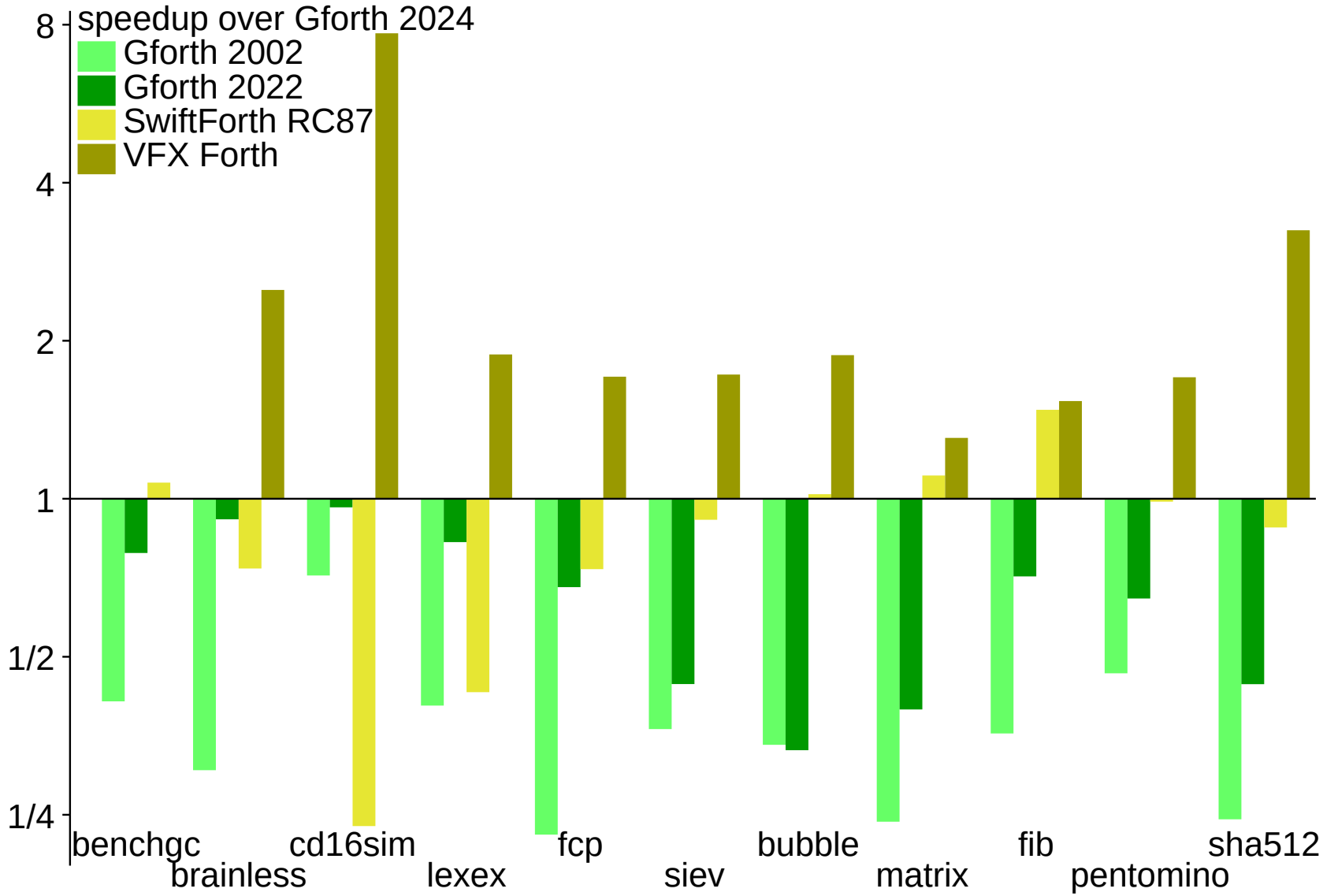
How to implement words (efficiently)

M. Anton Ertl, TU Wien

Motivation


speedup over Gforth 2024
Gforth 2002
Gforth 2022
SwiftForth RC87
VFX Forth
8
4
2
1
1/2
1/4
benchgc
brainless
cd16sim
lexex
fcp
siev
bubble
matrix
fib
pentomino
sha512

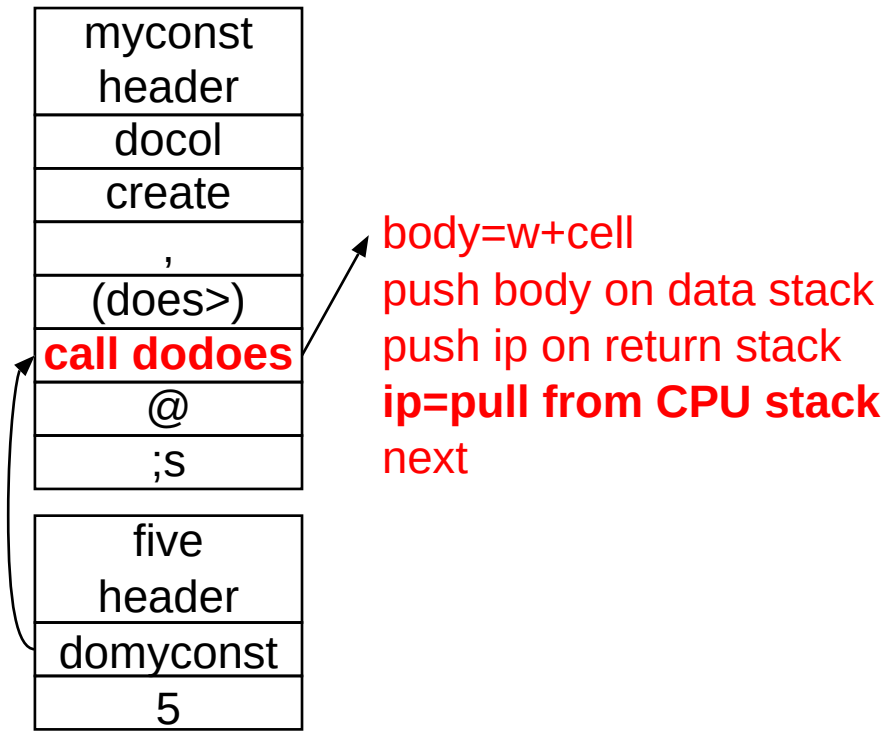
Call-pull: indirect-threaded origins

Problem

From one cfa/xt get:

- body address
- does-code address
- code address (dodoes)

```
: myconst ( x "name" -- )
  create ,
does> ( -- x )
  @ ;
5 myconst five
```



Call-pull in native-code systems

Problem

`five` is called

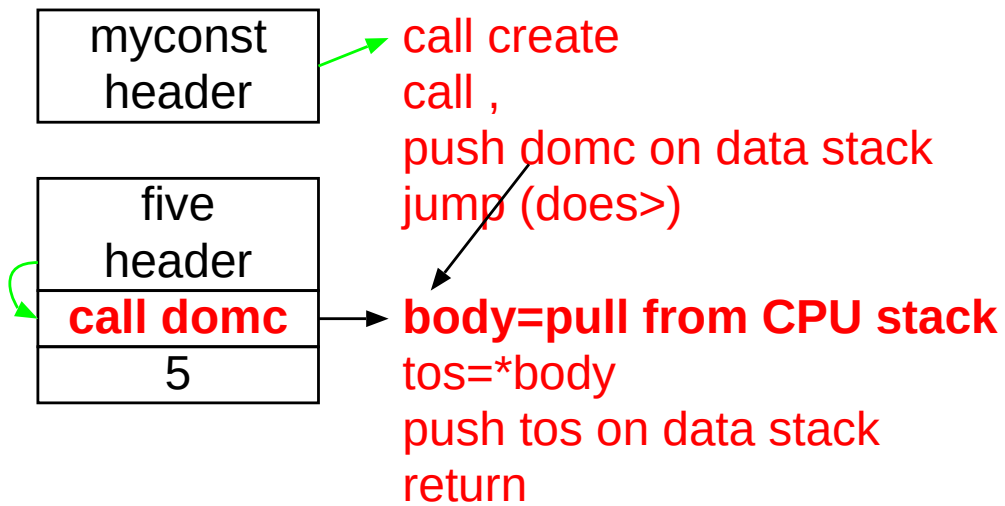
no CFA in W register

Solution?

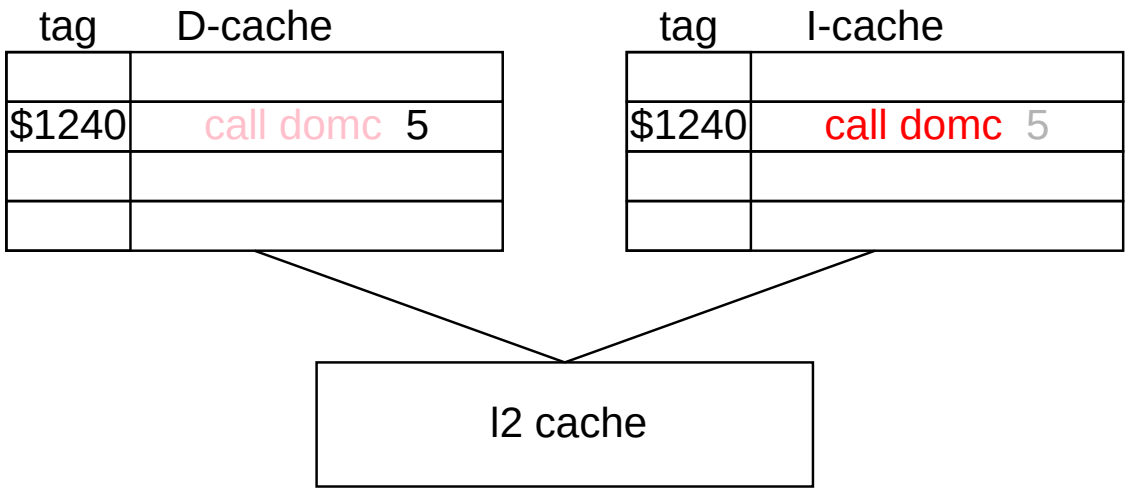
Use call-pull

also without `does>`

```
: myconst ( x "name" -- )
  create ,
does> ( -- x )
  @ ;
5 myconst five
```

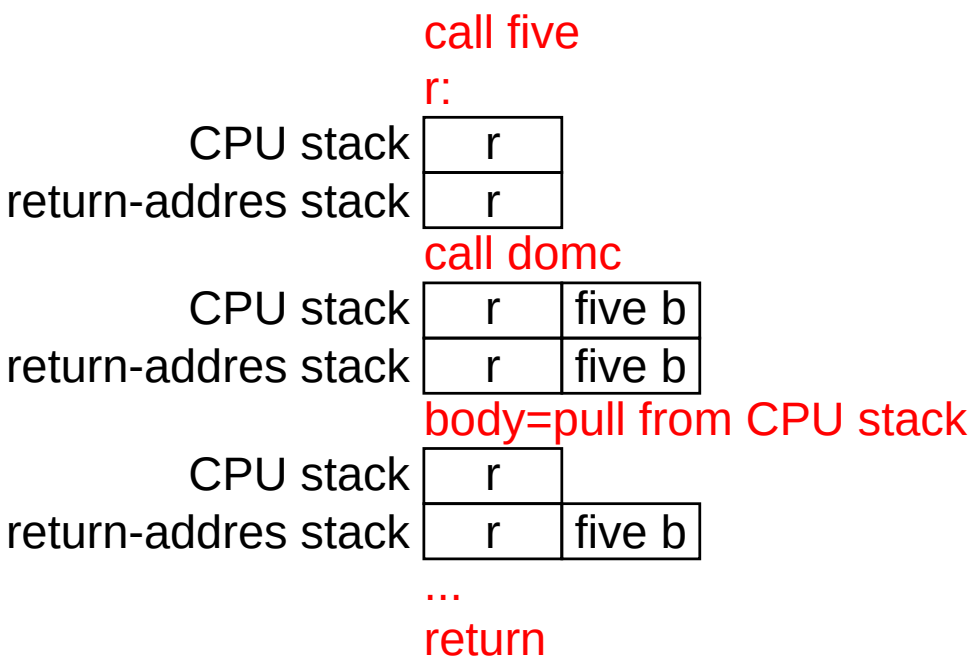


Performance pitfall: false sharing



- Granularity: cache lines (64B)
- Write invalidates the line in other cache(s)
- Usually D-caches of different cores
- Between I and D cache: on IA-32, AMD64, s390(x)
- Cost for round-trip:
  $\approx$ 400 cycles on Intel P
  $\approx$ 100 cycles on Ryzen 5800X
- True sharing at least as bad slow on all architectures

Performance pitfall: return misprediction



- return address-stack: hardware for branch prediction
- call-pull results in out-of-sync stacks usually one or more branch mispredictions
- Cost: $\approx$ 20–30 cycles per misprediction

Initial case

```
: d1 ( "name" -- )
    create 0 ,
does> ( -- addr )
;
d1 z1
: bench-z1-comp ( -- )
    iterations 0 ?do
        1 z1 +!
    loop ;
```

cycles	inst.	cache misses I	cache misses D	branch mispred	system
8.2	34.0	0.0	0.0	0.0	gforth
9.0	6.6	0.0	0.0	0.0	iforth
6.4	15.0	0.0	0.0	0.0	lxf
6.5	14.0	0.0	0.0	0.0	sf RC89
434.2	15.0	2.0	2.0	1.0	sf RC87
7.7	4.6	0.0	0.0	0.0	vfx

- Based on application (CD16sim)
- Slowness due to call-pull

Does only SwiftForth RC87 have such problems?

```
: d2 ( "name" -- )
    create 0e f,
does> ( -- )
    1e dup f@ f+ f! ;
d2 z2
: bench-z2-exec ( -- )
    ['] z2 iterations 0 ?do
        dup execute
    loop ;
```

cycles	inst.	cache misses I	cache misses D	branch mispred	system
10.4	49.0	0.0	0.0	0.0	gforth
449.5	49.6	2.0	2.1	0.0	iforth
13.5	19.0	0.0	0.0	0.0	lxf
428.3	26.0	2.0	2.0	1.0	sf RC89
249.5	30.0	2.0	1.0	1.0	sf RC87
228.2	16.6	1.0	1.0	1.0	vfx

- call-pull frequent for `execute`ing xts

Do such problems only occur with `does>`?

```
create x 0 ,
: bench-x-exec ( -- )
    ['] x iterations 0 ?do
        1 over execute +!
    loop drop ;
```

cycles	inst.	cache misses I	cache misses D	branch mispred	system
7.0	28.0	0.0	0.0	0.0	gforth
16.5	49.6	0.0	0.0	0.0	iforth
6.0	17.0	0.0	0.0	0.0	lxf
442.8	24.0	2.0	2.0	1.0	sf
221.1	17.6	1.0	1.0	1.0	vfx

- `Created` word implemented with call-pull

What about `defer` and `is`?

```
0 constant my0
defer w ' my0 is w
: bench-w-comp ( -- )
    ['] my0 ['] drop
    iterations 0 ?do
        w over is w
    loop
    2drop ;
```

cycles	inst.	cache misses I	cache misses D	branch mispred	system
7.0	22.5	0.0	0.0	0.0	gforth
9.2	19.6	0.0	0.0	0.0	iforth
427.0	21.5	2.0	1.0	0.3	lxf
435.9	19.5	2.7	2.0	1.0	sf
205.3	11.1	1.0	1.0	0.5	vfx

- True sharing on lxf thanks to jump-patching
- False sharing on SwiftForth and VFX thanks to call-pull

What about `defer` without `is`?

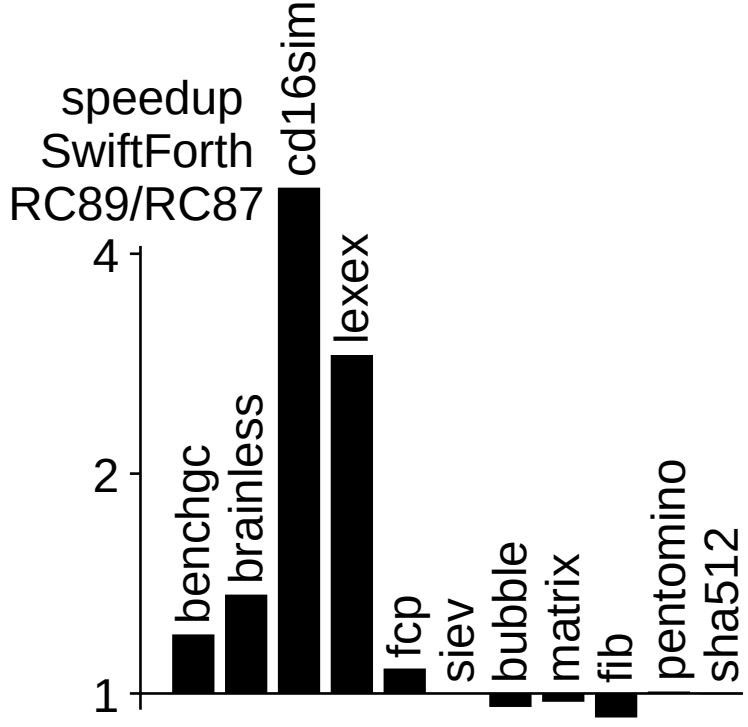
```
0 constant my0
defer w ' my0 is w
: bench-w-nois-comp ( -- )
    iterations 0 ?do
        w drop
    loop ;
' z1 is w bench-w-nois-comp
```

cycles	inst.	cache misses I	cache misses D	branch mispred	system
8.4	35.0	0.0	0.0	0.0	gforth
15.5	42.6	0.0	0.0	0.0	iforth
6.0	12.0	0.0	0.0	0.0	lxf
29.4	16.0	0.0	0.0	1.0	sf
27.2	11.6	0.0	0.0	1.0	vfx

- Return mispredictions in SwiftForth and VFX thanks to call-pull
- Similar for other uses of call-pull without writes

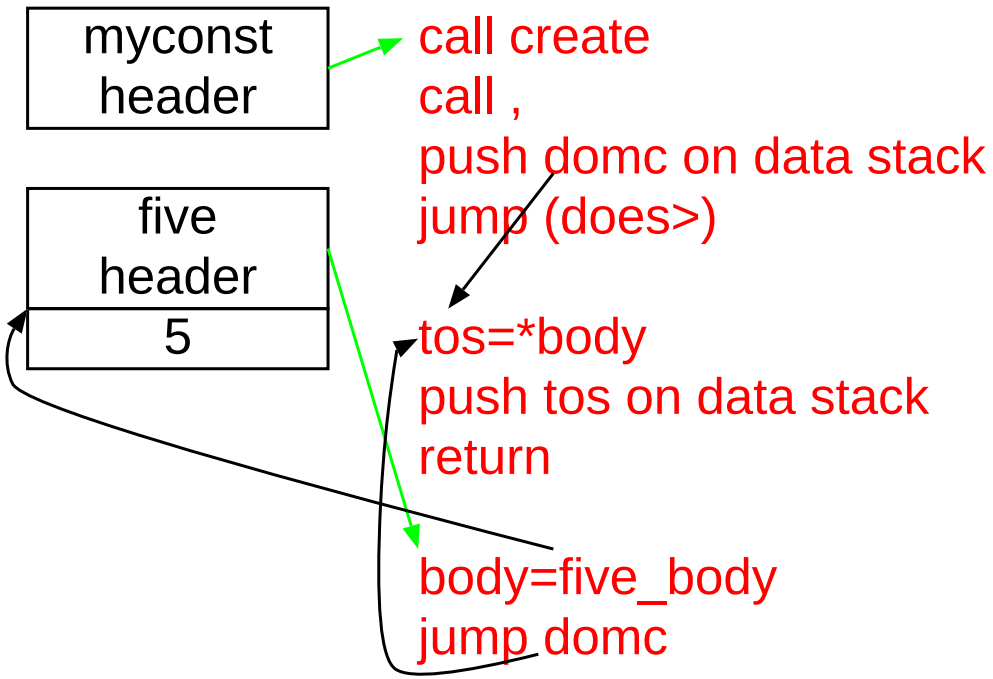
But what about applications?

- How do you know that an application is affected?
- Compare I-cache misses with other Forth systems
- Compare branch mispredictions with other Forth systems
- Implement words using techniques without these performance pitfalls
  Compare your Forth before and after



How to avoid call-pull? (1) Trampolines

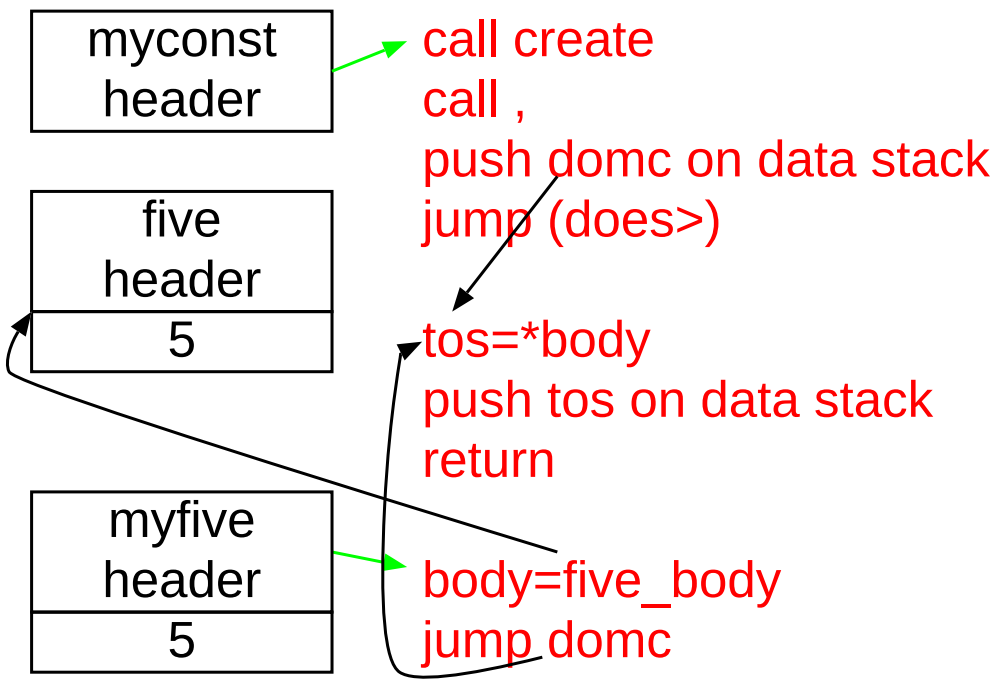
```
: myconst ( x "name" -- )
  create ,
does> ( -- x )
  @ ;
5 myconst five
```



- Keeps code separate from data
- Get body address without pull

How to avoid call-pull? (2) Intelligent `compile,`

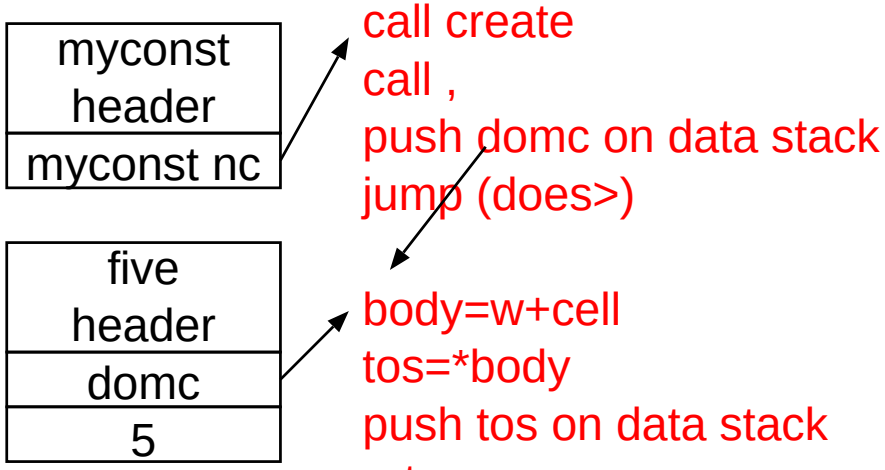
```
: myconst ( x "name" -- )
  create ,
does> ( -- x )
  @ ;
5 myconst five
: myfive five ;
```



- Compile address of body, then call to doer
  call is tail-call optimized in `myfive`
- Keeps code separate from data
- Get body address without pull
- Can be used to generate trampoline
- Does not help for `execute` or `defer`

How to avoid call-pull? (3) Code field

```
: myconst ( x "name" -- )
  create ,
does> ( -- x )
  @ ;
5 myconst five
```



- Set W to CFA, then perform indirect call
- Fine for `execute` and `deferred` words
- Slow for naively `compile,`d code
  but intelligent `compile,` avoids that

How to implement `defer`red words

- Just use a data field and an indirect jump
- Don't patch native-code jumps

Conclusion

- Call-pull slowdowns
  false sharing (100+ cycles)
  return mispredictions (20-30 cycles)
- Big slowdowns in microbenchmarks
- How much in applications?
  You know it when you fix it
- Avoiding call-pull is possible
  trampolines (used in lxf)
  intelligent `compile,` (used everywhere)
  code field
- Use data field for `defer`red words
- Microbenchmarks: `http://www.euroforth.org/ef24/papers/ertl.fs`