Compiler And Runtime Optimizations

This page lists optimizations done by the compilers. Note that these are not guaranteed by the language specification.

Interface values

Zero-width types in interface values

Putting a zero-width type in an interface value doesn't allocate.

- **gc:** 1.0+
- **gccgo:** ?

Word-sized value in an interface value

Putting a word-sized-or-less non-pointer type in an interface value doesn't allocate.

- **gc:** 1.0-1.3, but *not* in 1.4+
- **gccgo:** never

`string` and `[]byte`

Map lookup by `[]byte`

For a map `m` of type `map[string]T` and `[]byte b`, `m[string(b)]` doesn't allocate. (the temporary string copy of the byte slice isn't made)

- **gc:** 1.4+
- **gccgo:** ?

`range` over `[]byte(s)`

No allocation when converting a `string` into a `[]byte` for ranging over the bytes:

```
s := "foo"
for i, c := range []byte(s) {
    // ...
}
```

conversion for string comparison

No allocation done when converting a `[]byte` into a `string` for comparison purposes

```
var b1 string
var b2 []byte
var x = string(b1) == string(b2) // memeq
var y = string(b1) < string(b2)  // lexicographical comparison
```

- **gc:** 1.5+ (CL 3790)
- **gccgo:** ?

Escape analysis and Inlining

Use `-gcflags -m` to observe the result of escape analysis and inlining decisions for the gc toolchain.

(TODO: explain the output of `-gcflags -m`).

Escape analysis

Gc compiler does global escape analysis across function and package boundaries. However, there are lots of cases where it gives up. For example, anything assigned to any kind of indirection (`*p = ...`) is considered escaped. Other things that can inhibit analysis are: function calls, package boundaries, slice literals, subslicing and indexing, etc. Full rules are too complex to describe, so check the `-m` output.

- **gc:** 1.0+
- **gccgo:** 8.0+.

Function Inlining

Only short and simple functions are inlined. To be inlined a function must conform to the rules:

- function should be simple enough, the number of AST nodes must less than the budget (80);
- function doesn't contain complex things like closures, defer, recover, select, etc;
- function isn't prefixed by go:noinline;
- function isn't prefixed by go:uintptrescapes, since the escape information will be lost during inlining;
- function has body;
- etc.
- **gc:** 1.0+
- **gccgo:** -O1 and above.

Idioms

Optimized memclr

For a slice or array s, loops of the form

```
for i := range s {
    s[i] = <zero value for element of s>
}
```

are converted into efficient runtime memclr calls. Issue and commit.

- **gc:** 1.5+
- **gccgo:** ?

Non-scannable objects

Garbage collector does not scan underlying buffers of slices, channels and maps when element type does not contain pointers (both key and value for maps). This allows to hold large data sets in memory without paying high price during garbage collection. For example, the following map won't visibly affect GC time:

```
type Key [64]byte // SHA-512 hash
type Value struct {
    Name      [32]byte
    Balance   uint64
    Timestamp int64
}
m := make(map[Key]Value, 1e8)
```

- **gc:** 1.5+
- **gccgo:** ?