Экзаменационная работа

“Язык программирования Zig”

Выполнил студент группы Б20-505
Сорочан Илья

Московский Инженерно-Физический Интститут
Москва 2023

1 Введение

1.1 История и цель создания

Язык программирования Zig был создан в 2015 году американским программистом Эндрю Келли (Andrew Kelley) с целью предоставить разработчикам гибкий и мощный инструмент для создания высокопроизводительных системных приложений, который сочетает в себе простоту и эффективность.

Не смотря на свою ”свежесть он уже может быть использован для написания программного обеспечения. Одним из примеров служит bun, пакетный менеджер JavaScript, написанный на Zig. Согласно заявлениям разработчика это помогло уменьшить количество потребляемой памяти и ускорить работу программы по сравнению с остальными пакетными менеджерами.

Так же автор Zig помимо полноценной работы над языком участвует в конференциях. Одной из них является GOTO 2022.

1.2 Задачи, решаемые языком

Язык программирования Zig подходит для разработки высокопроизводительных системных приложений, таких как операционные системы, драйверы устройств, сетевые приложения, компиляторы, библиотеки и другие схожие системные проекты. Язык сочетает в себе простоту и эффективность, что является несомненным преимуществом.

Однако, Zig плохо подходит для разработки приложений, которые требуют быстрого прототипирования, так как язык является системным и тот уровень работы с ”железом который он предоставляет может не понадобиться или даже мешать.

Также стоит учитывать, что Zig является относительно новым языком программирования, и на данный момент у него ограниченное количество библиотек и инструментов. Это частично исправляется тем, что Zig может использовать библиотеки C (импортируются в одну строчку) и тем, что сообщество разработчиков языка активно работает над их созданием.

1.3 Производительность в тестах

Результаты тестов Zig неоднозначны. С одной стороны среди трех тестов в которых он присутствует (1.1, 1.2, 2.1) только в тесте 1.1 он занимает второе место – в остальных первое. С другой стороны, сравнивать производительность компилируемого и скриптового языков не корректно (Python, JavaScript и пр.), так как у первого есть преимущество. В то же время языки, с которыми чаще всего сравнивают Zig (C, C++, Rust) в тестах отсутствуют.

Единственный вывод, который можно сделать на основе этих данный это то, что Zig не является настолько неоптимизированным, что проигрывает интерпретируемым языкам.

2 Исследование языка

Параметры системы:
OS: *Manjaro Linux x86_64*
CPU: *6-core AMD Ryzen 5 4500U*
Версия компилятора: *0.11.0-dev.3704+729a051e9* (новейшая на данный момент)

Компиляция файла *main.zig* без оптимизации:

```
zig build-exe main.zig
```

С оптимизацией:

```
zig build-exe -O ReleaseFast -fstrip main.zig
```

2.1 Соглашения о вызовах функций

Возьмем следующую функцию в качестве примера:

```
fn function(b: usize) usize {
    return b + 1;
}
```

Тогда главный цикл будет иметь следующий вид:

```
var a: usize = 0;
for (0 .. 1000000000) |b|
    a = function(b);
```

Соглашение о вызове функции function можно задать в её объявлении следущим образом:

```
fn function(b: usize) callconv(.C) usize {
    return b + 1;
}
```

Здесь используется ключевое слово *callconv*, которому передается одна из констант *std.builtin.CallingConv*
Имя перечисления можно опустить (особенность языка). Среди доступных перечислений имеются (**не все элементы**):

- *C* – совместима с Си, она же *cdecl*;
- *Naked* – функция без пролога и эпилога. Не может вызываться из самого Zig, но может быть полезна при интегрировании в ассемблер;
- *Inline* – функция встраивается в место вызова;
- *Stdcall*, *Fastcall*, *Win64* и многие другие

При оптимизированной сборке данная программа работает за 0 милисекунд (наносекунды). При этом функция main отсутствует. Полагаю это связано с тем, что компилятор Zig старается исполнить как можно больше кода во время компиляции. Поэтому для проверки производительности вызовов будет использоваться отладочная сборка.

Соглашений о вызовах очень много, тем более, что некоторые из них поддерживают только определенную архитектуру процессора (*APCS*, *Kernel* и тд). Вот среднее время (ms) за 10 запусков для некоторых из них:

- *Undefined* (не указано) – 2820;

- *C* – 2820;
- *Win64* – 2820;
- *SysV* – 2819;
- *Inline* – 2058;

В тестах не участвовали *Stdcall* и *Fastcall*, так как они не поддерживают разрядность *x86_64*. Для того что бы получить 32-битный исполняемый файл, необходимо указать флаг *-target x86-linux-gnu*. С помощью него можно указывать цель компиляции (в том числе и кросс-компилировать). В моем случае OS linux и abi GNU. Аналогично проведем 10 запусков и подсчитаем среднее время (ms):

- *Undefined* (не указана) – 2818;
- *C* – 2817;
- *Stdcall* – 2995;
- *Fastcall* – 2991;
- *Inline* – 1950;

2.2 Ключевое слово *const*

В языке программирования Zig ключевое слово конст используется для объявления новых переменных следующим образом:

```
const a = 0;
const b: i32 = 1;
```

Тип константы можно не задавать, однако тип зачастую требуется при инициализации новой изменяемой переменной:

```
var a = 0;       // ERROR
var b: i32 = 1; // OK
```

При этом, если изменяемая переменная не изменяется, то компилятор так же выдаст ошибку:

```
var a: usize = 0;
var l = 1000000000; // error: variable must be const or comptime
for (0 .. l) |b|
    a = function(b);
std.debug.print("{}\n", .{a});
```

Однако, стоит отметить, что это требование можно обойти с помощью вызова функции. Например:

```
fn num() callconv(.Inline) usize {
    return 1000000000;
}

// some code

var l = num(); // OK
comptime l = num(); // OK
var d = std.math.pow(usize, 10, 9); // OK
```

Все же я полагаю это временной недоработкой языка и так как согласно идеологии и инструметам Zig как можно больше вычислений следует оставлять компилятору, то полагаю, что сравнивать исполнение с *const* и без него не имеет смысла. Потому что *const* следует использовать по возможности всегда.

2.3 Разыменовывание указателя

Пусть имеется следующая структура:

```
const Position = struct {
    x: i32,
    y: i32,
    z: i32
};
```

И функции, её меняющие:

```
fn function1(p: Position) Position {
    return .{
        .x = p.x + 1,
        .y = p.y,
        .z = p.z
    };
}

fn function2(p: *Position) void {
    p.x += 1;
}
```

Даже функция с копированием (хотя, почему это не так важно будет пояснено позже) в **отладочном** режиме выдает 0 мс (100 запусков). Следовательно в данном случае различия пренебрежительно малы.

Усложним структуру:

```
const Params = struct {
    n1: i32,
    n2: u32,
    n3: i32,
    n4: f32,
    n5: i64,
    n6: f32,
    n7: i32,
    n8: f32,
    n9: i16,
    n10: i32,
    n11: i64,
    n12: u32,
    n13: i32,
    n14: f64,
    n15: i32,
    n16: u32,
};
```

Однако ситуация никак не изменилась. Это связано с тем, что Zig сам выбирает как эффективнее передать структуру – по значению (скопировать) или по указателю.

2.4 Векторизация операций

Zig поддерживает паралельную работу с векторами булей, целых и дробных чисел и указателей при помощи SIMD инструкций. Они создаются при помощи функции *@Vector* и поддерживают следующие операции:

- Арифметические: +, -, /, *, @sqrt, @log и пр.;
- Побитовые: », «, &, |, , и пр.;
- Сравнения: >, <, = и пр.

Так же есть функции для работы со всем вектором:

- @splat – делает вектор из одного скаляра (повтор n раз);
- @reduce – использует оператор для получения скаляра из вектора. Доступны арифметические и логические операторы, а так же взятие максимума и минимума;
- @shuffle – собирает один вектор из нескольких с помощью специальной маски;
- @select – объединяет два вектора в один с помощью предиката (вектор булей).

Важно отметить, что если целевая платформа не поддерживает SIMD инструкции, то операции над каждым элеметом вектора выполняются последовательно.

Пример использования векторов:

```
const a = @Vector(4, i32){1, 2, 3, 4};
const b = @Vector(4, i32){5, 6, 7, 8};
const c = a + b; // c = [6, 8, 10, 12]
```

Приложение А

Использованные программные коды

Программа для замера времени соглашений о вызовах:

```
const std = @import("std");

const RUNS = 10;
const CALLCONV = std.builtin.CallingConvention.Inline;

fn function(b: usize) callconv(CALLCONV) usize {
    return b + 1;
}

pub fn main() !void {
    var sum: u64 = 0;
    for (0 .. RUNS) |_| {
        var timer = try std.time.Timer.start();
        var a: usize = 0;
        for (0 .. 1000000000) |b|
            a = function(b);
        sum += timer.read() / std.time.ns_per_ms;
    }
    std.debug.print("{d}\n", .{sum / RUNS});
}
```

Программа, иллюстрирующая *const*:

```
const std = @import("std");

fn function(b: usize) callconv(.Inline) usize {
    return b + 1;
}

fn num() callconv(.Inline) usize {
    return 1000000000;
}

pub fn main() !void {
    var a: usize = 0;
    // var l = 1000000000;
    comptime var l = num();
    for (0 .. l) |b|
        a = function(b);
    std.debug.print("{}\n", .{a});
}
```

Передача простых структур по ссылке и по значению:

```
const std = @import("std");
```

```
const RUNS = 1000;

const Position = struct {
    x: i32,
    y: i32,
    z: i32
};

fn function1(p: Position) Position {
    return .{
        .x = p.x + 1,
        .y = p.y,
        .z = p.z
    };
}

fn function2(p: *Position) void {
    p.x += 1;
}

pub fn main() !void {
    var sum: u64 = 0;
    for (0 .. RUNS) |_| {
        var timer = try std.time.Timer.start();

        var p: Position = undefined;
        p = std.mem.zeroInit(Position, p);

        p = function1(p);
        // function2(&p);

        sum += timer.read() / std.time.ns_per_ms;
    }
    std.debug.print("{d}\n", .{sum / RUNS});
}
```

Передача больших структур по ссылке и по значению:

```
const std = @import("std");

const RUNS = 1000;

const Params = struct {
    n1: i32,
    n2: u32,
    n3: i32,
    n4: f32,
    n5: i64,
    n6: f32,
    n7: i32,
    n8: f32,
    n9: i16,
    n10: i32,
```

```
    n11: i64,
    n12: u32,
    n13: i32,
    n14: f64,
    n15: i32,
    n16: u32,
};

fn function1(p: Params) Params {
    var q: Params = p;
    q.n14 = q.n8 + 4;
    return q;
}

fn function2(p: *Params) void {
    p.n14 = p.n8 + 4;
}

pub fn main() !void {
    var sum: u64 = 0;
    for (0 .. RUNS) |_| {
        var timer = try std.time.Timer.start();

        var p: Params = undefined;
        p = std.mem.zeroInit(Params, p);

        // p = function1(p);
        function2(&p);

        sum += timer.read() / std.time.ns_per_ms;
    }
    std.debug.print("{d}\n", .{sum / RUNS});
}
```