Объектно-ориентированное программирование

Object-oriented programming

XIII. Дизайн относительно данных

Data-oriented design

"Software is getting slower more rapidly than hardware becomes faster."

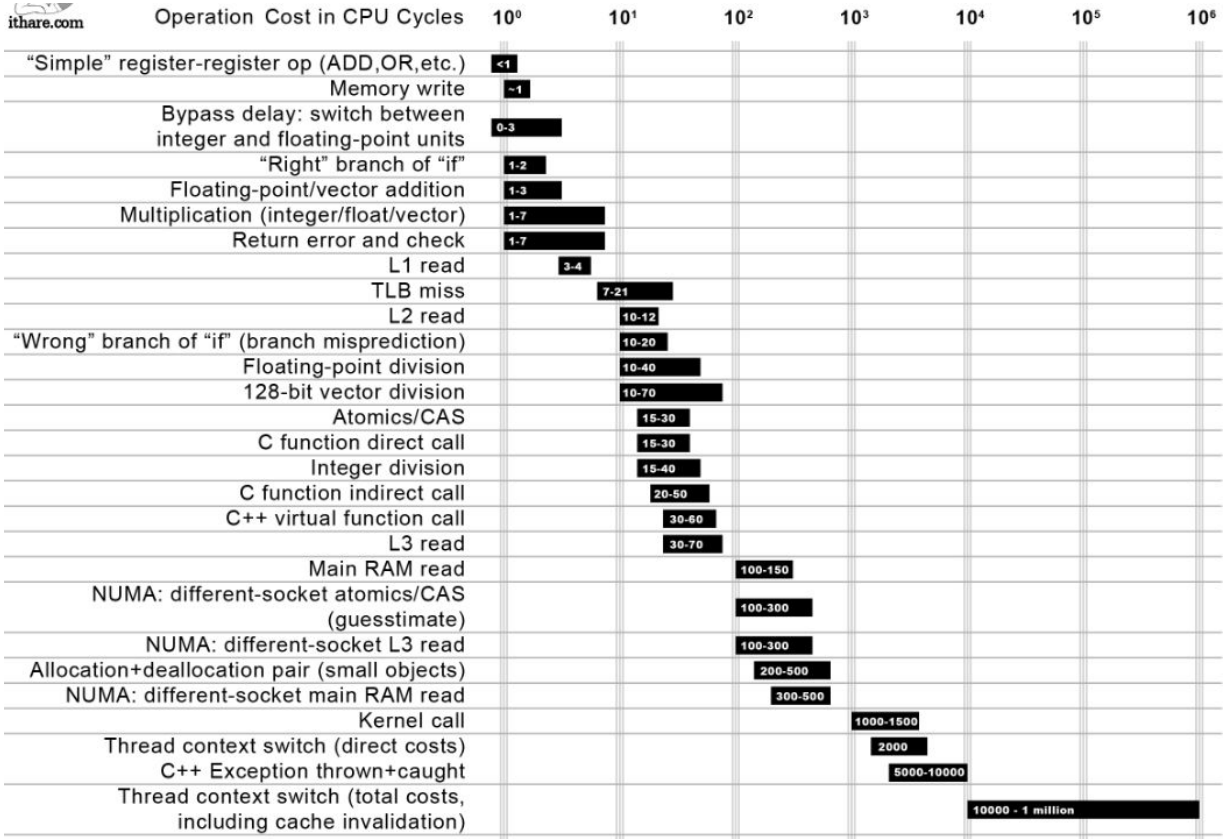
N. Wirth

leon.brinzan@iis.utm.md

https://youtu.be/rX0ItVEVjHc&t=4580s

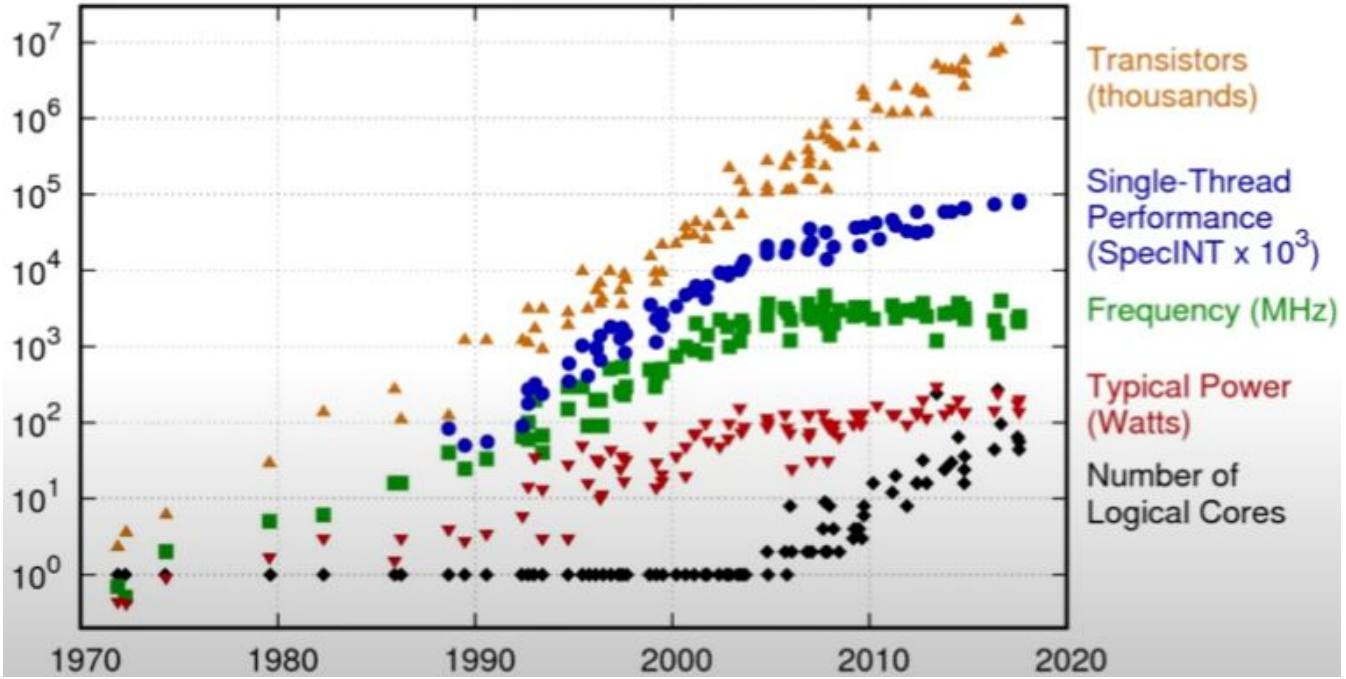
"You don't care how long it takes. Great. But **people who don't care** how long it takes is also **the reason why I have to wait** 30 seconds for Word to boot."

Мотивация



http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Тренды в архитектуре процессоров



https://youtu.be/ICKIMHCw--Y

Промежуточные выводы

1. Использованные данные кэшируются, кэш бывает нескольких уровней
2. Кэш сохраняет данные в “линию” (cache line), используя хэш-таблицу с открытой адресацией, количество “бакетов” сильно ограничено
3. Механизм угадывания пытается заранее определить линию, которая понадобится процессору
   a. кэшируется то, что понадобится в будущем (temporal locality)
   b. соседние байты тоже кэшируются на всякий случай (spatial locality)
4. Кэш не всегда общий для разных ядер/потоков
5. Не все типы данных одинаково хорошо кэшируются

https://youtu.be/Nz9SiF0QVKY

Хитрости чтобы выжать все до капли из процессора

Минимальный элемент без ветвления (останется не обнуленным):

```
int min(int a, int b) {
    return a * (a < b) + b * (b <= a);
}
```

```
int min(int a, int b) {
    // precondition: INT_MIN <= (a - b) <= INT_MAX
    return b + ((a - b) & ((a - b) >> (sizeof(int) * CHAR_BIT - 1)));
}
```

https://graphics.stanford.edu/~seander/bithacks.html

Двоичный поиск без рекурсии:

```
size_t bsearch(size_t needle, const size_t haystack[16]) {
    size_t i = (haystack[8] <= needle) ? 8 : 0;
    i += (haystack[i + 4] <= needle) ? 4 : 0;
    i += (haystack[i + 2] <= needle) ? 2 : 0;
    i += (haystack[i + 1] <= needle) ? 1 : 0;
    return i;
}
```

Выбор без ветвления:

```
#define  BRANCHLESS_IF(f,x) ((x) & -((typeof(x))!!(f)))

#define  BRANCHLESS_IF_ELSE(f,x,y)  (((x) & -((typeof(x))!!(f))) |
\
                                     ((y) & -((typeof(y)) !(f))))
```

Почему так не надо делать

```
void filter_loop(const std::vector<Type>& data, ...) {
    Type sum = 0;
    // ...
    {
        for(auto x : data) {
            if(x < 6) {
                sum += x;
            }
        }
    }
}
```

Современный компилятор знает лучше вас

Type = float:

```
pxor    xmm1, xmm1
movss   xmm2, DWORD PTR .LC4[rip]
mov     rax, rcx
cmp     rcx, rdx
je      .L9
movss   xmm0, DWORD PTR [rax]
comiss  xmm2, xmm0      ; if(x < 6)
jbe     .L10
addss   xmm1, xmm0      ; sum += x
```

Type = int:

```
xor     ebx, ebx
mov     rax, r8
cmp     r8, rdi
je      .L9
mov     edx, DWORD PTR [rax]
cmp     edx, 6          ; sum += x
lea     ecx, [rbx+rdx]
cmovl   ebx, ecx
add     rax, 4
cmp     rdi, rax
jne     .L11
```

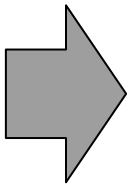
https://godbolt.org/z/nCx7st

DOD vs. OOD

```
struct Shape {
    Shape(const Shape&) = delete;
    Shape& operator=(const Shape&) = delete;
    virtual ~Shape() = 0;
    virtual void draw(Window&) const = 0;
    virtual float area() const = 0;
private:
    Color c;
    bool is_visible;
};

struct Circle : Shape {
    void draw(Window &w) const override {
        if(is_visible()) { ... }
    }
};

struct Square : Shape { ... };
```



```
std::vector<std::unique_ptr<Shape>> data;

for(auto& ptr : data)
    ptr->draw(window);
```

```
struct Button {
    std::string text;
    std::unique_ptr<Shape> shape;
};
```

```
struct Button {
    std::string text;
    std::variant<Circle, Square> shape;
};
```

DOD vs. OOD

- Вызовы извне данных в кэше (pointer indirection)
- Объем кэша используется неэффективно (alignment, padding)
- Виртуальные вызовы внутри работающего цикла (“hot” code)
- Объекты **is_invisible()** тоже обрабатываются

Объекты в памяти должны быть маленькими

```
int a;
bool b;
struct { int a; };
struct { bool b; };
union { int a; bool b; }
struct { int a; bool b; } // массив этих структур?
struct { int a; int *p; int b; }
struct { int a; int b; int *p; }
struct { int a; int b; int *p; bool b; }
```

https://vimeo.com/649009599

Объекты в памяти должны быть маленькими

```
int a;                                          // 4, 4
bool b;                                         // 1, 1
struct { int a; };
struct { bool b; };
union { int a; bool b; }                        // 4, 4
struct { int a; bool b; }                       // 4, 8
struct { int a; int *p; int b; }                // 8, 24
struct { int a; int b; int *p; }                // 8, 16
struct { int a; int b; int *p; bool b; }        // 8, 24
```

Относительная адресация против абсолютной

Трюк 1. Пользуйтесь целыми числами вместо указателей (x2 экономия памяти)

```
// вместо хранения конкретного адреса:
struct s { String *a; String *b; };
s.a = malloc(sizeof(String)); s.b = malloc(sizeof(String));

// хранится расстояние от одного объекта до другого:
struct s { unsigned a; unsigned b; }
String n[2]; s.a = 0; s.b = 1;
```

https://floooh.github.io/2018/06/17/handles-vs-pointers.html

Состояние объекта вне данных (out of band)

Трюк 2. Вместо флагов в данных, группируйте сами данные в таблицах

```
// Булевое поле вынуждает терять 3 байта на каждом объекте:
struct s { int a; bool b; }
s arraylist[100] = {}; // sizeof(arraylist) == 800
// сама информация может храниться без булевого поле:
struct s { int a; }
s a_list[50] = {};     // sizeof(a_list) == 200
s b_list[50] = {};     // sizeof(b_list) == 200
```

* Проверки флага тоже не нужны при такой схеме хранения

Отдельно про массивы

Трюк 3. Вместо массива разнотипных данных – структура всех данных, организованных по типу

```
enum v { fast, slow };
struct s { int *a; v b; } // sizeof(s) == 16
s arraylist[100] = {};    // sizeof(arraylist) == 1600
// информация о классификации может быть частью структуры:
struct s { int *as[100]; v vs[100]; };
s multiarray = {};        // sizeof(s) == 800 + 400
```

* Шаблон “Structure of Arrays” (Multi-array в некоторых языках)

Ассоциативные массивы

Трюк 4. Если составное поле объекта часто пустует (**sparse arrays**), его можно хранить отдельно в ассоциативном массиве

```
// вместо хранения пустых списков в объекте:
struct s { int a; int b; int list[4]; } // sizeof(s) == 24
s arraylist[100] = {};          // sizeof(arraylist) == 2400
// можно хранить в другой таблице по ключу:
struct s { int a; int b; };             // sizeof(s) == 8
s arraylist[100] = {};          // sizeof(arraylist) == 800
map<unsigned, int[4]> m = {};  // sizeof(m) == [0, (20 * n)]
```

* Не считая служебной информации самого контейнера map<>

Чуть более сложный пример

```
enum Color { yellow, red };
struct Shape { Color c; };                                  // 4
struct Window { int x; int y; bool visible; };              // 12
struct Popup { int i; union { Shape a; Window b; } j; }; // 16
```

Наследование?

```
enum Color { yellow, red };
struct Shape { Color c; };                                  // 4
struct Window { int x; int y; bool visible; };              // 12
struct Popup { int i; union { Shape a; Window b; } j; };    // 16
```

```
enum Color { yellow, red };
struct Popup { int i; enum { window, shape } tag; };        // 8
struct Shape { Popup base; Color c; };                      // 12
struct Window { Popup base; int x; int y; bool visible; }; // 20
```

Кодирование информации

```
enum Color { yellow, red };
struct Popup { int i; enum { window, shape } tag; };        // 8
struct Shape { Popup base; Color c; };                      // 12
struct Window { Popup base; int x; int y; bool visible; }; // 20
```

```
struct Base { int i; int index; };          // 8
struct Popup { Base b; enum {
    shape_yellow,
    shape_red,
    window_visible,
    window_hidden } tag; };                 // 12
struct WindowIrregular { int x; int y; }; // 8
```

DOD vs. OOD (продолжение)

```
struct Circle {
    Point center;
    float radius;
};

struct Square {
    Point top;
    float size;
};

struct Shape {
    int type;
    size_t index;
};

struct Shapes {
    std::vector<Circle> circles;
    std::vector<Squares> squares;
};
```

```
float area(const Shapes& geometry) {
    float res = 0;
    for(auto& c : geometry.circles) ...
    for(auto& s : geometry.squares) ...
    return res;
}
```

```
struct ShapesRender {
    std::vector<std::pair<Shape, Color>> visible;
};

void draw(Window& window, const ShapesRender& render,
          const Shapes& geometry)
{
    for(auto [id, color] : render.visible)
        draw(window, geometry.circles[id.index], color);
        draw(window, geometry.square[id.index], color);
}
```

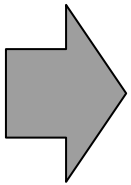
Structure of arrays (SoA)

```
struct Circle {
    Point center;
    float radius;
};

struct Square {
    Point top;
    float size;
};

struct Shape {
    int type;
    size_t index;
};

struct Shapes {
    std::vector<Circle> circles;
    std::vector<Squares> squares;
};
```



```
struct ShapesRender {
    std::vector<std::pair<Shape, Color>> visible;
};

struct ShapesRender {
    std::vector<int> type;
    std::vector<size_t> index;
    std::vector<Color> color;
};
```

Выводы

- Облегчайте процесс кэширования – храните данные в простых блоках байтов (contiguous memory)
- Группируйте данные в зависимости от порядка доступа к ним (packed cache space)
- Инструкции тоже кэшируются – редко используемый (cold) код должен вызываться отдельно от часто используемого (hot) кода
- Структуры массивов – ваши друзья


```zig
const std = @import("std");
const parseInt = std.fmt.parseInt;

test "parse integers" {
    const input = "123 67 89,99";
    const ally = std.testing.allocator;

    var list = std.ArrayList(u32).init(ally);
    // Ensure the list is freed at scope exit.
    // Try commenting out this line!
    defer list.deinit();

    var it = std.mem.tokenizeAny(u8, input, " ,");
    while (it.next()) |num| {
        const n = try parseInt(u32, num, 10);
        try list.append(n);
    }

    const expected = [_]u32{ 123, 67, 89, 99 };

    for (expected, list.items) |exp, actual| {
        try std.testing.expectEqual(exp, actual);
    }
}
```

https://ziglang.org/

https://youtu.be/NWMx1Q66c14