector used a read barrier was to protect the mutator from the changes to the object locations by the collector when objects are moved into tospace. But the Treadmill collector does not move objects while scanning.

There is no reason why the Treadmill needs a read barrier [5]. We could just use a write barrier that grays the new target of a modified pointer if it was originally white, or re-gray the black object, depending on our overall strategy.

Both operations need just an object relinked into a different segment of the Treadmill, this requires constant time. Also a write barrier is much cheaper than a read barrier.

H. Heterogenous Objects

A big problem with the Treadmill is that it assumes all objects are of the same size. This may be a perfectly valid assumption for many functional languages, but most other languages require quite a big spectrum of different object sizes.

The most straightforward solution appears to just ignore the problem, putting different sized objects into the Treadmill.

But this causes the problem that it is no longer possible to take just the first object in the free list, it has to be searched for an object big enough to satisfy the request, the object may have to be split and later adjacent free objects may have to be merged again in order to be able to satisfy big requests.

There are several solutions to this problem that can find a fitting free space in logarithmic time, but of course this is no longer real-time.

A tradeoff would be to round up the size of the objects to the next power of two and using seperate Treadmills for each object size [12]. This eliminates the need to search a free list, but produces fragmentation.

It also means that we have multiple free lists, so they don't become empty simultaniously. Because of this we have to recolor free objects explicitly, but this can be done lazily [5].

I. Memory Fragmentation

There is also the problem of memory fragmentation. In general every garbage collector produces some memory fragmentation, but for non-moving collectors it can easily get out of control.

There are two kinds of memory fragmentation [13]:

- *Internal fragmentation* is memory wasted by the algorithm, for example by header fields or rounding up object sizes. This memory can't be used by the mutator. It is usually very easy to give an upper bound on the memory lost by internal fragmentation.
- *External fragmentation* is memory that is free in theory, but it can't be used because it is too small to satisfy allocation requests. This memory can be used by the mutator, but only for small enough allocations. When the mutator does not need that much small objects, the memory is effectively lost.

In general it is not possible to give an upper bound on the memory lost to external fragmentation. Depending on the usage statistics it can grow arbitrarily large.

The problem of external fragmentation is always there when we have to deal with immobile objects of different sizes, and in general it can not be solved without moving objects [10].

The next section tries to optimize the Treadmill algorithm mainly by reducing fragmentation.

VII. The Treadmill on Embedded Systems

Lim, Pardyak and Bershad have implemented a garbage collector that is optimized for embedded systems [13].

For evaluating algorithms they used three performance metrics:

- latency
- overhead
- memory utilization

Latency is the length of mutator pauses caused by the collector or allocator.

Overhead is the total time spend collecting or allocating memory in relation to the total execution time.

Memory utilization is the percentage of memory actually usable by the mutator in relation to the total heap size.

Usually it is very easy to optimize one of these metrics. Most algorithms sacrifice higher memory utilization for lower latency or overhead. Incremental garbage collection reduces latency at the cost of a slightly higher overhead.

On desktop computers we can usually ignore memory utilization as long as we stay within reasonable bounds, because memory is cheap. But since memory is usually a very limited resource on embedded systems, the primary design goal of their algorithm was to minimize memory utilization without sacrificing overhead or real-time latency.

The collector is based on the Treadmill algorithm, since this algorithm is already optimized for real-time latency and low overhead by sacrificing memory utilization.

A. Overall Strategy

Known improvements to the Treadmill algorithm like a buddy allocator would produce a higher overhead or compromise it's real-time latencies.

The overall strategy of the algorithm is [13]:

- large objects are allocated from a seperate pool and aligned to pages to limit internal fragmentation
- free pages are located via a page-wise collection to enable fast merging, migration and remapping
- free pages are migrated between lists to eliminate page-level external fragmentation
- free pages are remapped into continuous ranges for large allocations
- page filling allocation directs allocations to under-utilized pages

These page-level optimizations can all be performed efficiently in constant time, so they do not compromise real-time latencies.

To counter memory waste by objects smaller than the page size, they allow arbitrary free-list sizes. Lim also compacts the object header, further reducing internal fragmentation.

B. Performance

Lim tested the garbage collector both in the SPIN operating system kernel and in the context of userspace applications. In all benchmarks the improved garbage collector used 25% to 60% less memory than the unimproved Treadmill collector.

In userspace applications they managed to further reduce the memory usage by 18% by implementing arbitrary free-list sizes and compacting the headers (see Section VII-I).

The latency of the algorithm is at most 15ms. It's important to note that this is a soft real-time bound. For any garbage collector implementation there always exists an allocation request that requires an unbounded amount of time to complete.

If the application requests more memory than is available when the collector has not yet finished marking, a full collection is forced. In the worst case the whole heap has to be scanned before the allocation request can be fulfilled.

C. The Segregated Treadmill Algorithm

With the Treadmill, allocation time is constant because all objects have the same size. This restriction can be relaxed by using segregated free lists for different object sizes, increasing in power of two steps [12]. Each free list is managed by a seperate Treadmill (see Fig. 7).

While having real-time latencies and a low overhead, the segregated Treadmill algorithm has poor memory utilization.

The worst case internal fragmentation is 50%. The external fragmentation can get extremely high because memory blocks are never merged to fulfill larger requests or split to fulfill smaller ones. Once comitted to a particular free list, pages are never moved to another one.

Because of excecllent time characteristics, Lim chose this segregated Treadmill algorithm as a starting point for developing their optimized collector.

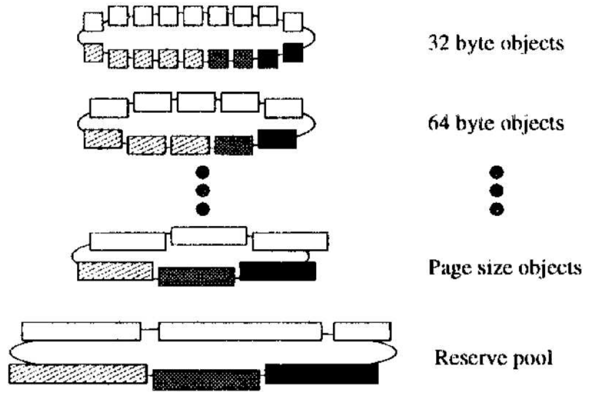

Fig. 7. Seperate Treadmills for small objects. [13]

D. Improving Memory Utilization

Coalescing memory to reduce internal fragmentation is hard with the Treadmill. It reclaims memory implicitly in constant time, so it is hard to identify free objects, and without that information it is hard to come up with global strategies to reduce fragmentation.

In the new algorithm, free memory information is maintained at the page level. For example, if all objects on a page are free, the page can be assigned to a different Treadmill. All computations are done on a limited number of pages per call in order to maintain real-time bounds.

All techniques presented try to improve peak memory utilization, that is, the amount of memory that a collector can allocate before having to complete a collection cycle. That is, for a given workload and collection frequency the collector needs a smaller heap, or more important, for a given heap size the collector can provide more memory before it has to reclaim garbage.

While in general reducing peak memory utilization increases latency and overhead of a single collection cycle, it also decreases the frequency of collection cycles, so actually the overhead is massively reduced.

The techniques presented operate on two layers. As already mentioned, the base layer operates on the page level and drastically improves memory utilization.

The second layer reduces internal fragmentation caused by small objects. It builds on top of the base layer to further reduce the memory footprint of workloads with many small allocations.

E. Large Objects

Objects smaller and up to one page in size are allocated from segregated Treadmills (see Fig. 7).

But in addition to the usual 50% worst case internal fragmentation, big objects can produce an extremely high amount of external fragmentation, because once freed their memory is still lost for smaller allocations.

Because of that, objects larger than a single page are allocated in a seperate *big object list*. In the big object list are regions of memory that are broken up into objects of the required size. All objects larger than a page are always allocated page-aligned.

The big object list itself is also managed by a Treadmill, but since there are different sized objects on the list, an allocation requires a search on the free list. This takes linear time, so it cannot be done in real-time. In Section VII-G we will see that allocation from the big object list can still be done in constant time utilizing page level optimizations.

```
Object allocate(int size) {
  if(size > PageSize) {
    return bigTreadmill->allocate(size);
  } else {
    size = roundUp(size);
    return smallTreadmill[size]->alloc();
  }
}

Object BigTreadmill::allocate(int size) {
  // search for big enough object
  Object ret = findObject(free, size);
  unlink(ret);
  insert(ret, free->ccw);
}
```

Because starting from page-sized objects we no longer increase object sizes in power of two steps, but in page-sized steps, the internal fragmentation is always less than one page. This is still 50% for the worst case szenario where every allocation request is one page plus one byte, but for larger objects internal fragmentation drops fast.

F. Page Migration

After taking care of internal fragmentation, we still have to solve the problem of external fragmentation, caused by the segregated Treadmill because it does not move objects from one free-list to another.

With each small object Treadmill we associate a list of pages, seperated into an allocated and a free page list. The pages in these lists contain the objects managed by the Treadmill. When an object is allocated, the page it resides on is moved from the free page list to the allocated page list.

During garbage collection, when an object is grayed, the page it resides on is also marked. At the end of the collection cycle, unmarked pages are moved to the free page list.

```
void SmallTreadmill::ScanObject(object) {
  for(pointer in object) {
    if(pointer->isWhite()) {
      unlink(*pointer);
      insert(*pointer, top);
      pointer->page->mark();
    }
  }

  scan = scan->ccw;
}

void SmallTreadmill::Flip() {
  while(scan != top)
    doSomeWork();

  bottom = top->cw;
  top = free->ccw;

  for(root in rootset) {
    unlink(root);
    insert(root, top);
  }

  for(page in allocatedPages) {
    if(!page->marked) {
      allocatedPages->remove(page);
      freePages->insert(page);
    }
  }

  scan = free->ccw;
}
```

Whenever a Treadmill runs out of free objects, a page from the free list of another Treadmill is reassigned. First the page is unlinked from the free page list, and with it all objects are unlinked from the associated Treadmill.

Then the page is reinitialized for holding objects of the different size. The page is inserted into the free list of the target Treadmill, and its objects are inserted into the free segment of the Treadmill (see Fig. 8).

```
Object SmallTreadmill::allocate() {
  if(free == bottom) {
    // search free page lists
    Page page = findFreePage();
    if(page) {
      for(object on page) {
        // unlink old objects
        unlink(object);
      }
      reinitialize(page, this->objSize);
      for(object on page) {
        // insert into our own free list
        insert(object, free);
      }
    } else {
      Flip();
    }
  }

  Object ret = free;
  free = free->cw;
  return ret;
}
```

Only when a free list is empty *and* there is no free page on any other Treadmill, garbage has to be reclaimed. By deferring the flip as long as possible, both the memory utilization and the overhead of the algorithm are greatly improved.

This algorithm is still real-time. Locating a free page is only bounded by the number of free page lists, which is a

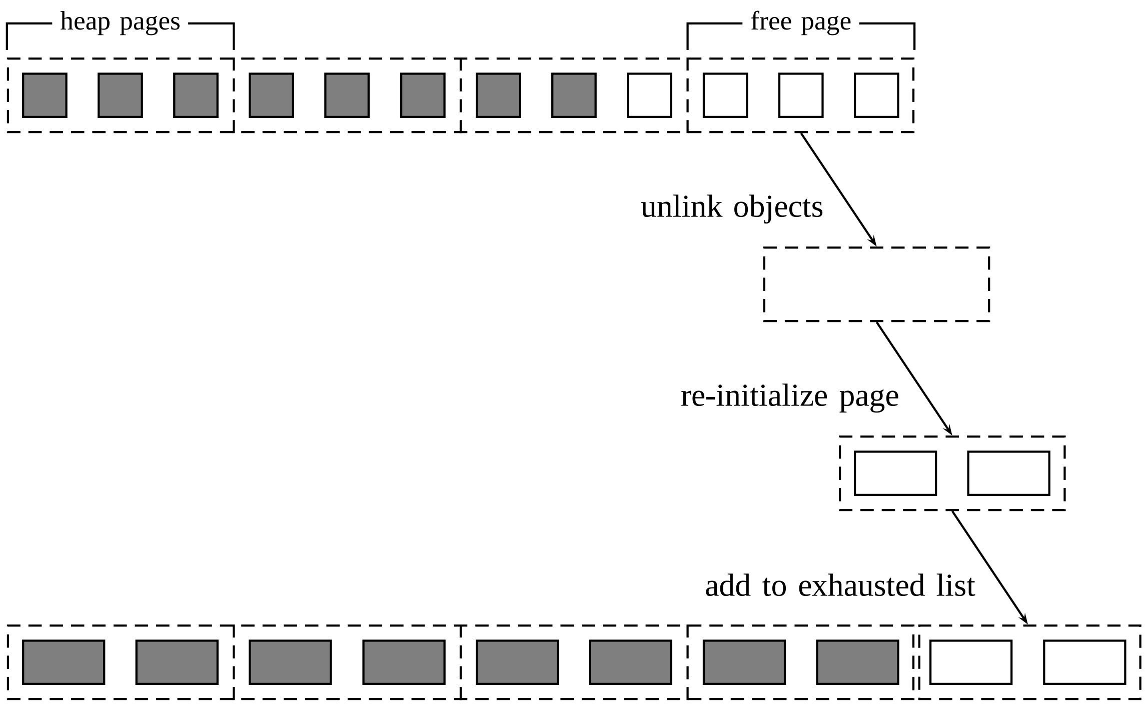

Fig. 8. Reassigning pages to different Treadmills. [13]

small constant. Unlinking the page is bounded by the count of objects previously in the page, reinitializing and linking into the target Treadmill is bounded by the count of new objects that fit in the page. Both are constant.

Reclaiming an arbitrary number of free pages can still be done in constant time by using a seperate page-level Treadmill, but this would increase the cost of allocation, so it was not implemented by Lim.

G. Page Remapping

We still have the problem of external fragmentation in the big object list. Long lived big objects can break up the space, resulting in many single free pages that are no longer usable for big object allocation.

The page migration algorithm of the previous section does not prevent this kind of external fragmentation, because it deals only with single pages. It never merges adjacent free pages to produce enough space for big objects.

Further it appears to be counterproductive to move free pages from the big object list back to the small Treadmills. This would further fragment the memory, using up continous space that can't be used for big object allocations later.

Compacting memory would solve this problem, but this would require moving objects, which conflicts with the goal of producing an in-place garbage collector.

To solve this, we just virtually remap free pages into a continuous range of memory. Whenever an allocation request larger than a page is made that can't be satisfied out of the big object Treadmill, the allocator finds enough free pages and remaps them into a continuous range.

Since now we effectively eliminated page level fragmentation, we also eliminated the problem that we can't move free pages from the big object list back to the small object Treadmills.

The same algorithm can be used for constant time allocation out of the big object list. Instead of searching for an object big enough on the free list, we just remap all free memory of the big object list into a single continuous block and allocate by incrementing a pointer.

The dead-white section of the big object Treadmill can now just be seen as a free page list for the purpose of both big object allocation and small object page migration.

```
void BigTreadmill::growContinuousBlock() {
  Page page = findFreePage();
  page->remap(allocTop);
  allocTop += PageSize;
}

Object BigTreadmill::allocate(int size) {
  while(allocFree + size > allocTop) {
    growContinuousBlock();
  }

  Object ret = allocFree;
  allocFree += size;
  return ret;
}
```

Of course, remapping of large numbers of pages can take unbounded time. But remapping can be done lazily, analogous

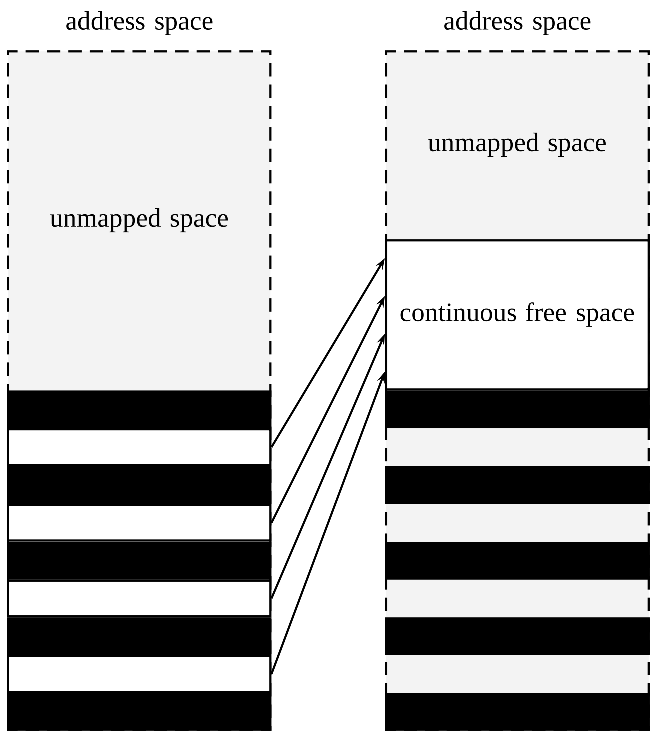

Fig. 9. Page remapping.

to lazy copying. Instead of remapping all pages at once, they are just reserved and remapped incrementally in the background, or when someone actually needs the memory, that is, at page fault time. Initialisation of newly allocated memory can also be deferred to the time when the page is actually remapped.

But since in typical workloads big object list allocations and remapping were rare, Lim did not implement lazy remapping.

H. *Page-filling Allocation*

Our page migration algorithm introduces an additional problem. We now have sub-page external fragmentation.

When an object from a page is allocated, this page is commited to this object size and can not be used by allocations of different sized objects.

In SPIN's workloads this is not a problem because there are some long-lived objects allocated during initialization and then a large number of a wide variety of sizes, so almost all memory that would be wasted over by this kind fragmentation gets used.

For different workloads that allocate only few objects of some sizes, there is a page-filling allocator. It remembers continuous chunks of memory within pages. When memory utilization is too low, the allocator tries to find a page with enough continuous range and assigns part of this page to another Treadmill.

This can be done on demand, so there is only overhead when it is actually needed.

I. *Small Objects*

For very small objects, the internal fragmentation can easily raise above 50%. The big object list reduces the problem for large objects, so for workloads with many large objects this can be ignored.

There are two optimizations for workloads with mainly very small objects.

First we can reduce the internal fragmentation caused by the object headers. The Treadmill algorithm requires two pointers per object in addition to any other header information. The size of these pointers can be reduced by using relative pointers. Objects that are far away from each other can be brought nearer together by page remapping.

With this method we can reduce the header size from 3 words to 1 or 2 words, depending on how much performance we are willing to sacrifice.

Another method is to reduce the internal fragmentation caused by rounding up object sizes by relaxing the power of two constraint. We just determine some set of object sizes at compile time that minimize rounding errors for our particular workload, and then use these object sizes instead of a power of two progression for our small object Treadmills.

VIII. Hardware Support

Software only garbage collection algorithms all have a problem with hard real-time performance. Read-barriers are very expensive and often lead to unpredictable pauses. Virtual memory techniques are even worse, since every operation may potentially cause a page fault.

The best software-only collectors are the Nettles and O'Toole replicating collector and Baker's Treadmill [5].

The Nettles collector has measured worst-case latency times of 50 microseconds, but propably it will performe worse in an environment where write operations are more frequent.

For these reasons it is believed that garbage collectors need hardware support for archieving hard real-time performance [5].

Because general purpose computers that rely on specialised architectures are usually not comercially successful, Nilsen and Schmidt propose a special garbage collected memory module that communicates with the system over a traditional memory bus [14].

For the CPU this memory module would look identical to a normal RAM module (see Fig. 10).

Their garbage collector is basically a variation of Baker's incremental copying collector [1], [2] with additional back-pointers for lazy copying.

The read barrier is performed by the memory module in hardware. If an object that is not yet fully copied is read by the CPU, the memory module automatically copies the object and stalls the CPU until it is ready. The CPU won't notice anything except some latency.

The simulated worst case latency is approximately one microsecond.

List of Figures

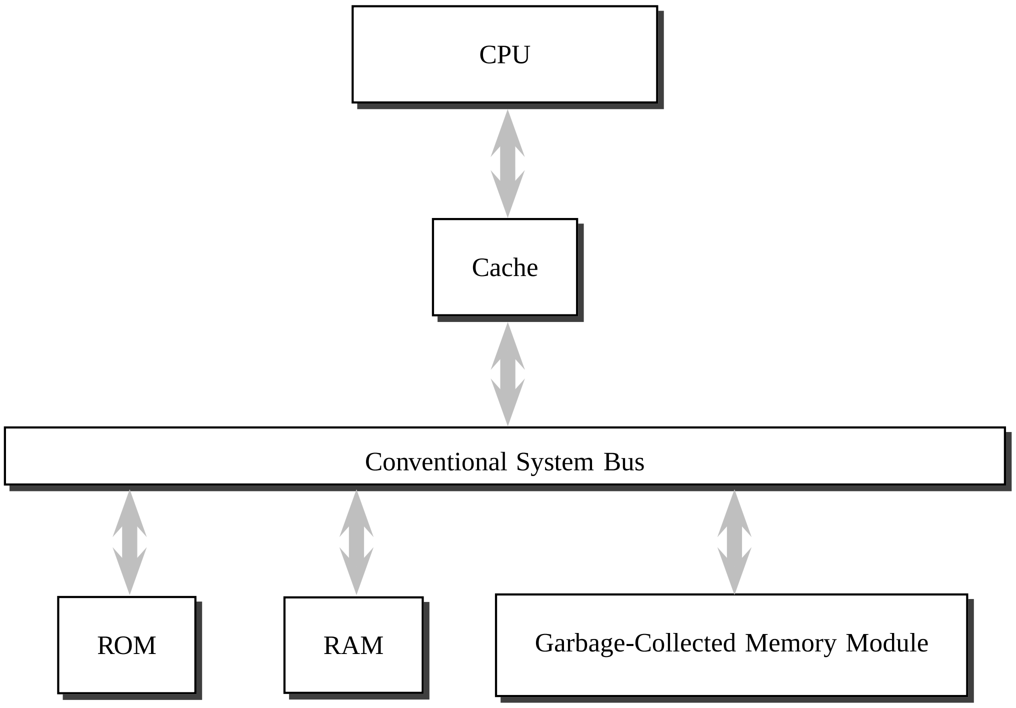

Fig. 10. Nilsens hardware architecture. [14]

REFERENCES

[1] H. G. Baker, “List processing in real time on a serial computer,” *Commun. ACM*, vol. 21, no. 4, pp. 280–294, 1978.

[2] C. Wirth, “Incremental garbage collection I,” in *Seminar aus Softwareentwicklung: Garbage Collection*, 2006.

[3] T. Würthinger, “Incremental garbage collection III,” in *Seminar aus Softwareentwicklung: Garbage Collection*, 2006.

[4] A. W. Appel, J. R. Ellis, and K. Li, “Real-time concurrent collection on stock multiprocessors,” in *PLDI ’88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 11–20.

[5] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1996.

[6] S. Nettles and J. O’Toole, “Real-time replication garbage collection,” in *PLDI ’93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 1993, pp. 217–226.

[7] L. Huelsbergen and J. R. Larus, “A concurrent copying garbage collector for languages that distinguish (im)mutable data,” in *PPOPP ’93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 1993, pp. 73–82.

[8] D. Doligez and X. Leroy, “A concurrent, generational garbage collector for a multithreaded implementation of ml,” in *POPL ’93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1993, pp. 113–123.

[9] D. Doligez and G. Gonthier, “Portable, unobtrusive garbage collection for multiprocessor systems,” in *POPL ’94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1994, pp. 70–83.

[10] H. G. Baker, “The treadmill: real-time garbage collection without motion sickness,” *SIGPLAN Not.*, vol. 27, no. 3, pp. 66–70, 1992.

[11] D. E. Knuth, *The Art of Computer Programming Vol. I: Fundamental Algorithms*, 2nd ed. Reading, MA: Addison-Wesley, 1973.

[12] P. R. Wilson and M. S. Johnstone, “Truly real-time non-copying garbage collection,” in *Proceedings of OOPSLA/EECOOP’93 Workshop on Garbage Collection in Object-Oriented Systems*, 1993.

[13] T. F. Lim, P. Pardyak, and B. N. Bershad, “A memory-efficient real-time non-copying garbage collector,” in *ISMM ’98: Proceedings of the 1st international symposium on Memory management*. New York, NY, USA: ACM Press, 1998, pp. 118–129.

[14] K. D. Nilsen and W. J. Schmidt, “Cost-effective object space management for hardware-assisted real-time garbage collection,” *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 338–354, 1992.