Soul of a New Machine

Jeff Chase
Duke University
Getting to Dune

• We want to talk about Dune…
• But let’s be sure we have the basics well in hand!
• The basics: **architectural foundations of protection**.
  – Protected mode (kernel mode) and “rings”
  – Kernel entry and exit: exceptions, interrupts, and handlers
  – Virtual memory maps (page tables) and the MMU
• And now we’re going to warp speed.
  – Intel **VT-x**: extensions for virtual machines
  – All the basics, but one instance per virtual machine context
  – And a new layer of trusted software (hypervisor) to coordinate
  – And Dune uses it in an “unexpected” way.
Processes and the kernel

• A (classical) OS lets us run programs as processes. A **process** is a running program instance (with a **thread**).
  – Program code runs with the CPU core in untrusted **user mode**.

• Processes are protected/isolated.
  – **Virtual address space** is a “fenced pasture”
  – **Sandbox**: can’t get out. **Lockbox**: nobody else can get in.

• The OS kernel controls **everything**.
  – Kernel code runs with the CPU core in trusted **kernel mode**.
Recap: OS protection

Know how a classical OS uses the hardware to protect itself and implement a **limited direct execution** model for untrusted user code.

- **Virtual addressing.** Applications run in sandboxes that prevent them from calling procedures in the kernel or accessing kernel data directly (unless the kernel chooses to allow it).

- **Events.** The OS kernel installs handlers for various machine events when it boots (starts up). These events include machine exceptions (faults), which may be caused by errant code, interrupts from the clock or external devices (e.g., network packet arrives), and deliberate kernel calls (traps) caused by programs requesting service from the kernel through its API.

- **Designated handlers.** All of these machine events make safe control transfers into the kernel handler for the named event. In fact, once the system is booted, these events are the only ways to ever enter the kernel, i.e., to run code in the kernel.
CPU mode: user and kernel

The current mode of a CPU core is represented by a field in a protected register.

We consider only two possible values: user mode or kernel mode (also called protected mode or supervisor mode).

If the core is in protected mode then it can:
- access kernel space
- access certain control registers
- execute certain special instructions

If software attempts to do any of these things when the core is in user mode, then the core raises a CPU exception (a fault).
x86 control registers

The details aren’t important.
See [en.wikipedia.org/wiki/Control_register]
Entering the kernel

• Suppose a CPU core is running user code in user mode:
  – The user program controls the core.
  – The core goes where the program code takes it…
  – …as determined by its register state (context) and the values encountered in memory.

• How does the OS get control back? How does the core switch to kernel mode?
  – CPU interrupts and exceptions (trap, fault)

• On kernel entry, the CPU transitions to kernel mode and resets the PC and SP registers.
  – Set the PC to execute a pre-designated handler routine for that exception type.
  – Set the SP to a pre-designated kernel stack.
## Exceptions and Interrupts

<table>
<thead>
<tr>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>caused by an instruction</td>
<td>caused by some other event</td>
</tr>
</tbody>
</table>

### Intentional
- **trap: system call**
  - open, close, read, write, fork, exec, exit, wait, kill, etc.

### Unintentional
- **fault**
  - invalid or protected address or opcode, page fault, overflow, etc.
- **interrupt**
  - caused by an external event: I/O op completed, clock tick, power fail, etc.

- **"software interrupt"**
  - software requests an interrupt to be delivered at a later time
User code runs on a CPU core in user mode in a user space. If it tries to do anything weird, the core transitions to the kernel, which takes over.

The kernel executes a special instruction to transition to user mode (labeled as “u-return”), with selected values in CPU registers.
Timer interrupts

The **system clock** (timer) interrupts periodically, giving control back to the kernel. The kernel can do whatever it wants, e.g., switch threads.
Native virtual machines (VMs)

- Slide a **hypervisor** underneath the kernel.
  - New OS layer: also called **virtual machine monitor (VMM)**.
- Kernel and processes run in a **virtual machine (VM)**.
  - The VM “looks the same” to the OS as a physical machine.
  - The VM is a sandboxed/isolated context for an entire OS.
- Can run multiple VM **instances** on a shared computer.
Virtualization in the Enterprise

- Consolidate under-utilized servers to reduce CapEx and OpEx
- Avoid downtime with VM Relocation
- Dynamically re-balance workload to guarantee application SLAs
- Enforce security policy

[Ian Pratt, Xen and the Art of Virtualization]
Implementing VMs

Recent CPUs support additional protected mode(s) for hypervisors (E.g., Intel VTx). When the hypervisor initializes a **VM context**, it selects some set of event types to **intercept**, and registers handlers for them.

Configured interceptions transfer control to a registered **hypervisor handler** routine. For example, a guest OS kernel accessing device registers may cause the physical machine to invoke the hypervisor to intervene.

In addition, the VM architecture has another level of indirection in the MMU page mappings (Intel’s **Extended Page Tables**). The hypervisor uses it to specify and restrict what parts of physical memory are visible to each guest VM.

A guest can map to or address a physical memory frame or command device DMA I/O to/from a physical frame if and only if the hypervisor permits it.

If any guest VM tries to do anything weird, then the hypervisor regains control and can see or do anything to any part of the physical or virtual machine state before (optionally) restarting the guest VM.
According to Dune

2.1 The Intel VT-x Extension

In order to improve virtualization performance and simplify VMM implementation, Intel has developed VT-x [37], a virtualization extension to the x86 ISA. AMD also provides a similar extension with a different hardware interface called SVM [3].

The simplest method of adapting hardware to support virtualization is to introduce a mechanism for trapping each instruction that accesses privileged state so that emulation can be performed by a VMM. VT-x embraces a more sophisticated approach, inspired by IBM’s interpretive execution architecture [31], where as many instructions as possible, including most that access privileged state, are executed directly in hardware without any intervention from the VMM. ...The motivation for this approach is to increase performance, as traps can be a significant source of overhead.

VT-x adopts a design where the CPU is split into two operating modes: VMX root and VMX non-root mode. VMX root mode is generally used to run the VMM and does not change CPU behavior, except to enable access to new instructions for managing VT-x. VMX non-root mode, on the other hand, restricts CPU behavior and is intended for running virtualized guest OSes.

Transitions between VMX modes are managed by hardware. When the VMM executes the VMLAUNCH or VMRESUME instruction, hardware performs a VM entry; placing the CPU in VMX non-root mode and executing the guest. Then, when action is required from the VMM, hardware performs a VM exit, placing the CPU back in VMX root mode and jumping to a VMM entry point. Hardware automatically saves and restores most architectural state during both types of transitions. This is accomplished by using buffers in a memory resident data structure called the VM control structure (VMCS).

In addition to storing architectural state, the VMCS contains a myriad of configuration parameters that allow the VMM to control execution and specify which type of events should generate VM exits. This gives the VMM considerable flexibility in determining which hardware is exposed to the guest. For example, a VMM could configure the VMCS so that the HLT instruction causes a VM exit or it could allow the guest to halt the CPU. However, some hardware interfaces, such as the interrupt descriptor table (IDT) and privilege modes, are exposed implicitly in VMX non-root mode and never generate VM exits when accessed. Moreover, a guest can manually request a VM exit by using the VMCALL instruction.

Virtual memory is perhaps the most difficult hardware feature for a VMM to expose safely. A straw man solution would be to configure the VMCS so that the guest has access to the page table root register, %CR3. However, this would place complete trust in the guest because it would be possible for it to configure the page table to access any physical memory address, including memory that belongs to the VMM. Fortunately, VT-x includes a dedicated hardware mechanism, called the extended page table (EPT), that can enforce memory isolation on guests with direct access to virtual memory. It works by applying a second, underlying, layer of address translation that can only be configured by the VMM. AMD’s SVM includes a similar mechanism to the EPT, referred to as a nested page table (NPT).

From Dune: Safe User-level Access to Privileged CPU Features, Belay et al., (Stanford), OSDI, October, 2012
Machine runs according to its configuration. If it encounters a condition with no valid configured action, it suspends processing and generates an exception for the controller.
IA Protection Rings (CPL)

- Actually, IA has four protection levels, not two (kernel/user).
- **IA/X86 rings (CPL)**
  - Ring 0 – “Kernel mode” (most privileged)
  - Ring 3 – “User mode”
  - Ring 1 & 2 – Other
- Linux only uses 0 and 3.
  - “Kernel vs. user mode”
- Pre-VT Xen modified to run the guest OS kernel to Ring 1: reserve Ring 0 for hypervisor.

[Fischbach]
Why aren’t (IA) rings good enough?

- CPL 3
- CPL 1
- CPL 0

Increasing Privilege Level

- Ring 0
- Ring 1
- Ring 2
- Ring 3

VM

hypervisor

guest kernel

???
A short list of pre-VT problems

Early IA hypervisors (VMware, Xen) had to emulate various machine behaviors and generally bend over backwards.

- IA32 page protection does not distinguish CPL 0-2.
  - Segment-grained memory protection only.
- Ring aliasing: some IA instructions expose CPL to guest!
  - Or fail silently…
- Syscalls don’t work properly and require emulation.
  - `sysenter` always transitions to CPL 0. (D’oh!)
  - `sysexit` faults if the core is not in CPL 0.
- Interrupts don’t work properly and require emulation.
  - Interrupt disable/enable reserved to CPL0.
Early approaches to IA VMMs

To implement virtualization on IA in the pre-VT days, you have to modify the guest OS kernel.

- To run a proprietary OS (i.e., Windows) you had to do it on the sly.
  - VMware: software binary translation (code morphing)
- If it’s an open-source OS (e.g., Linux) you can do it out in the open.
  - Xen: paravirtualization
    - Basically just a change to the HAL/PAL: a VM is just another machine type to support (<2% of the OS code).
- Of course you can always do full emulation! (pre-kvm qemu)
- All of this is washed away by VT.
- (Of course, hardware-based VMs were also done correctly on IBM machines a lonnnnng time ago.)
VT in a Nutshell

- **New VM mode bit**
  - Orthogonal to CPL (e.g., kernel/user mode)
- **If VM mode is off → host mode**
  - Machine “looks just like it always did” (“VMX root”)
- **If VM bit is on → guest mode**
  - Machine is running a guest VM: “VMX non-root mode”
  - Machine “looks just like it always did” to the guest, BUT:
  - Various events trigger gated entry to hypervisor (in VMX root)
  - A “virtualization intercept”: exit VM mode to VMM (VM Exit)
  - Hypervisor (VMM) can control which events cause intercepts
  - Hypervisor can examine/manipulate guest VM state and return to VM (VM Entry)
CPU Virtualization With VT-x

- Two new VT-x operating modes
  - Less-privileged mode (VMX non-root) for guest OSes
  - More-privileged mode (VMX root) for VMM

- Two new transitions
  - VM entry to non-root operation
  - VM exit to root operation

Execution controls determine when exits occur
- Access to privilege state, occurrence of exceptions, etc.
- Flexibility provided to minimize unwanted exits

VM Control Structure (VMCS) controls VT-x operation
- Also holds guest and host state
Dune in a Nutshell

• Provide safe user-level access to privileged CPU features
• Still a normal process in all ways (POSIX API, etc)
• Key idea: leverage existing virtualization hardware (VT-x)

Run App as a “process” in guest mode, and let it use all CPLs.
Hypervisor calls

VT VMCALL operation (an instruction) voluntarily traps to hypervisor. Similar to SYSCALL to trap to kernel mode.

Not needed for transparent virtualization: but Dune uses it to call the “real” kernel.
Memory management in Dune

- Configure the EPT to provide process memory
- User programs can then directly access the page table
Virtual Machines and Virtual Memory

Guest Virtual Addresses

Guest Page Tables

Guest Physical Addresses

Host Page Tables

Host Physical Addresses
“Classic Linux Address Space”

http://duartes.org/gustavo/blog/category/linux
x64, x86-64, AMD64: VM Layout

Figure 3.7: Virtual Address Configuration

<table>
<thead>
<tr>
<th>0xffffffffffffffffffffff</th>
<th>Reserved system area</th>
<th>End of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>0x80000000000000</td>
<td>Dynamic segments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Process segments</td>
<td>Beginning of memory</td>
</tr>
</tbody>
</table>

Figure 3.8: Conventional Segment Arrangements

<table>
<thead>
<tr>
<th>0x80000000000000</th>
<th>Dynamic segments</th>
<th>Source: System V Application Binary Interface AMD64 Architecture Processor Supplement 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack segment</td>
<td>N high addresses</td>
</tr>
<tr>
<td></td>
<td>…</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data segments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>0x400000</td>
<td>Text segments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Unmapped</td>
<td></td>
</tr>
</tbody>
</table>
Linux x86-64 VAS layout

The details aren’t important.
If we really want to know what the address space really looks like…

[On 64-bit linux]
Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process … Each row has the following fields:

- address - This is the starting and ending address of the region in the process's address space
- permissions - This describes how pages in the region can be accessed. There are four different permissions: read, write, execute, and shared. If read/write/execute are disabled, a '-' will appear instead of the 'r'/w'/x'. If a region is not shared, it is private, so a 'p' will appear instead of an 's'. If the process attempts to access memory in a way that is not permitted, a segmentation fault is generated. Permissions can be changed using the mprotect system call.
- offset - If the region was mapped from a file (using mmap), this is the offset in the file where the mapping begins. If the memory was not mapped from a file, it's just 0.
- device - If the region was mapped from a file, this is the major and minor device number (in hex) where the file lives.
- inode - If the region was mapped from a file, this is the file number.
- pathname - If the region was mapped from a file, this is the name of the file. This field is blank for anonymous mapped regions. There are also special regions with names like [heap], [stack], or [vdso]. [vdso] stands for virtual dynamic shared object. It's used by system calls to switch to kernel mode. [see http://www.trilithium.com/johan/2005/08/linux-gate/]

You might notice a lot of anonymous regions. These are usually created by mmap but are not attached to any file. They are used for a lot of miscellaneous things like shared memory or buffers not allocated on the heap. For instance, …the pthread library uses anonymous mapped regions as stacks for new threads.
VMs (or segments) are storage objects described by maps. A page table is just a block map for one or more VM segments. The page tables (e.g., one for each process) are stored in memory. As threads reference memory, the machine uses the current page table to translate the virtual addresses. The hardware hides the indirection from user programs.
Block maps

Large storage objects (e.g., files, segments) may be mapped so they don’t have to be stored contiguously in memory or on disk.

Idea: use a level of indirection through a map to assemble a storage object from “scraps” of storage in different locations.

The “scraps” can be fixed-size slots: that makes allocation easy because the slots are interchangeable (fixed partitioning). Fixed-size chunks of data or storage are called blocks or pages.

Examples: page tables that implement a VAS.

One issue now is that each access must indirect through the map…
From “Porting NetBSD to the AMD x86-64: a case study in OS portability”, Frank van der Linden
Example: Windows/IA32

- Two-level block map (page table) structure reduces the space overhead for block maps in sparse virtual address spaces.
  - Many process address spaces are small: e.g., a page or two of text, a page or two of stack, a page or two of heap.

- Windows provides a simple example of a hierarchical page table:
  - Each address space has a page directory ("PDIR")
  - The PDIR is one page: 4K bytes, 1024 4-byte entries (PTEs)
  - Each PDIR entry points to a map page, which MS calls a "page table"
  - Each map page ("page table") is one page with 1024 PTEs
  - Each PTE maps one 4K virtual page of the address space
  - Therefore each map page (page table) maps 4MB of VM: 1024*4K
  - Therefore one PDIR maps a 4GB address space, max 4MB of tables
  - Load PDIR base address into a register to activate the VAS
Two-level page table
32-bit virtual address
Two 10-bit page table index fields (PT1, PT2)
(10 bits represents index values 0-1023)

Step 1. Index PDIR with PT1
Step 2. Index 2L page table with PT2
What’s in a PTE

- **Dirty bit**: hardware sets on store to page, kernel clears.
- **Protection (writable, user)**: kernel sets to configure perms.
- **Reference bit**: hardware sets on access to page, kernel clears.
Virtual Address Translation

**Example only**: a typical 32-bit architecture with 4KB pages.

Virtual address translation maps a **virtual page number** (VPN) to a **page frame number** (PFN) in machine memory: the rest is easy.

Deliver fault to OS if translation is not valid and accessible in requested mode.
Virtual Addressing: Under the Hood

How to monitor page reference events/frequency along the fast path?
Virtual Memory in Virtual Machines

- Each guest runs its own kernel and manages its own page tables.
- So what prevents a guest from installing a mapping to anywhere in real/physical/machine memory?
  - What prevents a guest from messing with memory allocated to another guest?
- If a guest can initiate I/O on a DMA device, what prevents it from telling the device to move data to/from any region of memory?
- Various software-based VM systems have faked this, but doing it right requires hardware extensions (VT, VT-d).
Hardware Support for VM MMU

- A new page table!
- Maps GFN to HFN
  - Guest Frame Number to Host Frame Number
  - (Terminology varies)
- Hypervisor manages it
- Hypervisor loads map for the current VM
- Intel: Extended Page Table
- AMD: Nested Page Tables
Address Terminology

Addresses
- GVA: guest virtual address
- GPA: guest physical address
- SPA: system physical address

Or:
- GFN: Guest Frame Number
- HFN: Host Frame Number
Intel® Extended Page Table

- Guest can have full control over Intel® 64 page tables / events
  - CR3, INVLPG, page fault
- VMM controls Extended Page Tables
- CPU uses both tables
- EPT (optionally) activated on VM entry
  - When EPT active, EPT base pointer (loaded on VM entry from VMCS) points to extended page tables
  - EPT deactivated on VM exit
Filling EPT entry when EPT violate

1. Look up EPT TLB
2. EPT Violation VM Exit
3. VM Entry to Guest
Table 2: Average time (in cycles) of operations that have overhead in Dune compared to Linux.

<table>
<thead>
<tr>
<th></th>
<th>getpid</th>
<th>page fault</th>
<th>page walk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>138</td>
<td>2,687</td>
<td>35.8</td>
</tr>
<tr>
<td>Dune</td>
<td>895</td>
<td>5,093</td>
<td>86.4</td>
</tr>
</tbody>
</table>
What about I/O?

• How does a guest VM set up a DMA transaction for an I/O operation?
• What address does it give to the device to tell it where to read/write the data?
• Oops, we need to translate those addresses too.
  – *Another* page table (IOMMU)
• Virtualization is hard…
Benefits Of An IOMMU
Direct Device Assignment Example

- Guest OS
  - Virtual Device Driver
  - Virtual Device Emulator
    - Device Driver
    - Virtual Machine Monitor
    - Device Controller

- Guest OS
  - Virtual Device Driver
  - Device Controller

- Guest OS
  - Device Driver
  - Virtual Machine Monitor
  - IOMMU
  - Device Controller

Overhead reduced in path between Guest and Device
Benefits Of An IOMMU
Security and Isolation Example

System Memory

I/O Buffer

Device Controller

I/O Buffer

Device Controller

Malicious or Errant Write

System Memory

Protection Domain 1

I/O Buffer

Protection Domain 2

I/O Buffer

Device Controller

Device Controller

IOMMU

Write is blocked
Are we done?

- **Not quite.**
  - The latest hardware revs add support to enable a guest to receive interrupts without a VM exit.
  - Under certain conditions…
  - If the guest controls the device directly…
  - As allowed by the host…
  - **With this last fix, VM performance is now very close to bare-metal performance.**
    - See ELI paper, ASPLOS 2012, ACM Research Highlights
The Cause of x86 I/O Virtualization Overhead

- Privileged instructions trap to the hypervisor
- Traps cause an *Exit* (switch to the hypervisor context)
- Hypervisor emulates their behavior
- Hypervisor resumes the guest (switch to the guest context)

I/O intensive workloads cause many exits = overhead!
Direct Device Assignment

- Give the guest direct access to a physical device
- Devices nowadays know how to “self virtualize” (SR-IOV)
- Main I/O control and data paths do not require hypervisor intervention
Interrupt, Exits, Interrupt, Exits, Interrupt, Exits...

- No hypervisor intervention (?!)... except for handling more than **48,000** interrupts per second!
  - Exits due to external interrupts
  - Exits due to interrupt completions
Background: x86 Interrupt Handling

- I/O devices raise interrupts to notify the system software about incoming events
- CPU temporarily stops the currently executing code and jumps to a pre-specified interrupt handler

```
IDTR
Address
Limit

IDT Register

IDT
IDT Entry
IDT Entry
... 
IDT Entry

Interrupt Descriptor Table

IDT
IDT Entry
IDT Entry
... 
IDT Entry

Interrupt handlers

Handler for vector 1
Handler for vector 2
Handler for vector n
```
x86 Interrupt Handling in Virtual Environments

- **Host IDT**: handles physical interrupts
- **Guest IDT**: handles virtual interrupts
- If a physical interrupt – **including interrupts from assigned devices** – arrives while the guest is running the **CPU forces an Exit!**
ELI: Exitless Interrupts - Completion

- Guests write to the LAPIC EOI register
- Old LAPIC interface:
  - Hypervisor traps memory accesses $\rightarrow$ page granularity
- New LAPIC interface (x2APIC), required for Exitless Completions
  - Hypervisor traps accesses to MSRs $\rightarrow$ register granularity

ELI gives direct access only to the EOI register
Figure 2: Virtual memory in Dune.
Address Translation Uses

• **Process isolation**
  – Keep a process from touching anyone else’s memory, or the kernel’s

• **Efficient interprocess communication**
  – Shared regions of memory between processes

• **Shared code segments**
  – E.g., common libraries used by many different programs

• **Program initialization**
  – Start running a program before it is entirely in memory

• **Dynamic memory allocation**
  – Allocate and initialize stack/heap pages on demand
Address Translation (more)

• Program debugging
  – Data breakpoints when address is accessed

• Zero-copy I/O
  – Directly from I/O device into/out of user memory

• Memory mapped files
  – Access file data using load/store instructions

• Demand-paged virtual memory
  – Illusion of near-infinite memory, backed by disk or memory on other machines
Address Translation (even more)

• Checkpoint/restart
  – Transparently save a copy of a process, without stopping the program while the save happens

• Persistent data structures
  – Implement data structures that can survive system reboots

• Process migration
  – Transparently move processes between machines

• Information flow control
  – Track what data is being shared externally

• Distributed shared memory
  – Illusion of memory that is shared between machines
Virtual Memory Primitives for User Programs

Andrew W. Appel and Kai Li
Department of Computer Science
Princeton University

Abstract

Memory Management Units (MMUs) are traditionally used by operating systems to implement disk-paged virtual memory. Some operating systems allow user programs to specify the protection level (inaccessible, read-only, read-write) of pages, and allow user programs to handle protection violations, but these mechanisms are not always robust, efficient, or well-matched to the needs of applications.

We survey several user-level algorithms that make use of page-protection techniques, and analyze their common characteristics, in an attempt to answer the question, "What virtual-memory primitives should the operating system provide to user processes, and how well do today's operating systems provide them?"

This simple example of a user-mode fault handler is "dangerous," because it may lead the operating-system and hardware designers to believe that user-mode fault-handlers need not be entered efficiently (which is certainly the case for the "graceful error shutdown" example). But there are much more interesting applications of user-mode fault handlers. These applications exercise the page-protection and fault-handling mechanisms quite strenuously, and should be understood by operating-system implementors.

This paper describes several algorithms that make use of page-protection techniques. In many cases, the algorithms can substitute the use of "conventional" paging hardware for the "special" microcode that has sometimes been used. On shared-memory multiprocessors, the algorithms use page-protection hardware to achieve
User access to VM primitives

- TRAP - Handle page fault
- PROT1 - Protect a single page
- PROTN - Protect many pages
- UNPROT - Unprotect single page
- DIRTY - return list of dirty pages
- MAP2 - Map a page to two addresses
Concurrent Garbage Collection

[Source: Phil Howard]
Concurrent Garbage Collection
Concurrent Garbage Collection

Invariants

- Mutator sees only to-space pointers
- New objects contain to-space pointers only
- Objects in to-space contain to-space pointers only
- Objects in from-space contain from-space and to-space pointers

[Source: Phil Howard]
Concurrent Garbage Collection

• Use VM to protect from-space
• Collector handles access violations, validates objects and updates pointers
• Collector uses aliased addresses to scan in background

[Source: Phil Howard]
<table>
<thead>
<tr>
<th></th>
<th>GCBench</th>
<th>LinkedList</th>
<th>HashMap</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collections</td>
<td>542</td>
<td>33,971</td>
<td>161</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory use (MB)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation</td>
<td>938</td>
<td>15,257</td>
<td>10,352</td>
<td>1,753</td>
</tr>
<tr>
<td>Heap</td>
<td>28</td>
<td>1,387</td>
<td>27</td>
<td>1,737</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution time (ms)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>1,224</td>
<td>15,983</td>
<td>14,160</td>
<td>6,663</td>
</tr>
<tr>
<td>Dune</td>
<td>1,176</td>
<td>16,884</td>
<td>13,715</td>
<td>7,930</td>
</tr>
<tr>
<td>Dune TLB</td>
<td>933</td>
<td>14,234</td>
<td>11,124</td>
<td>7,474</td>
</tr>
<tr>
<td>Dune dirty</td>
<td>888</td>
<td>11,760</td>
<td>8,391</td>
<td>6,675</td>
</tr>
</tbody>
</table>

Table 6: Performance numbers of the GC benchmarks.
Software-based Distributed Shared Memory (DSM)

CPU
Memory
Mapping Manager

CPU
Memory
Mapping Manager

CPU
Memory
Mapping Manager

Shared Virtual Memory

[Source: Phil Howard]
Software-based DSM

- Consistent across nodes - each read gets the last value written
- Multiple readers/Single writer
- Handled the same as "regular" VM except for fetching and writing pages

[Source: Phil Howard]
Concurrent Checkpointing

- Stop all threads
- Save all thread states
- Save all memory
- Restart threads
- Stop all threads
- Save all thread states
- Make all memory read-only
- Restart threads
- Save pages in the "background" and mark as read/write

[Source: Phil Howard]
Figure 3: Average SPEC2000 slowdown compared to Linux for the sandbox, the sandbox using large pages, and Linux using large pages identically.