

Self modifying code in Linux kernel – what where and how

Evgeniy Paltsev

PaltsevEvgeniy@gmail.com

Embedded Linux Conference Europe 2019

October 28, 2019



Evgeniy Paltsev

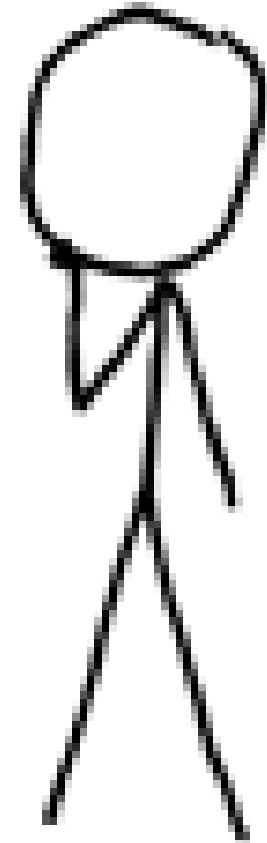
- Synopsys ARC team member
- Development, porting and maintenance of open-source projects for Synopsys ARC processors architecture
- Main focus is Linux kernel and U-Boot (drivers and platform support) with journeys to other projects like Weston, Buildroot, uClibc-ng, etc

READONLY

```
$ objdump --section-headers vmlinux
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.vector	00002000	90000000	90000000	00002000	2**2
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
1	.init.text	000196d8	90220000	90220000	00222000	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
2	.init.data	00005358	902396e0	902396e0	0023b6e0	2**5
			CONTENTS, ALLOC, LOAD, DATA			
3	.data..percpu	00006980	90240000	90240000	00242000	2**7
			CONTENTS, ALLOC, LOAD, DATA			
4	.text	004a82a8	90248000	90248000	0024a000	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			



Agenda

- Real use-cases for self modifying code in Linux kernel
- Deep dive – into static branches implementation for ARC

Mandatory cute / funny picture

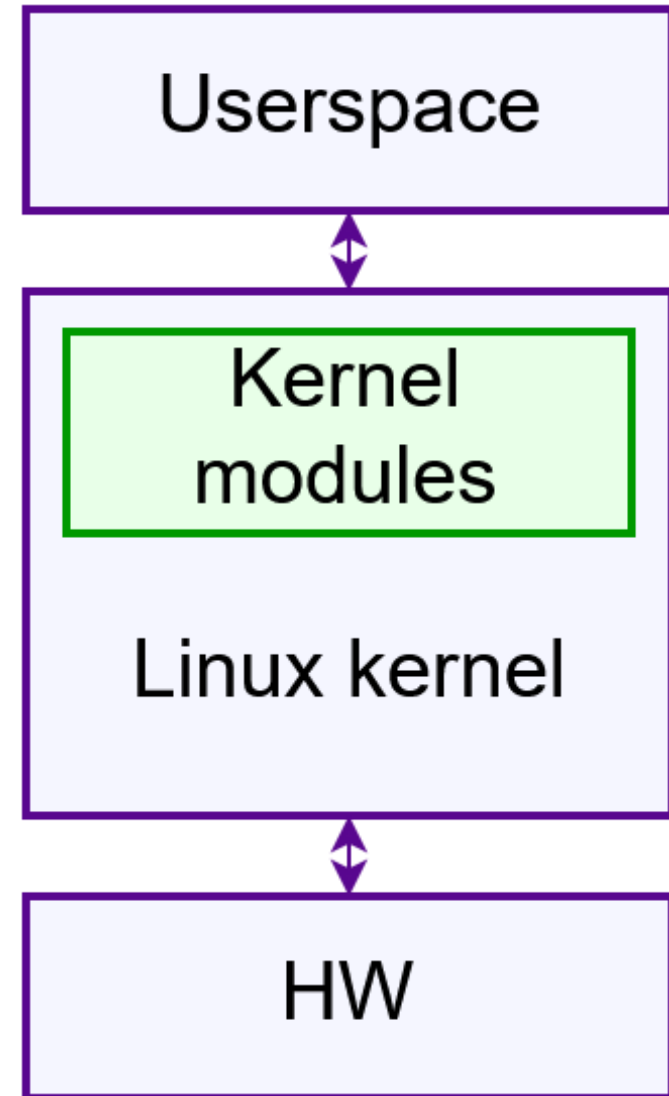


Real use-cases for self modifying code in Linux kernel



Kernel modules

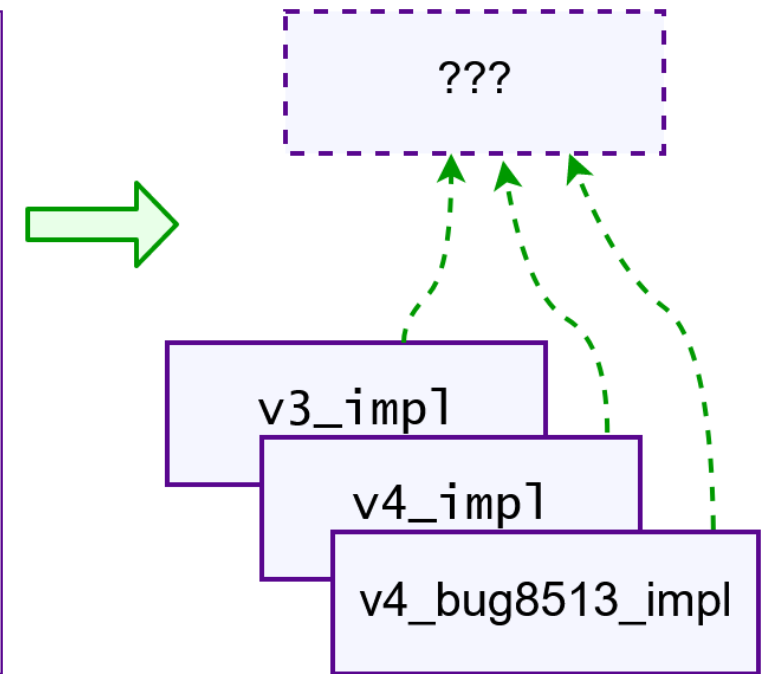
- Pieces of code that can be dynamically loaded and unloaded into the kernel
- Extend the functionality of the kernel without the need to reboot the system
- We modify both kernel (by adding external code) and module code (by load-linking)



Architecture-specific code rewriting

- Several architectures rewrite arch-specific code on early init phase to support HW with different
 - bugs
 - feature level
 - configurationin one image.
- Less overhead

```
switch(mmu_version) {  
  case MMU_V3:  
    v3_impl();  
    break;  
  case MMU_V4:  
    v4_impl();  
    break;  
  case MMU_V4_BUG8513:  
    v4_bug8513_impl();  
    break;  
}
```



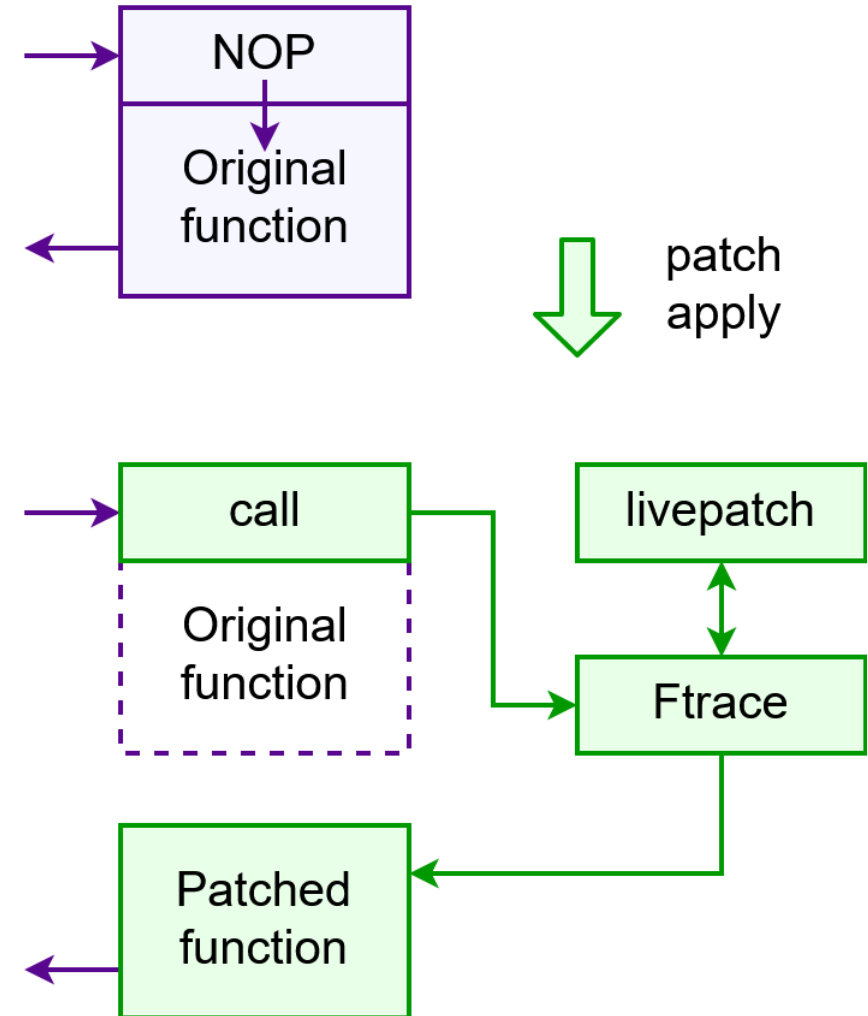
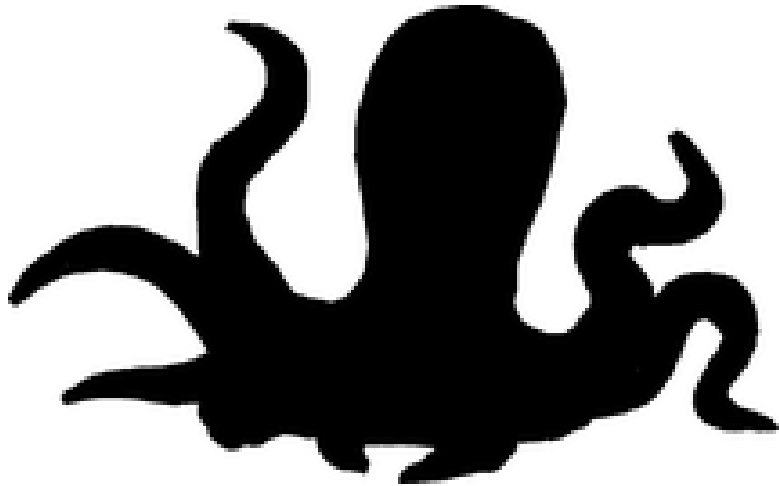
Making changes in code we'll execute is fine

Making changes in code we execute **right now** is way more interesting



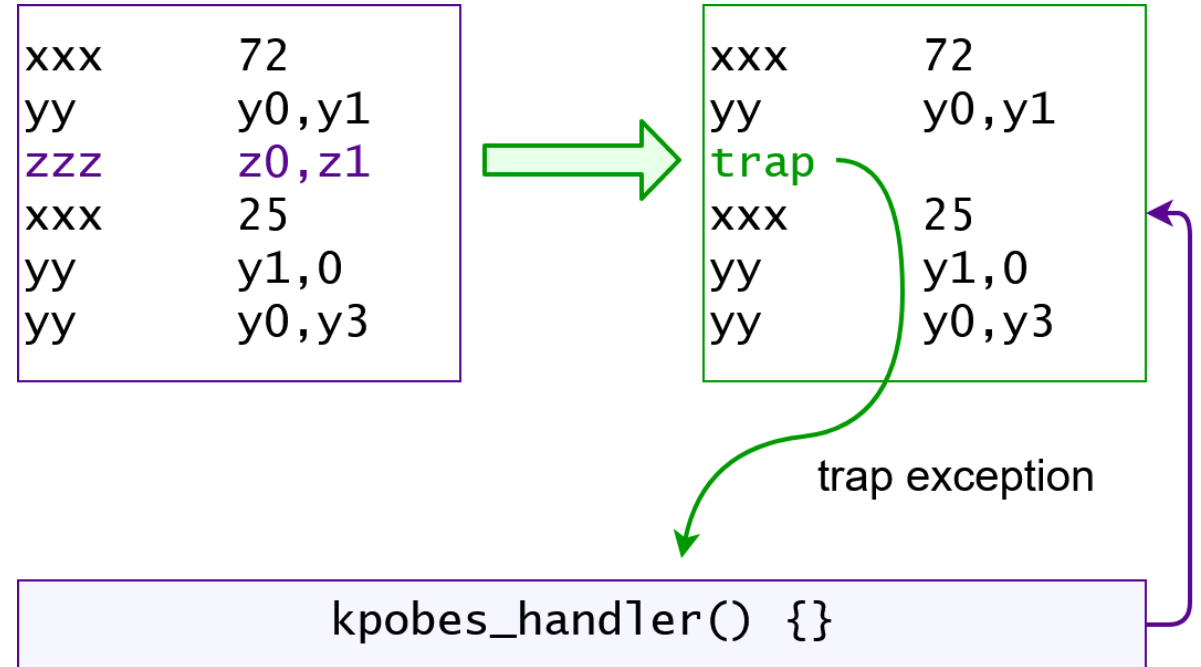
Livepatch

- Live kernel patching without system restart
 - apply kernel-related security updates
- Uses function call redirections (through Ftrace)



Kprobes

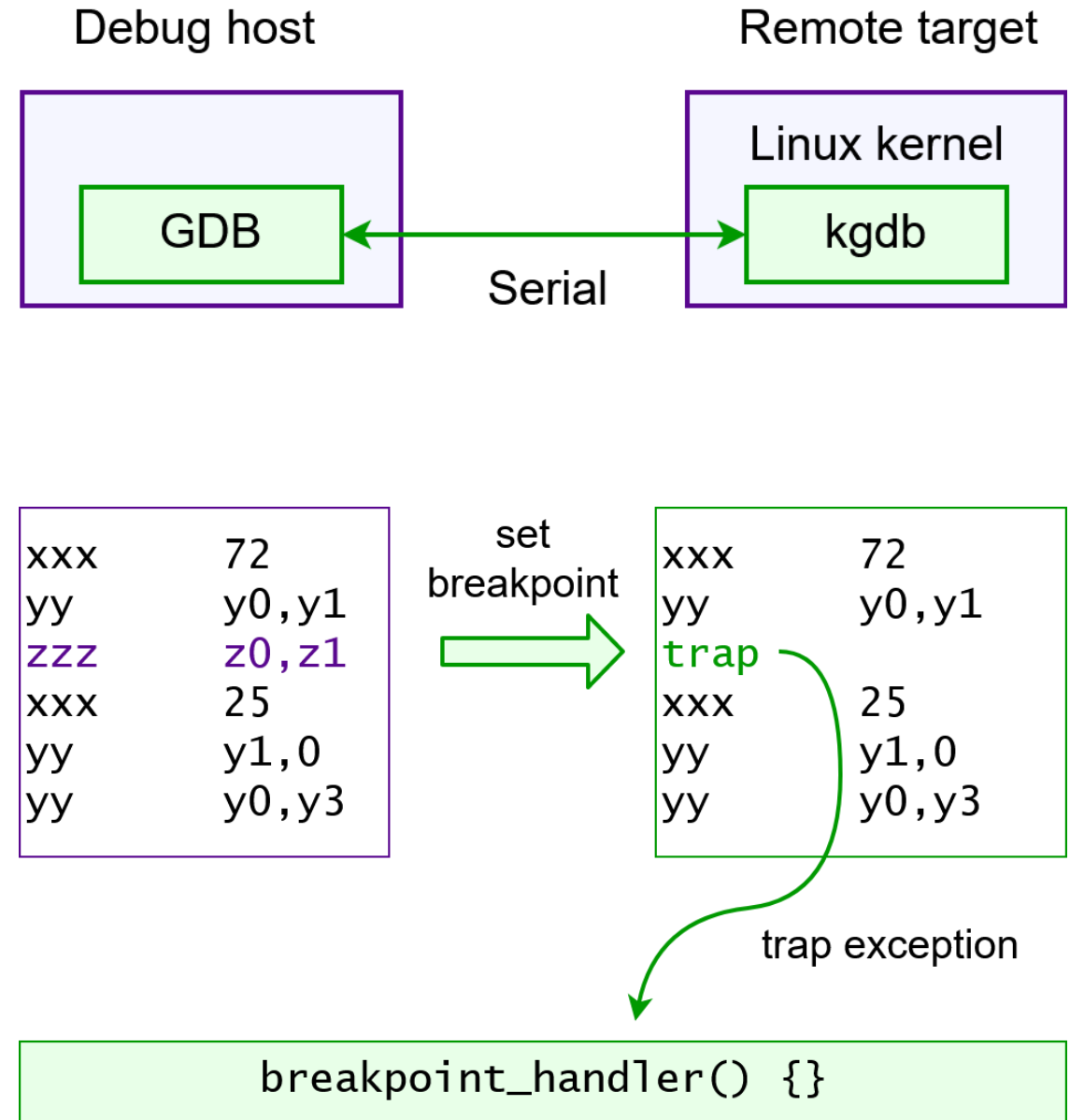
- Hooking Linux kernel functions to
 - monitor/trace events
 - collect debugging info
 - collect performance info
 - change execution path
- Allows to install pre-handlers / post-handlers for
 - kernel instructions
 - function-entry
 - function-return



KGDB

Kernel debugger

- Allows to
 - Set breakpoints
 - Check & modify data structures/memory/registers
 - Control kernel running flow
- May be used with
 - External GDB front-end (connected through serial)
 - Internal KDB front-end
- Questionable – reuse kernel code to debug kernel code
- Some features may modify kernel code – i.e. breakpoints



Static branches

The idea is to optimize branches where

- condition changes rarely
- branch itself is in hot codepath

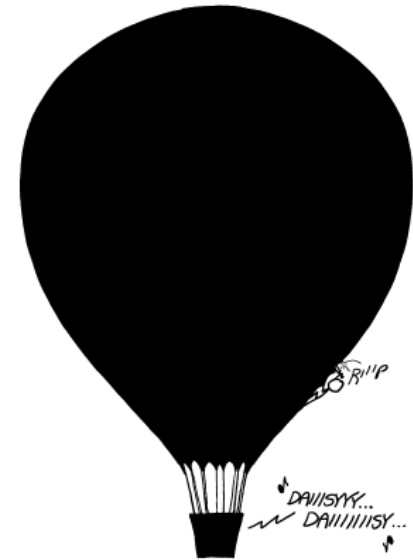
Example:

```
if (hw_feature_exists())  
    handle_feature();
```



Deep dive

- Examples are based on my static branches implementations for ARC
- Patches are under final review, most likely to be applied after minor fixies:
<https://lkml.org/lkml/2019/7/18/374>



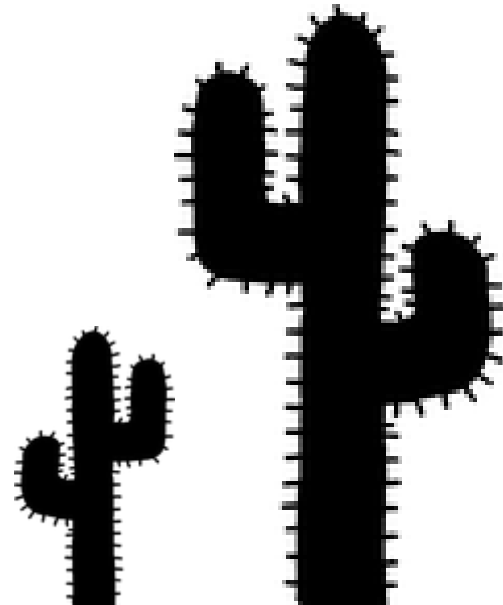
Static branches – use cases

- Originally – for use in tracepoints enabling / disabling code
- Now used in:
 - Tracepoints
 - Dynamic printk
 - Scheduler
 - Memory management
 - Network code
 - ...



Static branches – supported in major architectures

- Supported in different architectures (v5.3 kernel):
 - x86 / x86_64
 - ARM / ARM64
 - mips
 - parisc
 - powerpc
 - s390
 - sparc
 - xtensa



Static branches – implementation for ARC

Original idea – to make code faster

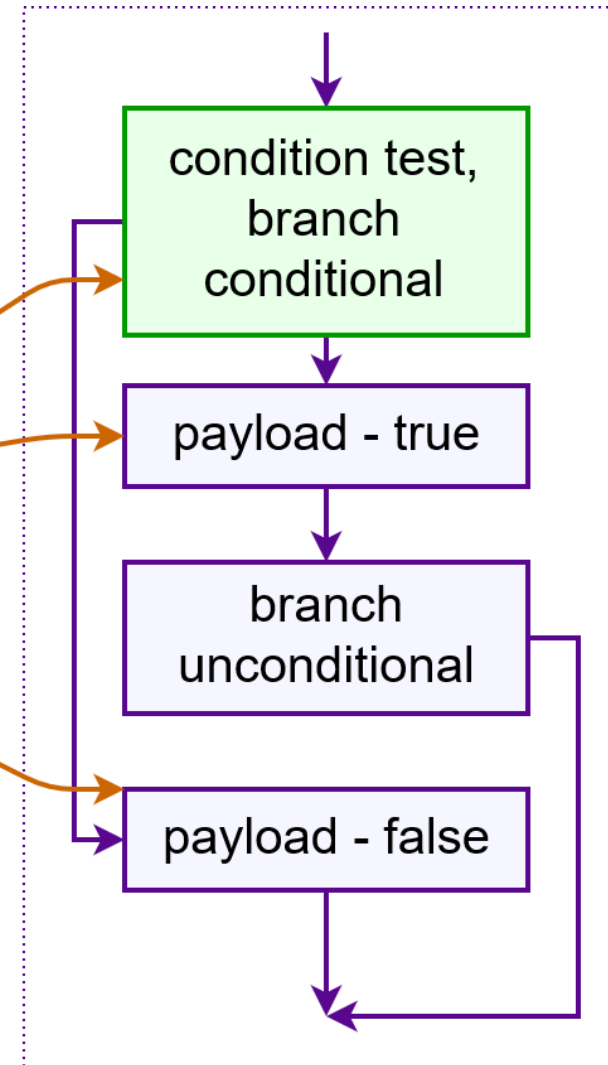
In our case – to make maintainer happy



Static branches – implementation

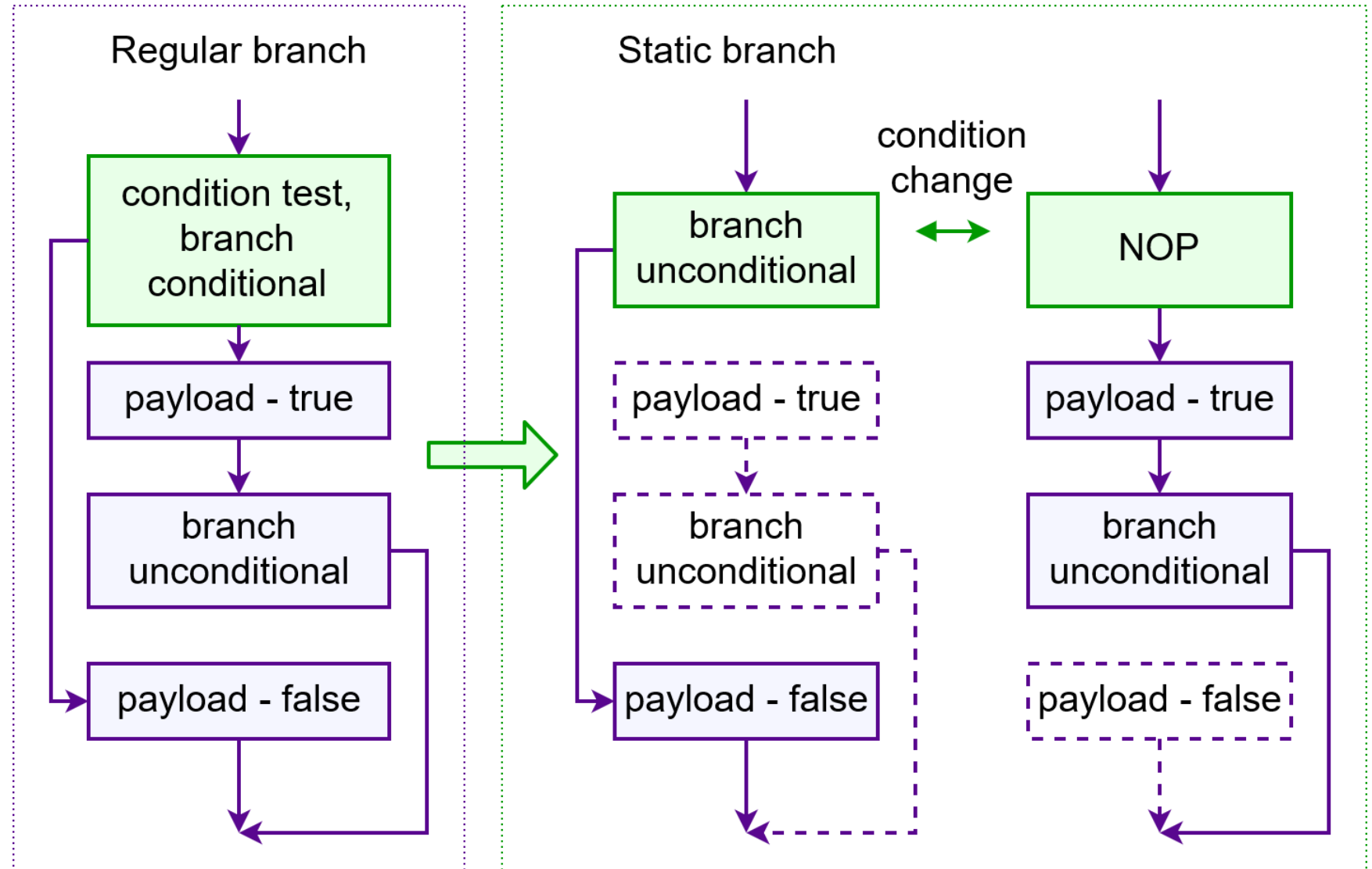
Regular branch:

```
if (branch_condition)
    true_payload();
else
    false_payload();
```



Static branches – implementation

Main idea – to switch between branches of ‘if’ statement by replacing BRANCH instruction with NOP (or vice versa)



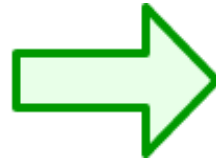
Static branches – user interface

Regular branches:

```
bool branch_condition = true;

if (branch_condition)
    true_payload();
else
    false_payload();

branch_condition = false;
```



Static branches:

```
DEFINE_STATIC_KEY_TRUE(cond);

if (static_branch_likely(&cond))
    true_payload();
else
    false_payload();

static_branch_disable(&cond);
```

- Bonus: we don't need to change anything in the code if we don't support static branches (or have them disabled)

Static branches – implementation interface

- Define
 - JUMP_LABEL_NOP_SIZE
 - struct jump_entry
- Implement functions

```
arch_jump_label_transform  
arch_static_branch  
arch_static_branch_jump
```

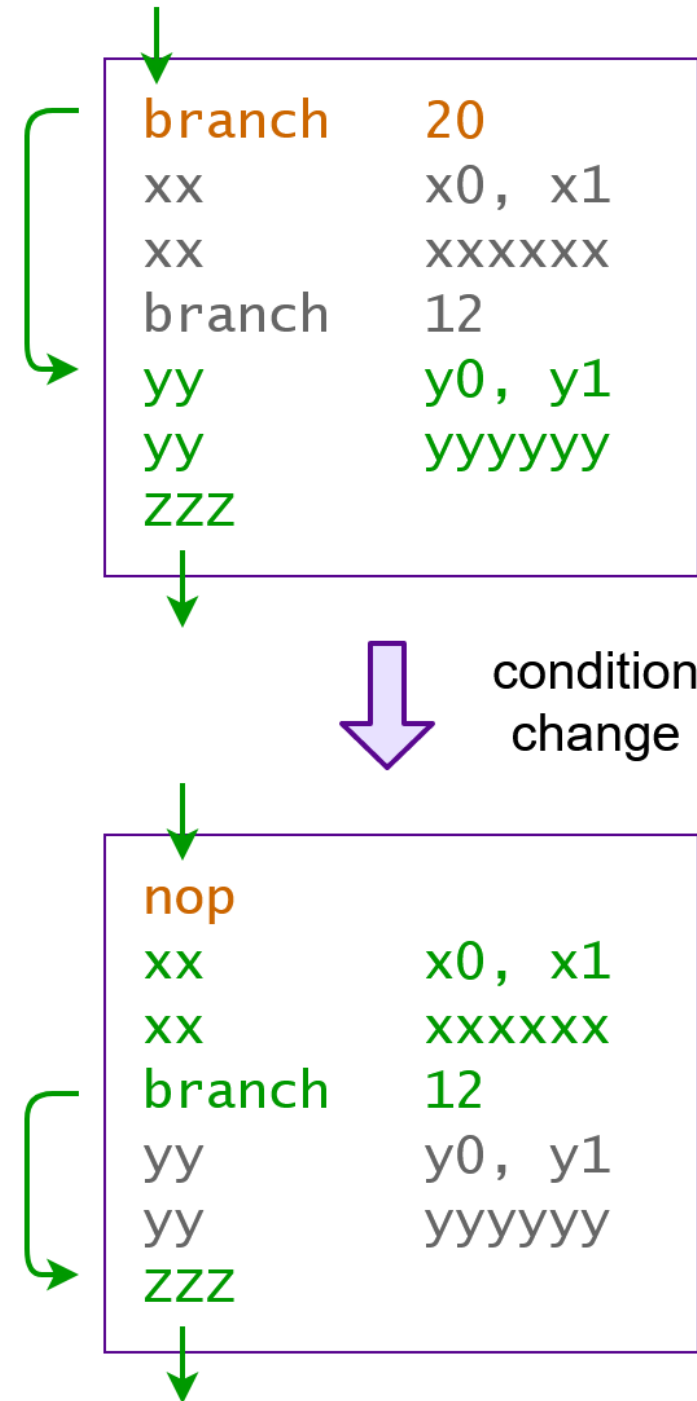


Static branches – implementation

- We have 32-bit NOP and 32-bit BRANCH
- All we need to do – replace one 32-bit instruction with another. **Atomically.**

- It's easy!
- We simply write the instruction to its address:

```
*instruction_address = NOP_32BIT;
```



Static branches – implementation

- **Ooops:** compiler can split our single 32-bit write for several small writes (i.e. two 16-bit writes)
- We don't have atomic update anymore.

- Solution:

```
WRITE_ONCE(*instruction_address, NOP_32BIT);
```

```
...  
store_word r0, [addr]
```



```
*addr = NOP_32BIT;
```



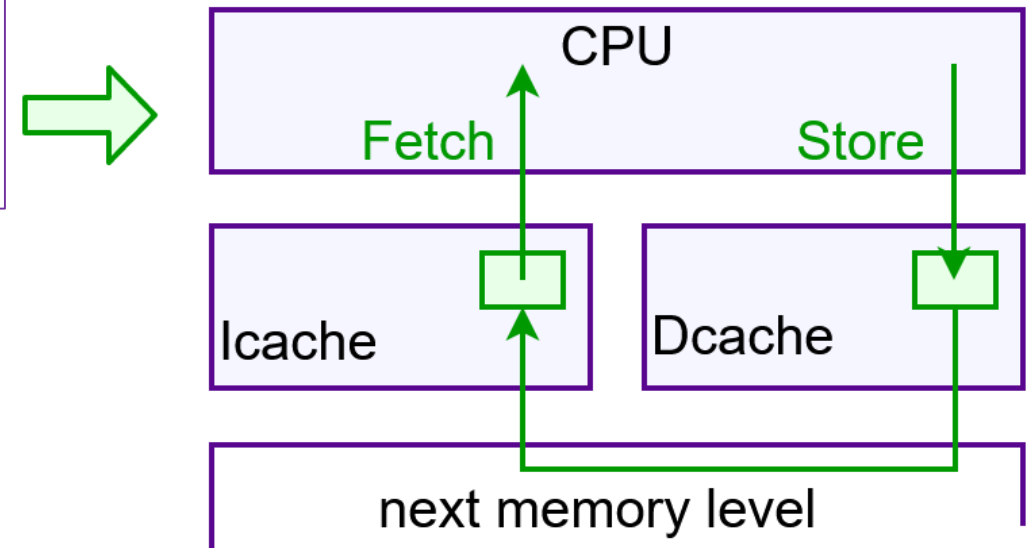
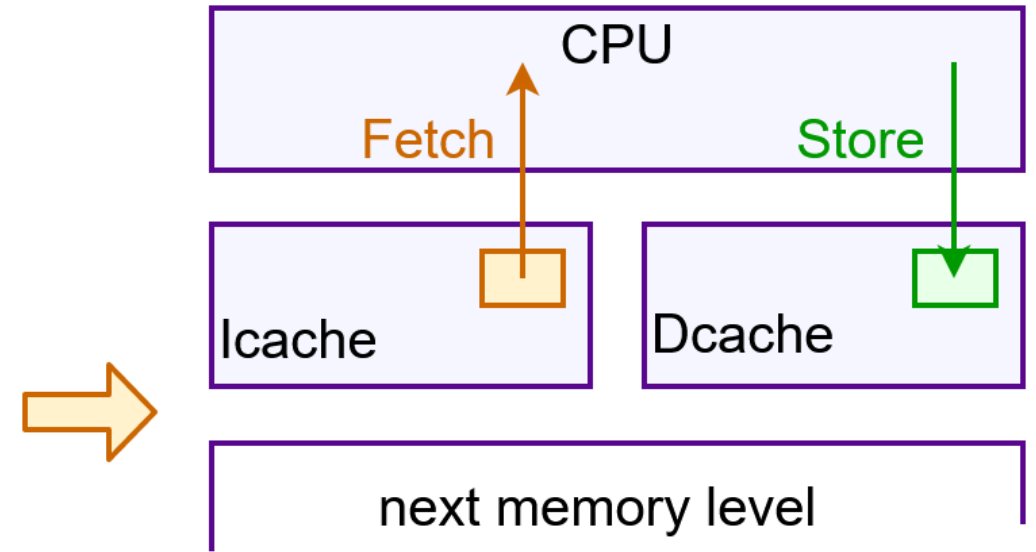
```
...  
store_half r0, [addr]  
store_half r1, [addr+2]
```

Static branches – implementation

- **Ooops:** we have caches. And instruction cache isn't coherent with data cache.
- No one knows when code will be really updated.

- Solution:

```
WRITE_ONCE(*instruction_address, NOP_32BIT);  
flush_data_cache_line();  
invalidate_instruction_cache_line();
```

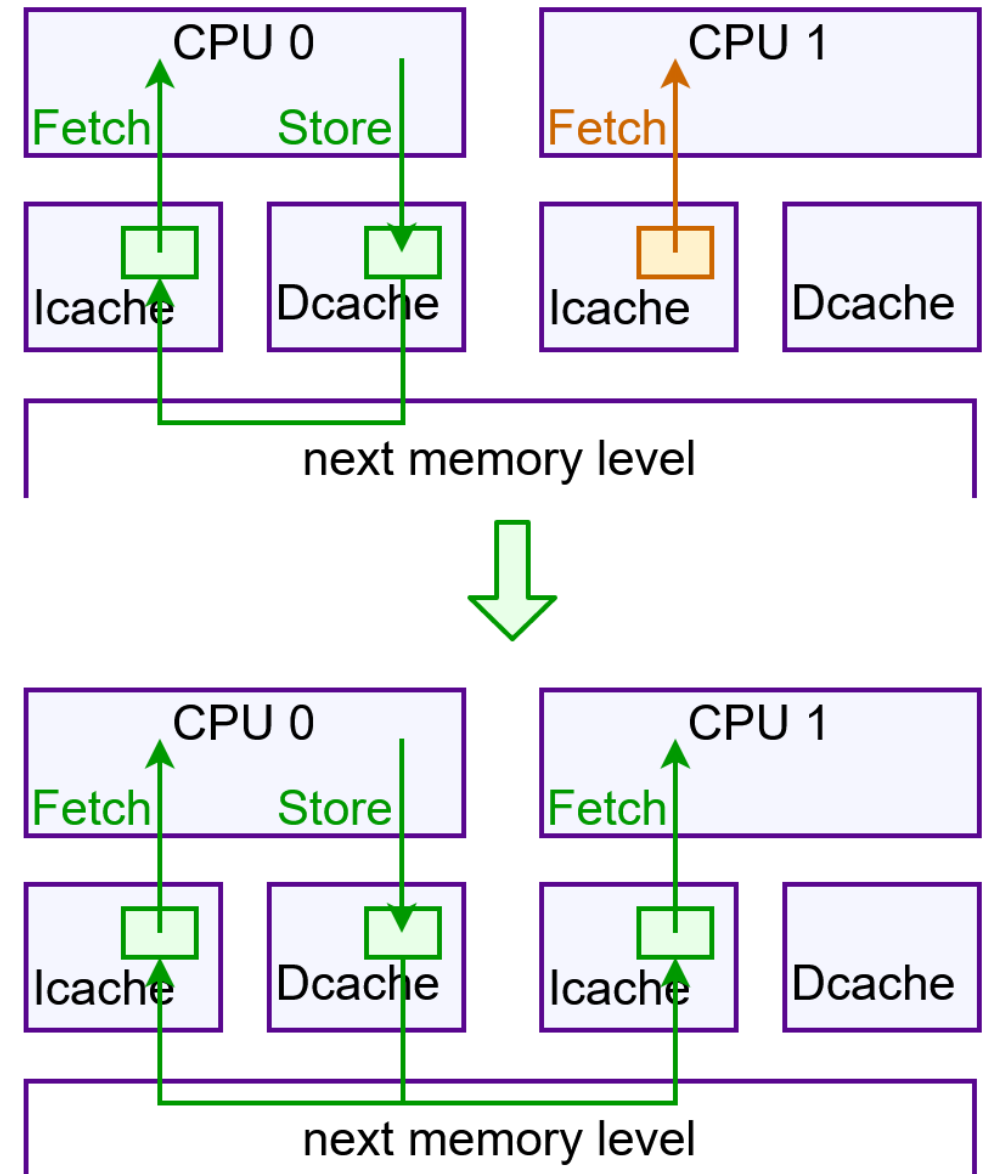


Static branches – implementation

- **Ooops:** we are in 2019. And we have multicore CPU.
- No one knows when code will be really updated for other CPUs.

- Solution:

```
WRITE_ONCE(*instruction_address, NOP_32BIT);  
flush_data_cache_line();  
on_each_cpu(invalidate_instruction_cache_line);
```



Static branches – implementation

- **Ooops:** instruction replace may be non-atomic if instruction cross cache line boundary
- We may execute partially updated instruction (2byte old + 2byte new)

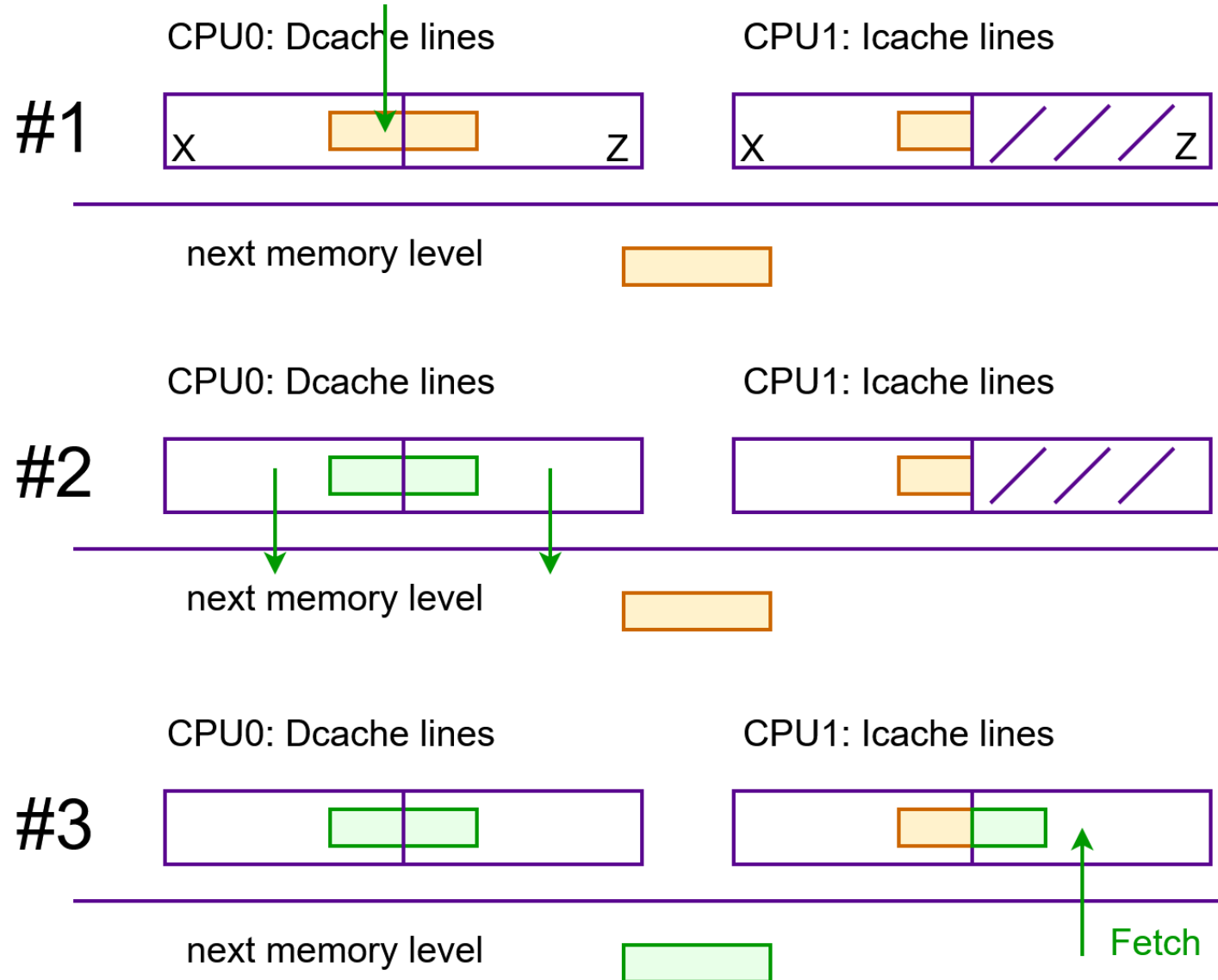
- Solution:

Force instruction to not cross cache line

- ".bundle_align_mode"
- ".balign"

assembler directives in

- arch_static_branch()
- arch_static_branch_jump()



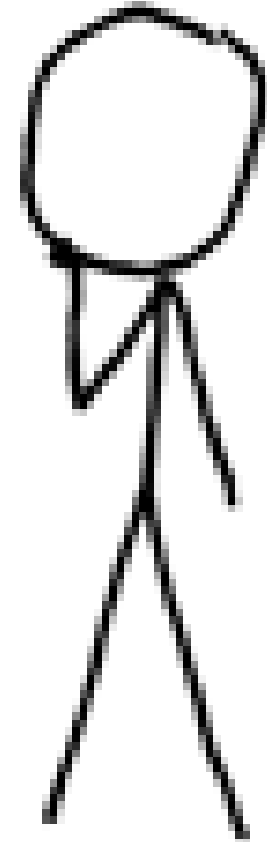
We started from simple writing of 32bit value

And ended up with

- manual cache management
- inter-processor communications
- special code alignment

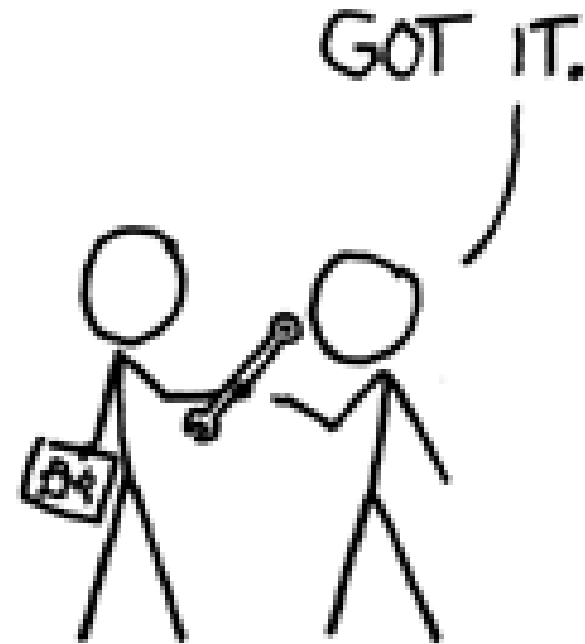
Static branches – implementation

- **Ooops:** circular dependencies
I.e. we use static branches to optimize cache ops but they are used in branch patching code.
- **Ooops:** static branches in early code
I.e. we want to update branch when static branches core isn't initialized.
- **Ooops:** interactions between different self-modifying code subsystems
I.e. we want to set breakpoint to our branch update instruction? - not a good idea.



Self modifying code in Linux kernel

- Powerful tool
- Use responsibly
- Don't rewrite yourself



Thanks!
Questions?



Images licensing notes

The slides [3, 10, 13-16, 21, 38, 29] include images which are based on or includes content from xkcd.com. Content from xkcd.com is licensed under the Creative Commons Attribution-NonCommercial 2.5 license

The slide [5] includes image from <http://www.flickr.com/photos/74743437@N00/3577220863/> which is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.0 Generic license