Efficient JIT-based remote execution

Google Summer of Code Project with LLVM by

Anubhab Ghosh Indian Institute of Information Technology, Kalyani, India

Mentors: Lang Hames, Vassil Vassilev, Stefan Gränitz

https://compiler-research.org

Code models

From the GCC man page for x86_64:

- -mcmodel=small
 - Generate code for the small code model: the program and its symbols must be linked in the lower 2 GB of the address space. Pointers are 64 bits. Programs can be statically or dynamically linked. This is the default code model.
- -mcmodel=large

-mcmodel=small

• Generate code for the large model. This model makes no assumptions about addresses and sizes of sections.

0000000000000000 <m< th=""><th>ain>:</th><th>000000000000000 <main< th=""><th>>:</th></main<></th></m<>	ain>:	000000000000000 <main< th=""><th>>:</th></main<>	>:
0: 55	push rbp	0: 55	push rbp
1: 48 89 e5 4: b8 00 00 00 00	mov rbp,rsp mov eax,0x0	1: 48 89 e5 4: b8 00 00 00 00 9: 48 ba 00 00 00 00	mov rbp,rsp mov eax,0x0 movabs rdx,0x0
9: e8 00 00 00 00 a: R_X86_64_PLT32	call e <main+0xe> func-0x4</main+0xe>	f: 00 00 00 00 b: R_X86_64_64 13: ff d2	func call rdx
e: 5d	pop rbp	15: 5d	pop rbp
f: c3	ret	16: c3	ret

-mcmodel=large

Shared Memory

- LLVM JIT supports running the generated code in a separate process.
 - The RPC scheme Executor Process Control (EPC) is used for communication with the target.
- Communication happens through file descriptors backed by pipes or sockets.
 - This includes all the generated code as well.
- Shared memory allows us to communicate directly with the executor process.
 - Physical memory pages are mapped into both processes.
 - We can write directly to the address space of the executor process avoiding copies. It can save a lot of overhead when moving large data structures like in case of Clang REPL.
 - A syscall is still required to signal events like finalization.
 - Only works when executor is running on top of the same underlying physical memory.

MemoryMappers

- MemoryMapper: an interface to map and unmap memory in the executor process with protections
 - Makes it easy to swap out the code transport mechanism
- InProcessMapper: Implementation directly using sys::Memory APIs
 - Manages memory when running in the same process
- SharedMemoryMapper: The primary shared memory implementation
 - Has POSIX and win32 shared memory support
 - Executor side implemented in SharedMemoryMappperService class

Implementations of this interface can be used by the Mapper JITLinkMemoryManager class struct AllocInfo {
 struct SegInfo {
 ExecutorAddrDiff Offset;
 const char *WorkingMem;
 size_t ContentSize;
 size_t ZeroFillSize;
 AllocGroup AG;

};

ExecutorAddr MappingBase; std::vector<SegInfo> Segments; shared::AllocActions Actions;

S; Beconvoc

virtual void reserve(size_t NumBytes, OnReservedFunction OnReserved) = 0; virtual char *prepare(ExecutorAddr Addr, size_t ContentSize) = 0; virtual void initialize(AllocInfo &AI,

```
OnInitializedFunction OnInitialized) = 0;
```

```
virtual void deinitialize(ArrayRef<ExecutorAddr> Allocations,
```

OnDeinitializedFunction OnDeInitialized) = 0;

virtual void release(ArrayRef<ExecutorAddr> Reservations,

OnReleasedFunction OnRelease) = 0;

Slab-Based Memory Allocator

- This class implements the JITLinkMemoryManager interface.
- It accepts a MemoryMapper and uses it for underlying memory management.
- It reserve()s a large chunk of memory on first allocate() and returns smaller areas from that.
 - The chunk is already mapped in the address space so allocate() is almost free.
 - \circ It avoids some overhead of going through EPC and repeatedly calling mmap().
 - As the whole chunk is contiguous, this provides compatibility with small memory model on most architectures.
- Freed blocks are also returned to the available memory pool for reuse.

How to use?

- It is already integrated in Ilvm-jitlink tool with both in-process and shared memory use case.
- MapperJITLinkMemoryManager :: CreateWithMapper() allows supplying a MemoryMapper.
- The SharedMemoryMapper requires ExecutorSharedMemoryMapperService to be enabled in the executor process.
- Creating a shared memory mappers requires a bit more set-up with the symbols of the service for RPC.

```
static std::unique_ptr<JITLinkMemoryManager> createInProcessMemoryManager() {
  return ExitOnErr(
     MapperJITLinkMemoryManager::CreateWithMapper<InProcessMemoryMapper>(
          SlabSize)):
Expected<std::unique_ptr<jitlink::JITLinkMemoryManager>>
createSharedMemoryManager(SimpleRemoteEPC &SREPC) {
  SharedMemoryMapper::SymbolAddrs SAs;
  if (auto Err = SREPC.getBootstrapSymbols(
          {{SAs.Instance, rt::ExecutorSharedMemoryMapperServiceInstanceName},
           {SAs.Reserve,
           rt::ExecutorSharedMemoryMapperServiceReserveWrapperName},
           {SAs.Initialize,
            rt::ExecutorSharedMemoryMapperServiceInitializeWrapperName},
           {SAs.Deinitialize.
            rt::ExecutorSharedMemoryMapperServiceDeinitializeWrapperName},
           {SAs.Release,
            rt::ExecutorSharedMemoryMapperServiceReleaseWrapperName}}))
   return std::move(Err);
 return MapperJITLinkMemoryManager::CreateWithMapper<SharedMemoryMapper>(
     SlabSize, SREPC, SAs);
```

How to use?

- It is already integrated in Ilvm-jitlink tool with both in-process and shared memory use case.
 - Look for createInProcessMapper() and createSharedMemoryMapper() in llvm/tools/llvm-jitlink/llvm-jitlink.cpp for an example.
- MapperJITLinkMemoryManager::CreateWithMapper() allows supplying a MemoryMapper.
- The SharedMemoryMapper requires ExecutorSharedMemoryMapperService to be enabled in the executor process.
- Creating a shared memory mappers requires a bit more set-up with the symbols of the service for RPC.

Thank You

Contact me at

- anubhabghosh.me@gmail.com
- LLVM Discord at argentite#0791
- Github <u>@argentite</u>
- LinkedIN: anubhab-ghosh-44b451194

GSoC Work Product with more details: https://gist.github.com/argentite/b265db7604a5ba3c487 83c42cefc6908

More interesting projects: <u>https://compiler-research.org</u>

Better explanation of code models: https://eli.thegreenplace.net/2012/01/03/understanding-t he-x64-code-models

