Enhanced Live Migration For Intensive Memory Loads

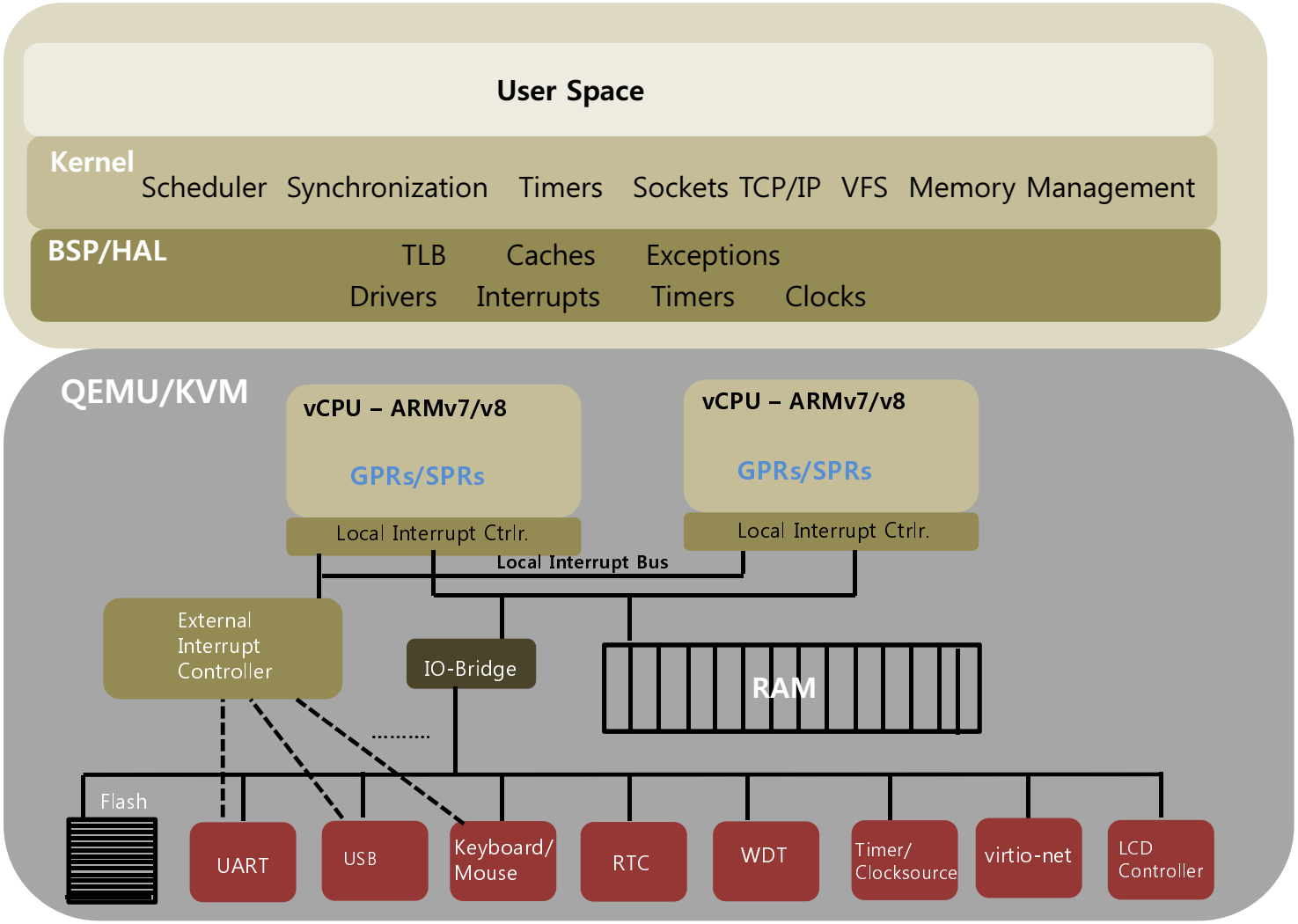
Mario Smarduch
Senior Virtualization Architect
Open Source Group
Samsung Research America (Silicon Valley)
m.smarduch@samsung.com

Agenda

- Cover Enhanced Live migration on ARMv8,7
- General Walkthrough live migration
- Enhancement to handle higher dirty page rate
- Validating Destination

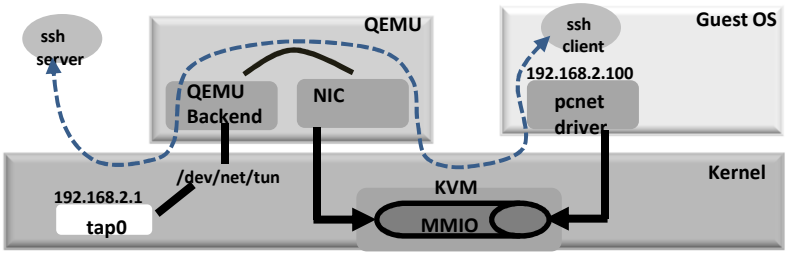
Machine Model & Migration

- Machine Model & State to Migrate
  - ❑ Extremely complex software



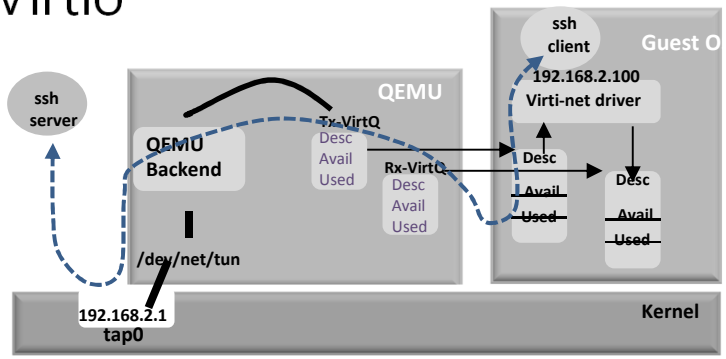
Type of Devices to Migrate

- Type of devices to migrate
  - MMIO



- NIC – conifuguration
- F.E.
  - ➢ – cmd regs, fifo_used, len
  - ➢ - irq level

  - Virtio



- Vring num, last idx available
- Promisc, guest offloads
- Status, macs, GPA of ring buffers

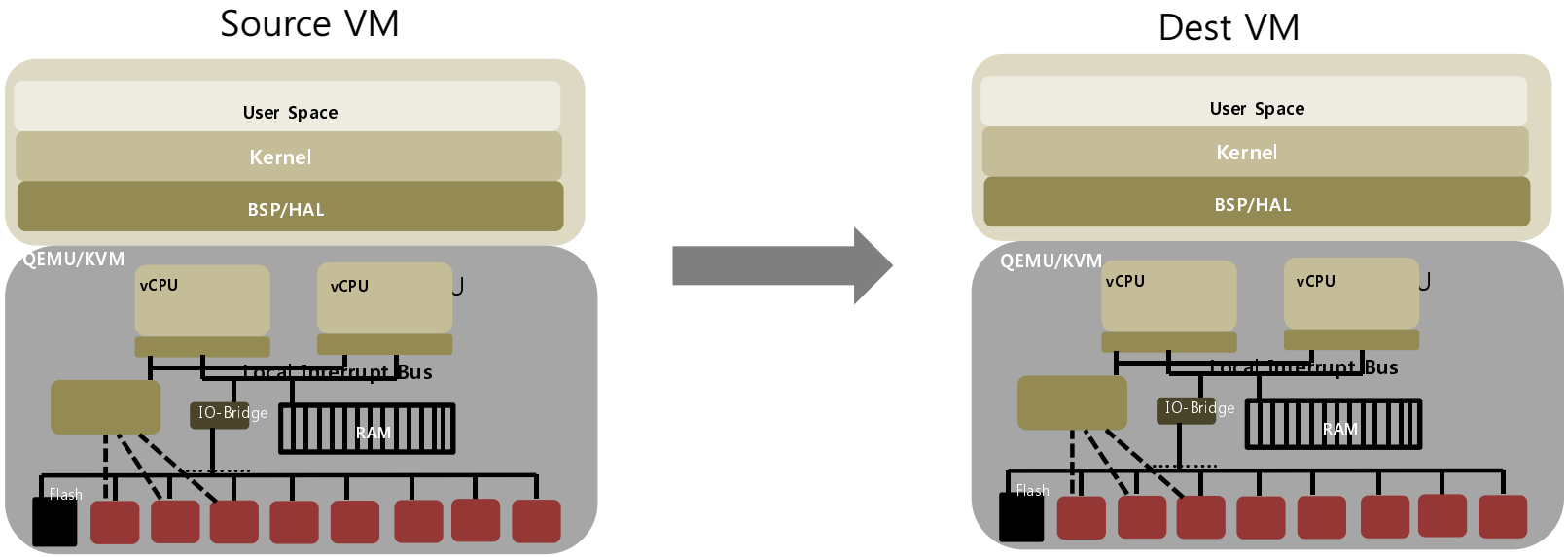
Type of Devices to Migrate

- CPU State
  - GPRs – x0-x30, vfp/simd
  - SPRs – SCTLR, Memory Attr., CPU Affinity, MMU PGD pointer, …
- Interrupt Controller
  - Num of IRQs – IO Interrupt contoller
  - Levels, CPU targets
  - Local Interrupt Ctrl. Reg, Prio. Mask Reg,
- Timers
  - Virtual counter offset
  - Current counter
  - Pending timer events

Who writes to memory?

- Guest – via shadow page tables
- QEMU
  - Virtio devices
- Host KVM – some features
  - Async PF – injects page faults – writes to guest
    - Guest mis – overlap
  - PV-EOI –
    - Writes to Guest – don't do EOI
- Mark all accesses
  - From all source

Migration State



- Dest QEMU
  - ❑ Boots Guest to identical default state – see '-incoming' & QEMU_OPTION_incoming
  - ❑ In Virgin state
  - ❑ Must migrate – source state changes to dest (will see how next)

- All State needs migration – abstracted differently
  - ❑ MMIO device defines VMStateDescrption
  - ❑ Live state GIC, Virtio – stream state to destination
  - ❑ Memory – complex
    - Several stages – setup, iteration, pending, complete

Migration State Declaration

- How do we abstract device migration state?
  - ❑ Simple Timer Example - static

```
static const VMStateDescription vmstate_sp804 {
    .name = …; version_id = ..;
    .fields = ….
    . VMSTATE_INT32_ARRAY(_f=level, _s=SP804State, _n=2) → put_int32()
```

  - ❑ Virtio-net example – dynamic tx & rx dev migration state

```
virtio_net_save(QEMUFile *f, …)
qemu_put_8s(f, &vdev->status);
qemu_put_8s(f, &vdev->status);
 ….
```

  - ❑ RAM Example – special case
    - Handlers

```
.save_live_setup = ….
.save_live_iterate = ….
.save_live_complete
```

RAM Structures

- RAMBlock
  - ❑ All mmap() regions

	Offset	Size
r/w ram	0	0x20000000
r/w ram	0x20000000	0x10000
	0x20010000	0x4000000
	0x24010000	0x4000000
r/w ram	0x28010000	0x2000000
r/w ram	0x2a010000	0x800000

- MemoryRegion
  - ❑ All regions – mmio, ram, flahs

GPA	ram_addr	size
0x80000000	0x0	0x20000000
0x8000000	0x20010000	0x4000000
0xC000000	0x24010000	0x4000000
0x14000000	0x28010000	0x2000000
0x18000000	0x2a010000	0x800000
0x2e000000	0x20000000	0x10000

- Create migration_bitmap – using RAMBlock last offset
- MemoryRegion – ram_addr to index
  - Also used for GVA → GPA and GPA → GVA
  - For migration – iterate RAMBlock, use MR – ram_addr to index dirty bit map

Live Migration High Level

- Devices to Migrate
  - ❑ Flow - (1) WP (2) fault (3) mark dirty log (4) scan (5) WP again
  - ❑ ram, cpu, int ctlr., audio, sd-card, timers, …..
  - ❑ Start with memory first – after completed do devices
  - ❑ High Level Migration –
    - ○ Setup Stage
      - Connect to peer, exchange versions
      - Determine size pages to migrate, dirty bitmap
      - Enable KVM dirty page logging
    - ○ Iteration Stage
      - Establish bandwidth, downtime – user configurable
      - Iterate – walk dirty bitmap – copy to peer
      - Cover RAMBlock –
        - ❖ if remaining RAM < bandwidth * downtime – move to complete
    - ○ Completion Stage
      - Copy remaining ram
      - Go through remaining vmstate_handlers – sync state
    - ○ Peer ready to run
  - ❑ Migration challenge
    - ○ Dirty page rate – primarily Guest memory access

Live Migration More Detail

Setup:
- Connect to Peer, Validate compativility
- Walk RAMBlock List find last page
  - Create 'migration_bitmap' – 696,384 bits for 1k page
  - Assume all memory dirty 'migration_dirty_pages' – 696,384
  - Enable KVM Dirty Page Logging – ARM 4k/64k

migration_bitmap_sync()
- for each writable memory region – get dirty bitmap (kvm_state)
- update to 'migration_bitmap' – convert page sizes
- **increment 'migration_dirty_page' – but not in init**

Iterate:
- xfer rate < bandwidth
- **migrate_bitmap_sync()**
- if remaining dirty memory < bandwidth * downtime
  - move to completion
- Iterate over RAMBlock list
- **migration_bitmap_find_and reset_dirty(block->mr, offset)**
  - **migration_dirty_bytes--;**
- send page to peer
- throttle

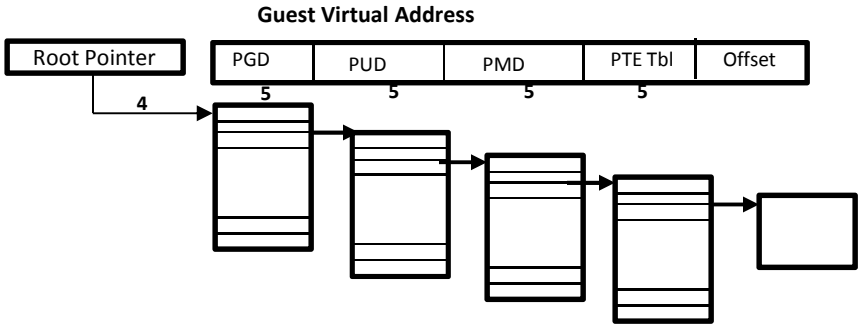
Completion:
- ram_save_complete() – copy remaing ram
- Iterate savevm_handlers – use 'ops' – stream, 'vmsd' – static
  - transfer remaining state

Live Migration Configurables

- xfer_rate < bandwidth
  - ❑ Bandwidth – default 3.2 Mbytes/s can – update QEMU
  - ❑ Governs rate of dirty memory scanning
- Downtime – bandwidth * downtime
  - ❑ Governs – convergence
  - ❑ greater downtime – greater service unavailability
  - ❑ Can update in QEMU (for better or worse)
- Expected Downtime
  - ❑ Based on dirty memory rate / bandwidth
  - ❑ As dirty memory rate increases so does downtime
- To speed up migration
  - Increase network bandwidth
  - Increste downtime
  - *Optimize Memory migration - dirty page tracking, compress*

Huge Pages and performance

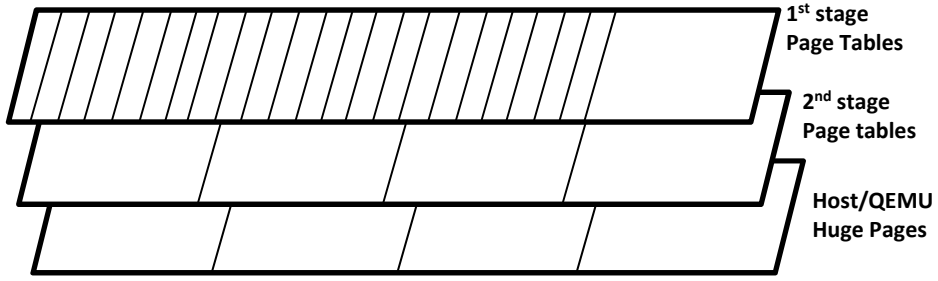
- Huge Pages
  - Guest Page faults very expensive – 6.5x penalty



  - Limit 2nd stage faults
  - Shorten page table walks
  - A must in virtualized env.
  - Depending on IPTW cache – 7% - 32% degradation
    - Also depending on workload

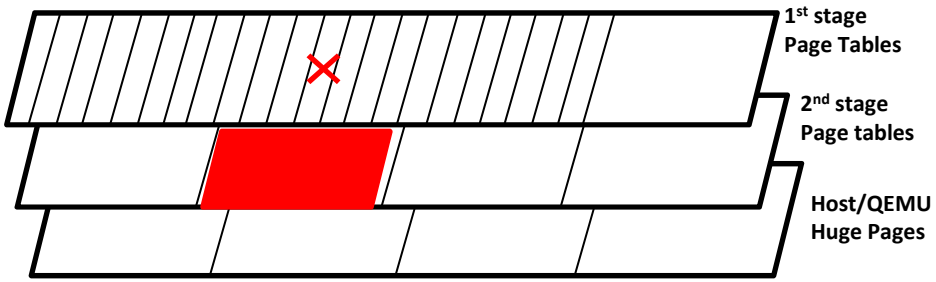
Relationship Between Page Tables

- Guest, Nested, Host page tables involved
  - ❑ THP, hugetlbfs, …
  - ❑ KVM maps in huge 2nd stage entries
  - ❑ ARMv8 4kb pg – 2MB; 64kb page – 512MB
  - ❑ One huge page TLB – covers 512 TLB entries



Migration & Huge Pages

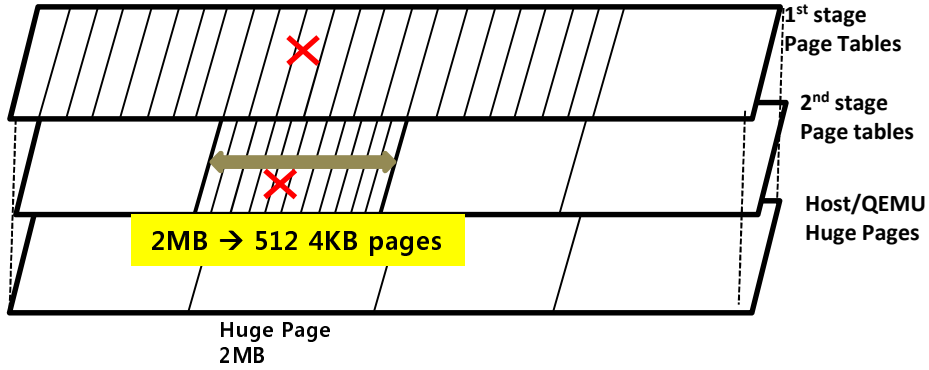
- Huge page granularity to large



- Write to 4k/64k page – assume huge page dirty
  - ❑ We WP – only know about 1st write
  - ❑ Near idle guest – dirtying pages – not migrateble
  - ❑ But we want the performance of huge pages

Run-time dissolve huge pages

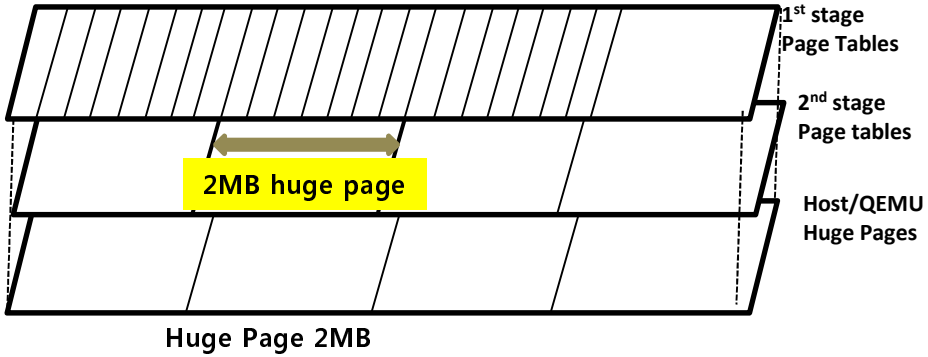
- Dissolve 2nd stage huge pages during migration



- 2nd stage handler – lazy dissolve
  - ❑ For most part 'arch/arm/kvm/mmu.c – 4.x kernel
    - arm64/arm – code shared
  - ❑ Discover fault in huge page backed by Host
  - ❑ Shoot down mapping
  - ❑ Future page faults force small pages
  - ❑ Track 4k page granules

Run-time dissolve huge pages

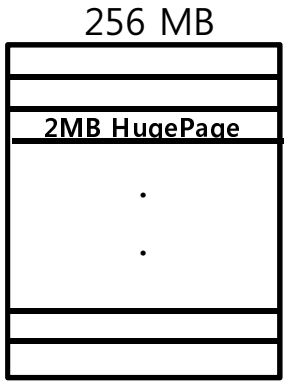
- Once migrated – go back to huge pages
  - ❑ Get the performance back



- Huge Page dissolve – degrades performance …
  - ❑ But we want to migrate and quick
  - ❑ We don't dissolve Read-Only pages

Performance Numbers

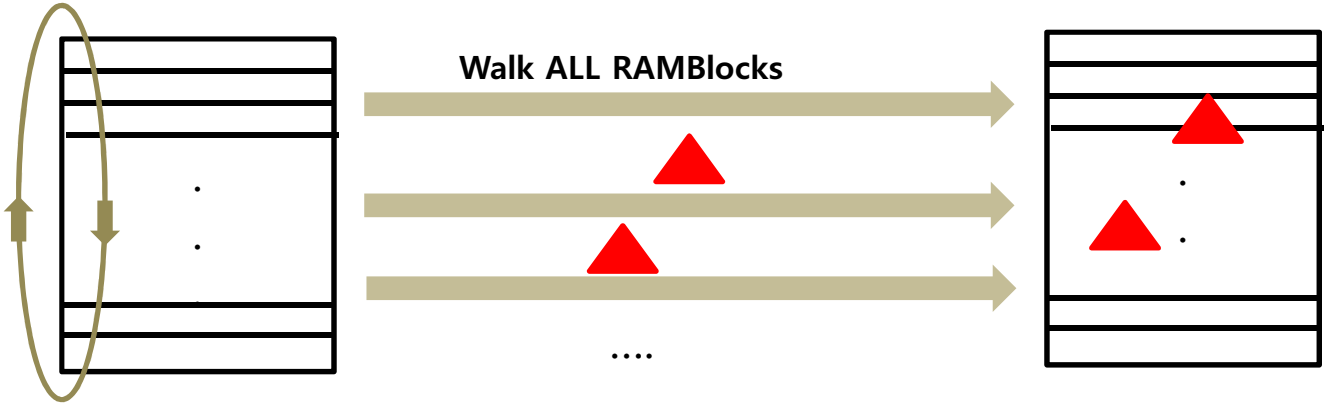
- 256MB memory mark huge page dirty



- With huge page won't auto converge – 1GigE
  - 128 * 2MB = 2Gbps
  - xfer_rate < downtime * bandwidth
  - Can dirty one byte or whole huge page
- With huge page dissolve
  - ~20K - 4k pages/second
  - System loaded lightly – dirty memory periodically
  - See – https://github.com/mjsmar/arm-dirtylog-tests
  - ARMv8 – QEMU support added, ARMv7 for quite a while

Accuracy Testing

- Should trust image on destination
  - ❑ Several sources write to guest memory
  - ❑ Validate destination identical to source



- Destination = initial iteration + delats
- Delta – Guest, QEMU, Host – several sources
- After convergence
  - ❑ Checksum source & destination – must match
  - ❑ Generic approach – associate checksum with each page
    - For repeated pages update checksum
  - ❑ Discover latent memory errors

Other

- Libvirt/Openstack
  - ❑ Enabled by default – nothing to be done

- NFV in mind – lots of memory data bases
- … But any memory intensive workload
- Configurations supported
  - ❑ ARMv8 - 4kb, 64kb Guest kernel → 4kb, 64kb host
    - ○ Huge page 2MB, 512MB
  - ❑ ARMv7 – 4kb Guest → 4kb host
    - ○ Huge page 2MB

Enhanced Live Migration For Intensive Memory Loads

Q & A

Thank you.

Mario Smarduch
Senior Virtualization Architect
Open Source Group
Samsung Research America (Silicon Valley)
m.smarduch@samsung.com