Program Optimization and Analysis

Chenyang Lu
CSE 467S
Program Transformation

HLL

compile

assembly

assemble

link

load

executable

Physical Address

Relative Address

Absolute Address

optimize

Analyze

Chenyang Lu

CSE 467S

2
What do we need to do?

- Understand optimization levels (-O1, -O2, etc.)

- Optimize HLL code.

- Analyze and optimize assembly code.

- Modifying compiler output requires care:
  - correctness;
  - loss of hand-tweaked code.
Goals

- Optimizing for execution time.
- Optimizing for energy/power.
- Optimizing for program size.
- They may conflict with each other!
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);
  ```

- Eliminate by control flow analysis, constant folding.
Function Call
Instructions (ARM7)

- Branch and link instruction:
  
  \[
  \text{BL~} \quad \text{foo} \quad \text{==} \quad \text{MOV~} \quad r14, \quad r15 \\
  \quad \text{B~} \quad \text{foo}
  \]

  - \( r15 \) contains the current PC
  - Copies current PC to \( r14 \).

- To return from subroutine:
  
  \[
  \text{MOV~} \quad r15, \quad r14
  \]
Stack

- Use a stack to keep track of
  - parameters,
  - return value,
  - return address.

- Caller and callee access the stack in a consistent order.
  - Different compilers/programmers may follow different orders.

- Access the stack (ARM7)
  - r13 always points to the top of stack
  - **Push:** STR r0, [r13, #4]!
  - **Pop:** SUB r13, #4
Stack Operations

- **Caller: call a function**
  - Push parameters to stack
  - BL (r15 → r14; jump)

- **Callee: receive a call**
  - Read parameters from stack
  - Overwrite top of stack with return address (r14)

- **Callee: return**
  - Load PC with return address (on top of stack)

- **Caller: receive a return**
  - Pop callee’s return address from stack
Nested function calls (ARM7)

```c
main() { f1(x); }
void f1(int a) { f2(a); }

; f1 is called by main()
LDR r0, [r13] ; load parameter into r0 from stack
STR r14, [r13] ; store f1’s return addr.

; f1 calls f2()
STR r0, [r13, #4]! ; push parameter for f2 to stack
BL f2 ; branch and link to f2

; return from f2()
SUB r13, #4 ; pop f2’s parameter off stack

; f1 returns to main()
LDR r15, [r13] ; restore register and return
```
Function Inlining

```c
int foo(a, b, c) { return a + b - c; }
```

```c
z = foo(w, x, y);
```

⇒

```c
z = w + x - y;
```

- Improve performance by eliminating function call overhead.
- May increase code size, but not always…
- Affect instruction cache behavior.
## Optimization: Inlining

Inlining improves performance and reduces code size.

### Why?

<table>
<thead>
<tr>
<th>App</th>
<th>Code size</th>
<th>Code reduction</th>
<th>Data size</th>
<th>CPU reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textit{inlined}</td>
<td>\textit{noninlined}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surge</td>
<td>14794</td>
<td>16984</td>
<td>12%</td>
<td>1188</td>
</tr>
<tr>
<td>Maté</td>
<td>25040</td>
<td>27458</td>
<td>9%</td>
<td>1710</td>
</tr>
<tr>
<td>TinyDB</td>
<td>64910</td>
<td>71724</td>
<td>10%</td>
<td>2894</td>
</tr>
</tbody>
</table>
Loops
Loop Unrolling

- Reduces loop overhead

```c
for (i=0; i<4; i++)
    a[i] = b[i] * c[i];

⇒

for (i=0; i<4; i+=2) {
    a[i] = b[i] * c[];
    a[i+1] = b[i+1] * c[i+1];
}
```
Loop Overhead on ARM7

; loop initiation code
MOV r0, #0 ; use r0 for loop counter
MOV r8, #0 ; use separate index for arrays
LDR r1, #4 ; buffer size
MOV r2, #0 ; use r2 for f
ADR r3, c ; load r3 with base of c[ ]
ADR r5, x ; load r5 with base of x[ ]

; loop;
L:  LDR r4, [r3, r8] ; get c[i]
    LDR r6, [r5, r8] ; get x[i]
    MUL r4, r4, r6 ; compute c[i]x[i]
    ADD r2, r2, r4 ; add into sum
    ADD r8, r8, #4 ; add one word to array index
    ADD r0, r0, #1 ; add 1 to i
    CMP r0, r1 ; exit?
    BLT L ; if i < 4, continue
Loop Fusion

Combines multiple loops:

```c
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
⇒
for (i=0; i<N; i++) {
    a[i] = b[i] * 5; w[i] = c[i] * d[i];
}
```

Necessary conditions

- Loops share a same index
- No dependencies between two loops
Code Motion

for (i=0; i<N*M; i++)
    z[i] = a[i] + b[i];
Array
One-Dimensional Array

C array name points to 0th element:

$$a[i] = *(a + i)$$
Two-Dimensional Array

- **Row-major layout:**

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

zptr = z; bptr = b;
for (i=0; i<N; i++)
    for (j=0; j<M; j++) {
        zind = i*M+j;
        bind = i*M+j;
        *(zptr+zind)=*(bptr+bind)
    }
zptr = z; bptr = b;
for (i=0; i<N; i++)
    for (j=0; j<M; j++) {
        zbind = i*M+j;
        *(zptr+zbind)=*(bptr+zbind);
    }
zptr = z; bptr = b; zbind = 0;
for (i=0; i<N; i++)
    for (j=0; j<M; j++) {
        *(zptr+zbind)=*(bptr+zbind);
        zbind++;
    }
Cache Analysis

- Loops use large quantities of data (arrays) → cache conflicts
Direct-Mapped Cache

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0xabcd</td>
<td>byte byte byte byte ...</td>
</tr>
</tbody>
</table>

`valid` = tag

Cache block

Tag | Index | Offset
---|-------|-------

hit

byte
Array Conflicts in Cache

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

main memory

<table>
<thead>
<tr>
<th>a[0,0]</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[0,0]</td>
<td>4099</td>
</tr>
</tbody>
</table>

cache

<table>
<thead>
<tr>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>4099</td>
</tr>
</tbody>
</table>

256
Array Conflicts

- Array elements conflict because they are in the same line.
- Solution: move one array.
Static Cache Locking

- Lock instructions in cache before execution.
- **Predictable** execution time.

- Similarly, lock code and data in memory to avoid paging.
Register Allocation

- Fit current variables in registers.

- Load once, use many times.
  - Reduce number of cache/memory accesses.
  - Improve performance.
  - Reduce energy consumption.
Register Lifetime Graph

1. \( w = a + b; \)
2. \( x = c + w; \)
3. \( y = c + d; \)
4. \( z = a - b; \)

*No. of needed register = 5*
After Rescheduling...

1. \( w = a + b; \)
2. \( z = a - b; \)
3. \( x = c + w; \)
4. \( y = c + d; \)

**no. of needed register = 4**

Cannot change dependencies between instructions!
Summary: Performance Optimization

- Use registers efficiently.

- Optimize loops.

- Optimize function calls.

- Optimize cache behavior:
  - Avoid instruction conflicts by rewriting code, rescheduling;
  - Move conflicting scalar/array data can be moved.
Execution Time Analysis

- Real-time systems must meet deadlines.
- Need to analyze execution time.
Execution Time

- Affected by program path and instruction timing

- Program path depends on input data.
  - Sensor readings
  - User input

- Instruction timing depends on
  - pipelining
  - cache behavior – memory can be x10 slower than cache!
for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];

• Loop initiation executed once.
• Loop test executed N+1 times.
• Loop body and index update executed N times.
Execution Time Metrics

- Difficult to predict execution time accurately.

- Average case
  - For “typical” data values
  - Soft real-time

- Worst case
  - For any possible input set
  - Hard real-time
  - Longest program path may NOT lead to worst-case execution time
Approaches

- Compile-time analysis: pessimistic
- Measurement: optimistic
Analysis

- Analyze optimized assembly/binary code, not high-level language (HLL) code
  - HLL statement $\rightarrow$ many assembly/binary instructions
  - Example: function calls

- Challenges
  - Program path depends on input data
  - Pipelining, cache effects are hard to predict
  - Analysis tends to be pessimistic
Measurement

CPU simulator
- I/O may be hard to measure.
- May not be totally accurate.

Time stamping
- Requires instrumenting program.
  - Timer granularity
    - `Gettimeofday` on Linux: ms
    - `Gethrtime` on Intel processors: read 64-bit clock cycle counter and return the number of clock cycles since CPU was powered up or reset.

Logic analyzer: limited logic analyzer memory depth.
Timing diagram of event propagation on Mote
Granularity: 50 microsecond
Trace-driven Analysis

- Record of the program path of a program.

- Help study cache behavior and power management policies.

- A useful trace
  - requires proper input values;
  - is large.
Trace Generation

- **Hardware capture**
  - Logic analyzer
    - Limited buffer space
    - Cannot observe on-chip cache
  - Hardware assist in CPU
    - Pentium supports automatic tracing of branches

- **Software**
  - PC sampling
  - Instrumentation instructions
  - Simulation
Goals

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

- **Goals**
  - Reduce memory cost;
  - Reduce power consumption.

- **Two opportunities:**
  - Data;
  - Instructions.
Reduce Data Size

- Reuse constants, variables, buffers in different parts of code.
  - Single-buffer in TinyOS.
  - Pack multiple flags in one byte.
  - Use shortest data type needed.
  - Requires careful verification of correctness.

```c
uint8_t i;
for(i = 0; i < 1000; i++) {
  ...
}
// This loop will never terminate
```

- Generate data using instructions.
Reduce Code Size

- Avoid loop unrolling.

- Inlining?
  - Size of function
  - Number of calls

- Choose CPU with compact instructions.
  - Digital Signal Processors (DSP) tend to have smaller code.

- Some CPUs support dense instruction set
  - ARM Thumb, MIPS-16
Code Compression

- Use statistical compression to reduce code size.
- Decompress on-the-fly.
- Need to handle jump addresses.
Reading

- Textbook 5.5, 5.6, 5.7, 5.8, 5.9.