Memory Management, Performance Analysis, and Optimization

Introduction
Memory – the place where instructions and various data are stored
Has a significant impact on the behaviour of our systems

When managing memory in embedded systems
An improperly designed memory and access scheme
Can determine if a design succeeds or fails

Major concerns
Avoid dangerous allocation
Result in loss of deterministic behavior
Creation of deadlock situations
Minimize or reduce overhead during memory allocation

Today it seems that in many applications
Memory almost free

Many cases when it’s not
➢ Weight
   Aircraft
   Spacecraft
➢ Cost
   Very high volume consumer products
   Automotive
   Televisions
➢ Power consumption
   Portable applications
   This is becoming increasingly critical
Note – these are all embedded applications

In such cases
Must optimize use of memory
Here use refers to amount

As we saw earlier
Memory comprised of several different colors
   Code space
   Data space
   System space
At a high level we are concerned about the affects of memory on:
- The time our program takes to execute
- The amount of memory our program needs
- Ability to meet critical time constraints

When we design our systems, we must consider and manage all of these.
As we've seen in most embedded applications, time is a critical parameter.

Focusing on time and runtime specifically, our memory management is concerned with the following:
- Managing process stack(s)
- Allocation of memory
  - Static
    - Partition of memory
      - Code
      - Data
      - System
  - Dynamic
    - Allocation of memory resources for processes

When managing memory in embedded systems, major concerns are:
- Avoid dangerous allocation
- Result in loss of deterministic behavior
- Creation of deadlock situations
- Minimize or reduce overhead during memory allocation

**Memory Loading**
Let’s begin by taking a top-level view of memory.

To be able to manage memory efficiently for:
- System
- User
Must have some idea of how much each needs
- Available (percentage)

Recall once again we optimize to focus most of resources.
On getting target task completed
We defined *memory loading* as
Percentage of usable memory being used for task

Here we are referring to main memory
Not the cache if used

**Main Memory Map**

Developing memory map
Useful for understanding memory allocation and use
We see an example in the accompanying figure

The map specifies the allocation and use of
Each location in the physical memory address space
At bare minimum memory map should identify
Data and code space
Typical memory map for a small 16 bit machine
Presented in accompanying figure

Memory map lists the addresses in memory
Allocated to each portion of the memory
Note this is primary physical memory address space
Not segregating memory types

**System Memory**
Contains
Operating system
Other system level programs and drivers

**Nonvolatile RAM**
Potentially optional memory dedicated to holding data
Must be retained if power lost

**Instructions / Firmware**
Holds user application
In embedded system
May be flash type memory

**Static Data Area**

![Memory Map Diagram](image-url)
Holds user application data
   Constants
   Other static data

Heap or Dynamic Data Area
   Holds dynamic user application data
   Data space allocated using malloc in C or new in C++
   Dynamic data structures
   Linked lists

Stack Area
   Holds automatic user application data
   Local variables
   Data passed into or returned from function call

Memory Mapped I/O
   Supports calling interface to external user peripherals
   Read and write operations to such peripherals
   Appear as reads and writes to memory locations

From a high level perspective
   Memory subsystem is comprised of two basic types
      RAM and ROM
      ROM is used to hold words that are not expected to change at runtime
      RAM is used to hold words that may change at runtime
   Portion of the RAM memory may be allocated for nonvolatile RAM
      Used for data that needs to be retained if power removed from system
   If the design is using memory mapped I/O
      Then all of physical memory will not be available for data or code

Note it is possible for required code and data space
   To exceed total available primary memory
   Techniques called
      *Virtual memory* and *Overlays*
      Support such situation

When dealing with such situations
   We must spend time bring these into primary memory
Total memory loading will be sum of individual loading
  Instructions
    Typically ROM
  Stack
  RAM

Loading given by
\[ M_T = M_I \times P_I + M_R \times P_R + M_S \times P_S \]

The values \( M_I \) reflect memory loading
  For each portion of memory
The values \( P_i \) represent the percentage of total memory
  Allocated for program
\( M_T \) will be expressed as a percent

**Example**

Let system be implemented as follows
- \( M_I = 15 \) megabytes
- \( M_R = 100 \) kilobytes
- \( M_S = 150 \) kilobytes

- \( P_I = 55\% \)
- \( P_R = 33\% \)
- \( P_S = 10\% \)

We get a value for \( M_T \) of
\[ M_T = 0.55 \times \frac{15}{15.25} + 0.33 \times \frac{0.1}{15.25} + 0.1 \times \frac{0.15}{15.25} \]
\[ M_T = 54\% \]

Observe
  We do not include memory mapped I/O / DMA space in calculation
  Fixed by hardware design

**Considerations**
  Allocate minimum amount necessary
  Be sure to allow room for future growth
  Leave remaining amount of RAM
  For application program(s)
**Instruction / Firmware Area**

This portion of memory space
Generally ROM of one form or another
Reason called firmware

Contains instructions to implement application
Memory loading computed by
Dividing number of user locations
By maximum allowable
We get

\[ M_I = \frac{U_I}{T_I} \]

**RAM Area**

This portion of memory space
Generally for storing
Program data of one form or another
Global variables
Occasionally used for storing
Instructions
Done to improve fetch speed
Support modifiable instructions
Generally we avoid such thing
Reason called firmware
Size determined at design time
Can only determine loading
After design completed

Memory loading computed by
Dividing number of user locations
By maximum allowable RAM area
We get

\[ M_R = \frac{U_R}{T_R} \]

**Stack Area**

This portion of memory space
Used to store
Context information
Auto variables
Depending upon design
Can have multiple stacks
In this area of memory
From point of view of memory loading calculations
Model as one single stack

Capacity generally determined at design time
Size based upon use at runtime

Can establish a bound
We assume a maximum number of tasks
Call that number \( t_{\text{max}} \)
We next assume maximum allocation for each task
Call allocation \( s_{\text{max}} \)
From these we compute maximum stack size
\[ U_S = s_{\text{max}} \times t_{\text{max}} \]

Memory loading computed by
Dividing \( U_S \)
By maximum allocated stack area

We get
\[ M_S = \frac{U_S}{T_S} \]

Memory Allocation
Occam’s razor applies
Never allocate more space that necessary
Most operating systems give control over stack size
Define size that is large enough without being too much
Be certain to understand problem
Stack overflow can be dangerous
If developing multithreaded to multitasking applications
Without purchased kernel
Carefully manage how stack allocated and deallocated

Working With Memory – System Space
Let’s begin with one portion of memory
This is working memory system needs at runtime
Does not include system code space
Recall we’re working with multiple tasks / threads
Whenever have context switch
May need to save current state
Why do we say may rather than must
Some architectures
Automatically save context
Switching to completely new context
Rather than
Saving old
Loading new

Restore state information on return

Implemented in several ways
✓ Task Control Block
  Best for full featured operating system
✓ Stack(s)
  Best for
    Interrupt only
    Foreground/Background

Task Control Block
Looking first at task control blocks

In task based approach
Each process represented in the operating system
By a data structure called Task Control Block – TCB
also known as a process control block

TCB contains all of the important information about the task
✓ A typical TCB contains following information
✓ Pointer (for linking the TCB to various queues)
✓ Process ID and state
✓ Program counter
✓ CPU registers
✓ Scheduling information (priorities and pointers to scheduling queues)
✓ Memory management information (tag tables and cache information)
✓ Scheduling information (time limits or time and resources used)
✓ I/O status information (resources allocated or open files)
TCB allocation may be static or dynamic
- Static allocation
  Typically used in embedded systems with no memory management
  Are a fixed number of task control blocks
  Memory is allocated at system generation time
  Placed in dormant or unused state.

When a task initiated
  TCB created
  Appropriate information entered

TCB is then placed into ready state by scheduler

From the ready state
  Will be moved to the execute state by dispatcher

When a task terminates
  Associated TCB returned to dormant state

With fixed number of TCBs
  No runtime memory management is necessary
  One must be cautious not to exhaust supply of TCBs

- Dynamic Allocation
  Variable number of task control blocks
  Allocated from the heap at runtime

When a task initiated
  TCB created
  Appropriate information entered

TCB is then placed into ready state by scheduler

From the ready state
  Will be moved to the execute state by dispatcher

When a task terminates
  Associated TCB memory is returned to heap storage
  With a dynamic allocation
  Heap management must be supported

Dynamic allocation suggests an unlimited supply of TCBs
However the typical embedded application has limited memory
Allocating too many TCBs can exhaust the supply

Dynamic memory allocation scheme
Generally too expensive for smaller embedded systems

Stacks
Stack is rather simple data structure used for storing information
Associated with task
Area set aside in memory
Part of system allocation

Information held in data structure similar to TCB
Denoted stack frame or activation record
Information is about the same
Illustrated in figure

When stack used must develop procedures
- Save information to stack
  Invoked by interrupt handler
  Before context switch
  Immediately after interrupts have been disabled
- Restore
  Upon return to old context
  Immediately before interrupts enabled
  If only single level of interrupt

Stack data type
Generally supports following operations
Push
  Add to top of stack
Pop
  Remove from top of stack
Peek
  Look at top of stack
We distinguish three kinds of stack:
- Runtime stack
- Application stack
- Multiprocessing stack

**Runtime Stack**
In stack-based systems:
- Runtime stack under system control
- Possibly shared by other processes
- Size known a priori
  - Must ensure not too many stack frames pushed on
  - Will result in Overflow condition
- Most certainly deterministic behavior
  - No dynamic creation
  - Generally try to build in buffer space
  - Defensive design
  - Want to accommodate unexpected

**Application Stacks**
This is an interesting approach:
In addition to runtime stack:
Useful to be able to work with multiple stacks:
That is we are working with:
- Runtime stack
- Application stacks
- Simplify management of multiple tasks in interrupt environment

Application stacks:
Different from stacks in multiprocessing design:

In such case on interrupt:
Runtime stack:
- Holds pointer to application stack
- Associated with initial or interrupting process
- Process works with new stack
  If interrupted:
  1. Context held on current application stack
  2. Pointer to new application stack
    - Placed on runtime stack

Save and restore
Interface functions must be modified
To store / restore wrt current context
As runtime stack is unwound
Provides very fast context switch

**Multiprocessing Stacks**
Similar to main runtime stack
When process started
Among other resources
Allocated its own stack space

In contrast to application stacks which are
Managed by foreground task
Assuming foreground / background vs. task/thread architecture

Process stack managed by owner process
Allocated from heap when process created
Returned to heap when process exits

**Working With Memory – Code and Data Space – Part 1**

Dynamic Memory Allocation
Usually interpreted as process of allocating memory at runtime
Associated with some extensible data structure
- Linked list
  Which works well for an embedded system
- Heap

C malloc and C++ new
Examples of such routines

Remember in embedded systems
We have two kinds of memory
ROM
RAM
May be cache or main memory

Here we are more concerned with managing main memory to accommodate
- Programs larger than main memory
- Multiple processes in main memory
- Multiple programs in main memory
Traditional virtual memory schemes
Are completely nondeterministic
Rarely used in real time systems
Particularly those with hard deadlines

Let’s look at several schemes
That work well in embedded systems

Swapping
Memory segregated into two portions
- System
- User
  Code and data space for user programs

Simplest method for accommodating multiple programs called **swapping**
System remains resident in memory
Assumes only single user program resident in memory at a time
  Same holds true for processes
  Typically only single task in memory
  At any one time

Assume program comprised of several processes
First process executing when second process must be run
  Similar to subroutine call
  More complex
First process suspended
  Swapped to secondary storage device
  Will use word secondary to refer to non-runtime memory
  Could be ROM somewhere
  Since dealing with firmware
  Context also saved
  We could be working out of cache
  Swap to main
Second process with context
  Loaded into user space
  Activated by dispatcher

Note such scheme potentially deadly in time critical systems
Timing can be deterministic if program / process well understood

If such scheme used
Want to ensure process/program execution time
Long compared to swap time
Processes
Chemical processing
Thermal control

Overlays
An overlay is a poor man’s version of virtual memory
Overlay will be in ROM
Used to accommodate program
Larger than main memory
Program segmented into number of sections called overlays
Usually one main section plus remaining overlays
Main section usually contains following
• Top level routine
• Code to perform overlay process
• Data segment for shared data
• Overlay segment
When overlay executed
New overlay segment replaces current overlay segment
In main memory
Care must be taken in designing such systems
• Code in each overlay selected carefully
• Cannot just cut program into pieces that fit
  Two halves of loop in different overlays
  Can result in thrashing
• Usually segmentation hand tailored

Multiprogramming
As name suggests multiprogramming
Permits one to run multiple programs in same memory space (RAM
We consider two versions
Programs in which number of tasks
Fixed
Variable
Fixed Number

Similar to paging
Useful scheme when number of tasks
Known in advance
True for many embedded systems
User space divided into number of fixed size partitions
Tasks must reside in contiguous partitions
Linking becomes extremely difficult otherwise
When preempted
Partition swapped to disk
Or slower memory for embedded systems
Or RAM
Or code portion address is saved
This means something different in embedded systems

As seen in virtual memory schemes
Using fixed size partitions
Leads to memory fragmentation
Fragmentation can happen in several ways
Consider fixed size partitions of 2 K
Let 3 jobs be brought into memory
J1 - 1.5k
J2 - 0.5K
J3 - 2.1K
Allocated successive partitions in memory
4 partitions required
Let J2 finish and be swapped out
What does this mean
Problems
J2
Leaving has left hole that can only be filled by job <= 2k
Consumes 2k partition leaving 3/4 unused
J3
Requires 2 partitions
2k
0.1k
Very wasteful of memory
If these 3 jobs consume last of memory
New job of 1k cannot enter
Although will fit

Variable Number
Similar to segments
Memory allocation for variable number of tasks scheme
Similar to paged virtual memory scheme
Allocation determined by process requirements
When loaded into memory
Works well when number of tasks
Unknown
Variable
Waste of memory by
Misfit into partition size virtually eliminated
Holes remains
Can eliminate by using compaction schemes
Moves pieces of used memory
Into contiguous locations

**Working With Memory – Code and Data Space – Part 2**

The Cache

Cache is a small fast memory that temporarily holds
Copies of blocks data and program instructions
From the main memory
Increased speed of cache memory over that of main memory components
Offers prospective for programs to execute much more rapidly
If the instructions and data can be held in cache

Many of today’s higher performance processors
Implemented around the Harvard architecture
Will internally support a both an
\textit{icache} (instruction cache)
\textit{dcache} (data cache)

Will now examine the concept of caching in greater detail
\checkmark Will look first at the ideas behind caching
\checkmark What cache is, why it works
\checkmark Will then examine several alternative caching schemes

**A Clever Observation – Locality of Reference**

Goal
Reduce number of accesses
Make each access short as possible
Utilized to much greater extent in today’s memories
Ideally would like to make all memory as fast as technology allows
Such action has associated cost

\textbf{-16-}
As noted memories near bottom are expensive
Support circuitry for such memories also expensive
Additional circuitry required
Power supplies to support

Almost all programs executed today written using procedural paradigm
If we analyze how such programs
Designed
Execute

Discover interesting phenomenon
Execution generally occurs
Sequentially
Small loops
Number of instructions
Means overall progress forward through program
Executing at much lower rate than
Access times of fastest memory
Put another way
With respect to entire program
We are executing within a small window
That moves forward through program
This is shown in diagram

Formally such phenomenon called
Locality of reference
We recognize program executing only few instructions
Within small window

Types of Locality
Temporal Locality
If an item has been accessed recently
It will tend to be accessed again soon
Want to keep more recently accessed items closer to the processor
When we must evict items to make room for new ones
Try to keep more recently accessed items

Spatial Locality
If an item has been accessed recently
Nearby items will tend to be accessed soon
Want to move blocks consisting of multiple contiguous words to upper level
Locality guides caching

Benefits of locality
If we can keep those few instructions
in fast memory
Program will appear to be executing out of fast memory
Gain benefits of such speed
Reduced cost

Important point
Approach works provided
Area within which we are executing is in window
Method can easily be defeated with
Large loops
Branches outside window

Terminology
Block
Minimum unit of information transfer between levels of the hierarchy
Block addressing varies by technology at each level
Blocks are moved one level at a time

Upper vs. lower level
Upper closer to CPU
Lower is farther away

Hit
Data appears in a block in that level
- Hit rate
  Percent of accesses hitting in that level
- Hit time
  Time to access this level
- Hit time
  Access time + Time to determine hit/miss

Miss
Data does not appear in that level
Must be fetched from lower level
- Miss rate
  Percent of misses at that level
  1 – hit rate
- Miss penalty
  Overhead in getting data from a lower level
Miss penalty
Lower level access time + Replacement time + Time to deliver to processor
Miss penalty is usually MUCH larger than the hit time

Cache Access Time
Average access time
Access time = (hit time) + (miss penalty) x (miss rate)
Want
High hit rate
Low hit time
Given in clock cycles
Since miss penalty is large
Average Memory Access Time – AMAT
Apply average access time to entire hierarchy.

Architecture
Let’s now look at portion of memory hierarchy
Presented earlier
We’ll not consider
Archival storage
ROM or CDROM
Registers
We will focus on
Hard Drive
RAM
Cache
Secondary memory
$2^{18}$ pages
Primary memory
$2^{10}$ pages
Page = 4 blocks
Block = 1 K words
Word = 4 bytes

Cache System Architecture
When using a caching scheme
Goal is to operate out of cache memory to greatest extent possible
Typically SRAM
When program execution needs instruction or data not in the cache
Must be brought in from main memory
Typically DRAM.
The block diagram for the architecture appears as follows

Let’s examine cache memory and caching techniques first
Once again idea
In one sense
Take advantage of locality of reference
Instructions
Data
To minimize access time
Larger sense
Can use caching techniques
Many places to optimize performance
Internet
Bit images cached locally to improve display speed
Network file systems
Temporarily maintain local copy
To avoid having to retransfer
Based upon assumption will be using again in near future

Designing a Cache System
Caching requires certain amount of local memory
Size determines how much information can be stored locally

**High Level Description**
Program begins executing
Encounters needed data or instruction
Check cache
If in cache
Have a *cache hit*
Use
Else
Have *cache miss*
Must go get from somewhere else
Bring in new block of data or instructions
If room left in cache
    Store block
Else
    Must make room
    Remove existing block
    If block has been modified
        Save changes
    Else
        Discard
    Write new block in its place

Important issues
✓ How do we know something is not in cache
✓ Where do we go to find something if not in cache
    What if not there
✓ How do we know if room left in cache
✓ How do we know if information in cache modified
✓ How do we select block to replace
✓ Selecting the block size

Caching – A Direct Mapped Implementation

A Specification
Will address each question as we build a cache
Implementation scheme called direct mapped cache

An initial thought
Determining or specifying block size

Block Size Tradeoff
In general larger block size takes advantage of spatial locality
However, larger block size means larger miss penalty
    Takes longer time to fill up the block
If block size is too big relative to cache size
    Miss rate will go up
    Too few cache blocks

First step is specification
The cache and main memory will store 32 bit words.
The cache size will be 64 K words.
The cache will be organized as 128 0.5 K word blocks.
The cache will implement a direct mapped replacement algorithm.
Memory addresses will be 32 bits.
Main memory size will be 128 M words.
Main memory will be organized as 2 K pages
Each page will hold 128 blocks.

Following diagrams illustrate initial view of system

**Preliminary Design**
During normal operation
- Instruction or data fetch from memory
- Data write to memory proceeds as discussed earlier
- Address (and data as appropriate)
  - Provided and the read or write operation is executed
When the target address is not found in the cache
- cache miss occurs
  - Under such circumstances required data or instruction
    - Must be copied into the cache from main memory

Rather than bring a single word into the cache
- When such a transfer is needed
  - Complete block containing the required word is brought in
- Destination to where the new block is copied
- Determined by the replacement algorithm designed into the cache
  - Design under discussion will use the *direct mapping* algorithm
    - One of the simpler ones.

Main memory page size set equal to cache size
- Each page will contain a corresponding number of blocks
  - Will therefore contain $size_{\text{main memory}} \mod size_{\text{cache}}$ pages
- Thus each main memory page will contain
- A block 0, block 1, etc.
When a block is brought into the cache from main memory
- It is placed into correspondingly numbered block in cache
  - Main memory
    - *block 0* will always be placed (or directly mapped)
      - Into the cache *block 0* slot
    - *block 1* will always be placed
      - Into the cache *block 1* slot, etc.
A Logical Interpretation of an Address

The system memory address is 32 bits

Only 18 will be needed for the cache address

If byte addressing is supported

The 18 cache address bits are interpreted as presented in following figure

Bits A1 – A0
Each data or instruction word is 32 bits or 4 bytes long
Bits A1 and A0 identify a byte within a word

Bits A10 – A2
Each block contains 512 words
Address bits A10...A2 identify a word in a block

Bits A17 – A11
The block address within cache identified by address bits A17..A11
These bits called the index into cache
Correspond to the block’s address within main memory page
Any main memory address with address bits A17...A11
Having the values [000 0000]
Will be mapped into block 0 in cache

Bits A31 – A17
Identify which main memory page the block came from
Called the Tag
These values stored in data structure called Tag Table
Used when testing to see if needed word in cache
Current design will use 11 of these bits
Remainder will be 0

Examining the Tag Table and Tag

Tag Table
Data structure that contains information
Essential to the proper management of the cache
Contains one record for each block in cache
For the current design the tag table will contain 128 entries
Typical information contained in each record includes

*Tag*
Subset of bits from the main memory address
Identifying the page (in main memory) where the block originated

*Valid Bit*
Flag indicating if the corresponding block contains valid data
When an application starts
Cache contains no relevant information – all blocks empty
*valid bit* associated with each block set to FALSE
When a block is brought into the cache
Valid bit is tested
If FALSE
New block is copied to the target location
Valid bit is set
Else If the valid bit is TRUE
Block must be checked for changes

*Dirty Bit*
Flag indicating if the corresponding block contains data that has been modified
When a new block is first brought into cache
Copy in cache and the copy in main memory are identical or *coherent*
If a change is made to any piece of data in the cache
Two blocks are no longer the same
Two main schemes for addressing the issue

*Write through*
Propagates any data change immediately to main memory
Ensures two remain consistent

*Delayed write*
Assumes that if a piece of data changed once
May change again in the near future
Time can be saved by not performing multiple write operations
to the same data
To identify that the data has changed
*Dirty bit* is set to TRUE

When a new block is brought into memory and *valid bit* is TRUE, *dirty bit* associated with the block checked.

If bit is FALSE:
- New block overwrites the old
- Else old block must be copied back to main memory before the new one is brought in.

**Time**

Time of day information may be stored with a tag entry indicating:
- When block brought into the cache
- When it was last accessed

Information is used in other replacement algorithms.

Direct mapping scheme does not require it.

**How Does it Work**

**Finding a word**

To find word in a direct mapped cache execute simple process:

1. Check tag table for bits A31 – A18
2. If present:
   - Use bits A17 - A11 to index into cache
   - Use bits A10 - A2 to index into block
   - Use bits A1 - A0 for byte access
   - If WRITE operation:
     - Set dirty bit in tag table
     - Modify word
   - Else:
     - Return word
3. Else:
   - Get block from primary memory
   - If block occupied:
     - Check dirty bit
     - If set:
       - Write block to primary memory
       - Write block to cache
       - Set occupied bit

**Data or instruction**

As noted earlier:

Most contemporary computers use two caches:
- Data – dcache
- Instruction – icache

Same principles work for both
Only extra work
Deciding which cache to use

*Performance*
Factors to consider in each case
With and without cache
With cache
With and without miss
Optimizing size
Affect of look ahead

Direct mapped cache problems
What if regularly used items happen to map to the same cache line
Thrashing
Continually loading into cache but evicting it before reuse

Cache Miss Types
Several different types of misses
Compulsory/Coldstart
First access to a block
Basically unavoidable
For long-running programs
Is a small fraction of misses
Capacity
The block needed was in the cache
Unloaded because too many other accesses intervened
Solution
Increase cache size
Remember bigger is slower and more expensive
Conflict
The block needed was in the cache
Was enough room to hold it and all intervening accesses
Blocks mapped to the same location knocked it out
Solution
Cache size
Associativity

Caching – An Associative Mapping Cache Implementation
Although simple the direct mapped algorithm
Can have some negative affects on system performance
   Particularly if real-time constraints must be satisfied
Easy to imagine situation in which different portions of an algorithm
   Are in different block 0's in main memory
Under such a circumstance
   Two blocks would be repeatedly interchanged

**Associative Mapping**
   A ssociative approach lets a new block be placed anywhere in the cache
   An associative search then executed to locate it
   Such an algorithm searches by content rather than by address

   **Associative search asks**
   ‘given a target, is it in memory and if so, where?’
   **Traditional search says,**
   ‘here is an address, return the contents of that address

Under associative mapping algorithm
   Organization and mapping from main memory may now appear as illustrated

![Diagram of main memory and cache](image)

   New main memory block can be placed anywhere in the cache
   As with the direct mapping algorithm
   **Tag table entry corresponding to the cache block will**
   **Contain the address information**
   Identifying main memory origin
   Linear search of the tag table is not feasible

   **Cache miss using an associative mapping scheme**
   Similar to direct mapping scheme

   **Two differ in**
   How the new block is brought into the cache
Which block is to be replaced
Since new block not constrained to specific location
Replacement offers more opportunities
Any of three schemes are commonly used
Each has advantages and disadvantages
Choice for implementation often governed by
Requirements of the application

With an associative mapping algorithm
Time is added to tag table record
Three of the more common algorithms utilize temporal locality of reference as a metric

*Least Recently Used (LRU)*
*Most Recently Used (MRU)*
*Random*

LRU also called a FIFO
First-in-first-out scheme
MRU is referred to as LIFO
Last-in-first-out
Third algorithm selects and removes a block at random
Typically 12% worse than LRU

FIFO algorithm is based upon the assumption
Oldest block in the cache
Least likely to be used in future
Should be the one to be replaced
LIFO algorithm assumes
Newest block
Least likely to be used in the future
Since it was just used
Therefore that one that should be removed

Tag table in associative mapping scheme
Uses an associative memory
Illustrated in the high level block diagram

To find word associative mapped cache
Tag and block portions of the memory address
Specify the target for the associative search
All entries in the tag table are searched in parallel
If a match found
  Tag table location identifies block in cache
  Containing the target word
  That information
  Combined with remainder of the memory address
  To retrieve the needed word
The problems with fully associative cache
  Long search times
  Which can potentially slow down the application
  Complexity and cost of the underlying logic
Caching - Block Set Associative Mapping Cache Implementation

An approach combining direct and associative mapping

Called block-set associative

Combines

Some of simplicity of direct mapping algorithm
Some of flexibility of associative algorithm

Under block-set associative scheme

Entry at a specific index

Expanded from a single block to multiple blocks

Such a collection is called a set

Number of blocks in each set

Determined by the specific implementation.

We will implement 2-way set associative scheme

Each set will have two blocks

Similarly a 4-way implementation

Would support sets containing four blocks

Cache for a 2-way implementation takes on form

The Design

Main memory address space

First organized as collection of \( m \) blocks

Consider main memory with 16 blocks

The \( m \) blocks then organized as a collection of \( n \) groups

Group number for each block

Computed as

\[
\text{groupNumber} = m \mod n
\]

For

\( n = 2 \) two groups of 8 blocks each
\( n = 4 \) four groups of 4 blocks each

\textit{Set number} in the cache corresponds to the main memory \textit{group number}

Cache set \( \) main memory group

For

\( n = 2 \) two groups of 8 blocks each and two sets of 8 blocks each
\( n = 4 \) four groups of 4 blocks each and four sets of 4 blocks each
Any block from main memory group $j$
Can be placed into cache set $j$
Set is now searched associatively
Search is far less complex
Because dealing with much smaller search space

For current system

*Cache*

64 K with 128 0.5K blocks organized as 64 two block sets.

*Main Memory*

256 K words organized as 512 0.5K blocks

The resulting groups are given

<table>
<thead>
<tr>
<th>Block</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>63</td>
<td>127</td>
</tr>
</tbody>
</table>

Our addresses have the following association

We can now see how a main memory address are
Mapped to a cache address
Computation of address follows in same manner as Direct and associative mappings

Motorola (now Freescale) 68040
- Implements separate 4 K Data and instruction caches
- Each uses 4 way block set associative scheme
- Cache has 64 sets
  - Each set has 4 blocks
  - Each block has 4 long words
- Uses
  - 1 valid bit per block
  - 1 dirty bit per word

ARM 710T
- Each uses 4 way block set associative scheme
  - Single unified cache
    - Holds both data and instructions
  - Cache has 64 sets
    - Each set has 4 blocks
    - Each block has 16 bytes
      - Four 32 bit words
  - Write through protocol used
  - Uses random replacement algorithm

Intel Pentium III
- Uses two cache levels
  - Level 1
    - Implements separate 16 K byte
Data and instruction caches
Data cache
4 way block set associative scheme
Supports
Write through or write back protocol
Instruction cache
2 way block set associative scheme
No write policy
Level 2
Implements separate 256 K byte
Unified cache
8 way block set associative scheme
256 bit cache bus
To connect to bus interface unit
To interconnect
Caches
Main memory
I/O

Intel Pentium 4
Uses three cache levels
Level 1
Implements separate
Data and instruction caches
Data cache
8 K byte
4 way block set associative scheme
64 byte blocks
Supports
Write through protocol
Instruction cache
Holds predecoded instructions
Level 2
Implements separate 256 K byte
Unified cache
8 way block set associative scheme
128 byte blocks
Supports
Write back protocol
Level 3
Intended for servers
Performance Evaluation

For the moment when talking about performance
  Focusing on time measures

When we analyze and quantify performance of our system
  We’re interested in the best information we can get
  We’d like exact times if they’re computable
    Measurable and bounded times if exact numbers not available
To accomplish this we can use any of a variety of methods

We identified three techniques for acquiring performance information
  Analytical modeling
  Simulation
  Measurement

Key consideration in deciding which of these to use
  Stage in the development cycle that the system is in
  Measurement only possible if the system or one similar to proposed system already exists
  Otherwise analytic modeling or simulation
    Only alternatives
    A set of criteria for selecting an evaluation technique

Presented in the Table 1 below,

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Analytic Modeling</th>
<th>Simulation</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage</td>
<td>Any</td>
<td>Any</td>
<td>Post-prototype</td>
</tr>
<tr>
<td>Time Required</td>
<td>Small</td>
<td>Medium</td>
<td>Varies</td>
</tr>
<tr>
<td>Tools</td>
<td>Analysis</td>
<td>Computer Languages</td>
<td>Instrumentation</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Low</td>
<td>Moderate</td>
<td>Varies</td>
</tr>
<tr>
<td>Trade-off Evaluation</td>
<td>Easy</td>
<td>Moderate</td>
<td>Difficult</td>
</tr>
<tr>
<td>Cost</td>
<td>Small</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Scalability</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 1
Performance Evaluation Selection Criterion

Must remember that any performance measurement only valid
  When it’s considered in context of environment
  It’s the environment that establishes system workload
Performance analysis cannot be delayed until
Design completed and product ready to ship
At that time it’s too late and the cost of recovery may be too high.

Evaluating Performance
During the early stages of a design
Very often, we’re constrained to use modeling techniques
When we’re developing that model
We should consider the following.
  o The model should be hierarchical
    This way, we will be able to model a complex system
    By decomposing it into simpler parts
    Useful techniques include
      Progressive refinement
      Abstraction
      Reuse of existing (known) components
  o The model should be able to express
    Concurrent and temporal interdependencies
    Among physical and modeled elements
    With such ability
      We can begin to understand the dynamic performance
      The interactions amongst the constituent elements
  o Ideally, the model should be graphical
    Such a presentation makes interaction easier
    Not absolutely necessary however
  o The model should be should have parameters
    That permit worst case and scenario analysis
    Thus, we will be able to do boundary condition analysis
  o The model should support movement in time
    That is, we should be able to (repeatedly) verify
      Performance of system model forwards and backwards in time
During the mid-states of the design
We now are starting to get some real components
We can begin using prototype modules and integrating them into subsystems

During the later stages of the development
We can begin integrating the pieces
Into larger and larger portions of the system
At this time, we’re now prepared to begin
Exercising integrated subsystems

Performance Optimization
Let’s now look at a few ways we can begin to
Improve system performance

Briefly examine
Time
Power
memory

Trade-offs
Often times performance is optimization issue
Involves trading several contradictory requirements
Speed
Memory size
Cost
Weight
Power
Time must be spent up front to thoroughly understand
Application
Constraints

Considerations
Questions to ask
• What is being optimized,
  If we can’t answer this question, we need to re-think what we are doing.
• Why is it being optimized,
  What is the intended affect of the optimization.
• What will be affect on overall program if the module we are trying to optimize is
  eliminated from program,
  Presume a module under study has zero execution time
If the effect on the performance of the system is minimal, there’s no point in spending the effort.

- Is the optimization appropriate to operating context? Don’t optimize for floating point performance if only working with ints.

**Common Mistakes**

- Expect improvement in one aspect to improve performance proportional to improvement. See comment above. A 100% improvement in aspect affects 1% of performance.

- Hardware independent metrics predict performance. Example code size. Remember the difference between macros and subroutines.

- Comparing performance based upon a couple of metrics. Example clock rate, clock cycles per instruction, instruction count. Higher clock rate or more instructions does not guarantee better performance.

- Using peak performance as measure.

- Using synthetic benchmarks. Code can be optimized to excel on benchmark but not encountered in real world.

**Optimizing in Time**

At this point we’ll assume by performance, refer to response time and time loading. Response time is time from submittal to completion. Time loading is percent of time CPU or memory doing work for us.

**Reducing Response Times and Time-Loading**

1. Perform measurements and computations at rate and significance consistent with rate of change of data, values of data, type of arithmetic, number of significant digits calculated. Often interacting with external world.
Temperature typically very slowly changing entity
Measuring change at sampling interval greater than 1 second
Wasting CPU cycles

2. Use lookup tables or Combinational logic
Look up is faster than computing
Make a measurement, scale, convert
Arithmetic and shifting operations
Can be logical rather than arithmetic
Scaling a value by a constant
Logical operation
Multiplying two int
Combinational logic problem
Learn from the compiler guys
Many tricks commonly used by compiler writers
Reduce code size
Improve speed performance

Be careful also
Optimizing can cause problems
Example
Put value in register
Assume several instructions value will be
There and unchanged
No need to reload
Value may have been modified by some other routine
Shared variables
C++ has
Volatile and const volatile qualifiers

3. Recursion vs. Iteration
When to use recursion?
Processing recursive data structures
Divide and Conquer Algorithms:
1. Divide problem into subproblems
2. Solve each subproblem recursively
3. Combine subproblem solutions

When to use iteration instead?
Non-recursive data structures
Problems without obvious recursive structure
Functions with a large "footprint"
Especially when many iterations are needed

In Theory...
Any iteration can be rewritten using recursion, and vice-versa (at least in theory)
But the rewrite is not always simple
Iteration is generally
- More efficient
- Faster
- Takes less memory

A compromise:
If the problem is naturally recursive
Design the algorithm recursively first
Later convert to iteration if needed for efficiency

Tail Recursion
Suppose the last action of a function
Make a call to itself
In stack based implementation
Local variables pushed onto the stack
As recursive call is initiated
When recursive call terminates
Local variables will be popped off the stack
Thereby restored to former values
Doing this last step is pointless
Since recursive call is last operation of function
Just restored values are discarded

When last action of function is recursive call to itself
Not necessary to use the stack
Since no local variables need to be saved
Instead
Set dummy calling parameters to new values
Branch to beginning of the function

If last executed statement of a function is a recursive call to the function itself, the call can be eliminated by assigning the calling parameters to the values specified in the recursive call then repeating the whole function.
If a single recursive call is at the very end of the function:
Known as tail recursion

Easy for a smart compiler to automatically rewrite using iteration

4. Macros and Inlining Functions

As noted in discussion of recursion
Each function call requires that we build a stack frame
  Store state in original context
  Store arguments being passed
  Store local variables in new context
  Store return value
  Store return address
Such a process can be very time consuming

The C language
Supports construct called macros based upon the #define directive
Allows body of function to be placed directly in code
  Avoids cost of function call
That form of #define directive
  Declares a formal parameter list
Parameterized macro
Invoked
  Writing name
  Left parenthesis
  1 actual arg for each formal parameter
    Separated by commas
  Right parenthesis
If no formal parameters
  Must include empty arg list
White space may appear
  Between
  Name
  Left parenthesis
Example

```c
#define sum(x,y) ((x) + (y))

x = sum(2*a, b) / sum (c,d);
x = sum(2 * g(a,b), h(a,b)) / sum (c,d);
```

The #undef macro
- Companion to #define
- Used to make `name` no longer defined
- Causes preprocessor to forget macro definition of name
- Once `name` is undefined
  - Can be given new definition using #define

The C++ language
- Supports similar construct called inline
- Motivation the same
  - Avoid the cost of function call
    - Replacing function call with function body
- Like other optimization techniques
  - Always have tradeoff
    - Here trading off speed for increased memory size

5. Returns from a function call
- As noted
  - Function call involves creating stack frame
    - Hold passed and returned values
  - Can reduce overhead
    - Using shared variables in global space
  - Must ensure
    - Mutually exclusive access however

- Flow of control optimization
  - In branches or switches
    - Avoid repeated
Jumps or tests

```
je $2
$1:    ...
$2: jmp $3 ...

je $3
$1:    ...
$2: jmp $3 ...
```

C code fragment

```
switch (y)
{
    case 0: x = x+1;
    case 1: x = x+2;
    case 2: x = x+1;
    case 2: x = x+1;
}
```

```
switch (y)
{
    case 0:
    case 2: x = x+1;
    case 1: x = x+2;
    case 1: x = x+2;
}
```

May also be able to set x to value before switch
Change if necessary

```
while(1)  // recall the if is a single expression
    if (light == ON)
        light = OFF;
    else
        light = ON;
```

```
while(1)
    light = ~light;
```
7. Use registers and caches

Languages like C and C++
Support register type variables
Usually advantageous to utilize such types
Register operations faster than memory operations
When working with C or C++
Register qualification on variable declaration
Requests compiler put variable into register
No guarantee compiler will comply
Can force by writing code in assembler

Some processors support caching
Use caching to store frequently used variables
More rapid access than general purpose memory

8. Loop management

Therefore make ideal target for optimizing
Easy to instrument
During analysis can determine which might be candidates for optimization
Let’s look at several techniques that might be used

Loop invariant optimization
Precalculate any values that will not change within
Block of repeated code
Some good compilers do this for you
Use precomputed value rather than recomputing each time
Can be particularly significant if
Operand require indirect memory access for example
Working with several arrays
Indices differ by integer value

Loops and Arrays
Arrays are commonly used data structure
Simple modification in how accessed
Can have large impact on performance

Consider following code fragment

Observe
Test parameter computed

```c
for(j = 0; j < x + 3 ; j++)
{
    a[j] = b[j] + c[j];
}
```
With each iteration of loop
For large value of x
Can accumulate substantial time
A good compiler should spot and optimize
Computed value in test parameter
Without relying on compiler
Move computation of context of loop

\[
\begin{align*}
\text{int tempVar} &= x + 3; \\
\text{for}(j = 0; j < \text{tempVar}; j++) \\
&\{ \\
&\quad a[j] = b[j] + c[j]; \\
&\}
\end{align*}
\]

Now value only computed single time

**Nested Loops**
Let's modify the above example into

\[
\begin{align*}
\text{for}(i = 0; i < k; i++) \\
&\{ \\
&\quad \text{for}(j = 0; i < m; j++) \\
&\quad \quad \{ \\
&\quad \quad \quad a[i][j] = b[i][j] + c[i][j]; \\
&\quad \quad \} \\
&\}
\end{align*}
\]

Nested loop on several multidimensional arrays

We can easily rewrite the code using pointers to simplify the code

\[
\begin{align*}
\text{int offset} &= 0; \\
\text{for}(i = 0; i < k; i++) \\
&\{ \\
&\quad \text{for}(j = 0; i < m; j++) \\
&\quad \quad \{ \\
&\quad \quad \quad *(aPtr + \text{offset}) = *(bPtr + \text{offset}) + *(cPtr + \text{offset}); \\
&\quad \quad \quad \text{offset}++; \\
&\quad \quad \} \\
&\}
\end{align*}
\]

The code fragment assumes
Starting at index 0 for all arrays
If such is not the case
Must modify code as follows

```
int offset = 0;
for(i = 0; i < k ; i++)
{
    for(j = 0; i < m ; i++)
    {
        offset = l * m + j;
        *(aPtr + offset) = *(bPtr + offset) + *(cPtr + offset);
    }
}
```

**Unroll Loops**
Consider the following simple code fragment

```
for(j = 0; j < 4; j++)
a[j] = a[j]*8;
```

We can unroll this several ways
Case 1

```
a[0] = a[0]*8;
a[1] = a[1]*8;
a[2] = a[2]*8;
a[3] = a[3]*8;
```

Case 2

```
for(j = 0; j < 2; j++)
{
    a[j] = a[j]*8;
a[j+1] = a[j+1]*8;
}
```

9. Use only necessary values
The X Windows mouse
Dragging a graphic wire frame
Slewing

10. Understand the algorithms you’re using
   Searching and sorting
   Can be very time intensive
   Hash tables
   Can be very effective in reducing
   Time to store and time to look up

11. Optimize common path or frequently used code block
   Most frequently used path or highly used code segment
   Should be most highly optimized

12. Use page mode accesses
   For main memory accesses
   Set row address
   Modify only column address
   In essence moving pages / blocks of data

*Hardware Accelerators*
   One technique that can be used to gain significant temporal performance increase
   Over software implementation
   Move some of functionality to hardware
   Termed *hardware accelerators*
   Accelerator often attached to CPU bus

Communication with CPU
   Accomplished through many of same techniques already discussed
   • Shared variables
     Implemented as data and control registers
     Located in accelerator
   • Shared memory locations
     May use DMA

   Using shared locations comes with all problems
   Previously discussed
   Must manage the shared variables
   Semaphores
   Monitors
Distinguish between accelerator and co-processor

Accelerator
- Does not execute instructions
- Interface appears as I/O
- Designed to perform specific function
- Implemented as
  - ASIC, FPGA, CPLD

Co-processor
- Integrated with CPU
- Executes set of instructions

When to use
- When there are functions whose operations do not map well onto CPU
  - Bit and bit field operations can be difficult
  - Differing precisions of arithmetic calculations
  - Very high speed arithmetic
    - FFT calculations
    - Multiplies
  - Very high speed search
    - Associative
- High demand input or output operations
- Tight timing constraints
- High throughput
- Streaming applications
  - High speed audio and video
    - Delays in time domain translate to distortion in frequency

Optimizing Memory Management

Introduction
- Memory - the place where instructions and data are stored
- Has a significant impact on the behaviour of our systems

An improperly designed memory and access scheme
- Can determine if a design succeeds or fails

At the high level we are concerned about the affects of memory on
- The time our program takes to execute
- The amount of memory our program needs
When we design our systems
We must consider and manage both of these

In most embedded applications
Time is a critical parameter

Focusing on time and runtime specifically
Our memory management is concerned with the following
Managing process stack(s)
Allocation of memory
Static
Partition of memory
Code
Data
System
Dynamic
Allocation of memory resources for processes

When managing memory in embedded systems
Major concerns
Avoid dangerous allocation
Result in loss of deterministic behavior
Creation of deadlock situations
Minimize or reduce overhead during memory allocation

Let’s look at one aspect of the problem

Caches and Performance
Based upon locality of reference characteristic of most contemporary programs
We can use small amounts of high speed memory
To hold subset of instructions and data
For immediate use
Give illusion
Program has unlimited amounts of high speed memory
In fact bulk of instructions and data
Held in memory with much longer cycle / access times
Than available in System CPU

Problem in real time embedded applications
Cache behaviour is non deterministic
Difficult to predict when will have
  Cache miss or hit
Consequence difficult to set
  Reasonable upper bounds on execution times for tasks
  In extreme cases
    Can assume every access is a miss
      Is overly pessimistic
    Can assume every access is a hit
      Is overly optimistic

Problem rooted in two sources
  • Conditional branches
    Although number of good branch prediction algorithms
      Cannot know for certain in advance
        Which branch will be taken
    Becomes important because
      One path may cause hit and the other a miss
    Paths and successful access may vary with iteration
    Problem exacerbated with pipelined architectures
      Pipelining used to prefetch data and instructions
        While other activities taking place
      Selection of alternate branch
        Requires pipe to be flushed and refilled
          Has potential for cache miss
            Thereby extending time delay

  • Preemption and Shared Access
    In multitasking or interrupt context
      One task may preempt another
    In such a context
      Preempting task may require
        Different blocks of data or instruction

Consequence
  May get significant number of cache misses
    As tasks switch

Similar situation arises during instruction cycle
  In von Neumann machine
    In such a machine instructions and data
Let's elaborate on the problem of shared access
Consider a direct mapping caching scheme
Recall that blocks or lines from main memory are mapped into cache
Modulo the cache size

If we have a 1K cache with blocks of 64 words
64 word blocks from main memory addresses 0, 1024, 2048, etc
All going to map to block 0 in cache

Assume the following memory map
Instructions loaded starting at location 1024
Data loaded starting at location 8192

Consider the following simple code fragment

```c
for (i = 0; i < 10; i++)
{
    a[i] = b[i] + 4;
}
```

On first access
Instruction access will miss and
Bring in appropriate block from main memory
Instruction will execute and have to bring in data
Data will miss and
Bring in appropriate block from main memory
Because block 0 is occupied
Data block will overwrite

Second access
Instruction access will again miss and
Bring in appropriate block from main memory again
Miss occurs because instructions had been overwritten by incoming data

Instruction will execute and have to bring in data again
Data will also miss again
Bring in appropriate block from main memory again
Because block 0 is again occupied
Data block will overwrite again
Process will repeat causing serious performance degradation
Performance is actually worse with cache
Not only have the main memory accesses
Also have time to search and manage cache

Possible solutions to shared access
1. Set associative rather than direct mapping scheme
   Can help to mitigate some of effects of direct mapping scheme

2. Move to Harvard or Aiken architecture
   Support instruction cache and data cache
   Advantage of such an approach
   Can support multiple accesses per clock cycle
   Two caches can be designed differently
   To different design parameters
   Via different architectures
   Direct
   Set Associative

One scheme may use to address preemption problem
Give each task
   Its own portion of cache
Scheme called Smart Memory Allocation for Real Time – SMART cache
   Designed as another approach
   Cache decomposed into
   Restricted portions
   Common portion
Critical task
   Assigned restricted portion(s) on start-up
   All cache accesses
   Restricted to those partitions and common area
   Task retains exclusive rights
   To restricted areas
   Until terminates or aborted
   Include preemption by other tasks
Method for assigning partitions
   Remains open problem
Various heuristic schemes utilized

**Optimizing for Power Consumption**

Today more and more embedded applications targeted towards

- Small hand held or other types of portable devices
- Common thread through all such applications
  - Need to have long battery life
  - Translates to low power consumption

Power consumption can be attacked in several ways

- Certainly hardware solution
  - Low power devices
  - Turning portions of system off
    - ACPI - Advanced Configuration and Power Interface
  - Surprisingly have software contribution as well

Let’s look at each and begin with a view into the software

**Software**

There are a number of places that we can attack

- From software point of view

Initial places to look

- The algorithms that we use
- Location of code
  - Memory accesses can have significant impact on power
- Using software to control subsystems

As we have been stating

To analyze then control particular aspect of performance

- Must be able to measure that aspect
  - Both before and after modification

**Measuring Power Consumption**

For the moment

Will assume goal is to reduce power consumed by processor

To such an end

- Measuring power consumption is two step process
  - 1. Identify the portion of code to be analyzed
  - Typically this will be a loop
Doesn’t need to be 
Measure the current consumed by the processor 
While the code is being exercised

2. Modify the loop such that the code comprising the loop is disabled 
   Ensure that the compiler hasn’t optimized loop out 
   Measure the current consumed by the processor 

Once we have identified power consumed 
Next step is to try to reduce if appropriate 

Studies have identified several software factors 
That contribute to processor power consumption 
Among the contributors we find 
• The kind of instruction 
• The collection or sequence of instructions executed 
• The locations of the instructions and their operands 

Memory system and transfers in and out 
Have been shown to be most expensive operation 
Performed by processor 
Here memory is referring to main memory not cache 
This is the DRAM in our system 
Using simple addition operation as reference we find

<table>
<thead>
<tr>
<th>Operation</th>
<th>Relative Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 Bit Add</td>
<td>1</td>
</tr>
<tr>
<td>16 Bit Multiply</td>
<td>3.6</td>
</tr>
<tr>
<td>8x128x16 SRAM Read</td>
<td>4.4</td>
</tr>
<tr>
<td>8x128x16 SRAM Write</td>
<td>9</td>
</tr>
<tr>
<td>I/O Access</td>
<td>10</td>
</tr>
<tr>
<td>16 Bit DRAM Memory Transfer</td>
<td>33</td>
</tr>
</tbody>
</table>

Evident from table 
Using cache can have significant affect on power consumption 
Assumes a cache hit 
Cache miss requires main memory access 
SRAM generally consumes more power than DRAM 
On per cell basis 
Cache is generally SRAM 
Want to optimize size of cache \
Want smallest that provides desired performance
Almost becomes empirical trade-off

Other optimizations
1. Power aware compilers
   Take an instruction level view of problem
   Modify schedule of bus activity

2. Use registers efficiently
   Bring value into register and leave there for reuse

3. Look for cache conflicts and eliminate if possible
   For instruction conflicts
   Rewrite if possible
   May have to move code
   Scalar data conflicts
   Move data to different locations
   Arrayed data
   Move to alternate location
   Change access pattern

4. Unroll loops
   Must be careful that unrolled loop doesn’t result in cache misses

5. Eliminate recursive procedures where possible
   Eliminates overhead of function call

Hardware
Another technique for managing the power consumption
In embedded applications
Draws from familiar schemes used at home
Turn off the portions of system not being used
Such a scheme has been used for years
In space program
Satellites
   Orbital and interplanetary
Shuttle
Mercury, Gemini, Apollo
Therein hardware
Battery powered
Must be recharged
Done via solar panels of one form or another
Today can fly from Seattle to Japan in 14 hours
Laptop computer or other such tools
Typical battery life 3-5 hour
Yep a laptop is still an embedded application
We are in continuous race between
Battery technology
Demand for more and more powerful features
All such features require power

Power Management Schemes
To begin to address problem
As part of design
Formulate power management strategy
On one extreme
Turn power off
In such state
Power consumption limited to leakage
As with other metrics
Sets a lower bound on consumption

Opposite extreme
Power to all parts of system on and all parts operating
In such state
Power consumption approaches maximum
Such condition sets upper bound
Softer than lower bound
See earlier discussion on software affects

Goal is somewhere in middle
Governed once again by requirements specification
Based upon such a goal
We segregate system components into two categories
Those that must remain powered up
Referred to as static components
Those that may be powered down
Referred to as dynamic components

Such a scheme sounds simple ... and is at the high level
Like everything else we are doing
We have certain trade-offs

We must
1. Decide which portions of the system to power down
   These may be
   - All dynamic components
   - Subgroups based upon need or not need
2. Recognize that components cannot be shut down instantly
3. Recognize that components cannot be powered up instantly

These factors can be expressed with a simple first cut graphically as

Let’s consider a topographic mapping satellite application
As satellite is circling the earth collecting data
   Data is sent to ground station at known points in orbit
       When over appropriate station
No reason to keep transmitter powered up
       When not in position to transmit
Further timing of orbit known with sufficient resolution
       Know in advance when will need to transmit

After passing ground station
   Shut down transmitter
   Re-enable shortly before reaching next download point
Locations of each ground station known in advance

Such fixed schedule scheme is among simplest
   Can be very effective
   Observe similar to
       Round robin schedule with no preemption
Next level of sophistication
   Recognize that schedule may not be fixed
   Problem now moves from deterministic to probabilistic
Use knowledge of
   - Current history
   - Understanding of problem
To anticipate when to shut dynamic portions of system down
   Denoted predictive shutdown
Observe that such a scheme commonly used in
   Branch prediction logic in instruction prefetch pipeline
Using such a scheme
Subject to shutdown or restart too early

Related idea
- Rather than set schedule
  - Control algorithm with associated timer
  - Monitors activities of devices to be dynamically controlled
- If timer expires
  - Device is powered down
- Device reactivated on demand
- We’ve already used such a scheme in a watchdog timer

Next level of sophistication
- Draws from basic queuing theory
- Under such a scheme we have
  - A resource or producer
    - Service provided by system whose power is being controlled
  - A consumer
    - Portion of the system that needs the service
  - A queue of service requests
  - A power manager
    - Monitors behaviour of system
      - Producer
      - Consumer
      - Queue
  - Can build schedule using Markov modeling
    - That maximizes
      - System computational performance
      - While satisfying specified power budget

Simple data / control flow diagram appears as
Let’s look at an example of simple power management
Operating system responsible for dynamically controlling power
In simple I/O subsystem
Dynamically controlled portion
Supports two modes
Off and On
Dynamic subcomponents
Consume 10 watts when on and 0 watts when off
Switching
2 seconds and 40 joules
Switch from OFF state to ON state
1 second and 10 joules
Switch from ON to OFF
Request has period of 25 seconds

Graphically we illustrate 3 alternative schemes

<table>
<thead>
<tr>
<th>Policy</th>
<th>Energy Consumed in 25 sec</th>
<th>Average Latency Per Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always ON</td>
<td>250 J (10 x 25 sec)</td>
<td>1 sec</td>
</tr>
<tr>
<td>Reactive Greedy</td>
<td>240 J</td>
<td>3 sec</td>
</tr>
<tr>
<td>Power Aware</td>
<td>140 J</td>
<td>2.5 sec</td>
</tr>
</tbody>
</table>

Observe that we have the same average throughput
Substantially reduced power consumption

Advanced Configuration and Power Interface – ACPI
Industry standard power management
Initial application was to PC
More specifically windows
Currently targeted to wider variety of operating systems

Standard
Provides some basic power management facilities
Provides interface to the hardware
Software more specifically the operating system on a system
Provides power management module
It is the responsibility of the OS
Specifying the power management policy for the system

The operating system uses the ACPI module
To send required controls to the hardware
Monitor the state of the hardware
As input to the power manager
We express the scheme in following block diagram

Standard supports five global power states

1. G0 - working state in which system is fully usable
2. G1 - sleeping state
   System appears to be off
Time required to return to working condition
Inversely proportional to power consumption

Substates
S1 - low wake-up latency
Ensures no loss of system context
S2 - low wake-up latency state
Has loss of CPU and system cache state
S3 - low wake-up latency state
All system state except for main memory is lost
S4 - lowest power sleeping state
All devices are off

3. G2 – soft off requires full OS reboot to restore system to full operational condition
4. G3 – hard off or full off
   Defined as physically off state
   System consumes no power
5. Legacy state
   System does not comply with ACPI