IVANNIKOV ISP RAS OPEN CONFERENCE
MOSCOW, 5-6 DECEMBER, 2019

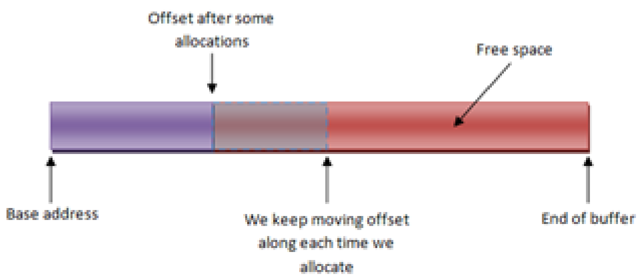
High-Performance Flexible Memory Allocators in Complex Projects

Iliya Trub

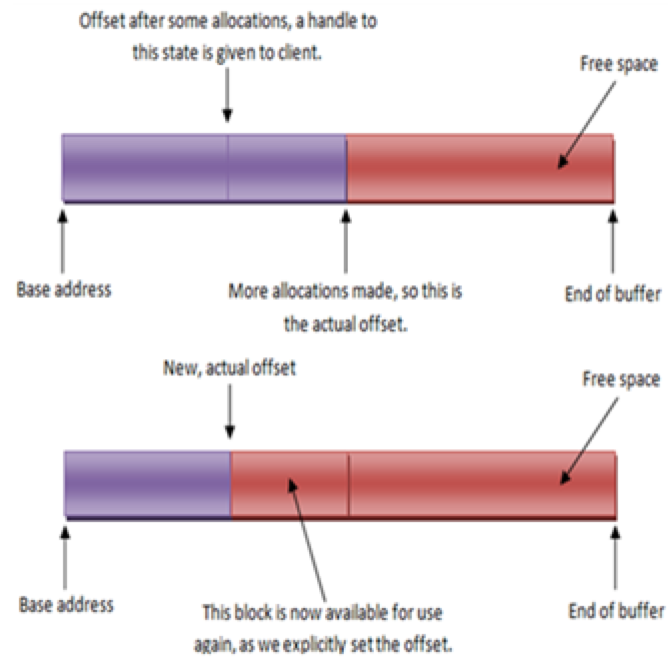
Senior Software Engineer,
AI Compiler Lab,
Samsung Research Russia, Moscow

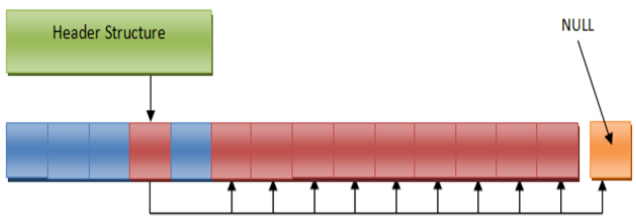
E-mail: itrub@yandex.ru; Phone.: +7-903-240-78-64

Typical custom allocators



Linear allocator





Pool allocator

Stack allocator

Two main questions to apply custom allocator(CA) to large project

- **what to change?** It means, that exhaustive analysis of source code and using algorithms is needed to find what program modules and data structures are appropriate for one or another kind of CA;
- **how to change?** It means the proper choice of CAs' design to make code understandable and expandable.

Custom allocators' basic design

Examples of *StackPool*-level methods:

void* StackPool::getMemory(size_t size) {

return poolLowLevel.getAsStack(size, debugMode);

}

void StackPool::releaseMemory(void *pointer) {

poolLowLevelreleaseAsStack(pointer, debugMode);

}

Examples of *CustomAllocator*-level :

ClientType Allocate(int number) {*

return static_cast<ClientType>*

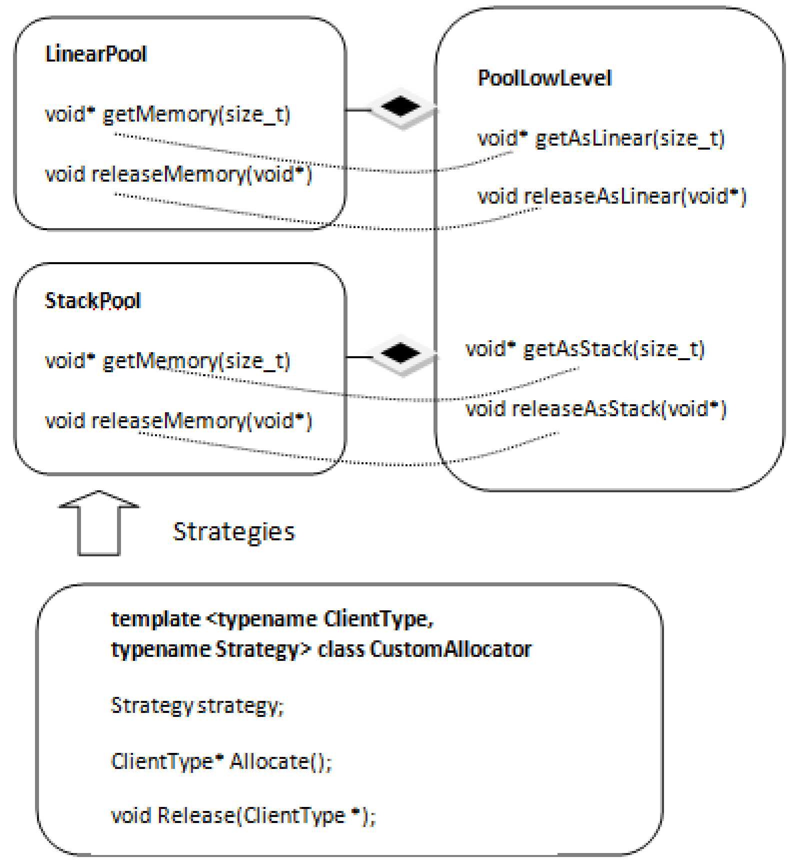
strategy.getMemory(number sizeof(ClientType)));*

}

void Release(ClientType ptr) {*

strategy.releaseMemory((void)ptr);*

}



Examples of application level declarations:

*CustomAllocator<std::vector<int>,LinearPool> *linearPool;*

*CustomAllocator<llvm::BitVector, StackPool> *stackPool;*

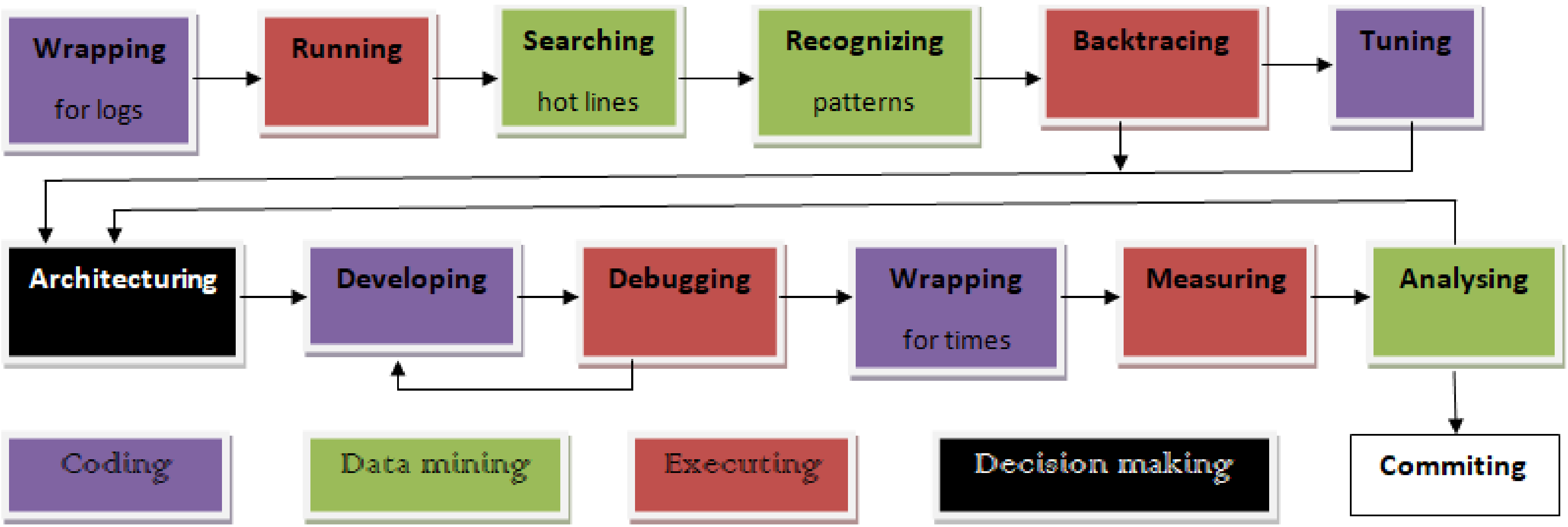
Class PoolLowLevel

- unsigned oneChunkSize;
- void *headChunk;
- unsigned offsetInCurrentChunk;
- void *currentChunk;
- static const size_t maxChunkSize;

```
void PoolLowLevel::releaseAsStack(void *pointer, bool debugMode) {
    if (debugMode) { //determining the size of last allocation
        unsigned *debugPtr = (unsigned *)((char *)pointer - sizeof(unsigned));
        unsigned lastSize = *debugPtr;
        assert((char*)pointer - (char*)currentChunk + lastSize ==
          offsetInCurrentChunk); //the key point of debug mode
        offsetInCurrentChunk -= lastSize+sizeof(unsigned);
    } else {    //rolling back – the only needed line in the most cases
        offsetInCurrentChunk = (char*)pointer - (char*)currentChunk;
    }
    if ((offsetInCurrentChunk == sizeof(void *) + sizeof(unsigned)) &&
          currentChunk != headChunk)) { //avoiding memory leak
        unsigned *offsetInPrevChunk = (unsigned *)((char *)currentChunk
          +sizeof(void*));
        unsigned prevOffset = *offsetInPrevChunk;
        void *oldPtr = headChunk;
        void *ptr = headChunk;
        while (ptr != nullptr) { //searching previous chunk
            ptr = *((void**)oldPtr);
            if (ptr == currentChunk) break;
            oldPtr = ptr;
        }
        currentChunk = oldPtr;
        offsetInCurrentChunk = prevOffset; //rolling back
        *offsetInPrevChunk=0;
    }  //avoiding memory leak
}
```

free-operation of stack allocator

The main workflow of custom allocators implementation



Profiling of memory operations

static void printMem(memoryKind kind, int size, char const logFile, char const*func, int line, void *ptr, void *newPtr, int amount);*

where

- ***kind*** is enumeration, which lists all operations such as *malloc*, *free*, *realloc*, *calloc*, *new*, *new[]*, *delete* and *delete[]*;
- ***size*** is the size of the required memory (0 for *free*s);
- ***memfile*** is the path to the source file;
- ***func*** represents the function/method in the source file;
- ***line*** is the line number in the source file;
- ***ptr** i*s the address of allocated/freed memory, the old address for *realloc*;
- ***newPtr*** is the address of the allocated memory for *realloc*;
- ***amount*** is the number of allocated objects for new.

Examples of wrappers

```
void *wrapper_malloc(char const* memfile, char const*  func,int line, int size)
{
   void *ptr=malloc(size);
   if (ptr) printMem(op_malloc,size,memfile,func,line,ptr,0,0);
   return(ptr);
}

void wrapper_free(char const* memfile, char const* func,int line, void *ptr)
{
   printMem(op_free,0,memfile,func,line,ptr,0,0);
   free(ptr);
}

void* operator new(size_t size, char const* memfile, char const* func, int line)
{
   void* ptr = std::malloc(size);
    if (ptr) printMem(op_new,size,memfile,func,line,ptr,0,0);
    return ptr;
}

void operator delete(void *ptr, char const* memfile, char const* func,int line)
{
    printMem(op_delete,0,memfile,func,line,ptr,0,0);
    free(ptr);
}
```

Examples of replacement

C-style operations

malloc(size) → **wrapper_malloc**(__FILE__, __FUNCTION__, __LINE__,size);

free(ptr) → **wrapper_free**(__FILE__,__FUNCTION__, __LINE__,ptr);

C++-style operations

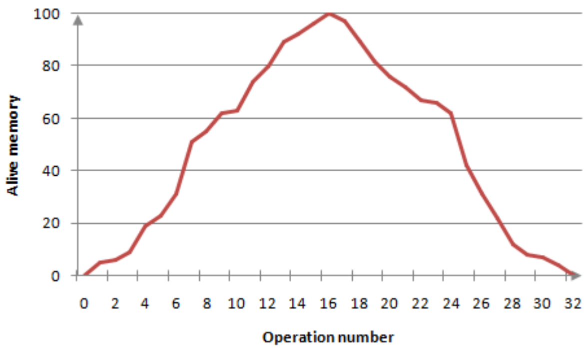
new className(<constructor arguments) → **new**(__FILE__,__FUNCTION__,__LINE)
className(<constructor arguments>);

delete(instance) → **operator delete**(instance,__FILE__, __FUNCTION__,__LINE__);

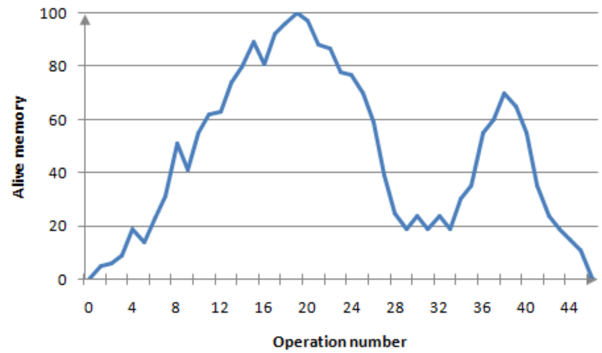
Manual recheck:

- to distinguish *new* and *new[];*
- to determine placement form of *new* when the memory is not allocated, so replacement is not needed;
- to determine if *new* is already overloaded somewhere. Replacement results in compilation error at best or crash at worst if overloading is so involved in inheritance, typedefs and templates and compiler cannot untangle it;
- the last item is especially difficult for *delete*, because we cannot easy recognise, which datatype *delete* is called.

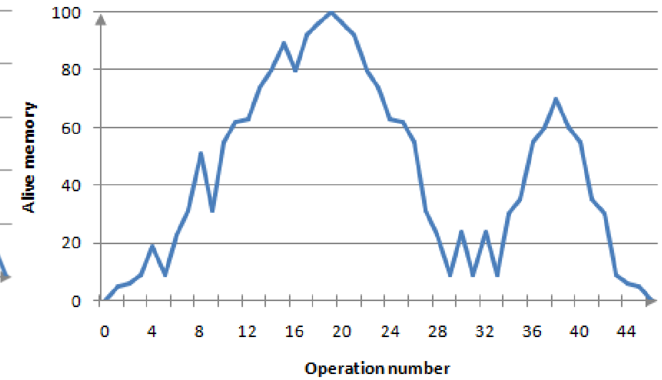
Recognising patterns



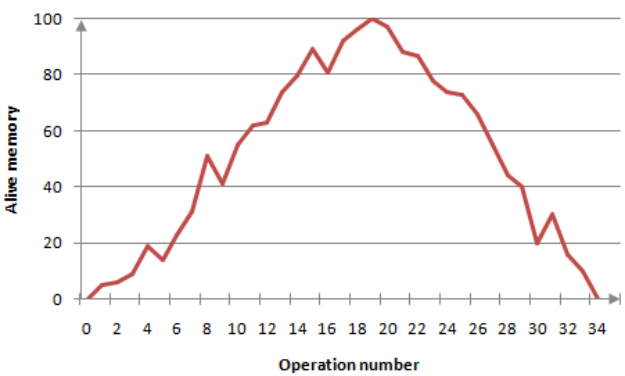
Linear good



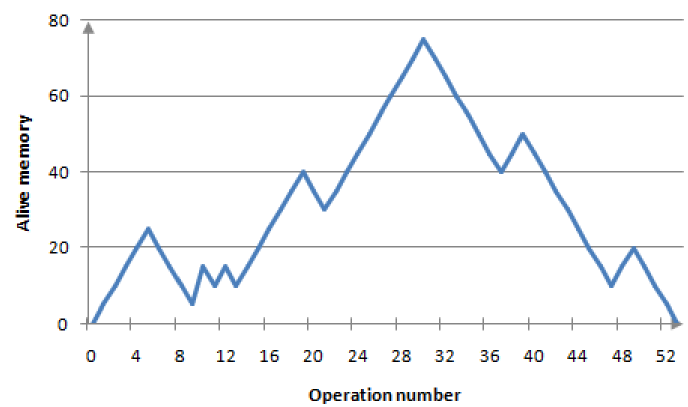
Linear bad



Stack



Linear satisfactory



Pool

Recognising and backtracing examples

Found patterns for LLVM-based shader compiler

Class	Strategy
llvm::StringMap	Linear
llvm::BitVector	Stack
llvm::Use	Linear
llvm::DenseMap::Buckets	Pool
llvm::DomTreeNodeBase	Linear
llvm::SmallVector	Stack

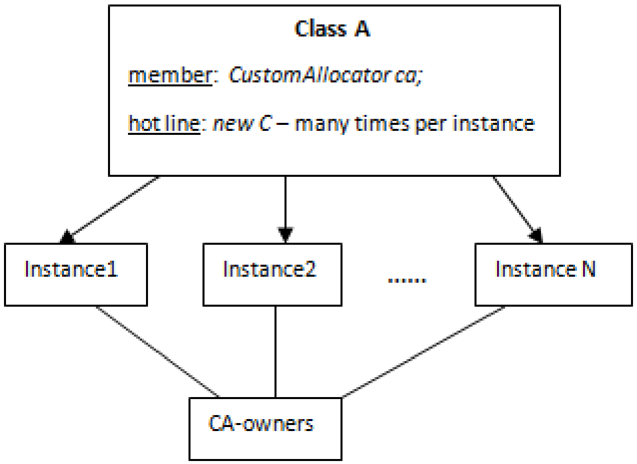
Results of hotline backtracing for llvm::BitVector

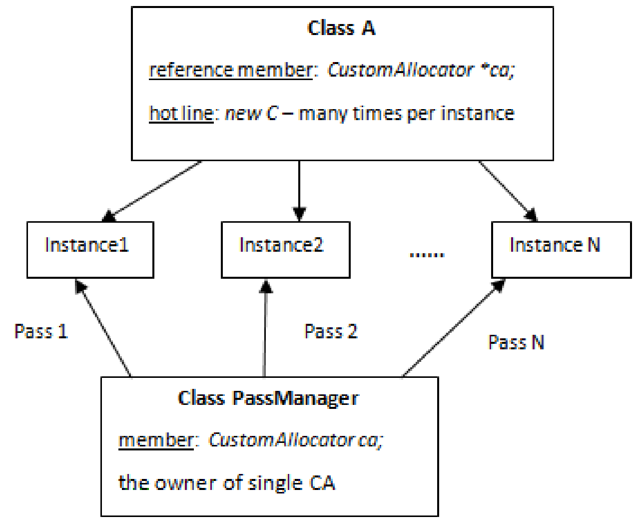
Pass	Percentage
MachineInstructionScheduler	45
OptimizeOutput	20
RegAllocGreedy	12
ValueConvergence	10
ShaderSplitting	7
DagToDagISel	4
MachineFunctionAnalysis	2

Architecturing. Main questions

1. What class will the owner of the CA instance belong so that it will be created and deleted ?
2. What class will the consumer of CA belong so that it will get and free allocated memory?
3. What is the optimal size of the chunk?
4. How to pass the reference on CA instance from the owner to the consumer?
5. How consumer will address CA in order to allocate memory or how to rewrite the hotline?
6. What *free*/*delete* operations we must rewrite to fit the rewritten hotline without a runtime error?

Architecturing. Patterns 1 and 2





Pattern1. Owner and consumer are the same

A: llvm::StringMap
C: llvm::SringMapEntry

Pattern2. One owner, few consumers

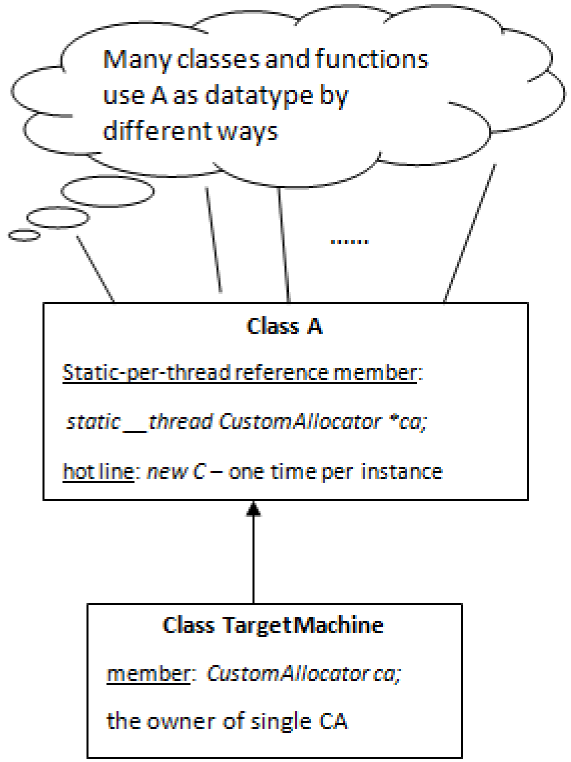
A: llvm::DominatorTree
C: llvm::DomTreeNodeBase

```
static void *operator new(size_t,
CommonAllocator<DomTreeNodeBase,LinearPool> *ca)
{
  return(ca->Allocate());
}
```

Call:

```
new(ca)DomTreeNodeBase(<constructor args>);
```

Architecturing. Pattern 3



Pattern 3. One owner, may consumers

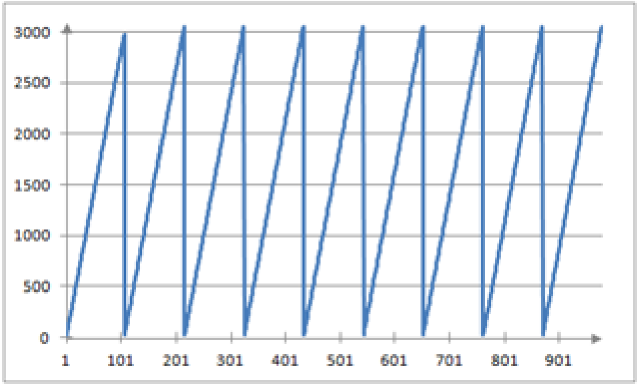
A: llvm::BitVector
C: inner memory of BitVector

Wrapping for times

The typical code:

```
struct timespec tv1, tv2;
long int k1, k2;
…………….
clock_gettime(CLOCK_REALTIME, &tv1);
<wrapped original or replaced alloc/free operation>
clock_gettime(CLOCK_REALTIME, &tv2);
k1 = tv2.tv_sec – tv1.tv_sec;
k2 = tv2.tv_nsec – tv1.tv_nsec;
if (k2<0) {k1--; k2+=1000000000; }
LOG << “Line ….” << (k1*1000000000+k2) << “\n”;
```

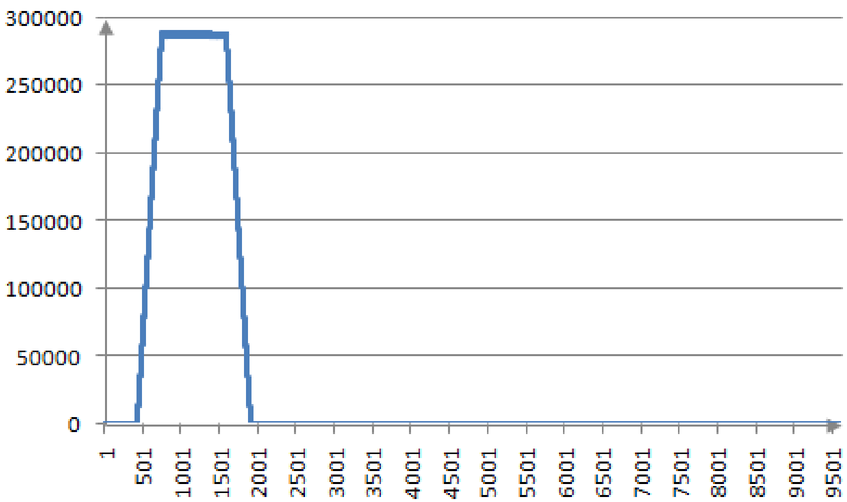
Experimental behaviour of linear allocator



Shader size is 16 Kb.
Allocates instances: llvm::DomTreeNodeBase (28 bytes).
Maximum volume is 3.1 Kb.
The length of trace is 978 operations.
Chunk size is 4 Kb.
Each increasing line: 109 sublines with the same step 28.

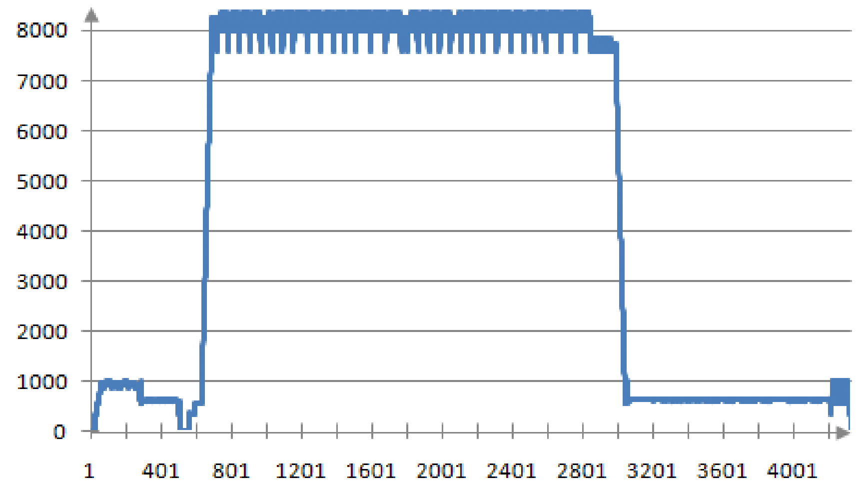
Experimental behaviour of stack allocator

Compiling large shader



Shader size is 16 Kb.
Allocates instances: llvm::BitVector.
Maximum volume is 290 Kb.
The length of trace is 9 622 operations.
Chunk size is 8.1 Kb.

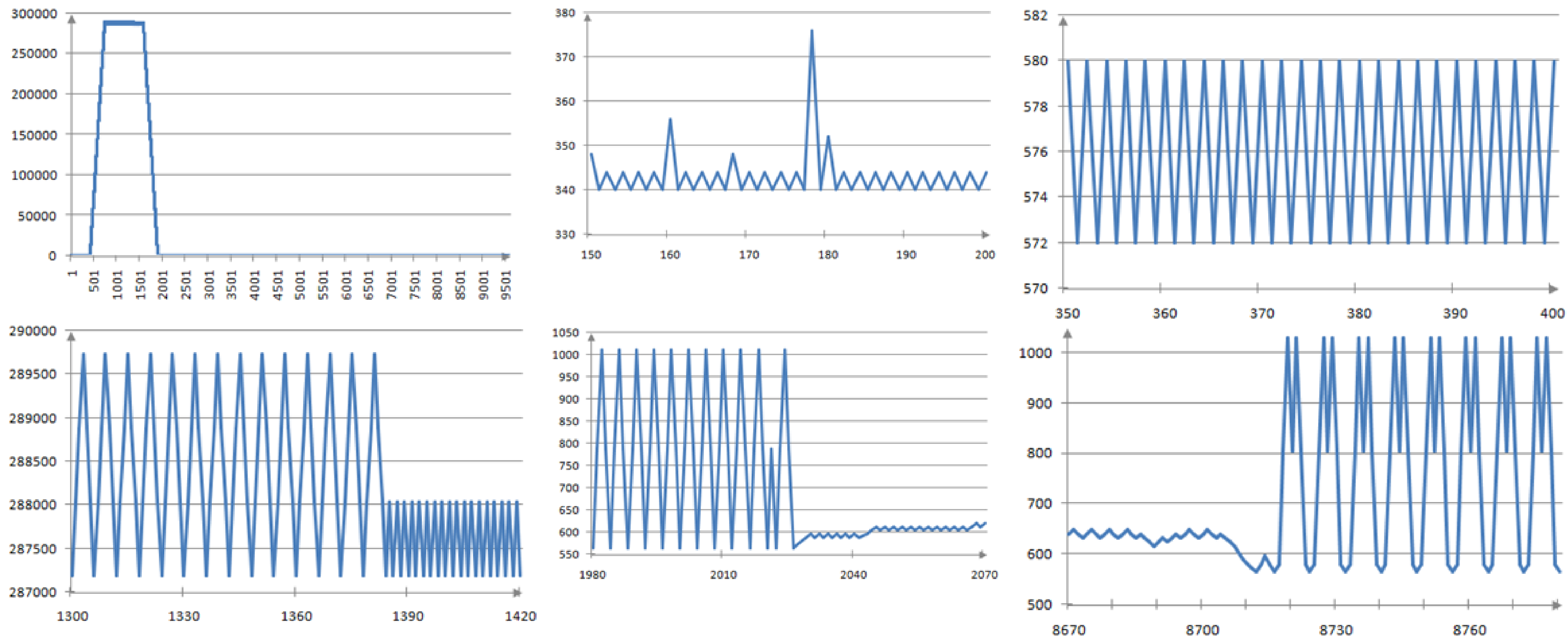
Compiling medium shader



Shader size is 5 Kb.
Allocates instances: llvm::BitVector.
Maximum volume is 8.4 Kb.
The length of trace is 4 326 operations.
Chunk size is 8.1 Kb.

Experimental behaviour of stack allocator

Compiling of large shader (refined)



Chunk size: 8100 bytes.
Total time for memory operations without stack allocator: **990 mcs**
Total time for memory operations without stack allocator: **500 mcs**
Variance of raw data: **3-4 %**.

Times for one memory operation

Preconditions:

- CPU frequency is 1700 MHz
- "*uname –a*" is "*Linux 4.4.0-145-generic #171-Ubuntu SMP x86_64 GNU/Linux*" .
- Allocation of 8 bytes is required

Times:

- *std::malloc()* takes **80-90** ns;
- stack allocator takes **30-40** ns;
- linear allocator takes **22-27** ns.

The total time of memory operations

Game shader set: 135 shaders, 1008 Kb.
Test shader set: 248 shaders, 235 Kb.
Chunk size is 4 Kb.
Sequence length: 30 executions.

llvm::DomTreeNodeBase
Allocator type: linear

Shader set	Trees	Nodes	Total time, ms, malloc/free	Total time, ms, linear allocator
Game	64	3151	43	30
Set	76	2218	23	17

llvm::BitVector
Allocator type: stack

Shader set	Covered passes	Total time, ms, malloc/free	Total time, ms, stack allocator
Game	7	65	33
Set	7	38	21

The total time of all game shaders compilation

	malloc/free, s	Custom allocators, s
Average time	4.35	3.95
Best time	4.05	3.70
Worst time	4.80	4.40

Task list for program transformation

- The stage "<u>Wrapping for logs</u>": the code transformation task of profiling memory operations;
- Stages "<u>Searching hotlines</u>" and "<u>Recognising patterns</u>": data mining task to identify hotlines and to choose/reject CA for each of them;
- The stage "<u>Backtracing</u>": text processing and classification task to identify consumers of memory which hotline allocates;
- The stage "<u>Developing</u>": the code transformation task to implement architecture;
- The stage "<u>Wrapping for times</u>": the code transformation task of profiling hotlines to measure the sum of consuming times;
- And at last the common control script which joins all stages together.