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**?
Some issue examples

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

```cpp
__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

SEPS'17 - MALT, Sébastien Valat
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
• Tau memory profiler
  – Do **not** use **snapshots**
  – Provides **min/average/max**
  – Support MPI

• Commercial tools:
  – Ensure ++
  – Purify++
  – Visual Studio Ultimate edition
What I want to provide

• Same approach than valgrind/kcachegind

• 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 (~>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 kcachgrind

- 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, AngularJS)
- Easier for remote usage
Global summary

- Provide a small summary
- Provide some warnings

<table>
<thead>
<tr>
<th>Show all details</th>
<th>Show help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical memory peak</td>
<td>66.7 MB</td>
</tr>
<tr>
<td>Virtual memory peak</td>
<td>158.1 MB</td>
</tr>
<tr>
<td>Requested memory peak</td>
<td>6.1 MB</td>
</tr>
<tr>
<td>Cumulated memory allocations</td>
<td>11.5 MB</td>
</tr>
<tr>
<td>Allocation count</td>
<td>172.2 K</td>
</tr>
<tr>
<td>Recycling ratio</td>
<td>1.9</td>
</tr>
<tr>
<td>Leaked memory</td>
<td>743.7 KB</td>
</tr>
<tr>
<td>Largest stack</td>
<td>0 B</td>
</tr>
<tr>
<td><strong>Global variables</strong></td>
<td>10.0 MB</td>
</tr>
<tr>
<td>TLS variables</td>
<td>48 B</td>
</tr>
<tr>
<td><strong>Global variable count</strong></td>
<td>421.0 K</td>
</tr>
<tr>
<td>Peak allocation rate</td>
<td>37.8 MB/s</td>
</tr>
</tbody>
</table>
Global summary: top 5 functions

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

### Allocated memory

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Allocs</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>202.4 MB</td>
<td><code>linear_solver_operators_m::solve_linear_system_deflated_pcg</code></td>
</tr>
<tr>
<td></td>
<td>25.6 MB</td>
<td><code>bnd_data_defs_m::find_bnd_data</code></td>
</tr>
<tr>
<td></td>
<td>21.8 MB</td>
<td><code>linear_solver_operators_m::solve_el_grp_pcg</code></td>
</tr>
<tr>
<td></td>
<td>13.0 MB</td>
<td><code>data_comm_m::copy_int_comm_to_data</code></td>
</tr>
<tr>
<td></td>
<td>18.1 MB</td>
<td><code>data_comm_m::update_int_comm</code></td>
</tr>
</tbody>
</table>

### Peak memory

<table>
<thead>
<tr>
<th>Date</th>
<th>SEPS'17 - MALT, Sébastien Valat</th>
</tr>
</thead>
<tbody>
<tr>
<td>24/11/2017</td>
<td>12</td>
</tr>
</tbody>
</table>
Call tree view

Navigable Call Tree

Searchable Symbol list

Filtering to show only useful nodes

Navigation buttons
Per thread statistics

Call per thread

Time per thread
Global variables

Distribution over binaries

- simple-case-first-linked
- libc-2.17.so
- libstdc++.so.6.0.17
- libm.so
- ld-2.17.so
- libdl-2.17.so
- libm-2.17.so
- libunwind-x86_64.so.0.0.1
- libunwind.so.0.0.1
- liblzma.so.5.0.5
- libpthread-2.17.so
- libdl-2.17.so
- libgcc_s.so.1
- libgomp.so.1.0.0
- libtool-0.158.so

Distribution over variables
Example on AVBP init phase

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

**Allocation rate**

---

```c
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)
```
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!

And mostly really small allocations!
Examples on YALES 2, small allocations:

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
Scatter plot

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

- **Main()**
  - Peak id = 0
  - On peak = 0
  - Current = 0 MB

- **Prepare()**
  - Peak id = 2
  - On peak = 10 MB
  - Current = 10 MB

- **Compute()**
  - Peak id = 2
  - On peak = 10 MB
  - Current = 10 MB

---

**Total Mem**
- 20 MB

**Global peak mem**
- 20 MB

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

**Total Mem**

- **15 MB**

**Global peak mem**

- **20 MB**

**Global peak ID**

- **2**

---

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

- **Peak id** = 0
- **On peak** = 0
- **Current** = 0 MB

**Prepare()** => **free(10 MB)**

- **Peak id** = 2
- **On peak** = 10 MB
- **Current** = 0 MB

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

- **Peak id** = 2
- **On peak** = 10 MB
- **Current** = 15 MB
Plan

• What we want

• Existing tools

• Our proposal

• Use cases

• Conclusion
Callgrind compatibility

- Can use kcachegrind
- Might be usefull for some users, cannot provide all metrics.
Existing tools / Commercials

- **IBM Purify++ / Parasoft 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

![Diagram showing how LD_PRELOAD works with malloc and free functions](image)

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

- Can use kcachegrind
- 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
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

Run description

<table>
<thead>
<tr>
<th>Executable:</th>
<th>simple-case-finstr-linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commande:</td>
<td>./simple-case-finstr-linked</td>
</tr>
<tr>
<td>Tool:</td>
<td>mact-0.0.0</td>
</tr>
<tr>
<td>Host:</td>
<td>localhost</td>
</tr>
<tr>
<td>Date:</td>
<td>2014-11-26 22:40</td>
</tr>
<tr>
<td>Execution time:</td>
<td>00:00:00.25</td>
</tr>
<tr>
<td>Ticks frequency:</td>
<td>1.8 GHz</td>
</tr>
</tbody>
</table>

Global statistics

<table>
<thead>
<tr>
<th>Physical memory peak</th>
<th>2.3 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual memory peak</td>
<td>103.7 MB</td>
</tr>
<tr>
<td>Requested memory peak</td>
<td>22 KB</td>
</tr>
</tbody>
</table>
Temporal metrics

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

- Example on YALES2 with gfortran:
• The tool maintain a **call stack tree**
• Profile **stats on leaves**
• 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

Display largest stack for thread ID

Stack space used by functions on peak

Thread ID

Stack size over time

MATT WebView

Thread ID: 0

2.5 KB
grid_io_hdf_m:dump_grid_data_to_hdf

25.8 KB
grid_io_hdf_m:dump_grid_to_hdf

21.3 KB
solver_incompressible

54.0 KB

Thread ID: 0

Time (seconds)

0.0 11.4 22.7 34.1 45.4 56.8 68.1 78.6

0 26.6 K

77.4
Example from YALES2 with gfortran issue

Many really small allocations
Existing tools / valgrind

- **Valgrind - massif**:
  - Link memory size to functions
  - Take **snapshots** over time.
  - Miss short live allocations
  - Only 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%
```
Existing tools

- **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

![Graph showing memory allocation over time](image-url)
Who give you this address ?

What you want ??
Some users expect 0x1A8E3 as answer to malloc(0) !! ???

Thank you.

QUESTIONS ?