Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm

Presented by Edward Raff

Some Terminology

- Backing Store: The page file on the disk / Swap space
- Tenuring: When a ~~*Professor*~~ Object survives long enough that it will probably be around for a while

Motivation

- Problems with GC
  - Stop the World.
    - Can be fine for mainframes and long running process. Not so good for application that need a fast response time
  - Current Algorithms do not know about Paging
    - May place common objects on different pages
    - May need to page in an object just to free it.
    - Paging is not a free lunch, and does not solve GC
- These are relevant issues to Personal Computing, and we want Smalltalk on every desktop!

Possible (Not)Solutions?

- Reference Counting: No pause times & no Paging issues
  - Pause times come back if we want compaction
  - Cyclic data structures need not apply.
- Mark Sweep
  - Scanning the objects thrashes the Page table.
- Scavenging (Incremental Semi Space)
  - Possible solution! But not as fast as we would like. Still not very Paging friendly.
  - Not as fast as we would like

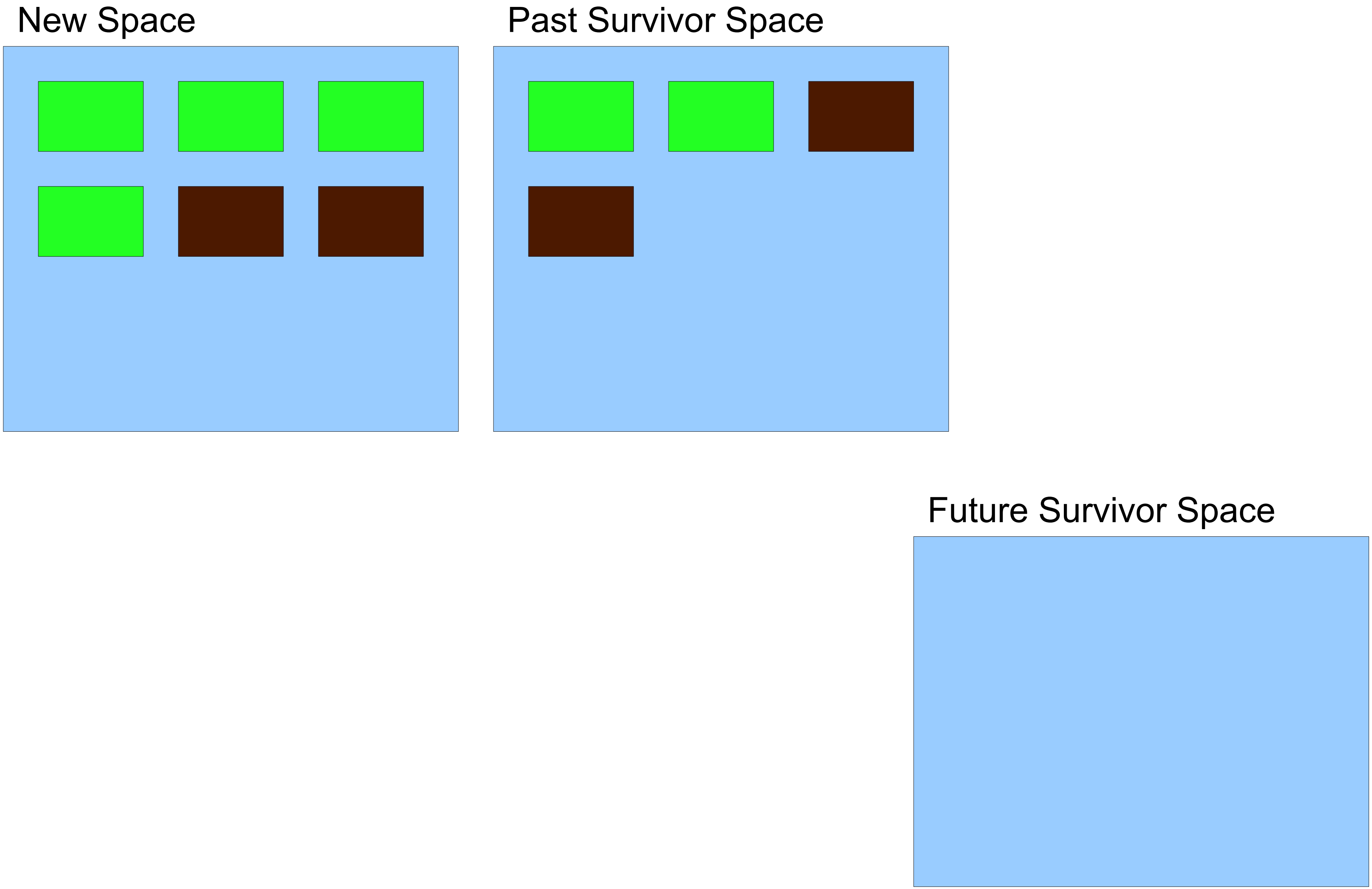
Tools

- Empirical observations we can apply
  - Most objects are short lived, Generational
    - Idea! New objects should never be paged out until they get promoted to an old generation.
  - We tend to allocate objects at a stead state.
    - Translates to reclaiming an average 7/8th of a byte per instruction
      - Exploit regression to the mean. If we just allocated an abnormally large number of objects, we can continue at the normal speed and be fine.

Solution: Generation Scavenging

- Segregate objects into Old and New.
  - Many modal → Bi Modal.
  - Old → New references get added to a *remembered set* (RS)
    - Stack Frames are always New
    - All live objects in the New space are a children of RS or registers
  - If an New object survives enough times, it gets tenured.
  - New Space is collected by Scavenging, Old space by Mark & Sweep
    - Starting to combine algorithms to get the best of both worlds (Immix combines 3 to get the best of 3 worlds)

New Area

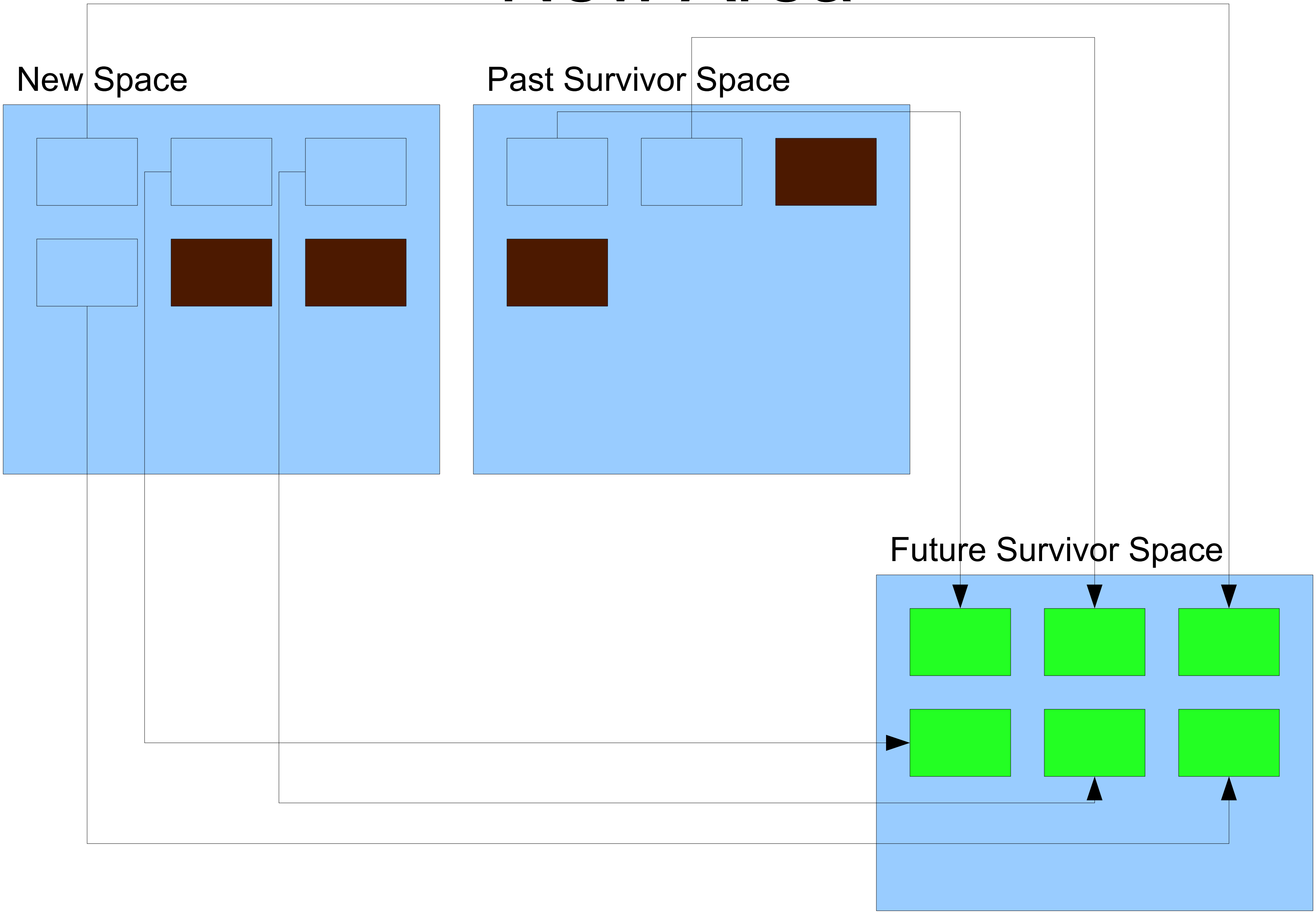


New Space

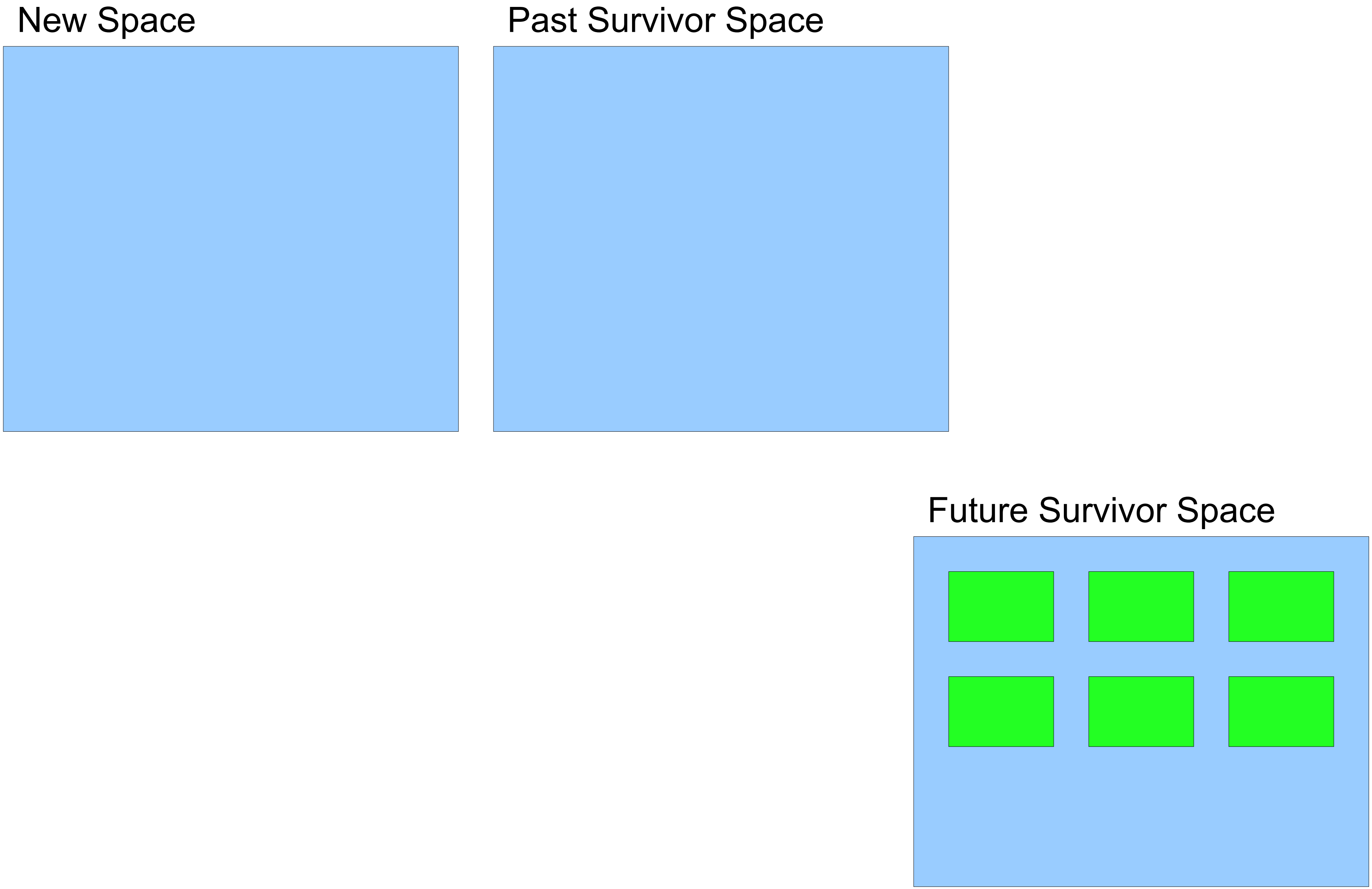
Past Survivor Space

Future Survivor Space

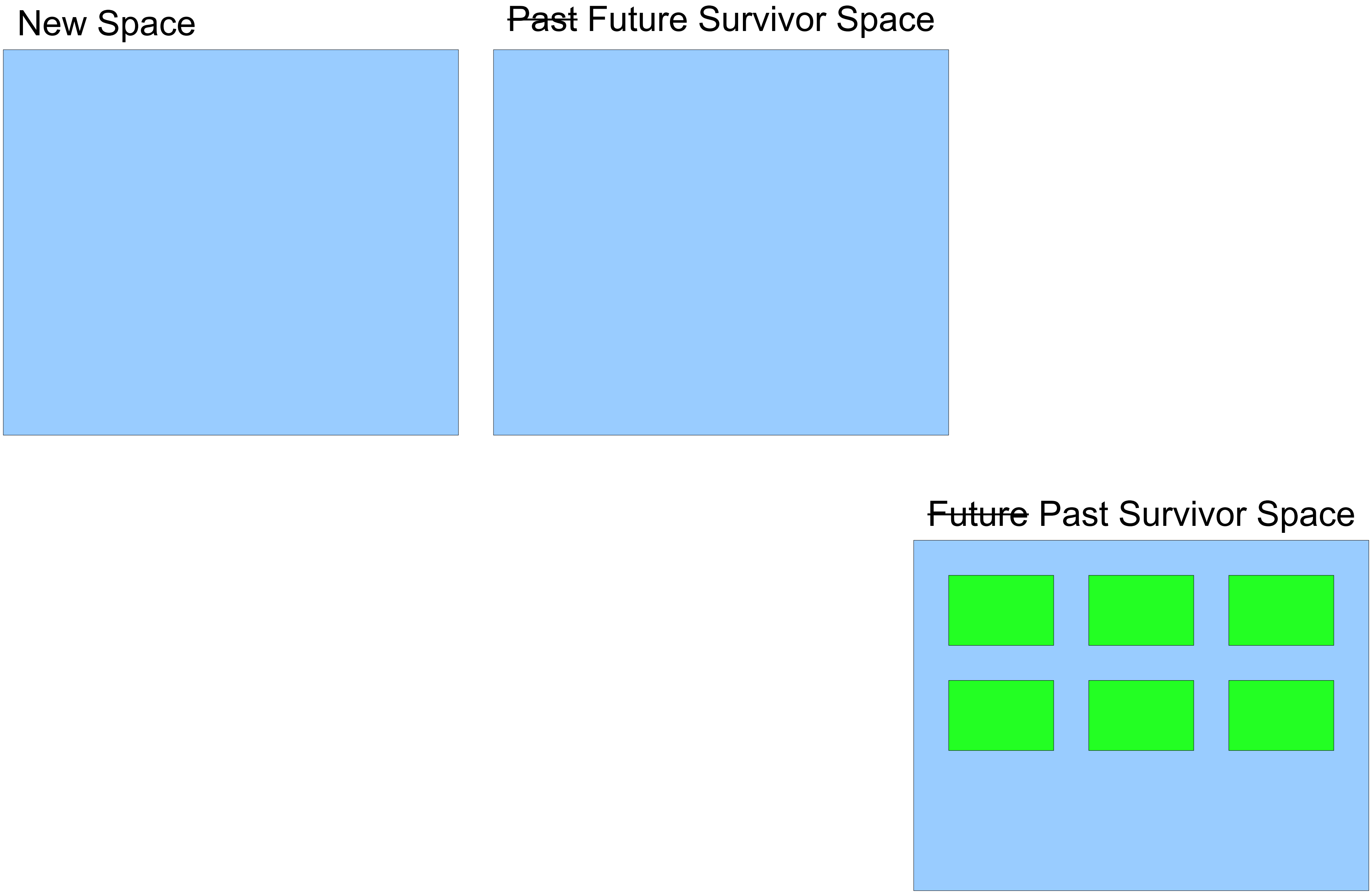
New Area



New Area



New Area



New Space

~~Past~~ Future Survivor Space

~~Future~~ Past Survivor Space

Why its better.

- Collection Time
  - New space is O(# Live Objects)
  - Old space is O(# Dead Object)
- Pause times small enough to be unnoticeable for Personal computes [But not for real-time applications]
- Spend the least time doing GC work
  - 1.5% of CPU time, opposed to 7% for Semi Space and 15% for Reference Counting
- Lower Memory use then Backer's

Caveat Lector: “Let the read beware”

- Implementation does not actually lock the pages for the New space
- Performance artificially inflated by slower Smalltalk runtime
- Tenuring problem, some ~~people~~ objects get tenure even though they become garbage soon.

Hardware Support

- Building a CPU (SOAR) with special instructions to make Smalltalk faster
  - Jazelle: ARM support for Java byte codes
  - The added instructions are tailored to their Smalltalk implementation

Strengths

- Lots of statistics, a small amount of theoretical work added in (Predicting CPU time without running it on the hardware..)
- Good idea, start of Nursery and Old generation concept (opposed to just generational).

Weaknesses

- Source of statistics are not explained, what test applications were run?
- No mention of the parameters used, let alone the method of determination
- No mention of the weaknesses or short falls in their method
  - Old generation is Mark Sweep, and will still cause page faults.
  - Pathologically bad *types* of programs?