Reducing GLSL Compiler Memory Usage (or Fitting 5kg of Potatoes in a 2kg Bag)

Ian Romanick <ian.d.romanick@intel.com>

Agenda

- Mesa project background
- Problems encountered
- How we fixed it

Mesa

- Open source OpenGL driver stack
- Hardware agnostic shading language compiler front-end and “middle” end

Mesa

- Current compiler started in 2010
  - `wc -l` says ~60k lines of C++
- Uses a talloc “clone” for memory management
  - We call it ralloc
  - Most of the compiler uses it like a mark-and-sweep garbage collector

I believe either Carl Worth or Eric Anholt talked about the compiler architecture and the memory management system at LCA in 2011.

Mesa

- Each pass will...
  - Create a new ralloc memory context
  - Remove old nodes from the IR
  - Add new nodes to the IR
  - Reparent all reachable nodes to the new context
  - Free the old context
    - Any nodes not reparented are automatically freed
    - With one caveat... more on that later

The Problem

- Many games compile MANY shaders at start-up
  - Several Unreal Engine 3 based games are known to compile more than 10,000 shaders
  - Developer build of Dota2 used over 4GB at start-up
  - Many DX games exceed the sandbox capacity in a VM on a Linux host

Different shaders are usually specializations for different effects, number of lights, etc.

The developer build of Dota2 is special because it compiles all of the possible shader permutations... even the ones that will not be used on the current configuration.

Solution

- Approach like an optimization problem
  - Collect some data
  - Look for big fish and low-hanging fruit
  - Fix a problem
  - Repeat until good enough

Instead of “where is the time spent,” determine “what is using storage.” A little more tricky.

Data Collection

- Representative workload
  - Apitrace of a single frame of Dota2
  - “All the shaders” → shader-db
- 3 sets of data collected:
  - Counts of reachable nodes
  - Peak memory usage
  - Data structure utilization

The Dota2 trace initially used ~76MB heap on 32-bit.
That includes the compiler, textures, models, etc.

apitrace

- https://github.com/apitrace/apitrace

shader-db

- http://cgit.freedesktop.org/mesa/shader-db/
- Data for compiler changes:

```
total instructions in shared programs: 5877951 -> 5877012 (-0.02%)
instructions in affected programs:     155923 -> 154984 (-0.60%)
```

For GLSL we have shader-db. Public and private shader-db repos have every shader we could scrape from every open-source project and every closed-source game we could find.

The combined repos represent on the order of 50,000 shaders.

Reachable Nodes

- For each compiled shader:
  - Iterate the reachable nodes in the IR tree
  - Count the number of nodes of each type
- Gives an estimate of which types use the most memory
- Easy using existing visitor infrastructure
  - http://en.wikipedia.org/wiki/Visitor_pattern

Peak Usage

- Valgrind massif FTW

  `valgrind --tool=massif ./a.out`

  - `ms_print` to visualize the results
  - http://valgrind.org/docs/manual/ms-manual.html
- Collect data for 32-bit and 64-bit!
  - Most games are still 32-bit
- Provides before / after data for commit messages

Data Structure Utilization

- pahole
  - https://kernel.googlesource.com/pub/scm/devel/pahole/pahole/
- Understand the output:

```
public:
    /* class ir_instruction      <ancestor>; */ /*     0     0 */
    /* XXX 32 bytes hole, try to pack */
    const class glsl_type  *   type;            /*    32     8 */
    const char  *              name;            /*    40     8 */
```

  - The “32 bytes hole” is the base class, not a real hole

Data Structure Utilization

- Collect data for 32-bit and 64-bit!

```
struct foo {
    char *c;
    int i;
    // Padding on 64-bit, but not on 32-bit
    double *d;
};
```

Free Unused Data

- Eliminate more unused variables

GLSL has many variables built in to the language, but most shaders use few (if any) of them.

By the count-the-nodes metric, dramatic improvement.

By the massif metric, no change.

Pseudo-leaks

- Symbol table is the ralloc context for variables
  - Optimizations can eliminate variables…
  - …but the memory is still reachable from the s.t.
  - Would have been a problem with a proper garbage collector too

Repack Structures

- Death by 1,000 cuts...

Sometimes-dynamic Data

- Time / space trade off

Don't have to be clever in the destructor because of ralloc.

Do have to be smart in the clone() method.

C++ getter / setter methods make these changes possible for non-trivial cases.

Better for small data that can be recomputed.

Use Dead Space

```
class base {
    uint8_t data;
};


class derived : public base {
    void *v;
}
```

Use Dead Space

```
class base {
    uint8_t data;
    uint8_t storage[sizeof(void *) - 1];
};

class derived : public base {
    void *v;
}
```

Be careful about use in subclasses... don't conflict.

Was going to be used for short variable names. This change was not accepted in Mesa, and was made mostly irrelevant by...

Static Flyweights for Common Data

- Most C programs have a variable called `i`

GLSL is no different. Why allocate unique storage for variable names that 10,000 shaders in an application will have?

What is Common?

- Log variable names from compiler

```
grep ^VARNAME output | cut -d' ' -f2 |\
    sort | uniq -c
```

Need a large corpus of representative data, and that's where shader-db comes in.

Performance Problem

- Hash look-up for every name allocation
  - Avoid as many calls to `strcmp` as possible

Bloom Filter

- The “trivial reject”
- N hashes, and M bits
  - If all N bits are set, the data is *probably* in the set
- See:
  - http://patchwork.freedesktop.org/patch/29762/
  - http://patchwork.freedesktop.org/patch/29766/

I used an explicit hash and an implicit hash (the length of the name).

I experimentally settled on 8,192 bits

For 6,749,837 calls to get_static_name, 161,649 strings were in the Bloom filter, and 931 of these were not in the set.

Less than a 1% false positive rate.

Results

Trimmed apitrace of Dota2:

	total(B)	useful-heap(B)	extra-heap(B)
Before (32-bit):	76,337,968	69,720,886	6,617,082
After (32-bit):	65,999,288	60,937,396	5,061,892
Before (64-bit):	106,986,512	98,112,095	8,874,417
After (64-bit):	92,433,072	85,309,100	7,123,972

Memory usage includes textures, vertex data, etc... and still 13% reduction on 32-bit.

Automate!

- Use `git rebase -i -x` to collect data for commit messages

Thanks