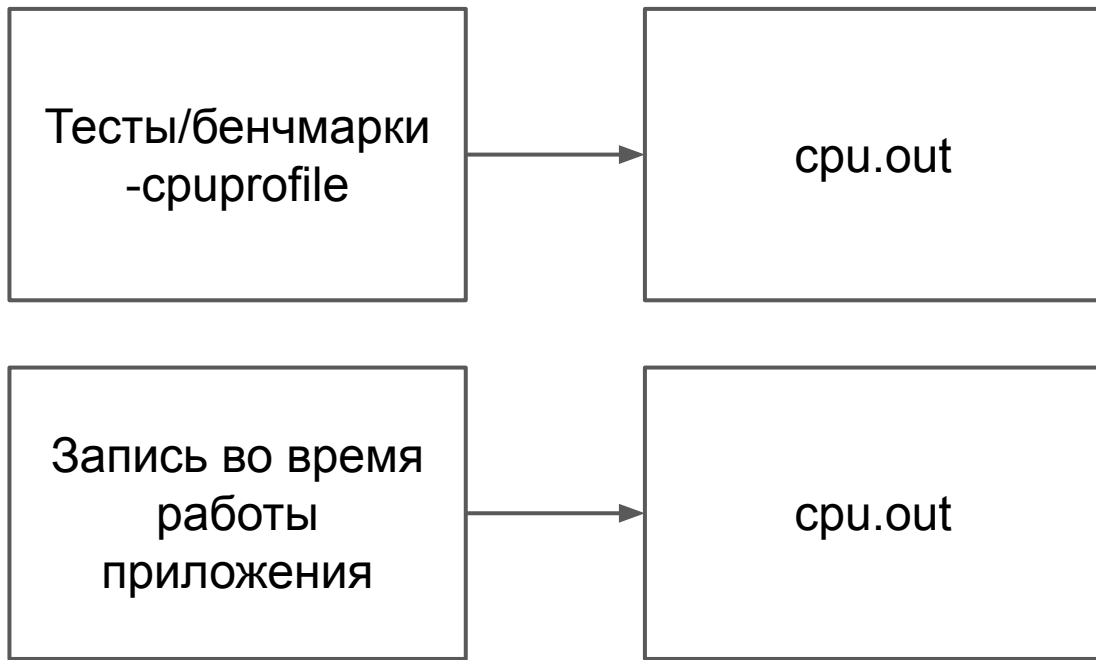
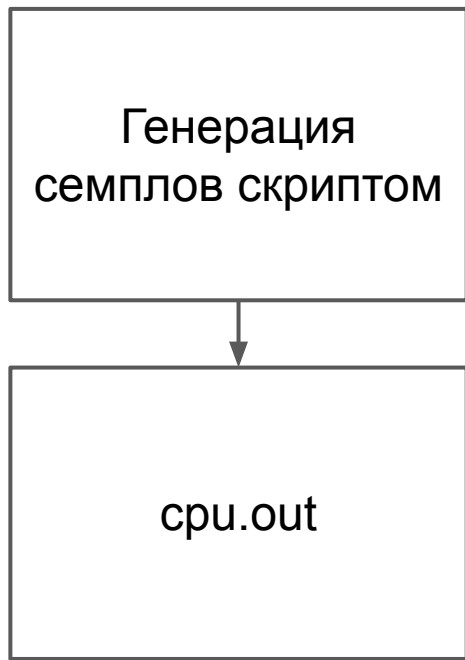
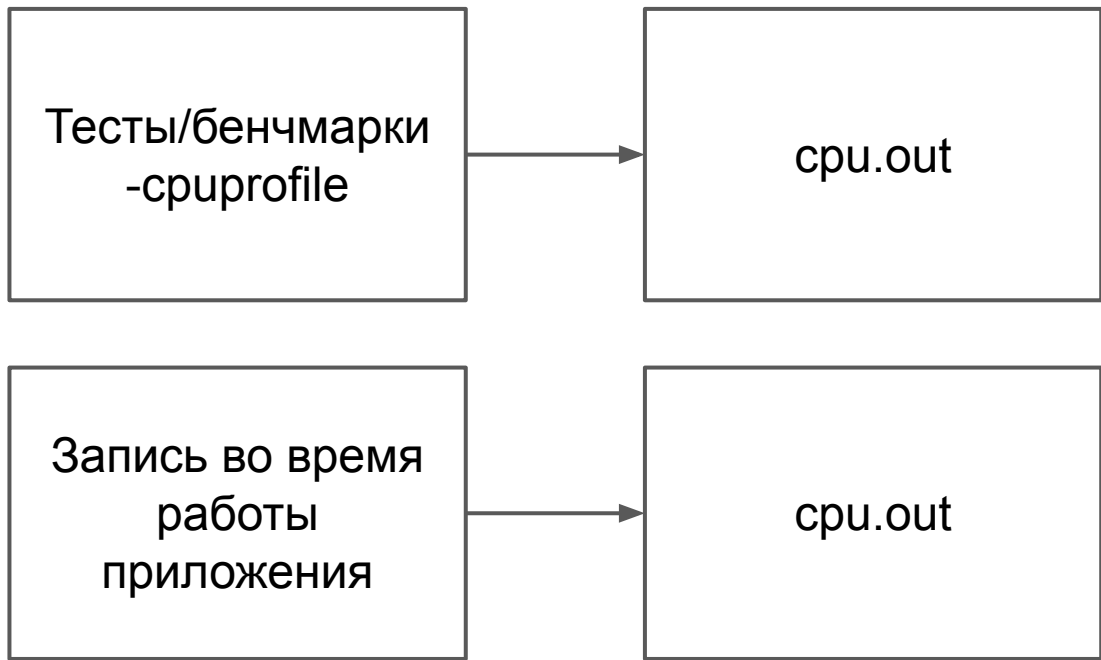


Работаем с CPU профилями как с данными  
@quasilyte 2022

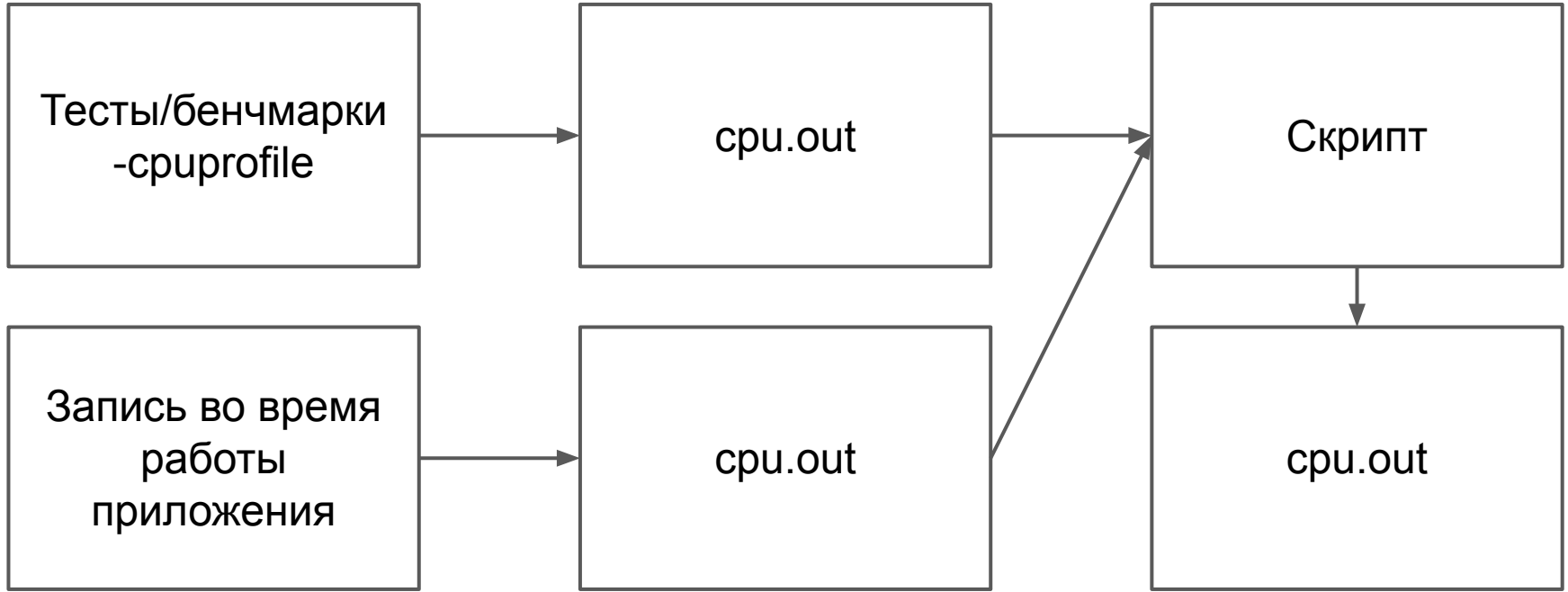
Откуда берутся CPU профили?



Источники профилей



Источники профилей



Источники профилей

Анализ через встроенные средства

```
func copyBytes(b []byte) []byte {  
    dst := make([]byte, len(b))  
    copy(dst, b)  
    return dst  
}
```

```
func copyBytes(b []byte) []byte {  
    dst := make([]byte, len(b))  
    copy(dst, b)  
    return dst  
}
```

```
func BenchmarkCopyBytes(b *testing.B) {  
    dst := make([]byte, 2022)  
    for i := 0; i < b.N; i++ {  
        copyBytes(dst)  
    }  
}
```



```
func copyBytes(b []byte) []byte {  
    dst := make([]byte, len(b))  
    copy(dst, b)  
    return dst  
}  
func BenchmarkCopyBytes(b *testing.B) {  
    dst := make([]byte, 2022)  
    for i := 0; i < b.N; i++ {  
        copyBytes(dst)  
    }  
}
```

```
go test -bench=. -count=20 -cpuprofile=cpu.out
```

(pprof) top 5

(pprof) top 5

37.56% runtime.mallocgc

12.16% runtime.memclrNoHeapPointers

9.35% runtime.memmove

8.47% runtime.scanobject

6.42% runtime.scanblock

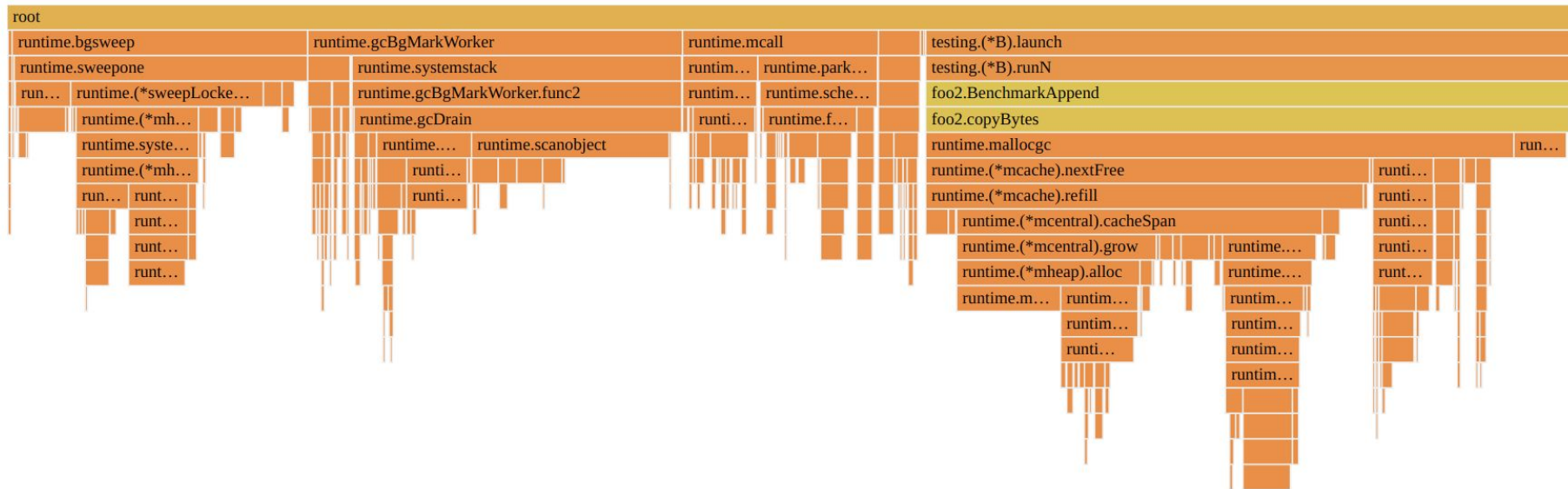
А где copyBytes?

```
(pprof) top 5  
37.56% runtime.mallocgc  
12.16% runtime.memclrNoHeapPointers  
9.35% runtime.memmove
```

```
func copyBytes(b []byte) []byte {  
    dst := make([]byte, len(b))  
    copy(dst, b)  
    return dst  
}
```

- **mallocgc** (аллокация)
- **memclrNoHeapPointers** (зачистка памяти)
- **memmove** (копирование)

# Флеймграф - это частичное решение



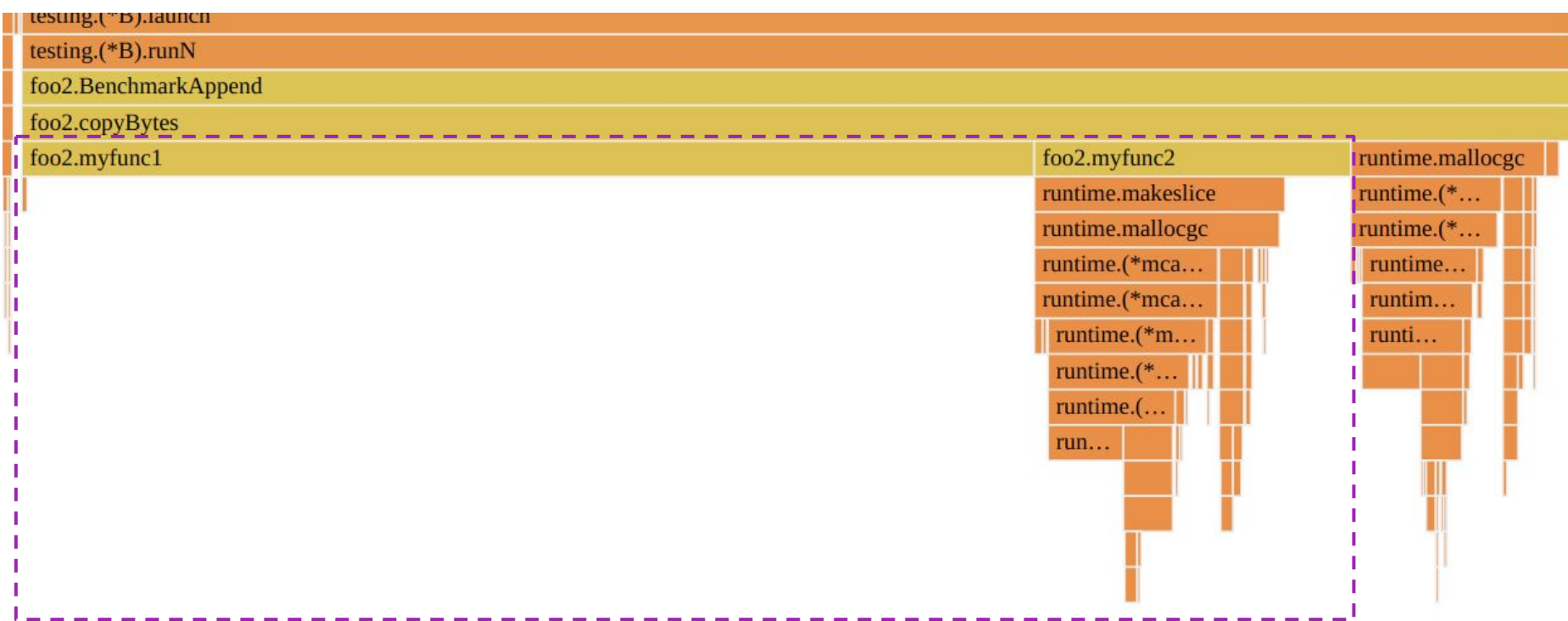
# Флеймграф - это частичное решение



```
func copyBytes(b []byte) []byte {  
    myfunc1(len(b) * 2)  
    myfunc2(193)  
    dst := make([]byte, len(b))  
    copy(dst, b)  
    return dst  
}
```

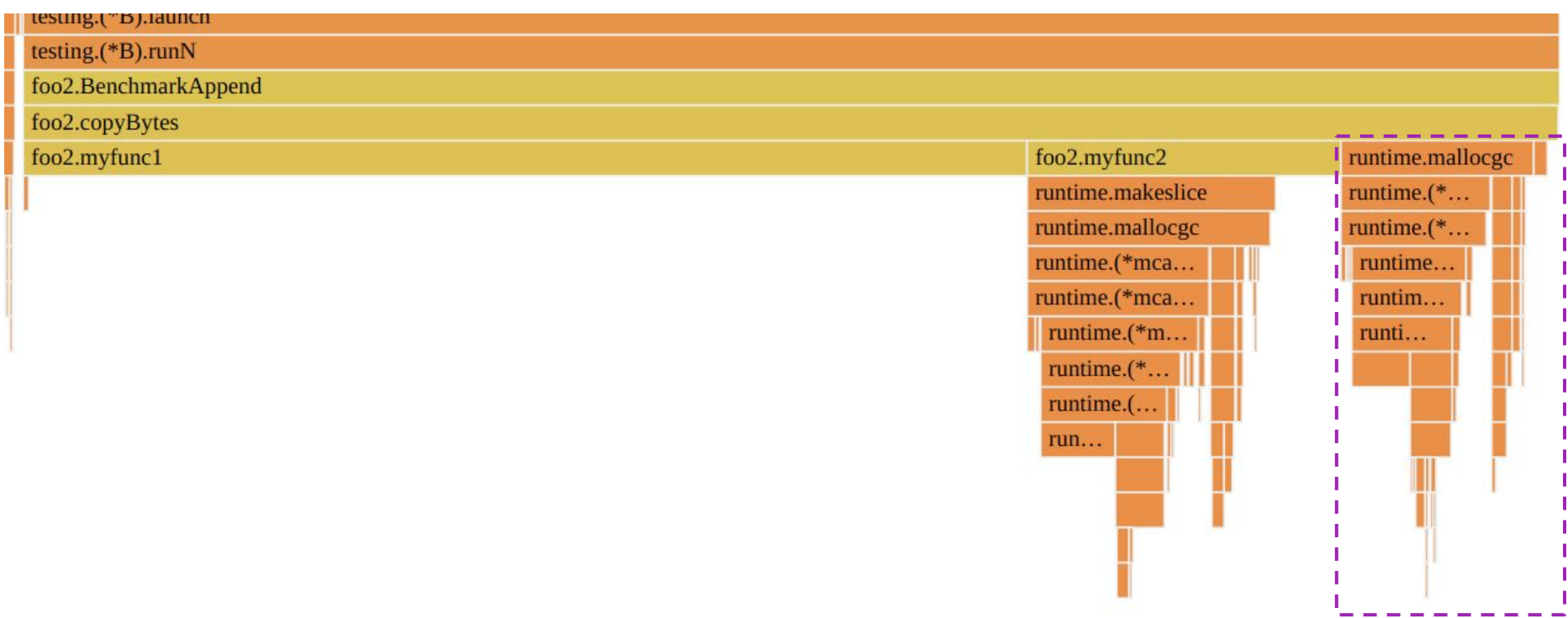






Вот эти функции нам не интересны.

Top -sum имеет тот же недостаток: он будет включать время работы myfunc1 и myfunc2 в copyBytes.



А вот интересующая нас часть.

Даже если данные в профиле репрезентативны,  
мы должны уметь правильно их анализировать.

А ещё важно задавать правильные вопросы.

“Как мне изменить *мой* код, чтобы приложение стало быстрее?”

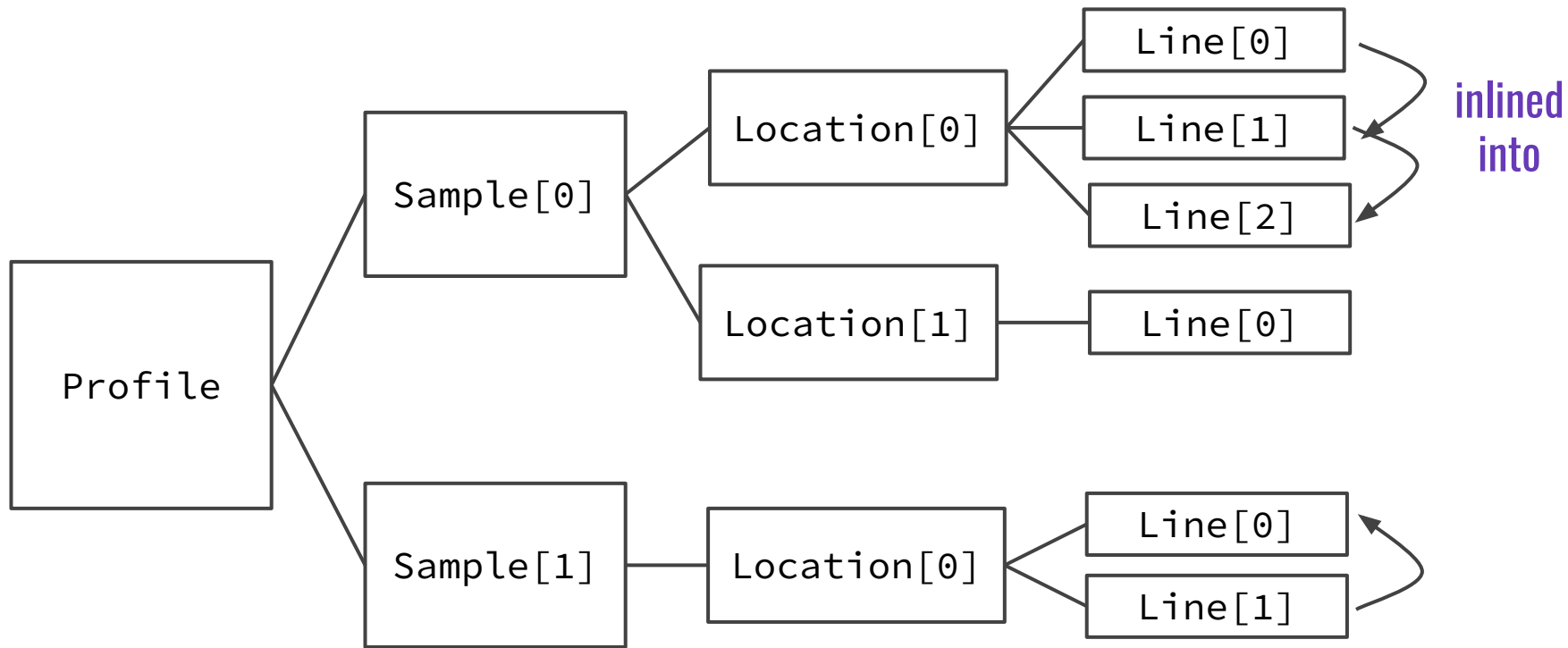
Нас редко интересуют детали рантайма, ведь мы не можем поменять их реализацию.

Строим свой top

# Алгоритм

1. Разобрать CPU профиль парсером
2. Произвести агрегацию по своему критерию
3. Вывести top-N результатов после сортировки

[github.com/google/pprof](https://github.com/google/pprof) - парсер (и клиент pprof)



Организация семплов в профиле

```
for _, sample := range p.Samples {  
    for _, loc := range sample.Location {  
        for _, l := range loc.Line {  
            sampleValue := sample.Value[1] // time/ns  
            funcName := l.Function.Name  
            filename := l.Function.Filename  
            line := l.Line  
            println(filename, line, funcName, sampleValue)  
        }  
    }  
}
```

Обход семплов: наивный подход



```
var stack []profile.Line
for _, sample := range p.Samples {
    stack = stack[:0] // reuse memory for every stack
    for _, loc := range sample.Location {
        stack = append(stack, loc.Line...)
    }
    for i, l := range stack {
        // handle l
    }
}
```

Обход семплов: улучшенный способ

```
var stack []profile.Line
for _, sample := range p.Samples {
    stack = stack[:0]
    for _, loc := range sample.Location {
        stack = append(stack, loc.Line...)
    }
}
```

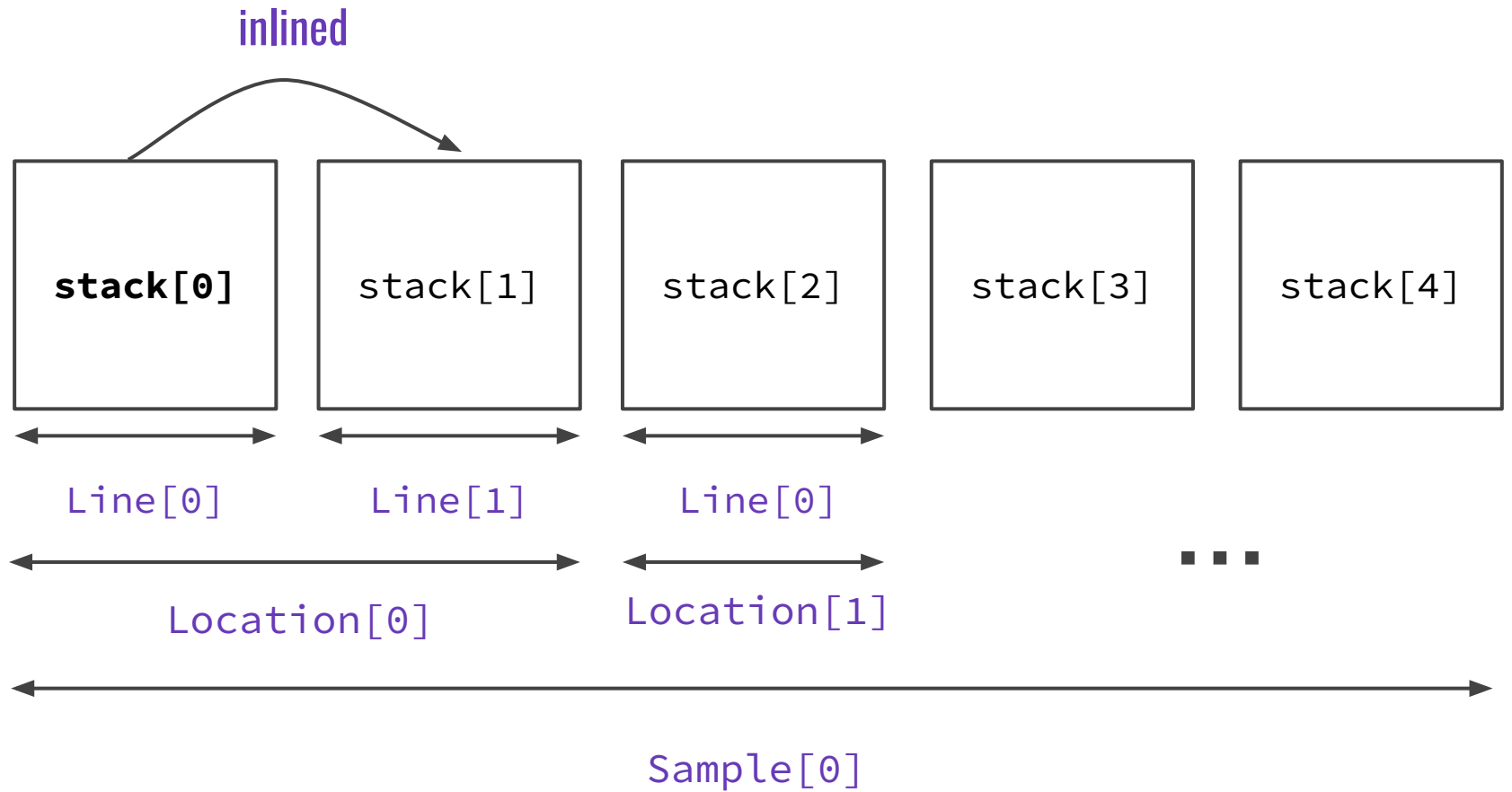
**stack**[0]

“self” значение, текущий фрейм

```
var stack []profile.Line
for _, sample := range p.Samples {
    stack = stack[:0]
    for _, loc := range sample.Location {
        stack = append(stack, loc.Line...)
    }
}
```

**stack**[1:]

Фреймы дальше по стеку - вызывающая сторона



`l.Function.Name`

`=>`

Имена функций и методов

`l.Function.Name`

`=>`

`somefunc`

Имена функций и методов

```
l.Function.Name
```

=>

```
somefunc
```

```
runtime.mallocgc
```

Имена функций и методов

`l.Function.Name`

`=>`

`somefunc`

`runtime.mallocgc`

`github.com/foo/pkg.Bar`

Имена функций и методов



`l.Function.Name`

`=>`

`somefunc`

`runtime.mallocgc`

`github.com/foo/pkg.Bar`

`github.com/foo/pkg.Bar.func1`

Имена функций и методов

`l.Function.Name`

`=>`

`somefunc`

`runtime.mallocgc`

`github.com/foo/pkg.Bar`

`github.com/foo/pkg.Bar.func1`

`github.com/foo/pkg.Bar.func1.2`

Имена функций и методов

`l.Function.Name`

`=>`

`somefunc`

`runtime.mallocgc`

`github.com/foo/pkg.Bar`

`github.com/foo/pkg.Bar.func1`

`github.com/foo/pkg.Bar.func1.2`

`github.com/foo/pkg.(*Bar).Method`

Имена функций и методов

`l.Function.Name`

`=>`

`somefunc`

`runtime.mallocgc`


`github.com/foo/pkg.Bar`

`github.com/foo/pkg.Bar.func1`

`github.com/foo/pkg.Bar.func1.2`

`github.com/foo/pkg.(*Bar).Method`

А здесь вообще неоднозначность:  
Метод `func1` из типа **Bar** или  
первая лямбда из функции **Bar**?



Имена функций и методов

```
// github.com/foo/pkg.(*Bar).Method  
sym := pprofutil.ParseFuncName(l.Function.Name)
```

```
println(sym.PkgPath)    // => github.com/foo/pkg  
println(sym.PkgName)    // => pkg  
println(sym.TypeName)   // => Bar  
println(sym.FuncName)   // => Method
```

sym.TypeName будет пустым для свободных функций

[github.com/quasilyte/pprofutil](https://github.com/quasilyte/pprofutil)

```
var stack []profile.Line
for _, sample := range p.Samples {
    stack = stack[:0] // reuse memory for every stack
    for _, loc := range sample.Location {
        stack = append(stack, loc.Line...)
    }
    for i, l := range stack {
        // stack[0] is a current func
    }
}
```

=>

```
pprofutil.WalkSamples(p, func(s pprofutil.Sample) {
    // s.Stack[0] is a current func
})
```

```
perFunc := make(map[string]int64)

pprofutil.WalkSamples(p, func(s pprofutil.Sample) {
    current := s.Stack[0]
    perFunc[current.Function.Name] += s.Value
})
```

Строим простенький “flat” как в pprof

```
perFunc := make(map[string]int64)

pprofutil.WalkSamples(p, func(s pprofutil.Sample) {
    for _, l := range s.Stack {
        perFunc[l.Function.Name] += s.Value
    }
})
```

Строим простенький “cum” (cumulative) как в pprof



```
type keyValue struct {  
    key string  
    value int64  
}  
sorted := make([]keyValue, 0, len(perFunc))  
for k, v := range perFunc {  
    Sorted = append(sorted, keyValue{key: k, value: v})  
}  
sort.Slice(sorted, func(i, j int) bool {  
    return sorted[i].value > sorted[j].value  
}))
```

Сортируем данные

```
for _, kv := range sorted[:n] {  
    fmt.Printf("%s: %v ns\n", kv.key, kv.value)  
}
```

Выводим top-N

## А зачем нам всё это?

- Произвольная фильтрация семплов
- Любая агрегация или слияние семплов
- Экспорт в любой формат данных
- Хитрый анализ между двумя или более профилей
- Создание своих клиентов для rprof (например для веба)
- Насыщение профилей новой информацией (об этом потом)

И это только в области преобразования семплов!

Runtime-fold flat агрегация


# Алгоритм

1. Сначала строим обычный flat (perFunc)
2. Далее сворачиваем стеки до первого пользовательского
3. Сортируем и выводим информацию по результатам шага 2

\* Два прохода не обязательны, но так будет проще

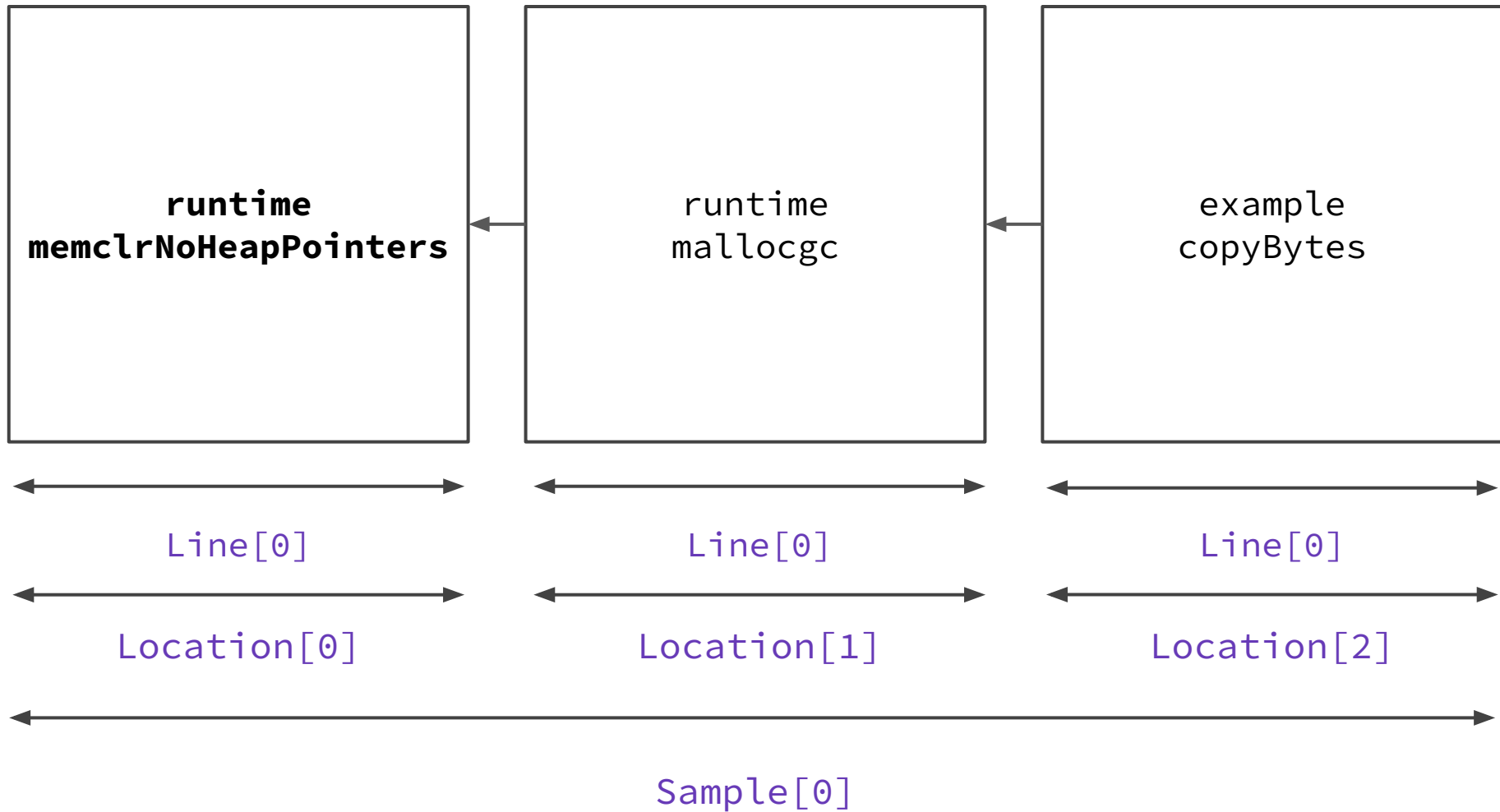
# Runtime-fold flat агрегация

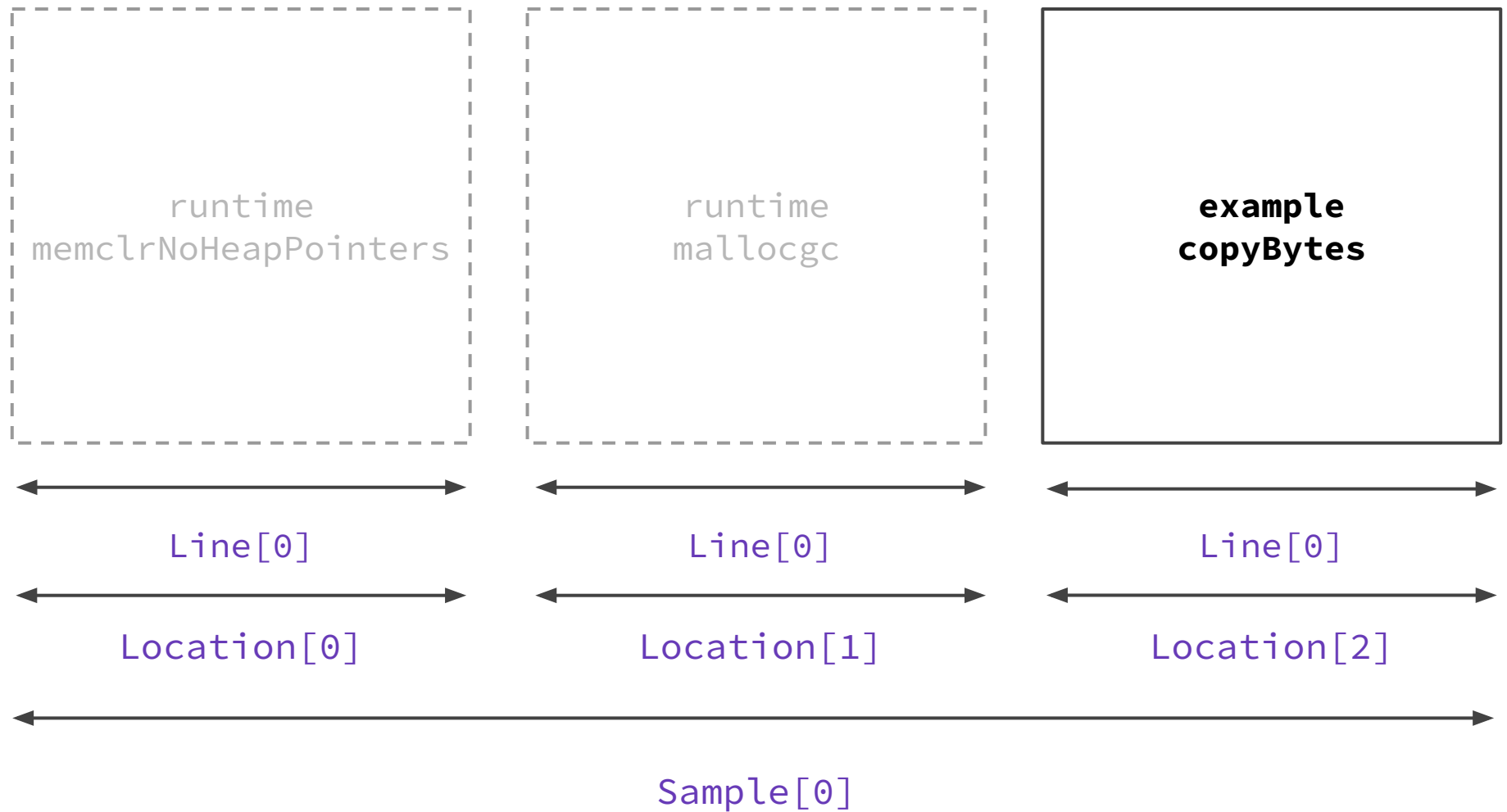
Допустим есть семпл с value=10000000ns:



```
[0] runtime.memclrNoHeapPointers  
[1] runtime.mallocgc  
[2] example.copyBytes  
[3] main.main
```

Мы добавим 10000000ns к **example.copyBytes**

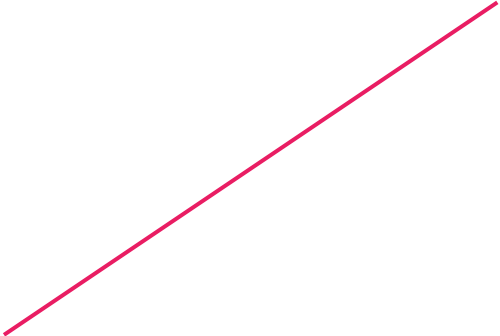






```
func isRuntimeFunc(s pprofutil.Symbol) bool {  
    return s.PkgPath == "runtime" || s.PkgName == ""  
}
```

Функции без пакета (ассемблер)  
будем считать частью рантайма



Предикат для функций из рантайма

```
perFunc := make(map[string]int64)

pprofutil.WalkSamples(p, func(s pprofutil.Sample) {
    current := s.Stack[0]
    perFunc[current.Function.Name] += s.Value
})
```

Проход 1: строим обычный flat

```
type pref func(l *profile.Line) bool

func findNode(s pprofutil.Sample, f pref) *profile.Line {
    for i := range s.Stack {
        if f(&s.Stack[i]) {
            return &s.Stack[i]
        }
    }
    return nil
}
```

Хелпер для поиска семплов

```
pprofutil.WalkSamples(p, func(s pprofutil.Sample) {
    current := s.Stack[0]
    l := findNode(s, func(l *profile.Line) bool {
        sym := pprofutil.ParseFuncName(l.Function.Name)
        return !isRuntimeFunc(sym)
    })
    if l == nil {
        return
    }
    perFunc[l.Function.Name] += s.Value
})
```

Проход 2: строим runtime-fold flat

```
(pprof) top 5  
37.56% runtime.mallocgc  
12.16% runtime.memclrNoHeapPointers  
9.35% runtime.memmove  
8.47% runtime.scanobject  
6.42% runtime.scanblock
```

А где copyBytes?

```
./runtime-aware cpu.out
```

```
6.25s example.copyBytes
```

```
40ms example.BenchmarkCopyBytes
```

```
20ms testing.(*B).runN
```

```
func copyBytes(b []byte) []byte {  
    myfunc1(len(b) * 2)  
    myfunc2(193)  
    dst := make([]byte, len(b))  
    copy(dst, b)  
    return dst  
}
```

```
./runtime-aware cpu2.out
```

```
7.39s example.myfunc1
```

```
2.3s example.myfunc2
```

```
1.58s example.copyBytes
```

```
40ms example.BenchmarkCopyBytes
```

```
20ms testing.(*B).runN
```

Добавляем информацию в CPU профили

# Bound checks

Информации о bound checks в CPU профиле в явном виде нет.

Все проверки индексов становятся self значением содержащей функции.

Но мы можем распознать эти фрагменты и добавить их в CPU профиль.

[Анализируем bound checks в Go по CPU профилю](#)



```
SUBQ $24, SP
MOVQ BP, 16(SP)
LEAQ 16(SP), BP
MOVQ BX, xs+40(FP)
```

```
CMPQ CX, AX
```

```
JLS bc_fail
```

```
MOVQ (BX)(AX*8), AX
```

```
MOVQ 16(SP), BP
```

```
ADDQ $24, SP
```

```
RET
```

```
bc_fail:
```

```
CALL runtime.panicIndex(SB)
```

Как выглядит bound check

```
CMPQ DI, BX      // начинается с CMPQ/TESTQ
JB boundcheck    // далее прыжок

// ...
```

### **boundcheck:**

```
MOVQ DI, AX      // передача аргументов
MOVQ BX, CX
CALL runtime.panicSliceB // вызов паники
```

Как распознать bound check

Для этого нам потребуется бинарник программы

cpu.out

+

program.exe

=

better\_cpu.out

А ещё нам нужен дизассемблер для целевой платформы

[golang.org/x/arch/x86/x86asm](https://golang.org/x/arch/x86/x86asm)

## Далее не очень очевидная магия

- Собираем адреса panic-функций из рантайма (debug/elf)
- Выполняем маппинг символа на загруженный адрес
- Для каждого семпла исполняем функцию проверки на bound check
- Функция проверки анализирует машинный код по адресу из семпла
- Для анализа машинного кода мы используем дизассемблер
- Для маппинга адресов используем информацию из CPU профиля

В упомянутой выше статье всё разобрано в подробностях.



# Затем мы добавляем в CPU профиль новые данные

1. Нужно добавить функцию `runtime.boundcheck`
2. Для этой функции нужно добавить семплов в нужных местах

```
// ID начинаются с 1, то есть p.Function[i].ID == i+1
id := len(p.Function) + 1
boundcheckFunc := &profile.Function{
    ID:          uint64(id),
    Name:        "runtime.boundcheck",
    SystemName:  "runtime.boundcheck",
    Filename:    "builtins.go", // Не имеет значения
    StartLine:   1,             // Не имеет значения
}

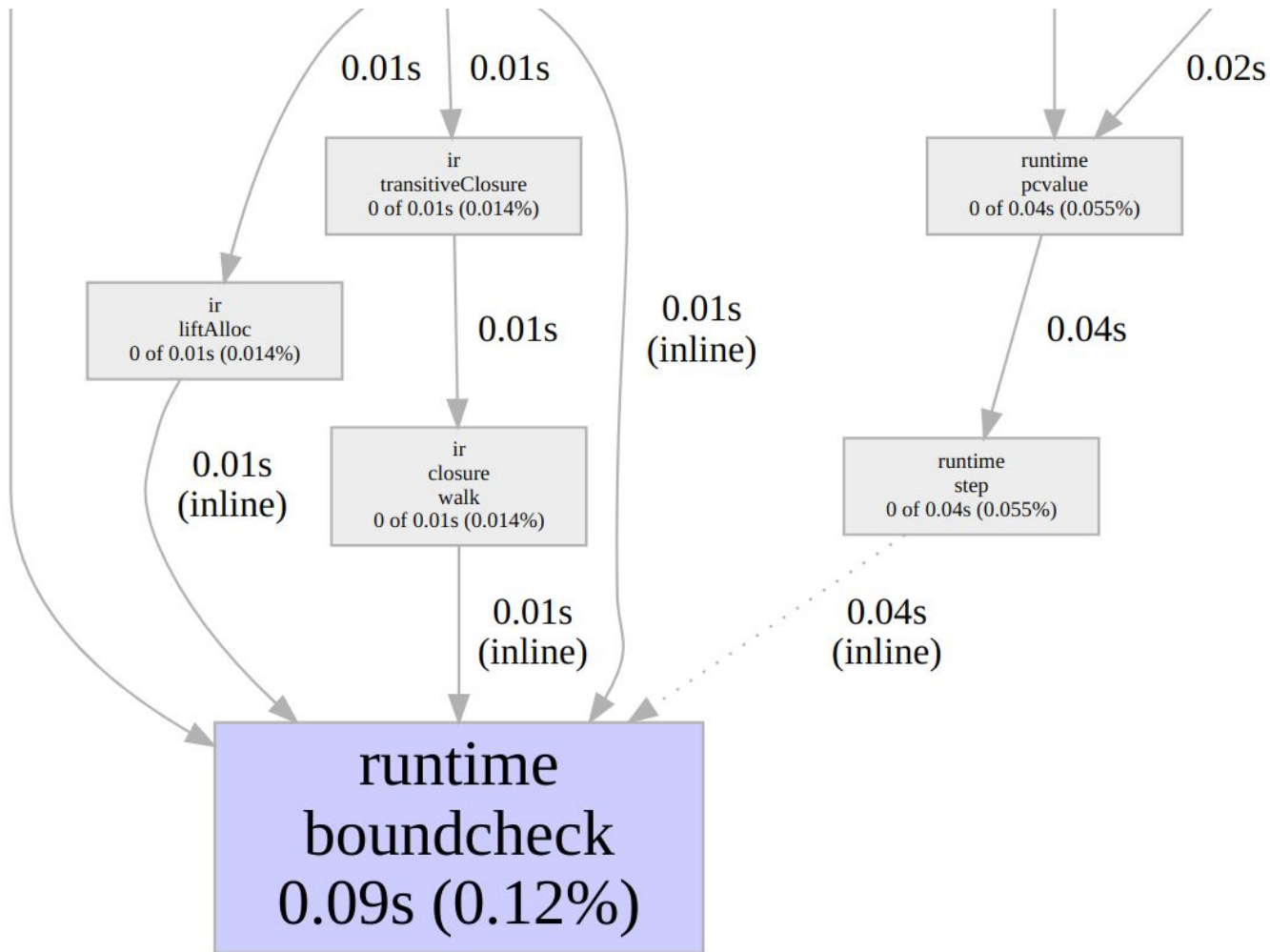
p.Function = append(p.Function, boundcheckFunc)
```

Добавляем функцию runtime.boundcheck

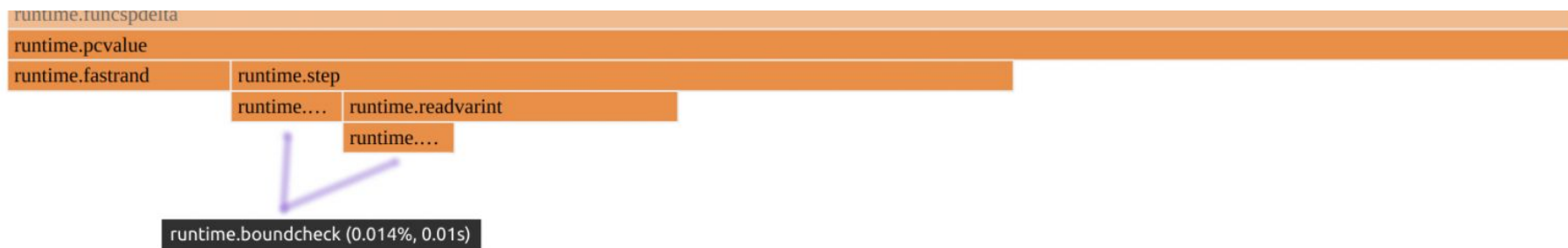


```
func insertLine(loc *profile.Location, fn *profile.Function) {  
    if loc.Line[0].Function == fn {  
        return // Видимо, мы уже вставляли сюда новый вызов  
    }  
  
    newLine := profile.Line{  
        Function: fn,  
        Line:     fn.StartLine,  
    }  
    loc.Line = append([]profile.Line{newLine}, loc.Line...)  
}
```

Функция вставки вызова в стек



0.59s	1.08%	65.23%	0.59s	1.08%	runtime.nextFreeFast (inline)
0.57s	1.04%	66.29%	0.57s	1.04%	runtime.futex
0.48s	0.88%	67.17%	0.49s	0.9%	bufio.(*Reader).ReadByte
0.43s	0.79%	67.96%	1.21s	2.21%	go/scanner.(*Scanner).scanComment
0.42s	0.77%	68.73%	3.41s	6.24%	go/scanner.(*Scanner).Scan
0.39s	0.71%	69.44%	0.39s	0.71%	runtime.markBits.isMarked (inline)
0.35s	0.64%	70.08%	1.41s	2.58%	runtime.greyobject
0.33s	0.6%	70.68%	0.33s	0.6%	runtime.heapBits.bits (inline)
0.30s	0.55%	71.23%	0.30s	0.55%	runtime.nilcheck (inline)
0.29s	0.53%	71.76%	1s	1.83%	go/build.(*importReader).readByte
0.29s	0.53%	72.29%	0.29s	0.53%	runtime.boundcheck (inline)
0.29s	0.53%	72.82%	0.74s	1.35%	runtime.gcWriteBarrier
0.29s	0.53%	73.35%	0.29s	0.53%	runtime.memmove
0.28s	0.51%	73.87%	0.41s	0.75%	runtime.mapaccess1_faststr
0.26s	0.48%	74.34%	1.26s	2.30%	go/build.(*importReader).peekByte



Строим индексы по CPU профилям

```
file.go:100 0.1s  
file.go:100 0.3s  
file.go:120 0.1s  
file.go:120 0.1s  
file.go:100 0.4s  
file.go:130 0.2s  
file.go:140 0.1s  
file.go:145 0.1s  
file.go:150 0.3s  
file.go:165 0.2s  
file.go:170 0.2s
```

Для начала берём все  
семплы из файла

```
file.go:100 0.8s  
file.go:120 0.2s  
file.go:130 0.2s  
file.go:140 0.1s  
file.go:145 0.1s  
file.go:150 0.3s  
file.go:165 0.2s  
file.go:170 0.2s
```

Суммируем семплы для  
одинаковых локаций

file.go:100	0.8s
file.go:150	0.3s
file.go:120	0.2s
file.go:130	0.2s
file.go:165	0.2s
file.go:170	0.2s
file.go:140	0.1s
file.go:145	0.1s

Сортируем семплы по их значениям



file.go:100	0.8s		L=5
file.go:150	0.3s		L=5
file.go:120	0.2s		L=4
file.go:130	0.2s		L=3
file.go:165	0.2s		L=3
file.go:170	0.2s		L=2
file.go:140	0.1s		L=1
file.go:145	0.1s		L=1

Разделяем их на  
категории (5 уровней)

<b>file.go:100</b>	<b>0.8s</b>	<b> </b>	<b>L=5</b>
<b>file.go:150</b>	<b>0.3s</b>	<b> </b>	<b>L=4</b>
<b>file.go:120</b>	<b>0.2s</b>	<b> </b>	<b>L=3</b>
<b>file.go:130</b>	<b>0.2s</b>	<b> </b>	<b>L=2</b>
file.go:165	0.2s		L=0
file.go:170	0.2s		L=0
file.go:140	0.1s		L=0
file.go:145	0.1s		L=0

По параметру threshold  
выбираем top N% записей

Допустим, threshold=0.5

# Где и как можно использовать эти индексы?

- Раскрашивание “горячих” зон в текстовом редакторе или IDE

## Где и как можно использовать эти индексы?

- Раскрашивание “горячих” зон в текстовом редакторе или IDE
- Фильтры для поиска по коду (искать только среди “горячих” мест)

## Где и как можно использовать эти индексы?

- Раскрашивание “горячих” зон в текстовом редакторе или IDE
- Фильтры для поиска по коду (искать только среди “горячих” мест)
- Ранжирование в разрезе локаций кода, а не в рамках символов

# Где и как можно использовать эти индексы?

- Раскрашивание “горячих” зон в текстовом редакторе или IDE
- Фильтры для поиска по коду (искать только среди “горячих” мест)
- Ранжирование в разрезе локаций кода, а не в рамках символов

[github.com/quasilyte/perf-heatmap](https://github.com/quasilyte/perf-heatmap) строит подобный индекс

```
478     start := -1 // valid span start if >= 0
479     for i := 0; i < len(s); {
480         size := 1
481         r := rune(s[i])
482         if r >= utf8.RuneSelf {
483             r, size = utf8.DecodeRune(s[i:])
484         }
485         if f(r) {
486             if start >= 0 {
487                 spans = append(spans, span{start, i})
488                 start = -1
489             }

```

[github.com/quasilyte/vscode-perf-heatmap](https://github.com/quasilyte/vscode-perf-heatmap)

Реклама [go-perfguard](#)



# Что такое perfguard?

Статический анализатор для Go, который использует CPU профили для нахождения горячих мест, которые можно переписать оптимальнее.

Чаще всего для диагностик имеются автофиксы.

Заметим, что perfguard исправляет только те места, которые по профилю выявляются как горячие!

```
for _, name := range names {  
    if ok, _ := regexp.Match("^go", []byte(name)); ok {  
        _, _ = fmt.Fprintln(w, "Hello", name)  
    }  
}
```

Вспомним код из предыдущего доклада

```
quasilyte@lispbook:go_perf_meetup$ perfguard optimize --heatmap cpu.out .  
main.go:126: regexpCompile (8.52s): regexp compilation should be avoided on the hot paths  
Affected samples time: 8.52s  
Found 1 issues (0 auto-fixable)  
quasilyte@lispbook:go_perf_meetup$
```

```
quasilyte@lispbook:go_perf_meetup$ perfguard optimize --heatmap cpu.out .  
main.go:128: regexpStringCopyElim (1.24s): goRegexp.Match([]byte(name)) => goRegexp.MatchString(name)  
Affected samples time: 1.24s  
Found 1 issues (1 auto-fixable)  
quasilyte@lispbook:go_perf_meetup$
```

## Рекомендация для замены типов

`bytes.Buffer => strings.Builder`

Если локальный `bytes.Buffer` использует только совместимое со `strings.Builder` API, а в результате мы делаем `Buffer.String()`, то можно заменить `bytes.Buffer` на `strings.Builder` и избежать лишней аллокации.

## Рекомендации по заменам функций

```
io.WriteString(w, fmt.Sprintf("%s:%d", f, l))
```

=>

```
fmt.Fprintf(w, "%s:%d", f, l)
```

## Удаление лишних копирований

`b = append(b, []byte(s)...) => b = append(b, s...)`

`copy(b, []byte(s)) => copy(b, s)`

`re.Match([]byte(s)) => re.MatchString(s)`

`w.Write([]byte(s)) => w.WriteString(s)`

И многие другие избыточные трансформации данных.

## Перестановка условий

`f() && isOK => isOk && f()`

Выгоднее сначала проверять дешёвые условие без побочных эффектов, и только потом вычислять более дорогие.



`map[T]bool -> map[T]struct{}`

Когда `map[T]bool` используется как множество (set), можно сэкономить немного памяти при использовании `map[T]struct{}`.

## Распознавание идиом (узнаваемых оптимизатором Go)

```
obj.cache = make(map[K]V, len(obj.cache))
```

=>

```
for k := range obj.cache {  
    delete(obj.cache, k)  
}
```

# Как много паттернов в perfguard на данный момент?

Примерно ~150 паттернов через ruleguard

+

7 вручную написанных анализатора

Подводим итоги

# CPU профили можно обрабатывать как данные

Есть библиотеки, позволяющие за 10-20 строк кода выполнить задачу по автоматизации анализа сложного CPU профиля.

Вместо того, чтобы тратить много времени на всматривание в CPU профиль, можно написать скрипт преобразования или воспользоваться утилитой типа qpprof.

## Утилита qpprof

- Runtime-fold flat агрегация
- Stdlib-fold flat агрегация
- Добавление boundcheck метрик профиль
- Добавление nilcheck метрик в профиль

[github.com/quasilyte/qpprof](https://github.com/quasilyte/qpprof)

Можно использовать как пример того, чему мы научились сегодня (подглядывать разрешается).

# Утилита pprofutils

Умеет выполнять разные преобразования форматов профилей.

[github.com/felixge/pprofutils](https://github.com/felixge/pprofutils)

# Утилита gogrep

Использует perf-heatmap для поиска кода только по горячим местам.

[profile-guided](#) поиск по коду



## Утилита perfguard

Использует perf-heatmap для определения горячих мест, чтобы давать советы по оптимизации только там, где это имеет значение.

Работаем с CPU профилями как с данными  
@quasilyte 2022