

MALT : MAlloc Tracker

A memory profiling tool

sebastien.valat@cern.ch

- We have **good profiling tool** for **timings**(eg. Valgrind or vtune)
- But for what **memory profiling**?
- Memory can be an issue :
 - **Availability** of the resource
 - **Performance**
- Three main questions :
 - How to reduce **memory footprint** ?
 - How to improve overhead of **memory management** ?
 - How to improve **memory usage** ?

- We want to point :
 - **Where** memory is allocated.
 - **Properties** of allocated chunks.
 - **Bad** allocation **patterns** for performance.

```
__thread Int gblVar[SIZE];  
int * func(int size)  
{  
    child_func_with_allocs();  
    void * ptr = new char[size];  
    double* ret = new double[size*size*size];  
    for (.....)  
    {  
        double* buffer = new double[size];  
        //short and quick do stuff  
        delete [] buffer;  
    }  
    return ret;  
}
```

Global variables and TLS

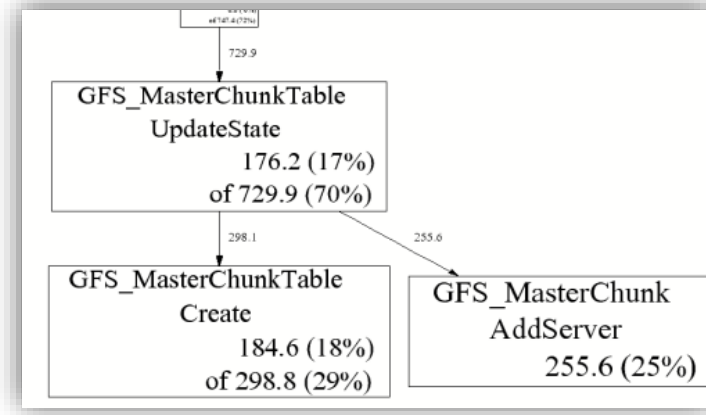
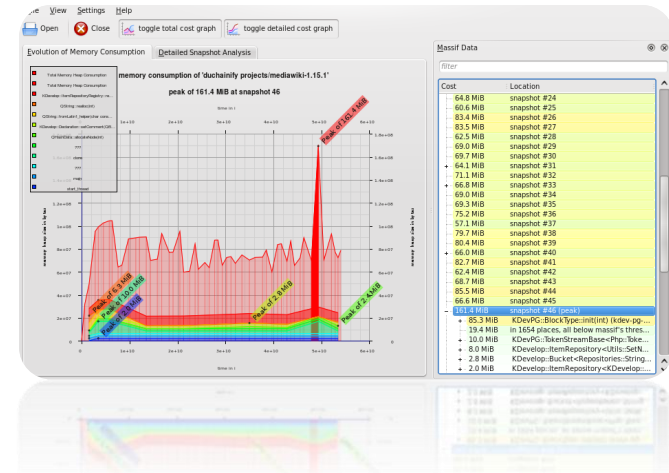
Indirect allocations

Leak

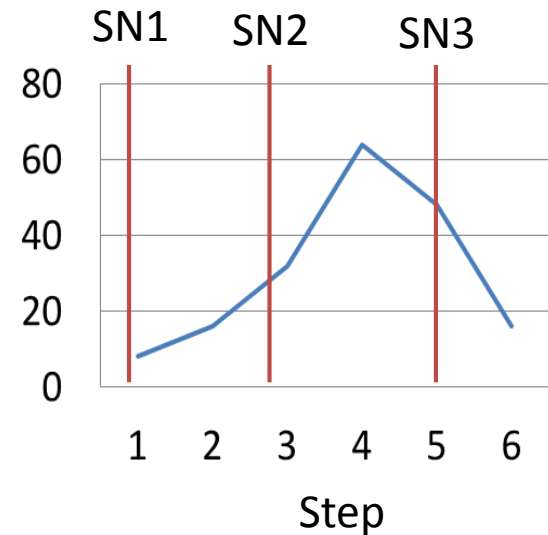
Might lead to swap for large size

Short life allocations

- Valgrind (massif)
 - Memory **over time** (snapshots) & **functions**
 - Memory per function **at peak**
 - Has a simple GUI
- Valgrind (memcheck)
 - **Leaks**
 - No real GUI
- Google heap profiler (tcmalloc)
 - Memory **over time** (snapshots)
 - Faster then valgrind
 - No GUI



- Tau memory profiler
 - Do **not** use **snapshots**
 - Provides **min/average/max**
 - Support MPI
- Commercial tools:
 - Ensure ++
 - Purify++
 - Visual Studio Ultimate edition



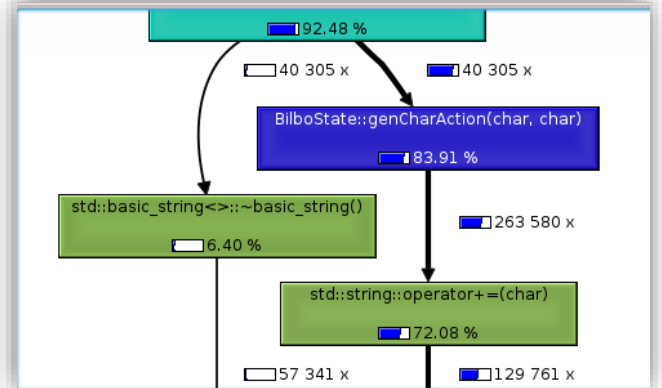
- Same **approach** than **valgrind/kcachgind**
- **Mapped** allocations on **sources lines** and **call stacks**
 - Using **profile** approach, **not snapshots**
- For **memory resource usage** :
 - Memory **leaks**
 - **Memory on peak**
- For **performance** :
 - Allocation **count** and **cumulated size**
 - Allocation **sizes** (min/average/max)
 - Chunk **lifetime** (min/average/max)

- Two approach implemented : **backtrace** and **instrumentation**
- **Backtrace** (default) :
 - Work out of the box
 - Manage all dynamic libraries
 - **Slow** for **large number of calls** ($\sim > 10M$)
- **Instrumentation** :
 - Need source **recompilation** (available) : *-finstrument-function*
 - Or tools for **binary instrumentation** : MAQAO / Pintool (experimental)
 - Faster for really large number of calls to malloc
 - **Only** provide stacks for the **instrumented** binaries

- List of functions with exclusive/inclusive costs

100.00	0.00	(0)	0x0000000000000001
97.96	0.00	1	0x0000000000000401
97.95	0.00	1	(below main)
97.79	0.01	1	main
96.53	0.18	14	BilboState::genOrd
93.73	1.03	1 345	BilboState::findBett
92.69	2.15	40 350	BilboState::countSt
90.54	1.94	40 350	BilboState::countLe
83.18	9.03	41 247	BilboState::genCha
72.50	12.36	270 850	std::string::operato
60.52	6.38	134 107	std::string::reserve
37.60	6.64	134 107	std::string::_Rep::_M
28.80	4.53	134 654	std::string::_Rep::_S
24.27	3.45	134 654	operator new(unsign

- Nice call tree



- Annotated sources

```

0.00 16 call(s) to 'std::string::size() const' (lib
0.00 1 call(s) to '_dl_runtime_resolve' (ld-2.20.
{
    //after 20 chars, try to move to the
    //if (i%10 == 0)
    //    cout << state.genOrderToM
965
966
967
968
969
970
971 0.00 WordCompression wordCompression
0.15 15 call(s) to 'checkForWordCompression(
972 0.00 SequenceCompression sequence
0.03 15 call(s) to 'checkForSequenceCompress
973 0.00 BiSequenceCompression biSequenc
0.01 15 call(s) to 'checkForBiSequenceCompr
974
  
```


- **Started with kcacegrind GUI.... But ...**
- **Display human readable units**
 - You prefer **15728640** or **15 MB** ?
 - I want to **compare to what I expect**.
- **Cannot handle non sum cumulative metrics**
 - **Inclusive** costs **only** rely on **+** operator
 - Some mem. metrics **requires max/min** (eg. lifetime)
- **No way to express time charts**
- **No way to express parameter distributions (eg. sizes).**

- Web technology (NodeJS, D3JS, JQuery, AngularS)
- Easier for remote usage

MATT WebView

Summary Alloc sites Time analysis Stack Alloc sizes Help

Allocated mem. ▾

Search

454.6 MB	__libc_start_main
454.6 MB	_start
454.6 MB	MAIN__
394.8 MB	incompressible_solver_...
394.8 MB	yales2_m::temporal_loop
249.1 MB	linear_solver_operators...
239.5 MB	linear_solver_operators...
59.3 MB	yales2_m::init_yales2
36.8 MB	data_comm_m::update_l...
31.4 MB	incompressible_numeric...
31.3 MB	data_diff_operators_m::...
29.7 MB	grid_io_m::create_grid
28.5 MB	data_buffers_m::use_dat...
27.0 MB	incompressible_numeric...
26.6 MB	bnd_data_defs_m::find_...

3 KB 635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
395 KB 656
657
658
659
660
661
662
663
664
665
666
667
668
669
670

```

/home/svalat/Applications/yales2-devel-2/src/main/yales2_m.f90
call add_new_inputfile(thesolver%first_inputfile,name,new_ptr=input
if (myrank == master) then
  call parse_inputfile(inputfile)
  call print_inputfile(inputfile)
end if
call cast_inputfile(master,inputfile)

end subroutine load_yales2_inputfile
!=====
subroutine temporal_loop()
  implicit none
  ! -----
  select case (solver%type)
  ! -----
  case (SOLVERTYPE_INCOMPRESSIBLE)
    call ics_temporal_loop(solver)
  ! -----
  case (SOLVERTYPE_VARIABLEDENSITY)
    call vds_temporal_loop(solver)
  ! -----
  case (SOLVERTYPE_IMPL_COMP)
    call cps_temporal_loop(solver)
  ! -----
  case (SOLVERTYPE_EXPL_COMP)
    call ecs_temporal_loop(solver)
  
```

Total :
Allocated memory : 2.5 KB
Freed memory : 2.2 KB
Max alive memory : 2.5 K
3 alloc : [14 B, 866 B, 2.2 KB]
2 free : [14 B, 1.1 KB, 2.2 KB]
Lifetime : [70.3 K, 181.4 G, 544.1 G] (cycles)

Function	Metric
__start	2.5 KB
__libc_start_main	2.5 KB
MAIN__	2.5 KB
yales2_m::init_yales2	2.5 KB
yales2_m::load_yales2_inputfile	2.5 KB
inputfile_defs_m::add_new_inputfile	2.5 KB
malloc	2.5 KB
_gfortran_string_trim	14.0 B

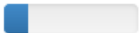
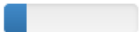
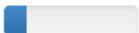
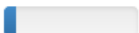
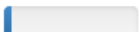
- Provide a small summary
- Provide some warnings

[Show all details](#) [Show help](#)


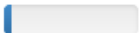
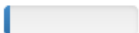
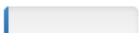
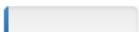
Physical memory peak	66.7 MB
Virtual memory peak	158.1 MB
Requested memory peak	6.1 MB
Cumulated memory allocations	11.5 MB
Allocation count	172.2 K
Recycling ratio	1.9
Leaked memory	743.7 KB
Largest stack	0 B
Global variables	10.0 MB 
TLS variables	48 B
Global variable count	421.0 K 
Peak allocation rate	37.8 MB/s

- Summarize **top functions** for some metrics
- Points to check
- Examples on YALES2

Alloc count

Ratio	Allocs	Function
	911.9 K	data_comm_m::copy_int_comm_to_data
	896.4 K	data_comm_m::copy_data_to_int_comm
	853.2 K	data_comm_m::update_int_comm
	484.9 K	sponge_layer_m::calc_sponge_layer_mask
	296.0 K	incompressible_numerics_m::ics_diffuse_velocity_rk_4th

Allocated memory

Ratio	Allocs	Function
	202.4 MB	linear_solver_operators_m::solve_linear_system_deflated_pcg
	26.6 MB	bnd_data_defs_m::find_bnd_data
	21.8 MB	linear_solver_operators_m::solve_el_grp_pcg
	19.0 MB	data_comm_m::copy_int_comm_to_data
	18.1 MB	data_comm_m::update_int_comm

Peak memory

MATT WebView

Inclusive/Exclusive

Metric selector

Per line annotation

Search

Allocated mem. ▾

- 28.4 KB __libc_start_main
- 28.4 KB _start
- 28.2 KB main
- 12.5 KB testMaxAlive()
- 6.9 KB recurseA(int)
- 6.3 KB testThreads()
- 1.0 KB funcB()
- 1.0 KB testRecuseIntervdA(l...
- 1.0 KB testRecuseIntervdB(l...
- 704.0 B funcC()
- 704.0 B testParallelWithRecur...
- 128.0 B OutOfMainAlloc
- 128.0 B __cxx_global_var_init1
- 128.0 B global constructors ke...
- 128.0 B __libc_csu_init

704 B

```

53 int * ptr = new int[16];
54 *(char*)ptr = 'c'; //required otherwise new compilers will remove malloc/free
55 delete [] ptr;
56
57
58 /***** FUNCTION *****/
59 void funcB()
60 {
61     void * ptr = malloc(32);
62     *(char*)ptr = 'c'; //required otherwise new compilers will remove malloc/free
63     free(ptr);
64     funcC();
65 }
66
67 /***** FUNCTION *****/
68 void funcA()
69 {
70     void * ptr = malloc(16);
71     *(char*)ptr = 'c'; //required otherwise new compilers will remove malloc/free
72     free(ptr);
73     funcB();
74 }
75
76 /***** FUNCTION *****/
77 void recurseA(int depth)
78 {
79     if (depth > 0)
80     {
81         void * ptr = malloc(64);
82         *(char*)ptr = 'c'; //required otherwise new compilers will remove malloc/free
83         free(ptr);
84         recurseA(depth-1);
85     }
86 }
87
88 /***** FUNCTION *****/

```

352 B

704 B

16 B

96 B

7 KB

6 KB

Total :

Allocated memory : 96 B

Freed memory : 96 B

Max alive memory : 96

2 alloc : [32 B, 48 B, 64 B]

2 free : [32 B, 48 B, 64 B]

Lifetime : [41.3 K, 42.1 K, 42.9 K] (cycles)

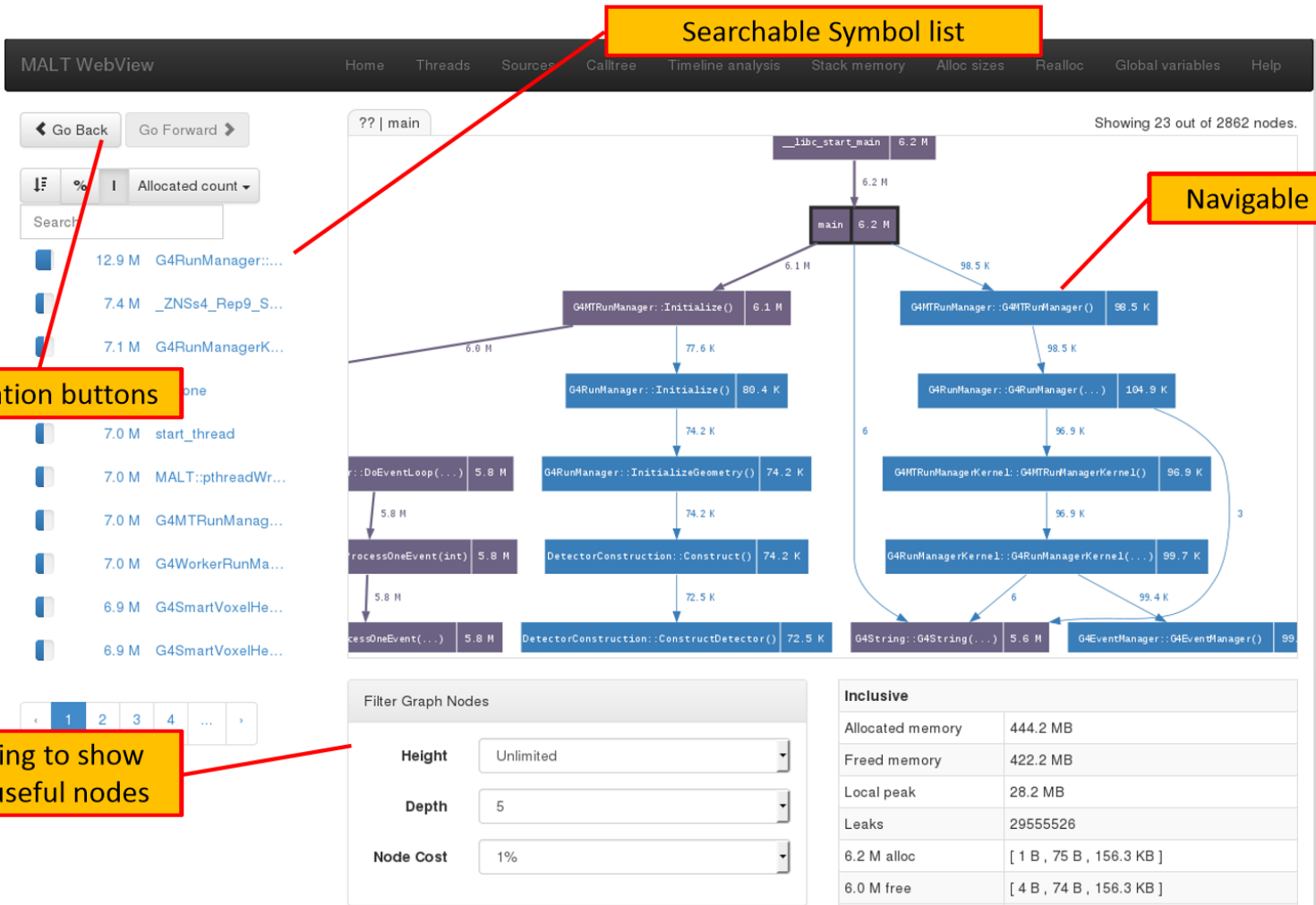
Function	Metric
▼ _start	96.0 B
▼ __libc_start_main	96.0 B
▼ main	96.0 B
▼ funcA()	96.0 B
▼ funcB()	96.0 B
▼ malloc	32.0 B
funcC()	

24/11/2017

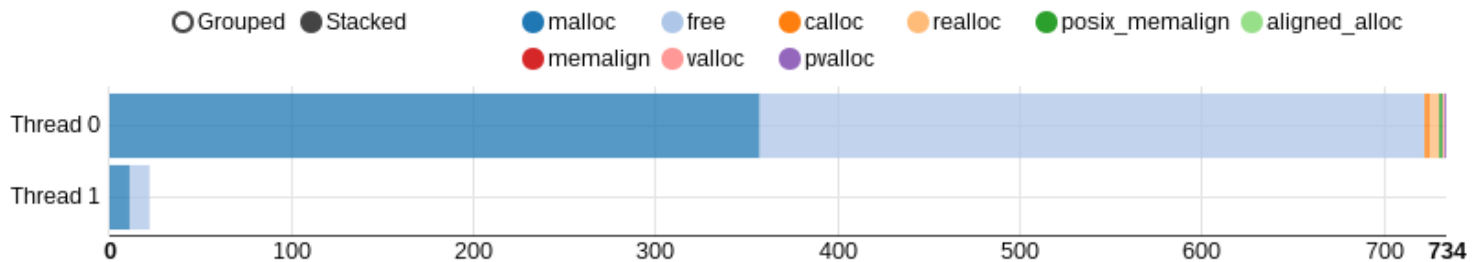
Symbols

Details of symbol or line

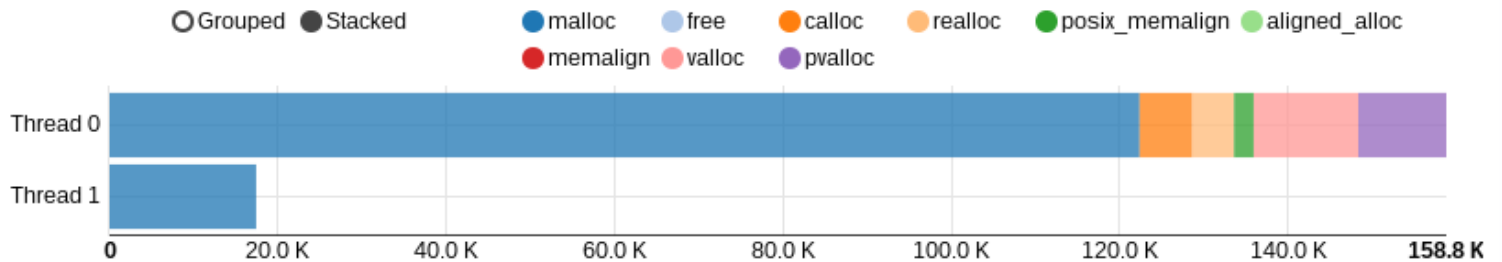
Call stacks reaching the selected site.



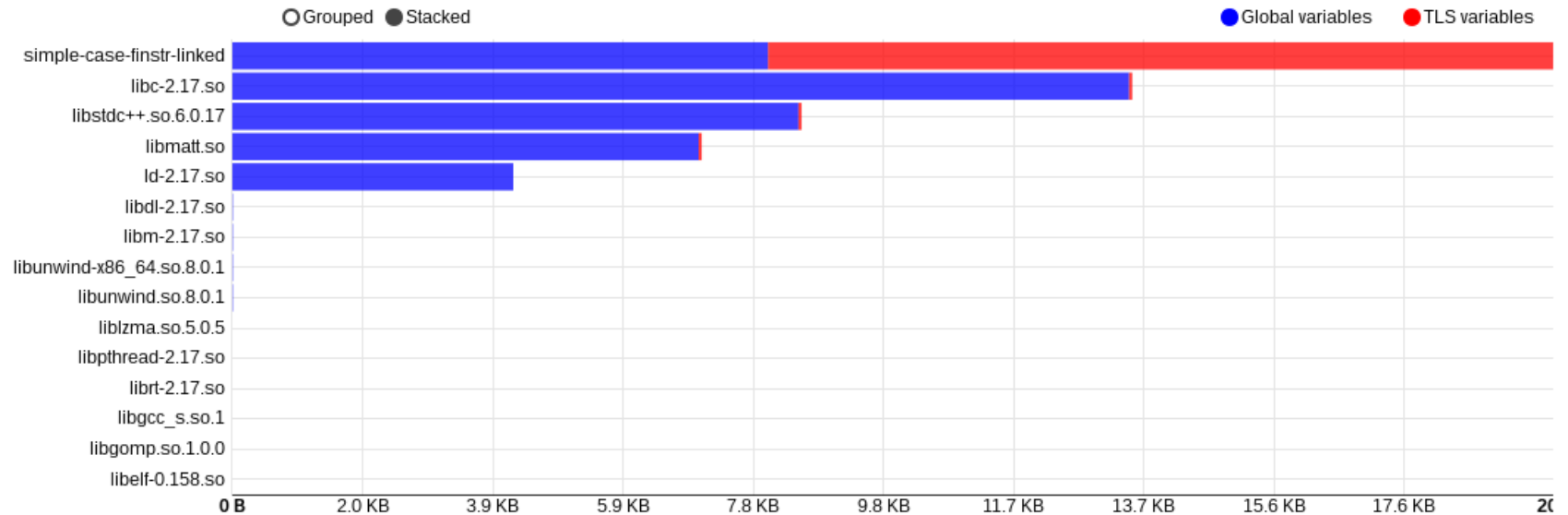
Call per thread



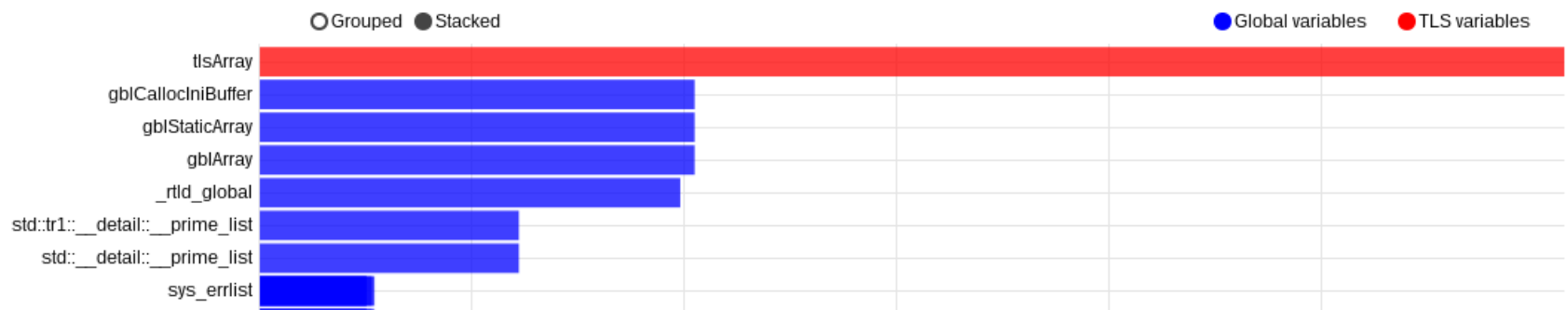
Time per thread



Distribution over binaries

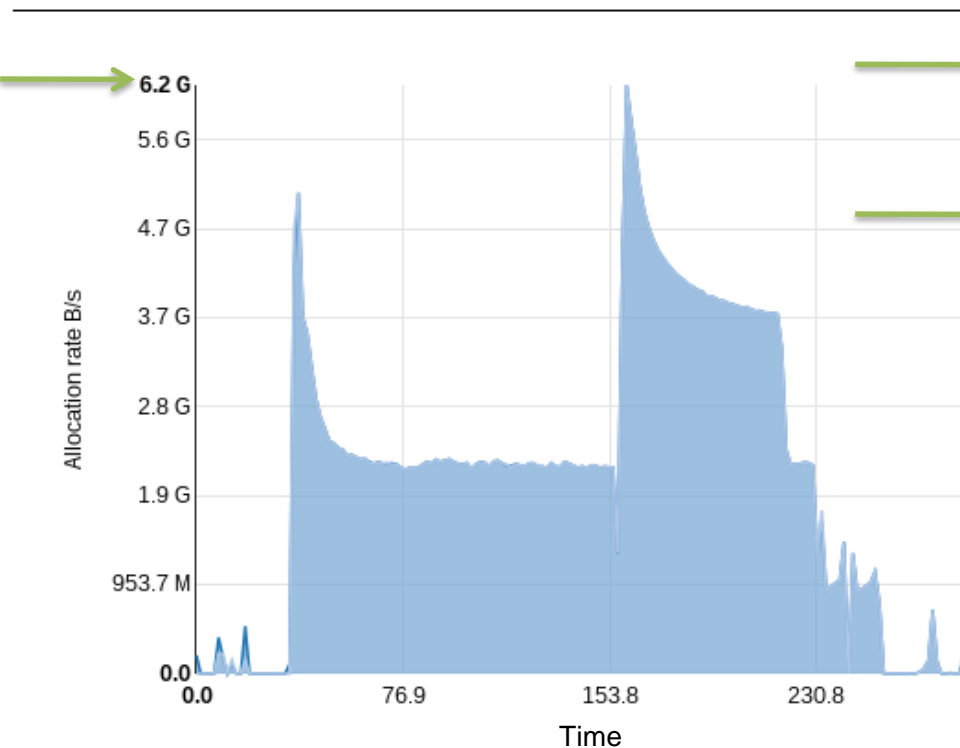


Distribution over variables



- Issue with **reallocation** on init
- Detected with **allocation rate** & **cumulated allocated mem.**

Allocation rate



```

99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
CALL assert(capacity==size(array), &
            'array and capacity variable are not
IF (needed_size>capacity) THEN
  IF (ALLOCATED ( temp) ) DEALLOCATE(temp)
  ALLOCATE ( temp(capacity))

  DO i=1,capacity
    temp(i)=array(i)
  END DO

  DEALLOCATE ( array)
  ALLOCATE ( array(new_cap))

  DO i=1,capacity
    array(i)=temp(i)
  END DO

  capacity=new_cap
END IF

```

Total :

Allocated memory : 56.8 GB ←
 Max alive memory : 135.7 M ←
 3.5 K alloc : [16.0 KB , 16.3 MB , 33.7 MB] ←
 Lifetime : [107.8 K , 26.7 M , 476.7 M] (cycles) ←

Own :

Allocated memory : 56.8 GB
 Max alive memory : 135.7 M
 3.5 K alloc : [16.0 KB , 16.3 MB , 33.7 MB]
 Lifetime : [107.8 K , 26.7 M , 476.7 M] (cycles)

Function
▶ _start

- Issue only occur with **gfortran**, ifort uses stack arrays.

MATT WebView

Allocation count ▾

Search

911.9 K data_comm_m::copy_i...

896.4 K data_comm_m::copy_...

Search intensive alloc functions

Huge number of allocation for a line programmer think it doesn't do any !

```

892 do i=1,nitem el_grp
893   el_grp_ind = el_grp_index2int_comm_index%val(1,i)
894   int_comm_ind = el_grp_index2int_comm_index%val(2,i)
895   el_grp_r2%val(1:dim1,el_grp_ind) = int_comm_r2%val(1:dim1,int_comm_ind)
896 end do

```

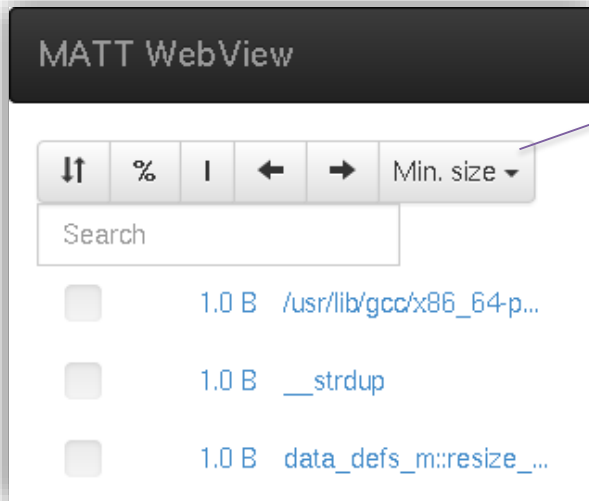
608 K

Total :

Allocated memory : 9.5 MB
 Freed memory : 9.5 MB
 Max alive memory : 432
 608.0 K alloc : [16 B , 16 B , 16 B]
 608.0 K free : [16 B , 16 B , 16 B]
 Lifetime : [24.5 K , 39.9 K , 37.8 M] (cycles)
 Own: 24/11/2017
 Allocated memory : 9.5 MB

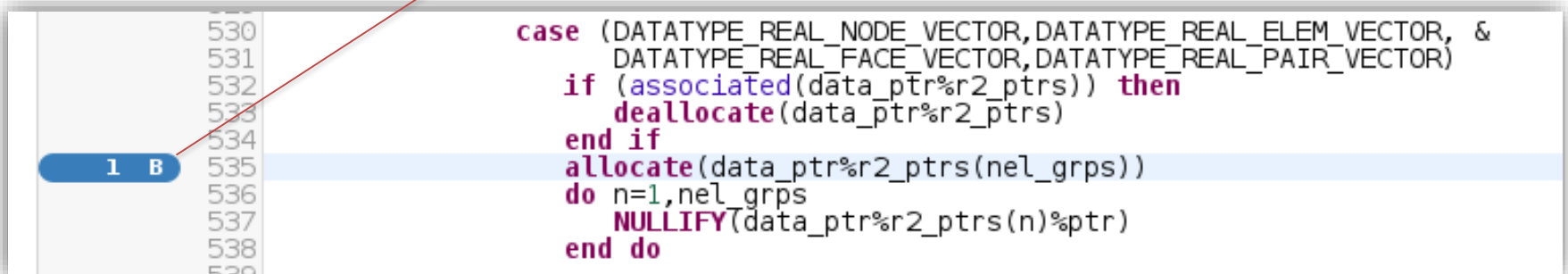
And mostly really small allocations !

- Examples on YALES 2, small allocations :



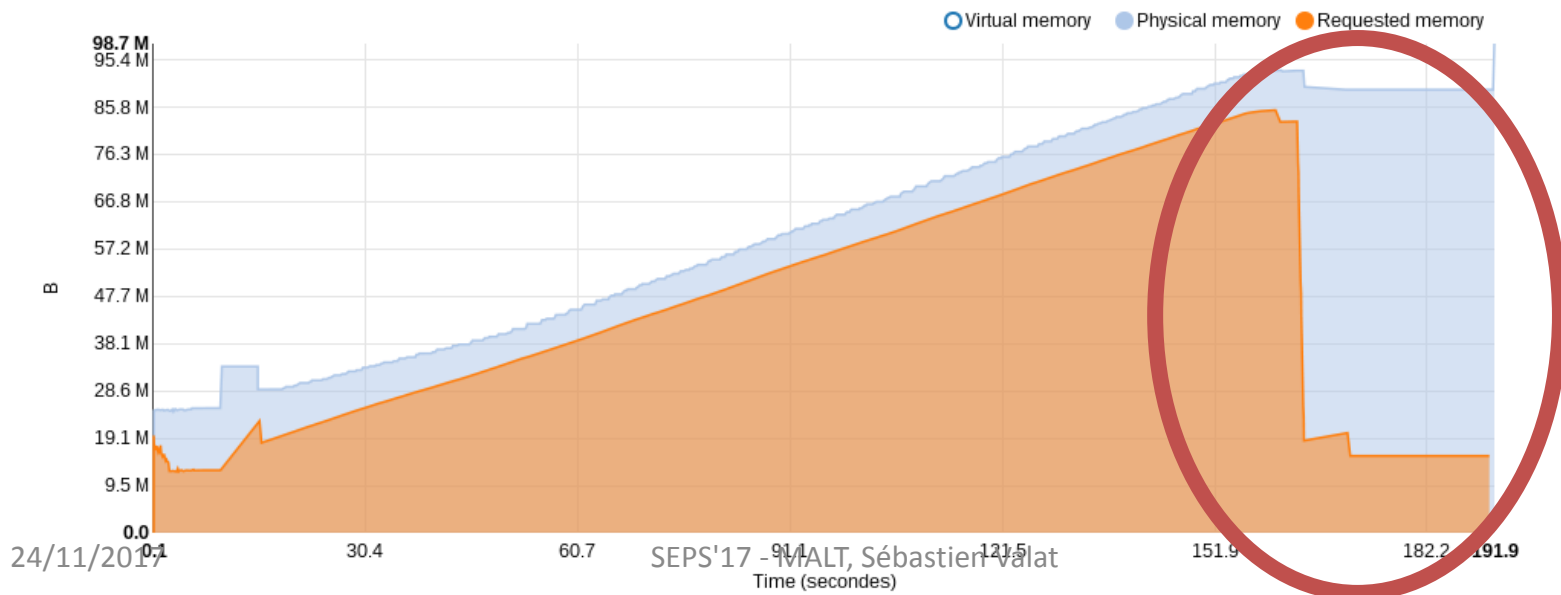
Search for the minimal chunk size.

Many codes produce allocations of 1B.
OK with moderation.



- Example of **fragmentation** detection
- Using the time chart with **physical**, **virtual** and **requested** memory
- **Solution** : **avoid interleaved** allocation of chunks with **different lifetime**.
- Looking on **source annotation** : most of them **can be avoided**.

Memory allocated over time



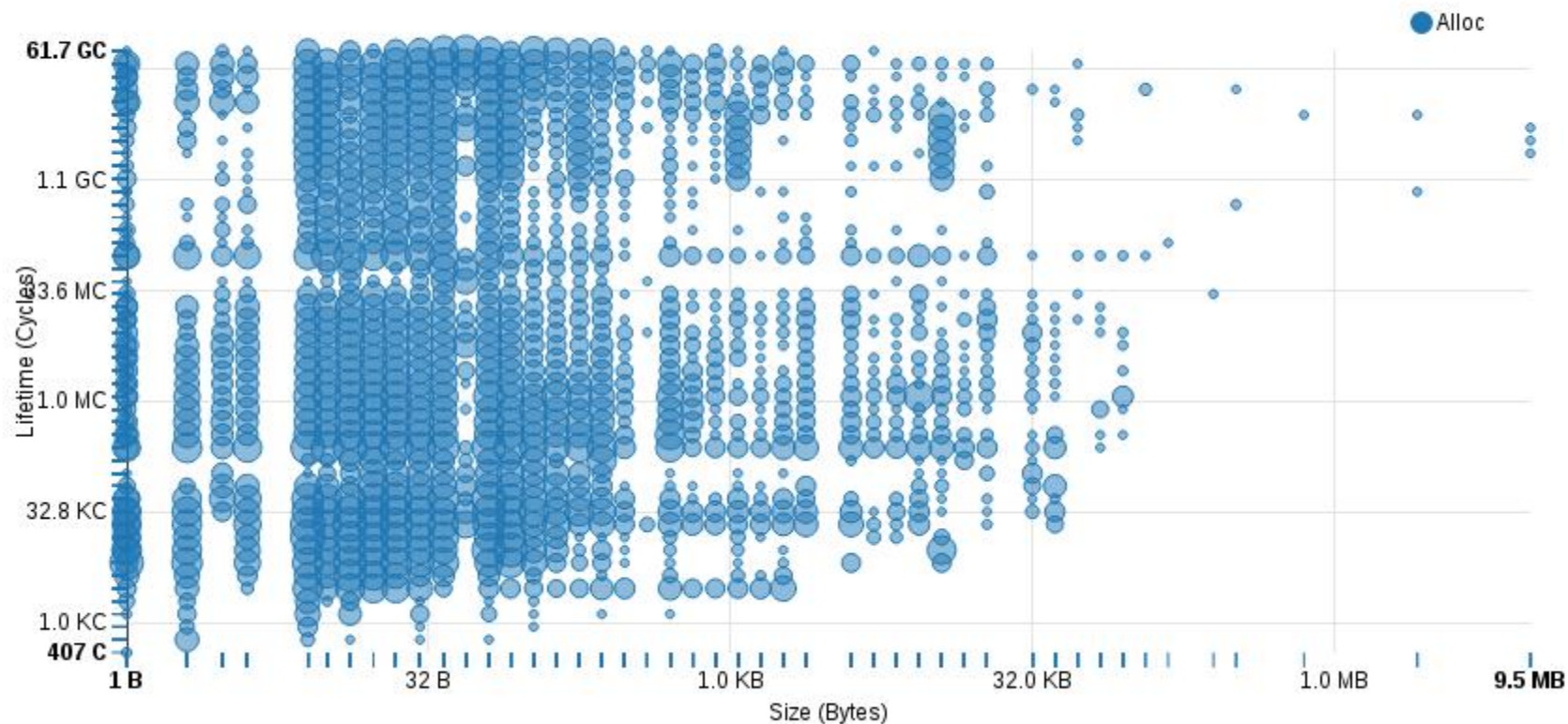
- We can provide a usefull tool
 - **Merging** properties of **all others tools**
 - Extending by some **new features**
 - Mostly adding **properties of allocations**
 - Direct source **code annotations**
- Already found interesting real and unexpected use cases
- Future work :
 - Integrate **traces** into the view (already get all the backend stuff)
 - Add **NUMA** informations (at lease statistics about usage)
 - Hope to get **Open Source** release soon

Thank you.

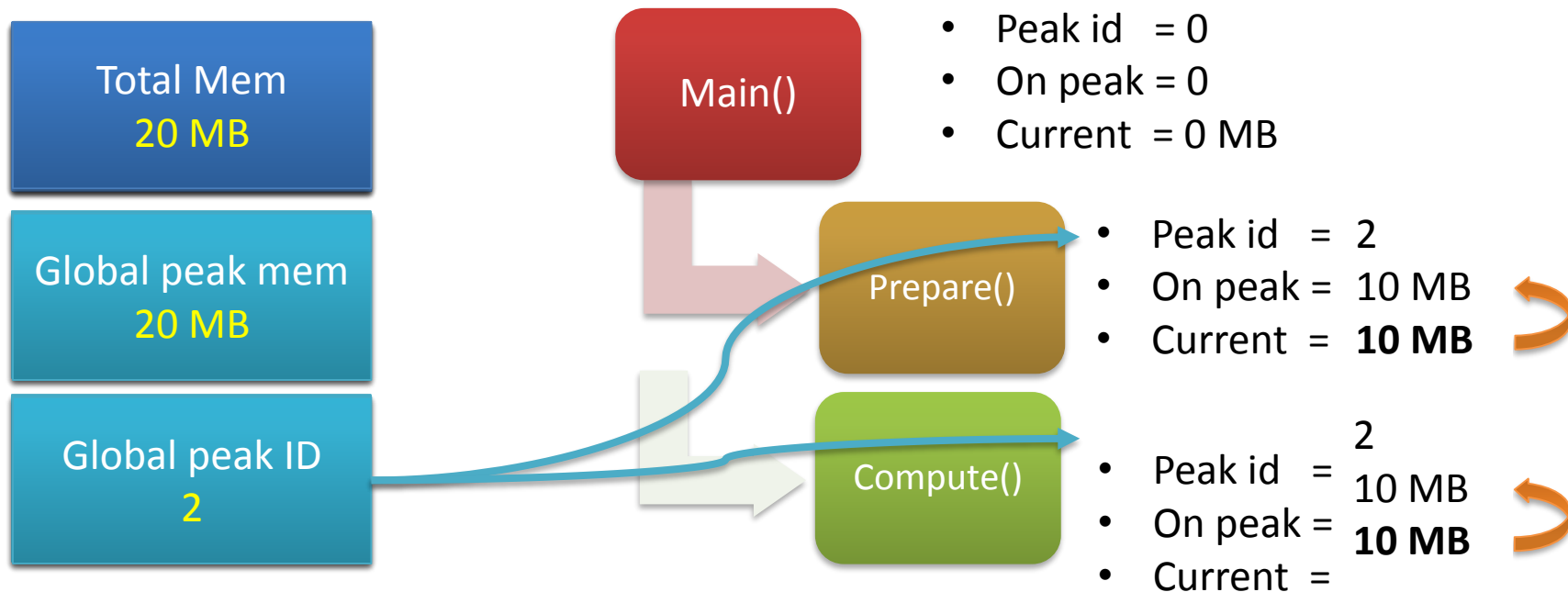
QUESTIONS ?

BACKUP

Lifetime over size

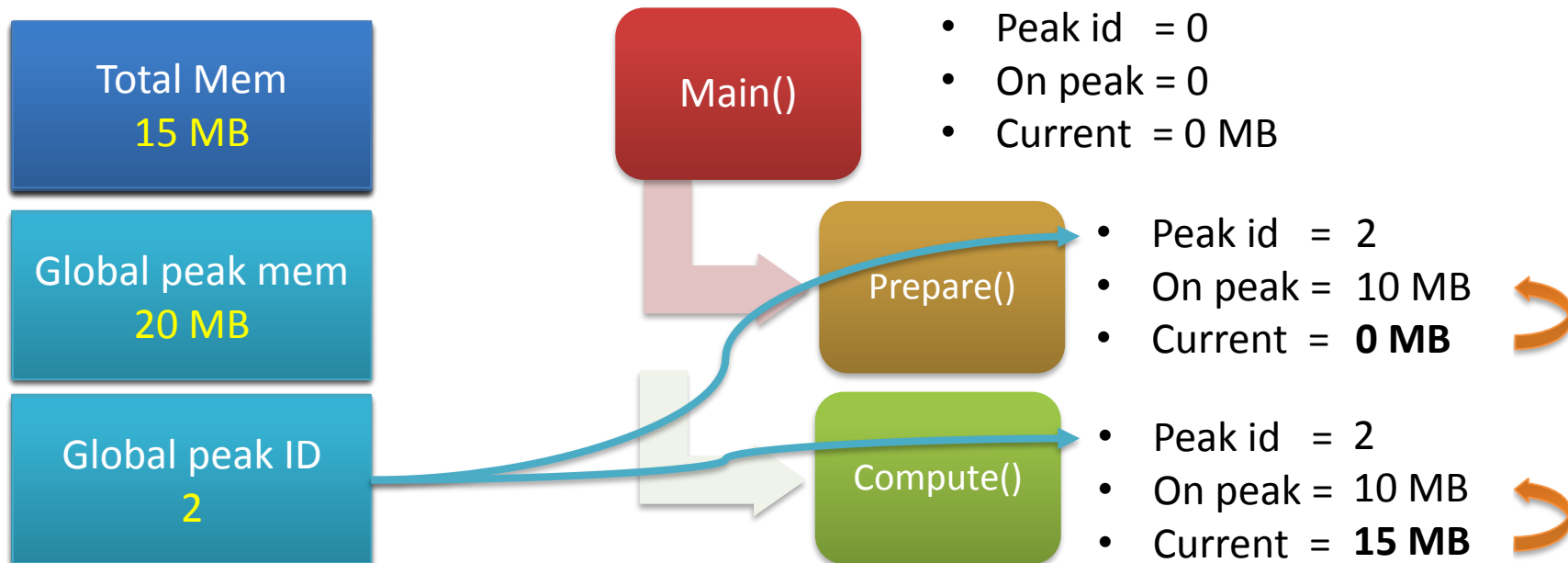


1. *Take peak snapshot on all new memory increase...*
2. Snapshot on free calls with 1% cutoff (valgrind – massif)
3. **Lazy updating** => exact peak at low cost



1. *Capture statistics on all new memory increase...*
2. Capture on free then with 1% cutoff (valgrind – massif)
3. **Lazy updating**

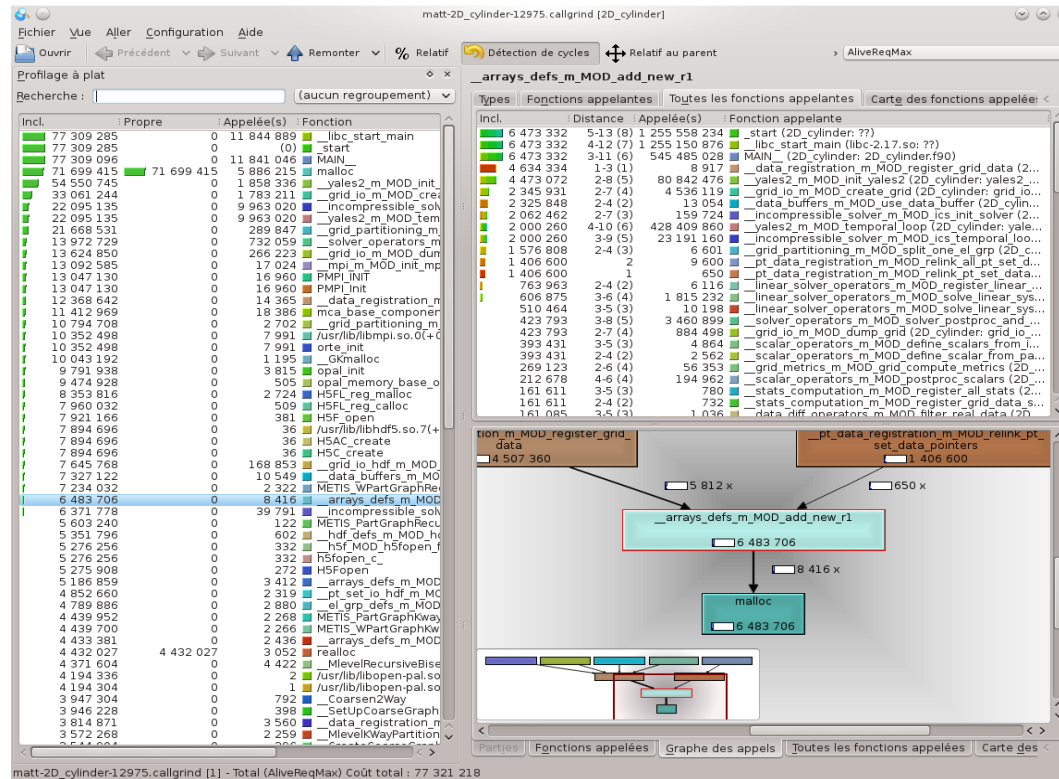
Compute() => malloc(**5 MB**)



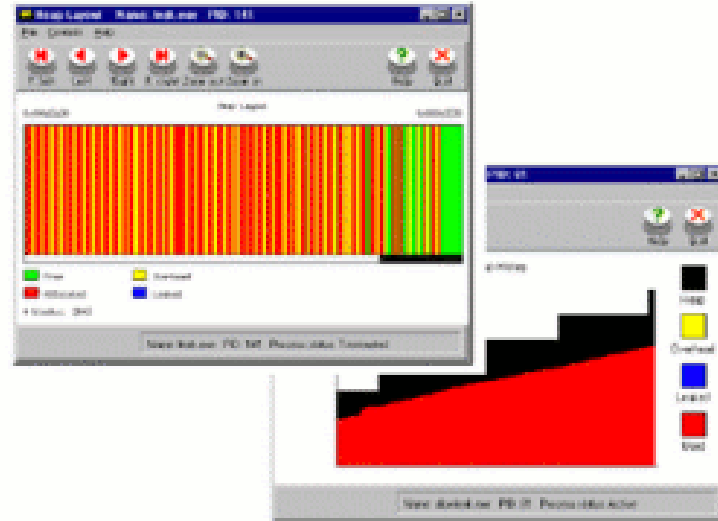
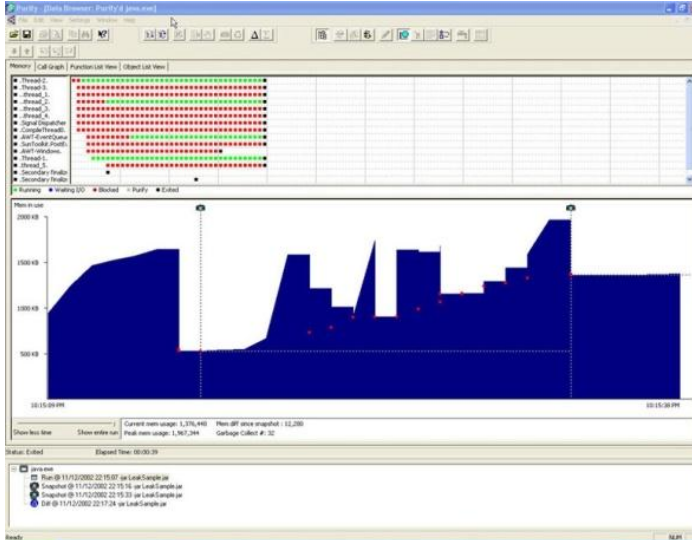
- What we want
- Existing tools
- Our proposal
- Use cases
- Conclusion

Callgrind compatibility

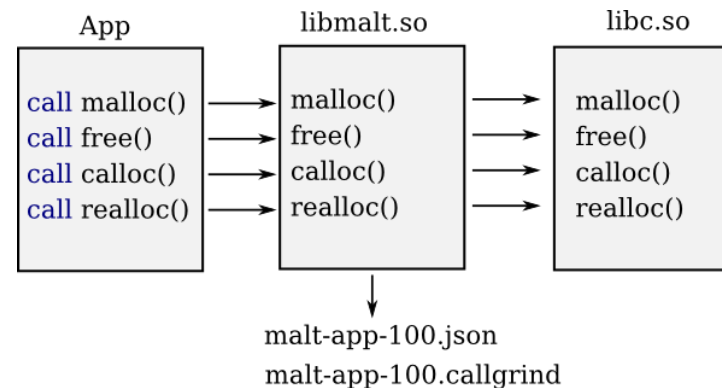
- Can use kcachgrind
- Might be usefull for some users, cannot provide all metrics.



- **IBM Purify++ / Parsoft Insure++**
 - Commercial
 - Leak detection, access checking, memory debugging tools.
 - Use binary or source instrumentation.
 - Windows / Redhat
- **Visual Studio Memory profiler**
 - Nice but windows only and commercial



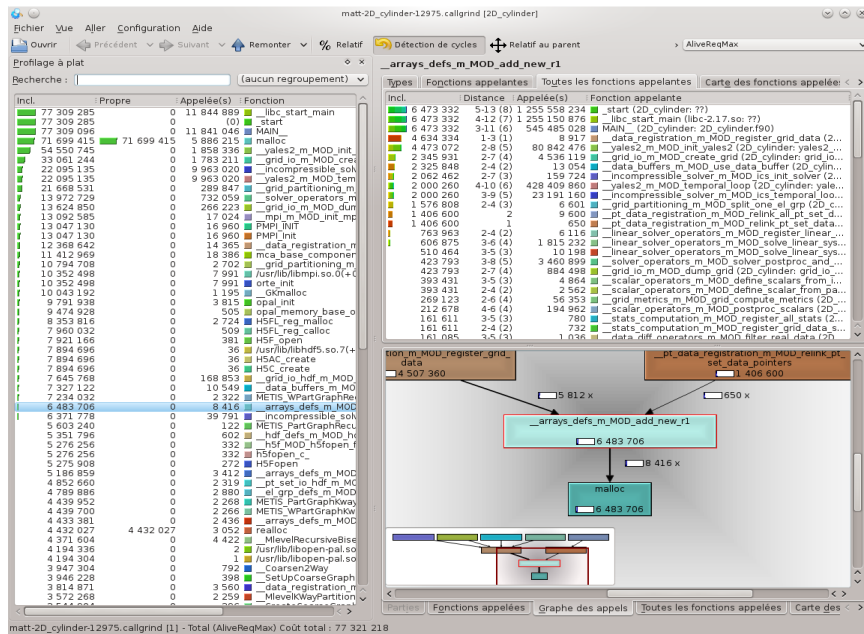
- Use **LD_PRELOAD** to intercept malloc/free/... as Google heap profiler



- Project allocations on call stacks
- Generate JSON output file
- Build profile so size is limited by call tree

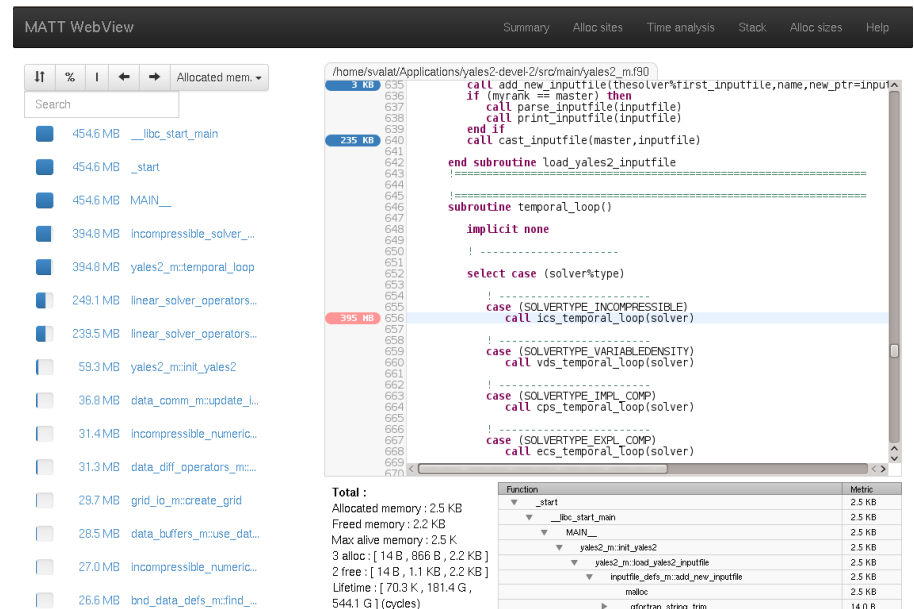
Callgrind compatibility

- Can use kcachgrind
- Might be useful for some users, cannot provide all metrics.



Own web view

- Get all metrics
- Web technology (NodeJS, D3JS, JQuery, AngularJS)
- Easier for remote usage
- Can be used for shared working



- Backtrace mode :

```
# Optionally recompile with debug flag to get source lines :  
cc -g ...  
# Run your program  
${PREFIX}/bin/malt [--config=file.ini] YOUR_PRGM [OPTIONS]
```

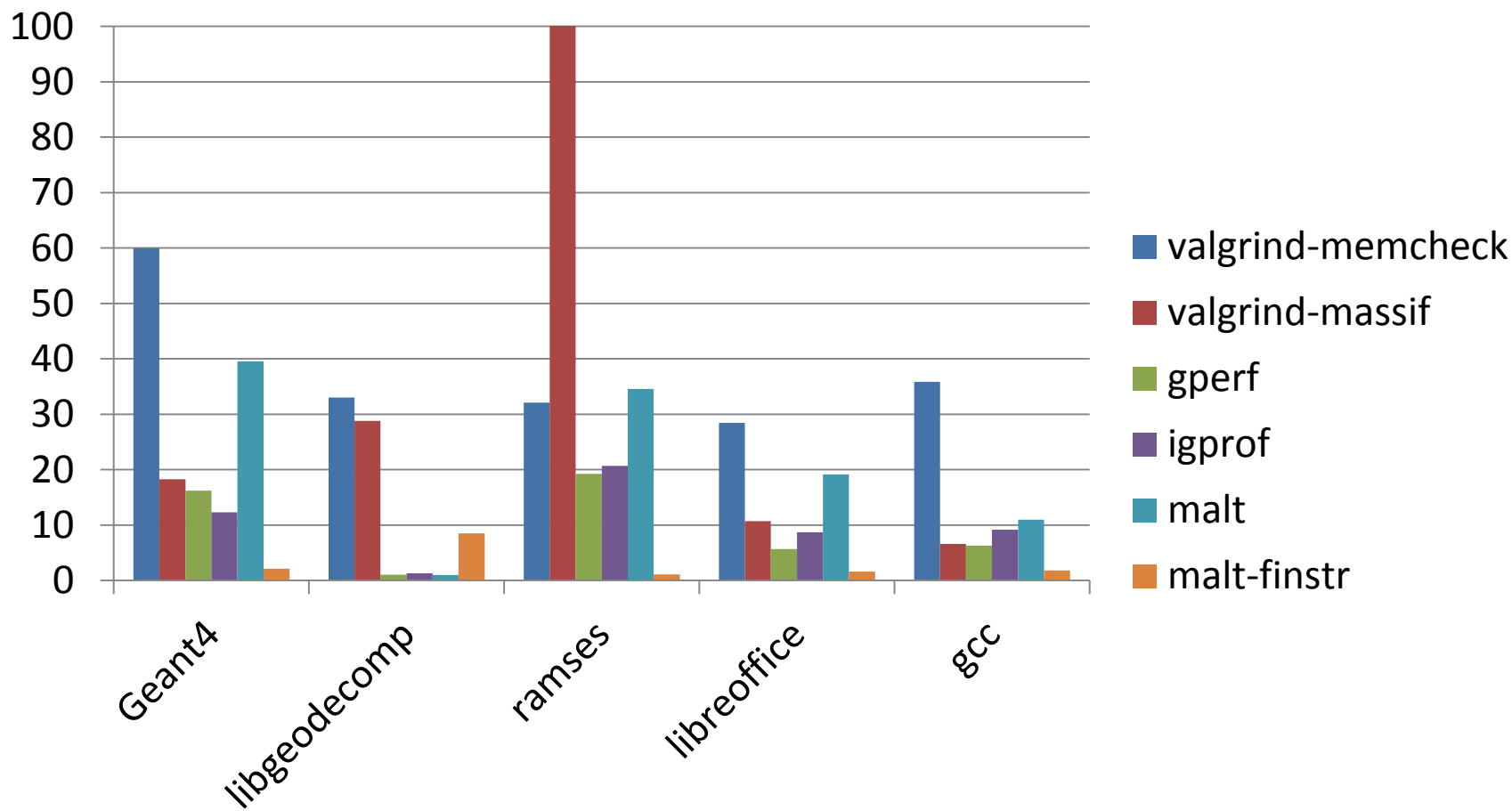
- Function tracking with -finstrument-function :

```
# Recompile with instrumentation flag :  
cc -finstrument-function -g ...  
# Run  
${PREFIX}/bin/malt --stack=enter-exit [--config=file.ini] YOUR_PRGM [OPTIONS]
```

- Use the web view :

```
#Launch the server  
malt-webserver -i malt-{YOUR_PRGM}-{PID}.json  
# Connect with your browser on http://localhost:8080
```


- Add NUMA statistics
- Provide virtual/physical ratio
- Estimate page fault costs
- Exploit traces in GUI for deeper analysis
 - Alive allocations at a certain time
 - Fragmentation analysis
 - Time charts from call sites
 - Usage over threads for call sites



EXECUTION TIME

00:00:00.25

PHYSICAL MEMORY PEAK

2.3 MB

ALLOCATION COUNT

379

AVAILABLE PHYSICAL MEMORY

4.1 Gb

Run description

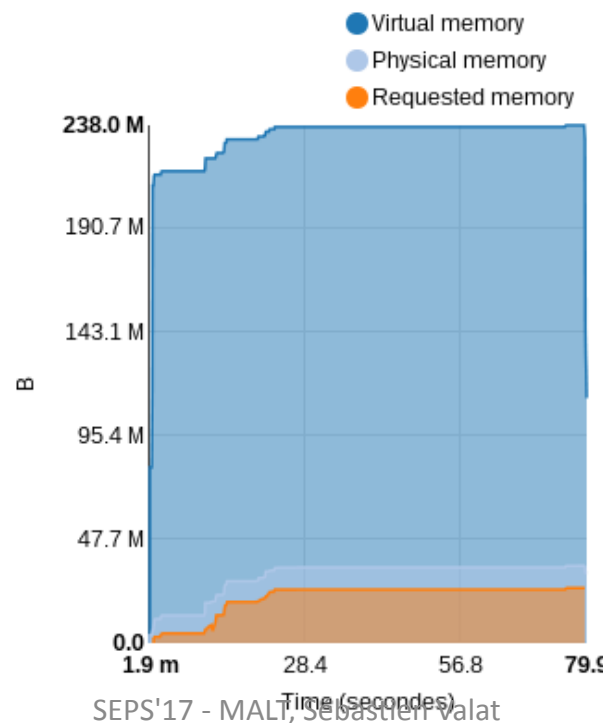
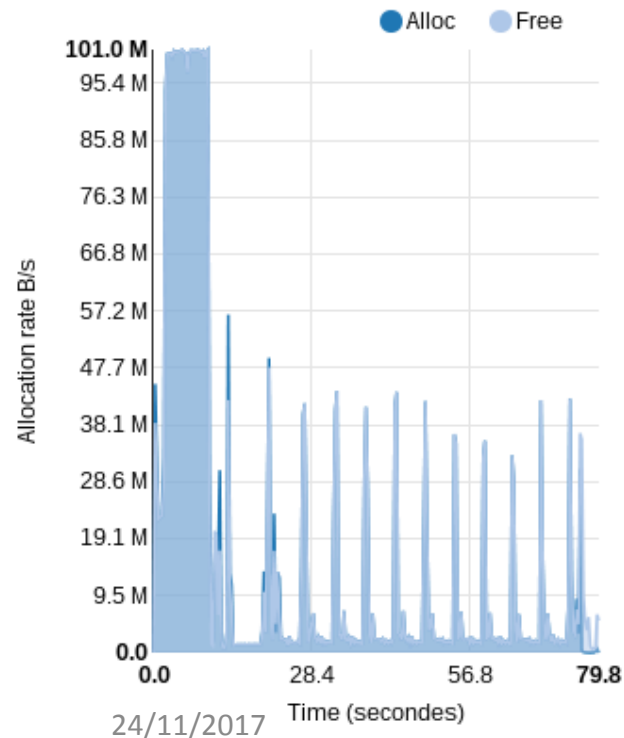
Executable :	simple-case-finstr-linked
Commande :	<code>./simple-case-finstr-linked</code>
Tool :	matt-0.0.0
Host :	localhost
Date :	2014-11-26 22:40
Execution time :	00:00:00.25
Ticks frequency :	1.8 GHz

Global statistics

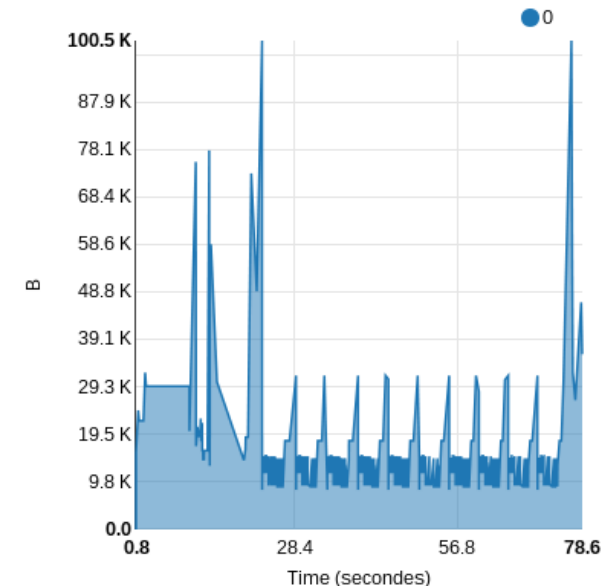
[Show all details](#) [Show help](#)

Physical memory peak	2.3 MB
Virtual memory peak	103.7 MB
Requested memory peak	2.3 MB

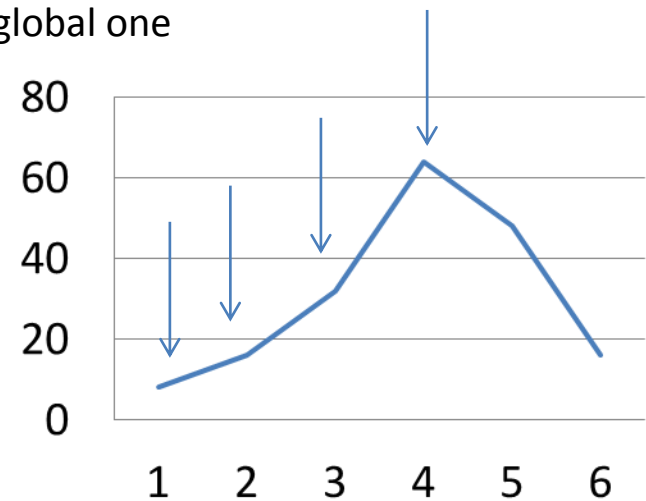
- **Profile over time :**
 - Allocation rate
 - **Physical / Virtual / Requested** memory
 - **Stack size** for each **thread** (require function instrumentation)
- **Example on YALES2 with gfortran :**



2.1 54.0 KB 29.8 KB 23.6 7.9
Kt grid_io_hdf_m::dump_grid_data_to_hdf grid_io_hdf_m::dump_KKE KB



- The tool maintain a **call stack tree**
- Profile **stats on leafs**
- On **new global peak**, need **to copy** each **local current contribution**
- Need to **walk over** the wall **tree** each time ?
- **Do lazy update :**
 - Keep track of **last local peakId** on each leaf
 - On leaf update, compare the **local peakId** and the global one
 - If not same : remember the old local contribution



Display largest stack for thread ID

MATT WebView

Summary

Alloc sites

Time analysis

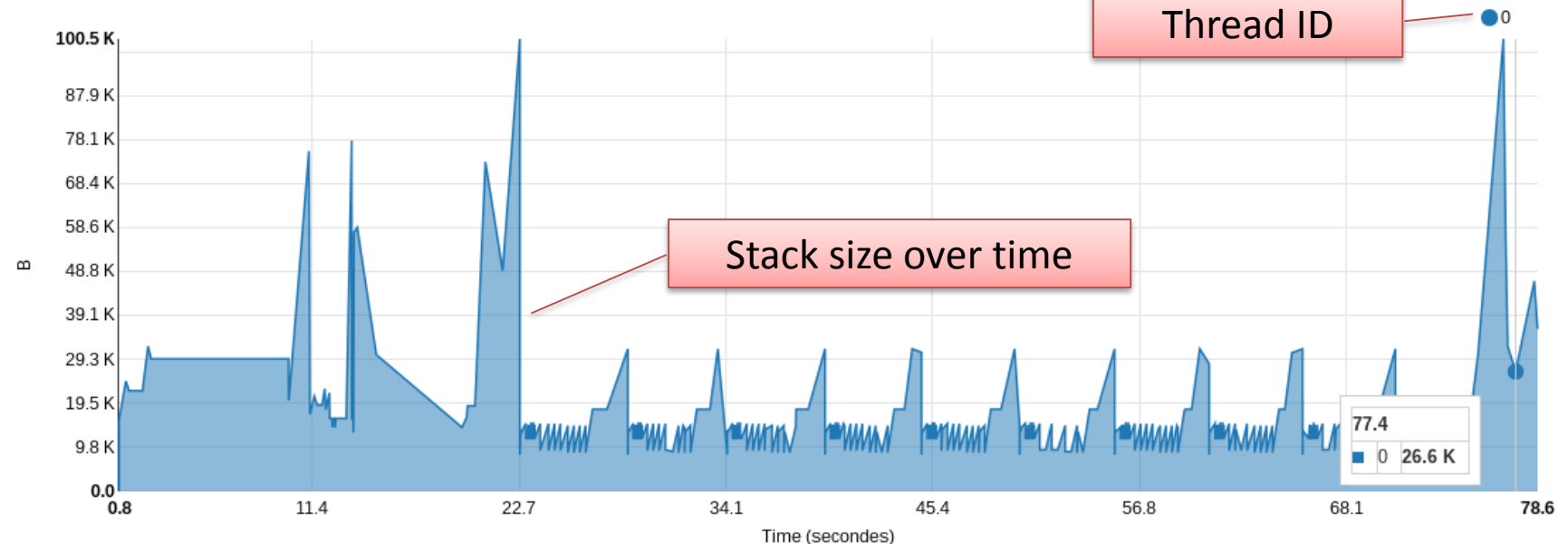
Stack

Alloc sizes

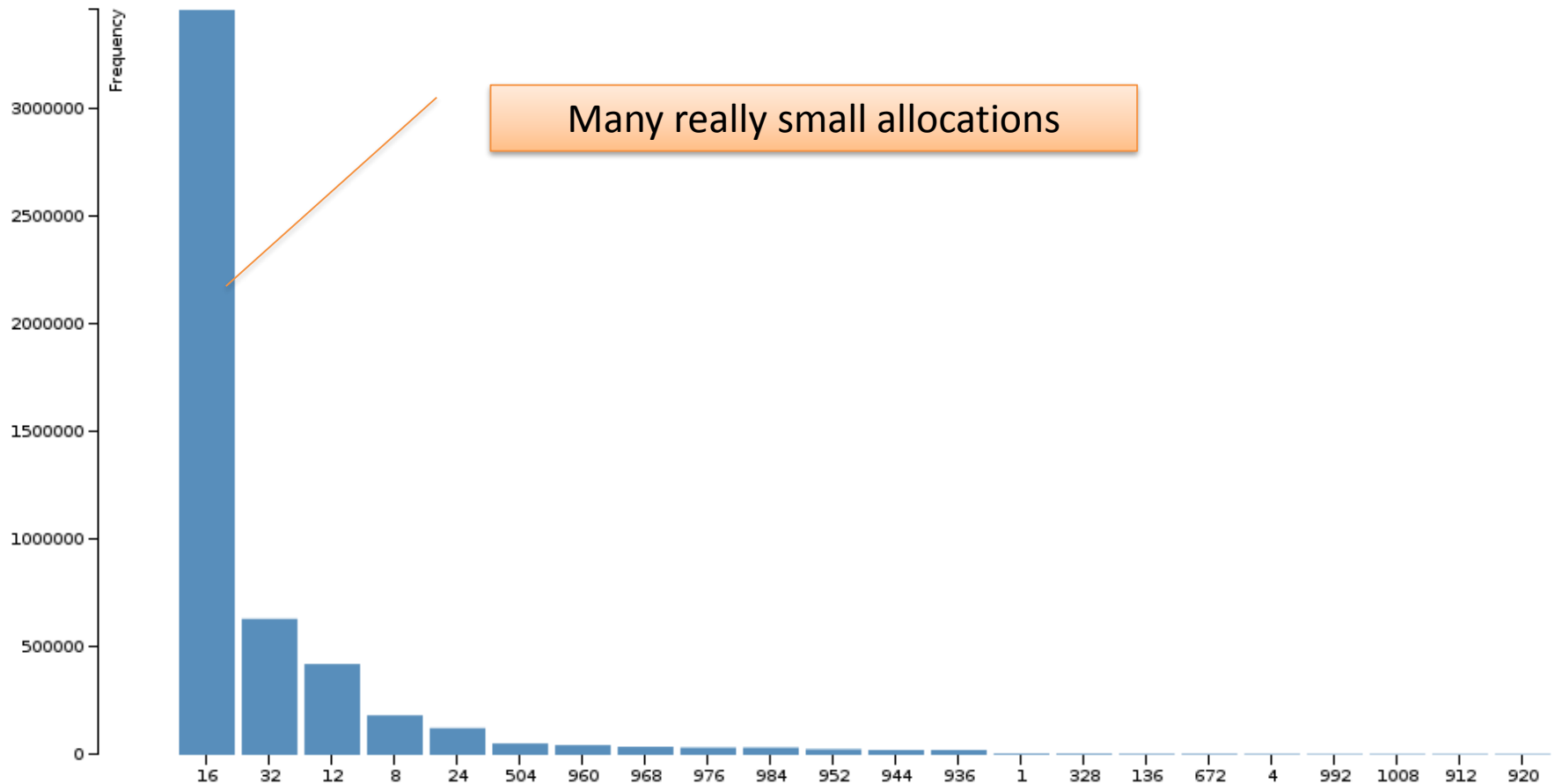
Help

Thread ID : 0

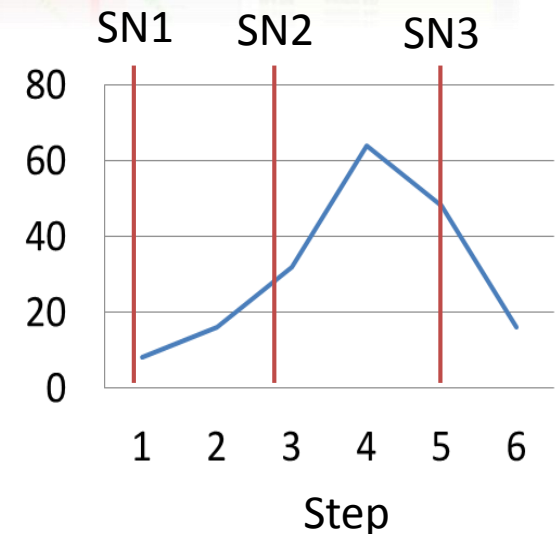
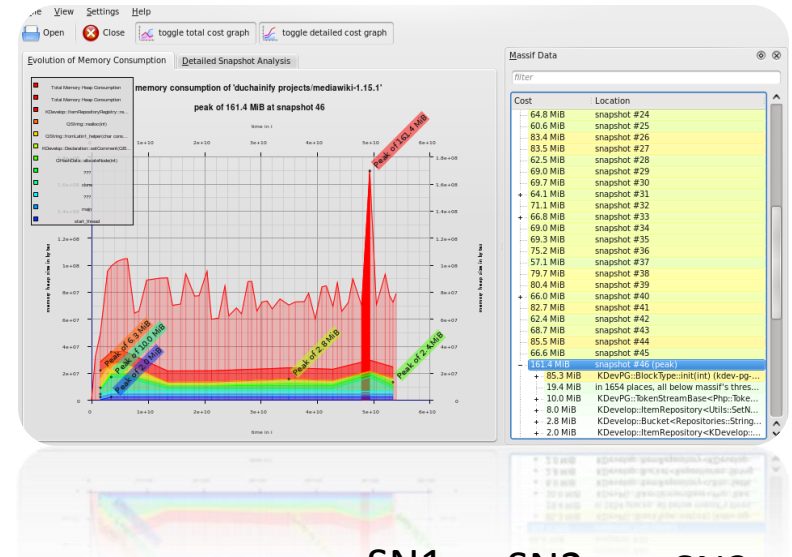
Stack space used by functions on peak



Example from YALES2 with gfortran issue

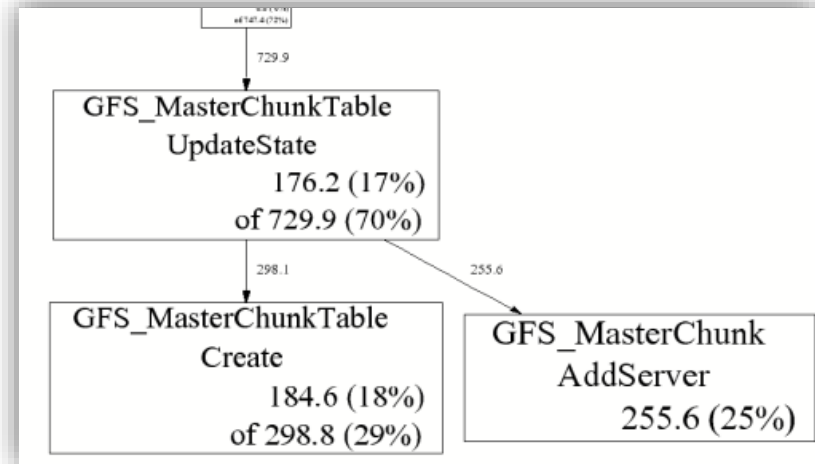


- **Valgrind - massif :**
 - Link memory size to functions
 - Take **snapshots** over time.
 - Miss short live allocations
 - Oly interested in memory size
 - Slow, not parallel.
- **Valgrind - memcheck :**
 - Leak detection
 - Slow, not parallel.



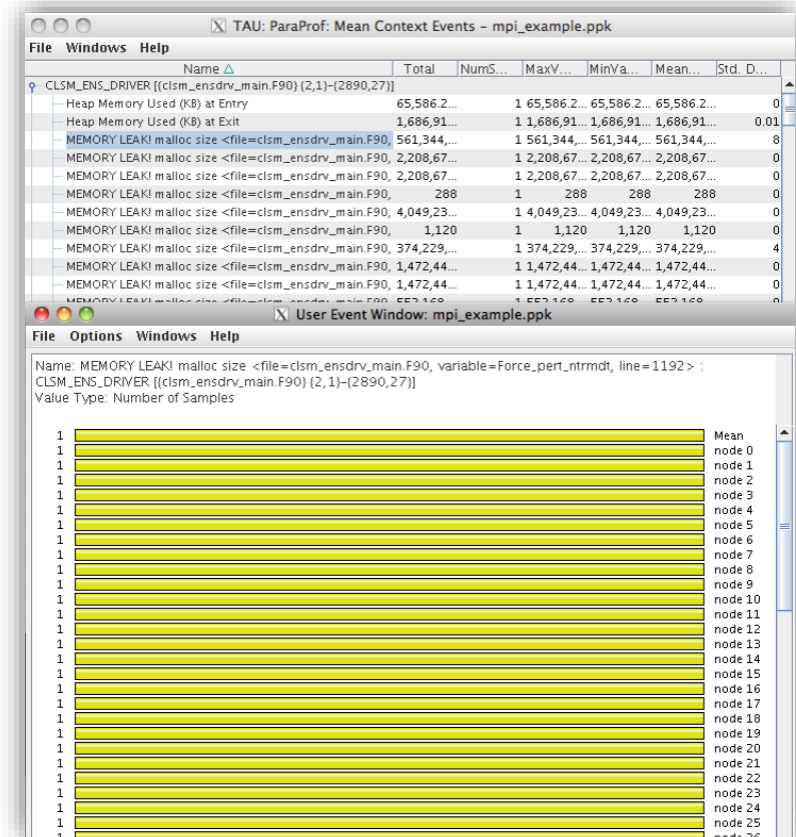
- Google heap profiler (tcmalloc):

- Small overhead.
- Similar metric than massif
- Only provide **snapshots** of **allocated memory per stacks**.
- Peak might not be captured.
- **Lack of a real GUI to use it.**



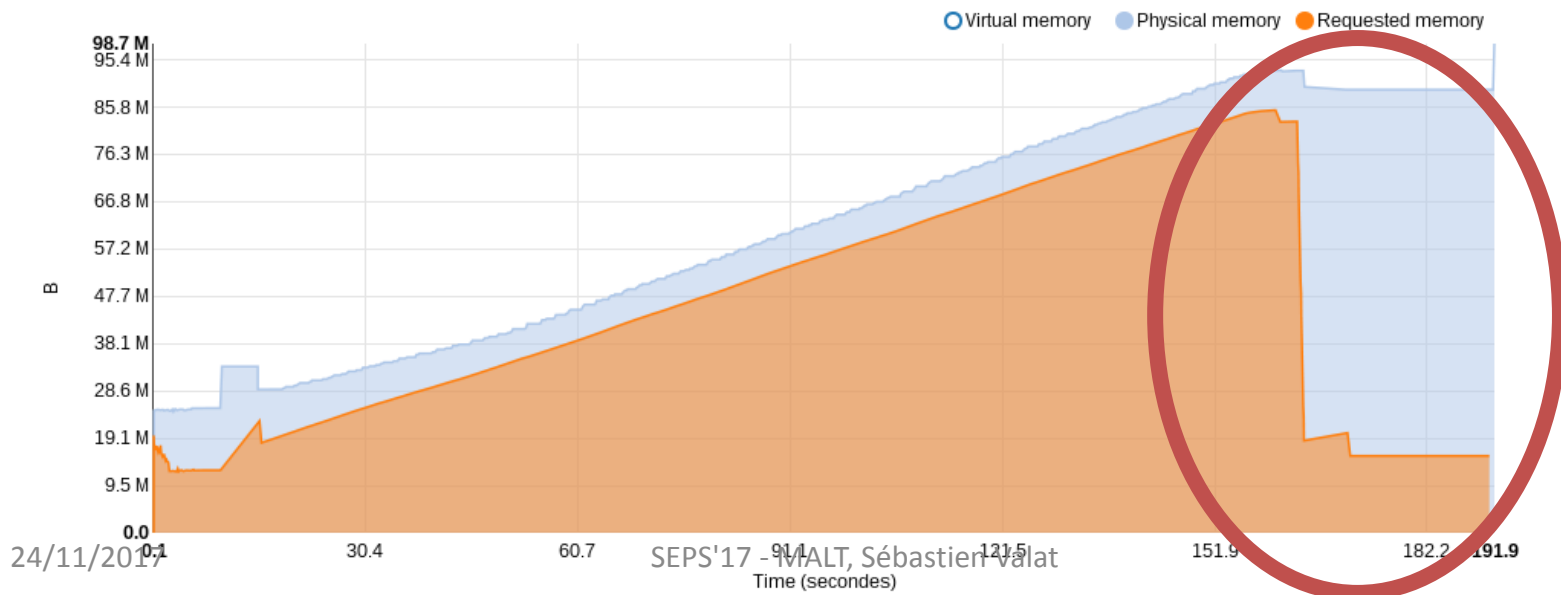
```
% pprof gfs_master profile.0100.heap
255.6 24.7% 24.7% 255.6 24.7% GFS_MasterChunk::AddServer
184.6 17.8% 42.5% 298.8 28.8% GFS_MasterChunkTable::Create
176.2 17.0% 59.5% 729.9 70.5% GFS_MasterChunkTable::UpdateState
169.8 16.4% 75.9% 169.8 16.4% PendingClone::PendingClone
76.3 7.4% 83.3% 76.3 7.4% __default_alloc_template::_S_chunk_alloc
49.5 4.8% 88.0% 49.5 4.8% hashtable::resize
```

- **TAU memory profiler**
 - Provide **profiles** (not snapshots)
 - Provide leaks
 - Done for HPC/MPI
 - **Lack easy matching with sources**

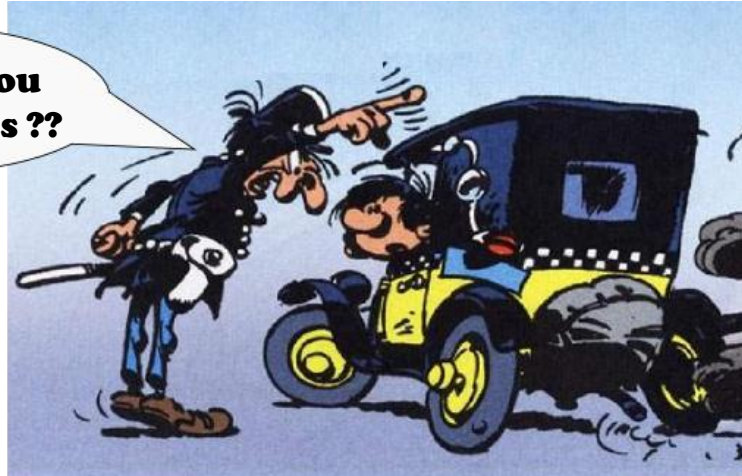


- Example from **Dassault mini-app** from Loïc Thébault and Eric Petit.
- **Fragmentation** can **prevent** from **returning physical pages** to OS
- **Solution** : **avoid interleaved** allocation of chunks with **different lifetime**.
- We observed with the **source annotation** that **most of them can be avoided**.

Memory allocated over time



**Who give you
this address ??**



Thank you.

QUESTIONS ?

