**Program design and analysis**

- Design patterns
- Representations of programs
- Assembly and linking

**Design patterns**

- Design pattern: generalized description of the design of a certain type of program.
- Designer fills in details to customize the pattern to a particular programming problem.

**List design pattern**

- Class diagram
- State diagrams
- Sequence diagrams
- etc.

**Design pattern elements**

- State machine is useful in many contexts:
  - parsing user input
  - responding to complex stimuli
  - controlling sequential outputs

**State machine example**

- State machine diagram with states and transitions.
State machine pattern

- State machine
- State
- Output step (input)

Circular buffer pattern

- Circular buffer
- Init()
- Add(data)
- Data head()
- Data element (index)

Circular buffer

- X1 X2 X3 X4 X5 X6
- T1 T2 T3
- Data stream
- X5 X6 X7 X4

Models of programs

- Source code is not a good representation for programs:
  - Clumsy;
  - Leaves much information implicit.
- Compilers derive intermediate representations to manipulate and optimize the program.

Data flow graph

- DFG: data flow graph.
- Does not represent control.
- Models basic block: code with one entry and exit.
- Describes the minimal ordering requirements on operations.

Single assignment form

- \( x = a + b; \)
- \( y = c - d; \)
- \( z = x \ast y; \)
- \( y = b + d; \)
- \( y_1 = b + d; \)

- Original basic block
- Single assignment form
Data flow graph

\[
x = a + b; \\
y = c - d; \\
z = x \times y; \\
y_1 = b + d;
\]

single assignment form

Control-data flow graph

- **CDFG**: represents control and data.
- Uses data flow graphs as components.
- Two types of nodes:
  - decision;
  - data flow.

Data flow node

Encapsulates a data flow graph:

\[
x = a + b; \\
y = c + d;
\]

Write operations in basic block form for simplicity.

Control

Equivalent forms

CDFG example

```
if (cond1) bb1();
else bb2();
bb3();
switch (test1) {
  case c1: bb4(); break;
  case c2: bb5(); break;
  case c3: bb6(); break;
}
```
**for loop**

for (i=0; i<N; i++)
    loop_body();

```
for loop
i=0;
while (i<N) {
    loop_body(); i++; }
equivalent
```

**Assembly and linking**

- Last steps in compilation:
  - HLL compile
  - assembly
  - assemble
  - load
  - executable link

**Multiple-module programs**

- Programs may be composed from several files.
- Addresses become more specific during processing:
  - Relative addresses are measured relative to the start of a module;
  - Absolute addresses are measured relative to the start of the CPU address space.

**Assemblers**

- Major tasks:
  - Generate binary for symbolic instructions;
  - Translate labels into addresses;
  - Handle pseudo-ops (data, etc.).
- Generally one-to-one translation.

**Assembly labels**

```
ORG 100
label1  ADR r4,c
```

**Symbol table**

```
ADD r0,r1,r2   xx  0x8
xx ADD r3,r4,r5 yy  0x10
CMP r0,r3
yy SUB r5,r6,r7
assembly code  symbol table
```

**Symbol table generation**

- Use program location counter (PLC) to determine address of each location.
- Scan program, keeping count of PLC.
- Addresses are generated at assembly time, not execution time.
Symbol table example

\[
\begin{array}{llll}
\text{D r0},r1,r2 & xx & 0x8 \\
\text{D r3},r4,r5 & yy & 0xa \\
\text{P r0},r3 & \\
\text{B r5},r6,r7 & \\
\end{array}
\]

Two-pass assembly

- **Pass 1:**
  - generate symbol table
- **Pass 2:**
  - generate binary instructions

Relative address generation

- Some label values may not be known at assembly time.
- Labels within the module may be kept in relative form.
- Must keep track of external labels---can’t generate full binary for instructions that use external labels.

Pseudo-operations

- Pseudo-ops do not generate instructions:
  - **ORG** sets program location.
  - **EQU** generates symbol table entry without advancing PLC.
  - **Data statements** define data blocks.

Linking

- Combines several object modules into a single executable module.
- Jobs:
  - put modules in order;
  - resolve labels across modules.

Externals and entry points
Module ordering

- Code modules must be placed in absolute positions in the memory space.
- Load map or linker flags control the order of modules.

| module1 | module2 | module3 |

Dynamic linking

- Some operating systems link modules dynamically at run time:
  - shares one copy of library among all executing programs;
  - allows programs to be updated with new versions of libraries.

Program design and analysis

- Compilation flow.
- Basic statement translation.
- Basic optimizations.
- Interpreters and just-in-time compilers.

Compilation

- Compilation strategy (Wirth):
  - compilation = translation + optimization
- Compiler determines quality of code:
  - use of CPU resources;
  - memory access scheduling;
  - code size.

Basic compilation phases

- HLL
  - parsing, symbol table
  - machine-independent optimizations
  - machine-dependent optimizations
  - assembly

Statement translation and optimization

- Source code is translated into intermediate form such as CDFG.
- CDFG is transformed/optimized.
- CDFG is translated into instructions with optimization decisions.
- Instructions are further optimized.
Arithmetic expressions

\[ a \times b + 5 \times (c - d) \]

Expression

Control code generation

if \((a + b > 0)\)
  \(x = 5;\)
else
  \(x = 7;\)

Procedure linkage

- Need code to:
  - call and return;
  - pass parameters and results.
- Parameters and returns are passed on stack.
  - Procedures with few parameters may use registers.

Procedure stacks

```
growth
```

```
proc1

proc1(int a) {
  proc2(5);
}
```

FP frame pointer

SP stack pointer

accessed relative to SP
**ARM procedure linkage**

- **APCS (ARM Procedure Call Standard):**
  - r0-r3 pass parameters into procedure. Extra parameters are put on stack frame.
  - r0 holds return value.
  - r4-r7 hold register values.
  - r11 is frame pointer, r13 is stack pointer.
  - r10 holds limiting address on stack size to check for stack overflows.

**Data structures**

- Different types of data structures use different data layouts.
- Some offsets into data structure can be computed at compile time, others must be computed at run time.

**One-dimensional arrays**

- C array name points to 0th element:
  - `a` → `a[0]` → *(a + 1)*

**Two-dimensional arrays**

- Column-major layout:
  - `a[0,0]` → `a[0,1]` → `...` → `a[1,0]` → `a[1,1]` → *(a*M+j)*

**Structures**

- Fields within structures are static offsets:
  - `struct`  
    - `int field1;`  
    - `char field2;`  
    - `mystruct;`  
  - `struct mystruct a, *aptr = &a;`

**Expression simplification**

- Constant folding:
  - `8+1 = 9`
- Algebraic:
  - `a*b + a*c = a*(b+c)`
- Strength reduction:
  - `a*2 = a<<1`
**Dead code elimination**

- Dead code:
  ```c
  #define DEBUG 0
  if (DEBUG) dbg(p1);
  ```
- Can be eliminated by analysis of control flow, constant folding.

**Procedure inlining**

- Eliminates procedure linkage overhead:
  ```c
  int foo(a,b,c) { return a + b - c;}
  z = foo(w,x,y);
  ⇒
  z = w + x - y;
  ```

**Loop transformations**

- Goals:
  - reduce loop overhead;
  - increase opportunities for pipelining;
  - improve memory system performance.

**Loop unrolling**

- Reduces loop overhead, enables some other optimizations.
  ```c
  for (i=0; i<4; i++)
  a[i] = b[i] * c[i];
  ⇒
  for (i=0; i<2; i++) {
    a[i*2] = b[i*2] * c[i*2];
    a[i*2+1] = b[i*2+1] * c[i*2+1];
  }
  ```

**Loop fusion and distribution**

- Fusion combines two loops into 1:
  ```c
  for (i=0; i<N; i++)
  a[i] = b[i] * 5;
  for (j=0; j<N; j++)
  w[i] = c[i] * d[i];
  ⇒
  for (i=0; i<N; i++) {
    a[i] = b[i] * 5; w[i] = c[i] * d[i];
  }
  ```
- Distribution breaks one loop into two.
  - Changes optimizations within loop body.

**Loop tiling**

- Breaks one loop into a nest of loops.
  - Changes order of accesses within array.
    - Changes cache behavior.
Loop tiling example

for (i=0; i<N; i++)
for (j=0; j<N; j++)
c[i] = a[i][j]*b[j];

Array padding

- Add array elements to change mapping into cache:

 before

\[ \begin{array}{c}
  a[0,0] \\
  a[0,1] \\
  a[0,2] \\
  a[1,0] \\
  a[1,1] \\
  a[1,2] \\
\end{array} \]

 after

\[ \begin{array}{c}
  a[0,2] \\
  a[1,2] \\
\end{array} \]

Register allocation

- Goals:
  - Choose register to hold each variable;
  - Determine lifespan of variable in the register.
- Basic case: within basic block.

Register lifetime graph

w = a + b;  \quad t=1
x = c + w;  \quad t=2
y = c + d;  \quad t=3

Instruction scheduling

- Non-pipelined machines do not need instruction scheduling: any order of instructions that satisfies data dependencies runs equally fast.
- In pipelined machines, execution time of one instruction depends on the nearby instructions: opcode, operands.

Reservation table

- A reservation table relates instructions/time to CPU resources.

<table>
<thead>
<tr>
<th>Time/instr</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>instr2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>instr3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>instr4</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Software pipelining

- Schedules instructions across loop iterations.
- Reduces instruction latency in iteration \(i\) by inserting instructions from iteration \(i-1\).

Software pipelining in SHARC

- Example:
  for \((i=0; i<N; i++)\)
  \[\text{sum } += a[i]*b[i];\]
- Combine three iterations:
  - Fetch array elements \(a, b\) for iteration \(i\).
  - Multiply \(a, b\) for iteration \(i-1\).
  - Compute sum for iteration \(i-2\).

Software pipelining in SHARC, cont’d

```c
/* first iteration performed outside loop */
ai=a[0]; bi=b[0]; p=ai*bi; ai=a[1]; bi=b[1];
/* initiate loads used in second iteration; remaining loads will be performed inside the loop */
for (i=2; i<N-2; i++) {
  sum += p; /* make sum using p from last iteration */
  p = ai*bi; /* multiply for next iteration’s sum */
  ai=a[i]; bi=b[i]; /* fetch for next cycle’s multiply */
  sum += p; /* fetch for next cycle’s multiply */
}
sum += p;
```

Software pipelining timing

<table>
<thead>
<tr>
<th>time</th>
<th>ai=a[i]; bi=b[i];</th>
<th>p = ai*bi;</th>
<th>ai=a[i]; bi=b[i];</th>
<th>p = ai*bi;</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum += p;</td>
<td>sum += p;</td>
<td>sum += p;</td>
<td>sum += p;</td>
<td></td>
</tr>
</tbody>
</table>

Iteration i-2 | Iteration i-1 | Iteration i

Instruction selection

- May be several ways to implement an operation or sequence of operations.
- Represent operations as graphs, match possible instruction sequences onto graph.

Using your compiler

- Understand various optimization levels (-O1, -O2, etc.)
- Look at mixed compiler/assembler output.
- Modifying compiler output requires care:
  - correctness;
  - loss of hand-tweaked code.
**Interpreters and JIT compilers**

- **Interpreter**: translates and executes program statements on-the-fly.
- **JIT compiler**: compiles small sections of code into instructions during program execution.
  - Eliminates some translation overhead.
  - Often requires more memory.

**Program design and analysis**

- Optimizing for execution time.
- Optimizing for energy/power.
- Optimizing for program size.

**Motivation**

- Embedded systems must often meet deadlines.
  - Faster may not be fast enough.
- Need to be able to analyze execution time.
  - Worst-case, not typical.
- Need techniques for reliably improving execution time.

**Run times will vary**

- Program execution times depend on several factors:
  - Input data values.
  - State of the instruction, data caches.
  - Pipelining effects.

**Measuring program speed**

- CPU simulator.
  - I/O may be hard.
  - May not be totally accurate.
- Hardware timer.
  - Requires board, instrumented program.
- Logic analyzer.
  - Limited logic analyzer memory depth.

**Program performance metrics**

- **Average-case**:
  - For typical data values, whatever they are.
- **Worst-case**:
  - For any possible input set.
- **Best-case**:
  - For any possible input set.
- Too-fast programs may cause critical races at system level.
What data values?

What values create worst/average/best case behavior?
- analysis;
- experimentation.

Concerns:
- operations;
- program paths.

Performance analysis

Elements of program performance (Shaw):
- execution time = program path + instruction timing
- Path depends on data values. Choose which case you are interested in.
- Instruction timing depends on pipelining, cache behavior.

Program paths

Consider for loop:

```c
for (i=0, f=0, i<N; i++)
    f = f + c[i]*x[i];
```

- Loop initiation block executed once.
- Loop test executed N+1 times.
- Loop body and variable update executed N times.

Instruction timing

Not all instructions take the same amount of time.
- Hard to get execution time data for instructions.
- Instruction execution times are not independent.
- Execution time may depend on operand values.

Trace-driven performance analysis

- Trace: a record of the execution path of a program.
- Trace gives execution path for performance analysis.
- A useful trace:
  - requires proper input values;
  - is large (gigabytes).
Trace generation
- Hardware capture:
  - logic analyzer;
  - hardware assist in CPU.
- Software:
  - PC sampling.
  - Instrumentation instructions.
  - Simulation.

Loop optimizations
- Loops are good targets for optimization.
- Basic loop optimizations:
  - code motion;
  - induction-variable elimination;
  - strength reduction (x*2 -> x<<1).

Code motion
```
for (i=0; i<N*M; i++)
  z[i] = a[i] + b[i];
```

Induction variable elimination
- Induction variable: loop index.
- Consider loop:
  ```
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      z[i][j] = b[i][j];
  ```
- Rather than recompute i*M+j for each array in each iteration, share induction variable between arrays, increment at end of loop body.

Cache analysis
- Loop nest: set of loops, one inside other.
- Perfect loop nest: no conditionals in nest.
- Because loops use large quantities of data, cache conflicts are common.

Array conflicts in cache
```
main memory
```
Array conflicts, cont’d.

- Array elements conflict because they are in the same line, even if not mapped to same location.
- Solutions:
  1. move one array;
  2. pad array.

Performance optimization hints

- Use registers efficiently.
- Use page mode memory accesses.
- Analyze cache behavior:
  1. instruction conflicts can be handled by rewriting code, rescheduling;
  2. conflicting scalar data can easily be moved;
  3. conflicting array data can be moved, padded.

Energy/power optimization

- Energy: ability to do work.
  1. Most important in battery-powered systems.
- Power: energy per unit time.
  1. Important even in wall-plug systems—power becomes heat.

Measuring energy consumption

- Execute a small loop, measure current:
  ```c
  while (TRUE) a();
  ```

Sources of energy consumption

- Relative energy per operation (Catthoor et al):
  1. memory transfer: 33
  2. external I/O: 10
  3. SRAM write: 9
  4. SRAM read: 4.4
  5. multiply: 3.6
  6. add: 1

Cache behavior is important

- Energy consumption has a sweet spot as cache size changes:
  1. cache too small: program thrashes, burning energy on external memory accesses;
  2. cache too large: cache itself burns too much power.
Optimizing for energy

- First-order optimization:
  - high performance = low energy.
- Not many instructions trade speed for energy.

Optimizing for energy, cont'd.

- Use registers efficiently.
- Identify and eliminate cache conflicts.
- Moderate loop unrolling eliminates some loop overhead instructions.
- Eliminate pipeline stalls.
- Inlining procedures may help: reduces linkage, but may increase cache thrashing.

Optimizing for program size

- Goal:
  - reduce hardware cost of memory;
  - reduce power consumption of memory units.
- Two opportunities:
  - data;
  - instructions.

Data size minimization

- Reuse constants, variables, data buffers in different parts of code.
  - Requires careful verification of correctness.
- Generate data using instructions.

Reducing code size

- Avoid function inlining.
- Choose CPU with compact instructions.
- Use specialized instructions where possible.

Code compression

- Use statistical compression to reduce code size, decompress on-the-fly:
Program design and analysis

- Program validation and testing.

Goals

- Make sure software works as intended.
  - We will concentrate on functional testing---performance testing is harder.
- What tests are required to adequately test the program?
  - What is "adequate"?

Basic testing procedure

- Provide the program with inputs.
- Execute the program.
- Compare the outputs to expected results.

Types of software testing

- Black-box: tests are generated without knowledge of program internals.
- Clear-box (white-box): tests are generated from the program structure.

Clear-box testing

- Generate tests based on the structure of the program.
  - Is a given block of code executed when we think it should be executed?
  - Does a variable receive the value we think it should get?

Controllability and observability

- Controllability: must be able to cause a particular internal condition to occur.
- Observability: must be able to see the effects of a state from the outside.
Example: FIR filter

- Code:
  
  ```
  for (firout = 0.0, j =0; j < N; j++)
    firout += buff[j] * c[j];
  if (firout > 100.0) firout = 100.0;
  if (firout < -100.0) firout = -100.0;
  ```

- Controllability: to test range checks for `firout`, must first load circular buffer.
- Observability: how do we observe values of `buff`, `firout`?

Path-based testing

- Clear-box testing generally tests selected program paths:
  1. control program to exercise a path;
  2. observe program to determine if path was properly executed.
- May look at whether location on path was reached (control), whether variable on path was set (data).

Example: choosing paths

- Two possible criteria for selecting a set of paths:
  1. Execute every statement at least once.
  2. Execute every direction of a branch at least once.
- Equivalent for structured programs, but not for programs with `gos`.

Branch testing strategy

- Exercise the elements of a conditional, not just one true and one false case.
- Devise a test for every simple condition in a Boolean expression.

Loop testing

- Common, specialized structure---specialized tests can help.
- Useful test cases:
  1. skip loop entirely;
  2. one iteration;
  3. two iterations;
  4. mid-range of iterations;
  5. n-1, n, n+1 iterations.
Black-box testing

- Black-box tests are made from the specifications, not the code.
- Black-box testing complements clear-box.
  - May test unusual cases better.

Types of black-box tests

- Specified inputs/outputs:
  - select inputs from spec, determine required outputs.

- Random:
  - generate random tests, determine appropriate output.

- Regression:
  - tests used in previous versions of system.

Evaluating tests

- How good are your tests?
  - Keep track of bugs found, compare to historical trends.
  - Error injection: add bugs to copy of code, run tests on modified code.