Program Design and Analysis

José Costa

Software for Embedded Systems

Departamento de Engenharia Informática (DEI)
Instituto Superior Técnico

2015-10-20

Outline

- 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

Program-Level Performance Analysis

- Need to understand performance in detail
  - real-time behavior, not just typical
  - on complex platforms

- Program performance $\neq$ CPU performance
  - the way we use pipeline or cache
  - we must analyze the entire program

Program Design and Analysis

- Optimizing for Execution Time
- Optimizing for Energy/Power
- Optimizing for Program Size

- Optimizing for Execution Time
- Optimizing for Energy/Power
- Optimizing for Program Size

Run Times Will Vary

Program execution times depend on several factors

- Input data values
  - different values, different execution paths
- Cache behavior
  - also dependent on input values
- Instruction level
  - floating-point operations
  - pipelining effects

Measuring Program Speed

- CPU simulator
  - clearly slower than executing
  - I/O may be hard
  - may not be totally accurate
- Hardware profiler/timer
  - requires board with timer connected to bus
  - instrumented program to start/stop timer
- Logic analyzer
  - connected to cpu bus
  - relies on indentifiable event on bus
  - limited logic analyzer memory depth

Program Performance Metrics

- Average-case execution time
  - for typical input data values, whatever they are
- Worst-case execution time
  - longest execution time for any possible input sequence
  - important to assert deadlines
- Best-case execution time
  - for any possible input sequence
  - relevant to better timing of the tasks

Performance Analysis

- Elements of program performance:
  - execution time = program path + instruction timing
- Path depends on data values
  - choose which case you are interested in
- Instruction timing depends on pipelining and cache behavior

Programs and Performance Analysis

TÉCNICO LISBOA

- It is hard to get accurate estimates of total execution time from a high-level language
- 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
- However, some aspects of program performance can be estimated by looking at the HLL program

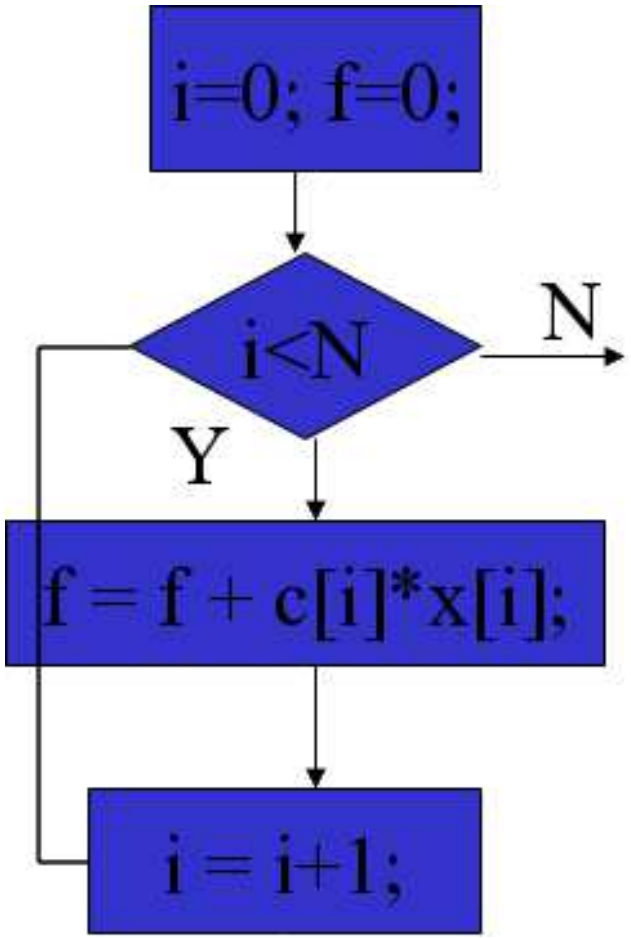
Program Paths

- Consider for loop

```
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

Control Data FlowGraph (CDFG)



Measuring the Longest Path Length

- Must work on the optimized CDFG
  - compiler can change the original CDFG

- Choosing the longest path may not correspond to the longest execution time
  - the time also depends on the timing of the instructions
  - the simplest estimate is to assume that every instruction takes the same amount of time

Instruction Timing

- Not all instructions take the same amount of time
  - but we can look them up
- Instruction execution times are not independent
  - but we can consider their effects
- Execution time may depend on operand values
  - this is more tricky but usually the variances are not that large

Trace-Driven Performance Analysis

TÉCNICO LISBOA

- 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

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

Software Performance Optimization

TÉCNICO LISBOA

How to optimize software performance

- Loop optimizations
- Cache optimizations
- Other performance optimizations strategies

Loop Optimizations

- Loops are good targets for optimization

Basic loop optimizations:

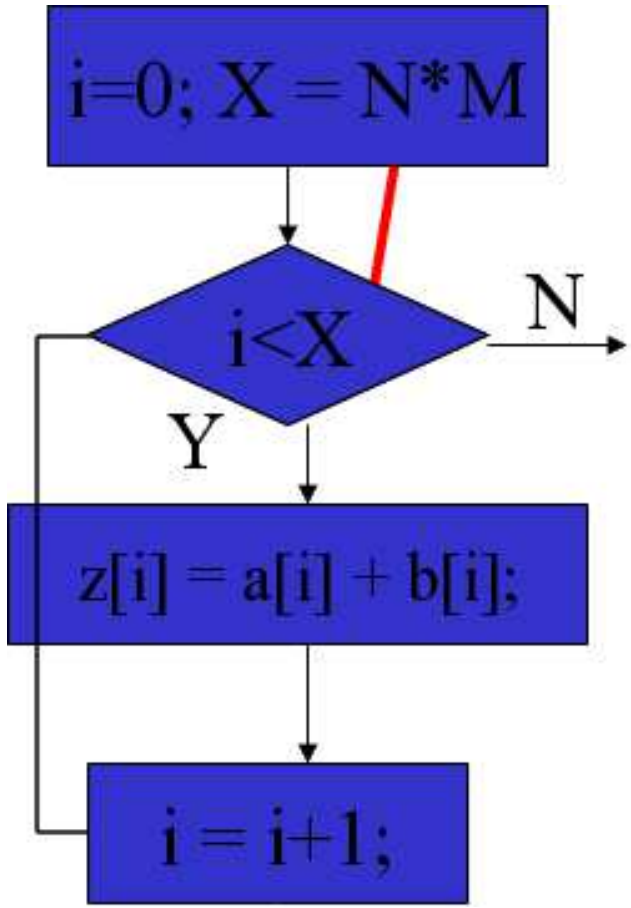
- code motion
- induction-variable elimination
- strength reduction (x*2 $\rightarrow$ x<<1)

Code Motion

- Move unnecessary code out of a loop
- Consider loop:

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

- Don't recompute N*M in each iteration



Induction Variable Elimination

- Induction variable: variable that depends on loop index
- Consider loop:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    z[i][j] = b[i][j];
```

- Don't recompute i*M+j for each array in each iteration
- Share induction variable between arrays, or
- 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
- Changing the order of the loops can sometimes optimize cache performance

Array Conflicts in Cache (1/3)

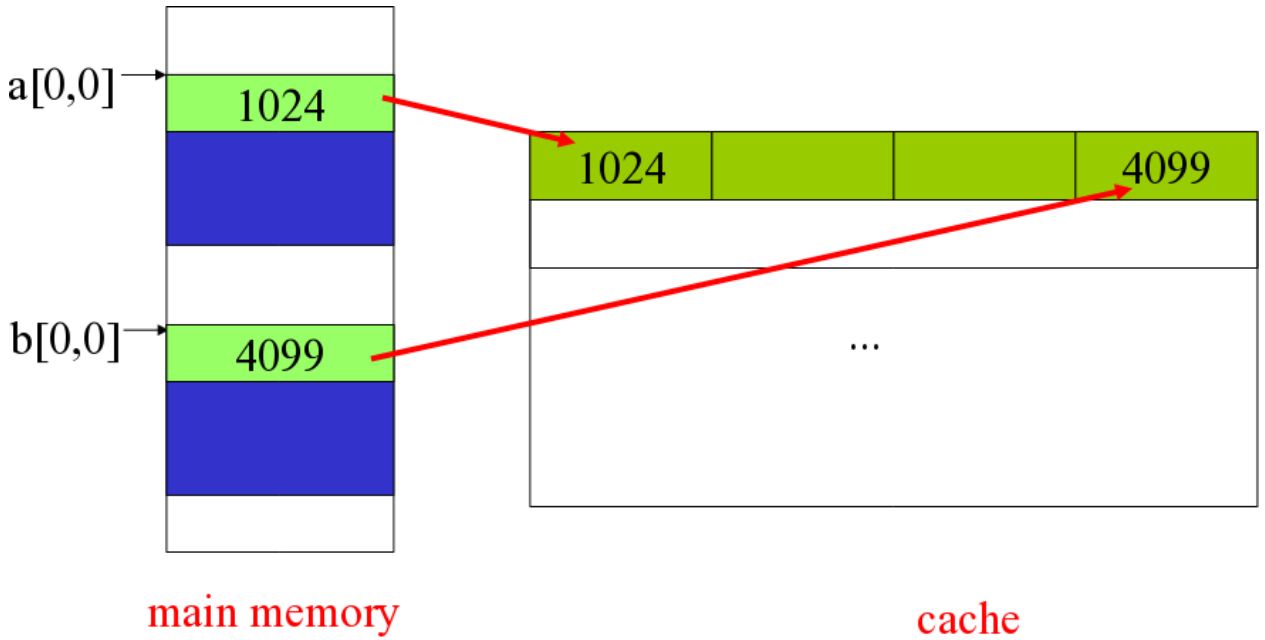
```
for (j=0; j<M; j++)
  for (i=0; i<N; i++)
    a[i][j] = b[i][j] * c;
```

Can be changed to:

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

Array Conflicts in Cache (2/3)

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



Array Conflicts in Cache (3/3)

- 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 Strategies

- Does the code really need to be accelerated?
  - maybe that is not the code where your program spends the most time
- Profilling may help you determine which part of your code needs to be optimized
- You may be able to optimize the algorithm
  - fewer instructions
  - use of static memory instead of allocated
- Look at the implementation of the program

Performance Optimization Hints

- Use registers efficiently
- Use page mode memory accesses
  - rearrange the variables so that more can be referenced contiguously
- Analyze cache behavior
  - instruction conflicts can be handled by rewriting code, rescheduling
  - conflicting scalar data can easily be moved
  - conflicting array data can be moved, padded

- Optimizing for Execution Time
- Optimizing for Energy/Power
- Optimizing for Program Size

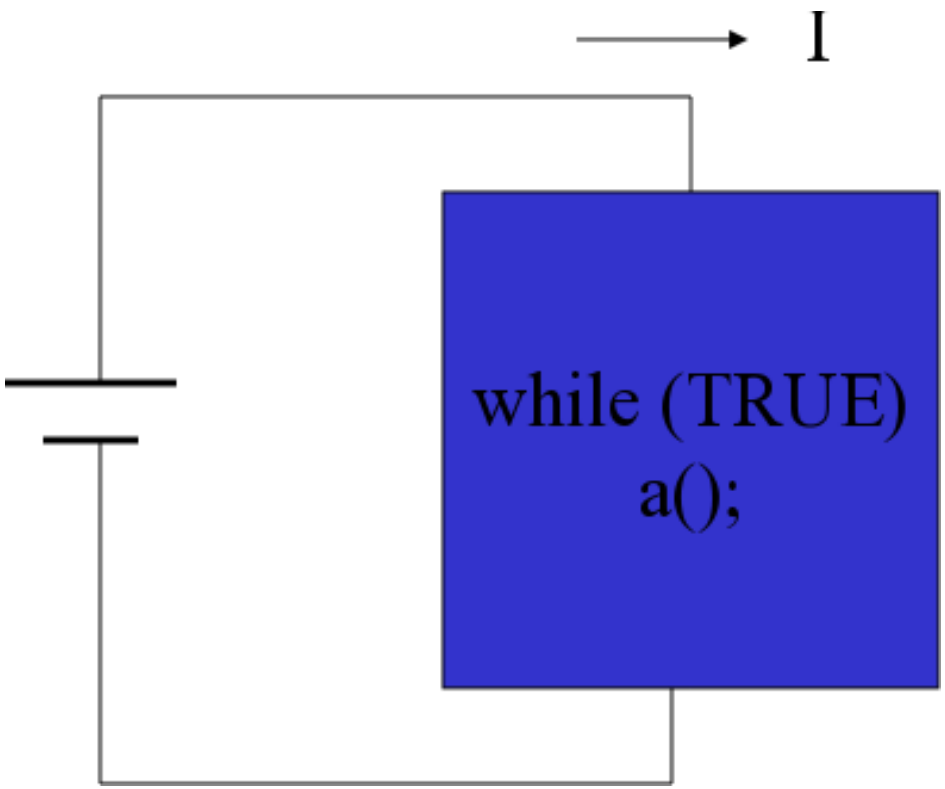
Energy/Power Optimization

- 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'98):
  - 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

Optimizing for Energy (1/2)

- When there is not an energy profile of the hardware:
  - first-order optimization: high performance = low energy

- When there is an energy profile of the hardware:
  - consider the simplification of tasks in higher consumption modules, and
  - increase processing time in lower consumption modules

Optimizing for Energy (2/2)

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

- Optimizing for Execution Time
- Optimizing for Energy/Power
- Optimizing for Program Size

Optimizing for Program Size

- Goal:
  - reduce hardware cost of memory
  - reduce power consumption of memory units

- Two opportunities:
  - data
  - instructions

Data Size Minimization

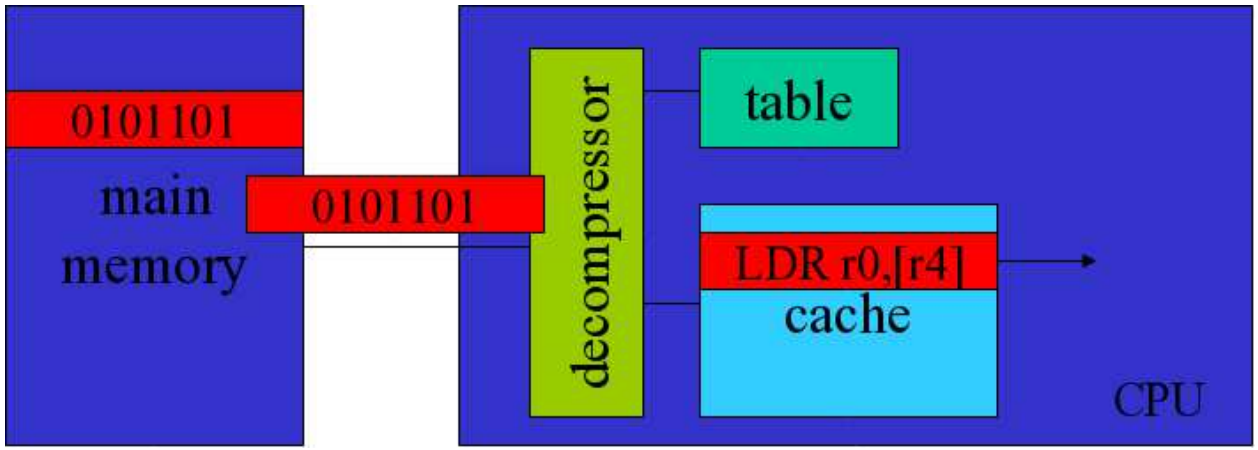
- Reuse constants, variables, data buffers in different parts of code
  - requires careful verification of correctness
- Generate data using instructions
- Both techniques must be used very carefully because they tend to obscure program logic

Reducing Code Size

- Avoid function inlining
- Choose CPU with compact instructions
  - may conflict with execution performance
- Use specialized instructions where possible
  - against RISC "philosophy"

Code Compression

- Use statistical compression to reduce code size, decompress on-the-fly



References

Computers as Components: Principles of Embedded Computing System Design , Marylin Wolf. Morgan Kaufmam. Ch. 5.6, 5.7 and 5.8

Next Class

- Program Design and Analysis - Validation and Testing