VProbes Update

What’s New Since ESX 5.1

Restricted

Outline

Language Features

- Exceptions
- Loops, iterators
- Binary logging
- Aggregate formatting
- Shared integers
- Preprocessing
- Probe globbing

Fault Injection

- Modify register state
- Modify memory
- Skip functions
- Inject virtual MCEs

Usability Enhancements

- External types
- Local variables and arguments
- Line number probing

Language Features

Exception Handling

- **Intercept runtime errors**
  - Typically builtin failures
  - Memory read errors, full bag on insert, empty bag on remove, etc.
- **Programatically handle errors**
  - Trap and ignore errors
  - Print error message
  - Assign default values
- **Syntax similar to C++**
- **Default behavior:**
  - Terminate the current probe
  - Print an error message

```
try {
   value = *address;
   printf(“%d”, value);
} catch (VmkMemAccessError) {
   printf(“Memory read error\n”);;
}
```

Loops

- Previously:
  - Iterative computations expressed using recursion
- Standard loop statements now available:
  - For, while, do-while
  - Break and continue
- Example:

```
memmodel guest64;
void guestbt() {
   printf(“%#x\n”, RIP);
   for (rbp = RBP; rbp; rbp = *(uint64*)rbp) {
       printf(“%#x\n”, *(uint64*)(rbp + 8));
   }
}
```

Bag Iterators

- Iterate over all key-value pairs in bags
- Syntax inspired by Python
- Example:

```
void printbag(bag b) {
    int key, val;
    foreach (key, val) in b {
        printf(“%d: %d\n”, key, val);
    }
}
```

Avoid Infinite Loops

- Bound time per probe fire
- Older releases:
  - Upper bound on function calls per probe (400)
- Newer releases use a cost model:
  - Assign costs to each builtin, loop back-edges, function calls
  - Impose a maximum budget per probe (aimed at about 50us per probe)

```
try {
    for (i = 0; ; i += 1) {}
} catch {
    printf(“Loop ran for %d iterations\n”, i);
}
```

Multiple Returns

- Multiple return points in functions and probes
- Return different values:

```
int squareSum(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n * n + squareSum(n - 1);
    }
}
```

- Early probe returns

```
VMK1Hz {
    if (PCPU == 0) {
        return;
    }
    ...
}
```

Binary Logging

- `printf` outputs formatted text
- `writeint` outputs binary data

```
      writeint (uint64 data, int numBytes)
```

- Provides some form of output compression
- Use repeated calls to write out entire structures
- Possible binary output formats:
  - Fixed-size output records
  - Variable-size records (must also write record length)

- Problem: difficult to distinguish errors from binary data
  - Vprobe output split into output and error streams
  - Stdout/stderr on ESX; vprobe.out/vprobe.err on hosted

Formatting Aggregate Output

- Aggregates
  - Data type for building Histograms, bucketed by integer and string keys.
  - Print using `logaggr(a)`, clear using `clearaggr(a)`
  - Example:

```
aggr ioStats[1][1];
bag  ioSize, ioStart, ioType;
FS3AsyncIOStart(int len, void *o, int read, int guid) {
   ioSize[guid]  <- len / 1024;
   ioStart[guid] <- TSC;
   ioType[guid]  <- read;
}
FSAsyncIOEnd(int guid, int status) {
   ioStats[ioSize[guid], ioType[guid] ? "READ" : "WRITE”]
                <- cycles2ms(TSC - ioStart[guid]) / 1000;
}
VMK1Hz logaggr(ioStats);
```

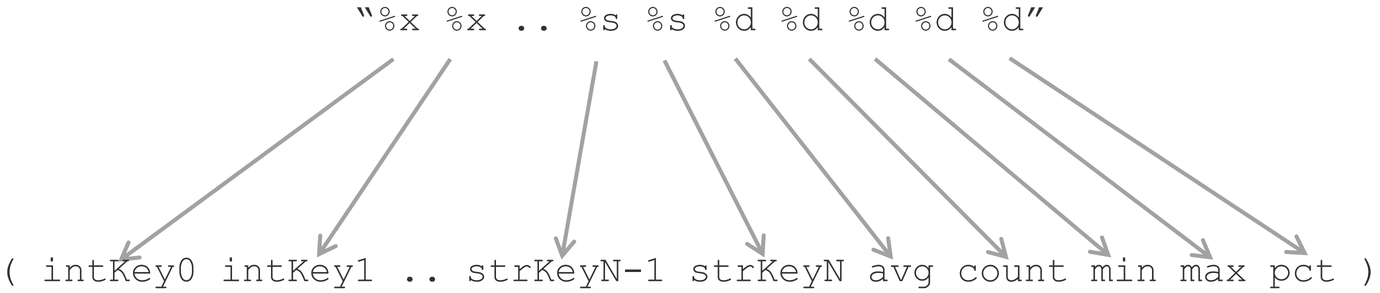
Formatting Aggregate Output - logaggr

- `logaggr(ioStats)`
  - Fixed format, tabular.
  - Each row is a bucket in the aggregate.

```
intKey0            strKey0        avg     count       min       max    pct%
              0x2c READ             7        92         0       210    0.4%
              0x18 READ            23        31         0       273    0.4%
              0x40 WRITE           45        18         0       208    0.5%
               0xc WRITE           38        32         0        89    0.8%
               0x8 WRITE           32        38         0        87    0.8%
               0x4 WRITE            7       165         0       158    0.8%
               0x4 READ             7       170         0       292    0.8%
              0x60 WRITE           30        46         0       210    0.9%
               0x8 READ            23        69         0       175    1.1%
               0xc READ            25        65         0       218    1.1%
              0x80 READ             5      2482         0       727    8.8%
              0xa0 WRITE           50      2127         0       308   72.0%
```

Formatting Aggregate Output – printa

- `printa(aggr[, format-str])` *[Available in 2014]*
  - C `printf` style format string “applied” to each bucket in the Histogram before printing.
  - Arguments to the format string are implied,



```
“%x %x .. %s %s %d %d %d %d %d”

( intKey0 intKey1 .. strKeyN-1 strKeyN avg count min max pct )
```

- Explicit fields selectors: `“%<field-identifier>$<format-specifier>”`
  - Inspired by Posix extensions to `printf` for numbered arguments.
  - Field Identifiers:
    - `%0$, %1$, %2$ …`
    - `%avg$, %count$, %min$, %max$, %pct$`

Formatting Aggregate Output - Examples

- `printa(ioStat, “%d”)`

```
intKey0 strKey0       avg     count       min       max   pct%
     96 WRITE          30        46         0       210   0.9%
      8 READ           23        69         0       175   1.1%
     12 READ           25        65         0       218   1.1%
    128 READ            5      2482         0       727   8.8%
    160 WRITE          50      2127         0       308  72.0%
```

- `printa(ioStat, “%d KB | %8s | %6x %-6d %06d %6d”)`

```
intKey0 KB |  strKey0 |    avg count     min    max   pct%
     96 KB |    WRITE |   0x1e 46     000000    210   0.9%
      8 KB |     READ |   0x17 69     000000    175   1.1%
     12 KB |     READ |   0x19 65     000000    218   1.1%
    128 KB |     READ |    0x5 2482   000000    727   8.8%
    160 KB |    WRITE |   0x32 2127   000000    308  72.0%
```

Formatting Aggregate Output – Examples Contd.

- `printa(ioStat, “%d KB | %8s | %avg$d | %count$d”)`

```
intKey0 KB |  strKey0 |       avg |     count
     96 KB |    WRITE |        30 |        46
      8 KB |     READ |        23 |        69
     12 KB |     READ |        25 |        65
    128 KB |     READ |         5 |      2482
    160 KB |    WRITE |        50 |      2127
```

- `printa(ioStats, "{ len: %=d, type: \"%=s\",  avg: %avg$=d,  count: %count$=d },”)`

```
{ len: 96, type: "WRITE",  avg: 30,  count: 46 },
{ len: 8, type: "READ",  avg: 23,  count: 69 },
{ len: 12, type: "READ",  avg: 25,  count: 65 },
{ len: 128, type: "READ",  avg: 5,  count: 2482 },
{ len: 160, type: "WRITE",  avg: 50,  count: 2127 },
```

Shared Integers

- New storage classes for integers:
  - `pervm` – shared in a VM, across all VCPU and VMX threads
  - `pervmk` – shared in the VMkernel, across all PCPUs
  - `perhost` – shared in the ESX host, across all VMs and the Vmkernel
- New built-ins `fetchadd` and `cmpxchg`
- Reads and writes are fenced for sequentially consistency

```
pervm int hvExitCtr;

HVExit {
    fetchadd(hvExitCtr, 1);
}

VMXUnload {
    printf(“%u\n”, hvExitCtr);
}
```

Shared Integers – Example

```
// excerpt from --
// bora/devkits/tools/ddv/scripts/ddv_common.emt
pervmk int bigDDVLock;

int ddvSpinLockAcquire()
{
  int lockAcquired;
  if (!cmpxchg(bigDDVLock, 0, 1)) {
     lockAcquired = ddvSpinLockRetry(1);
  } else {
     lockAcquired = SPLOCK_SUCCESS;
  }
  return lockAcquired;
}
```

Emmett Pre-Processor (EPP)

- Much like the C pre-processor
- Macros: `#define, #undef`
  - `-D IDENT[=VAL]` vprobe app switch
  - No support for macro arguments
- File inclusion: `#include`
  - `-I PATH` vprobe app switch
- Conditional inclusion: `#if, #ifdef, #elif, #else`
- Misc: `#error, #warning, #line`
- Pre-defined macros : `__VERSION__, __ESX__, __DESKTOP__`
- Part of Emmett compiler, written in OCaml.
  - CPP doesn’t work due to distribution issues (Windows, OSX, Linux, ESX)

EPP – Example

```
// excerpt from
// bora/devkits/tools/ddv/scripts/ddv_common.emt

#if (__VERSION__ < 2013)
int ddvCurModID()
{
   ModInfoStack *modInfoStack;
   modInfoStack = curworld()->modStack;
   return (modInfoStack ? modInfoStack->modID : DUT_MOD_ID_INVALID);
}
#define CUR_MOD_ID ddvCurModID()
#else
#define CUR_MOD_ID LASTMODID
#endif /* (__VERSION__ < 2013) */
```

Probe Globbing

- Define probe points using regex patterns
- Available in the VMK domain
- Example:

```
aggr histo[0][1];
VMK:ENTER:Timer_* { histo[PROBENAME]++; }
VMK:VMKUload      { printa(histo); }
```

- Output:

```
strKey0                                        count  pct%
VMK:ENTER:Timer_WorldPreCleanup                    1   0.0%
VMK:ENTER:Timer_RemoveSync                        33   0.0%
VMK:ENTER:Timer_ModifyOrAddTC                     60   0.0%
VMK:ENTER:Timer_SysUptimeUS                       63   0.0%
VMK:ENTER:Timer_SysUptime                        121   0.0%
VMK:ENTER:Timer_UpdateOneShot                    312   0.1%
VMK:ENTER:Timer_Pending                          362   0.1%
VMK:ENTER:Timer_Stats                            705   0.3%
VMK:ENTER:Timer_Remove                           813   0.4%
VMK:ENTER:Timer_AddTC                            917   0.4%
VMK:ENTER:Timer_SignedTCToMS                     954   0.5%
VMK:ENTER:Timer_AddTCWithLockDomain             1279   0.6%
VMK:ENTER:Timer_EnableStatsTimer                3560   1.8%
VMK:ENTER:Timer_Interrupt                       5884   3.1%
VMK:ENTER:Timer_BHHandler                      26241  13.9%
VMK:ENTER:Timer_GetCycles                     146500  78.0%
```

Fault Injection

Fault Injection Framework

- Destructive builtins that modify system state
- `setphysgpr()`
  - Modify physical register state
  - Available in the VMM and VMK domains
- `setgpr()`
  - Modify guest register state (i.e., virtual registers)
  - Available in the VMM and GUEST domains
- `setvmw()`
  - Modifies memory contents
  - Available in the VMK domain
- `genmce()`
  - Inject MCE’s into the guest

Fault Injection Framework

- `skipfunction()`
  - Modifies control flow
  - Skip execution of the current function
  - Available in VMK, at function entry points
- Example

```
VMK:ENTER:vmklinux_9.vmklnx_kmalloc {
   if (injectFault) {
      /* set return value as NULL */
      setphysgpr(REG_RAX, 0);
      /* skip execution of vmklnx_kmalloc*/
      skipfunction();
}
```

VMK Watchpoints

- Program physical DR's for read/write watchpoints
- Builtins: `setwatchpoint()`, `removewatchpoint()`
- Static probe: WatchpointHandler
  - Fire probe when the watchpoint triggers
- Limitations
  - Use NMIs to propagate DR updates to other PCPUs
  - VMM is not NMI hardened
  - Hence, this is an unsupported feature

VMK Watchpoints Example

```
pervmk int rsp, wpid;
ENTER:Timer_Interrupt {
    if (PCPU == 0) {
        rsp = getphysgpr(REG_RSP);
        wpid = setwatchpoint(rsp, 0x2/*8 bytes*/, 0x1/*WRITE*/);
    }
}
WatchpointHandler(int addr) {
    string bt;
    vmwstack(bt);
    printf(“Write watchpoint at %p, backtrace %s\n”, addr, bt);
}
EXIT:Timer_Interrupt {
    if (PCPU == 0) {
        removewatchpoint(rsp);
    }
}
```

Usability Enhancements

External Types

- Let’s write a script that prints `PRDA->lastIntIdx` every second, for every PCPU
  - You must manually determine `lastIntIdx`’s offset and declare a sparse type

```
typedef struct PRDA {
   @0x54 lastIntIdx;
} PRDA;

VMK1Hz {
   PRDA *p = PRDA_GET_ADDR;
   printf(“PCPU %d: %d\n”, PCPU, p->lastIntIdx);
}
```

External Types

- New external type support allows referencing VMK types without the need for declaration:

```
VMK1Hz {
   $vmkernel.PRDA *p = PRDA_GET_ADDR;
   printf(“%d\n”, p->lastIntIdx);
}
```

- Notation: *$MODULE.TYPE*
  - `$vmkernel.World_Handle`
  - `struct $vsan.BatchResult`
  - `union $vmkernel.PCIECapReg`
  - `enum $tcpip4.VmkNetDomain = $tcpip4.VMK_INET_DOMAIN`

Line Numbers

- Let’s write a script that probes a line of function World_New() and prints the PCPU#

```
/* bora/vmkernel/main/world.c */
3012: VMK_ReturnStatus
3013: World_New(World_InitArgs *args,
3014:           World_ID       *worldID)
      ...
3035: if (args->typeFlags & WORLD_USER) {
3036:    initTable = userTableInit;
      ...
```

- You must manually map world.c:3036 to the corresponding function-relative byte offset

```
VMK:OFFSET:World_New:0xA8 {
   printf(“PCPU %d\n”, PCPU);
}
```

Line Numbers

- New line number probing support eliminates need for manual line number to byte offset translation
- File-relative line numbers:

```
VMK:OFFSET:World_New:L3036 {
   printf(“PCPU: %d\n”, PCPU);
}
```

- Function-relative line numbers:

```
VMK:OFFSET:World_New:F23 {
   printf(“PCPU: %d\n”, PCPU);
}
```

Local Variables and Arguments

- Now we want to inspect `args->p2mCacheSize` at the same probe point (world.c:3036)

```
/* bora/vmkernel/main/world.c */
3012: VMK_ReturnStatus
3013: World_New(World_InitArgs *args,
3014:           World_ID       *worldID)
      ...
3035: if (args->typeFlags & WORLD_USER) {
3036:    initTable = userTableInit;
      ...
```

Local Variables and Arguments

- Must manually determine:
  - the offset of `p2mCacheSize` in `World_InitArgs`
  - where `args` is stored at the probe point (stack? register?)

```
typedef struct World_InitArgs {
   @0x80 uint32 p2mCacheSize;
} World_InitArgs;

VMK:OFFSET:World_New:0xA8 {
   World_InitArgs *args = getphysgpr(REG_RBX);
   printf(“%d\n”, args->p2mCacheSize);
}
```

Local Variables and Arguments

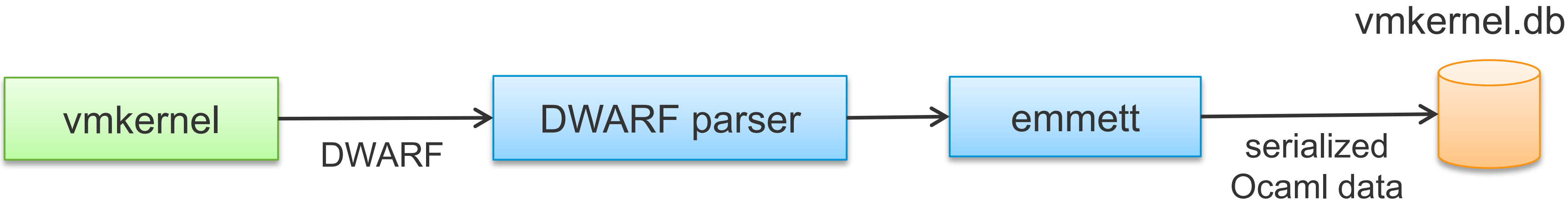
- New local variable inspection support allows target local variables to be directly referenced by name:

```
VMK:OFFSET:World_New:L3036 {
   printf(“%d\n”, $args->p2mCacheSize);
}
```

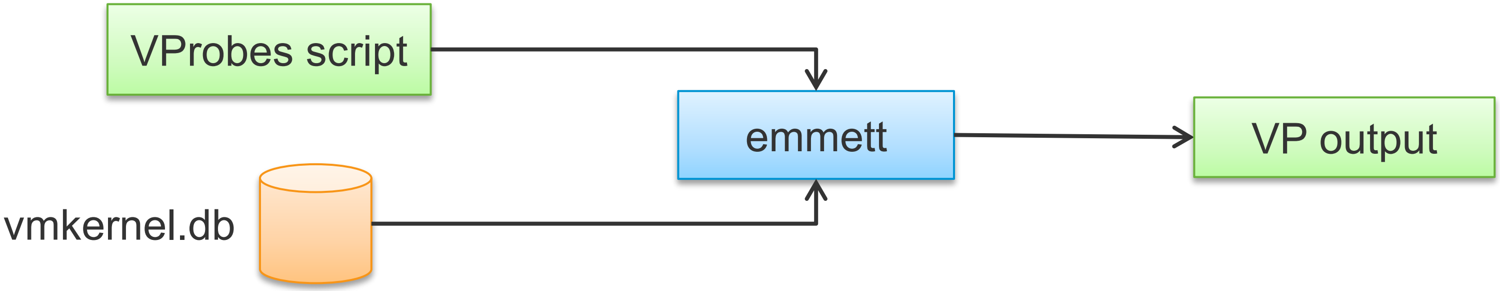
  - emmett translates the name reference to the correct register/memory lookup under-the-hood
  - Arguments and local variables are treated the same way
  - No support yet for static locals

Symbol Databases

- Pre-requisite to external types, line numbers and local variable support
- Symbol database creation (once per build):
  - `bora/vmcore/support/vprobes/util/createEmtSymbolDb.py`



- Symbol database usage:



Useful Links

- **User Guide:**

  http://engweb.eng.vmware.com/monitor/vprobes/doc/html/userGuide.html

- **Mailing list**

  vprobes@vmware.com

- **Web site**

  http://vprobes.eng.vmware.com