Measuring performance (I)

- Response time: how long does it take to execute a certain application/a certain amount of work
- Given two platforms $X$ and $Y$, $X$ is $n$ times faster than $Y$ for a certain application if

$$n = \frac{\text{Time}_Y}{\text{Time}_X}$$ (1)

- Performance of $X$ is $n$ times higher than performance of $Y$ if

$$n = \frac{\text{Time}_Y}{\text{Time}_X} = \frac{1}{\text{Perf}_Y} = \frac{\text{Perf}_Y}{\text{Perf}_X}$$ (2)
Measuring performance (II)

- Timing how long an application takes
  - **Wall clock time/elapsed time**: time to complete a task as seen by the user. Might include operating system overhead or potentially interfering other applications.
  - **CPU time**: does not include time slices introduced by external sources (e.g. running other applications). CPU time can be further divided into
    - **User CPU time**: CPU time spent in the program
    - **System CPU time**: CPU time spent in the OS performing tasks requested by the program.

Measuring performance

- E.g. using the UNIX `time` command
Amdahl’s Law

- Describes the performance gains by enhancing one part of the overall system (code, computer)

\[
\text{Speedup} = \frac{\text{Time}_{\text{org}}}{\text{Time}_{\text{enh}}} = \frac{\text{Perf}_{\text{org}}}{\text{Perf}_{\text{enh}}} \quad (3)
\]

- Amdahl’s Law depends on two factors:
  - Fraction of the execution time affected by enhancement
  - The improvement gained by the enhancement for this fraction

\[
\text{Time}_{\text{enh}} = \text{Time}_{\text{org}}((1 - \text{Fraction}_{\text{enh}}) + \frac{\text{Fraction}_{\text{enh}}}{\text{Speedup}_{\text{enh}}}) \quad (4)
\]

\[
\text{Speedup}_{\text{overall}} = \frac{\text{Time}_{\text{org}}}{\text{Time}_{\text{enh}}} = \frac{1}{(1 - \text{Fraction}_{\text{enh}}) + \frac{\text{Fraction}_{\text{enh}}}{\text{Speedup}_{\text{enh}}}} \quad (5)
\]
Amdahl’s Law (IV)

Amdahl’s Law - example

• Assume a new web-server with a CPU being 10 times faster on computation than the previous web-server. I/O performance is not improved compared to the old machine. The web-server spends 40% of its time in computation and 60% in I/O. How much faster is the new machine overall?

\[
\text{Fraction}_{\text{enh}} = 0.4 \\
\text{Speedup}_{\text{enh}} = 10 \\
\text{using formula (5)}
\]

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enh}}) + \frac{\text{Fraction}_{\text{enh}}}{\text{Speedup}_{\text{enh}}}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.6} = 1.56
\]
Amdahl’s Law - example (II)

Example: Consider a graphics card
- 50% of its total execution time is spent in floating point operations
- 20% of its total execution time is spent in floating point square root operations (FPSQR).

Option 1: improve the FPSQR operation by a factor of 10.
Option 2: improve all floating point operations by a factor of 1.6

\[
\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22
\]

\[
\text{Speedup}_{\text{FP}} = \frac{1}{(1 - 0.5) + \frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23 \quad \rightarrow \quad \text{Option 2 slightly faster}
\]

CPU Performance Equation

- Micro-processors are based on a clock running at a constant rate
- Clock cycle time: CC<sub>c</sub>
  - length of the discrete time event in ns
- Equivalent measure: Rate CPU, = \frac{1}{CC_{\text{time}}}
  - Expressed in MHz, GHz
- CPU time of a program can then be expressed as

\[
\text{CPU} \text{time} = \text{no} \times \text{cycles} \times CC_{\text{time}} \quad (6)
\]

or

\[
\text{CPU} \text{time} = \frac{\text{no} \times \text{cycles}}{\text{CPU}}, \quad (7)
\]
CPU Performance equation (II)

- **CPI**: Average number of clock cycles per instruction
- **IC**: number of instructions

\[
CPI = \frac{no_{cycles}}{IC} \quad (8)
\]

- Since the CPI is often known (average), the CPU time is

\[
CPU_{\text{time}} = IC \times CPI \times CC_{\text{time}} \quad (9)
\]

- Expanding formula (6) leads to

\[
CPU_{\text{time}} = \frac{\text{instructions}}{\text{program}} \times \frac{no_{cycles}}{\text{instruction}} \times \frac{\text{time}}{no_{cycles}} \quad (10)
\]

CPU performance equation (III)

- According to (7) CPU performance is depending on
  - Clock cycle time → Hardware technology
  - CPI → Organization and instruction set architecture
  - Instruction count → ISA and compiler technology
- Note: on the last slide we used the average CPI over all instructions occurring in an application
- Different instructions can have strongly varying CPI’s

\[
\text{no}_{\text{cycles}} = \sum_{i=1}^{n} IC_i \times CPI_i \quad (11)
\]

\[
CPU_{\text{time}} = \left( \sum_{i=1}^{n} IC_i \times CPI_i \right) \times CC_{\text{time}} \quad (12)
\]
The average CPI for an application can then be calculated as

\[ CPI = \sum_{i=1}^{n} \frac{IC_i \times CPI_i}{IC_{total}} = \sum_{i=1}^{n} \frac{IC_i}{IC_{total}} \times CPI_i \]  

(13)

\( \frac{IC_i}{IC_{total}} \): Fraction of occurrence of that instruction in a program

Example (I)

• (Page 50/51 in the 5th Edition)
  Consider a graphics card, with
  - FP operations (including FPSQR): frequency 25%, average CPI 4.0
  - FPSQR operations only: frequency 2%, average CPI 20
  - all other instructions: average CPI 1.3333333

• Design option 1: decrease CPI of FPSQR to 2
• Design option 2: decrease CPI of all FP operations to 2.5

Using formula (13):

\[ CPI_{org} = \sum_{i=1}^{n} \frac{IC_i}{IC_{total}} \times CPI_i = (4 \times 0.25) + (1.333333 \times 0.75) = 2.0 \]

\[ CPI_1 = CPI_{org} - enh = 2.0 - 0.02(20 - 2) = 1.64 \]

\[ CPI_2 = \sum_{i=1}^{n} \frac{IC_i}{IC_{total}} \times CPI_i = (2.5 \times 0.25) + (1.333333 \times 0.75) = 1.625 \]
Example (II)

- Slightly modified compared to the previous section: consider a graphics card, with
  - FP operations (excluding FPSQR): frequency 25%, average CPI 4.0
  - FPSQR operations: frequency 2%, average CPI 20
  - all other instructions: average CPI 1.33
- Design option 1: decrease CPI of FPSQR to 2
- Design option 2: decrease CPI of all FP operations to 2.5

Using formula (13):

$$
CPI_{org} = \sum_{i=1}^{n} \frac{IC_i}{IC_{total}} \times CPI_i = (4 \times 0.25) + (20 \times 0.02) + (1.33 \times 0.73) = 2.3709
$$

$$
CPI_1 = \sum_{i=1}^{n} \frac{IC_i}{IC_{total}} \times CPI_i = (4 \times 0.25) + (2 \times 0.02) + (1.33 \times 0.73) = 2.0109
$$

$$
CPI_2 = \sum_{i=1}^{n} \frac{IC_i}{IC_{total}} \times CPI_i = (2.5 \times 0.25) + (20 \times 0.02) + (1.33 \times 0.73) = 1.9959
$$

Dependability

- Module reliability measures
  - MTTF: mean time to failure
  - FIT: failures in time
    $$
    FIT = \frac{1}{MTTF}
    $$
    (14)
  - Often expressed as failures in 1,000,000,000 hours
  - MTTR: mean time to repair
  - MTBF: mean time between failures
    $$
    MTBF = MTTF + MTTR
    $$
    (15)

- Module availability:
  $$
  M_A = \frac{MTTF}{MTTF + MTTR}
  $$
  (16)
Dependability - example

- Assume a disk subsystem with the following components and MTTFs:
  - 10 disks, MTTF=1,000,000h
  - 1 SCSI controller, MTTF=500,000h
  - 1 power supply, MTTF=200,000h
  - 1 fan, MTTF=200,000h
  - 1 SCSI cable, MTTF=1,000,000h

- What is the MTTF of the entire system?
- What is the probability, that the system fails within a 1 week period?

Dependability - example (II)

- Determine the sum of the failures in time of all components

\[
FIT_{\text{system}} = 10 \times \frac{1}{1,000,000} + \frac{1}{500,000} + \frac{1}{200,000} + \frac{1}{200,000} + \frac{1}{1,000,000} \\
= \frac{10 + 2 + 5 + 5 + 1}{1,000,000} = \frac{23}{1,000,000} = 23,000 \\
\text{MTTF}_{\text{system}} = \frac{1}{FIT_{\text{system}}} = \frac{1}{23,000} = 43,500h
\]

- Probability that the system fails within a 1 week period:

\[
p_{\text{system}}^{\text{week}} = \frac{24 \times 7}{43,500} \approx 0.00386 = 0.386%
\]
Dependability - example (III)

- What happens if we add a second power supply and we assume, that the MTTR of a power supply is 24 hours?
- Assumption: failures are not correlated
  - MTTF of the pair of power supplies is the mean time to failure of the overall system divided by the probability, that the redundant unit fails before the primary unit has been replaced
  - MTTF of the overall system:
    \[
    FIT_{system} = \frac{1}{MTTF_{power}} + \frac{1}{MTTF_{power}} = \frac{2}{MTTF_{power}}
    \]
    \[
    MTTF_{system} = \frac{1}{FIT_{system}} = \frac{MTTF_{power}}{2}
    \]

- Probability, that 1 unit fails within MTTR: \( MTTR / MTTF_{power} \)

\[
MTTF_{pair} = \frac{MTTF_{power}}{2MTTR} = \frac{MTTF_{power}^2}{2 \times 24} \approx 830,000,000
\]
Dependability - example (III)

• More generally, if
  - power supply 1 has an MTTF of $MTTF_{power_1}$
  - power supply 2 has an MTTF of $MTTF_{power_2}$

  \[
  FIT_{system} = \frac{1}{MTTF_{power_1}} + \frac{1}{MTTF_{power_2}}
  \]

  \[
  MTTF_{system} = \frac{1}{FIT_{system}}
  \]

  \[
  MTTF_{pair} = MTTF_{system} \left( \frac{MTTR}{\min(MTTF_{power_1}, MTTF_{power_2})} \right)
  \]

  Or if either power_1 or power_2 have been clearly declared to be the backup unit

  \[
  MTTF_{pair} = MTTF_{system} \left( \frac{MTTR}{MTTF_{power backup}} \right)
  \]

From 1st quiz five years ago

• In order to minimize the MTTF of their mission critical computers, each program on the space shuttle is executed by two computers simultaneously. Computer A is from manufacturer X and has a MTTF of 40,000 hours, while computer B is from manufacturer Y and has a different MTTF. The overall MTTF of the system is 4,000,000 hours.
  - How large is the probability that the entire system (i.e. both computers) fails during a 400 hour mission?
  - What MTTF does the second/backup computer have if the MTTF of the overall system is 4,000,000 hours, assuming that Computer A is failing right at the beginning of the 400 hour mission and the MTTR is 400h (i.e. the system can only be repaired after landing)?

• Solution will be posted on the web, but try first on your own!
Choosing the right programs to test a system

- Most systems host a wide variety of different applications
- Profiles of certain systems given by their purpose/function
  - Web server:
    - high I/O requirements
    - hardly any floating point operations
  - A system used for weather forecasting simulations
    - Very high floating point performance required
    - Lots of main memory
    - Number of processors have to match the problem size calculated in order to deliver at least real-time results

Choosing the right programs to test a system (II)

- Real application: use the target application for the machine in order to evaluate its performance
  - Best solution if application available
- Modified applications: real application has been modified in order to measure a certain feature.
  - E.g. remove I/O parts of an application in order to focus on the CPU performance
- Application kernels: focus on the most time-consuming parts of an application
  - E.g. extract the matrix-vector multiply of an application, since this uses 80% of the user CPU time.
Choosing the right programs to test a system (III)

- Toy benchmarks: very small code segments which produce a predictable result
  - E.g. sieve of Eratosthenes, quicksort

- Synthetic benchmarks: try to match the average frequency of operations and operands for a certain program
  - Code does not do any useful work
What is SPEC?

• The Standard Performance Evaluation Corporation (SPEC) is a non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers. SPEC develops suites of benchmarks and also reviews and publishes submitted results from our member organizations and other benchmark licensees.

• For more details see http://www.spec.org

SPEC groups

- Open Systems Group (desktop systems, high-end workstations and servers)
  • CPU (CPU benchmarks)
  • JAVA (Java client and server side benchmarks)
  • MAIL (mail server benchmarks)
  • SFS (file server benchmarks)
  • WEB (web server benchmarks)
- High Performance Group (HPC systems)
  • OMP (OpenMP benchmark)
  • HPC (HPC application benchmark)
  • MPI (MPI application benchmark)
- Graphics Performance Groups (Graphics)
  • Apc (Graphics application benchmarks)
  • Opc (OpenGL performance benchmarks)
Why do we need benchmarks?

- Identify problems: measure machine properties
- Time evolution: verify that we make progress
- Coverage: help vendors to have representative codes
  - Increase competition by transparency
  - Drive future development (see SPEC CPU2000)
- Relevance: help customers to choose the right computer

SPEC-Benchmarks

- All SPEC benchmarks are publicly available and well known/understood
  - Compiler can introduce special optimizations for these benchmarks, which might be irrelevant for other, real-world applications.
    → user has to provide the precise list of compile-flags
    → user has to provide performance of base (non-optimized) run
  - Compiler can use statistical informations collected during the first execution in order to optimize further runs (Cache hit rates, usage of registers)
- Benchmarks designed such that external influences are kept at a minimum (e.g. input/output)
SPEC CPU2000

- 26 independent programs:
  - CINT2000 - integer benchmark: 12 applications (11 in C and 1 in C++)
  - CFP2000 - floating-point benchmark: 14 applications (6 in Fortran-77, 4 in Fortran-90 and 4 in C)
- Additional information is available for each benchmark:
  - Author of the benchmark
  - Detailed description
  - Documentation regarding Input and Output
  - Potential problems and references.

CINT2000

CINT2000 (Integer Component of SPEC CPU2000):

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Language</th>
<th>Category</th>
<th>Full Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>C</td>
<td>Compression</td>
<td>HTML Text</td>
</tr>
<tr>
<td>175.vpr</td>
<td>C</td>
<td>FPGA Circuit Placement and Routing</td>
<td>HTML Text</td>
</tr>
<tr>
<td>176.gcc</td>
<td>C</td>
<td>C Programming Language Compiler</td>
<td>HTML Text</td>
</tr>
<tr>
<td>181.mcf</td>
<td>C</td>
<td>Combinatorial Optimization</td>
<td>HTML Text</td>
</tr>
<tr>
<td>186.crafty</td>
<td>C</td>
<td>Game Playing: Chess</td>
<td>HTML Text</td>
</tr>
<tr>
<td>197.parser</td>
<td>C</td>
<td>Word Processing</td>
<td>HTML Text</td>
</tr>
<tr>
<td>252.eon</td>
<td>C++</td>
<td>Computer Visualization</td>
<td>HTML Text</td>
</tr>
<tr>
<td>253.perl</td>
<td>C</td>
<td>PERL Programming Language</td>
<td>HTML Text</td>
</tr>
<tr>
<td>254.gss</td>
<td>C</td>
<td>Group Theory, Interpreter</td>
<td>HTML Text</td>
</tr>
<tr>
<td>255.vortex</td>
<td>C</td>
<td>Object-oriented Database</td>
<td>HTML Text</td>
</tr>
<tr>
<td>256.bz2</td>
<td>C</td>
<td>Compression</td>
<td>HTML Text</td>
</tr>
<tr>
<td>300.twolf</td>
<td>C</td>
<td>Place and Route Simulator</td>
<td>HTML Text</td>
</tr>
</tbody>
</table>
**CFP2000**

**CFP2000 (Floating Point Component of SPEC CPU2000):**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Language</th>
<th>Category</th>
<th>Full Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>108.xmpwise</td>
<td>Fortran 77</td>
<td>Physics / Quantum Chromodynamics</td>
<td>HTML Text</td>
</tr>
<tr>
<td>171.swim</td>
<td>Fortran 77</td>
<td>Shallow Water Modeling</td>
<td>HTML Text</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>Fortran 77</td>
<td>Multi-grid Solver: 3D Potential Field</td>
<td>HTML Text</td>
</tr>
<tr>
<td>173.appu</td>
<td>Fortran 77</td>
<td>Parabolic / Elliptic Partial Differential Equations</td>
<td>HTML Text</td>
</tr>
<tr>
<td>177.mesa</td>
<td>C</td>
<td>3-D Graphics Library</td>
<td>HTML Text</td>
</tr>
<tr>
<td>178.gadget</td>
<td>Fortran 90</td>
<td>Computational Fluid Dynamics</td>
<td>HTML Text</td>
</tr>
<tr>
<td>179.att</td>
<td>C</td>
<td>Image Recognition / Neural Networks</td>
<td>HTML Text</td>
</tr>
<tr>
<td>183.equake</td>
<td>C</td>
<td>Seismic Wave Propagation Simulation</td>
<td>HTML Text</td>
</tr>
<tr>
<td>187.facerec</td>
<td>Fortran 90</td>
<td>Image Processing: Face Recognition</td>
<td>HTML Text</td>
</tr>
<tr>
<td>188.annmp</td>
<td>C</td>
<td>Computational Chemistry</td>
<td>HTML Text</td>
</tr>
<tr>
<td>189.lucas</td>
<td>Fortran 90</td>
<td>Number Theory / Primality Testing</td>
<td>HTML Text</td>
</tr>
<tr>
<td>191.ima3d</td>
<td>Fortran 90</td>
<td>Finite-element Crash Simulation</td>
<td>HTML Text</td>
</tr>
<tr>
<td>200.sintrack</td>
<td>Fortran 77</td>
<td>High Energy Nuclear Physics Accelerator Design</td>
<td>HTML Text</td>
</tr>
<tr>
<td>301.apsi</td>
<td>Fortran 77</td>
<td>Meteorology: Pollutant Distribution</td>
<td>HTML Text</td>
</tr>
</tbody>
</table>

**Example for a CINT benchmark**

**256.bzip2**

**SPEC CPU2000 Benchmark Description File**

<table>
<thead>
<tr>
<th>Benchmark Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>256.bzip2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julian Seward <a href="mailto:jseward@acm.org">jseward@acm.org</a></td>
</tr>
</tbody>
</table>
Performance metrics

- Two fundamentally different metrics:
  - speed
  - rate (throughput)
- For each metric results for two different optimization level have to be provided
  - moderate optimization
  - aggressive optimization
  \[ 4 \text{ results for CINT2000} + 4 \text{ results for CFP2000} = 8 \text{ metrics} \]
- If taking the measurements of each application individually into account:
  \[ 2^2 \times (14+12) = 104 \text{ metrics} \]
Performance metrics (II)

- All results are relative to a reference system
- The final results is computed by using the geometric mean values

\[
\begin{align*}
\text{Speed: } & \quad SPE_{\text{int/fp}} = \sqrt[n]{\prod_{i=1}^{n} \left( \frac{t_{\text{ref}}^{i}}{t_{\text{run}}^{i}} \right) \times 100} \\
\text{Rate: } & \quad SPE_{\text{int/fp}} = \sqrt[n]{\prod_{i=1}^{n} \left( \frac{t_{\text{ref}}^{i}}{t_{\text{run}}^{i}} \right) \times 1.16 \times N}
\end{align*}
\]

with:
- \(n\): number of benchmarks in a suite
- \(t_{\text{ref}}^{i}/t_{\text{run}}^{i}\): execution time for benchmark \(i\) on the reference/test system
- \(N\): Number of simultaneous tasks

Reporting results

- SPEC produces a minimal set of representative numbers:
  - Reduces complexity to understand correlations
  - Easies comparison of different systems
  - Loss of information
- Results have to be compliant to the SPEC benchmarking rules in order to be approved as an official SPEC report
  - All components have to available at least 3 month after the publication (including a runtime environment for C/C++/Fortran applications)
  - Usage of SPEC tools for compiling and reporting
  - Each individual benchmark has to be executed at least three times
  - Verification of the benchmark output
  - A maximum of four optimization flags are allowed for the base run (including preprocessor and link directives)
  - Disclosure report containing all relevant data has to be available
### CFP2000 Result

<table>
<thead>
<tr>
<th>Model</th>
<th>CPU</th>
<th>Memory</th>
<th>OS</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acer Incorporated</td>
<td>SPECint 2000 = 1664</td>
<td>SPECint base 2000 = 1650</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>CPU</th>
<th>Memory</th>
<th>OS</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### CINT2000 Result

<table>
<thead>
<tr>
<th>Model</th>
<th>CPU</th>
<th>Memory</th>
<th>OS</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acer Incorporated</td>
<td>SPECint 2000 = 1807</td>
<td>SPECint base 2000 = 1806</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>CPU</th>
<th>Memory</th>
<th>OS</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes/Tuning Information

- Compiler: `-O2` is recommended for all `gcc` and `g++` compilers.
- Options: `-g` is recommended for debugging.
- Environment: `LD_LIBRARY_PATH` needs to be set properly.
- Libraries: `libgcc_s.so` is required for debugging.
- Flags: `-Werror` is recommended for all platforms.

**Standard Packages/Components**:
- GCC
- LDD
- LDconfig
- SED
- GDB
- Make

**Expert Settings**:
- `-O2 -g` for debugging.
- `-O2 -Wall -Werror` for maximum optimization.
- `-Wno-unused-parameter` to disable unused parameter warnings.
- `-Wno-unused-variable` to disable unused variable warnings.

**Tips**:
- Use `LD_PRELOAD` to load shared libraries before running the application.
- Use `valgrind` for memory profiling.
- Use `gprof` for profiling.

**Best Practices**:
- Use `autoconf` to automatically generate configuration files.
- Use `m4` for macro expansion.
- Use `yacc` for parsing.
- Use `lex` for lexical analysis.

**Troubleshooting**:
- Check `make` logs for build errors.
- Check `gdb` logs for debugging.
- Use `man` for detailed information on tools.

**Tools**:
- `man` for command line help.
- `info` for detailed documentation.
- `man pages` for extended documentation.
- `makeinfo` for generating documentation.

**Resources**:
- `gcc` manuals for detailed compiler options.
- `gdb` manuals for debugging.
- `valgrind` manuals for memory profiling.
- `gprof` manuals for profiling.

**Notes**:
- Ensure `gcc` version 4.8 or later is used.
- Use `g++` for C++ applications.
- Use `ldd` to check library dependencies.

**Components**:
- GCC
- LD
- LDconfig
- SED
- GDB
- Make
- Linker
- Debugger
- Compiler

**Expert Settings**:
- `-O2 -g` for debugging.
- `-O2 -Wall -Werror` for maximum optimization.
- `-Wno-unused-parameter` to disable unused parameter warnings.
- `-Wno-unused-variable` to disable unused variable warnings.

**Tips**:
- Use `LD_PRELOAD` to load shared libraries before running the application.
- Use `valgrind` for memory profiling.
- Use `gprof` for profiling.

**Best Practices**:
- Use `autoconf` to automatically generate configuration files.
- Use `m4` for macro expansion.
- Use `yacc` for parsing.
- Use `lex` for lexical analysis.

**Troubleshooting**:
- Check `make` logs for build errors.
- Check `gdb` logs for debugging.
- Use `man` for detailed information on tools.

**Tools**:
- `man` for command line help.
- `info` for detailed documentation.
- `man pages` for extended documentation.
- `makeinfo` for generating documentation.

**Resources**:
- `gcc` manuals for detailed compiler options.
- `gdb` manuals for debugging.
- `valgrind` manuals for memory profiling.
- `gprof` manuals for profiling.

**Notes**:
- Ensure `gcc` version 4.8 or later is used.
- Use `g++` for C++ applications.
- Use `ldd` to check library dependencies.