Linux Scalability Issues

Tim Chen

Andi Kleen

Dave Hansen

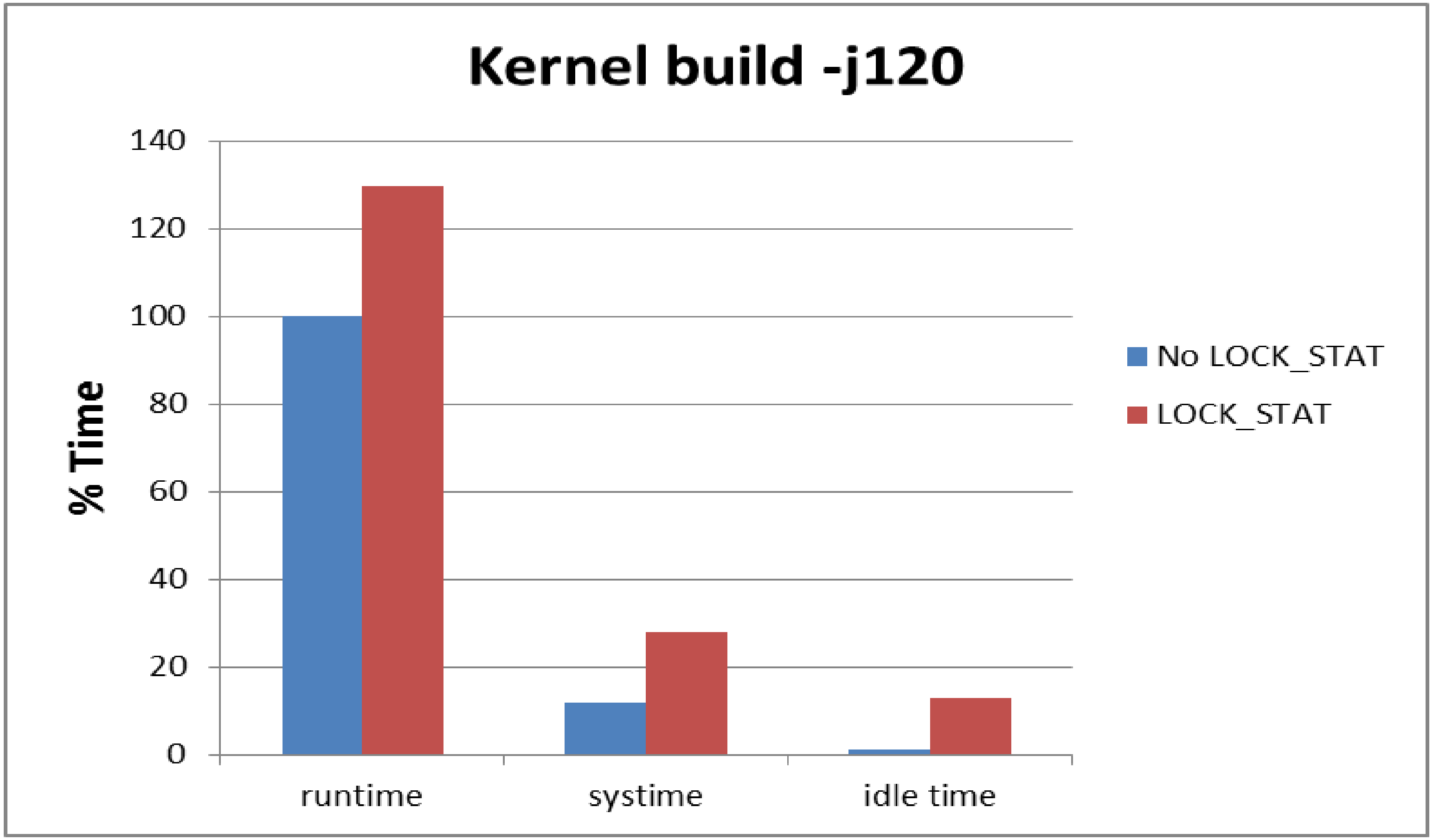
9/18/2013 @ Linux Plumbers Conference

The Enemy to Scalability

- Writes on shared data is the enemy!
- Writes to shared structures are expensive, be it a spinlock, r/w lock, atomic counter, etc..., Cache line bouncing between cpus really slow things down
- Even very short hold time on a lock is expensive.
  - e.g. A recent change put ext4 inode on a sorted LRU list for reclaim. LUR list lock caused a simple file copy workload putting page cache pressure spend >90% time in lock contention.
- Will like to have good tool that can be run with minimal overhead

LOCK STAT scales poorly (better tool?)

- LOCK STAT collection has large overhead due to the LOCKDEP infrastructure. Kernel build took 30% longer on a 60 core system with make -j 120. System time increase from 12% to 28% and idle from 1% to 13%



LOCK_STAT Need Improvements

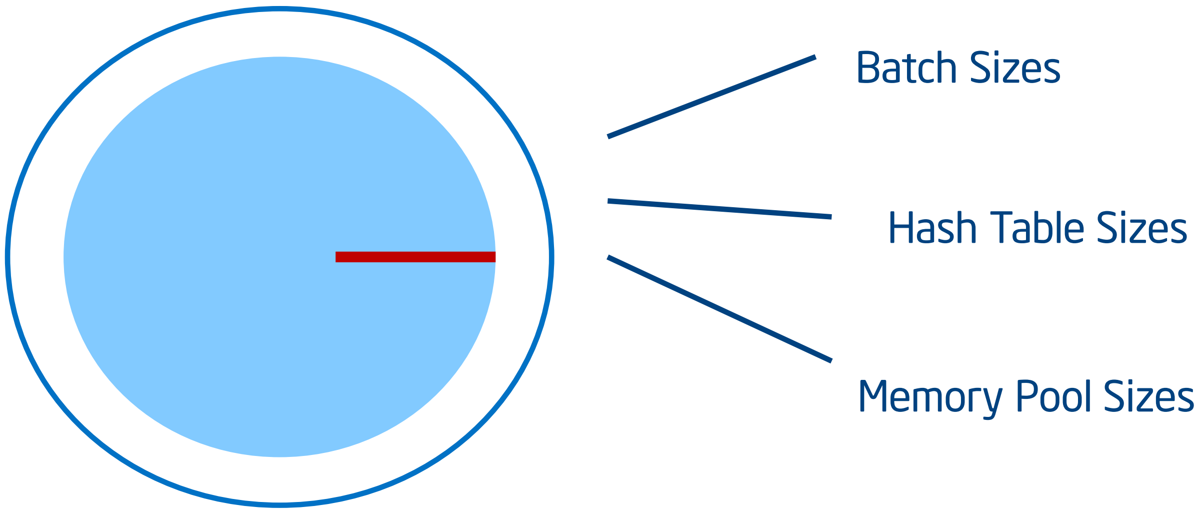
- Lighter weight without all the lock correctness check overhead, and changes workload behaviors. (e.g. rcu_read_lock became real lock with LOCK_STAT)
- Lower overhead allow usage on production kernel, very useful for debugging

```
+  16.29%            cc1  cc1                        [.] 0x0000000000433e1e
-   7.83%            cc1  [kernel.kallsyms]          [k]
__lock_acqu.reisrn.n1
   - __lock_acquire.isra.31
      - 99.83% lock_acquire
         + 19.56% __mem_cgroup_count_vm_event
        + 13.05% __mem_cgroup_try_charge
        + 7.87%  _raw_spin_lock
        ...
```

MAGIC Number Tuning

Make magic numbers scale according to machine size

- A single knob for humans
- Raw knobs for auto-tuning



MAGIC Number Tuning

- Lots of magic numbers sprinkled throughout the kernels
- Batch sizes
  - e.g. PAGEVEC_SIZE (=14) (batching LRU pages op), mem cgroup charge batch size (=32), TASK_RSS_EVENTS_THRES (=64, update mm counters every 64 pg fault)
- Hash Table sizes
  - FUTEX hash table size (4 bit, 8 bit)
  - INET listen connection table size (32), UNIX socket hash table size (256)
- Pool size
  - e.g. KERN_MSGPOOL = 37 , per cpu page pool (pcp capped 0.5M)

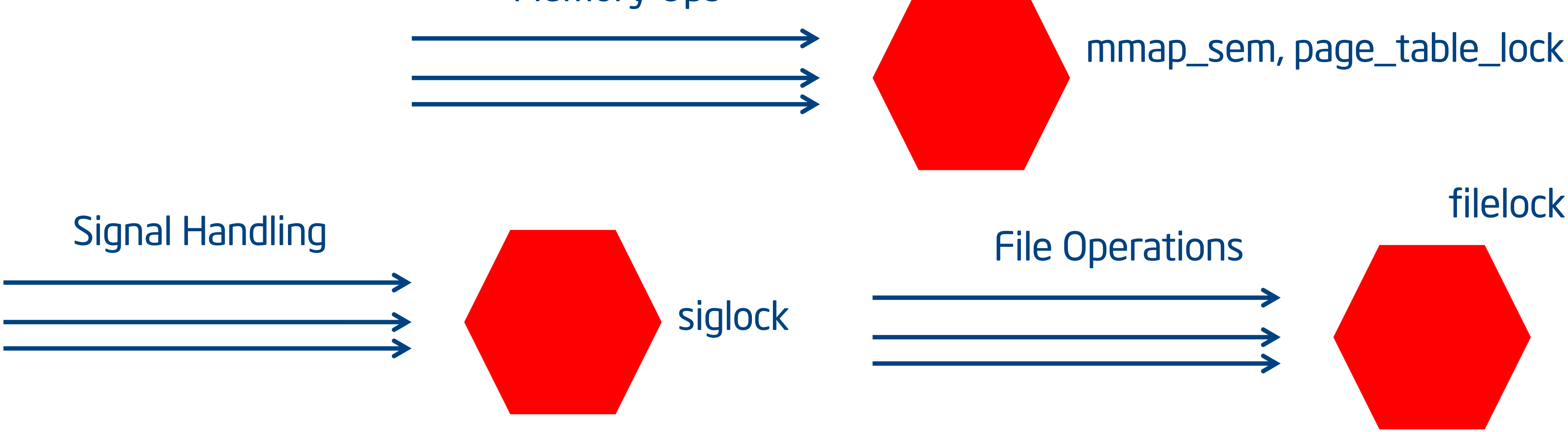
Coarse Grained Locality Framework

- A single shared structure is often cause for scalability bottleneck
- But locality granularity on a per cpu basis is too fine grained for many circumstances
- Will be useful to have a general framework to allow something in between
- Infrastructure that support shared data for a group of cpus

Multi-Threaded App Scaling Issues

mmap_sem, page_table_lock, sighand->siglock and file_lock are shared in task_struct

- Heavily mmap_sem, page_table_lock contended when multiple threads are doing page faults, mmap
- Contended when many signals are sent to individual threads in thread group, in multi-threaded applications (Per-thread siglock that doesn't slow down single thread case?)
- files_open management contending on file_lock when a lot of files open/close by each thread



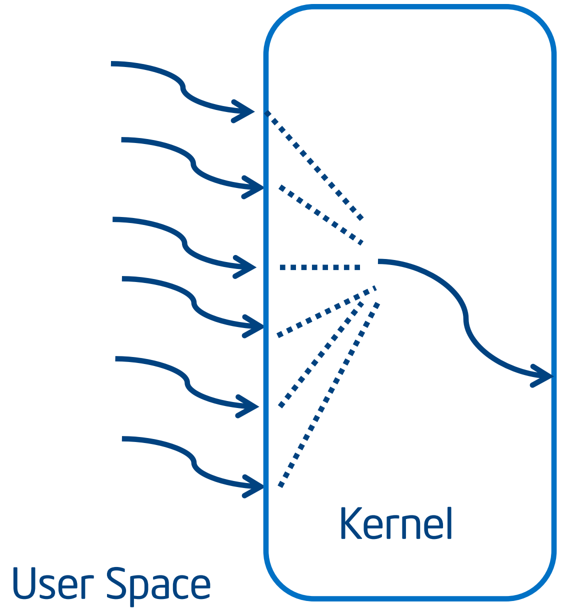
VM Scalability Issues

- Per Page Ops during TLB Flushing and Page Reclaim
  - Lots of cross CPU IPI, on a per page basis, not batched in shrink_page_list
- mmap_sem is contended too often for threads in a process
- Page fault has significant cost (page allocation & clearing)
  - there is *NO* way to avoid using page faults to populate file mappings. MADV_WILLNEED does _readahead_, but will not prefault anything
  - When we detect pattern of continued page faults, can we pre-allocate pages? fault pages in batch?
  - Have a pool of pre-cleared pages (with non-temporal instructions) so they are ready to use. Need evaluations to see if this will help as page cleared may need to be used immediately and be in cache.
- Fork operations contend on root of anon_vma tree for rmap. Current rw_sem did not perform as well as previous mutex and spin_lock implementation

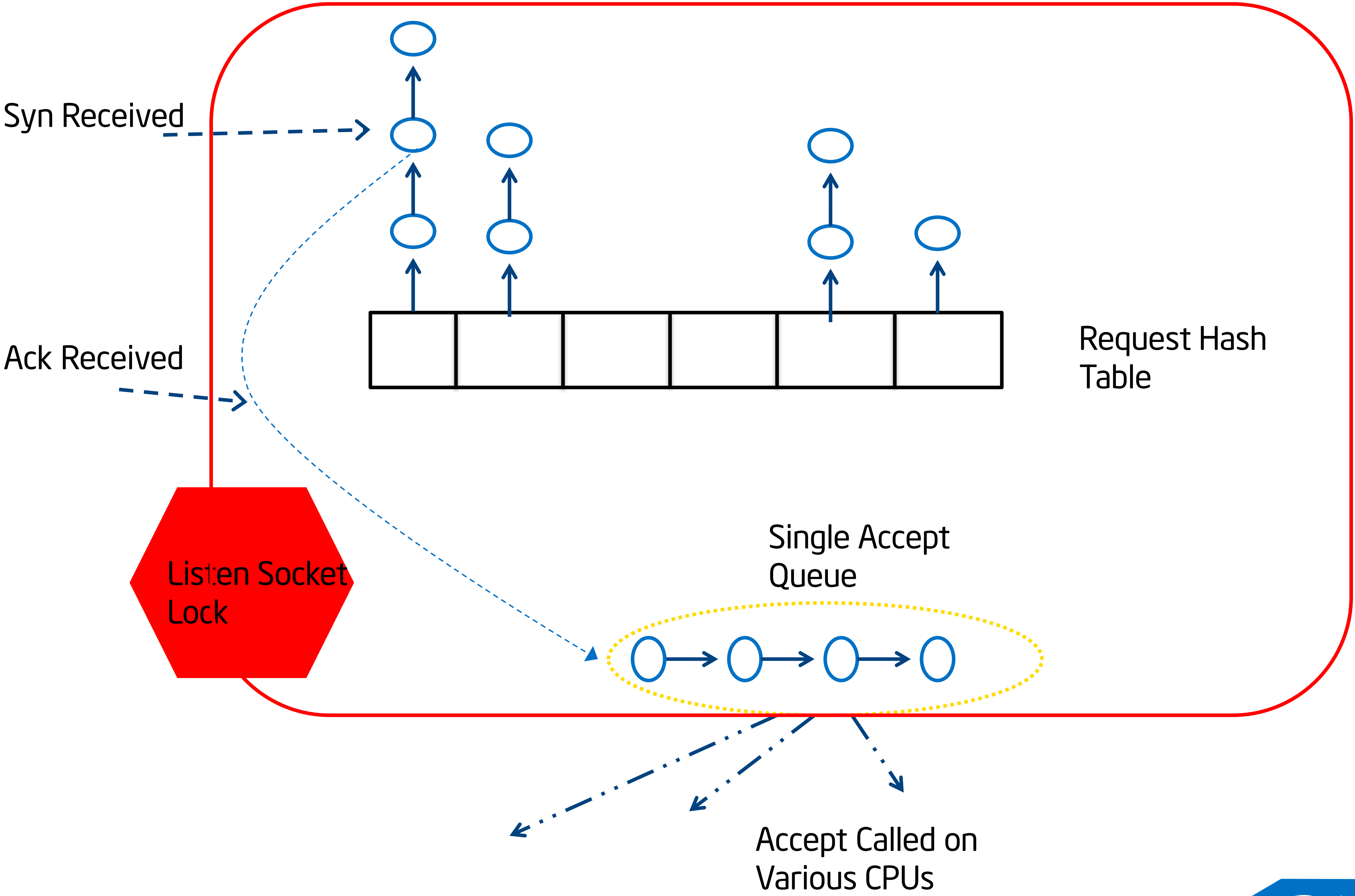
Massively Parallel Platform

Platform has large number of smaller cores running highly parallelized userspace program

Kernel operations that are single threaded could become bottleneck and block (e.g. I/O operations)



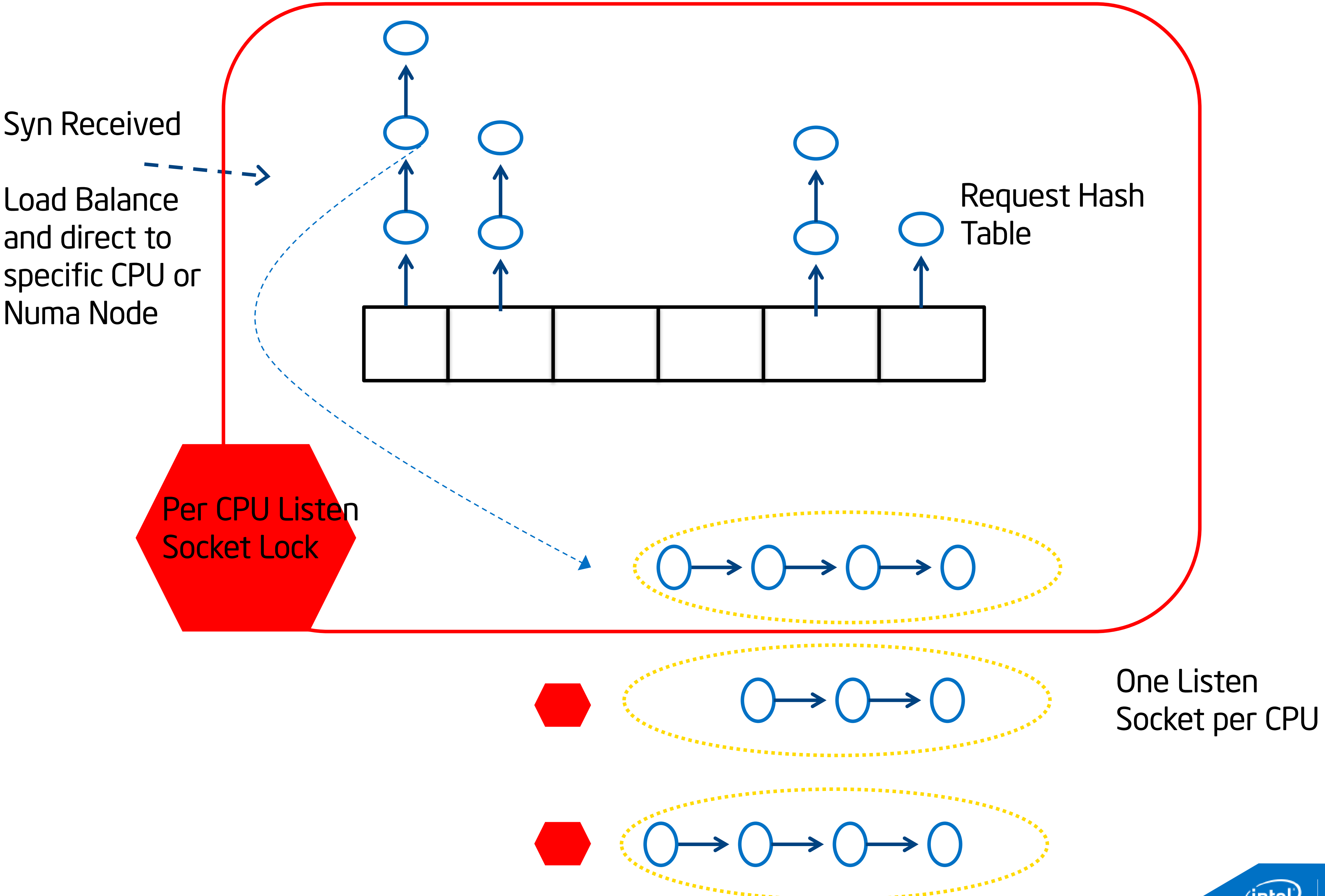
Single-Accept Connection Socket



Multi-Accept Connection Socket

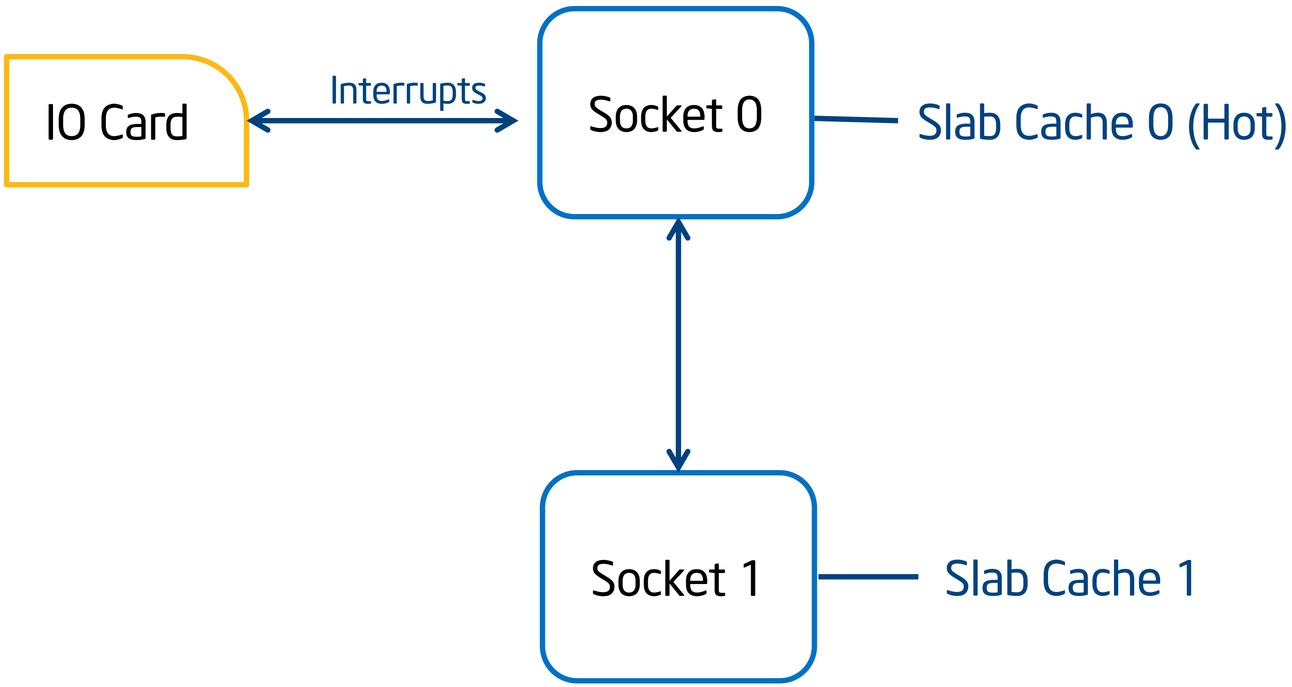
- Single lock on listen socket protecting the socket's single backlog queue and hash table for accept.
- Does not scale well with large number of connect requests handled by different CPUs.
- Socket lock contention and bouncing of socket structure data between CPUs.
- Possible Mitigations:
  - Lockless ring buffer, array queue?
  - Per CPU listen socket/accept queue cloning
  - Per CPU group based listen socket/accept queue cloning (more locking and code changes)

Multi-Accept Socket



Conflict between NUMA affinity and SLAB locality

- For best NUMA affinity, bind the IO interrupts to specific node in the system
- This creates contention at the node for allocation of objects on SLAB



Reclamation of dentries, inodes sb_lock bottleneck

- Heavy page cache pressure from reading large files lead to reclamation of memory with shrinkers
- Superblock shrinker is responsible for counting and reclaiming dentries, inodes and file system specific cache
- Single sb_lock in whole OS taken for counting available cached objects and reclaiming them, scaling problem when available objects near 0.
- Possible mitigations:
  - Don't hold sb_lock when counting
  - Break up sb_lock?