Performance Measurements in HPC: Techniques, Side Effects, and Use

Hessisches Kompetenzzentrum
für Hochleistungsrechnen

brainware for science

Agenda

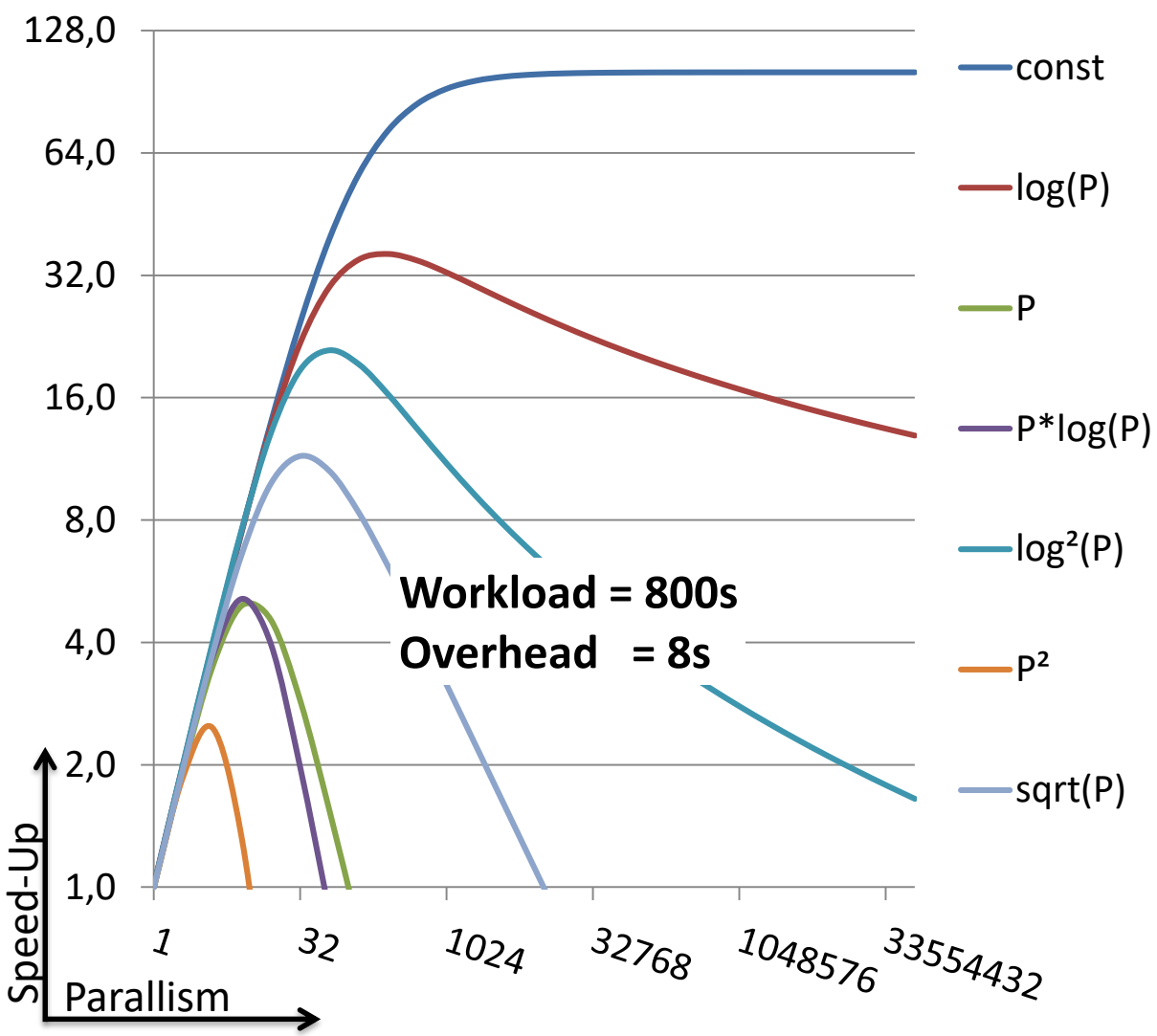
- Why is performance analysis critical?
- On metrics and models
- Understanding and context
- Measurement techniques
  - Sampling
  - Instrumentation
  - Overhead and perturbation
  - Optimizations
- Measurement-data storage and processing
- Tangent on *Performance Analysis Tools*
- Conclusion

Checking performance is important

- $\text{Performance} = \frac{\text{Work}}{\text{Time}}$
- $\text{Speed}up = \frac{Work}{Overhead(P) + \frac{Work}{P}}$
- $\text{Efficiency} = \frac{\text{Work}}{\text{Units of Investment}}$

Simplified

- Optimize performance by doing more work in the same time
- Doing more work, requires
  - Using all processing units at peak capacity
  - More processing units

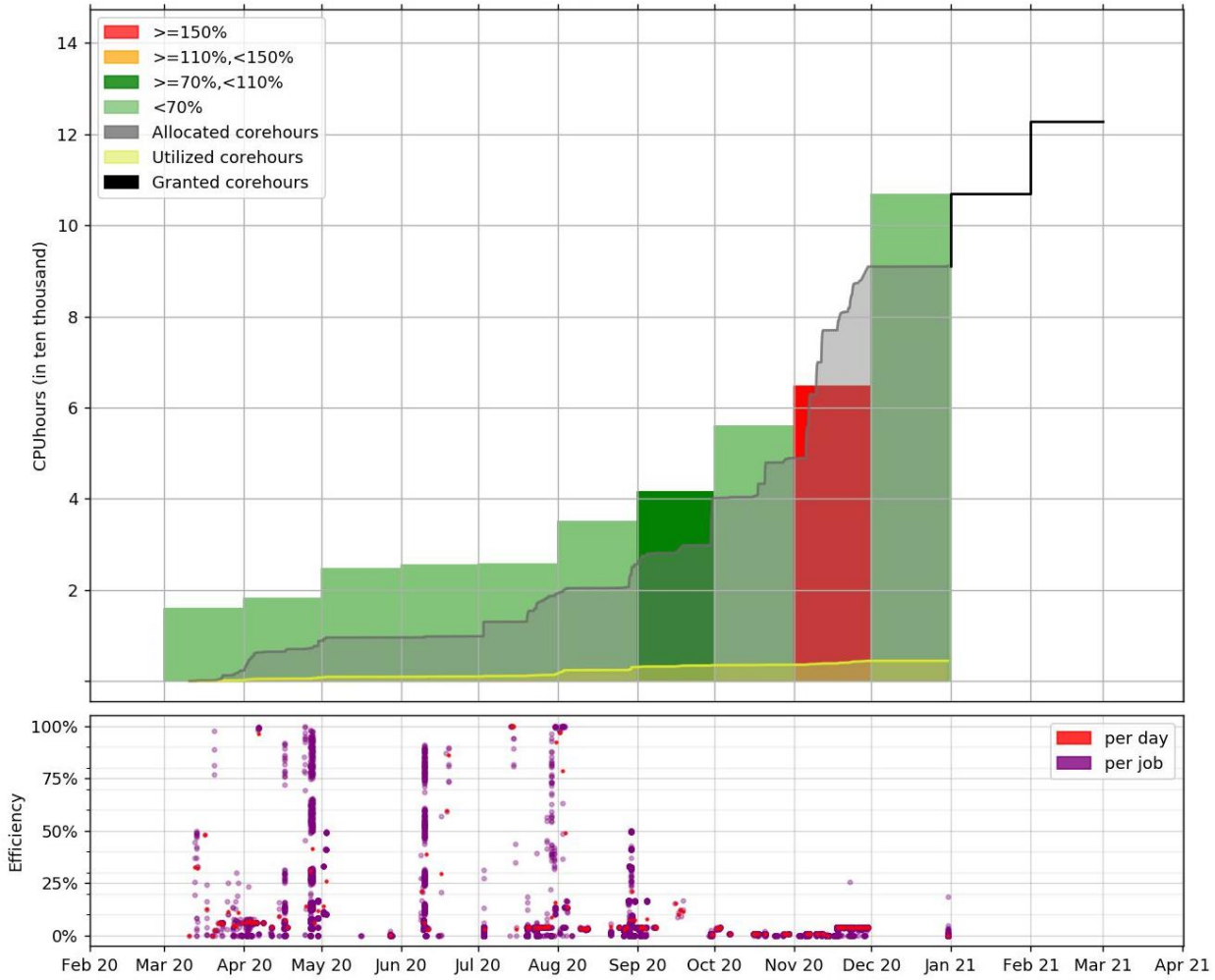


Checking performance is important

- $\text{Performance} = \frac{\text{Work}}{\text{Time}}$
- $\text{Speed}up = \frac{Work}{Overhead(P) + \frac{Work}{P}}$
- $\text{Efficiency} = \frac{\text{Work}}{\text{Units of Investment}}$

Simplified

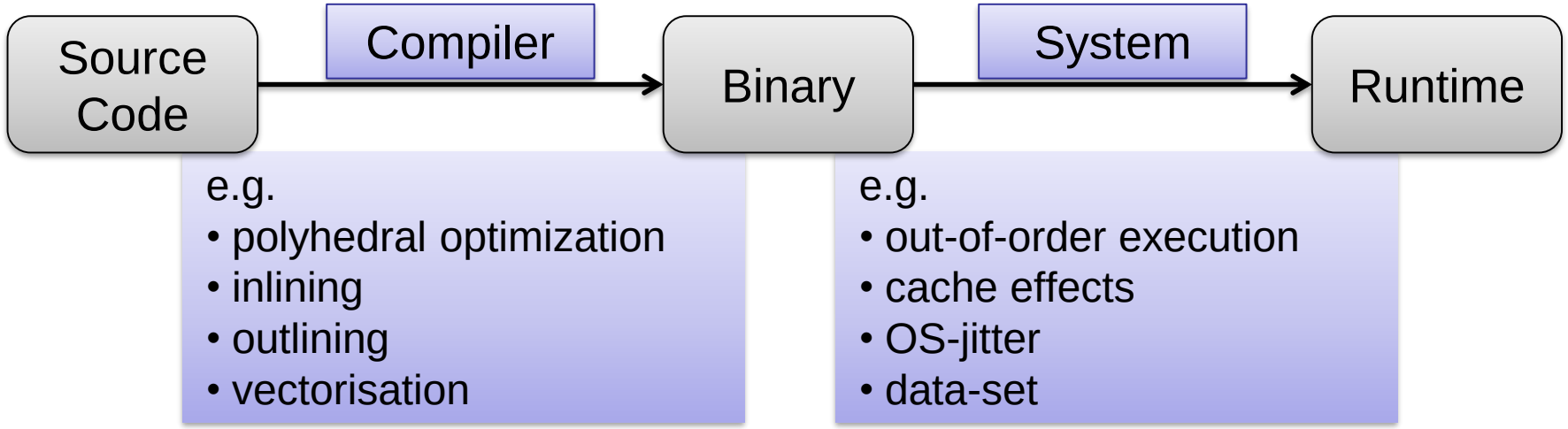
- Optimize performance by doing more work in the same time
- Doing more work, requires
  - Using all processing units at peak capacity
  - More processing units



Why is performance analysis challening

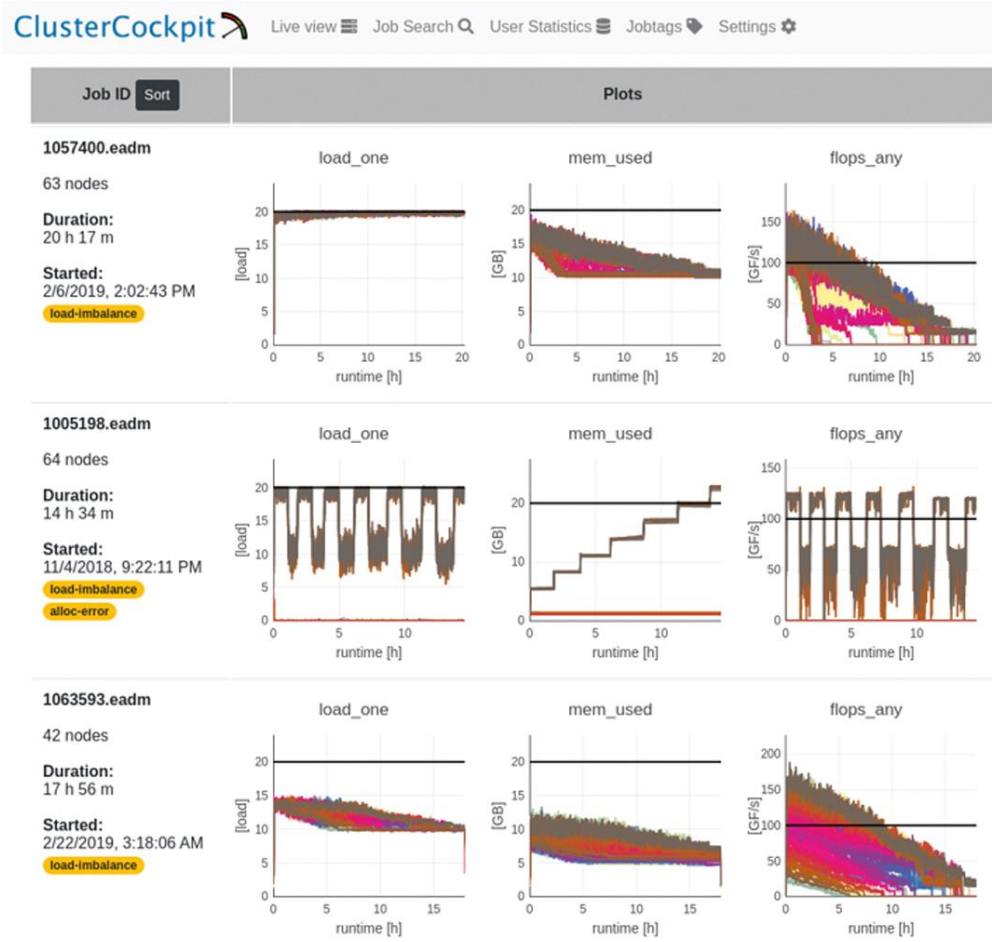
- Parallel performance is a compound result of
  - Program logic & algorithms
  - Data-set
  - System & Machine properties
  - Compilers and libraries

- Compounding layered effect of influences:



Ideal and reality

- Utilization of hardware resources
- Ideal:
  - Fully loaded memory
  - Saturated network-connection
  - 100% CPU utilization
  - …
- This is usually not possible:
  - Phases in a program
  - Circumventing specific limitations or hardware comes at a price: overhead
  - Conserve energy
    - Compromise with performance



Src: https://www.rrze.fau.de/2020/01/webanwendung-zum-jobspezifischen-performance-monitoring-eine-erfolgsgeschichte-keine-ressourcen-verschwendung-gezielte-optimierung/

Checking your assumptions

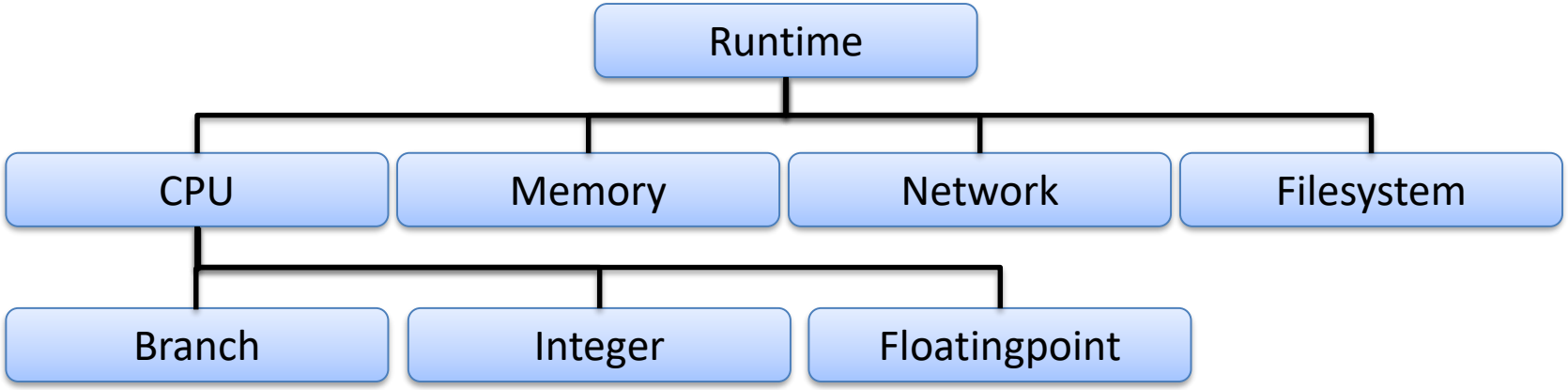
- Determine what part of your program is the dominant aspect
- Computer wisdom: make the dominant case fast
- Verify that this is the dominant aspect

- Observation from past projects, where the "perceived" hotspot
  - Was not the **only** hot-spot
  - Was not the hot-spot at all
  - Was slowed down by non-essential work
    - I/O
    - Screen-output

On models and metrics

Analytical models:

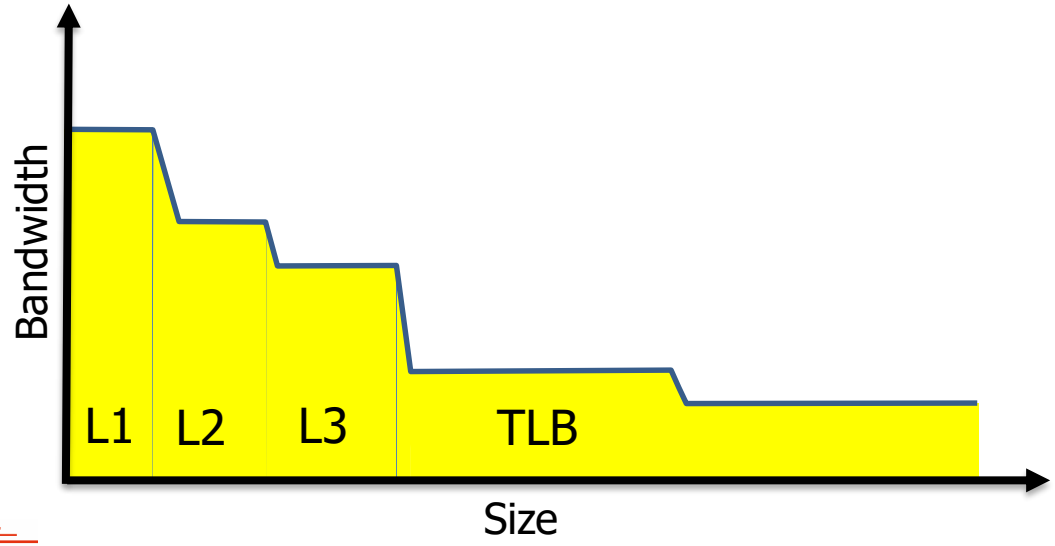
- Simple model
  - $Runtime = \frac{Number\ of\ Instructions}{\frac{Instructions}{Second}} + \frac{\frac{Memory}{Instruction}}{\frac{Memory}{Second}} + \frac{Memory\ to\ Communicate}{\frac{Memory}{Second}} + \cdots$
- Metrics aim to capture and overt aspects of this expression
- Each of the terms above can (and should) be split into more precise sub-terms



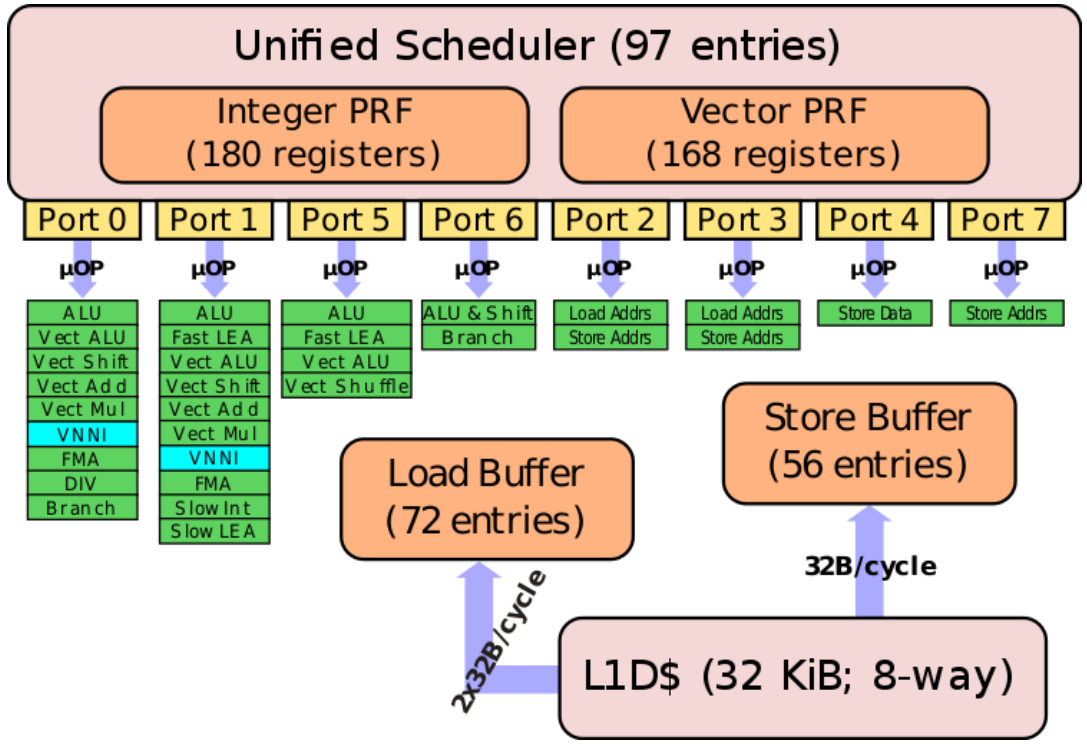
Architecture knowledge

- Understanding of the systems architectural capabilities: everything is complex

Memory



CPU design



Src.: https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake

Frequently used Metrics

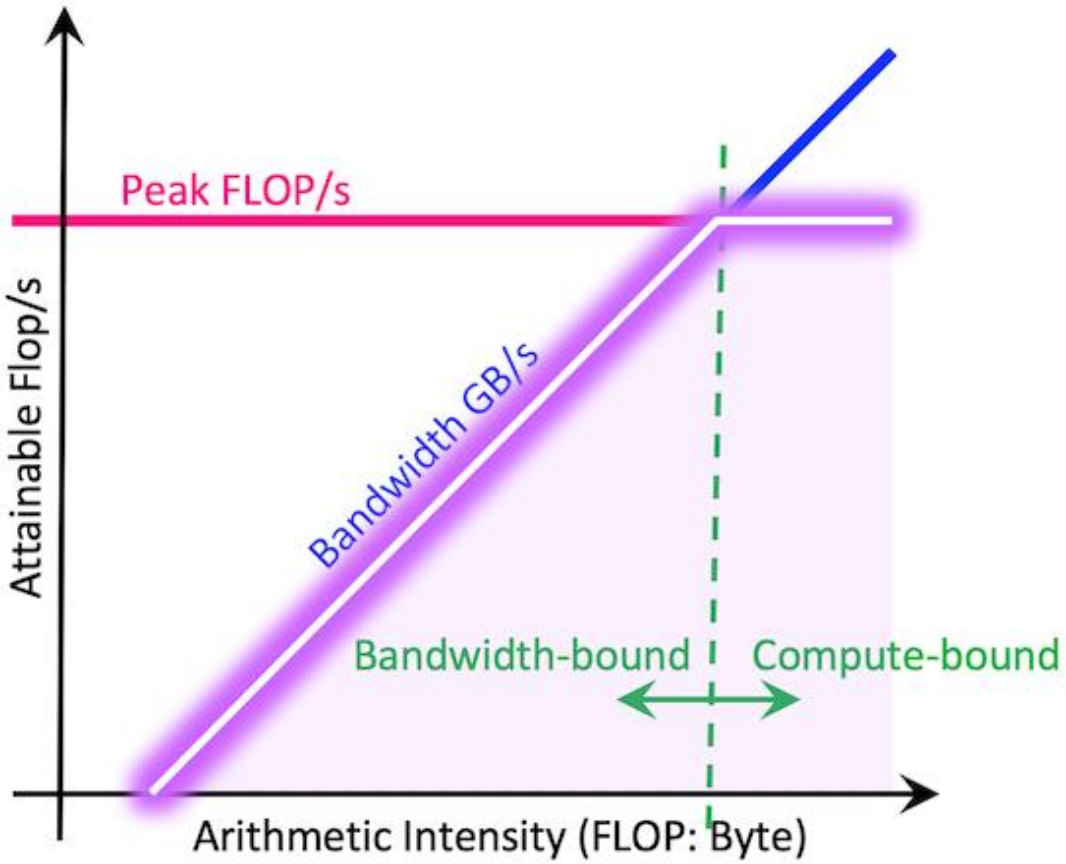
- CPU
  - Cache hits & misses
  - FLOPS (Floating-point Operations Per Second)
  - Branch misses
  - …
- Memory:
  - Memory bandwidth & Memory latency
  - Aim to saturate memory bandwith
  - …
- Network:
  - Network bandwidth & Network latency

```
================================================
 PAPI Preset Events
================================================
   Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  Yes   No   Requests for a snoop
PAPI_CA_SHR  0x8000000a  Yes   No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  Yes   No   Requests for exclusive access to clean cache line
PAPI_CA_INV  0x8000000c  No    No   Requests for cache line invalidation
PAPI_CA_ITV  0x8000000d  Yes   No   Requests for cache line intervention
PAPI_L3_LDM  0x8000000e  Yes   No   Level 3 load misses
PAPI_L3_STM  0x8000000f  No    No   Level 3 store misses
PAPI_BRU_IDL 0x80000010  No    No   Cycles branch units are idle
PAPI_FXU_IDL 0x80000011  No    No   Cycles integer units are idle
PAPI_FPU_IDL 0x80000012  No    No   Cycles floating point units are idle
PAPI_LSU_IDL 0x80000013  No    No   Cycles load/store units are idle
PAPI_TLB_DM  0x80000014  Yes   Yes  Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  Yes   No   Instruction translation lookaside buffer misses
PAPI_TLB_TL  0x80000016  No    No   Total translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  Yes   No   Level 1 load misses
PAPI_L1_STM  0x80000018  Yes   No   Level 1 store misses
PAPI_L2_LDM  0x80000019  Yes   No   Level 2 load misses
PAPI_L2_STM  0x8000001a  Yes   No   Level 2 store misses
PAPI_BTAC_M  0x8000001b  No    No   Branch target address cache misses
PAPI_PRF_DM  0x8000001c  Yes   No   Data prefetch cache misses
PAPI_L3_DCH  0x8000001d  No    No   Level 3 data cache hits
PAPI_TLB_SD  0x8000001e  No    No   Translation lookaside buffer shootdowns
PAPI_CSR_FAL 0x8000001f  No    No   Failed store conditional instructions
PAPI_CSR_SUC 0x80000020  No    No   Successful store conditional instructions
PAPI_CSR_TOT 0x80000021  No    No   Total store conditional instructions
PAPI_MEM_SCY 0x80000022  No    No   Cycles Stalled Waiting for memory accesses
PAPI_MEM_RCY 0x80000023  No    No   Cycles Stalled Waiting for memory Reads
PAPI_MEM_WCY 0x80000024  Yes   No   Cycles Stalled Waiting for memory writes
….
```

Roof-line model

- Lay-mans summary:

  Performance limited either by
  - compute or
  - memory
- Arithmetic intensity $\frac{Work}{Byte}$
- Goal: Get to the ridge-line
  - Move to the right: do more work with the same data
  - Move to the left: do less work with the same data



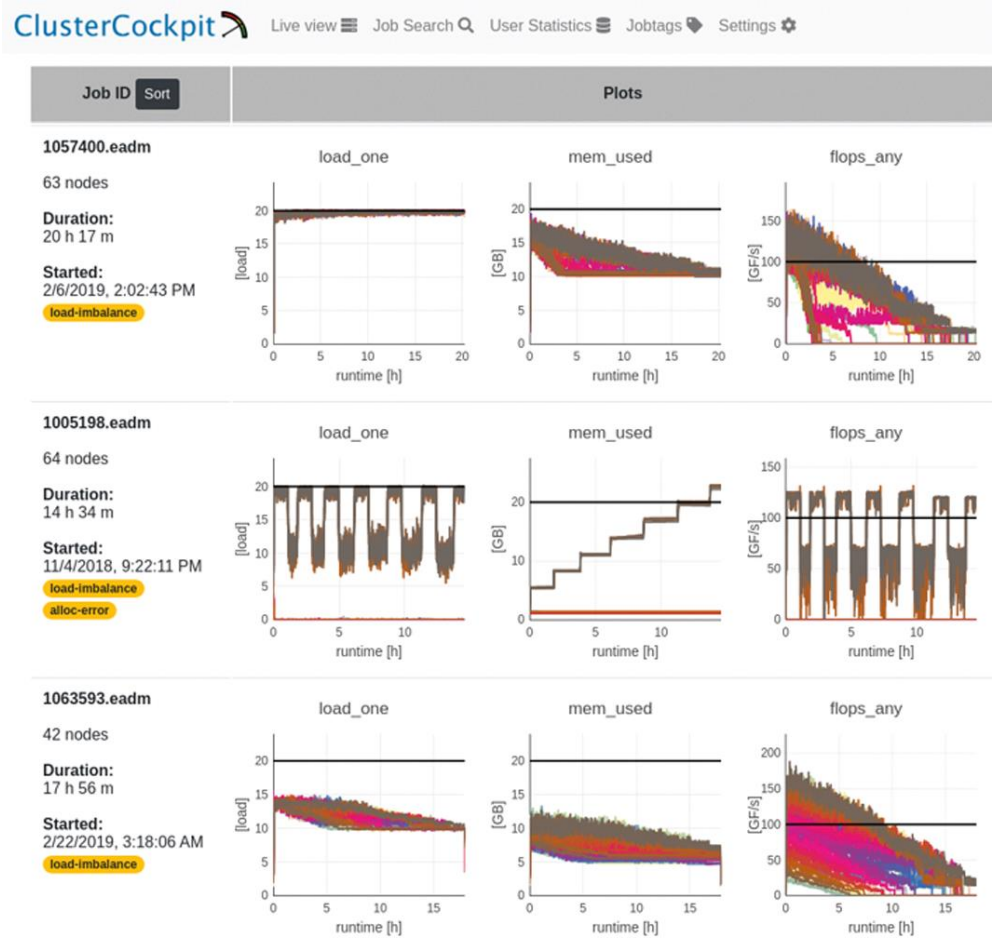
Src. https://docs.nersc.gov/tools/performance/roofline/

Applies to a steady-state

Understanding and Context

- Metrics gatherd for a „whole run" have little expressiveness
- Context during which a metric was gathered
- Correlate change in metric to specific code regions

- Requirements:
  - How it „should" run
    What the hardware can do
  - Observe actual execution



Src: https://www.rrze.fau.de/2020/01/webanwendung-zum-jobspezifischen-performance-monitoring-eine-erfolgsgeschichte-keine-ressourcen-verschwendung-gezielte-optimierung/

Measurement Techniques

- Process isolation prevents direct observation of a programs state
  - Modify the target program before execution
  - Use a loader to gain control of the application
    - Keyword: LD_PRELOAD mechanism
- Terminology:
  - **Hook**: Facility to potentially measure
  - **Probe**: A measurement device, connecting to a hook
  - **Measurement**: Gathering of data and context

- Resolving the temporal state of the program
  1) Sampling
  2) Instrumentation

SAMPLING

Sampling

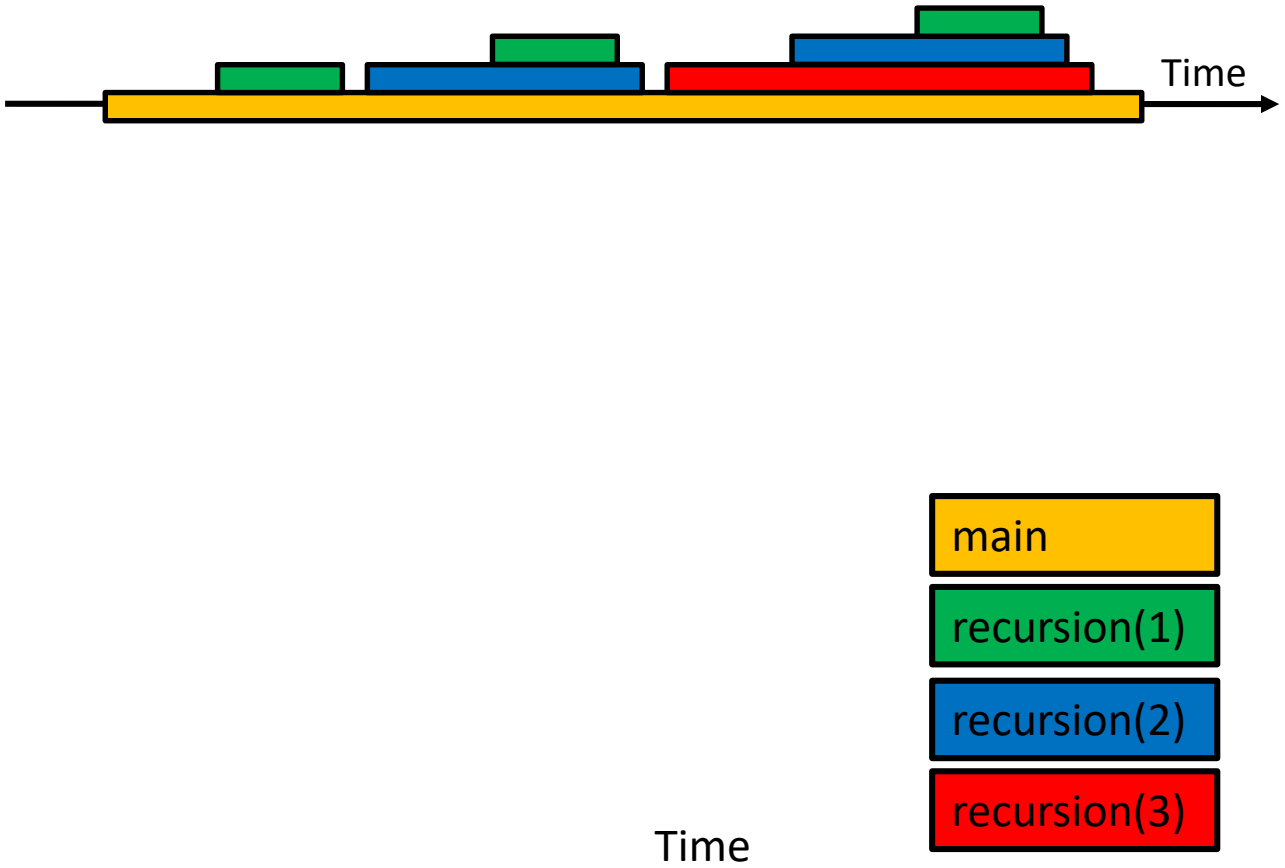
- Taking measurements at regular (or irregular) intervals
- Basic approach:

```
1) gain access to the target programs process
2) program an time or hardware-counter based event generator
3) for each event:
4)    identify current location
5)    create a measurement
6)    process and store data
7)    reset event generator
```

Demonstrator

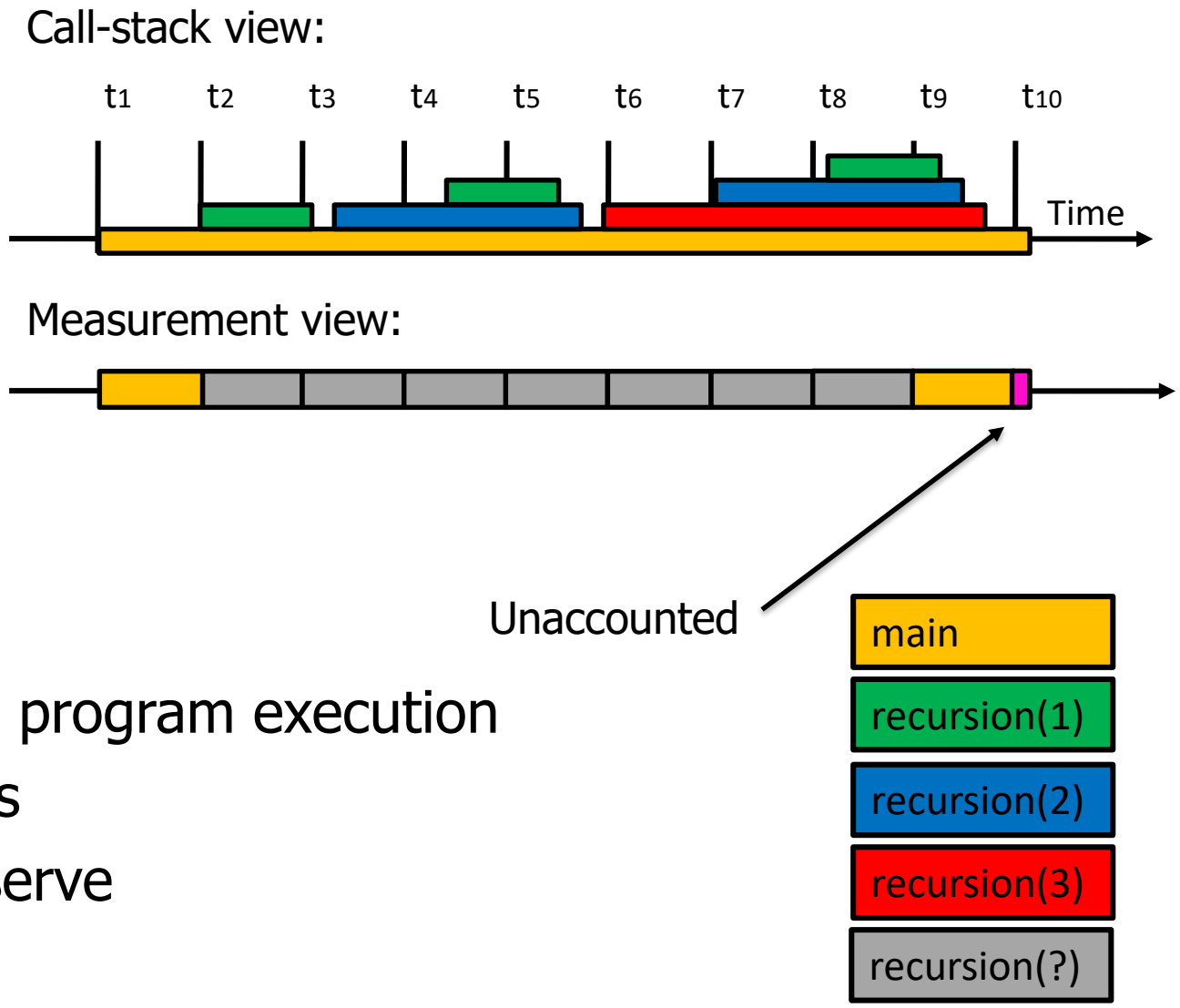
```
void recursion(int depth){
    if (depth>1)
        return recursion(depth-1);
    return;
}

int main(int argc, char ** argv){
    for (int depth=1;depth<4;depth++)
    {
        recursion(depth);
    }
}
```

Call-stack view:



Basic sampling

```
void recursion(int depth){
    if (depth>1)
        return recursion(depth-1);
    return;
}

int main(int argc, char ** argv){
    for (int depth=1;depth<4;depth++)
    {
        recursion(depth);
    }
}
```

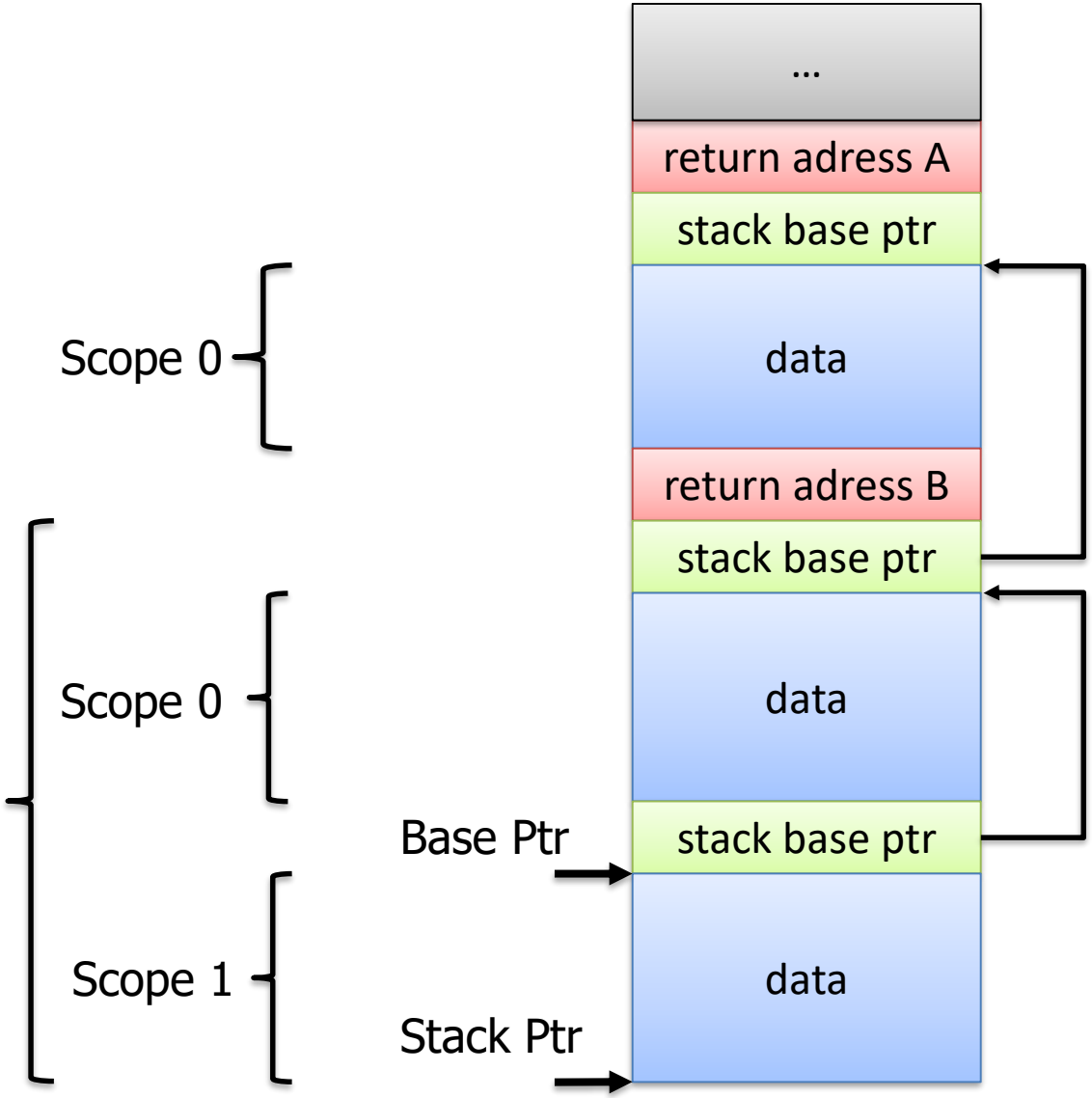


- Sampling „statistically" observes the program execution
- Not guaranteed to observe all details
- Frequency: 2x smallest event to observe

Unwinding

- Unwinding: analysis of the call-stack to identify the call-path to the current function
- Basic approach:
  - Analyze entries before stack base-pointer
  - Exists an return-adress and does it point to a viable source-position?
  - Repeat

```
void some_function(){
    int scope;
    {
        int scope;
    }
}
```



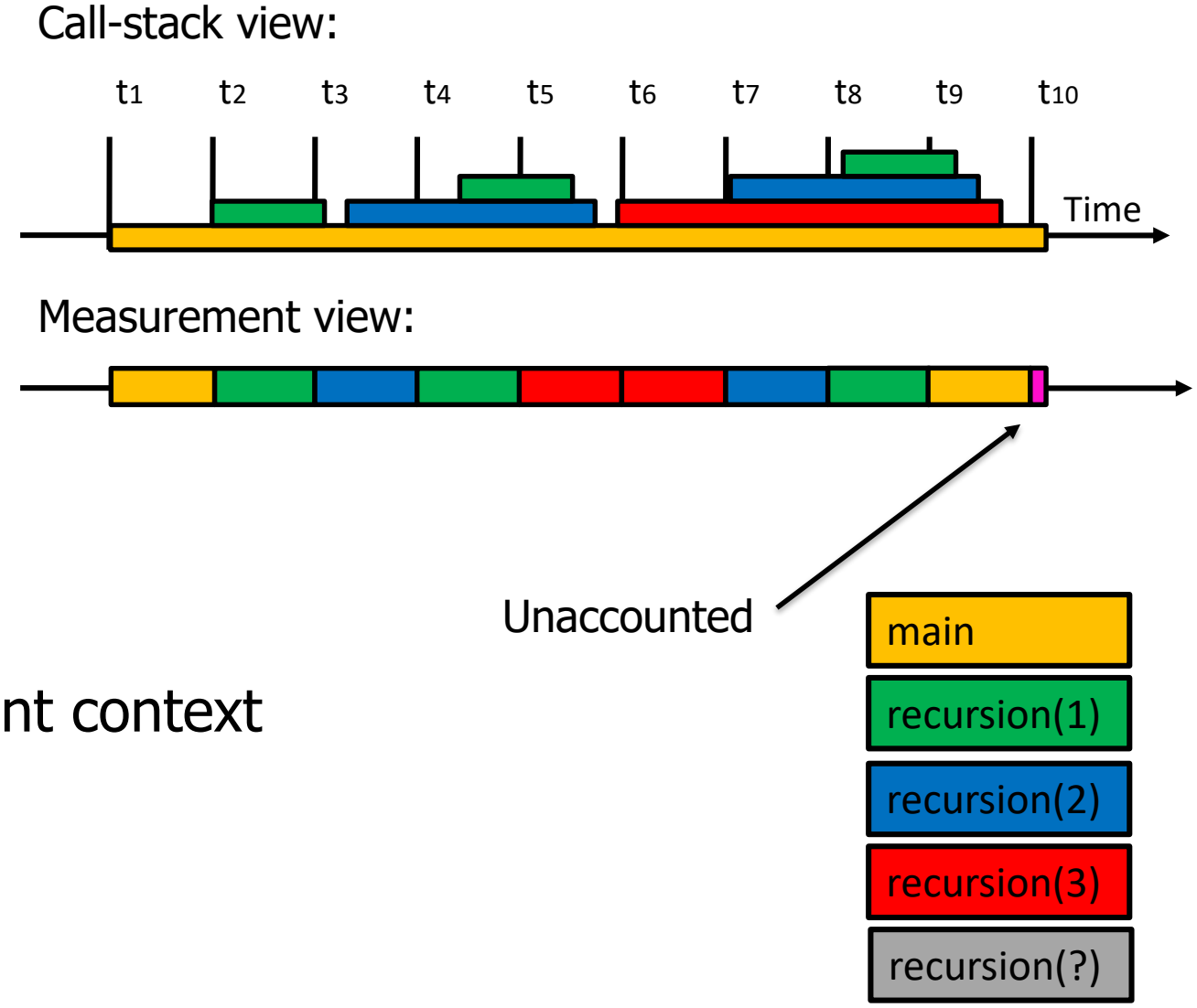
Sampling (revised)

- Taking measurements at regular (or irregular) intervals
- Revised approach, with unwinding

```
1) gain access to the target programs process
2) program an time or hardware-counter based event generator
3) for each event:
4)     identify current location
5)     obtain call-context, e.g. via unwinding
6)     create a measurement
7)     process and store data
8)     reset event generator
```

Sampling with unwinding

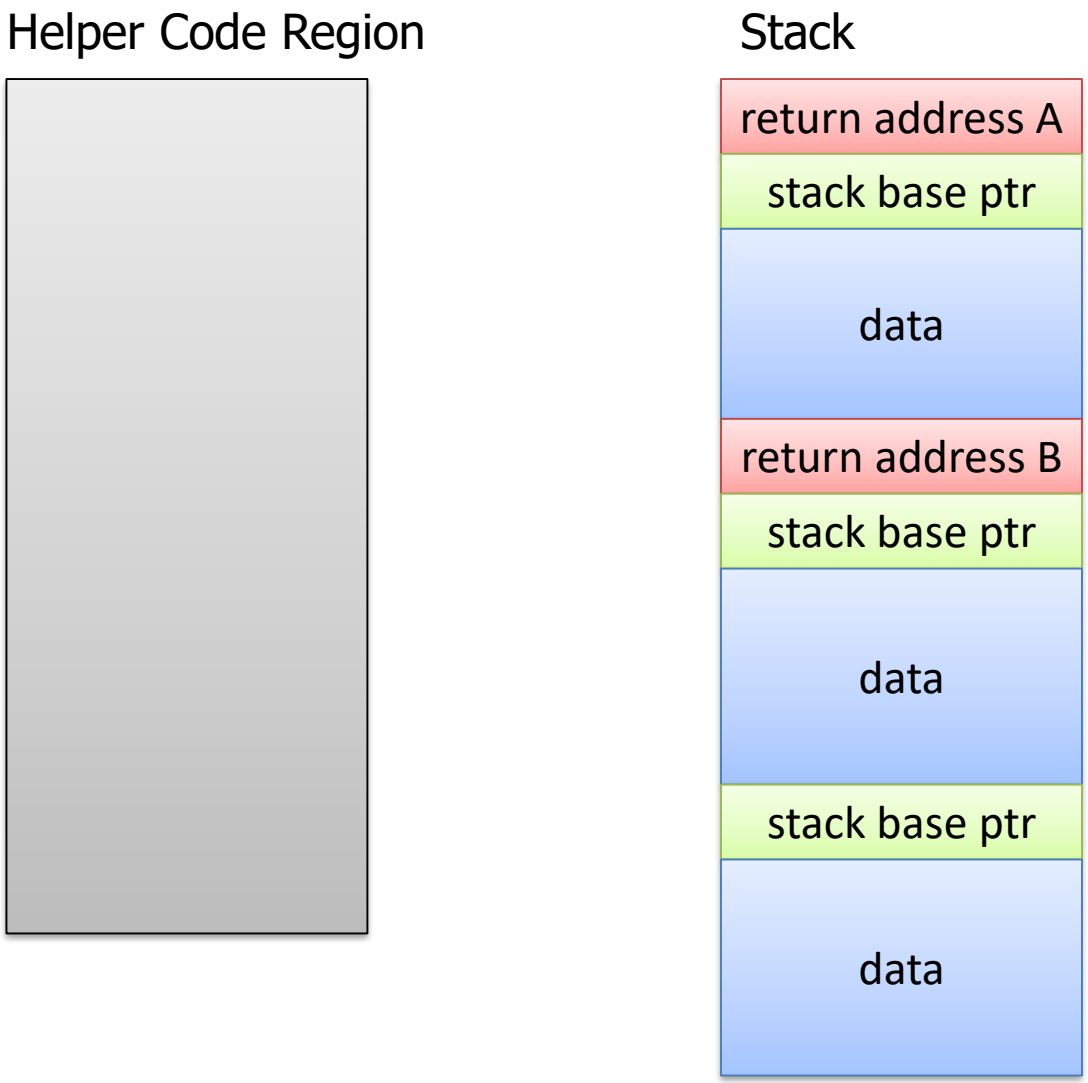
```
void recursion(int depth){
    if (depth>1)
        return recursion(depth-1);
    return;
}

int main(int argc, char ** argv){
    for (int depth=1;depth<4;depth++)
    {
        recursion(depth);
    }
}
```



- Unwinding resolves most of the event context
- End still unaccounted

Trunks and Thunks

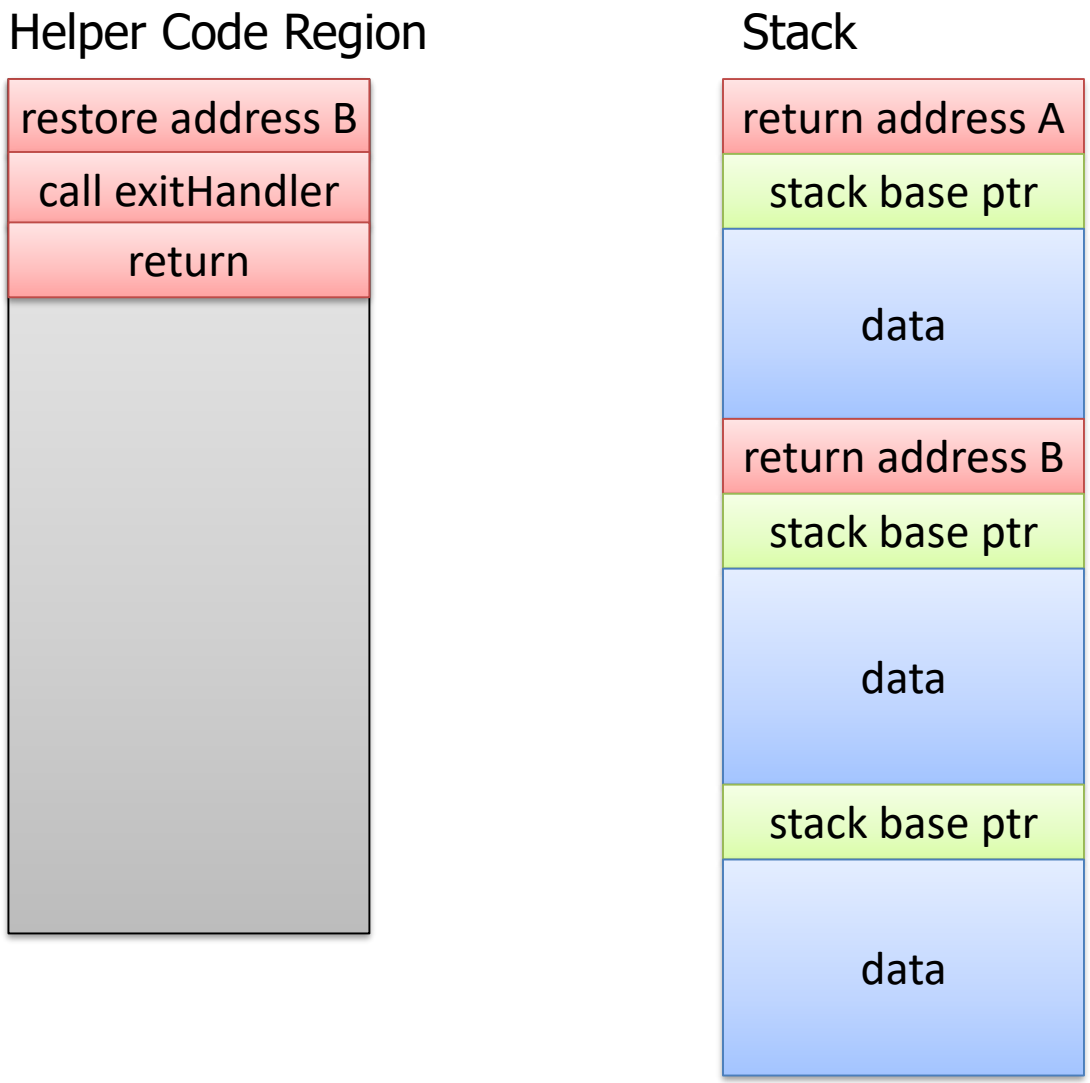
- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler



DOI: 0.1109/IPDPS.2011.67

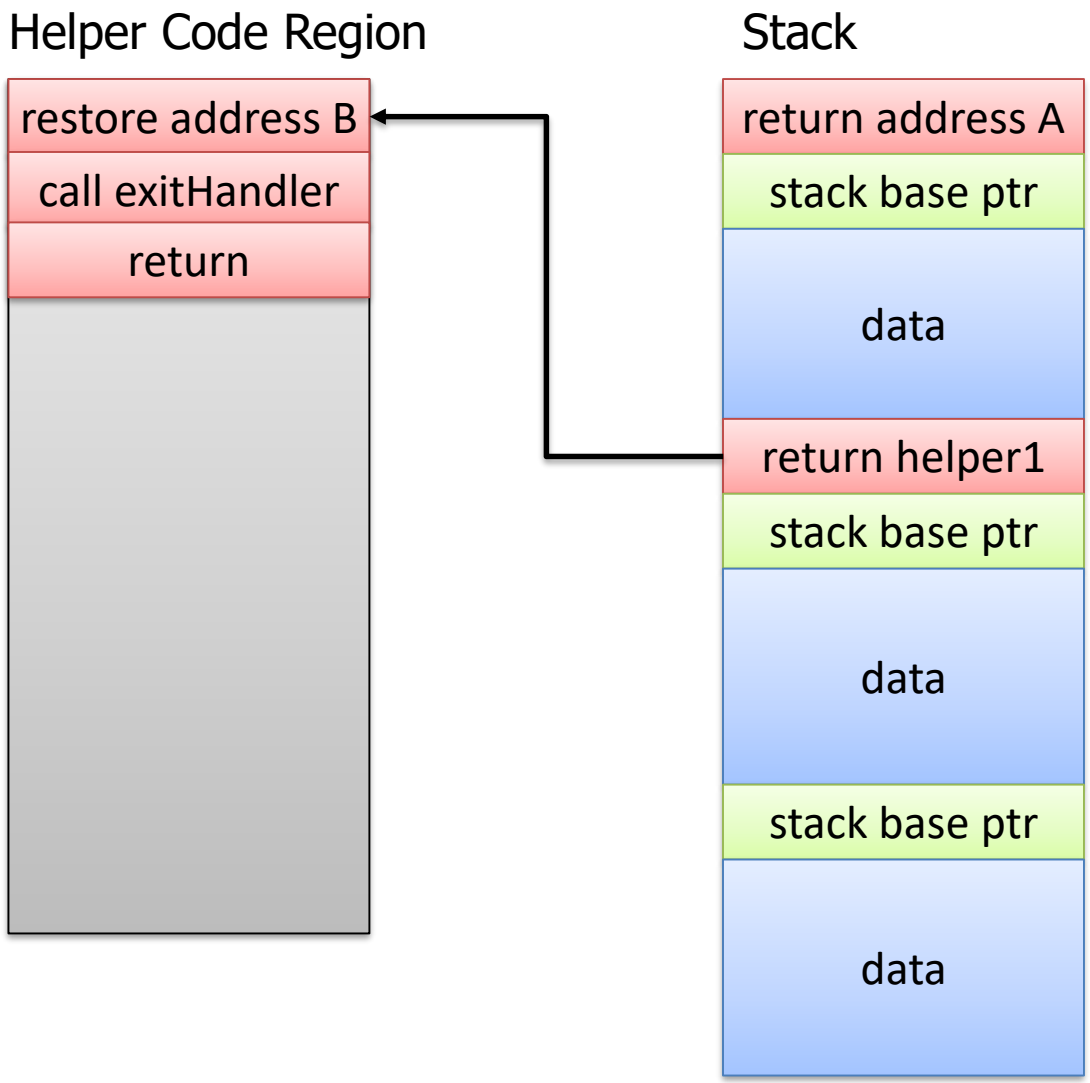
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler



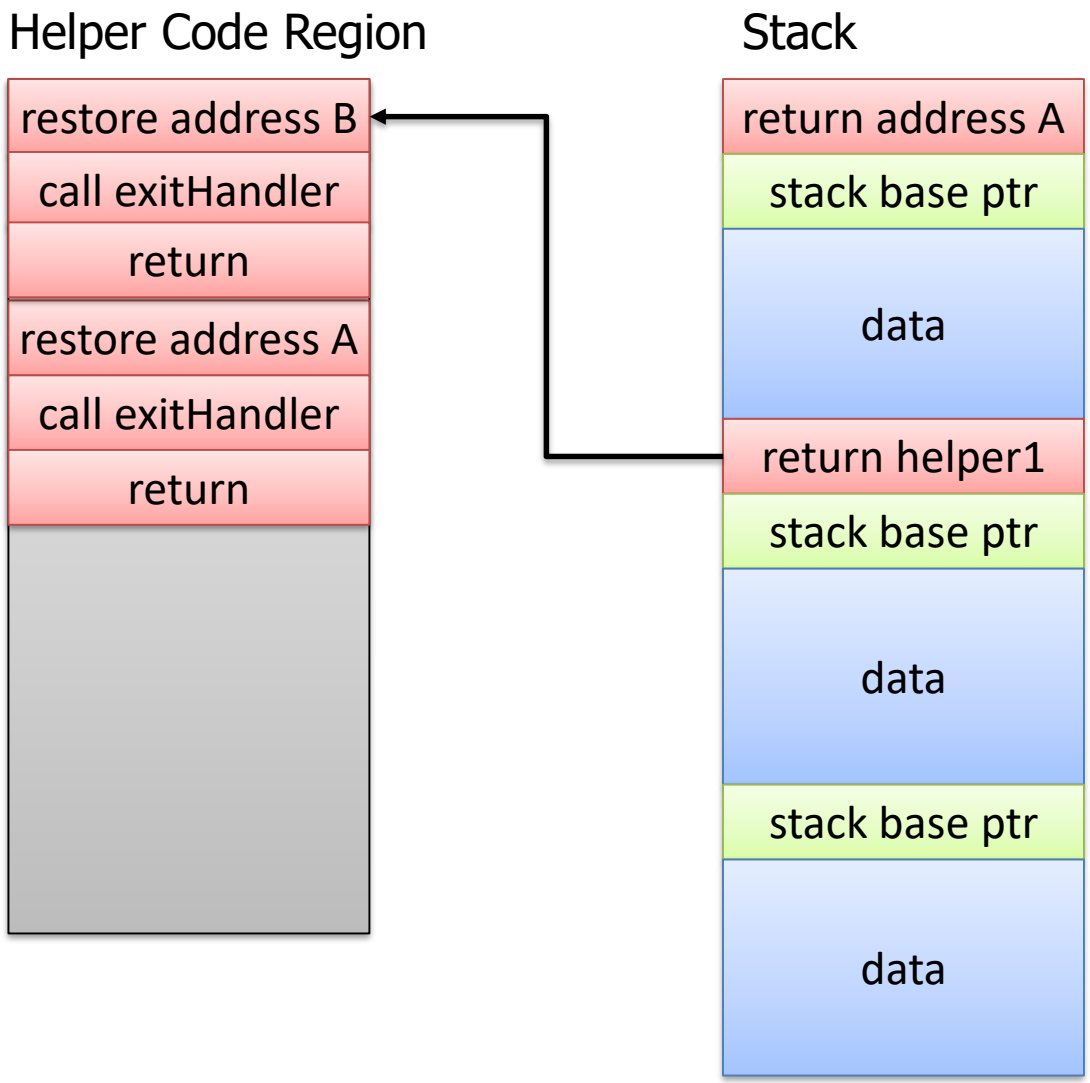
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler



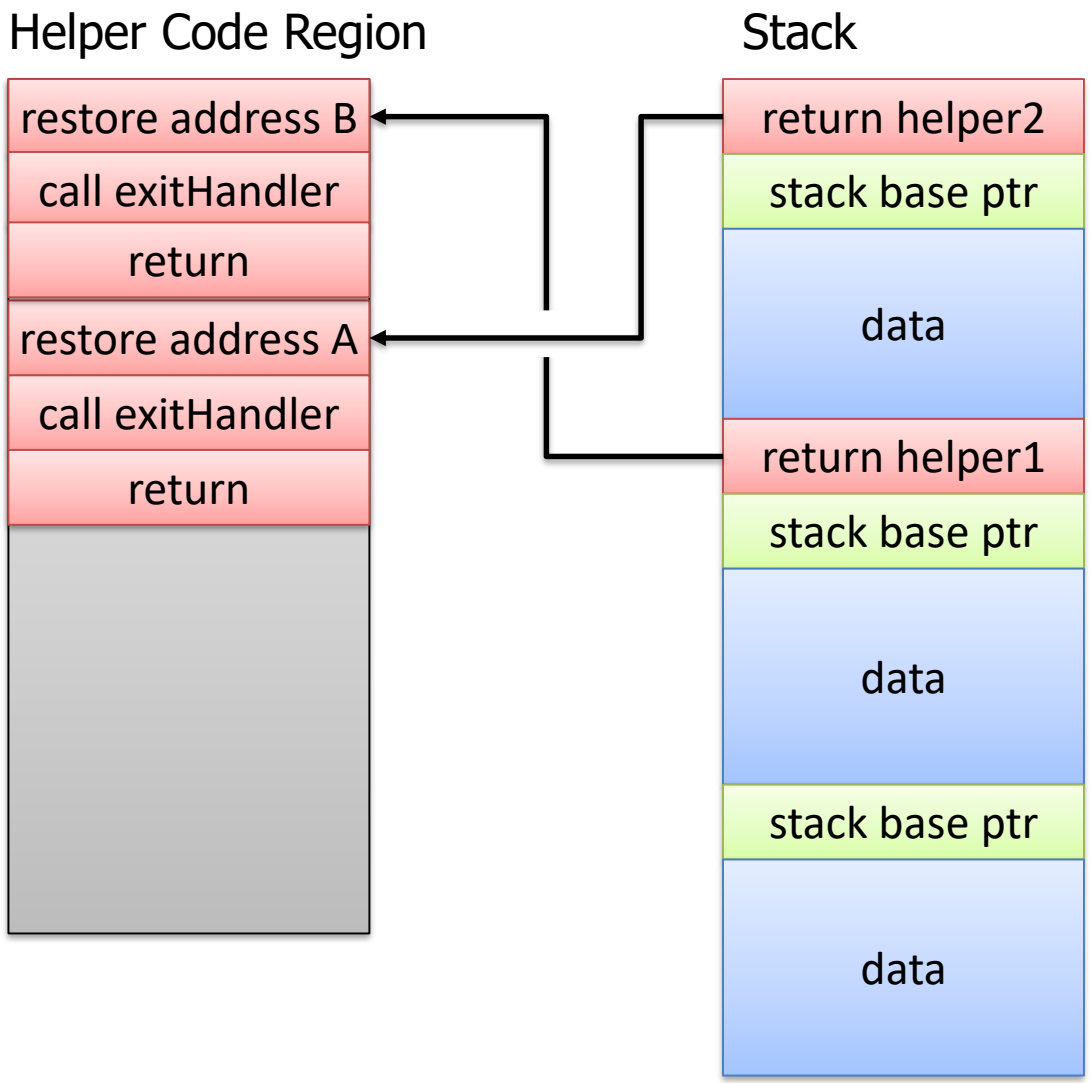
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler



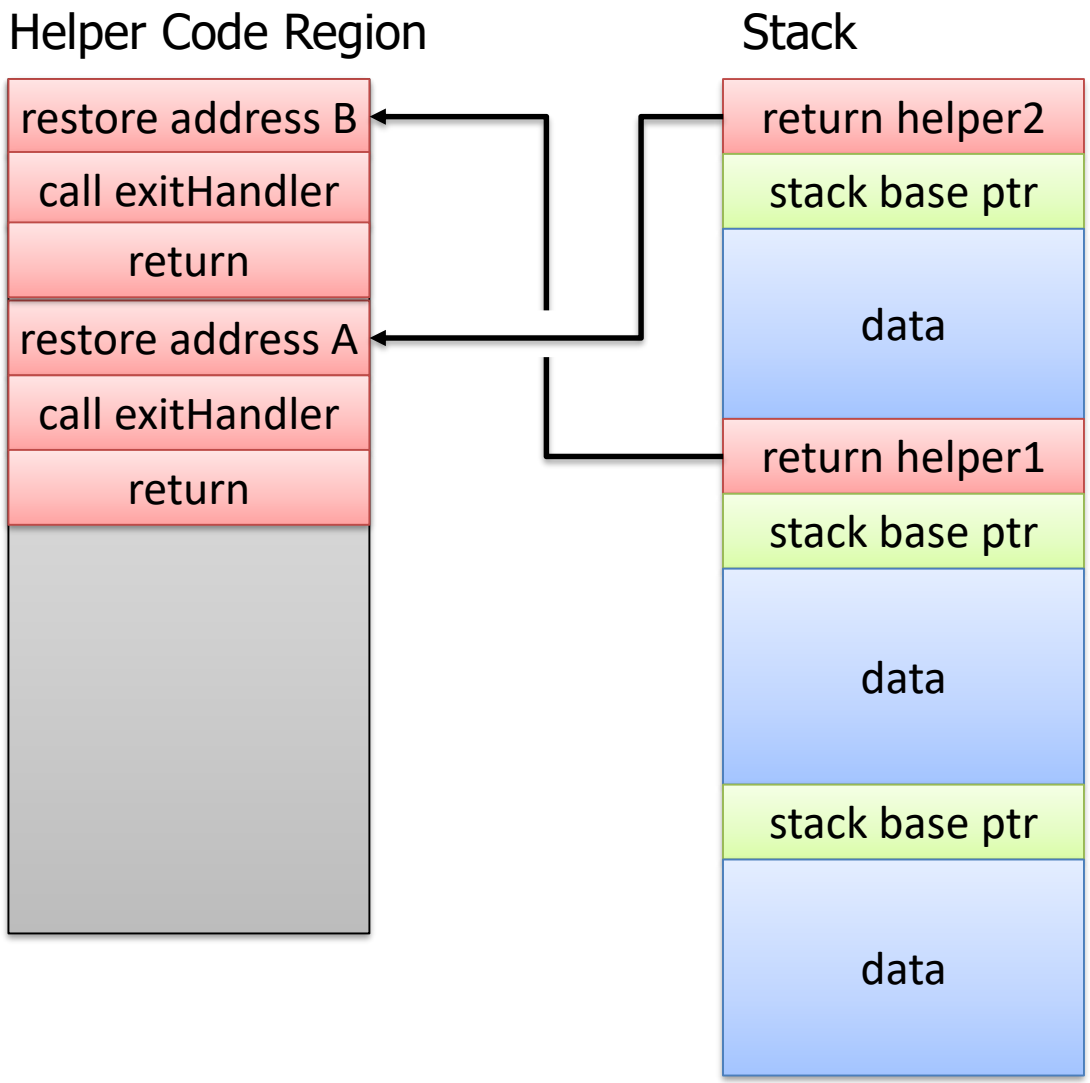
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler



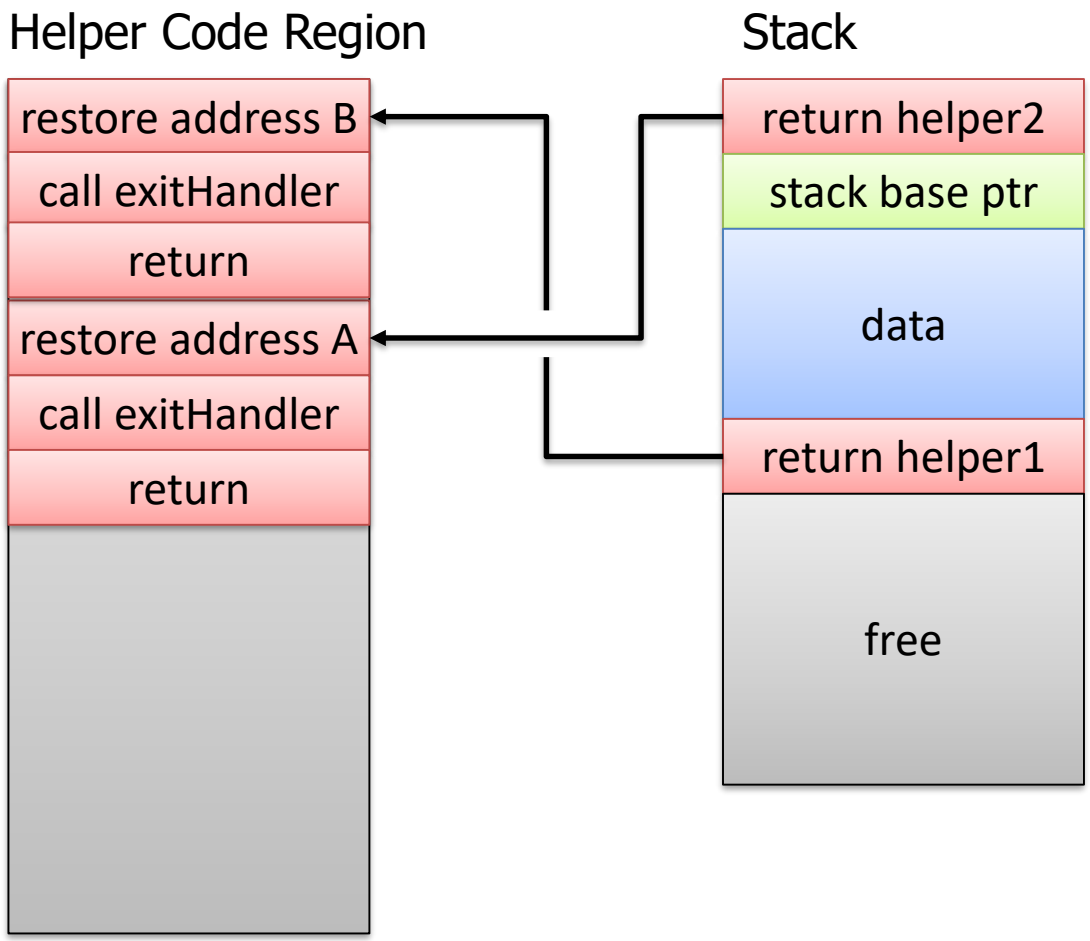
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



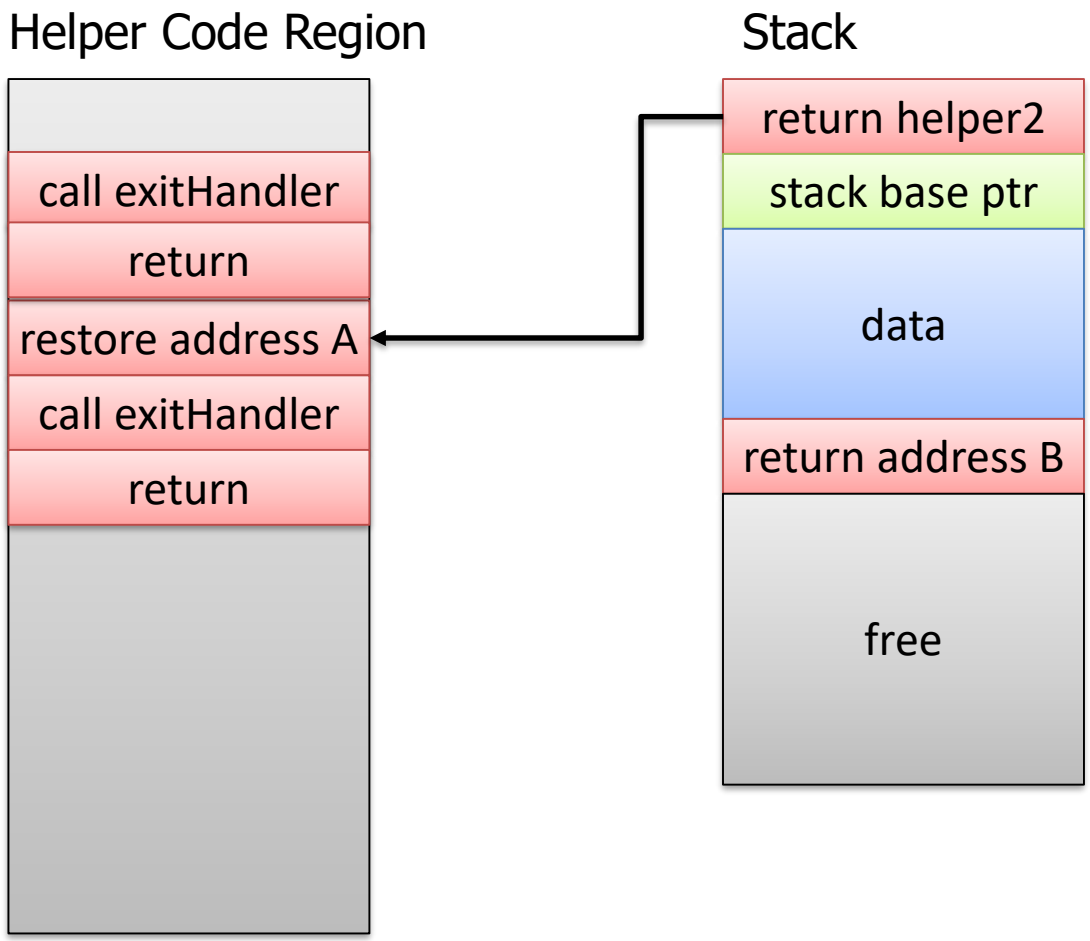
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



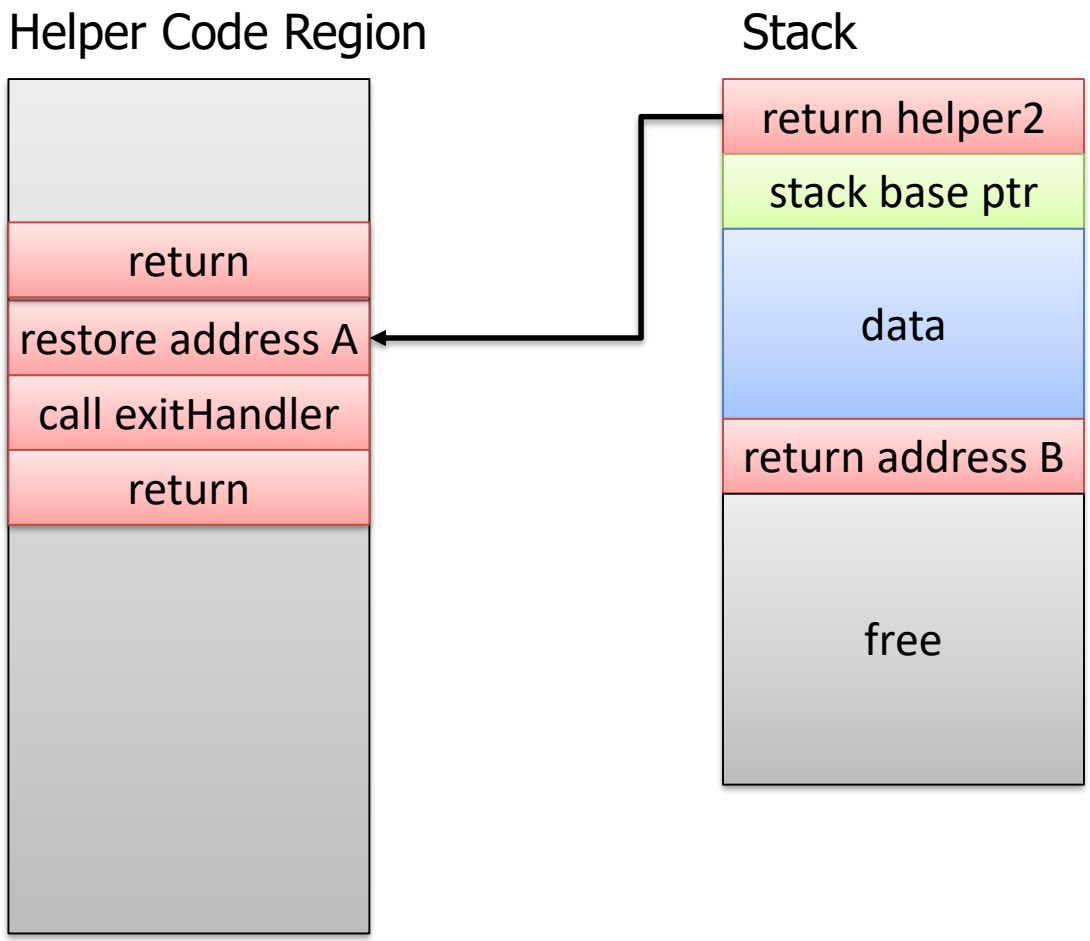
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



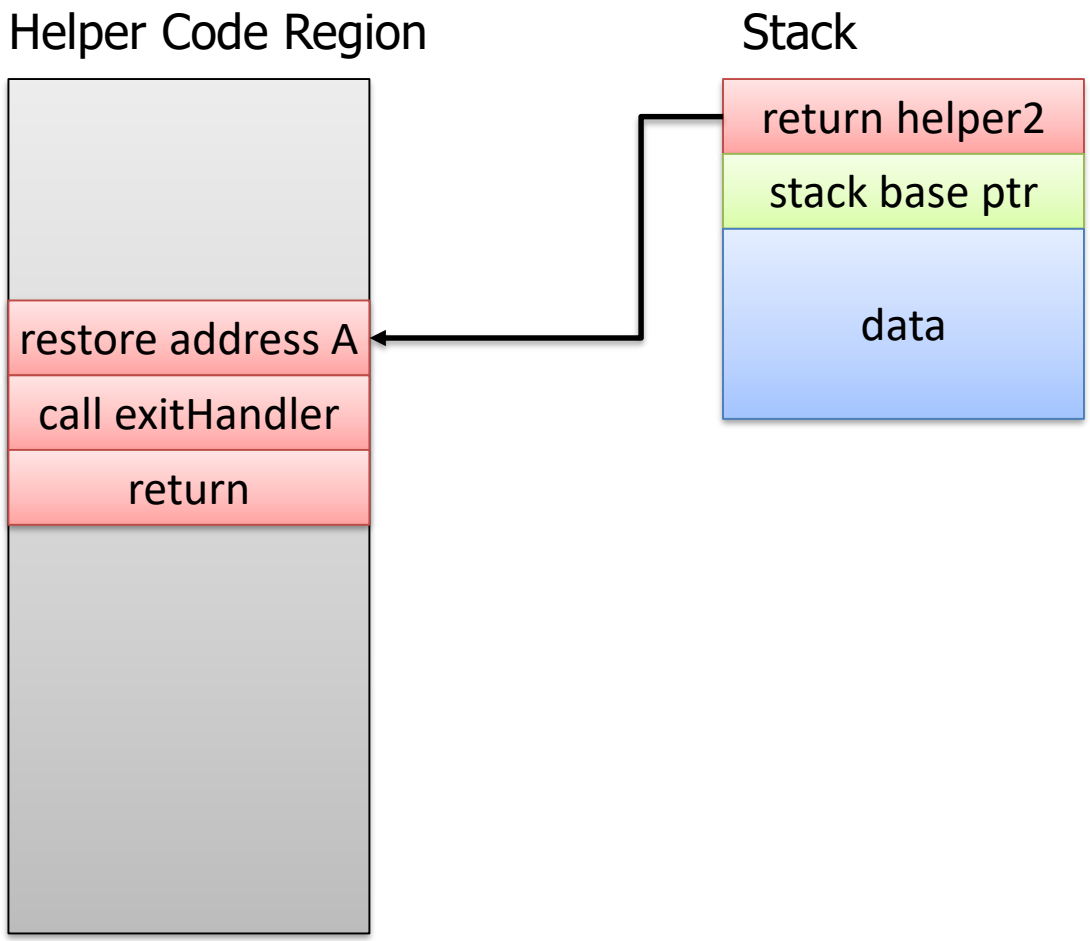
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



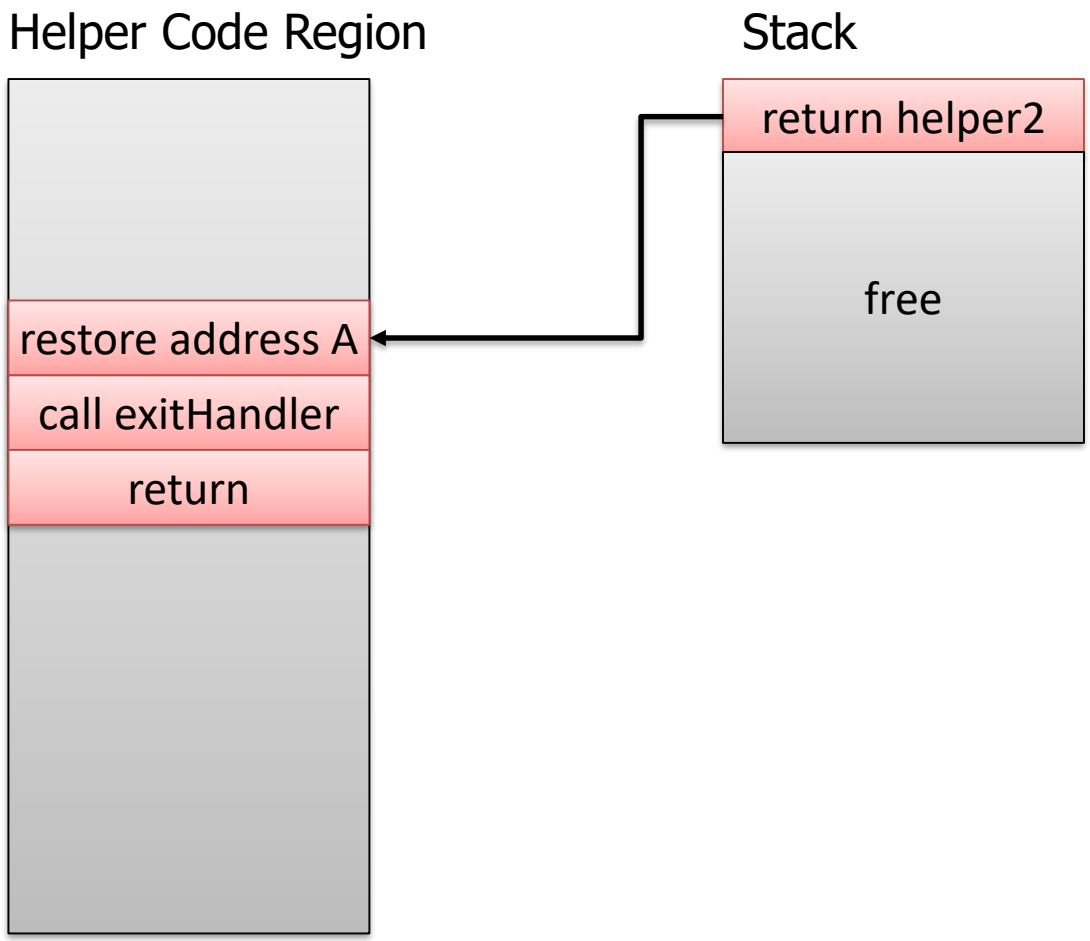
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



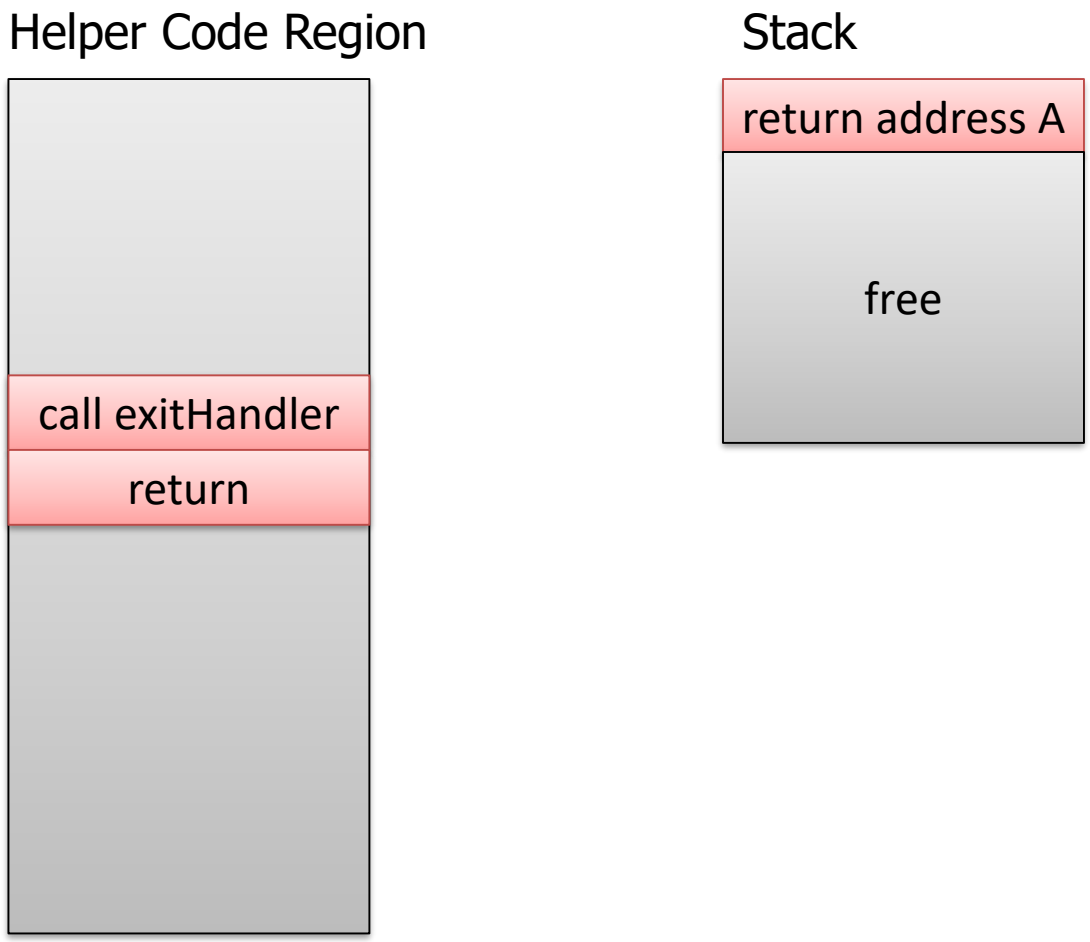
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



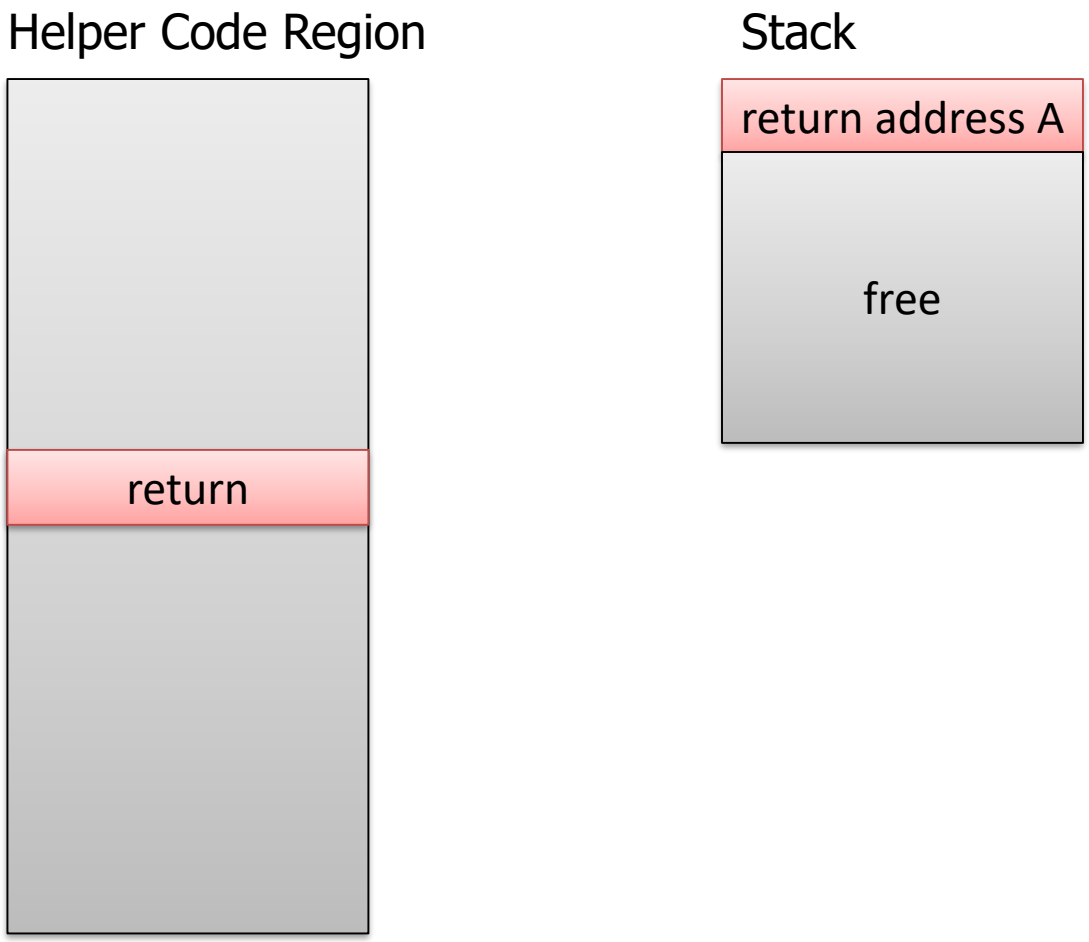
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



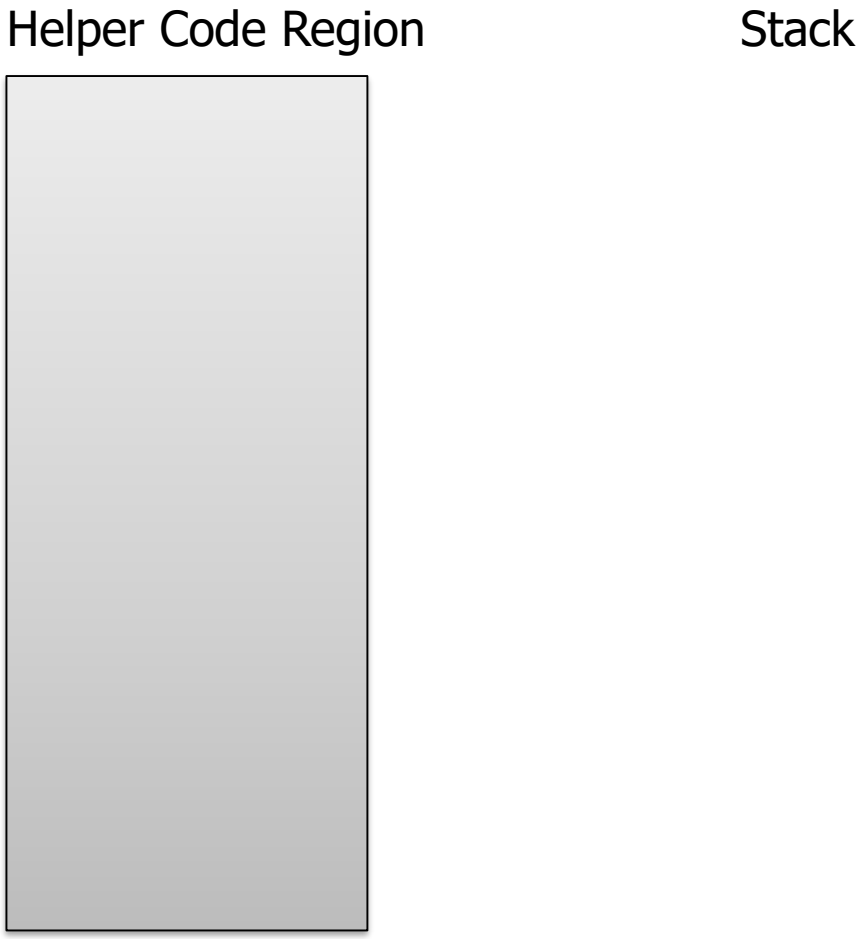
Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



Trunks and Thunks

- When unwinding a functions stack frame for the first time:
  - Preserve current return-address
  - Add call to exitHandler
  - Modify return address to point to an function exit handler
- Function exit handler logs exit, performs measurement, and returns to original



Sampling (revised, thunks)

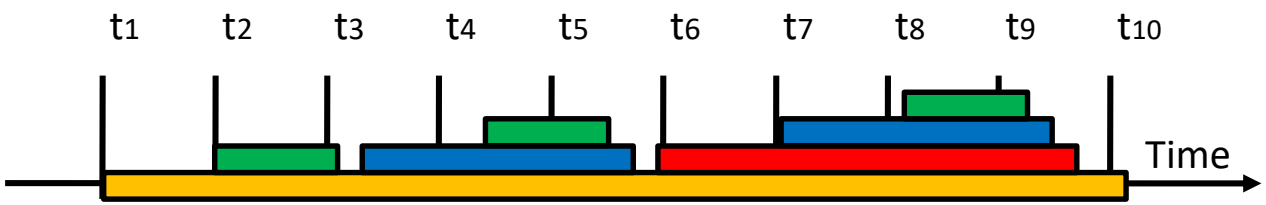
- Taking measurements at regular (or irregular) intervals
- Revised approach, with unwinding and thunks/trunks

```
1) gain access to the target programs process
2) program an time or hardware-counter based event generator
3) for each event:
4)     identify current location
5)     obtain call-context, e.g. via unwinding
6)     if no thunk/thrunk
7)         establish thunk
8)         obtain call-context, e.g. via unwinding
9)     create a measurement
10)    process and store data
11)    reset event generator
```

Sampling with unwinding and return-address tracking

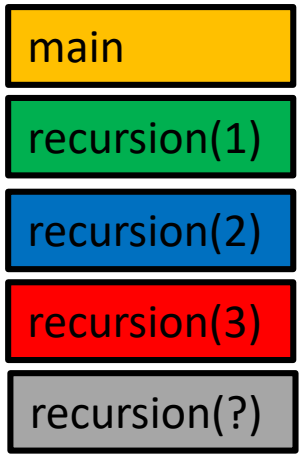
```
void recursion(int depth){
    if (depth>1)
        return recursion(depth-1);
    return;
}

int main(int argc, char ** argv){
    for (int depth=1;depth<4;depth++)
    {
        recursion(depth);
    }
}
```

Call-stack view:



Measurement view:

- „Thunking/Trunking“ allows to accuarately track function exits



INSTRUMENTATION

Instrumentation principle

- Example: Function Instrumentation

```
int main(){
  ... some code ...
  return callA();
}
```

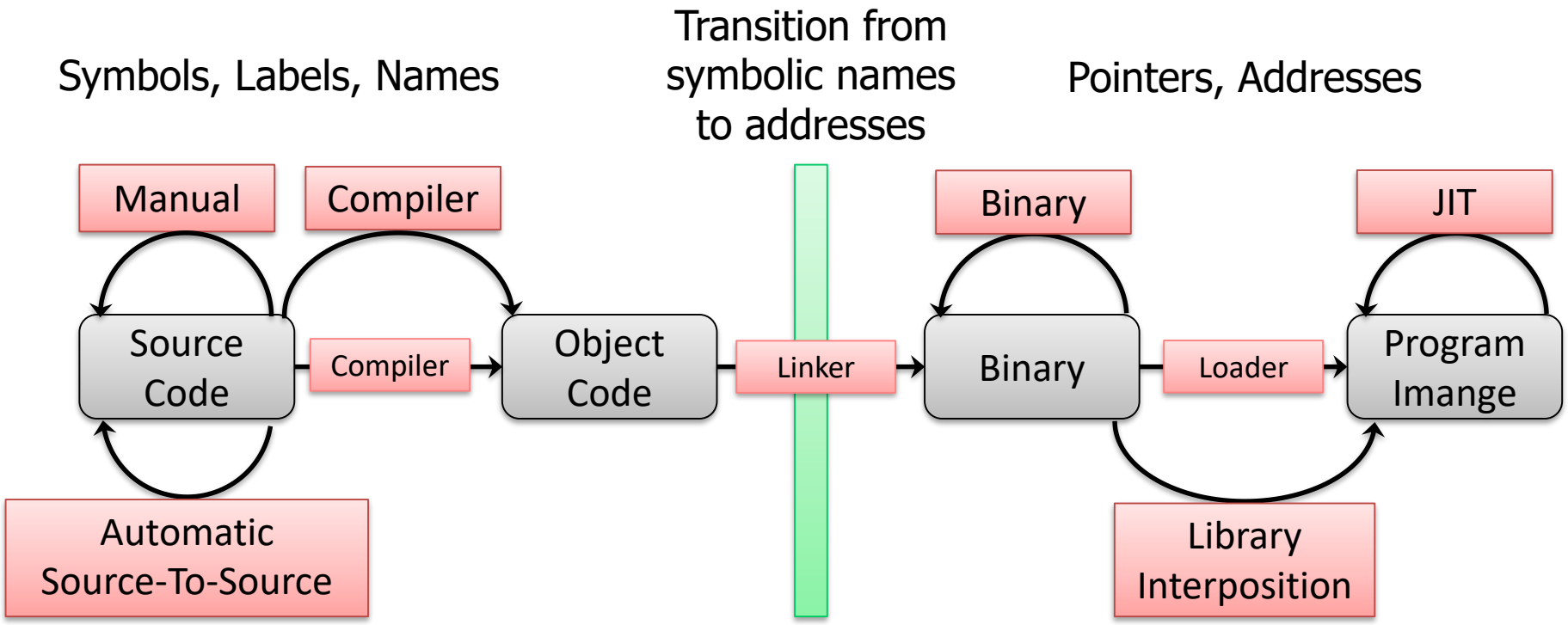
Function Instrumentation

```
int main(){
  startMeasurement("main");
  ... some code ...
  auto retVal = callA();
  stopMeasurement("main");
  return retVal;
}
```

```
void startMeasurement(char * name){
  auto time=getTime();
  logTime(START, time, name);
}

void stopMeasurement(char * name){
  auto time = getTime();
  logTime(STOP, time, name);
}
```

- **Other instrumentation sites of interest**

Instrumentation Opportunities

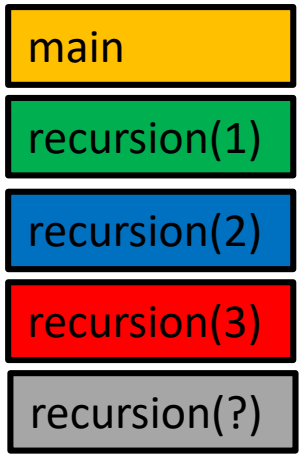


Basic Instrumentation – Variant A

```
void recursion(int depth){
    if (depth>1)
        return recursion(depth-1);
    return;
}

int main(int argc, char ** argv){
    for (int depth=1;depth<4;depth++)
    {
        recursion(depth);
    }
}
```

Call-stack view:

Measurement view:



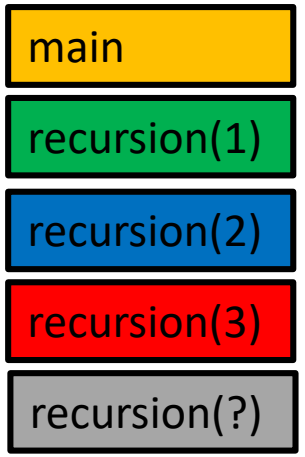
Instrumentation before optimization

Basic Instrumentation – Variant B

```
void recursion(int depth){
    if (depth>1)
        return recursion(depth-1);
    return;
}

int main(int argc, char ** argv){
    for (int depth=1;depth<4;depth++)
    {
        recursion(depth);
    }
}
```

Call-stack view:

Measurement view:



Instrumentation after optimization:

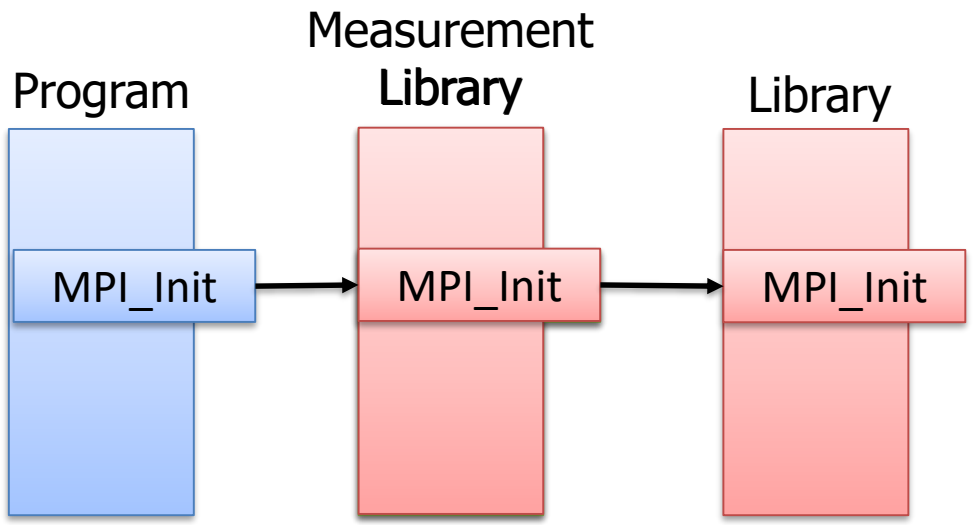
- Inlining removed function calls

Manual Instrumentation / PreInstrumented Libraries

- Manual instrumentation
  - Using specific interface, add hooks
  - Easy to forget entry/exit convention or miss them
  - Code-rewrite necessary
- Libraries may offer explicit profiling interface
  - Similar to manual instrumentation
- Library interposition
  - aka library-wrapping
  - Custom measurement library with identical interface
  - Internal forwarding of calls to original variant
  - For dynamic libraries, no recompilation is necessary

Manual Instrumentation / PreInstrumented Libraries

- Manual instrumentation
  - Using specific interface, add hooks
  - Easy to forget entry/exit convention or miss them
  - Code-rewrite necessary
- Libraries may offer explicit profiling interface
  - Similar to manual instrumentation
- Library interposition
  - aka library-wrapping
  - Custom measurement library with identical interface
  - Internal forwarding of calls to original variant
  - For dynamic libraries, no recompilation is necessary



Compiler Instrumentation

- Compiler add hooks at predifined locations:
  - Function-instrumentation
  - **Critical: adding hooks alters optimization decision by the compiler**
  - **Critical: adding hooks before or after optimizations**
  - Hook example (GNU):

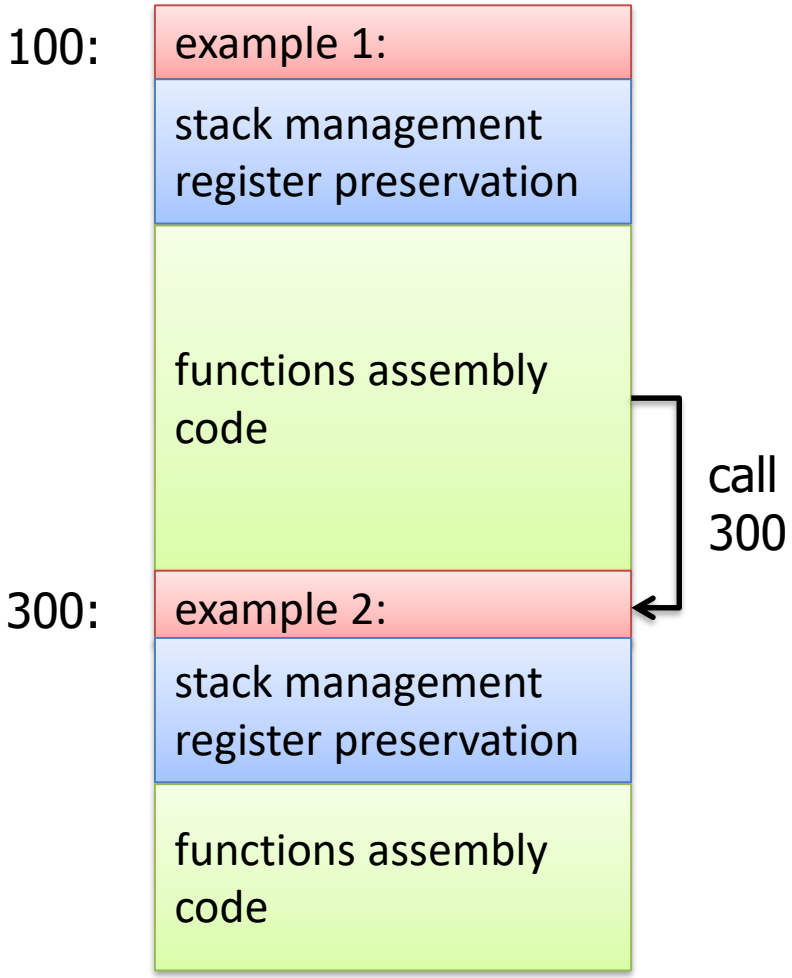
```
void __cyg_profile_func_enter(void * fn,void * call_site);
void __cyg_profile_func_exit (void * fn,void * call_site);

void callee(){
    __cyg_profile_func_enter(callee,&&callsiteA);
    ...
    __cyg_profile_func_exit(callee,&&callsiteA);
}
void caller(){
callsiteA:
    callee();
}
```

function address

call-site

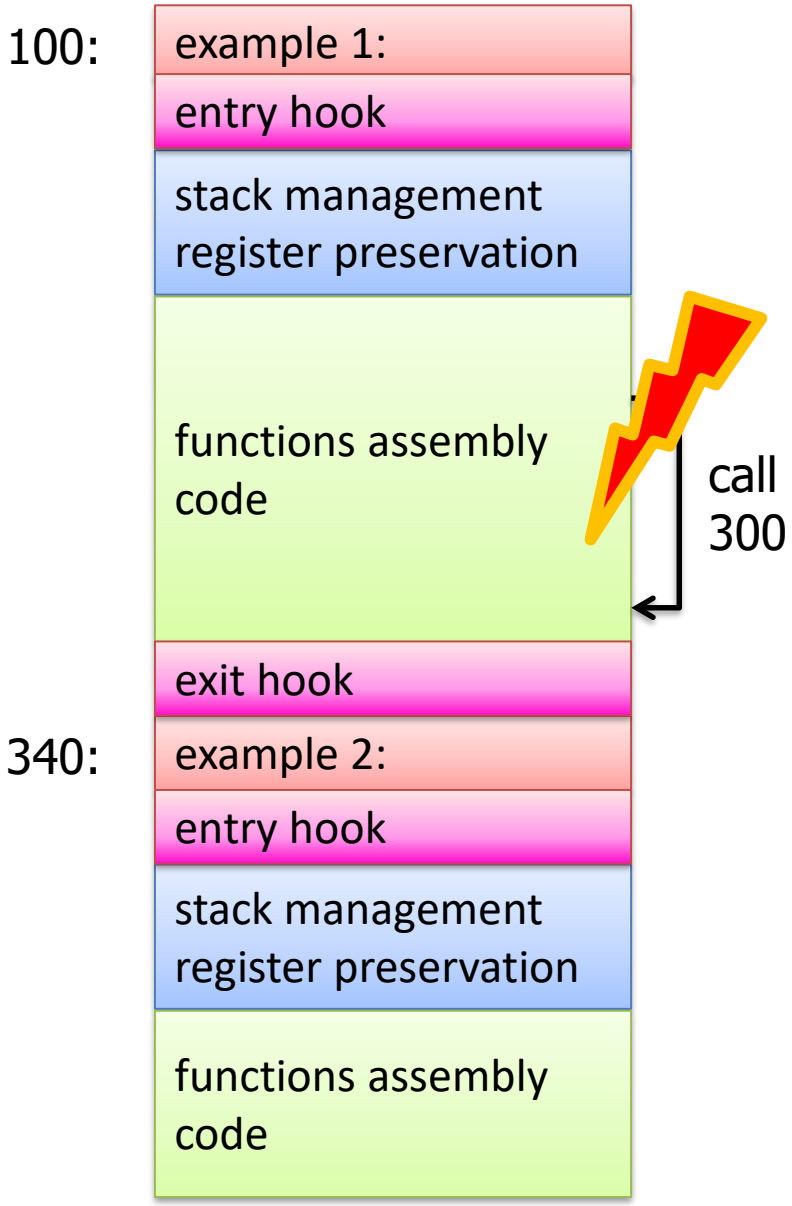
Binary Instrumentation

- Binary is modified to add hooks
- Challenge:
  some assembly calls are relative
- Challenge:
  new architectures with new instructions require update of instrumentation tool
- Invalidate or counteract compiler optimizations



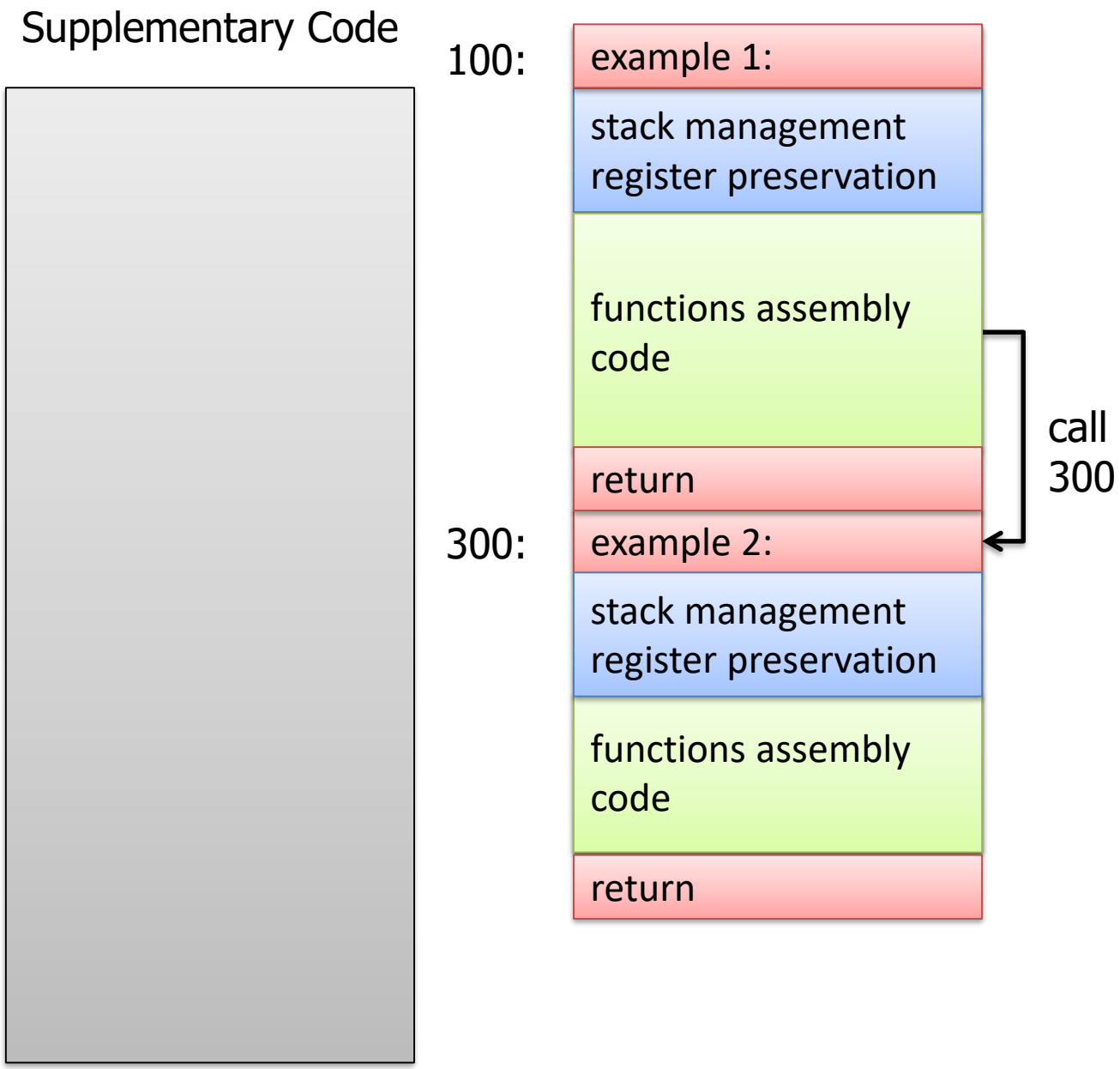
Binary Instrumentation

- Binary is modified to add hooks
- Challenge:
  some assembly calls are relative
- Challenge:
  new architectures with new instructions require update of instrumentation tool
- Invalidate or counteract compiler optimizations



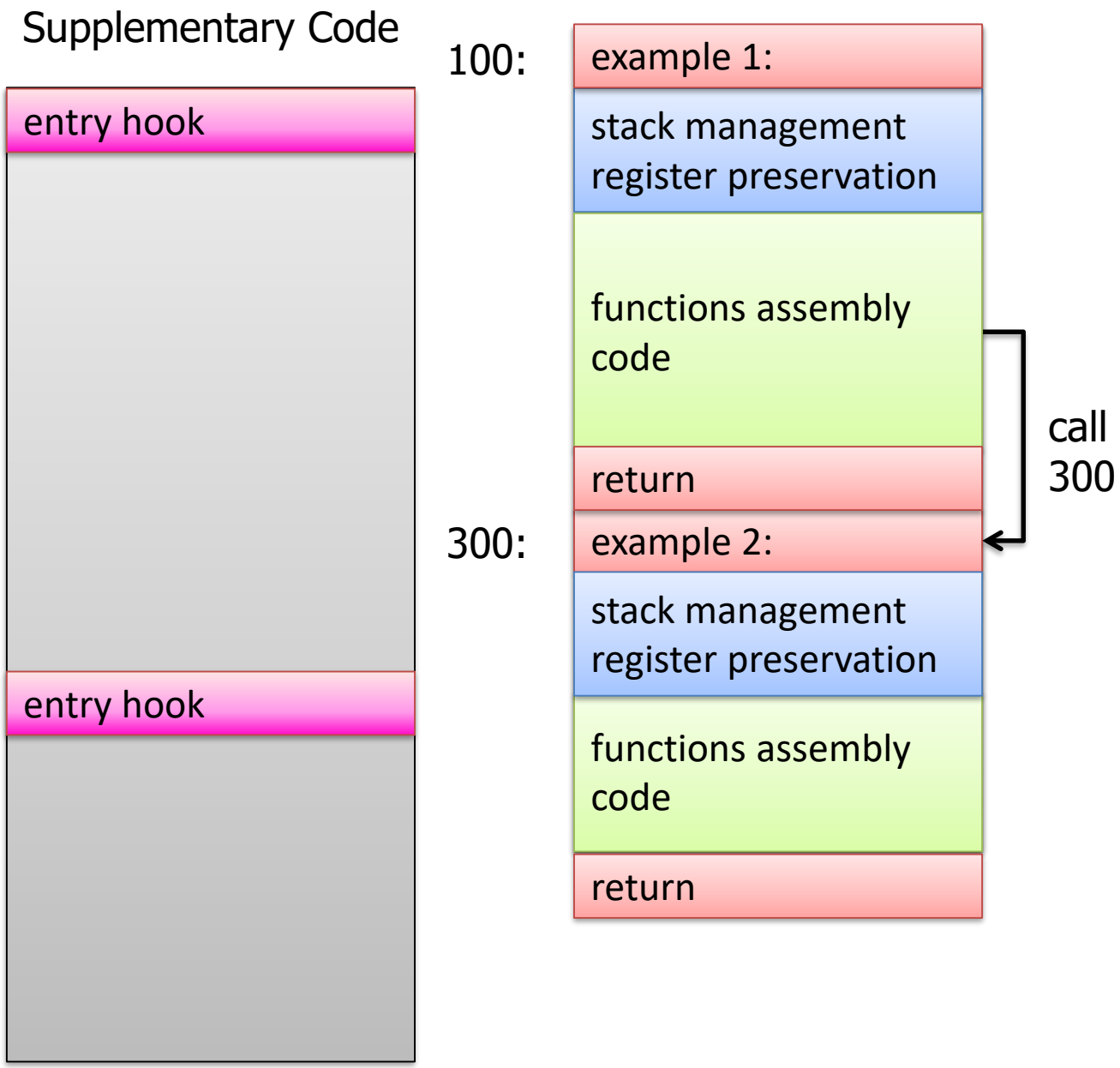
Binary Instrumentation

- Basic approach:
  - Create supplementary code region



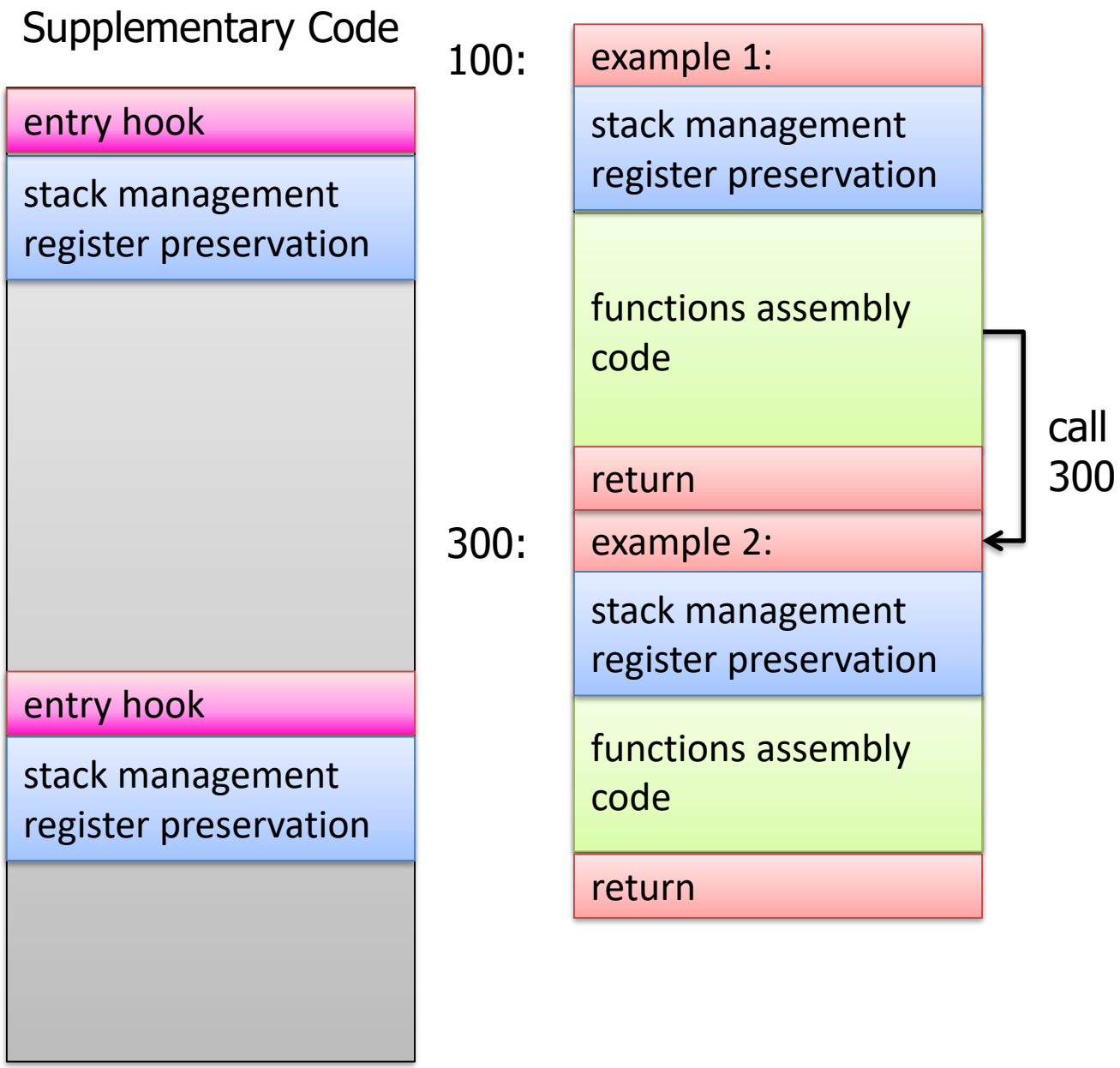
Binary Instrumentation

- Basic approach:
  - Create supplementary code region
  - Place entry hooks



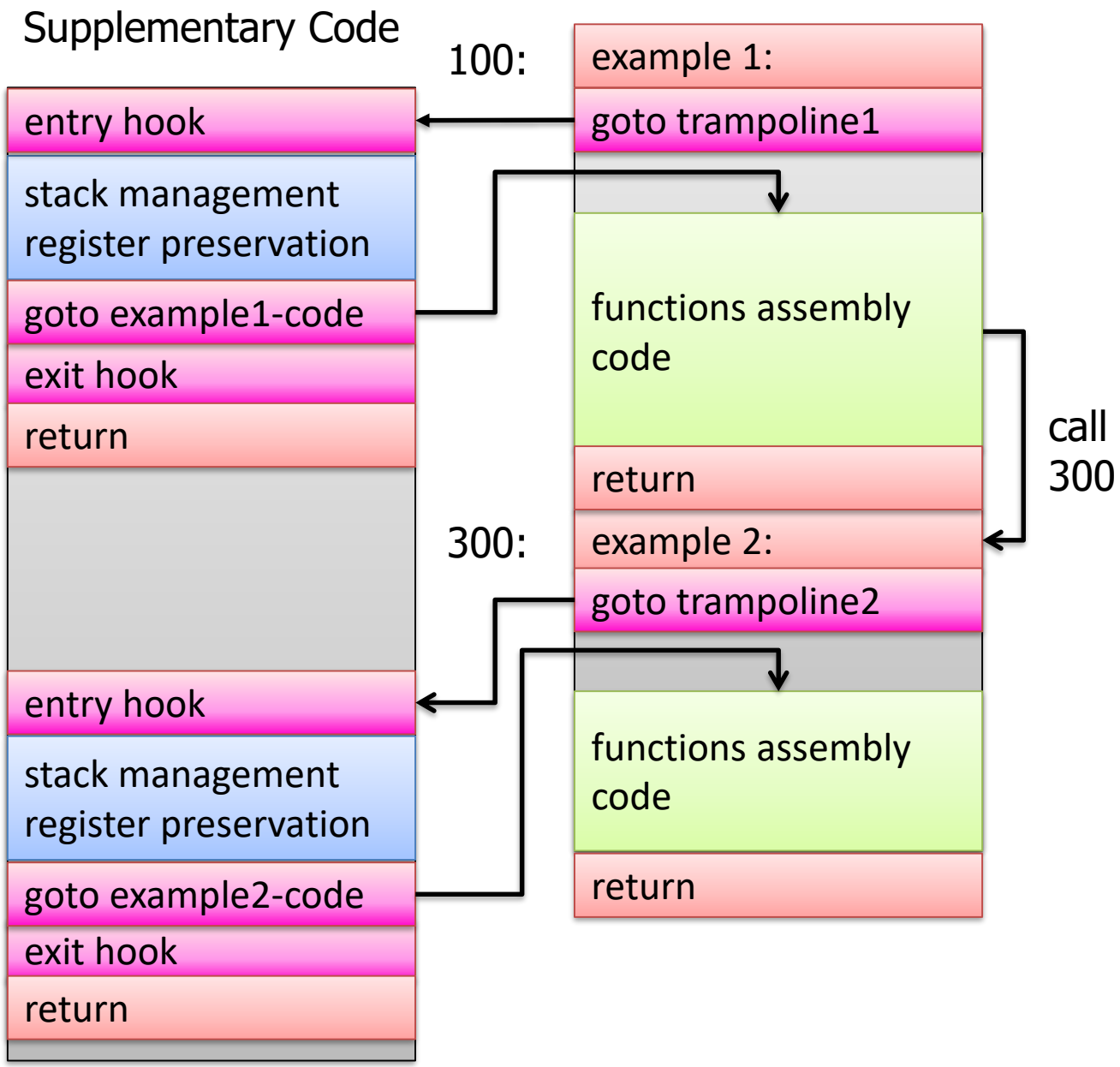
Binary Instrumentation

- Basic approach:
  - Create supplementary code region
  - Place entry hooks
  - Backup beginning of the function
  - Cut out beginning of the function to make room for a „trampoline“



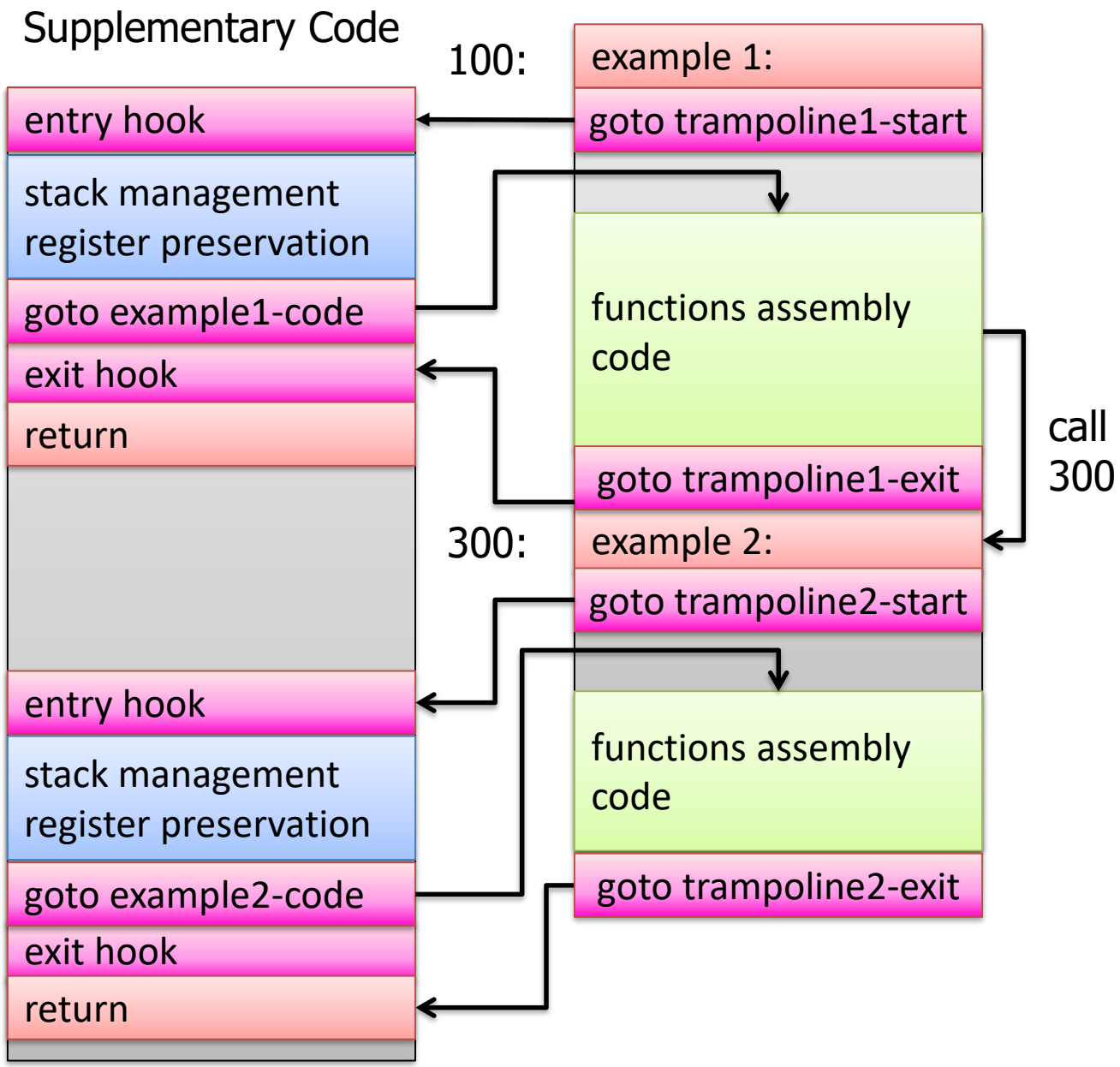
Binary Instrumentation

- Basic approach:
  - Create supplementary code region
  - Place entry hooks
  - Backup beginning of the function
  - Cut out beginning of the function to make room for a „trampoline"
  - Redirect program flow to and from the relocated code-fragments
  - Add exit hook and return statement



Binary Instrumentation

- Basic approach:
  - Create supplementary code region
  - Place entry hooks
  - Backup beginning of the function
  - Cut out beginning of the function to make room for a „trampoline"
  - Redirect program flow to and from the relocated code-fragments
  - Add exit hook and return statement
  - Replace returns with jumps to the exit hook

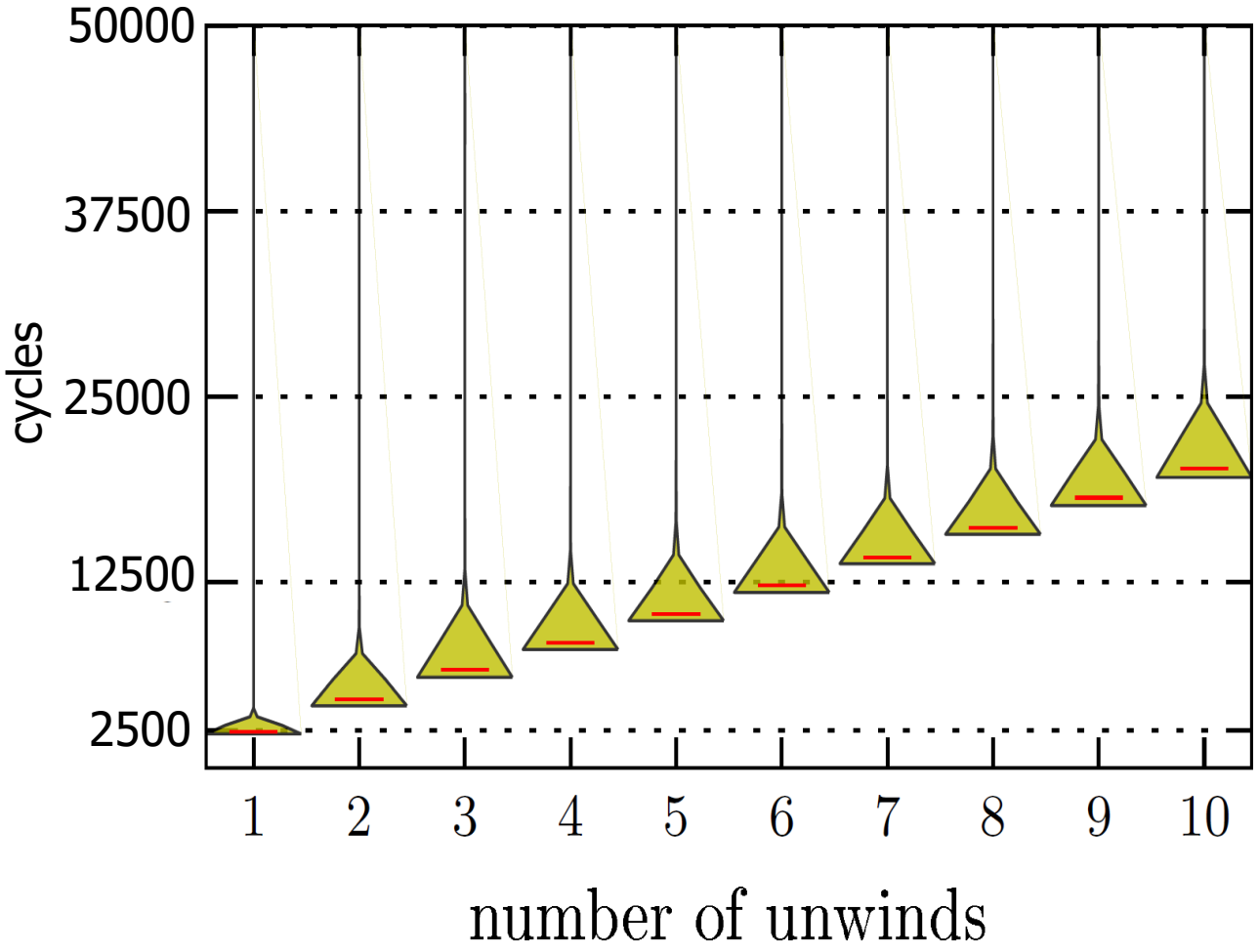


OVERHEAD AND PERTURBATION

Overhead of sampling

- $O_{Sampling} = \sum_{locations} O_{Unwind}(loc.) * n(loc.)$
  - » $O_{Unwinding} \approx 4*10^{-6}$ s,
  - » $\sum_{locations} n(loc.)$/second ≈ 200Hz
- Unwinding is not for free
- Example:
  - Runtime: 10000s
  - Samplerate: 200Hz
  - Average unwinding-depth: 10

  Overhead:
  100000s*200Hz*7.5µs = 15s



DOI: 10.1109/ICPPW.2014.33

CPU: 2,5GHz

Overheads of Instrumentation

- $O_{Instrumentation} = O_{Probe} * n_{invocations}$
- Cost of a single probe:

GNU	Baseline	PIN Direct	PIN Library	Compiler Direct	Compiler Library	Dyninst COBI
Time (ns)	3.0	8.6 **+5.6**	90.3 **+87.3**	5.2 **+2.2**	6.8 **+3.8**	143.1 **+140.1**
Instructions	17	58 **+31**	90 **+73**	24 **+7**	34 **+17**	108 **+91**
Cycles	7	21	225	13	17	357
Br.Ins.	3	9	13	7	9	8
Br.Ucn.	2	6	11	6	8	7
Stl.Icy.	0	0	117	1	0	0

  - $O_{Probe} \approx 4 * 10^{-9}$ s

Examples of number of function calls

Benchmark	Runtime	Number of Functions	Number of Function Calls
444.namd	392	100	1,62E+06
453.povray	155	809	1,41E+06
464.h264ref	62	319	2,55E+08
403.gcc	288	2734	3,40E+08
447.dealII	302	5491	1,17E+08
458.sjeng	488	73	1,11E+10
473.astar	338	129	7,66E+09
433.milc	437	112	1,73E+07
450.soplex	205	769	4,72E+09
482.sphinx3	532	203	1,25E+09
lulesh	102	254	6,50E+10
miniFE	46	552	2,80E+10
DROPS	63	7257	1,70E+11

- $O_{Instrumentation} = O_{Probe} * n_{invocations}$
- Cost of a single probe:
  - $O_{Probe} \approx 4 * 10^{-9}$ s
- Number of probe-calls:
  - 1,41 E+06 < $n_{invocations}$ < 1,70 E+11

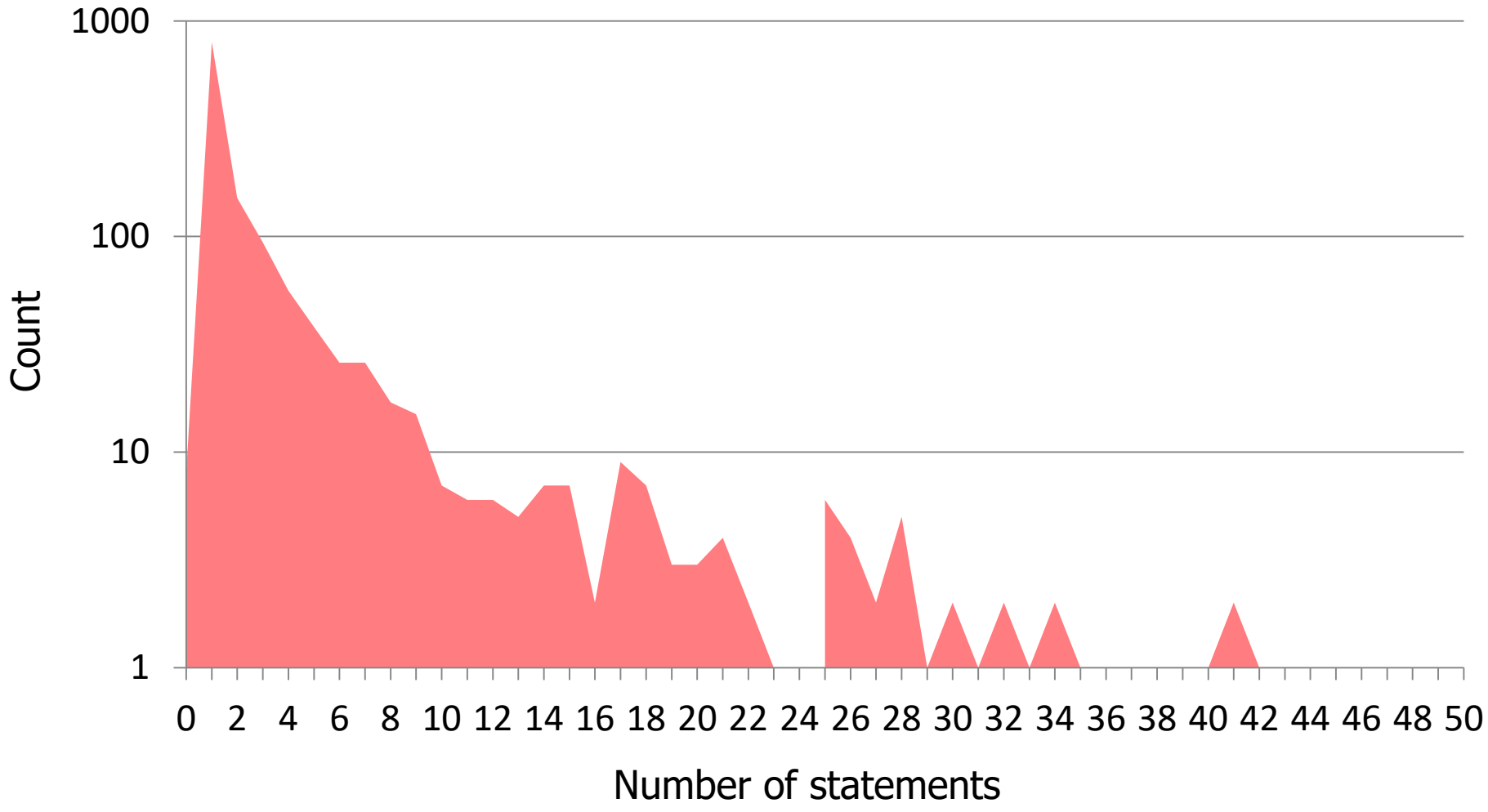
Examples of number of function calls

Benchmark	Runtime	Number of Functions	Number of Function Calls	Overhead	Overhead in %
444.namd	392	100	1,62E+06	1,30E-02	0%
453.povray	155	809	1,41E+06	1,13E-02	0%
464.h264ref	62	319	2,55E+08	2,04E+00	3%
403.gcc	288	2734	3,40E+08	2,72E+00	1%
447.dealII	302	5491	1,17E+08	9,36E-01	0%
458.sjeng	488	73	1,11E+10	8,88E+01	18%
473.astar	338	129	7,66E+09	6,13E+01	18%
433.milc	437	112	1,73E+07	1,38E-01	0%
450.soplex	205	769	4,72E+09	3,78E+01	18%
482.sphinx3	532	203	1,25E+09	1,00E+01	2%
lulesh	102	254	6,50E+10	5,20E+02	510%
miniFE	46	552	2,80E+10	2,24E+02	487%
DROPS	63	7257	1,70E+11	1,36E+03	2159%

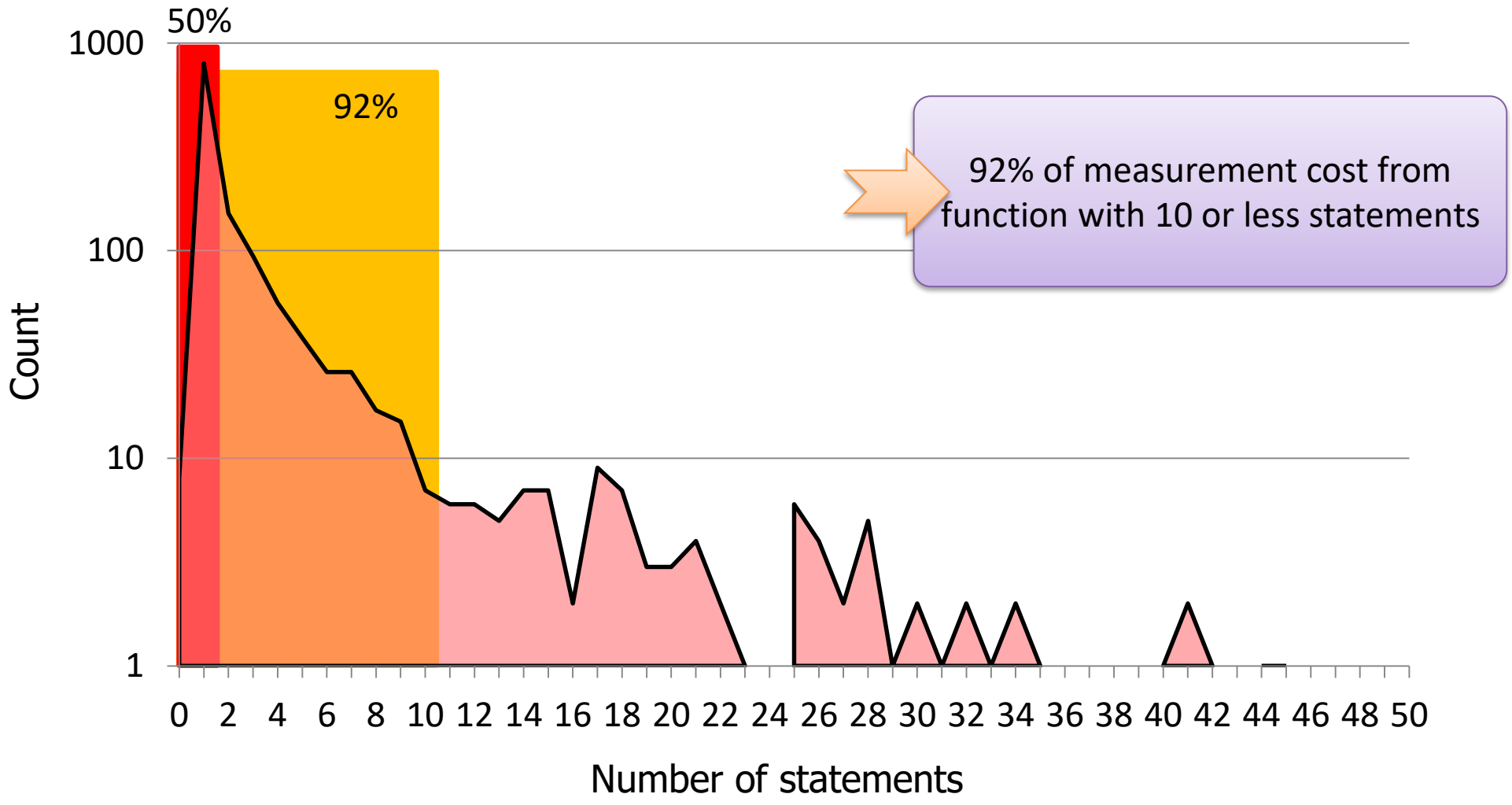
Examples of number of function calls

Benchmark	Runtime	Number of Functions	Number of Function Calls	Overhead	Overhead in %
444.namd	392	100	1,62E+06	1,30E-02	0%
453.povray	155	809	1,41E+06	1,13E-02	0%
464.h264ref	62	319	2,55E+08	2,04E+00	3%
403.gcc	288	2734	3,40E+08	2,72E+00	1%
447.dealII	302	5491	1,17E+08	9,36E-01	0%
458.sjeng	488	73	1,11E+10	8,88E+01	18%
473.astar	338	129	7,66E+09	6,13E+01	18%
433.milc	437	112	1,73E+07	1,38E-01	0%
450.soplex	205	769	4,72E+09	3,78E+01	18%
482.sphinx3	532	203	1,25E+09	1,00E+01	2%
lulesh	102	254	6,50E+10	5,20E+02	510%
miniFE	46	552	2,80E+10	2,24E+02	487%
DROPS	63	7257	1,70E+11	1,36E+03	2159%

Function statistics: coral.lulesh



Function statistics: coral.lulesh



Probe Cost Evaluation

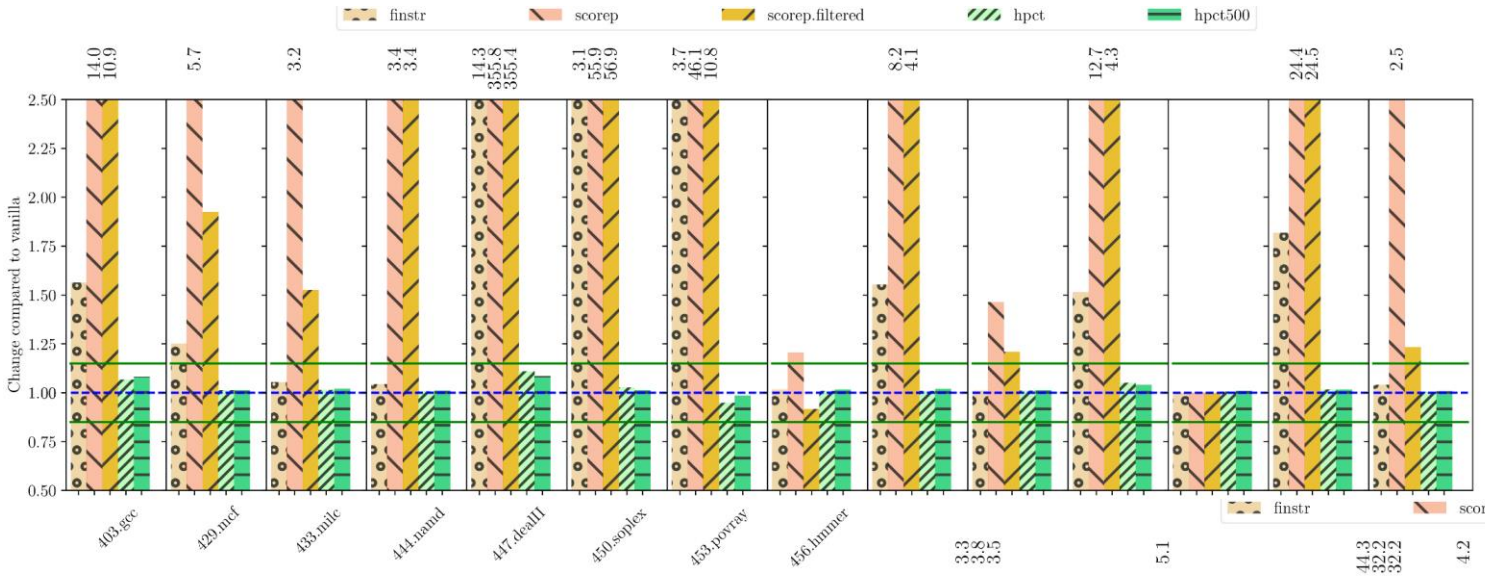
	GNU			Intel		
	Baseline	ScoreP Filtered	ScoreP	Baseline	ScoreP Filtered	ScoreP
Time (ns)	3.0	12.0 **+9.0**	177.1 **+174.1**	4.7	9.7 **+3.0**	182.0 **+177,3**
Instructions	17	85 **+66**	1022 **+1005**	24	68 **+44**	1038 **+970**
Cycles	7	30	442	11	24	454
Br.Ins.	3	19	251	5	14	253
Br.Ucn.	2	12	151	4	8	153
Stl.Icy.	0	0	31	2	1	39

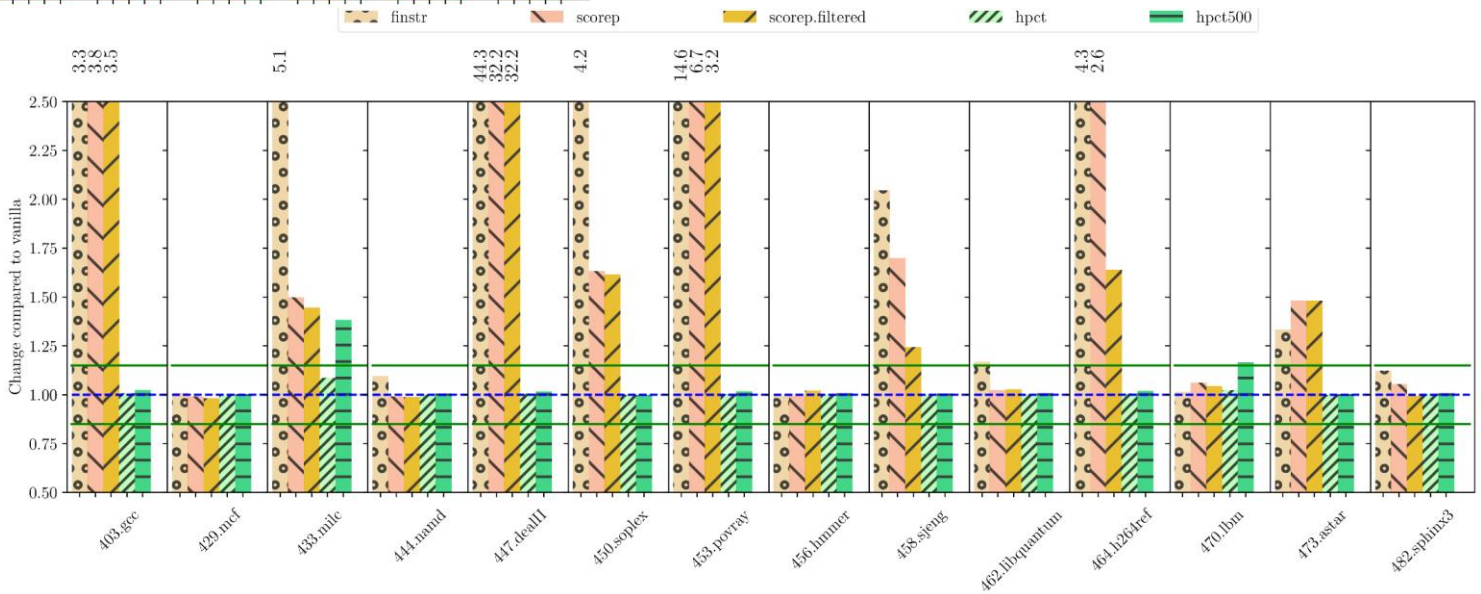
Key insight:
To maintain a specific measurement overhead, a critical amount of work has to be performed for every measurement inverval (entry / exit pair)

Side-Effects of Instrumentation / Sampling

- Hard to quantify!
- Instrumentation:
  - Existance of hooks alters optimization decisions
- Sampling:
  - Unwinding and thunks/trunks impact metrics considerabl
  - Traversing the stack for analysis impacts ist cache locality
- Measurement and processing of data at runtime
  - Impact data-cache

Runtime vs Mispredicted branches

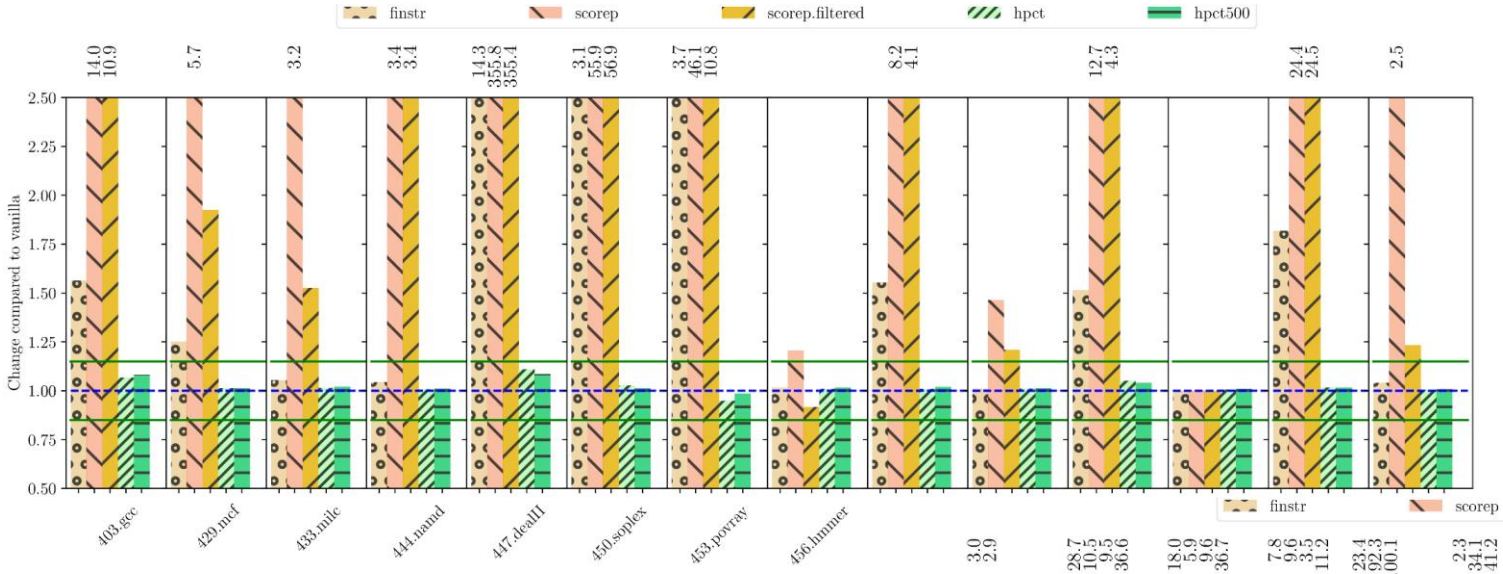


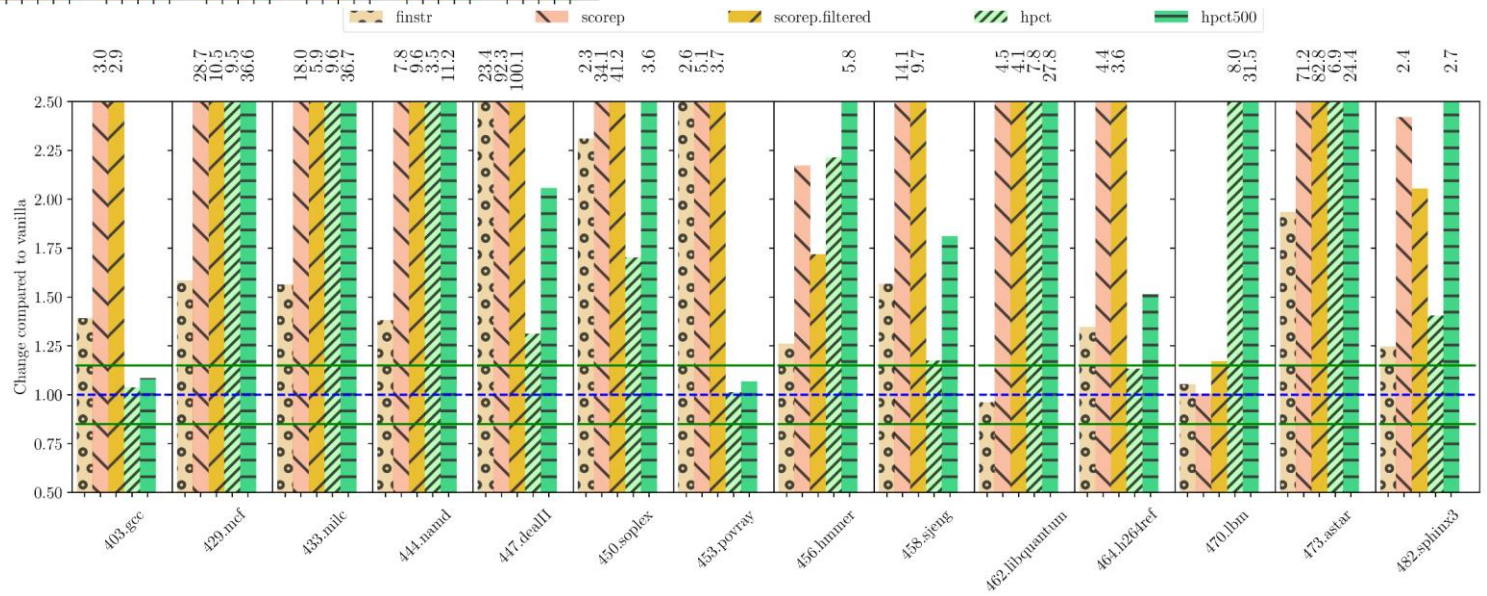


- No filtering or other measures taken to reduce overhead
- Instrumentation clearly perturbes branch-predictoin

DOI: 10.1145/3141865.3141869

Runtime vs Mispredicted L1-Cache misses



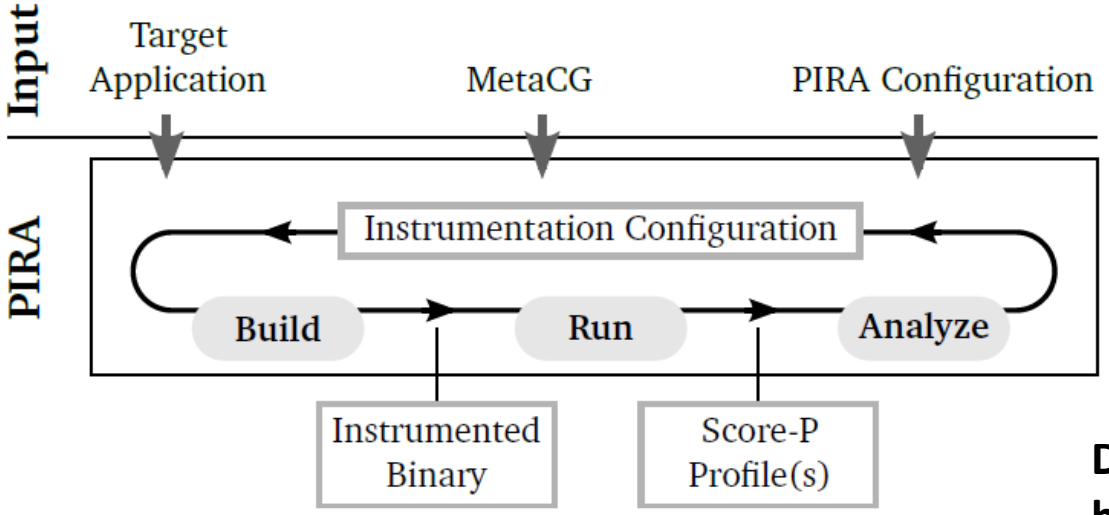


- No filtering or other measures taken to reduce overhead
- Instrumentation clearly perturbes L1-cache misses
- Sampling, although no runtime dialation, perturbes L1 cache misses

DOI: 10.1145/3141865.3141869

Countering Overhead (and Perturbance)

- Instrumentation:
  - Tailor the measurement to the problem
  - Filtering
  - PIRA



DOI: 10.1145/3281070.3281071
https://github.com/tudasc/pira

- Unwinding in instrumentation
  - ScoreP supports unwinding

Profiling

- Recording of aggregated information
  - Time
  - Counts
    - Function/method invocations
    - Hardware counter values

- Collects only process-local information
- Methods to create a profile
  - Interval timer / program counter sampling (statistical approach)
  - Direct measurement (deterministic approach)

Tracing

- Recording information about significant points (events) during execution of the program
  - Entering/leaving of a code region (function, loop, …)
  - Sending/receiving a message ...
- Save information in event record
  - Time stamp, location ID, event type
  - Plus event specific information
- Event trace := stream of event records sorted by time

- Can be used to reconstruct the dynamic behavior

  ⇨ Abstract execution model on level of defined events

Tracing vs. Profiling

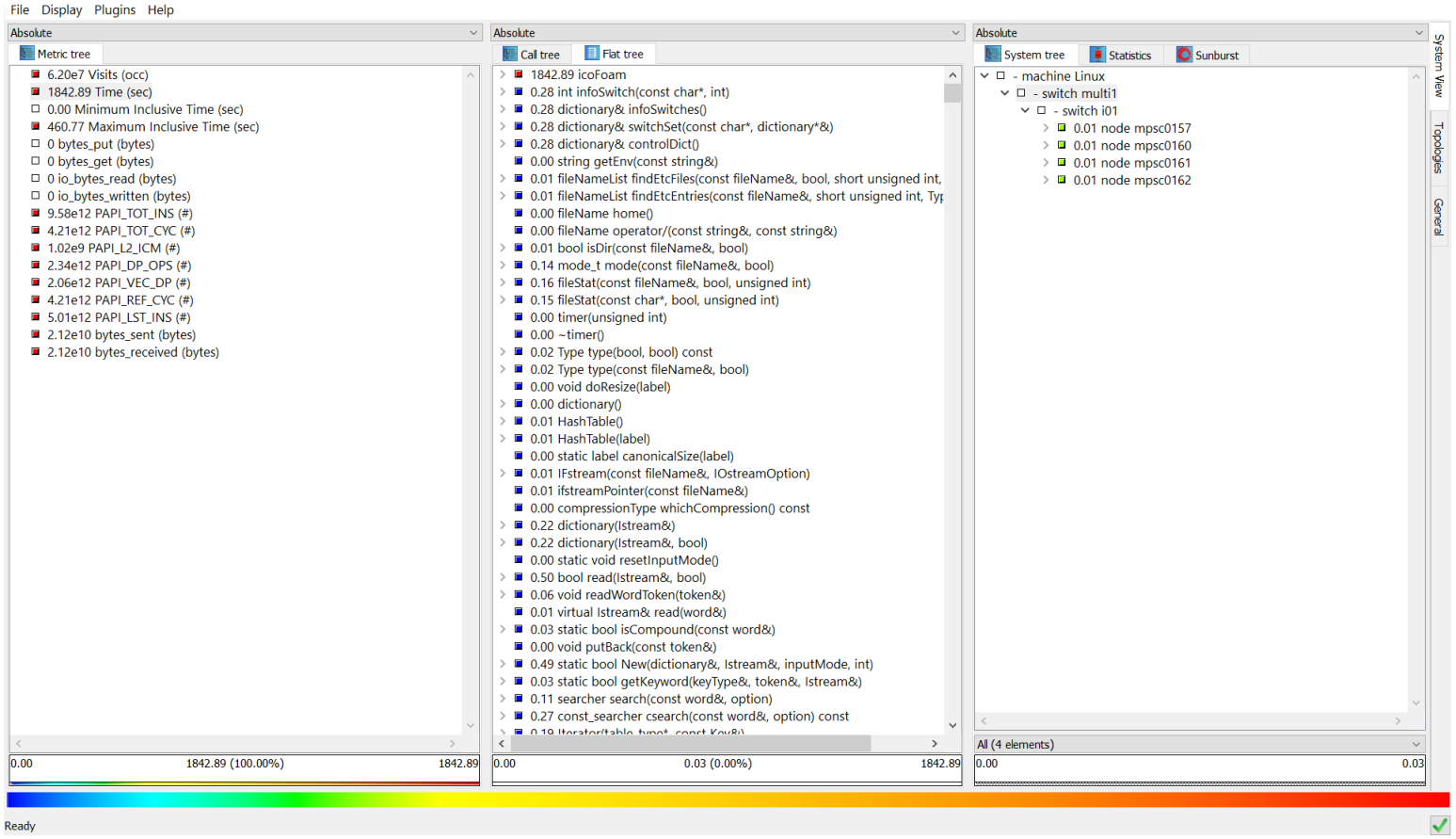
- Tracing Advantages
  - Event traces preserve the temporal and spatial relationships among individual events (⇨ context!)
  - Allows reconstruction of dynamic behaviour of application on any required abstraction level
    - Automatic analysis
    - Visualization
  - Most general recording technique
    - Profile data can be constructed from event traces
- Disadvantages
  - Traces can become very large
  - Flushing of trace buffers to file at runtime can cause perturbation

Tangent: Analysistools

- Trace visualizer
  - Vampir
  - Paraver
  - HPC Toolkit
  - Intel Trace Analyzer and Collector
  - …
- Profile visualizer
  - Cube
  - Intel Amplifyer
  - …
- Analysis & Modeling tools
  - Extra-P

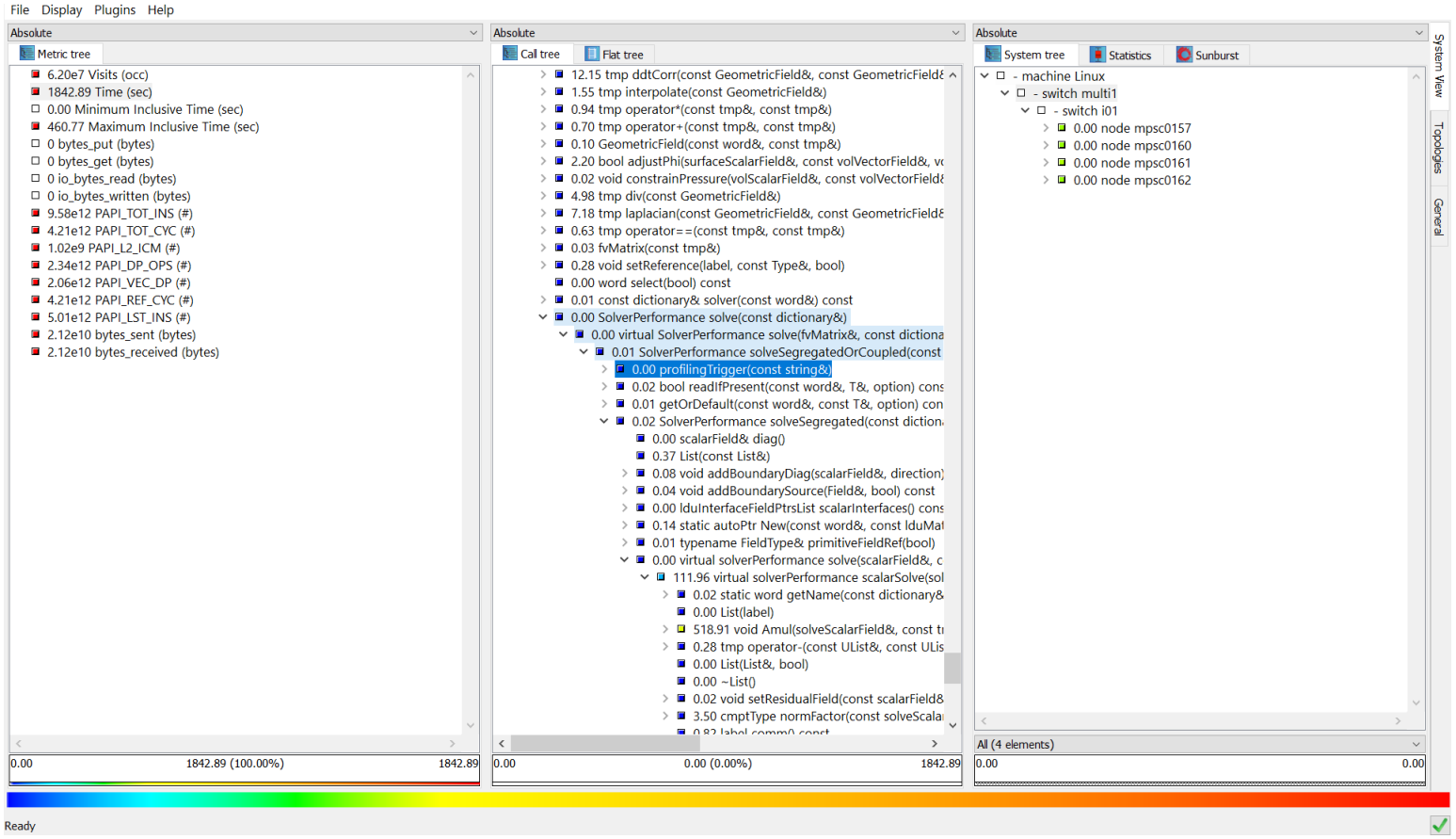
Tangent: Profiles and Cube

- Profiles present aggregate information
- Flat profiles:
  - Each function listed once



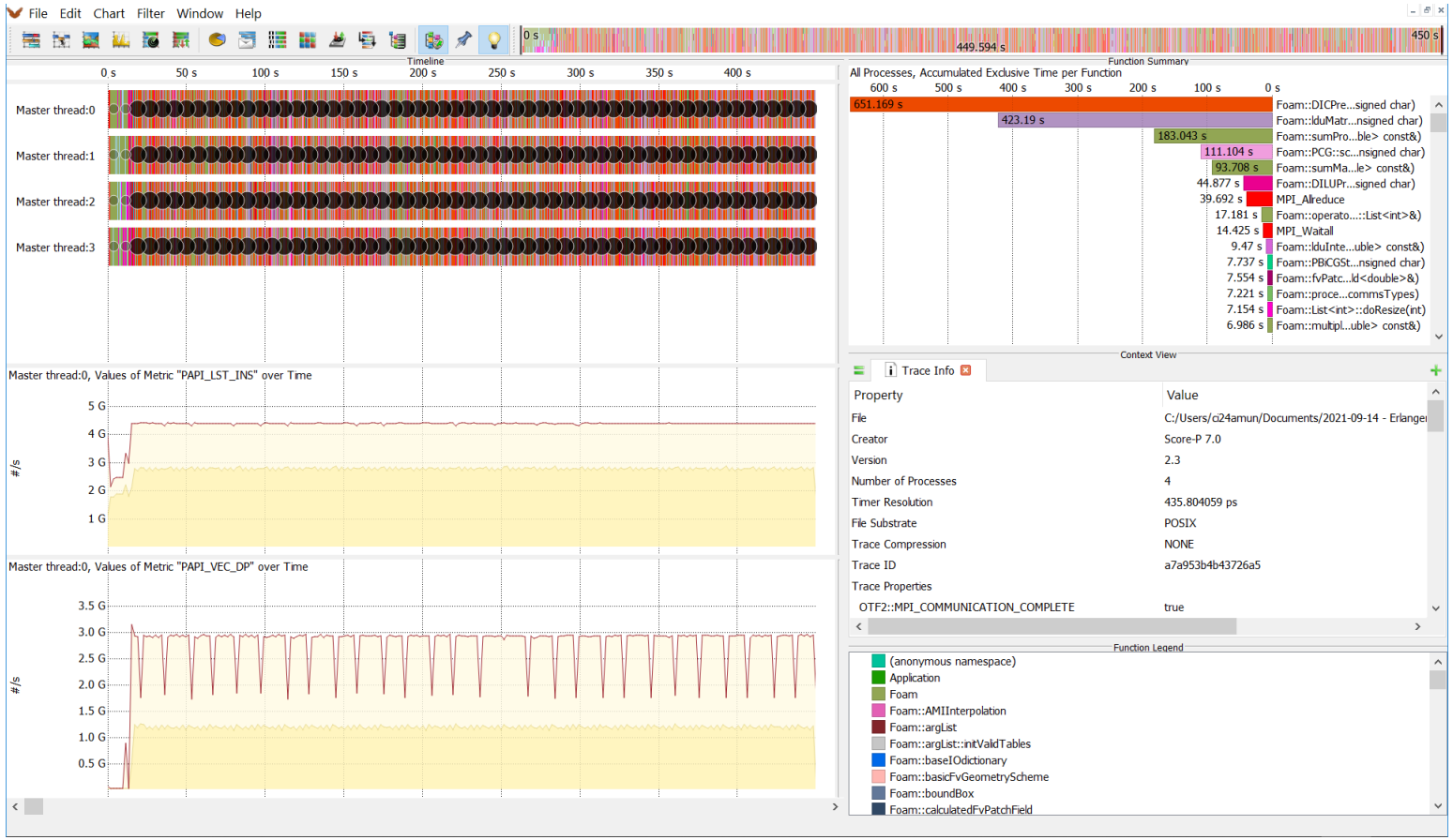
Tangent: Profiles and Cube

- Profiles present aggregate information
- Flat profiles:
  - Each function listed once
- Tree profiles
  - Each call-context is listed
- Great for identifying hot-spots
- Low storage requirements



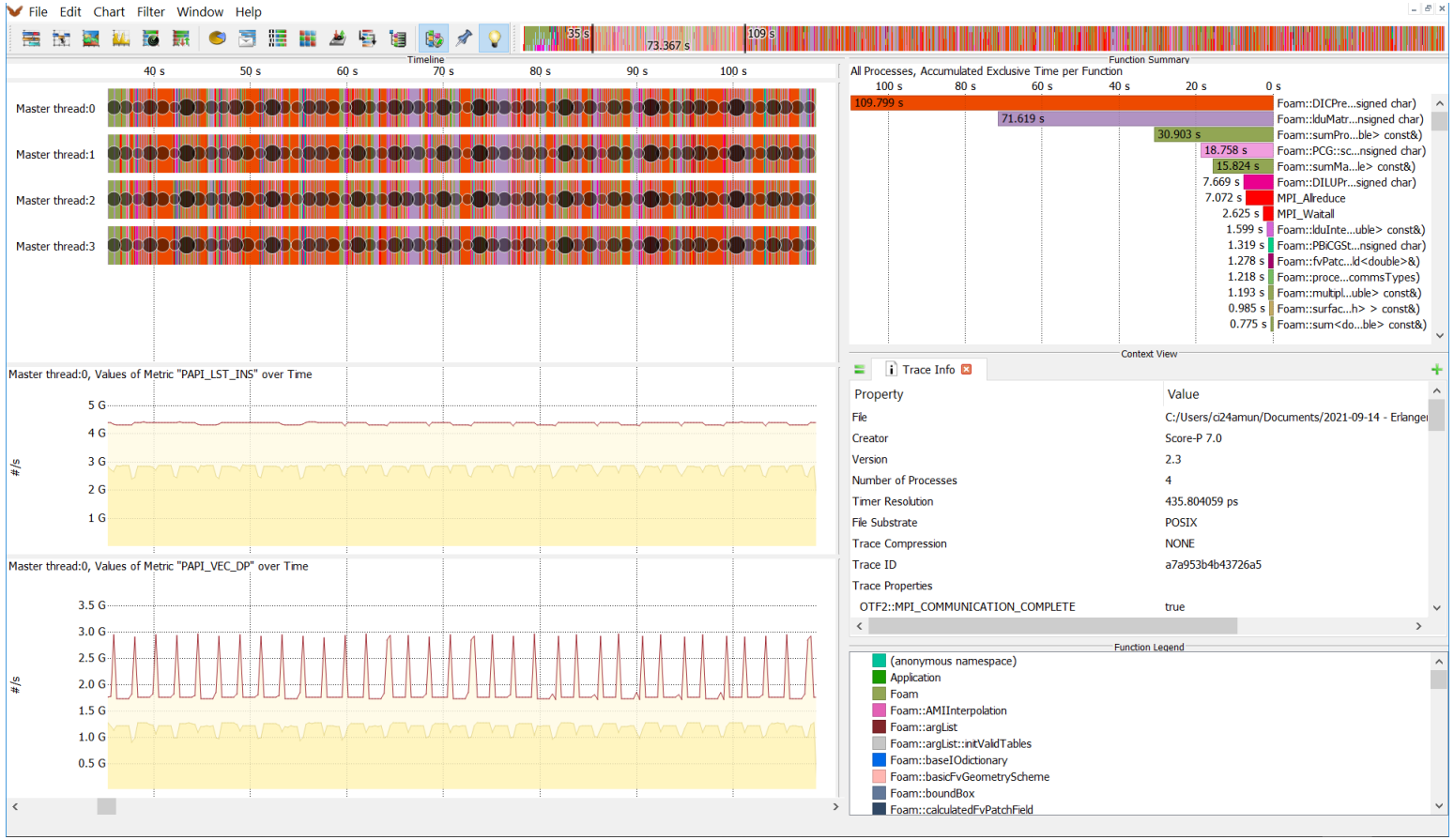
Tangent: Traces and Vampir

- Traces present the temporal evolution
- Each event is available for inspection
- Large to huge storage requirements



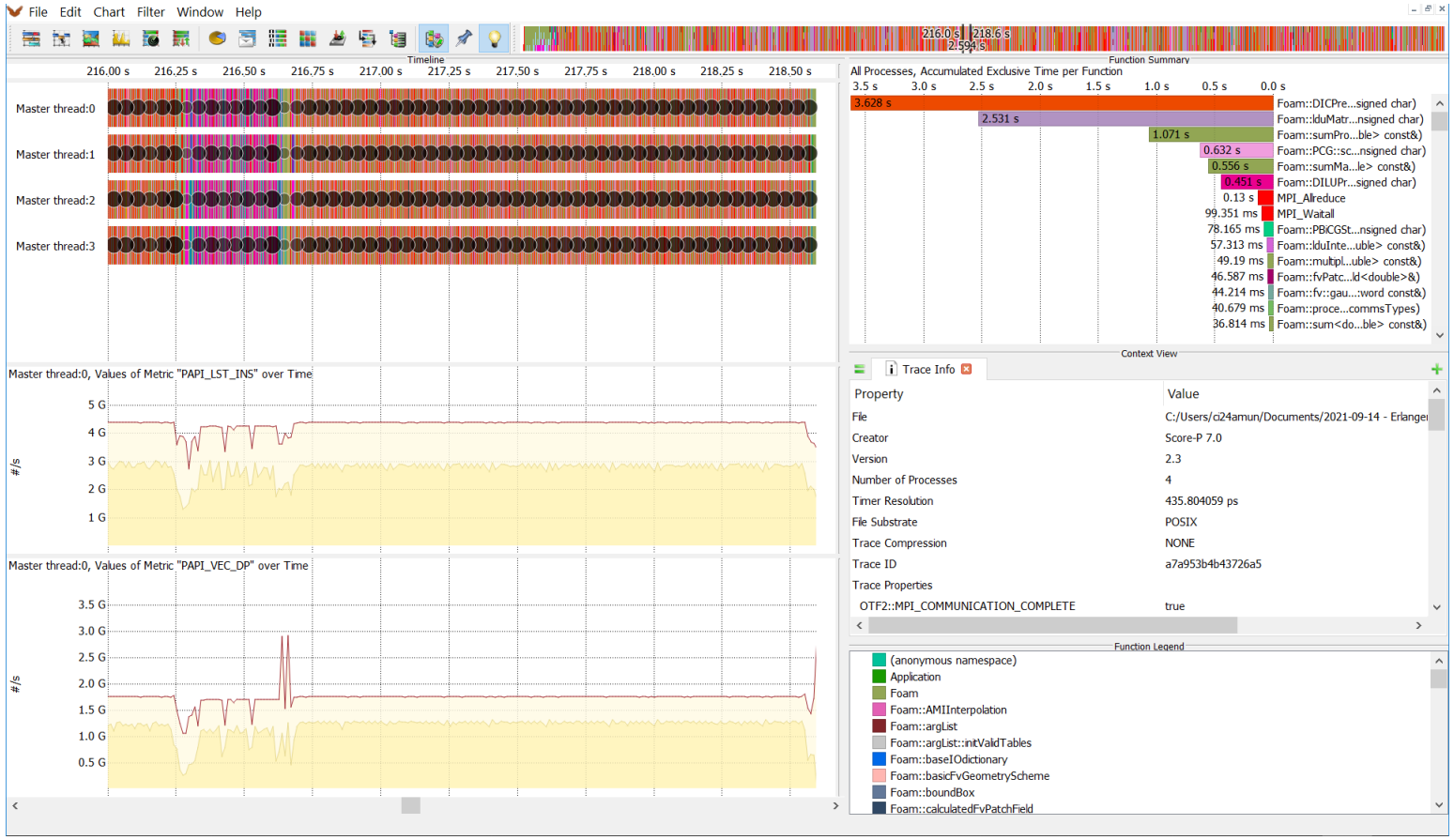
Tangent: Traces and Vampir

- Traces present the temporal evolution
- Each event is available for inspection
- Large to huge storage requirements



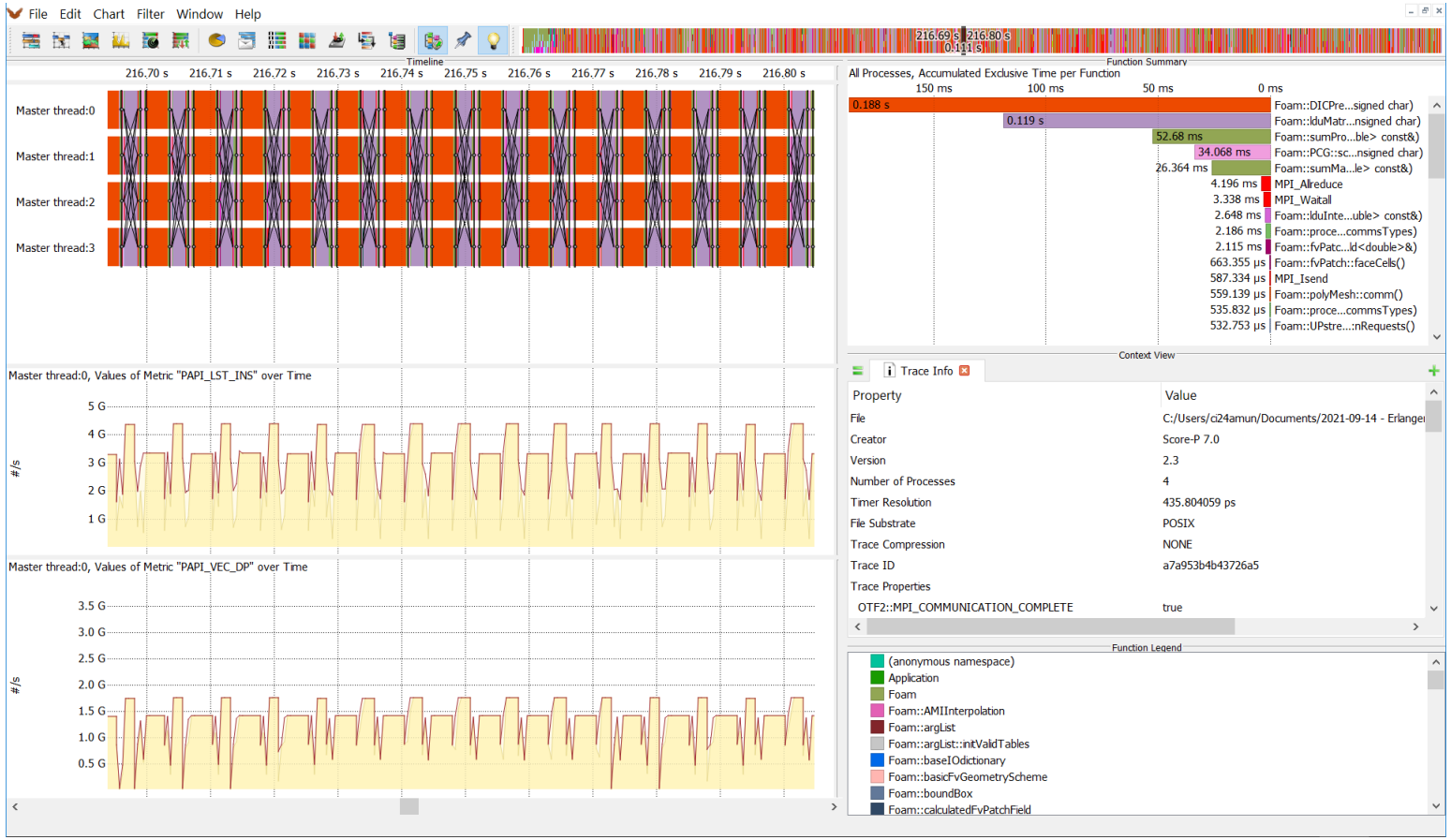
Tangent: Traces and Vampir

- Traces present the temporal evolution
- Each event is available for inspection
- Large to huge storage requirements
- Instrumentation based traces are not limited to sampling frequency



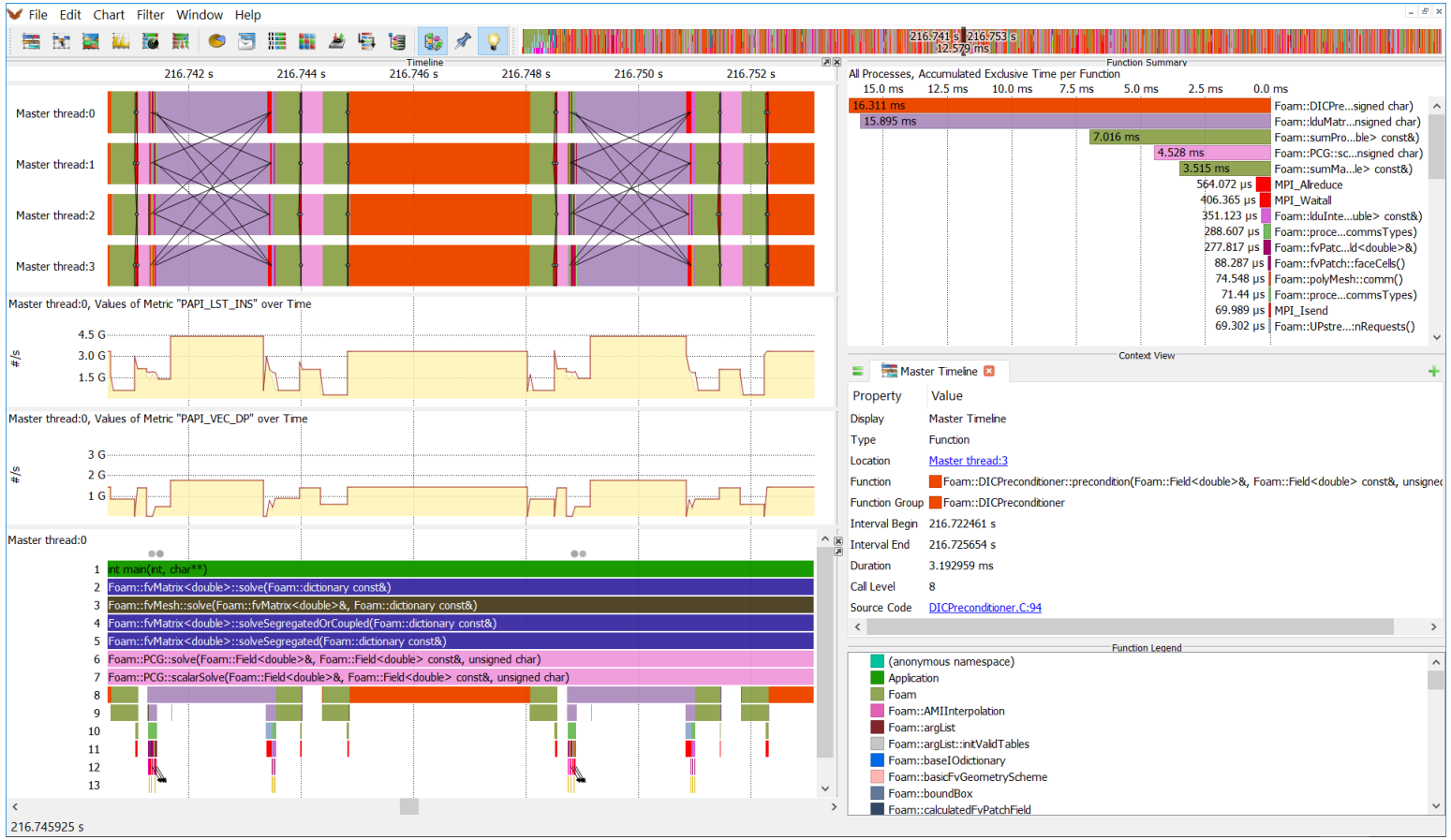
Tangent: Traces and Vampir

- Traces present the temporal evolution
- Each event is available for inspection
- Large to huge storage requirements
- Instrumentation based traces are not limited to sampling frequency



Tangent: Traces and Vampir

- Traces present the temporal evolution
- Each event is available for inspection
- Large to huge storage requirements
- Instrumentation based traces are not limited to sampling frequency



- Call-depth: 13
- Smallest entity 50ys

Example: ExtraP: Semi-empirical performance models

```
main() {
   foo()
   bar()
   compute()
}
```

Input

Measurement

- All functions

Performance measurements (profiles)

$p_1 = 128$
$p_2 = 256$
$p_3 = 512$
$p_4 = 1{,}024$
$p_5 = 2{,}048$
$p_6 = 4{,}096$

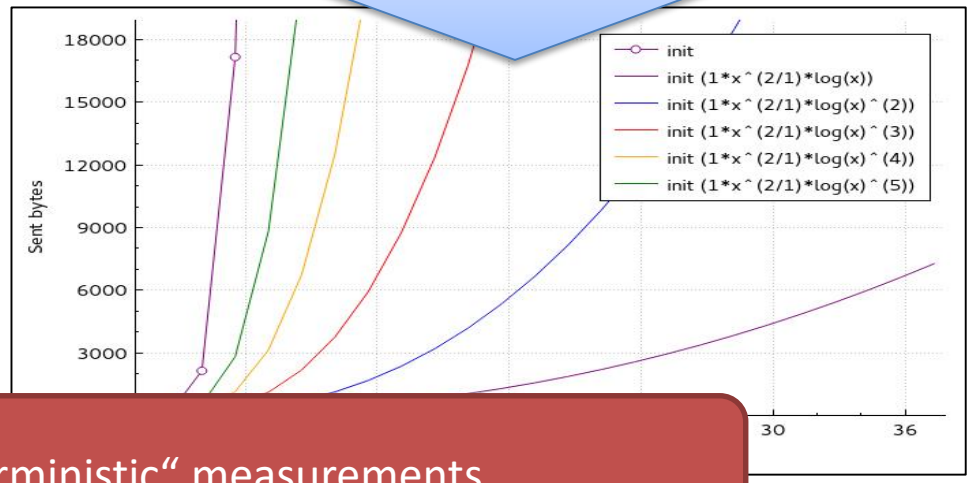
Automated modeling

Output

Ranking:

- Asymptotic
- Target scale p_t

1. foo
2. compute
3. main
4. bar
[ ]



Requires „deterministic“ measurements

Take-Away (Part I)

- Always create a baseline for any measurement
  - Gather the same metrics, that are of interest
  - Compare the baseline to the measurement intended for analysis
- Monitor region to probe cost
  - Measuring regions of code less than 10.000 cycles will generate overhead larger than 10%
- Check assumptions
  - The „target" hot-spot usually is not the only one

Take-Away (Part II)

- Sampling:
  - Taking a sample along its context is more expensive than an instrumentation event
  - Measurement frequency, and in consequence overhead is easy to control
- Instrumentation:
  - Each event is in principle cheaper than a sampling event
  - Controlling event rate is a challenge
- Each technique imprints on specific metrics
- Measurement technique should support use of data
  - Overview and monitoring: sampling is a good choice
  - But ..
  - Analysis and precise tracking of specific locations: Instrumentation is preferable

Questions?

Hessisches Kompetenzzentrum
für Hochleistungsrechnen