Everywhere you imagine. RENESAS

Superpages Revisted:

Transparent Application of Large TLBs on Embedded Systems

2009-04-06

Renesas Technology
Paul Mundt
paul.mundt@renesas.com
lethal@linux-sh.org

The Coverage Problem

- TLBs on embedded systems tend to roughly be around 32 or 64 entries on average.
  - PAGE_SIZE is generally 4k or 8k.
    - 8k is commonly seen on platforms with aliasing VIPT L1 caches that wish to avoid aliasing altogether.
- These same systems support a wide range of page sizes to maximize TLB coverage in a controlled environment.
  - Very few embedded platforms make use of this fact in the kernel today.
  - The classic embedded problem, the desire for small page sizes as well as a reduction in the amount of time spent servicing TLB misses.
    - While generally spending the entire product lifetime under heavy memory pressure, often without swap.
  - For anyone unfortunate enough to be taking notes, this is what we are focusing on today.

The Coverage Problem

- Most of these systems lack global PTE bits and must wire TLB slots with fixed translations to preserve across a TLB flush.
  - A scheme often used/abused for fixmaps.
  - This all depletes entries and reduces coverage even more.
- Some systems employ special support for fixed section mappings.
  - Often PMD-granular section mapping.
    - Which generally see a reduction in permission bits over their PTE counterparts.
  - Other times extra TLBs are added.
    - These may include more emphasis on caching behaviour and less on access control.
      - A common case for mapping lowmem cached/uncached and flipping section bits in ioremap().
    - And may or may not include a miss exception.
      - Generally pre-faulted with fixed mappings, or piggybacked on top of the first-level TLB miss.
    - Some of these will use their own page tables.
  - Making superpage promotion/demotion even more tedious.

Ways that large TLBs are presently used

- ioremap() and friends
- cached/uncached lowmem mapping
- hugetlbfs/libhugetlb
- Sparsemem vmemmap
- Architecture “extensions” for abusing software-managed TLB entry loading.

Ways that large TLBs are presently not used

- Dynamically
  - ie, Page frame coalescing for order promotion and other academic evils.
- Generally
  - Huge pages and large contiguous physical blocks of memory are a scarce resource, and generally cease to be available shortly after system startup.
    - Fortunately workloads are generally fixed, access patterns and behaviour is well understood, and performance degradation is quantifiable.
  - Combined with things like ZONE_MOVABLE, it's still possible to get back to a state where large TLBs can still be used often enough to provide a measurable performance boost.

Overview of the SH-4A/SH-X2 MMU Architecture

- Semi-harvard TLB layout.
  - 64-entry Unified TLB containing I/D-TLB entries.
  - 4-entry I-TLB, loaded from the U-TLB and managed entirely by hardware.
- PMB (Privileged Space Mapping Buffer)
  - 16-entry section-mapping TLB.
  - No miss exception, so must be either statically configured or loaded from a special page table piggybacking the TLB miss.
  - Limited access control capabilities, only usable for kernel mappings.
    - Usable for user mappings if using single-address space mode for uClinux on MMU parts where the role of the MMU is scaled back.
- Wide range of page sizes.
  - 4k/8k/64k/256k/1MB/4MB/64MB in TLB.
    - 512MB also on SH-5 TLB, generally used for lowmem mapping.
  - 16MB/64MB/128MB/512MB in PMB.
- 64-bit PTEs for extended TLB mode, primarily for extended protection bits, which also necessitates a 64-bit pgprot.

ioremap() and friends

- Nothing terribly interesting to note here.
- The largest possible sizes are used and scaled down from.
  - Often establishing both PMD and PTE mappings, depending on size.
- This is also a tie-in for multiple TLBs.
  - SH-X2 and later parts first attempt PMB mappings to avoid building up TLB-bound page tables unnecessarily.
  - Has the undesirable side-effect that things like unmap_vm_area() have a lot of linear scanning to do before finding TLB-bound page tables (if at all) when doing PMB tear-down.
    - Not a hot path by any stretch of the imagination, so only mildly bothersome.

cached/uncached lowmem mapping

- Greatly simplifies ioremap() and friends in that caching attributes for a given mapping can be adjusted by flipping the high bits.
- Some SH parts (and most MIPS, too) support identity-mapped segmentation with varying cache attributes natively without having to go through the TLB.
  - While this makes changing caching attributes easy and greatly reduces TLB misses, it also wreaks havoc on available virtual address space, negating the usefulness of things like sparsemem vmemmap, crimping vmalloc space, etc.
  - Others simply set up mappings to provide the same functionality.

hugetlbfs/libhugetlb

- Generally requires static reservation of entries at boot time.
- Variable page sizes are supportable, but the optimal sizes will generally depend on target workload.
- While there is no “one size fits all”, and huge pages are a scarce resource, they are also the simplest interface for applications getting a handle on large TLBs.
  - Still requires careful profiling of the workload to determine optimal number of reservations, sizes, etc.
- Most applications have no awareness of huge pages, so a more transparent option is needed for bridging the two.
  - So we opt for the simplest solution any time a minor userspace problem arises -- hacking the libc directly and telling applications what they really want.

libhugetlb and uClibc, together at last

- uClibc contains a pluggable allocator backend that makes experimenting with various implementations fairly trivial, which makes it a good match for experimenting with libhugetlb.

- With integration of libhugetlb in the allocator backend, large allocations that match the huge page size are automatically backed with huge pages if they are available.
  - This behaviour can be enabled/disabled explicitly for applications, or simply left to the hugetlb code to try and do the best job it can.

- Even with fairly dumb implementation, applications using large allocs for anonymous memory get a measurable performance gain.
  - In sample workloads this has resulted in gains from 5-25% on a loaded system.
  - If application access patterns are more random, or only small allocations are employed by the system workload, performance degradation is marginal.
    - However, memory is wasted if more pages are reserved than are used.

Sparsemem vmemmap

- Similar to the ioremap() implementation, where the mappings are established using the largest possible size and scaling down.
- Far too heavy on virtual address space to be of general use on constrained platforms.
- .. but a reasonable trade-off for platforms that value more effective TLB utilization over address space conservation.
  - If you aren't counting individual pages of available virtual address space while subsequently losing count of the number of bounce buffers, this can mean you!

Magical Architecture Extensions (MAE)

- Systems with software-managed TLBs allow for much greater flexibility in how the TLB miss is serviced.
  - Which may not even entail loading an entry in to the TLB that took the initial miss.
  - A combination of order recording and page hinting can trivially be used for platforms with enough free PTE bits to transparently construct a TLB entry that matches the entry on the initial page fault.
    - In practice this can be optimized down to a handful of additional instructions given that the page table walking is taking place regardless of order hint.
      - Presently this does not handle PTE->PMD transitions, which remains something to investigate in future work.
    - Page hinting introduced by s390 for aiding guests and hypervisors.
  - In the case of faulting in a section mapping, alignment hinting can offer suggestions whether walking an additional page table is worth the extra overhead or not.
    - In the case where access is mispredicted, overhead increases, but integrity is maintained as the TLB is still loaded as a fallback.

Magical Architecture Extensions (MAE)

- In the case of hardware loaded TLBs, fewer optimizations can be made, requiring PTEs to be order encoded, and requiring many more generic code changes.
  - While many optimizations can still be made for these platforms, the lack of software-management prohibits any intelligent decision making at TLB miss time.
  - On the other hand, it is not always a net benefit, as the system needs to support page sizes that match the most frequently used page orders.
    - It always depends on the workload.
  - Fortunately on embedded systems, these things are measurable!

Summary

- While there are many trivial options for using large TLBs more transparently on embedded systems, there has been little to no effort in this area.
  - In the past interest in this area has been an all or nothing.
  - Empirical testing on the other hand suggests that even making use of large TLBs for anonymous memory is measurable enough to make it worthwhile.
  - Even if TLBs grow larger and the coverage problem is lessened, systems will still prefer a tiny page size.
- For embedded systems in general, having control over the applications and workloads that the system is exposed to allows for careful profiling and figuring out where the bottlenecks are.
  - While being under constant TLB pressure is not often considered a big problem on modern systems, measurement has shown that there is still significant performance degradation on even the most typical embedded workloads.

Questions?

Concerns?