HEP C++ meets reality

Giulio Eulisse & Lassi Tuura
Northeastern University

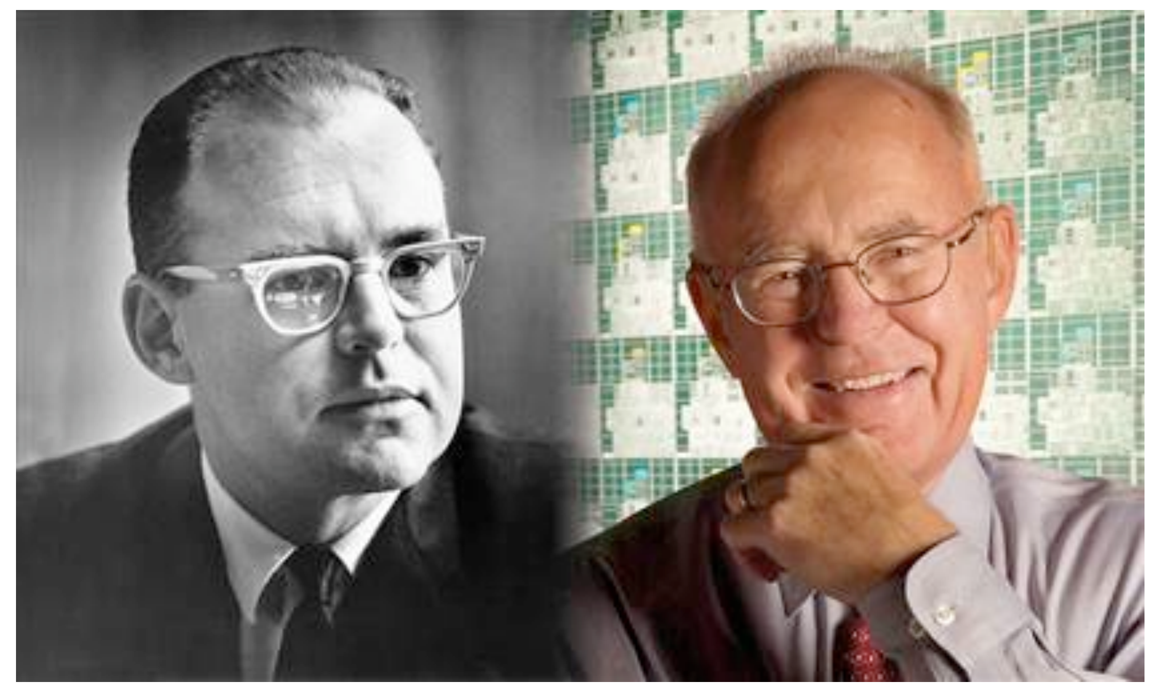
Peter Elmer
Princeton

on behalf of the CMS Computing & Offline Projects

Moore’s law

*Hardware advances double computing power every **18 months***

Gordon E. Moore, Intel Co-founder



Proebsting's Law

Compiler Advances Double Computing Power Every ***18 Years***

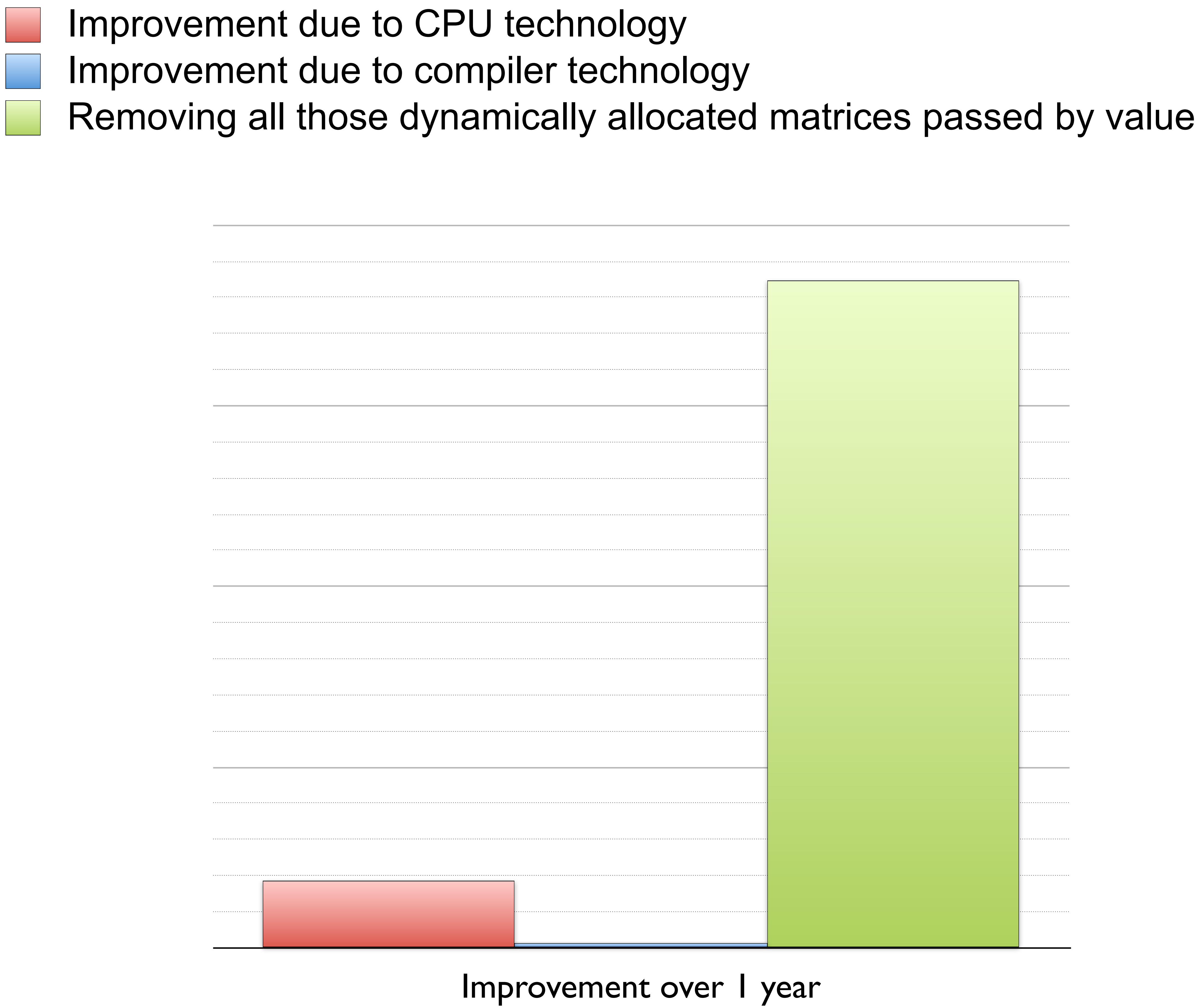
Todd Proebsting

Director of the Centre for Software Excellence
Microsoft's Platform and Services Division

Tuura's Law

CMS manages to get ***an order of magnitude*** *improvement every 18 months by ~~removing some broken piece of code~~ improving algorithms*

Lassi Tuura: two offices down my corridor


Improvement due to CPU technology
Improvement due to compiler technology
Removing all those dynamically allocated matrices passed by value
Improvement over 1 year

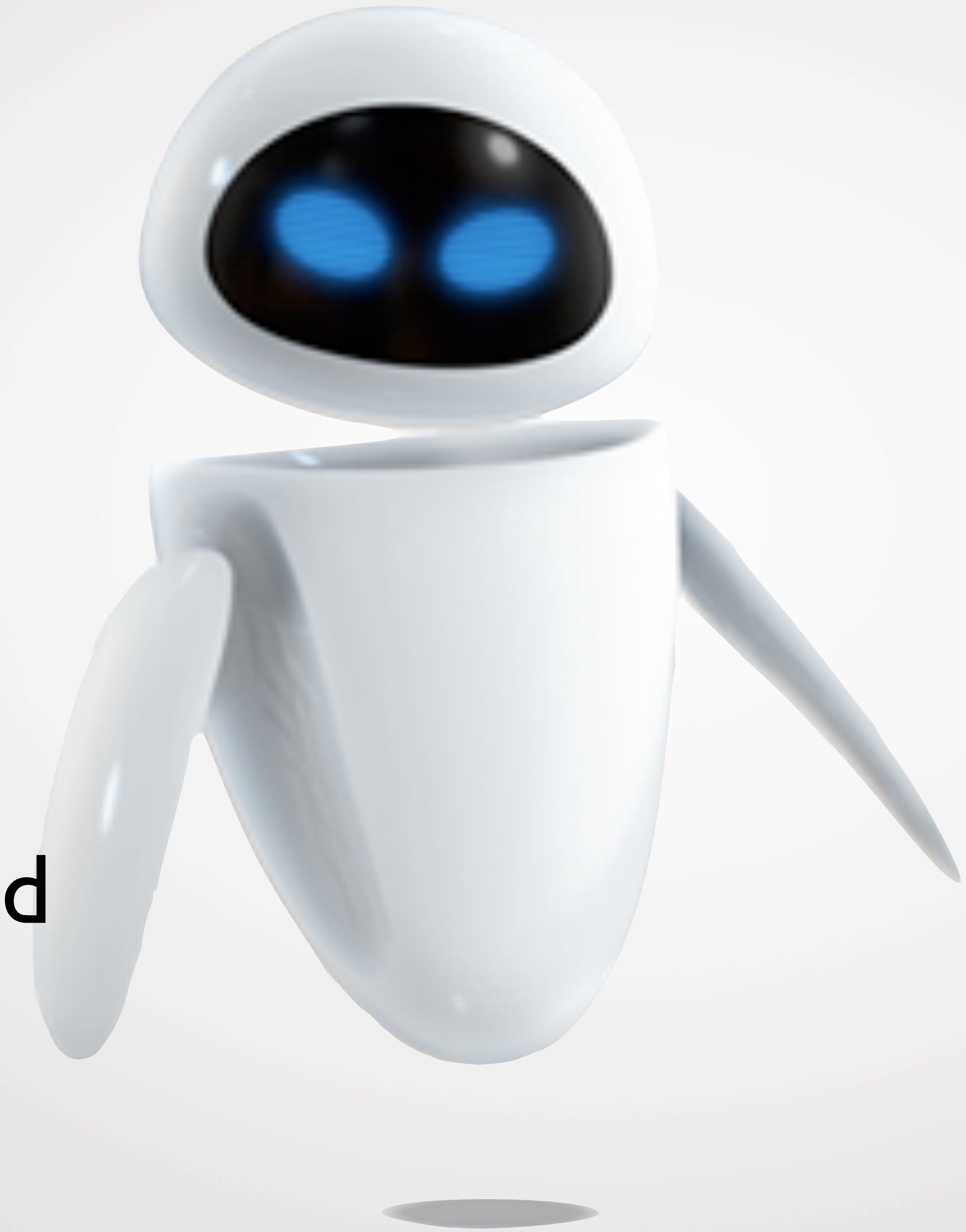
CPU dream

Memory access has zero latency

Flat address space

Branches are cheap

Processors are never code-starved



CPU reality

CPU reality

- Memory latency is huge
  Various levels / geometries of cache memory to improve access to memory
- Translating memory addresses from virtual to physical memory does not come for free
  TLBs to simplify / speedup address translation
- Branches are not cheap
  Branch Prediction Units try to guess program flow in advance to mitigate the cost of branches
- Code is not immune from latency problems, if one is not careful
  No, your CPUs did not attend the first semester OO design lectures you went to...
- ...and now they are complicating things even more with multi-core...
  See Vincenzo's talk (http://indico.cern.ch/contributionDisplay.py?contribId=520&confId=35523)

Modern CPU specs

2/4 cores

L1 cache

from 16KB to 64KB

L2 cache

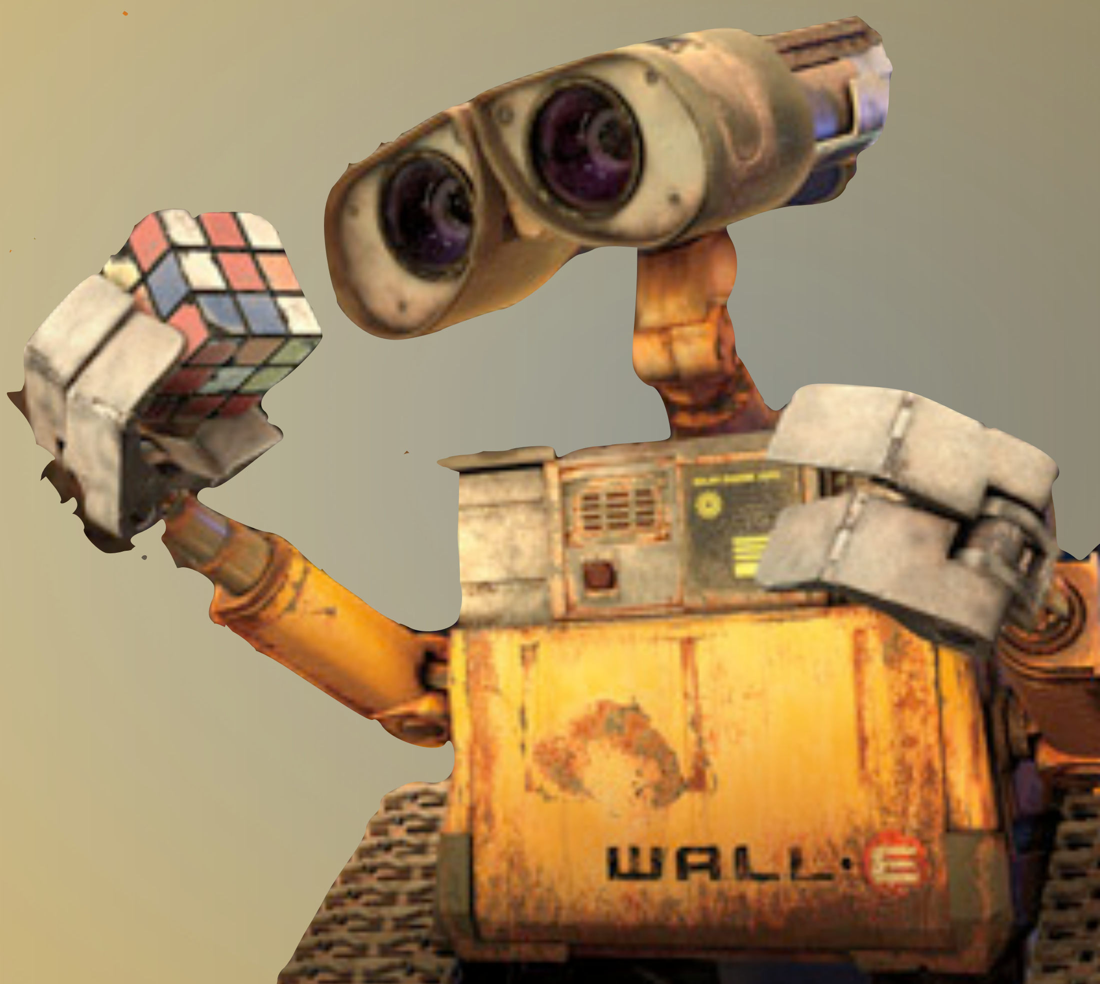
from 512KB to 8MB

TLBs

from 8 to 512 entries for 4KB pages

Branch Prediction Unit

1GB / 2GB memory per core



HEP C++ (e.g. CMSSW)

150MB of size (w/o externals)

50MB of actual code

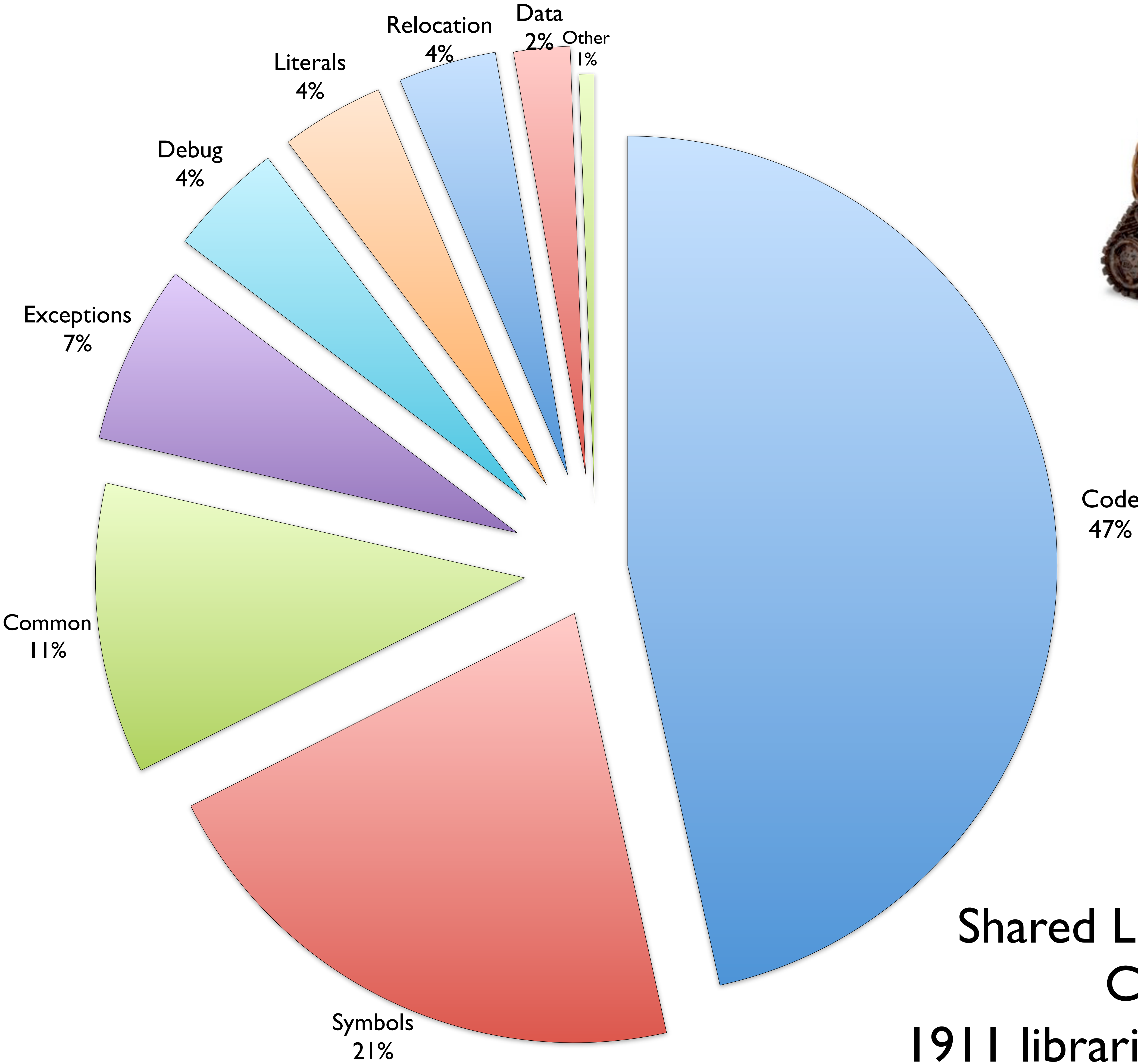
O(500) libraries loaded

O(50k) symbols

Very deep call stacks.

lots of inter-library calls

O(1GB) VSIZE



Shared Library Sections
CMSSW 3.1.0p4
1911 libraries – ∑ 511 MB

CPU vs. HEP

2/4 cores

L1 cache

from 16KB to 64KB

L2 cache

from 512KB to 8MB

TLBs

from 8 to 512 entries for 4KB pages

Branch Prediction Unit

1GB / 2GB memory per core

150MB of size (w/o externals)

50MB of actual code

O(500) libraries loaded

O(50k) symbols

Very deep call stacks.

lots of inter-library calls

O(1GB) VSIZE

First naive observation

There is a lot of code out there

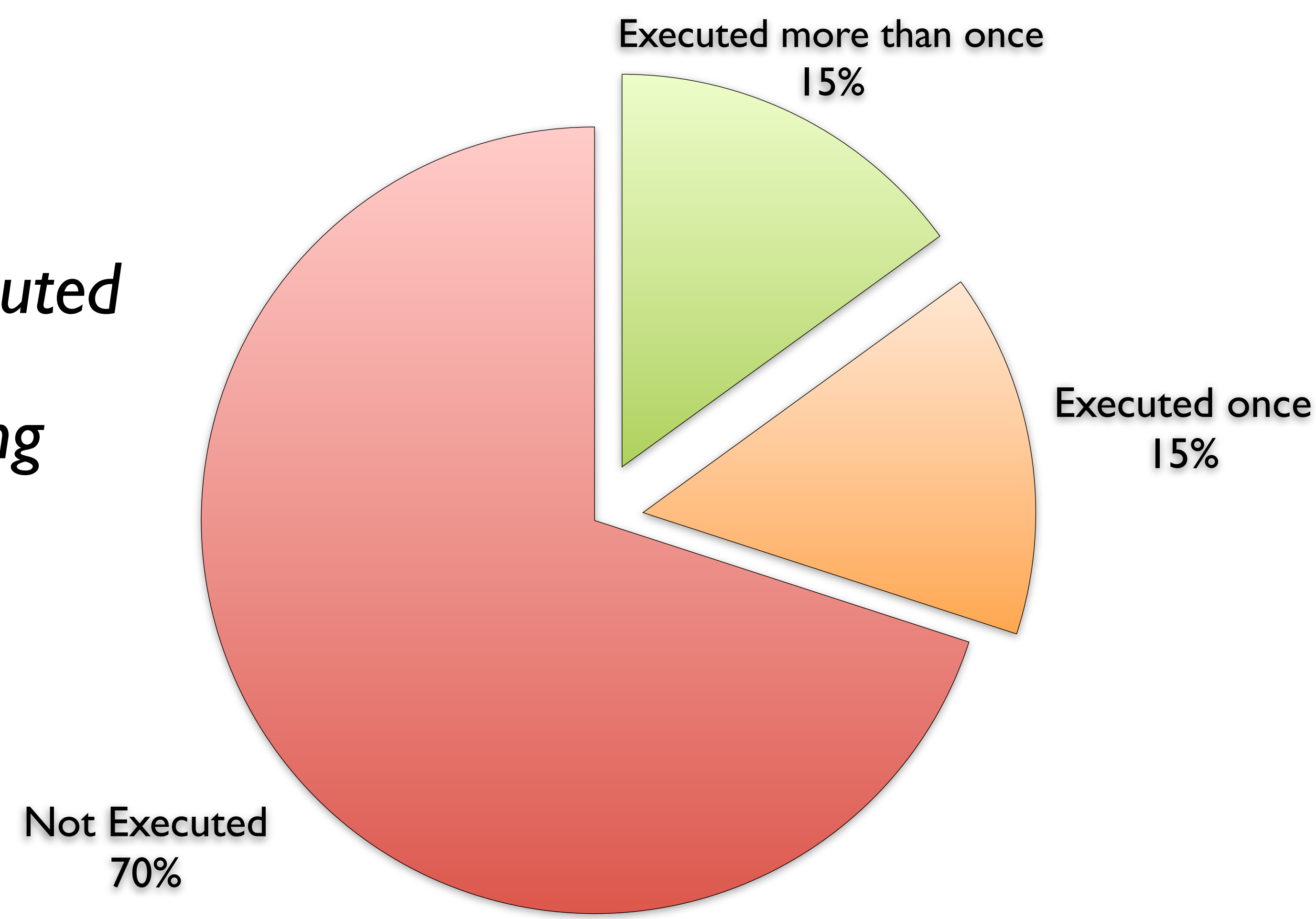
Do we really need 150MB of code?

Source coverage of standalone* CMSSW executable:

Only 30% of source code is actually executed

15% of it is dictionaries constructors being executed only once

What about the remaining 70%?



*does not include externals, only source code for the tested workflow included

https://eulisse.web.cern.ch/eulisse/blog/performance/2008/10/building-a-single-executable-for-cmsrun.html

Reasons

Naive programming

Over-generic designs

C++ idiosyncrasies and abuses

Exceptions, debug code and boundary conditions

Mea culpa!

Very simple example

```
void parseSomeString (const std::string &text)
{
...
}
```

Perfectly valid, correct and clean C++...

Too bad I was passing it a `const char *` *90% of the times...*

→ *the compiler created an implicit temporary* `std::string`*, inline, for each call.*

C++ produces code

C++ is not an abstract language to model a problem. C++ produces actual code which runs on real hardware.

All the animals are equal, but some animals are more equal than others

Not all logically equivalent implementations give the same results performance wise.

Understand what the compiler does

Understanding how C++ source code translates into machine code is crucial if you are interested in having the compiler generate performant / compact code.

Code bloat: simple stuff

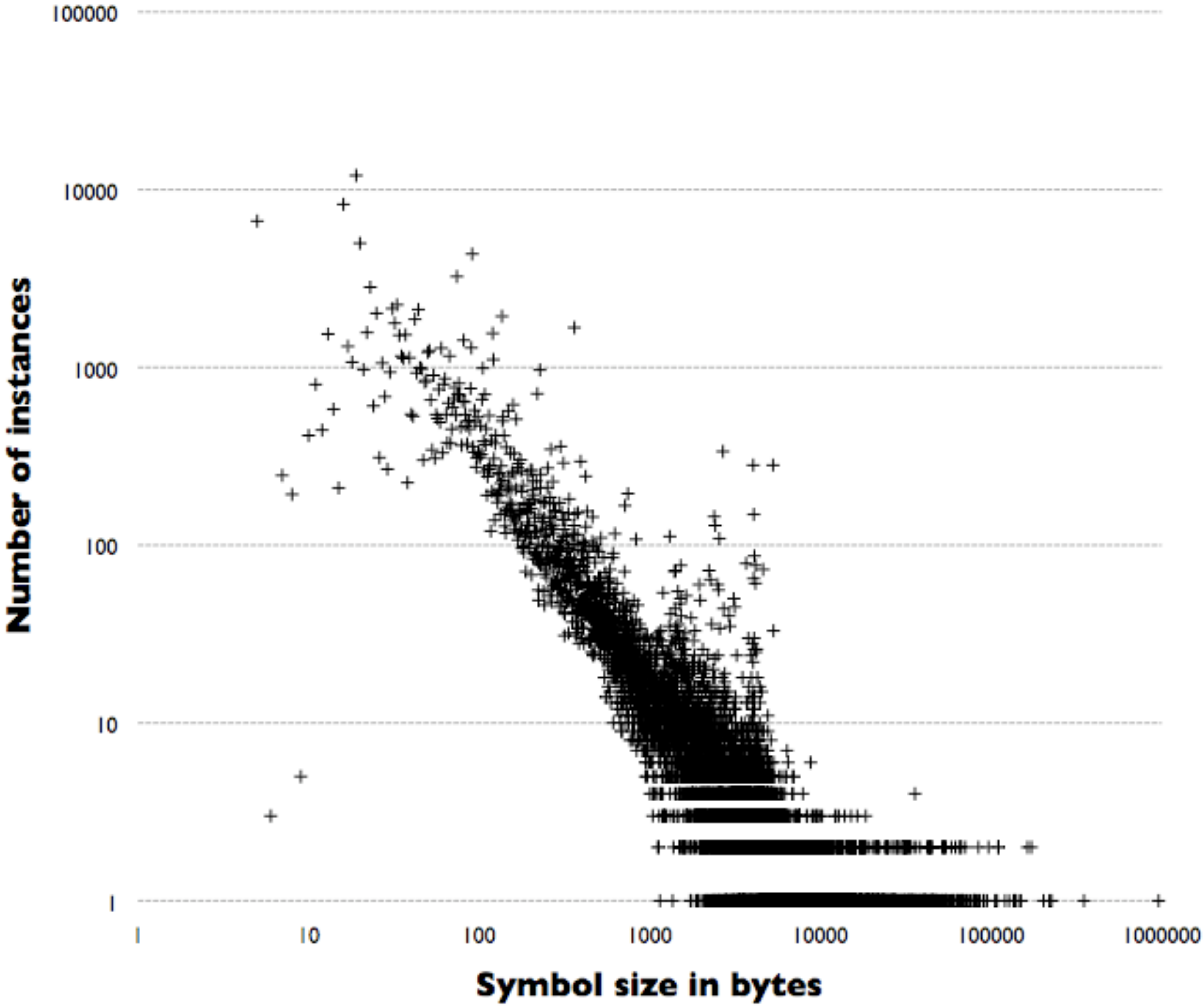
ROOT/REFLEX dictionaries

- Interpreter dictionaries
  - *consider using* `--dataonly` *flag while generating dictionaries*
- Naive mistakes
  - *compiling dummy objects*
  - *long symbols names when compiling files in long paths*

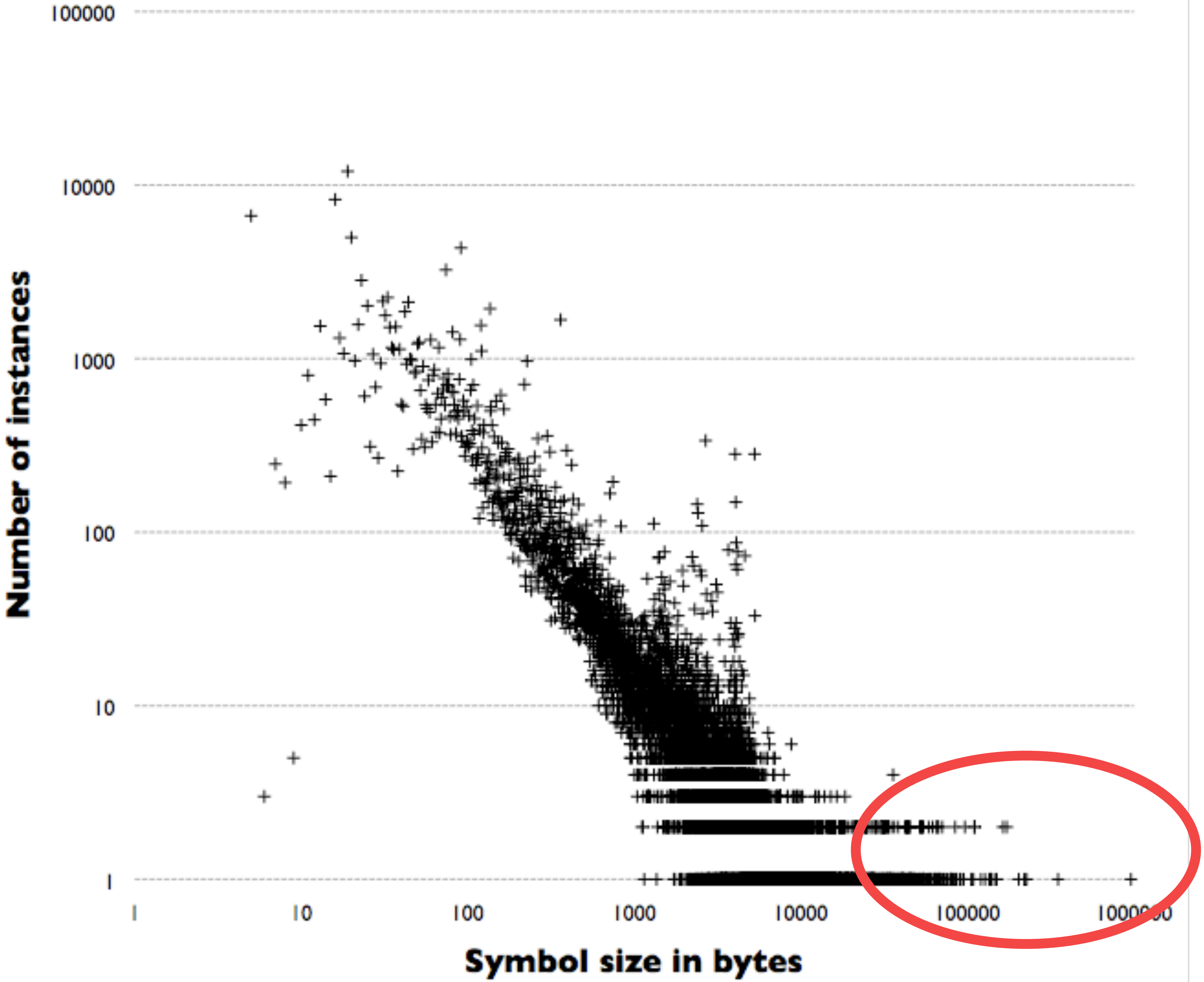
Naive programming

- `statics` are not cheap
  - *Public symbols*
  - *Initialization code in header: e.g.* `#include <iostream>` *in header files*
  - *Guard variables and associated code*
- Inline `std::string` creation from `char *`
- Objects passed by value

Normal function size vs. instances



Normal function size vs. instances



Implicit destructors bloat the code

```
int someVeryLongMethod ()
{
   Klass object;
   Klass object2;
   ...
   if (someCondition)
   {
      object.doSomething();
      throw Exception();
   }
   ...
   if (someCondition)
   {
      object2.doSomethingElse();
      throw Exception();
   }
}
```

This method was 120KB for no apparent reason

https://eulisse.web.cern.ch/eulisse/blog/performance/2008/11/cost-of-implicit-destructor-and-inlined-code.html

Implicit destructors bloat the code

```
int someVeryLongMethod ()
{
   Klass object;
   Klass object2;
   ...
   if (someCondition)
   {
      object.doSomething();
      throw Exception();
   }
   ...
   if (someCondition)
   {
      object2.doSomethingElse();
      throw Exception();
   }
}
```

`Klass` had a implicit destructor.

`Klass` member variables had expensive destructors (`vectors`, `maps`, `strings`).

Compilers tend to inline implicit destructors.

https://eulisse.web.cern.ch/eulisse/blog/performance/2008/11/cost-of-implicit-destructor-and-inlined-code.html

Implicit destructors bloat the code

```
int someVeryLongMethod ()
{
   Klass object;
   Klass object2;
   ...
   if (someCondition)
   {
      object.doSomething();
      throw Exception();
   }
   ...
   if (someCondition)
   {
      object2.doSomethingElse();
      throw Exception();
   }
}
```

The compiler also needs to destroy all the objects that go out of scope....

...for every exit path...

... and compilers are not that good at understanding that two exit paths are the same...

https://eulisse.web.cern.ch/eulisse/blog/performance/2008/11/cost-of-implicit-destructor-and-inlined-code.html

Implicit destructors bloat the code

```
int someVeryLongMethod ()
{
   Klass object;
   Klass object2;
   ...
   if (someCondition)
   {
      object.doSomething();
      throw Exception();
   }
   ...
   if (someCondition)
   {
      object2.doSomethingElse();
      throw Exception();
   }
}
```

Destructors inlined everywhere!

https://eulisse.web.cern.ch/eulisse/blog/performance/2008/11/cost-of-implicit-destructor-and-inlined-code.html

Implicit destructors bloat the code

```
// In the .h

class Klass
{
  ..
 ~Klass(void);
  ..
};


// In the .cc

Klass::Klass(void) {}
```

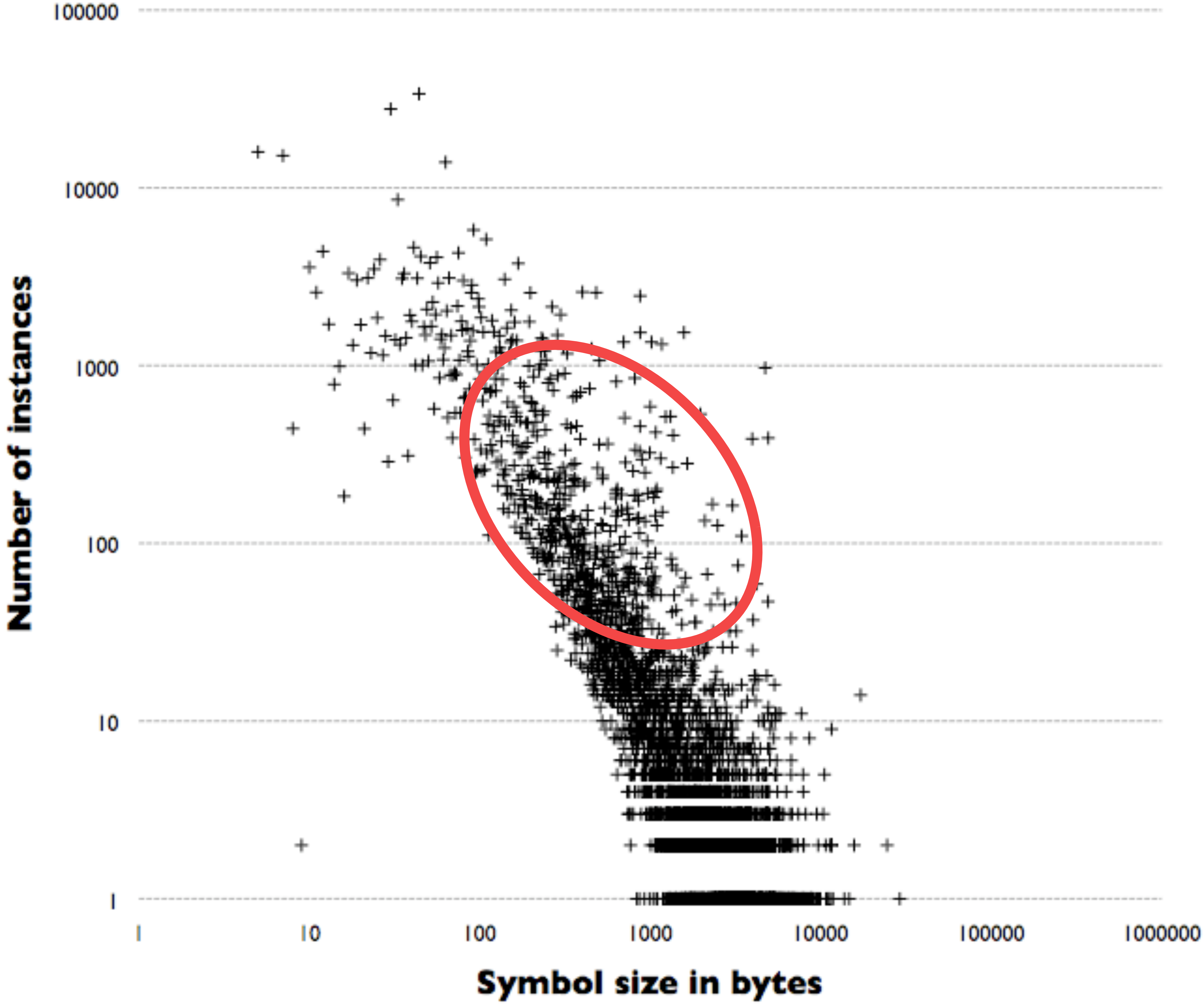
Adding an explicitly out-of-line destructor saved 100KB (from one single method!!!)

https://eulisse.web.cern.ch/eulisse/blog/performance/2008/11/cost-of-implicit-destructor-and-inlined-code.html

Giulio's 1st Observation on Optimization

Code bloat correlates very well with bad coding practices

Inlined function size vs. instances



Bloat from `templates`

Tricky to spot

Small cost for a given template class method might become large when you integrate over the use template parameters.

`template` invariant code

The compiler will not factor out template invariant code from a template class , each template instance will get a copy of the same exact code.

Particularly relevant for `templates` over event product type

CMS has O(400) different physics object classes

Symbols proliferation

```
template <class T>
class SomeKlass
{
  ...
  void methodWhichDoesNotDependOnT (void){}
};
...

SomeKlass<Product1> p1;
SomeKlass<Product2> p2;
SomeKlass<Product3> p3;

p1.someMethodWhichDoesNotDependOnT();
p2.someMethodWhichDoesNotDependOnT();
p3.someMethodWhichDoesNotDependOnT();
```

Compiler will produce (unnecessarily) separate code for each `ProductN`.

```
template <class T>
class SomeKlass
{
  ...
  void methodWhichDoesNotDependOnT (void){}
};
...

SomeKlass<Product1> p1;
SomeKlass<Product2> p2;
SomeKlass<Product3> p3;

p1.someMethodWhichDoesNotDependOnT();
p2.someMethodWhichDoesNotDependOnT();
p3.someMethodWhichDoesNotDependOnT();
```

...and will do so for each library where `SomeKlass<T>` is used...

```
class SomeKlassBase
{
   ...
   void methodWhichDoesNotDependOnT (void){}
};

...

template <class T>
class SomeKlass :SomeKlassBase
{
   ...
};
```

Introducing a non-template base class might be a good solution.

https://eulisse.web.cern.ch/eulisse/blog/performance/2008/10/cost-of-over-templated-code.html

Giulio's 2nd Observation on Optimization

Many small (related) symbols correlate very well with bad coding practices

perfmon2

Very high resolution profiler for Linux

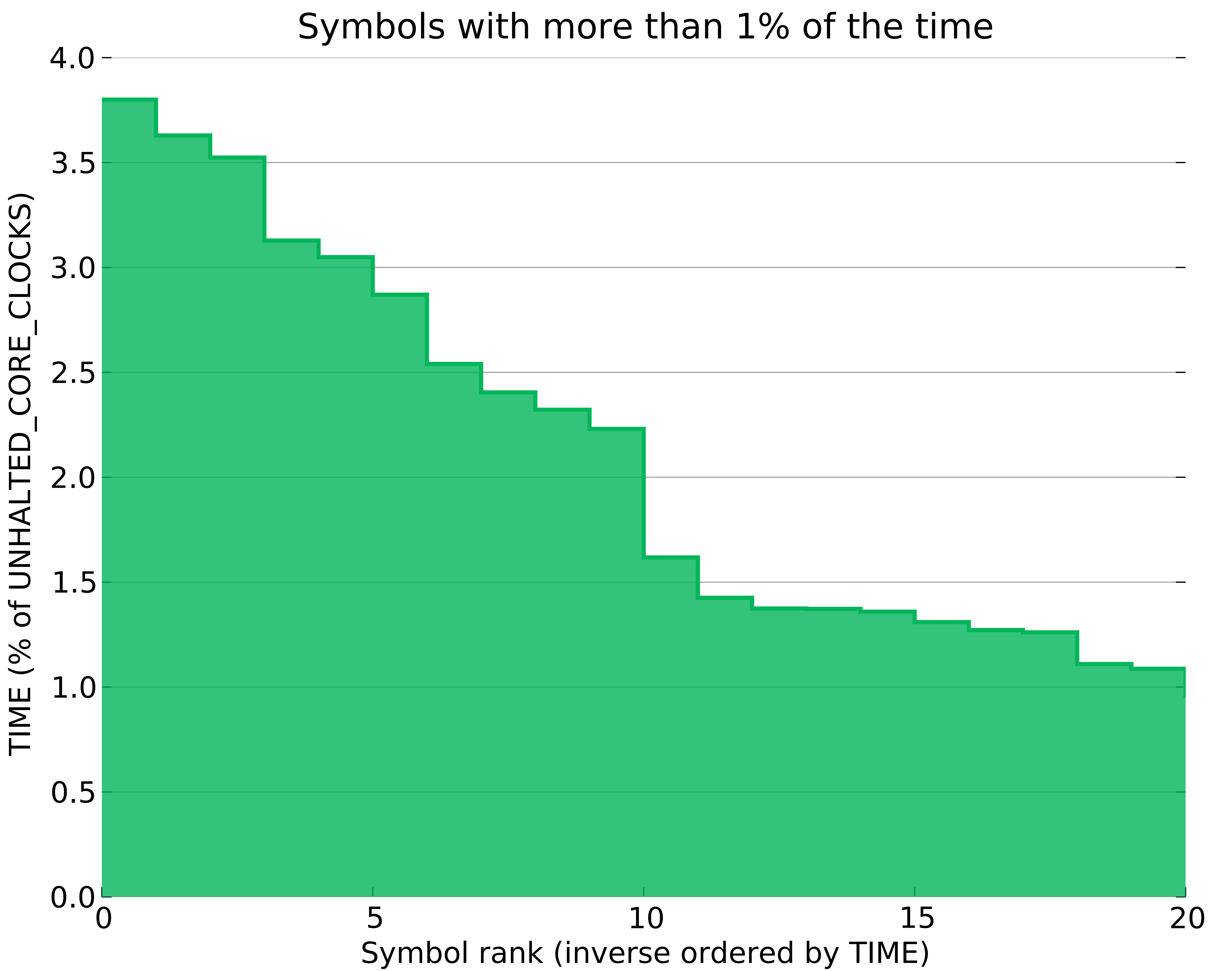
- *uses processor performance counters*

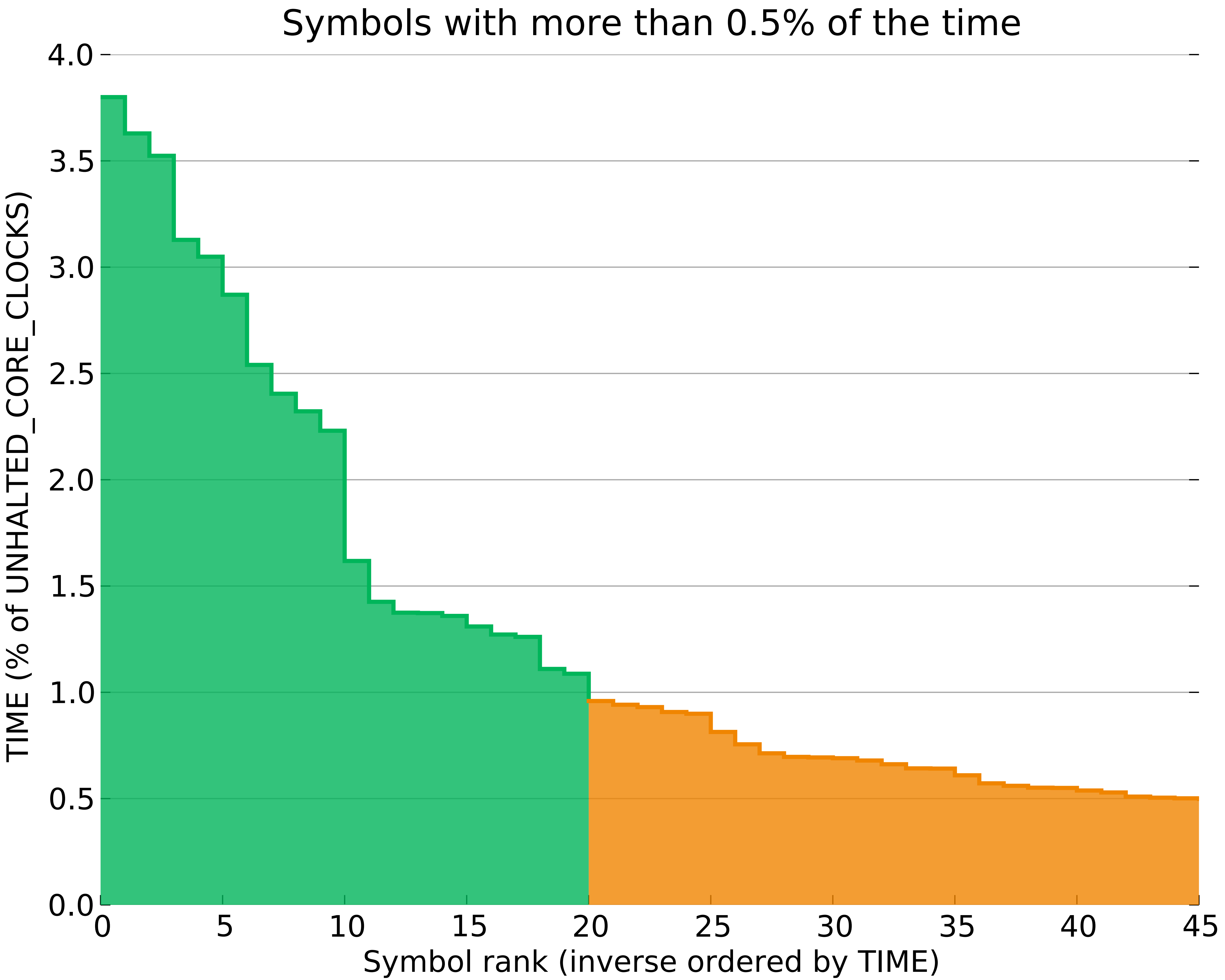
Monitors every aspect of a CPU

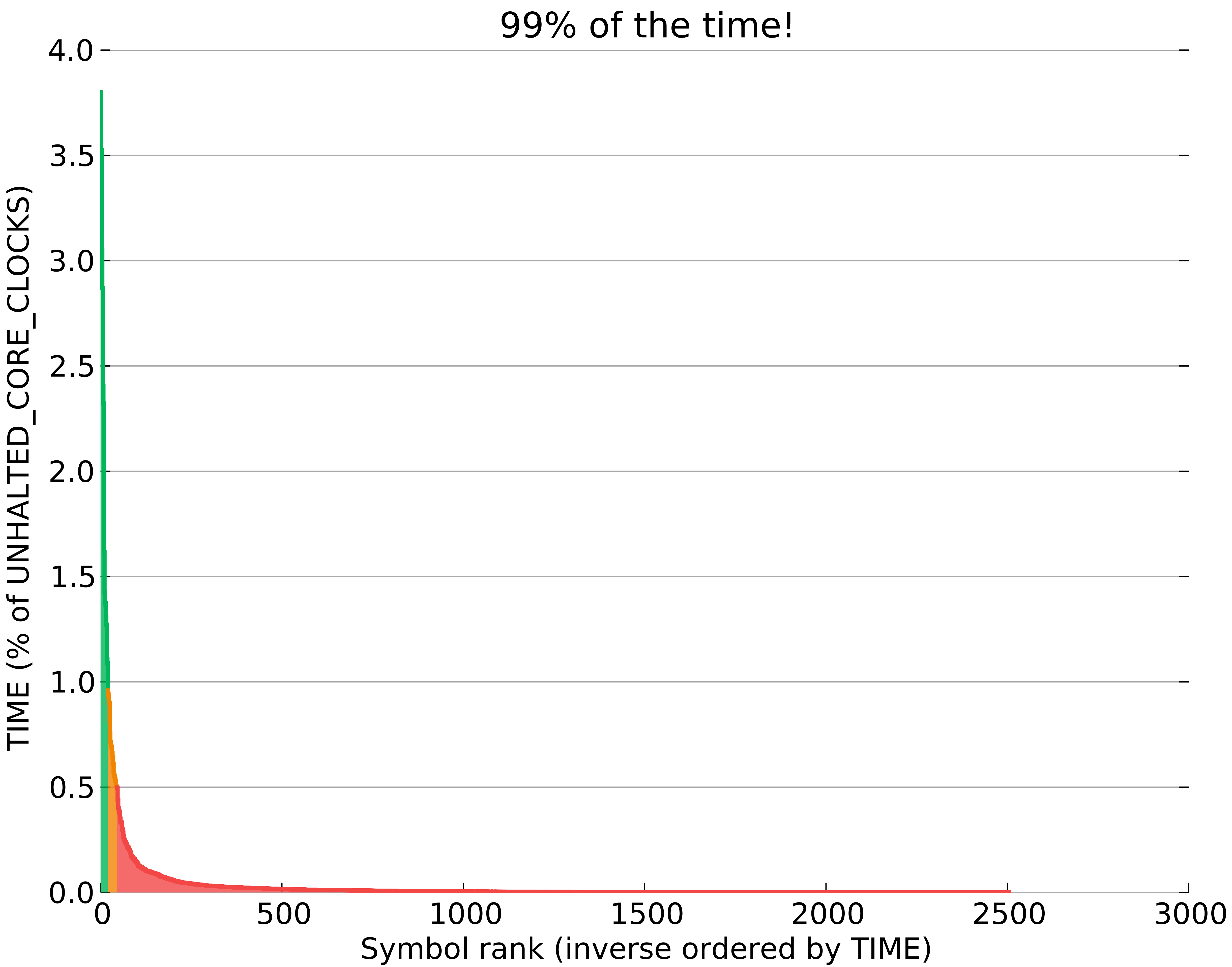
- *Retired instructions*
- *Mispredicted branches*
- *Cache misses*
- *etc.*

See talk by Andrzej Nowak

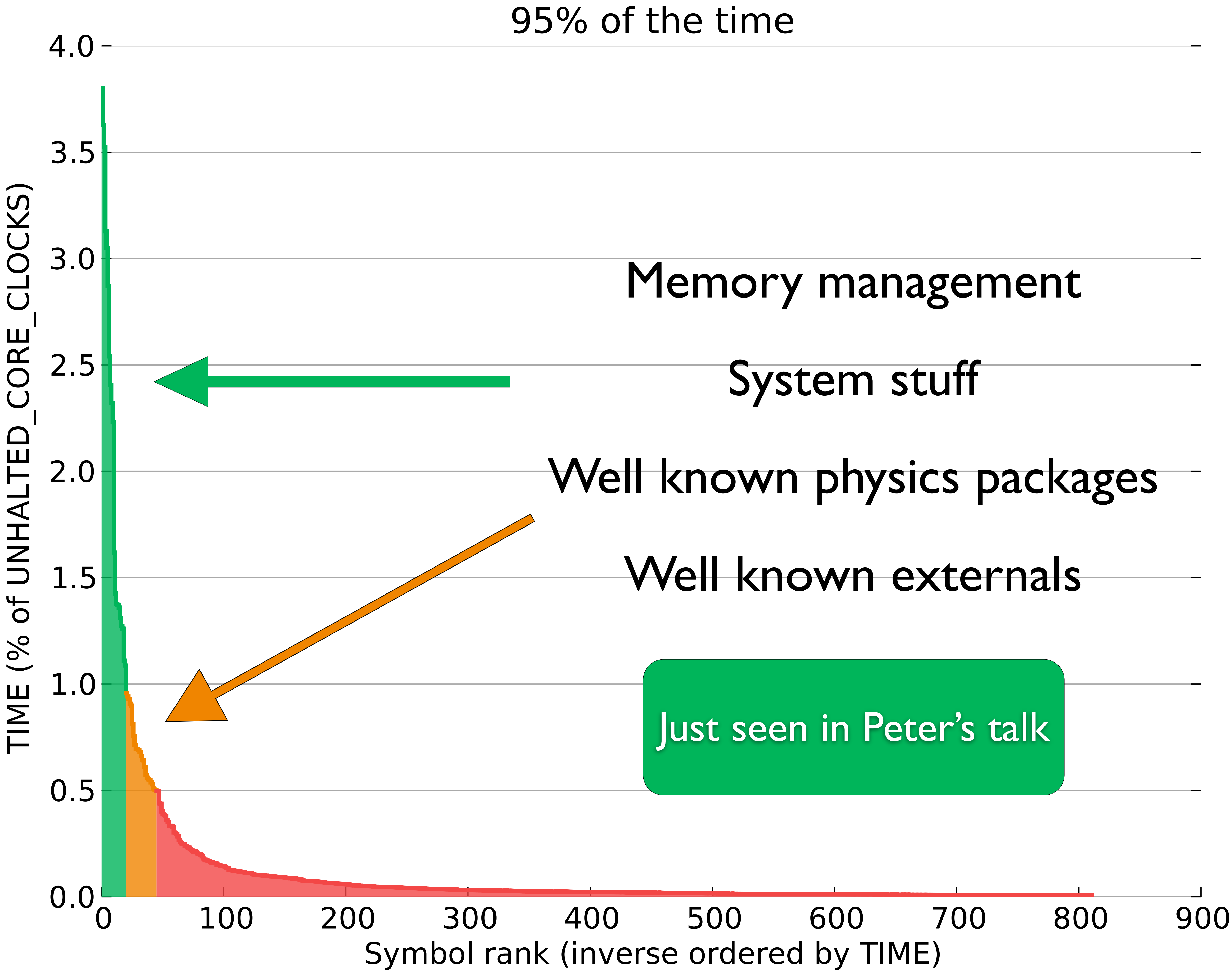
- *http://indico.cern.ch/contributionDisplay.py?contribId=436&confId=35523*

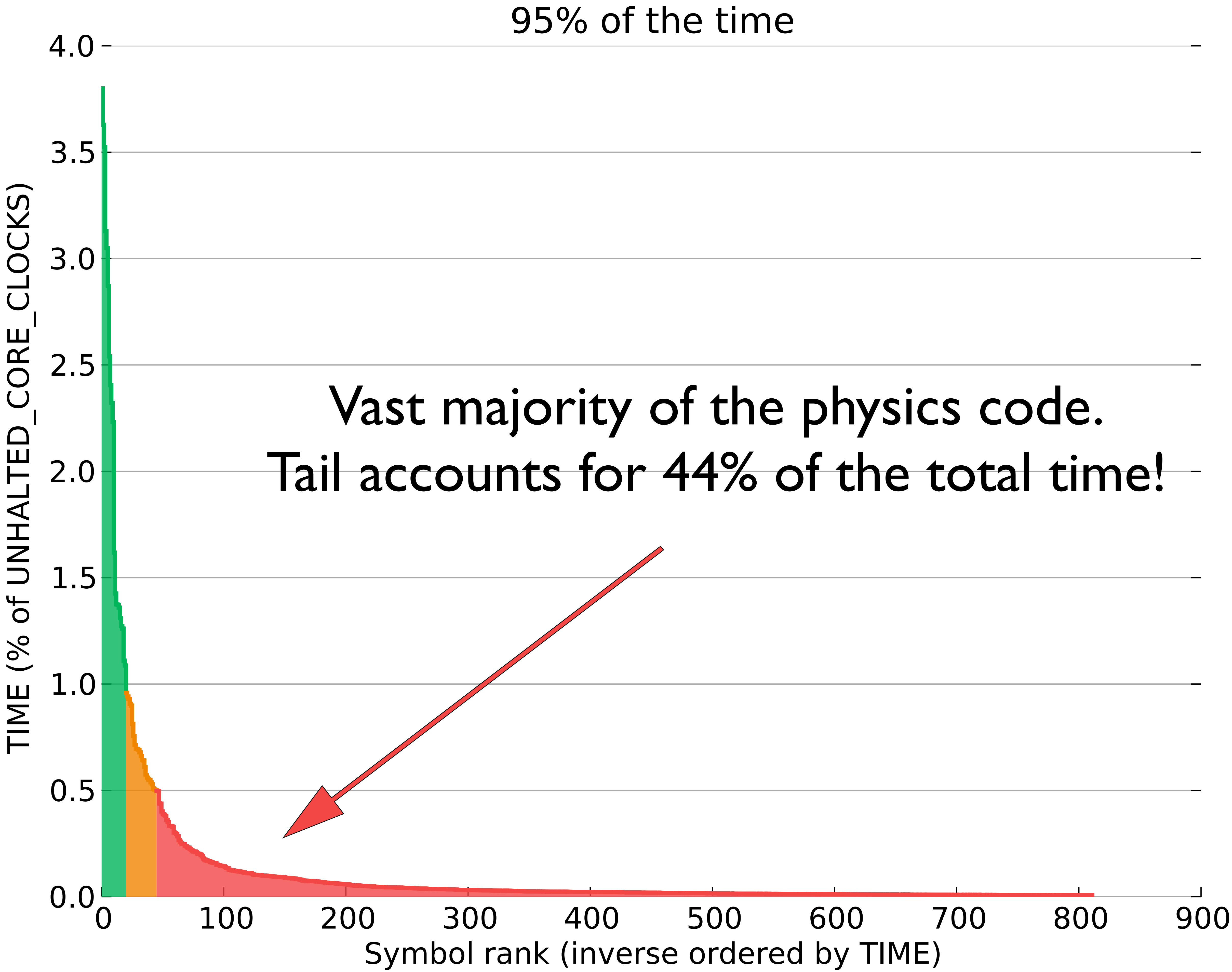

Symbols with more than 1% of the time
TIME (% of UNHALTED_CORE_CLOCKS)
Symbol rank (inverse ordered by TIME)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
5
10
15
20


Symbols with more than 0.5% of the time
TIME (% of UNHALTED_CORE_CLOCKS)
Symbol rank (inverse ordered by TIME)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
5
10
15
20
25
30
35
40
45

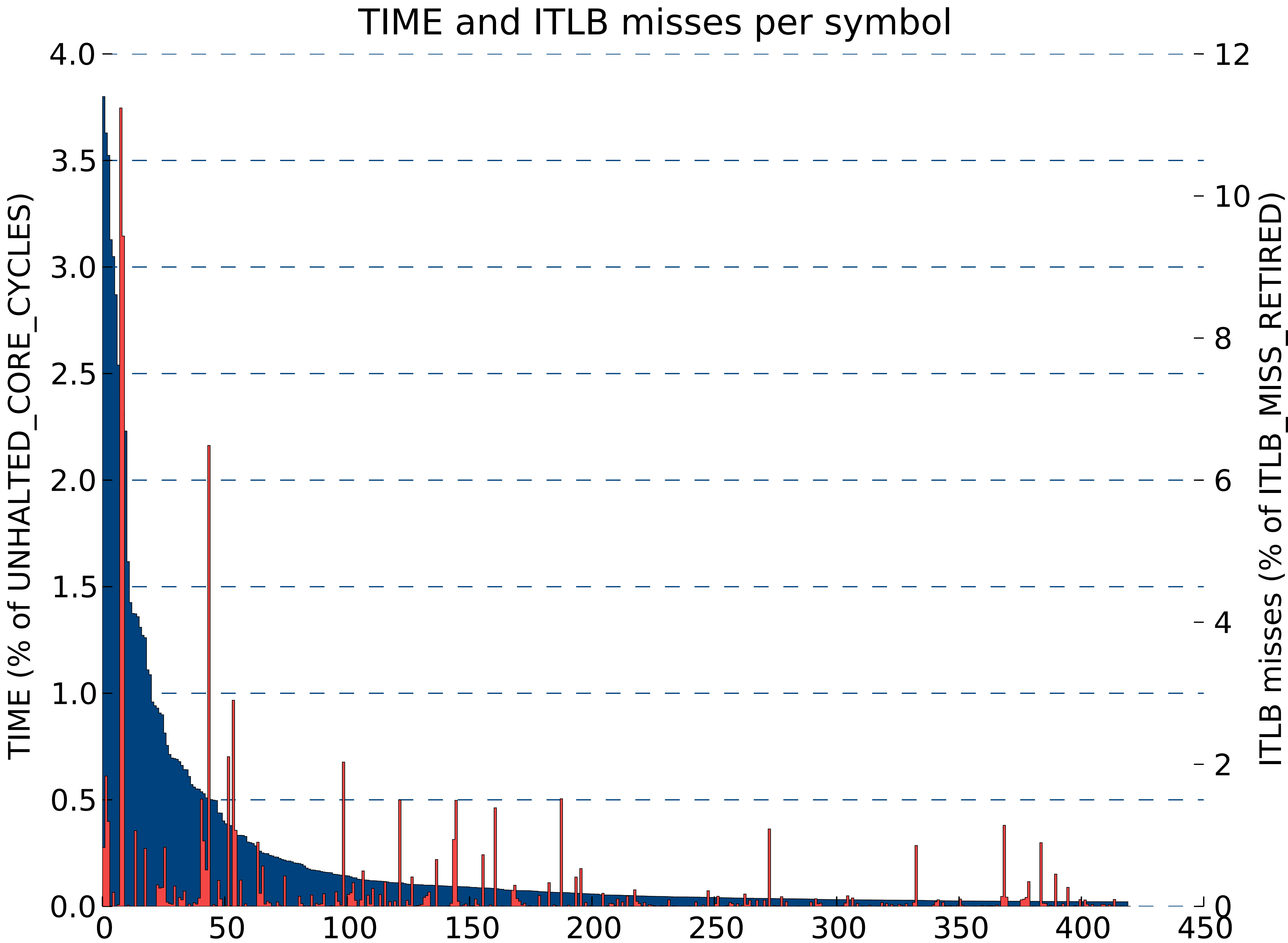


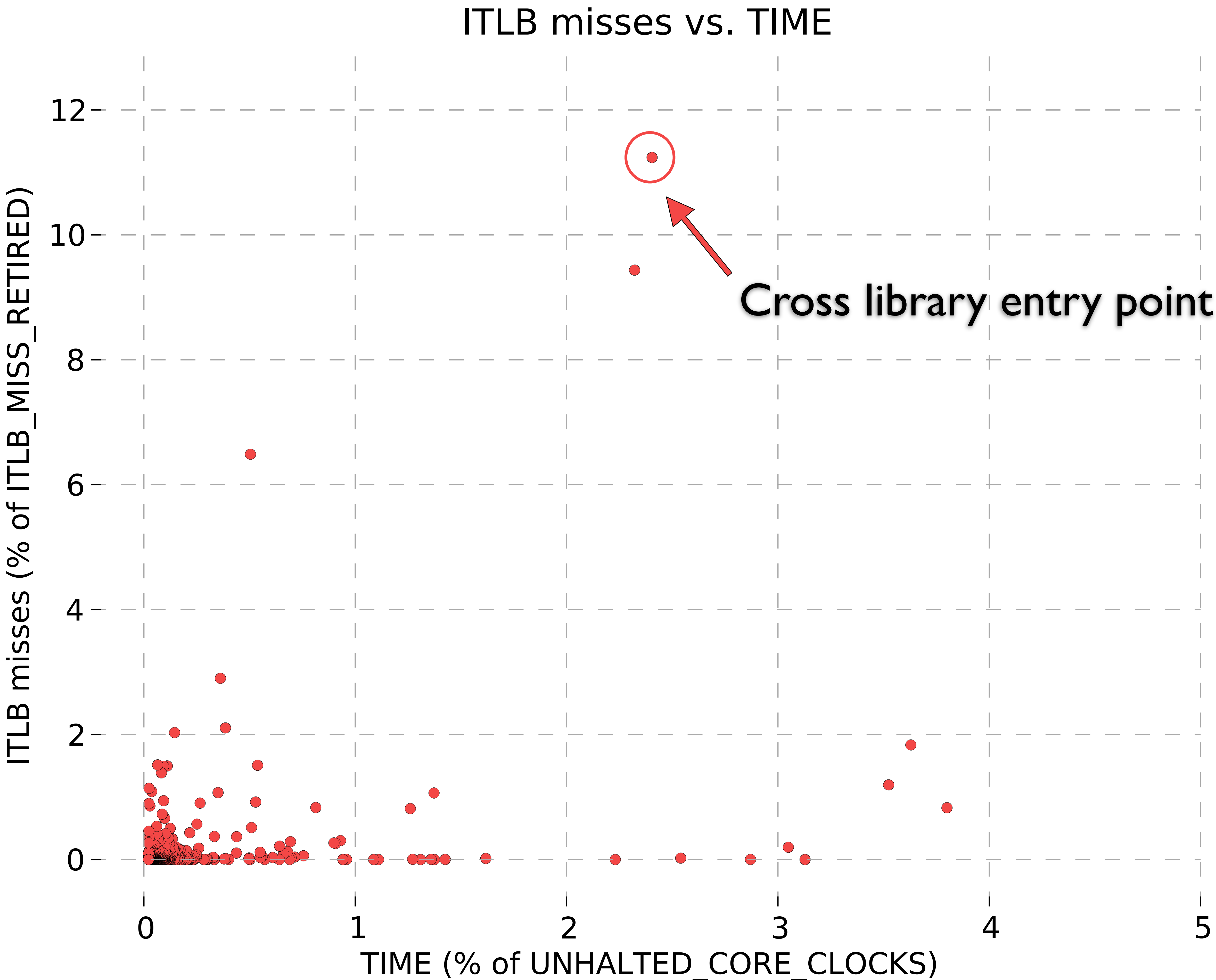
99% of the time!
TIME (% of UNHALTED_CORE_CLOCKS)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
500
1000
1500
2000
2500
3000
Symbol rank (inverse ordered by TIME)

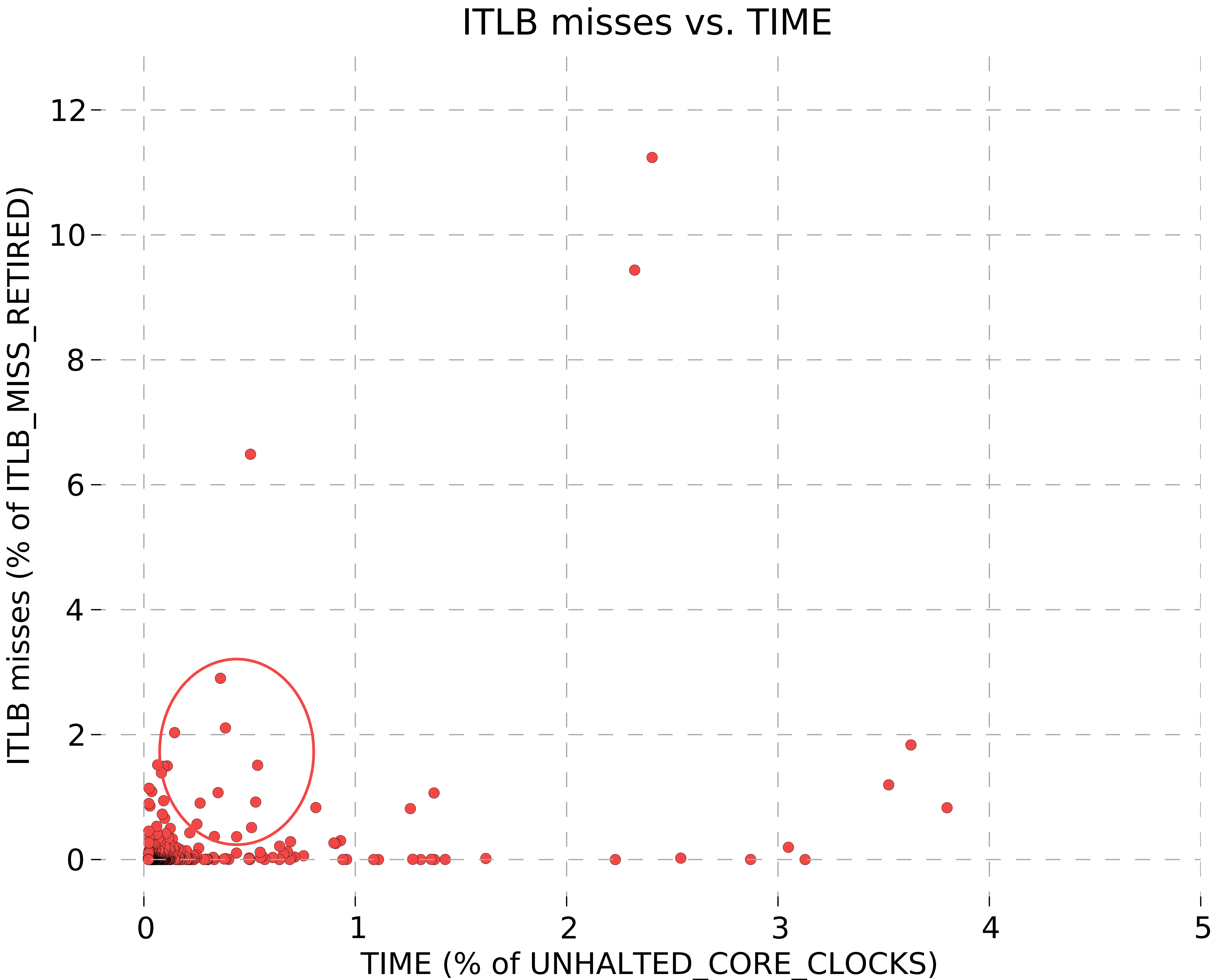

95% of the time
TIME (% of UNHALTED_CORE_CLOCKS)
Symbol rank (inverse ordered by TIME)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
100
200
300
400
500
600
700
800
900
Memory management
System stuff
Well known physics packages
Well known externals
Just seen in Peter's talk


95% of the time
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
TIME (% of UNHALTED_CORE_CLOCKS)
0
100
200
300
400
500
600
700
800
900
Symbol rank (inverse ordered by TIME)
Vast majority of the physics code.
Tail accounts for 44% of the total time!

We need to correlate results!




ITLB misses vs. TIME
Cross library entry point
ITLB misses (% of ITLB_MISS_RETIRED)
TIME (% of UNHALTED_CORE_CLOCKS)
0
2
4
6
8
10
12
0
1
2
3
4
5



ITLB misses vs. TIME
ITLB misses (% of ITLB_MISS_RETIRED)
TIME (% of UNHALTED_CORE_CLOCKS)
0
2
4
6
8
10
12
0
1
2
3
4
5

Function local `statics`.

```
template <class T, unsigned int D>
class A {
public:
  A() :b(0) { Init(); }
..

  void Init() {
    static B<D> flo;
    b = &flo;
  }
 B *b;
};
```

The static is obviously initialized only once...

Function local `statics`.

```
template <class T, unsigned int D>
class A {
public:
  A() :b(0) { Init(); }
..

  void Init() {
    static B<D> flo;
    b = &flo;
  }
  B *b;
};
```

...but some compilers (e.g. gcc 3.4.5) put `Init()` inline into the constructor.

Function local `statics`.

```
template <class T, unsigned int D>
class A {
public:
  A() :b(0) { Init(); }
..

  void Init() {
    static B<D> flo;
    b = &flo;
  }
  B *b;
};
```

..a trivial constructor brings the whole (size) cost of a (relatively) complex, one-time, initialization..

Concrete case: `SMatrix`

`ROOT::Math::SMatrix` uses exactly this coding pattern.

```
template <class T, unsigned int D>
class MatRepSym {
public:
  MatRepSym() :fOff(0) { CreateOffsets(); }
..

  void CreateOffsets() {
    static RowOffsets<D> off;
    fOff = &off;
  }
};
```

Concrete case: `SMatrix`

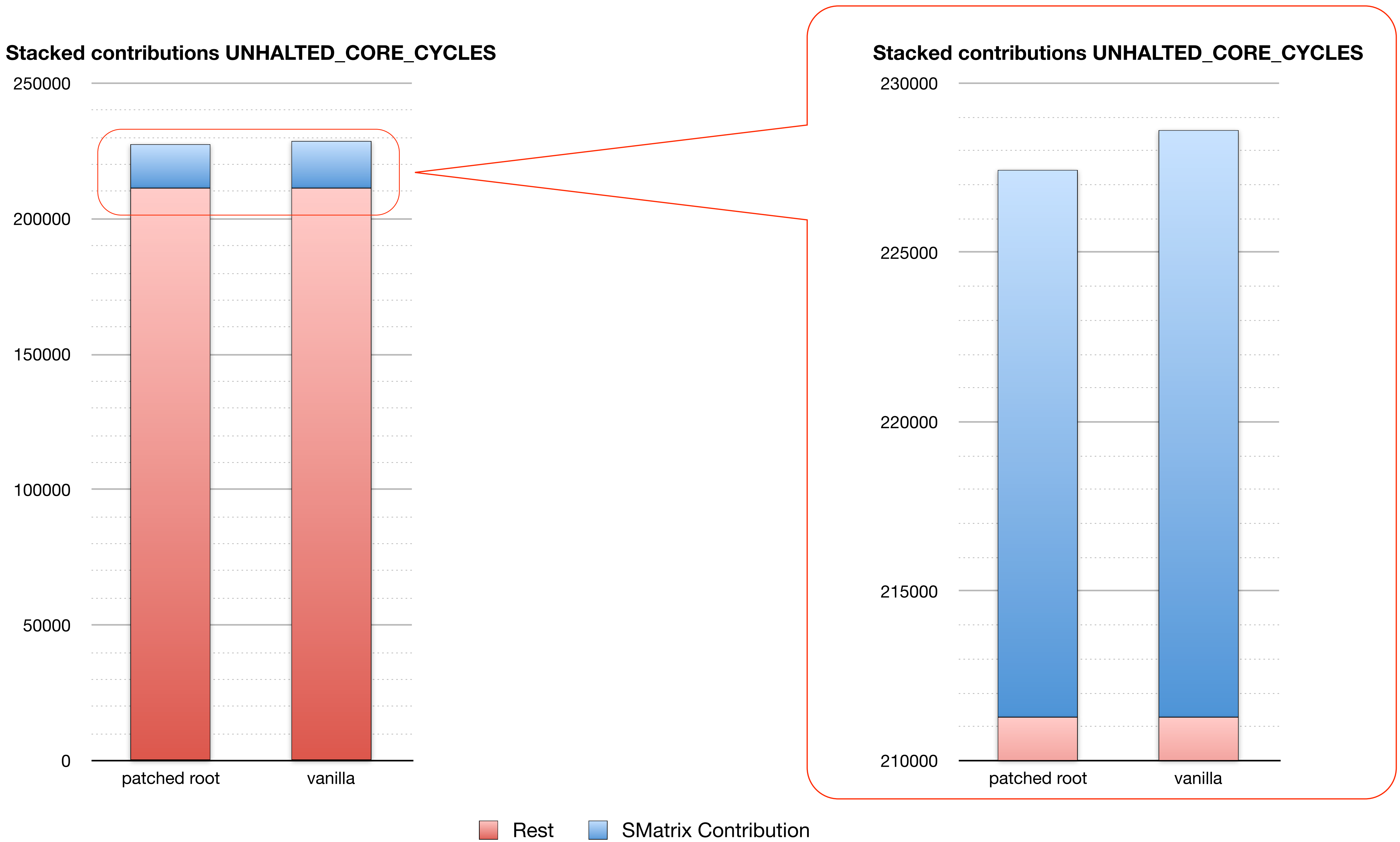
```
template <class T, unsigned int D>
class MatRepSym {
public:
  MatRepSym() :fOff(0) { CreateOffsets(); }
..

  void CreateOffsets() {
    static RowOffsets<D> off;
    fOff = &off;
  }
};
```

Forcing the compiler to put `CreateOffset()` out of line

```
struct RowOffsetsBase
{
protected:
  static void init(int *v, int *offsets, unsigned int D);
};

template<unsigned int D>
struct RowOffsets {
struct RowOffsets : RowOffsetsBase {
  RowOffsets() {
    this->init(v, fOff, D);
  }
..
};

template <unsigned int D> struct SymMatrixOffsets
{
protected:
  static RowOffsets<D> offsets;
};

template <unsigned int D>
RowOffsets<D>
SymMatrixOffsets<D>::offsets;

template <class T, unsigned int D>
class MatRepSym : SymMatrixOffsets<D> {
public:

  MatRepSym() :fOff(&SymMatrixOffsets<D>::offsets) { }
  ..
};
```

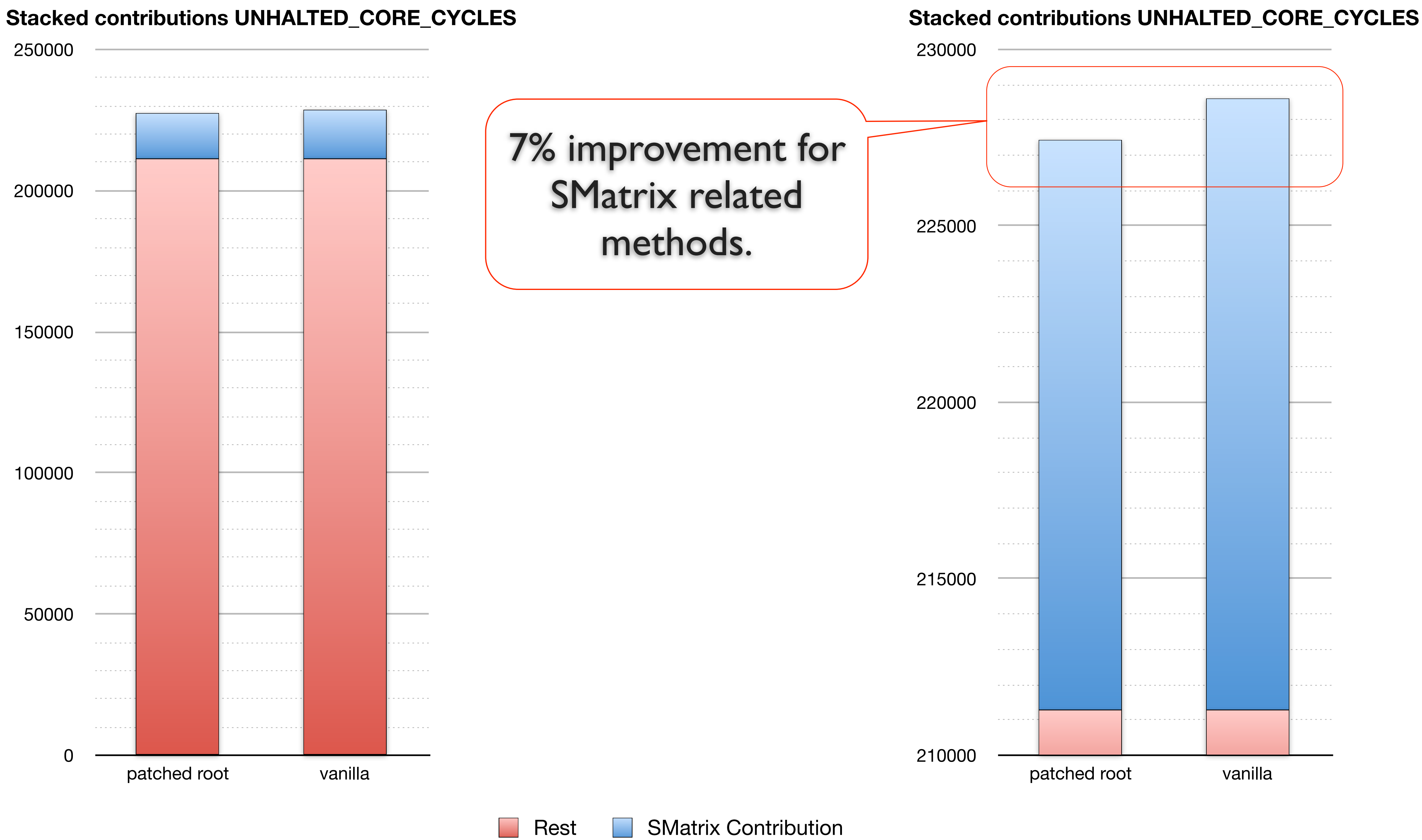
Profiling results

Profiling with pfmon the runtime and ITLB misses for SMatrix related symbols



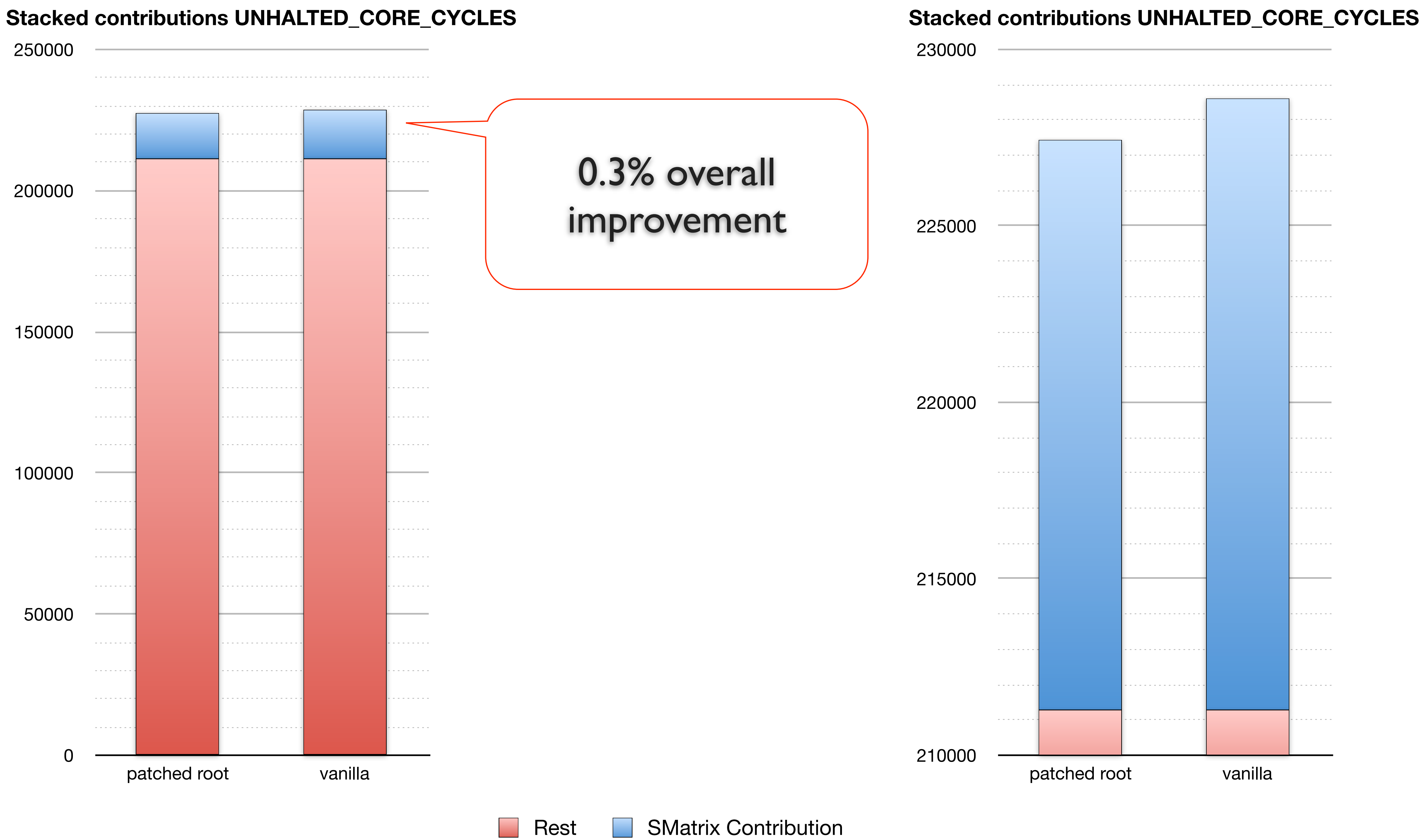
Profiling results

Profiling with pfmon the runtime and ITLB misses for SMatrix related symbols

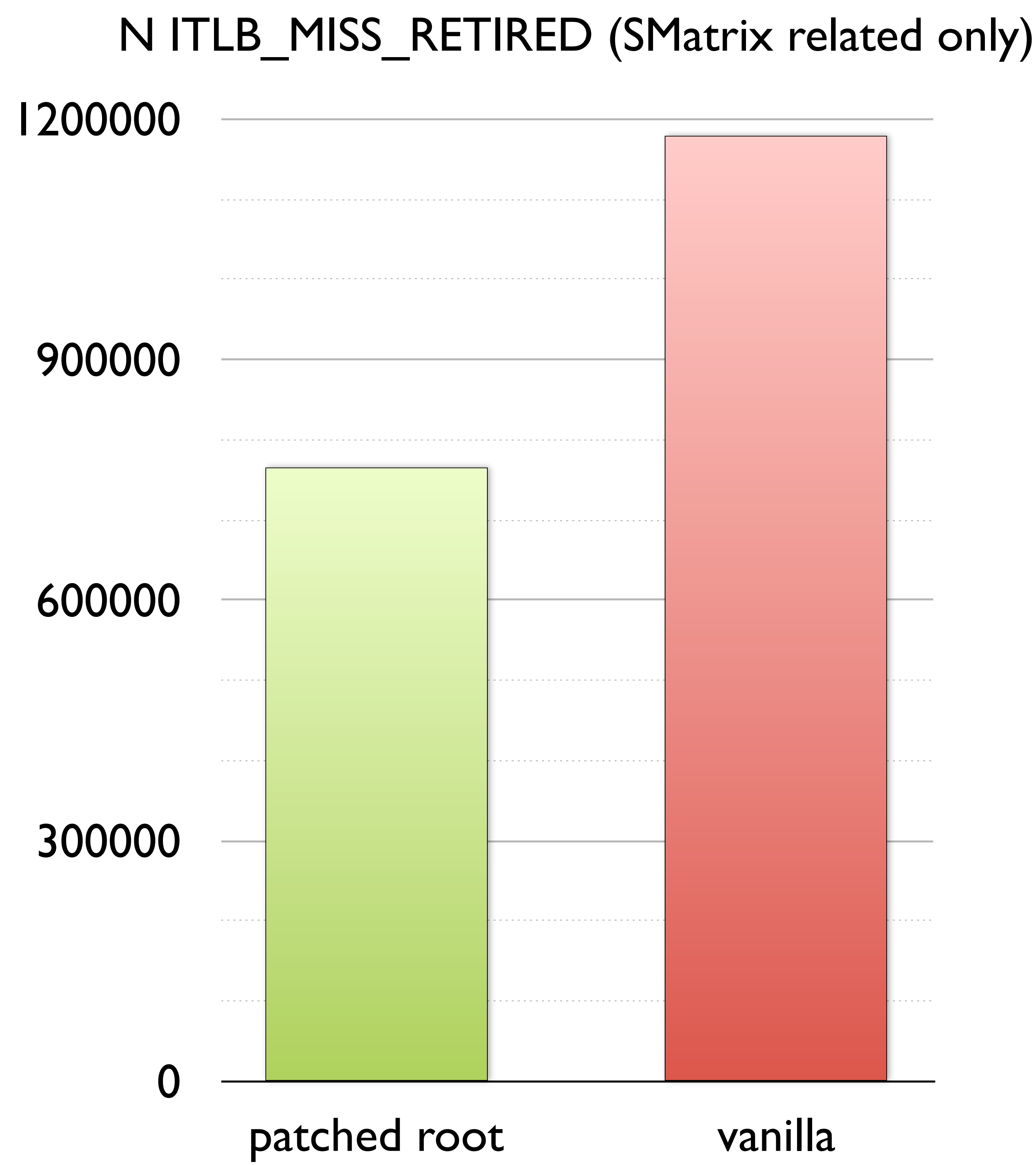


Profiling results

Profiling with pfmon the runtime and ITLB misses for SMatrix related symbols



Profiling results



Giulio's 3rd Observation on Optimization

Code which is never executed still affects performance

Lessons learned

Hardware matters

C++ is not an abstract language

Physical packaging and build procedures matter

dynamic libraries are not just a matter of subdividing code

Non-executed code might have side effects

pollutes caches
pollutes itlb
forces longer jumps

Things never work as you think

always profile your software...
...and try to understand what it actually does!

Performance Taskforce Chronicles:

https://eulisse.web.cern.ch/eulisse/blog/performance

more details about my findings