The What, The Why and the Where To of Anti-Fragmentation

Mel Gorman

Andy Whitcroft

IBM LTC – Ireland & UK

Legal

- This work represents the view of the author and does not necessarily
- represent the view of IBM.
- IBM, IBM (logo), e-business (logo), pSeries, e (logo) server, and xSeries
- are trademarks or registered trademarks of International Business Machines
- Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service
- marks of others.

Overview

- What is fragmentation
- What is anti-fragmentation
- Why do we care
- Anti-fragmentation Implementations
- Linear reclaim
- Results
- Future direction
- Questions

What is Fragmentation

- External fragmentation is the inability to grant large contiguous allocations even if enough pages are free overall
- Binary buddy allocator is fast but behaves poorly in the face of fragmentation
- Result is high-order allocations fail after the system is running for some time

So why care?

- HugeTLB pages cannot be allocated long after boot
  - Result: Variable page support is relatively primitive
  - Little data available on time spent with TLB misses
- Memory hot-remove is almost non-existent
  - Patches exist, operation usually fails
- Drivers must use small pages
  - Many operations must be artificially split for those without “real” hardware

Large pages are desirable

- TLB misses are expensive but may be reduced with greater TLB reach
- Some studies have shown performance increases between 3% and 60% for some workloads *when enough large pages were available*
- Stream (memory bandwidth benchmark) on POWER5(TM)
  - ~30% gain backing data & text segments
  - ~19% gain using large pages for malloc()
- Gains/Regressions on x86 are CPU dependant

Memory hot-remove is desirable

- Virtualised environments can grow and shrink their memory as demand requires
- Currently dependant on the balloon driver

Anti-Fragmentation Vs Defragmentation

- Defragmentation is an active process for moving pages around in memory to rearrange currently free memory into contiguous blocks
- Anti-fragmentation keeps the system in a state where page reclaim will free memory in contiguous blocks
- Defragmentation is expensive, might not work
- Anti-fragmentation can incur a runtime cost and might break down depending on implementation and workload

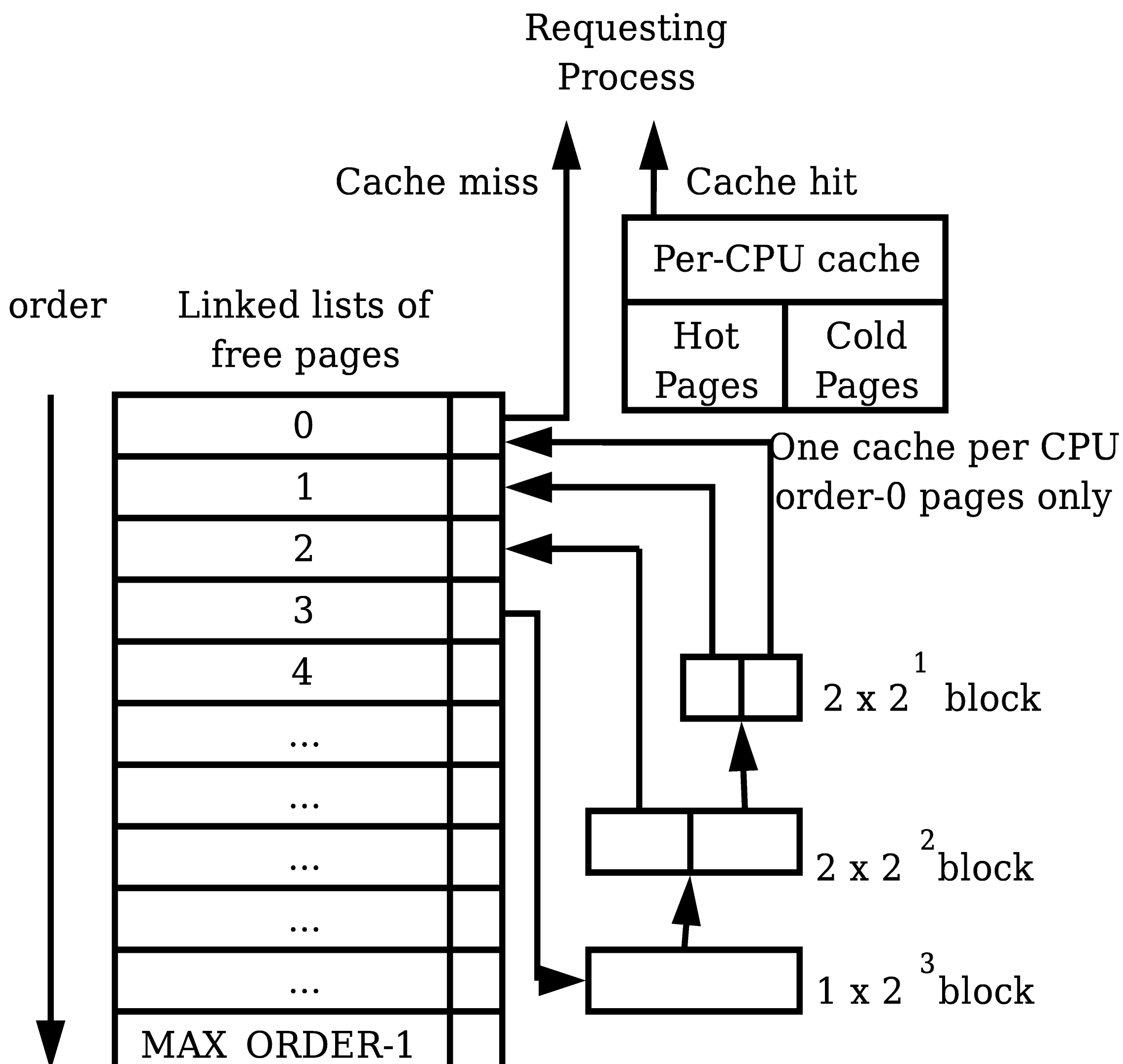
Anti-Fragmentation Vs Defragmentation

- Full fragmentation avoidance requires knowledge of the future.
- Alternatively, all allocated memory must be reclaimable by the kernel.
  - Change use of physical addresses to virtual addresses
  - Drivers required to return all memory on demand
  - Smart pointers
  - Not going to happen

Anti-fragmentation

- Cluster allocations of related types together
  - Clustering by size or allocation time is not suitable for an OS
  - Types are Kernel Non-Reclaimable (KernNoRclm), Kernel Reclaimable (KernRclm) and Easily Reclaimable (EasyRclm)
- On reclaim, the KernRclm and EasyRclm areas should free as contiguous blocks
- Flag caller type with GFP flags

Current Buddy Allocator



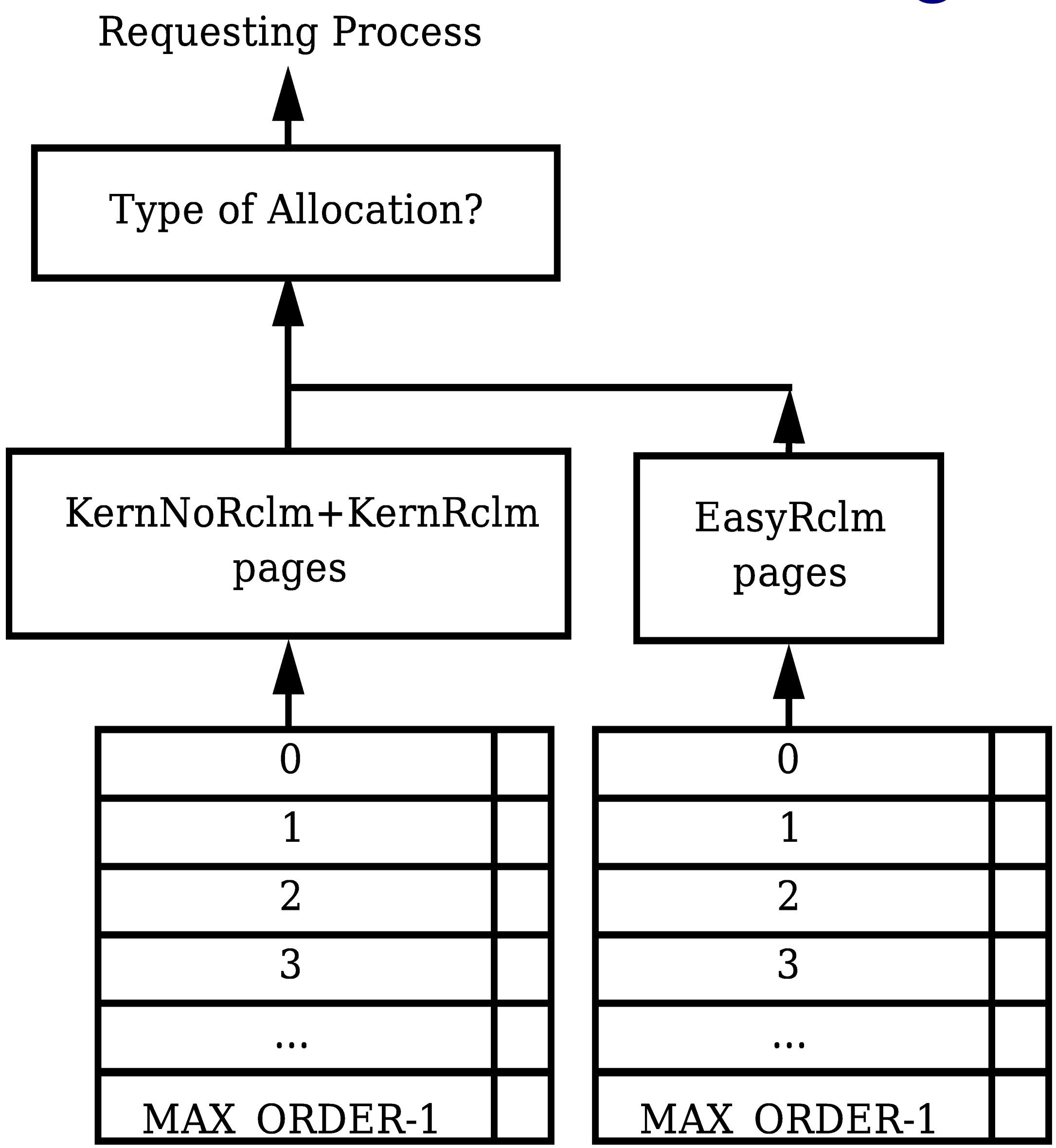
First Implementation: List-based/sub-zones

- Normally one free page list per order per zone
- Add one list to split kernel and user allocations
- Similarly add additional list for per-CPU allocator
- Allocations use their preferred free lists
- Otherwise fallback

List-based/Sub-zones: Fallback

- Simply fallback to the “other” list
- Always try and steal the largest free block to minimise future fallbacks
- If splitting a large block, free buddies are placed on allocation types free lists

List-based/Sub-zones: Diagram



List-based/sub-zones: Fallback

- Most important that KernNoRclm allocations do not fallback often
  - Kernel allocations can spike depending on the workload – updatedb an obvious culprit
  - On desktop loads, caching data is a large percentage of the persistent allocations

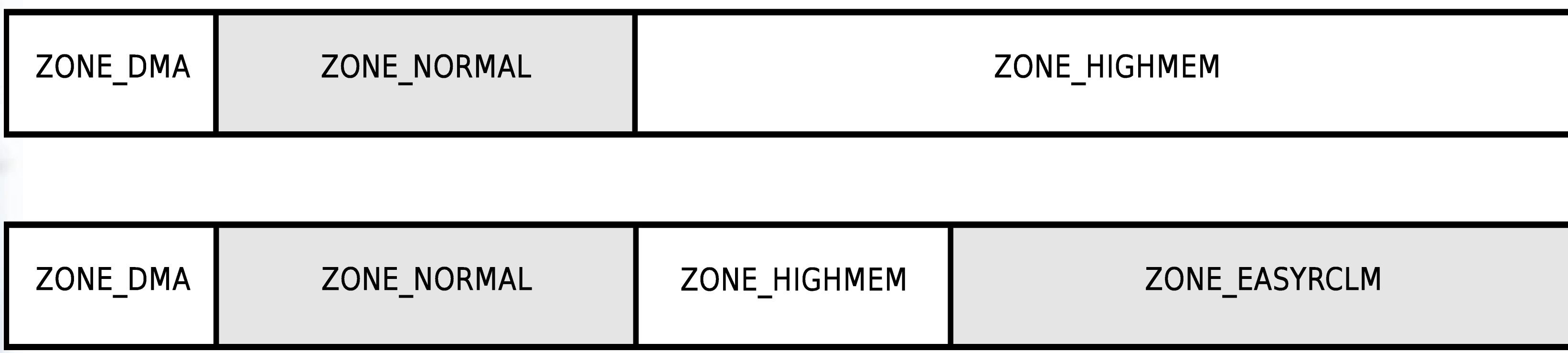
List-based/sub-zones: Result

- Overhead in the fast path
- Broke down over time, regressed to standard behaviour after stress testing
- Got hammered on lkml
- Considered too complex for little gain
- Suggestion to implement the same idea with zones

Implementation: Zone-based

- Create ZONE_EASYRCLM
- Split allocations into kernel and user allocations
- Place user allocations in EasyRclm zone with fallback allowed to kernel zone
- Do not allow kernel to fallback
- Sizing zones was architecture-specific mess
  - Led to development of arch-independent zone-sizing

Zone-based: Diagram



Zone-based: Comparison to list-based

- More robust than list-based – guaranteed availability
  - RDMA may be a problem
- Does not affect the hot-path
- Simpler implementation
- Requires configuration at boot-time
- Inflexible, only helps large page allocations

Zone-based: Status

- Trying to get arch-independent zone sizing into -mm
- Tests show that the zone can reliably allocate large pages within that zone
- Dynamic huge page pool resizing patch built on top

List-based: Revisited

- Revisited because of zone-based inflexibility
- One free page list per order per zone
- Add two more lists for KERNRCLM and EASYRCLM
- Similarly add additional lists for per-CPU allocator
- Allocations use their preferred free lists
- If free page is unavailable, fallback

List-based Revisited: Fallback

- Trickiest part of implementation to get right.
- Order of fallback determined by allocation type.
- Always try and steal the largest free block to minimise future fallbacks. Free buddies are placed on allocation types free lists.
- If large block is being split, move all free pages from that MAX_ORDER_NR_PAGES block to the allocation types free lists.
- If a kernel allocation, reclaim all EASYRCLM pages in that MAX_ORDER_NR_PAGES block.

List-based Revisited: Result

- Still complex
- Performance varies +/- 1%
- No longer breaks down during our tests
  - Requires min_free_kbytes to be about 10%
- *Substantially* higher success rates
- On desktop after several hours under load
  - 28% allocatable as large pages with mem=512MB
  - 37% allocatable as large pages with mem=768MB
  - 77% allocatable as large pages with 2GB!
- Retry merge with greater large page transparency

Linear Reclaim

- With page groupings, there is a reasonable chance reclaimable contiguous blocks exist
- LRU-reclaim could find them *eventually*
  - In tests, LRU-reclaim gave very variable results under load
- Linearly scan memory searching for blocks that are likely to be reclaimable and reclaim them
- Gave much more reproducible results under load
- Trashes less

Metrics

- Absolute availability
  - How many large pages can be allocated
- Unusable free space index
  - Indicates how much of the available free space can be used for a large page allocation
  - 0 == good, 1 == bad
- Fragmentation index
  - Measured at time of failure
  - Check if failure is due to no memory or fragmentation
  - 0 == good, 1 == bad

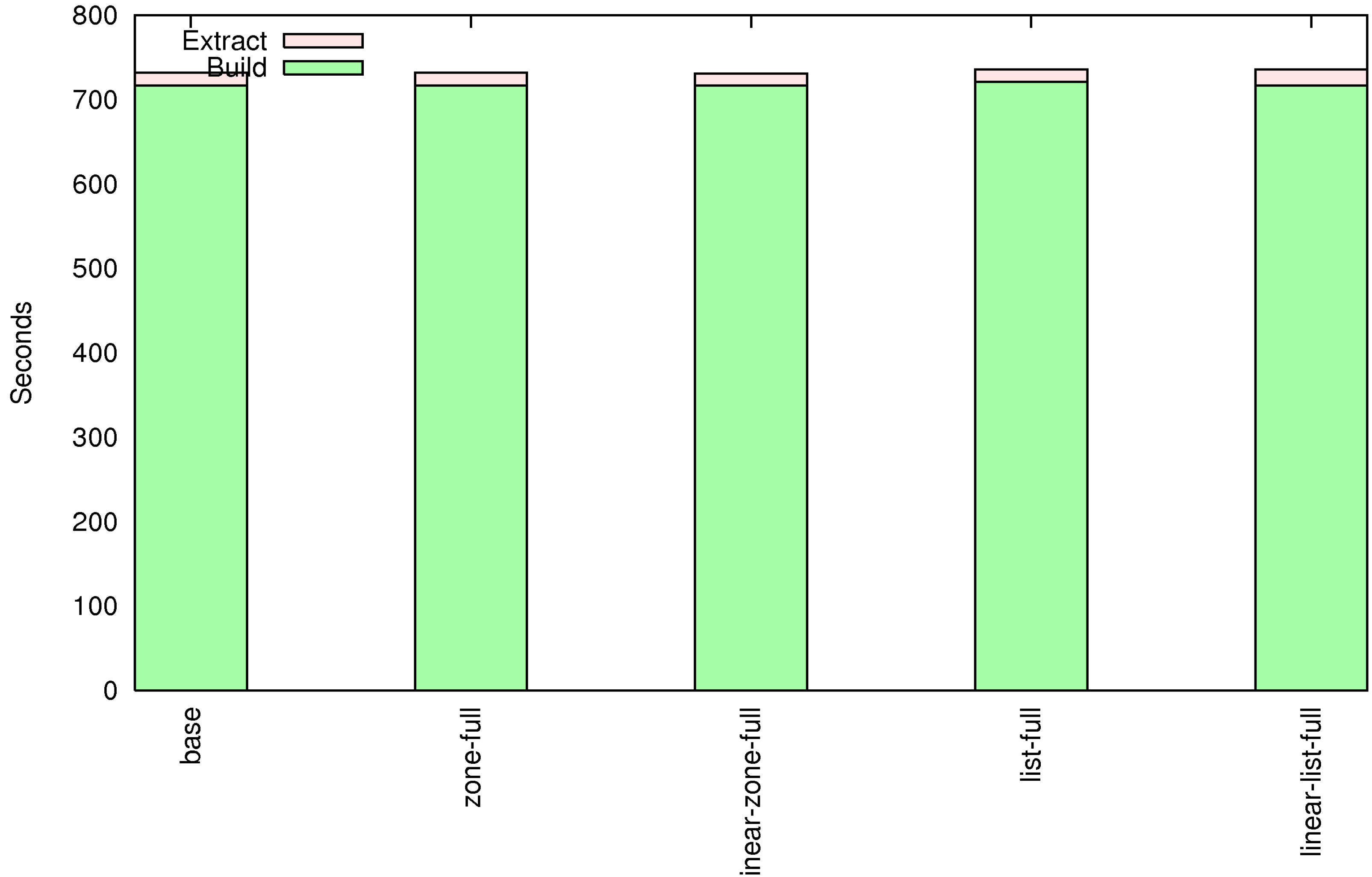
Test Scenario

- Expand and compile linux-2.6.14
  - Measure time to complete
- Run AIM9 as a microbenchmark on VM operations
- Run HugeTLB Capability Test
  - Compile one kernel in parallel for every 200 MB of memory
    - 7 simulataneous compiles on X86. 17 on PPC64
  - Allocate pages during compile
  - Allocate pages after compile
  - Allocate pages after dumping caches
- Run High Order Allocate Stress test
  - Compile one kernel in parallel for every 200 MB of memory
  - Allocate pages during compile
  - Allocate pages after dumping caches

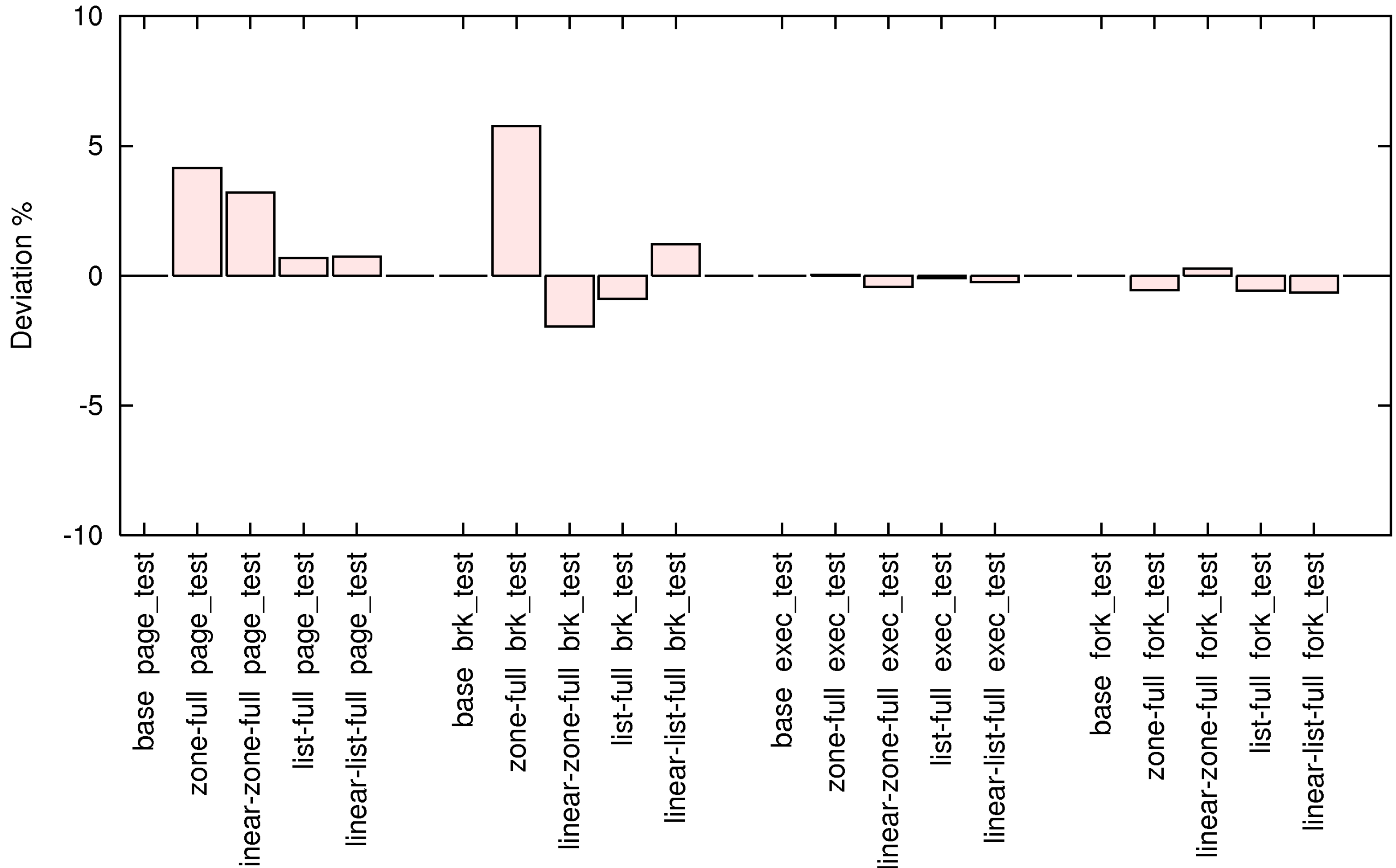
Results

- Results here are different from the paper
- In the paper, list-based broke down very quickly and was useless on PPC64.
- Figures for list-based here are the “Revisited” implementation
- Zone-based figures based on systems with 30% memory given over to ZONE_EASYRCLM

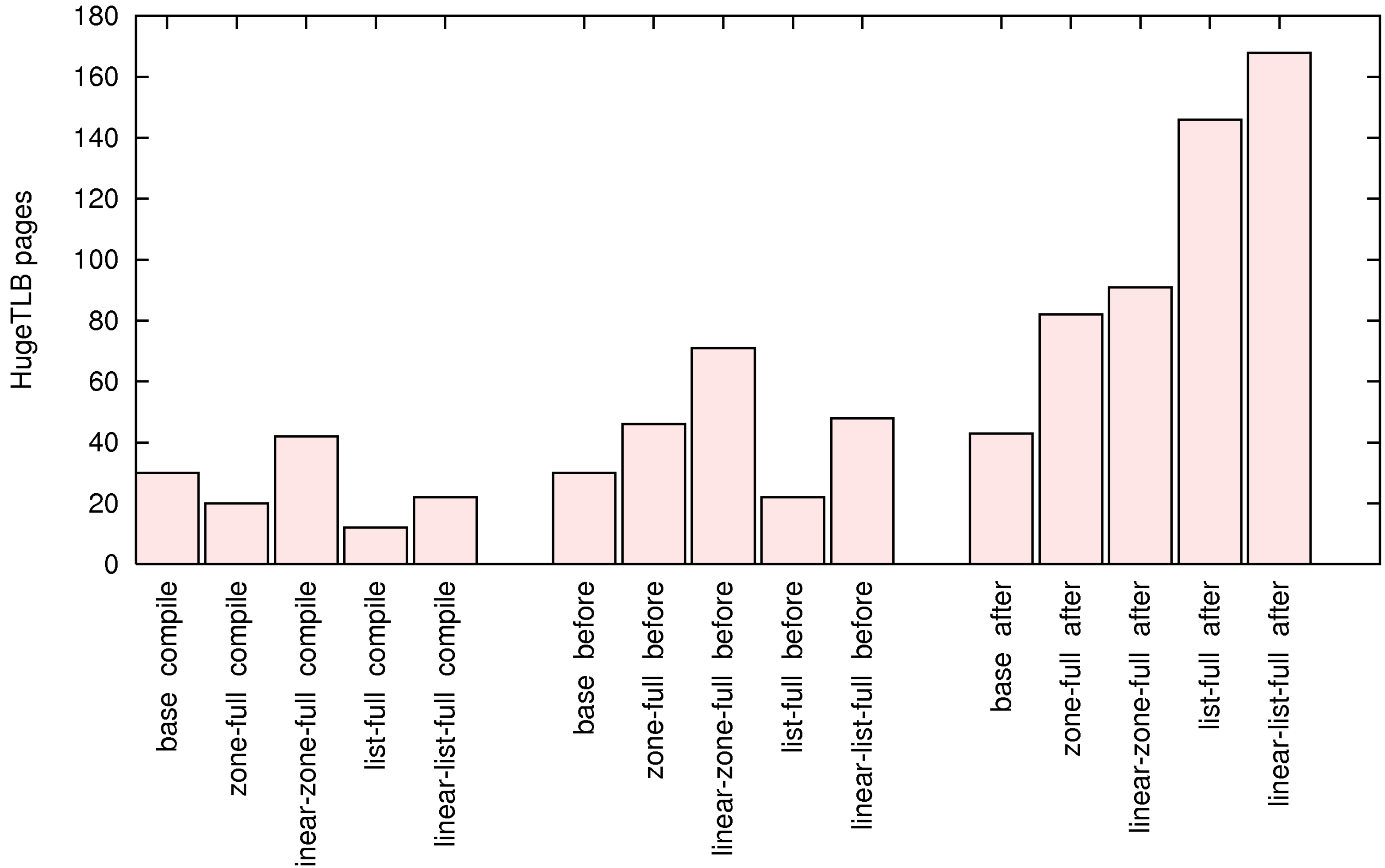
PPC64 Kernel Build timings



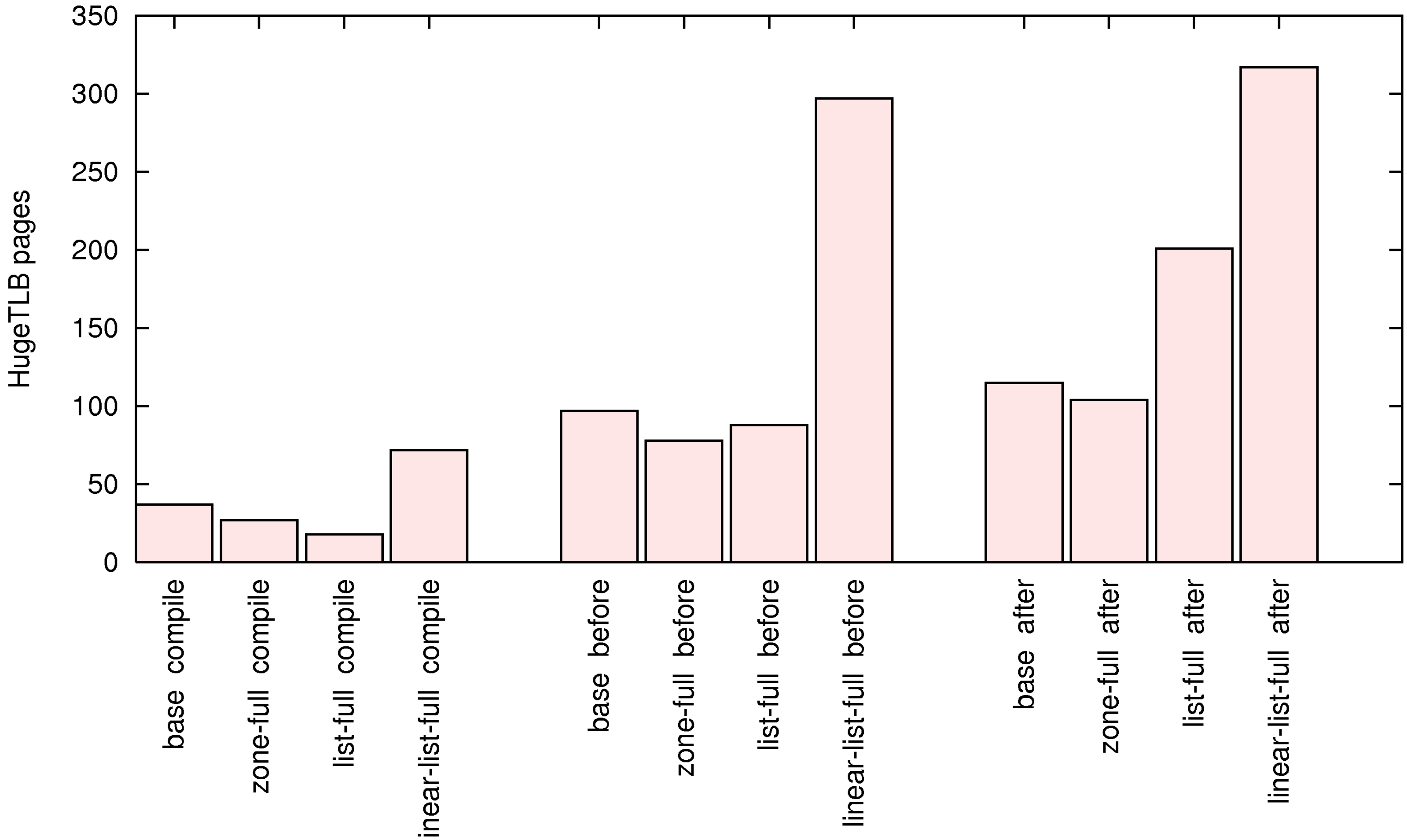
PPC64 AIM9 Results



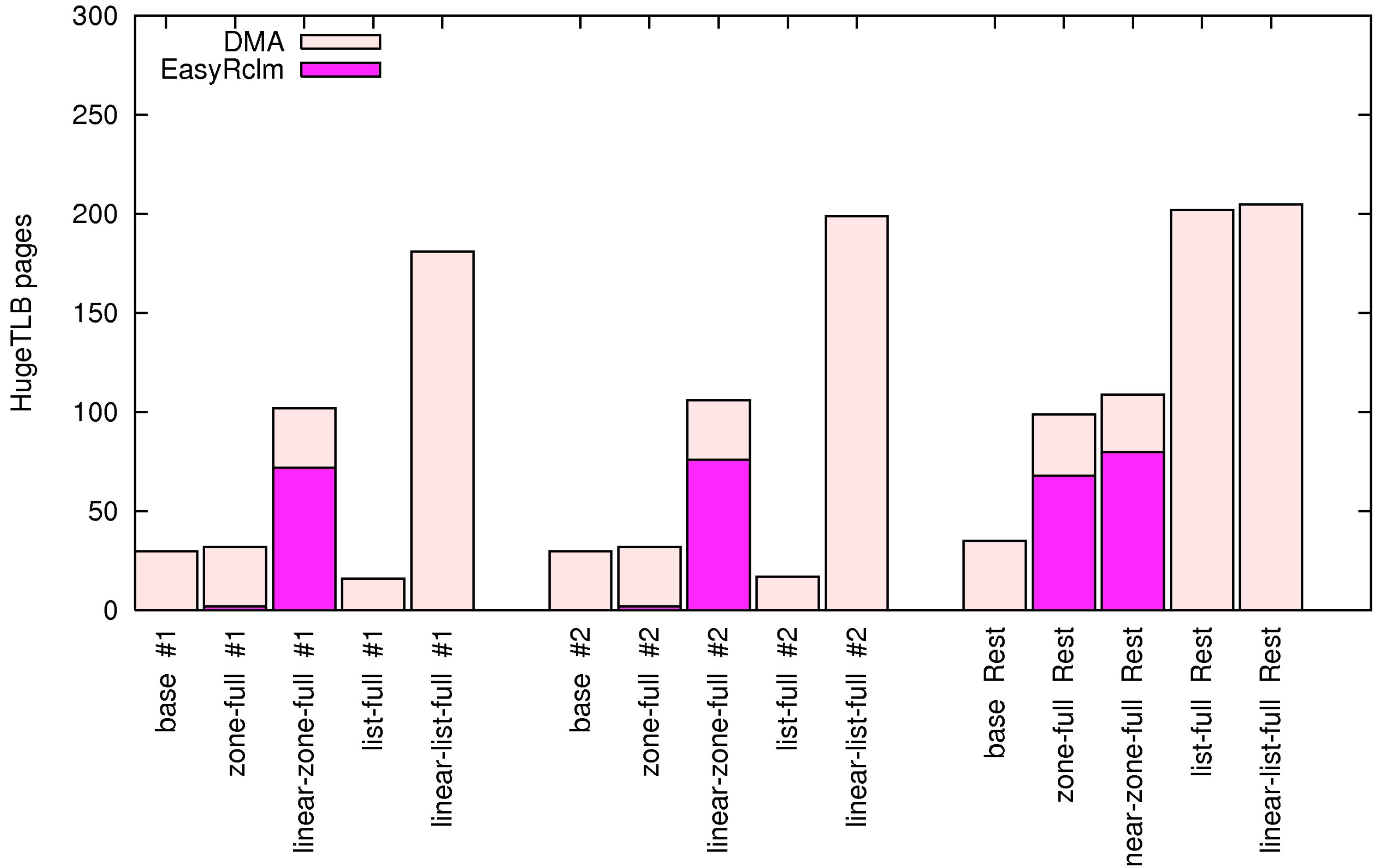
PPC64 HugeTLB Allocation via Proc



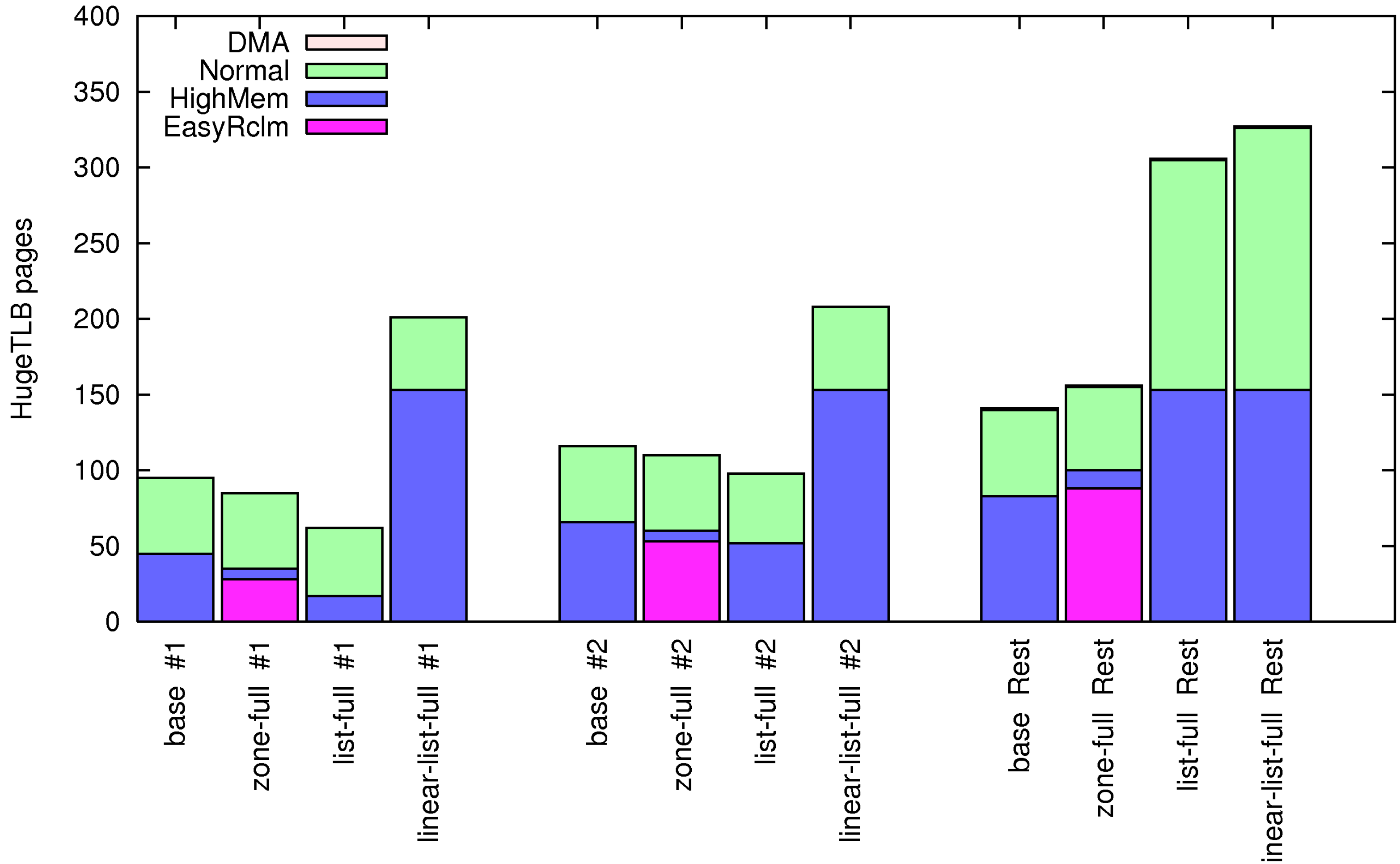
X86 HugeTLB Allocation via Proc



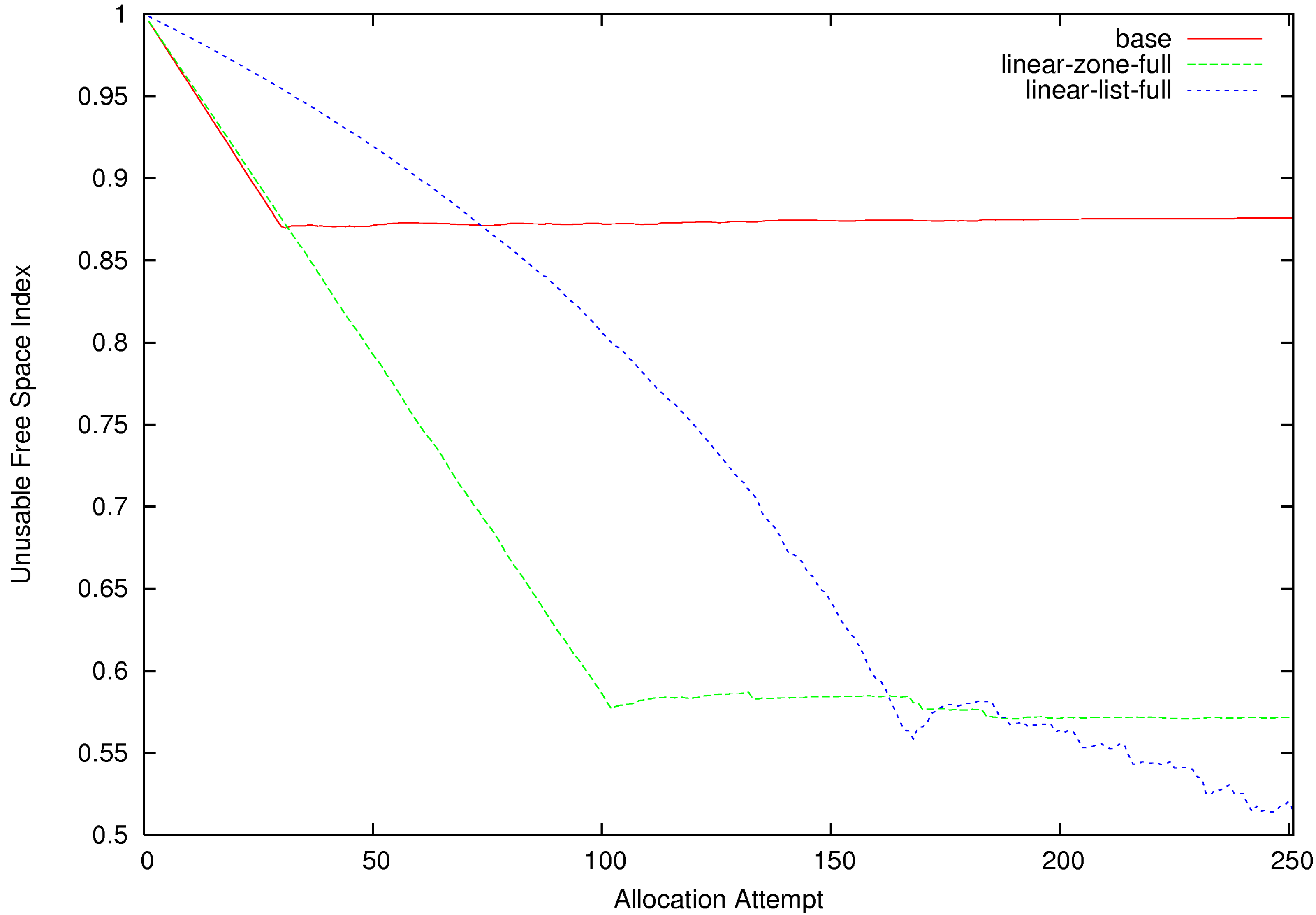
PPC64 Stress Large page alloc



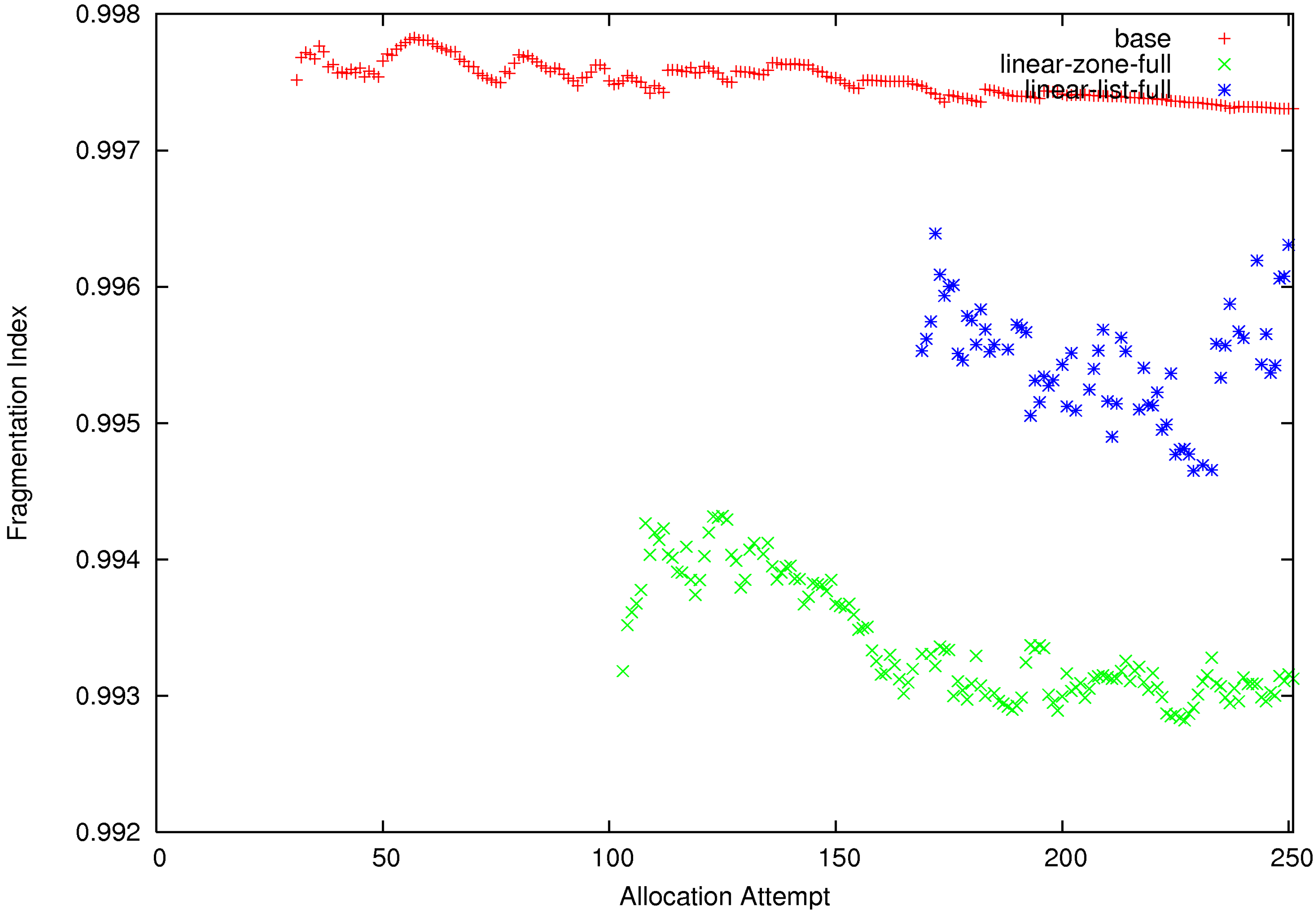
X86 Stress Large page alloc



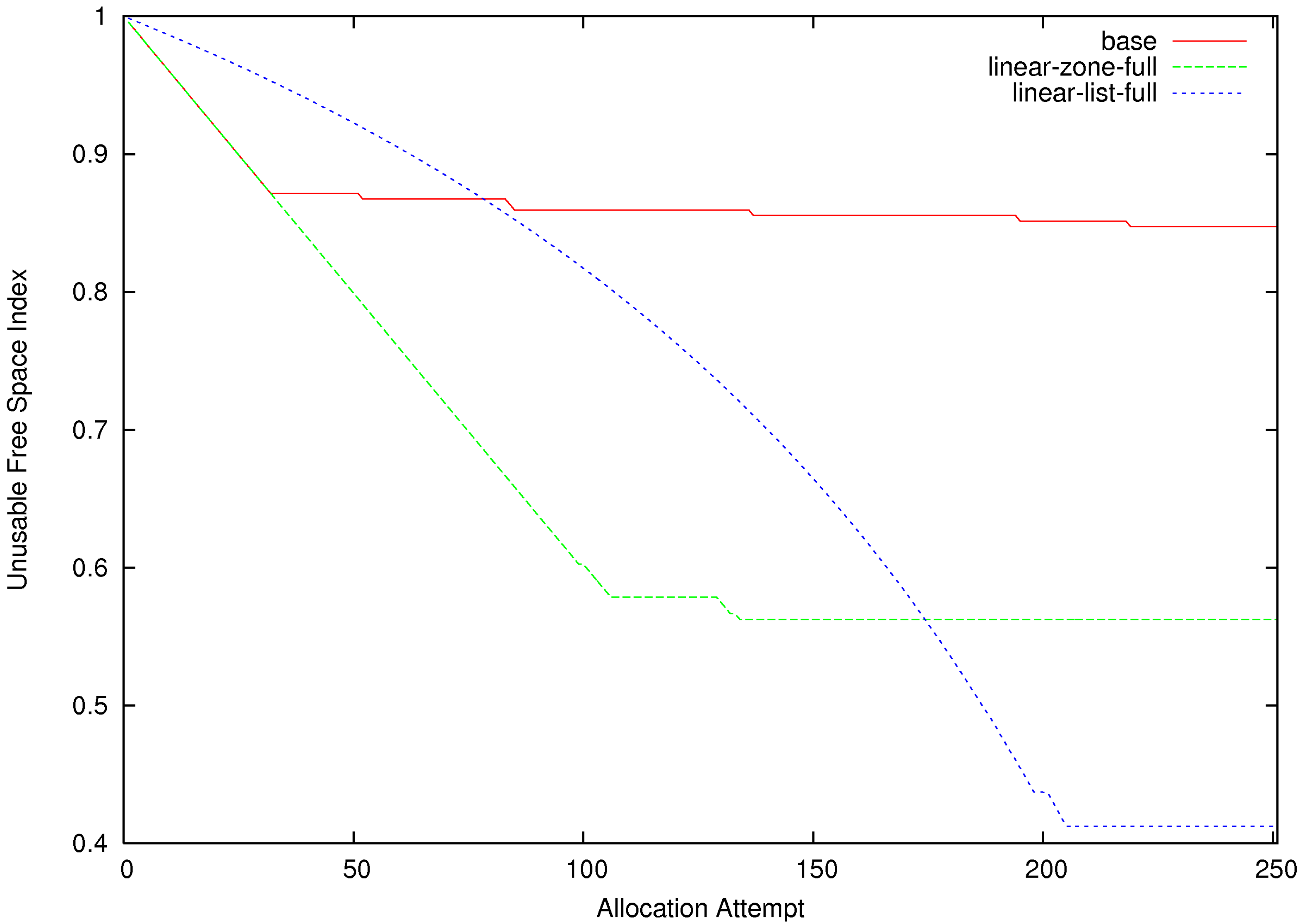
Unusable Free Space Index: Compile



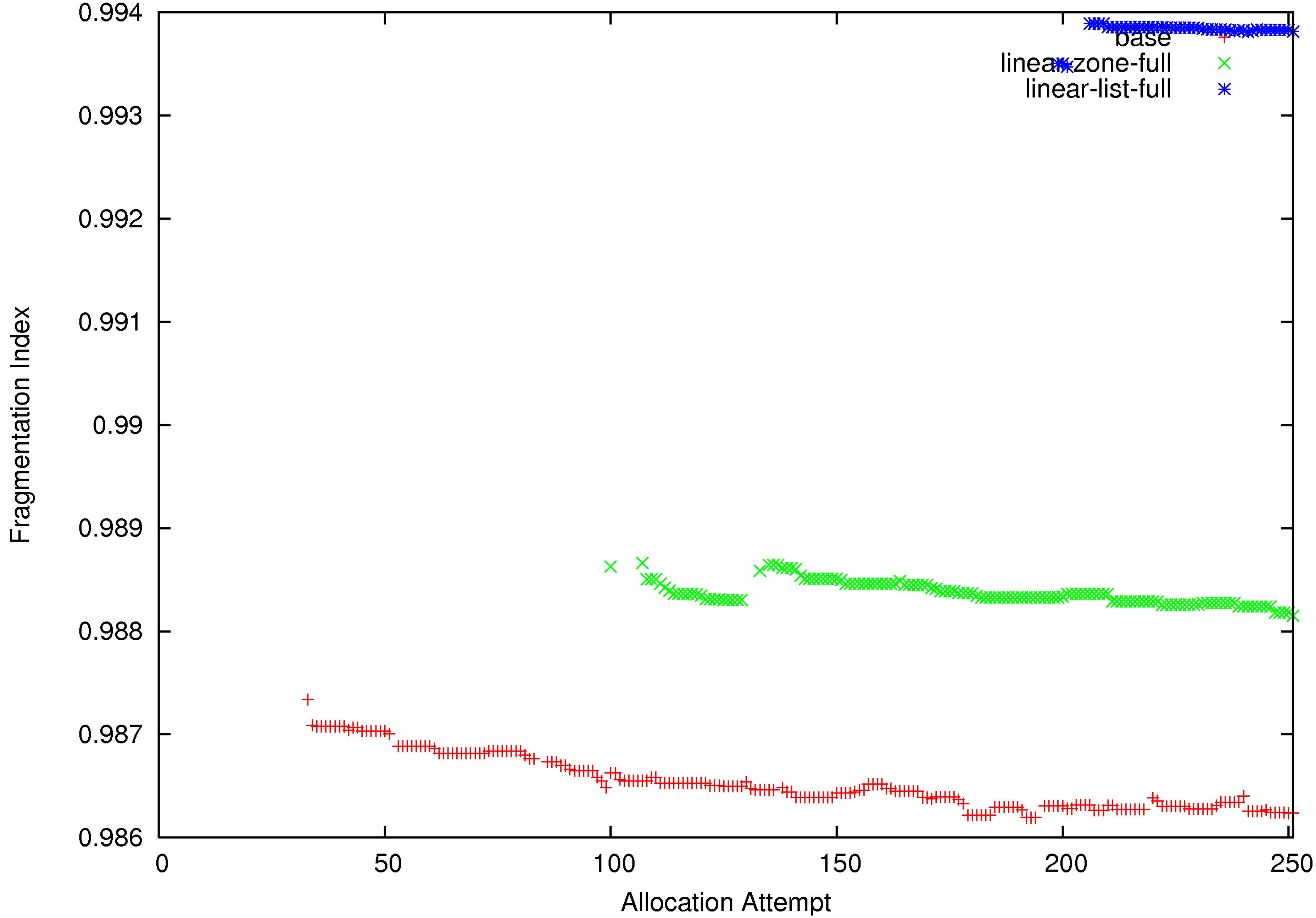
Fragmentation Index: Compile



Unusable Free Space: Rest



Fragmentation Index: Rest



Dynamic Huge Page Pool Resizing

- Patch to allow the huge page pool to grow and shrink
- Restricted to the size of ZONE_EASYRCLM
- Fairly reliable
- Unreleased because of number of pre-requisite patches

Future Direction

- Try and get zone-based integrated
  - slow going, lot of churn in -mm
- Make zone-based a bit more flexible
- Revisit hotplug remove for supporting hot-remove of ZONE_EASYRCLM
- Work on greater transparency for large pages
  - BIG job here
- Work on benchmarks that justify use of large pages

Questions, comments, flamage?

Backup slides

Old list-based Vs New List-based

- Old list-based
  - HugeTLB pages at end 0 large pages
  - Stress high-order alloc 14 large pages
    - Almost totally useless
- New list based without reserve
  - HugeTLB pages at end 99 large pages
  - Stress high-order alloc 71 large pages
    - Decaying slowly
- New list based with reserve
  - HugeTLB pages at end 145 large pages
  - Stress high-order alloc 202 large pages
    - No longer decaying
- New list based with reserve and linear-reclaim
  - HugeTLB pages at end 158 large pages
  - Stress high-order alloc 206 large pages
    - Note that linear helped allocate more HugeTLB pages via proc