

MALT : MALloc Tracker

A memory profiling tool



- 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** ?

- I wanted 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 (auto it : iter_Items)
    {
        double* buffer = new double[size];
        //short and quick do stuff
        delete [] buffer;
    }
    return ret;
}
    
```

Global variables and TLS

Indirect allocations

Leak

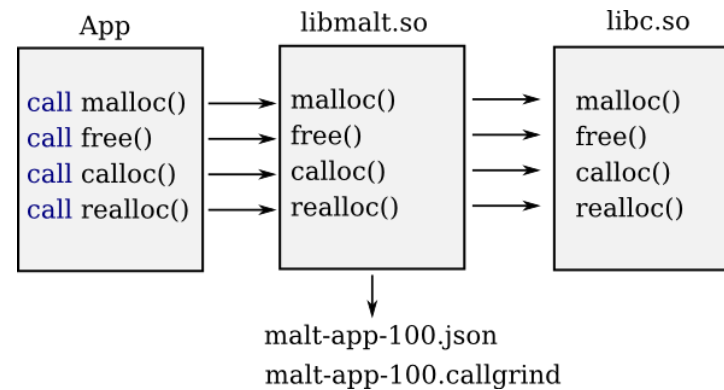
Might lead to swap for large size

C++11 auto induced allocs

Short life allocations

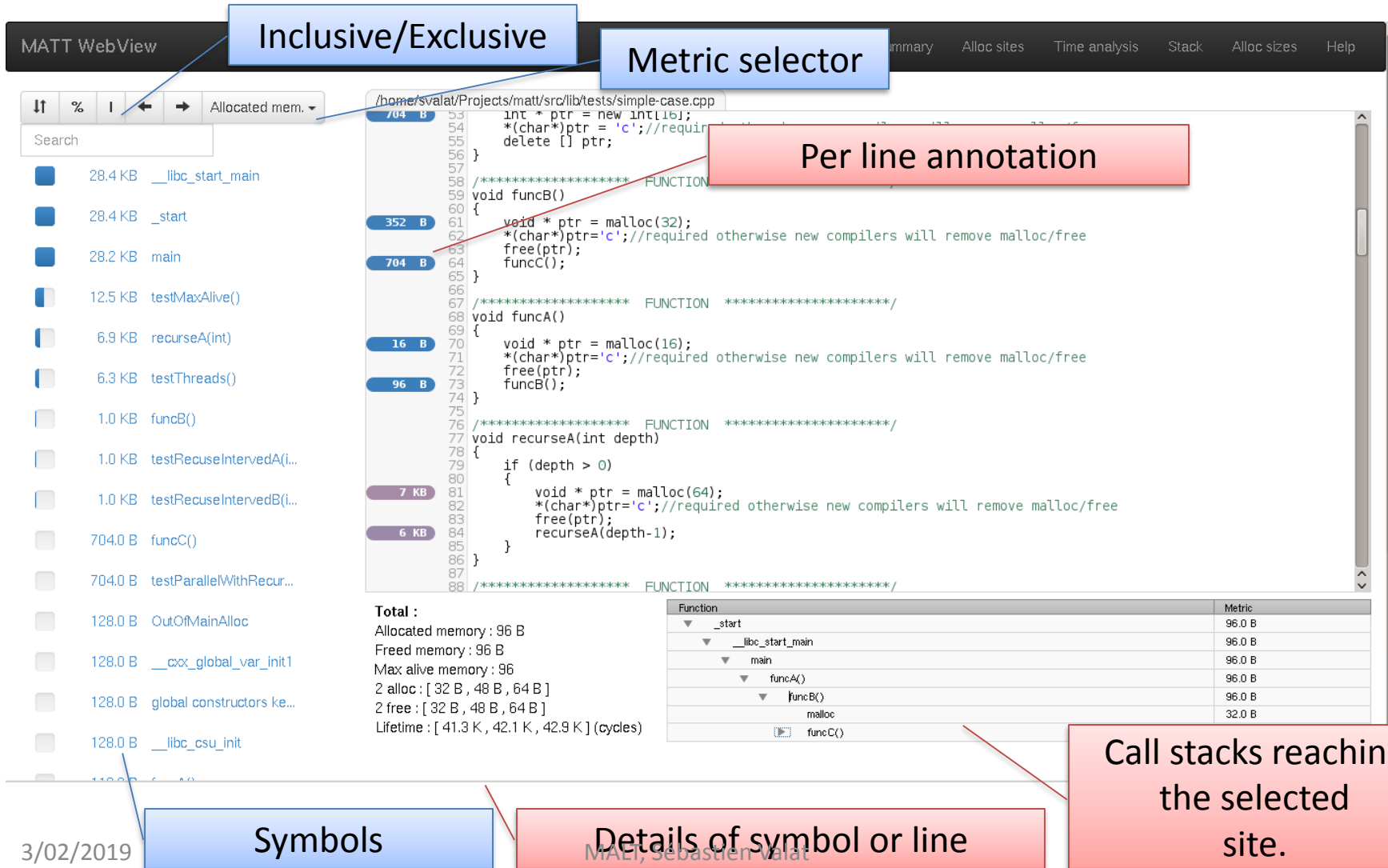
- Same **approach** than **valgrind/kcachgrind**
- **Mapped** allocations on **sources lines** and **call stacks**
- **Using a web-based GUI**
 - I started with **kcachgrind**
 - But wanted more flexibility and time charts

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



- **Map** allocations on **call stacks**
- **Build & consolidate summary** metrics
- Generate **JSON** output file

Web technology (NodeJS, D3JS, JQuery, AngularS)



MATT WebView | Inclusive/Exclusive | Metric selector | Summary | Alloc sites | Time analysis | Stack | Alloc sizes | Help

Allocated mem. ▾

Search

- 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(i...
- 1.0 KB testRecuseIntervdB(i...
- 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

```

/home/svalat/Projects/matt/src/lib/tests/simple-case.cpp
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 *****/
    
```

Per line annotation

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()	

Symbols | **Details of symbol or line** | **Call stacks reaching the selected site.**

3/02/2019 | MATT, Sébastien Valat

MALT WebView
Home Threads Sources Calltree Timeline analysis Stack memory Alloc sizes Realloc Global variables Help

Navigation buttons

Go Back Go Forward

Allocated count

Search

- 12.9 M G4RunManager:...
- 7.4 M _ZN5s4_Rep9_S...
- 7.1 M G4RunManagerK...
- 7.0 M start_thread
- 7.0 M MALT::pthreadWr...
- 7.0 M G4MTRunManag...
- 7.0 M G4WorkerRunMa...
- 6.9 M G4SmartVoxelHe...
- 6.9 M G4SmartVoxelHe...

Filtering to show only useful nodes

Searchable Symbol list

Navigable Call Tree

Showing 23 out of 2862 nodes.

Filter Graph Nodes

Height Unlimited

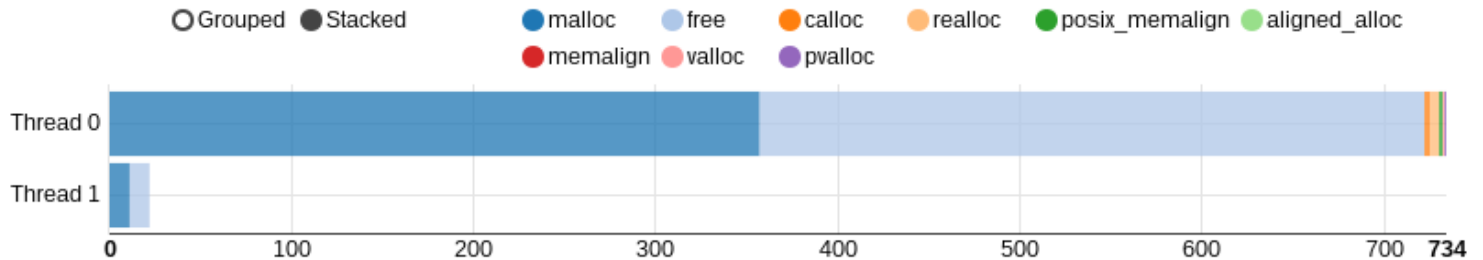
Depth 5

Node Cost 1%

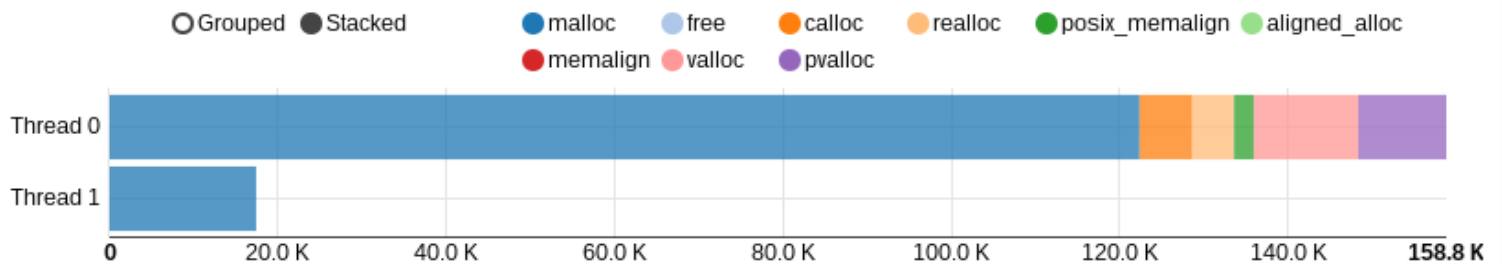
Inclusive

Allocated memory	444.2 MB
Freed memory	422.2 MB
Local peak	28.2 MB
Leaks	29555526
6.2 M alloc	[1 B , 75 B , 156.3 KB]
6.0 M free	[4 B , 74 B , 156.3 KB]

Call per thread

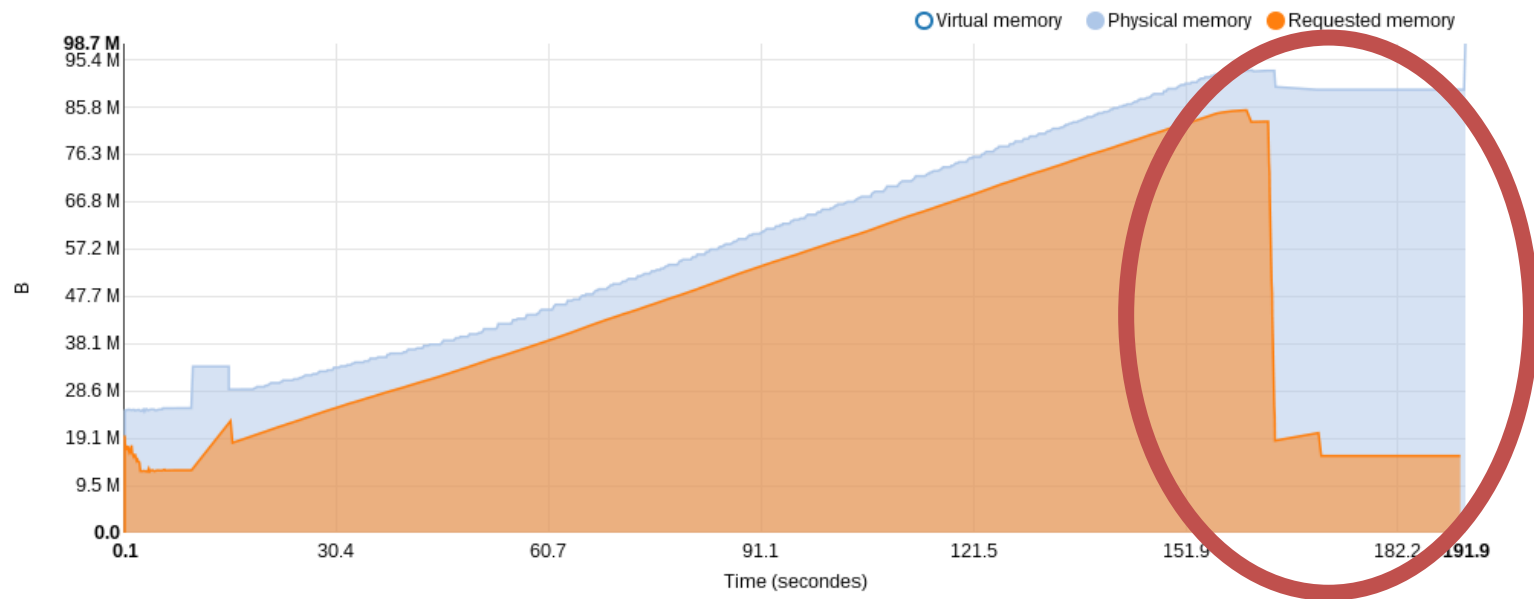


Time per thread

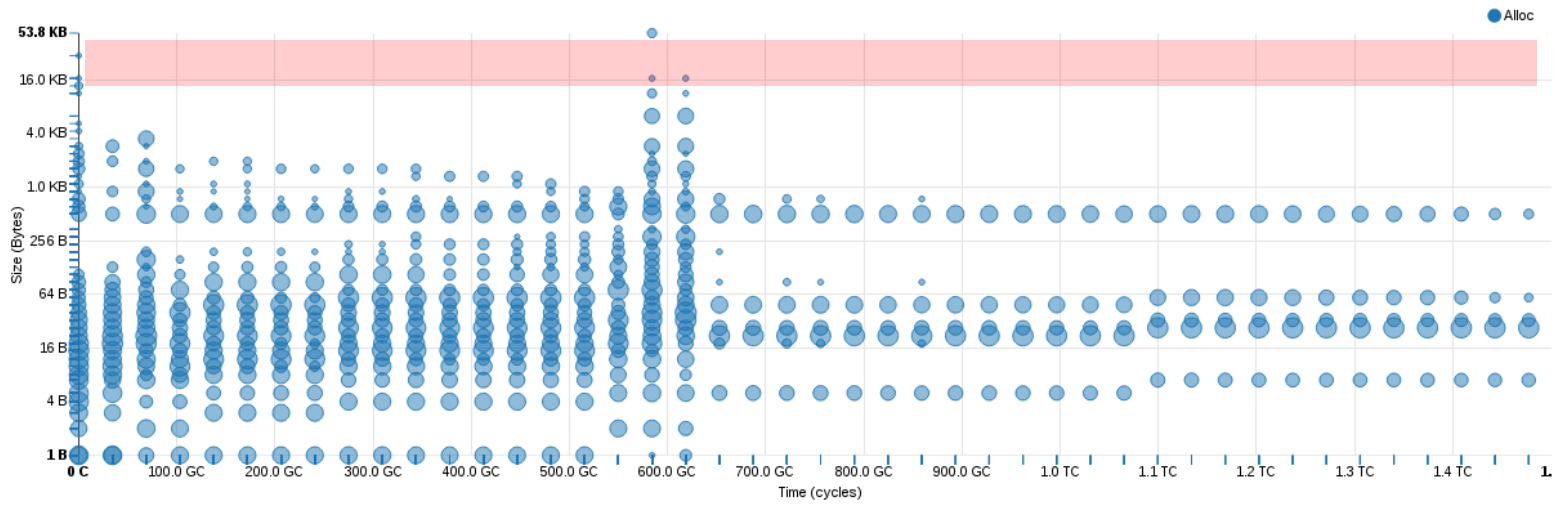


- Memory consumption over time
 - Physical
 - Virtual
 - Requested (malloced)

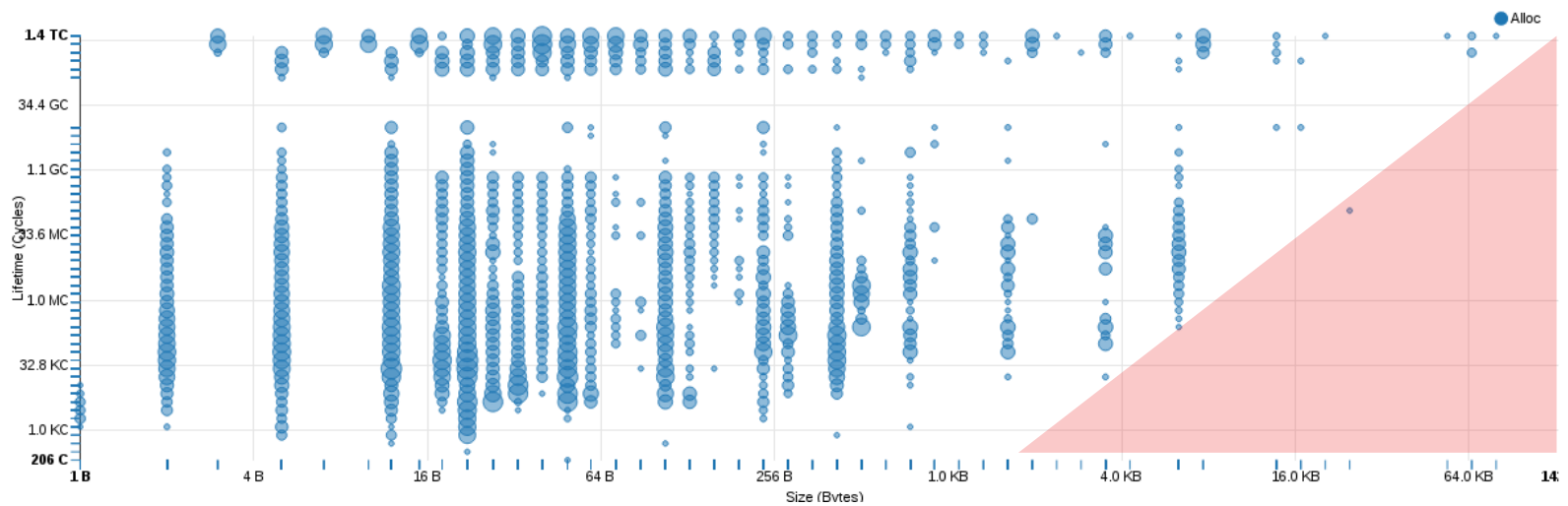
Memory allocated over time



Size over time

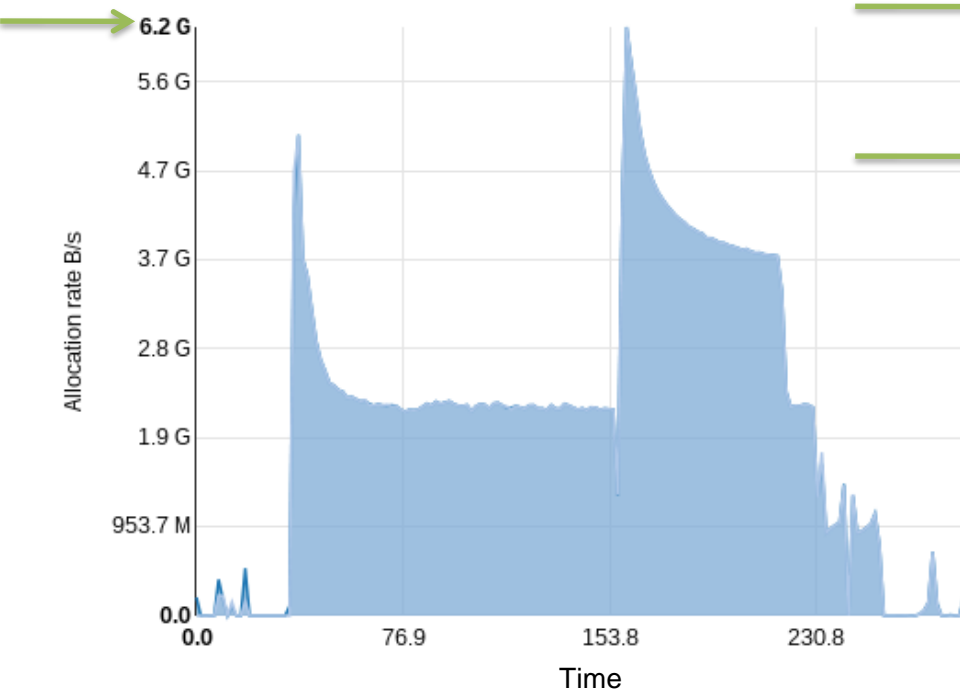


Lifetime over size



- 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

- Optionally recompile with debug flags :

```
gcc -g ...
```

- Run

```
malt [--config=file.ini] YOUR_PRGM [OPTIONS]
```

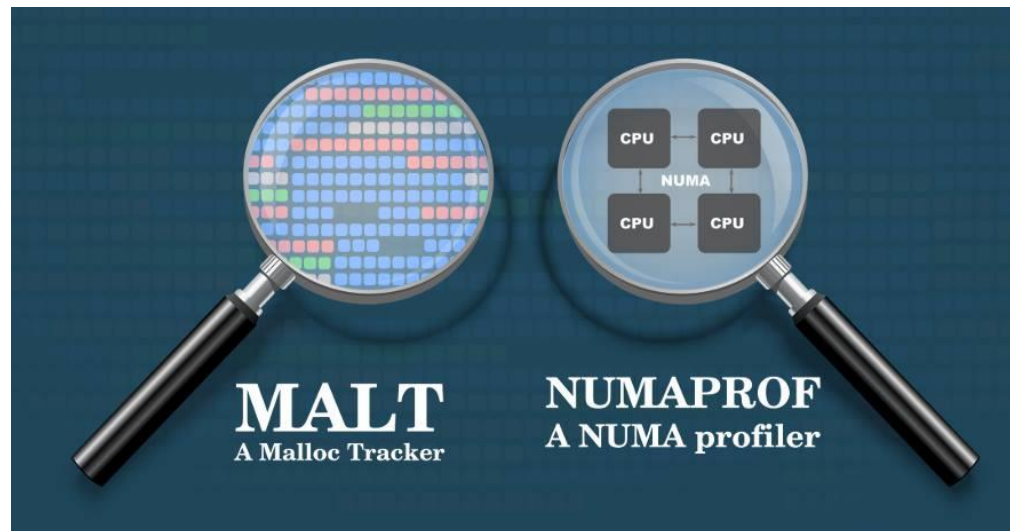
- Use the web view && <http://localhost:8080>:

```
malt-webview -i malt-{YOUR_PRGM}-{PID}.json
```

- In case there is a QT wrapper embedding NodeJS + Webkit

```
malt-qt -i malt-{YOUR_PRGM}-{PID}.json
```

- **Open sourced** since one year on <https://github.com/memtt>
- Co-hosted with a **similar tool** :
NUMAPROF for **Non Uniform Memory Access** profiling.



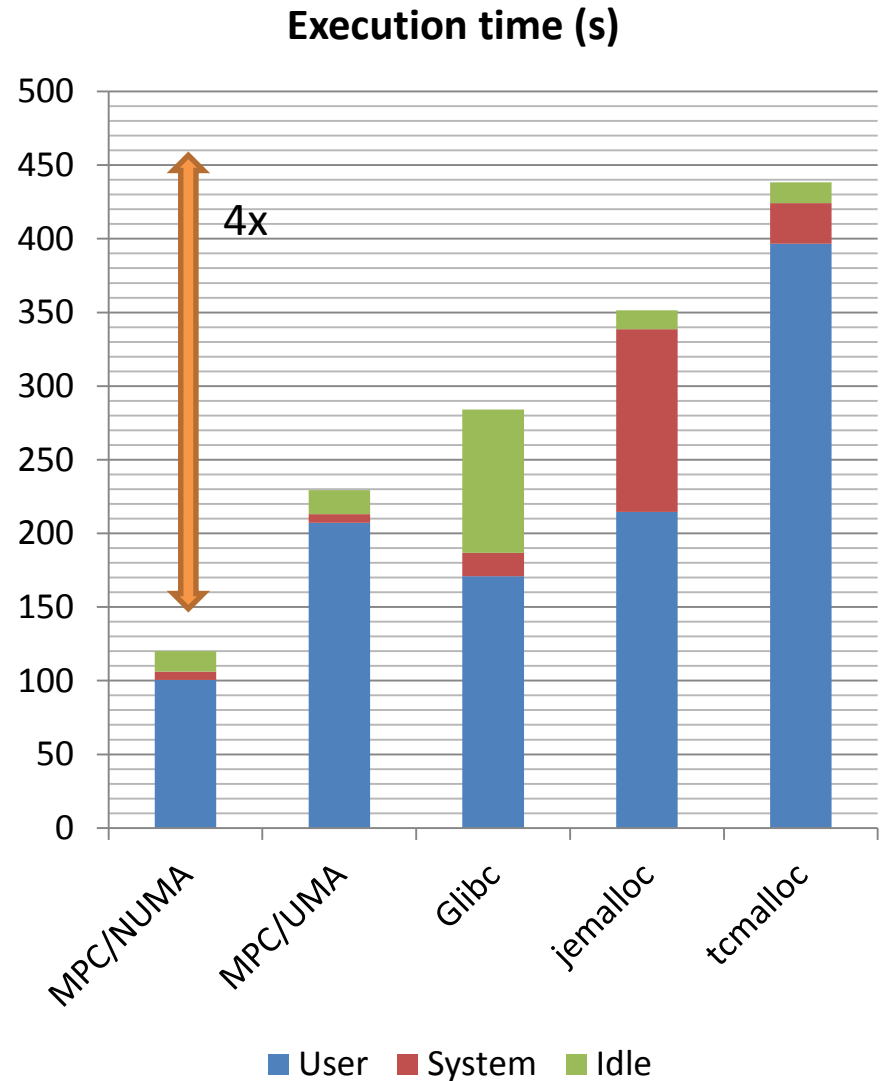
- My **research** on memory management for **HPC** : <http://svalat.github.io/>

Thank you.

QUESTIONS ?

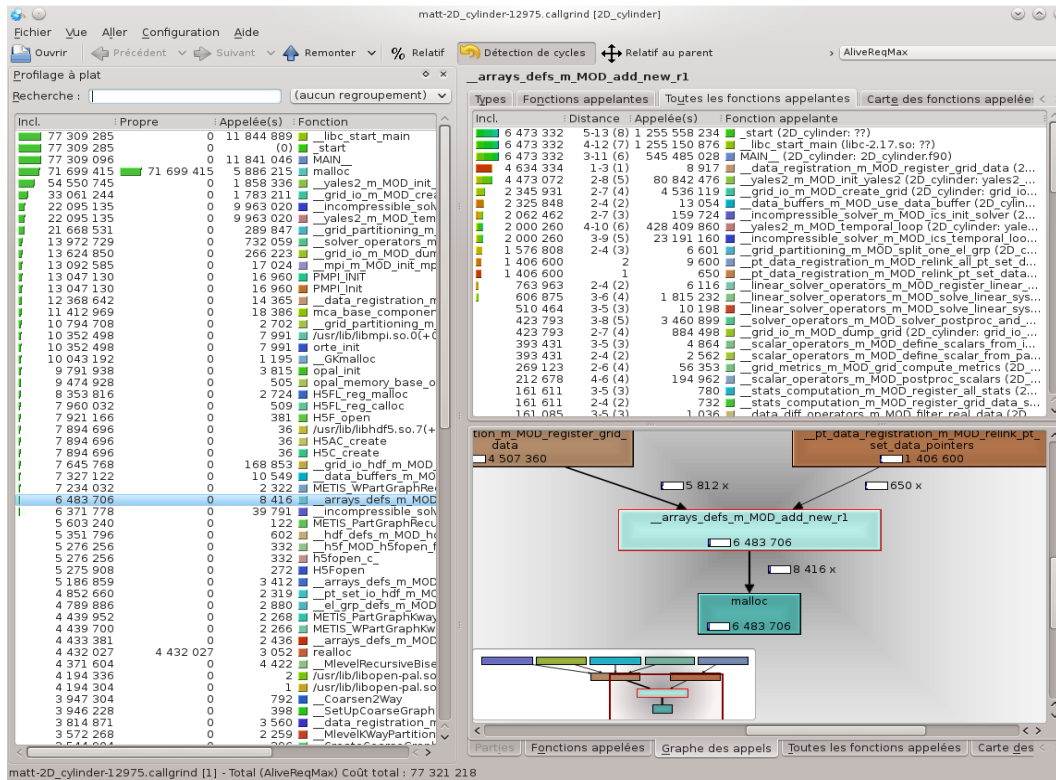
BACKUP

- **Memory management** can have **huge impact** on performance
- **Extreme case** on a 1.5 million C++ lines **HPC simulation** app. on a **16 processors** server
- Can see **10-15%** improvement on **MySQL** by **changing allocator**



Callgrind compatibility

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



- **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).

- 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

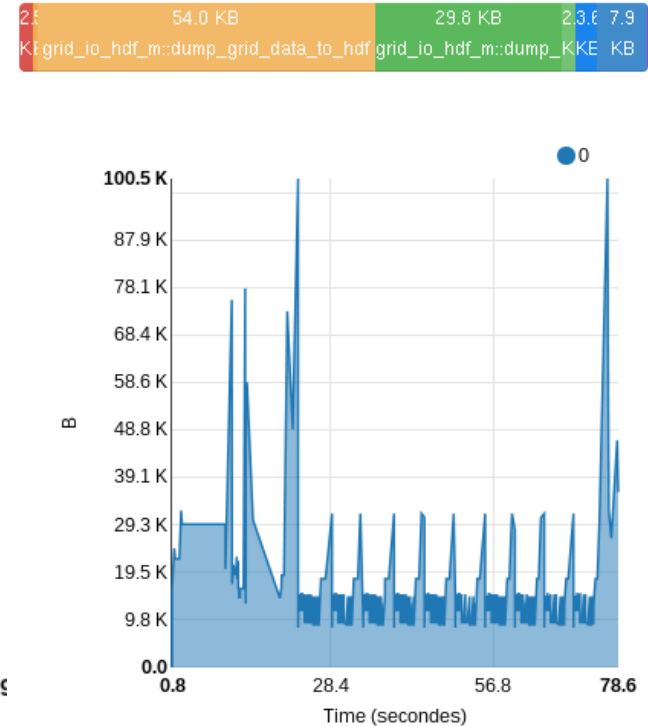
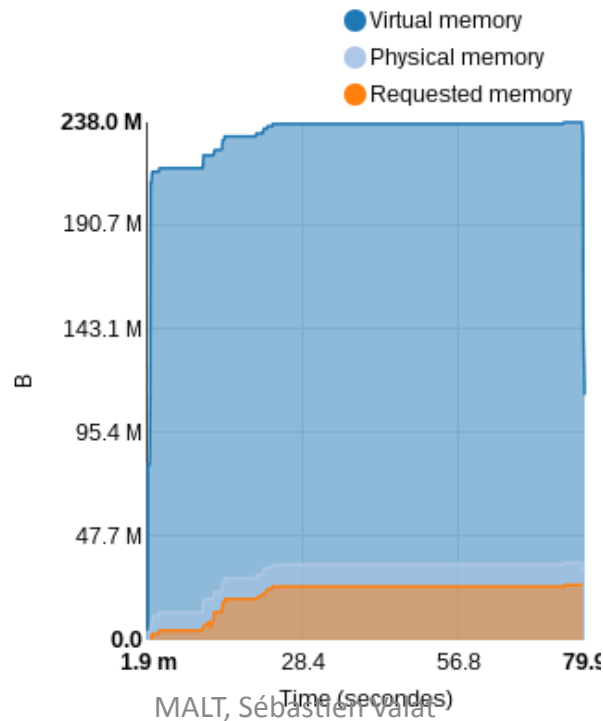
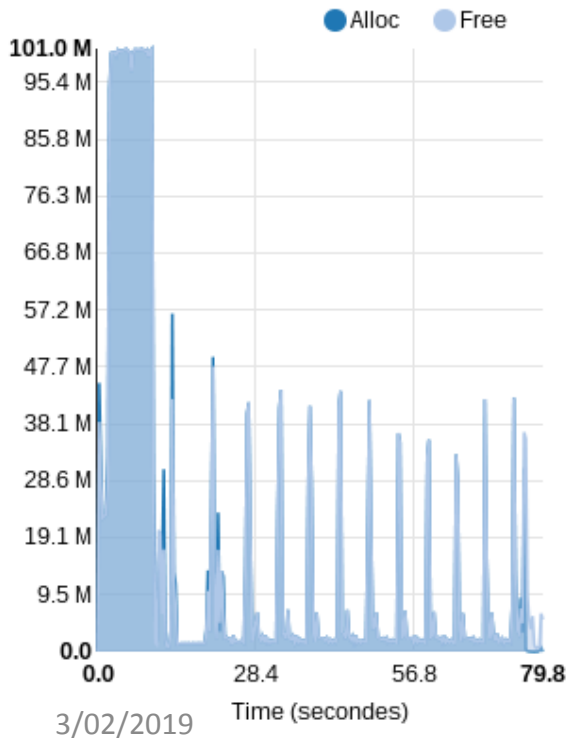
Global statistics

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

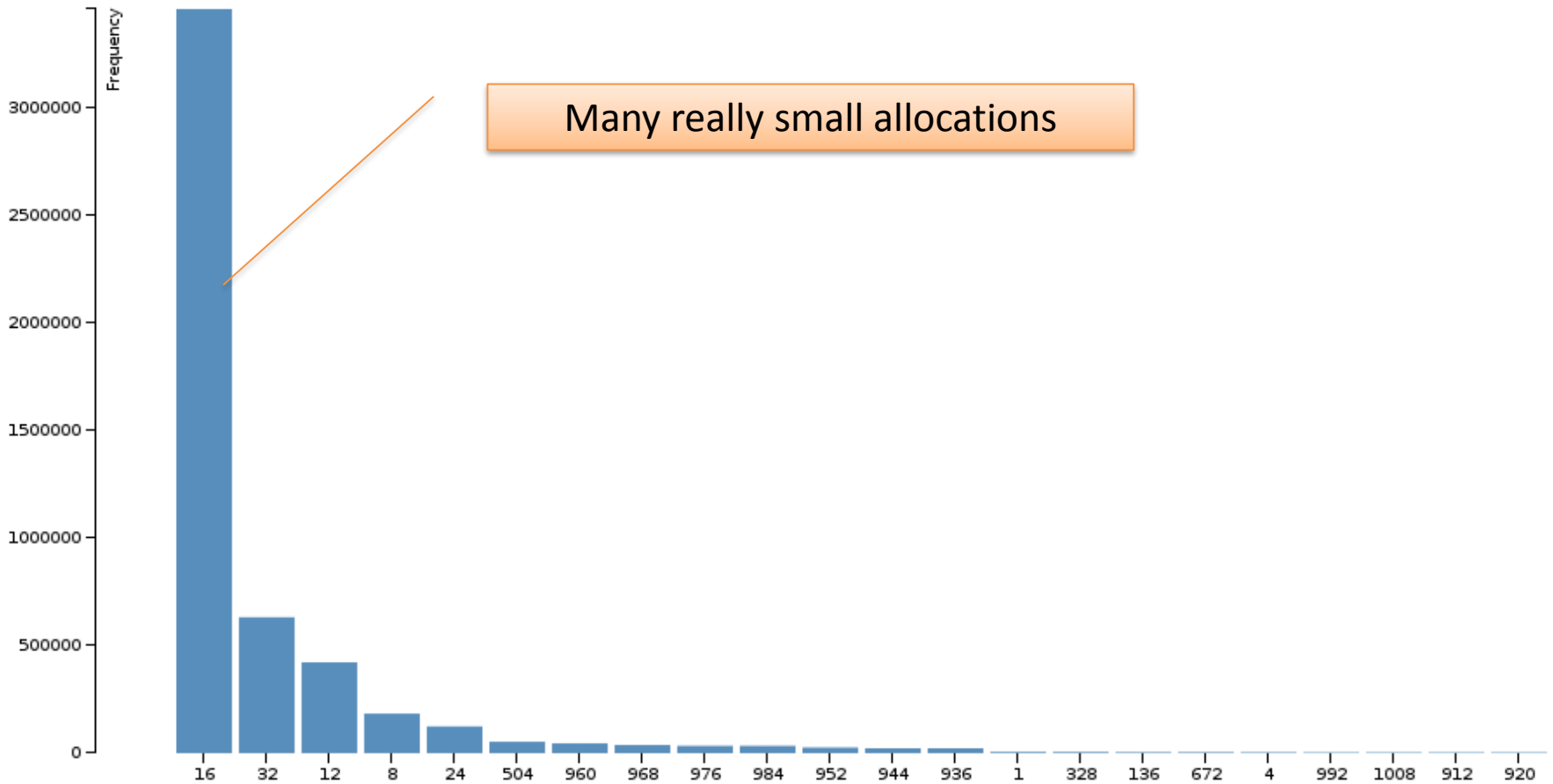
Physical memory peak	2.3 MB
Virtual memory peak	103.7 MB
Requested memory peak	2.8 KB

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

- **Example on YALES2 with gfortran :**

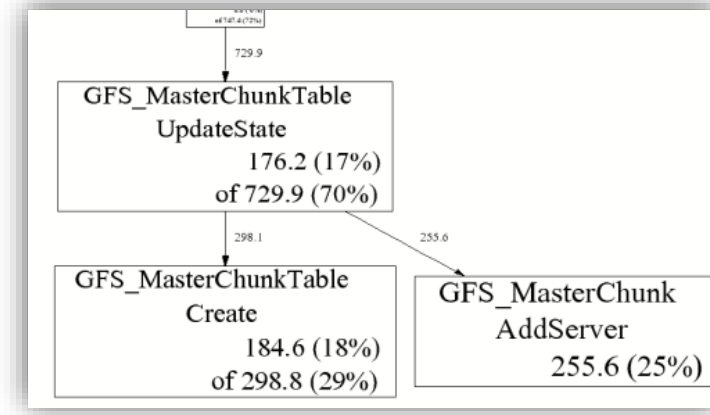
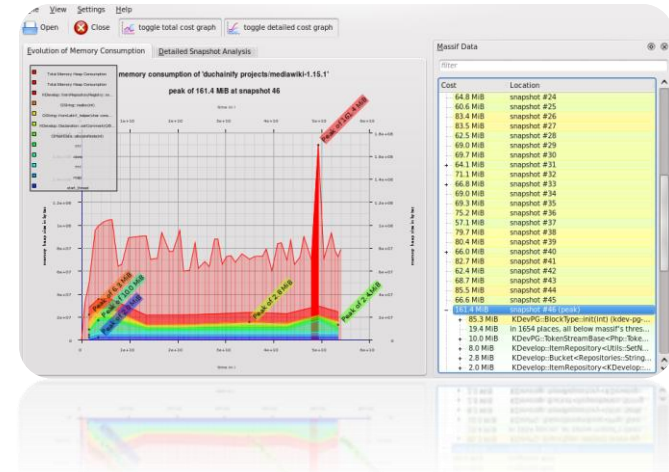


Example from YALES2 with gfortran issue

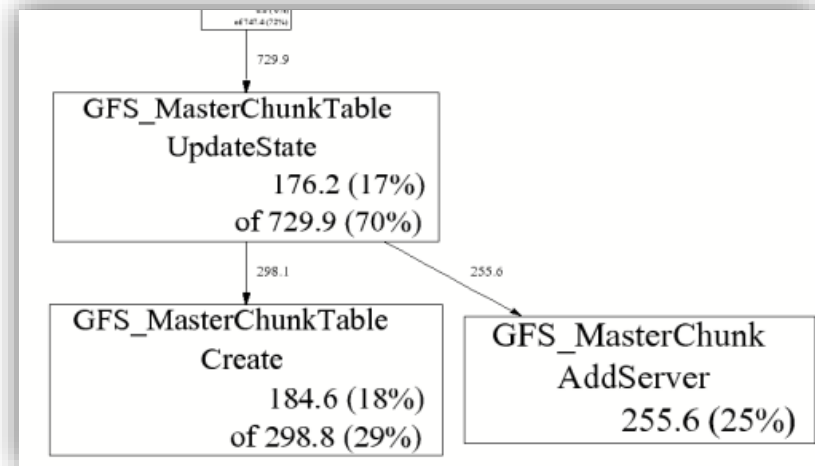


EXISTING TOOLS

- 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



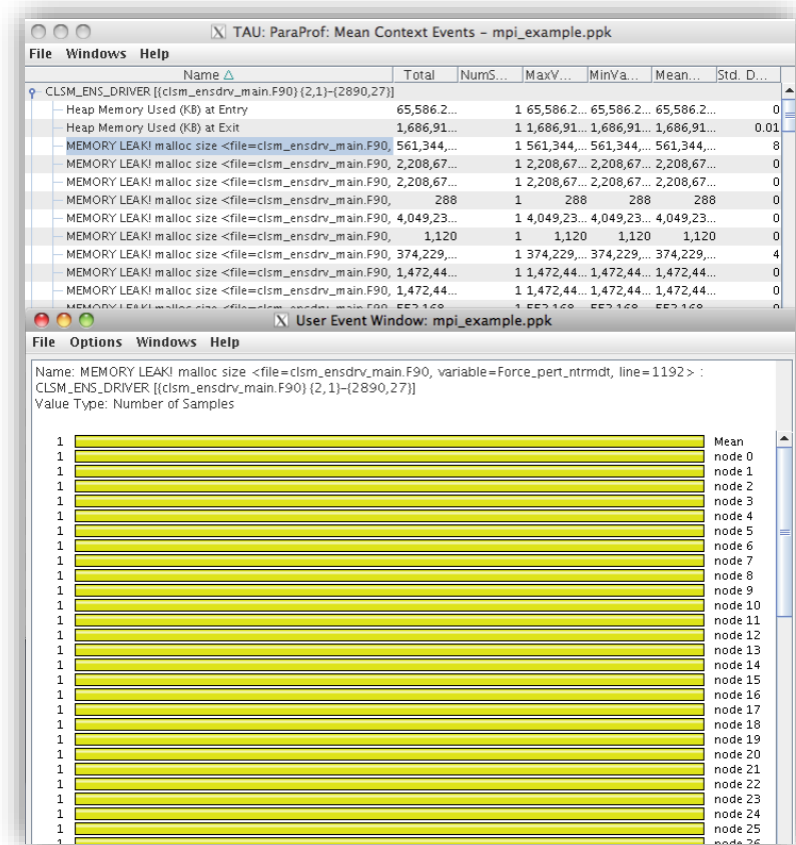
- **Google heap profiler (tcmalloc):**
 - **Small overhead.**
 - Similar metric than massif
 - Only provide **snapshots** of **allocated memory per stacks.**
 - Peak might not be captured.
 - **Lack of a real GUI to use it.**



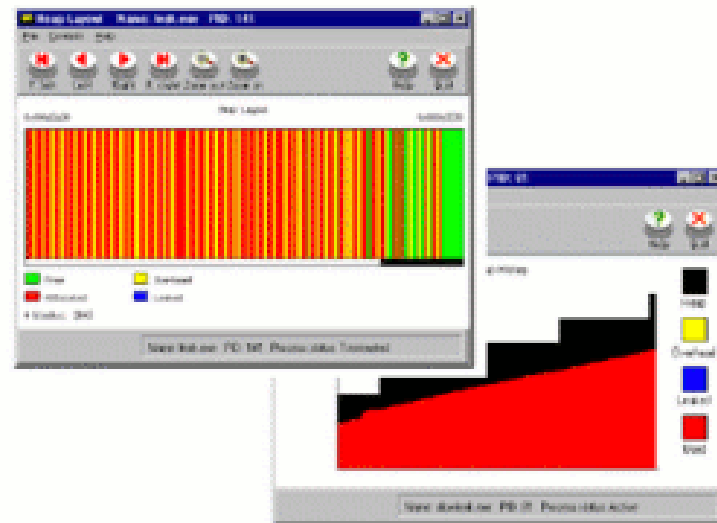
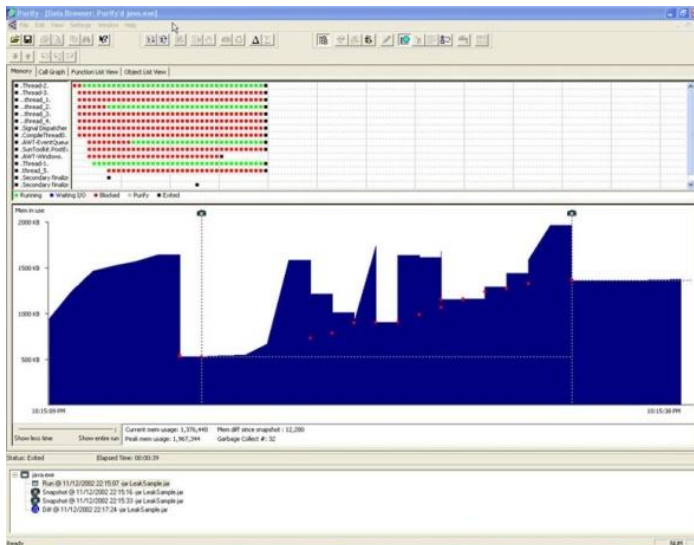
```

% 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
 - Follow stacks
 - Track leaks
 - Parallel, done for HPC/MPI
 - Lack easy matching with sources
- **FOM**



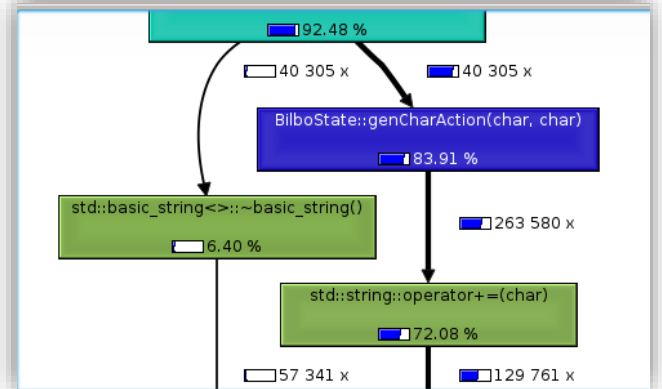
- **IBM Purify++ / Parasoft Insure++**
 - Commercial
 - Leak detection, access checking, memory debugging tools.
 - Use binary or source instrumentation.
 - Windows / Redhat
- **Visual Studio Ultimate Edition Memory profiler**
 - Nice but windows only and commercial



- 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

- List of functions with exclusive/inclusive costs
- Nice call tree
- Annotated sources

100.00	0.00	(0)	0x0000000000001
97.96	0.00	1	0x0000000000401
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(unsig



```


0.00 16 call(s) to 'std::string::size() const' (libstdc++6)
0.00 1 call(s) to '_dl_runtime_resolve' (ld-2.20)
{
  //after 20 chars, try to move to the next word
  //if (i%10 == 0)
  //  cout << state.genOrderToM
969
970 //check for compression
971 WordCompression wordCompression
0.15 15 call(s) to 'checkForWordCompression' (libstdc++6)
972 SequenceCompression sequence
0.03 15 call(s) to 'checkForSequenceCompression' (libstdc++6)
973 BiSequenceCompression biSequenceCompression
0.01 15 call(s) to 'checkForBiSequenceCompression' (libstdc++6)
974

```

SOME VIEWS

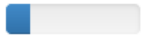
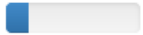



- Provide a small summary
- Provide some warnings

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


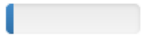
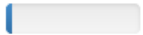
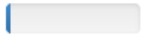

Physical memory peak	66.7 MB
Virtual memory peak	158.1 MB
Requested memory peak	6.1 MB
Cumulated memory allocations	11.5 MB
Allocation count	172.2 K
Recycling ratio	1.9
Leaked memory	743.7 KB
Largest stack	0 B
Global variables	10.0 MB 
TLS variables	48 B
Global variable count	421.0 K 
Peak allocation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

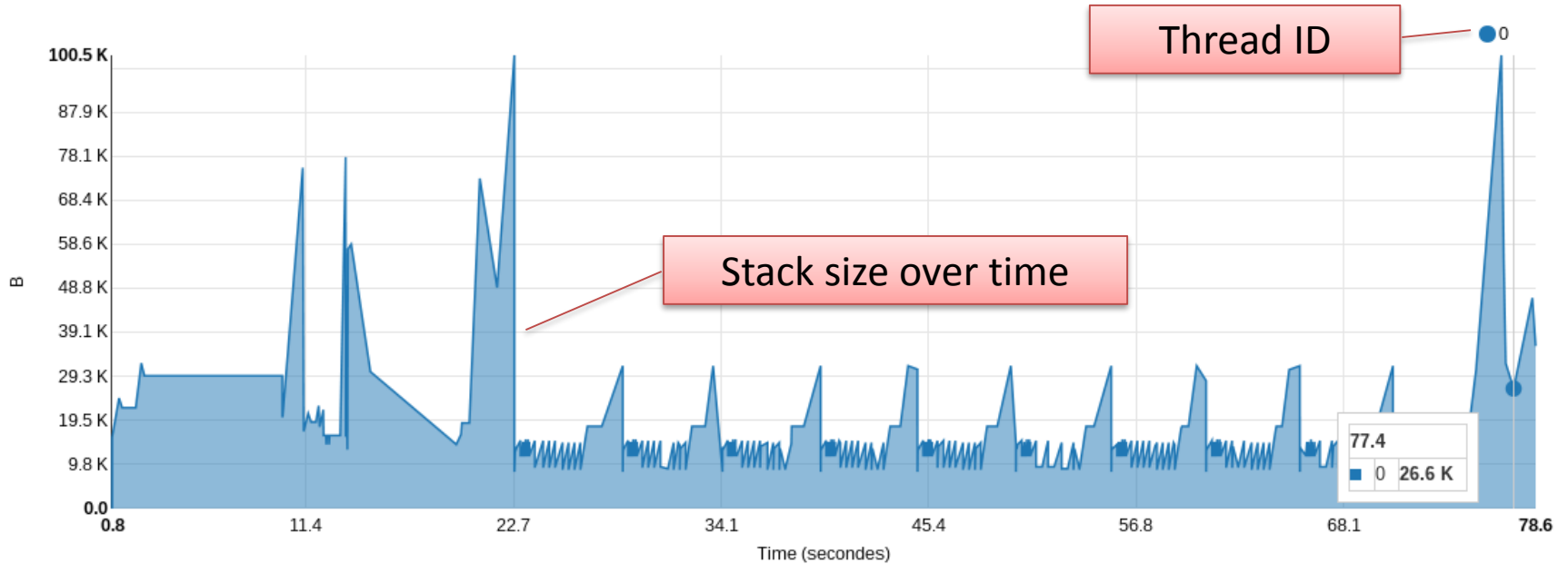
Display largest stack for thread ID

MATT WebView

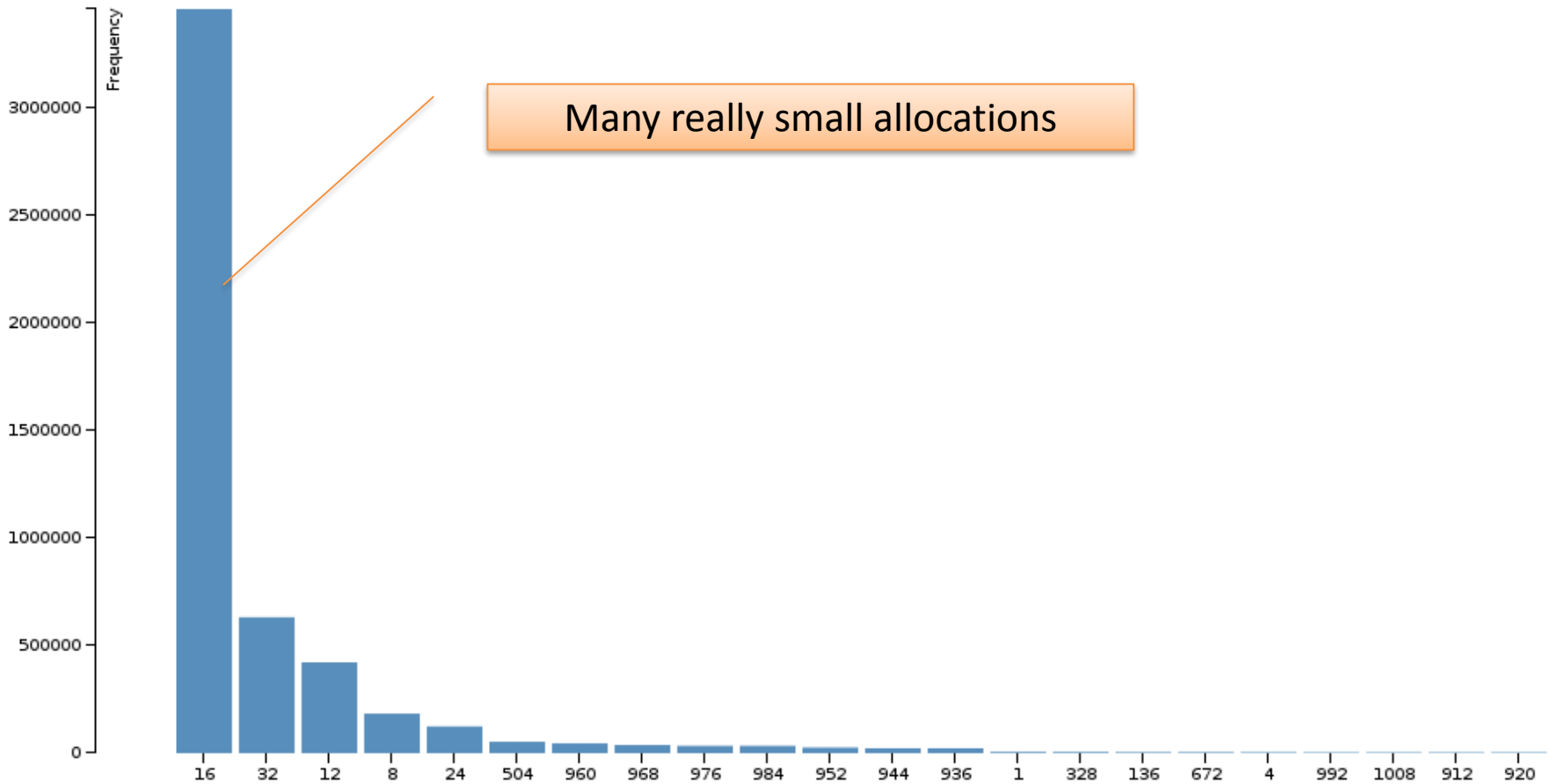
Summary Alloc sites Time analysis Stack Alloc sizes Help

Thread ID: 0

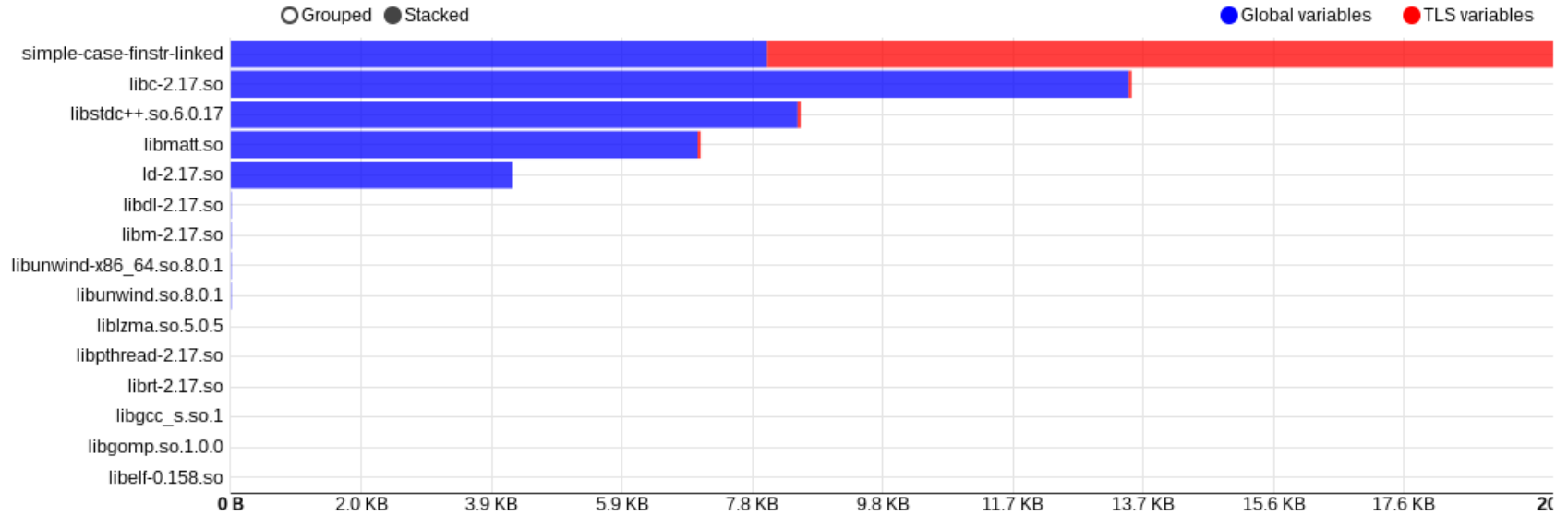
Stack space used by functions on peak



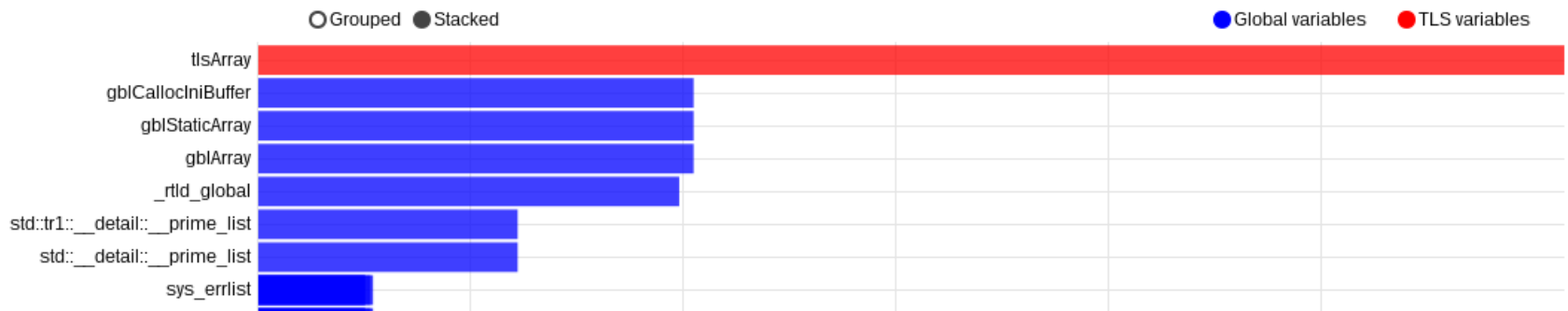
Example from YALES2



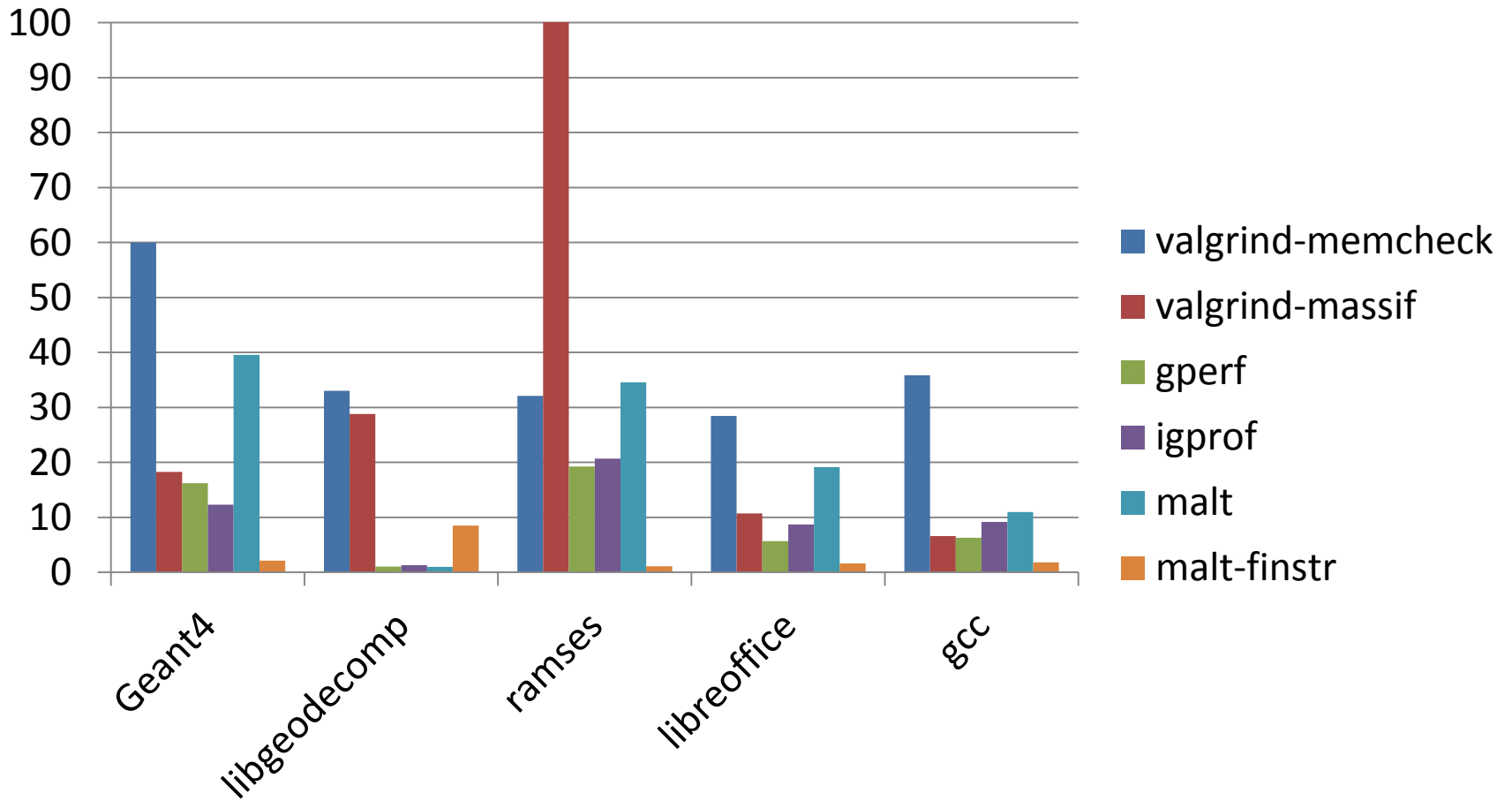
Distribution over binaries



Distribution over variables



REAL CASES



- 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 !

```

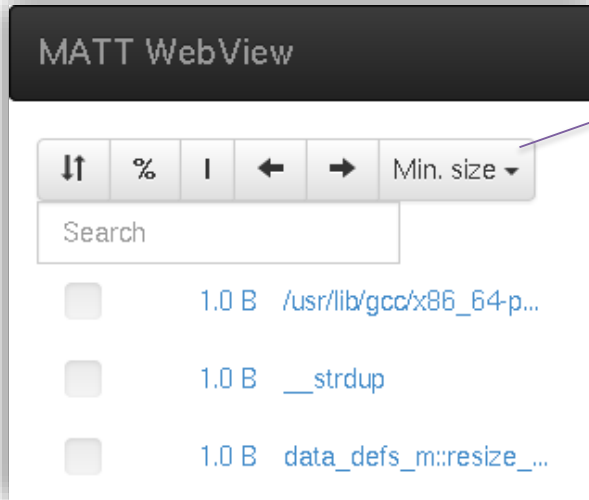
892       do i=1,nitem_el_grp
893         el_grp_ind = el_grp_index2int_comm_index%val(1,i)
894         int_comm_ind = el_grp_index2int_comm_index%val(2,i)
895         el_grp_r2%val(1:dim1,el_grp_ind) = int_comm_r2%val(1:dim1,int_comm_ind)
896       end do
    
```

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: 03/02/2019
 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.

```

530
531
532
533
534
535
536
537
538
539

```

```

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
  allocate(data_ptr%r2_ptrs(nel_grps))
  do n=1,nel_grps
    NULLIFY(data_ptr%r2_ptrs(n)%ptr)
  end do

```

1 B

- 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

