Overview

PART I
Uniprocessors

- Processors
  - Processor architectures
  - Instruction set architectures
  - Instruction scheduling and execution
- Data storage
  - Memory hierarchy
  - Caches and TLB
- Compiler optimizations for uniprocessors

PART II
Multiprocessors

- Parallel models
- Types of parallel computers
  - Pipeline and vector machines
  - Shared memory multiprocessors
  - Distributed memory multicomputers
  - Message passing multicomputers
Levels of Parallelism

Superscalar and multicore processors

- Instruction-level parallelism
  - Processor core
  - Processor core with HT
  - Executes multiple instructions (operations) at the same time
- Hyper-threading
  - Processor core
  - Executes the work of two threads
- 2 cores on 1 chip
  - Processor core
  - Executes two threads with shared on-chip resources

Multiprocessing
- 2 processors
  - 1 computer
- Distributed processing
  - 3 computers

Less parallelism

More parallelism (=stacked)
# Processor Families

<table>
<thead>
<tr>
<th>Family</th>
<th>Instruction Set Architecture (ISA)</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISC</td>
<td>DEC VAX</td>
<td>VAX-11/780</td>
</tr>
<tr>
<td></td>
<td>Intel 80x86 (IA-32)</td>
<td>Intel Pentium Pro</td>
</tr>
<tr>
<td>Vector</td>
<td>CRAY</td>
<td>CRAY T90</td>
</tr>
<tr>
<td></td>
<td>Convex</td>
<td>Convex C-4</td>
</tr>
<tr>
<td>RISC</td>
<td>HP PA-RISC</td>
<td>PA-8600</td>
</tr>
<tr>
<td></td>
<td>SGI MIPS</td>
<td>MIPS R10000</td>
</tr>
<tr>
<td></td>
<td>DEC Alpha</td>
<td>Alpha21264</td>
</tr>
<tr>
<td></td>
<td>Sun Sparc</td>
<td>Sun UltraSparc-III</td>
</tr>
<tr>
<td></td>
<td>IBM PowerPC</td>
<td>IBM Power3</td>
</tr>
<tr>
<td>VLIW</td>
<td>Multiflow</td>
<td>Multiflow Trace</td>
</tr>
<tr>
<td></td>
<td>Cydrome</td>
<td>Cydrome Cydra-5</td>
</tr>
<tr>
<td></td>
<td>Intel IA-64</td>
<td>Intel Itanium</td>
</tr>
</tbody>
</table>
CISC, RISC, VLIW, and Vector Processor Families

- CISC (*complex instruction set computer*)
  - CISC ISAs offer specialized instructions of various (and variable) length
  - Instructions typically executed in *microcode*

- RISC (*reduced instruction set computer*)
  - No microcode and relatively few instructions
  - Many RISC processors are *superscalar* (for instruction-level parallelism)
  - Only load and store instructions access memory
  - Single common instruction word length
  - 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 (not necessarily parallel)
  - Multiple vector instructions can be chained
Superscalar Architectures

Data
- FPR
- GPR

Functional units
- FPU
- ALU

Program and control
- Control
- Comparison result
- Branch
- Instruction Decode
- L1 instruction cache

Load/store
- L1 data cache

L2 cache

Memory system

HPC Fall 2010
An instruction pipeline increases the instruction bandwidth

Classic 5-stage pipeline:
- IF instruction fetch
- ID instruction decode
- EX execute (functional units)
- MEM load/store
- WB write back to registers and forward results into the pipeline when needed by another instruction

Five instructions are in the pipeline at different stages
Instruction Pipeline Example

- Example 4-stage pipeline
- 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
  - Etc.
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 or more cycles

- **Example**
  - Suppose a stall of one cycle occurs in the IF stage of the purple instruction
  - Cycle 3: Purple cannot be decoded and a no-operation (NOP) is inserted
N-way Superscalar RISC

- RISC instructions have the same word size
  - Can fetch multiple instructions without having to know the instruction content of each
- N-way superscalar RISC processors fetch N instructions each cycle
  - Increases the instruction-level parallelism

Two-way superscalar RISC pipeline
A CISC Instruction Set in a Superscalar Architecture

- Pentium processors translate CISC instructions to RISC-like µOps
  - Higher instruction bandwidth
  - Maintains instruction set architecture (ISA) compatibility
- Pentium 4 has a 31 stage pipeline divided into three main stages:
  - Fetch and decode
  - Execution
  - Retirement

Simplified block diagram of the Intel Pentium 4
Instruction Fetch and Decode

- Pentium 4 decodes instructions into \( \mu \)Ops and deposits the \( \mu \)Ops in a trace cache
  - Allows the processor to fetch the \( \mu \)Ops trace of an instruction that is executed again (e.g. in a loop)
- Instructions are fetched:
  - Normally in the same order as stored in memory
  - Or fetched from branch targets predicted by the branch prediction unit
- Pentium 4 only decodes one instruction per cycle, but can deliver up to three \( \mu \)Ops per cycle to the execution stage
- RISC architectures typically fetch multiple instructions per cycle

Simplified block diagram of the Intel Pentium 4
Instruction Execution Stage

- Executes multiple μOps in parallel
  - Instruction-level parallelism (ILP)
- The scheduler marks a μOp for execution when all operands of the μOp are ready
  - The μOps on which a μOp depends must be executed first
  - A μOp can be executed out-of-order in which it appeared
- Pentium 4: a μOp is re-executed when its operands were not ready
- On Pentium 4 there are 4 ports to send a μOp into
- Each port has one or more fixed execution units
  - Port 0: ALU0, FPMOV
  - Port 1: ALU1, INT, FPEXE
  - Port 2: LOAD
  - Port 3: STORE

Simplified block diagram of the Intel Pentium 4
Retirement

- Looks for instructions to mark completed
  - Are all \( \mu \text{Ops} \) of the instruction executed?
  - Are all \( \mu \text{Ops} \) of the preceding instruction retired? (putting instructions back in order)

- Notifies the branch prediction unit when a branch was incorrectly predicted
  - Processor stops executing the wrongly predicted instructions and discards them (takes \( \approx 10 \) cycles)

- Pentium 4 retires up to 3 instructions per clock cycle

Simplified block diagram of the Intel Pentium 4
Software Optimization to Increase CPU Utilization

- Processors run at maximum speed 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

- The processor can only improve #1 to a certain level with *out-of-order scheduling* and partly #2 with *hardware prefetching*

- Compiler optimizations effectively target #1-3

- The programmer can help improve #1-5
Instruction Latency and Throughput

- **Latency**: the number of clocks to complete an instruction when all of its inputs are ready
- **Throughput**: the number of clocks to wait before starting an *identical* instruction
  - Identical instructions are those that use the same execution unit
- The example shows three multiply operations, assuming there is only one multiply execution unit
- Actual typical latencies (in cycles)
  - Integer add: 1
  - FP add: 3
  - FP multiplication: 3
  - FP division: 31

\[ a = u \times v; \quad b = w \times x; \quad c = y \times z \]
Instruction Latency Case Study

- Consider two versions of Euclid’s algorithm
  1. Modulo version
  2. Repetitive subtraction version
- Which is faster?

```c
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;
    }
}

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;
        }
    }
}
```

Modulo version

Repetitive subtraction
Instruction Latency Case Study

- Consider the cycles estimated for the case $a=48$ and $b=40$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>#</th>
<th>Latency</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modulo</td>
<td>2</td>
<td>68</td>
<td>136</td>
</tr>
<tr>
<td>Compare</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Branch</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Other</td>
<td>6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>14</td>
<td></td>
<td>148</td>
</tr>
</tbody>
</table>

**Modulo version**

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>#</th>
<th>Latency</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Compare</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Branch</td>
<td>14</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>Other</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>24</td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

**Repetitive subtraction**

<table>
<thead>
<tr>
<th>Modulo version</th>
<th>Repetitive subtraction</th>
<th>Blended version</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.55 sec</td>
<td>14.56 sec</td>
<td>12.14 sec</td>
</tr>
</tbody>
</table>
Data Dependences

- 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
- The example shows a RAW dependence
- WAR and WAW dependences exist because of storage location reuse (overwrite with new value)
  - WAR and WAW are sometimes called false dependences
  - RAW is a true dependence
Data Dependence Case Study

- Removing redundant operations may increase the number of dependences
- Example: two versions to initialize a finite difference matrix
  1. Recurrent version with lower FP operation count
  2. Non-recurrent version with fewer dependences
- Which is fastest depends on effectiveness of loop optimization and instruction scheduling by compiler (and processor) to hide latencies and the number of distinct memory loads

```
dxi=1.0/h(1)
do i=1,n
  dxo=dxi
  dxi=1.0/h(i+1)
diag(i)=dxo+dxi
  offdiag(i)=-dxi
endo
```

With recurrence

```
dxi=1.0/h(i)
do i=1,n
  dxo=1.0/h(i)
  dxi=1.0/h(i+1)
diag(i)=dxo+dxi
  offdiag(i)=-dxi
endo
```

Without recurrence
**Case Study (1)**

*Intel Core 2 Duo 2.33 GHz*

\[
dxi = \frac{1.0}{h[0]};
\]

\[
\text{for (i=1; i<n; i++)}
\]

\[
\{ 
\text{dxo = dxi;}
\]
\[
\text{dxo = \frac{1.0}{h[i]};}
\]
\[
\text{dxo = \frac{1.0}{h[i+1]};}
\]
\[
\text{diag[i] = dxo+dxo;}
\]
\[
\text{offdiag[i] = -dxo;}
\]
\[
}\n\]

\[
\text{for (i=1; i<n; i++)}
\]

\[
\{ 
\text{dxo = \frac{1.0}{h[i]};}
\]
\[
\text{dxo = \frac{1.0}{h[i+1]};}
\]
\[
\text{diag[i] = dxo+dxo;}
\]
\[
\text{offdiag[i] = -dxo;}
\]
\[
}\n\]

\[
gcc -O3 fdinit.c
time ./a.out
\]
\[
0.135 \text{ sec}
\]

\[
gcc -O3 fdinit.c
time ./a.out
\]
\[
0.270 \text{ sec}
\]

*One cross-iteration flow dependence but fewer memory loads*

*One more memory load but fewer deps*

*Dependence spans multiple instructions, so has no big impact*
Case Study (2)

Timing comparison of non-optimized kernels
recurrent version: 1.856 sec
non-recurrent version: 7.315 sec

Timing comparison of compiler optimized kernels
recurrent version: 0.841 sec
non-recurrent version: 0.841 sec

```
f77 -g fdinit.f -o fdinit
collect -o fdinit.er ./fdinit
```

UltraSparc IIIi 1.2 GHz

```
f77 -fast -g fdinit.f -o fdinit
collect -o fdinit.er ./fdinit
```

9/8/10

HPC Fall 2010
Instruction Scheduling

- 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
- The example shows an optimal instruction schedule, assuming there is one multiply and one add execution unit
  - Note that $m = k + n$ uses the old value of $n$ (WAR dependence)
  - Advanced processors remove WAR dependences from the execution pipeline using register renaming

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

- Register renaming performed by a processor removes unnecessary WAR dependences caused by register reuse
  - 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

\[
\begin{align*}
  r_0 &= r_1 + r_2 \quad \text{Sequential} \\
  r_2 &= r_7 + r_4 \\
  r_0 &= r_1 + r_2; \quad r_2' = r_7 + r_4 \quad \text{Parallel} \\
  \ldots \\
  r_2 &= r_2'
\end{align*}
\]

Pitfall: it does not help to manually remove WAR dependences in program source code by introducing extra scalar variables, because the compiler’s register allocator reuses registers assigned to these variables and thereby reintroduces the WAR dependences at the register level.
Data Speculation

- A load should be initiated as far in advance as possible to hide memory latency.
- When a store to address A1 is followed by a load from address A2 then there is a RAW dependence when A1=A2.
  - A compiler assumes there is a RAW dependence if it cannot disprove A1=A2.
- The *advanced load instruction* allows a process to ignore a potential RAW dependence and sort out the conflict at run time when the store address is known and is the same.
Branch Prediction

- **Branch prediction** is a architectural feature that enables a processor to fetch instructions of a target branch
  - When predicted correctly there is no branch penalty
  - When not predicted correctly, the penalty is typically >10 cycles

Branch prediction uses a history mechanism per branch instruction by storing a compressed form of the past branching pattern

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

*Simple branch pattern that is predicted correctly by processor*

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

*Random branch pattern that is difficult to predict*
Improving Branch Prediction

- In a complicated branch test in C/C++ move the simplest to predict condition to the front of the conjunction
  \[ \text{if (i == 0 && a[i] > 0)} \]
  This example also has the added benefit of testing the more costly \( a[i] > 0 \) less frequently.

- Rewrite conjunctions to logical expressions
  \[ \text{if (t1==0 && t2==0 && t3==0)} \]
  \[ \Rightarrow \text{if ( t1|t2|t3 == 0 )} \]

- Use max/min or arithmetic to avoid branches
  \[ \text{if (a >= 255)} a = 255; \]
  \[ \Rightarrow a = \text{min(a, 255)}; \]

- Note that in C/C++ the \texttt{cond?then:else} operator and the \&\& and \|\| operators result in branches!
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
  - Exceptions are ignored
- A check operation verifies that whether the load triggered an exception (e.g. bus error)
  - Reraise the exception

```plaintext
if (i<n)
  x = a[i]

if (i>n) jump to skip
load a[i] into r0
skip:
  ...

speculative load a[i] into r0
if (i>n) jump to skip
check spec load exceptions
skip:
  ...
```
Data Storage

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

- Storage systems are organized in a hierarchy:
  - Speed
  - Cost
  - Volatility

- Memory Hierarchy
  - faster
  - cheaper per byte
  - volatile
Performance of Storage

- 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

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

- CPU
  - TLB
  - registers

- L1 Cache
- I-cache
- D-cache

- L2 Cache
- Unified

- Main Memory
  - Memory Bus

- Disk
  - I/O Bus

- On/off chip
- CPU
- Memory Bus
- I-cache
- D-cache
- L1 Cache
- Unified
- Main Memory
- Disk
Memory Access

- A logical address is translated into a physical address in virtual memory using a page table
  - The translation lookaside buffer (TLB) is an efficient on-chip address translation cache
  - Memory is divided into pages
  - Virtual memory systems store pages in memory (RAM) and on disk
  - Page fault: page is fetched from disk

- L1 caches (on chip):
  - I-cache stores instructions
  - D-cache stores data

- L2 cache (E-cache, on/off chip)
  - Is typically unified: stores instructions and data
Locality in a Memory Reference Pattern

- **The working set model**
  - A process can be in physical memory if and only if all of the pages that it is currently using (the most recently used pages) can be in physical memory
  - *Page thrashing* occurs if pages are moved excessively between physical memory and disk

- Operating system picks and adjusts working set size
  - OS attempts to minimize page faults
Translation Lookaside Buffer

Logical address to physical address translation with TLB lookup to limit page table reads (page table is stored in physical memory)

Logical pages are mapped to physical pages in memory (typical page size is 4KB)
Finding TLB Misses

```
collect -o testprog.er -h DTLB_miss,on ./testprog
```
Caches

- **Direct mapped cache**: each location in main memory can be cached by just one cache location
- **N-way associative cache**: each location in main memory can be cached by one of $N$ cache locations
- **Fully associative**: each location in main memory can be cached by any cache location
- Experiment shows SPEC CPU2000 benchmark cache miss rates
Cache Details

- An $N$-way associative cache has a set of $N$ cache lines per row
- A cache line can be 8 to 512 bytes
  - Longer lines increase memory bandwidth performance, but space can be wasted when applications access data in a random order
- A typical 8-way L1 cache (on-chip) has 64 rows with 64 byte cache lines
  - Cache size = $64\text{ rows} \times 8\text{ ways} \times 64\text{ bytes} = 32768\text{ bytes}$
- A typical 8-way L2 cache (on/off chip) has 1024 rows with 128 byte cache lines
Cache Misses

- Compulsory misses
  - Caused by the first reference to a datum
  - Misses are effectively reduced by prefetching

- Capacity misses
  - Cache size is finite
  - Misses are reduced by limiting the working set size of the application

- Conflict misses
  - Replacement misses are caused by the choice of victim cache line to evict by the replacement policy
  - Mapping misses are caused by level of associativity

Compulsory miss:
reading `some_static_data[0]`

```
for (i = 0; i < 100000; i++)
    X[i] = some_static_data[i];
```

Capacity miss:
`X[i]` no longer in cache

```
for (i = 0; i < 100000; i++)
    X[i] = X[i] + Y[i];
```

Conflict miss:
`X[i]` and `Y[i]` are mapped to the same cache line (e.g. when cache is direct mapped)
False Sharing

- On a multi-core processor each core has an L1 cache and shares the L2 cache with other cores
- *False sharing* occurs when two caches of processors (or cores) cache two different *non-shared data items* that reside on the *same cache line*
- *Cache coherency protocol* marks the cache line (on all cores) dirty to force reload
- To avoid false sharing:
  - Allocate non-shared data on different cache lines (using `malloc`)
  - Limit the use of global variables
Finding Cache Misses

```
collect -o test.memory.1.er -S off -p on -h ecstall,on,cycles,on ./cachetest -g memory.erg
collect -o test.memory.2.er -S off -h icstall,on,dcstall,on ./cachetest -g memory.erg
```
Compiler Optimizations

- General optimizations
- Single loop optimizations
- Nested loop optimizations
- Global optimizations
## General Optimizations

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduce memory instructions</th>
<th>Reduce FP instructions</th>
<th>Reduce integer instructions</th>
<th>Reduce branch instructions</th>
<th>Reduce memory latencies and/or bandwidth</th>
<th>Memory latencies</th>
<th>Reduce effect of instruction latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register allocation</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++ register type</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++ asm macro</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++ include &lt;math.h&gt;</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniqueness of memory addr</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant folding and propagation</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common subexpression elimination</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strength reductions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Increase FMA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill branch delay slots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Register Allocation

Example

\[
\begin{align*}
\text{a} &:= \text{read}(); \\
\text{b} &:= \text{read}(); \\
\text{c} &:= \text{read}(); \\
\text{a} &:= \text{a} + \text{b} + \text{c}; \\
\text{if } (\text{a}<10) &\{ \\
\quad &\text{d} := \text{c} + 8; \\
\quad &\text{write} (\text{c}); \\
\} \text{ else if } (\text{a}<20) &\{ \\
\quad &\text{e} := 10; \\
\quad &\text{d} := \text{e} + \text{a}; \\
\quad &\text{write} (\text{e}); \\
\} \text{ else } &\{ \\
\quad &\text{f} := 12; \\
\quad &\text{d} := \text{f} + \text{a}; \\
\quad &\text{write} (\text{f}); \\
\} \\
\text{write} (\text{d});
\end{align*}
\]
Register Allocation

- **Register pressure**: when there is an insufficient number of registers to hold all live variables, then register values have to be stored in memory (*register spill*).
- This results in lower performance when spill occurs frequently such as within a loop.

Construct interference graph

```
<table>
<thead>
<tr>
<th>f</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>c</td>
</tr>
</tbody>
</table>
```

Solve

```
1 — 2 — 3
```

Assign registers:

- \( a = r_2 \)
- \( b = r_3 \)
- \( c = r_1 \)
- \( d = r_2 \)
- \( e = r_1 \)
- \( f = r_1 \)

```
r_2 := \text{read}();
r_3 := \text{read}();
r_1 := \text{read}();
r_2 := r_2 + r_3 + r_1;
if (r_2 < 10)
  \{ \ r_2 := r_1 + 8;
    \text{write}(r_1);
  \}
else if (r_2 < 20)
  \{ \ r_1 := 10;
    r_2 := r_1 + r_2;
    \text{write}(r_1);
  \}
else
  \{ \ r_1 := 12;
    r_2 := r_1 + r_2;
    \text{write}(r_1);
  \}
\text{write}(r_2);
```
C/C++ Register Data Type

- The C/C++ register data type is a hint to the compiler to store the scalar variable in a register to avoid register spill
- Additionally prevents potential pointer aliases to this variable

```c
register int i;
int k = 10;
int a[100];
int *p = init(a, &k);
for (i = 0; i < 100; i++)
    *p++ = k;
```

```c
int *init(int a[], int *k)
{
    return &a[*k];
}
```

Does not return pointer that includes a memory area that covers `k` but compiler may not have that information

Pointer arithmetic can make it difficult for a compiler to disprove that `p` does not point to other integer variables such as `k` though it can never point to `i`
Uniqueness of Addresses

- **Aliases** are references to the same data via two variables, function arguments, or pointers
- Aliases prevent a compiler from moving and optimizing load/stores
  - Data speculation gives the compiler some optimization opportunities
- Compilers treat *potential* aliases as a fait accompli
  - Use compiler options
  - Use C/C++ `restrict` keyword when pointer variables and argument pointers always point to different objects

```c
void copy_dt(dt * restrict p, dt * restrict q, int n)
{
    int i;
    for (i = 0; i < n; i++)
        p[i] = q[i];
}
```

Memory accesses cannot overlap
Uniqueness of Addresses (cont’d)

- Compiler options
  - gcc -fargument-noalias -fargument-noalias-global
  - suncc -xrestrict
  - icc -restrict

- Use additional compiler options to force strict aliasing rules in C/C++ that prohibit the use of pointers of different types to point to the same address
  - gcc -fstrict-aliasing
  - suncc -falias-level=basic

```c
int *p;
double x;
p = (int*) &x; // prohibited
```

Register allocation is effective, because `p`’s pointer dereferences cannot refer to `x`
Dead Code Elimination

\[ i = 0; \]
\[ \text{if (} i \neq 0 \text{) } b = x+y \]

- Remove unreachable code

- Remove useless assignment

\[ \text{b = a + 1} \]
\[ a = b + c \]
\[ \ldots \]

Assuming \( a \) is dead (has no use in code below)

But what if another process reads \( a \)?
Compilers are \textbf{brain dead}: most optimizations are not multi-task safe!
Constant Folding and Propagation

- Folds arithmetic operations on constants into a single constant
- Constant assignments are eliminated when possible and the constants are propagated to the register uses
Common Subexpression Elimination

- Remove redundant computations
- Increases the number of dependences and therefore may also increase register pressure and hamper scheduling

```
a = b + c
b = a - d
c = b + c
d = a - d
```

```
a = b + c
b = a - d
c = b + c
d = b
```

```
t1 = b * c
t2 = a - t1
t3 = b * c
t4 = t2 + t3
```

```
t1 = b * c
t2 = a - t1
t4 = t2 + t1
```
Strength Reductions

- Replace expensive operations with cheaper ones
  - Replace integer multiplications and divisions with bit shifts
  - Replace multiplications with additions
  - Replace floating point division by multiplication when denominator is a constant

```
r3 = r4 / 3.0
r5 = 2 * r4
r6 = r4 / 4
r6 = r4 >> 2
```

Replace with

```
r3 = r4 * 0.333
r5 = r4 + r4
```
FMA

- Many processors have compound floating-point fused multiply and add (fma) instructions and fused negate multiply and add (fnma) instructions

- Compiler options:
  - suncc -fma=fused
  - icc -IPF-fma

- Example: complex arithmetic
  - \( x = (x_r, x_i), y = (y_r, y_i), z = (z_r, z_i), u = (u_r, u_i), \) compute \( u = x\cdot y + z \)

\[
\begin{align*}
  f_1 & = x_r\cdot y_r \\
  f_2 & = x_r\cdot y_i \\
  f_3 & = x_i\cdot y_i \\
  f_3 & = f_1 - f_3 \\
  f_4 & = x_i\cdot y_r \\
  f_4 & = f_4 + f_2 \\
  u_r & = z_r + f_3 \\
  u_i & = z_i + f_4
\end{align*}
\]

Without fma

\[
\begin{align*}
  f_{1a} & = x_r\cdot y_r + z_r \\
  f_{2a} & = x_r\cdot y_i + z_i \\
  u_r & = -x_i\cdot y_i + f_{1a} \\
  u_i & = x_r\cdot y_r + f_{2a}
\end{align*}
\]

With fma instructions
Fill Branch Delay Slots

- **Branch delay slots** reduce the penalty of pipeline stalls caused by mispredicted branches.
- Compiler fills branch delay slots when optimizing code:
  - gcc -fdelayed-branch
- One or more instructions that occur after a branch instruction are always executed:
  - These instructions were already fetched in the pipeline and in the IF, ID, and possibly the EX stage.

```c
for (i = 0; i < n; i++)
    s = s + 3.0

i = -1
loop:
    s = s + 3.0
    i = i + 1
    if (i<n) jump to loop
    nop ! Branch delay slot

i = -1
loop:
    i = i + 1
    if (i<n) jump to loop
    s = s + 3.0
```
# Single Loop Optimizations

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduce memory instructions</th>
<th>Reduce FP instructions</th>
<th>Reduce integer instructions</th>
<th>Reduce branch instructions</th>
<th>Reduce memory latencies and/or memory bandwidth</th>
<th>Reduce effect of instruction latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Induction variable optimization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefetching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Test promotion in loops</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop peeling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Loop fusion</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Loop fission</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Copying</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Block and copy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Unrolling</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Software pipelining</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Loop invariant code motion</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array padding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Optimizing reductions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

9/8/10  HPC Fall 2010
Induction Variable Optimization

- Loop counters and other variables that are iteratively updated are *induction variables*
- *Loop strength reduction* replaces expressions by induction variables
- *Induction variable elimination* removes redundant induction variables

```plaintext
DO i = 0,n,2
   IA(i) = i*k+m
ENDDO

ic = m
is = 2*k
DO i = 0,n,2
   IA(i) = ic
   ic = ic+is
ENDDO

j = 1
DO i = 0,n
   IA(j) = IB(i)
   j = j+1
ENDDO

DO i = 0,n
   IA(i+1) = IB(i)
ENDDO
```
Prefetching

- **Hardware prefetching**: the processor detects a sequential, possibly up/down strided access pattern to memory and prefetches D-cache lines
- **Software prefetching**: specialized instruction inserted by the compiler or added by programmer (see example) to load D-cache line, or to load specific choice of caches
  - gcc -fprefetch-loop-arrays
  - suncc -xprefetch=auto
  -icc -prefetch (IA64 only)

```
for (i = 0; i < n; i++)
a[i] = (b[i] + 1) * 2;
```

*Original loop*

```
#include <sun_prefetch.h>
static double a[N], b[N];
int i, m = (n/8) * 8;
for (i = 0; i < m; i += 8) {
  sun_prefetch_read_many(&b[i]+256);
  sun_prefetch_write_many(&a[i]+256);
  a[i] = (b[i] + 1) * 2;
  a[i+1] = (b[i+1] + 1) * 2;
  a[i+2] = (b[i+2] + 1) * 2;
  a[i+3] = (b[i+3] + 1) * 2;
  a[i+4] = (b[i+4] + 1) * 2;
  a[i+5] = (b[i+5] + 1) * 2;
  a[i+6] = (b[i+6] + 1) * 2;
  a[i+7] = (b[i+7] + 1) * 2;
}
#pragma unroll(1)
for (i = m+1; i < n; i++)
a[i] = (b[i] + 1) * 2;
```

*New code with prefetch*
Test Promotion in Loops

- Test promotion, also called loop unswitching, moves a loop-independent test out of the loop and duplicates the loop body.
- Removes branches that can otherwise interfere with pipelining, loop parallellization and vectorization.

```
DO I = 1,N
  IF (A .GT. 0) THEN
    X(I) = X(I) + 1
  ELSE
    X(I) = 0
  ENDIF
ENDDO

IF (A .GT. 0) THEN
  DO I = 1,N
    X(I) = X(I) + 1
  ENDDO
ELSE
  DO I = 1,N
    X(I) = 0
  ENDDO
ENDIF
```
Loop Peeling

- Some numerical codes include loops that handle boundary conditions in the first and/or last iteration
- Loop peeling takes the first/last iteration out of the loop and duplicates the loop body for these cases

```
DO I = 1,N
  IF (I .EQ. 1) THEN
    X(I) = 0
  ELSEIF (I .EQ. N) THEN
    X(I) = N
  ELSE
    X(I) = X(I) + Y(I)
  ENDIF
ENDDO

X(1) = 0
DO I = 2,N-1
  X(I) = X(I) + Y(I)
ENDDO
X(N) = N

First and last iteration peeled, assuming N > 2
```
Loop Fusion

- Fuse two loops into one by merging the loop bodies
  - Checks for array-based dependences between loops: if a dependence exists then loop fusion may not be possible
  - Resulting loop must not be too large (may lead to register spill)

```c
for (i = 0; i < n; i++)
    temp[i] = x[i] * y[i];
for (i = 0; i < n; i++)
    z[i] = w[i] + temp[i];
```

```c
DO I = 2, N
    B(I) = T(I) * X(I)
ENDDO
DO I = 2, N
    A(I) = B(I) - B(I-1)
ENDDO
```

```c
for (i = 0; i < n; i++)
    { temp = x[i] * y[i];
      z[i] = w[i] + temp;
    }
```

```
scalarized
```
Loop Fission

- Loop fission (loop distribution) splits a loop into two or more loops
- Compiler computes the acyclic condensation of the dependence graph to find a legal order of the split loops

```
S1 DO I = 1, 10
S2 A(I) = A(I) + B(I-1)
S3 B(I) = C(I-1)*X + Z
S4 C(I) = 1/B(I)
S5 D(I) = sqrt(C(I))
S6 ENDDO
```

![Dependence graph](image)

![Acyclic condensation](image)

9/8/10
Copy array to a new “safe” location to avoid cache thrashing when two or more arrays in a loop map to the same cache line.

```c
double X[N], Y[N];
for (I=0; I<N; I++)
    Y[I] = Y[I] + X[I];
```

*Original loop suffering from cache thrashing*
Block and Copy

- Use blocking to reduce memory storage overhead when arrays are large and much larger than the cache

double X[N], Y[N];
for (I=0; I<N; I++)
    Y[I] = Y[I] + X[I];

Original loop suffering
from cache thrashing
Loop Unrolling

- Unroll loop by a constant, usually between 2 and 8 iterations
  - gcc -funroll-loops -funroll-all-loops -fvariable-expansion-in-unroller
  - suncc -unroll=n
  - icc -unroll

```
DO I=1,N
  Y(I) = Y(I) + X(I)
ENDDO

DO I=1,N,4
  Y(I) = Y(I) + X(I)
  Y(I+1) = Y(I+1) + X(I+1)
  Y(I+2) = Y(I+2) + X(I+2)
  Y(I+3) = Y(I+3) + X(I+3)
ENDDO
```

Unrolled by compiler
assuming N-1 is multiple of 4
Software Pipelining with Rotating Registers

- Increase the dependence height by increasing the distance between load/stores
- Registers are renumbered by the processor with each iteration of the loop
- This is a form of software pipelining
  - Register values stay alive for a fixed iteration window thereby increasing the distance between load/store dependences
  - Dependences are overlapped in each iteration and spread across the loop iterations

```fortran
DO I = 0,N
   Y(I) = X(I)
ENDDO

DO I = 0,N
   load X(I) into r0
   store r0 to Y(I)
ENDDO

load X(0) into r3
load X(1) into r2
load X(2) into r1
DO I = 0, N-3
   load X(I+3) into r0
   store r3 to Y(I)
   rotate r0 to r3
ENDDO
store r2 into Y(N-2)
store r1 into Y(N-1)
store r0 into Y(N)
```
Modulo Scheduling

- A form of software pipelining that copies the chunk of instructions of the loop body to execute them in parallel in each iteration of the new loop
- Uses a rotating register file
- Consider the loop:

\[
\text{DO } i = 0, 6 \\
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{E} \\
\text{F} \\
\text{G} \\
\text{H} \\
\text{ENDDO}
\]
Loop Invariant Code Motion

- A loop-invariant expression does not change its value from one iteration to another
- Loop-invariant code motion moves loop-invariant code out of the loop to be executed only once
  - Hoisting: moves code before the loop
  - Sinking: moves code after the loop

DO I=1,N
  Y(J) = Y(J) + 2 * K * X(I)
ENDDO

Load and store from same address repeatedly

IF (N .GE. 1)
  t1 = Y(J)
  t2 = 2*K
ENDIF
DO I=1,N
  t1 = t1 + t2 * X(I)
ENDDO
IF (N .GE. 1)
  Y(J) = t1
ENDIF
Array Padding

- Memory is organized in banks that are a power of two in size
- Avoid allocating arrays that are a power of two in size
- Array padding: when two or more arrays share cache lines, then padding the array with additional leading elements may help to move the start of the array access up

Before padding

After padding
Optimizing Reductions

- Optimize reductions: rewrite reduction loops or use library calls to highly optimized code or parallelize the reduction
- Not all compilers support this kind of optimization

```
a = 0.0
DO i=1,n
   a = a + X(i)
ENDDO
```

```
a = 0.0
b = 0.0
c = 0.0
d = 0.0
DO i=1,n,4
   a = a + X(i)
   b = b + X(i+1)
   c = c + X(i+2)
   d = d + X(i+3)
ENDDO
a = a + b + c + d
```

Unrolled and optimized assuming n-1 is multiple of 4
## Nested Loop Optimizations

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduce memory instructions</th>
<th>Reduce FP instructions</th>
<th>Reduce integer instructions</th>
<th>Reduce branch instructions</th>
<th>Reduce memory latencies and/or bandwidth</th>
<th>Reduce effect of instruction latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop interchange</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Outer loop unrolling</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Unroll and jam</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blocking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Block and copy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Loop Interchange

- *Loop interchange* exchanges outer with inner loop
- Checks for array-based dependences loops: if a dependence exists then loop interchange may not be possible
- May require loop fusion of inner loops first

```fortran
DO I = 1, N
  DO J = 1, M
    A(I,J) = A(I,J-1) + B(I,J)
  ENDDO
ENDDO
```

```fortran
DO J = 1, M
  DO I = 1, N
    A(I,J) = A(I,J-1) + B(I,J)
  ENDDO
ENDDO
```

*Note: Fortran uses column-major storage while C uses row-major storage*
Outer Loop Unrolling

- Unrolls the outer loop, usually followed by fusion of inner loops
- Reduces branches and may reduce memory operations

```
DO J = 1, N
    DO I = 1, M
        A(I, J) = A(I, J) + X(I) * Y(J)
    ENDDO
ENDDO

DO J = 1, N, 2
    DO I = 1, M
        A(I, J) = A(I, J) + X(I) * Y(J)
    ENDDO
    DO I = 1, M
        A(I, J+1) = A(I, J+1) + X(I) * Y(J+1)
    ENDDO
ENDDO
```

Unrolled twice and optimized assuming \( N \) is multiple of 2

Need to load \( X[I] \) only once
Unroll and Jam

- **Unroll and jam** refers to unrolling inner and outer loops and jamming them back together in ways that aim to reduce the number of memory operations.

```plaintext
DO K = 1, N, 2
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K) = C(I,K) + A(I,J) * B(J,K)
    ENDDO
    DO I = 1, N
      C(I,K) = C(I,K) + A(I,J+1) * B(J+1,K)
    ENDDO
  ENDDO
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K+1) = C(I,K+1) + A(I,J) * B(J,K+1)
    ENDDO
    DO I = 1, N
      C(I,K+1) = C(I,K+1) + A(I,J+1) * B(J+1,K+1)
    ENDDO
  ENDDO
ENDDO
```

**Step 1: unroll outer loops twice**
Unroll and Jam (cont’d)

Step 2: jam the inner loops back together with loop fusion and combine assignments

DO K = 1, N, 2
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K) = C(I,K) + A(I,J) * B(J,K)
      C(I,K) = C(I,K) + A(I,J+1) * B(J+1,K)
      C(I,K+1) = C(I,K+1) + A(I,J) * B(J,K+1)
      C(I,K+1) = C(I,K+1) + A(I,J+1) * B(J+1,K+1)
    ENDDO
  ENDDO
ENDDO

DO K = 1, N, 2
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K) = C(I,K) + A(I,J) * B(J,K) + A(I,J+1) * B(J+1,K)
      C(I,K+1) = C(I,K+1) + A(I,J) * B(J,K+1) + A(I,J+1) * B(J+1,K+1)
    ENDDO
  ENDDO
ENDDO
ENDDO
Blocking

- Blocking aims to decrease cache misses in nested loops

```plaintext
DO J = 1,N
  DO I = 1,N
    Y(I) = Y(I) + A(I,J)
  ENDDO
ENDDO

NBLOCK = 1000
DO IOUTER = 1,N,BLOCK
  DO J = 1,N
    DO I = IOUTER,MIN(N,N+BLOCK-1)
      Y(I) = Y(I) + A(I,J)
    ENDDO
  ENDDO
ENDDO
ENDDO
```
Block and Copy

Combination of blocking with array copying

\[
\begin{align*}
\text{DO } & K = 1, N \\
\text{DO } & J = 1, N \\
\text{DO } & I = 1, N \\
\text{C}(I, J) & = C(I, J) + A(I, K) * B(K, J) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

Perform \( A \times B \) in blocks where each block multiply uses a local temporary copy of \( A, B, \) and \( C \) that fit in cache

\[
\begin{align*}
\text{DO } & I = 1, N, NBLOCK \\
\text{DO } & J = 1, N, NBLOCK \\
! & \text{Copy C block at } (I, J) \text{ to } T3 \\
! & \text{DO } K = 1, N, NBLOCK \\
! & \text{Copy A block at } (I, K) \text{ to } T1 \\
! & \text{Copy B block at } (K, J) \text{ to } T2 \\
! & \text{CALL MATMUL}(NBLOCK, NBLOCK, T1, T2, T3) \\
\text{ENDDO} \\
\text{ENDDO} \\
! & \text{Copy T3 to C block at } (I, J) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
Global Optimizations

- **Interprocedural analysis** (IPA) determines properties of a function code to help optimize caller’s code
  - For example, aliases can be resolved by determining which variables and parameters are actually changed by the function

- **Interprocedural optimizations** (IPO)
  - **Inlining** (*inline* keyword in C/C++) expands a function in-line to avoid function calling overhead and to enable optimizations on the function’s body in the current code region
    - gcc -finline-functions
    - suncc -xinline=auto
    - icc -finline -finline-functions
  - **Cloning** creates specialized copies of functions that are optimized based on actual parameter values
Practical Suggestions

- Split code in compute-intensive part and the rest (such as file access)
- Write clear code to help compiler detect opportunities for optimization
- Avoid bulky loops
- Use regular data structures whenever possible and set them up so that they are accessed element by element in loops to improve spatial locality
- Avoid global data (statically allocated data)
- Keep branch conditions simple
Further Reading

- [HPC] pages 7-56, 81-124
- Optional (see instructor for material):
  - [OPT] pages 51-121