CPUs and Compilers

2024 Fall ECE454: Computer Systems Programming — Lecture 3

Jon Eyolfson — 1.0.0

Let’s Re-visit the History of CPU Architectures

To optimize our programs, we need to know a bit more about the hardware

ECE552 is the course to take if you’re interested in architecture

How Do We Measure a Single CPU Core?

Does higher clock frequency mean a faster core?

No, there’s many factors to consider including:

- IPS (instructions per second, you can prefix with B for billions)
- FLOPS (float point operations per second)
- IPC (instructions per clock)
- CPI (cycles per instruction)

CPI is the reciprocal of IPC

Note: these are all measures of throughput

Intel’s First CPU—4004

It was a: 4-bit processor @ 108KHz, with 2300 transistors

No:
- Virtual memory
- Interrupts
- Pipelining

It also had 3 stack *registers*

Continued until the Intel 8086 in 1978,
a 16-bit CPU @ 10 MHz (beginning of x86 instruction set)

Faster clocks meant CPI can stay the same, but the end result is faster

Old CPUs Executed Instructions Sequentially

Typical processor units:

- IF: Instruction Fetch
- ID: Instruction Decode
- EX: Execute
- MEM: Memory access
- WB: Register write back

Instr \ **Cycle**	1	2	3	4	5	6	7	8	9	
1	IF	ID	EX	MEM	WB					
2						IF	ID	EX	MEM	WB

CPI = 5, in this case

RISC and Pipelining

The idea behind pipelining is that every processor unit runs on each cycle

Instr \ Cycle	1	2	3	4	5	6	7	8	9
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB

Once the pipeline is full, ideally CPI = 1

Note modern pipelines are deep (around 19 for Zen)

Branchs Cause Issues

If we're decoding a branch instruction, what's the next one we fetch?
- It depends on the result!

Solution 1: Stall
- Delay the fetch until we know the result

It turns out branches are 10-15% of instructions for a typical program

Effective CPI >1

Branch Prediction

Modtern CPUs have very good branch predictors
- Instead of stalling, make an educated guess
  - (could use tables to store history)

The code after the branch executes speculatively (speculative execution)

This pushes the CPI back closer to 1 (branch predictors are quite good)

Can we get a CPI <1?

Going Superscalar—Instruction Level Parallelism

Multiple instructions execute in the same cycle

This is possible if the two instructions are independent
- This makes the decode more complicated

Cycle	Instructions (Finished)
1	① ②
2	③
3	④ ⑤

Assuming the numbers correspond to the sequential order in our program

We Also Have Out of Order Execution

Instructions that cause a cache miss are very slow, we can execute others!

Cycle	Instructions (Finished)
1	1 2
2	3 6
3	4 5

Simultaneous Multithreading (SMT) is Another Option

For normal threaded applications we may have:

Cycle	Instructions (Finished)
1	1 2
2	3
3	4 5
Context Switch	
4	1
5	2
6	3 4

For SMT (also called Hyperthreading), if resources are ideal the other thread could use, execute them

Cycle	Instructions (Finished)
1	1 2
2	3 1
3	4 2
4	3 4 5

Architecture Summary

Technique	CPI
Basic	n
Pipelining	»1
Branch Prediction	>1
ILP	<1
Out of Order Execution	«1
SMT	«1

What About the Compiler?

Let’s look at the optimizations the compiler performs

Optimizations relate to performance, avoid doing these by yourself, since:

- You’ll likely waste time
- Make your code more unreadable

Compiler’s have a host of optimization options, we’ll look at `gcc`

GCC Optimization Levels

-O1 (-O)
Reduce code size and execution time
No optimizations that increase compiliation time

-O2
All optimizations except space-speed tradeoff ones

-O3
All optimizations

-O0 (default, plain)
Fastest compilation time, debugging performs as expected

Disregard Standards, Acquire Speedup

`-Ofast`
All `-O3` optimizations and non-standard compliant optimizations, namely `-ffast-math`

Turns off exact implementations of IEEE or ISO rules/specifications for math functions

Generally, if you don’t care about the exact results, you can use this for a speedup

Scalar Optimizations are the First Class, e.g. Constant Folding

```
i = 1024 * 1024
```

The compiler will not emit code that does the multiplication at runtime, it will simply use the computed value

```
i = 1048576
```

Enabled at all optimization levels

Common Subexpression Elimination

`-fgcse`

Perform a global common subexpression elimination pass
This pass also performs global constant and copy propagation
Enabled with `-O2`, `-O3`

Example:

```
a = b * c + g;
d = b * c * d;
```

Instead of computing `b * c` twice, we compute it once, and reuse the value in each expression

Constant Propagation

Moves the constant values from definition to use

- Valid if there's no redefinition of the variable

Example:

```c
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

with constant propagation would produce:

```c
int x = 14;
int y = 0;
return 0;
```

Copy Propagation

Replaces direct assignments with their values, usually required to run after common subexpression elimination

Example:

```
y = x
z = 3 + y
```

with copy propagation would produce:

```
z = 3 + x
```

Dead Code Elimination

`-fdce`

Remove any code that is guaranteed not to execute

Enabled at all optimization levels

Example:

```
if (0) {
    z = 3 + x;
}
```

would not be included in the final executable

Loop Optimizations are Next, e.g. Loop Unrolling

`-funroll-loops`

Unroll any loops with a set number of iterations

Example:

```
for (int i = 0; i < 4; ++i)
    f(i)
```

would be transformed to:

```
f(0)
f(1)
f(2)
f(3)
```

Loop Interchange

`-floop-interchange`

Example: in C the following:

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

would be transformed to this:

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

since C is **row-major** (meaning a[1][1] is beside a[1][2]), the other possibility is **column-major**

Loop Fusion

Example:

```
for (int i = 0; i < 100; ++i)
    a[i] = 4

for (int i = 0; i < 100; ++i)
    b[i] = 7
```

would be transformed to this:

```
for (int i = 0; i < 100; ++i) {
    a[i] = 4
    b[i] = 7
}
```

There is a trade-off here between data locality and loop overhead, the opposite of this is called **loop fission**

Loop-Invariant Code Motion

Moves invariants out of the loop
- Also called loop hoisting

Example:

```
for (int i = 0; i < 100; ++i) {
    s = x * y;
    a[i] = s * i;
}
```

would be transformed to this:

```
s = x * y;
for (int i = 0; i < 100; ++i) {
    a[i] = s * i;
}
```

This reduces the amount of work we have to do for each iteration of the loop

Other Optimizations, e.g. Devirtualization (1)

`-fdevirtualize`

Attempt to convert calls to virtual functions to direct calls

Enabled with `-O2`, `-O3`

Virutal functions impose some overhead, for instance in C++, you must read the objects RTTI (run-time type information) then effectively branch to the correct function

Devirtualization (2)

Example:

```
class A {
    virtal void m();
};

class B : public A {
    virutal void m();
};

int main(int argc, char *argv[]) {
    std::unique_ptr<A> t(new B);
    t.m();
}
```

could eliminate reading the RTTI and just insert a call to B's `m`

Scalar Replacement of Aggregates

`-fipa-sra`

Replace references by values when appropriate

Enabled at `-O2` and `-O3`

Example:

```
{
    std::unique_ptr<Fruit> a(new Apple);
    std::cout << color(a) << std::endl;
}
```

could be optimized to:

```
std::cout << "Red" << std::endl;
```

if the compiler knew what `color` does

Aside: Three Address Code

TAC is a representation of intermediate code used by compilers, mostly used for analysis and optimization

Statements represent one fundamental operation (for the most part, we can consider each operation atomic)

Useful to reason about your program, and easier to read than assembly (as long as you seperate out memory reads/writes)

Statements have the form: *result* $:=$ *operand*$_1$ *operator operand*$_2$

GIMPLE

GIMPLE is the three address code used by `gcc`

To see the GIMPLE representation of your compilation use the `-fdump-tree-gimple` flag

To see all of the three address code generated by the compiler use `-fdump-tree-all`, you'll probably just be interested in the optimized version

Use this if you want to reason about your code at a low-level without having to read assembly

The `restrict` Keyword in C/C++

"A new feature of C99: The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables."

For C99 standard in `gcc` use the `-std=c99` flag (we use C23)

If you declare a pointer with `restrict`, you are ensuring to the compiler that the pointer will never alias (another pointer will not point to the same data) for the lifetime of the pointer

Example of `restrict` (1)

If you have a bunch of pointers declared with `restrict`, you are saying that these will never point to the same data
   Below is the Wikipedia example, would declaring all these pointers as `restrict` generate better code?

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
    *ptrA += *val;
    *ptrB += *val;
}
```

Example of `restrict` (2)

Let's look at the GIMPLE instead

```
D.1609 = *ptrA;
D.1610 = *val;
D.1611 = D.1609 + D.1610;
*ptrA = D.1611;
D.1612 = *ptrB;
D.1610 = *val;
D.1613 = D.1612 + D.1610;
*ptrB = D.1613;
```

Is there any operation here that could be left out if all the pointers represent different data?

Example of `restrict` (3)

If `ptrA` and `val` are different, you don't have to reload the data on **line 6**

Otherwise you would since you could call `updatePtrs(&x, &y, &x);`

If you change the arguments to, you will get the optimized version

```
void updatePtrs(int* restrict ptrA, int* restrict ptrB,
                int* restrict val);
```

Note: you can get the optimization by just declaring `ptrA` and `val` as `restrict`, `ptrB` isn't needed for this optimization

Summary of `restrict`

Use `restrict` whenever you know the pointer will not alias another pointer (also declare as `restrict`)

- The compiler may be not able to know whether pointers alias, so you must provide this

This allows the compiler to do better optimization for your code (and therefore run faster)

- Caveat: don’t lie to the compiler, or else you will get undefined behaviour

Aside: this not the same as `const`, `const` data can still be changed through a different pointer

Aliasing and Pointer Analysis

We've seen using `restrict` to tell the compiler variables do not alias

Pointer analysis tracks the variables in your program to determine whether or not they alias

- If they don't alias, we can reorder them and do other types of optimizations

Call Graph

A directed graph that shows relationships between functions

Relativity simple in C, hard for virtual function calls (C++/Java)

Virtual calls require pointer analysis

Importance of Call Graphs

Having the call graph allows us to know if the following can be optimized:

```c
int n;

int f() { /* opaque */ }

int main() {
    n = 5;
    f();
    printf("%d\n", n);
}
```

We could propagate the constant value 5, as long as we know that `f()` does not write to `n`

Tail Recursion Elimination

`-foptimize-sibling-calls`

Optimize sibling and tail recursive calls

Enabled at `-O2` and `-O3`

Example:

```
int bar(int N) {
    if (A(N))
        return B(N);
    else
        return bar(N);
}
```

We can just replace the call to `bar` by a `goto` at the compiler level, this way we avoid having overhead of a function call and increasing our call stack

Branch Predictions

GCC attempts to guess the probability of branches in order to do the best code ordering

You can use `__builtin_expect(expr, value)` to help GCC, if you know the run-time characteristics of your program

Example (in the Linux kernel):

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

Architecture Specific

Two common ones `march` and `mtune` (`march` implies `mtune`)
- These enable using specific instructions that not all CPUs may support (AVX512, etc.)

Example: `-march=arrowlake`
- Good to use on your local machine, not so much for shipped code

Now We Know What the Compiler is Doing

A feel of what the optimization levels do

What some of the compiler optimizations are

Full list: `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

Compiler Explorer: `https://godbolt.org/` is a great tool too!