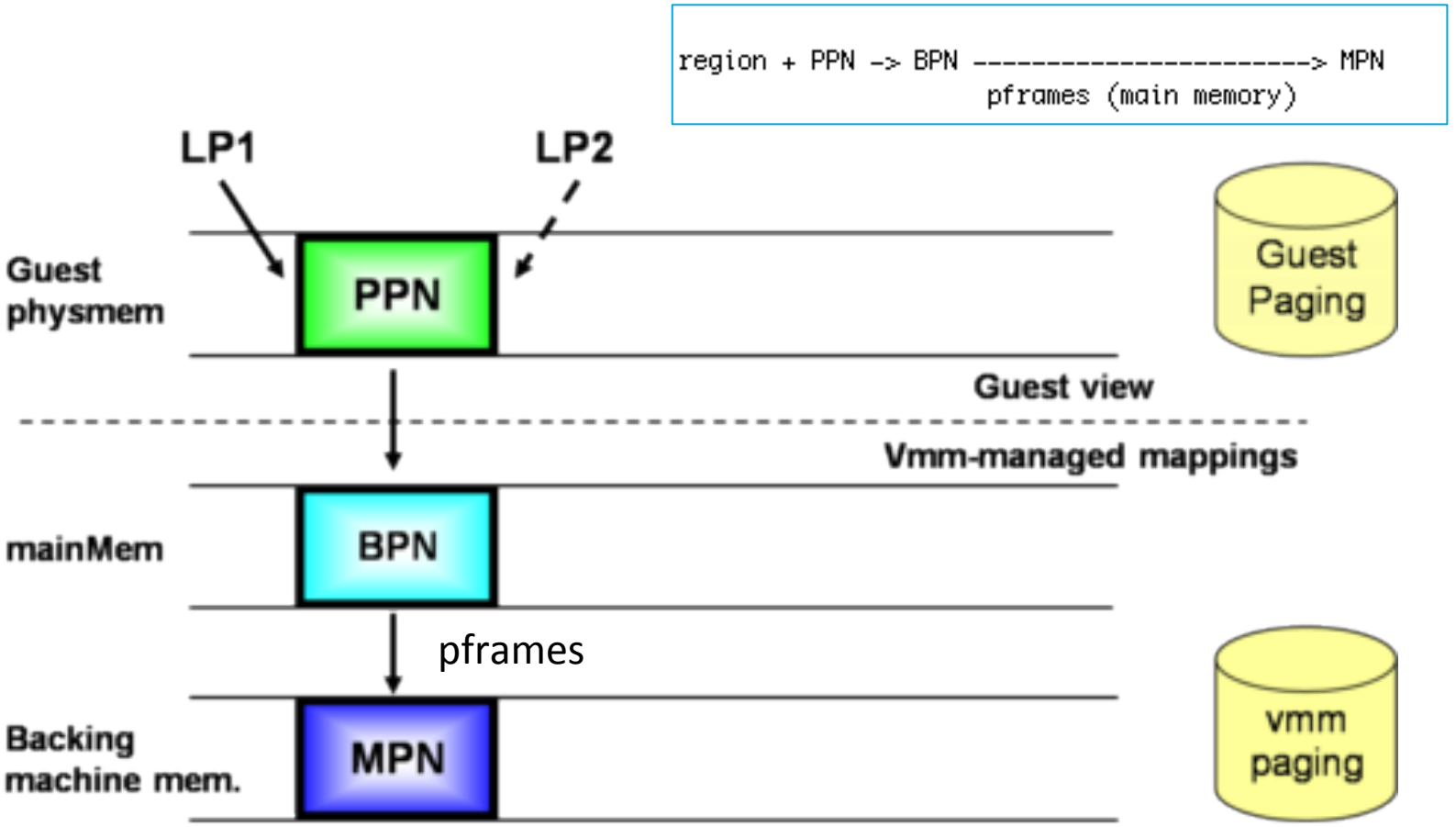
FSR memory handoff

Ricardo Koller


region + PPN -> BPN ----------> MPN
pframes (main memory)
LP1
LP2
Guest physmem
PPN
Guest Paging
Guest view
Vmm-managed mappings
mainMem
BPN
pframes
Backing machine mem.
MPN
vmm paging

BPN to MPN

1. monitor asks kernel for BPN → MPN
2. get pageNum out of BPN
3. get PFrame directory for VM
4. get PFrame for pageNum
5. get MPN out of PFrame

```
BusMem_TranslateBPN(bpn) → returns MPN
    BusMem_GetMPN(bpn)
        Platform_BPN2MPN // hosted or vmkernel
            if (BPN_IsMainMem(bpn))
                VMKCall_GetPFrameMPN
                    VmMemPf_GetPFrameMPN(bpn, &mpn)
                        pgNum = Alloc_BPNToMainMemPage(world, bpn);
                            pgNum ← bpn & BPN_MAINMEM_PAGE_MASK
                        PFrame_Lock(world, pgNum); // any access should be locked, per VM
                        PFrame_Get(world, pgNum, &pframe);
                            PFrame_Lookup(world, pgNum, &dirMPN, &pageIndex)
                                PFrame_Info *pframes = &Alloc_AllocInfo(world)->pframes
                                    *pageIndex = PAGE_2_PAGEINDEX(pgNum);
                                    *dirIndex  = PAGE_2_DIRINDEX(pgNum);
                            *pframe = (PFrame*)Kseg_MapValidMPNFast(dirMPN) + pageIndex;
                            // need to map the MPN to be able to access that memory
                                pframe← (void*)(dirMPN + DIRECT_MAP_VADDR)
                        *mpn = PFrame_GetMPN(pframe);
                            mpn = MPN64_FROM_VAL(PFrame_GetIndex(f));
                                // for COW, REGULAR, SWAP_OUT
                                mpn← f->unmappedInVMX.indexLowBits
                        PFrame_Unlock(world, pgNum);
```

BusMem_TranslateBPN(bpn) → returns MPN

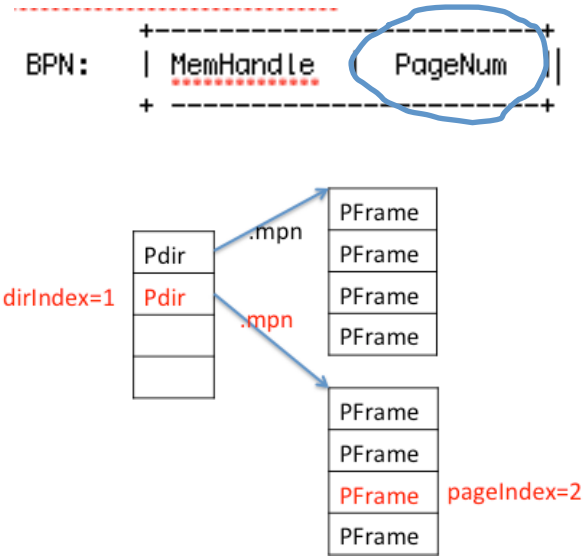
BusMem_GetMPN(bpn)

Platform_BPN2MPN // hosted or vmkernel

if (BPN_IsMainMem(bpn))

VMKCall_GetPFrameMPN

VmMemPf_GetPFrameMPN(bpn, &mpn)



pgNum = Alloc_BPNToMainMemPage(world, bpn);

pgNum ← bpn & BPN_MAINMEM_PAGE_MASK

PFrame_Lock(world, pgNum); // any access should be locked, per VM

PFrame_Get(world, pgNum, &pframe);

PFrame_Lookup(world, pgNum, &dirMPN, &pageIndex)

PFrame_Info *pframes = &Alloc_AllocInfo(world)->pframes

*pageIndex = PAGE_2_PAGEINDEX(pgNum);

*dirIndex = PAGE_2_DIRINDEX(pgNum);

pframe = (PFrame)Kseg_MapValidMPNFast(dirMPN) + pageIndex;

// need to map the MPN to be able to access that memory

pframe← (void*)(dirMPN + DIRECT_MAP_VADDR)

*mpn = PFrame_GetMPN(pframe);

mpn = MPN64_FROM_VAL(PFrame_GetIndex(f));

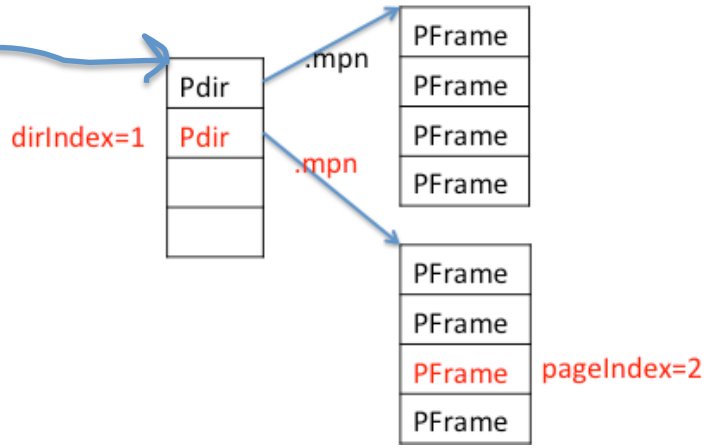
// for COW, REGULAR, SWAP_OUT

mpn← f->unmappedInVMX.indexLowBits

PFrame_Unlock(world, pgNum);

per world

```
// PFrame direcotry and related info
typedef struct PFrame_Info {
   // PFrame directory lock
   SP_SpinLock           pdirLock;

   // PFrame domains
   PFrameDomain         *domains;
   uint32                domainMask;

   // Guest mainMem memory pages
   uint32                numMainMemPages;

   // During FSR, we will set default type to remote
   Bool                  defaultPageTypeIsRemote;
   PageNum               defaultPageTypeIsRemoteMaxPgNum;

   /*
    * Two-level index from pgNum -> MPN pframe content.
    * 'pages' is array of MPNs containing the second level of index.
    */
   PDir_Info            *pdirInfo;

   // Contains extended PFrames used for storing per-PageNum debug data
   MPN64                *pages2;

   // Page directory MPN usage
   uint32                numPDirEntries;
   uint32                numXMapPages;
   uint32                numL3PTPages;
   uint32                numPDirEntries2;
   uint32                numXMapPages2;
} PFrame_Info;
```



BusMem_TranslateBPN(bpn) → returns MPN

BusMem_GetMPN(bpn)

Platform_BPN2MPN // hosted or vmkernel

if (BPN_IsMainMem(bpn))

VMKCall_GetPFrameMPN

VmMemPf_GetPFrameMPN(bpn, &mpn)

pgNum = Alloc_BPNToMainMemPage(world, bpn);

pgNum ← bpn & BPN_MAINMEM_PAGE_MASK

PFrame_Lock(world, pgNum); // any access should be locked, per VM

PFrame_Get(world, pgNum, &pframe);

PFrame_Lookup(world, pgNum, &dirMPN, &pageIndex)

PFrame_Info *pframes = &Alloc_AllocInfo(world)->pframes

*pageIndex = PAGE_2_PAGEINDEX(pgNum);

*dirIndex = PAGE_2_DIRINDEX(pgNum);

pframe = (PFrame)Kseg_MapValidMPNFast(dirMPN) + pageIndex;

// need to map the MPN to be able to access that memory

pframe← (void*)(dirMPN + DIRECT_MAP_VADDR)

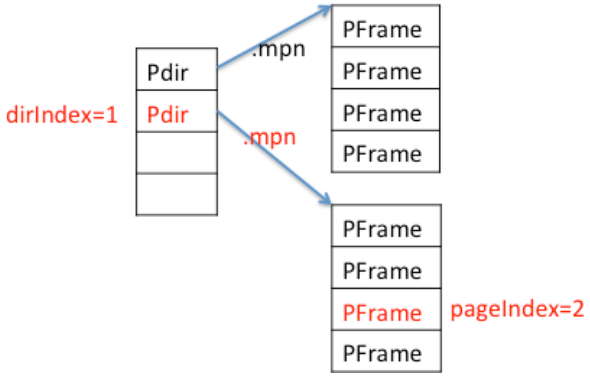
*mpn = PFrame_GetMPN(pframe);

mpn = MPN64_FROM_VAL(PFrame_GetIndex(f));

// for COW, REGULAR, SWAP_OUT

mpn← f->indexLowBits + f->indexHighBits << 32

PFrame_Unlock(world, pgNum);



```
typedef union PFrame {
   uint64 pte;
   PFrameMappedInVMX     mappedInVMX;
   PFrameUnmappedInVMX   unmappedInVMX;
} PFrame;
```

MPN = indexHighBits << 32 | indexLowBits

```
typedef struct {
   // 8-bits
   uint64 isMappedInVMX    : 1;    // bit  00    --  always set to 0
   uint64 isCptBusy        : 1;
   uint64 isCptDone        : 1;
   uint16 isExposedToVMM   : 1;
   uint64 type             : 4;

   // 8-bits
   union {
      struct {
         uint8 indexHighBits  : 6;
         uint8 zipBlock       : 2;
      } __attribute__((__packed__)) zipped;

      struct {
         uint8 indexHighBits  : 6;
         uint8 vmkSwapped     : 1;
         uint8 unused         : 1;
      } __attribute__((__packed__)) swapped;

      struct {
         uint8 indexHighBits  : 6;
         uint8 vmkSwapped     : 1;
         uint8 unused         : 1;
      } __attribute__((__packed__)) swapin;

      struct {
         uint8 indexHighBits    : 6;
         uint8 isBallooned      : 1;
         uint8 isAppBallooned   : 1;
      } __attribute__((__packed__)) pshared;

      struct {
         uint8 indexHighBits    : 6;
         uint8 isCowHint        : 1;
         uint8 isBackedByLPage  : 1;
      } __attribute__((__packed__)) regular;

      struct {
         uint8 indexHighBits  : 6;
         uint8 flags          : 2;
      } __attribute__((__packed__)) common;
   } extra;

   // 16-bits
   uint16 pinCount         : 14;
   uint16 isInvalGoingOn   :  1;
   uint16 isNotAtMigDest   :  1;

   // Lower 32-bits of a MPN/index
   uint32 indexLowBits;
} __attribute__((__packed__)) PFrameUnmappedInVMX;
```

per world

New world

```
// PFrame direcotry and related info
typedef struct PFrame_Info {
   // PFrame directory lock
   SP_SpinLock           pdirLock;

   // PFrame domains
   PFrameDomain         *domains;
   uint32                domainMask;

   // Guest mainMem memory pages
   uint32                numMainMemPages;

   // During FSR, we will set default type to remote
   Bool                  defaultPageTypeIsRemote;
   PageNum               defaultPageTypeIsRemoteMaxPgNum;

   /*
    * Two-level index from pgNum -> MPN pframe content.
    * 'pages' is array of MPNs containing the second level of index.
    */
   PDir_Info            *pdirInfo;

   // Contains extended PFrames used for storing per-PageNum debug data
   MPN64                *pages2;

   // Page directory MPN usage
   uint32                numPDirEntries;
   uint32                numXMapPages;
   uint32                numL3PTPages;
   uint32                numPDirEntries2;
   uint32                numXMapPages2;
} PFrame_Info;
```

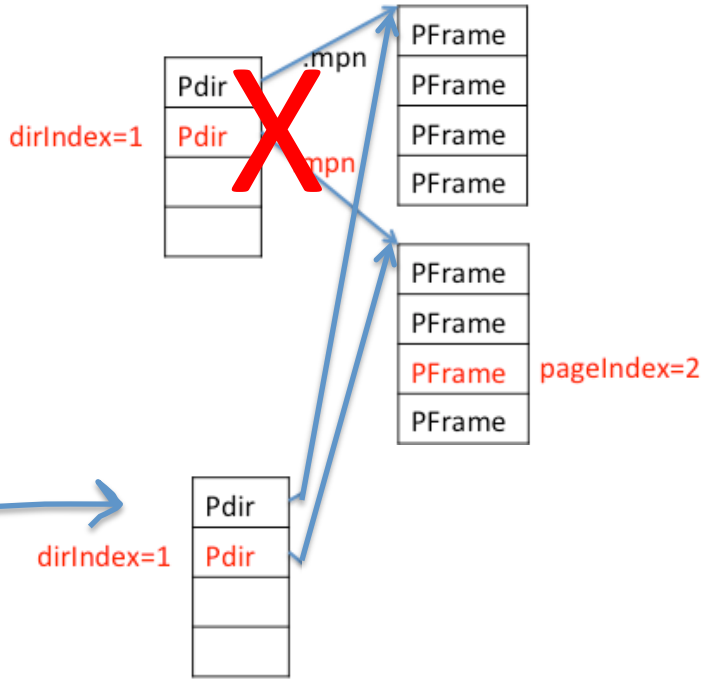

```
// PFrame direcotry and related info
typedef struct PFrame_Info {
   // PFrame directory lock
   SP_SpinLock           pdirLock;

   // PFrame domains
   PFrameDomain         *domains;
   uint32                domainMask;

   // Guest mainMem memory pages
   uint32                numMainMemPages;

   // During FSR, we will set default type to remote
   Bool                  defaultPageTypeIsRemote;
   PageNum               defaultPageTypeIsRemoteMaxPgNum;

   /*
    * Two-level index from pgNum -> MPN pframe content.
    * 'pages' is array of MPNs containing the second level of index.
    */
   PDir_Info            *pdirInfo;

   // Contains extended PFrames used for storing per-PageNum debug data
   MPN64                *pages2;

   // Page directory MPN usage
   uint32                numPDirEntries;
   uint32                numXMapPages;
   uint32                numL3PTPages;
   uint32                numPDirEntries2;
   uint32                numXMapPages2;
} PFrame_Info;
```

FSR memory handoff

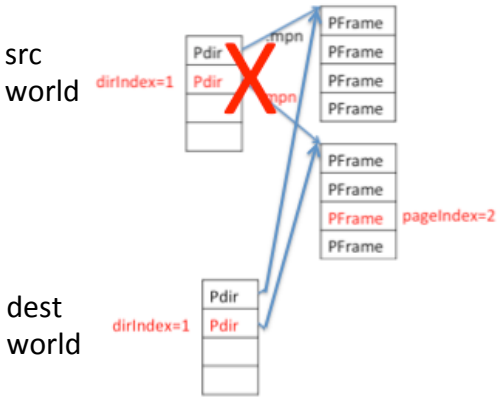
```
FSR_ResumeDone
    FSRPrepareForMemoryTransfer
        FSRVerifyReservationsToTransfer
            if (fsr->memMinToTransferMB > MemSched_GroupGetAlloc->min)   FAIL
            if (fsr->cpuMinToTransferMHz > CpuSched_GroupGetAlloc.min)   FAIL
        VmMemMigrate_PrepareForMemoryTransfer
            Swap_HasMigOrCptSwappedPages
                if Swap_GetMigSwapFile(world) has used slots   FAIL
                if Swap_GetCptFile(world) has used slots   FAIL
            Swap_Disable(Swap_GetInfo(srcWorld))
                swapInfo->swapperEnabled = FALSE
    FSRTransferSwapFile
    World_SetPanicNoCore // point of no return
    FSRTransferMemoryAndReservations // if fail, calls FSRCancelPrepareForMemoryTransfer
        FSRTransferReservations(fsr, srcWorld, dstWorld)
        VmMemMigrate_TransferMemory(srcWorld, dstWorld)
```

new world

```
// PFrame direcotry and related info
typedef struct PFrame_Info {
   // PFrame directory lock
   SP_SpinLock           pdirLock;

   // PFrame domains
   PFrameDomain         *domains;
   uint32                domainMask;

   // Guest mainMem memory pages
   uint32                numMainMemPages;

   // During FSR, we will set default type to remote
   Bool                  defaultPageTypeIsRemote;
   PageNum               defaultPageTypeIsRemoteMaxPgNum;

   /*
    * Two-level index from pgNum -> MPN pframe content.
    * 'pages' is array of MPNs containing the second level of index.
    */
   PDir_Info            *pdirInfo;

   // Contains extended PFrames used for storing per-PageNum debug data
   MPN64                *pages2;

   // Page directory MPN usage
   uint32                numPDirEntries;
   uint32                numXMapPages;
   uint32                numL3PTPages;
   uint32                numPDirEntries2;
   uint32                numXMapPages2;
} PFrame_Info;
```



```
VmMemMigrate_TransferMemory
    VmMemMigrateTransferMemory
        for (pgNum = 0; pgNum < Alloc_NumMainMemPages(srcWorld); pgNum += ALLOC_PDIR_PAGES)
            status = PFrame_Lookup(srcWorld, pgNum, &srcMPN, &pageIndex)
            status = PFrame_Lookup(dstWorld, pgNum, &dstMPN, &pageIndex)
            VmMemMigrateTransferPageDir(srcWorld, dstWorld, pgNum, &srcMPN, &dstMPN)
                VmMemMigrateRemovePageDirFromSource(srcWorld, pgNum, srcMPN)
                    for (index = 0; index < ALLOC_PDIR_PAGES; index++)
                        BackMap_Unset(mpn, srcWorld, pgNum + index)
                        VmMem_UpdateFreeStats(srcWorld, pgNum + index, mpn)
                        // STATS ← simulate a page free
                PFrame_DirSet(srcWorld, pgNum, INVALID_MPN64, INVALID_MPN64)
                    // PDIR ← map source PDir to nothing
                PFrame_DirSet(dstWorld, pgNum, srcMPN)
                    // PDIR → redirect the new PDirs to the source PDir MPN
                    Alloc_Info *info = Alloc_AllocInfo(world); // world is dest
                    PFrame_Info *pframes = &info->pframes; // world is dest
                    uint32 dirIndex  = PAGE_2_DIRINDEX(pgNum);
                    PTE_SET((uint64 *)&pframes->pdirInfo[dirIndex], PFrameMakePTE(srcMPN))
                VmMemMigrateAddPageDirToDest(dstWorld, pgNum, srcMPN, dstMPN, freeSrcPFrame)
                    *dstMPN = srcMPN;
                VmMemMigrateInitPageDirsOnDest(srcWorld, dstWorld, pgNum, *dstMPN)
                    dir = Kseg_MapMPN(dstMPN)
                    for (pageIndex = 0; pageIndex < ALLOC_PDIR_PAGES; pageIndex++)
                        VmMemMigrateAddPageToDest(pframe)
                            if pframe.type == MEMSCHED_VMPAGE_REGULAR
                                PFrame_SetRegular(dstWorld, pframe, pgNum, mpn)
                                VmMem_UpdateAllocStats(dstWorld, pgNum, mpn, 1)
                                // STATS → simulate a page alloc
```



where are shadow page tables copied/ invalidated?