Memory Saving Techniques

Annual Concurrency Forum Meeting
Fermilab

February 5, 2013

1. Session Introduction
2. Kernel-compressed Memory

Assumptions

- The number of cores grows faster than the amount of memory
- Event-level parallelism:
  - Memory consumption per event has to decrease
- **Orthogonal to other parallelization efforts**

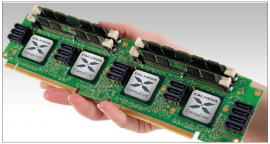
Assumptions The number of cores grows faster than the amount of memory
Event-level parallelism:
Memory consumption per event has to decrease
Orthogonal to other parallelization efforts

- On the Grid: 2 GB per core

Introduction

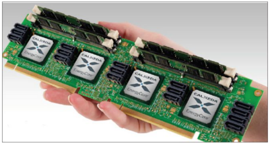
Assumptions	The number of cores grows faster than the amount of memory
	Event-level parallelism:
	Memory consumption per event has to decrease
	Orthogonal to other parallelization efforts

- On the Grid: 2 GB per core
- ARM servers (or: hyper-threading): 1 GB per core/thread

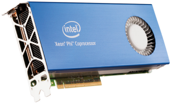


Assumptions
The number of cores grows faster than the amount of memory
Event-level parallelism:
    Memory consumption per event has to decrease
Orthogonal to other parallelization efforts

- On the Grid: 2 GB per core
- ARM servers (or: hyper-threading): 1 GB per core/thread



- Xeon Phi (MIC): 100 MB per core



- GPUs: order of magnitude less memory per core, change of memory model

Explored Memory Saving Techniques

Summary of so far explored memory saving techniques:

Memory Sharing

- Fork and copy-on-write
  Fork should be done reasonably late
- Kernel SamePage Merging
  Sharing is done automatically at the cost of speed
- Multi-threaded application (Geant4-MT)
  Can go beyond page-wise sharing in the fork model

Reduction of Memory Consumption

- Kernel Compressed Memory
  (*zRam*, *frontswap*, *cleancache*)
  Virtual swap area used to compress unused memory
- X32 ABI: x86_64 semantics with 32bit pointers
  Restricts address space to 4 GB
  (which should be acceptable)

These techniques are all (relatively) non-intrusive

Job scheduling

- For memory sharing: jobs with similar input data should be co-scheduled
- In general: a *good mix* of jobs should be scheduled

Techniques provided by the Linux kernel

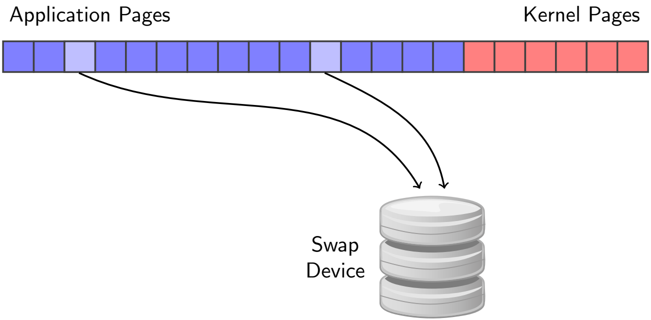
- Many of the new features are not available in SL6
- Virtual Machines can be used to couple a new kernel with an SL6 user land
- Automatically adjusting kernel parameters can be difficult

New platforms

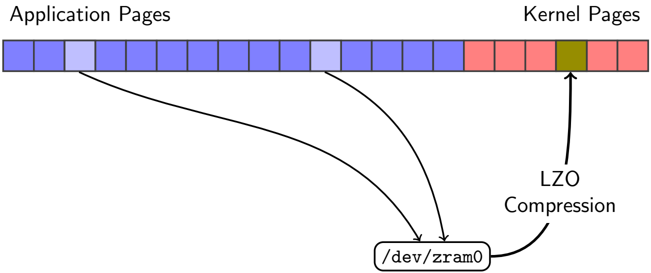
- There might be a need to recompile (and verify) the software stack for **ARM** and/or **X32**

Kernel-compressed Memory – Principle

- Kernel module *compcache* / *zram* provides a virtual block device for swapping
- Originally developed for "small" devices (Netbooks, phones, . . .)
- Part of Kernel >= 2.6.34, can be compiled for SLC6 (with drawbacks)

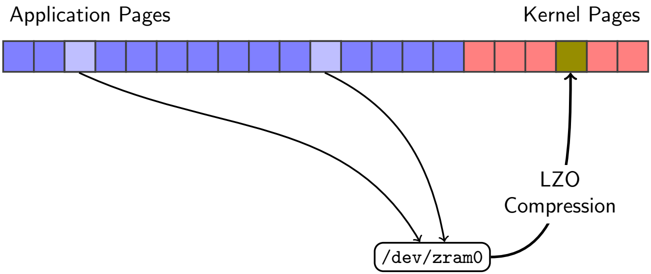


- Kernel module *compcache* / *zram* provides a virtual block device for swapping
- Originally developed for "small" devices (Netbooks, phones, . . .)
- Part of Kernel >= 2.6.34, can be compiled for SLC6 (with drawbacks)



Kernel-compressed Memory – Principle

- Kernel module *compcache* / *zram* provides a virtual block device for swapping
- Originally developed for "small" devices (Netbooks, phones, . . .)
- Part of Kernel >= 2.6.34, can be compiled for SLC6 (with drawbacks)

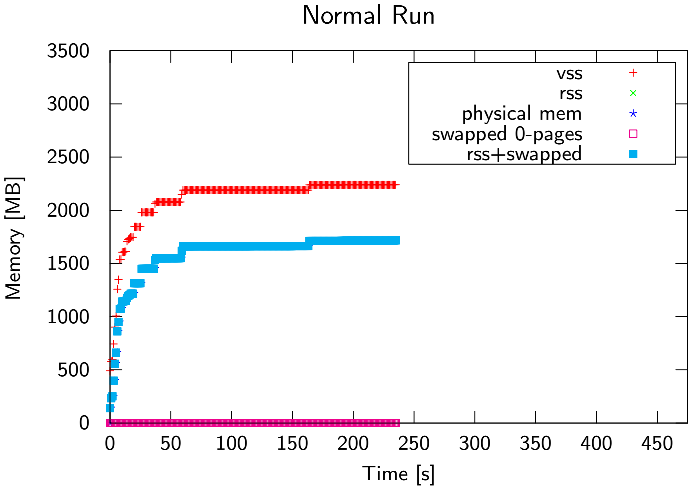


Change in strategy: not swap at all $\mapsto$ swap whenever possible (`/proc/sys/vm/swappiness`)

The system memory pressure and the swappiness are not fine-grained enough handles for measurements

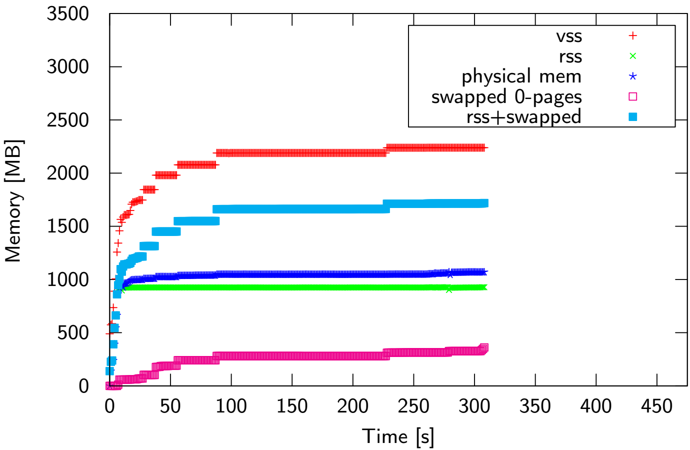
Linux cgroups allow to put the application into a limited memory container:

```
$ mkdir /sys/fs/cgroup/memory/restricted
$ echo $((150*1024*1024)) > \
    /sys/fs/cgroup/memory/restricted/memory.limit_in_bytes
$ echo $PID > /sys/fs/cgroup/memory/restricted/tasks
```



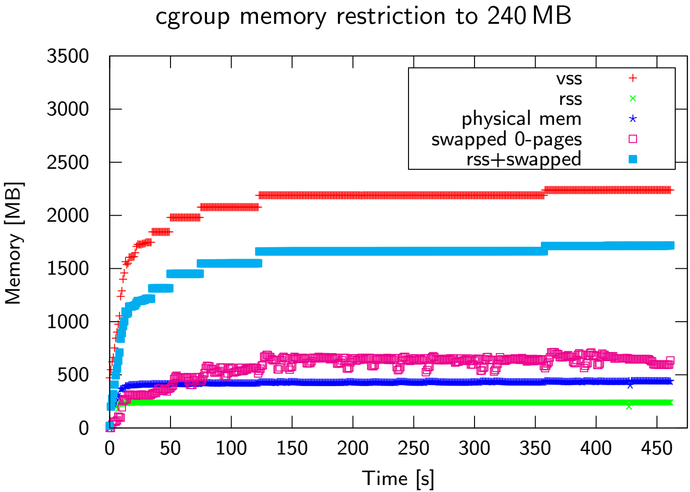
Normal Run
vss
rss
physical mem
swapped 0-pages
rss+swapped
Memory [MB]
3500
3000
2500
2000
1500
1000
500
0
0
50
100
150
200
250
300
350
400
450
Time [s]

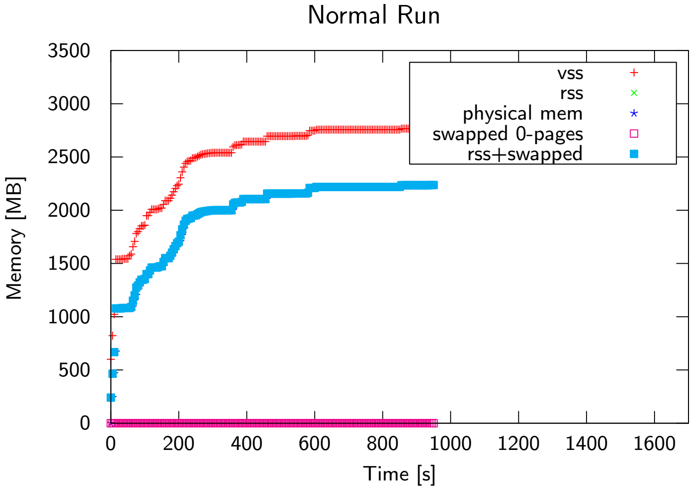
cgroup memory restriction to 950 MB



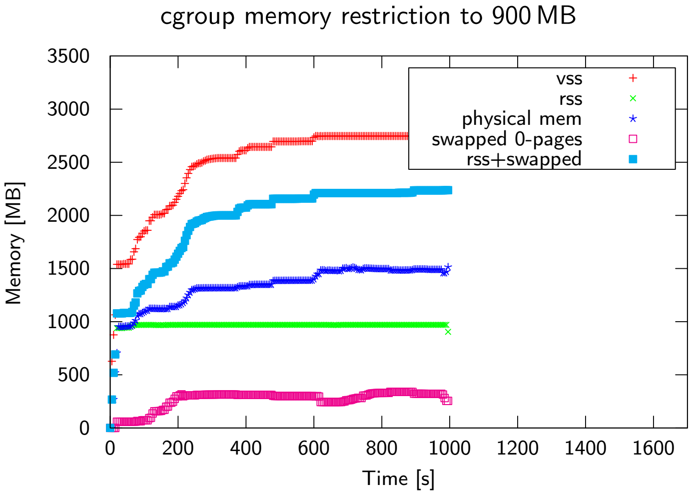
Kernel-compressed Memory – Figures

AliRoot reconstruction of 2 simulated pp Events (v5-04-25-AN)

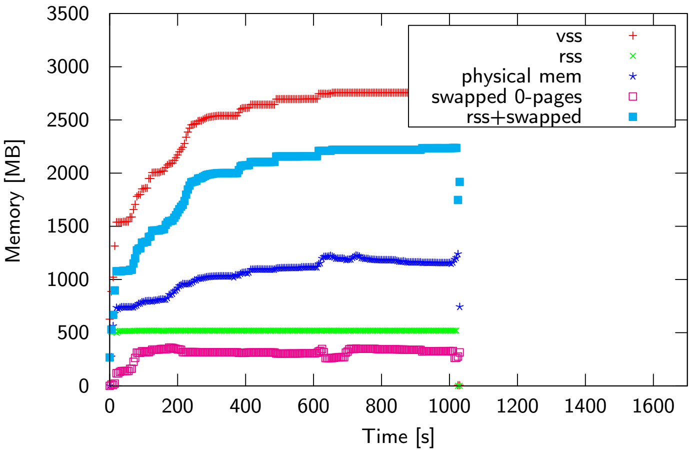


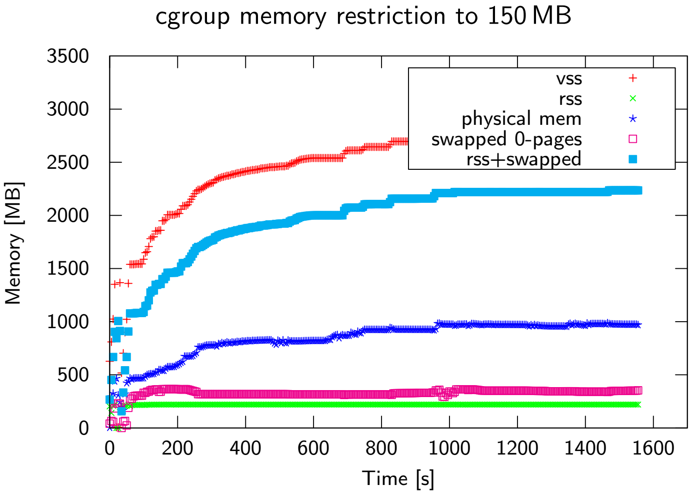


Normal Run
Memory [MB]
Time [s]
0
500
1000
1500
2000
2500
3000
3500
0
200
400
600
800
1000
1200
1400
1600
vss
rss
physical mem
swapped 0-pages
rss+swapped


cgroup memory restriction to 900 MB
Memory [MB]
Time [s]
0
500
1000
1500
2000
2500
3000
3500
0
200
400
600
800
1000
1200
1400
1600
vss
rss
physical mem
swapped 0-pages
rss+swapped

cgroup memory restriction to 450 MB




cgroup memory restriction to 150 MB
Memory [MB]
Time [s]
vss
rss
physical mem
swapped 0-pages
rss+swapped
0
500
1000
1500
2000
2500
3000
3500
0
200
400
600
800
1000
1200
1400
1600

X-Check: scan through a core dump of the application

Can we get rid of these hundreds of Megabytes of continuous zeros?

- No change by using automatic garbage collection (Boehm's GC)
- Zero pages in LHCb DaVinci: $\approx$ 700 MB out of 2.3 GB
- Zero pages in CMS reconstruction
  - 180 MB out of 900 MB without output
  - 280 MB out of 1.4 GB with output

Idea: Inspect `memset()` calls >4 kB

Dead pages (AliRoot reco)

- $\approx$ 40 % zero pages traced back to source code
- Breaks down to half a dozen memsets with high impact
- No hits after detector initialization
- Scattered over uses of TClonesArray

Remaining zero pages

- Excluded: `read()`, `mmap()`
- Excluded: ROOT buffers
- Measurement uncertainties at memset boundaries
- Only literal `memset()` covered, standard constructors:

```
int *a =
  new int[1024*1024]();
```

Next Steps

1. Forensics: Track back large zero-runs to a `malloc()`
2. How to choose zram parameters for an optimal tradeoff wrt. throughput?

   Perhaps zram can also be used as an “overflow” mechanism to make sure that a job finishes