SW众核芯片模拟器

1 总体概况

SW26010存储层次图如图1所示。对于单个核组包括了一个主核(控制核心MPE : management processing unit)，和64个从核阵列(CPE : computer processing element)。MPE是一个负责处理程序的逻辑密集部分和系统资源的控制的通用处理器。64个CPE则是一些轻量级计算核心，被排布成8\*8阵列。主核有完整的多级cache，从核只有本地局存LDM，主核从核都可以访问外存DDR。对于每个从核LDM地址空间与DDR地址空间相互独立。



图1 目前SW26010存储层次图

对于一个核组的模拟，本文采用了一个单核的SimpleScalar来模拟主核作为一个独立的进程，将CMP配置为众核模拟从核作为另一个进程，两个进程并行模拟。

本文主要介绍了SimpleScalar、CMP安装使用过程，主从核之间及从核与从核之间数据批量传输的设计实现过程。

2 安装与使用

2.1 Simplescalar（主核）安装

安装包下载地址：https://gitee.com/xiaozhenli001/simplescalar\_install.git

1. 安装环境信息

操作系统：Ubuntu 14.10 / Ubuntu 18.04

环境变量配置：

编辑用户目录下的.bashrc文件，在该文件最后加上以下语句

1. export IDIR= “要安装的目录”
2. export HOST=i686-pc-linux
3. export TARGET=sslittle-na-sstrix
4. export PATH=$PATH:$IDIR/sslittle-nasstrix/bin

编辑完成后，执行指令“source .bashrc”使设置的环境变量生效。

1. 安装过程
2. 在安装目录下创建文件夹，下载安装包到指定文件夹下。
3. 执行 sudo apt-get install build-essential flex bison 命令来安装使用过程所需的软件包。
4. 解压Simpletools-2v0
   1. 执行以下指令：

cd $IDIR

tar xvfz simpletools-2v0.tgz

rm-rf gcc-2.6.3

执行完上述指令后，在$IDIR目录下会得到ssbig-na-sstrix和sslittle-na-sstrix文件夹，它们中都包含了一个include文件夹和一个lib文件夹。

1. 安装SimpleUtils-990811
   1. 执行以下指令：

cd $IDIR

tar xvfz simpleutils-990811.tar.gz

cd simpleutils-990811

* 1. 修改ld目录下的ldlex.l文件，将文件中的“yy\_current\_buffer”替换成“YY\_CURRENT\_BUFFER”。
  2. 进入simpleutils-990811目录，执行以下指令：

./confing --host=$HOST --target=$TARGET --with-gnu-as --with-gnu-ld --prefix=$IDIR

make

make install

1. 安装simplesim-3v03
   1. 执行以下命令：

cd $IDIR

tar xvfz simplesim-3v0e.tgz

cd simplesim-3.0

make config-pisa

make

1. 安装gcc交叉编译器
   1. 执行以下命令：

cd $IDIR

tar xvfz gcc-2.7.2.3.ss.tar.gz

cd gcc-2.7.2.3

./configure --host=$HOST --target=$TARGET --with-gnu-as --with-gnu-ld --prefix=$IDIR

* 1. 修改当前目录读写权限：“chmod -R +w .”
  2. 修改Makefile文件，在130行末添加 “-I/usr/include”
  3. 修改protoize.c文件，把60行处的<varargs.h>改为<stdarg.h>
  4. 修改obstack.h文件，在341行处，把“\*((void \*\*)\_\_o->next\_free)++”改为“\*((void \*\*)\_\_o->next\_free++)”
  5. 执行以下命令：

cp ./patched/sys/cdefs.h ../sslittle-na-sstrix/include/sys/cdefs.h

cp ../sslittle-na-sstrix/lib/libc.a ../lib/

cp ../sslittle-na-sstrix/lib/crt0.o ../lib/

* 1. 编译：make LANGUAGES=”c c++” CFLAGS=”-O” CC=”gcc”

此时编译会出现一个错误，修改insn-output.c文件，在675、750、823行末加“\”。

再次编译会出现类似“\*\*\* buffer overflow detected \*\*\*”的错误，此时将安装包install needed文件夹中的两个文件放到$IDIR/sslittle-na-sstrix/bin目录下，执行“chmod +x ar ranlib”指令修改它们的权限。

再次编译，若出现“recipe for target ‘libgcc1.null’ failed”的错误，执行“sudo apt-get install gcc-multilib”指令，安装成功后再次编译。

* 1. 在$IDIR/simplescalar/gcc-2.7.2.3目录下，执行“make enquire”指令。

若出现“badly punctuated parameter list in `#define`”的错误，修改“/usr/include/sys/sysmacros.h”文件第62行为“#define \_\_SYSMACROS\_DM1(ARGS…) \_\_glibc\_macro\_warning(##ARGS)”后再次执行“make enquire”命令。

若出现“undefined reference to `\_\_isoc99\_sscanf”的错误，在Makefile文件第995行“$(ENQUIRE\_CFLAGS)”后面加上“-D\_GNU\_SOURCE”,后重新编译。

若出现“/bin/sh:1:cc:not found”的错误，将Makefile第60行CC=cc修改为CC=gcc，然后重新编译。

* 1. 安装交叉编译器

执行“../simplesim-3.0/sim-safe ./enquire -f >! float.h-cross”指令。

* 1. 安装完成后替换simplesim-3.0目录下的sim-outorder.c文件。

2.2 CMP（从核）安装

安装代码下载地址：https://gitee.com/xiaozhenli001/cmp.git

1. 安装环境信息

操作系统：Ubuntu 14.10 / Ubuntu 18.04

环境变量配置：

编辑用户目录下的.bashrc文件，在该文件最后加上以下语句

1. Export PATH=“/cmp安装位置/opt/crosstool/gcc-3.4.3-glibc-2.3.5/alpha-unknown-linux-gnu/bin:$PATH”。
2. 安装过程
3. 在安装目录下创建文件夹，下载cmp代码到指定文件夹。
4. 进入“./cmp/cmp\_org\_huq\_all\_trace”执行make指令编译仿真器，编译完成后屏幕最后一行出现“my work is done here…”仿真器编译成功。

2.3 CMP（从核）使用

1. 进入“cmp/ cmp\_org\_huq\_all\_trace”目录，执行make指令编译仿真器，输出“my work is done here…”编译成功。
2. 进入“cmp/benchmark/matrix”目录下，执行make指令编译benchmark，编译完成后，在当前目录下执行“./matrix.sh”指令，等待程序执行。

2.4 SimpleScalar（主核）使用

1. 修改sim-outorder.c文件sim-main()函数最后打开文件的路径为当前cmp安装路径。
2. 进入“simplescalar/simpelsim-3.0”目录下，执行make指令编译SimpleScalar，编译完成后屏幕最后一行出现“my work is done here…”SimpleScalar编译成功。
3. 进入“simplescalar/benchmark”目录下，执行“$IDIR/bin/sslittle-na-sstrix-gcc -o matrix matrix.c”指令对程序进行交叉编译。
4. 编译无误后，执行“../../simplesim-3.0/sim-outorder matrix”执行，等待程序执行。

3 DMA、RMA读写内部原理

3.1 DMA读写设计

3.1.1 DMA 读

DMA 读是由从核发起的系统调用，其功能是从主存批量读取数据到从核本地局存 LDM。



图2 DMA读时序模型

DMA读时序模型如图2所示。DMA读是单从核模式，由单个从核发起DMA读的系统调用，此时从核线程挂起，生成读请求事件，然后将其注入网络，该读请求事件将通过popnet进行网络仿真到达目的路由器处（此处我们将网络拓扑映射后ID号为00的空结点作为目的路由节点用于时序模拟。下同），在目的路由器收到请求事件后，将源、目的节点交换位置，生成响应事件注入网络，经过popnet返回到与原核绑定的路由器上，此时从核线程恢复。到此，该RMA读的时序模型结束。



图3 DMA读时序模型函数调用流程图



图4 DMA读时序模型响应流程函数调用图

DMA读时序模型函数调用流程如图3所示。该系统调用在sys\_syscall()中，应用程序在调用MTA\_DMA\_READ(src,dst,len)时触发相应的系统调用，其中src参数表示发起读请求的从核源地址，dst参数表示要读取的主存数据的目的地址，len参数表示要读取的数据长度。此时，将freeze变量置1，running变量置0，表示将当前从核线程挂起。在源地址处生成一个网络数据包（其中包含了源、目的节点、数据包大小、读写参数等），通过scheduleThroughNetwork(event)函数将包注入网络，经过网络仿真后，在目的路由器处判断是否收到数据包，在确认收到数据包后，目的路由器交换数据包的源、目的地址生成响应事件注入网络，生成的网络数据包回传过程如图4所示。源路由器在收到目的路由器的响应数据后，整个网络传输过程模拟结束。此时，将freeze变量置0，running变量置1，表示当前从核线程恢复，DMA读时序结束。



图5 DMA读功能模型

DMA读功能模型如图5所示。DMA读功能模型是由从核发起的DMA读请求事件，发起DMA读请求事件后，将请求按照规定的结构写入mmap映射的共享内存区域，等待主核数据。在主核数据到达后，从核读取共享内存中的数据，将文件中的数据写到从核局存目的地址。

3.1.2 DMA写

DMA写是由单个从核发起的系统调用，其功能是将该从核局存LDM的数据批量写入主存。



图6 DMA写时序模型

DMA写时序模型如图6所示。DMA写是单从核模式，由单个从核发起DMA写系统调用，生成写请求事件，然后将其注入网络，该写请求事件包将通过popnet进行网络仿真到达目的路由节点，该DMA写时序模型结束。与DMA读不同的是，DMA写无需挂起线程，也无需目的路由器生成响应事件。



图7 DMA写时序模型函数调用

DMA写时序模型函数调用流程如图7所示。该系统调用在sys\_syscall()中，应用程序在调用MTA\_DMA\_WRITE(src,dst,len)时触发相应的系统调用，其中src参数表示发起读请求的从核源地址，dst参数表示要写入的主存地址，len参数表示要写入的数据长度。在源地址处生成一个网络数据包（其中包含了源、目的节点、数据包大小、读写参数等），通过scheduleThroughNetwork(event)函数将包注入网络。经过网络仿真模拟后，在目的路由器处判断是否收到数据包，在确认收到数据包后，整个网络传输过程模拟结束，DMA读时序结束。



图8 DMA写功能模型

DMA写功能模型如图8所示。DMA写功能模型由从核发起DMA写请求，发起写请求数据后，将请求事件放入事件队列中，然后将主存目的地址、数据、处理位等按照规定的格式写入mmap映射的共享内存区域，主核每隔一定的时间检查共享区域，发现共享区域发生变化后将数据根据目的地址写入相应的位置，从而DMA写过程完成。

3.2 RMA读写设计

3.2.1 RMA 读

RMA读是由从核发起的系统调用，其功能是将一个从核本地局存LDM中的数据读取到另外一个从核的本地局存LDM中。

RMA读时序模型与DMA读时序模型大致相同。二者所不同的是，RMA读将读请求事件注入网络经popnet网络仿真后，直接到达目的从核在网络映射后的节点。RMA读时序模型函数调用流程与DMA读时序模型函数调用流程一致，在此不再赘述。



图9 RMA读功能模型

RMA读功能模型如图9所示。RMA读功能模型是由单从核发起的RMA读请求，在发起RMA读请求后，将请求放入事件队列中，等待目的路由器，即将一个从核本地局存LDM中len个数据长度的数据写入目标从核的本地局存LDM中。



图10 RMA读功能模型函数调用流程图

RMA读功能模型函数调用流程如图10所示。该系统调用在sys\_syscall()中，应用程序在调用MTA\_RMA\_READ(src,dst,len,threadid)后触发相应的系统调用，在时序模型将发起的读请求事件注入网络后，通过调用dir\_eventq\_insert(event)将该事件插入到事件队列中，然后通过MEM\_READ(MEM,ADDR,TYPE)函数，读取到从ADDR开始的长度为len的TYPE类型数据，最后通过调用MEM\_WRITE(MEM,ADDR,TYPE,DATA)函数，将读取到的TYPE类型的DATA数据写入从ADDR开始的向后偏移len个数据长度的目的从核本地局存中。

3.2.2 RMA 写

RMA写是由单个从核发起的系统调用，其功能是将该从核局存LDM的数据批量写入主存。

RMA写时序模型及RMA写时序模型的函数调用流程与DMA写时序模型及DMA写时序模型的函数调用流程基本一致，故在此不在赘述。



图11 RMA写功能模型

RMA写功能模型如图11所示。RMA写功能模型由从核发起RMA写请求，在发起请求后，将请求插入事件队列中，最后将局存源地址开始的大小为len个数据长度的数据写到目的从核局存中，从而完成RMA的写功能。



图12 RMA写功能模型函数调用流程图

RMA写功能模型函数调用流程如图12所示。该系统调用同样在sys\_syscall()中，应用程序调用MTA\_RMA\_WRITE(src,dst,len,threadid)时触发相应的系统调用，在时序模型将写请求事件注入网络后，通过调用dir\_eventq\_insert(event)函数将事件插入事件队列中，通过调用MEM\_WRITE(MEM,ADDR,TYPE,DATA)函数将本地局存的TYPE类型的DATA数据写入以ADDR地址开始的目的从核局存中。至此，RMA写功能模型完成。

4 DMA、RMA读写具体实现

4.1 从核内部

从核内部主要是对上述RMA、DMA读写设计的代码实现。

1. 对syscall.c文件的修改

在syscall.c文件中，对RMA、DMA的系统调用进行了定义和实现。首先，选择一个系统未使用过的系统调用号对系统所要实现的系统调用进行定义。然后在sys\_syscall()函数中对所定义的系统调用进行具体实现。RMA读系统调用的实现如图13所示。对于RMA、DMA读而言，系统调用首先要生成一个事件，然后从传值寄存器中获取到相应的参数后，对事件的各种参数进行初始化设置并将当前线程挂起等待读取的数据传回，同时使用quiesceStruct结构体记录使当前线程挂起的请求源地址和请求线程ID。接着将事件通过scheduleThroughNetwork()函数注入网络进行网络仿真，最后通过dir\_eventq\_insert()函数将事件插入事件队列。

对于DMA、RMA写而言，与DMA、RMA读所不同的是，DMA、RMA写无需将线程挂起，将事件注入网络并插入事件队列即可。



图13 RMA读系统调用的实现

1. 对cache.c文件的修改

对cache.c文件的修改包含了对cache\_access()函数和dir\_operation()函数的修改。

dir\_operation()函数用于线程在完成网络仿真后，对插入事件队列中的事件根据其operation参数值进行相应的操作。对于RMA、DMA读而言，由于读请求要保证数据从目的路由器返回到源节点才能进行后续的读数据操作，所以首先要进行的处理是交换源、目的地址，修改operation值为MTA\_READ\_REPLY，其余参数保持不变，而后同样进行注入网络、插入事件队列的操作。在MTA\_READ\_REPLY中，表示源节点已成功收到目的节点所发出的响应数据，此时，我们只需根据源、目的地址以及读取的数据长度读出对应的数据并写入从核局存。相比于DMA读而言，RMA读过程更加简单，仅需通过MEM\_READ()读取目的局存数据，而后通过MEM\_WRITE()将读取到的数据写到源从核局存中去即可。对于DMA、RMA写而言，在目的节点收到写请求数据包后无需生成响应包，故我们仅需根据源地址、数据长度读取从核局存中的数据，而后写入目的地址。所不同的是，RMA写是将数据写入目的从核局存中，而DMA写则是将数据按规定格式写入共享内存中，等待主核读取。

4.2 主核内部

由于DMA读、写都是单从核模式，都是由从核发起的，所以主核内部仅需在固定的周期内对从核发起读写请求文件进行响应即可。即对主核sim-outorder.c文件进行修改即可。在sim-outorder.c文件中sim\_main()函数是程序的入口函数，每执行一次sim\_cycle会自增一次，每隔固定的sim\_cycle对共享文件进行一次读操作。对于从核发起的写请求，在判断请求未被处理后，主核读取数据和目的地址，通过MEM\_WRITE()将数据写入。对于从核发起的读请求，在判断请求未被处理后，主核通过MEM\_READ()读取到数据后，将数据及源、目的地址等写入到文件中去。

5 Benchmark的编写

本benchmark主要功能是使用上述实现的DMA、RMA读写系统调用实现两个矩阵之间的相乘。对于从核程序，首先我们通过系统调用fork出4个线程代表四个从核，定义四个数组分别表示四个从核的地址空间。0号从核程序如图15所示。对于0号从核，首先通过DMA\_READ()将主核上的矩阵第一列读入0号从核的地址空间，接着将读入的数据与从核上的初始矩阵进行运算。运算完成后判断四个核是否都已完成对应任务，如果完成，则通过DMA\_WRITE()将数据写入主核，通知主核从核已完成相应的操作。最后通过DMA\_WRITE()将运算结果写回主核。对于其余三个从核，与0号从核不同的是需要通过RMA\_READ()将0号从核的矩阵数据读入各自的地址空间，其余操作流程与0号从核一致。对于主核程序，仅需要通过简单的循环给矩阵进行初始化即可。



图15 从核矩阵乘benchmark