MALT : MALloc Tracker

A memory profiling tool

THE QUESTION

Questions

- 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** ?

Some issue examples

- 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

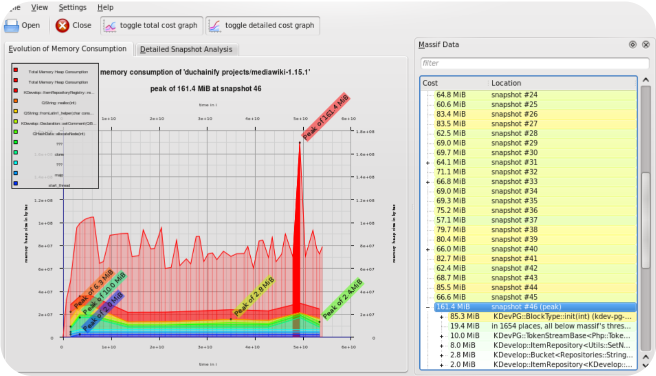
Might lead to swap for large size

Short life allocations

EXISTING TOOLS

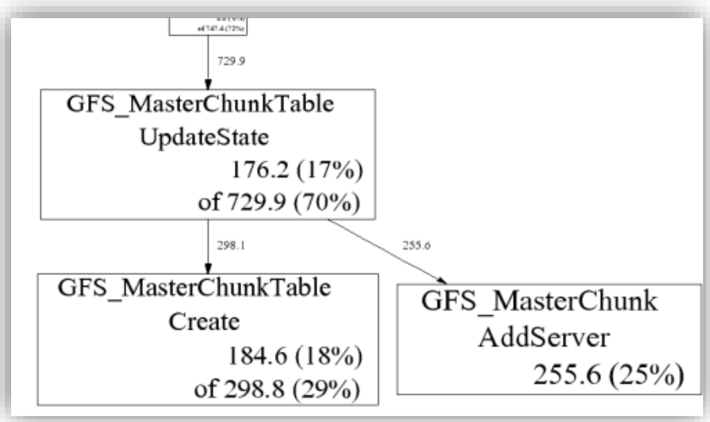
Existing tools

- Valgrind (massif)
  - Memory **over time** (snapshots) & **functions**
  - Memory per function **at peak**
  - Has a simple GUI



- Valgrind (memchek)
  - **Leaks**
  - No real GUI

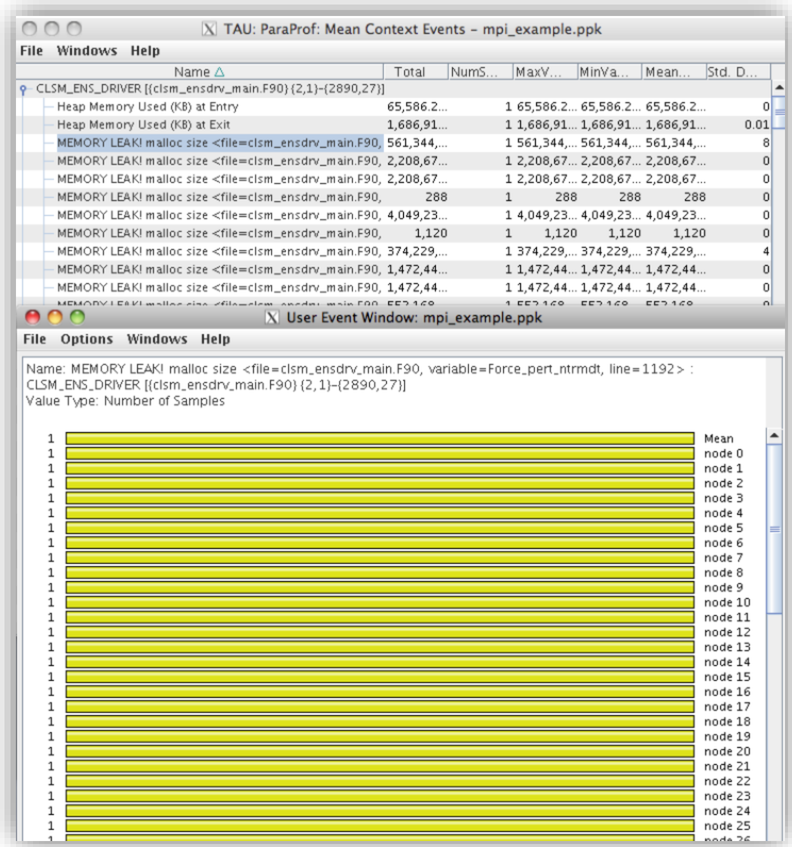
- Google heap profiler (tcmalloc)
  - Memory **over time** (snapshots)
  - Faster then valgrind
  - No GUI



Existing tools

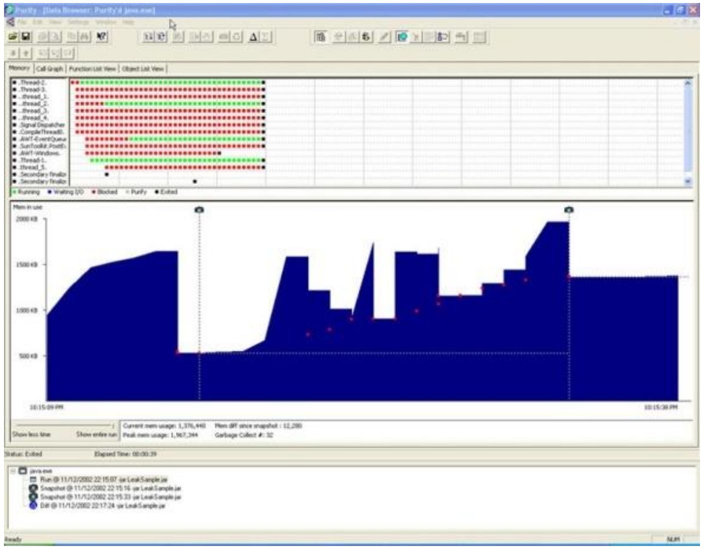
- **TAU memory profiler**
  - Provide profiles
  - Follow stacks
  - Track leaks
  - Parallel, done for HPC/MPI
  - Lack easy matching with sources

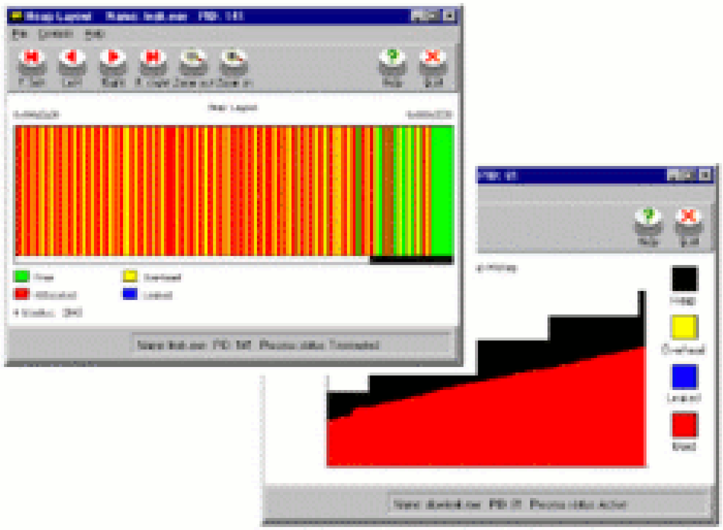
- **FOM**



Existing tools / Commercials

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





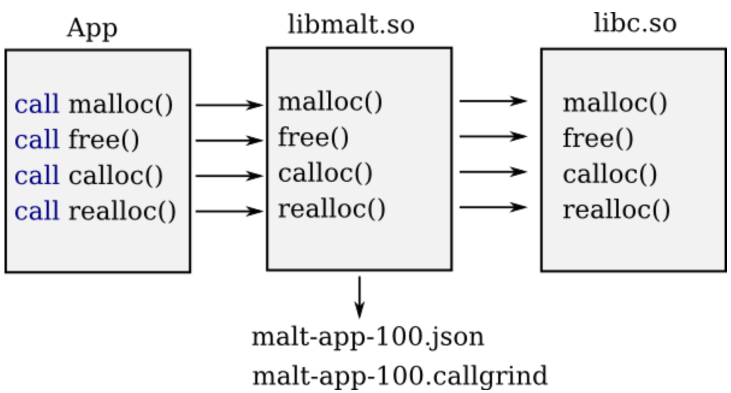
WHERE I GO

What I want to provide

- 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)

How it works

- 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

Stack tracking

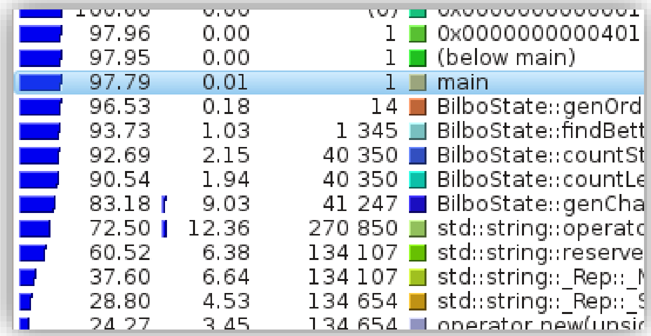
- Two approach implemented : **backtrace** and **instrumentation**

- **Backtrace** (default) :
  - Work out of the box
  - Manage all dynamic libraries
  - **Slow** for **large number of calls** (~>10M)

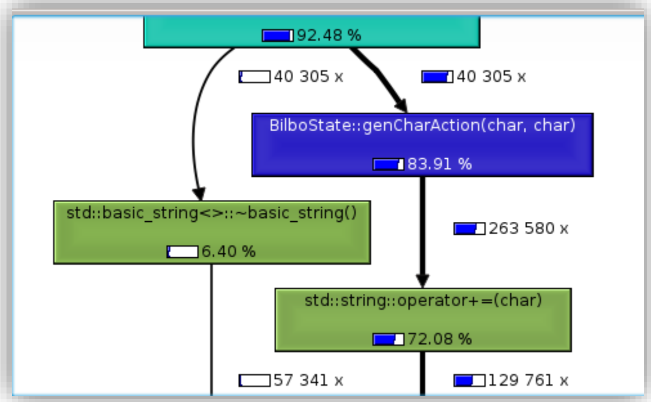
- **Instrumentation :**
  - Need source **recompilation** (available) : *-finstrument-function*
  - Or tools for **binary instrumentation** : MAQAO / Pintool (experimental)
  - Faster for really large number of calls to malloc
  - **Only** provide stacks for the **instrumented** binaries

What is good in kcachegrind

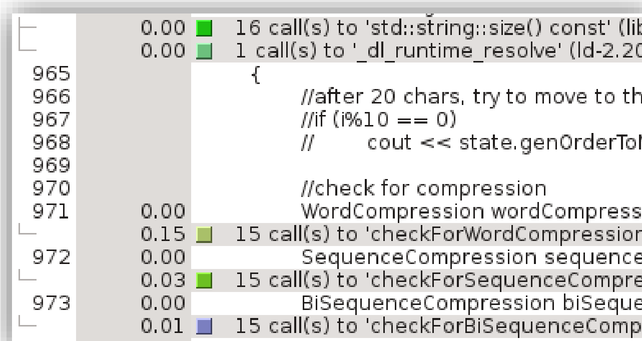
- List of **functions** with **exclusive/inclusive** costs



- Nice **call tree**



- **Annotated** sources



What is missing to kcachegrind

- **Started** with **kcacegrind** GUI…. But …

- Display **human readable** units
  - You prefer **15728640** or **15 MB** ?
  - I want to **compare to what I expect**.

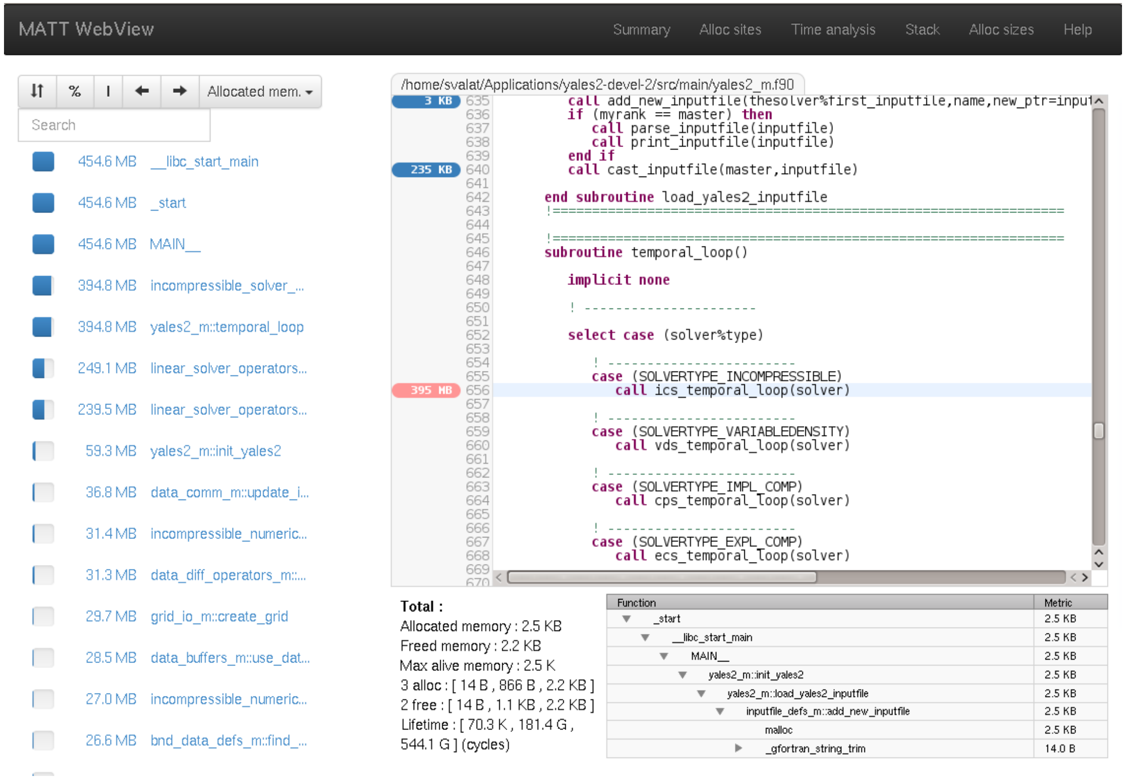
- Cannot handle **non sum cumulative metrics**
  - **Inclusive** costs **only rely** on **+ operator**
  - Some mem. metrics **requires max/min** (eg. lifetime)

- No way to express **time charts**

- No way to express **parameter distributions** (eg. sizes).

Our GUI : based on web tech.

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



SOME VIEWS

Global summary

- 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

Global summary : top 5 functions

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

Alloc count

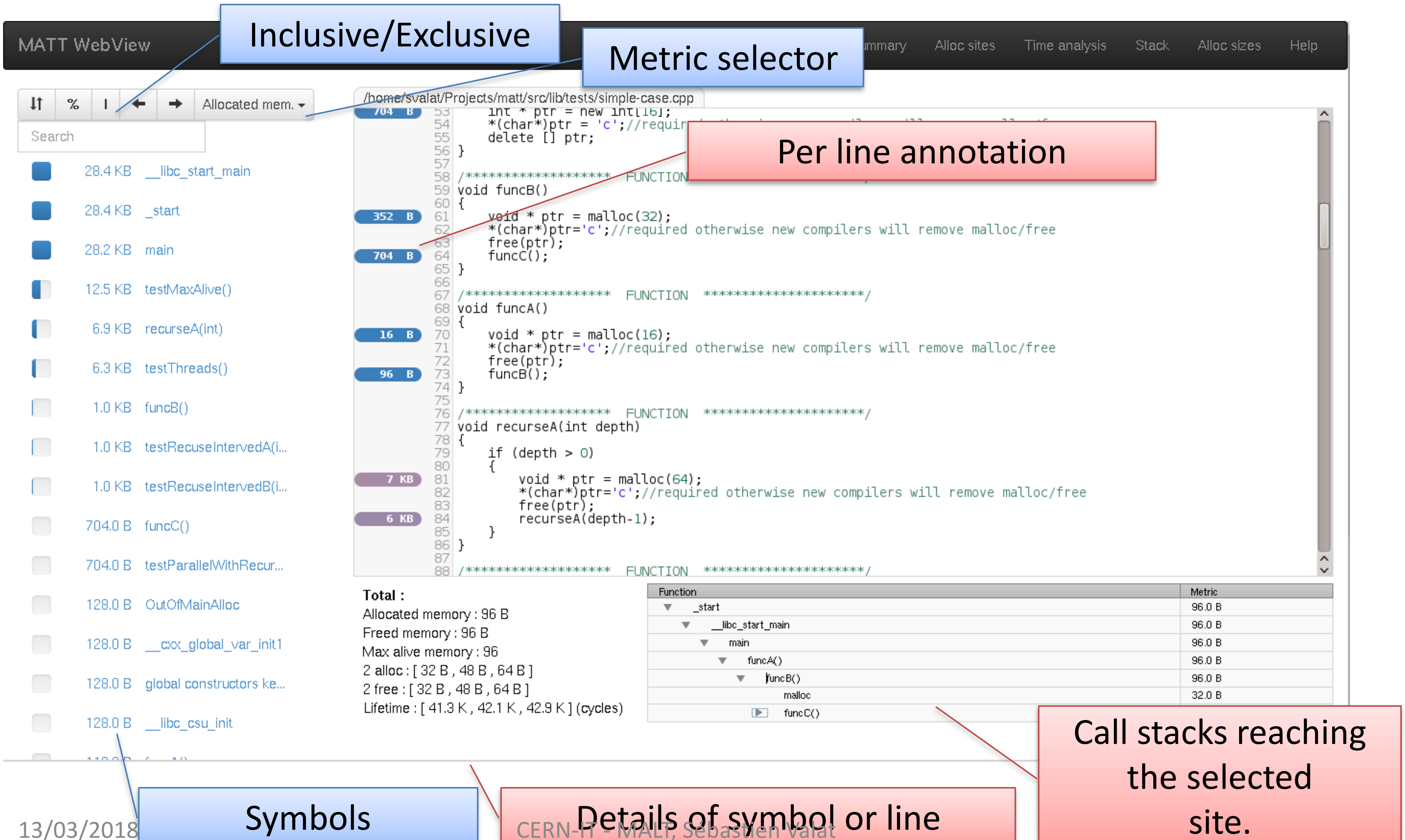
Ratio	Allocs	Function
	911.9 K	data_comm_m::copy_int_comm_to_data
	896.4 K	data_comm_m::copy_data_to_int_comm
	853.2 K	data_comm_m::update_int_comm
	484.9 K	sponge_layer_m::calc_sponge_layer_mask
	296.0 K	incompressible_numerics_m::ics_diffuse_velocity_rk_4th

Allocated memory

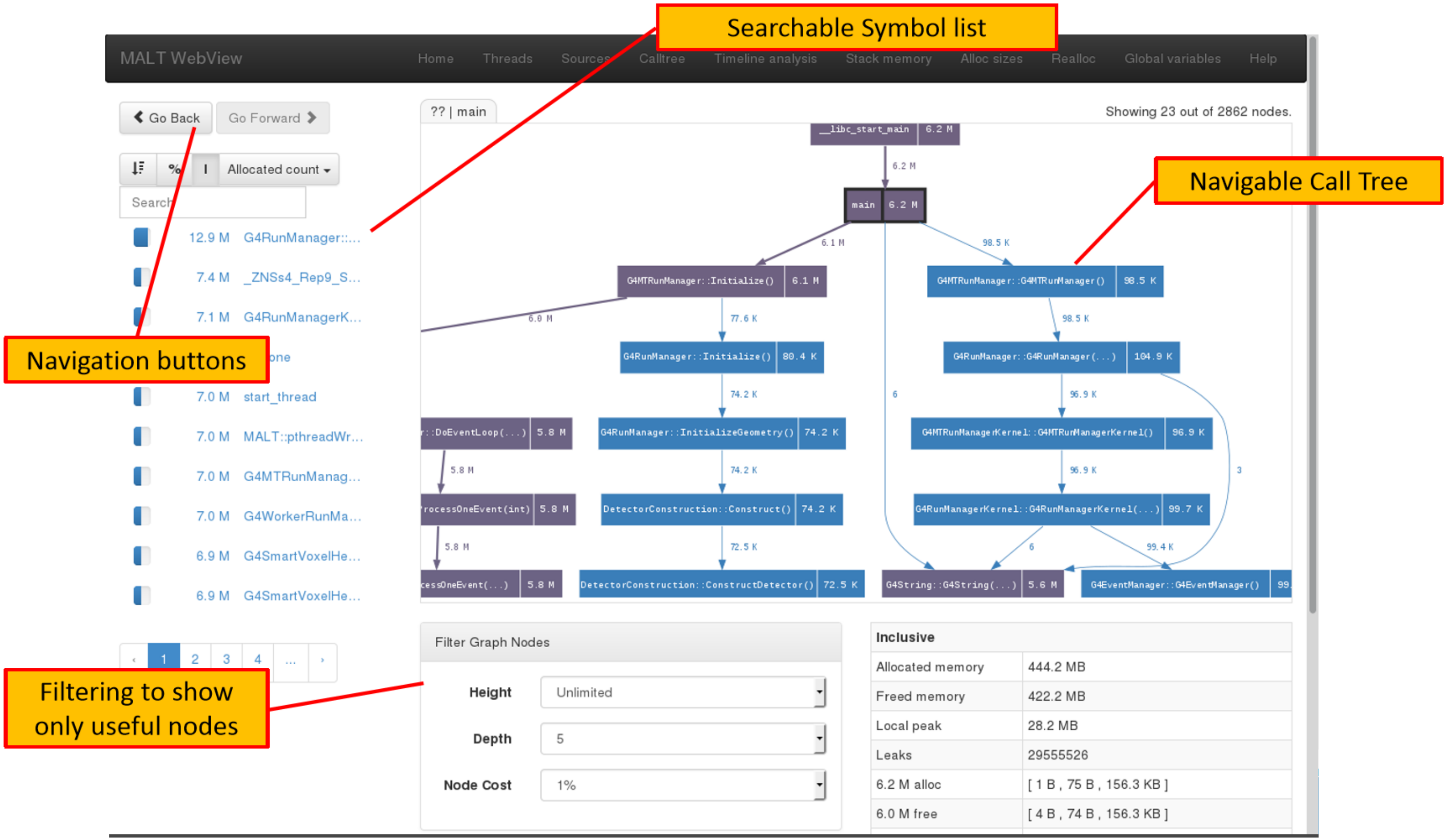
Ratio	Allocs	Function
	202.4 MB	linear_solver_operators_m::solve_linear_system_deflated_pcg
	26.6 MB	bnd_data_defs_m::find_bnd_data
	21.8 MB	linear_solver_operators_m::solve_el_grp_pcg
	19.0 MB	data_comm_m::copy_int_comm_to_data
	18.1 MB	data_comm_m::update_int_comm

Peak memory

Source annotations

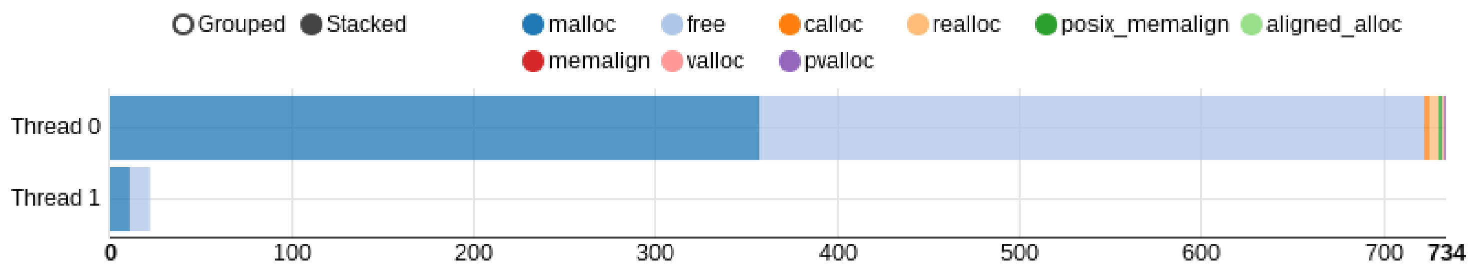


Call tree view

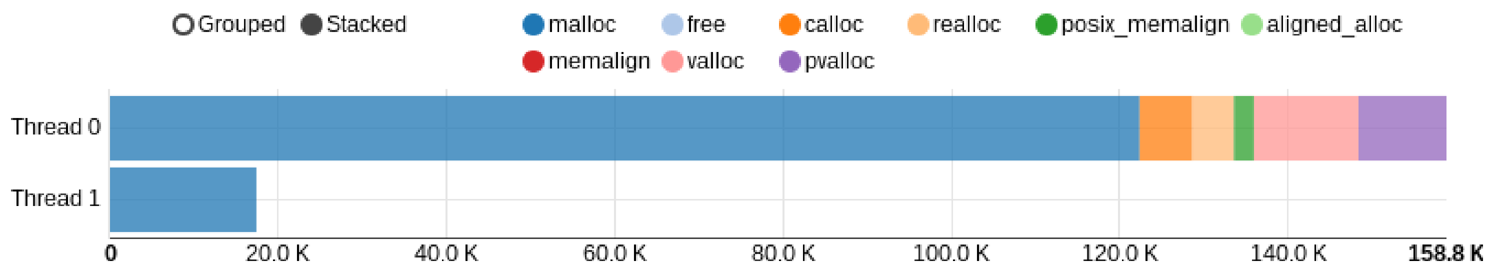


Per thread statistics

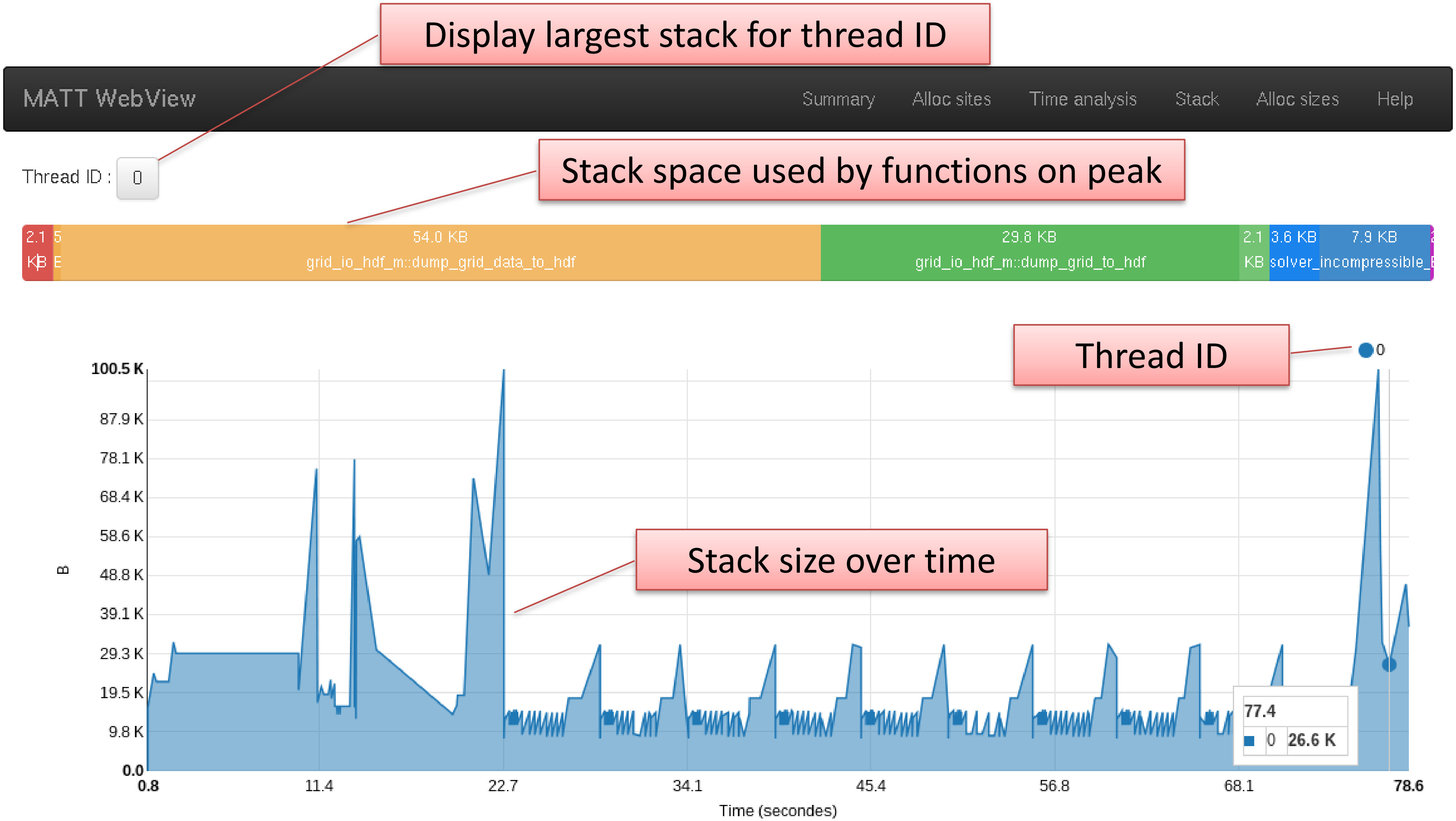
Call per thread



Time per thread

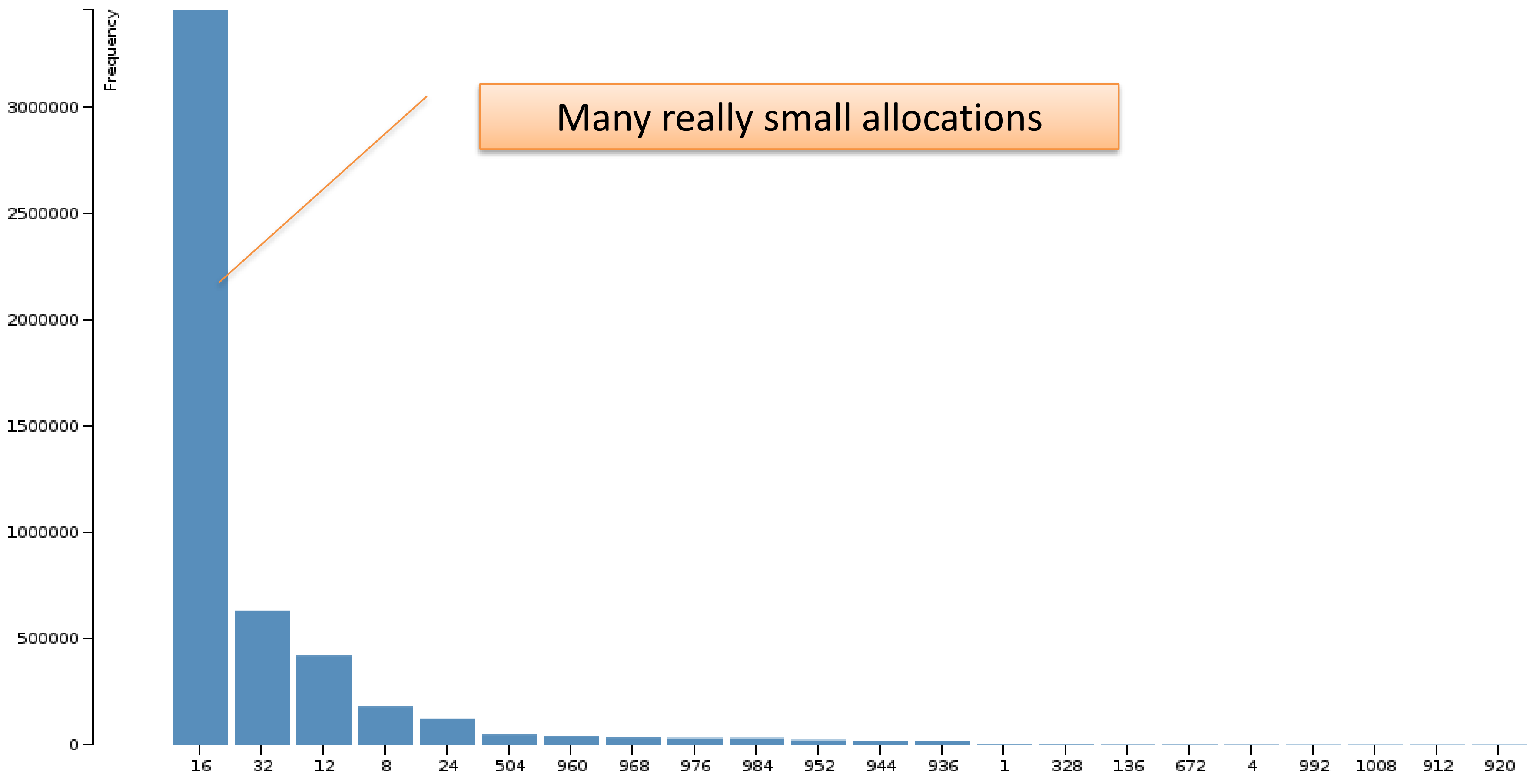


Tracking stack memory

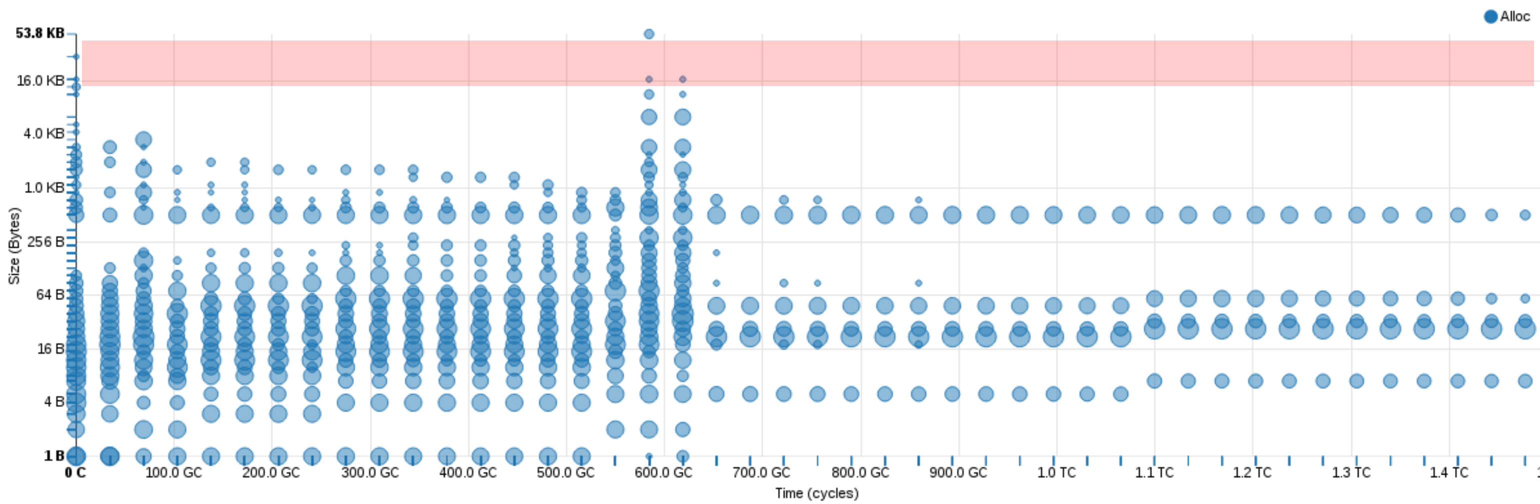


Chunk size distribution

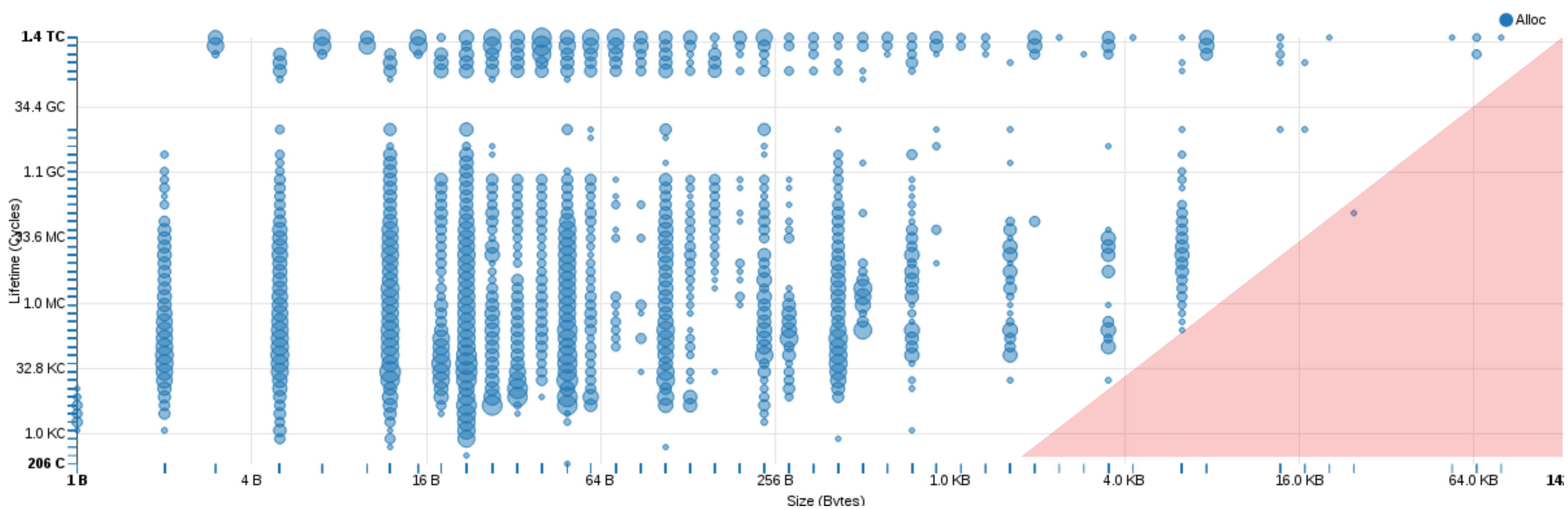
Example from YALES2



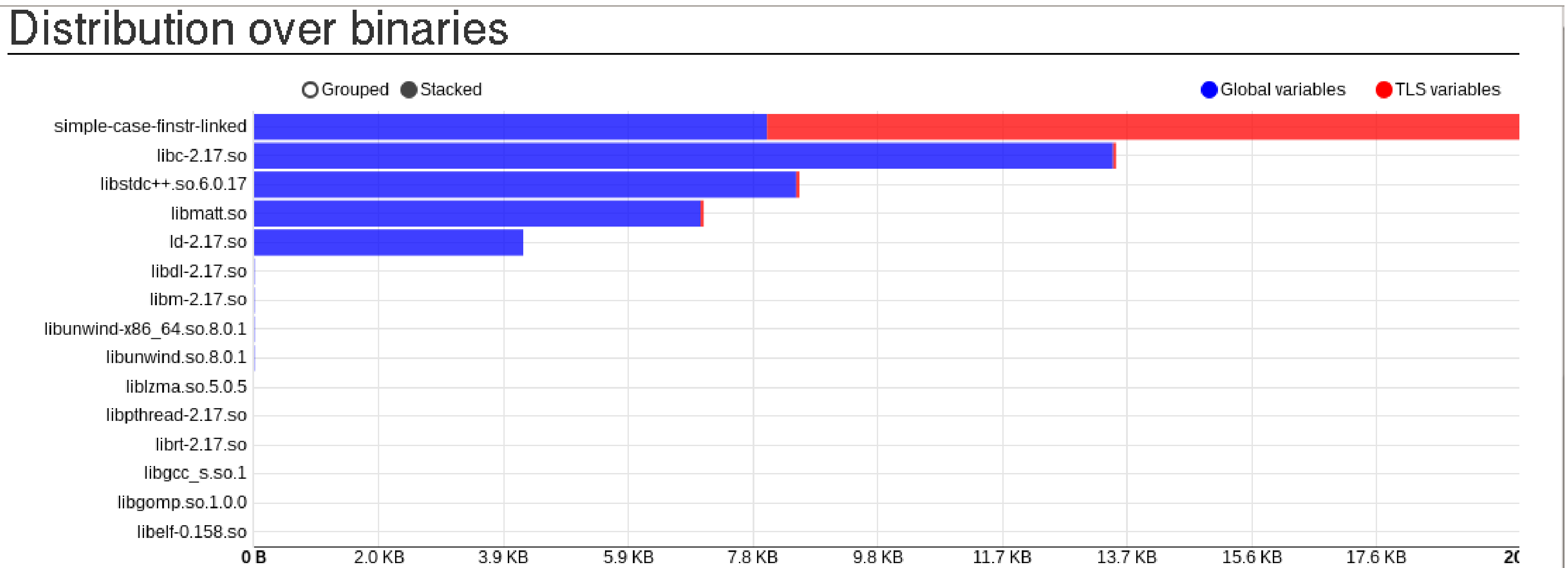
Size over time

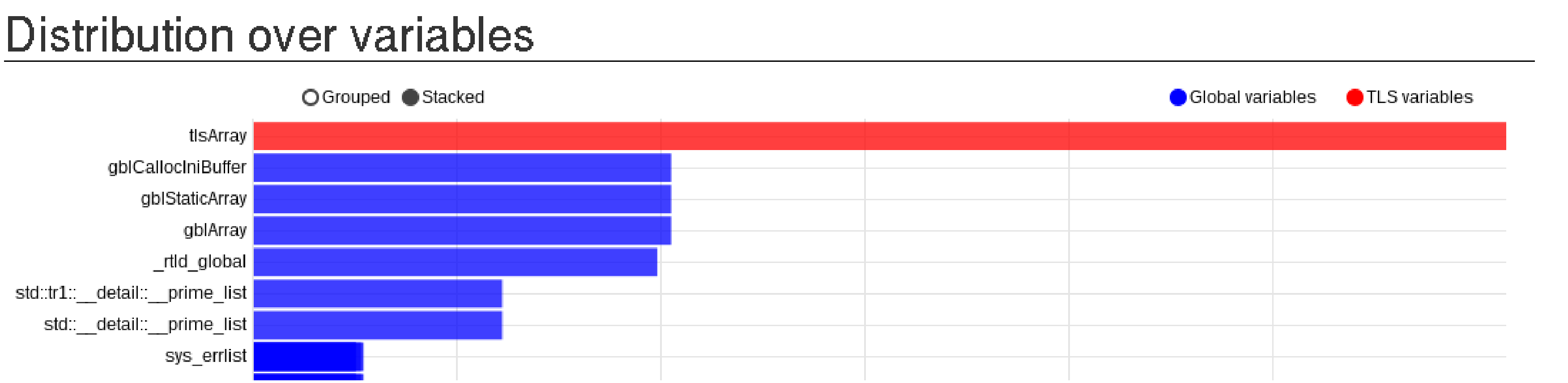


Lifetime over size



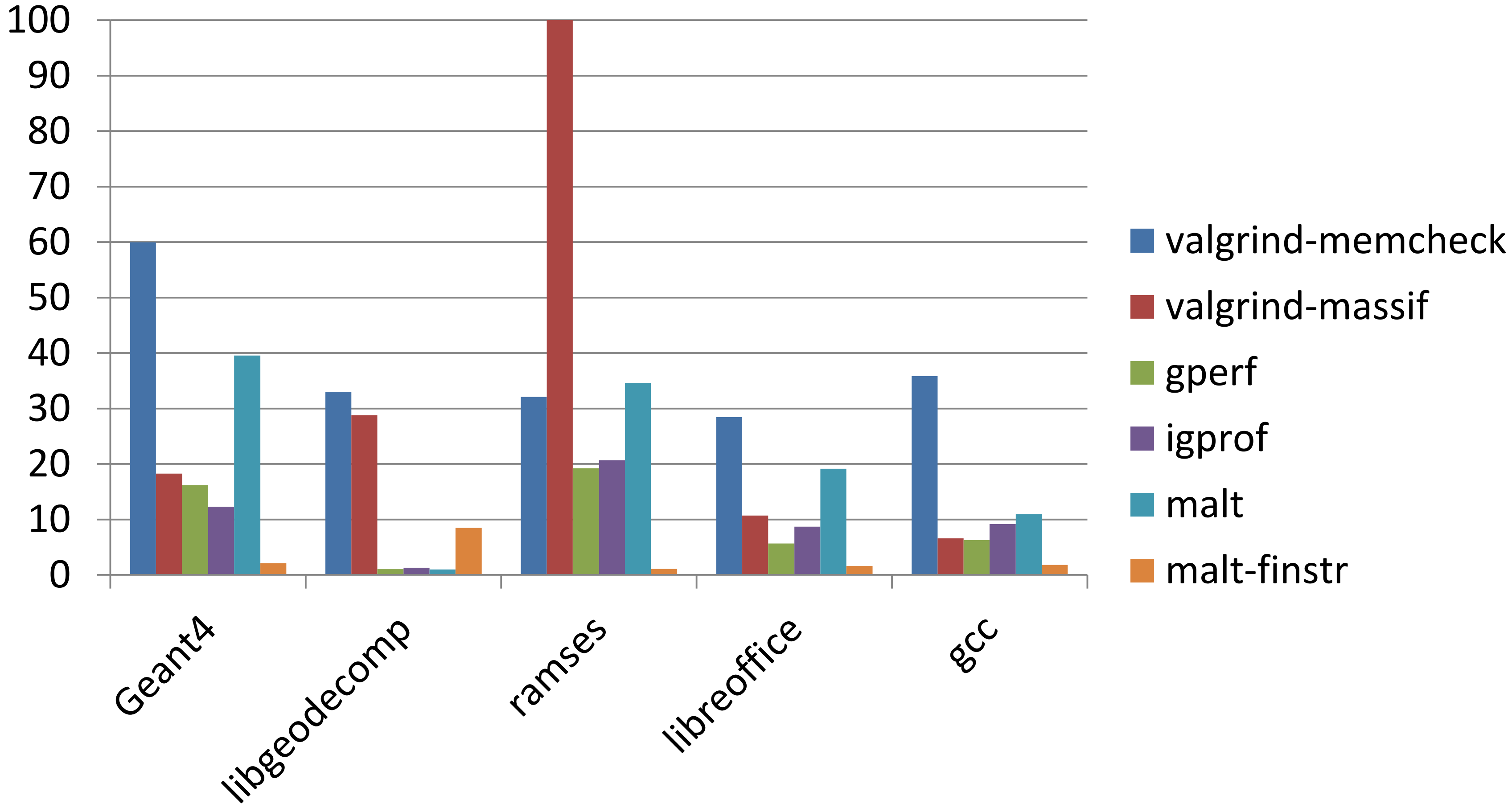
Global variables





REAL CASES

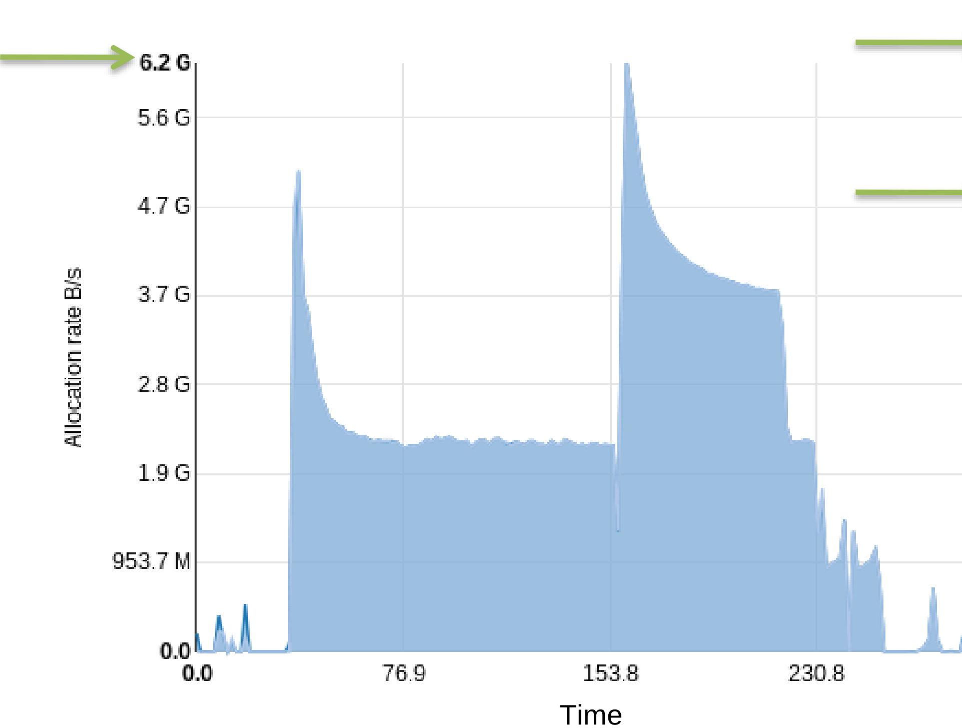
Performance



Example on AVBP init phase

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

Allocation rate



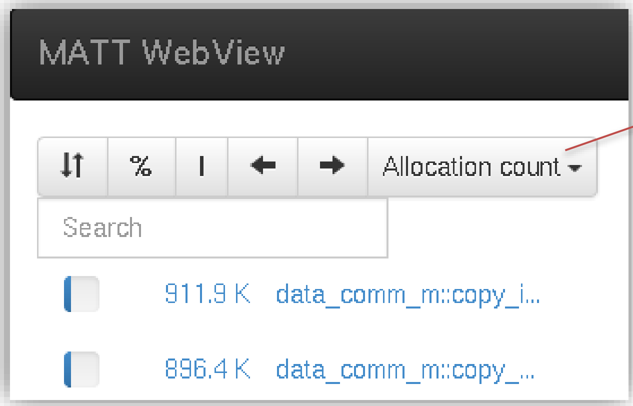

```
CALL assert(capacity==size(array),&
      'array and capacity variable are not

IF (needed_size>capacity) THEN
  IF (ALLOCATED ( temp) ) DEALLOCATE(temp)
  ALLOCATE ( temp(capacity))                 ! 57 GB

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

  DEALLOCATE ( array)
  ALLOCATE ( array(new_cap))                 ! 57 GB

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

  capacity=new_cap
END IF
```

Function
_start

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)

Allocatable arrays on YALES2

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



Search intensive alloc functions

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

608 K

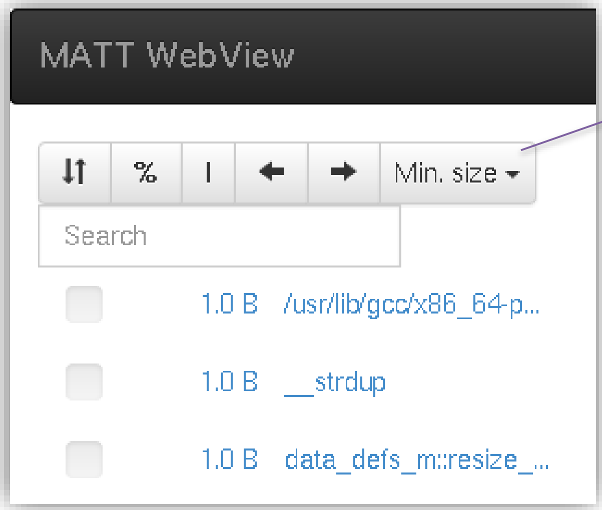
```
do i=1,nitem_el_grp
   el_grp_ind = el_grp_index2int_comm_index%val(1,i)
   int_comm_ind = el_grp_index2int_comm_index%val(2,i)
   el_grp_r2%val(1:dim1,el_grp_ind) = int_comm_r2%val(1:dim1,int_comm_ind)
end do
```

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 :
Allocated memory : 9.5 MB

And mostly really small allocations !

We can found allocs of 1B !

- Examples on YALES 2, small allocations :



Search for the minimal chunk size.

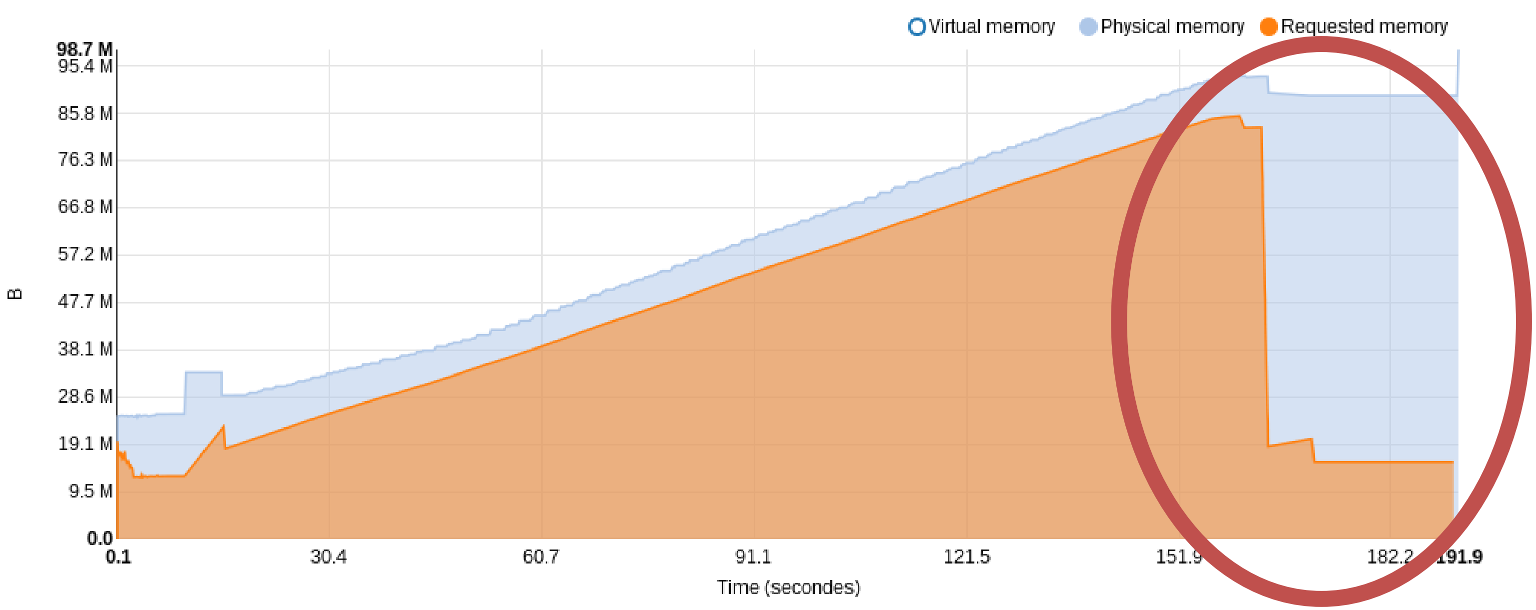
Many codes produce allocations of 1B.
OK with moderation.

```
          case (DATATYPE_REAL_NODE_VECTOR,DATATYPE_REAL_ELEM_VECTOR, &
                DATATYPE_REAL_FACE_VECTOR,DATATYPE_REAL_PAIR_VECTOR)
             if (associated(data_ptr%r2_ptrs)) then
                deallocate(data_ptr%r2_ptrs)
             end if
1 B          allocate(data_ptr%r2_ptrs(nel_grps))
             do n=1,nel_grps
                NULLIFY(data_ptr%r2_ptrs(n)%ptr)
             end do
```

Fragmentation issue

- 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



Conclusion

- 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** infos (at lease statistics about usage)
  - Is now **Open Source** : http://memtt.github.io/

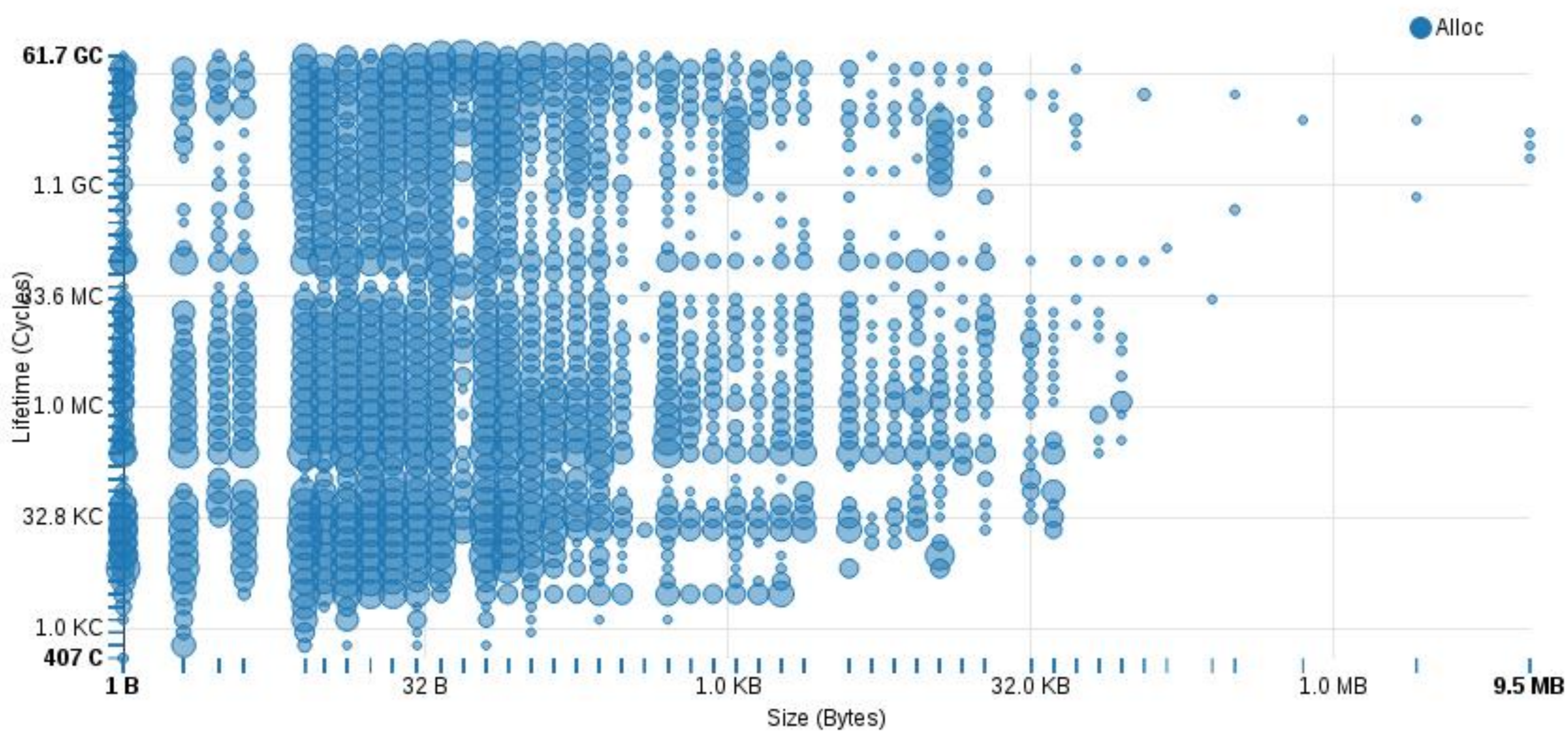
Thank you.

QUESTIONS ?

BACKUP

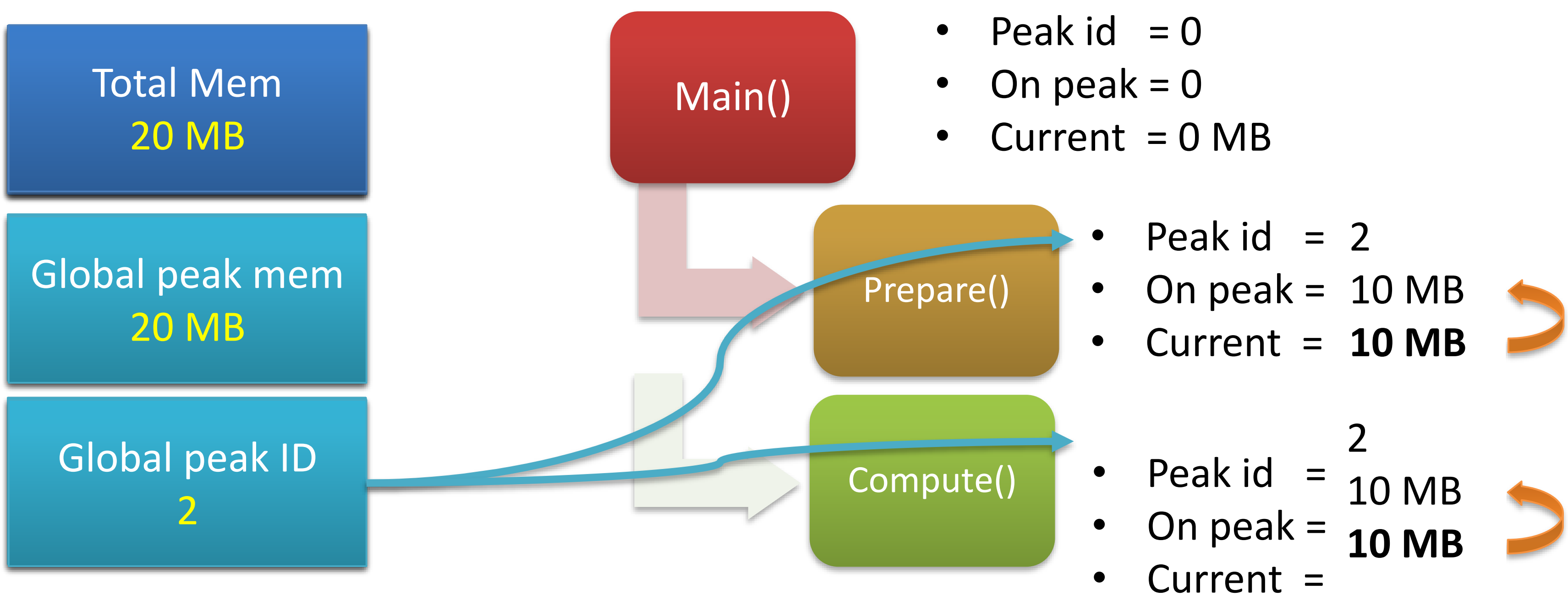
Scatter plot

Lifetime over size



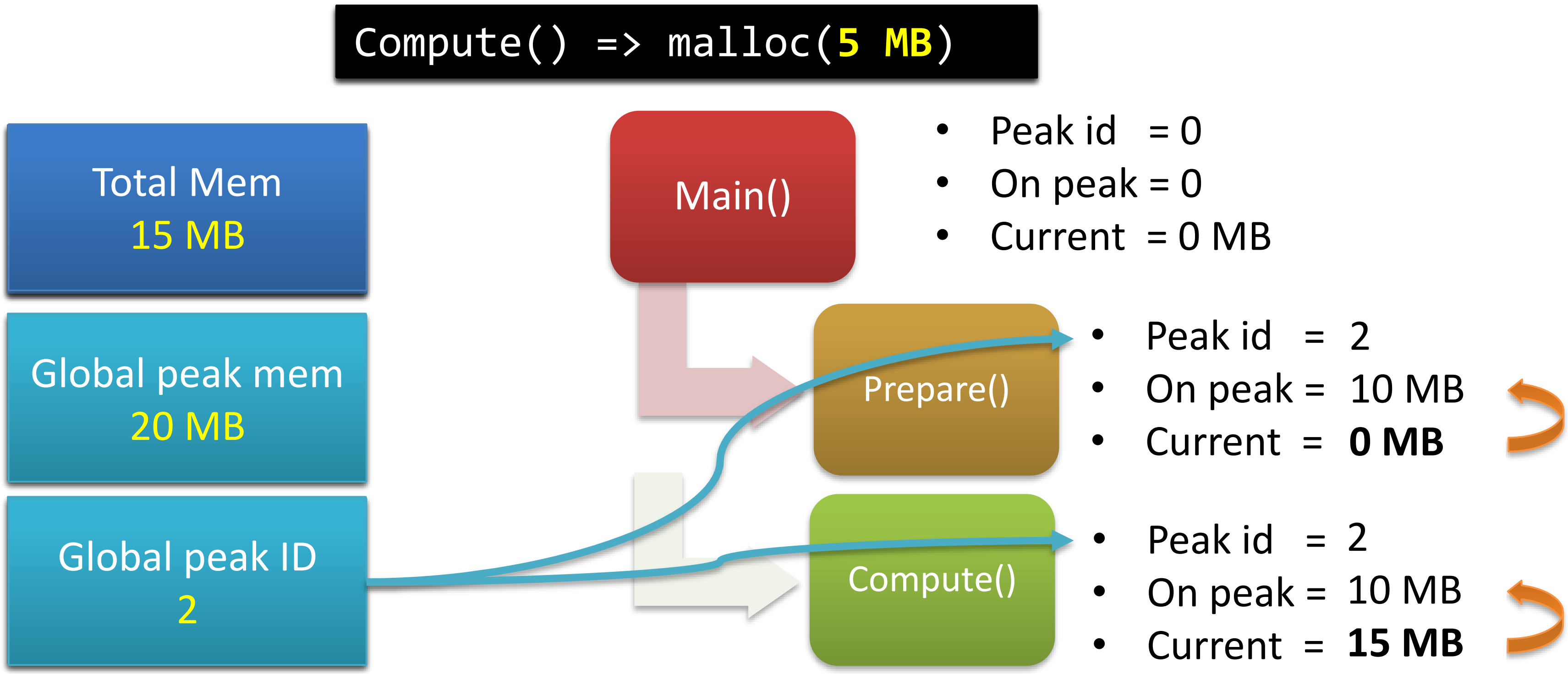
Memory peak extraction

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



Memory peak extraction

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



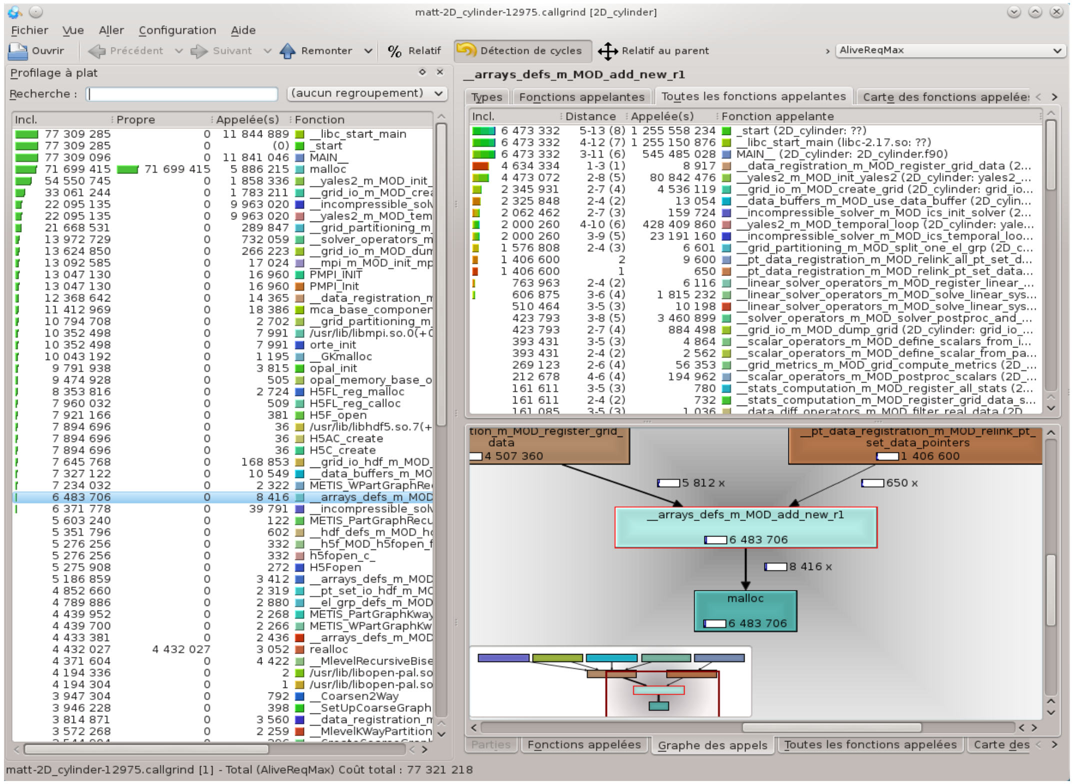
Plan

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

Output, first idea, kcachegrind

Callgrind compatibiltiy

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



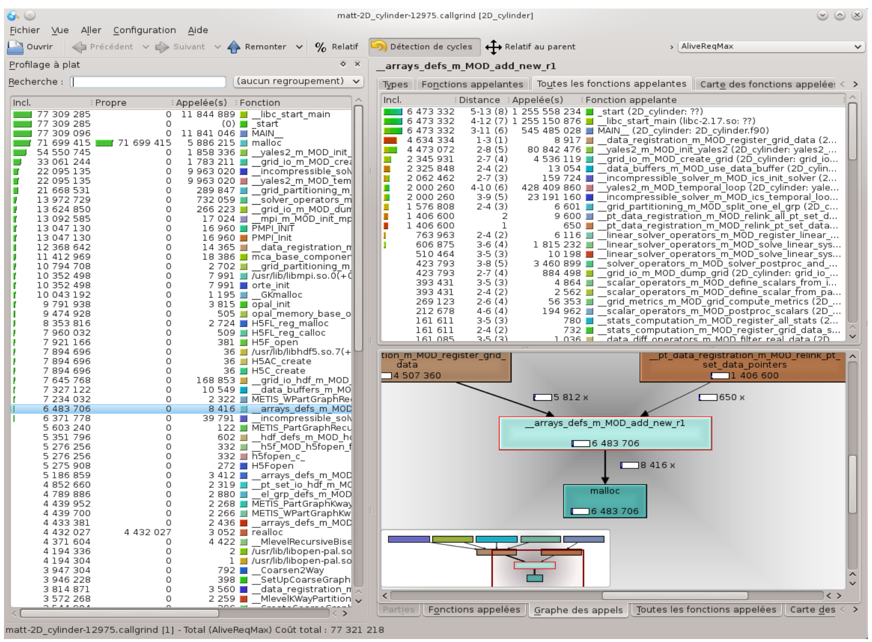
Global output

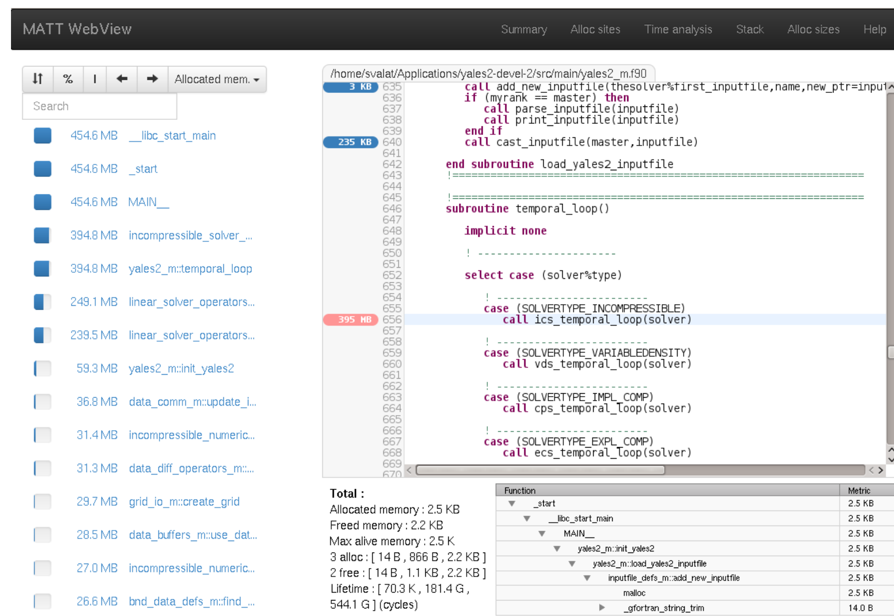
Callgrind compatibiltiy

- Can use kcachgrind
- Might be usefull 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





Usage

- 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
```

Exascale computing research

Ideas of improvement

- 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

Global summary

EXECUTION TIME	PHYSICAL MEMORY PEAK	ALLOCATION COUNT	AVAILABLE PHYSICAL MEMORY
00:00:00.25	2.3 MB	379	4.1 Gb

Run description

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

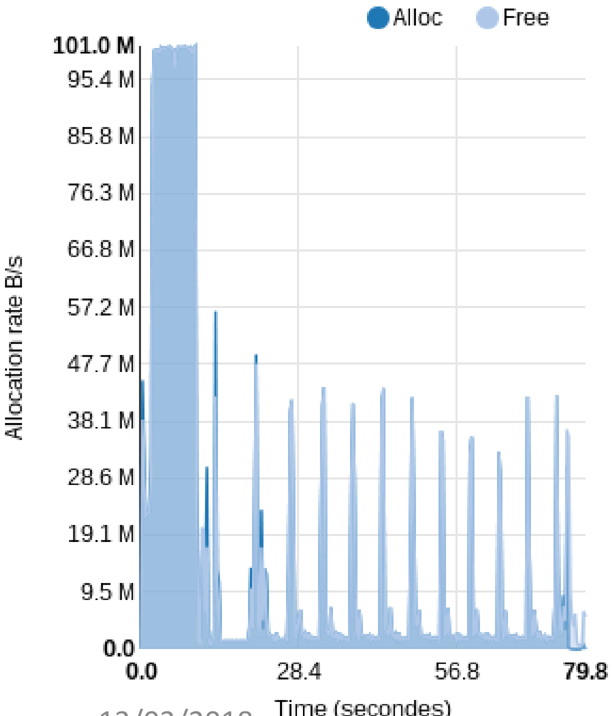
Global statistics

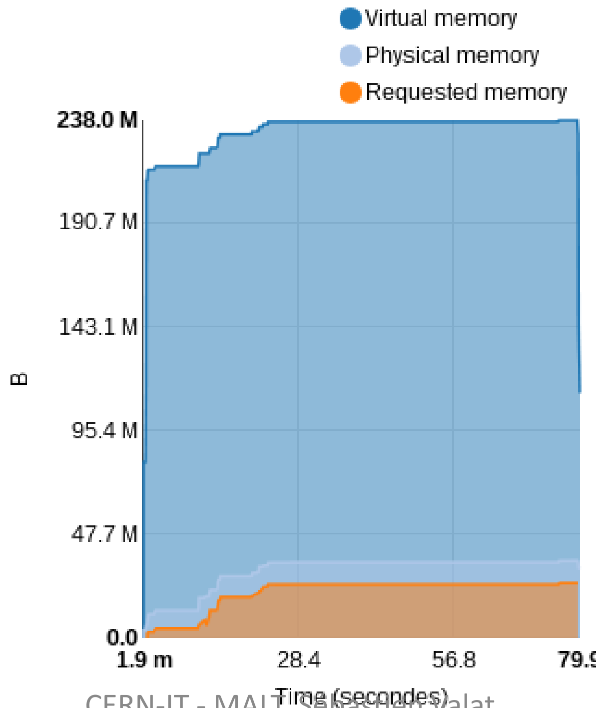
Show all details | Show help

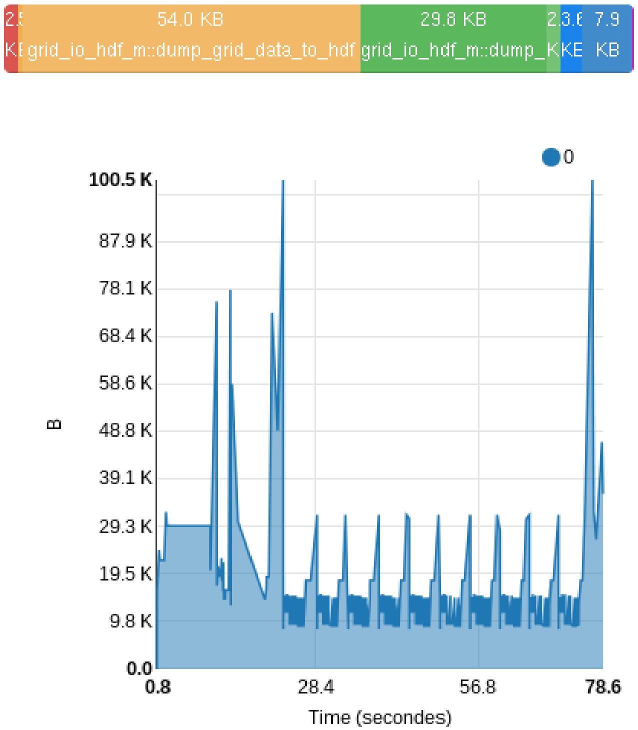
Physical memory peak	2.3 MB
Virtual memory peak	103.7 MB

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

- **Example on YALES2 with gfortran :**



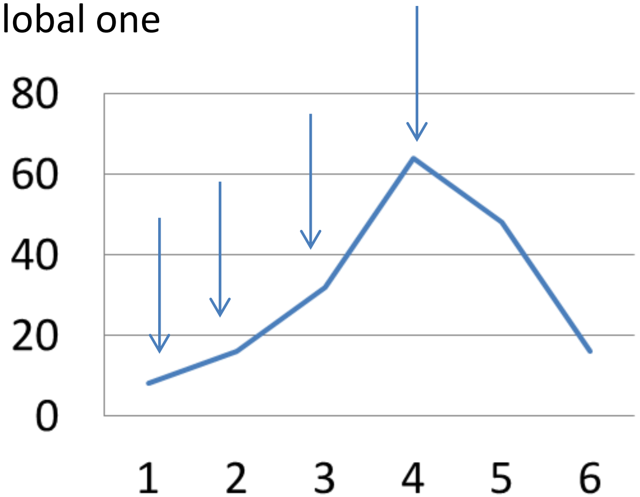




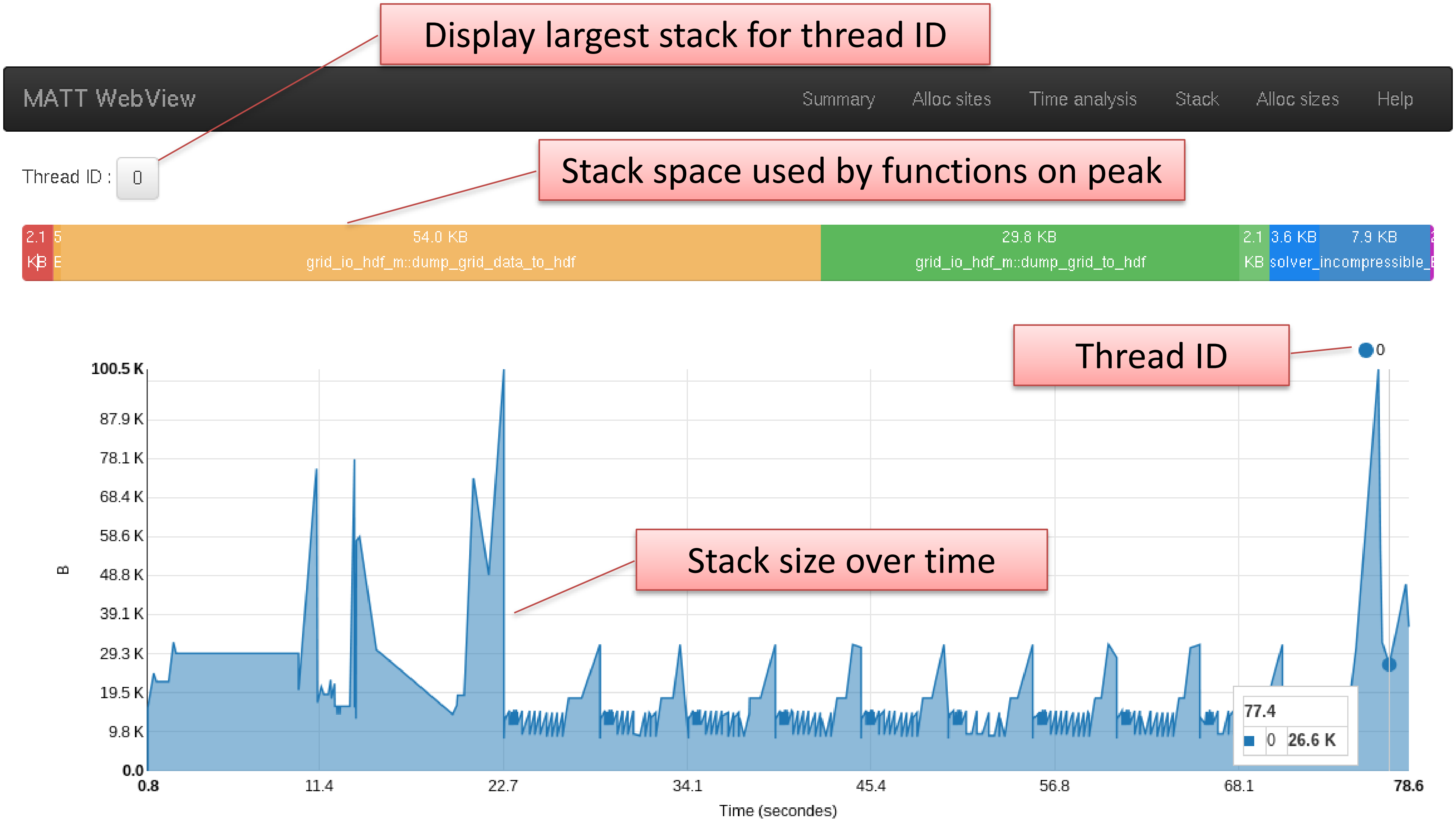
About global peak

- 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

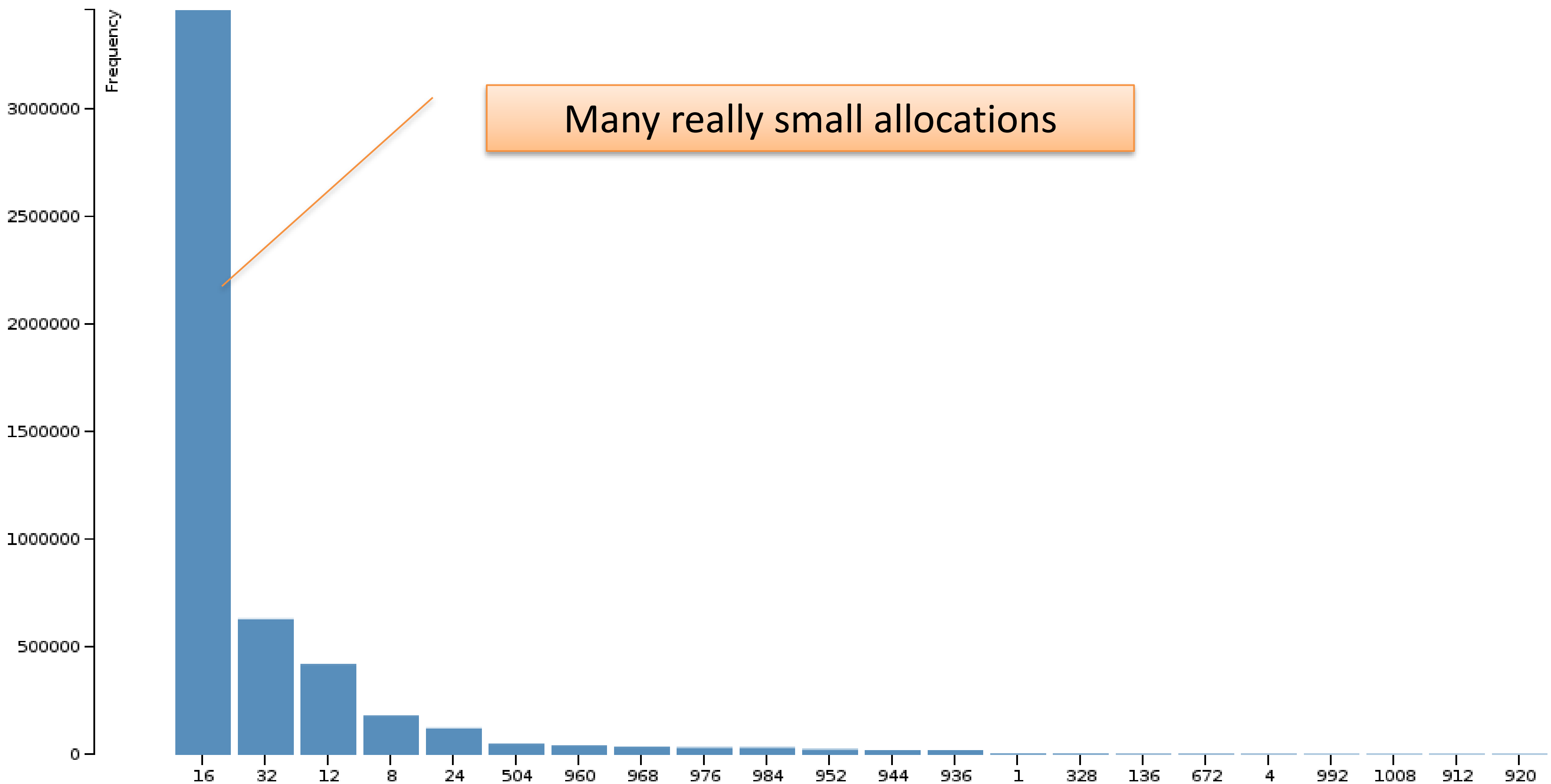


Tracking stack memory



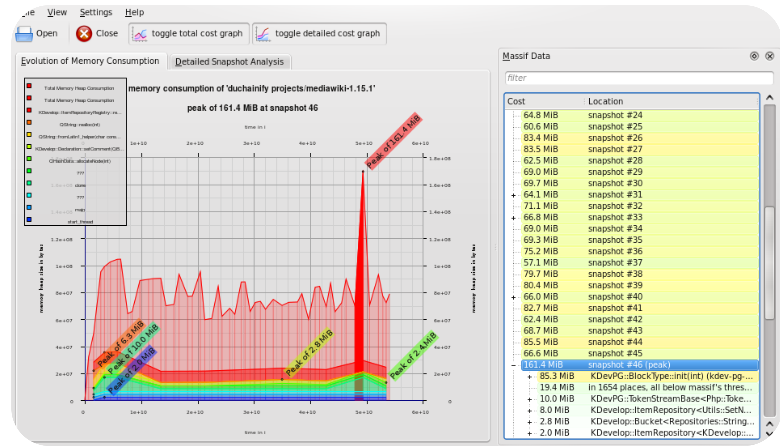
Chunk size distribution

Example from YALES2 with gfortran issue

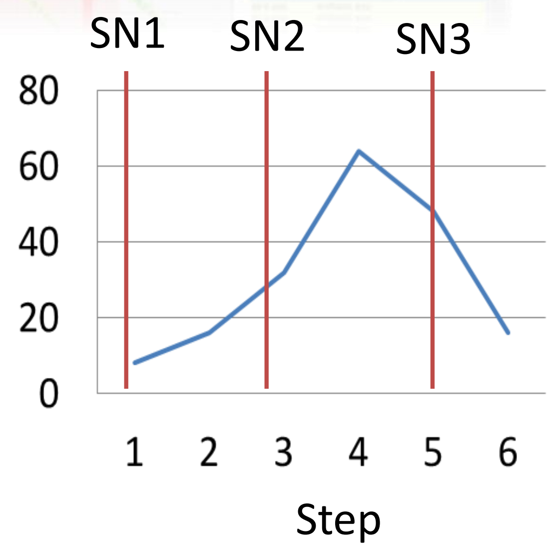


Existing tools / valgrind

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

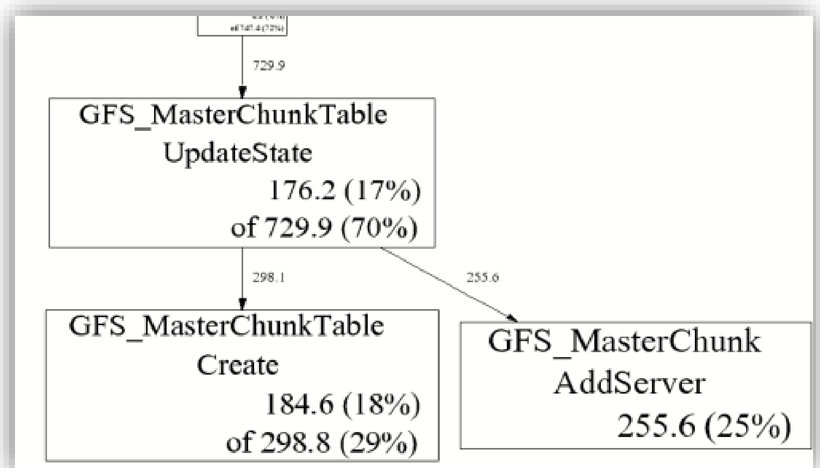


- **Valgrind - memcheck :**
  - Leak detection
  - Slow, not parallel.



Existing tools / Google heap profiler

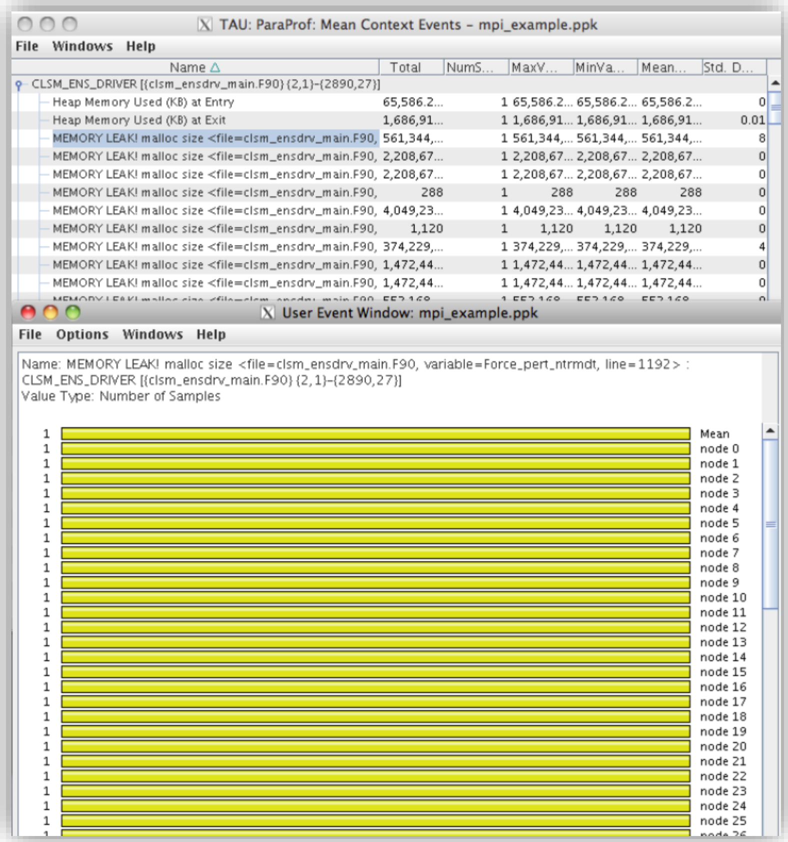
- **Google heap profiler (tcmalloc):**
  - **Small overhead.**
  - Similar metric than massif
  - Only provide snapshots of **allocated memory per stacks**.
  - Peak might not be captured.
  - Lack of a real GUI to use it.




```
% 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
```

Existing tools

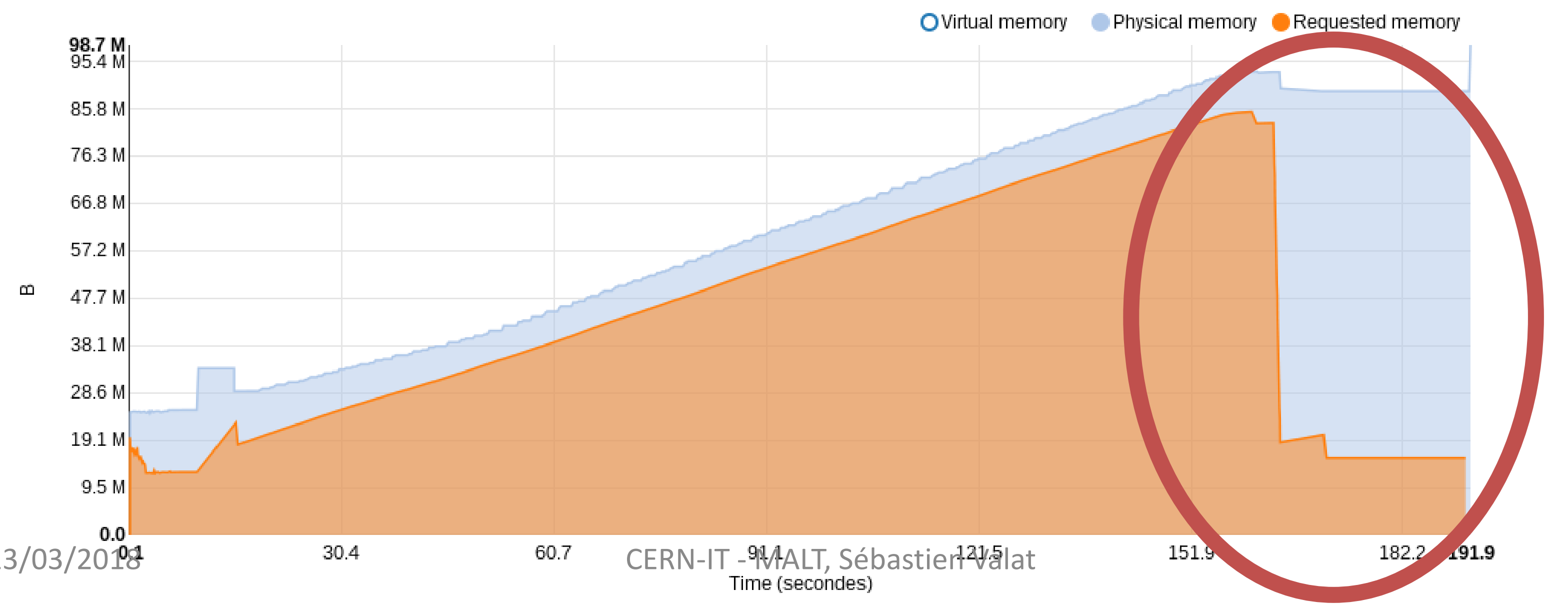
- **TAU memory profiler**
  - Provide **profiles** (not snapshots)
  - Provide leaks
  - Done for HPC/MPI
  - Lack easy matching with sources



Fragmentation issue

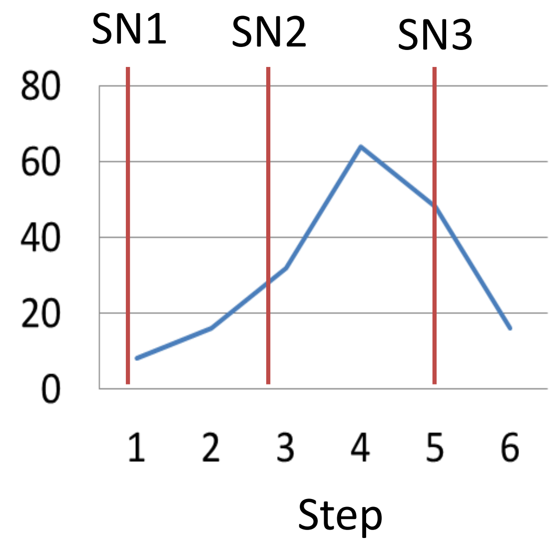
- Example from **Dassault mini-app** from Loïc Thébault and Eric Petit.
- **Fragmentation** can **prevent** from **returning physical pages** to OS
- **Solution** : **avoid interleaved** allocation of chunks with **different lifetime**.
- We observed with the **source annotation** that **most of them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

Memory allocated over time



Existing tools

- Tau memory profiler
  - Do **not** use **snapshots**
  - Provides **min/average/max**
  - Support MPI



- Commercial tools:
  - Ensure ++
  - Purify++
  - Visual Studio Ultimate edition