Removing Frame Pointers from ESXi

What We Have Now

- **VMM:**
  - No frame pointers in release builds
  - Frame pointers available in obj, opt, stats builds
- **Vmkernel, userworlds:**
  - All builds are compiled with frame pointers
- **Ole filed PR 236488 (Feb 2008)**

  “don't use frame pointers for vmk64 (at least not for release builds)”
- **Ole filed PR 928007 (Sep 2009)**

  “can we turn frame pointers off for vmkernel release builds?”

- **Goal: remove frame pointers from vmkernel (core, modules, and drivers) to improve performance on critical paths**

Frame Pointer Refresher

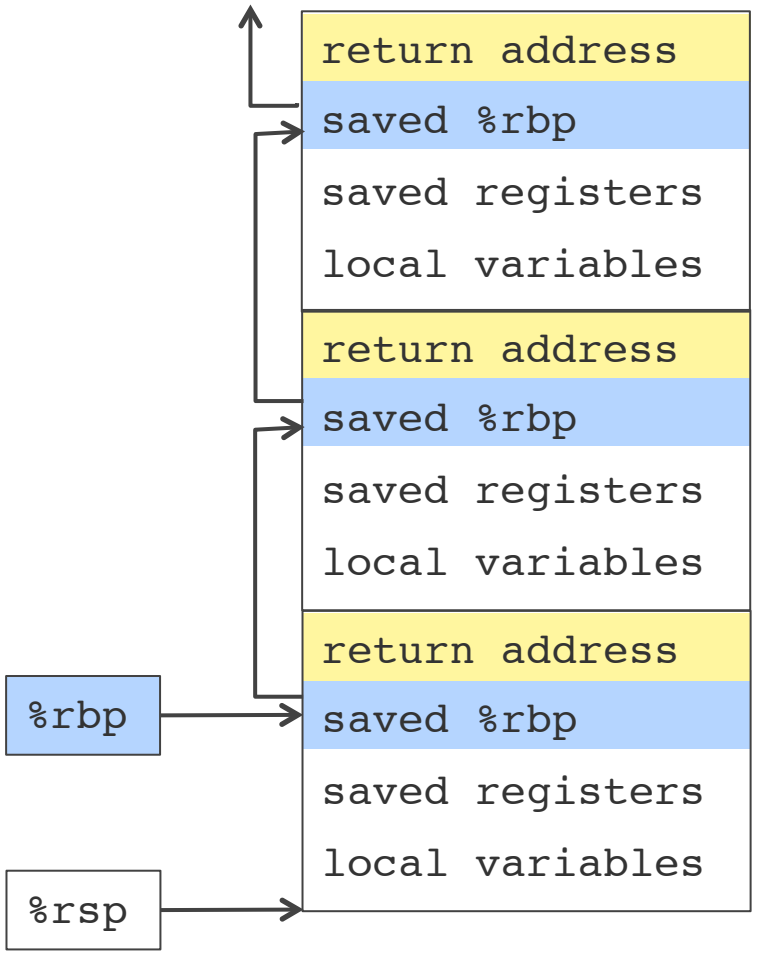
Function disassembly

```
push %rbp
mov  %rsp, %rbp

...

leave
ret
```

Stack Layout



What We Gain

- **Fewer instructions executed**
  - Remove three instruction per function:

```
push %rbp;  mov %rsp,%rbp;  leave
```

- **Less icache pressure**
  - vmkernel code size decreases by 111Kb (or 1.6%)
- **One additional register**
  - %rbp

- **Code should run faster**
- **Removing frame pointers is trivial:**

```
<     '-fno-omit-frame-pointer',
---
>     '-fomit-frame-pointer',
```

What We Might Lose

- **Stack walk ability**
  - Backtrace collection in vmkstats, VProbes, Panic() all rely on frame pointers
  - Developers, CPD: ability to manually walk stacks
- **Not affected:**
  - gdb backtraces, which rely on information that gcc provides in ***.eh_frame***
- **Can we make stack walks work?**
  - Option 1: link against libunwind, glibc (expensive, meant for debuggers)
  - Option 2: build our own stack unwind tables, ensure fast stack walks

How About Other OSes

- **Linux**
  - Frame pointers by default, can be disabled via CONFIG_FRAME_POINTER

- **Microsoft**
  - "*Debugging in the (Very) Large: Ten Years of Implementation and Experience*", from SOSP '09

"The decision to disable frame-pointer omission (FPO), in Windows XP SP2, was originally quite controversial. Programmers believed FPO was crucial to performance. However, extensive internal testing across a wide range of both desktop and server benchmarks showed that FPO provided no statistically provable benefit but significantly impaired post-mortem debugging of unexpected call stacks. Ultimately it was decided that the benefit of improving post-mortem debugging outweighed the cost of disabling FPO."

Is It Worth Our Effort?

- **It could impact latency-critical paths**
- **Performance team ran an experiment:**
  - 16 Linux VMs, each VM sends data to a another VM on a different ESX host
  - Both large message and small messages
  - Throughput (Gbps) increased 3%, cpu usage (Cpu/Gbits) decreased 10%
  - Performance impact was measurable

Evaluating the Options

- **Option 1: link against libunwind, glibc**
  - Very slow: between 10 and 200 Kcycles per backtrace (vs <1Kcycles)
  - Okay for debuggers, less so for profilers
  - Not clear how to specify the starting point (if not current location)
  - No solution for manual stack walks

- **Option 2: build custom *vmkunwind* tables**
  - Simple table structure, fast walks
  - Tables derived from the *eh_frame* information
  - Construct tables either at build time or module load time
  - Provide gdb macros for manual stack walks

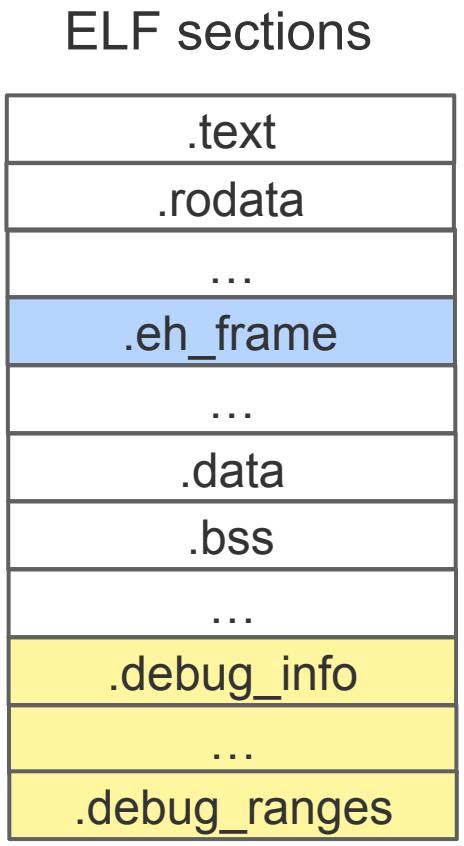
Current Plan

- **Build custom vmkunwind tables (option 2)**
- **Transparent change for clients that use `Util_StackWalk()` and `Util_Backtrace()`**
  - VProbes, Panic already use this interface
  - Vmkstats has recently switched to `Util_StackWalk()`
- **Userworlds unchanged, still have frame pointers**
  - UW stack walks use `World_StackWalk()`
- **New stack walker will work with and without frame pointers**
- **Backwards compatible**
- **Provide gdb macros for manual stack walks**

DWARF, Unwind Tables, and Stack Walks

DWARF Sections

- **DWARF = standard debugging format**
  - Used in the highlighted sections
- **Several debug_* sections**
  - Line number, type information
  - Used by debuggers
  - All removed when stripping the executable
- **The eh_frame section**
  - Contains call frame information
    - How to compute frame address
    - How to restore callee-saved registers
  - Regardless of whether the code is compiled with frame pointers
  - The section is not stripped



The eh_frame Section (Raw Data)

```
$ readelf –wf vmkernel

000000c8 0000002c 000000cc FDE cie=00000000 pc=418000000148..4180000001bb
  DW_CFA_advance_loc: 1 to 418000000149
  DW_CFA_def_cfa_offset: 16
  DW_CFA_offset: r6 (rbp) at cfa-16
  DW_CFA_advance_loc: 1 to 41800000014a
  DW_CFA_def_cfa_offset: 24
  DW_CFA_offset: r3 (rbx) at cfa-24
  DW_CFA_advance_loc: 7 to 418000000151
  DW_CFA_def_cfa_offset: 32
  DW_CFA_advance_loc1: 103 to 4180000001b8
  DW_CFA_def_cfa_offset: 24
  DW_CFA_advance_loc: 1 to 4180000001b9
  DW_CFA_def_cfa_offset: 16
  DW_CFA_restore: r3 (rbx)
  DW_CFA_advance_loc: 1 to 4180000001ba
  DW_CFA_def_cfa_offset: 8
  DW_CFA_restore: r6 (rbp)
```

FDE = Frame Description Entry (one per function)
CFA = Canonical Frame Address

The eh_frame Section (More Readable)

```
$ readelf –wF vmkernel

FDE cie=00000000 pc=4180000000148..4180000001bb
   LOC            CFA      rbx   rbp   ra
0000418000000148 rsp+8    u     u     c-8          push %rbp
0000418000000149 rsp+16   u     c-16  c-8          push %rbx
000041800000014a rsp+24   c-24  c-16  c-8          mov
                                                   sub  $8, %rsp
0000418000000151 rsp+32   c-24  c-16  c-8          mov
                                                   ...
                                                   add  $8, rsp
00004180000001b8 rsp+24   c-24  c-16  c-8          pop  %rbx
00004180000001b9 rsp+16   c-16  c-8                pop  %rbp
00004180000001ba rsp+8    c-8                      ret
```

The eh_frame Section (More Readable)

```
$ readelf –wF vmkernel

FDE cie=00000000 pc=4180000000148..4180000001bb
   LOC            CFA      rbx   rbp   ra
0000418000000148 rsp+8    u     u     c-8          push %rbp
0000418000000149 rsp+16   u     c-16  c-8          push %rbx
000041800000014a rsp+24   c-24  c-16  c-8          mov
                                                   sub  $8, %rsp
0000418000000151 rsp+32   c-24  c-16  c-8          mov
                                                   ...
                                                   add  $8, rsp
00004180000001b8 rsp+24   c-24  c-16  c-8          pop  %rbx
00004180000001b9 rsp+16   c-16  c-8                pop  %rbp
00004180000001ba rsp+8    c-8                      ret
```

Custom Tables

- **DWARF targets debuggers**
  - Complex instruction format, decoding is expensive
  - Tells how to restore all callee-saved registers

- **What we want:**
  - Simple table structure
  - Fast searches
  - Just stack walking, no register restore
  - Handle code with and without frame pointers

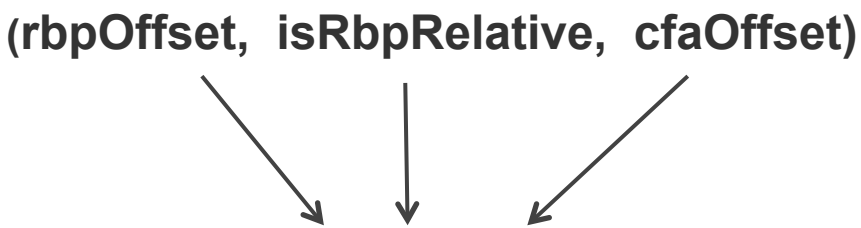
Vmkunwind Table

- **The table maps each RIP to a data triple:**

  (rbpOffset, isRbpRelative, cfaOffset)

- **The frame address (CFA) is:**
  - %rbp + cfaOffset, if isRbpRelative
  - %rsp + cfaOffset, otherwise
- **The new %rbp value is:**
  - CFA + rbpOffset, if rbpOffset ≠ 0
  - Unchanged, otherwise
- **The new %rsp value is:**
  - CFA

Example




```
0000418000000148   ( 0, 0,  8)      push %rbp
0000418000000149   (16, 0, 16)      push %rbx
000041800000014a   (16, 0, 24)      mov
                                    sub  $8, %rsp
0000418000000151   (16, 0, 32)      mov
                                    ...
                                    add  $8, %rsp
00004180000001b8   (16, 0, 24)      pop  %rbx
00004180000001b9   (16, 0, 16)      pop  %rbp
00004180000001ba   ( 0, 0,  8)      ret
```

Stack Walking Algorithm

Old:

```
StackWalk(rip, rbp):
  while validAddr(rbp):
    callback(rip);
    rip = *(VA*)(rbp + 8);
    rbp = *(VA*)rbp;
```

New:

```
StackWalk(rip, rbp, rsp):
  while validAddr(rsp):
    callback(rip);
    (rbpOff,rbpRel,cfaOff) =
        LookupTable(rip);
    if rbpRel:
      rsp = rbp;
    rsp += cfaOffset;
    rip = *(VA*)(rsp – 8);
    if rbpOff:
      rbp = *(VA*)(rsp - rbpOff);
```

Overall Table Structure

Module Table

Start Addr	End Addr	Table Pointer
0x100	0x200	
0x300	0x400	
…		

RIP	rbpOff	rbpRel	cfaOff
0x100	0	0	8
0x101	16	0	16
0x1ff	0	0	8

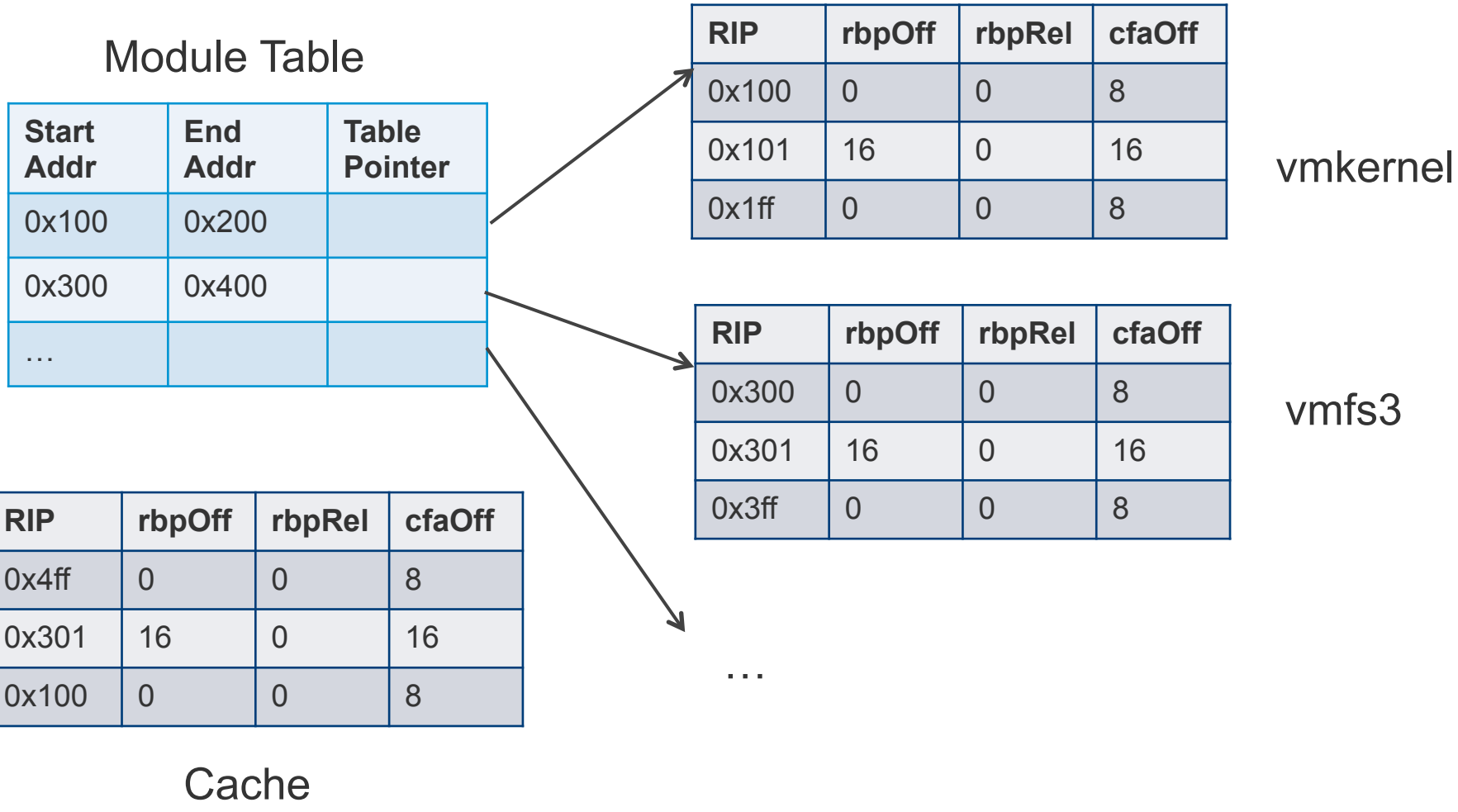
vmkernel

RIP	rbpOff	rbpRel	cfaOff
0x300	0	0	8
0x301	16	0	16
0x3ff	0	0	8

vmfs3

…

Overall Table Structure



Issues and Corner Cases

- **Backward compatibility**
- **Assembly functions**
  - Assembler supports directives for manually constructing tables
  - Switch to another stack (e.g., gate_entry)
- **Overlapping function addresses in eh_frame:**
  - Functions from different text sections, e.g., .text and .init.text
- **No-return functions**
  - Address pushed on stack is beyond function address range

Preliminary Results: Backtrace Collection Time

	Min Cycles	Average Cycles
With frame pointers	775	2011
With frame pointers + Optimized stack reads	450	717
No frame pointers + Optimized stack reads	1295	3293
No frame pointers + Optimized stack reads + With caching	494	730

Preliminary Results: Table Sizes

Module	With Frame Pointers	Without Frame Pointers
vmkernel	392 KB	640 KB
tcpip4	64 KB	112 KB
vmfs3	24 KB	40 KB
vsan	40 KB	64 KB
e1000	16 KB	24 KB
usbnet	8 KB	8 KB

Status

- **We have a prototype**
  - Constructs tables at build time
- **Next steps:**
  - Gradually check in, address corner cases, new issues
  - Alok works on constructing the tables at module load time
  - Provide gdb macros
- **Pre-requisites:**
  - Switch to gcc 4.6 (PR 908423)
  - Fix vmkernelCheckInstructions (PR 814365)