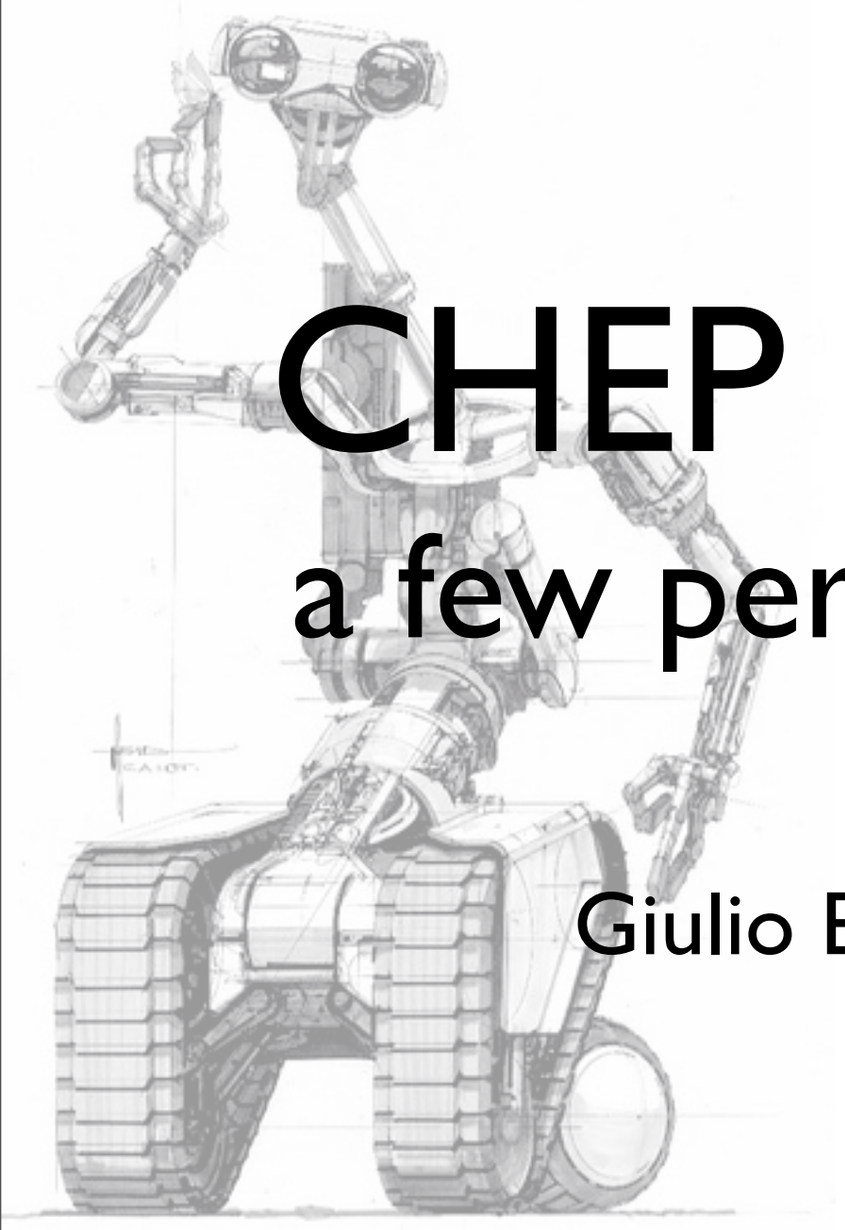


CHEP 09 - RELOADED

a few performance related notes

Giulio Eulisse, Northeastern University

References

Full bells & whistles talk from CHEP09

http://indico.cern.ch/getFile.py/access?contribId=266&sessionId=59&resId=0&materialId=slides&confId=35523

My performace related blog

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

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

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

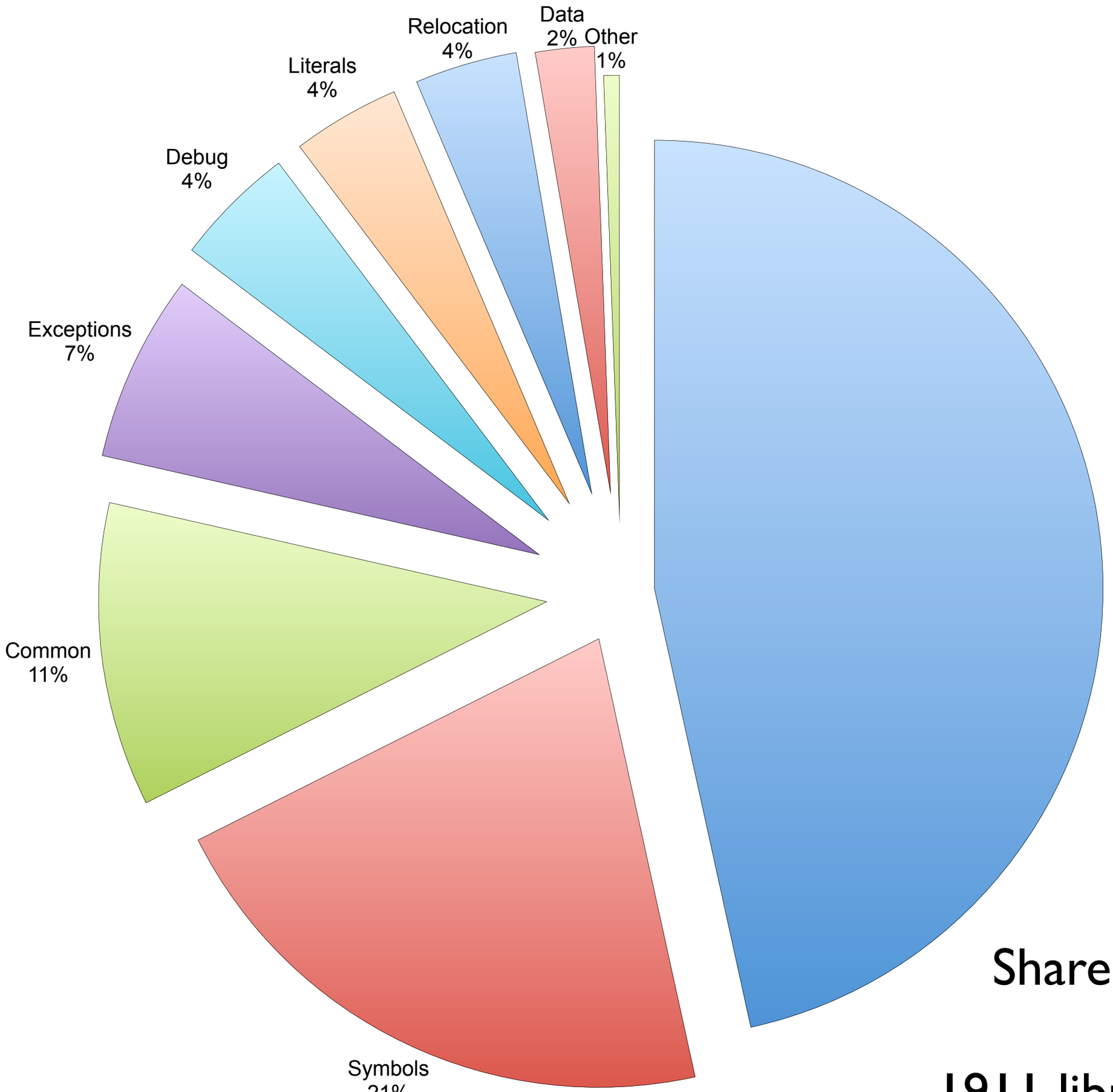
O(500) libraries loaded

O(50k) symbols

Very deep call stacks.

lots of inter-library calls

O(1GB) VSIZE



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

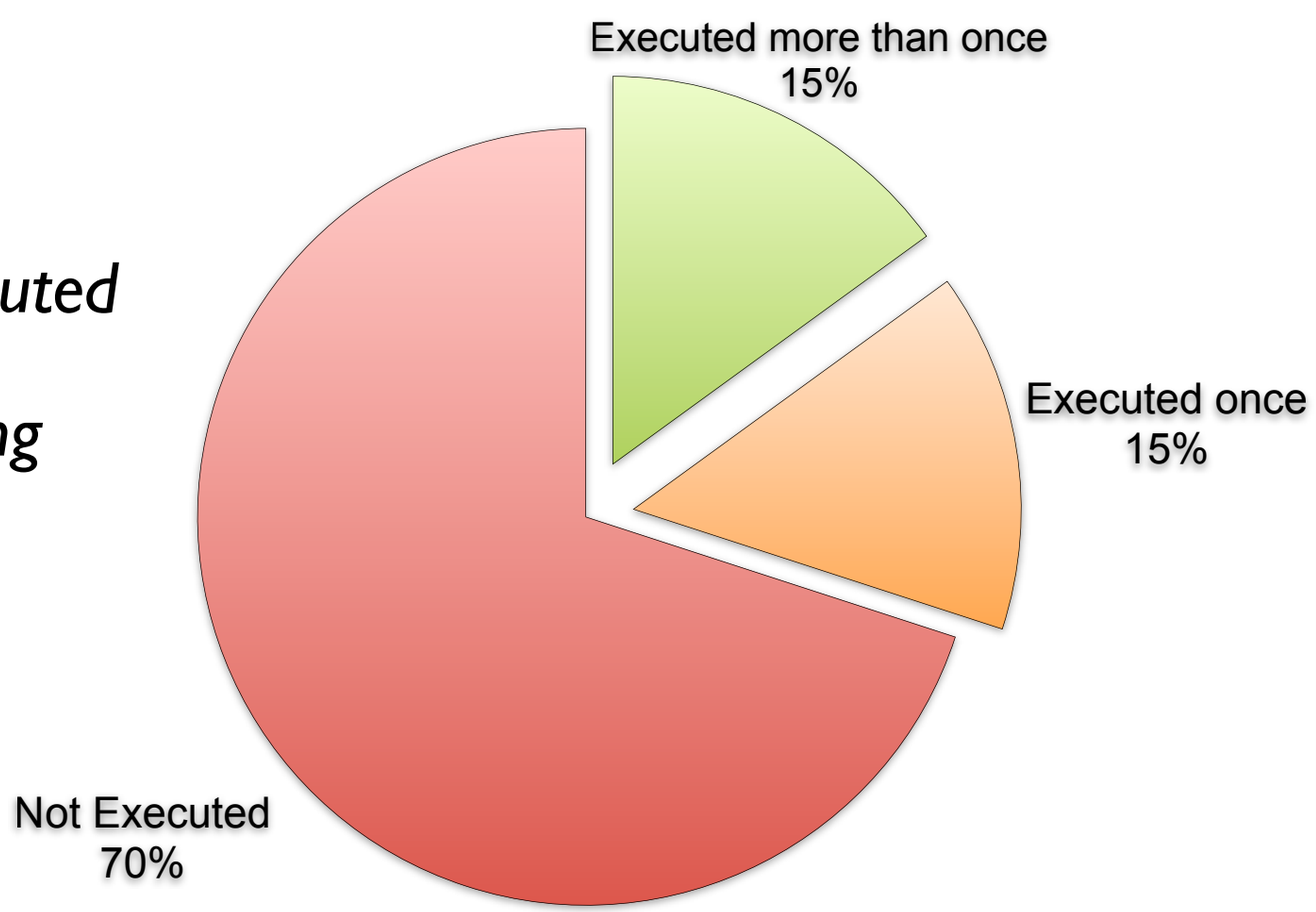
Do we really need 200MB of code?

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

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

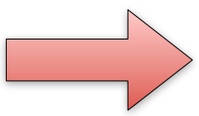
Mea culpa!

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.*

Concrete case: HCTypeTag

Just an example: hetero containers code in CMSSW

```
 91                :        template<class T, class Group >
 92                :        void
 93       1051794  :        HCTypeTagTemplate<T,Group>::doRegistration() {
 94       1051794  :           static bool registrationDone = false;
 95       1051794  :           if(!registrationDone){
 96           357  :              registerName(className(), typeid(T));
 97           357  :              registrationDone=true;
 98                :           }
 99                :        }
100                :        }
101                :    }
```

Concrete case: HCTypeTag

doRegistration is called 1M times (once per HCTypeTag objects construction)

```
91                :        template<class T, class Group >
92                :        void
93          1051794 :      HCTypeTagTemplate<T,Group>::doRegistration() {
94          1051794 :         static bool registrationDone = false;
95          1051794 :         if(!registrationDone){
96              357 :            registerName(className(), typeid(T));
97              357 :            registrationDone=true;
98                :         }
99                :      }
100               :      }
101               :  }
```

Concrete case: HCTypeTag

it does something only an handful of times

```
 91                :        template<class T, class Group >
 92                :        void
 93       1051794  :        HCTypeTagTemplate<T,Group>::doRegistration() {
 94       1051794  :           static bool registrationDone = false;
 95       1051794  :           if(!registrationDone){
 96           357  :              registerName(className(), typeid(T));
 97           357  :              registrationDone=true;
 98                :           }
 99                :        }
100                :        }
101                :    }
```

Concrete case: HCTypeTag

1 byte for the bool, 30 for guard variables, relocation etc.

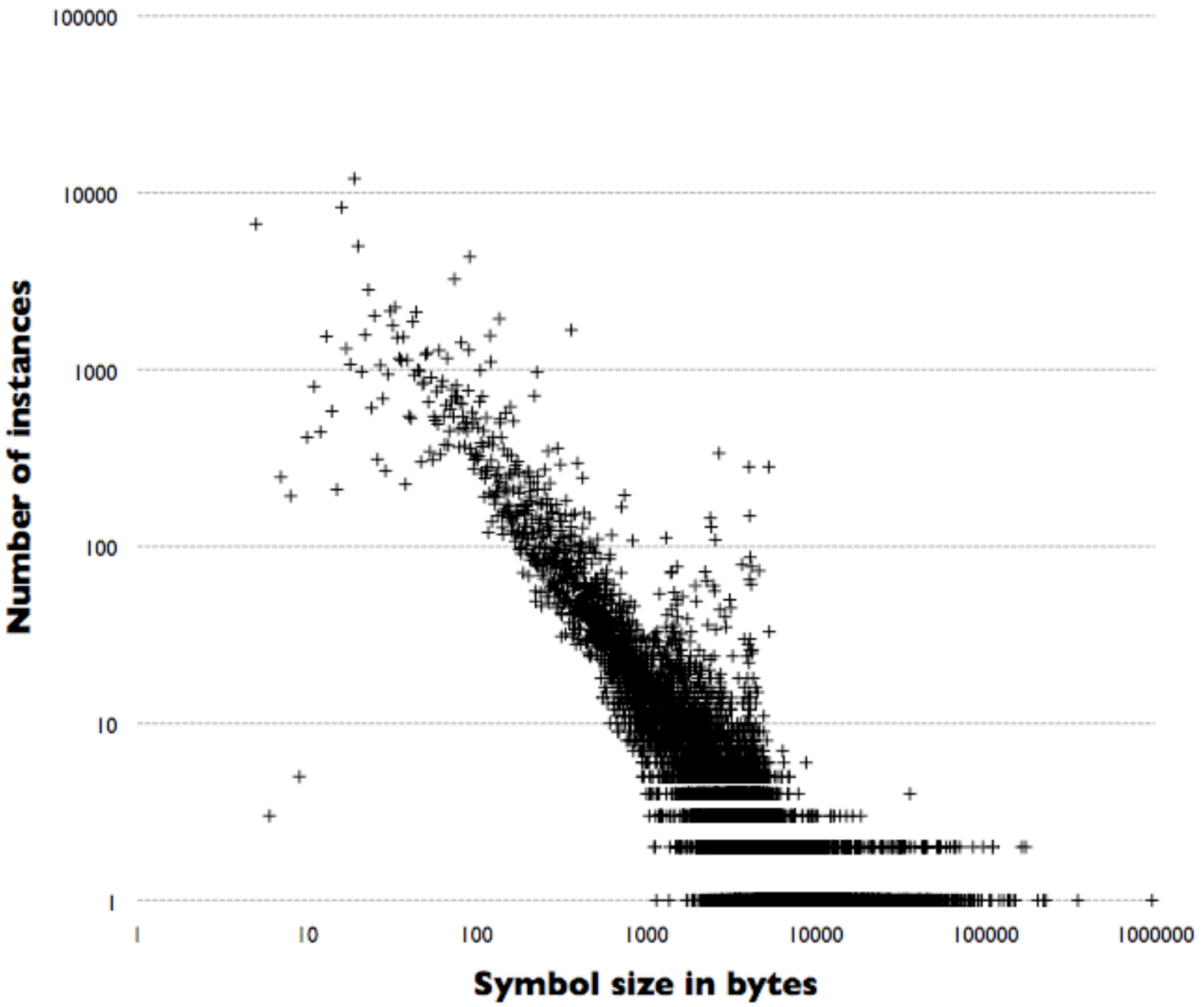
```
91                :        template<class T, cla s Group >
92                :        void
93          1051794 :        HCTypeTag<T,Group>::doRegistration() {
94          1051794 :          static bool registrationDone = false;
95          1051794 :          if(!registrationDone){
96              357 :            registerName(className(), typeid(T));
97              357 :            registrationDone=true;
98                  :          }
99                  :        }
100                 :      }
101                 :  }
```

Concrete case: HCTypeTag

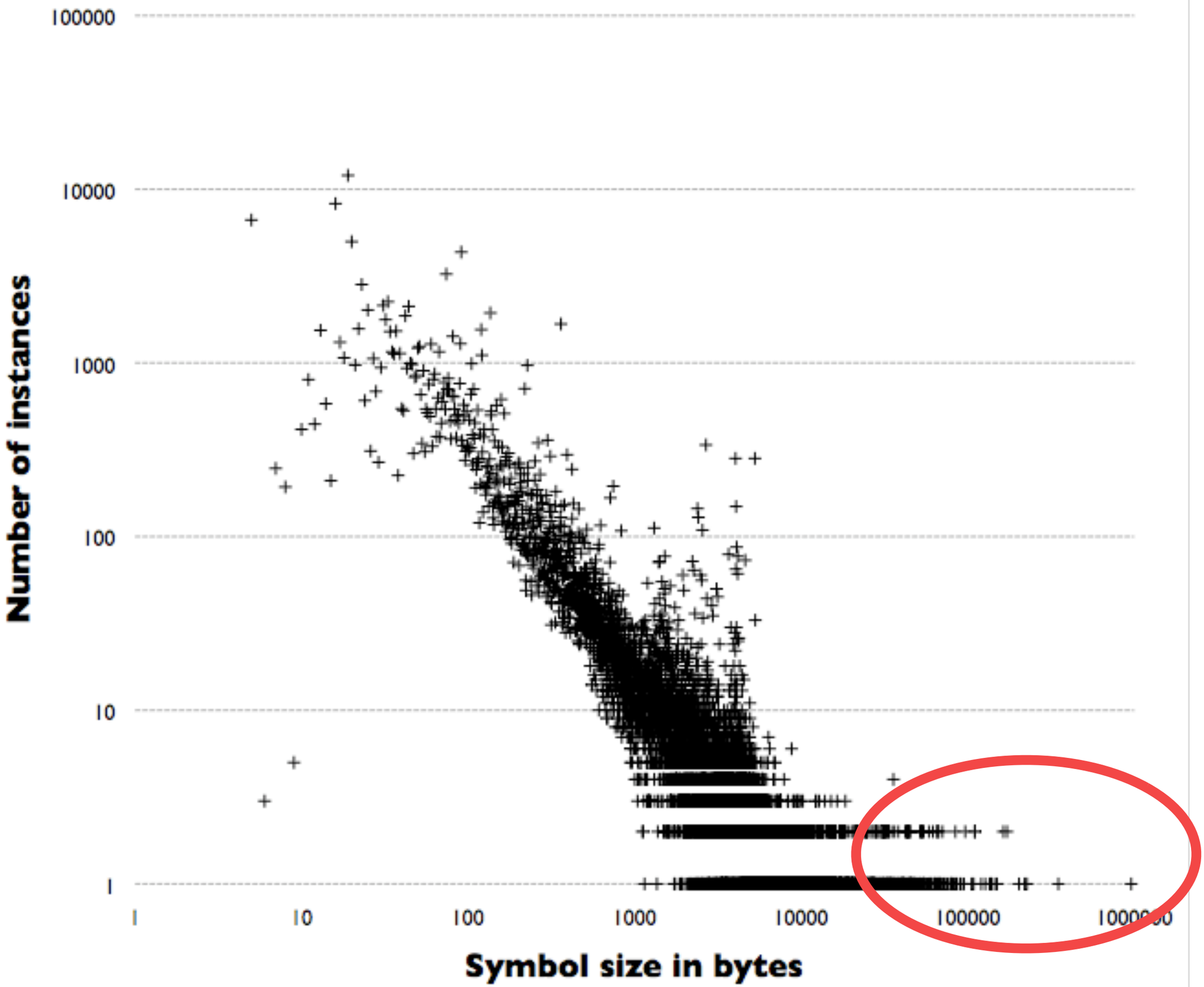
moreover it still could be mispredicted or cause other branches to be mispredicted by polluting the branch cache (this is 2 spurious branches!)

```
91                :        template<class T, class Group >
92                :        void
93          1051794 :        HCTypeTagTemplate<T,Group>::doRegistration() {
94          1051794 :          static bool registrationDone = false;
95          1051794 :          if(!registrationDone){
96              357 :            registerName(className(), typeid(T));
97              357 :            registrationDone=true;
98                :          }
99                :        }
100               :      }
101               :  }
```

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

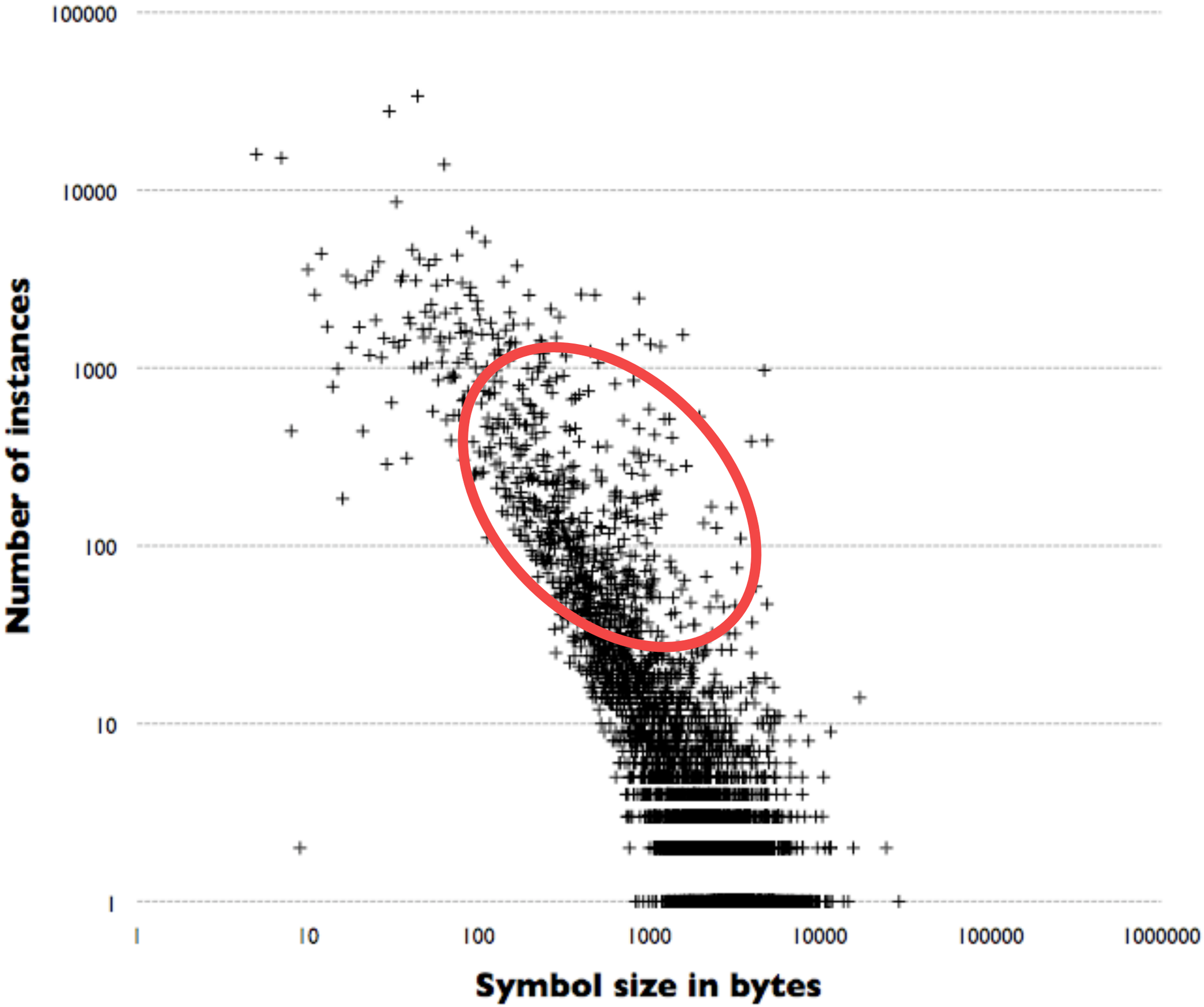
Implicit destructors bloat the code

```
// 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

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

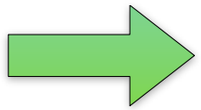
Exception & templates are a match made in hell

It's particularly bad to have template / inlined code throwing exceptions. The compiler (gcc) is simply not able to cope with it.

See also: http://gcc.gnu.org/ml/gcc/2009-04/msg00266.html

Simple solution: always have exceptions thrown in an out-of-line method or even better completely separate.

```
if (someError)
{
  throw MyExeceptionClass();
}
```



```
if (someError)
{
  throwMyExeceptionMethod();
}
```

Profiling with perfmon

Good part:

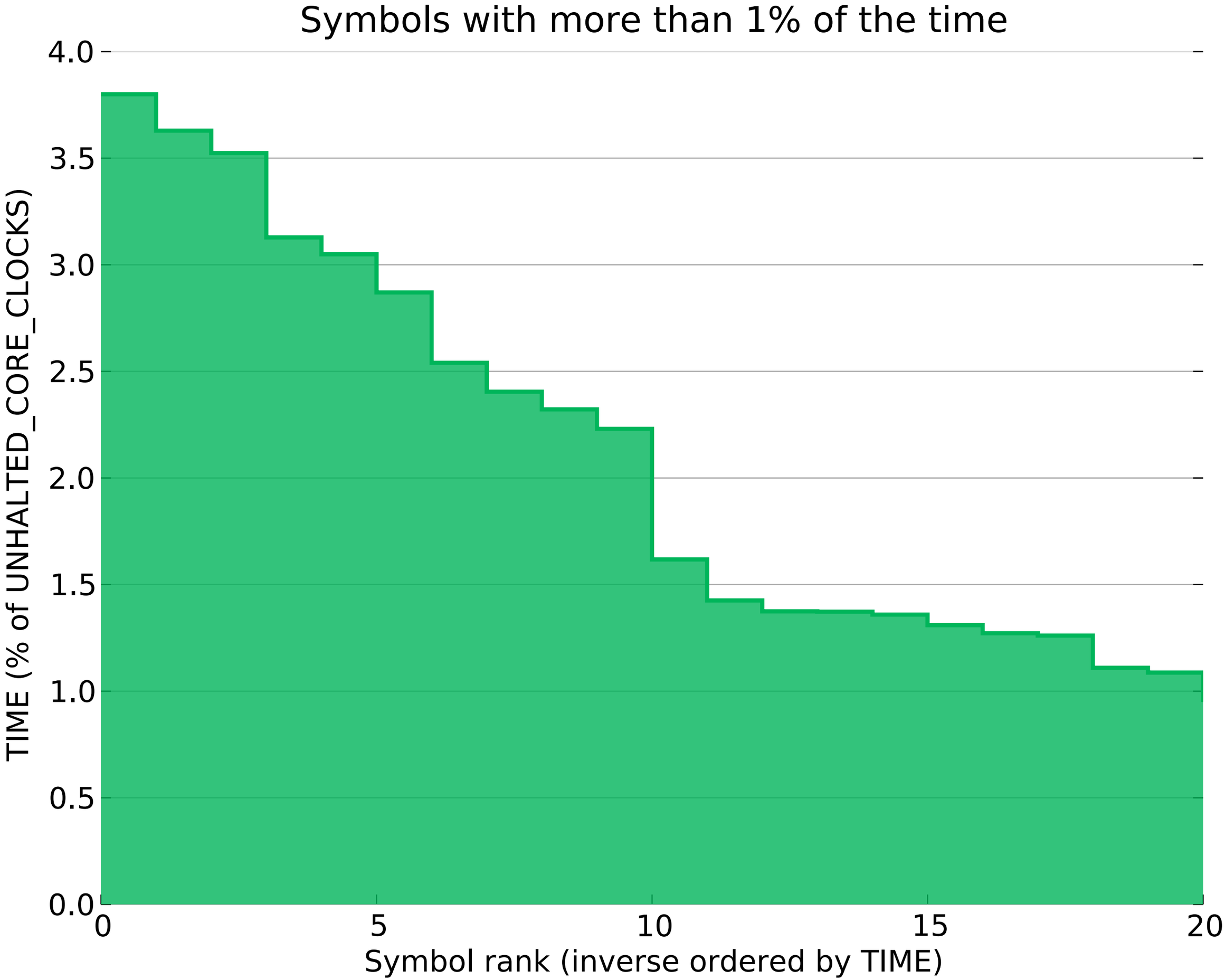
Accurate

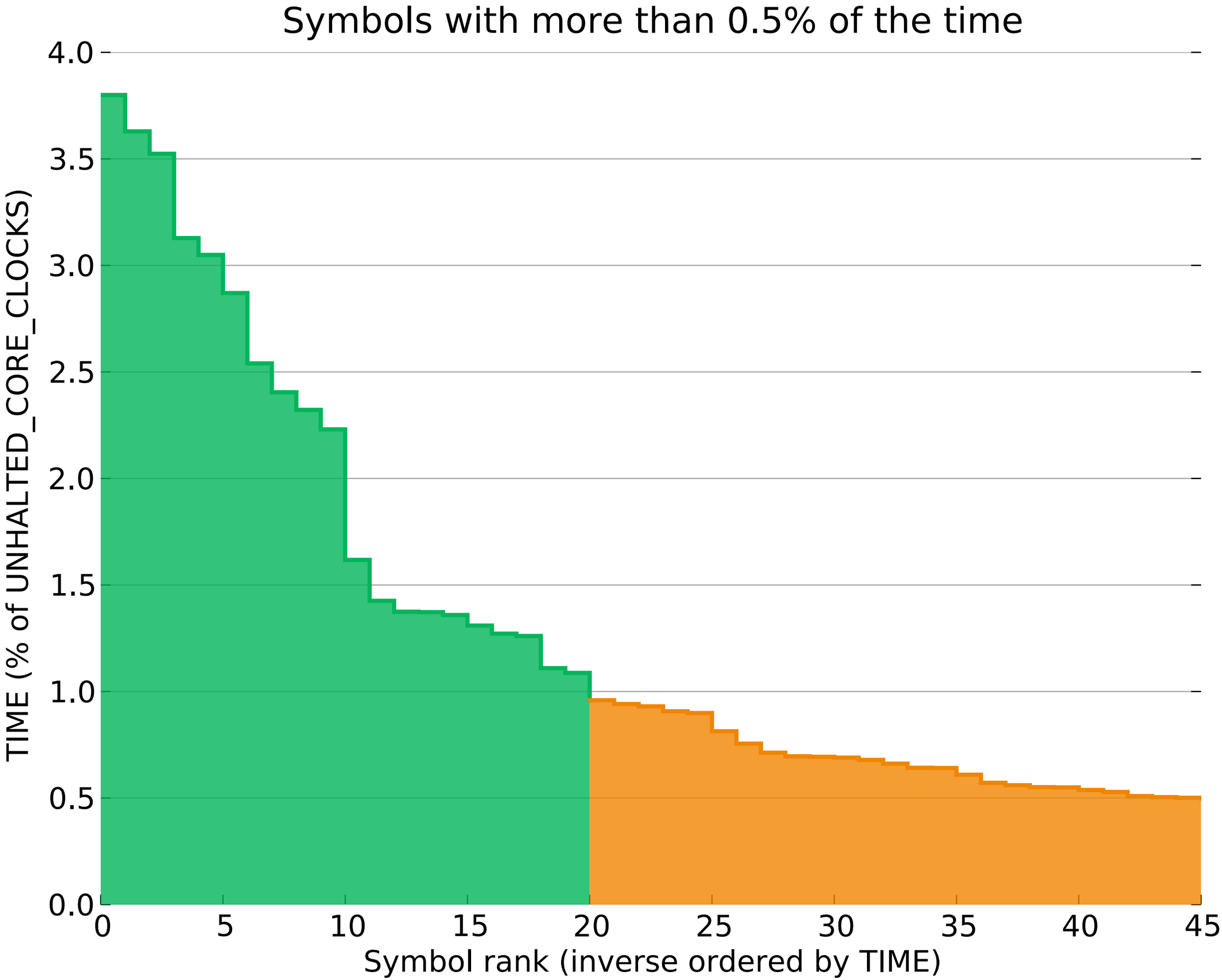
Lots of information

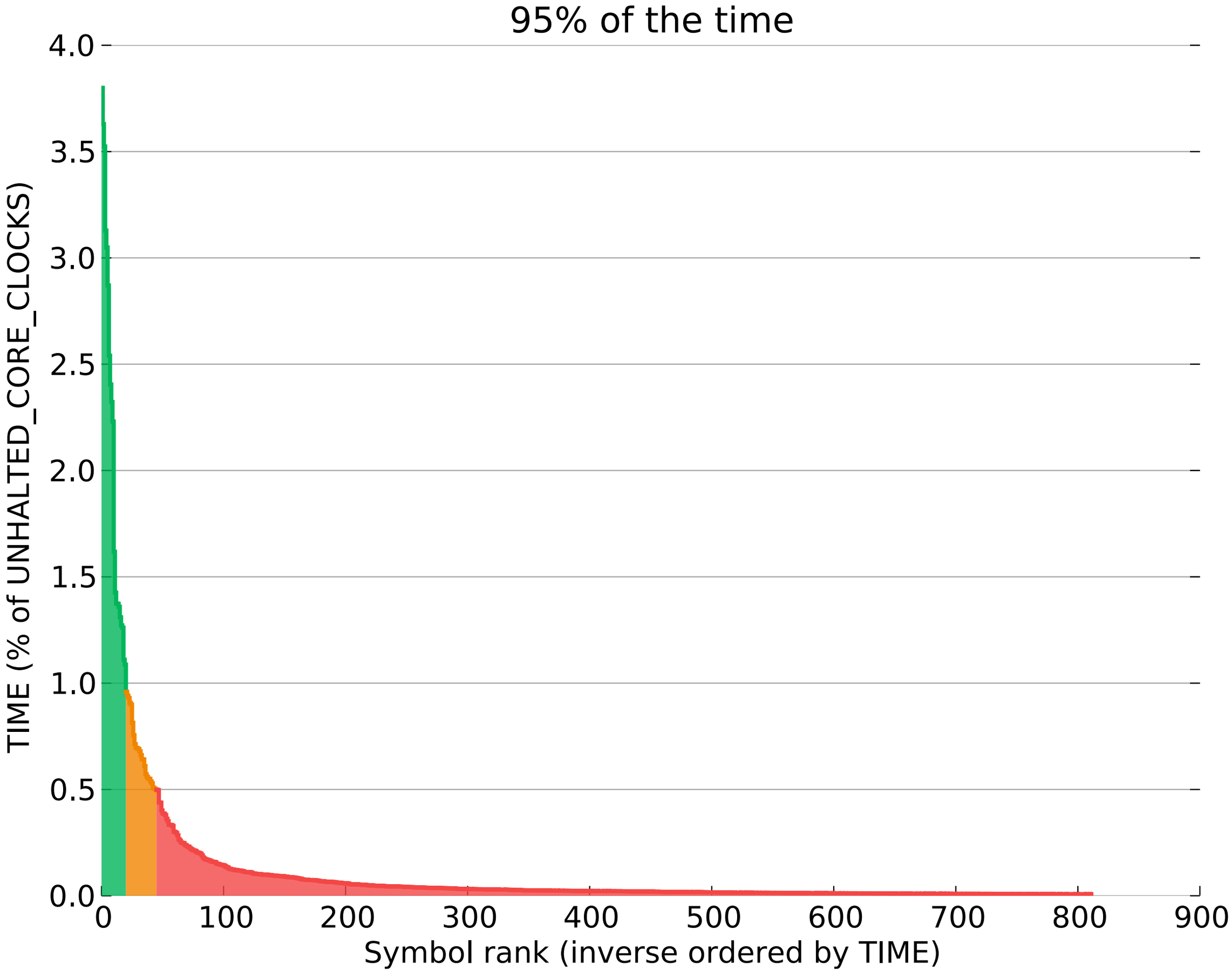
Bad part:

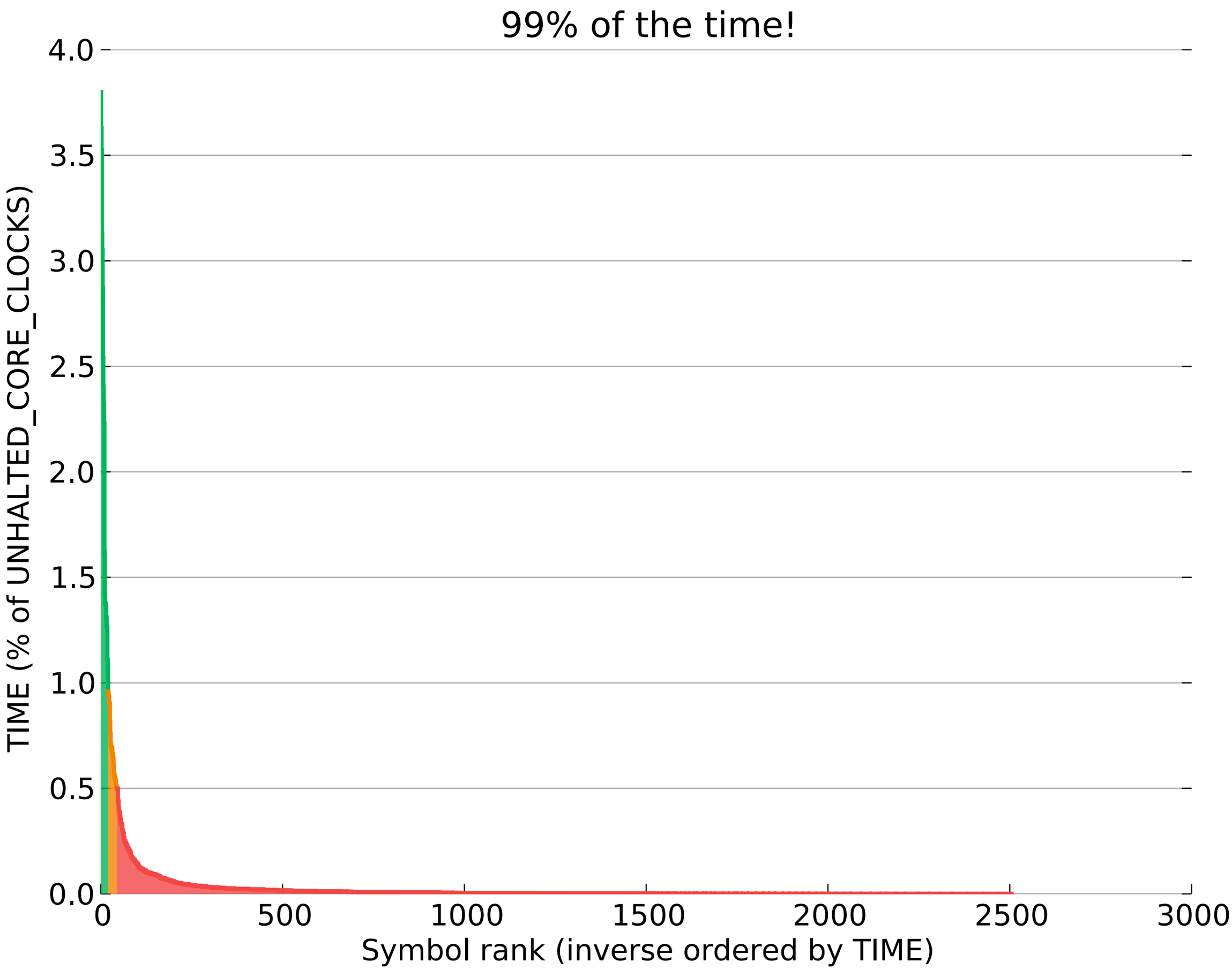
Flat profile (by hardware design?!?!)

Too much information


Symbols with more than 1% 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
5
10
15
20
Symbol rank (inverse ordered by TIME)


Symbols with more than 0.5% 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
5
10
15
20
25
30
35
40
45
Symbol rank (inverse ordered by TIME)


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)


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

Memory management

System stuff

Well known physics packages

Well known externals

See Peter's talk @ CHEP09

TIME (% of UNHALTED_CORE_CLOCKS)

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0

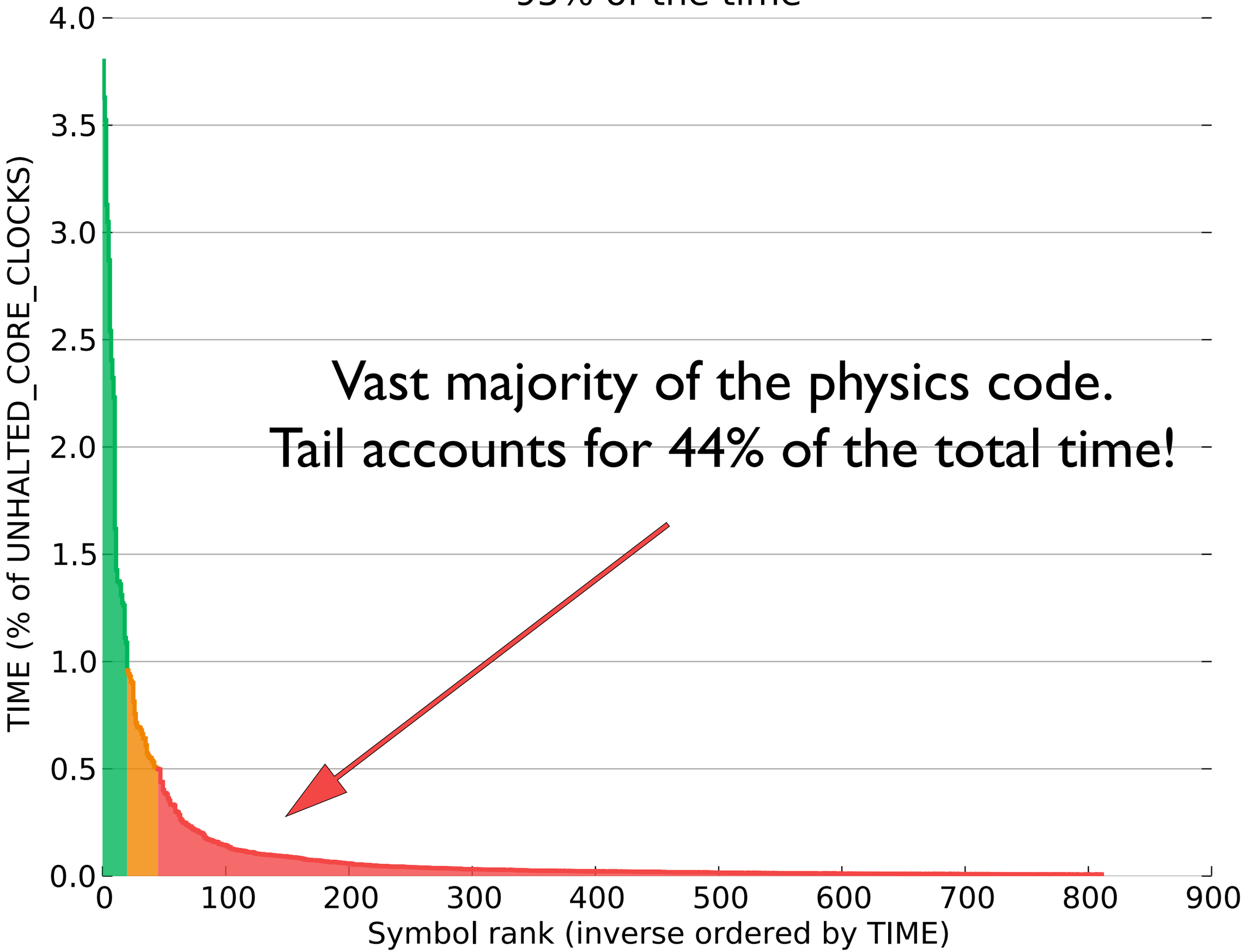
0 100 200 300 400 500 600 700 800 900

Symbol rank (inverse ordered by TIME)

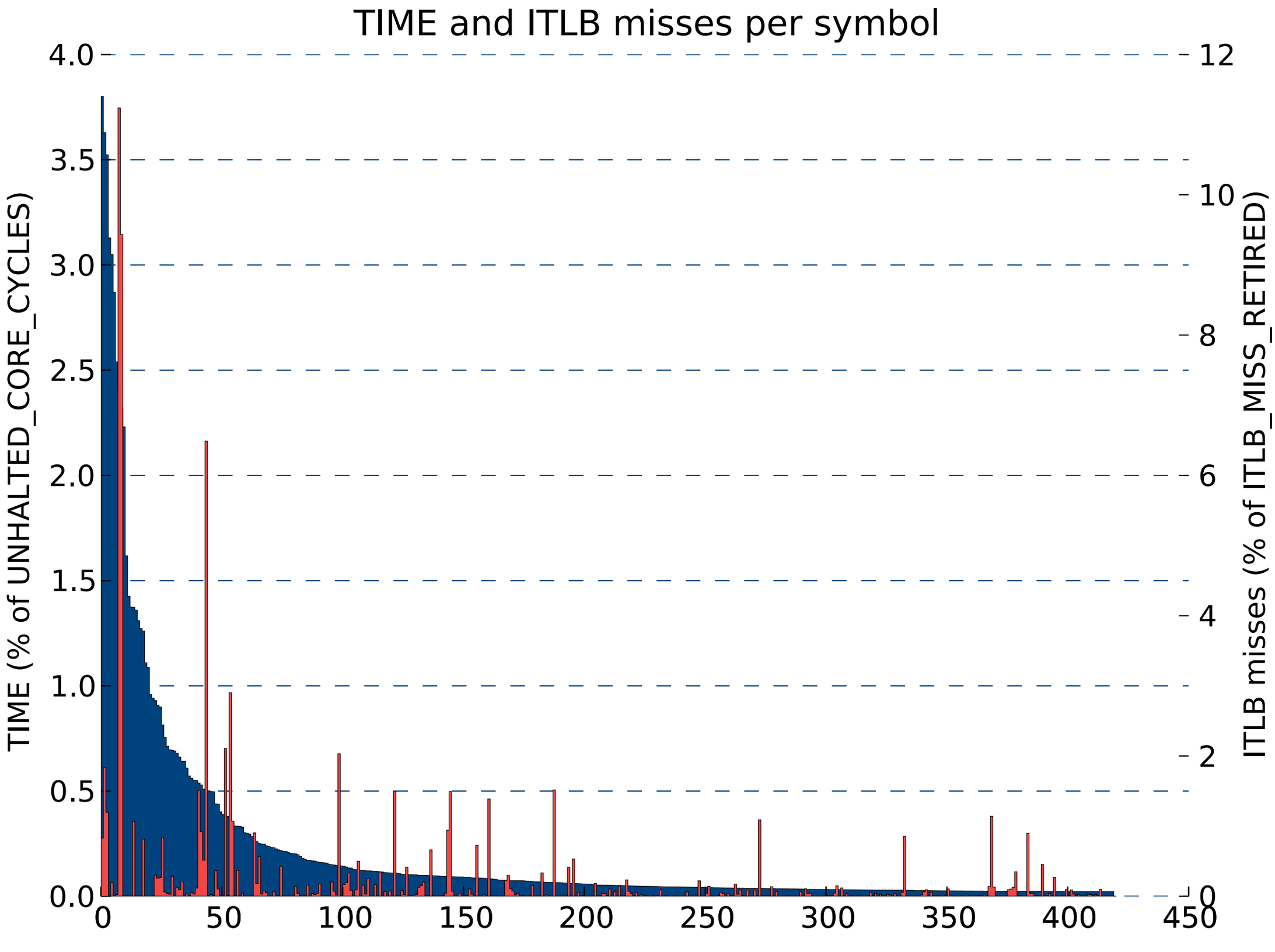
95% of the time

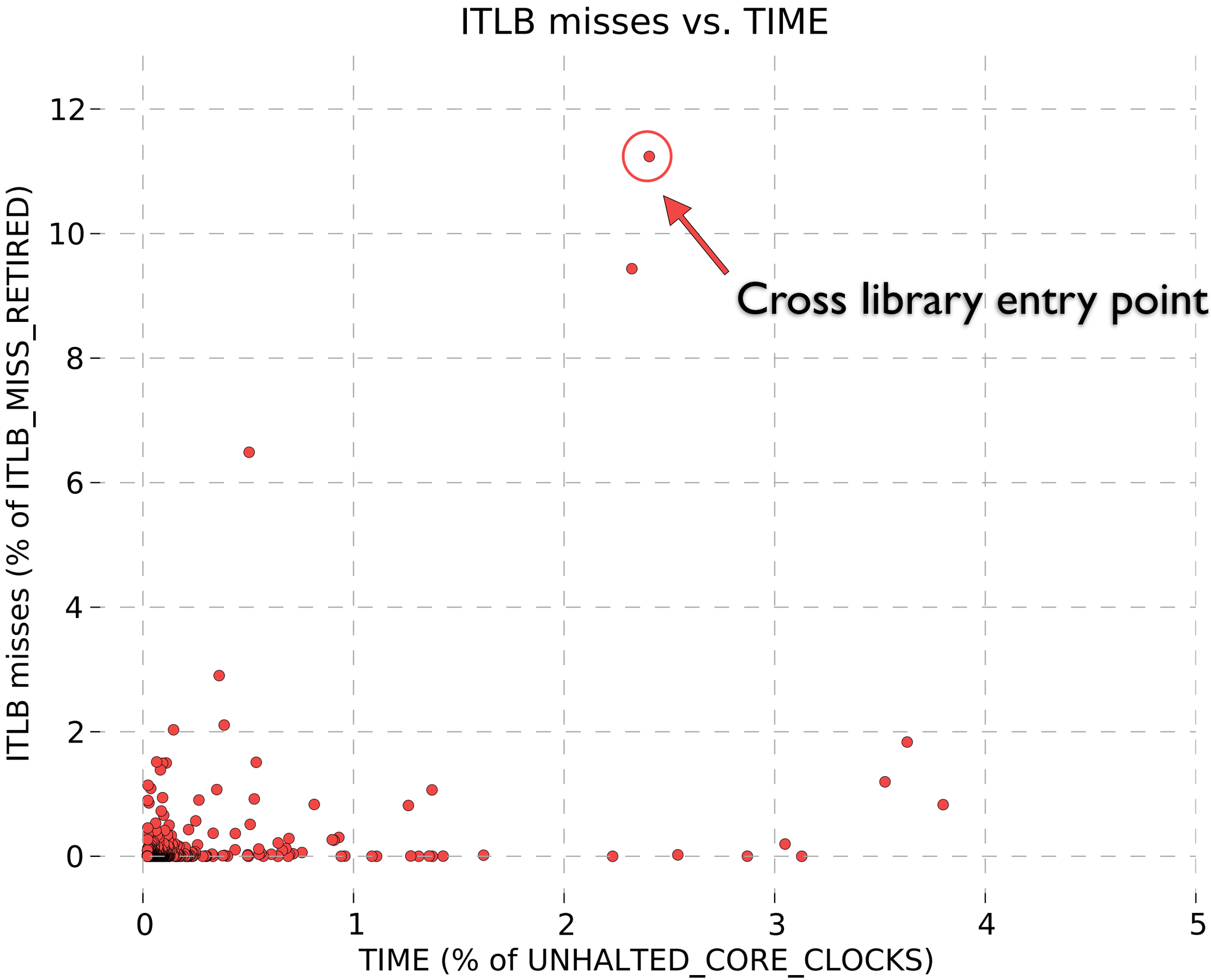
Vast majority of the physics code.

Tail accounts for 44% of the total time!

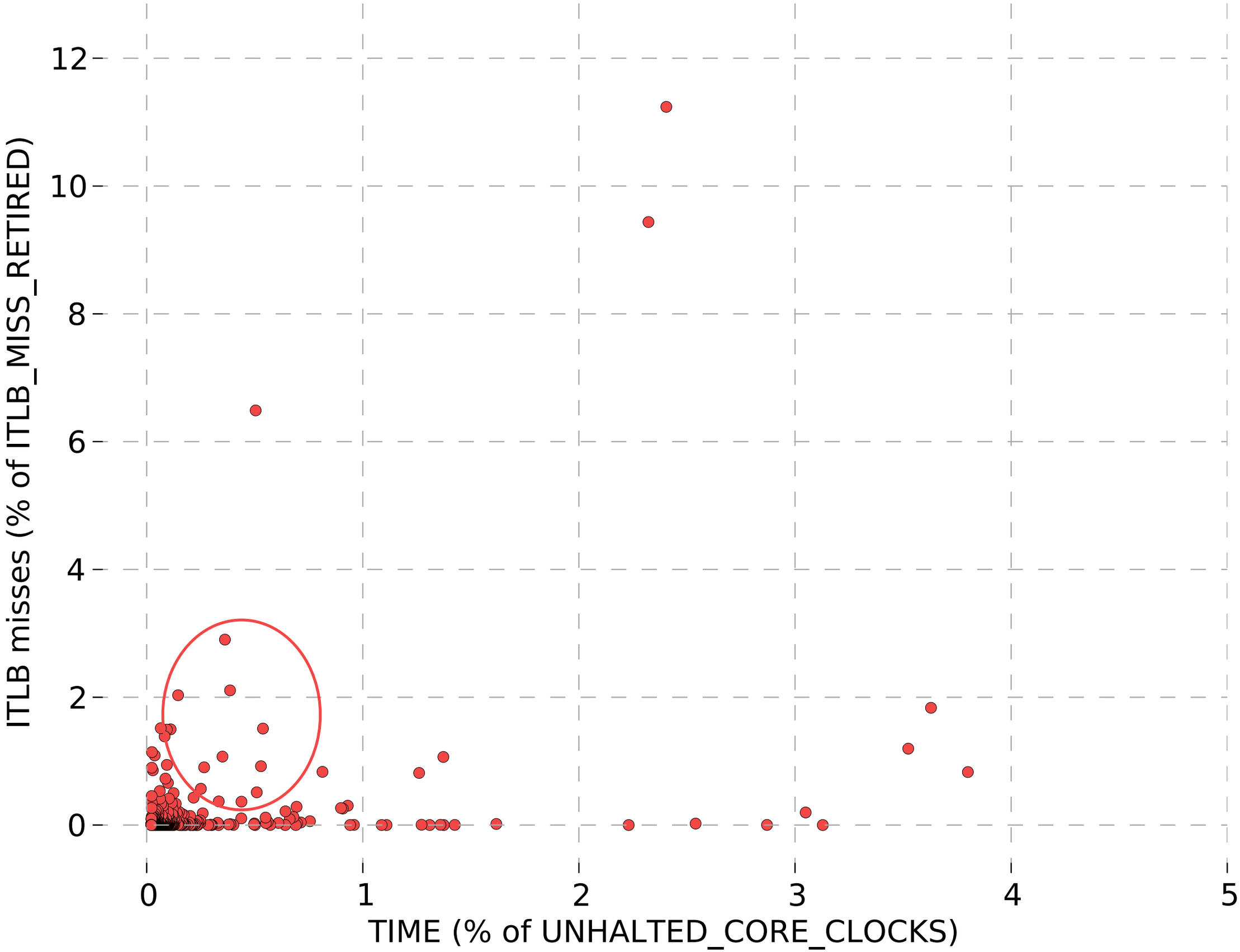


We need to correlate results!




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
Cross library entry point

ITLB misses vs. TIME



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..

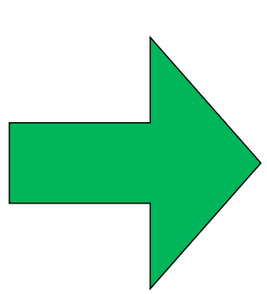
Concrete case: `SMatrix`

`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;
  }
};
```

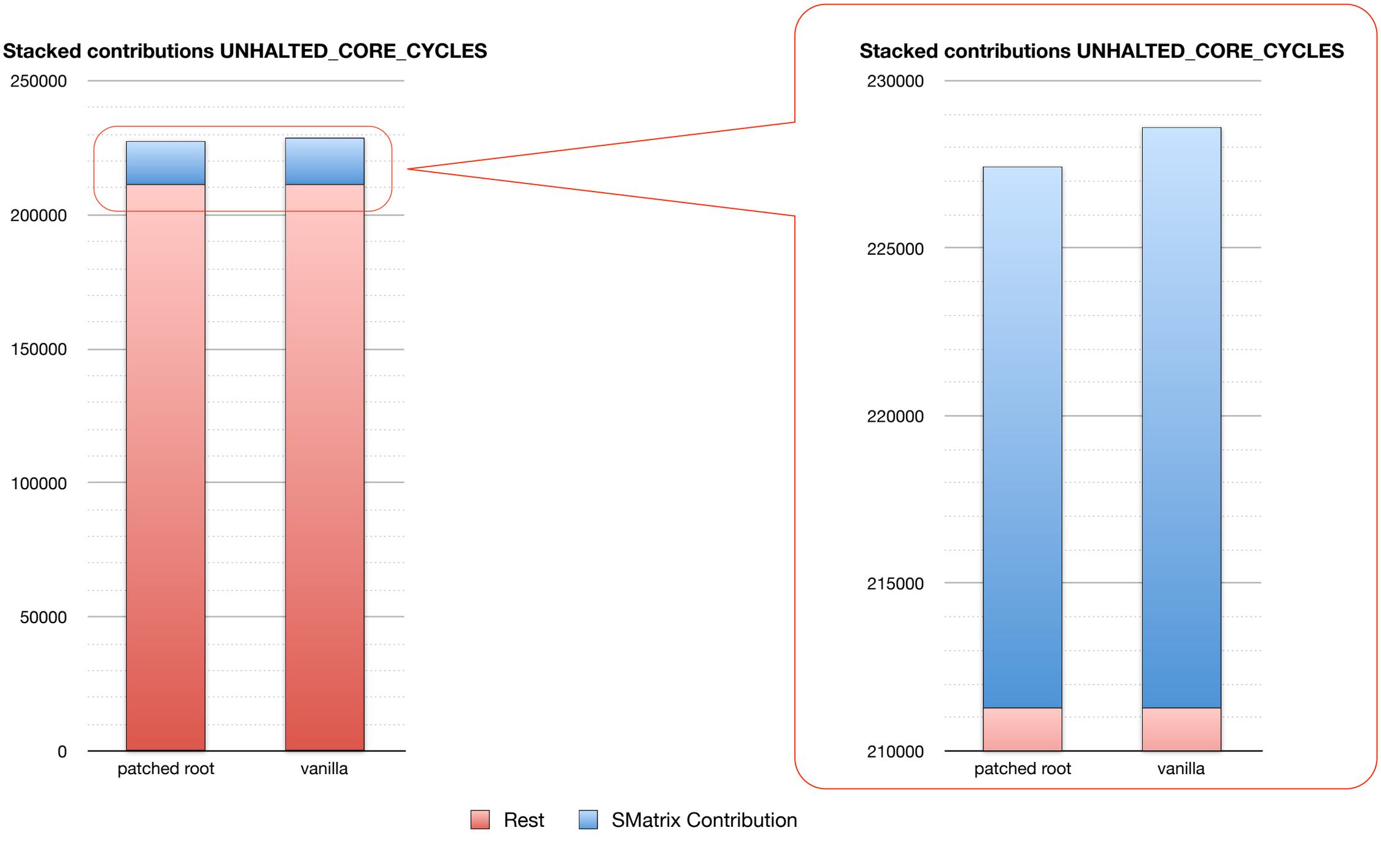


```
struct RowOffsetsBase
{
protected:
  static void init(int *v, int *offsets, unsigned int D);
};

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) { }
  ..
};
```

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

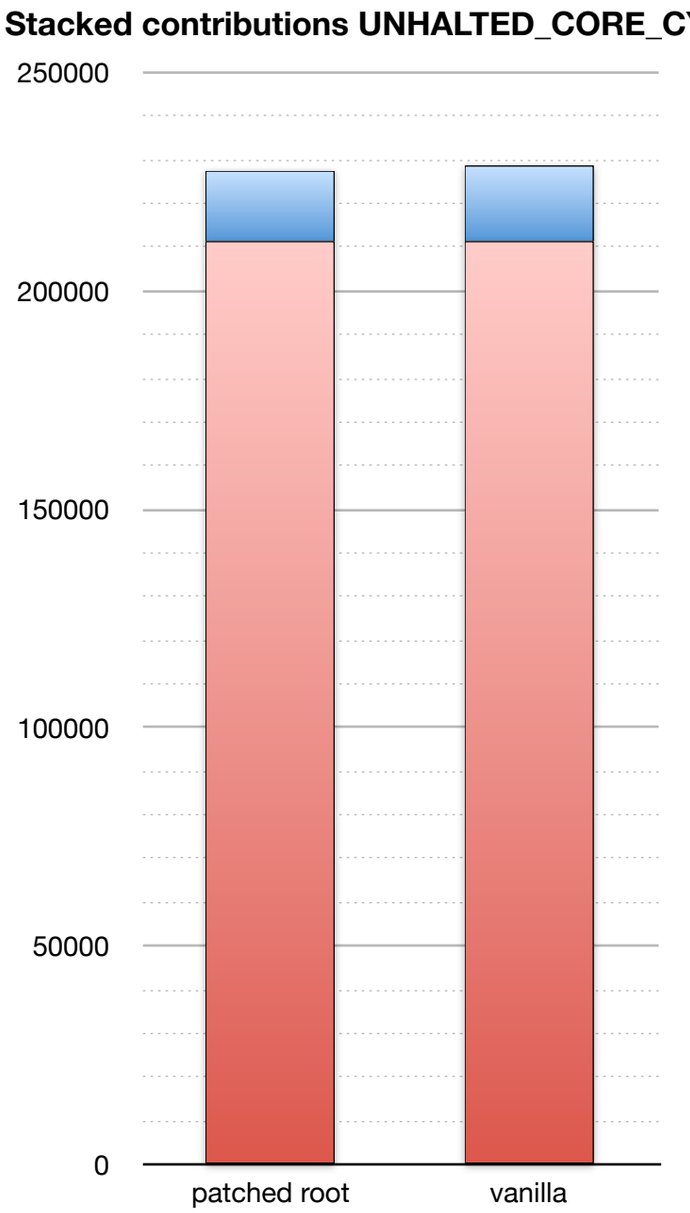
Profiling results

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

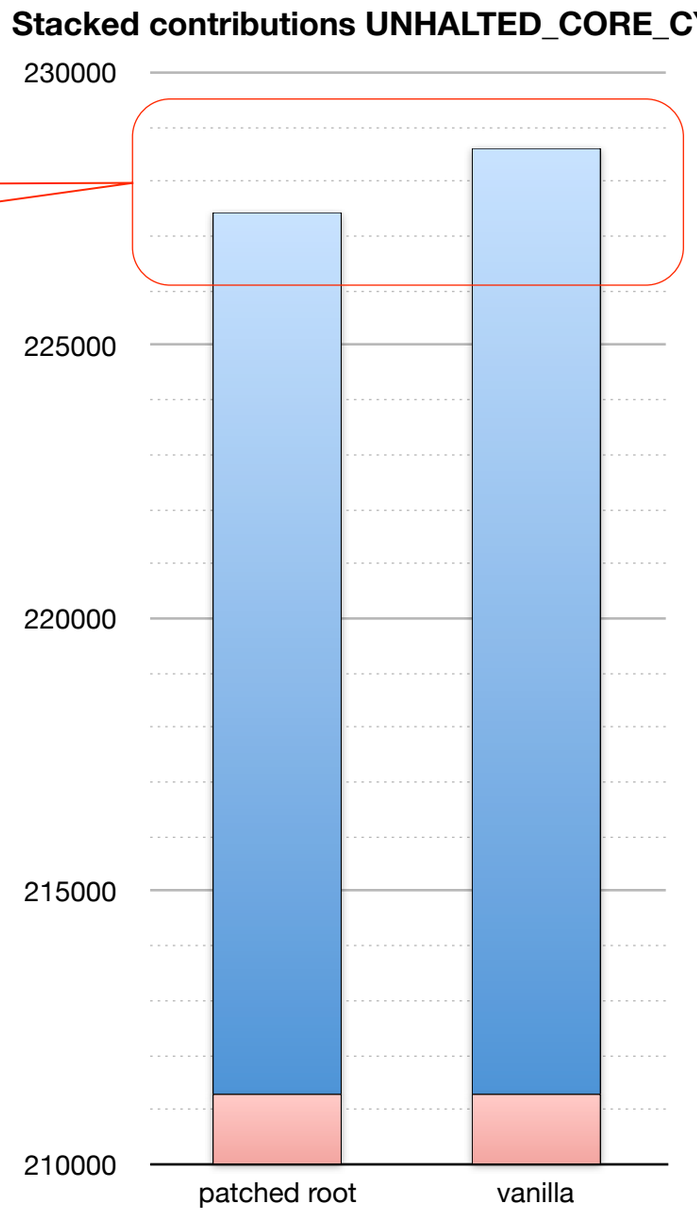


Profiling results

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



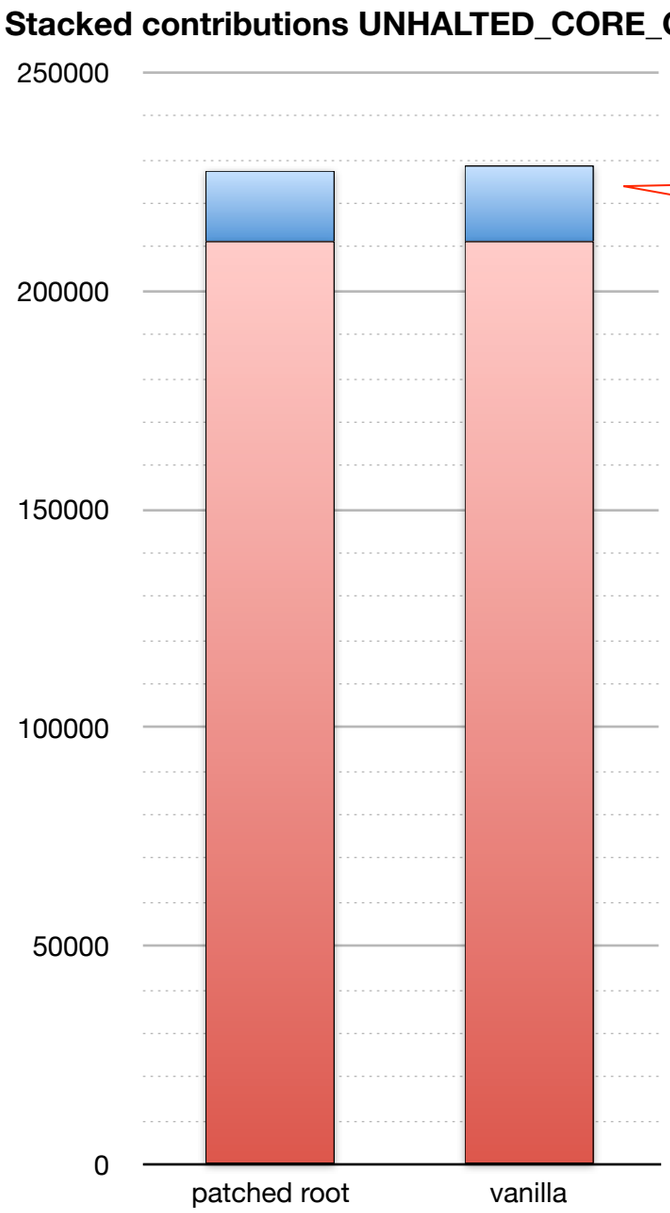
7% improvement for SMatrix related methods.



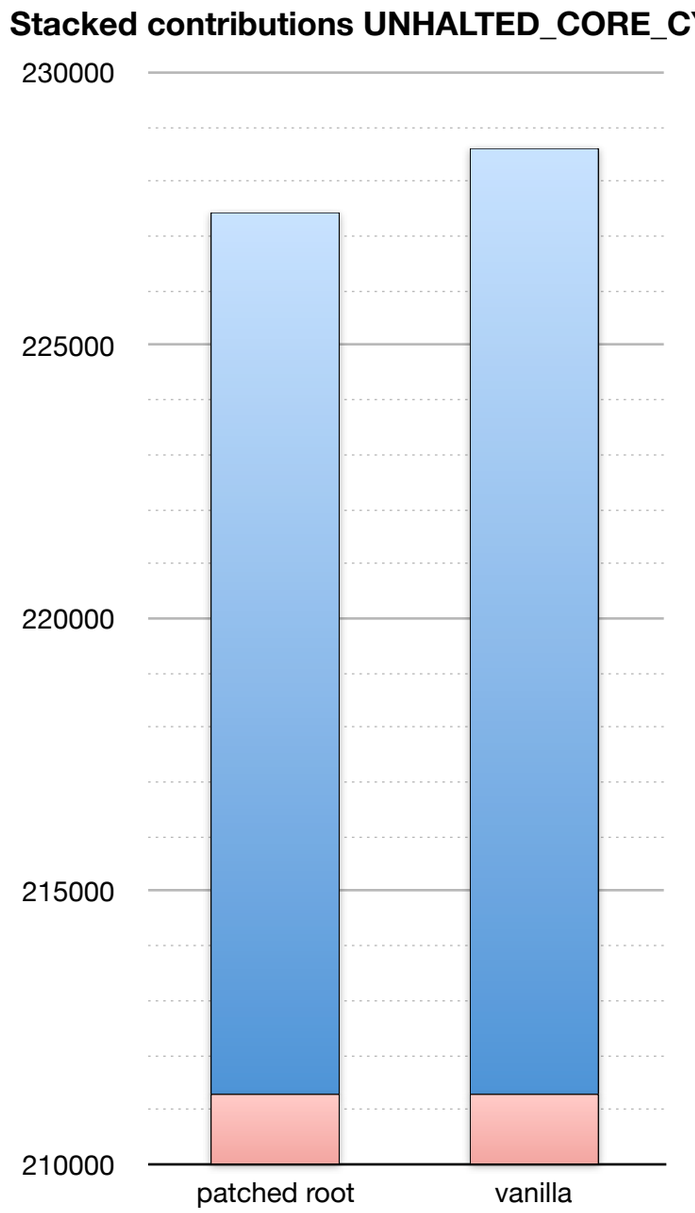
Rest SMatrix Contribution

Profiling results

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

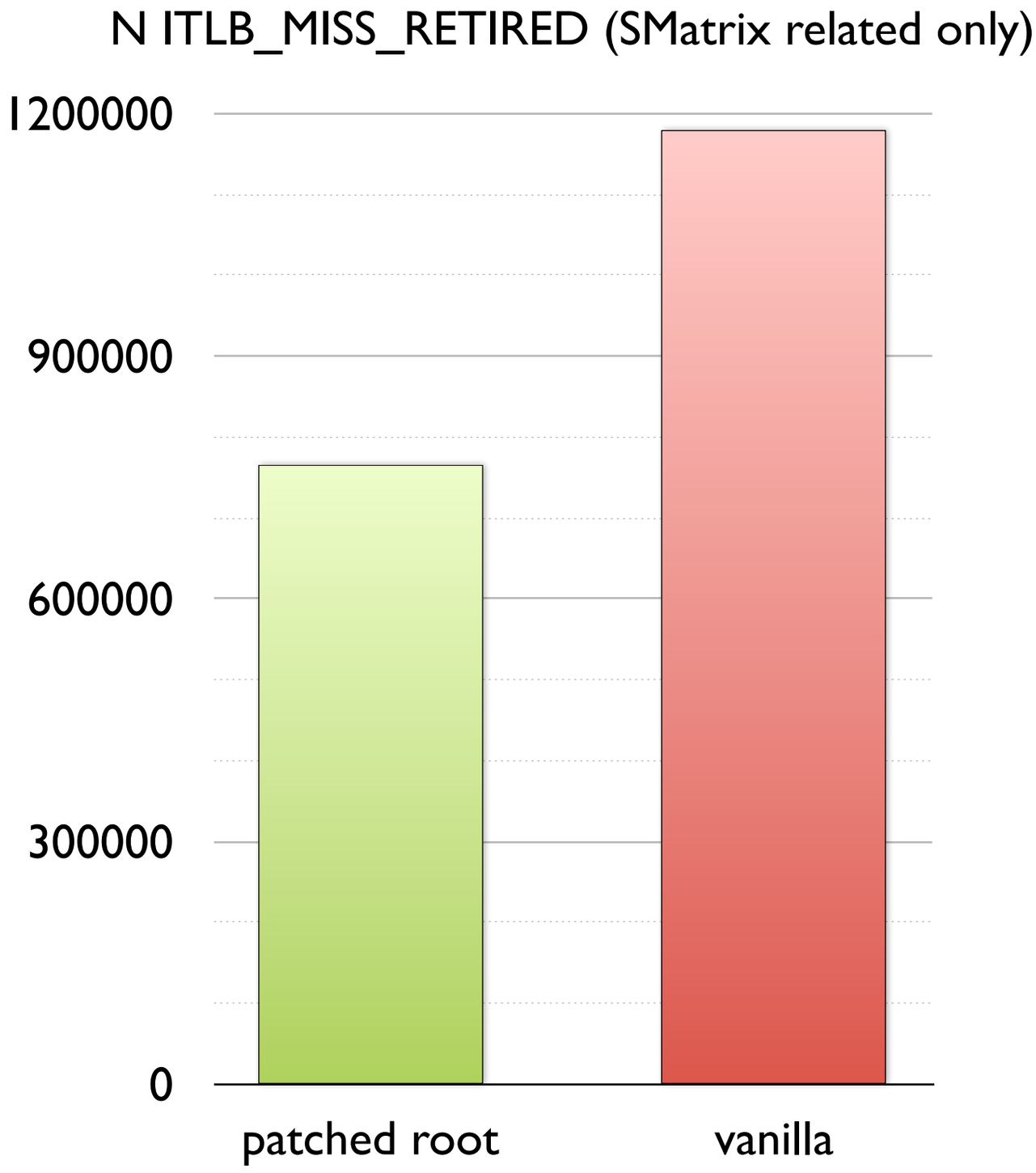


0.3% overall improvement



Rest SMatrix Contribution

Profiling results



Another example
&
what I'd like to have as profile analysis tool

```
const CartesianTrajectoryError& cartesianError() const {
  if (!hasError()) throw TrajectoryStateException(
    "FreeTrajectoryState: attempt to access errors when none available");
  if (!theCartesianErrorValid)
    createCartesianError();
  return theCartesianError;
}
```

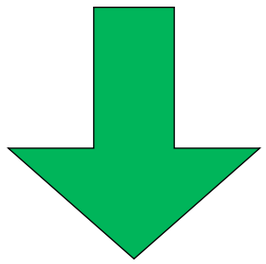
```
const CartesianTrajectoryError& cartesianError() const {
  if (!hasError()) throw TrajectoryStateException(
    "FreeTrajectoryState: attempt to access errors when none available");
  if (!theCartesianErrorValid)
    createCartesianError();
  return theCartesianError;
}
```

ICACHE misses are in `hasError()` (method size is small)

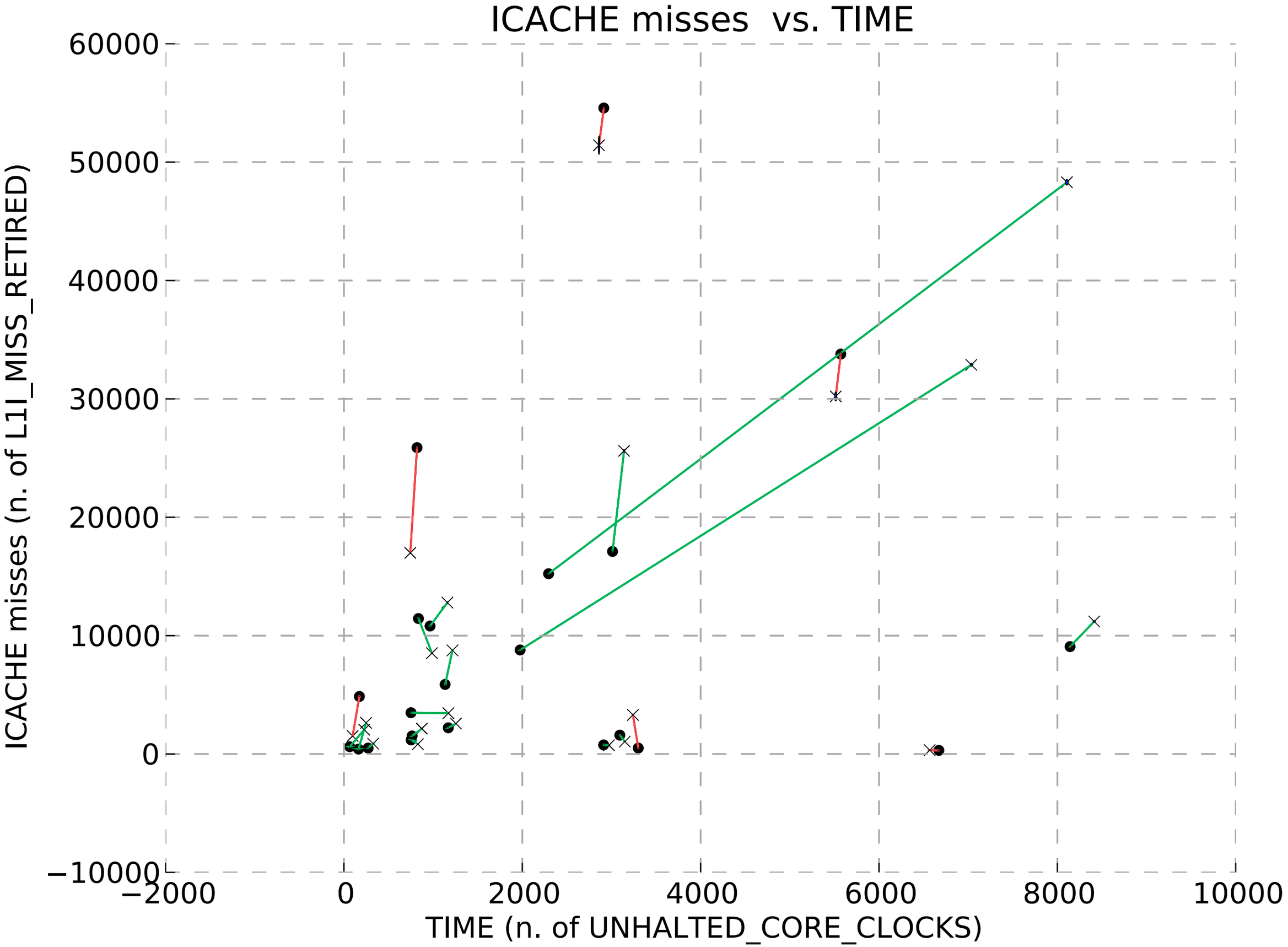
```
const CartesianTrajectoryError& cartesianError() const {
  if (!hasError()) throw TrajectoryStateException(
    "FreeTrajectoryState: attempt to access errors when none available");
  if (!theCartesianErrorValid)
    createCartesianError();
  return theCartesianError;
}
```

`cartesianError()` (which is large because of the throw) pushes `hasError()` out of the cache!

```
const CartesianTrajectoryError& cartesianError() const {
  if (!hasError()) throw TrajectoryStateException(
    "FreeTrajectoryState: attempt to access errors when none available");
  if (!theCartesianErrorValid)
    createCartesianError();
  return theCartesianError;
}
```



```
const CartesianTrajectoryError& cartesianError() const {
  if (!hasError()) throwMissingErrors();
  if (!theCartesianErrorValid)
    createCartesianError();
  return theCartesianError;
}
```


ICACHE misses vs. TIME
ICACHE misses (n. of L1I_MISS_RETIRED)
TIME (n. of UNHALTED_CORE_CLOCKS)
60000
50000
40000
30000
20000
10000
0
−10000
−2000
0
2000
4000
6000
8000
10000

What I would like to get from a profiling GUI:

Correlate information

Scatter plot between time & some quantity that identifies a reason for the time (e.g. cache misses, mispredicted branches)

“diff” history

Ability to overlay results before and after patching

Ability to filter out symbols that don’t change

Navigate results

Pop-up information about a given arrow (symbol name, actual counts, other counters, etc)