SCALABLE COMPUTING LABORATORY

VANDERBILT UNIVERSITY

SC3260 / SC5260

Introduction to HPC

Lecture by: Ana Gainaru

Course Organization

Two lecturers

- **Ana Gainaru**
- **Hongyang Sun**

Plan (tentative) → 24 lectures, 1 midterm and 3 classes for project presentation

- Overview of HPC
- Introduction to parallel computing
- Introduction to bash scripting
- Performance analysis
- Shared memory
- GPU
- Interconnect and communication patterns
- Distributed memory
- Scheduling
- Fault tolerance

Resources at: https://github.com/vanderbiltscl/SC3260_HPC

What to expect

- The course material is dense. We are giving only a broad overview of the field.
- The slides and additional material are sufficient to pass the course. We will recommend research papers and books for those who want a more advanced knowledge of HPC.
- Some classes will be very practical, some will present basic and advanced theoretical concepts of HPC. Try to look back and get the global view.
- We expect at the end of the course students to be able to submit jobs and to understand internals on HPC architectures and applications.

Ask questions !

We encourage you to ask questions about previous classes at the beginning of each class.

Grading

Three evaluation steps:

- Homeworks. There will be 3 homeworks:
  - Shared memory (Cilk programming language) February 14th
  - Shared memory (GPU programming) March 13th
  - Distributed memory (MPI programming) March 27th
- Midterm
  - Will include all the material up to spring break March 11th
- Semester long project.
  - Teams of 2-3 students
  - Presentations will be held on the last 2 classes April 15th and 20th
- [For graduate students only] Research paper presentations
  Scattered during the Scheduling Fault Tolerance lectures (April 6/8/13th)

Resources

All the lecture slides and research articles are available on the github page.
https://github.com/vanderbiltscl/SC3260_HPC

- If you have questions:

mailto: *ana.gainaru@vanderbilt.edu, hongyang.sun@vanderbilt.edu*

- Prerequisites:

Basic algorithmic techniques (dynamic programming, greedy algorithms)
Knowledge of C/C++

Advertisement

We are working in the broad field of HPC (application and middleware levels).

- If you are interested in doing research related to any of the concepts presented in this course
  - Research assistanships or summer intern positions
  - (either in our team or we can help choose other locations)

Take a look at our research topics: http://scl.vuse.vanderbilt.edu

Why this course?

- Almost all computers today use some form of parallelism
- Designing algorithms and programs by taking parallelism into account from the start is generally more efficient
- High performance in many applications is critical: we want to use our hardware as efficiently as possible

Parallel computing

What is parallel computing to you?

Parallel computing

What is parallel computing to you?

Three critical notions: application, machine, objective function.

The software is a solution for connecting “applications” and “machines” while trying to optimize an "objective function".

Course Topics

- Aspects of HPC architecture and networks
- Parallel algorithms
  - How to partition a problem for parallel computing
- Batch scripting and job submission
- Performance modeling
  - Computation and communication
- Parallel scientific applications
  - Molecular simulations, quantum chemistry
- Distributed memory (MPI) programming
- Multithreaded programming (Cilk, OpenMP, etc.)
- Coprocessor/accelerator programming

Sequential programming

Traditionally, programs were written for sequential computation.

- A problem is divided into a set of separate instructions
- Instructions are executed sequentially one after another
- The execution is done on only one processing unit
- At every given time unit there is only one instruction being executed

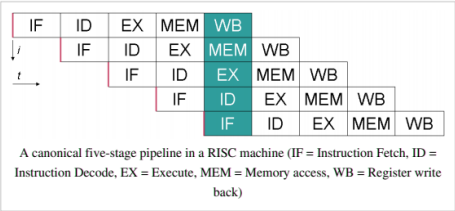
Is your laptop using parallel computing?

Sequential programming

Traditionally, programs were written for sequential computation.

- A problem is divided into a set of separate instructions
- Instructions are executed sequentially one after another
- The execution is done on only one processing unit
- At every given time unit there is only one instruction being executed

Is your laptop using parallel computing?
What about a machine with only one core?



A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

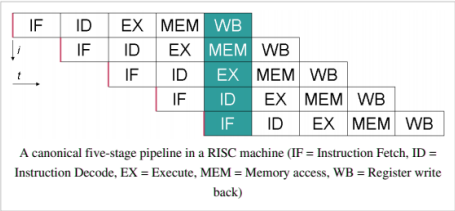
Source: http://www.cse.unt.edu/ tarau/teaching/parpro/papers/Parallel%20computing.pdf

Instruction-level parallelism from pipelining

- Some processors can issue more than one instruction at a time (superscalar processors)

VANDERBILT UNIVERSITY

Sequential programming



A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

Source: http://www.cse.unt.edu/ tarau/teaching/parpro/papers/Parallel%20computing.pdf

What about the order of execution?

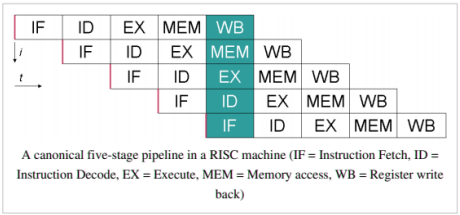
```
int a = compute_a();
int b = compute_b();

if(a > 4)
{
  c = a + 1;
}
```

Instruction-level parallelism from pipelining

- Some processors can issue more than one instruction at a time (superscalar processors)

VANDERBILT UNIVERSITY

Sequential programming



A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

Source: http://www.cse.unt.edu/ tarau/teaching/parpro/papers/Parallel%20computing.pdf

Instruction-level parallelism from pipelining

- Some processors can issue more than one instruction at a time (superscalar processors)

What about the order of execution?

```
int a = compute_a();
int b = compute_b();

if(a > 4)
{
  c = a + 1;
}
```

Reordering buffers inside each modern CPUs

VANDERBILT UNIVERSITY

Parallel programming

In the simplest form, parallel computing is represented by the simultaneous use of multiple resources when solving a problem

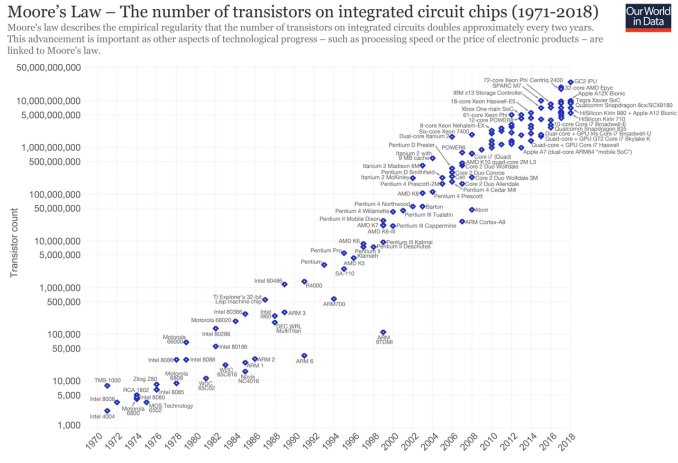
- The problem is divided into a set of sub-problems that can be resolved concurrently
- Each sub-problem is again divided into a set of separate instructions
- These instructions are executed on several processing units
- Usually there is some form of coordination / control system

1. **Applications can be divided in different ways**
2. **Parallel architectures can have different configurations**

Moore's Law

Exponential growth of computing power

- Has been true for the past 45+ years
- Many say Moore's law nears its physical limits



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

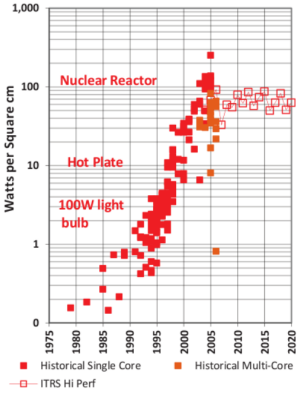
Licensed under CC-BY-SA by the author Max Roser.

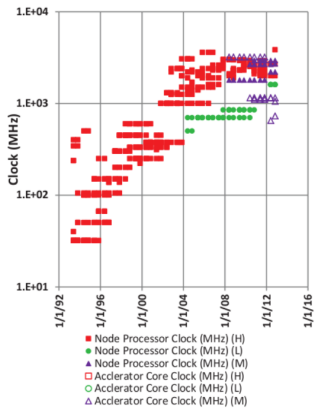
Source: Wikipedia

VANDERBILT UNIVERSITY

Moore's Law

We have a hard time powering these systems





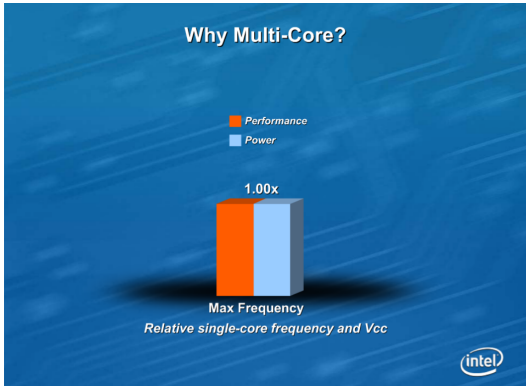
Source: Exascale computing trends, Kogge and Shalf, 2013

VANDERBILT UNIVERSITY

Single Core Limit

Limitations in increasing the frequency of single cores

- Transistor density increases power
- Increases number of bit flips



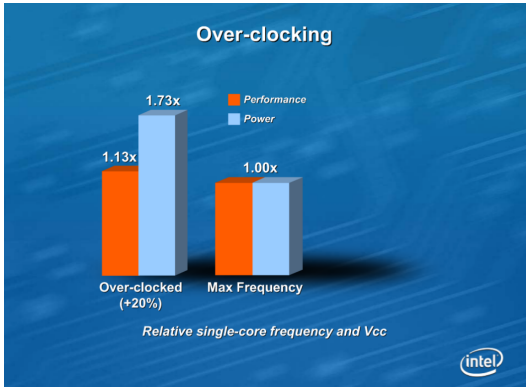
Source: Intel (Algorithmique du Parallélisme slides Guillaume Pallez)

VANDERBILT UNIVERSITY

Single Core Limit

Limitations in increasing the frequency of single cores

- Transistor density increases power
- Increases number of bit flips



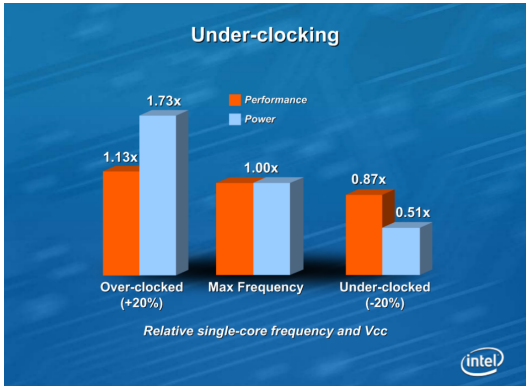
Source: Intel (Algorithmique du Parallélisme slides Guillaume Pallez)

VANDERBILT UNIVERSITY

Single Core Limit

Limitations in increasing the frequency of single cores

- Transistor density increases power
- Increases number of bit flips



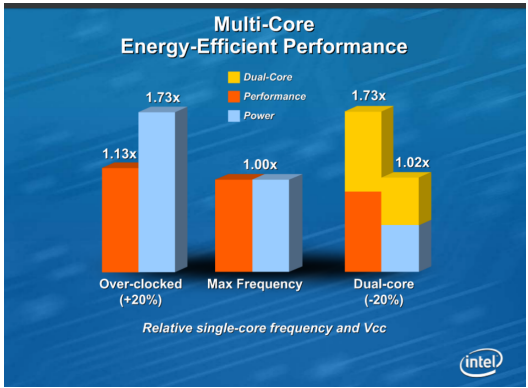
Source: Intel (Algorithmique du Parallélisme slides Guillaume Pallez)

VANDERBILT UNIVERSITY

Single Core Limit

Limitations in increasing the frequency of single cores

- Transistor density increases power
- Increases number of bit flips



Source: Intel (Algorithmique du Parallélisme slides Guillaume Pallez)

VANDERBILT UNIVERSITY

Overview

Uniprocessors and Compiler Optimizations

- Uniprocessors
  - Processor architectures
  - Instruction set architectures
  - Instruction scheduling and execution
- Data storage
  - Memory hierarchy
  - Caches, TLB
- Compiler optimizations

Multiprocessors and Parallel Programming Models

- Multiprocessors
  - Pipeline and vector machines
  - Shared memory
  - Distributed memory
  - Message passing
- Parallel programming models
  - Shared vs distributed memory
  - Hybrid
  - BSP

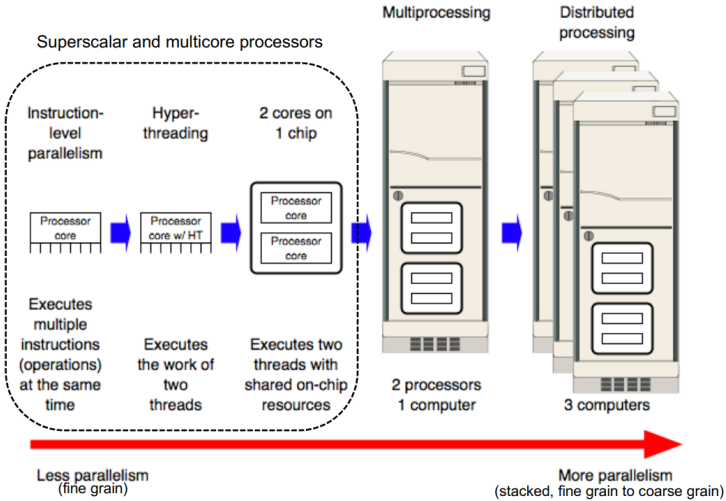
Slides based on

Software Optimization for High Performance Computing: Creating Faster Applications by K.R. Wadleigh and I.L. Crawford

ISC5318 course on High Performance Computing Spring 2017, Florida State University

VANDERBILT UNIVERSITY

Levels of Parallelism

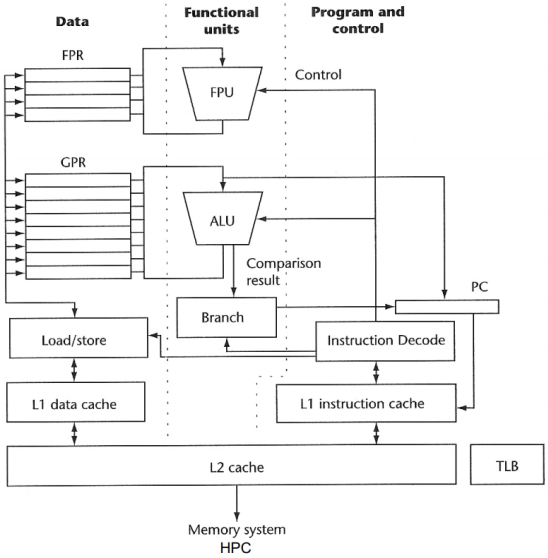


VANDERBILT UNIVERSITY

Processor Families

- **CISC** (complex instruction set computer)
  - Specialized instructions of various (and variable) length
  - Single instructions can execute several low-level operations (e.g. load from memory, an arithmetic operation, and a memory store)
- **RISC** (reduced instruction set computer)
  - No microcode and relatively few instructions of the same word length (only load and store instructions access memory)
  - Many RISC processors are superscalar (for instruction-level parallelism)
  - More registers than CISC
- **VLIW** (very long instruction word)
  - Bundles multiple instructions for parallel execution
  - Dependences between instructions in a bundle are prohibited
  - More registers than CISC and RISC
- **Vector Machines** Single instructions operate on vectors of data

Superscalar Architecture



Instruction Pipeline

A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

- An instruction pipeline increases the instruction bandwidth

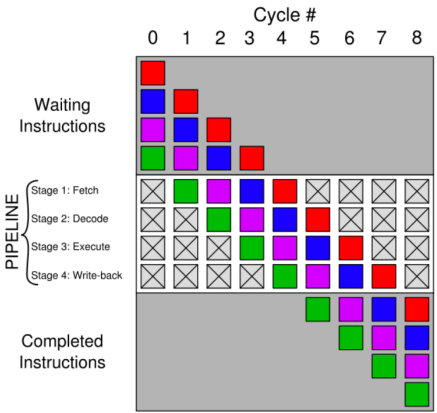
For full utilization

Five instructions are in the pipeline at different stages

WB write back to registers and forward results into the pipeline when needed by another instruction

VANDERBILT UNIVERSITY

Instruction Pipeline Example



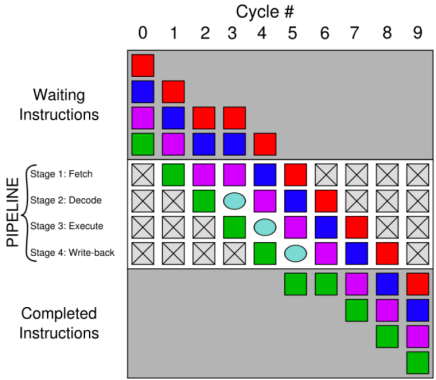
- Example 4-stage pipeline: IF, ID, EX, WB
- Four instructions (green, purple, blue, red) are processed in this order
  - Cycle 0: instructions are waiting
  - Cycle 1: Green is fetched
  - Cycle 2: Purple is fetched, green decoded
  - Cycle 3: Blue is fetched, purple decoded, green executed
  - Cycle 4: Red is fetched, blue decoded, purple executed, green in write-back

For full utilization

- The four instructions need to be independent
- Every stage needs to take one cycle

VANDERBILT UNIVERSITY

Instruction Pipeline Hazards



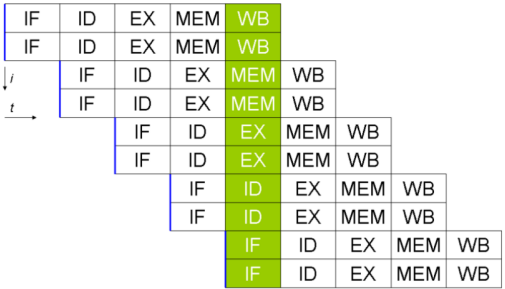
- **Pipeline hazards**
  - From data dependences
  - Forwarding in the WB stage can eliminate some data dependence hazards
  - From instruction fetch latencies (e.g. I-cache miss)
  - From memory load latencies (e.g. D-cache miss)
- A hazard is resolved by **stalling the pipeline**, which causes a bubble of one ore more cycles

Example One cycle stall in the IF stage of the purple instruction.

Cycle 3: Purple cannot be decoded
and a no-operation (NOP) is inserted

VANDERBILT UNIVERSITY

N-way Superscalar RISC



- Increases the instruction-level parallelism
  - RISC instructions have the same word size
  - N-way superscalar RISC processors fetch N instructions each cycle

Example:
2-way superscalar RISC pipeline parallelism

Increase pipeline hazards

VANDERBILT UNIVERSITY

Software Optimization to Increase CPU Throughput

Processors run at maximum speed (high instruction per cycle rate (IPC)) when

1. There is a good mix of instructions (with low latencies) to keep the functional units busy
2. Operands are available quickly from registers or D-cache
3. The FP to memory operation ratio is high (FP : MEM > 1)
4. Number of data dependences is low
5. Branches are easy to predict

Software Optimization to Increase CPU Throughput

Processors run at maximum speed (high instruction per cycle rate (IPC)) when

1. There is a good mix of instructions (with low latencies) to keep the functional units busy
2. Operands are available quickly from registers or D-cache
3. The FP to memory operation ratio is high (FP : MEM > 1)
4. Number of data dependences is low
5. Branches are easy to predict

Hardware

Out-of-order scheduling can improve #1

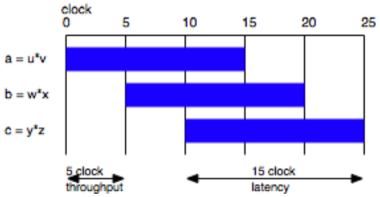
Hardware prefetching can improve #2

Compiler optimizations effectively target #1-3

The programmer can help improve #1-5

Instruction Latency and Throughput

```
a=u*v; b=w*x; c=y*z
```



- **Latency**
  - Time from the start of the instruction until the result is available
  - An instruction using a can start after 15 cycles
- **Throughput**
  - Time until you can start another operation of the same type
  - Another multiplication can start every 5 cycles

Example x64 processor

A single 32-bit **division** has a throughput of one instruction every six cycles with a latency of 26 cycles.

A **multiplication** has a throughput of one instruction every cycle and a latency of 3 cycles."

VANDERBILT UNIVERSITY

Instruction Latency Example

Euclid's algorithm

- Two versions of the implementation
  - Modulo version
  - Repetitive subtraction version

```
int find_gcf1(int a, int b)
{
   while (1)
   {
      a = a % b;
      if (a == 0) return b;
      if (a == 1) return 1;
      b = b % a;
      if (b == 0) return a;
      if (b == 1) return 1;
   }
}
```

Modulo version

```
int find_gcf2(int a, int b)
{
   while (1)
   {
         if (a > b)
               a = a - b;
         else if (a < b)
               b = b - a;
         else
           return a;
    }
}
```

Repetitive subtraction version

Which one is faster?

Instruction Latency Example

Modulo version

```
int find_gcf1(int a, int b)
{
   while (1)
   {
      a = a % b;
      if (a == 0) return b;
      if (a == 1) return 1;
      b = b % a;
      if (b == 0) return a;
      if (b == 1) return 1;
   }
}
```

Repetitive subtraction version

```
int find_gcf2(int a, int b)
{
   while (1)
   {
          if (a > b)
              a = a - b;
          else if (a < b)
              b = b - a;
          else
            return a;
    }
}
```

Instruction count for a=48 and b=40

Modulo 2
Compare 3
Branch 3

Total: 8

Subtract 5
Compare 5
Branch 14

Total: 24

Instruction Latency Example

Modulo version

```
int find_gcf1(int a, int b)
{
   while (1)
   {
      a = a % b;
      if (a == 0) return b;
      if (a == 1) return 1;
      b = b % a;
      if (b == 0) return a;
      if (b == 1) return 1;
   }
}
```

Repetitive subtraction version

```
int find_gcf2(int a, int b)
{
   while (1)
   {
          if (a > b)
                a = a - b;
          else if (a < b)
                b = b - a;
          else
            return a;
    }
}
```

Instruction count for a=48 and b=40

Latency of 30 for division, 1 for the rest

Modulo 2 - Cycles 60
Compare 3 - Cycles 3

Branch 3 - Cycles 3

Total: 8 - Cycles 66

Subtract 5 - Cycles 5
Compare 5 - Cycles 5

Branch 14 - Cycles 14

Total: 24 - Cycles 24

VANDERBILT UNIVERSITY

Instruction Latency Example

Modulo version

```
int find_gcf1(int a, int b)
{
   while (1)
   {
      a = a % b;
      if (a == 0) return b;
      if (a == 1) return 1;
      b = b % a;
      if (b == 0) return a;
      if (b == 1) return 1;
   }
}
```

Total instructions: 8 - Total cycles 66

Repetitive subtraction version

```
int find_gcf2(int a, int b)
{
   while (1)
   {
         if (a > b)
               a = a - b;
         else if (a < b)
               b = b - a;
         else
           return a;
    }
}
```

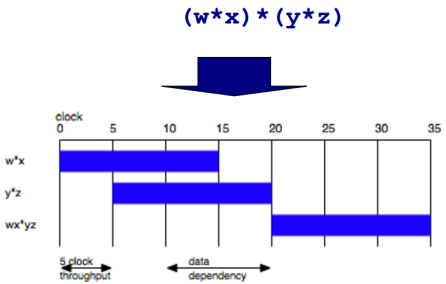
Total instructions: 24 - Total cycles 24

Execution time for all values of a and b in [1..9999]

Modulo version takes 18.55 sec
Repetitive subtraction version takes 14.56 sec

VANDERBILT UNIVERSITY

Data dependencies



- Instruction level parallelism is limited by data dependences
- Types of dependences:
  - RAW: read-after-write also called flow dependence
  - WAR: write-after-read also called anti dependence
  - WAW: write-after-write also called output dependence

What kind of dependency is in the example?

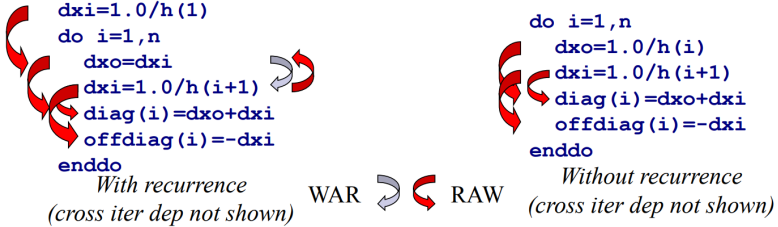
- WAR and WAW dependences exist because of storage location reuse
  - overwrite with new value
  - WAR and WAW are called false dependences
  - RAW is a true dependence

VANDERBILT UNIVERSITY

Data dependencies

Removing redundant operations by reusing space may increase the number of dependences.

- Example: two code versions to initialize a finite difference matrix
  - Recurrent version with lower FP operation count
  - Non-recurrent version with fewer dependences



Which is fastest depends on loop optimization and instruction scheduling by compiler (and processor)

VANDERBILT UNIVERSITY

Data dependency example

Running on Intel Core 2 Duo 2.33 GHz

```
dxi = 1.0/h[0];
for (i=1; i<n; i++)
{
   dxo = dxi;
   dxi = 1.0/h[i+1];
   diag[i] = dxo+dxi;
   offdiag[i] = -dxi;
}
```

`gcc -O3 fdinit.c time ./a.out`

Total execution time 0.135 sec

```
for (i=1; i<n; i++)
{
  dxo = 1.0/h[i];
  dxi = 1.0/h[i+1];
  diag[i] = dxo+dxi;
  offdiag[i] = -dxi;
}
```

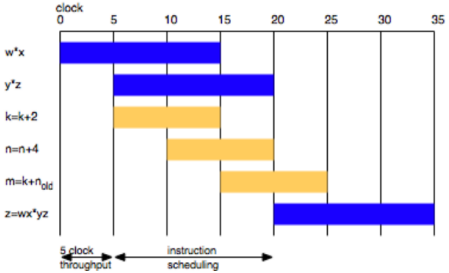
`gcc -O3 fdinit.c time ./a.out`

Total execution time 0.270 sec

- On the right there is one more memory load but fewer dependencies.
- On the left one cross-iteration RAW but fewer memory loads
- Overall the compiler/hardware could decrease the impact of the dependence
  - It spans multiple instructions

Instruction Scheduling

```
z = (w*x)*(y*z);
k = k+2;
m = k+n
n = n+4;
```



- Instruction scheduling can hide data dependence latencies
  - With **static scheduling** the compiler moves independent instructions up/down to fixed positions
  - With **dynamic scheduling** the processor executes instructions out of order when they are ready
- Example of one multiply and one add execution unit
  - **Most modern processors use register renaming to remove WAR dependences**
  - Note that line 3 and 4 can be executed out of order
    - m=k+n uses the old value of n

VANDERBILT UNIVERSITY

Hardware Register Renaming

Sequential

```
R0 = R1 + R2
R2 = R3 + R4
```

Parallel

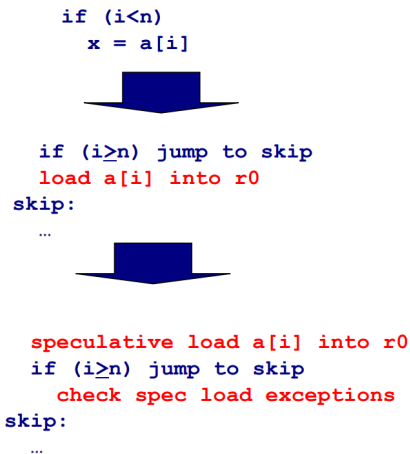
```
R0 = R1 + R2
R2'= R3 + R4
...
R2 = R2'
```

- Hardware register renaming can remove WAR dependences
  - Fixed number of registers are assigned by compiler to hold temporary values
- A processor's register file includes a set of hidden registers
  - A hidden register is used in place of an actual register to eliminate a WAR hazard
- The WB stage stores the result in the destination register

Pitfall

It **does not** help to manually remove WAR dependences by introducing extra scalar variables. The compiler's register allocator reuses registers assigned to these variables and thereby reintroduces the WAR dependences at the register level.

Control speculation



- Control speculation allows conditional instructions to be executed before the conditional branch in which the instruction occurs
  - Hide memory latencies
- A speculative load instruction performs a load operation
  - Enables loading early
- A check operation verifies that whether the load trigged an exception (e.g. bus error)

VANDERBILT UNIVERSITY

Hardware Branch Prediction

```
for (a=0; a<100; a++)
{
   if (a % 2 == 0)
      do_even();
   else
      do_odd();
}
```

Simple branch pattern `a % 2` is predicted correctly by processor after only a few iterations

```
for (a=0; a<100; a++)
{
   if (flip_coin() == HEADS)
      do_heads();
   else
      do_tails();
}
```

Complex patterns `flip_coin() == HEADS` are difficult to predict

Branch prediction enables a processor to pre-fetch instructions of a target branch

- Branch prediction uses a history mechanism per branch instruction by storing a compressed form of the past branching pattern
  - When predicted correctly there is no branch penalty
  - When not predicted correctly, the penalty is typically >10 cycles

VANDERBILT UNIVERSITY

Ways to Help the Branch Predictor

- In a complicated branch test in C/C++ move the simplest to predict condition to the front of the conjunction

```
if (i == 0 && a[i] > 0)
```

- Avoid branches if possible

`a = min(a, 255);` instead of `if (a >= 255) a = 255;`

- Rewrite conjunctions to logical expressions (not true all the time !!)

`if ( (t1|t2|t3) == 0 )` instead of `if (t1==0 && t2==0 && t3==0)`

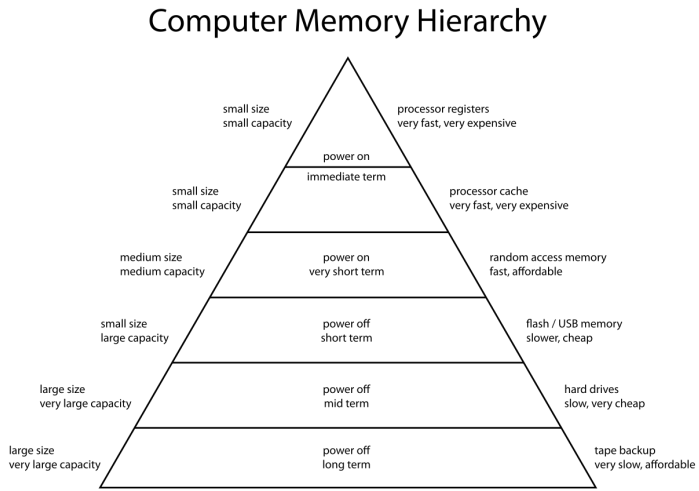
Note that cond?then:else and the && and || operators result in branches!

A lot of optimizations are already done by the compiler !
Optimize without reducing readability as much as possible

Data Storage

- Memory hierarchy
- Performance of storage
- CPU and memory
- Virtual memory, TLB, and paging
- Cache

Memory Hierarchy



Source: Wikipedia

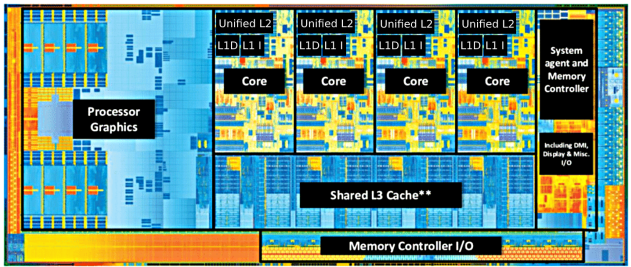
Storage systems are organized in a hierarchy

- Speed
- Cost
- Volatility

VANDERBILT UNIVERSITY

Memory Performance

- Registers are fast, typically one clock cycle to access data
- Cache access takes tens of cycles
- Memory access takes hundreds of cycles
- Movement between levels of storage hierarchy can be explicit or implicit



- Registers, latency 1 cycle
- L1 32 kB, latency 3 cycles
- L2 256 kB, latency 10 cycles
- L3 8 MB shared, latency 40 cycles
- DRAM, latency hundreds of cycles
- Disk, latency millions cycles

VANDERBILT UNIVERSITY

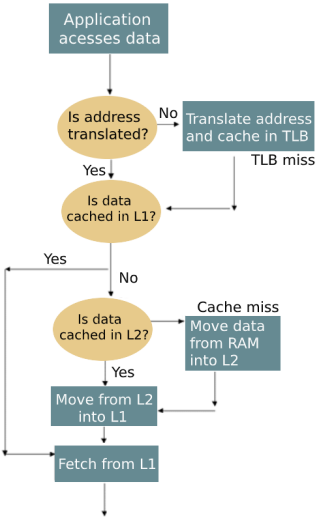
Memory Performance

- Registers are fast, typically one clock cycle to access data
- Cache access takes tens of cycles
- Memory access takes hundreds of cycles
- Movement between levels of storage hierarchy can be explicit or implicit

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

- Registers, latency 1 cycle
- L1 32 kB, latency 3 cycles
- L2 256 kB, latency 10 cycles
- L3 8 MB shared, latency 40 cycles
- DRAM, latency hundreds of cycles
- Disk, latency millions cycles

Memory Hierarchy



- A logical address is translated into a physical address
  - The translation lookaside buffer (TLB) is an on-chip address translation cache

Cache hit

Data is found in cache (**Hit rate:** fraction found in the cache; **Hit time:** time to access the cache)

Cache miss

Data is not found in cache and must be copied from a lower level (**Miss penalty** : time to replace a block from lower level)

- Access time : time to access lower level
- Transfer time : time to transfer block

Average memory-access time (AMAT)

= Hit time + Miss rate x Miss penalty (ns or clocks)

VANDERBILT UNIVERSITY

Example Cache Performance

- Instruction miss rate %2
- Data miss rate %4
- Ideal CPI (cycles per instruction) is 2
- Miss penalty 40 cycles
- %36 of instructions are load/store

How much faster would the system be if it had a perfect cache that never missed?

Instruction miss cycles = #I x 0.02 x 40 = 0.80 #I
Data miss cycles = #I x 0.36 x 0.04 x 40 = 0.58 #I
Total memory stall cycles = 0.80 #I + 0.58 I = 1.38 #I

$CPI_{real} = CPI_{ideal} + 1.38 = 2 + 1.38 = 3.38$

$$\frac{CPUPerformance_{real}}{CPUPerformance_{ideal}} = \frac{3.38}{2} = 1.69$$

Example Cache Performance

- Instruction miss rate %2
- Data miss rate %4
- Ideal CPI (cycles per instruction) is 2
- Miss penalty 40 cycles
- %36 of instructions are load/store

How much faster would the system be if it had a perfect cache that never missed?

Instruction miss cycles = #I x 0.02 x 40 = 0.80 #I
Data miss cycles = #I x 0.36 x 0.04 x 40 = 0.58 #I
Total memory stall cycles = 0.80 #I + 0.58 I = 1.38 #I

$CPI_{real} = CPI_{ideal} + 1.38 = 2 + 1.38 = 3.38$

$$\frac{CPUPerformance_{real}}{CPUPerformance_{ideal}} = \frac{3.38}{2} = 1.69$$

Ways of increasing the performance

- Decrease the miss rate
  - Hardware, compiler and software
- Decrease the penalty of a miss
  - Hardware
- Decrease the penalty of a hit
  - Hardware

Hardware optimization techniques

- **Decrease the miss rate**
  - Use larger blocks / larger cache
  - Use more associativity
  - Victim cache
  - Prefetch
- **Decrease the penalty of a miss**
  - Use smaller blocks
  - Multilevel Caches
  - Use a non-blocking cache
- **Decrease the penalty of a hit**
  - Give priority to read over write misses

Ways of increasing the performance

- Decrease the miss rate
  - Hardware, compiler and software
- Decrease the penalty of a miss
  - Hardware
- Decrease the penalty of a hit
  - Hardware

Compiler and software ways of decreasing the miss rate

Four compiler techniques to improve cache locality

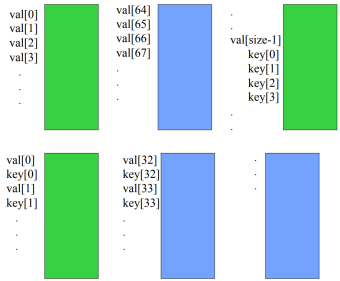
- Merging arrays
- Loop interchange
- Loop fusion
- Loop blocking / tiling

Merging arrays

```
int val[size];
int key[size];
```

It's usually more efficient to store together containers whose elements are used frequently together

- Example: at least 2 blocks must be in cache to begin using the arrays instead of 1 when arrays are merged
- The compiler has limitations founding all the merged arrays
  - Think about memory accesses when designing your data structures



VANDERBILT UNIVERSITY

Loop Interchange and Loop Fusion

Loop interchange is needed to accesses memory in a continuous fashion

- In the example below, the code is accessing x by columns, so two consecutive accesses are spaced 1000 words apart in memory

Loop fusion is needed to minimize blocks bouncing in and out of cache

- In the example below, the array x is used in each loop, so needs to be loaded into cache twice.

```
for(j=0; j<1000; j++)
  for(i=0; i<1000; i++)
        x[i][j] *= a;

for(j=0; j<1000; j++)
  for(i=0; i<1000; i++)
        y[i][j] = x[i][j] + b;
~
```

```
for(i=0; i<1000; i++)
        x[a[i]] *= 2;

for(i=0; i<compute_end(); i++)
        y[i] = x[a[i]] + 10;
~
```

Modern compilers do a good job with loop interchange and fusion

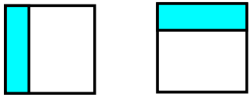
- Help the compiler by not making it difficult to understand the loop patterns

VANDERBILT UNIVERSITY

Loop Blocking / Tiling

```
for(j=0; j<1000; j++)
  for(i=0; i<1000; i++)
        x[i][j] = y[j][i];
~
```

- Loop interchange would exploit spatial reuse for either x or y, depending on which loop is outermost.



```
for(idx_j=0; idx_j<1000; idx_j+=50)
  for(idx_i=0; idx_i<1000; idx_i+=50)
        for(j=idx_j; j<idx_j+50; j++)
          for(i=idx_i; i<idx_i+50; i++)
                x[i][j] = y[j][i];
~
```

Blocking/tiling breaks loops into strips, then interchanges to form blocks

- Block sizes are selected so that they are small enough to exploit locality carried on both loops

Conclusion

Write clean and predictable code !

- Be aware of what the hardware is doing
- Be aware of what the compiler can optimize
- Test performance before deciding to optimize something
- Try to keep readability high

Further reading

1. **CPU Hardware Optimizations** in *The Software Optimization Cookbook*, pages 51-141 *Software Optimization for High Performance Computing: Creating Faster Applications*, pages 7-56
2. **Compiler Optimizations for Uniprocessors** in *Software Optimization for High Performance Computing: Creating Faster Applications*, pages 81-124

Tool to test the assembly code generated by the compiler

https://godbolt.org/