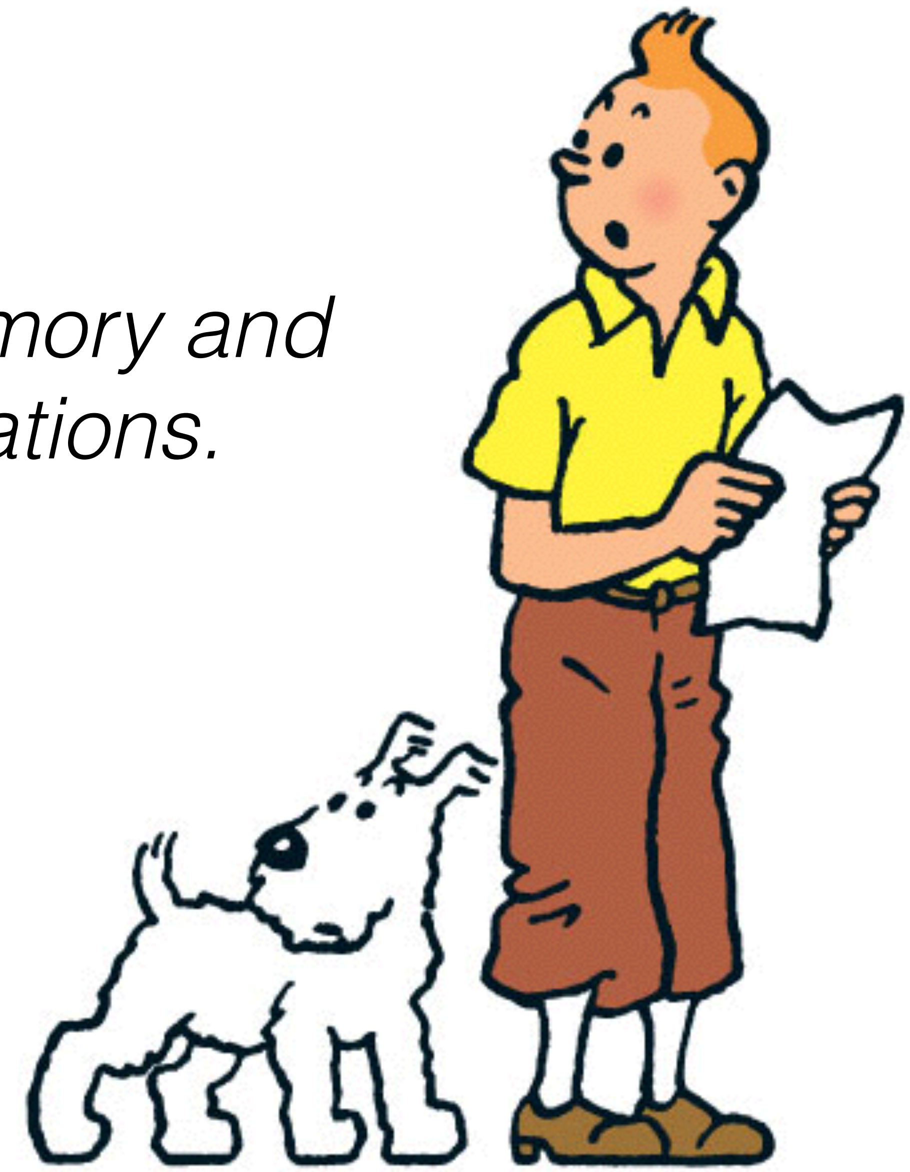
IgProf

The ignominious profiler. A generic memory and performance profiler for linux applications.

http://igprof.org

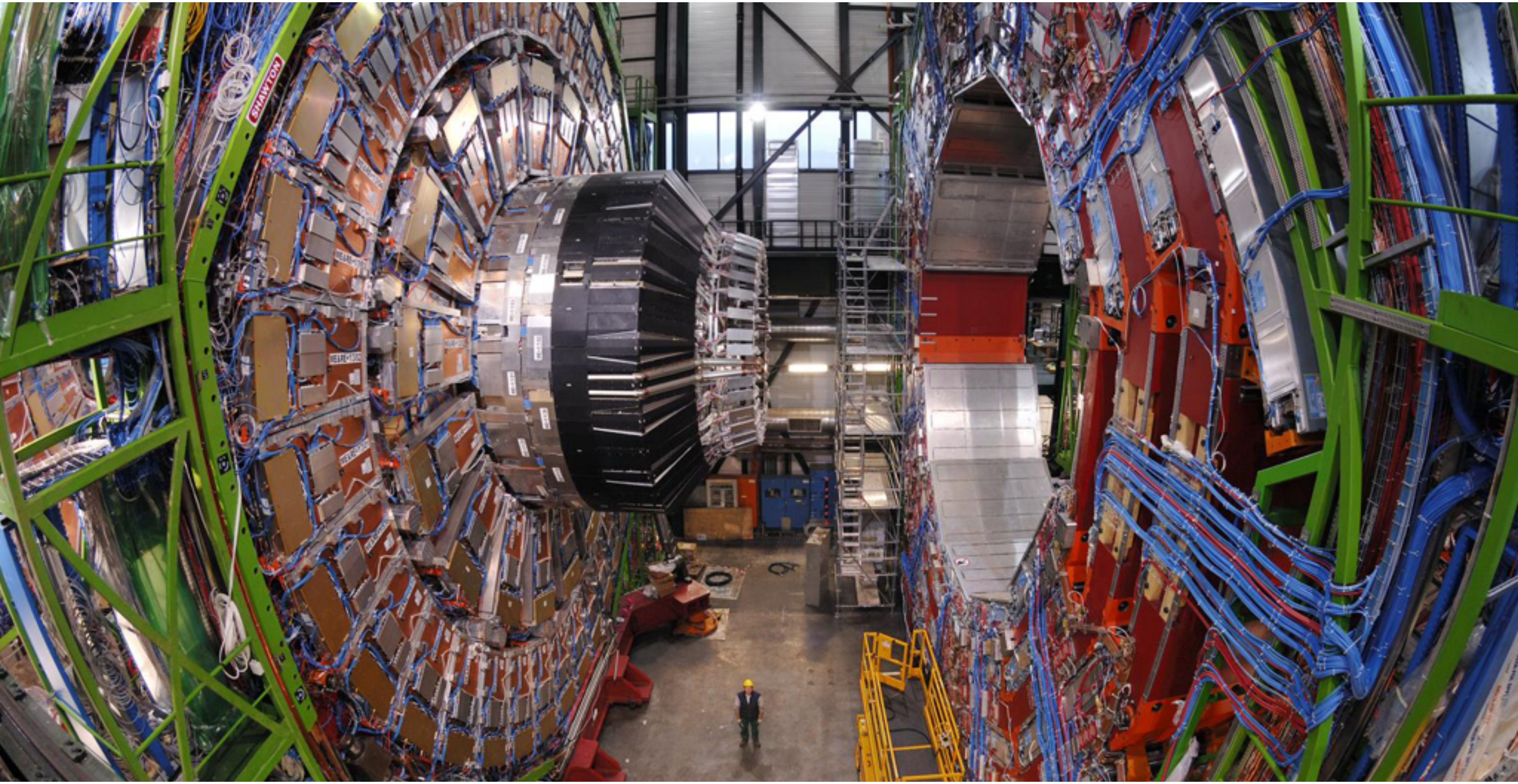


Giulio Eulisse
Fermi National Accelerator Laboratory

Opinions expressed in this presentation are mine, only mine, and solely mine…. buahahahah

Why a profiler?

Where I work

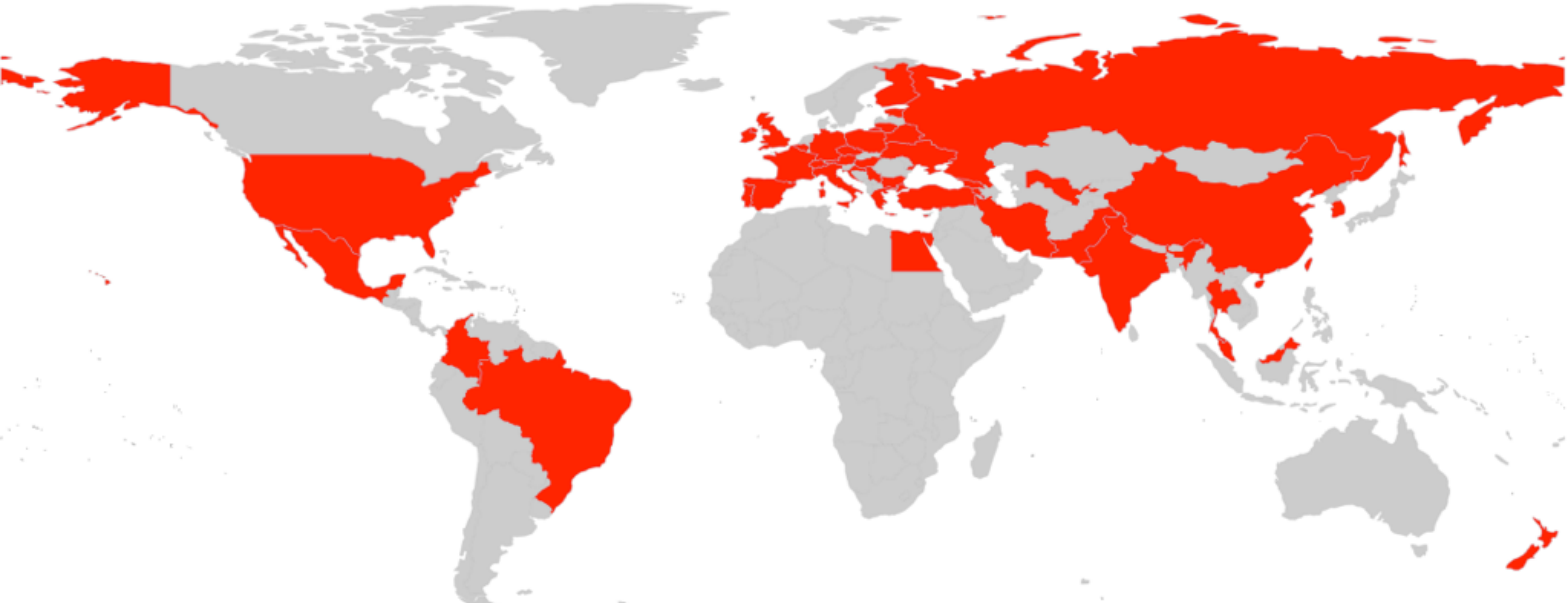


CMS experiment @ CERN LHC

Where I work

Over 4000 researchers...

Where I work

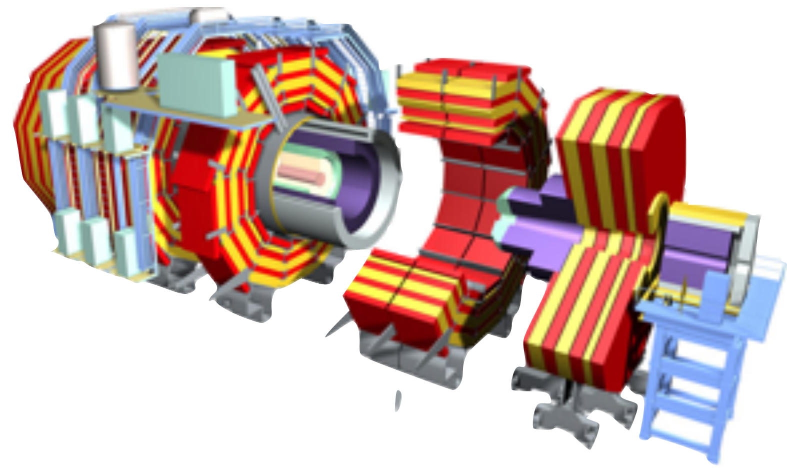


43 countries, 191 institutes

Source: http://cms.web.cern.ch/content/cms-collaboration

What do we do?

+

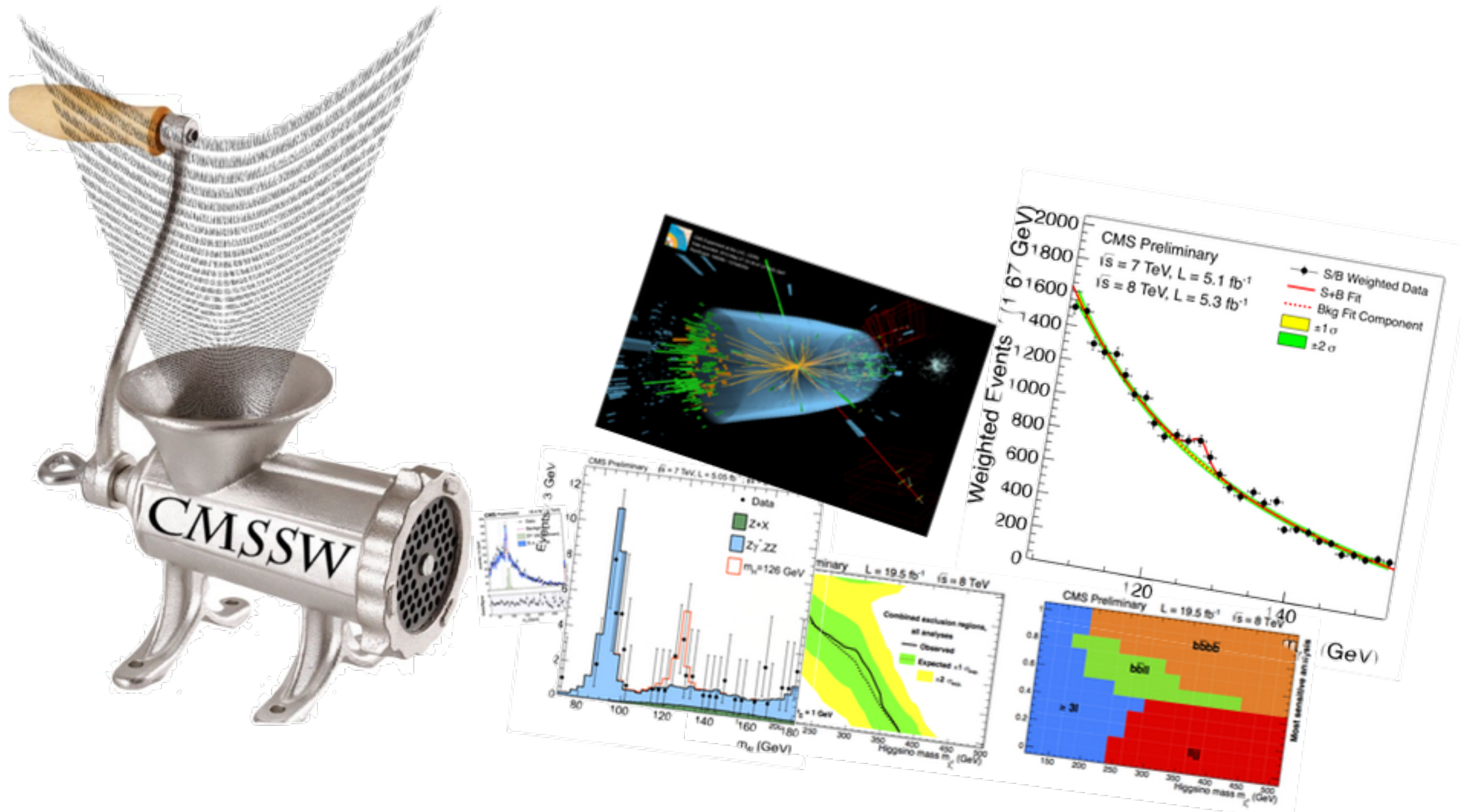


=

Mostly, we use 1.3 terawatt hours yearly to smash particles and we take digital pictures of the result with a sophisticated camera. Every single collision, a “RAW Event”, is roughly 1 MB. LHC delivers 10^9 events per second (i.e. 1 PB) when running, which become a few hundreds evt/s after a first level of (HW) “trigger”. We stored to disk roughly 10^9 events in 2011-2013 period*

Source: http://home.web.cern.ch/about/engineering/powering-cern

What do we do?



We process detector data with in custom written software and quickly obtain PhD theses and conference papers with it.

CMSSW...

Over 5M SLOCs custom code

Large C++ / Python / Fortran codebase, developed over almost two decades by over 1300 researchers ranging from master students to Nobel prize candidates. Software comprises complex pattern recognition algorithms, data analysis tools and simulations. Small core of professional software engineers mostly doing application framework, release integration and QA.

Large and diverse application set

Many different "workflows" depending on the the kind of simulation / analysis being done. Over 3000 shared objects chained via python configuration language. Each workflow has ~300 MB of CODE sections, sparse in ~600 shared libraries. Roughly 600'000 symbols present in the process image. Over 300 level deep call-trees are the norm. 2GB RSS footprint on average, large memory churn (up to 1M allocation per second). No single offender.

Large working dataset

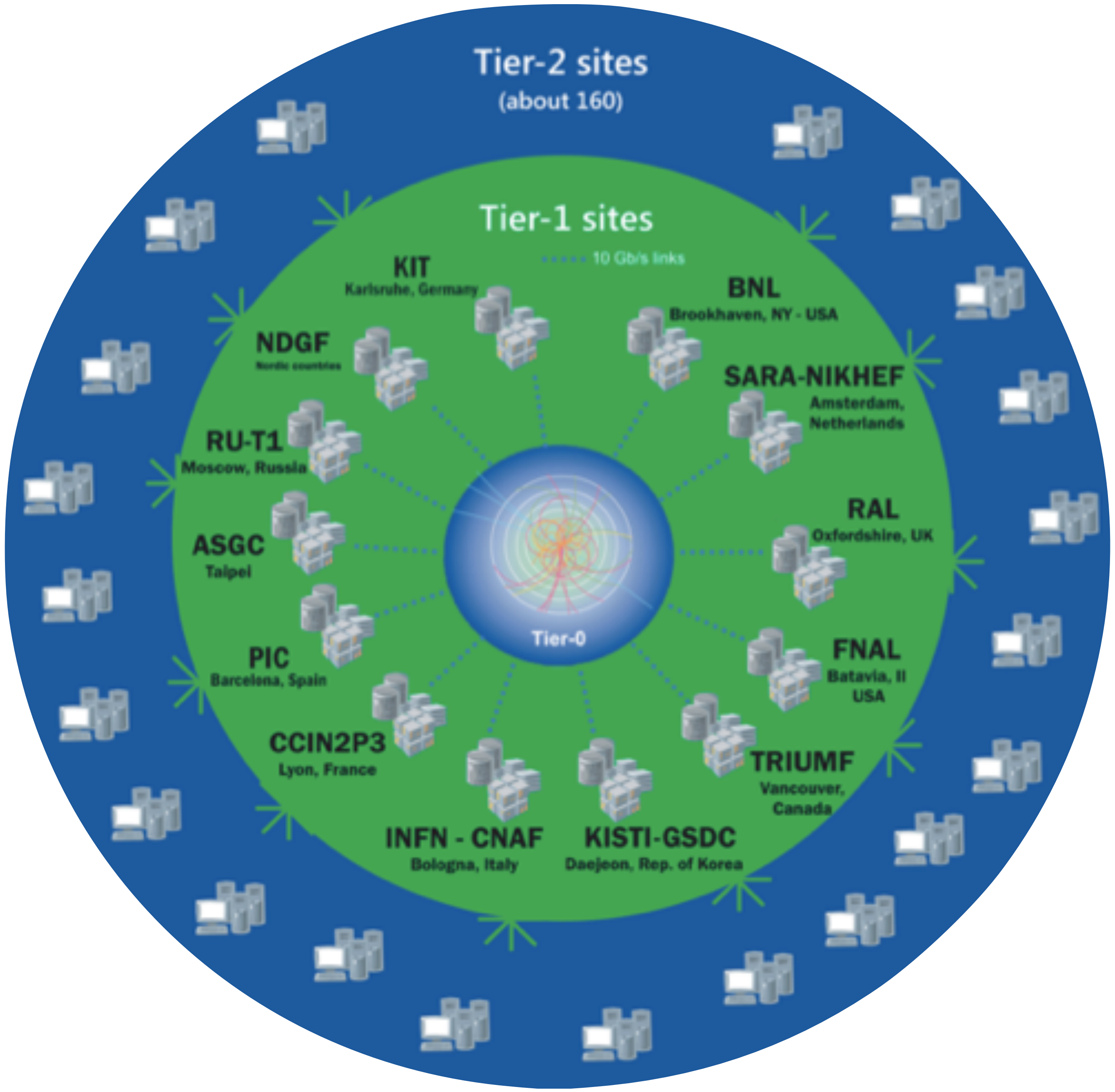
~12 PB of RAW data acquired in 2011-13, more than double that if we count the processed via the WLGC Grid using 100k cores sparse over 5 continents. We expect a 2-3 order of magnitude increase in data volume in the next 10 - 15 years.

...the WLGC Grid...

Tier-0 (CERN): data recording, reconstruction and distribution

Tier-1: permanent storage, re-processing, analysis

Tier-2: Simulation, end-user analysis



nearly 170 sites, 40 countries

~350'000 cores

500 PB of storage

> 2 million jobs/day

10-100 Gb links

WLCG:

An International collaboration to distribute and analyse LHC data

Integrates computer centres worldwide that provide computing and storage resource into a single infrastructure accessible by all LHC physicists.

CMSSW in perspective

	CMSSW (CMSSW_7_3_0)	Firefox (34.0.5)	OpenOffice Writer (4.1.1)	PovRay (3.7)	Clang (3.5)
SLOCs	5.5M	6.4M	4.7M	0.6M	0.9M
Initial release	2005	2002	2002	1991	2007
Contributors	>1300	>1200	>140	~40	~200
Typical memory footprint (RSS)	~2 GB	~0.3 GB	~0.2 GB		~0.2 GB
Primary languages	C++, Python, Fortran	C / C++, Javascript	C++ / Java	C / C++	C / C++ / ObjC

A bit of history

- We needed a **memory leak** detector, because that's where the first loop issues of big software stacks always are.
- In **2003** a brave duo (Lassi Tuura and yours truly) decided waiting for Valgrind (a fantastic tool, BTW) to finish was not an option.
- **MemProfLib** was born

MemProfLib

A **one** day prototype:

- Malloc Hooks

  Used `__malloc_hook` & c. to keep track of allocation / deallocations. Allocation not `free`d by the end of the program were reported as possible leaks. Code injected into programs via standard `LD_PRELOAD` mechanism, using `atexit` to trigger the dump.

- Flat output

  xml (sigh) output, analyzed by some XSLT magic (double sigh).

- Instant gratification

  It's quite amazing how much something like this can already catch. Always prove yourself you can do something cool before actually over-designing your tools.

...any reference to real facts or
persons is purely coincidental...

```
...
auto foo = new std::vector<SomeClass *>(); // sigh…
for (int i = 0; i < 1000000; ++i)
 foo->push_back(new SomeClass()); // double sigh
delete foo; // triple sigh..
...
```

ignominy |ˈɪɡnəmɪni|

noun [mass noun]

public shame or disgrace: *the ignominy of being imprisoned*.

ORIGIN mid 16th cent.: from French ***ignominie*** or Latin ***ignominia***, from ***in- 'not'*** + a variant of ***nomen 'name'***.

The Ignominious Profiler was born...

ignominy |ˈɪɡnəmɪni|
noun [mass noun]
ORIGIN mid 16th cent.: from French ***ignominie*** or Latin ***ignominia***, from ***in-*** ***‘not’*** + a variant of ***nomen*** ***‘name’***.

Key design decisions

Performance & **memory** profiling, with **backtraces**

Should work in managed environment

- ***No kernel support*** *required*
- ***No superuser*** *privileges required*

Target audience: people in CMS

- *Low overhead must be able to allow* ***interactive usage***
- *Results must be understandable to* ***non software professionals***

Target application: CMS software

- *Support for dynamic code / libraries*
- *Multiplatform: x86 / x86_64 / ARM32 / ARM64*

Reworking the internals

- Dynamic instrumentation (IgHook)
- Memory (by hooking into malloc) and performance profiler (via SIGPROF / SIGALRM)
- Full backtrace information (via libunwind)
- Analyser tool (igprof-analyse)
- Simple web frontend (igprof-navigator)

Dynamic instrumentation

Avoid extensions and platform specific APIs

`__malloc_hook` *& c. are glibc specific.*

Flexibility

We wanted to hook into more than just malloc (e.g. read / write statements).

Safety

We have to hook into various places to make sure we can catch things which interfere with the profiler (e.g. `fork`*), or even attempts to disable it (e.g. explicit calls to* `signal`*). Moreover we can safely hook into* `exit` *and* `_exit` *and dump the profile at that point.*

Dynamic instrumentation

We sit on the shoulders of giants

(1) *Jeffrey Richter, "Load Your 32-bit DLL into Another Pro- cess's Address Space Using INJLIB", Windows System Journal, Vol 9 No 5, May 1994.*

(2) *Shaun Clowes, "injectso: Modifying and Spying on Running Processes Under Linux and Solaris", The Black Hat Briefings, 2001, Amsterdam, http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt*

(3) *"DynInst: An Application Program Interface (API) for Run-time Code Generation", http://www.dyninst.org/*

(4) *https://github.com/rentzsch/mach_inject*

(5) *https://github.com/rentzsch/mach_override*

Dynamic instrumentation

Symbol start

```
41 56    push   %r14
41 55    push   %r13
41 54    push   %r12

55       push   %rbp
53       push   %rbx
…
rest of function body
…









c3       retq
```

Almost every symbol has a preamble to save registers, frame pointer, etc. We need enough bytes to replace part of it with a jump to our own code.

Trampoline structure

Original

```
e9 9f 3c 01 00  jmpq    0x41f004
90              nop
55              push    %rbp
53              push    %rbx
```

```
55              push    %rbp
53              push    %rbx
…
rest of function body
…

c3      retq
```

Trampoline

Jump to wrapper function

```
41 56   push    %r14
41 55   push    %r13
41 54   push    %r12
```

Jump to original code

Patch area (we might have copied `%rip` relative instructions in the preamble)

Trampoline structure

Original

```
e9 9f 3c 01 00   jmpq    0x41f004
90               nop
55               push    %rbp
53               push    %rbx
```

```
55               push    %rbp
53               push    %rbx
…
rest of function body
…

c3       retq
```

Trampoline

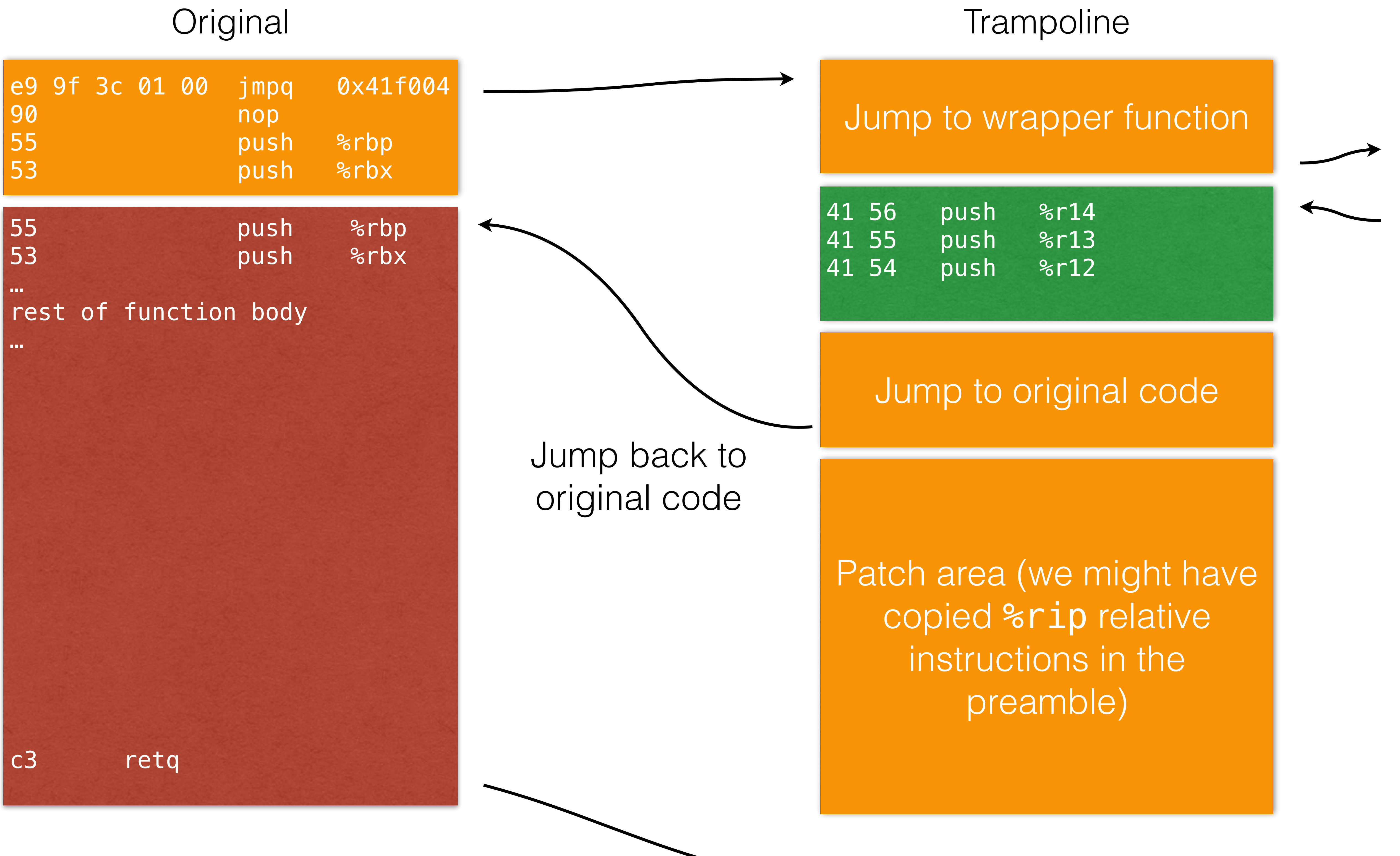
Jump to wrapper function

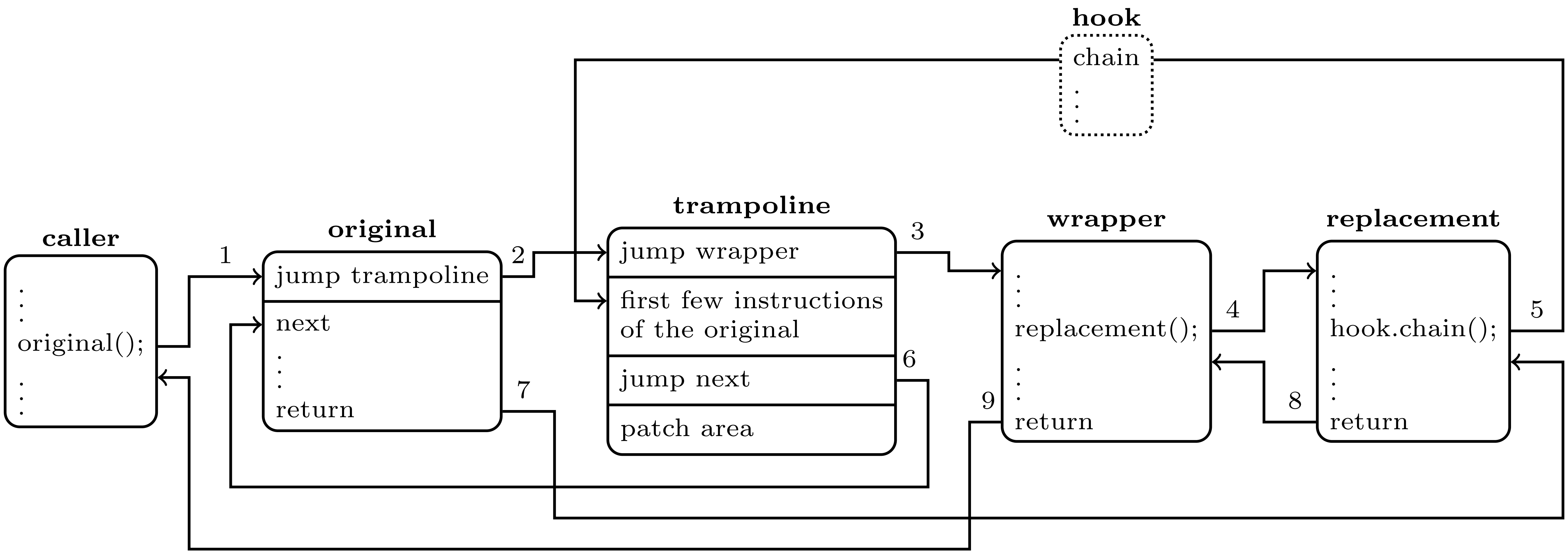
```
41 56    push    %r14
41 55    push    %r13
41 54    push    %r12
```

Jump to original code

Patch area (we might have copied `%rip` relative instructions in the preamble)

Trampoline structure





from Filip Nybäck thesis

Issues with DI

Relocability

`%rip` *relative instructions in the preamble need to be properly relocated.*

x86 assembly complexity

Parsing the preamble on x86 is not trivial, due to complexity of instruction set.

Short branches on ARM64

On ARM64 we likely have only 1 instruction (4 bytes) available for the jump. We limit trampoline distance so that we can use a `B` *instruction (±128MB jump ought to be enough for everyone…).*

Specificity to the rescue

The problem is simplified by the fact that usually we care about an handful of symbols so the phase space of the problem is usually limited (`malloc`*,* `calloc`*,* `exit`*,* `signal`*,* `fork`*, etc.). For this reason the above are not usually an issue for standard memory and performance profiling, however they might make generic instrumentation complicated (if not impossible at all).*

Memory profiling

Hooks into `malloc` & c

It only profiles heap allocations do not expect your 64MB array on stack to popup.

Three different kind of counters:

- `MEM_TOTAL`: sum of allocations in a call-path
- `MEM_LIVE`: sum of allocations from a given call-path, still present when profile dumps results
- `MEM_MAX`: largest **single** allocation in call-path

For each counter we store the number of calls and the allocated bytes. “Peak” mode also available.

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
 malloc(1);
...
<igprof dumps report>
```

MEM_LIVE and MEM_TOTAL are the same if the is no deallocation

	Counts	Calls	Peak
MEM_LIVE	10	10	10
MEM_TOTAL	10	10	10
MEM_MAX	1	10	1

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
 malloc(1);
...
<igprof dumps report>
```

MEM_MAX tracks single allocations

	Counts	Calls	Peak
MEM_LIVE	10	10	10
MEM_TOTAL	10	10	10
MEM_MAX	1	10	1

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
 malloc(n);
...
<igprof dumps report>
```

Counts are the sum of allocations, calls are how many times we called `malloc`

	Counts	Calls	Peak
MEM_LIVE	55	10	55
MEM_TOTAL	55	10	55
MEM_MAX	10	10	10

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
  free(malloc(n));
...
<igprof dumps report>
```

Once you start deallocating the difference between `MEM_LIVE` and `MEM_TOTAL` is obvious

	Counts	Calls	Peak
MEM_LIVE	**0**	0	**10**
MEM_TOTAL	55	10	55
MEM_MAX	**10**	10	10

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
  free(malloc(n));
...
<igprof dumps report>
```

MEM_LIVE in peak mode gives the largest block of memory at one point in time

	Counts	Calls	Peak
MEM_LIVE	**0**	0	**10**
MEM_TOTAL	55	10	55
MEM_MAX	**10**	10	10

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
{
  void *x = malloc(1);
  <igprof dumps report*>
  free(x);
}
...
```

Dumps can be triggered by calling a public symbol or by writing to a file specified by the **-D** option. In this particular case you get n reports.

	Counts	Calls	Peak
MEM_LIVE	n	n	n
MEM_TOTAL	Σ1..n	Σ1..n	Σ1..n
MEM_MAX	n	n	n

MEM_LIVE becomes a "*likely leaks*" checker!

Allocations are not what they seem

Malloc overhead profiler (`-mo`)

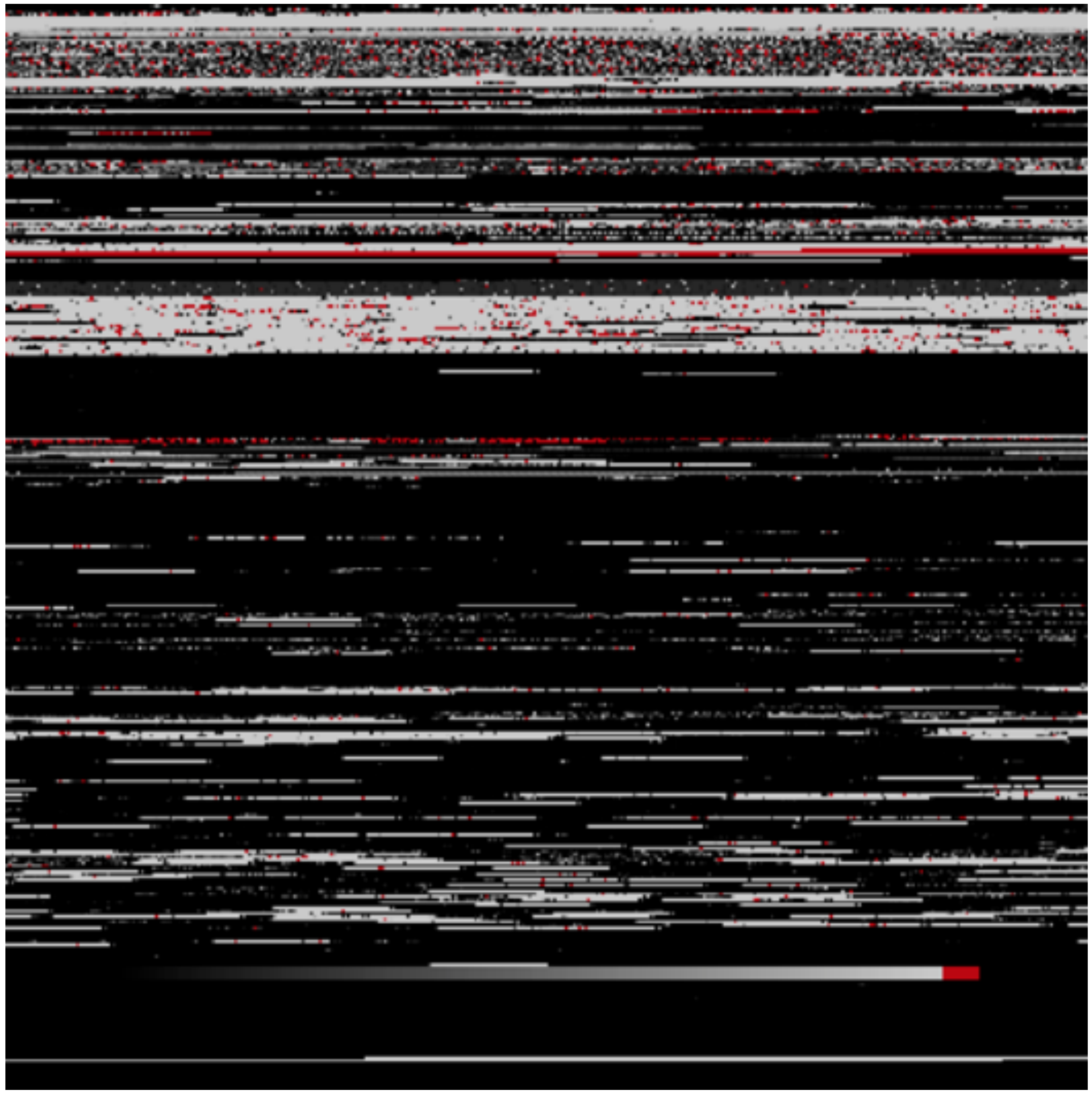
It's pretty common especially in "naive" Object Oriented code to underestimate the overhead of small allocations. For example `malloc` *on x86_64 allocates at 16 bytes borders and keeps at least one extra pointer for each allocation. The* `-mo` *option tries to account for "unseen" (by the programmer) overhead of small allocations via malloc.*

RSS profiling

When you are trying to reduce the RSS of your application (for example because your batch systems kills you if you are over 2GB of RSS), it's important to remember that memory always gets mapped into real memory in pages. A single 1 byte allocation in a new page will swap in the associated 4Kb page.

IgProf dumps contain the heap dump, and can report pages rather than bytes.

Allocations are not what they seem



Some CMSSW memory map

More goodies

File descriptor profiling

The previous concepts can actually be extended for any workflow which handles generic “resources” with a “cost” attached. For example you can count the number of writes / reads to a file by hooking into read / write.

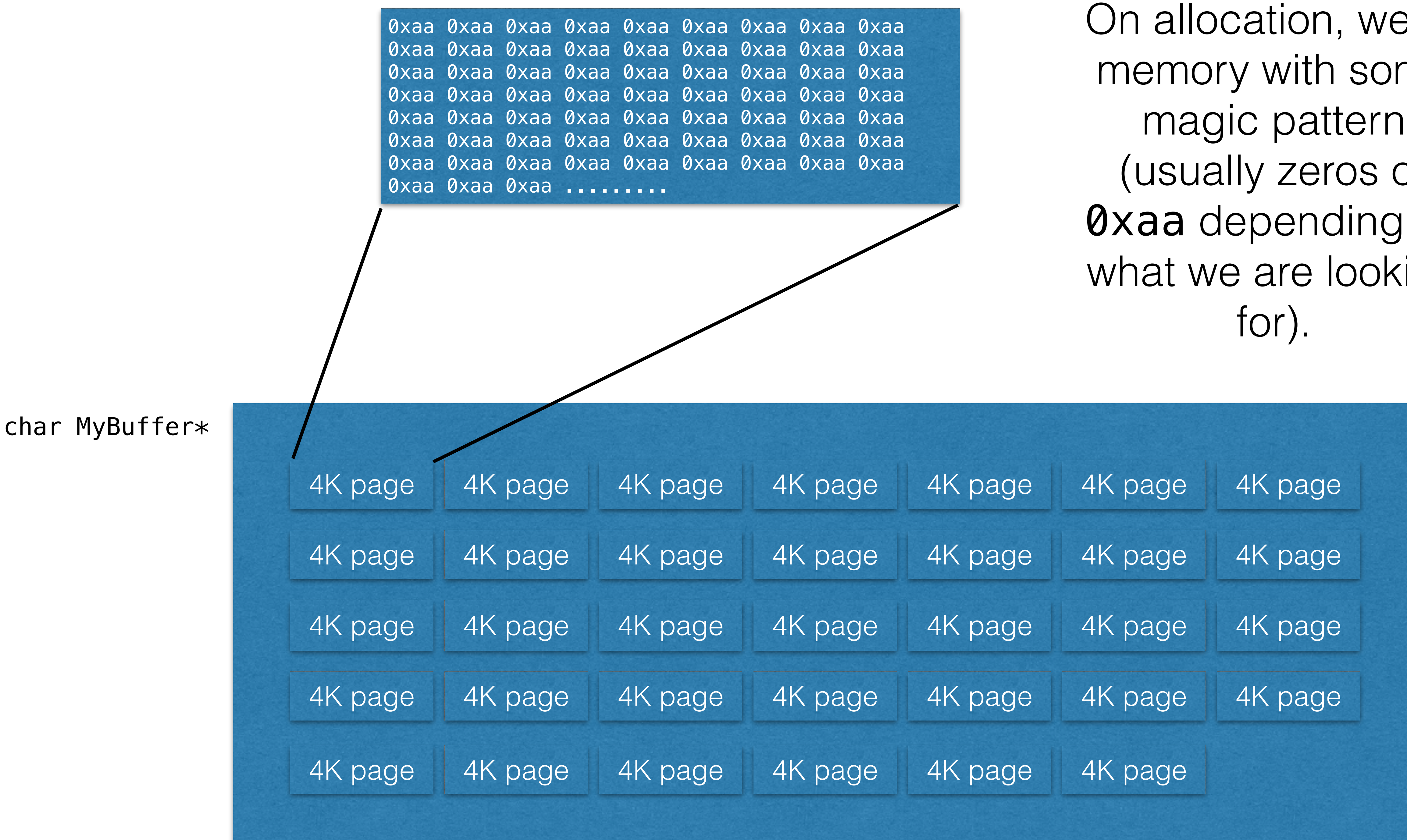
Tracing exceptions

A very common pattern we had was to use C++ exceptions as a way to communicate between different parts of the program. This is both slow and leads to unmaintainable code.

Empty memory profiler

Work done by Jakob Blomer @ CERN / SFT. Useful to tune your I/O buffers. On allocation, fill memory with some magic pattern (usually zeros or `0xaa` depending on what we are looking for). On free, scan for the same magic pattern counting untouched 4KB pages. At profile dump we report untouched pages.

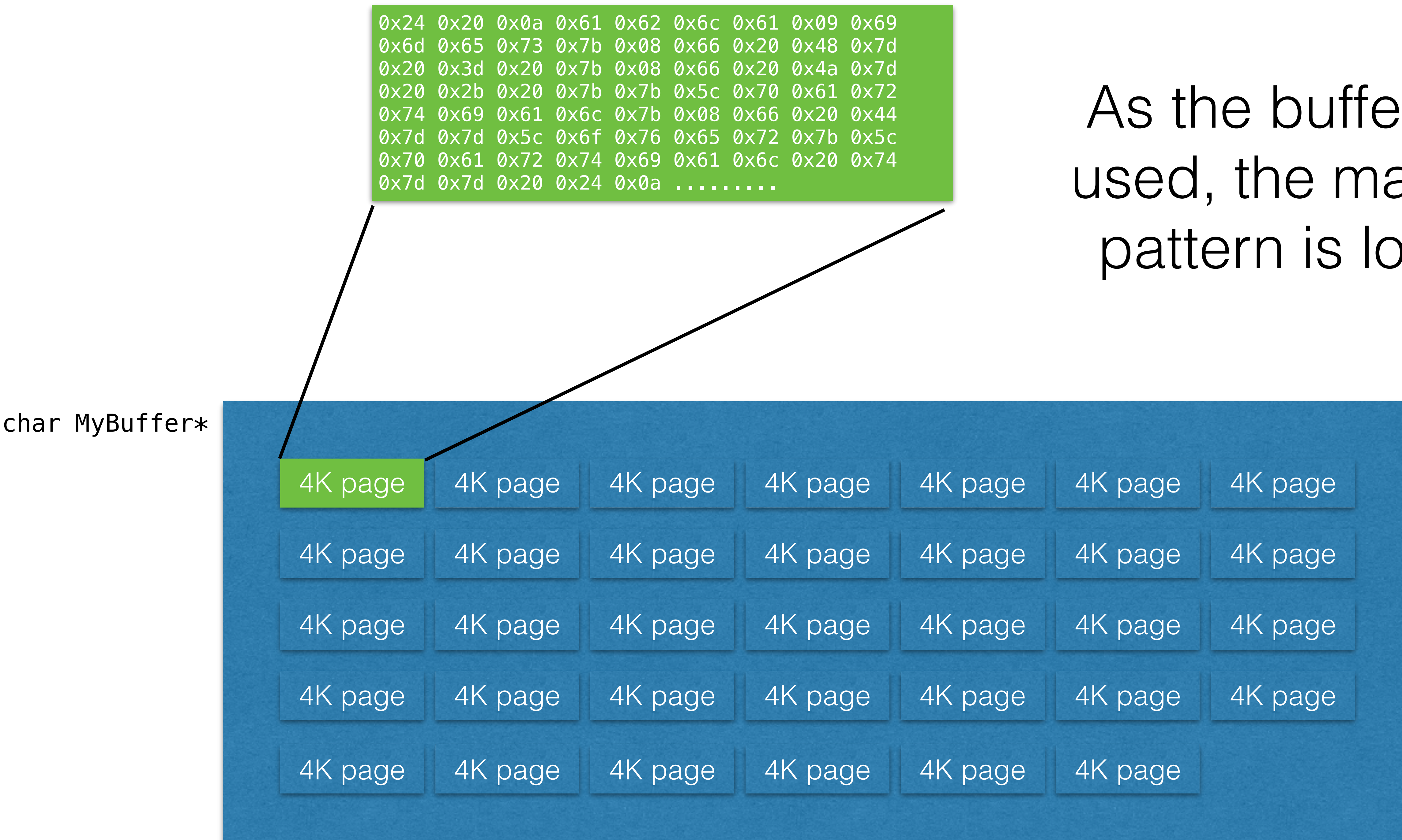
Empty memory profiling



On allocation, we fill memory with some magic pattern (usually zeros or `0xaa` depending on what we are looking for).

*for simplicity sake let's assume the buffer is 4KB aligned, works similarly if not.

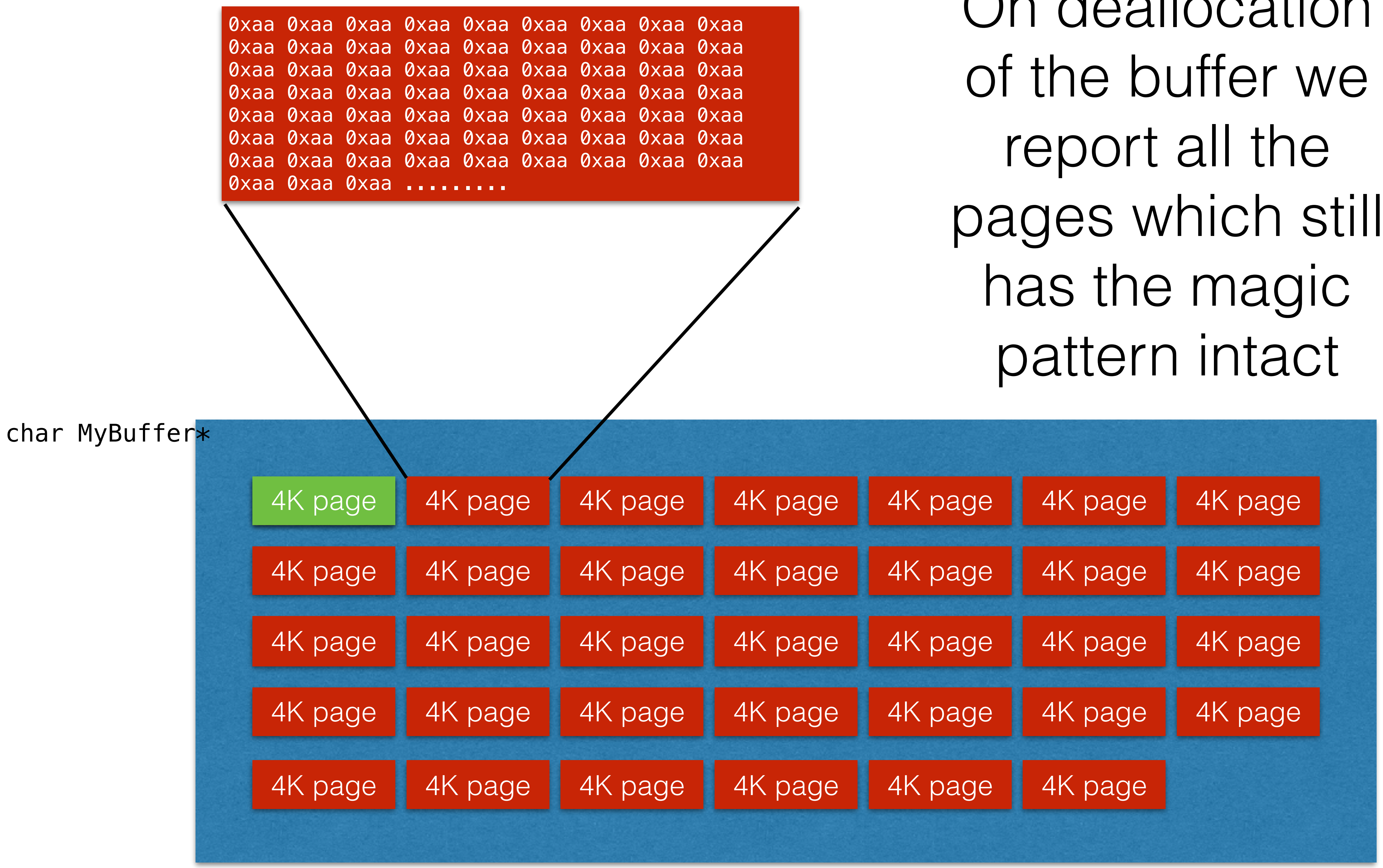
Empty memory profiling



As the buffer is used, the magic pattern is lost.

*for simplicity sake let's assume the buffer is 4KB aligned, works similarly if not.

Empty memory profiling



On deallocation of the buffer we report all the pages which still has the magic pattern intact

*for simplicity sake let's assume the buffer is 4KB aligned, works similarly if not.

Performance profiling

How it works

Uses SIGPROF to have time uniform callbacks every ~ 1/100s. Callback stores the backtrace of where the signal happened. Supports both CPU and wall-clock time. Biggest advantage is the limited interference with the program itself.

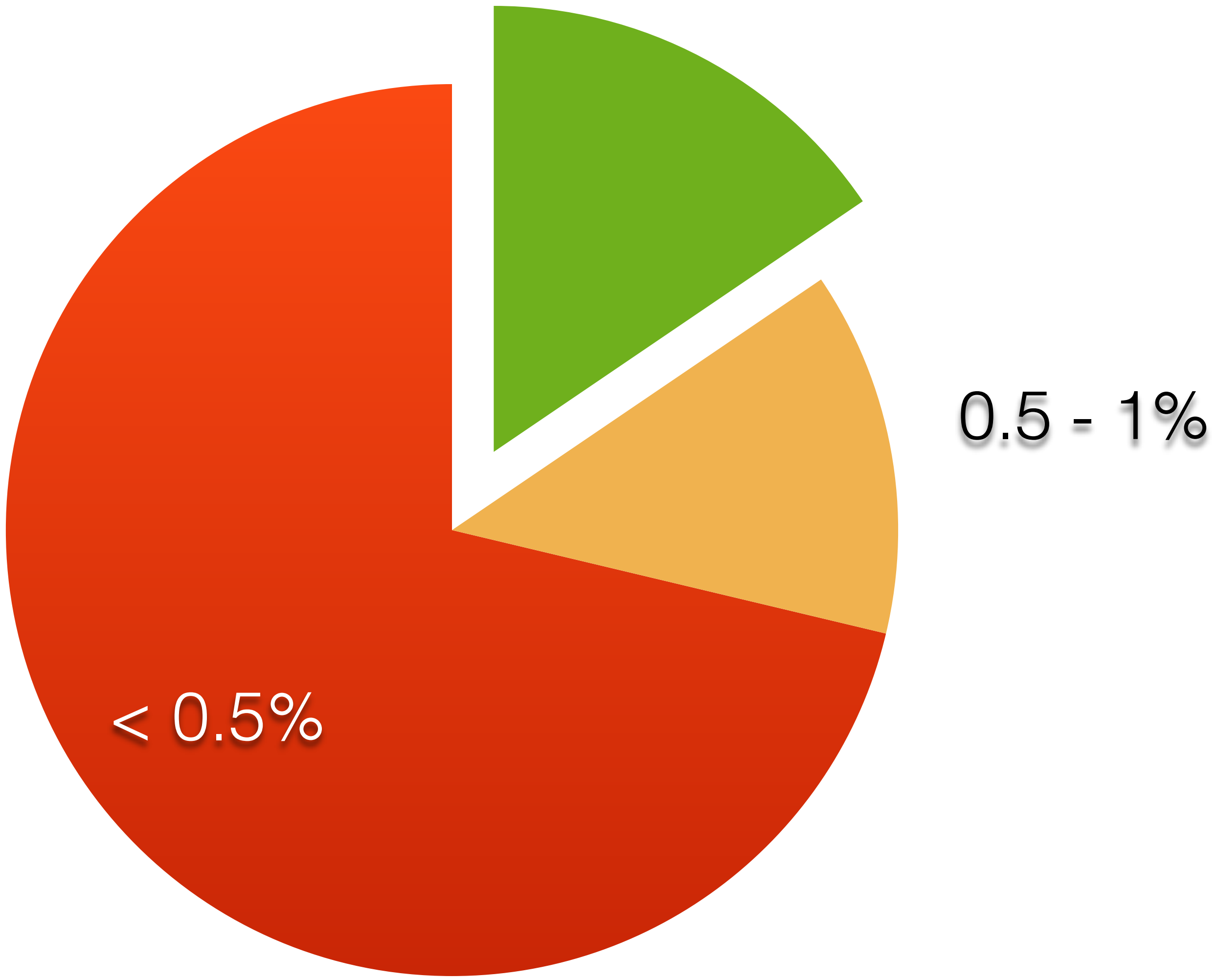
It does converge

If you wait long enough, this actually converges to the right distribution of time spent in any given function. Works brilliantly for repetitive payloads. Unsurprisingly results correlate with `MEM_TOTAL`*.*

Not so easy

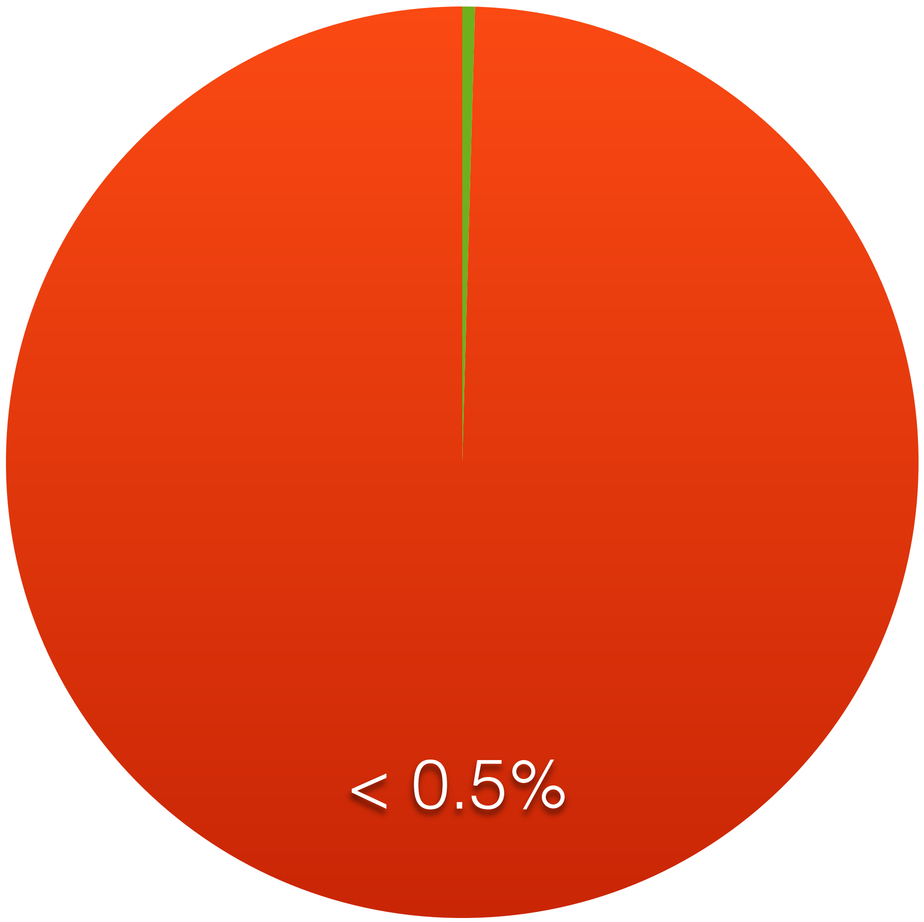
Of course the difficult part here is making sure you are not interrupting something which shall not be interrupted. For example igprof has problems when a signal happens in `dl_iterate_phdr`*. Another issue was making sure that we accounted for the time spent in* `fork` *due to the large RSS which had to be duplicated.*

Time spent in functions with more than 1% of contribution



Vast majority of the time is spent in functions which themselves have little contributions. Say thanks to C++ encapsulation.

Number of functions with more than 0.5% of contribution



…and things look even worse if we count the symbols, rather than their contribution… Only 27 symbols have more than 0.5% of the time spent in them, out of ~6000 which gets counts...

Call tree

```
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```

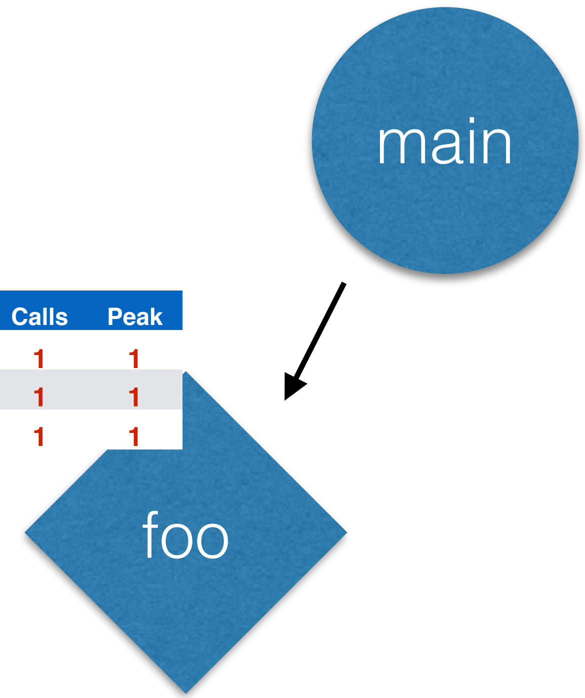
In large programs what is actually interesting is to know not only which functions allocated most of the memory, but why.

Call tree

```
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```



	Counts	Calls	Peak
MEM_LIVE	1	1	1
MEM_TOTAL	1	1	1
MEM_MAX	1	1	1

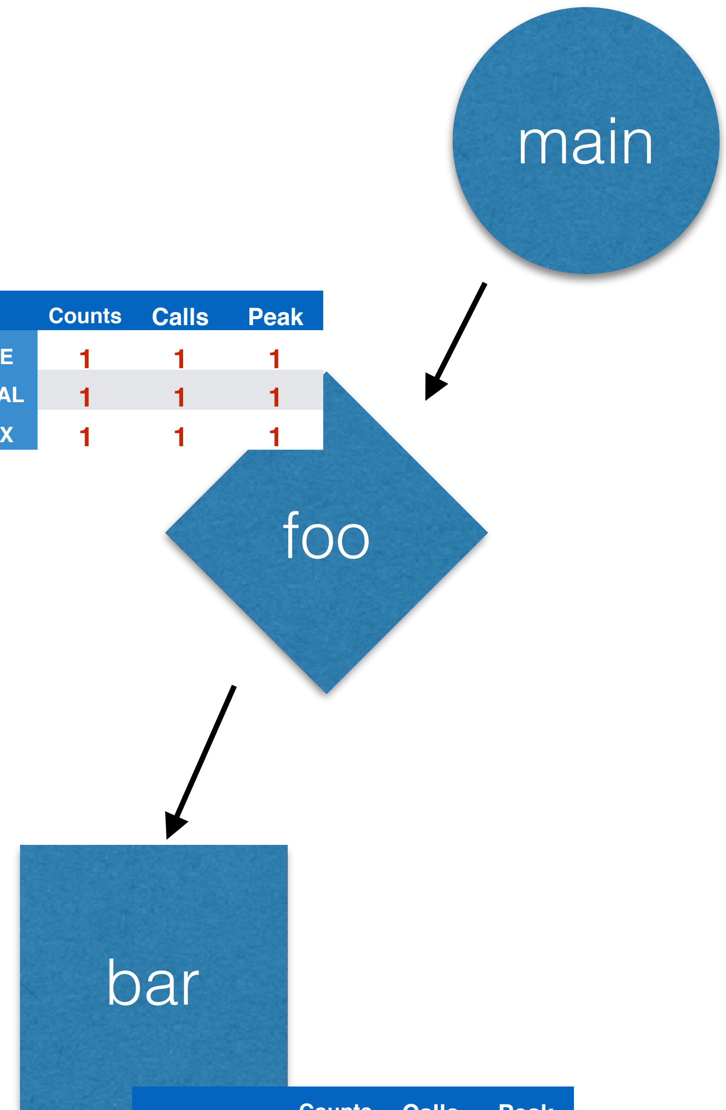
Every time an instrumented function is called we determine the full calltree.

Call tree

```c
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```



	Counts	Calls	Peak
MEM_LIVE	1	1	1
MEM_TOTAL	1	1	1
MEM_MAX	1	1	1

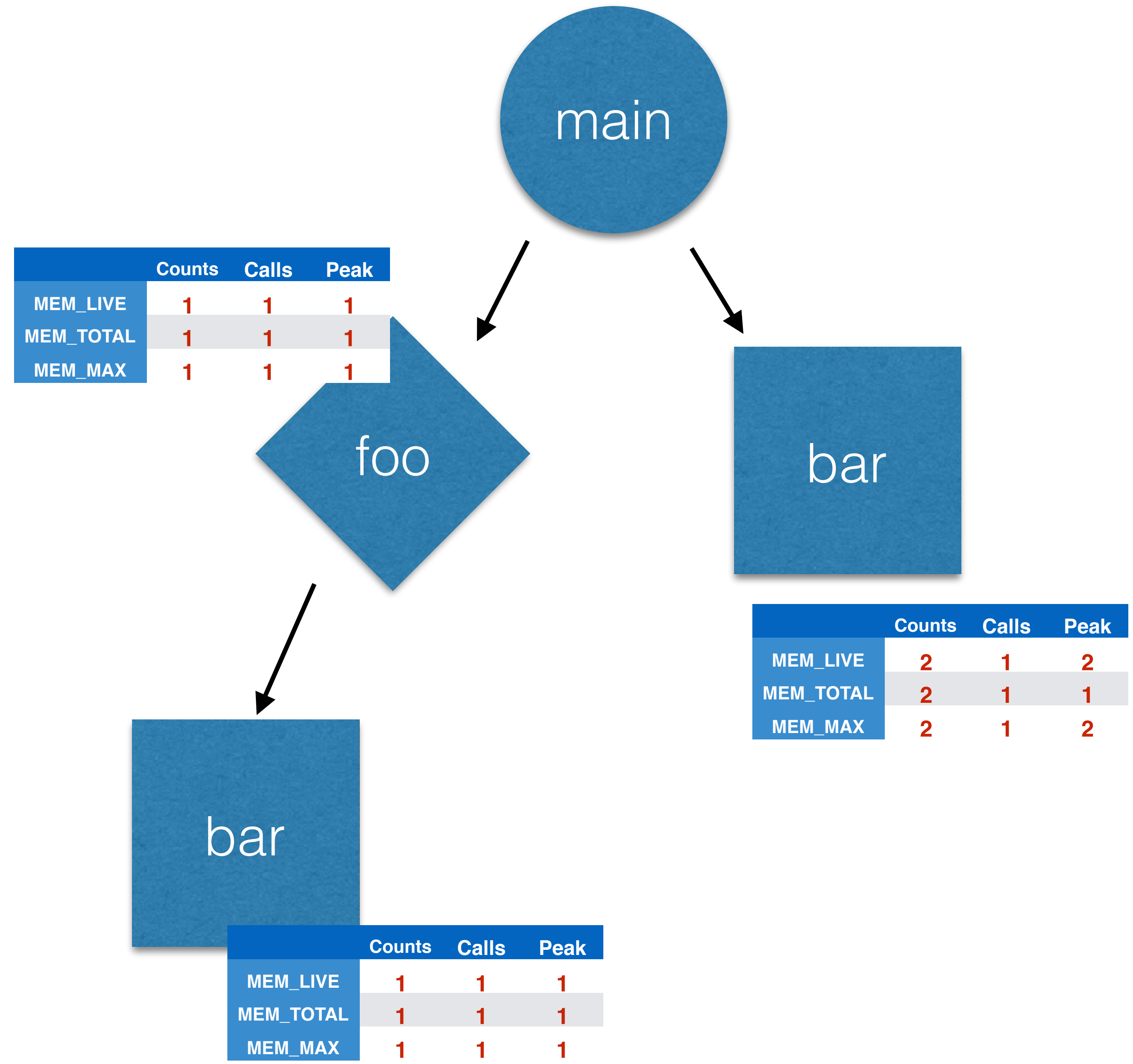
	Counts	Calls	Peak
MEM_LIVE	1	1	1
MEM_TOTAL	1	1	1
MEM_MAX	1	1	1

Call tree

```
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```



	Counts	Calls	Peak
MEM_LIVE	1	1	1
MEM_TOTAL	1	1	1
MEM_MAX	1	1	1

	Counts	Calls	Peak
MEM_LIVE	2	1	2
MEM_TOTAL	2	1	1
MEM_MAX	2	1	2

	Counts	Calls	Peak
MEM_LIVE	1	1	1
MEM_TOTAL	1	1	1
MEM_MAX	1	1	1

Fast path backtracing

Backtrace using libunwind

This was initially done using `backtrace()`*, but then we switched to* ***libunwind*****, due to reliability issues.*

Optimizing libunwind

In our application we had up to 1M allocations per second which made memory profiling too slow due to many unwindings. Lassi implemented a fast path which does not do the full DWARF unwind but a simpler stack walk with fallback to the full blown method only when it does not work.

Contributing back to libunwind

- *x86_64 implementation by Lassi Tuura*
- *ARM32 / ARM64 implementation by Filip Nybäck (30x speedup reported!)*
- *Volunteers wanted to implement it on Power Architecture ;-)*

*: http://www.nongnu.org/libunwind/

Generic Instrumentation

Why not expanding beyond malloc?

Give user the flexibility to hook into any generic function. Needs to know the mangled symbol name and register signature.

Precise results

- *CALL_TIME: total time spent in the requested function (via RDTSC)*
- *CALL_COUNT: number of times a function has been called*

Support GCC `-finstrument-functions`

When recompiling is an option, GCC and others support instrumenting every function (including inlines) via the `-finstrument-functions` *flag which introduces* `__cyg_profile_func_enter` *and* `__cyg_profile_func_exit` *entry / exit points.*

Overhead too large to actually think about doing it for every compilation unit, probably more interesting to understand exactly who calls what.

PAPI Support

What is it?

Performance Application Programming Interface (PAPI) provides an interface and methodology for use of the performance counter hardware found in most major microprocessors. Very nice interface to read performance counters (using either of perf_events, PerfCtr, Perfmon). Requires kernel component so it's optional in igprof. http://icl.cs.utk.edu/papi/.

How it works?

Similar to the Performance profiler, we get counted at regular intervals and check PAPI counters at that point. If the counter overflows we count 1 in the igprof results. While this should converge, deciding the overflow level is currently completely empiric.

Energy measurements

Recently there has been big noise about "Power efficient computing". Intel provides nice counters which allow to measure power consumption with a low granularity (RAPL). Given PAPI in particular supports RAPL, we have started doing tests using it. The main problem is that energy consumption is really a global quantity, not a local (to the application) one. Not clear if it simply correlates to CPU performance. Different metrics (e.g. how many times CPU changes state)?

Analyzing results

igprof-analyse

Dumping a 50MB gzipped file full of profile data is useless if you cannot extract information from it. igprof-analyse takes a profile dump and produces (somewhat) human readable reports from it. Tries hard to be accurate when doing symbol name demangling, using `nm` *and* `gdb`*.*

It also aggregates call-paths, allows applying various filters on the call-tree, both by changing contents of single nodes and by merging nodes together.

Exports results in a gprof like text file, sqlite db, or JSON object

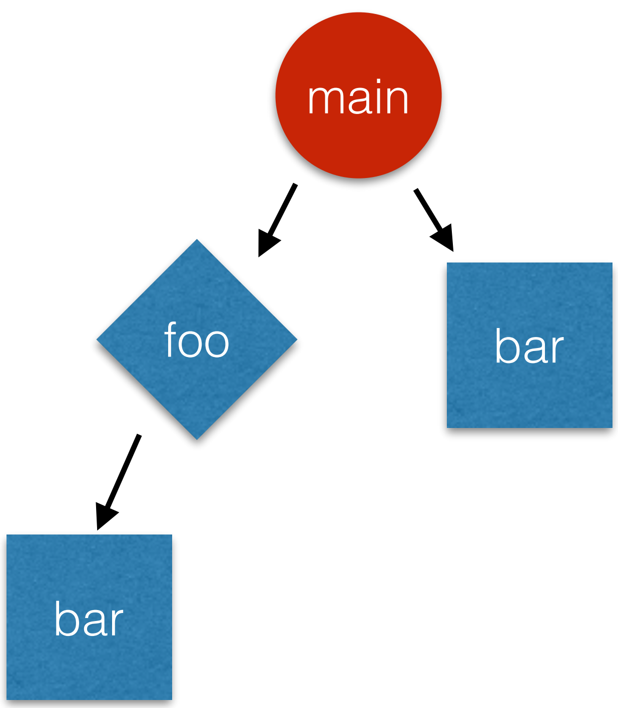
igprof-navigator

A poor man cgi script / python server which allows you to navigate results via a web interface. Uses the sqlite report as input.

Analyzing results

gprof like report format

For each node in the callpath, we aggregate edge information for nodes going into the selected node (i.e. callers) and nodes going out (i.e. callees). Considered node symbol is indented to highlight it.



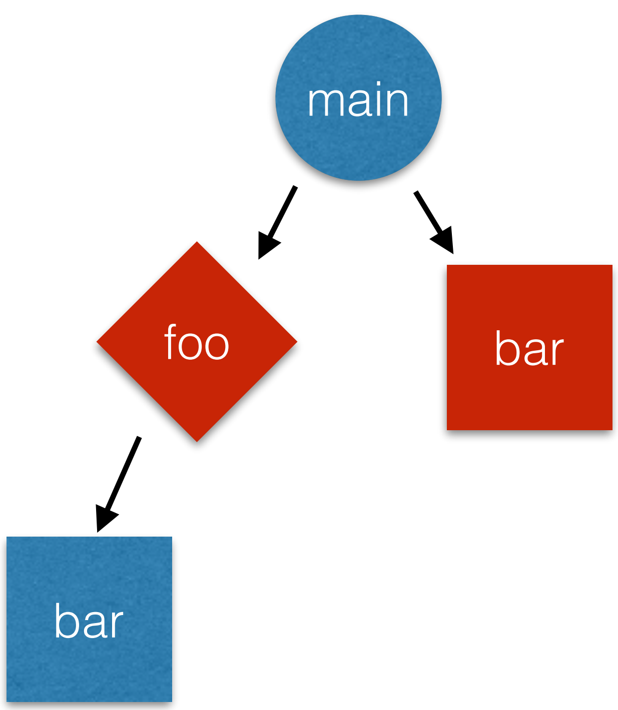

```
Rank % total      Self   Self / Children  Calls / Total   Function

[1]     100.0        4      0 / 4             3              main
         50.0  ........     2 / 3             1 / 2            bar(int) [2]
         50.0  ........     2 / 2             2 / 2            foo() [3]
```

Analyzing results

gprof like report format

Lines below the selected symbol are callees.



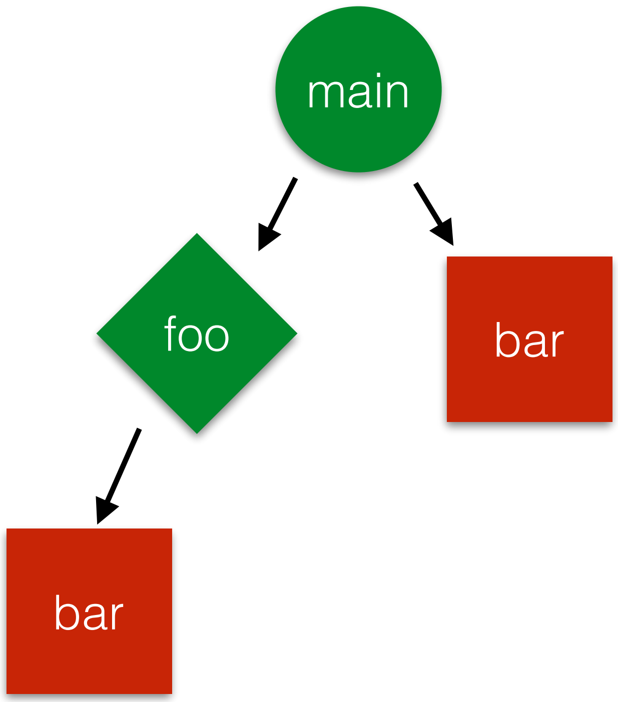

```
Rank % total      Self   Self / Children  Calls / Total    Function

[1]      100.0          4      0 / 4            3            main
          50.0  ........      2 / 3            1 / 2          bar(int) [2]
          50.0  ........      2 / 2            2 / 2          foo() [3]
```

Analyzing results

gprof like report format

Lines below the selected symbol are callers.



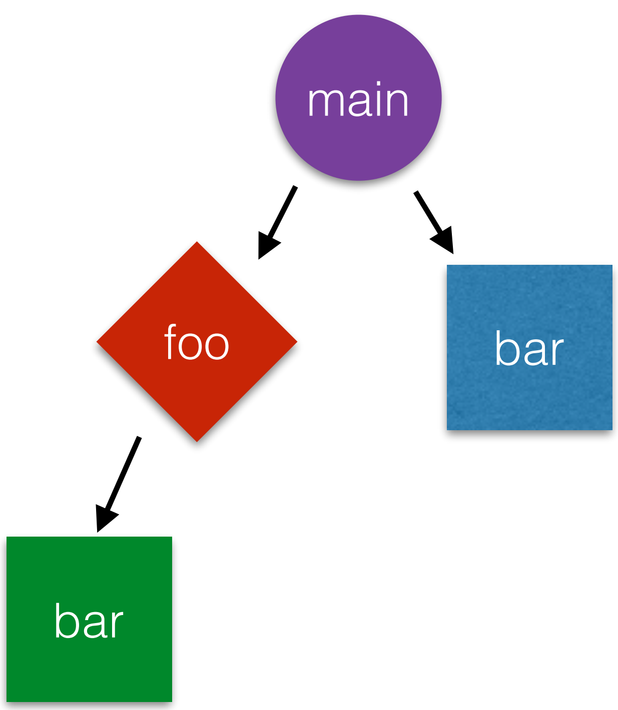

```
Rank % total      Self   Self / Children  Calls / Total    Function

        25.0  ........      1 / 2             1 / 2           foo() [3]
        50.0  ........      2 / 4             1 / 3           main [1]
[2]     75.0         3      3 / 0             2              bar(int)
```

Analyzing results

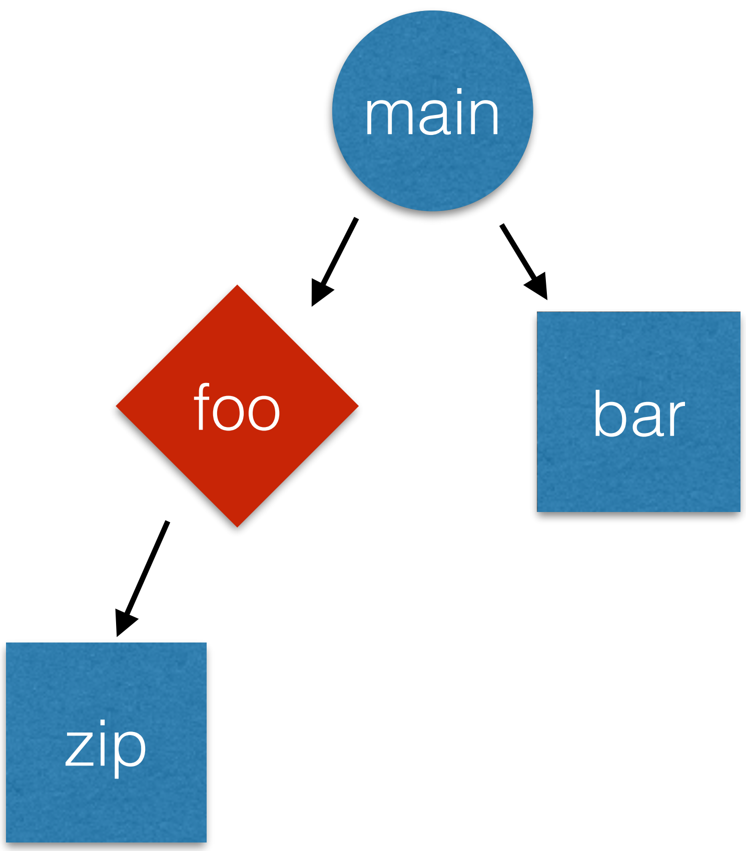
gprof like report format

Lines below the selected symbol are callers.



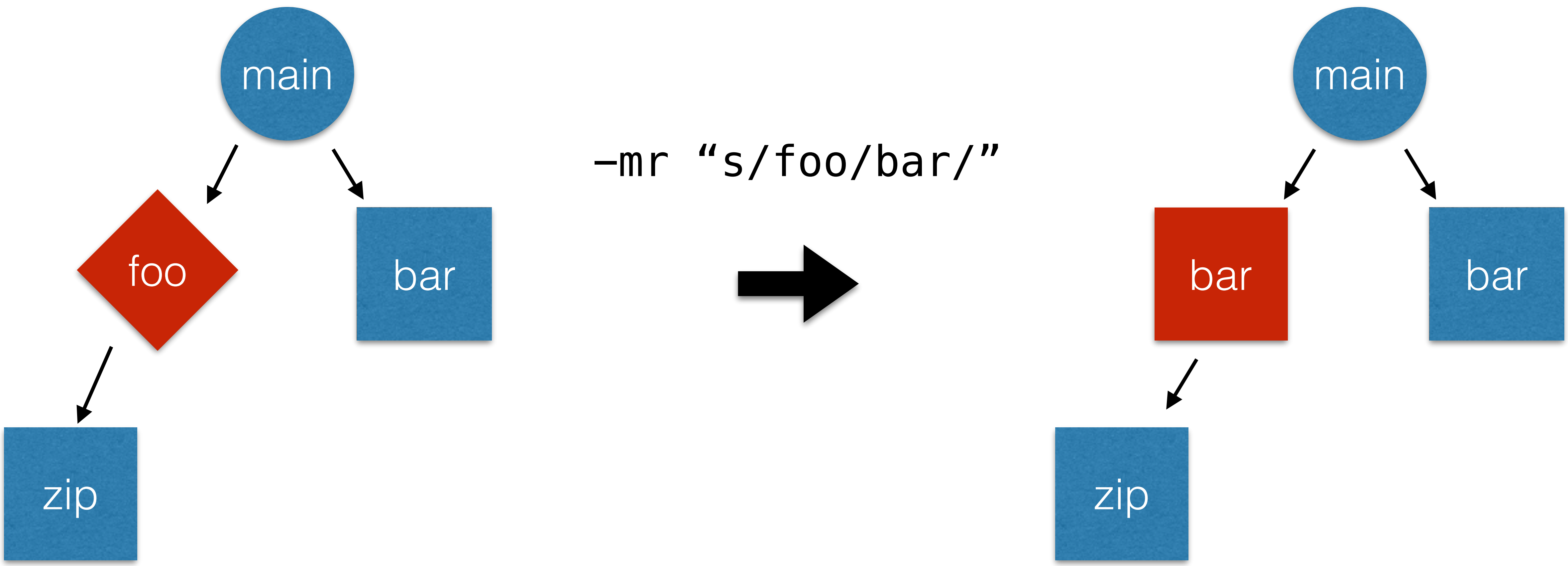
Rank	% total	Self	Self / Children	Calls / Total	Function
	50.0	....	2 / 4	2 / 3	main [1]
[3]	50.0	2	1 / 1	2	foo()
	25.0	....	1 / 3	1 / 2	bar(int) [2]

Merging nodes in reports

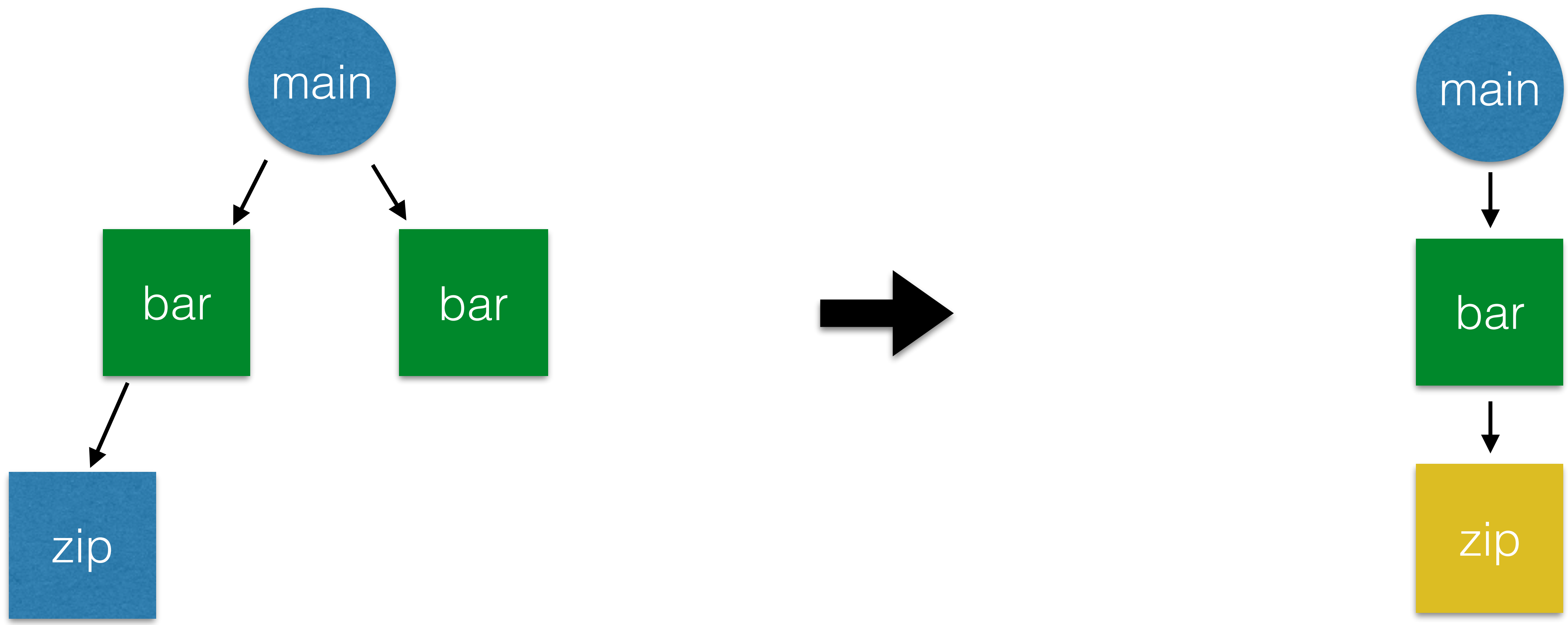


Sometimes it can be useful to rename nodes so that slightly different versions of the same are merged together.

Merging nodes in reports



Merging nodes in reports



Merging nodes in reports

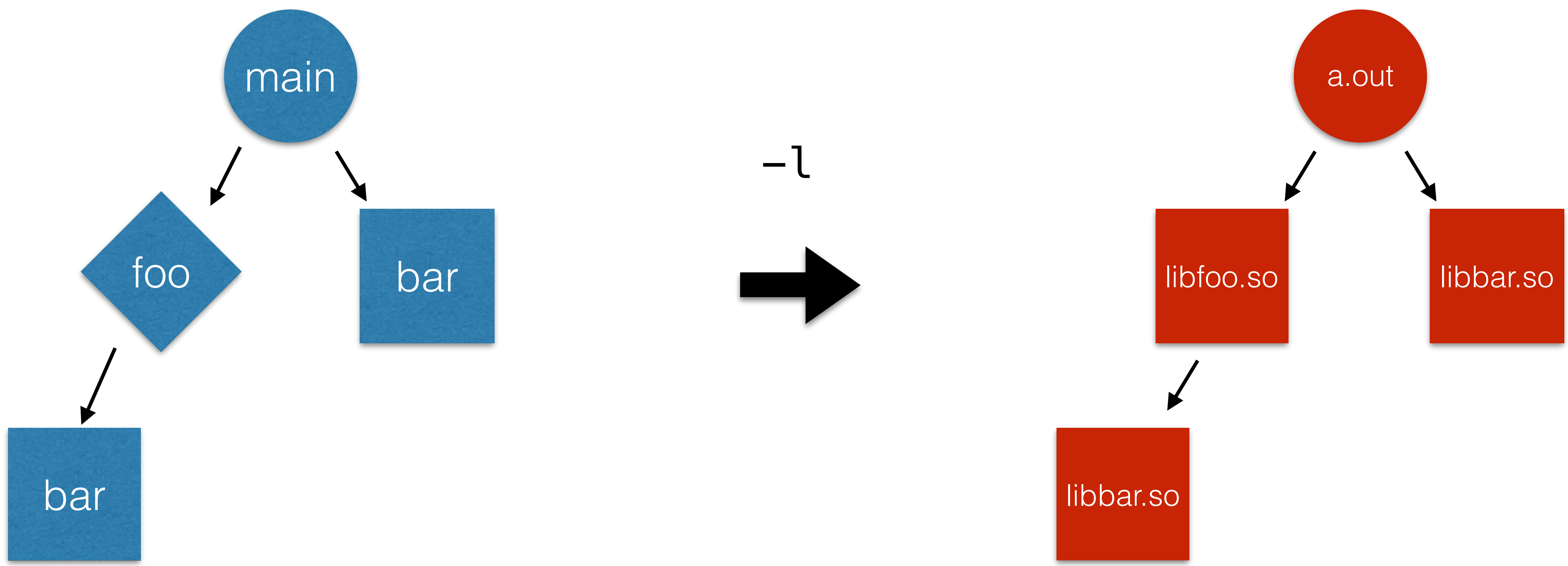
Surviving template-ed designs

This is actually extremely useful in case of heavily template designs where (for example) you have methods template-ed on their inputs.

```
class Foo {
   template <typename A>
    void useA(const A &a) { <do something with A> }
};
```

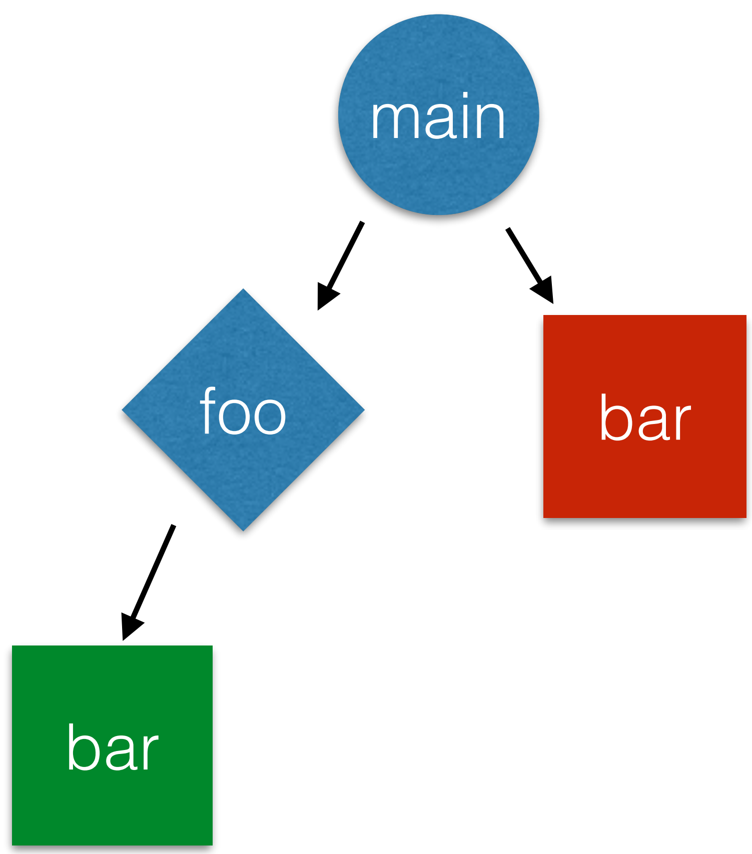
While normally each `useA<A>` *would appear as a single instance (different symbol!) using something like* `–mr "s/.*useA.*/Foo::useA/"` *allows merging all the various small contribution from each instantiation into one single contribution.*

Merging libraries



Might not be so interesting for few libraries, but it is if you have 600 between libraries and plugins

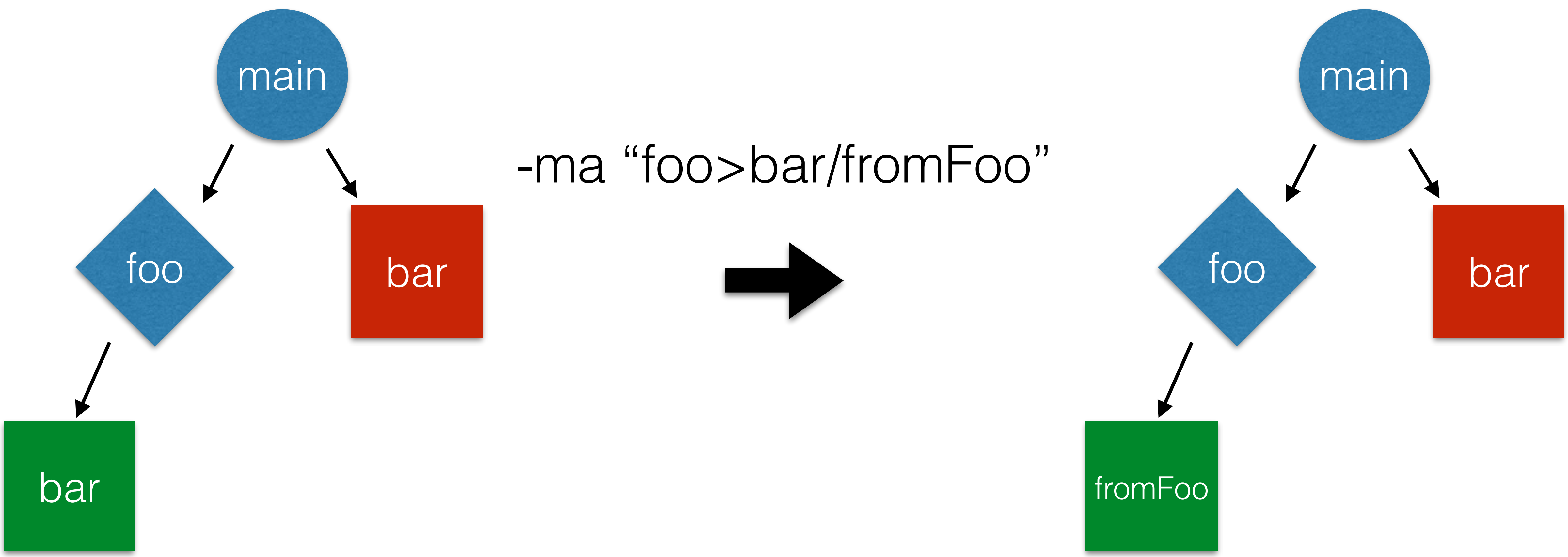
Splitting by ancestor



Sometimes we want to distinguish between same symbol which gets invoked from different call path.

One simple solution is to rename symbols associated to a node according to their parents.

Splitting by ancestor



IgProf web

Instant gratification

`igprof-navigator` *is a simple CGI script which we use to present reports as webpages. Especially in an environment of non software developers, it is important to have a clickable report at which experts can use to drive non-experts through performance optimization.*

Clang examples

MEM_LIVE, x86_64, dumped while compiling a monster machine generated file: http://cern.ch/go/k9Sw

Firefox benchmark (RoboHornet) examples

MEM_TOTAL, x86_64: http://cern.ch/go/nG7C , PERF_TICKS, x86_64: http://cern.ch/go/H9ct

CMSSW examples

…and for the High Energy Physics enthusiasts among you, a few CMS experiment software examples:

PERF_TICKS, x86_64: http://cern.ch/go/NWg9, PERF_TICKS, ARM64: http://cern.ch/go/7M9D

IgProf web

igprof_pp_25202.0_step3 - x86_64, igprof-navigator

Back to profiles index

Counter: PERF_TICKS, first 1000 entries

Sorted by cumulatative cost

(Sort by self cost)

Rank	Total %	Cumulative	Symbol name
1	100.00	2,130.54	<spontaneous>
3	97.40	2,075.14	__libc_start_main
2	97.40	2,075.14	_start
4	97.36	2,074.30	main
5	97.31	2,073.21	main::{lambda()#1}::operator()() const
6	94.11	2,005.03	edm::EventProcessor::runToCompletion()
7	94.11	2,005.02	boost::statechart::state_machine<statemachine::Machine, statemachine::Starting, std
12	92.60	1,972.82	edm::EventProcessor::readAndProcessEvent()
11	92.60	1,972.82	statemachine::HandleEvent::readAndProcessEvent()
10	92.60	1,972.82	statemachine::HandleEvent::HandleEvent(boost::statechart::state<statemachine::Handl
9	92.60	1,972.82	boost::statechart::state<statemachine::HandleEvent, statemachine::HandleLumis, boos
8	92.60	1,972.82	boost::statechart::simple_state<statemachine::FirstLumi, statemachine::HandleLumis,
15	92.60	1,972.81	edm::EventProcessor::processEventsForStreamAsync(unsigned int, std::atomic<bool>*)
14	92.60	1,972.81	edm::StreamProcessingTask::execute()
13	92.60	1,972.81	tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::local_wait_fc
16	92.59	1,972.62	edm::EventProcessor::processEvent(unsigned int)

IgProf web

Counter: PERF_TICKS

Rank	% total	Counts: to / from this	Counts: Total	Paths: Including child / parent	Paths: Total	Symbol name
	0.06	1.28	21.22	1	1	RecoMuonValidator::analyze(edm::Event const&, ed
	0.21	4.48	6.95	1	1	ObjectSelector<SingleElementCollectionSelector<s
	0.27	5.78	36.41	1	1	MuonTrackValidator::analyze(edm::Event const&, e
	1.35	28.77	30.36	1	1	MTVHistoProducerAlgoForTracker::fill_recoAssocia
	5.21	111.03	817.76	1	1	MultiTrackValidator::analyze(edm::Event const&,
[31]	**7.10**	**19.08**	**132.26**	**5**	**5**	**TrackingParticleSelector::operator()(TrackingPar**
	2.51	53.38	56.47	5	12	TrackingParticle::charge() const
	1.82	38.74	80.06	5	311	log
	0.88	18.76	26.10	4	14	ROOT::Math::Cartesian3D<double>::Eta() const
	0.61	13.07	14.19	5	9	TrackingParticle::vertex() const
	0.28	6.02	6.14	5	8	TrackingParticle::momentum() const
	0.05	1.08	1.08	4	4	TrackingParticle::numberOfTrackerLayers() const
	0.03	0.62	2.99	2	5	TrackingParticle::eventId() const
	0.03	0.58	1.86	2	15	TrackingParticle::pdgId() const

Back to summary

Contributing to FOSS

We use Free and Open Source Software

CMS experiment is a eager user of FOSS, and we evangelise with other experiments and CERN about its benefits. We actively report bugs and provide patches to tools like gcc, glibc and others.

We write FOSS

IgProf itself is GPLv2. Lassi and Filip contributed back to libunwind the fast path tracing which is now in upstream and available for everyone to use.

We train to use FOSS

We see lots of students coming through CERN and they all get a full immersion in using FOSS tools for their work. Kids get in they only know about Visual Studio and get out being vim wizards.

Working on IgProf

Patches & ideas welcome

CERN / CMS institutes summer programs

CERN / CMS institutes have various student programs, where IgProf is usually one of possible projects

CERN @ GSoC

CERN has been mentoring organization for the last few years: http://cern.ch/go/KM7W.

IgProf was part of GSoC2014

Great work by ***Filip Nybäck*** *from Aalto University*

- *ARM64 port itself.*
- *Fast path backtrace in libunwind also for ARM64*
- *PAPI support and initial energy profiling as a bonus*

Ideas for GSoC 2015?

Support for more architectures

We have a POWER7 system and we might get a POWER8 dev board. x32 support is also missing and requested by a few people. This should probably in a contribution to libunwind as well.

“Big Data” igprof

When running igprof on the validation of your integration builds you end up with a bunch of data which from which to extract sensible information. Right now this has to be done by hand. The idea would be pushing igprof reports to some key-value storage and the use automated tools to spot problems in your code.

Python / Javascript backtrace support

Right now, when profiling mixed C / C++ code invoking / invoked by scripting languages like python or javascript, igprof will happily show the cost of the interpreter / JIT itself, not the actual python code. The idea would be to instrument…

Profile more counters via PAPI

Pick your favorite

	License	x86	ARM	Power Architecture	HW counters	Generic Instrumentation	Heap	Sampling	kernel / root
igprof	GPL	✔	✔		✔*	✔	✔	✔	
gprof	GPL	✔	✔	✔				✔	
Google Performance Tools	BSD	✔	✔	✔			✔	✔	
Oprofile	GPL	✔	✔		✔			✔	✔
perfctr	GPL	✔	✔	✔	✔			✔	✔
perfmon2	BSD	✔	✔	✔	✔			✔	✔

...and many others...

Thanks!

Andreas Pfeiffer, Chris Jones, **David Abdurachmanov**, Filip Nybäck, Ian Bird, Jakob Blomer, Jukka Nurminen, **Lassi Tuura**, Liz Sexton-Kennedy, Lothar Bauerdick, Lucas Taylor, Matevž Tadel, Mikko Kurkela, Peter Elmer, René Meusel, Robert Lupton, Shahzad Muzaffar, Stephen Reucroft, Vincenzo Innocente

Contact info

http://igprof.org

giulio.eulisse@cern.ch

@ktf

@ktf