# 基于多芯粒集成的 X86 指令集共享式储存仿真器说明书

## 1. 简介

### 1.1. 仿真器简介

多芯粒系统是一种能降低芯片设计成本并提高复杂片上系统良率的新设计范式。多芯粒系统的设计空间比单芯片的片上系统大得多。为了支持早期的设计空间探索，仿真器显得尤为重要。然而，由于以下原因，大规模的多芯粒系统无法使用现有的开源多核/众核仿真器进行模拟：1）缺乏精确的芯粒间互连模型；2）无法支持精确互连模型的大规模并行模拟。

因此，本文提出了一种由 gem5 和 popnet 构建的基于多芯粒集成的 CPU 共享式储存仿真器，能够支持精确互连模型的大规模并行模拟。

仿真器架构为所有的 chiplet（gem5）都连接到同一块共享储存上，如图 1。其中 chiplet0 的共享储存和其他 chiplet 的共享储存统称为共享储存。所有 chiplet 对共享储存的读写权限均是相同的。

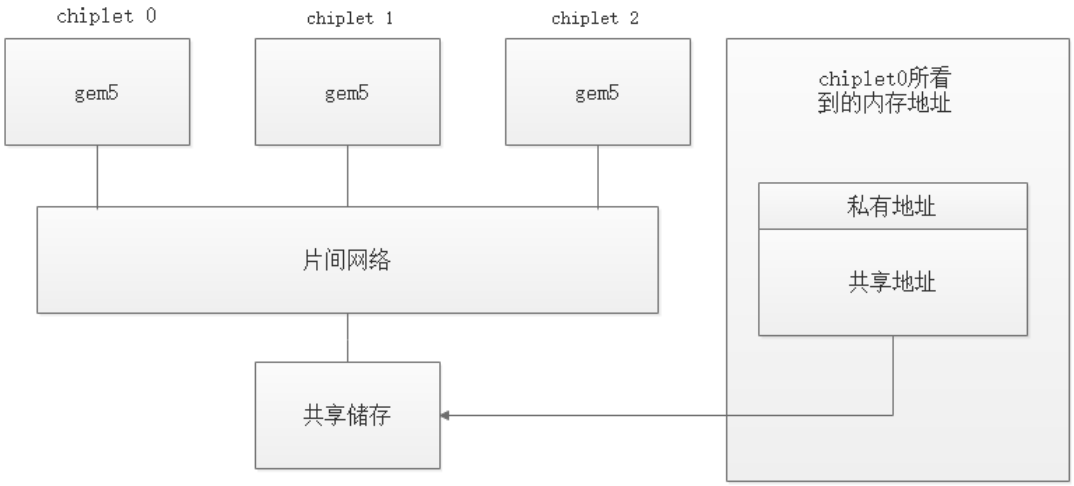


图 1 仿真器架构参考图

### 1.2. 基于多芯粒集成 CPU 共享式储存仿真器

基于多芯粒集成 CPU 共享式储存仿真器是一款用于模拟多芯粒系统中 CPU 芯粒的仿真软件。它的主要功能是：模拟在 CPU 组成的多芯粒系统中，一个操作系统中某个或多个应用程序运行的过程，并给出运行结果以及各项性能指标，如程序的运行时间等。

## 1.3. Gem5 简介

gem5 是融合了原有的 gems（a memory timing simulator）和 m5 (a CPU simulation framework) 而成的一个新的芯粒模拟器，gem5 提供灵活的模块设计，并且保证了非常高模仿在功能和时间上的正确性。在此之上，gem5 还支持操作系统级别的仿真。

gem5 底层是一个以事件为驱动的模拟器。

官网：https://www.gem5.org/

关于 gem5 结构的简单解释（如图 2），我们使用 gem5 模拟器启动一个操作系统，并且在操作系统里面启动应用程序，在此之上，我们可以测量测试程序的运行时间等参数。在本仿真器中，时间等参数又程序自己定义函数测定。


图 2 gem5 结构的简单介绍

### 1.3.1 gem5 的原理

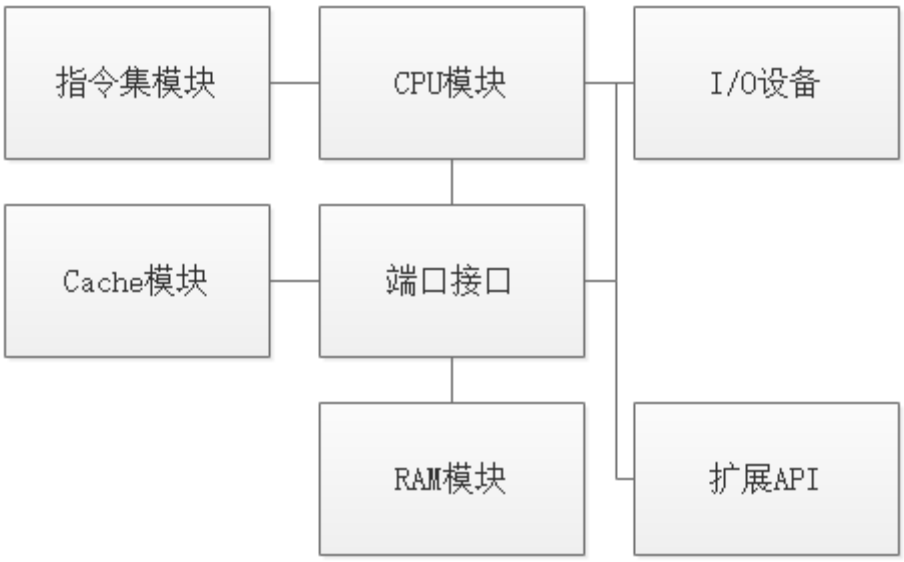


图 3 gem5 底层结构介绍

gem5 模拟器在原理框架上分为许多模块，主要的模块如图 3 所示。

在指令集模块，里面包含 ARM, X86, RISC-V, SPARC, POWER 指令集，此模块在编译时便会确定，例如后面使用 scons 命令编译 gem5 时，其中包含了 build/x86/gem5.opt 字段，其中的 x86 便指定了指令集模块。

在 CPU 模块，里面有 Decoder, Interrupts, Memory 小模块，注意，此处的 memory 模块仅仅是为了与端口接口连接而存在的模块，并不是 CPU 里面的 Cache 模块。Decoder 可以理解为一张函数映射表，它描述了操作系统对硬件保留的低地址（寄存器）操作时的调用情况，后面在讲解 m5 调用时将会详细讲解该调用情况。Interrupts 是 CPU 的中断模块，与一般的单片机的向量中断类似。

在 I/O 模块中，里面是各种 I/O 的模拟函数，比如读取硬盘，总线，网卡等硬件的模拟函数。

在 Cache 模块，模块当中是 cache 的模拟函数，他们可以像正常的 cache 一样，模拟高速的读写储存。

在 RAM 模块，里面包含 DDR4 LPDDR5 HBM GDDR5 等储存的模拟函数，他们都有一致的对外接口。

扩展 API 是除了之前提到模块的所有其他未分类模拟函数的集合。

而最重要的是端口接口模块，接口模块中有一个 cycle 变量，每次比如指令集，CPU，I/O 等模块里面的模拟函数被调用，他们都会把本次模拟函数中定义的时间与接口模块的 cycle 做加法。而又因为 gem5 是单线程程序，保证了时序性上的正确性。

而为了实现不同的 chiplet 的通信，我们修改了指令集模块以及 CPU 模块当中的 Decoder。具体为：1）在指令集模块内部添加函数 gadia_call()，gadia_receive()。2）在 CPU 模块的 Decoder 模块中，在空闲的地址段添加映射。gadia_call() 映射到 0x01455，gadia_receive()映射到 0x01456。3）添加函数 m5_gadia_call()和 m5_gadia_receive()到

ROOT/include/m5ops.h 文件 4）添加函数映射 M5OP(m5_gadia_call,M5OP_RESERVED1）和 M5OP(m5_gadia_receive, M5OP_RESERVED2)到 ROOT/include/gem5/asm/generic/m5ops.h 文件。其中把 gem5 文件夹定为 ROOT。

注：上一段的 4 中，具体为什么函数映射要写成那样，读者可以直接阅读上一段中提到的 m5ops.h 文件，其实 M5OP_RESERVED1 和 M5OP_RESERVED2 对应的 gem5 硬件地址是 0x01455 和 0x01456。

### 1.3.1 目前多芯粒版本 gem5 的仿真原理

如图 4 所示，有 3 个 chiplet 的共享存储被划分 3 个区域。对于 chiplet 而言，每个 chiplet 拥有自己独一无二的编号和自己的独特的共享储存区域，但是该储存区域可以被任何人读写。每个共享储存区域以 communication 开头表征，被所有 chiplet 共享。然而，当 chiplet 数目较多时，会因为搜索范围过大而读操作慢。并且，共享存储由文件模拟，繁杂的文件锁容易使得程序崩溃。综合上述两者原因，共享存储被规划成多块。即共享储存被拆分多个 communication 区域。图 4 中 chiplet0 独特的共享储存是图 4 的 communication0 区域。图 4 中 chiplet1 独特的共享储存是图 4 的 communication1 区域。图 4 中 chiplet2 独特的共享储存是图 4 的 communication2 区域。但是，不论是 communication0，communication1 亦或是其他 communication 开头的区域，他们都可以被任何的 chiplet 访问。当 chiplet 需要从其他 chiplet 接收数据时，它会以自身 chiplet 编号对共享储存对应的区域进行检索，当检索到储存中有之前未读且停留在共享储存最久的数据时，它会将其读入自己的储存中，并记录该数据已读，而具体的共享储存结构，请参考图 4。

例如图 4 中，假设初始时共享储存刚刚被初始化完成（内容全部为空），chiplet0 向 chiplet1 发送字段 1，共享储存的 communication1 文件中会被追加“仿真器 cycle 0 1 1 \n”字段，其中“仿真器 cycle”由 chiplet 自己确定。当 chiplet1 发出读请求时，chiplet1 只会搜索 communication1 文件（它不会去搜索其他共享储存的区域），然后 chiplet1 会读到刚刚追加的那一行。

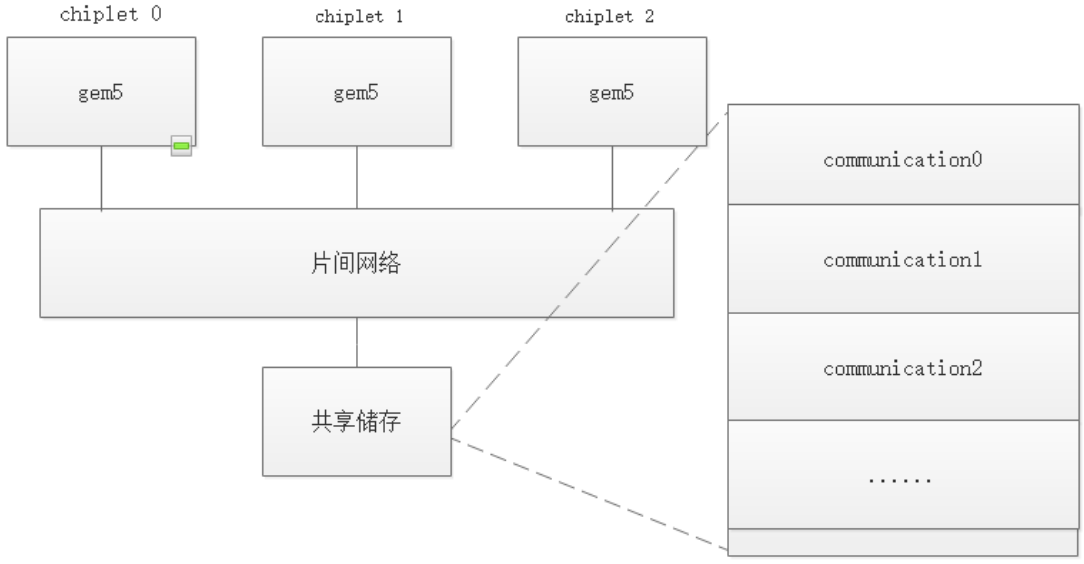


图 4 gem5 共享存储结构图

#### 1.3.1.1 共享储存读操作

在功能模型上，读操作包括根据 chiplet 编号搜索储存，查找最久的未读数据以及标记本次读入的数据（若没有读入数据则跳过）。在 gem5 中，gadia_receive(a)函数执行读操作，a 表示本次要搜索的 chiplet 编号对应的储存。每次执行该函数时，gem5 将访问 communicationa 文件，并读取该文件中最久没有被读取过内容读取完成之后将标记该数据已读并返回该数据，否则返回特定的数字(uint64_t)-1 的值。

在时序模型上，读操作的时序由写操作的时序模型一起计算。

#### 1.3.1.2 共享储存写操作

在功能模型上，写操作包括根据 chiplet 的编号查找对应储存地址，写入数据。在 gem5 中，gadia_call(a, b, c, d)函数执行写操作，其中 a 为当前 chiplet 编号，b 为数据将要传送的目标 chiplet 编号，c 为数据本身，d 为是否初始化共享储存。当 d 参数不要求初始化共享储存空间时，gem5 将访问文件 communicationb ，并在其内容追加内容 “gem5 当前 cycle a b c \n”。

在时序模型上，每次写操作执行时，每个 chiplet 会自动检测当前的 cycle 数并将其一起写入共享储存，在仿真完成之后将交由 popnet 计算出共享储存的读写操作的延迟 cycle 数。

### 1.4. popnet 简介

popnet 是一款开源的互连网络模拟器，能够根据网络节点间的通信记录信息（trace 文件）计算出网络传递数据包的平均延迟以及总能耗。

## 2. 仿真器的安装与使用

### 2.1. gem5 的安装

#### 2.1.1 运行依赖安装

Gcc(version >= 7.0)
Clang(6 – 10)
SCons(version >= 3.0)
Python3.6++

#### 2.1.2 Ubuntu18.04 一键安装依赖命令

$ sudo apt install build-essential git m4 scons zlib1g zlib1g-dev \ libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev \ python3-dev python3-six python libboost-all-dev pkg-config

#### 2.1.3 编译

使用 cd 命令切换到 ROOT 下，运行$ scons ./build/X86/gem5.opt -j [NumberOfThread]

[NumberOfThread]是读者期望该编译使用多少个线程进行编译，去掉-j 选项默认单线程，建议读者采用多线程编译，一般选为电脑 CPU 核数 - 1 最佳。

## 2.2 popnet 的安装

### 2.2.1 下载 popnet 源码

$ git clone https://gitee.com/hic_0757/popnet_modified.git

### 2.2.2 编译 popnet 仿真器

$ make

## 2.3. 仿真器的输入输出文件

### 2.3.1 单个 chiplet 的输入为一个程序的二进制可执行文件，要求如下：

程序必须放入 img 格式的文件当中（具体放置方法后面示例以及制作 img 有对应的说明

用户必须记住自己将程序放入到 img 格式文件中的位置

在 gem5 的操作系统启动完成之后，用户需要自行使用 cd 命令移动到 B 中自己记住的程序位置并自己启动程序

在 gem5 开始多芯粒仿真后，用户需要在逻辑上定义各个 chiplet 的编号（不允许重复）以便 chiplet 之间通信，具体该如何定义以及原因，将在 2.5 中作详细的说明。

### 2.3.2 单个 chiplet(gem5)的输出文件

用户可以得到在 ROOT 目录下的通信 trace 文件，他们是以 communication 开头，以数字（假设为 X）结尾。该文件 communicationX 表示的意思是 X 编号 chiplet 所收到的所有信息。

### 2.3.3 popnet 的输入文件

popnet 的输入文件包括 trace 文件与启动脚本文件。trace 文件记录了不同芯粒之间通信记录，它由多行组成，每一行的结构如下：

```
T sx sy dx dy n
```

T: 数据发送的时间
sx sy: 发送数据的芯粒在网络中的坐标
dx dy: 接收数据的芯粒在网络中的坐标

n: 本次发送数据的数据包大小

启动脚本文件记录了启动 Popnet 的各项配置，它的内容如下：

```
./popnet -A 9 -c 2 -V 3 -B 12 -O 12 -F 4 -L 1000 -T 20000 -r 1 -I ./random_trace/bench -R 0
```

-A 9: 互联网络的大小，代表每个维度上的芯粒数

-c 2: 互联网络的维数，2 代表网络是 2 维的，3 代表网络是 3 维的

-B 12: 输入缓冲区的大小

-O 12: 输出缓冲区的大小

-F 4: flit 大小

-L 1000: 线路长度，以 um 为单位

-T 20000: 仿真周期

-r 1: 随机数种子

-I ./random-trace/bench: trace 文件位置

-R 0: 选择拓扑结构，0，1，2 分别代表不同的拓扑结构

启动脚本文件需要本仿真器用户根据所需要仿真的多芯粒系统进行确定。如何得到 trace 文件将在使用部分进行详细说明。

### 2.3.4 popnet 的输出

Popnet 的仿真结果由控制台打印输出，如图 5 所示：

```
****************************************************
total finished:       11983
average Delay:        37.1263
total mem power:      27.7021
total crossbar power: 2.1895
total arbiter power:  0.00207568
total link power:     3.51097
total power:          33.4046
****************************************************
```

图 5. Popnet 运行结束时控制台输出

此外为了后文计算总 cycle，我们需要的图 5 里的 total finished 和 average Delay。

第一行 total finished 表示总的数据发送量（以数据包为单位），第二行 average Delay 表示每个包平均经过多少个时钟周期数的延时到达目标芯粒。第三行至第六行表示在不同环节的功耗，第七行表示总功耗。

## 2.4 仿真器的时间计算

在 3.1.3 和 3.1.2 中都会提到，我们的测试程序需要自己记录时间，一般采用在程序初始化完成之后，使用 m5_gadia_call()，让 chiplet 向自己发送数据为 0 的包（实际上这个包不会被 gem5 接受，gem5 会默认跳过自己向自己发送的包）。而 trace 文件的四个数字格式我们假设为“cycle src target data”，下列文本是某测试程序的 communication0 文件内容，其中 cycle0 和 cycle1 是为了方便公式计算而标记的，实际的 trace 文件并不会出现

```
2149756845383500(cycle0) 0 0 0
2169756845384000 1 0 -2
    .
    .
    .
```

```
2169824918589000 1 0 -2
2169824918589500(cycle1) 0 0 0
```

在拿到 gem5 的输出（trace）文件后（这里假设上述的 communication0 文件），用文本软件打开，找到两条自己向自己发送的那条 trace，如文件中的 2169756845383500 0 0 0 和 2169824918589500 0 0 0。

仿真器总 cycle 计算公式为 = (cycle1 – cycle0)/1000（该结果需向下取整） + (total finished)*(average Delay)

(total finished)和(average Delay)皆出现在图 5。

结合图 5 的输出和上述计算公式，当前测试程序的时序的计算示例为：

总 cycle = （2169824918589500 – 2169756845383500）/1000 + 11983*37.1263 = 68,073,206(此结果向下取整) + 444,884.4529 = 68,518,090.4529

注：在多个 chiplet 一起计算的时候，我们该选取哪个 trace 文件？这取决于程序本身的设计，比如在 3.1.3 的示例代码中，chiplet0 选择等待 chiplet1 完成矩阵乘法后，才结束程序，此时我们要选择 communication0 文件来计算时序。同理，若让 chiplet1 等待 chiplet0 完成矩阵乘法，则要选择 communication1 文件。一句话来说就是要选择逻辑上最后完成程序任务的 chiplet 的 trace 文件

## 2.5 gem5 仿真器间的通讯函数


图 6. 由 m5 函数调用 gem5 函数的示例图

### 2.5.1 通讯函数调用的简单介绍

测试程序想要调用通讯函数时，它必须通过一个叫 m5 的库来调用底层的硬件模拟函数。图 7 为调用过程的简单介绍，下面将以 m5_gadia_call()为例子，介绍它的调用过程。

gem5 内置一张函数映射表使得 m5 能够调用到 gem5 的函数，大致结构如图 6 所示（参数在此省略），首先 m5 对函数发起 m5_gadia_call()函数的调用请求，然后操作系统先根据函数映射表找到对应的位于 gem5 的函数地址，然后去执行 gadia_call()函数。

而本仿真器对函数映射进行了修改操作，使得我们可以在自己的代码中通过 m5 调用自定义的函数。

### 2.5. 2 在 m5 端的函数定义

实现自己的代码时，读者在修改/调用自定义的函数时，应该 include 位于 ROOT/include/gem5 的 m5ops.h 头文件，本程序提供的自定义函数为 m5_gadia_call，以及 m5_gadia_receive 两个函数。

```
uint64_t result = m5_gadia_receive(0);
```

在上句代码中，m5_gadia_receive 来自上述的 m5ops.h，其参数类型 uint64_t，它的意义为查询是否有给编号 0 Chiplet 的信息，如果没有，则返回-1 的 uint64_t 值，若有，则返回在没有读过消息的队列中最久的消息。

```
m5_gadia_call(a, b, c, d);
```

在上句代码中，m5_gadia_call 来自 m5ops.h，其参数的意义表示为：a 为本 Chiplet 编号，b 为要传递消息的 Chiplet 编号，c 为数据本身，d 表示是否初始化共享储存。以上参数类型皆为 uint64_t

## 2.5. 3 在 gem5 端的函数定义

实现被调用的函数的定义以及实现分别位于 ROOT/src/sim/pseudo_inst.hh，以及 pseudo_inst.cc。调用端口的函数对应关系分别为 m5_gadia_call -> gadia_call 以及 m5_gadia_receive ->gadia_receive。他们都是通过操作系统的地址映射，从而实现了函数的调用关系。

读者可根据自己的实际需求修改在 gem5 端的函数实际行为。但在编译自己写的程序时，请使用$$(CXX_COMPILER) -lm5 -L[ROOT/include] [源代码文件]

## 2.6 gem5 的使用

## 2.6.1 仿真器的工作流程

本仿真器工作流程如图 7 所示

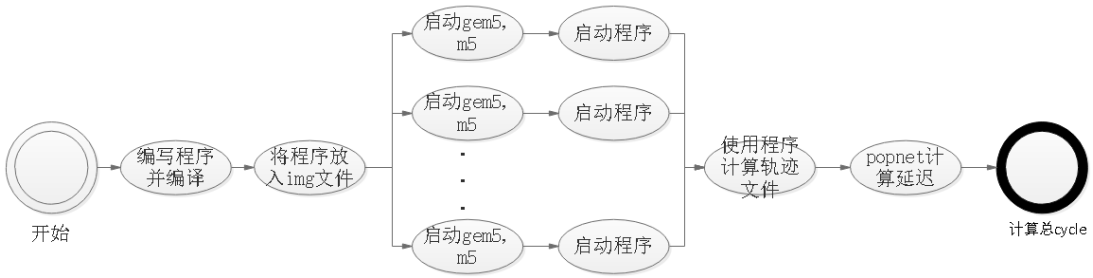


图 7 仿真器工作流程图

Step1:编写多机版本的程序。

Step2:使用 2.8 中提到的挂载 img 文件方法将 img 挂载在电脑上。

Step3:将刚才生成的可执行文件复制到 Step2 中的 img 文件里面，并记住该

位置。并卸载 img 文件。

Step4:启动 gem5，m5。

Step5:在 m5 中找到 Step3 放入的可执行文件并启动。

Step6:在 m5 中输入 ChipletNumber 和 TotalChipletNumber。(示例代码 3.1.2 中定义并要求输入。但也可能根据代码的实际情况有不同的输入情况，此处给出的仅是示例代码的参考输入)。

Step7:正常编译附录 A 的程序并将其放入 ROOT 内。

Step8:运行 Step7 的程序会得到以 bench 开头的 trace 文件。他们全是 popnet 要求的输入文件。

Step9:以 Step8 中 bench 开头的文件为输入，使用 popnet 计算片间通信。

Step10: 当 popnet 将所有 trace 记录处理完成后，会显示仿真结果。根据 2.4 的示例计算将总 cycle 计算出来。

### 2.6.2. 运行一个 gem5

首先，使用 cd 命令切换到 ROOT 下，运行

$ ./build/X86/gem5.opt ./configs/example/fs.py --kernel=[vmlinux 二进制文件] --disk-image=[img 格式的系统镜像文件] -n [cpu 核数]

其次，使用 m5 连接 gem5 使用 cd 命令切换到 ROOT 下，运行

$./utils/tern/m5term 127.0.0.1 [在启动 gem5 之后，这个数字会给出]

数字可以在 gem5 弹出的信息中找到，例如下面这条就是 3456

system.pc.com_1.device: Listening for connections on port 3456

### 2.6.3 运行多个 gem5

如果读者想启动 n 个 gem5，则运行 4 中的命令 n 次即可

### 2.6.4 在 gem5 中运行自己的程序

读者在 10 中运行自己的镜像时，把程序文件放入自己想放入的文件夹中后。读者可以在 m5 找到自己之前放入的文件夹中找到并启动

## 2.7 popnet 的使用

将附录 A 的代码复制到你的工作文件夹下，并用 g++正常编译它，再把 gem5 的所有以 communication 开头的输出文件与编译出来的程序放在同一个文件夹下，运行编译出来的程序，输入仿真器模拟时候的总 chiplet 数量，程序会自动计算，分类 trace 并生成 popnet 能够直接使用的 trace 文件，然后将生成的 bench 开头的所有文件作为 popnet 的输入文件，以 **2.3.3** 的教程，启动 popnet 即可。

## 2.8 运行程序示例

首先编译自己写的程序，假设目录如图 8 结构，而之后的为代码文件的内容

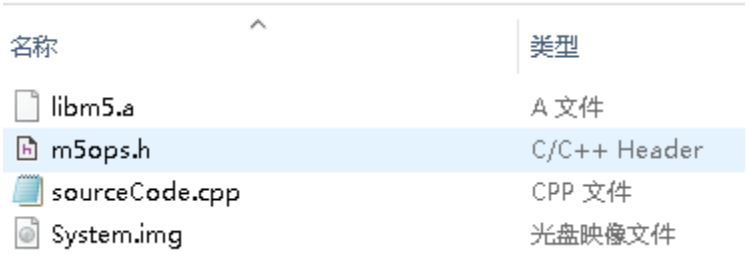


图 8 示例目录结构图

sourceCode.cpp 代码内容

```
#include “m5ops.h”

Int main(){

    m5_gadia_call(0, 1, 2, 0);

    uint64_t result = m5_gadia_receive(0);

    return 0;

}
```

首先使用 cd 切换到该目录下，运行$g++ -lm5 -L./ ./sourceCode.cpp -o a.out

这里给出一种移动文件进入 img 文件的方式

$sudo mount -o loop,offset=1048576 ./System.img /mnt

$sudo cp ./a.out /mnt/

$sudo umount /mnt

使用 4 中的教程启动 gem5，使用 m5 连接后

$/a.out

注：libm5.a 位于 ROOT/include 目录下。m5ops.h 位于 ROOT/include/gem5 目录下

# 3. 自己编写程序与工作负载的映射

## 3.1 分析程序的任务如何并行化

设计程序的第一步是分析程序的任务如何并行化。并行化分为两个层级：如何并行化程序使其适合芯粒运行；如何并行化程序使其负载能够映射到每个芯粒上。

### 3.1.1 如何并行化程序使其适合芯粒运行

根据前面介绍芯粒的架构特点，我们需要将一个程序划分为不同 chiplet 独立工作的范围。

程序的输入：大小为 m*n 的输入矩阵 A 与大小为 n*k 的输入矩阵 B。

矩阵的输出：结果矩阵 Q。

矩阵乘法的操作过程如图 9 所示

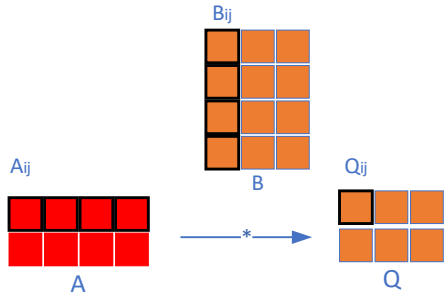


图 9. 矩阵乘法的计算过程

操作过程的两个特点如下：首先，对于结果矩阵 Q 来讲，位于（i，j）位置的数值与位于其他位置，如（i+1,j），的数据没有依赖关系。其次，计算结果矩阵中每个位置的数值时，操作完全相同。

因此，我们可以根据结果矩阵 Q 中的大小来人为划分不同芯粒的工作，使得每个芯粒能够计算出一个或者一部分矩阵 Q 里面的值。

所以在之后的 3.1.3 的矩阵代码乘法示例当中，以两个 chiplet 为例子计算两个 500*500 规模 A,B 矩阵的乘法。每个 chiplet 都分别保存着 A,B 矩阵的全部信息。在划分上也非常的简单，一个 chiplet 计算 Q 从第 1 行到第 250 行的结果，而另一个 chiplet 计算从 250 行到 500 行的结果。最后在再把他们合并。而具体的分解过程如图 10

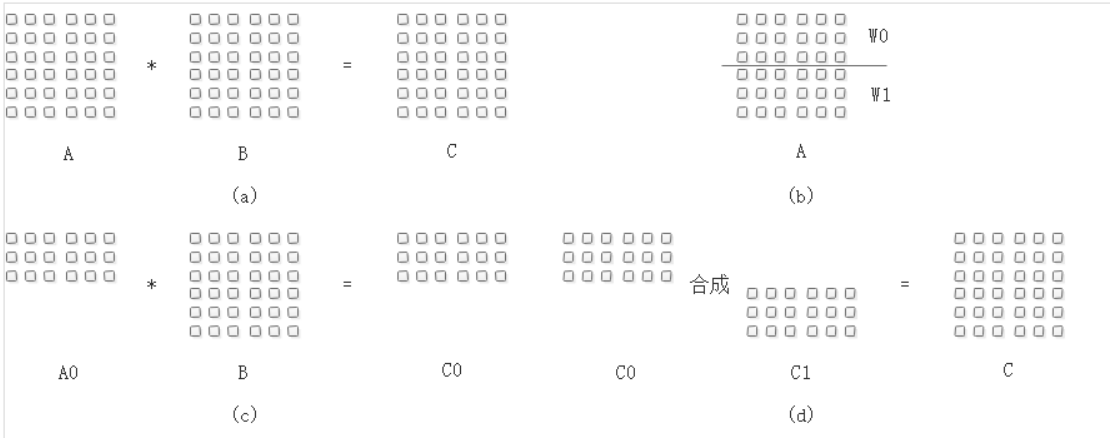


图 10　(a) 未被分解的原始矩阵; (b)对输入矩阵的切分;
(c)分解过后，较小的矩阵乘法;(d) C0 与 C1 合成等于正确结果

第一步，如图 10(a) 原始矩阵 A,B 皆是 500*500 的矩阵，在单机情况下，A*B 将会得到 C 矩阵，C 也是 500*500 规模的矩阵。

第二步，如图 10(b)，以两个 chiplet 为例子，将 A 矩阵的第 1-250 行分割成 W0，第 251-500 行分割为 W1。即 W0 是规模大小为 250*500 的矩阵，W1 是规模大小为 250*500 的矩阵。

第三步，如图 10(c)，对于 chiplet0 而言（假设我们把 W0 分配给 chiplet0）。chiplet0 先计算出 A0(W0)*B 的结果 C0 后，chiplet0 将不断从共享储存中请求读数据。对于 chiplet1 而言（假设 W1 分配给了 chiplet1），chiplet0 先计算出 A0(W0)*B 的结果 C1 后，chiplet1 将不断将结果写入到共享储存中。

第四步，如图 10.d，chiplet0 收到来自 chiplet1 的数据，并把 C0 和 C1 有序地合成一个大矩阵 C,C 的规模是 500*500。并且，如此并行化计算出来的结果矩阵 C 与串行化的相同。

第五步，两个 chiplet 的工作负载映射如图 11，将 W0 矩阵映射到 chiplet0，W1 矩阵映射到 chiplet1。chiplet1 和 chiplet2 都保存了完成的 B 矩阵，然后 chiplet0 计算出 C0，chiplet1 计算出 C1，最后将 C1 和 C0 合成为 C。

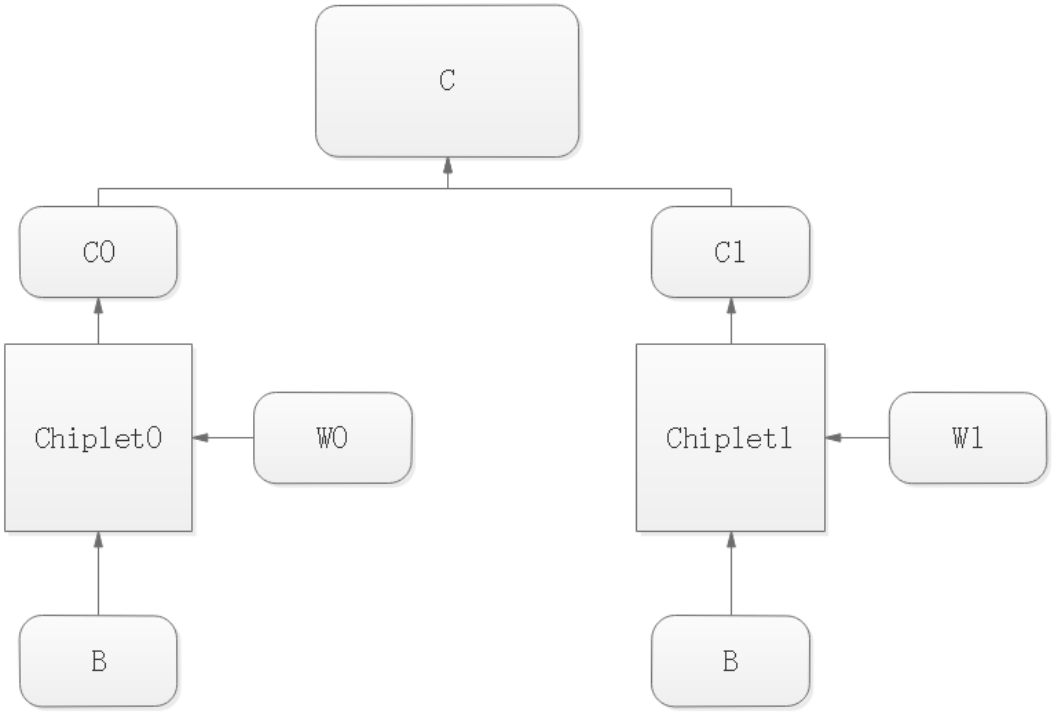


图 11 矩阵乘算法在两个 chiplet 的工作负载映射示例

### 3.1.2 编程示范

我们给 gem5 输入进去的是一个二进制的可执行程序，我们必须在程序代码中定义自己的 chiplet 编号，而一般流程应该是首先声明 chiplet 编号，根据 chiplet 编号分配不同的工作（此处是核心代码区域）。具体编程样板如下：

```
// 代码样板示例
unsigned int chipletNumber = -1; // 初始化变量
unsigned int totalChipletNumber = -1; // 初始化变量
void main(){
    // 初始化程序于此省略
    std::cin >> chipletNumber; // 输入 chiplet 编号
    std::cin >> totalChipletNumber; // 输入总的 chiplet 数量
    m5_gadia_call(chipletNumber, chipletNumber , 0, 0);// 用于记录开始 cycle
    If( chipletNumber == 0){
        Chiplet0_workload(); // chiplet0 的工作负载
/*可以使用 m5_gadia_call()或者 m5_gadia_receive()来同步 chiplet，比如可以大家约定代码先在这里暂停，然后每个 chiplet 使用 m5_gadia_call()给各个 chiplet 发送完成的信息（这里
```

你可以自己定义一个数字作为信号）。发送完成之后再使用 m5_gadia_receive()检测是否全部初始化程序完成（这里可以使用 totalChipletNumber 变量来作为一个完成的标志，比如一个 chiplet 收到了 totalChipletNumber – 1 次此完成信号，就开始跑）。

*/

```
    }else if(chipletNumber == 1){
        Chiplet1_workload();   // chiplet1 的工作负载
    }else if(chipletNumber == 2){
        Chiplet2_workload(); // chiplet2 的工作负载
    }// 其他则仿造即可
    m5_gadia_call(chipletNumber, chipletNumber , 0, 0);// 用于记录结束 cycle
}
// 代码示例
```

### 3.1.3 矩阵乘代码示例

/*

该示例是矩阵乘的工作负载映射代码，此代码的目的是让读者能够更加深入的了解在 gem5 上的编程。

程序将图 10 的所有内容都已经实现，A 矩阵是 matrix，B 矩阵是 matrix1，C 矩阵是 matrix2，他们的规模都是 500*500。与图 10 中不同的是，虽然代码把 A 分成了 W0 和 W1，但是它们都保存了 A 矩阵的全部内容而且 chiplet0 的工作负载是 W1, chiplet1 的是 W0。最后，代码仅仅是在做乘法的时候，不同的 chiplet 用了 A 矩阵不同的行向量。

*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include "gem5/m5ops.h"
/*
```

本示例示范 2 个 chiplet 做 500*500 的矩阵乘工作，假设结果为 C，则 C 的大小为 500*500，用一维矩阵储存，则 chiplet0 计算矩阵乘 C 索引从 0 到 500*249 的结果，而 chiplet1 计算剩下的结果。

```
*/
extern "C"
{
    const int N = 500;
    int main(){

        // 程序初始化开始
        long long *martrix = (long long *)malloc(N * N * sizeof(long long));
        for (int i = 0; i < N * N; i++){
            srand(i);
            martrix[i] = rand()%10;
        }
        long long *martrix2 = (long long *)malloc(N * N * sizeof(long long));
        for (int i = 0; i < N; i++){
            for (int j = 0; j < N; j++){
                srand(i + j);
                martrix2[i * N + j] = rand()%10;
            }
        }
        long long *martrix3 = (long long *)malloc(N * N * sizeof(long long));
        for (int i = 0; i < N; i++){
            for (int j = 0; j < N; j++){
                for (int k = 0; k < N; k++)
                    martrix3[i * N + j] = 0;
            }
        }
        // 初始化结束
```

```
// 初始化 gem5 的相关信息
int chipletNumber = -1;
std::cin >> chipletNumber;
// 完成 gem5 的相关信息初始化
m5_gadia_call(chipletNumber, chipletNumber, 0, 0); // 记录启始 cycle

// 示例的 totalChipletNumber 为 2，故不显式的写出来。
if (chipletNumber == 0){
   for (int i = N / 2; i < N; i++){
      for (int j = 0; j < N; j++){
         for (int k = 0; k < N; k++)
            martrix3[i * N + j] = martrix3[i * N + j] + martrix[i * N + k] * martrix2[k * N + j];
      }
   }
      m5_gadia_call(chipletNumber, 1, -2, 0);
      int position = 0;
      while(true){
            int result = (int)m5_gadia_receive(chipletNumber);
            // 检测 chiplet1 是否完成了矩阵乘的工作
            If(result == -1)
                  continue;
            else if(result == -2)
                  break;
            else{
                  martrix3[position] = result;
                  position++;
            }
```

```
            }
            m5_gadia_call(chipletNumber, chipletNumber, 0, 0); // 记录结束 cycle
            return 0;
        }// the following is responsible for collect
        else if( chipletNumber == 1){
          for (int i = 0; i < N/2; i++){
            for (int j = 0; j < N; j++){
              for (int k = 0; k < N; k++)
                martrix3[i * N + j] = martrix3[i * N + j] + martrix[i * N + k] * martrix2[k * N + j];
                // chiplet 1  把结果写入到共享储存中。
                m5_gadia_call(chipletNumber, 0, (int)martrix3[i * N + j], 0);
            }
          }
        // 告诉 chiplet0,  chiplet1 已经完成矩阵乘法
          m5_gadia_call(chipletNumber, 0, -2, 0);
          return 0;
        }
    }
}
```

## 4. img 文件和 vmlinux 文件的获取

gem5 全系统模拟需要硬盘 img 文件和内核文件，img 文件里面储存着 gem5 将要启动的操作系统及其文件系统。而 vmlinux 文件是内核文件。

1）img 文件可以直接下载

https://drive.google.com/file/d/1HGyHj7fM_kXXDzG7W6Och-A9vuW03jQF/view?usp=sharing

2）可以自行制作 img 文件

使用 qemu 创建

```
qemu-img create ubuntu-test.img 8G
```

使用 parted 命令创建 MBR 表 MSDOS

```
sudo parted ubuntu-test.img
```

成功进入之后

```
mklable

MSDOS

quit
```

安装 Ubuntu

```
qemu-system-x86_64 -hda ./ubuntu-test.img -cdrom [要安装的 ubuntu 的 iso 安装文件] -m 1024 -enable-kvm -boot d
```

就普通安装，但是在给磁盘分区时要注意以下问题

不要有除了/分区以外的任何分区（至少 boot 分区和 swap 分区是不能有的，其余的欢迎探索）。

根目录分区要选择 bootable。

不能有逻辑分区！

3）部署 img 文件，跑安装好的 Ubuntu，进去部署，亦是在此阶段把自己要跑的程序文件移入到该镜像文件

```
qemu-system-x86_64 -hda ./ubuntu-test.img -m 1024 -enable-kvm
```

Ubuntu 15 版本以上直接跑以下代码

```
wget http://cs.wisc.edu/~powerjg/files/gem5-guest-tools-x86.tgz
tar xzvf gem5-guest-tools-x86.tgz
cd gem5-guest-tools/
sudo ./install
```

Ubuntu 15 版本以下先跑上面的代码，肯定会报错的。报错之后再使用 Upstart 去启动 gem5.service，即 update-rc.d gem5.service enable

4）vmlinux 文件可以下载 https://drive.google.com/file/d/1JQ89qmqQU1_KG1zu-A9yEwFJbG3mLo0x/view?usp=sharing

5）也可以前往 https://www.gem5.org/documentation/general_docs/gem5_resources/ 官方网站并在 Linux kernels 一行即可下载自己想要的 vmlinux 文件

## 5. 附录 A

```
// 此程序用于统计共享储存的通信文件并输出 popnet 的输入文件
// 该程序的工作原理是 1 将所有共享存储的通信记录一次性读入程序中。2 读入完成后根据通信记
// 录的目标 chiplet 编号对通信记录分类。3 分类之后按 cycle 排序。4 选择 chiplet0 的 cycle，将其
// 他 chiplet 的 cycle 向 chiplet0 对齐，即同步大家的 cycle。4 将所有 chiplet 的 cycle 排序，将通信
// 记录输出成 trace 文件"bench"。5 将各自 chiplet 在 3 中分类的而形成的类别，分文件输出。
#include <vector>
#include <iostream>
#include <cstdlib>
#include <sstream>
#include <string>
#include <fstream>

class CommunicationRecord;
class Chiplet {
public:
    void setTime(unsigned long long time) {
        this->chipletStartCycle = time;
    }
    Chiplet():chipletStartCycle(0),chipletNumber(-1) {
        //do nothing
    }
    int chipletNumber;
    unsigned long long chipletStartCycle;
    std::vector<CommunicationRecord*> relatedRecord;// if the srcCore number is the
ChipletNumber, the record will be put here
                                                    //
};

class CommunicationRecord {
public:
    CommunicationRecord()
        :cycle(0), srcCore(0), targetCore(0) {

    }
    CommunicationRecord(unsigned long long _cycle, int _srcCore, int _targetCore) {
        this->cycle = _cycle;
        this->srcCore = _srcCore;
        this->targetCore = _targetCore;
    }
```

```
        unsigned long long cycle;
        int srcCore;
        int targetCore;
};

/*
* this Program is aimed to generate the popnet trace file from the communication record
*/
std::string communicationBaseFileName = "communication";
int TotalChipletNumber = 0;
unsigned long long chipletNumberZeroStartCycle;
void sortTheChipletRecord(unsigned long long* cycle, int* sequency, int length);

// 输出bench0.0 bench0.1 等文件
void outputSenderFile(Chiplet* ChipletSet) {
        for (size_t j = 0; j < ::TotalChipletNumber; j++) {
                std::ofstream file;
                int chipletNumber = j;
                std::string baseName = "bench.0.";
                baseName += char(chipletNumber + '0');
                file.open(baseName.c_str(), std::ios::out);
                // look for the record whose srcCore is j
                std::vector<CommunicationRecord*> tmpRecordSet;
                for (size_t i = 0; i < ::TotalChipletNumber; i++) {
                        for (size_t k = 0; k < ChipletSet[i].relatedRecord.size(); k++) {
                                CommunicationRecord* tmp = ChipletSet[i].relatedRecord[k];
                                if (tmp->srcCore == chipletNumber) {
                                        tmpRecordSet.push_back(tmp);
                                }
                        }
                }

                // sort the record
                CommunicationRecord* record2sort = new CommunicationRecord[tmpRecordSet.size()]();
                int* sequency = new int[tmpRecordSet.size()]();
                unsigned long long* cycle = new unsigned long long[tmpRecordSet.size()];
                for (size_t j = 0; j < tmpRecordSet.size(); j++) {
                        record2sort[j] = *tmpRecordSet[j];
                        sequency[j] = j;
                        cycle[j] = tmpRecordSet[j]->cycle;
                }
                sortTheChipletRecord(cycle, sequency, tmpRecordSet.size());
                for (int i = 0; i < tmpRecordSet.size(); i++) {
                        record2sort[i] = *tmpRecordSet[sequency[i]];
```

```
            }

            // write to the sender file
            for (int i = 0; i < tmpRecordSet.size(); i++) {
                unsigned long long cycle = unsigned long long(((record2sort[i]).cycle -
chipletNumberZeroStartCycle) / 1000);
                int targetCore = record2sort[i].targetCore;
                file << cycle << " 0 " << j << " 0 " << targetCore
                    << " 5" << std::endl;
            }
            file.close();
        }

}
// 输出bench文件
void outputTheFile(CommunicationRecord* Record, int length) {
        std::ofstream file;
        file.open("bench", std::ios::out);
        for (size_t i = 0; i < length; i++){
            unsigned long long cycle = unsigned long long((Record[i].cycle -
chipletNumberZeroStartCycle)/1000);
            int srcCore = Record[i].srcCore;
            int targetCore = Record[i].targetCore;
            file << cycle << " 0 " << srcCore << " 0 " << targetCore
                << " 5" << std::endl;

        }
        file.close();


}

// 用于生成全部trace文件
void generatePopnetTraceFile(Chiplet* chipletSet) {
        int recordSize = 0;
        for (size_t i = 0; i < ::TotalChipletNumber; i++){
            recordSize += chipletSet[i].relatedRecord.size();
        }
        //bool* recordComplete = new bool[::TotalChipletNumber]();
        int* ptr2ChipletRecord = new int[::TotalChipletNumber]();
        CommunicationRecord* TotalRecord = new CommunicationRecord[recordSize]();


        for (size_t i = 0; i < recordSize; i++) {
```

```
            bool init = false;
            CommunicationRecord tmpRecord = CommunicationRecord(0, 0, 0);
            int targetChipletNumber = -1;
            for (size_t j = 0; j < ::TotalChipletNumber; j++) {
                if (ptr2ChipletRecord[j] >= chipletSet[j].relatedRecord.size()) {
                    continue;
                }
                if (!init) {
                    tmpRecord = *chipletSet[j].relatedRecord[ptr2ChipletRecord[j]];
                    init = true;
                    targetChipletNumber = j;
                }

                // compare the time and decide the min
                if ((chipletSet[j].relatedRecord[ptr2ChipletRecord[j]])->cycle < tmpRecord.cycle) {
                    tmpRecord = *chipletSet[j].relatedRecord[ptr2ChipletRecord[j]];
                    targetChipletNumber = j;
                }

            }
            TotalRecord[i] = tmpRecord;
            ptr2ChipletRecord[targetChipletNumber]++;

        }
        outputTheFile(TotalRecord, recordSize);


        outputSenderFile(&chipletSet[0]);


}


void swap(int* first, int* second) {
        int* tmp = first;
        second = tmp;
        first = second;

}
void swap(unsigned long long* first, unsigned long long* second) {
        unsigned long long* tmp = first;
        second = tmp;
        first = second;
```

```
}


void sortTheChipletRecord(unsigned long long* cycle, int* sequency, int length) {
	//buble sort because not requring high performance
	for (int i = 0; i < length - 1; i++)
	{
		for (int j = 0; j < length - 1; j++)
		{

			if (cycle[j] > cycle[j + 1]) {
				swap(&cycle[j], &cycle[j + 1]);
				swap(&sequency[j], &sequency[j + 1]);
			}
		}
	}

}



// 排序时间，防止出现cycle无法对齐的情况
void sortChipletTime(Chiplet* chipletSet) {
	for (size_t i = 0; i < ::TotalChipletNumber; i++){
		// for each chiplet
		Chiplet* currentChiplet = &chipletSet[i];
		int size = currentChiplet->relatedRecord.size();
		CommunicationRecord* tmp = new CommunicationRecord[size];
		int* sequency = new int[size]();
		unsigned long long* cycle = new unsigned long long[size];
		for (size_t j = 0; j < size; j++){
			tmp[j] = *currentChiplet->relatedRecord[j];
			sequency[j] = j;
			cycle[j] = currentChiplet->relatedRecord[j]->cycle;

		}
		sortTheChipletRecord(cycle, sequency, (chipletSet[i]).relatedRecord.size());

		currentChiplet->relatedRecord.clear();

		for (int i = 0; i < size; i++) {
			currentChiplet->relatedRecord.push_back(&tmp[sequency[i]]);
		}
```

```
        }
    }

    // 查找chiplet的开始时间，因为gem5启动操作系统一般需要几十万cycle的开销。
    void initChipletStartPoint(Chiplet* chiplet, bool isMainChiplet = false, unsigned long long
    mainChipletCycle = 0) {
        // here is some stuff that something must be fixed in the future
        chiplet->setTime(chiplet->relatedRecord[0]->cycle);
        if (!isMainChiplet) {
            unsigned long long base = chiplet->relatedRecord[0]->cycle;
            chiplet->relatedRecord[0]->cycle = chiplet->relatedRecord[0]->cycle - base +
    mainChipletCycle + (rand()%20)*1000;
            for (size_t i = 1; i < chiplet->relatedRecord.size(); i++) {
                chiplet->relatedRecord[i]->cycle = chiplet->relatedRecord[i]->cycle - base +
    mainChipletCycle;
            }
        }else {

        }




    }


    void processOneFile(Chiplet* chipletSet, int currentChipletNumber) {
        std::ifstream myFile;
        std::string realFile = communicationBaseFileName + (char)(currentChipletNumber + '0');
        myFile.open(realFile.c_str(), std::ios::in);
        std::stringstream ss;
        std::string line = "";

        while (std::getline(myFile, line)) {
            ss.clear();
            ss.str(line);
            unsigned long long cycle;
            int srcCoreNumber;
            int targetCoreNumber;
            int data;// we don't need it but for skipping
            ss >> cycle >> srcCoreNumber >> targetCoreNumber >> data;

            if (targetCoreNumber == -1) {// the message is for all
```

```
                CommunicationRecord* tmp = new CommunicationRecord(cycle, srcCoreNumber,
currentChipletNumber);
                for (size_t i = 0; i < ::TotalChipletNumber; i++) {
                    chipletSet[i].relatedRecord.push_back(tmp);
                }

            }else {
                chipletSet[targetCoreNumber].relatedRecord.push_back(new
CommunicationRecord(cycle, srcCoreNumber, currentChipletNumber));

            }


        }

        myFile.close();


}


int main() {

        std::cout << "Enter the TotalChipletNumber" << std::endl;
        std::cin >> ::TotalChipletNumber;
        //bool* chipletInit = new bool[TotalChipletNumber]();
        Chiplet* myChipletSet = new Chiplet[::TotalChipletNumber]();
        for (int i = 0; i < TotalChipletNumber; i++) {
            myChipletSet[i].chipletNumber = i;
        }

        for (int i = 0; i < ::TotalChipletNumber; i++) {
            processOneFile(myChipletSet, i);
        }


        sortChipletTime(myChipletSet);


        initChipletStartPoint(&myChipletSet[0], true);
        for (size_t i = 1; i < ::TotalChipletNumber; i++) {
            initChipletStartPoint(&myChipletSet[i], false, myChipletSet[0].chipletStartCycle);

        }
```

```
        chipletNumberZeroStartCycle = myChipletSet[0].relatedRecord[0]->cycle;


        generatePopnetTraceFile(myChipletSet);


        return 0;
}
```