nvmain学习记录

目标：

> 理解**gem5**与**NVMain**之间是如何通信的，还有**nvmain**具体是如何配置**NVM**的

gem5+NVMain混合编译qpush无报错(+parsec负载)
nvmain wiki
nvsim main page
NVMain Extension for Multi-Level Cache Systems
内存芯片信息
访存时序参数
NVMain运行机制

同一个rank内部不同chip的bank之间是什么关系？bank和subarray又是什么关系？subarray和array是一样的吗？
访存过程是burst length是什么意思？
What is DRAM burst size? in quora

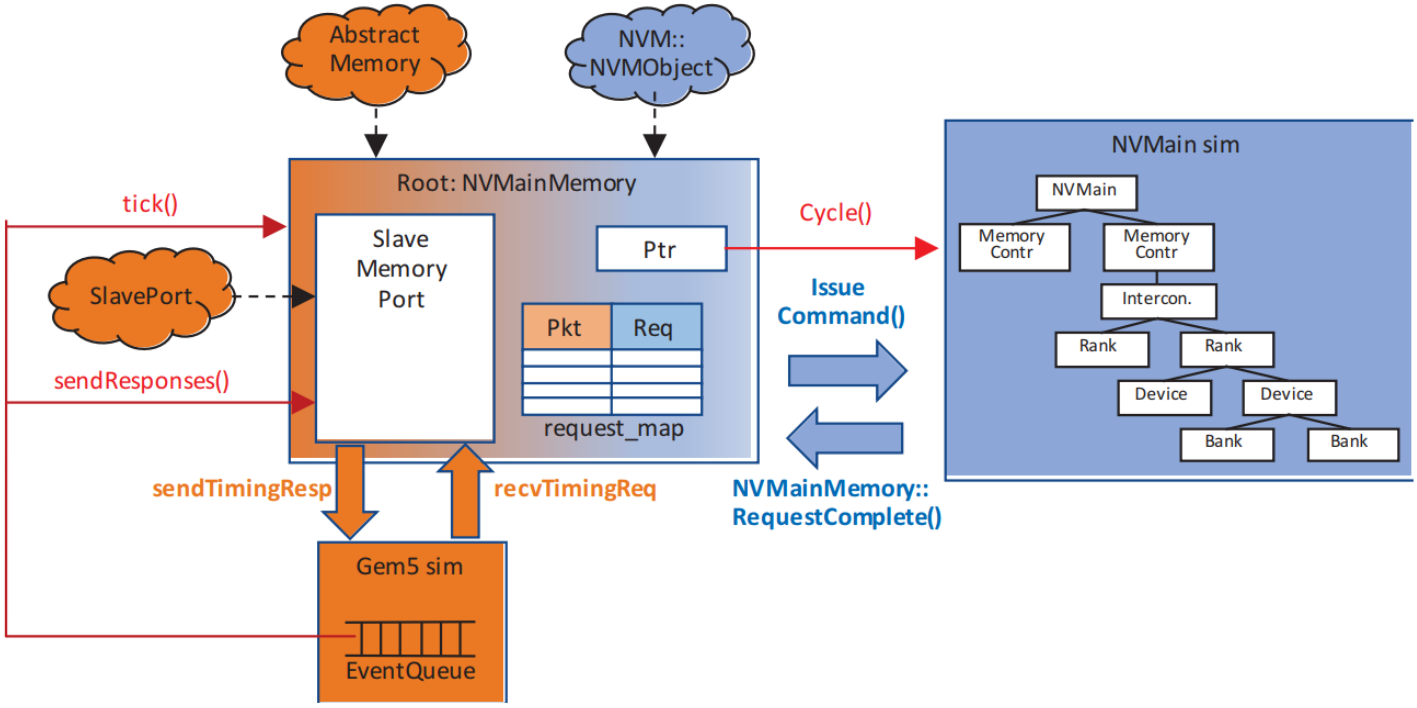
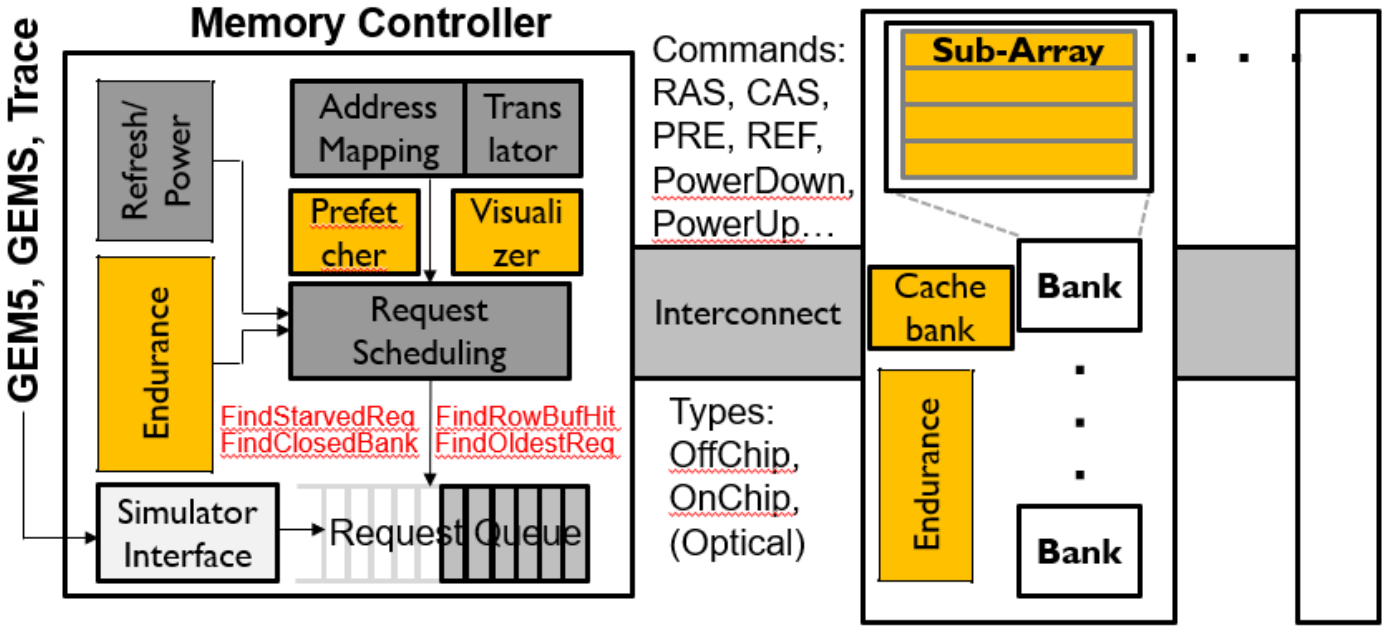


Fig. 4. Software architecture of the gem5/NVMain co-simulation.



DFPC代码实现

主要修改的地方：

```
$ pwd
path to nvmain
```

./Config: NVM的配置文件，改成8GB

./DataEncoders/FlipNWrite/FlipNWrite.cpp FlipNWrite的实现，好像没有修改（自己阅读这个部分代码的时候觉得有点问题）

./MemControl/FRFCFS/FRFCFS.cpp

./MemControl/FRFCFS/FRFCFS.h： 在这个里面实现对写回cache line的压缩

./NVM/nvmain.cpp

./NVM/nvmain.h： 暂不清楚改了什么

./include/NVMDataBlock.h 添加了是否压缩的标志位

./include/NVMDataBlock.cpp

NVMain使用方法

```
oxygen@oxygen-virtual-machine:~/workspace/nvmain$ ./nvmain.fast  -h
Usage: nvmain CONFIG_FILE TRACE_FILE CYCLES [PARAM=value ...]
```

可以看到运行NVMain的时候使用-h参数可以知道NVMain的使用方法，这里特别提到了可以选择主动指定参数的方法，因此当需要开启FlipNWrite的时候不需要每次去NVM的config文件中修改，直接在运行的时候指定数据编码方式就可以了，但是**gem5和NVMain一起运行的时候好像不太方便指定NVMain中的配置变量，之后再试试**

悄悄提一下并不需要用-h参数，只要输入的参数个数小于4个就会输出这个提示

```
    if( argc < 4 )
    {
        std::cout << "Usage: nvmain CONFIG_FILE TRACE_FILE CYCLES [PARAM=value ...]"
            << std::endl;
        return 1;
    }
```

因此可以在运行NVMain的命令行中指定DataEncoder的类型

```
oxygen@oxygen-virtual-machine:~/workspace/nvmain$ ./nvmain.fast Config/PCM_ISSCC_2012_4GB.config Tests/Traces/hello_world.nvt 1000000 Dat
```

那么如果在命令行和配置文件中均给出了某个变量的配置，那么谁的优先级更高呢？--命令行中的优先级更高，因为是先读取了配置文件然后再判断命令行中的参数是否多于4个，多则读取多的配置信息，同时对于覆盖的变量会输出提示

```
std::cout <<  "Overriding "  << clParam <<  " with '"  << clValue <<  "'"  <<  std::endl;
```

关于NVMain输出信息的位置：
stat流是否打开了，打开了就将输出信息全部导入到stat流中，否则输出到标准输出

```
std::ostream& refStream = (statStream.is_open()) ? statStream : std::cout;
stats->PrintAll( refStream );
```

而stat流的设置是在NVM config文件中设置的（或者在命令行中指定，直接在命令行中使用>>重定向？？？），需要在配置文件中给出 `StatsFile path/to/statfile`

```
if( config->KeyExists( "StatsFile" ) )
{
        statStream.open( config->GetString( "StatsFile" ).c_str(), std::ofstream::out | std::ofstream::app );
}
```

在stat设置的相邻位置有个判断IgnoreData的，而且在PCM_ISSCC_2012_4GB.config中是给出了这个参数的配置的 `IgnoreData true` ，后面看看这是干什么的
在./traceSim/traceMain.cpp中根据IgnoreData标志位决定是否设置NVMainRequest中的data和oldData成员（用使用FlipNWrite怕是不要忽略数据哦，试了下设置IgnoreData=false的时候，FlipNWrite的一些输出确实开始不为0了，但是统计结果中的flipNWriteReduction还是总是为0%，感觉有点问题）

```
if( !IgnoreData ) request->data = tl->GetData( );
if( !IgnoreData ) request->oldData = tl->GetOldData( );
```

使用grep的-v选项可以避免每次都输出烦人的Warning信息，虽然作者只是想要提醒这里使用的是默认值，应该也可以从代码上来改这个，因为gem5和NVMain一起执行跑FS的时候可能不太好使用这种方式，后面看看从代码上修改的话好不好

```
./nvmain.fast Config/PCM_ISSCC_2012_4GB.config Tests/Traces/hello_world.nvt 1000000 DataEncoder=FlipNWrite  |grep -v "Warning"
```

NVMain源码分析

目前看了（后面需要整理一下文件的组织逻辑，把这个部分转成**Latex**格式）
单独运行**NVMain**的时候程序的**main**函数位于**./traceSim/traceMain.cpp**中，从逻辑上应该先从这里开始分析，还有对

NVMObject的分析应该处于靠前的位置，因为很多对象都是继承自**NVMObject**
还有一个问题是**NVMain**中**include**了一些**gem5**文件加下的一些头文件，而且路径有点奇怪，到底是什么操作导致**NVMain**能够或者错误地包含到**gem5**地头文件呢？
另外的一个分析重点就是**./Simulators/gem5/nvmain_mem.cc**
$pwd
/home/oxygen/workspace/nvmain

- ./src/Config.h&.cpp
  - 读取./Config中的NVM配置信息（目前看到配置信息中有效的行中包含的配置信息格式是相同的，即为有效行中开始是由参数名称和参数值组成的，后面可以由分号开始的注释信息）到Config中保存
  - 看到Config.cpp
  - params
  - FlipNWrite，减少对NVM写操作时发生的比特翻转
    - 在 `./DataEncoders/FlipNWrite/FlipNWrite.cpp` 代码中写函数是由 `ncycle_t FlipNWrite::Write` 实现的，实际上并没有设置一个是否flipped的标志位，而是将发生flipped的位置的地址记录了下来，而只是将发生翻转写操作的地址记录下来了**invertdata**函数很诡异，甚至感觉写错了

config和para对配置文件的读取和参数设置
NVMDataBlock实现NVM数据块的读写（好像有地方把NVMDataBlock的size设置成了64B）
NVMAddress对NVM地址的设置
NVMainRequest对NVM主存读写请求的一些具体信息
NVMObject
EventQueue（里面涉及到NVMObject中的父子关系，以及回调函数，因此先去看看NVMObject）
parent和children的都是保存NVMObject类型的指针，但是children是vector，这说明了可能有多个孩子但是双亲只有一个，为什么？还有很多函数的参数都是NVMainRequest *？getchild怎么和NVMainRequest的地址扯上关系了？？？里面的decoder又是什么？大概有点理解了，这里的父子关系是主要依次包括了memory、channel与memory controller、channel、rank、bank之间的相互作用关系
config.cpp中也包括一个hooklist，添加方法是在NVM config文件中添加 `AddHook *` ，这样就把token *加到hooklist中了，hook的类型目前只有三种（./Utils/HookFactory.cpp）： `Visualizer`、`PostTrace`和`CoinMigrator` ,暂不清楚这三种的具体意义是什么，不过出了NVM config都默认把AddHook这条配置注释了

./traceSim/TraceMain.cpp里面控制了NVMain的主要输出（main函数也在里面）
其中config->SetSimInterface(simInterface);设置了模拟器接口，而在TraceMain.cpp中 SimInterface *simInterface = new NullInterface( );其被设置为单独运行。而在./Simulators/gem5/nvmain_mem.cc中m_nvmainConfig->SetSimInterface(m_nvmainSimInterface);其中m_nvmainSimInterface = new NVM::Gem5Interface( );接口设置成了gem5，之前**gem5**和**NVMain**一起测试的时候**NVMain**部分的输出全部为**0**，之后看看代码

配置文件中的TraceReader参数决定了NVMain获取负载的方式

```
TraceReader NVMainTrace
```

./traceSim/traceMain.cpp中设置了trace为NVMainTrace

```
GenericTraceReader *trace = NULL;
if( config->KeyExists( "TraceReader" ) )
    trace = TraceReaderFactory::CreateNewTraceReader(
            config->GetString( "TraceReader" ) );
else
    trace = TraceReaderFactory::CreateNewTraceReader( "NVMainTrace" );

trace->SetTraceFile( argv[2] );
```

./traceReader/TraceReaderFactory.cpp中判断CreateNewTraceReader的参数是否为NVMainTrace或者RubyTrace，这两种方式才会正确执行负载阅读器

```
GenericTraceReader *TraceReaderFactory::CreateNewTraceReader( std::string reader )
{
    GenericTraceReader *tracer = NULL;

    if( reader == "" )
        std::cout << "NVMain: TraceReader is not set in configuration file!"
            << std::endl;

    if( reader == "NVMainTrace" )
        tracer = new NVMainTraceReader( );
    else if( reader == "RubyTrace" )
        tracer = new RubyTraceReader( );

    if( tracer == NULL )
        std::cout << "NVMain: Unknown trace reader `" << reader << "'."
            << std::endl;

    return tracer;
}
```

./traceReader/NVMainTrace/NVMainTraceReader.cpp中读取NVMain负载文件，其中负载文件每行的组织为时钟周期数，读写操作标识符，地址，64B大小的新要写入的数据，线程号（在hello_world.nvt中都被设置为0）
里面会根据负载文件读取NVM负载版本号，默认为0，而为0时会将旧数据直接设置为0

```
/* Zero out old data in 1.0 trace format. */
oldDataBlock.SetSize( 64 );

uint64_t *rawData = reinterpret_cast<uint64_t*>(oldDataBlock.rawData);
memset(rawData, 0, 64);
```

而当负载版本号不为0的时候负载文件中新写入数据的下一列就是旧数据

```
bool GetNextAccess( TraceLine *nextAccess );
```
读取负载配置文件的下一行配置该函数将传入的nextAccess指向的TraceLine，获得各个参数之后执行nextAccess->SetLine(nAddress, operation, cycle, dataB
```
int  GetNextNAccesses( unsigned int N, std::vector<TraceLine *> *nextAccess );
```
这个函数就是执行多次GetNextAccess，将接下来的多行负载配置文件读取出来放到保存TraceLine 指针的一个容器中

和这个函数关系比较紧的是NVM负载的配置文件和TraceLine.h和TraceLine .cpp，这两个文件中定义TraceLine对象和基本的操作
./src/TagGenerator.h&.cpp中定义了TagGenerator，这个对象保存了字符串和整数组成的map，整数是调用构造函数时给出的 `TagGenerator( int startId );`，每加入一个为一个模块打上一个tag，下一个的整数就是前一个加一。这个有什么用呢？？？（代码里面给出的解释是： `Generate tags at runtime that are unique to all modules in a system` ）

```
std::map<std::string, int> tagNames;
```

- Gem5Interface和NullInterface 都是SimInterface的子类（看看NVMain与gem5的接口的实现，肯定能够找到为什么gem5和NVMain混合模拟的时候NVMain的输出不正确）
  SimInterface.h&.cpp实现了GetDataAtAddress和SetDataAtAddress，分别用来给出地址获取数据和给出地址和数据对NVMDataBlock进行设置，同时会对每个地址保存一个访问次数的统计值

```
class Gem5Interface : public SimInterface
class NullInterface : public SimInterface
```

NullInterface .h&.cpp正如其名，虽然重新定义了下面几个函数，但是都只是返回一个常数，比如0或者false

```
unsigned int GetInstructionCount( int core );
unsigned int GetCacheMisses( int core, int level );
unsigned int GetCacheHits( int core, int level );

bool HasInstructionCount( );
bool HasCacheMisses( );
bool HasCacheHits( );
```

Gem5Interface.h&.cpp
实现了打*的三个函数，其他有返回值的函数只是简单地返回一个常数
在GetCacheMisses函数中，首先判断是否定义了RUBY: `#ifdef RUBY` ，定义了和没定义是不一样的逻辑（**为什么？**，这个RUBY好像是多核cache一致性的一种），关于GetInstructionCount和GetCacheMisses有点没看很懂，两个函数都是涉及到list<Stats::Info *>类型： `std::list<Stats::Info *> &allStats = Stats::statsList();` ，Info类的定义位于/PathToGem5/src/base/state/info.hh中（这个文件夹下的文件感觉都是用于gem5输出的），GetCacheHits倒是看懂了，比较简单，直接计算上一级和这一级cache的不命中次数之差就是这一级cache的命中次数了

```
*unsigned int GetInstructionCount( int core );
*unsigned int GetCacheMisses( int core, int level );
*unsigned int GetCacheHits( int core, int level );
unsigned int GetUserMisses( int core );//有这个函数，但是没有真正实现它的功能，因为作者说现在还无法区分普通用户和超级用户的访问

bool HasInstructionCount( );
bool HasCacheMisses( );
bool HasCacheHits( );

int  GetDataAtAddress( uint64_t address, NVMDataBlock *data );
void SetDataAtAddress( uint64_t address, NVMDataBlock& data );
```

（这里的逻辑是按照看NVMainTraceReader.cpp中出现的对象来看的）

- 在NVMain.h&.cpp中，看到在析构函数中，通过判断numChannels来决定释放多少个内存控制器，**这里有个小小的疑问：不是一个控制器可以有多个通道的吗？**
  里面有两个记录Cofig指针的成员变量，展示比较奇怪的是channelConfig应该完全可以由config推导出来吧，为什么要新增一个Config *？

```
Config *config;
Config **channelConfig;
```

NVMain.h&.cpp中先后配置了 `Config *config` 、 `AddressTranslator *translator` 、 `Config **channelConfig` （还细心的判断了channel的配置文件的路径是不是从/开始的，不是就要将上配置文件的路径加到channel配置文件前，作者肯定是个暖男

了，虽然感觉配置文件一般不会写成绝对路径，特别是在NVMain代码给出的情况下，但是还是称赞一波作者）
和 `MemoryController **memoryControllers` （控制器的类型感觉就是那些排队算法什么FRFCFS之类的），NVMain模块的名字默认为defaultMemory，在traceMain中调用的时候也显示给出的是 `nvmain->SetConfig( config, "defaultMemory", true );`。添加了NVMain和memoryControllers的父子关系指向后，注册了memoryControllers的状态信息（注册信息的先后应该和最后NVMain的输出顺序有关系，这个后面结合rank、bank和subarray的代码看看）

- 对于AddressTranslator/Decoder的设置，有三种类型： `Default`、`DRCDecoder`和`Migrator` ，在3D DRC中Decoder配置为DRCDecoder，在混合内存中，Decoder配置为Migrator，其余的没有设置Decoder信息，被设置为默认的Default。那么AddressTranslator到底是用来干什么的呢？（**需要再次提一下这里提到了JEDEC-DDR标准，其中bus width为64b，burst length为8，其中的猝发长度是什么意思？**）其中的ReverseTranslate貌似是把row，col，bank，rank，channel，subarray字段的信息拼装到一起并根据TranslationMethod翻译成物理地址。Translate函数则恰好相反，其将物理地址翻译成对应的每个内存域地址信息（事实上有多个包含不同参数的Translate函数的重载，其目标大致都是把地址、请求转换成内存域地址信息）
- TranslationMethod和AddressTranslator有什么联系与区别？TranslationMethod设置了地址翻译的格式，主要设置了row，col，bank，rank，channel，subarray所占位宽与相对顺序。其中SetAddressMappingScheme函数根据配置文件（当配置文件中不存在相应的配置时，则为某个构造函数中设置的初始值，但是这样说太麻烦了，后面直接说是配置文件吧）中的 `AddressMappingScheme R:RK:BK:CH:C` 来设置重新设置地址映射的顺序（NVMain的输出信息中会有部分信息重复出现两次，虽然前后输出的信息稍微有点不同，但是输出重复信息总归看着觉得不是很好）。总的来说就是TranslationMethod中的bitWidths、count和order成员分别保存row，col，bank，rank，channel，subarray寻址所需位宽，实际个数和地址映射时的相对顺序
- 综合考虑一下上面两个小节的内容，首先TranslationMethod根据配置文件中地址映射模式信息来设置row，col，bank，rank，channel，subarray所占位宽与相对顺序。然后AddressTranslator可以根据TranslationMethod将物理地址转换成内存域地址信息，或者反过来row，col，bank，rank，channel，subarray字段的信息拼装到一起并根据TranslationMethod翻译成物理地址。以Config/PCM_ISSCC_2012_4GB.config为例，其中BANKS:4，RANKS:1，CHANNELS:1，ROWS:16384，COLS:1024（tBURST:4），SUBARRSY:1。地址映射模式为： `AddressMappingScheme R:RK:BK:CH:C` ，因此32bits的地址顺序依次为：row(14)，rank(0)，bank(2)，channel(0)，col(10)，lowcol(3)，busoffset(3)，这里一个问题是：配置文件中给出 `tBURST 4 ; length of data burst` 而在构造函数AddressTranslator中给出的 `burstLength = 8;/* the default burst length is 8 to comply with JEDEC-DDR */` 这两者之间到底是什么关系？在计算lowcol时用的值应该是8
- EventQueue.h&.cpp，里面涉及到Event、EventQueue和GlobalEventQueue三个对象。其中event成员函数除了 `void Event::SetRecipient( NVMObject *r )` 外都比较简单，其核心代码如下：

```
std::vector<NVMObject_hook *>& children =  r->GetParent( )->GetTrampoline( )->GetChildren( );
std::vector<NVMObject_hook *>::iterator it;
NVMObject_hook *hook =  NULL;
for( it =  children.begin(); it !=  children.end(); it++ )
{
        if( (*it)->GetTrampoline() == r )
        {
                hook = (*it);
                break;
        }
}
```

看看上面这段操作时干什么的？最后hook是Parent的children成员中包含指向r信息的一个NVMObject_hook指针。
`void EventQueue::InsertEvent( Event *event, ncycle_t when, int priority )` 在最后的插入事件的函数中，这个nextEventCycle到底是什么意思？这种插入方法感觉不太对劲，主要是感觉这个nextEventCycle 有什么意义不清楚。对劲了，**nextEventCycle 就是记录了开始时钟周期数最小的事件的时钟周期**

```
if( when < nextEventCycle )
{
        nextEventCycle = when;
}
```

看完整个InsertEvent系列函数之后，原来插入的方法是通过提供的信息构造一个Event对象出来，在加上时间和优先级插入到一个map中，这个map对每个时钟周期都存在一条优先级List，插入时根据事件的时钟周期信息找到对应的List，然后再根据优先级大小关系插入到List中的恰当位置（优先级越高会插入到List越靠前的位置）（注意这里
的 `NVMObject_hook *recipient` ，看看到底这个钩子是怎么样发挥作用的）

- InsertCallback函数会组装出一个Event对象，然后调用最终的InsertEvent（即是：InsertEvent(event, when, priority);）将回调事件（type被设置为EventCallback）插入到EventQueue中。需要注意的是对于构造一个回调事件不需要设置Event的req成员，但是需要额外设置回调方法
  *去除事件时RemoveEvent需要根据时钟周期信息和事件本身在map中定位到需要去除的事件，这件事情本身并不复杂，需要注意的时候删除元素可能会导致List甚至整个map为空的情况，下面更新nextEventCycle又是为了什么？
  nextEventCycle到底是什么意思？

```
if( eventMap.empty() )
{
        nextEventCycle =  std::numeric_limits<uint64_t>::max( );
}
else
{
        nextEventCycle =  eventMap.begin()->first;
}
```

注意一下Loop函数，感觉实际上就是在向前推进currentCycle，从Loop进入Process函数时，条
件 `nextEventCycle == currentCycle` 都是得到保证了
在Process函数中按优先级顺序处理map开始时钟为nextEventCycle对应的eventList（或者说从前向后从eventList中取出event）。根据event的类型，recipient指向的NVMObject执行对应的操作（Cycle、RequestComplete和GetCallback），然后更改lastEventCycle的值为nextEventCycle，以记录上一个处理eventList的开始时钟周期，并将nextEventCycle指向map中的下一个非空值。同时需要额外注意Cycle的参数是**nextEventCycle - lastEventCycle**，这是什么意思？总的来说Loop向前推进currentCycle，然后Process将所有开始时钟周期为currentCycle的事件按照其类型执行相应的操作，最后nextEventCycle后移指向最近的下一个EventList（也会去掉已经完成指向的EventList）
GlobalEventQueue有成员变
量 `std::map<EventQueue *, double> eventQueues;` ， `void GlobalEventQueue::AddSystem( NVMain *subSystem, Config *config )` 将NVMain的事件队列（EventQueue）（NVMain对象的EventQueue成员继承自NVMObject）加
到 `std::map<EventQueue *, double> eventQueues;` 中，其中第二参数为从配置文件中获得的CLK参数
`ncycle_t GlobalEventQueue::GetNextEvent( EventQueue **eq )` 函数在eventQueues指向的多个EventQueue中找出下一个事件的时钟周期数最小的那个EventQueue，其将这个最小的时钟周期数作为返回值（需要稍微注意一下，由于内存系统和CPU采用了不同的时钟频率，因此这里返回的时钟周期数是乘了放大因子转换成CPU的时钟周期数了），同时让eq指向那个具有最小下一个事件开始的时钟周期数的EventQueue
`void GlobalEventQueue::Cycle( ncycle_t steps )` 有点类似Loop函数，先GetNextEvent找到包含最早的下一个事件的EventQueue，然后这里大致是先使用localQueueSteps推进包含最早下一个事件的EventQueue，然后使用globalQueueSteps推进其他剩余EventQueue，但是这里存在很大的困惑。感觉最终的效果是都推进
到 `ncycle_t nextEvent = GetNextEvent( &nextEventQueue );` 了，**这里到底是为何什么？**
`void GlobalEventQueue::Sync( )` 将eventQueues中所有的EventQueue对象进行同步，也就是将其中包含的所有时钟周期数小于（**有可能会出现大于的情况吗？**）GlobalEventQueue的currentCycle矫正到内存时钟后的值的EventQueue对象向前推进

到相同的时钟周期（GlobalEventQueue的currentCycle矫正到内存时钟后的值）
到这里EventQueue就看完了，现在重新回到traceMain.cpp

到主流程中来的时候再看看AddSystem函数，把NVMain对应的EventQueue插入到GlobalEventQueue的eventQueues中了，然后对应的时钟频率设置成了配置文件中的 `CLK*1000000` 。然后 `nvmain->SetConfig( config, "defaultMemory", true );` 中创建了内存控制器。然后类似的一级一级地再SetConfig中创建了Interconnect、rank、bank、subarray（最后在subarray中加入了EnduranceModelFactory和CreateNewDataEncoder），还有几个比较重要地步骤是simInterface（单独运行NVMain时被设置为NullInterface）的设置，TraceReaderFactory（这个在配置文件中均配置为NVMainTrace），然后用从命令中获取到的trace文件路径来配置trace。用命令行中的模拟时钟周期数来配置simulateCycles（会根据CPU时钟和内存时钟的差异将该值进行修正）。嘿嘿，激动人心的时刻就要到来了，广告之后马上回来>>>>>>不！开玩笑的。
首先时判断当前时钟周期是否到达命令行设置的时钟周期数限制 `while( currentCycle <= simulateCycles || simulateCycles == 0 )` ，然后开始从NVMain负载配置文件中读取下一条traceline（失败则尝试drain，这里需要说明一下只有所有子类drain的返回值均为true的时候，这个父类的drain的返回值才会为true），然后就是利用读取到的traceline构造request，根据该条traceline的开始时钟周期时间向前推进globalEventQueue（或者直接说增加currentCycle），然后 `GetChild( )->IsIssuable( request )` 试探当前内存控制器是否可以接受下一条命令，不行就继续向前推进globalEventQueue，可以的话就增加 `outstandingRequests` ，然后将访存请求发送下面的对象 `GetChild( )->IssueCommand( request );` 。**这里没有看到显示地设置request的cycle，不知道是怎么回事?**
在NVMObject.h中CalculateStats被声明为virtual：

```
virtual  void  CalculateStats( );
```

在NVMObject.cpp中递归调用所有孩子的CalculateStats函数：

```
void NVMObject::CalculateStats( )
{
    std::vector<NVMObject_hook *>::iterator it;

    for( it = children.begin(); it != children.end(); it++ )
    {
        (*it)->CalculateStats( );
    }
}
```

随后设置输出流，然后将所有的stats打印到输出流中（这里可以想一想打印的顺序问题，是和注册的顺序有关吗？先注册其孩子的stats然后再注册自己的，这样是不是就保证了先输出孩子的stats然后输出父亲的，起到先详细到最底层，然后是上一层的总体统计信息）：（未完）

```
GetChild( )->CalculateStats( );//return  children[0];使用这个函数的时候需要确保只有一个孩子，因此系统里面只有一个NVMain对象，那感觉GlobalEventQ
std::ostream& refStream = (statStream.is_open()) ? statStream :  std::cout;
stats->PrintAll( refStream );
```

nvmain->SetConfig(config, "defaultMemory", true); 这条线大概可以一直追下去到subArray

- 一个有点奇怪的事情是在traceMain.cpp和nvmain.cpp中均设置了SimInterface的config成员

```
traceMain.cpp 170-171
simInterface->SetConfig( config, true );
nvmain->SetConfig( config, "defaultMemory", true );
nvmain.cpp 118-119
if( config->GetSimInterface( ) !=  NULL )
config->GetSimInterface( )->SetConfig( conf, createChildren );
```

NVMain在SetConfig函数中开始配置channels个memoryControllers，包含对channelConfig的设置（根据配置文件中是否存在CONFIG_CHANNEL_i，i为数字，来决定是否读取相应的通道配置文件或者直接使用之前的NVM配置文件中的信息），依据通道配置文件中的控制器类型确定创建什么类型的控制器

```
memoryControllers[i] =
MemoryControllerFactory::CreateNewController( channelConfig[i]->GetString( "MEM_CTL" ) );
```

这条路还没分析完，现在先转到**memorycontroller**去

`if( p->PrintPreTrace || p->EchoPreTrace )` NVMain.cpp的225行这里可能是要生成.nvt类型的负载

while(!GetChild( )->IsIssuable(request))
GetChild( )->IssueCommand(request);
这两条线应该差不多

还有outstandingRequests是如何在命令完成之后减少的，这也需要追下去一下

src/MemoryController.h&.cpp以及衍生出的各种排队算法，这里以MemControl/FCFS/FCFS.h&.cpp为例进行分析