CSci 2021: Review Lecture 2

Stephen McCamant
University of Minnesota, Computer Science & Engineering

Quiz 2 topics (in one slide)

- CPU architecture
  - Y86 instructions
  - Control logic and HCL
  - Sequential Y86
  - Pipelined Y86
- Code optimization
  - Machine-independent techniques
  - Instruction-level parallelism
- Memory hierarchy and caches
  - Memory and disk technologies
  - Locality and how to use it
  - Cache parameters and operation
  - Optimizing cache usage

Outline

Topics in CPU architecture

Topics in code optimization

Topics in memory hierarchy and caches

Discussion problems

Y86 instructions

- Simplified subset of x86, simpler encoding
- 32-bit only, 8 registers
- Four kinds of moves, only one addressing mode
- Add, subtract, bitwise and, bitwise xor
- Conditional jump and move based on equality and signed comparison
- Call, return, push, pop
- Halt and two fatal errors, no exceptions

Logic design for control

- Combinational circuits:
  - Compute a function of bits, no memory
  - Acyclic network of AND, OR, and NOT gates
  - Also includes word-sized comparison, multiplexors, and ALU
- Stateful elements:
  - (Clocked) registers
  - Random-access memory
  - State updates occur on rising clock edge only

Hardware design in HCL

- Simple language for specifying control circuits
- Two types: Boolean and word
- Comparison and logic operators (no side-effects or "short circuiting")
- Core construct: sequential conditional
  - $[C_1 : V_1; C_2 : V_2; \ldots 1 : V_n]$
  - "Else" case written 1

Sequential Y86

- Whole state update function is one big combinational circuit
- Express behavior of each instruction using smaller computations
- Processing split into stages for organization:
  - Fetch, decode, execute, memory, write back, PC update
- Simplest, but requires long cycle time (slow)

Pipelining basics

- Split processing into stages, and work on multiple instructions at once
- Reduces cycle time and increases hardware utilization
- Pipeline registers hold data between stages
- Performance concerns: balanced stages, and not too many
- Correctness concerns: must have same final behavior

Pipelining techniques

- *Hazards*: dependencies introduce danger of incorrect results
- Branch prediction: guesses result of conditional jumps
- Stalling: hold up instructions until data ready
  - Simple, but introduces a lot of delay
  - Used for return instruction in Y86
- Cancelling: kill incorrect instructions
  - Must happen before they have side-effects
  - Used for branch mis-predictions
- Forwarding: copy data to a different stage right as needed

Outline

Topics in CPU architecture

Topics in code optimization

Topics in memory hierarchy and caches

Discussion problems

Principles of optimization

- Concentrate on the program parts that run the most
  - Amdahl's law bounds possible speedup
  - Array-style programs: concentrate on inner loops
  - Complex programs: use a profiler
- Know what the compiler can and can't do
  - Compiler can be smart, but is careful about correctness
  - Functions and pointers (aliasing) block optimization
- Watch out for algorithmic problems

Machine-independent optimizations

- Move computations out of loops
- Avoid abstract functions in time-critical code
- Use temporary variables to reduce memory operations
- Unroll loops to reduce bookkeeping overhead

Instruction-level parallelism

- Modern processors are *super-scalar*
  - Can do more than one thing at once
- And *out-of-order*
  - In a different sequence than the original instructions
- Multiple *functional units*, each with different throughput and latency

Exposing loop parallelism

- To reduce latency, avoid a long *critical path*
- Functional unit throughput is an ultimate limit
- Unroll to allow optimization between iterations
- Techniques to shorten the critical path:
  - Re-associate associative operators
  - Replace a single accumulator with multiple parallel accumulators

Outline

Topics in CPU architecture

Topics in code optimization

Topics in memory hierarchy and caches

Discussion problems

RAM technologies

- SRAM: several (e.g. 6) transistors per bit
  - Faster
  - More expensive, less dense
  - Used for caches
- DRAM: one capacitor and transistor per bit
  - Must be periodically refreshed
  - Cheaper, more dense
  - Slower
  - Used for main memory
  - Typical DIMM organized by chips, rows, and columns

Disks and SSDs

- (Spinning) hard drives
  - Highest capacity
  - Random access time limited by seek and rotation latencies
  - Always read or write an entire sector at a time
- Solid-state (flash) drives
  - Technology descended from EEPROMs
  - Random-access reads are very fast
  - Can only rewrite by erasing large blocks
  - Random-access writes require recopying, slower

Spatial and temporal locality

- Spatial locality: memory accesses are close together in location
  - Best case: sequential accesses
- Temporal locality: the same location is accessed repeatedly close together in time
  - Set of locations being used is called the *working set*
- Because of locality, different locations have very different chances of being accessed next

Memory hierarchy

- Devices have trade-off between access time and capacity
  - Differences of many orders of magnitude
- Combine small+fast devices with big+slow ones in a hierarchy
- Because of locality, most uses are in small+fast device
- Must move data between levels
  - Keeping a copy at a higher level is called *caching*
  - First example: caches between CPU core and memory

Cache parameters

- Data is moved in blocks of size $B = 2^b$
- Organize cache into $S = 2^s$ sets of lines
- A set contains $E = 2^e$ lines, each of which can contain one of the same blocks
  - $E = 1$: direct mapped
  - $E > 1$: E-way set associative
  - $S = 1$: fully associative
- Total capacity $C = S \cdot E \cdot B$
- b and s also give a division of addresses into $m = t + s + b$

Cache operations: read

- Use s bits as an index to choose a set
- Check all lines in the set (hardware: in parallel), to see if any is valid and has a matching tag
- If yes, it's a *hit*: block offset indicates which bytes desired
- If not present, it's a *miss*
  - Fetch data from lower level (e.g., main memory)
  - Insert newly read data, usually *evicting* another block

Cache operations: write

- Look for a matching line as for a read
- If a hit, update contents of cache block
  - *Write-back* policy: do not copy to lower levels until evicted (opposite is write-through)
- If a miss, the common *write-allocate* policy copies the block into the cache
  - Exploits locality in write-only accesses

Cache usage optimizations

- Overall goals: maximize locality, minimize working set
- Use more compact data representations
- Prefer stride-1 data accesses
  - E.g., for a matrix, iterate over indexes in outer-to-inner order
- Temporally group accesses to the same data values
  - For 2-D data, group by blocks (tiles) instead of rows

Outline

Y86 "compiling"

```
int ary[10][10];
ary[i][j]++;
```

`ary` is in `%eax`, `i` is in `%ebx`, `j` is in `%ecx`.
Step 1: write a formula for `&ary[i][j]`

Y86 "compiling"

```
int ary[10][10];
ary[i][j]++;
```

`ary` is in `%eax`, `i` is in `%ebx`, `j` is in `%ecx`.
Step 1: write a formula for `&ary[i][j]`

```
4*(j + 10 * i) + ary
```

Y86 "compiling", pt. 2

`ary` is in `%eax`, `i` is in `%ebx`, `j` is in `%ecx`.
`4*(j + 10 * i) + ary`

```
rrmovl %ebx, %esi # esi = i
addl   %esi, %esi # esi = 2*i
addl   %esi, %esi # esi = 4*i
addl   %ebx, %esi # esi = 5*i
addl   %esi, %esi # esi = 10*i
addl   %ecx, %esi # esi = 10*i + j
addl   %esi, %esi # esi = 2*(10*i + j)
addl   %esi, %esi # esi = 4*(10*i + j)
addl   %eax, %esi # esi = ary + 4*(10*i + j)
```

Y86 "compiling", pt. 3

Instructions for `(*%esi)++`

Y86 "compiling", pt. 3

Instructions for `(*%esi)++`

```
mrmovl  0(%esi), %edi   # Load into %edi
irmovl  1, %edx
addl    %edx, %edi      # %edi++
rmmovl  %edi, 0(%esi)   # Store back
```

Optimization

Why does the following program run slowly?

```
char *concat(char *a, char *b) {
    char *c = malloc(strlen(a) + strlen(b) + 1);
    strcpy(c, a); strcat(c, b);
    free(a); free(b); return c;
}
int main(int argc, char **argv) {
    char *buf = strdup("");
    char *linebuf = 0; size_t len = 0; int i;
    while (getline(&linebuf, &len, stdin) != -1)
        buf = concat(buf, strdup(linebuf));
    for (i = strlen(buf) - 1; i >= 0; i--)
        putchar(buf[i]);
    return 0;
}
```

Cache parameters

The following caches all have 64-byte blocks:

	C	E	S
A.	32 KB	1	512
B.	32 KB	8	64
C.	32 KB	512	1

- Which cache needs the most gates?
- Which cache has the fastest hit time?
- Which cache has the lowest miss rate?
- Which cache is found in a real Core i7?

Cache parameters

The following caches all have 64-byte blocks:

	C	E	S
A.	32 KB	1	512
B.	32 KB	8	64
C.	32 KB	512	1

- Which cache needs the most gates? C
- Which cache has the fastest hit time?
- Which cache has the lowest miss rate?
- Which cache is found in a real Core i7?

Cache parameters

The following caches all have 64-byte blocks:

	C	E	S
A.	32 KB	1	512
B.	32 KB	8	64
C.	32 KB	512	1

- Which cache needs the most gates? C
- Which cache has the fastest hit time? A
- Which cache has the lowest miss rate?
- Which cache is found in a real Core i7?

Cache parameters

The following caches all have 64-byte blocks:

	C	E	S
A.	32 KB	1	512
B.	32 KB	8	64
C.	32 KB	512	1

- Which cache needs the most gates? C
- Which cache has the fastest hit time? A
- Which cache has the lowest miss rate? C
- Which cache is found in a real Core i7?

Cache parameters

The following caches all have 64-byte blocks:

	C	E	S
A.	32 KB	1	512
B.	32 KB	8	64
C.	32 KB	512	1

- Which cache needs the most gates? C
- Which cache has the fastest hit time? A
- Which cache has the lowest miss rate? C
- Which cache is found in a real Core i7? B