# An Efficient Non-Moving Garbage Collector for Functional Languages

Katsuhiro Ueno    Atsushi Ohori    Toshiaki Otomo [*]

Research Institute of Electrical Communication
Tohoku University
{katsu, ohori, o-toshi}@riec.tohoku.ac.jp

## Abstract

Motivated by developing a memory management system that allows functional languages to seamlessly inter-operate with C, we propose an efficient non-moving garbage collection algorithm based on bitmap marking and report its implementation and performance evaluation.

In our method, the heap consists of sub-heaps $\{H_i \mid c \leq i \leq B\}$ of exponentially increasing allocation sizes ($H_i$ for $2^i$ bytes) and a special sub-heap for exceptionally large objects. Actual space for each sub-heap is dynamically allocated and reclaimed from a pool of fixed size allocation segments. In each allocation segment, the algorithm maintains a bitmap representing the set of live objects. Allocation is done by searching for the next free bit in the bitmap. By adding meta-level bitmaps that summarize the contents of bitmaps hierarchically and maintaining the current bit position in the bitmap hierarchy, the next free bit can be found in a small constant time for most cases, and in $\log_{32}(\mathit{segmentSize})$ time in the worst case on a 32-bit architecture. The collection is done by clearing the bitmaps and tracing live objects. The algorithm can be extended to generational GC by maintaining multiple bitmaps for the same heap space. The proposed method does not require compaction and objects are not moved at all. This property is significant for a functional language to inter-operate with C, and it should also be beneficial in supporting multiple native threads.

The proposed method has been implemented in a full-scale Standard ML compiler. Our benchmark tests show that our non-moving collector performs as efficiently as a generational copying collector designed for functional languages.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—memory management (garbage collection)

***General Terms*** Algorithms, Languages, Performance

***Keywords*** Non-moving Garbage Collection, Bitmap Marking, Generational Collectors, SML#

[*] The third author's current affiliation: Hitachi Advanced Digital, Inc.

## 1. Introduction

The general motivation of this research is to develop a memory management system for a functional language that supports seamless interoperability with the C language. We have partially achieved this goal through type-directed compilation for *natural data representations* in ML [25]. Under this scheme, ML records and arrays as well as atomic types such as `int` and `real` have the same representation as in C and are therefore directly read or updated by a C program without any data representation conversion. This has been embodied in our SML# compiler [33]. In this implementation, one can directly import a C function and call it with data structures such as records, mutable arrays, and function closures (for call-backs) created in SML#. The following code is a fragment of a demo program distributed with SML#.

```
val glNormal3dv =
    dlsym (libgl, "glNormal3dv")
    : _import (real * real * real) -> unit
...
  map (fn (vec, ...) => (glNormal3dv vec; ...))
      [((1.0, 0.0, 0.0), ...), ...]
```

This code dynamically links `glNormal3dv` function in the OpenGL library as an ML function of type `real * real * real -> unit` and uses it with other SML# code. SML# compiles `(1.0, 0.0, 0.0)` to a pointer to a heap block containing 3 double precision floating point numbers. Since this record has the same representation as a `double` array of C, `glNormal3dv` works correctly. In addition to this natural data representation, each SML# heap object has a header containing object layout information indicating whether each field is a pointer or not. This information is used by the SML# garbage collector (GC) to exactly trace live objects. By adopting the strategy that the SML# GC only traces and collects SML# objects and leaves management of C allocated objects to C code, we should be able to achieve seamless interoperability between SML# and C.

The solution so far is however only partial in that data structures that are passed to foreign functions must be allocated in a special non-moving area. SML# compiler, as well as most other functional language compilers, has used copying garbage collection (GC) based on the commonly accepted belief being that, for functional programs requiring large amount of short-lived data, the Cheney's copying collector [8] would be the most efficient for their minor collection. However, any (precise) copying GC requires that the runtime system must be able to locate and update all the pointers to each heap allocated data. This prohibits functional programs from inter-operating with foreign functions or any programs that use local memory space not accessible from the garbage collector. To side-step this problem, the programmer must explicitly request GC not to move those data that are passed to external code. This "object pinning" approach is not only cumbersome but also dangerous. This problem should be painfully familiar to anyone who has tried to write a functional program that interacts with a foreign library that requires callbacks or locally stores object pointers passed from the caller. For a language with rather limited interoperability, "object pinning" might be performed automatically, but for an ML-style language that provides seamless interoperability, automatic "object pinning" is difficult, if not impossible. To see the problem, suppose a C function is called with an array and a call-back function. Since both the C function and the call-back function can freely mutate the array, the runtime system can only safely estimate that the set of reachable objects from the array passed to the C function to be the set of all the live objects in the entire heap, including even those that may be created later by the call-back function.

To solve this problem, we would like to develop a *non-moving* garbage collection algorithm suitable for functional languages. Since functional programs rely heavily on efficient allocation and collection, a new non-moving GC algorithm should ideally be as efficient as currently widely used copying GC with generational extension [21, 35]. The purpose of this paper is to develop such a garbage collection algorithm, to implement it for a full-scale ML compiler, and to evaluate its performance through extensive benchmarks to verify the feasibility of the algorithm.

An obvious non-moving candidate is mark and sweep GC [24]. Compared with copying GC, however, simple mark and sweep GC tends to exhibit the following general weaknesses.

- *Fragmentation and slow allocation.* Functional programs require objects of various sizes with various life time. Due to this property, the heap space becomes fragmented very quickly, resulting in slow allocation and reclamation. To avoid this problem, practical variant of mark-and-sweep GC algorithms sometimes perform costly compaction at sweep phase. In contrast, copying GC automatically performs compaction and yields a very fast allocator, which has only to check the heap boundary and to advance the allocation pointer.
- *High sweep cost.* Mark and sweep GC requires to sweep the heap, which takes time proportional to the size of the heap. Again this is in contrast with copying GC whose collection cost is proportional to the amount of live data.
- *Difficulty in developing efficient generational GC.* While it is straightforward to combine mark and sweep GC with other GC to form a generational GC, developing generational mark and sweep GC requires certain amount of additional machinery, which would result in slower GC time [10].

Perhaps due to these problems, mark and sweep GC seems to be considered not suitable for primary GC (e.g. minor GC in generational GC) of functional programs, which produces large amount of short-lived data of varied size very rapidly.

The main new technical contribution of this paper is to develop a variant of mark and sweep GC that is as efficient as Cheney's copying GC and its generational extensions, and to demonstrate its feasibility for functional languages thorough implementation and evaluation. Our basic strategy is well known *bitmap marking* (see [37] and [17] for a survey.) We associate a heap with bitmaps that represent the set of live objects. This structure alone does not yield an efficient allocation and collection. We have developed a series of new data structures and algorithms that overcome the weakness of mark and sweep GC mentioned above. The following is a summary of the features of our GC algorithm.

1. *A fragmentation-avoided and compaction-free heap.* Fragmentation occurs when objects have varied sizes; if all the objects were of the same size, then the heap could be a fixed size array that can be managed by a bitmap without incurring fragmentation. An extreme idea is to set up a separate heap for each object size. Of course we cannot prepare sufficiently large separate heaps for all possible object sizes, so we prepare a series of sub-heaps $\{H_i \mid c \leq i\}$ of exponentially increasing allocation block sizes, i.e. each $H_i$ consists of allocation blocks of $2^i$ bytes. Actual allocation space of $H_i$ is dynamically maintained as a list of fixed-size *segments*. Each segment contains an array of $2^i$-byte blocks. $2^c$ is the minimum allocation block size in bytes. In SML#, the minimum size of non-empty objects is 8 ($2^3$) byte, so we fix $c$ to be 3 in this paper.

   These structures eliminate the fragmentation problem associated with mark and sweep collection. Since a segment is a fixed size array, it is efficiently maintained by a bitmap where each bit corresponds to one block. Moreover, size of each $H_i$ is dynamically adjusted through allocation and reclamation of segments.

   In our scheme, an allocation block in $H_i$ in general contains an object smaller than $2^i$ bytes, and some amount of memory is unused. Our observation is that, we can avoid costly compaction at the expense of locally wasting space in each allocation block, and that this cost is quite acceptable in practice. Our evaluation shows that the space usage is better than that of copying GC in most cases.

   Since we cannot prepare $H_i$ for unbounded $i$, we set a bound $B$ of $i$ and allocate exceptionally large objects of size more than $2^B$ to a special sub-heap $\mathcal{M}$. $B$ can be as large as the size of one segment. According to our experimentation, $B = 12$ (i.e. at most 4096 bytes) appears to be sufficient for functional programs (e.g. covers most of allocation requests.) Hence, in our system, the heap consists of 10 sub-heaps, $H_3, \ldots H_{12}$. Since the special sub-heap $\mathcal{M}$ occupies a very small portion of the entire heap and can be managed by any non-copying GC method, we do not consider this further.
2. *Efficient allocation.* A bitmap can be used not only for marking but also for allocation by searching for a free bit in a bitmap. Simple search for a free bit in a bitmap takes, in the worst case, the time proportional to the number of busy bits. Perhaps due to this problem, most of bitmap based GC algorithms use some form of free list for allocation. We solve this problem by adding meta-level bitmaps that summarize the contents of bitmaps. The set of all bitmaps form a hierarchically organized tree. The sequence of bits at the leaf level in the tree is the ordinary bitmap representing liveness of the set of allocation blocks, and the sequence of bits at an intermediate level summarizes the bitmap one level below. On this bitmap tree, we maintain a data structure representing both the next bit position to be examined and the corresponding block address. This structure corresponds to the allocation pointer of copying GC, so we call it an *allocation pointer*. The whole organization is depicted in Figure 2 whose details will be explained in Section 2. By constructing a series of optimized searching algorithms on this data structure, the next free bit can be found in a small constant time for most cases, and in $\log_{32}(\textit{segmentSize})$ time in the worst case.
3. *Small GC cost.* Sweeping is done by clearing bitmaps. The total collection cost of a bitmap making GC is the sum of the costs for clearing bitmaps, tracing live objects, and setting the bits in the bitmaps. Among them, the bitmap clearing requires $N/32$ steps where $N$ is the total number of allocation blocks. Our extensive evaluation shows that bitmap clear time is negligible (about 1% of the GC time), and therefore, in practice, the total cost of our GC is dominated by the tracing and marking cost. So in practice our GC algorithm behaves similarly to that of copying GC. In most of the benchmark tests we performed, the

total GC cost was smaller than those of simple copying collector and generational copying collector.

4. *Non-moving generational GC.* By adapting the idea of *partial collection* proposed by Demers et.al. [10], our bitmap marking GC scales up to generational GC without moving objects. This is based on our observation that a bitmap represents a subset of the set of objects in a heap. Generational GC can be realized by maintaining a separate bitmap for each generation. The partial collection with "sticky bit" technique presented in [10] coincides with a special case where tenuring threshold is 1 and the number of generations is 2. Pointers to younger generations from older generations are tracked using a *write barrier* and a *remembered set*. In the above special case, the remembered set can be allocated in the collector's trace stack due to the property that the remembered set can be flushed after minor collection. The resulting implementation of the special case does not require any additional memory space other than those used in the non-generational version.

We have implemented the data structures and algorithms in a Standard ML compiler, and have done extensive benchmark tests. We have evaluated and compared them against a simple Cheney's copying GC, and a generational copying GC for functional languages described in [29], and have obtained the following results: (i) segment-based dynamic sub-heap size adjustment automatically achieves optimal hand-tuned sub-heap size assignment, (ii) the bitmap marking GC is as efficient as Cheney's copying GC, and (iii) the generational extension outperforms the non-generational bitmap GC, and shows comparable or better performance results compared to generational copying GC. These results demonstrate that our development have achieved the goal of developing a non-moving GC method for functional language.

The proposed method has additional advantage of supporting multiple native threads without much additional machinery. Our segment-based heap organization allows each concurrent thread to allocate objects in a shared global heap without any locking. The detailed implementation and evaluation for multithread extension is out of the scope of the present paper, but our preliminary implementation of multithread extension shows promising result.

All the implementation is done in our SML# compiler, which compiles the full set of Standard ML language (with the (mostly) seamless interoperability with C) into x86 native code. SML# version 0.40 or later includes the GC algorithms reported in this paper, and they are available and enabled by default in x86 native compilation mode.

The rest of the paper is organized as follows. Section 2 presents the bitmap marking GC algorithm. Section 3 extends the GC method to generational GC. Section 4 reports implementation and performance evaluation. Section 5 compares the contribution with related works. Section 6 concludes the paper.

## 2. The bitmap marking GC

This section describes the details of the data structures and algorithms for the bitmap marking GC. This description involves low-level bit manipulation. To achieve efficient allocation with bitmaps, we found it essential to design data structures and algorithms in detail. Accurately reporting them requires us to present them in details, sometimes referring to bit-level manipulation. For this reason, we use C like syntax in describing the algorithms.

### 2.1 The GC-mutator interface

Our GC is an exact tracing GC for functional languages. It can be used with any compiler that produces objects with their layout information and maintains an exact root set. In order to make accurate comparison with other GC algorithms, we design our GC algorithm with a uniform interface to a compiler. For this purpose, the implementation of our GC provides a function for the mutator to register a *root set enumerator*, which takes a function on a heap object pointer of the following type `void trace_fn(void **);` and applies this function to each object in a root set. At the beginning of each GC, the collector calls all the registered root set enumerators with an appropriate `trace_fn` function. In the case of Cheney collector, for example, this would be a function to move an object from one semi-space to the other; in our GC, this is a function that marks the bit corresponding to an object.

An allocation operation is designed as a function which takes a request size in bytes and returns a pointer to a newly allocated object. This is the most general style of allocator required for natural data representation. Since the size of natural data representation is different according to their types, the size of some records in a polymorphic function can only be determined at runtime. For example, consider a function of type `'a -> 'a * 'a`. This function returns a 8 byte record for an `int` argument and a 16 byte record for an `real` argument. Of course, for monomorphic cases, where sizes are statically known, we can inline-expand the allocation function to mutator code to produce an optimized allocation code. For the inline-expansion, however, compiler requires detailed knowledge of the allocation function, which may vary according to GC algorithms. To make the compiler independent of GC methods, in this paper, we turn off the inlining.

### 2.2 The structure of the heap space and allocation strategy

In what follows, we assume that one machine word is 32-bit long. This is not a restriction; any other word size can equally be used.

We use a given allocation space as a pool of fixed size allocation areas, called *segments*, and set up the entire heap as follows.

$$heap = (\mathcal{M}, \mathcal{S}, (H_3, \ldots, H_{12}))$$

$\mathcal{M}$ is a special sub-heap for large objects explained earlier. $\mathcal{S}$ is a free segment pool, i.e. set of unused segments. Each $H_i$ is a sub-heap for $2^i$ byte allocation blocks. Each sub-heap has the following structure.

$$H_i = (SegList_i, P_i)$$

$SegList_i$ is a list of segments currently belonging to $H_i$. $P_i$ is an allocation pointer for sub-heap $H_i$ whose structure is defined in subsection 2.4.

A segment $S_i$ initialized for $H_i$ has the following structure.

$$S_i = (Count_i, Blks_i, BitMap_i, Twork_i)$$

where $Count_i$ is the number of already allocated blocks in this segment, $Blks_i$ is an array of $2^i$ byte allocation blocks, $BitMap_i$ is a bitmap tree, and $Twork_i$ is the working area used for tracing live objects. The number of allocation blocks in a segment is derived from the segment size, which is statically fixed. We write $Size_i$ for the number of allocation blocks in $S_i$. We assume that every segment is aligned to power-of-2 boundary for fast bit-level address computation.

We write $Blks_i(k)$ for the $k$-th block in $Blks_i$. Let $L_i = \lceil \log_{32}(Size_i) \rceil$. This determines the height of the bitmap tree in a segment $S_i$. The bitmap tree has the following structure

$$BitMap_i = (BM_i^0, \ldots, BM_i^{L_i-1})$$

where $BM_i^j$ is the $j$-th level bitmap which is a sequence of bits organized as an array of 32 bit words. We write $BM_i^j(k)$ to denote the $k$-th bit and $BM_i^j[k]$ the $k$-th word in the $j$-th level bitmap. The least significant bit in $BM_i^j[k]$ is the bit $BM_i^j(32 \times k + 0)$ and its most significant bit is $BM_i^j(32 \times k + 31)$. The leaf-level bitmap

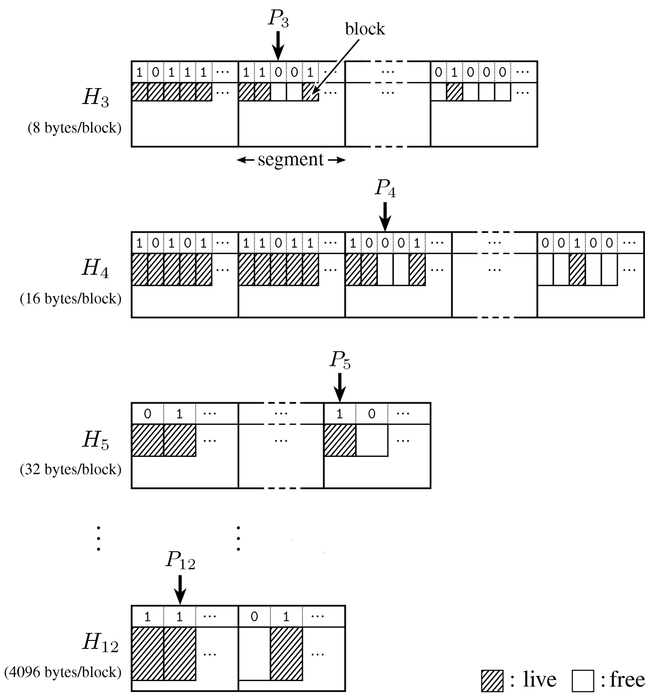

**Figure 1.** Heap structure

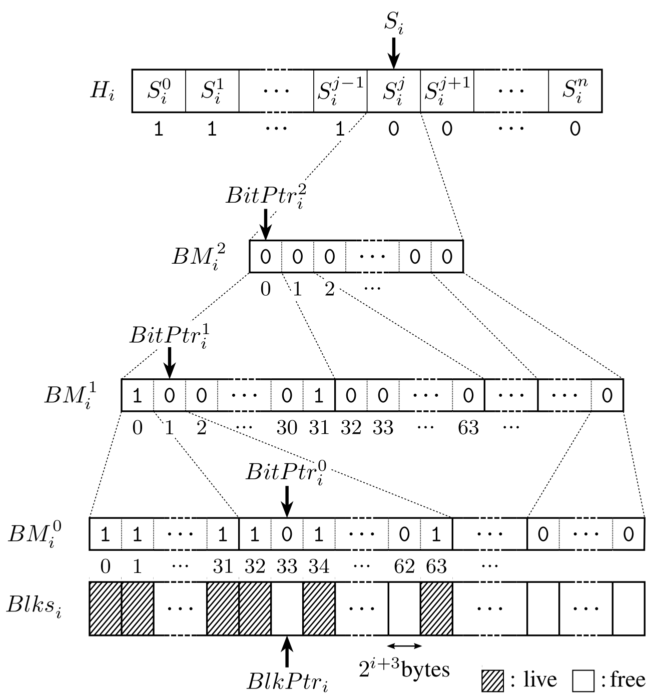

**Figure 2.** Segment structure

$BM_i^0$ represents the liveness of $Blks_i$, namely,

$$BM_i^0(k) = \begin{cases} 1 & Blks_i(k) \text{ is live,} \\ 0 & Blks_i(k) \text{ is free.} \end{cases}$$

As we shall explain below, meanings of 1 and 0 are chosen in such a way that free bit search can be implemented efficiently. The $j+1$-th level bitmap $BM_i^{j+1}$ represents whether each word in $BM_i^j$ has free entry or not, namely,

$$BM_i^{j+1}(k) = \begin{cases} 1 & \text{all bits in } BM_i^j[k] \text{ are 1,} \\ 0 & \text{otherwise.} \end{cases}$$

So for example if all the bits of the top-level bitmap $BM_i^{L_i-1}$ is 1 then all the blocks in $Blks_i$ are live, and there is no space left.

## 2.3 The allocation strategy

With the above structure, $SegList_i$ in $H_i$ forms a single allocation area managed by one (virtual) hierarchically organized tree of bitmaps. The list itself is regarded as the root bitmap of the entire bitmap tree where each bit indicates whether each segment is full or not, and each segment in the list is regarded as an immediate sub-tree of the root bitmap. This structure guarantees that the next free block in $H_i$ can be found in $\log_{32}(Size_i)$ time in the worst case.

In addition, we need to make average allocation as efficient as possible so that it can be comparable to "bump allocation" in Cheney GC. To this end, we observe that in most cases the block array $Blks_i$ is very sparsely used after GC. So we adopt the following strategy.

1. We sequentially allocate the next free block in $H_i$.
2. To make the typical case of allocation fast, we maintain a position information of the next candidate of allocation block. If this block is free then the allocator simply returns this next block and advances the position information.
3. If the next block is live, then the allocator searches for the next free bit using the bitmap tree. To perform this search efficiently, we maintain the next bit position information for higher-level bitmaps.

The allocation pointer $P_i$ in $H_i$ introduced in the previous subsection is for this purpose, whose structure is given below.

$$\begin{aligned} P_i &= (S_i, BitPtrs_i, BlkPtr_i) \\ BitPtrs_i &= (BitPtr_i^0, \ldots, BitPtr_i^{L_i-1}) \end{aligned}$$

$S_i$ is a pointer to the *active* segment in $H_i$, i.e. the segment in which blocks are being allocated. $BitPtrs_i$ are *bit pointers* indicating the next bit positions in $S_i$ to be examined. $BitPtr_i^0$ indicates the next bit position to be tested in the leaf-level bitmap $BM_i^0$. For each $0 \le j \le L_i - 2$, $BitPtr_i^{j+1}$ points to the parent bit representing the bitmap word that includes the bit pointed by $BitPtr_i^j$. $BlkPtr_i$ is a *block pointer* indicating the block address corresponding to the bit pointed by $BitPtr_i^0$. Using these pointers, allocation is done as fast as "bump allocation" when the next block is free.

Figure 1 and 2 shows the structures of the set of sub-heaps and a segment, respectively.

## 2.4 The allocation algorithm

In writing algorithms below, we simply write $SegList_i$ instead of a more verbose notation such as $H_i.SegList$, to indicate that this is the segment list belonging to $H_i$. Similar notations are used for all the other elements including $BitMap_i$, $BM_i^j$ etc.

The structure of the allocation algorithm is given in Figure 3. `alloc(`$n$`)` first locates the sub-heap $H_i$ by `findSubHeapBySize` function. If the target CPU has a native instruction counting trailing 0 bits of an integer (such as `BSR` instruction of x86 machine), this function can be implemented by a sequence of a few instructions. In general cases, this is implemented through a decision tree designed according to allocation ratio of blocks of various sizes. Our benchmark tests show the following average allocation ratio (in number of requests) of first 4 sizes.

```
alloc($n$) {
  $H_i$ = findSubHeapBySize($n$);
  temp = tryAlloc($H_i$);
  if (temp == Fail) {
    bitmapMarkingGC();
    temp = tryAlloc($H_i$); }
  return temp;
}

tryAlloc($H_i$) {
  if (isMarked($P_i$)) {
    if (findNextFreeBlock($P_i$) == Fail) {
      if (nextSegment($SegList_i, P_i$) == Fail) {
        if (newSegment($H_i$) == Fail) {
          return Fail; } }
      findNextFreeBlock($P_i$); } }
  temp = $BlkPtr_i$;
  inc($P_i$);
  return temp;
}

newSegment($H_i$) {
  $S$ = allocSegment($i$);
  if ($S$ == Fail) return Fail;
  append $S$ to $SegList_i$;
  $P_i$ = firstBlockOfLastSegment($SegList_i$);
  return Success;
}

inc($P_i$) {
  incBlkPtr($BlkPtr_i$);
  incBitPtr($BitPtr_i^0$);
}
```

**Figure 3.** Allocation algorithm (top-level)

| $H_i$ | $H_0$ | $H_1$ | $H_2$ | $H_3$ |
|---|---|---|---|---|
| alloc % | 11.6 | 51.3 | 31.4 | 4.5 |

From them, we coded `findSubHeapBySize(`$n$`)` so that it determines $H_i$ for about 93% of allocation requests by 2 comparisons. The algorithm then tries to allocate a block in $H_i$ using `tryAlloc(`$H_i$`)`. If it fails then it invokes the bitmap marking garbage collector. `tryAlloc(`$H_i$`)` first checks whether the block pointed by $P_i$ is marked. If not, then it simply returns the block address stored in $P_i$ and increment $P_i$ by `inc` function. If the next block is already marked, then it searches for the next free bit in the active segment. If the search fails then it advances $P_i$ to the first block of the next segment in the $SegList_i$. If there is no next segment then it tries to dynamically allocate a new segment from the free segment pool by `newSegment` function. Then it tries to search a free bit in the next or new segment. The collection algorithm described later organizes $SegList_i$ so that the search in the next segment never fails. We note that under our strategy of sequentially allocating blocks in $H_i$, it is not needed to set any bit at allocation. The auxiliary functions used in Figure 3 such as `nextSegment` should be clear from their names.

The remaining thing is to define `findNextFreeBlock` as efficient as possible. For this purpose, we have examined and experimented a number of algorithms in detail. The following is the one that we found the fastest.

We implement $BlkPtr_i$ as a machine address of a block, similar to the allocation pointer in copying GC. So `incBlkPtr` just increments the address. $BitPtr_i^j$ is an abstract data structure whose implementation is the following two word data

$$BitPtr_i^j = (Idx_i^j, Mask_i^j)$$

where

- $Idx_i^j$ is an index into the bitmap array $BM_i^j$, and
- $Mask_i^j$ is a 32 bit word in which only one bit is set, indicating the bit position.

For example, if $BitPtr_i^0 = (1, 8)$ (in decimal notation) then it denotes the 3rd bit in the second bitmap word, and corresponds to the 35th block. For bit pointers, we define the following primitives.

- `incBitPtr(`$BitPtr_i^j$`)` increments $BitPtr_i^j$.
- `indexToBitptr(`$x$`)` converts a bit index $x$ to a bit pointer.
- `bitptrToIndex(`$BitPtr_i^j$`)` converts a bit pointer to a bit index.
- `isMarked(`$BitPtr_i^j$`)` tests whether the bit in $BM_i^j$ at $BitPtr_i^j$ is 1.
- `blockAddress(`$BitPtr_i^0$`)` converts a bit pointer to the corresponding block address.

The last two are used in the context of some particular $S_i$. These primitives can be implemented through efficient bit manipulation (e.g. see [36] for bit manipulation techniques.) Similar primitives are defined for $P_i$, which contains $BitPtr_i^0$. In Figure 3 and the following, we also use these primitives for $P_i$ as well as for $BitPtr_i^j$.

We implement `findNextFreeBlock` by using the bit pointers and their primitives. This function finds the next free bit in the bitmap tree of a segment and adjust $BitPtrs_i$ by performing the following steps.

1. Search for a 0 bit in the current bitmap word after $BitPtr_i^0$. If one is found then advance $BitPtr_i^0$ by setting its $Mask_i^0$ accordingly, and return. Otherwise, perform the following two steps.
2. Compute the next free bitmap word index $k$ after $Idx_i^0$ at level 0 by traversing the bitmap trees at level 1 and above. Set $Idx_i^0$ to $k$ and $Mask_i^0$ to 1.
3. Set $Mask_i^0$ to the position of the first 0 bit in $BM_i^0[Idx_i^0]$. This always succeeds and yields the new mask.

The second step may in turn require to compute the free bitmap word indexes and the masks at level 1 and above. So we define the above operation as a procedure `forwardBitPtr(`$j$`)` that recursively computes the bit pointer of level $j$ using the bitmaps of level $j+1$ and above. Then the step 2 above can be implemented as follows.

2.1 Calculate the $j+1$ level bit pointer $BitPtr_i^{j+1}$ that points to the bit corresponding to the $Idx_i^j$-th word in the $j$ level bitmap $BM_i^j$.

2.2 Call `forwardBitPtr(`$j+1$`)` to advance $BitPtr_i^{j+1}$ to the next free bit position.

2.3 Set $BitPtr_i^j$ to the first 0 bit in the bitmap word $BM_i^j[k]$ where $k =$ `bitptrToIndex(`$BitPtr_i^{j+1}$`)`.

Figure 4 shows the structure of the bit search algorithm, which uses the following operation.

- `nextMask(`$BitPtr_i^j$`)` returns a bit mask $Mask$ such that a bit pointer ($Idx_i^j$, $Mask$) points to the next free bit of $BitPtr_i^j$ in the same word. If no free bit is found, it returns `Fail`. This operation is used in the context of some particular $S_i$.

This operation can be implemented efficiently through bit manipulations as well as bit pointer primitives defined earlier.

```
findNextFreeBlock($P_i$) {
  $Mask_i^0$ = nextMask($BitPtr_i^0$);
  if ($Mask_i^0$ == Fail) {
    if (forwardBitPtr($P_i$,0) == Fail) {
      return Fail; } }
  $BlkPtr_i$ = blockAddress($S_i$, $BitPtr_i^0$);
  return Success;
}

forwardBitPtr($P_i$,$j$) {
  if ($j + 1 \geq L_i$) return Fail;
  $BitPtr_i^{j+1}$ = indexToBitptr($Idx_i^j$);
  $Mask_i^{j+1}$ = nextMask($BitPtr_i^{j+1}$);
  if ($Mask_i^{j+1}$ == Fail) {
    if (forwardBitPtr($P_i$,$j+1$) == Fail) {
      return Fail; } }
  $Idx_i^j$ = bitptrToIndex($BitPtr_i^{j+1}$);
  $Mask_i^j$ = 1;
  $Mask_i^j$ = nextMask($BitPtr_i^j$);
  return Success;
}
```

**Figure 4.** Free bit search algorithm

## 2.5 The bitmap marking GC algorithm

When tryAlloc fails to allocate a block in certain $H_i$, the garbage collector is invoked. In addition to standard marking collection, after marking phase, the collector rearranges the newly marked $SegList_i$ in such a way that every segment after the position $P_i$ has at least one free block. This operation ensures that searching a free block in successive segments of the allocation pointer never fail. This arrangement also effectively performs *compacting* the bitmap tree in very low cost by moving all the filled sub-trees to the left of the allocation pointer.

Figure 5 shows the bitmap marking collection algorithm. It performs the following.

1. For each $H_i$, clear all the bitmaps in all the segments by writing 0 to each word of in their bitmap trees. This corresponds to sweeping the entire heap area.
2. For each object in the root set, mark the object, increment the live object count $Count_i$, and record it to the trace stack. The trace stack is implemented as a linked list using $Twork$ work areas.
3. While the trace stack is not empty, pop the trace stack. For each object reference stored in a field of the popped object, mark and push the referenced object to the trace stack, and increment the count.
4. When the marking is complete, then the algorithm reclaims empty segments, and reconstructs $SegList$ by rearranging the remaining non-empty segment. The allocation pointer is put at the beginning of the first segment which has at least one free block.

Marking an object involves the following operations.

1. Find the bit position corresponding to the object.
2. Check whether the bit has already been set. If not, set the bit and propagate this change to upper levels in the bitmap tree.

It uses the following auxiliary function.

- findBitptr($O$) finds the segment $S_i$ in which $O$ is allocated and the bit pointer corresponding to $O$ as follows. Let $addr(X)$ be the starting address of $X$. First, it computes $addr(S_i)$ by bit-level computation from $addr(O)$. Since every segment is aligned to power-of-2 boundary, this is done by masking lower bits of $addr(O)$ to 0. Second, it computes the block index $k$ of $O$ in the block array $Blks_i$ of $S_i$ by address arithmetic. indexToBitptr($k$) is the desired bit pointer.

```
bitmapMarkingGC() {
  for each $H_i$ { clearAllBitmapsAndCount($H_i$); }
  traceLiveObjects();
  for each $H_i$ { rearrangeSegList($H_i$); }
}

traceLiveObjects() {
  for each $O$ in $rootSet$ { markAndPush($O$); }
  while (stackIsNotEmpty()) {
    $O$ = pop();
    for each $O'$ in PointerFields($O$) {
      markAndPush($O'$); } }
}

markAndPush($O$) {
  $S_i, BitPtr_i^0$ = findBitptr($O$);
  if (not isMarked($BitPtr_i^0$)) {
    setBit($P_i$,0);
    incrementSegmentCount($S_i$);
    push($O$); }
}

rearrangeSegList($H_i$) {
  empty = $\{S_i^j \mid S_i^j \in SegList_i, Count_i^j = 0\}$
  filled = $\{S_i^j \mid S_i^j \in SegList_i, Count_i^j = Size_i\}$
  unfilled = $\{S_i^j \mid S_i^j \in SegList_i, 0 < Count_i^j < Size_i\}$
  reclaim all segments in empty;
  $SegList_i$ = concatSegList(filled, unfilled);
  set $P_i$ to the first $S_i^j$ in $SegList_i$ such that $Count_i^j < Size_i$;
}

setBit($P_i$,$j$) {
  $BM_i^j$[$Idx_i^j$] = $BM_i^j$[$Idx_i^j$] | $Mask_i^j$;
  if ($j + 1 < L_i$ and $BM_i^j$[$Idx_i^j$] == 0xFFFFFFFF) {
    $BitPtr_i^{j+1}$ = indexToBitptr($Idx_i^j$);
    setBit($P_i$,$j+1$); }
}
```

**Figure 5.** Bitmap marking GC algorithm

Any other auxiliary functions used in Figure 5 should be clear from their names.

## 3. Generational extension

In order for our non-moving bitmap marking GC to become a practical and truly better alternative to copying GC, we would like to extend it to generational GC. It is of course straightforward to combine our bitmap marking GC with any other GC to form a generational GC by moving objects between two heaps managed by two separate GC, but the resulting system loses the non-moving property. Our goal is to develop a new method that achieves the desired effect of generational GC without actually moving objects. This section presents one such general method.

### 3.1 General strategy

Demers et.al. [? ] observed that generational GC is a partial GC that collects some subset of objects, and presented two variants of generational mark and sweep GC. This general idea can be adapted to develop a generational extension of our bitmap marking GC. Our strategy is to maintain *multiple bitmap trees for the same sub-heap*: one for each generation.

In an abstract view, generational GC manages the set of live objects in $H_i$ as $n$ disjoint sets $\mathcal{G}_i^1, \ldots, \mathcal{G}_i^n$, called generations. We note that a bitmap tree $BitMap_i$ of $H_i$ can represent any subset of the set of all blocks in $H_i$, and therefore that each generation $\mathcal{G}_i^k$ can be represented by a separate bitmap tree $BitMap_i^k$ of $H_i$. These observations yield the following strategy. We use $n$ bitmap trees $BitMap_i^1, BitMap_i^2, \ldots, BitMap_i^n$ for the same sub-heap $H_i$. Let $[\![BitMap_i^k]\!]$ be the set of objects represented by $BitMap_i^k$. The GC algorithm maintains the following invariant for each $k$:

$$[\![BitMap_i^k]\!] = \mathcal{G}_i^k \cup \mathcal{G}_i^{k+1} \cup \cdots \cup \mathcal{G}_i^n$$

or equivalently, $\mathcal{G}_i^k = [\![BitMap_i^k]\!] \setminus [\![BitMap_i^{k+1}]\!]$. The generational GC algorithm can then be realized as follows.

1. Every object is assigned a generation number $k$ and a "tenure" counter indicating the number of minor collections it survived.
2. The mutator allocates an object in $\mathcal{G}_i^1$ with its tenure counter initialized to 0.
3. Reclamation of $\mathcal{G}_i^k$ is done by performing the following steps. If $k = n$, then the algorithm clears all the bitmap trees for $H_i$. If $k < n$ then it overrides $BitMap_i^k$ with $BitMap_i^{k+1}$. This corresponds to sweeping $\mathcal{G}_i^k$ by clearing the bits of objects in $\mathcal{G}_i^k$. The algorithm then traces the set of live objects. If it finds an unmarked live object, it marks the object in $BitMap_i^k$ and increments its tenure counter. If the tenure counter reaches the tenuring threshold, $tenureAge$, then the algorithm also marks it in $BitMap_i^{k+1}$. This corresponds to promoting the object to $\mathcal{G}_i^{k+1}$. When the tracing completes, the algorithm resets $BitMap_i^j$ to $BitMap_i^k$ for all $j$ smaller than $k$.
4. Reclamation of $\mathcal{G}_i^1$ (minor GC) is invoked at some interval determined by a GC policy. When $\mathcal{G}_i^1$ is filled up to some predefined level, the algorithm estimates the oldest generation $\mathcal{G}_i^k$ that needs to be collected, and invokes the collection of $\mathcal{G}_i^k$.

## 3.2 The GC algorithm

In this paper we restrict the number of generations to 2. Then GC is either minor collection of $\mathcal{G}_i^1$ or major collection of $\mathcal{G}_i^2$. For this simple case, the major tuning parameters are the effective "size of minor heap" and the frequency of minor collections. In our segment based bitmap GC, these two can be controlled by the following two parameters.

- *minorSize*: the number of segments which can be allocated before the minor collection.
- *fillLimit*: the percentage limit to which each segment can continue to be filled after minor GC. Segments that are filled beyond this limit are regarded as full and will not be used for further allocation.

Each segment $S_i$ is extended with $oldBitMap_i$ corresponding to $BitMap_i^2$. A generational GC algorithm is obtained by extending the `tryAlloc` function and defining a new procedure `minorGC` for minor collection. Figure 6 defines these functions. The algorithm for the major collection is the same as `bitmapMarkingGC` defined before. The allocation function `alloc` is also the same as before, except that it uses the new `tryAlloc`. The `traceLiveObjects` is also the same, except that it uses the new `markAndPush` that manages the tenure counter and promotion. The underlined part in the figure 6 is required refinement to the non-generational version. The `setBit` is extended so that it can mark bits in $oldBitMap_i$.

```
tryAlloc($H_i$) {
  if (isMarked($P_i$)) {
    if (findNextFreeBlock($P_i$) == Fail) {
      increment the minor GC counter;
      if (minor GC counter $\geq$ $minorSize_i$) minorGC();
      if (findNextFreeBlock($P_i$) == Fail) {
        if (nextSegment($SegList_i, P_i$) == Fail) {
          if (newSegment($H_i$) == Fail) {
            return Fail; } }
        findNextFreeBlock($P_i$); } } }
  temp = $BlkPtr_i$;
  inc($P_i$);
  return temp;
}

minorGC() {
  for each $H_i$ {
    for each $S_i^j$ in Sub($SegList_i, P_i, minorSize$) {
      overwrite $BitMap_i^j$ with $oldBitMap_i^j$; } }
  traceLiveObjects();
  for each $H_i$ { rearrangeSegListMinor($H_i$); }
  reset the minor GC counter;
}

markAndPush($O$) {
  $S_i, BitPtr_i^0$ = findBitptr($O$);
  if (not isMarked($BitPtr_i^0$)) {
    setBit($P_i$);
    incrementSegmentCount($P_i$);
    push($O$);
    increment the tenure counter of $O$;
    if (the tenure counter of $O \geq tenureAge$) {
      reset the tenure counter of $O$;
      setBit($P_i$) in $oldBitMap_i$; } }
}

rearrangeSegListMinor($H_i$) {
  filled = $\{S_i^j \mid S_i^j \in$ Sub$(SegList_i, P_i, minorSize)$,
                   $Count_i^j \geq fillLimit \times Size_i\}$
  unfilled = $\{S_i^j \mid S_i^j \in$ Sub$(SegList_i, P_i, minorSize)$,
                   $Count_i^j < fillLimit \times Size_i\}$
  list = concatSegList(filled, unfilled);
  replace Sub($SegList_i, P_i, minorSize$) with list;
  set $P_i$ to the first $S_i^j$ in $SegList_i$
      such that $Count_i^j < fillLimit \times Size_i$;
}
```

**Figure 6.** Generational bitmap marking GC algorithm

Sub$(SegList_i, P_i, n)$ is a reference to the sub-list of $SegList_i$ consisting of $n$ segments immediately preceding $P_i$ in the list.

If we further restrict $tenureAge$ to 1, a particularly simple variant exists. As we noted earlier, allocation only advances the bit pointer without setting the bits corresponding to newly allocated blocks. This means that $BitMap_i$ is unchanged during allocation. Since $tenureAge$ is 1, $BitMap_i$ after minor GC is the new contents of $oldBitMap_i$. So we do not actually need $oldBitMap_i$. The simple variant is obtained from the algorithm in Figure 6 by eliminating the bitmap overriding operation in `minorGC`, and using the same `markAndPush` function as non-generational GC defined in Section 2.5. The resulting algorithm coincides with the collection algorithm using "sticky mark bits" described in [10]. This simplified generational GC algorithm can be implemented with the same data structure and the support functions as the non-generational version.

## 4. Implementation and Evaluation

We have implemented the bitmap marking GC algorithm presented in Section 2 and its generational extension presented in Section 3. For comparison purpose, we have also implemented a plain Cheney copying collector, and a generational copying GC algorithm for Standard ML described in [29]. In this section, we outline our implementation and show the detailed evaluation results.

### 4.1 Implementation

Our implementation of non-generational GC takes the segment size and the total usable memory size as parameters, allocates the specified memory using `mmap` system call, and sets up the free segment pool and the 10 sub-heaps. The special sub-heap $\mathcal{M}$ for large objects is implemented as a mark and sweep GC, whose allocation area is dynamically obtained and freed using C library functions `malloc` and `free`. Since the usage of this special heap is very low in our benchmark tests, this special heap does not affect much on performance results. So we will not include performance data of this heap.

The free segment pool $\mathcal{S}$ is implemented as a free list of segments.

In addition to the above standard implementation, we have also implemented a special version of non-generational bitmap-marking GC. In this special version, instead of managing the allocation space as a collection of segments, each sub-heap $H_i$ consists of a single large segment whose size is statically determined. This special implementation first reads an environment variable that holds 10 numbers $R_3, \ldots, R_{12}$ representing percentage ratio of the sizes of sub-heaps $H_3, \ldots, H_{12}$, and sets up the single segment for each $H_i$. As we shall report in the next subsection, this version is used to evaluate performance of sub-heap size adjustment through segment-based dynamic allocation and reclamation.

In what follows, we refer to the standard implementation of our non-generational bitmap marking GC as **bitmap** (or **bm**), and the special version as **bitmap(static)**.

We have implemented the simplified version of generational GC with 2 generations and $tenureAge = 1$, described in Section 3.2, by extending the standard implementation **bitmap**. The only major addition is a mechanism for a *write barrier* and a *remembered set* to keep track of pointers to young objects from outside. Since $tenureAge$ is 1 and any object survived a minor collection is promoted to the old generation, the remembered set can be flushed after minor GC. Taking advantage of this fact, we allocate a remembered set in the collector's trace stack. As mentioned before, our trace stack is implemented as a linked list using $Twork$ work areas. This is done by assigning a unique pointer slot in $Twork$ to each object. This implementation allows us to determine whether a given object is already in the list or not by checking whether the pointer is non-null. This automatically eliminates duplication in the remembered set. A write barrier can then be incorporated in the generational collector as follows. A write barrier code takes a young object that is to be referred from the old generation due to mutation, and marks it and pushes it to the trace stack. Minor collector simply traces objects using the trace stack whose initial contents is the remembered set. In what follows this implementation is referred to as **bitmap(gen)** (or **bm(gen)**).

Our implementation of Cheney collector uses the given memory as their two semi-spaces. In addition, we modified the standard Cheney algorithm so that it also uses the special sub-heap $\mathcal{M}$ for large objects, just the same way as in **bitmap**, **bitmap(static)** and **bitmap(gen)**. We have carefully tuned and optimized the allocation and collection functions of the Cheney collector, consulting the generated x86 assembly code, to make the Cheney collector performs the best as itself for x86 machine. We note that our change of including a special sub-heap for large objects makes the Cheney GC slightly faster in most cases, especially for those programs that use large objects. So this change is in favor of the Cheney collector. We also note that this change adds two x86 machine instructions to the optimized Cheney allocator, which should have increased allocation overhead by very small amount. According to our evaluation, this overhead is below margin of error. We refer to this implementation as **cheney** (or **cp**).

Our implementation of a generational copying GC for Standard ML mostly follows [29], which is one of standard generational GC algorithm for functional languages, except that it uses a fixed heap layout instead of dynamically allocating to-spaces, and that our implementation uses exact remembered set instead of card-marking. As in [29], the tenuring threshold is fixed to 1. The number of generations can be set to a number given as a parameter. The default number of generations used in Standard ML of New Jersey compiler is 5, and our GC only have 2 generations. So we have compared our GC against this copying GC with 2 generations and 5 generations. We refer to them as **copy(2g)** (or **cp(2g)**) and **copy(5g)** (or **cp(5g)**) respectively.

### 4.2 Performance Evaluation

We evaluate performance of our algorithms using publicly available benchmark programs for Standard ML, omitting benchmarks such as `fib, tak, ntakl` whose memory usage is very small. The evaluation has been done on an Intel Xeon 5150 2.66GHz processor with 8GB memory running SUSE Linux Enterprise Server 10. Execution time is measured by the system timer.

We organize our performance evaluation using the following general parameters and measures.

- *Heap size*. The entire heap size allocated by the runtime system except for special sub-heap $\mathcal{M}$. This size includes the allocation area as well as all the administrative data for GC. For the Cheney collector, this is the size of the sum of "to" and "from" spaces. For our GC, this is the size of the sum of allocation blocks, bitmap trees, and a trace stack combined.
- *Live object ratio*. The average percentage ratio of the total size of live objects after GC against the heap size.
- *Memory occupancy ratio*. The average percentage ratio of the amount of memory actually used for objects just before GC against the entire heap size. In the case of the Cheney GC, this is 50% at best. Since the allocator aligns each objects to double word boundary, the actual occupancy ratio is lower than 50%. The ratio of our GC depends on the sub-heap configuration.
- *Total time*. The *user* time (in seconds) spent for the program to run.
- *GC time*. The time (in seconds) spent in GC.

In our method, sweeping is done by clearing bitmaps, which is proportional to the heap size with very small constant factor. According to all the results, the bitmap clear time is in the range of 2.4% – 0.06% of the GC time, and in most cases they are less than 1% of the GC time. So we can safely conclude that bitmap clear time is negligible. We omit the detailed data.

Through preliminary evaluations and comparisons, we found that performance depends on the live object ratio, and that the set of benchmarks can be roughly grouped into the following three categories.

- Group I of those that only produce very short-lived data, and show very low live object ratio.

| Group | benchmark | size (MB) | **bitmap(static)** | | | **bitmap** | | | **cheney** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | time | live | occ. | time | live | occ. | time | live | occ. |
| I | count_graphs | 2 | 15.27 | 6.01 | 67.8 | 15.15 | 6.02 | 55.44 | 15.77 | 6.02 | 48.7 |
| | cpstak | 2 | 1.33 | 5.52 | 64.2 | 1.26 | 5.56 | 51.63 | 1.43 | 5.55 | 48.9 |
| II | knuth_bendix | 12 | 0.74 | 10.01 | 63.4 | 0.76 | 10.45 | 61.1 | 0.75 | 10.17 | 46.1 |
| | ratio_regions | 20 | 37.82 | 12.21 | 54.0 | 37.24 | 11.97 | 62.3 | 38.21 | 11.95 | 47.4 |
| III | gcbench | 65 | 3.39 | 9.80 | 64.6 | 3.49 | 10.61 | 65.9 | 3.87 | 10.25 | 42.9 |
| | perm9 | 190 | 3.03 | 14.20 | 53.2 | 3.32 | 22.36 | 57.6 | 3.49 | 16.91 | 41.6 |

**Table 1.** Performance of segment-based sub-heap size adjustment

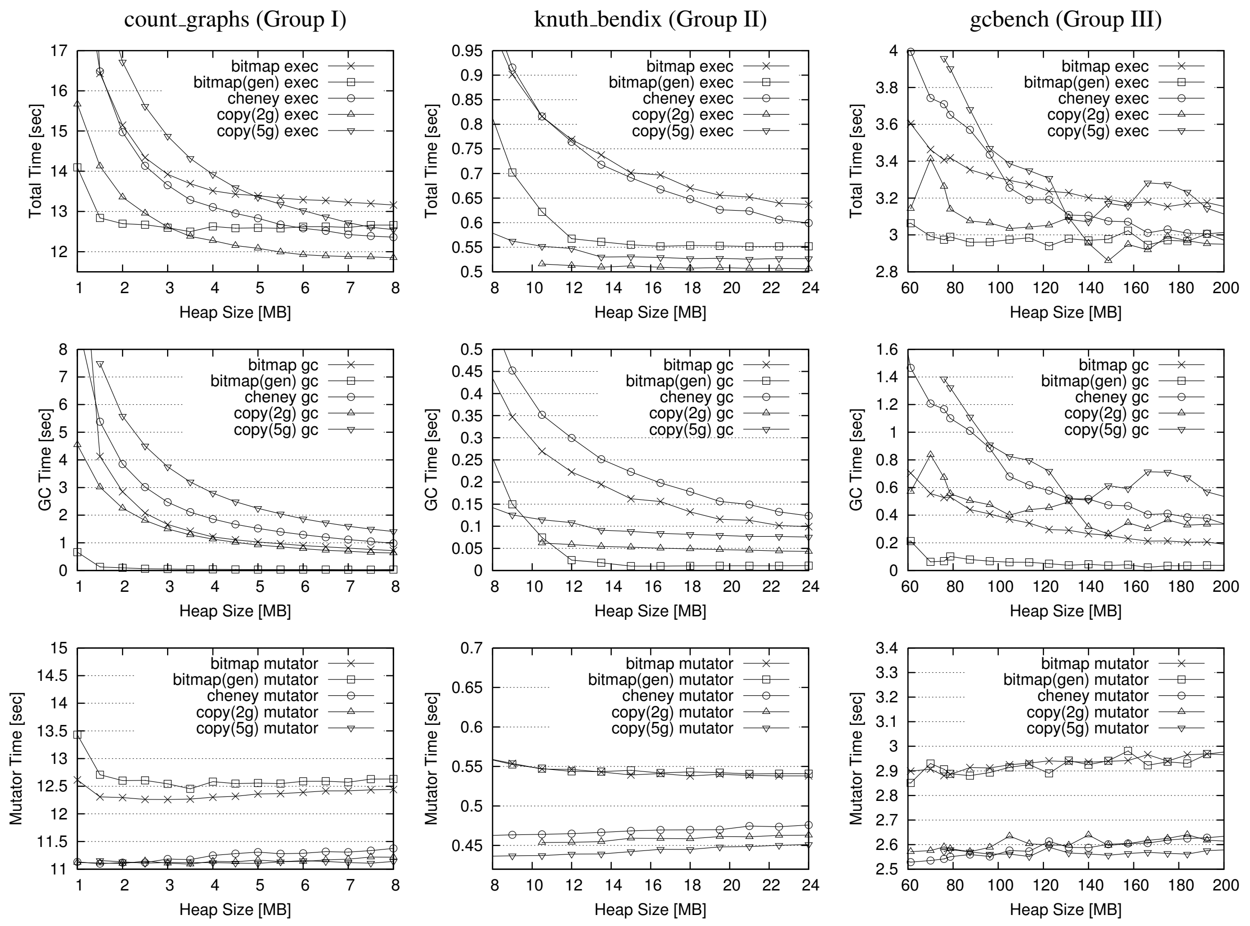

**Figure 7.** Performance of **bitmap(gen)** on typical benchmarks

- Group III of those that require some amount of memory and show high live object ratio.
- Group II is in between Group I and Group III.

We sometimes use these groups in evaluation and comparison.

Our first evaluation is the performance of automatic sub-heap size adjustment through dynamic segment allocation and reclamation. Our claim of fragmentation-avoided heap depends on this performance. If sub-heap size adjustment is not optimal in our GC then large amount of memory would have been wasted. To evaluate this, we have first estimated the best possible sub-heap size configuration for each benchmark using our **bitmap(static)** implementation; we have run each benchmark test on **bitmap(static)** a number of times (30 – 80 times) with different sub-heap size configurations by changing the environment variable we mentioned above. For each benchmark, we have selected the sub-heap size configuration that shows the best performance as our estimate of the optimal sub-heap size configuration for **bitmap(static)**. We have then compared performance of this hand-tuned **bitmap(static)** against **bitmap** and **cheney**. We omit the details of the result and only show a summary of overall performance of two typical benchmarks in each group in Table 1. We witnessed that all the others show similar behavior. The results show the following.

- The hand-tuned **bitmap(static)** performs better than **cheney** in most cases. The GC time is always shorter, and the total time is slightly better in most cases.
- The hand-tuned **bitmap(static)** shows significantly better memory usage for almost all the cases.
- Performance of the standard **bitmap** is as good as the best hand-tuned **bitmap(static)** in memory usage, GC time and the execution time.

These results verify that the dynamic segment allocation mechanism we have developed and implemented performs as efficiently as the best possible sub-heap size assignment obtained through hand-tuning. We thus conclude that this mechanism achieves almost optimal heap size adjustment automatically and dynamically.

Prior to overall performance evaluation, we searched for optimal values of the tuning parameters for the standard version **bitmap** and the generational version **bitmap(gen)**.

The segment size is a tuning parameter common to **bitmap** and **bitmap(gen)**. We have evaluated the performance of **bitmap** with various segment sizes from $32\mathrm{KB}$ to $512\mathrm{KB}$. The results show that GC performance is not sensitive to segment size except for the cases where the total heap size is rather small compared to segment size. We omit the detailed data, and only state our conclusion that the segment size of $128\mathrm{KB}$ is generally acceptable for all the benchmarks we used. All the data of **bitmap** and **bitmap(gen)** in this section are taken with $128\mathrm{KB}$ segments.

For **bitmap(gen)**, we need to set its tuning parameters *fillLimit* and $minorSize$. To determine the optimal values for *fillLimit*, we have evaluated the benchmarks against values of $0.2, 0.3, \ldots, 0.9$. According to the results, which we omit here, both the GC time and the total time of the most of benchmarks are not sensitive to this parameter, and $0.5$ is reasonable for all the cases. So we set *fillLimit* to $0.5$. To determine the optimal value of $minorSize$, we have evaluated the benchmarks with different $minorSize$ values from $1$ to the total number of segments. The benchmark test results, which we omit, indicate that larger $minorSize$ value is generally favorable. So we set $minorSize$ to be unlimited.

With these parameters, we have evaluated our generational GC **bitmap(gen)** and compared the results with those of the copying collectors **copy(2g)** and **copy(5g)**. For this purpose, we have run each of the benchmark programs with various heap sizes, ranging from the minimum runnable size to sufficiently large size.

Figure 7 shows the results of a typical benchmark of each of the three groups. It shows the total execution time (exec), the GC time (gc), and the mutator time (mutator) i.e. the difference of exec and gc against various heap sizes. From these sample results, we observe the following properties.

- For both generational and non-generational versions, the GC time of bitmap marking GC methods is shorter than that of copying GC methods. In particular, the GC time of **bitmap(gen)** is the shortest in most cases. Among the copying GC methods, **copy(2g)** is the shortest in GC time.
- The mutator time of bitmap GCs is longer than that of copying GCs, as expected. This difference should be due to the overhead of bitmap searching against bump pointer allocation. The difference of mutator time between non-generational and generational versions is very small.
- The total execution time of **bitmap(gen)** is shorter than that of **bitmap**. This shows that our generational GC scheme has the expected advantage for mark and sweep GC. Among copying GC methods, **copy(2g)** is the shortest in the total time.
- We cannot draw any definite conclusion on the relative strength of **bitmap(gen)** against **copy(2g)**; sometimes **bitmap(gen)** outperforms **copy(2g)** and conversely depending on benchmarks and heap size.

In addition to those 3 benchmarks shown in Figure 7, we evaluated the other benchmarks against varying heap sizes. We analyze these evaluation results and try to extract general property of our bitmap GC method, in particular, the relative strength of **bitmap(gen)** against **copy(2g)**.

To make a proper analysis of the benchmark data, we normalize them according to the relative heap size against the minimum runnable heap size. For most of the benchmarks, the minimal heap size of a bitmap marking GC is smaller than the corresponding copying GCs in both simple and general version. So we take the minimal heap size of each benchmark as the minimal heap with which **bitmap(gen)** can run. To eliminate anomalous or singular results due to too small or large heap, we sample the benchmark results with heap sizes from 2 to 6 times of this minimum heap size unit. Table 2 shows the summary of the average execution time over varying heap sizes for each benchmark. This table shows the following.

| Group | benchmark | min | **bm** | **bm(gen)** | **cp** | **cp(2g)** | **cp(5g)** |
|---|---|---|---|---|---|---|---|
| I | barnes_hut | 1 | 1.24 | 1.17 | 1.20 | 1.13 | 1.29 |
| | count_graphs | 1 | 13.75 | 12.60 | 13.32 | 12.40 | 14.25 |
| | cpstak | 1 | 1.09 | 0.89 | 1.06 | 0.92 | 1.16 |
| | diviter | 1 | 3.40 | 3.15 | 3.36 | 3.22 | 3.53 |
| | divrec | 1 | 4.03 | 3.75 | 3.80 | 3.64 | 3.94 |
| | fft | 2 | 1.61 | 1.52 | 1.55 | 1.49 | 1.57 |
| | life | 1 | 0.59 | 0.57 | 0.56 | 0.55 | 0.58 |
| | logic | 1 | 2.26 | 2.09 | 2.22 | 2.16 | 2.65 |
| | mandelbrot | 1 | 1.05 | 0.97 | 1.00 | 0.97 | 1.03 |
| | nucleic | 1 | 0.13 | 0.12 | 0.13 | 0.12 | 0.14 |
| | puzzle | 2 | 27.40 | 24.99 | 26.62 | 24.04 | 25.08 |
| | ray | 1 | 0.95 | 0.84 | 0.91 | 0.84 | 0.95 |
| | simple | 2 | 1.82 | 1.69 | 1.73 | 1.64 | 1.76 |
| II | boyer | 5 | 0.09 | 0.09 | 0.10 | 0.09 | 0.10 |
| | knuth_bendix | 6 | 0.65 | 0.55 | 0.62 | 0.51 | 0.52 |
| | lexgen | 8 | 1.21 | 1.12 | 1.06 | 0.99 | 1.01 |
| | mlyacc | 9 | 0.20 | 0.19 | 0.18 | 0.17 | 0.17 |
| | ratio_regions | 13 | 35.47 | 38.16 | 33.60 | 33.05 | 36.55 |
| | smlboyer | 5 | 0.75 | 0.73 | 0.77 | 0.80 | 0.85 |
| | tsp | 7 | 0.51 | 0.52 | 0.51 | 0.61 | 0.66 |
| | vliw | 2 | 1.81 | 1.37 | 1.52 | 1.30 | 1.54 |
| III | gcbench | 35 | 3.25 | 2.98 | 3.23 | 3.03 | 3.33 |
| | perm9 | 95 | 2.95 | 2.99 | 3.07 | 3.58 | 4.29 |

(min: minimum heap size in MB for **bm(gen)**. **bm**, **bm(gen)**, **cp**, and **cp($n$g)**: average total execution time with each GC method in seconds.)

**Table 2.** Performance summary

- **bitmap(gen)** outperform **bitmap** in almost all cases, as expected.
- **copy(2g)** shows the best performance among the copying GC methods in most cases.

Figure 8 shows the ratio of total execution time of **bitmap(gen)** against that of **copy(2g)** (smaller is better) under 2, 4, and 6 times of the minimum heap size. These results show that, for smaller heap size, **bitmap(gen)** outperforms **copy(2g)**. While the performance of **bitmap(gen)** is getting slightly worse than that of **copy(2g)** when the heap size increases, the ratios are around 1 in most of benchmarks. From these data, we can conclude that the performance of **bitmap(gen)** is comparable to that of **copy(2g)**.

These results are quite satisfactory; they show that our non-moving generational GC is a viable alternative to generational copying GC for functional languages.

## 5. Related works

This section compares our approach with some more related works that are not mentioned in Section 1.

Supporting efficient GC is essential for any optimizing functional language compilers. The current state-of-the-art in this field appears to be a generational GC whose minor (younger) collector is a variant of Cheney copying collector. OCaml, MLton, and old version of Haskell use a hybrid of copying GC with mark-compact GC [7, 31]. SML/NJ [29], Chez Scheme [14], and the latest Glasgow Haskell Compiler [23] use generational copying GC. In all

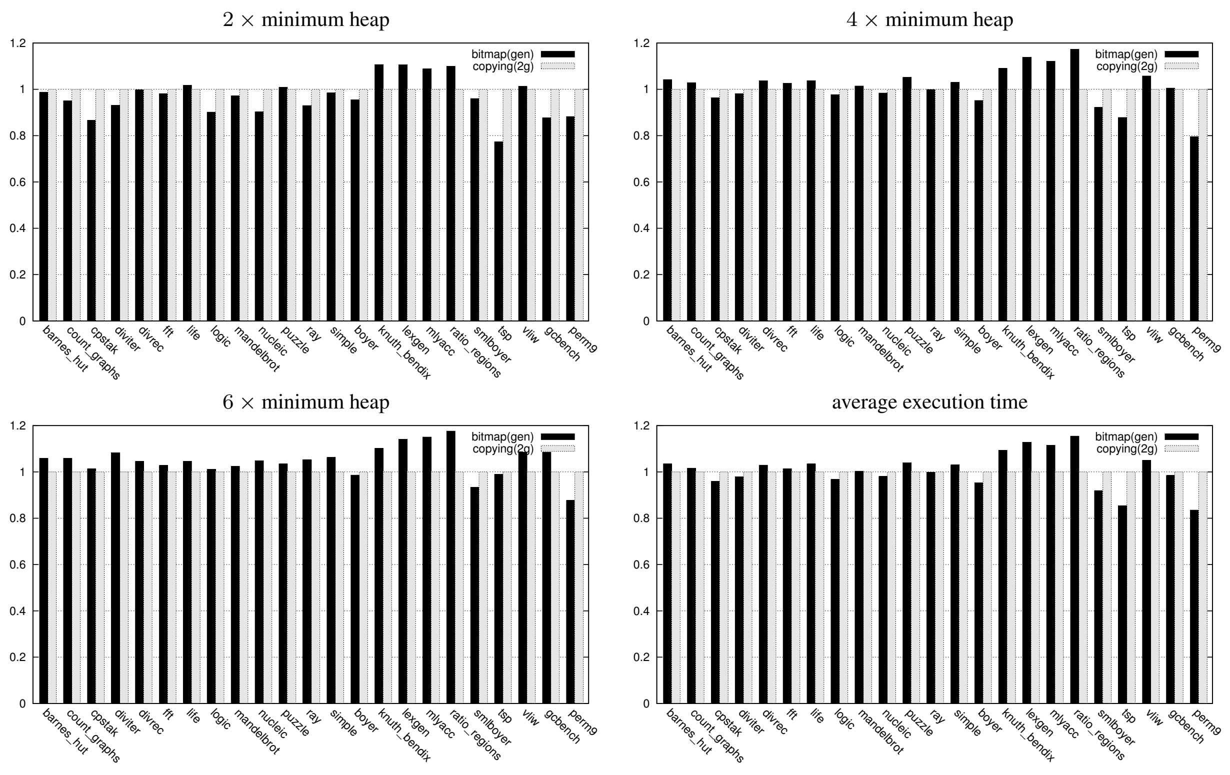

**Figure 8. bitmap(gen)** vs. **copy(2g)**

these systems, the Cheney's copying collector is used for minor GC and object-moving is inevitable, which makes inter-operation with other languages cumbersome and error-prone. A new contribution of our work is to present a non-moving alternative. Our benchmark tests including those that generate large amount of short-lived data such as `cpstak` show the feasibility of our approach.

Mark and sweep GC has also been extensively studied. Major issues have been efficient data structure for live objects, efficient allocation, avoiding fragmentation through compaction, and efficient sweeping.

The idea of using separate bitmaps to represent live objects is well known. Examples include bitwise sweep [12], bitmapped fits [37], and mark-and-deferred-sweep [38] based on lazy sweeping [16]. The main focuses of these works and others are on reducing the GC time and the cost for free list construction by avoiding scanning the entire heap. In contrast, our GC method uses bitmaps not only for representing live objects but also for allocation. Allocation is done using bitmaps without any other data structure.

Reducing sweep cost has been one of central issue, and a number of approaches have been proposed and implemented. Examples include mark during sweep [28], Treadmill [1], lazy sweeping [5, 16, 38], and selective sweeping [9]. Mark and split [30] eliminates subsequent sweep operations. In the context of sweeping, we can say that our approach eliminates the need of complex algorithms and data structures for sweeping and compaction, and yet achieves very fast sweeping; sweep is done simply by clearing bitmaps, whose cost is shown to be negligible in our benchmark tests.

One approach for efficient allocation, widely used in many GC methods [4, 6, 11, 15, 22, 29, 34], is to divide the heap into multiple areas of different size and manage those areas as a pool of *Big Bag of Pages* (BiBoP) [18] and some form of segregated free list such as [20]. In general perspective, our method of dividing the heap into sub-heaps of fixed size segment shares the same motivation with these previous works. In particular, obtaining a bit pointer from an object address mentioned in Subsection 2.5 can be regarded as an instance of encoding object meta-data in BiBoP. Our major new contribution is to prepare a sequence of sub-heaps of exponentially increasing block sizes. With this configuration, we avoid costly compaction at a reasonable expense of locally wasting space in each allocation block.

The problem of fragmentation and efficient compaction have also been extensively investigated. Example of recent works include [2, 4, 19, 26]. However, in most of the proposed approaches, possibility of fragmentation remains and for long running program, object-moving compaction is inevitable. SCHISM/CMR [27] attempts to eliminate fragmentation by fixing the size of an allocation unit, and decomposing objects into multiple units. Although this eliminates fragmentation in the main heap, it needs to introduce a separate copying heap for meta-level data to represent object structures. In contrast, our GC does not require any other structure than the heap itself, and still does not create fragmentation. So compaction is not needed by construction. Our observation is that our sub-heap organization achieves satisfactory compaction-free heaps at the acceptable expense of locally wasting allocation space.

Our goal is to develop an efficient non-moving collector. A pioneering work on this area is Boehm-Weiser conservative GC [6] developed for C. Since it is not possible to locate all the pointers of objects allocated by C programs, object-moving GC is impossible. Our non-moving constraint has the same origin: to inter-operate with C, we cannot move objects that are accessed from C programs. The problem domain is however rather different. Our aim is not to collect objects allocated in the C heap, but to collect only objects that are allocated in the functional-language heap. We hope that this opens up a new possibility of using GC technologies developed for functional languages to develop a new method satisfying non-moving constraint. Precise relationship between our GC method and those of conservative GC and its variants including mostly-copying GC [3, 32] is a topic for further investigation.

## 6. Conclusions and further development

Motivated by developing a memory management system that allows functional languages to seamlessly inter-operate with C, we have proposed a new garbage collection algorithm based on bitmap marking and have reported its implementation and performance evaluation. The proposed method realizes *a fragmentation-avoided heap* through a family of sub-heaps for objects of $2^i$ bytes $(3 \leq i \leq 12)$, and *efficient allocation* through hierarchically organized bitmaps. This method can be extended to non-moving generational GC by maintaining separate bitmaps for the same allocation segments. In our extensive benchmark tests, our method is comparable to simple and generational copying GC in most cases. These results achieves our goal of developing a non-moving GC method that is as efficient as copying GC currently used in functional languages.

This is our first step towards a satisfactory memory management system suitable for functional languages that support various practical features including interoperability with C. A number of further issues remain to be investigated, including:

1. complete implementation and evaluation of generational GC,
2. further efficient allocation techniques such as bunched allocations, and
3. efficient multithreading support for modern shared memory multi-core processors.

The first and second one can be implemented without much difficulty. The last one is more challenging and requires new data structures and algorithms. We have already done preliminary design and implementation of multithreading extension of our GC method. Its detailed development and evaluation is outside of the scope of the present paper. In what follows, we briefly described the extension, focusing on the advantage offered by our new GC method.

There are a number of issues in developing efficient allocation and GC method that support multithreading on a modern shared memory multi-core architecture. Here we focus on the reduction of synchronizations overhead at allocation time. Obtaining a lock at each allocation time would be prohibitively expensive. An existing proposal for a functional language [13] is to use thread-local heaps. However, introduction of thread local heaps in an ML-style imperative functional language complicates the algorithms and would incur overhead in both allocation and collection due to the need of maintaining the invariant that there should be no pointer from a global heap to a local heap. In contrast, our GC method allows each thread to allocate a block in a global heap directly without any locking for most of the cases. As described in section 2, our allocation algorithm searches for a free block within the current active segment using an allocation pointer and a bitmap for that segment. A multithread extension can then be obtained simply by giving each thread its own active segment. Since the allocation pointer and the bitmap in each segment is local to that segment, each thread can allocate objects without any locking at all until the active segment becomes full. When the current active segment for a thread becomes full, the thread acquires exclusive lock on the sub-heap and attempts to obtain a new segment for the active allocation segment of the thread. If the free segment pool is empty, GC is invoked. The simple strategy to extend our GC to support multithreading is stop-the-world: suspending all threads before starting the bitmap marking GC. We have already experimentally implemented this approach and obtained preliminary evaluation result showing that it achieves expected efficient allocation in a multithreaded program running on a multi-core processor.

We expect that non-moving property should also be beneficial in developing concurrent GC, and that the combination of the above approach with concurrent and parallel GC mechanisms (see [17] for survey) would yield a satisfactory non-moving concurrent GC for functional languages, which can work with native threads.

## References

[1] H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–70, 1992.

[2] K. Barabash, O. B.-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27:1097–1146, 2005.

[3] J. F. Bartlett. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1(6):3–12, 1988.

[4] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2008.

[5] H.-J. Boehm. Reducing garbage collector cache misses. In *Proceedings of the International Symposium on Memory management*, pages 59–64, 2000.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[7] E. Chailloux, P. Manoury, and B. Pagano. *Developing applications with Objective Caml*. O'Reilly, April 2000. French version.

[8] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11), 1970.

[9] Y. C. Chung, S.-M. Moon, K. Ebcioğlu, and D. Sahlin. Selective sweeping. *Software Practice and Experience*, 35(1):15–26, 2005.

[10] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: framework and implementations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 261–269, 1990.

[11] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the International Symposium on Memory Management*, pages 37–48, 2004.

[12] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: a case study of building efficient, scalable Jvms. *IBM Systems Journal*, 39(1):151–174, 2000.

[13] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, 1993.

[14] R. K. Dybvig. The development of Chez Scheme. In *Proceedings of the ACM International Conference on Functional Programming*, pages 1–12, 2006.

[15] R. K. Dybvig, D. Eby, and C. Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical report, Indiana University, 1994.

[16] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1082, 1982.

[17] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.

[18] G. L. Steele Jr. Data representation in PDP-10 MACLISP. *MIT AI Memo 421*, September 1977.

[19] H. Kermany and E. Petrank. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 354–363, 2006.

[20] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[21] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[22] S. Marlow, T. Harris, R. P. James, and S. P. Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the International Symposium on Memory Management*, pages 11–20, 2008.

[23] S. Marlow, S. P. Jones, and S. Singh. Runtime support for multicore Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 65–78, 2009.

[24] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[25] H.-D. Nguyen and A. Ohori. Compiling ML polymporphism with explicit layout bitmap. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 237–248, 2006.

[26] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the International Symposium on Memory management*, pages 159–172, 2007.

[27] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–159, 2010.

[28] C. Queinnec, B. Beaudoing, and J.-P. Queille. Mark during sweep rather than mark then sweep. In *Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, pages 224–237, 1989.

[29] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical report, AT&T Bell Laboratories Technical Memo, 1994.

[30] K. Sagonas and J. Wilhelmsson. Mark and split. In *Proceedings of the International Symposium on Memory Management*, pages 29–39, 2006.

[31] P. M. Sansom and S. L. P. Jones. Generational garbage collection for Haskell. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 106–116, 1993.

[32] F. Smith and G. Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the International Symposium on Memory Management*, pages 68–78, 1998.

[33] SML# compiler. http://www.pllab.riec.tohoku.ac.jp/smlsharp/.

[34] D. Spoonhower, G. Blelloch, and R. Harper. Using page residency to balance tradeoffs in tracing garbage collection. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 57–67, 2005.

[35] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[36] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[37] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, pages 1–116, 1995.

[38] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 87–98, 1990.