A new Mono GC

Paolo Molaro
lupus@novell.com

October 25, 2006

Current GC: why Boehm

Ported to the major architectures and systems

Featurefull

Very easy to integrate

- Handles managed pointers in unmanaged land
- Some support for typed objects
- Finalization semantics mostly match
- Weak references

Hightly tuned

Thread-local alloc

Current GC: why not

We don't need or use some of the features

- Incremental (handling of signals...)
- Resource usage (public API)

Finalization semantics not the same as needed

- Finalization of objects involved in cycles (which order?)

Weak reference support

- Track resurrection

Heap fragmentation

Zeroing overhead (atomic alloc ...)

Pause times

New GC: needs

Exact finalization and weak reference semantics

Small heap sizes and low-pause time (desktop apps)

Object pinning (automatic and API-controlled)

Fast allocation (bump-pointer and inlinable in managed code)

Large object handling

Precise type tracking

New GC: needs (continued)

Interior pointers on the stack

Appdomain unload issues (free all the objects in the appdomain)

Allocation of non-moving objects (interned strings...)

Nice to the user (no big memory chunk)

Still allow easy embedding of Mono

- Keeping the only reference to a managed object in the C stack is fine

Thread support

Implementation

Generational

- Old generation and nursery (the new generation)

Moving

- From the nursery to the old generation
- Compacting of the old generation

Stop the world collection

Large object space

- Collected with mark&sweep during major collections

Fixed heap

- For interned strings (and maybe later other types)
- Collected with mark&sweep during major collections

GetHashCode ()

Objects now can move, can't use the object address as hash code

Data stored in the lock word of the header

When the object is locked, the hash code is moved inside the fat lock structure

Load+check+shift in the fast path

- Additional check + load otherwise

Interlocked op when first setting the hash

mono_object_hash() for runtime hash tables

Pinned objects

How to create them:

- corlib API: GCHandle
- embedding API: mono_gchandle_new (obj, TRUE)
- fixed C# statement (PINNED flag for local IL vars)
- during P/Invoke calls

Additional cases

- objects and pointers found in the untyped C stack of the runtime and embedding applications
- objects the runtimes doesn't want to move (interned strings, Thread objects, currently Type handles)

Few of them in practice

Pinned object issues

Will fragment the heap

Will cause more collections

Will slowdown allocations and collections

Make sure you pin objects only when really needed and for as little time as possible

Runtime enhancements:

- Type information about registers and managed stack frames will allow to consider the referenced objects not pinned
- Unmanaged stack frames will still pin referenced objects for hassle-free runtime code and mono embedding

Pinned objects: finding them

Given a (possibly interior) pointer

Objects are allocated sequentially

- Need to start from the beginning of an heap area, visiting each object until the correct one is found
- During allocation, every 4KB save the object start
  - Need to scan at most 4 KB of memory to find object
  - 4 KB size is subject to change
  - The pinning addresses are sorted and the last pinned object is cached so usually this scanning is very fast

When found set the PIN flag or add to pinned objects array

The nursery

Most objects born here

Typically 512 KB - 2 MB for common apps

- Servers might want several Mbs

Bump-pointer allocation style

- Fast and can be inlined in managed code

When full, trigger a minor collection

- Unless a major collection is needed (old generation is full, too)

Divided in smaller chunks due to fragmentation or thread-local allocation

Nursery and pinning

Pinned objects fragment the nursery

Chunks of free nursery space between pinned objects used for allocations

When the nursery is completely pinned, allocate objects in the old generation

- Alternatives include: enlarge the nursery, allocate a new nursery

Pin objects only when aboslutely needed for short amouts of time

Track references precisely on the stack

Minor collection

Stop the world

Identify pinning objects

Scan the roots (including pinned objects)

- Copy to old generation as you go
- Place pointer to new copy in the old object's place

Scan the copied objects (they are roots, too)

- Check the finalization and weak-ref lists

Prepare for new allocs

Restart the world

Poke the finalizer thread

Stopping the world

Needed to not allow the mutators to see non-coherent data in objects

- forwarding pointers
- flags in the object header
- two copies of the same object

No support for safe points (yet)

Implemented with signals

- win32/OSX have proper OS support

Less than 1ms for heaps up to 100MB on1.6PM

Parallelize major collection later

The roots

Static managed variables

Registered roots (runtime and embedding data strutcures)

Handle tables

Remembered sets

- Old-generation objects referencing nursery objects

Pinned objects

Runtime stack

- Need type info for registers and managed stack frames

How to copy an object

Ensure we have enough room in the old generation

We need to ensure every reference points to the new object

For each pointer field

- obj.field = copy_object (obj.field);
- Place a forwarding pointer in the header
  - It's the pointer to the new home of the object
- If an object has a forwarding pointer it's already copied
- Pinned objects return the same pointer in copy_object()

The marking stack

We need to recursively copy the objects referenced by copied object

Recursion not a safe thing in the GC

- The runtime stack could be very small and the recursion very deep

The copied objects area is an explicit stack

Once the roots are scanned, scan the copied area, until no more objects are copied

- Gray objects are marked but the fields have not been traced

Finalization

When the roots are scanned, the non-copied objects are garbage

- unless they are pinned or need finalization

Copy finalizable objects to the old generation so they survive

- Recursively copy from their fields, too
- Loop until no more finalizable objects are found dead

Put finalizable objects in a separate list

Weak-refs don't need copying

- handled immediately

Prepare for new allocs

Create list of free fragments in nursery

memset the memory to 0 bytes

- too much overhead doing it at each object alloc
- too much overhead doing it for the whole nursery
  - touches too much memory and trashes the data cache
  - do it only a fragment at a time
- we need unused nursery areas to be zeroed before collections for the pinned-objects finding algorithm

Assign fragments to threads

- to each as much as they need (and resources allow)

Major collection

Currently copy-based

- Later use mark/copy to reduce memory requirements

Identify pinned objects in the whole heap

Remembered sets are cleared

Scan the roots as usual

- Large and fixed objects are just marked by setting a flag in the object header

Sweep unmarked objects

Free unused sections

Finalize/Prepare/Restart

Object sweeping

Walk the list of large objects and free the unmarked ones

Unset the mark flag for the others

Fixed objects are in pinned chunks

- freed by putting in a free-list
- unmarked as well

Both the large object list and the number of fixed objects should be small

Memory sections

Heap divided in sections

More flexible than using a big virtual area

Easier for the user

- Expands as needed
- Fair to other code in the same program competing for virtual address space

Of course it's harder to implement

- some performance issues
- likely the default on 64 bit systems (huge virtual memory space)

Pinned chunks

Data structure for GC-internal data and for fixed object allocation

Only for small objects

Each page has objects of same size

Allocation using freelists

Page assigned to a size as needed

Limited fragmentation issues (few sizes, mostly runtime and GC-controlled)

Mini-Boehm GC included

Large objects

Expensive to move

Currently used for object size >= 64 KB

Collected only during major collections

- using mark and sweep

Allocated with mmap

- already cleared, no need to touch pages

Need for tuning knobs

- how much memory do we allow in large objects before a major collection?

Object layout

Keep references close to each other

- Better cache locality
- Better encoding of reference bitmaps

Degenerate cases...

- Doctor, it hurts
- We should set reasonable limits for untrusted code

The JVM does have a much better time with this...

- no structs
- no fixed or sequential layout

GC descriptors

Several types needed:

- strings
- vectors
- bitmaps
- run-length encoding
- large bitmaps

Fast handling of ptrfree objects and arrays

Stored in MonoVTable

Write barriers

Keep track of references to nursery objects from the old generation: they become roots for a minor collection

Complicated by structs

- a reference can be stored in the heap or on the stack
- copy of structs

Custom ones optimized for each case

- array copy
- struct handling

API/ABI changes

A few simple changes needed for runtime hackers and embedders

- Objects can move, so the GC needs to know about all references or they must be pinned someway
  – No more valid to know an object is kept alive and store a reference in malloc memory
- Field and array element setting must go through write barriers
  – Only references or value types with references
- Interior pointer issues
  – A pointer to the end of an object is not a valid interior pointer
- New API:
  – MONO_OBJECT_SETREF (object, fieldname, value)
  – mono_array_setref (array, index, value)

TODO list

Use mark-compact for old generation collection

- Reduces memory required for old gen collection

Use mark-table based write barriers

- Faster and inlineable

Thread-local allocation

- Removes lock overhead

Inline allocation in jitted managed code

- No managed <> unmanaged transitions

Precise stack walk

- Reduce risk of false positive GC references

Fix the runtime

Preliminary results

Still untuned, so expect improvements

- write barriers, thread-local alloc, alloc code inlining

Some (rare:-) test cases: 5-7x speedup

Usually (with thread-local-alloc): 10-50% speedup

Usually smaller heap (but minimum is larger)

Degenerate cases:

- many pinned objects
- many reference stores (write barriers)
- long linked lists

Kernel help

Thread start/stop

- need to be able to get context of stopped thread

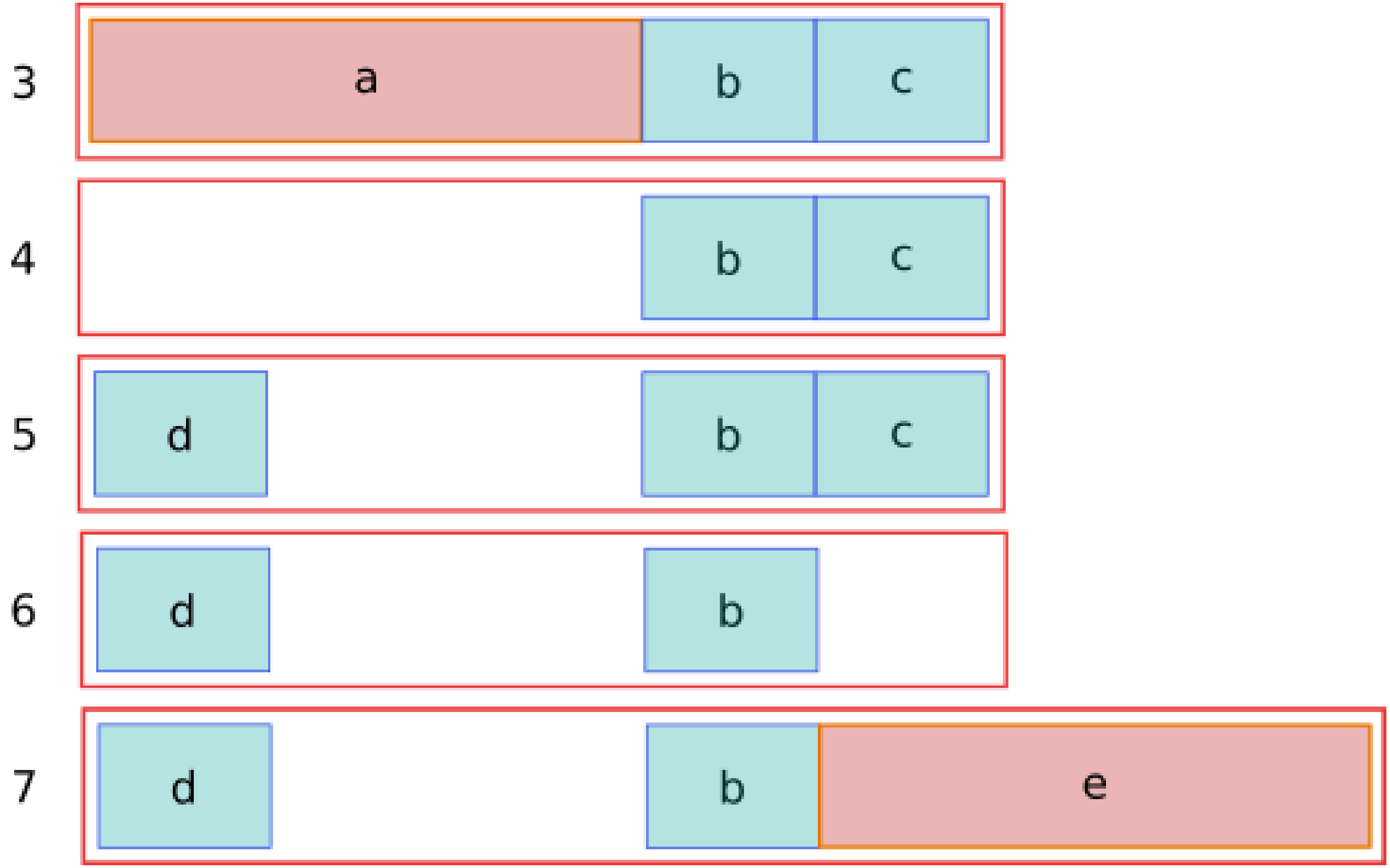
mprotect (addr, size, MAP_DISCARD);

- drop the pages from memory
- will be cleared when read again

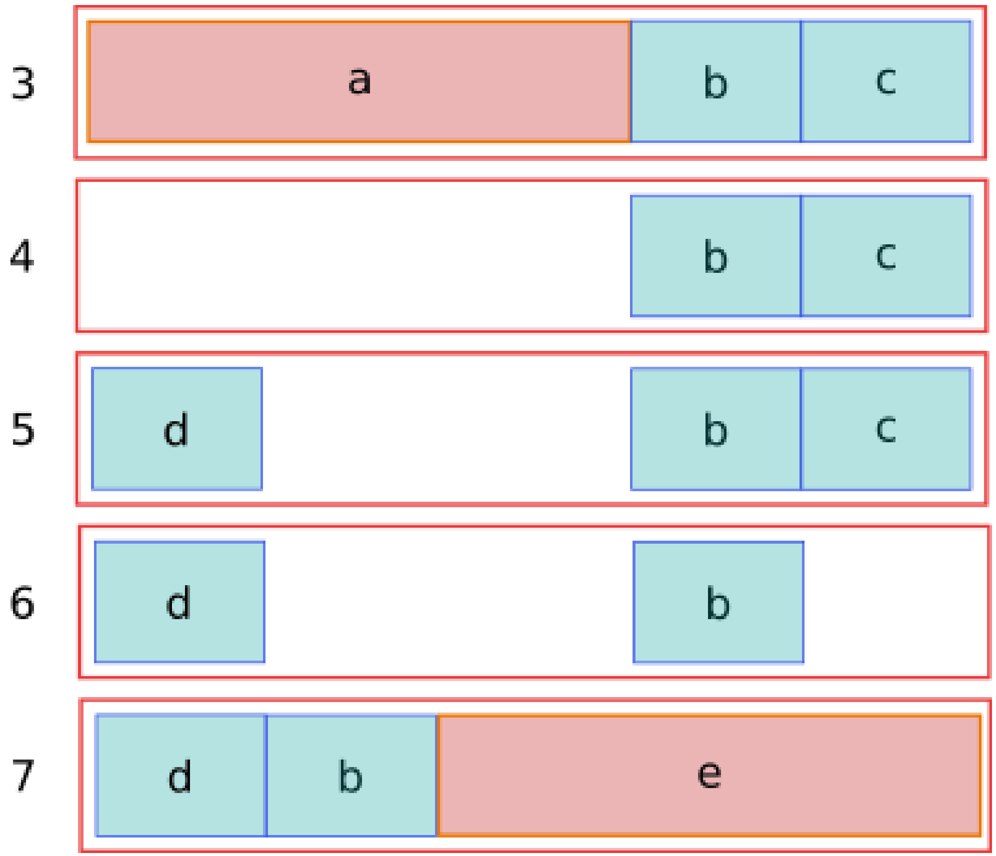
Page tables dirty bit access

- More coarse write barrier support
- No need for write barriers
- Fixed (and possibly too large) mark window

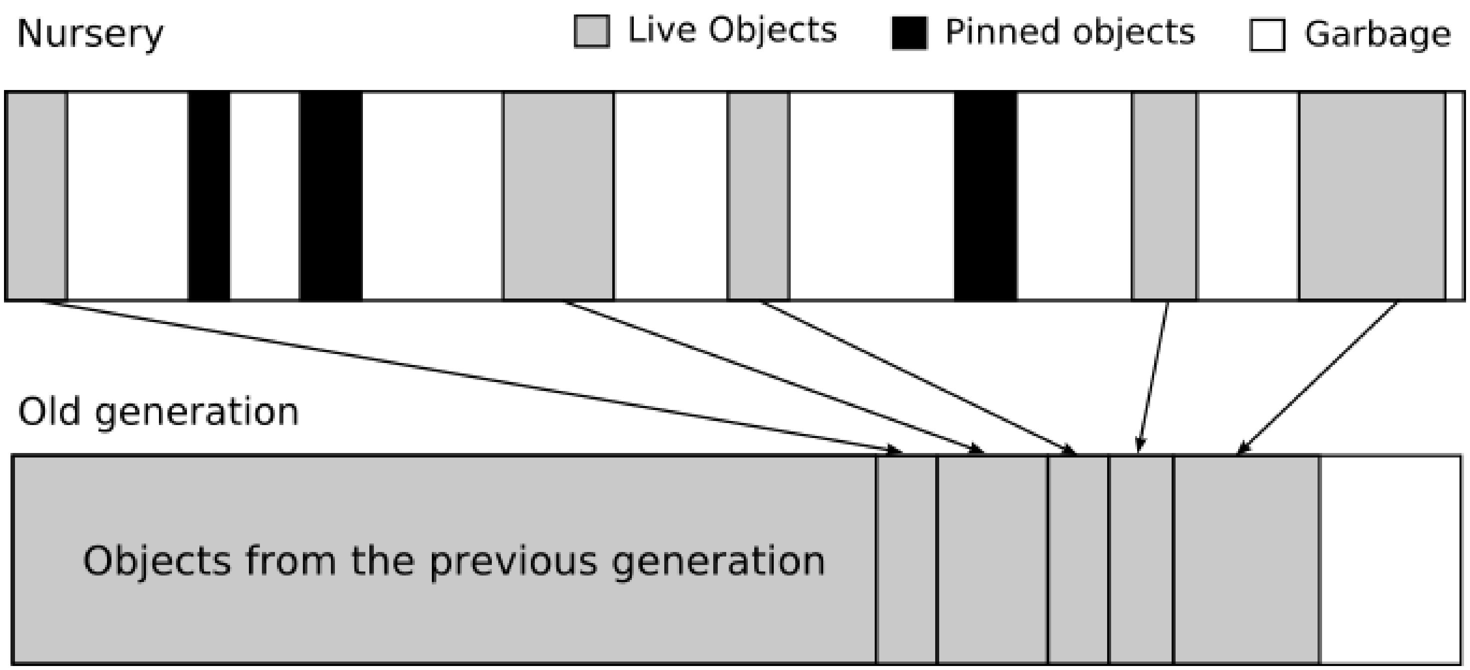
Compacting Garbage Collector.



Compacting Collector



Nursery.



Nursery
Live Objects
Pinned objects
Garbage
Old generation
Objects from the previous generation

Nursery and Pinned Objects.