Motivation 1: DRAM a “Cache” for Disk

The full address space is quite large:
- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

Disk storage is ~30X cheaper than DRAM storage
- 8 GB of DRAM: ~ $12,000
- 8 GB of disk: ~ $200

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk.
Levels in Memory Hierarchy

<table>
<thead>
<tr>
<th></th>
<th>CPU regs</th>
<th>Cache</th>
<th>Memory</th>
<th>Disk Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>200 B</td>
<td>32 KB</td>
<td>4MB</td>
<td>20 GB</td>
</tr>
<tr>
<td>speed</td>
<td>3 ns</td>
<td>6 ns</td>
<td>60 ns</td>
<td>8 ms</td>
</tr>
<tr>
<td>$/Mbyte</td>
<td>$100/MB</td>
<td>$1.50/MB</td>
<td>$0.05/MB</td>
<td></td>
</tr>
<tr>
<td>block size</td>
<td>8 B</td>
<td>32 B</td>
<td>8 KB</td>
<td></td>
</tr>
</tbody>
</table>

larger, slower, cheaper


DRAM vs. SRAM as a “Cache”

DRAM vs. disk is more extreme than SRAM vs. DRAM
- access latencies:
  - DRAM is ~10X slower than SRAM
  - disk is ~100,000X slower than DRAM
- importance of exploiting spatial locality:
  - first byte is ~100,000X slower than successive bytes on disk
  vs. ~4X improvement for page-mode vs. regular accesses to DRAM
- “cache” size:
  - main memory is ~100X larger than an SRAM cache
- addressing for disk is based on sector address, not memory address
**Impact of These Properties on Design**

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- Line size?
- Associativity?
- Replacement policy (if associative)?
- Write through or write back?

What would the impact of these choices be on:

- Miss rate
- Hit time
- Miss latency
- Tag overhead

**Locating an Object in a “Cache”**

1. Search for matching tag
   - SRAM cache
     - Object Name
     - Tag Data
     - Data

2. Use indirection to look up actual object location
   - Virtual memory
     - Location Table
     - Data

---

Given the table and diagram, the focus is on understanding how cache organization affects performance metrics such as miss rate, hit time, miss latency, and tag overhead.
A System with Physical Memory Only

Examples:
- most Cray machines, early PCs, nearly all embedded systems, etc.

CPU's load or store addresses used directly to access memory.

A System with Virtual Memory

Examples:
- workstations, servers, modern PCs, etc.

Address Translation: the hardware converts *virtual addresses* into *physical addresses* via an OS-managed lookup table (*page table*)
Page Faults (Similar to "Cache Misses")

What if an object is on disk rather than in memory?
• Page table entry indicates that the virtual address is not in memory
• An OS trap handler is invoked, moving data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.

Servicing a Page Fault

Processor Signals
Controller
• Read block of length P starting at disk address X and store starting at memory address Y

Read Occurs
• Direct Memory Access
• Under control of I/O controller

I / O Controller Signals
Completion
• Interrupt processor
• Can resume suspended process
Motivation #2: Memory Management

Multiple processes can reside in physical memory. How do we resolve address conflicts?

(Virtual) Memory Image for Alpha Process

<table>
<thead>
<tr>
<th>Address</th>
<th>Virtual Component</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 03FF 8000 0000</td>
<td>Reserved</td>
<td>0</td>
</tr>
<tr>
<td>0000 0001 2000 0000</td>
<td>Stack</td>
<td>N-1</td>
</tr>
<tr>
<td>0000 0000 0001 0000</td>
<td>Not yet allocated</td>
<td>M-1</td>
</tr>
</tbody>
</table>

e.g., what if two different Alpha processes access their stacks at address 0x11fffff80 at the same time?

Soln: Separate Virtual Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks - "Pages" (both virtual and physical)
- Each process has its own virtual address space - operating system controls how virtual pages are assigned to physical memory

Virtual Addresses

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Virtual Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VP 1</td>
</tr>
<tr>
<td>N-1</td>
<td>VP 2</td>
</tr>
</tbody>
</table>

Physical Addresses

<table>
<thead>
<tr>
<th>Process 2</th>
<th>Physical Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PP 1</td>
</tr>
<tr>
<td>N-1</td>
<td>PP 2</td>
</tr>
<tr>
<td></td>
<td>PP 7</td>
</tr>
<tr>
<td></td>
<td>PP 10</td>
</tr>
</tbody>
</table>

(Read-only library code)
Motivation #3: Protection

Page table entry contains access rights information
- hardware enforces this protection (trap into OS if violation occurs)

<table>
<thead>
<tr>
<th>Process i:</th>
<th>Process j:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>VP 0:</td>
</tr>
<tr>
<td>Read?</td>
<td>Write?</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Page Tables

<table>
<thead>
<tr>
<th>Process i:</th>
<th>Process j:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1:</td>
<td>VP 1:</td>
</tr>
<tr>
<td>Read?</td>
<td>Write?</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

| VP 2:      | VP 2:      |
| Read? | Write? | Physical Addr |
| No   | No    | XXXXXXXX      |
| ...  | ...   | ...           |

Memory

VM Address Translation

\[ V = \{0, 1, \ldots, N-1\} \text{ virtual address space} \]
\[ P = \{0, 1, \ldots, M-1\} \text{ physical address space} \]

\[ MAP: V \rightarrow P \cup \{\emptyset\} \text{ address mapping function} \]

\[ MAP(a) = a' \text{ if data at virtual address } a \text{ is present at physical address } a' \text{ in } P \]
\[ = \emptyset \text{ if data at virtual address } a \text{ is not present in } P \]

Processor

Addr Trans Mechanism

Main Memory

Secondary memory

Fault handler

missing item fault

physical address

OS performs this transfer (only if miss)
### VM Address Translation

**Parameters**
- \( P = 2^p \) = page size (bytes). Typically 1KB-16KB
- \( N = 2^n \) = Virtual address limit
- \( M = 2^m \) = Physical address limit

Notice that the page offset bits don’t change as a result of translation.
Address Translation via Page Table

Page Table Operation

Translation
- Separate (set of) page table(s) per process
- VPN forms index into page table

Computing Physical Address
- Page Table Entry (PTE) provides information about page
  - if (Valid bit = 1) then page in memory.
    - Use physical page number (PPN) to construct address
  - if (Valid bit = 0) then page in secondary memory
    - Page fault
      - Must load into main memory before continuing

Checking Protection
- Access rights field indicate allowable access
  - e.g., read-only, read-write, execute-only
- Typically support multiple protection modes (e.g., kernel vs. user)
- Protection violation fault if don’t have necessary permission
**Integrating VM and Cache**

Most Caches "Physically Addressed"  
• Accessed by physical addresses  
• Allows multiple processes to have blocks in cache at same time  
• Allows multiple processes to share pages  
• Cache doesn’t need to be concerned with protection issues  
  - Access rights checked as part of address translation

Perform Address Translation Before Cache Lookup  
• But this could involve a memory access itself  
• Of course, page table entries can also become cached

**Speeding up Translation with a TLB**

"Translation Lookaside Buffer" (TLB)  
• Small, usually fully associative cache  
• Maps virtual page numbers to physical page numbers  
• Contains complete page table entries for small number of pages
Address Translation with a TLB

- process id
- virtual page number
- page offset
- virtual address
- N-1
- p-1
- 0
- physical page number
- valid dirty tag
- physical address
- byte offset
- tag
- index
- data
- TLB hit
- cache hit
- TLB
- Cache

Alpha AXP 21064 TLB

- page size: 64KB
- hit time: 1 clock
- miss penalty: 20 clocks
- TLB size: ITLB 8 PTEs, DTLB 32 PTEs
- placement: Fully associative

Alpha AXP 21064 TLB

- page size: 64KB
- hit time: 1 clock
- miss penalty: 20 clocks
- TLB size: ITLB 8 PTEs, DTLB 32 PTEs
- placement: Fully associative
**TLB-Process Interactions**

**TLB Translates Virtual Addresses**
- But virtual address space changes each time have context switch

**Could Flush TLB**
- Every time perform context switch
- Refill for new process by series of TLB misses
- ~100 clock cycles each

**Could Include Process ID Tag with TLB Entry**
- Identifies which address space being accessed
- OK even when sharing physical pages

**Using the TLB to help with LRU - How?**

---

**Virtually-Indexed Cache**

**Cache Index Determined from Virtual Address**
- Can begin cache and TLB index at same time

**Cache Physically Addressed**
- Cache tag indicates physical address
- Compare with TLB result to see if match
  - Only then is it considered a hit

**What extra info needs to be included in cache?**
**When Can Cache be virtually tagged?**
Generating Index from Virtual Address

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual page number</td>
<td>page offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>29</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical page number</td>
<td>page offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Size cache so that index is determined by page offset
- Can increase associativity to allow larger cache
- E.g., early PowerPC’s had 32KB cache
  - 8-way associative, 4KB page size

Page Coloring
- Make sure lower k bits of VPN match those of PPN
- Page replacement becomes set associative
- Number of sets = \(2^k\)

Alpha Physical Addresses

<table>
<thead>
<tr>
<th>Model</th>
<th>Bits</th>
<th>Max. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>21064</td>
<td>34</td>
<td>16GB</td>
</tr>
<tr>
<td>21164</td>
<td>40</td>
<td>1TB</td>
</tr>
<tr>
<td>21264</td>
<td>44</td>
<td>16TB</td>
</tr>
</tbody>
</table>

Why a 1TB (or More) Address Space?
- At $1.00 / MB, would cost $1 million for 1 TB
- Would require 131,072 memory chips, each with 256 Megabits
- Current uniprocessor models limited to 2 GB
Massively-Parallel Machines

**Example: Cray T3E**
- Up to 2048 Alpha 21164 processors
- Up to 2 GB memory / processor
- 8 TB physical address space!

**Logical Structure**
- Many processors sharing large global address space
- Any processor can reference any physical address
- VM system manages allocation, sharing, and protection among processors

**Physical Structure**
- Memory distributed over many processor modules
- Messages routed over interconnection network to perform remote reads & writes

---

Alpha Virtual Addresses

**Page Size**
- 8KB (and some multiples)

**Page Tables**
- Each table fits in single page
- Page Table Entry 8 bytes
  - 4 bytes: physical page number
  - Other bytes: for valid bit, access information, etc.
- 8K page can have 1024 PTEs

**Alpha Virtual Address**
- Based on 3-level paging structure
- Each level indexes into page table
- Allows 43-bit virtual address when have 8KB page size
Tree Structure
- Node degree ≤ 1024
- Depth = 3

Nice Features
- No need to enforce contiguous page layout
- Dynamically grow tree as memory needs increase

Mapping an Alpha 21064 Virtual Address

PT size: 1024 PTEs

10 bits 13 bits

13 bits

21 bits

PTE size: 8 bytes
### Virtual Address Ranges

<table>
<thead>
<tr>
<th>Binary Address</th>
<th>Segment</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1...1 11 xxxx...xxx</td>
<td>seg1</td>
<td>Kernel accessible virtual addresses</td>
</tr>
<tr>
<td>1...1 10 xxxx...xxx</td>
<td>kseg</td>
<td>Kernel accessible physical addresses</td>
</tr>
<tr>
<td>0...0 0x xxxx...xxx</td>
<td>seg0</td>
<td>User accessible virtual addresses</td>
</tr>
</tbody>
</table>

- Information maintained by OS but not to be accessed by user
- No address translation performed
- Used by OS to indicate physical addresses
- Only part accessible by user program

### Address Patterns

- Must have high order bits all 0’s or all 1’s
- Currently 64-43 = 21 wasted bits in each virtual address
- Prevents programmers from sticking in extra information
- Could lead to problems when want to expand virtual address space in future

### Alpha Seg0 Memory Layout

#### Regions
- **Data**
  - Static space for global variables
  - Allocation determined at compile time
    - Access via $gp
- **Text**
  - Stores machine code for program
- **Stack**
  - Implements runtime stack
  - Access via $sp
- **Reserved**
  - Used by operating system
    - Shared libraries, process info, etc.

#### Alpha Seg0 Memory Layout Diagram

```
3FF FFFF FFFF
3FF 8000 0000
001 4000 0000
001 2000 0000
000 0001 0000
```

- **Reserved**
- **Not yet allocated**
- **Dynamic Data**
- **Static Data**
- **Not used**
- **Text (Code)**
- **Stack**
- **Not yet allocated**
- **Reserved**

$sp$
### Alpha Seg0 Memory Allocation

#### Address Range
- User code can access memory locations in range
  - 0x0000000000010000 to 0x000003FFFFFF
- Nearly $2^{32} = 4.3980465 \times 10^{12}$ byte range
- In practice, programs access far fewer

#### Dynamic Memory Allocation
- Virtual memory system only allocates blocks of memory as needed
- As stack reaches lower addresses, add to lower allocation
- As break moves toward higher addresses, add to upper allocation
  - Due to calls to `malloc`, `calloc`, etc.

---

### Minimal Page Table Configuration

#### User-Accessible Pages
- VP4: Shared Library
  - Read only to prevent undesirable interprocess interactions
  - Near top of Seg0 address space
- VP3: Data
  - Both static & dynamic
  - Grows upward from virtual address 0x140000000
- VP2: Text
  - Read only to prevent corrupting code
- VP1: Stack
  - Grows downward from virtual address 0x120000000

---
### Partitioning Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>0x001 2000 0000</th>
<th>0x001 4000 0000</th>
<th>0x3FF 8000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1:</td>
<td>0</td>
<td>0</td>
<td>511</td>
</tr>
<tr>
<td>Level 2:</td>
<td>576</td>
<td>640</td>
<td>576</td>
</tr>
<tr>
<td>Level 3:</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Mapping Minimal Configuration**

- **VP1**: 0x0000 03FF 8000 0000
- **VP2**: 0x0000 0001 2000 0000
- **VP3**: 0x0000 0001 4000 1FFF
- **VP4**: 0x0000 0001 2000 1FFF
- **VP5**: 0x0000 0001 1FF 5000
Increasing Heap Allocation

Without More Page Tables
- Could allocate 1023 additional pages
- Would give ~8MB heap space

Adding Page Tables
- Must add new page table with each additional 8MB increment

Maximum Allocation
- Our Alphas limit user to 16B data segment
- Limit stack to 32MB

Expanding Alpha Address Space

Increase Page Size
- Increasing page size 2X increases virtual address space 16X
  - 1 bit page offset, 1 bit for each level index

Physical Memory Limits
- Cannot be larger than kseg
  - VA bits ≥ PA bits
- Cannot be larger than 32 + page offset bits
  - Since PTE only has 32 bits for PPN

Configurations
- Page Size  8K  16K  32K  64K
- VA Size    43  47  51  55
- PA Size    41  45  47  48
### Page Size Trade-offs

- As page size increases?

- As page size decreases?

### Main Theme

**Programmer's View**
- Large “flat” address space
  - Can allocate large blocks of contiguous addresses
- Process “owns” machine
  - Has private address space
  - Unaffected by behavior of other processes

**System View**
- User virtual address space created by mapping to set of pages
  - Need not be contiguous
  - Allocated dynamically
  - Enforce protection during address translation
- OS manages many processes simultaneously
  - Continually switching among processes
  - Especially when one must wait for resource
    » E.g., disk I/O to handle page fault