Using your compiler

- Understand various optimization levels (-O1, -O2, etc.)
- **armcc**
  - **-Ospace** perform optimisations to reduce image size at the expense of increased execution time.
  - **-Otime** perform optimisations to reduce execution time at the expense of a larger image.

Gcc optimization levels

- In order to control compilation-time and compiler memory usage, and the trade-offs between speed and space for the resulting executable, GCC provides a range of general optimization levels.
- chosen with the command line option -O*LEVEL*, where *LEVEL* is a number from 0 to 3.
- -O0 or no -O option (default) : GCC does not perform any optimization and compiles the source code in the most straightforward way possible. This is the best option to use when debugging a program and is the default if no optimization level option is specified.

Gcc –O1

- Turns on the most common forms of optimization that do not require any speed-space tradeoffs.
- With this option the resulting executables should be smaller and faster than with -O0.
- The more expensive optimizations, such as instruction scheduling, are not used at this level.
- Compiling with the option -O1 can often take less time than compiling with -O0, due to the reduced amounts of data that need to be processed after simple optimizations.

Gcc –O2

- Turns on further optimizations, in addition to those used by -O1, which include instruction scheduling.
- Only optimizations that do not require any speed-space tradeoffs are used, so the executable should not increase in size.
- Take longer to compile programs and require more memory than with -O1.
- Best choice for deployment of a program, because it provides maximum optimization without increasing the executable size.
- It is the default optimization level for releases of GNU packages.

Gcc –O3

- Turns on more expensive optimizations, such as function inlining, in addition to all the optimizations of the lower levels -O2 and -O1.
- May increase the speed of the resulting executable, but can also increase its size.
- Under some circumstances where these optimizations are not favorable, this option might actually make a program slower.

Gcc –funroll-loops

- Turns on loop-unrolling, and is independent of the other optimization options.
- It will increase the size of an executable.
- Whether or not this option produces a beneficial result has to be examined on a case-by-case basis.

Gcc -Os

- -Os :This option selects optimizations which reduce the size of an executable. The aim of this option is to produce the smallest possible executable, for systems constrained by memory or disk space.
- In some cases a smaller executable will also run faster, due to better cache usage.

Using Gcc

- It is important to remember that the benefit of optimization at the highest levels must be weighed against the cost. The cost of optimization includes greater complexity in debugging, and increased time and memory requirements during compilation. For most purposes it is satisfactory to use -O0 for debugging, and -O2 for development and deployment.

Interpreters

- Interpreter: translates and executes program statements on-the-fly.

JIT compilation

- Also known as dynamic translation for improving the runtime performance
- JIT builds upon two ideas in run-time environments
  - Bytecode compilation
  - Dynamic compilation
- It converts code at runtime prior to executing it natively, for example bytecode into native machine code.
  - Source code -> bytecode -> native code
- A VM interprets the bytecode
- A JIT compiler can be used to speed up execution of bytecode.
  - Startup delay
  - Extra memory required

Java

- JVM
- JIT (just-in-time) compiler: 처음 본 것은 compile
- AOT (ahead-of-time) complier: 어플리케이션 다운로드후 모든 코드 컴파일
- DAC (dynamic adaptive compiler): advanced JIT, a combination of JIT and bytecode interpreter
  - JIT based on the profiled results of interpretation
  - 실제는 Profiling 하지 않고 모든 method들을 compile하기 때문에 overhead가 클 수 있음

Javascript (JS)

- A script language executed in web browser to reduce the load of the server
  - Dynamically typed
  - Prototype based
  - First-class functions with inner functions and closures

Javascript (JS)

- A script language executed in web browser to reduce the load of the server
- Dynamically typed
  - Variable type이 runtime에 결정
  - Object filed와 function을 arrray 형태로 접근
- Prototype based (class 사용하는 않는 oop)
  - Class에 비해 runtime에 변형 대치 쉬움
- Support first-class functions
  - Closure를 활용하여 지원
  - Function을 runtime에 생성, data structure에 저장 다른 function의 argument 또는 return 값으로 전달

First-class

- An object is first-class in the context of a particular programming language if it
  - Can be stored in variables and data structures
  - Can be passed as a parameter to a subroutine
  - Can be return as the result of a subroutine
  - Can be constructed at runtime
  - Has intrinsic identity (independent of any given name)

Closure

- A closure is the local variables for a function kept alive after the function has returned
- A closure is a stack frame is not deallocated when the function is returned as if a stack frame were malloc’ed instead of being on the stack.
- In Javascript, you can think of a function reference variable as having a pointer to the function as well as a hidden pointer to a closure of the function.

Javascript (JS)

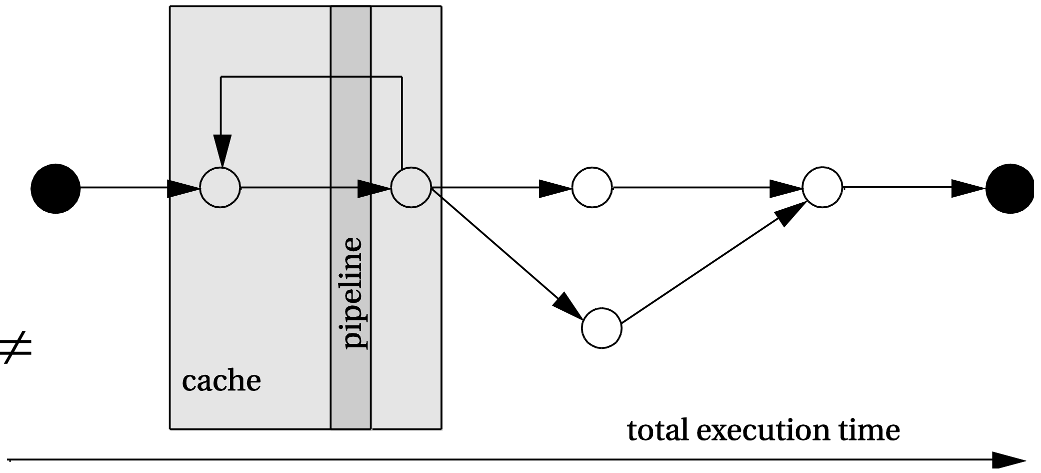
- Dynamic language
- JS bytecode : 121 개
  - 복잡하여 항상 handler function (CTI)로 처리하는 것: 50개
  - 간단한 data type 경우는 native code 생성, 복잡한 data type 경우는 handler function을 이용하는 것: 54개
  - 항상 native code 생성하는 것: 17개
- Javascript: script executed in web browser

5.6 Program level performance analysis

- Optimizing for:
  - Execution time.
  - Energy/power.
  - Program size.
- Program validation and testing.

Program-level performance analysis

- Need to understand performance in detail:
  - Real-time behavior, not just typical.
  - On complex platforms.
- Program performance ≠ CPU performance:
  - Pipeline, cache are windows into program.
  - We must analyze the entire program.



Complexities of program performance

- Varies with input data:
  - Different-length paths.
- Cache effects.
- Instruction-level performance variations:
  - Pipeline interlocks.
  - Fetch times.

How to measure program performance

- Simulate execution of the CPU.
  - Makes CPU state visible.
- Measure on real CPU using timer.
  - Requires modifying the program to control the timer.
- Measure on real CPU using logic analyzer.
  - Requires events visible on the pins.
- Performance analysis and monitoring using hardware counters

Program performance metrics

- Average-case execution time.
  - Typically used in application programming.
- Worst-case execution time.
  - A component in deadline satisfaction.
- Best-case execution time.
  - Task-level interactions can cause best-case program behavior to result in worst-case system behavior.

Elements of program performance

- Basic program execution time formula:
  - execution time = program path + instruction timing
- Program path: a sequence of instruction executed by the program
- Instruction timing: determined based on the sequence of instructions traced by the program path, which takes into account data dependencies, pipeline behavior, and caching

Data path + instruction timing

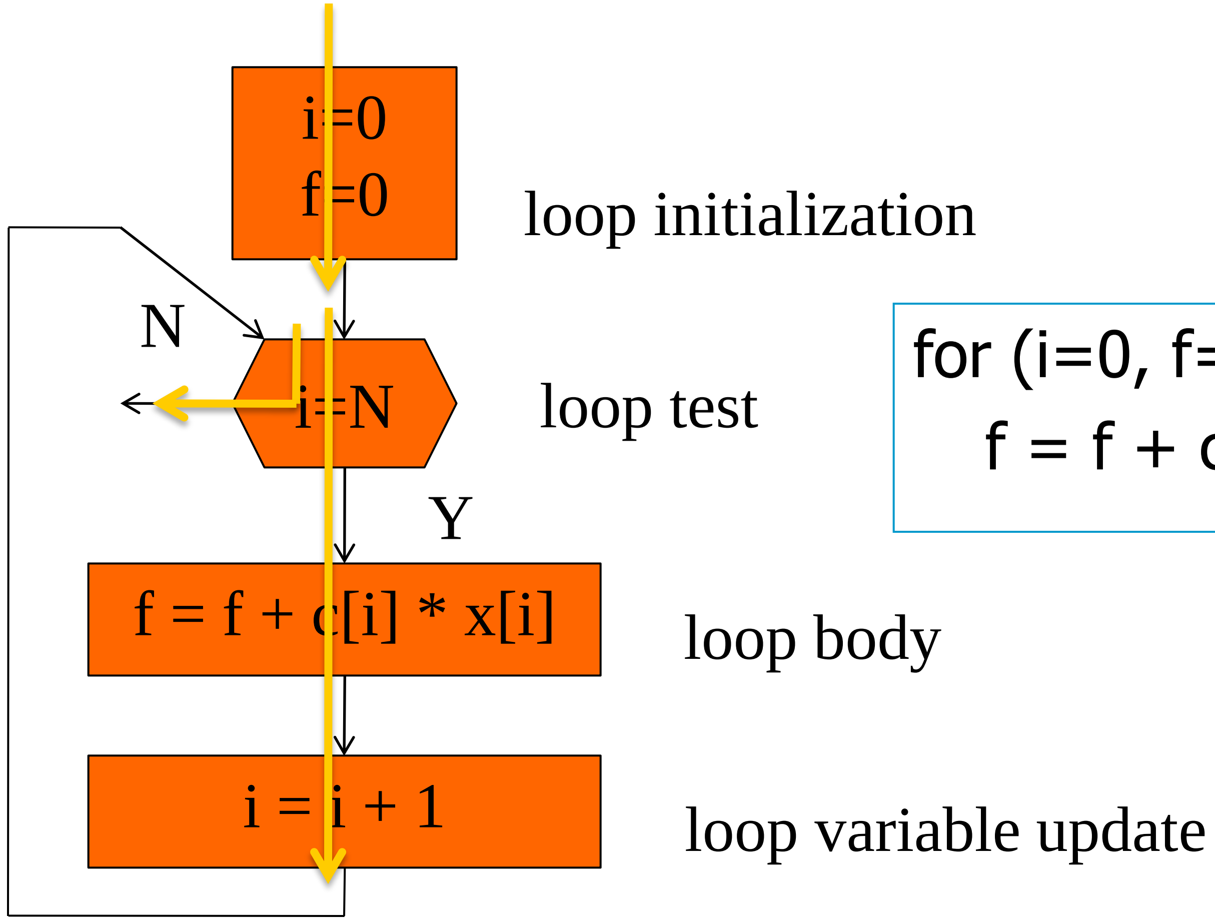
- Solving these two problems independently helps simplify analysis.
  - Easier to separate on simpler CPUs.
- Accurate performance analysis requires:
  - Assembly/binary code.
  - Execution platform.

Data-dependent paths in an if statement

```
if (a || b) { /* T1 */
    if ( c ) /* T2 */
        x = r*s+t; /* A1 */
    else y=r+s; /* A2 */
    z = r+s+u; /* A3 */
    }
else {
    if ( c ) /* T3 */
        y = r-t; /* A4 */
}
```

a	b	c	path
0	0	0	T1=F, T3=F: no assignments
0	0	1	T1=F, T3=T: A4
0	1	0	T1=T, T2=F: A2, A3
0	1	1	T1=T, T2=T: A1, A3
1	0	0	T1=T, T2=F: A2, A3
1	0	1	T1=T, T2=T: A1, A3
1	1	0	T1=T, T2=F: A2, A3
1	1	1	T1=T, T2=T: A1, A3

Paths in a loop



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

Instruction timing

- Not all instructions take the same amount of time.
  - Multi-cycle instructions.
  - Fetches.
- Execution times of instructions are not independent.
  - Pipeline interlocks.
  - Cache effects.
- Execution times may vary with operand value.
  - Floating-point operations.
  - Some multi-cycle integer operations.

Mesaurement-driven performance analysis

- Not so easy as it sounds
- Drawbacks
  - Must actually have access to the CPU.
  - Must know data inputs that give worst/best case performance.
  - Must make state visible.
- Need CPU or its simulator
- Still an important method for performance analysis.

Input: Feeding the program

- Need to know the desired input values.
- May need to write software scaffolding
  - that generates the input values and examines the outputs to generate feedback-driven inputs.
- Performance can be measured directly on the hardware or by using a simulator.

Trace-driven measurement

- Trace-driven:
  - Instrument the program.
  - Save information about the path.
- Requires modifying the program.
- Trace files are large.
- Widely used for cache analysis.

Physical measurement

- In-circuit emulator allows tracing.
  - Affects execution timing.
- Logic analyzer can measure behavior at pins.
  - Address bus can be analyzed to look for events.
  - Code can be modified to make events visible.
- Particularly important for real-world input streams.

CPU simulation

- Some simulators are less accurate.
- Cycle-accurate simulator provides accurate clock-cycle timing.
  - Simulator models CPU internals.
  - Simulator writer must know how CPU works.

SimpleScalar FIR filter simulation

```
int x[N] = {8, 17, … };
int c[N] = {1, 2, … };
main() {
    int i, k, f;
    for (k=0; k<COUNT; k++)
        for (i=0, f=0 ; i<N; i++)
            f += c[i]*x[i];
}
```

N	total sim cycles	sim cycles per filter execution
100	25854	259
1,000	155759	156
1,0000	1451840	145

Loop set up: 1
Loop body: N
Loop test: N+1

Performance optimization 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.

Programs and performance analysis

- Best results come from analyzing optimized instructions, not high-level language code:
  - non-obvious translations of HLL statements into instructions;
  - code may move;
  - cache effects are hard to predict.

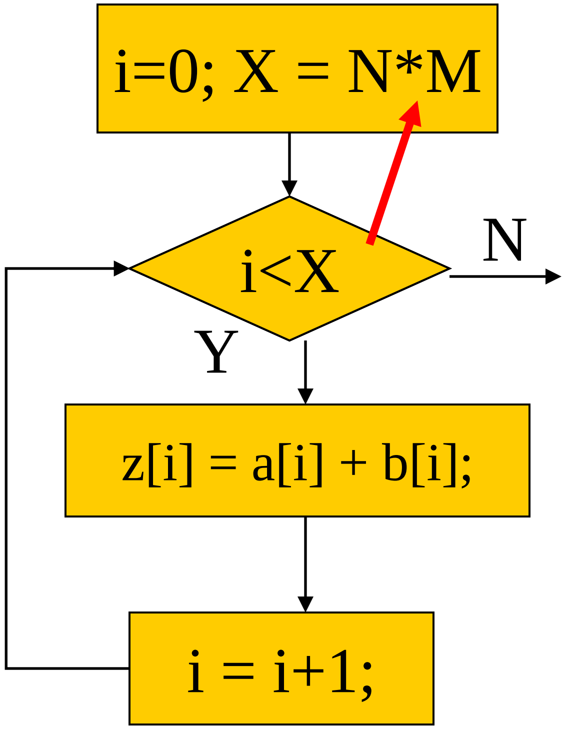
Loop optimizations

- Loops are good targets for optimization
  - Why?
- 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];
```

Performed (NM-1) times



Induction variable elimination

- Induction variable: its value is derived form the 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.

Strength reduction

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      zbinduct = i*M + j;
      *(zptr + zbinduct) = *(bptr + zbinduct);
```

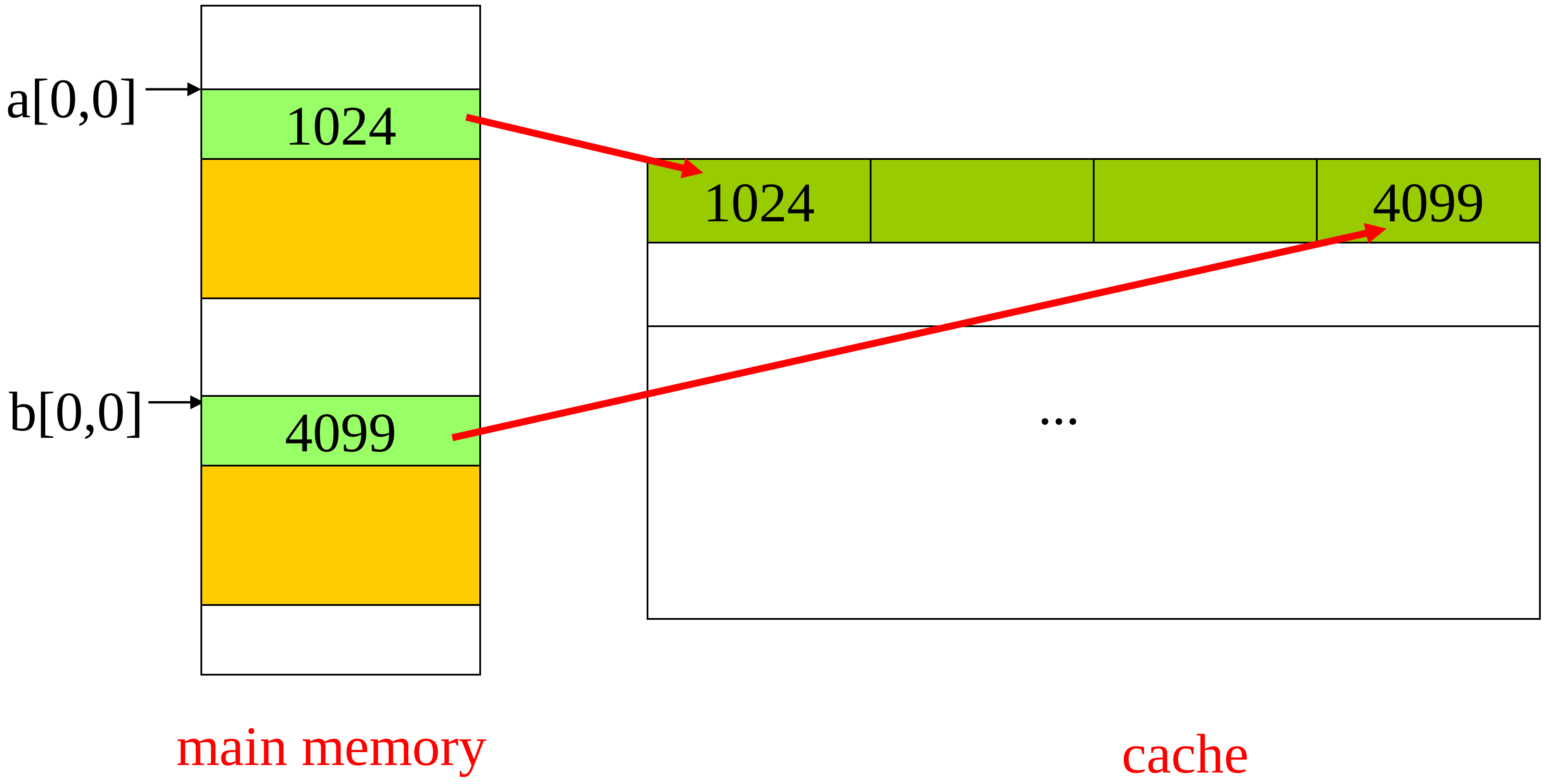
- Better code with strength reduction

```
xbinduct = 0;
for (i=0; i<N; i++)
    for (j=0; j<M; j++) {
       *(zptr + zbinduct) = *(bptr + zbinduct);
      zbinduct++;
    }
}
```

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



Array conflicts, cont'd.

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

Performance optimization hints

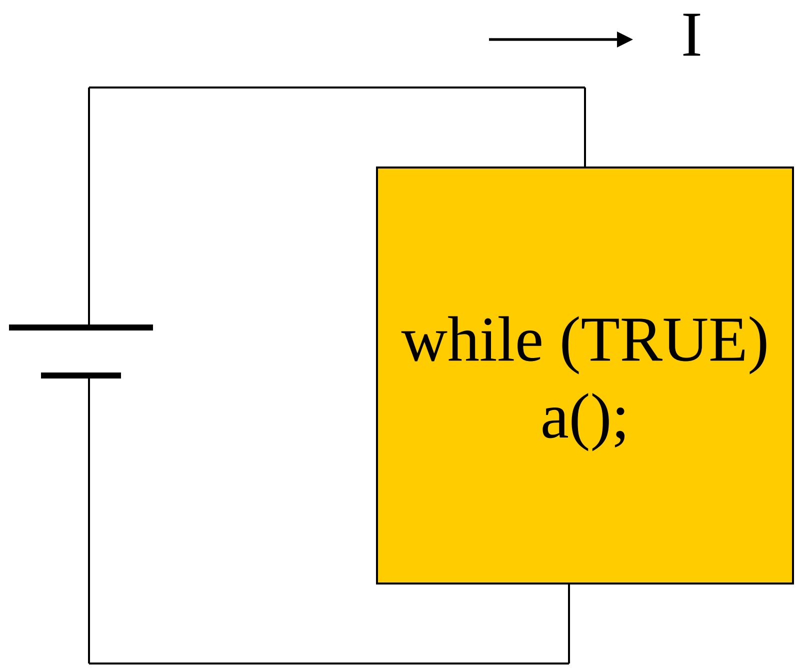
- Use registers efficiently.
- Use page mode memory accesses.
- Analyze cache behavior:
  - instruction conflicts can be handled by rewriting code, rescheudling;
  - conflicting scalar data can easily be moved;
  - conflicting array data can be moved, padded.

Energy/power optimization

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

Measuring energy consumption

- Execute a small loop, measure current:



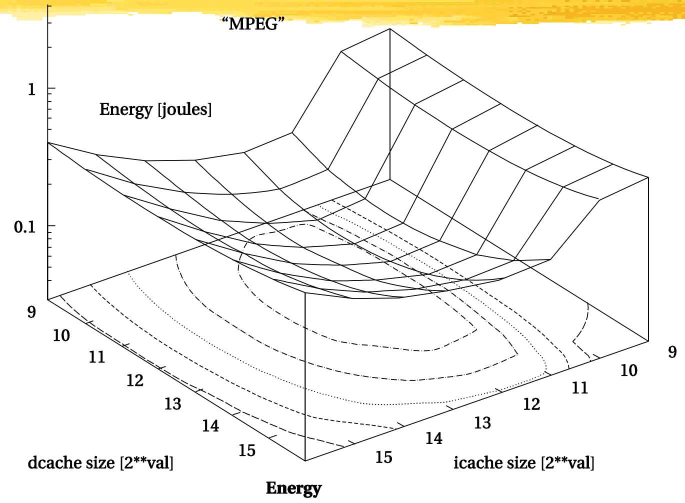
Sources of energy consumption

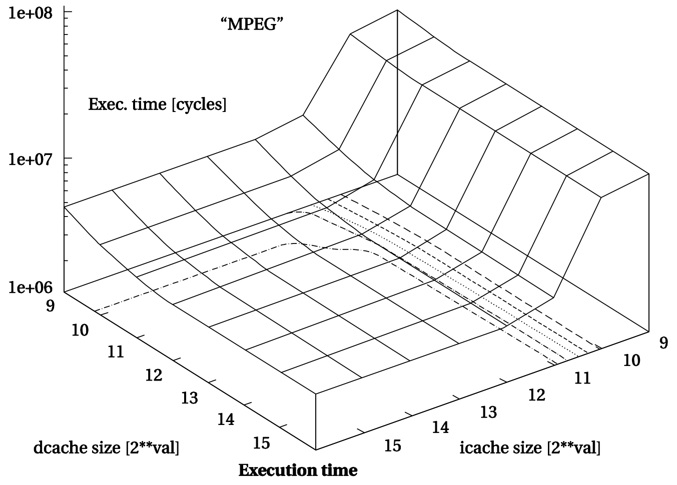
- Relative energy per operation (Catthoor et al):
  - memory transfer: 33
  - external I/O: 10
  - SRAM write: 9
  - SRAM read: 4.4
  - multiply: 3.6
  - add: 1

Cache behavior is important

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

Cache sweet spot





[Li98] © 1998 IEEE

Optimizing for energy

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

Optimizing for energy

- 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.

Efficient loops

- General rules:
  - Don’t use function calls.
  - Use unsigned integer for loop counter.
  - Use <= to test loop counter.
  - Make use of compiler---global optimization, software pipelining.

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.

Program validation and testing

- Validation
  - Does it work as it is intended?
- Test
  - Concentrate here on functional verification.
- Major testing strategies:
  - Black box doesn’t look at the source code.
  - Clear box (white box) does look at the source code.

Clear-box testing

- Examine the source code to determine whether it works:
  - Can you actually exercise a path?
  - Do you get the value you expect along a path?
- Testing procedure of 3 steps:
  - Controllability: provide program with inputs.
  - Execute.
  - Observability: examine outputs.

FIR filter with limiter

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

- What if we want to test whether the limiting code works?
- Controllability problem:
  - Must fill circular buffer with desired N values.

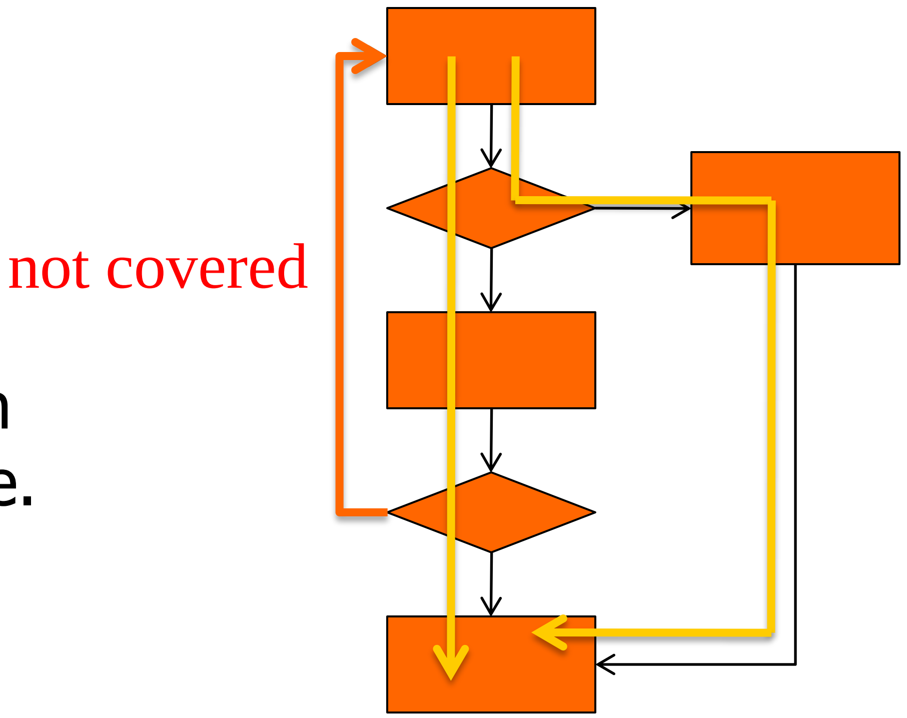
- What if we want to test the FIR filter itself?
- Observability problem:
  - Want to examine firout by setting a breakpoint before limit testing.

Execution paths and testing

- Paths are important in functional testing as well as performance analysis.
- In general, an exponential number of paths through the program.
  - Show that some paths dominate others.
  - Heuristically limit paths.

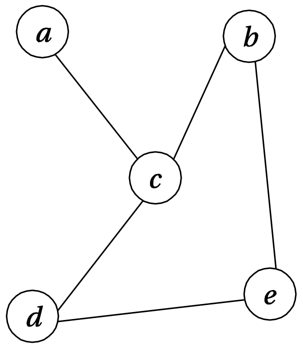
Choosing the paths to test

- Possible criteria:
  - Execute every statement at least once.
  - Execute every branch direction at least once.
- Equivalent for structured programs.
- Not true for gotos.



Basis paths

- Approximate CDFG with undirected graph.
- Undirected graphs have basis paths:
  - All paths are linear combinations of basis paths.



Graph

$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 1 \\ c & 1 & 1 & 0 & 1 & 0 \\ d & 0 & 0 & 1 & 0 & 1 \\ e & 0 & 1 & 0 & 1 & 0 \end{array}$$

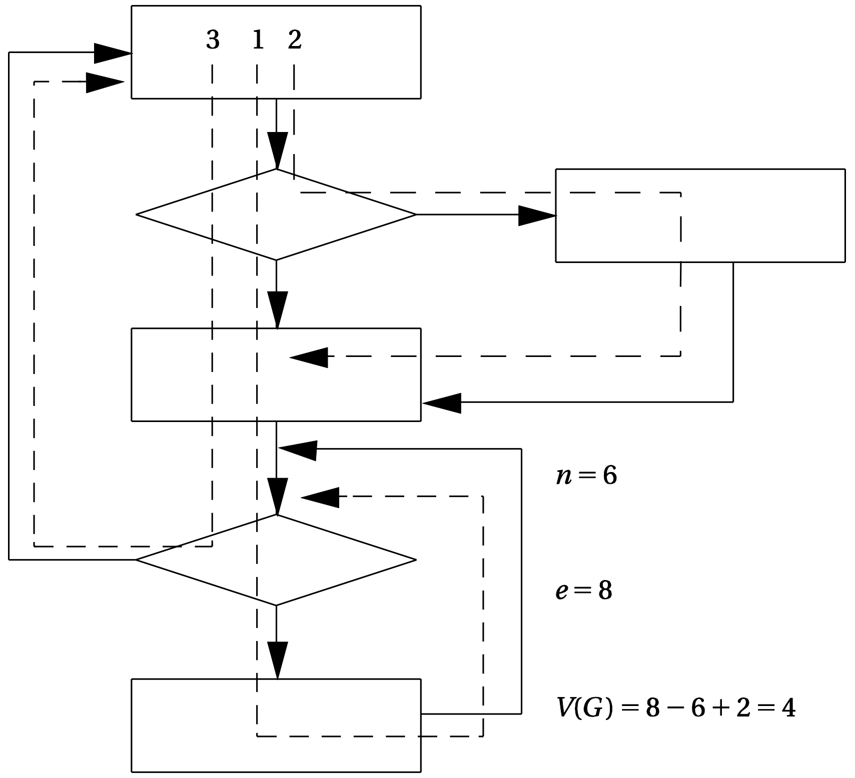
Incidence matrix

$$\begin{array}{c|ccccc} a & 1 & 0 & 0 & 0 & 0 \\ b & 0 & 1 & 0 & 0 & 0 \\ c & 0 & 0 & 1 & 0 & 0 \\ d & 0 & 0 & 0 & 1 & 0 \\ e & 0 & 0 & 0 & 0 & 1 \end{array}$$

Basis set

Cyclomatic complexity

- Cyclomatic complexity is a bound on the size of basis sets:
  - e = # edges
  - n = # nodes
  - p = number of graph components
  - M = e – n + 2p.



Branch testing

- Heuristic for testing branches.
    - Exercise true and false branches of conditional.
    - Exercise every simple condition at least once.

Branch testing example

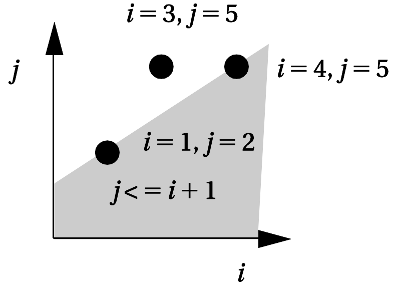
- Correct:
  - if (a || (b >= c)) { printf("OK\n"); }
- Incorrect:
  - if (a && (b >= c)) { printf("OK\n"); }
- Test:
  - a = F
  - (b >=c) = T
- Example:
  - Correct: [0 || (3 >= 2)] = T
  - Incorrect: [0 && (3 >= 2)] = F

Another branch testing example

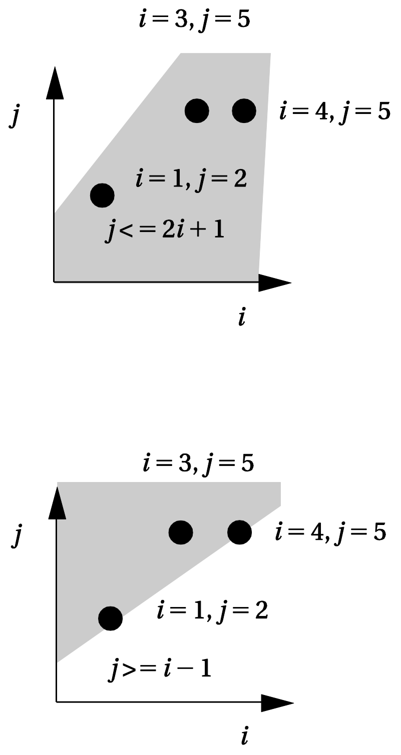
- Correct:
  - if ((x == good_pointer) && x->field1 == 3)) { printf("got the value\n"); }
- Incorrect:
  - if ((x = good_pointer) && x->field1 == 3)) { printf("got the value\n"); }
- Incorrect code changes pointer.
  - Assignment returns new LHS in C.
- Test that catches error:
  - (x != good_pointer) && x->field1 = 3)

Domain testing

- Heuristic test for linear inequalities.
- Test on each side + boundary of inequality.



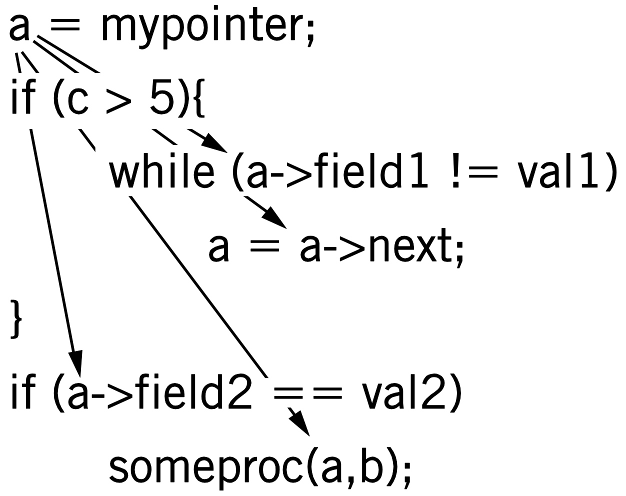
Correct test



Incorrect tests

Def-use pairs

- Variable def-use:
  - Def when value is assigned (defined).
  - Use when used on right-hand side.
- Exercise each def-use pair.
  - Requires testing correct path.



```
a = mypointer;
if (c > 5){
    while (a->field1 != val1)
        a = a->next;
}
if (a->field2 == val2)
    someproc(a,b);
```

Loop testing

- Loops need specialized tests to be tested efficiently.
- Heuristic testing strategy:
  - Skip loop entirely.
  - One loop iteration.
  - Two loop iterations.
  - # iterations much below max.
  - n-1, n, n+1 iterations where n is max.

Black-box testing

- Complements clear-box testing.
  - May require a large number of tests.
- Tests software in different ways.

Black-box test vectors

- Random tests.
  - May weight distribution based on software specification.
- Regression tests.
  - Tests of previous versions, bugs, etc.
  - May be clear-box tests of previous versions.

How much testing is enough?

- Exhaustive testing is impractical.
- One important measure of test quality---bugs escaping into field.
- Good organizations can test software to give very low field bug report rates.
- Error injection measures test quality:
  - Add known bugs.
  - Run your tests.
  - Determine % injected bugs that are caught.