System Optimization and Performance Analysis

Introduction

Is performance important?
Yes! Faster, smaller, cheaper, lower power, is better
Assuming we meet all the specified constraints
Very important in embedded systems
Often we have to trade-off speed and memory size
To be able to do so, we need concrete numbers
Such techniques become more important with larger systems
We use a different approach when dealing with small systems

Wait – what is performance?
As we progress we’ll see

Must keep things in perspective

If we let our first thoughts focus on software
Remember for any algorithm or method
Has basic housekeeping
Constituent steps
Choice of language or processor can have ± 100% affect
Therefore at first cut not really significant
These are details
Can usually wait for faster machine and problem goes away
Choice of fundamental algorithm
Can make vital difference

Other considerations
- Development time
- Ease of maintenance
- Extensibility

Approaches to improving software performance
- Tighter code
- 'Better' algorithms
- 'Better' data structures

There’s much more

An embedded system
Comprises collection of hardware and software components
Such a set intended to provide desired
 Behaviour
 Service

(Embedded) system design
 Process of implementing desired behaviour or services
 With such an implementation we want to
 Optimize certain aspects of behaviour or services
 Subject to set of constraints

Optimization and performance of a system
 Mean many things to many people
 While certainly can say inherently good
  What do they mean?
  What are we optimizing?
  What kind of performance are we examining?
  What is performance?

First steps at understanding
 With any design
  Many aspects which we can optimize
  Many different measures of performance

**Performance or Efficiency Measures**

Overview
 Performance or efficiency
  Usually means "time" (to run) or "space" (memory used)
 Simply put performance means meeting the specification
  To meet specification one must have specification
 Task becomes one of
  Identifying level at which performance is to be measured
  Identifying meaningful parameters at that level
  Selecting reasonable and proper values
  As designers this is our job

How to measure efficiency
 - Run the program
 - See how long it takes
 - See how much memory it uses
 - Tools are available
Lots of variability when running the program
- What input data?
- What hardware platform?
- What compiler?
- What compiler options?

Just because one program is faster than another right now
Will it always be faster?

Let’s begin to quantify
Several major areas can begin to focus on
✓ Complexity
✓ Time
✓ Power consumption
✓ Memory management and memory size
✓ Cost
✓ Weight

We will examine several of these

For each such measure
Must consider
- Best or min case
  When referring to time important in many scheduling algorithms
  When referring to cost
    Becomes value below which cannot remove any more parts
  Similar argument for power
- Average case
  Gives typical measure
  Often sufficient
- Worst case
  Largest or longest value of a particular measure
  When referring to time
    Sets an upper bound on a schedule

Typical embedded system
Comprises root hardware platform
May include number of peripheral devices as well
Goal of hardware is to implement specified behaviour or service(s)
From embedded point of view

- We consider **hardware** to comprise
  - Computational elements
  - Communication subsystem
  - Memory
  - Power comes in as adjunct to all

- We consider **software (firmware)** to be
  - Algorithms and Data Structures
  - Control and Scheduling

To optimize performance of combined system

Must consider performance of each hardware and software constituent

At the end of the day

Simply put performance means meeting the specification

To meet specification one must have specification

When we optimize one aspect of the system or another

Again we are optimizing against a specification

Based upon our specification task becomes one of

- Identifying level at which performance is to be measured
- Identifying meaningful parameters at that level
- Selecting reasonable and proper values

As designers this is our job

**Limitations**

We want to look at each of these aspects of performance in turn

Certainly our goal is to ‘improve’ performance

Before we attack the problem of performance improvement

Let’s step back and quantify limits of what can be done

From Amdahl’s Law we write

\[
\frac{T_{\text{total}}}{T_{\text{improved}}} = \frac{T_{\text{total}}}{(T_{\text{total}} - T_{\text{component}}) + \frac{T_{\text{component}}}{n}}
\]

- \(T_{\text{total}}\) – System metric prior to improvement
- \(T_{\text{improved}}\) – System metric after improvement
- \(T_{\text{component}}\) – Contribution of the component to be improved to the system metric
- \(n\) – the amount of the improvement
**Example**

Consider a system with the following characteristics
- Task to improve executes in 100 time units
- Desire 80 time units
- Algorithm to be improved used 40 time units

\[
\frac{100}{80} = \frac{100}{(100 - 40) + \frac{40}{n}}
\]

Simplifying gives a value of 2 for n
- The algorithm speed will have to be improved to 20 time units

**Example**

Consider a system with the following characteristics
- Task to improve executes in 100 time units
- Desire 50 time units
- Algorithm to be improved used 40 time units

\[
\frac{100}{50} = \frac{100}{(100 - 40) + \frac{40}{n}}
\]

Simplifying gives a value of -4 for n
- The algorithm speed will have to run in negative time to meet the new spec
- Clearly impossible

With such restrictions in mind
- Let’s now look at the various performance measures
- We’ll begin on the software side and at a high level
- Let’s look at assessing the complexity of our
  - Algorithms
  - Programs

**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 affect 1% of performance
    Minimal
- Hardware independent metrics predict performance
  Example code size
  Remember the difference between macros and subroutines
- Comparing performance based upon 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
  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
  Time from submittal to completion
  Time loading
  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
   #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;
}

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

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

```c
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

```c
for(j = 0; j < x + 3 ; j++)
{
    a[j] = b[j] + c[j];
}
```

Observe
Test parameter computed
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

```c
int tempVar = x + 3;
for(j = 0; j < tempVar ; j++)
{
    a[j] = b[j] + c[j];
}
```

Now value only computed single time

*Nested Loops*
Let’s modify the above example into
Nested loop on several multidimensional arrays
We can easily rewrite the code using pointers to simplify the code:

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

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

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

*Unroll Loops*
Consider the following simple code fragment:

```c
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 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

**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
Share same physical memory
May force cache misses based upon
Instruction and data fetch

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 over write

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 over write 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