sv6 for RV64 SMP - 课程设计最终报告

计53 谭闻德 尹宇峰

选题概述

近年来，人工智能及大数据的众多应用为计算能力提出了新的需求，而处理器核心频率的提高已经遇到困难，现如今，多核心处理器、并行化已成为提高计算系统性能的主要手段。事实上，在2013年，Austin T. Clements等人[1, 2]就提出过一种新的手段，用于优化并行系统软件的性能。此项工作提出了一个名为Commuter的工具，借助符号化执行技术[10, 11]以及SMT求解器[12]，用于挖掘系统规范中潜在的并行化优化点；同时，他们用此工具辅助设计实现了一个并行度很好的精简的操作系统，名为sv6。sv6在x86-64指令集架构上实现，考虑到目前新兴的RISC-V指令集架构[3, 4]更开放、灵活性更好、硬件设计更简洁，为此，本课程设计基于上述工作，将上述工作提出的sv6移植到RISC-V 64位多核平台，以期能够推动RISC-V并行架构的发展。同时，本课程设计过程深入理解Commuter的设计与实现，并对其论文中描述含糊的地方做了更详细的分析与阐述，最后对全文做了总结。此外，本课程设计还基于Commuter对POSIX socket API的UDP（协议）部分进行了建模，未来还可以用此模型对实际操作系统的网络子模块进行并行性测试与优化。

由此，本课程设计的主体工作分成以下三个部分：

1. **移植** 将sv6操作系统从x86-64移植到RISC-V 64位（以下简称RV64）对称多处理器（以下简称SMP）环境，包括模拟器环境以及真实硬件环境。
2. **分析** 详细阅读文献[2]及其代码，分析和理解Commuter工具的设计与实现，并对其论文中描述含糊的地方做了更详细的分析与阐述，最后对全文做了总结，方便后人阅读理解。
3. **建模** 利用Commuter工具对POSIX socket API的UDP（协议）部分进行建模（从传输层到应用层）。

相关工作

这一小节简要介绍本课程设计会用到的前人的一些已有工作。

RISC-V

RISC-V [3, 4]是一个全新的指令集架构，它更开放、灵活性更好、硬件设计更简洁。截止2018年05月25日，RISC-V指令集规范的用户态指令集规范已发布2.2版，特权指令集规范草稿已发布1.10版。

sv6

sv6操作系统[1]是一款基于xv6，并行度很好的精简的类POSIX的研究性质的操作系统。此外，sv6使用C++开发。

经过我们的调研，没有人曾尝试过将sv6移植到RV64 SMP环境，这也说明了我们的工作具有开创性。

符号化执行与Z3求解器

通过符号化执行的方法[10, 11]，可以将一个函数的输入参数指定为符号值，函数的每一个路径可以产生一个路径条件逻辑表达式，通过Z3等SMT求解器[12]进行求解，就可以知道这个路径是否可达，是否会产生特定的情况（例如，某些情况是断言失败）。同时还可以给出函数返回值的表达式。

Commuter

Austin T. Clements等人在2013年提出的Commuter [2]能够利用上面提到的符号化执行技术以及Z3求解器，完成对系统调用接口规范模型的分析，为设计良好并行化的接口给出建设性意见，此外Commuter还能生成测试代码来检测实际操作系统的系统调用并行性，详细的论述分析请见下一小节。

我们的工作

设计方案

本课程设计分为以下9个步骤分阶段进行：

1. 准备开发环境 — 第6周

把RV64 SMP的工具链，包括编译器GCC、模拟器CPU等环境准备好。

这是基础性的工作，为进行RV64的开发，必不可少。

2. 学习RV64 — 第6周

学习RV64用户态指令集架构和特权指令集架构。

这也是基础性的工作，为进行RV64的开发，必不可少。当然，若之后遇到问题，可以进一步查阅文档。

3. 动手实践RV64 SMP — 第6周

基于bbl编写几个RV64 SMP的小例子，注重SMP的启动以及管理。这个环节旨在熟悉RV64 SMP的一些关键性问题，例如谁负责启动AP（除了启动处理器之外的处理器核心称为辅助处理器，即AP），启动AP后的状态如何设置等。

4. 开始移植 — 第6周至第11周

开始将sv6移植到RV64单核环境。

这个阶段的重点以及难点都在于将原有的x86-64架构相关的部分在RV64架构下重新实现，包括虚拟内存管理、中断和异常（包括系统调用）处理、原子操作指令、寄存器上下文、内核栈设置、外设中断配置、设备驱动程序等。

5. 多核移植 — 第11周

进一步将sv6移植到RV64多核系统。

事实上，我们在单核移植时已经充分考虑多核的情况，所以多核的移植过程十分简短。

6. 真实硬件测试 — 第12周

从辉总[15]处借来了RV开发板一套，含12V 2A DC5.5电源一个、装板子的瓦楞纸质包装盒一个、保护用粉色塑料垫一张、RV板一个。RV开发板HiFive Unleashed [13]上安装了Freedom U540 SoC [14]。在RV开发板上成功运行移植的sv6，功能测例除浮点运算、信号以及几个运行较慢的测例外，均分别能够在模拟器以及真实硬件上测试通过。

注：感谢辉总大力支持！

7. 学习并实践符号化执行方法、Z3求解器 — 第10周

学习符号化执行方法以及Z3求解器的使用。

Commuter工具使用到了符号化执行方法以及SMT求解器，为了更好地完成sv6的优化，更好地写出接口规范，需要学习这两项内容。

8. 深入分析和理解Commuter的设计与实现 — 第10周至第13周

详细阅读文献[2]及其代码，分析和理解Commuter工具的设计与实现，并对其论文中描述含糊的地方做了更详细的分析与阐述，最后对全文做了总结，方便后人阅读理解。

9. 基于Commuter对POSIX socket API建模 — 第13周

利用Commuter工具对POSIX socket API的UDP（协议）部分进行建模（从传输层到应用层）。

主要工作是接口规范的编写。

小组分工

1. 移植sv6到RV64 SMP环境

谭闻德主导，尹宇峰辅助。

具体而言：

- 谭闻德负责部分编码工作、调试并修复bug、在真实硬件上测试及调试
- 尹宇峰负责小部分编码工作、review谭闻德编写的代码、参与调试并修复bug、编写谭闻德不愿意写的代码、与其他小组进行联络

2. 深入分析和理解Commuter的设计与实现

尹宇峰主导，谭闻德辅助。

具体而言：

- 尹宇峰负责阅读Commuter相关文献、分析Commuter设计与实现的原理及思路、编写分析报告
- 谭闻德负责阅读Commuter相关文献、与尹宇峰讨论Commuter相关问题、修改分析报告

3. 基于Commuter对POSIX socket API建模

尹宇峰主导，谭闻德辅助。

具体而言：

- 尹宇峰负责研究POSIX socket API规范、参与编写部分模型行为级描述文件
- 谭闻德负责研究POSIX socket API规范、参与编写部分模型行为级描述文件

具体细节

sv6移植

代码请见GitHub仓库：https://github.com/twd2/sv6 。

具体而言，sv6的移植工作可以分为如下几个阶段。每个阶段都有对应的陷阱，这会在本节逐一记录。注意，本文为总结性质的文档，所以如下几个阶段不一定严格按照物理时间顺序进行。

编译

为了开展移植工作，我们决定先将原始的x86-64架构下的sv6运行起来。由于sv6是2014年写的，而现在是2018年，所以用现在的编译器编译sv6会出现如下一些小问题，需要修复。

1. 编译器选项的`-std=c++0x`需要改为`-std=c++11`
2. 关闭`-Werror`选项，因为warning不易修复
3. 注释掉用到了`has_trivial_default_constructor`的代码段，因为此类模板已被废弃
4. sv6是C++语言编写的，而C++运行时代码用到了fs段寄存器，但是操作系统内核中对此寄存器没有很好地配置，因此需要去除对于fs段寄存器的使用。我们的方法是对编译器生成的二进制代码，将其中使用到fs段寄存器的指令替换为空操作。这样不会有运行错误是因为C++运行时代码仅仅使用fs段寄存器进行栈完整性检查，相应指令替换为空操作后，相当于跳过了检查而已。
5. `__thread`关键字在新标准下已被废弃，需要改为`thread_local`

修复后，原始的x86-64架构下的sv6可以正常运行。

接着，我们需要在RV64架构下编译sv6。容易知道，由于sv6是操作系统，因此它含有相当一部分和x86-64架构有关的汇编代码以及驱动程序，它们无法直接通过编译。为此，我们采用了如下方法来尽快使得代码能够编译，以便于我们能够尽快开始动态调试：

1. 将sv6中x86-64汇编代码部分注释并替换为死循环，以便于后续将对应的功能在RV64汇编下重新实现（事后反思，我们认为将这些代码替换为`panic`能够使得移植更加容易，因为死循环的定位略微麻烦）
2. 删去大量与x86-64分段机制相关的代码
3. 删去巨量与x86-64架构相关的驱动程序，包括ACPI、AHCI、APIC等。
4. 将其他无法编译的代码变为注释

至此，sv6可以在RV64编译环境下成功编译了。

链接生成可启动代码

在实际开始工作前，我们并没有意识到链接的过程会有问题。事实上，还是由于sv6是C++编写的，我们在链接的时候遇到了缺少库的问题，加上相应的`libgcc.a`库即可解决。

另外一个值得一提的问题与Code Model有关，RV64的Code Model分为两种：medlow以及medany。

选择medlow，会让编译器生成lui+12位立即数作为偏移的指令来进行寻址。其中，lui加载20位立即数到指定寄存器的高20位。此模型总计可以寻址的范围为32位，为0~4G。因此，程序只能放在低4G的空间内。

选择medany，则会让编译器生成auipc+12位立即数作为偏移的指令来进行寻址。其中，auipc指令也含有20位立即数，它左移12位再和当前PC相加，存入指定寄存器。这个模式可以寻址的范围同样为32位，但是这是相对于PC而言的，为PC-2G~PC+2G。只要偏移量的绝对值不超过2G，则程序放在内存的任何位置都可以正常寻址。例如，在我们的sv6移植中，内核代码放在高地址区域，为0xffffffc000000000，此时操作系统的代码、用到的库等所有程序都必须要用medany编译。否则，由于lui+12位立即数作为偏移的指令无法表示大地址，在链接的时候会出现"relocation truncated to fit: R_RISCV_HI20 against symbol"等错误，这也是我们遇到的问题。

我们起初编译的运行时库使用了默认的medlow参数，在与内核链接时，由于内核代码放在高地址区域，链接失败，重新用medany编译运行时库就解决了此问题。

此外，我们还遇到一个与链接关系不大，但是产生了链接错误的问题，与原子操作有关。具体而言，我们在编译sv6内核时，遇到了链接错误"对'`__atomic_compare_exchange_1`'未定义的引用"。

经过研究，这是因为RV64的A扩展指令集没有对8位整数或16位整数的原子操作的支持，从而导致GCC不支持`__atomic_compare_exchange_1`或`__atomic_compare_exchange_2`。

其中，`_1`或`_2`指的是操作数的长度是1字节或2字节，由于RV64支持32位整数或64位整数的原子操作，`__atomic_compare_exchange_4`或`__atomic_compare_exchange_8`有定义，也就不会产生链接错误。

这个问题的根本原因是代码中使用了`std::atomic<char>`或`std::atomic<short>`等类，间接用到了8位整数或16位整数的原子操作。

经过仔细分析，确定修改数据长度不会造成副作用后，我们将`std::atomic<char>`或`std::atomic<short>`等类修改为`std::atomic<int>`解决了此问题。

在单核环境运行

可以启动后，要将之前被注释、替换为死循环或者能够正常编译但对于RV64不适用的代码段对应的功能逐一在RV64汇编下重新实现，主要包括如下几点：

1. 页表的填写：RV64的页表格式与x86-64有细微区别，做相应修改即可。需要注意的陷阱是，RV64的页表项并不像x86-64是页面的物理地址，而是物理地址右移了2位。
2. `trapframe`以及`context`结构体：容易知道，RV64的寄存器及其调用约定与x86-64截然不同，因此需要修改这两个结构体中的字段，以及所有用到这两个结构体的代码。
3. 中断和系统调用的处理
4. 特权指令：RV64与x86-64的特权指令集差异很大，因此基本上需要重新实现所有和特权指令有关的代码。
5. 其余用汇编编写的代码：这部分包括内核线程的入口代码、用户进程的入口代码等。

此外，我们选择Berkeley Boot Loader（bbl）作为内核的boot loader，它提供一些SBI供操作系统内核使用，例如串口输入输出、IPI（处理器间调用）以及TLB shootdown。

在多核环境运行

多核与单核的不同在于处理器核心之间需要进行交互。具体而言，有系统启动时的同步互斥、IPI以及TLB shootdown等问题需要解决。由于bbl提供了很好的IPI以及TLB shootdown接口，这部分实现起来不是很困难。

此外，对于每一个处理器核心，需要保存自己的数据结构，考虑到sv6本身是多核操作系统，并且我们在单核移植的时候已经考虑到这一点，开启多核就不需要特殊处理了。

在真实硬件运行

真实硬件和模拟器的区别主要在于如下两方面，不能够很好的处理它们的话，就会产生bug。

第一，真实硬件启动时内存的内容是随机的，而模拟器一般会将内存空间全部清零。为此，需要在内核启动时清零bss段等内存区域。

第二，真实硬件有缓存以及TLB，模拟器一般不会特意模拟这些特性。为此，内核修改共享内存或者页表后，需要使用相应指令刷新缓存或TLB。

此外，对于本课程设计所使用的硬件环境而言，还存在如下两个问题：

1. RV开发板的第0个hart是被屏蔽的，所以真正可用的hart从编号1开始，需要特殊处理。
2. 根据规范，RV的实现可以不支持由硬件自动设置页表项的A位和D位，移植到模拟器的时候我们注意到了这一点，但是由于QEMU支持自动设置，我们也就没有特别处理。而RV开发板的实现是不支持的，需要软件设置，为简化实现，考虑到sv6没有交换分区，我们在创建页表项的时候直接将A位和D位置1。

其他

我们在移植过程中顺便修复了sv6本身的一个bug。具体而言，是buddy system中链表实现的bug，此bug能够导致在某种情况下，元素不能被正确插入链表，从而导致某些物理页面块不能被buddy system正确识别，表现与内存泄漏十分相似。

Commuter分析与理解

注：本小节基于文献[2]及其代码进行论述，由于时间及篇幅限制，我们略去了其中SIM条件证明等部分，但是这不会影响我们对于Commuter工具的整体理解和分析。

Commuter简述

本小节根据[2]中的文献Abstract、1. Introduction以及5. Analyzing interfaces using COMMUTER章节整理得到。

在论文中，作者指出当今的操作系统判断并行性的方式较为简单：在实际操作系统不同的核上运行测试代码来检测并行性的瓶颈。这样的方法有着明显的弊端：1. 我们不清楚最重要的并行性的瓶颈在哪里；2. 在操作系统开发阶段的后期，设计层面的修改（例如修改系统调用的接口）显得不切实际。为此作者提出了一个问题：我们是否可以仅仅通过系统调用的行为描述（specification）来判断系统调用与系统调用之间的并行性?

接着，作者提出了一条规则：如果对于一组系统调用，他们之间的任意调用顺序不会影响最后的系统状态和返回结果（即SIM条件），那么这些系统调用便可以被并行执行。这一规则的提出使得上述问题得到了解决（作者使用了名为Analyzer的工具来判断并行性）。

Commuter的提出可以帮助我们分析和设计系统调用的行为描述，并设计相应的实现到达并行效果。例如说，在同一个文件目录下，多个进程调用open系统调用，在类POSIX文件系统中这是无法并行的，这是因为open系统调用需要返回最小的文件描述符，但是当我们把行为描述修改成任意一个合法的文件描述符，他们就可以实现并行了。

更进一步地，针对在行为描述层面证明可以被并行的系统调用序列，作者利用TestGen工具生成了相应的测试代码，并在实际操作系统（sv6和Linux）中利用MTrace工具运行测试代码，以此来检测系统调是否因为不必要的存储共享降低了OS的并行性，同时帮助定位了OS并行性的瓶颈。最后，针对测试代码的结果，作者给出了几个可以帮助提升OS并行性的例子。

总的来说，Commuter的核心设计分成了三个部分Analyzer、TestGen、MTrace，其中Analyzer负责分析系统调用序列是否满足SIM条件，针对系统调用序列中满足条件的path condition，传递给TestGen；TestGen针对这些path condition实例化测试代码`TestGen.c`；MTrace运行在CPU的修改版本，额外记录了操作系统访问物理内存的信息，MTrace通过分析是否有缓冲行的访问冲突来判断实际OS的系统调用之间的并行性。原文中也提到充分理解Commuter是一件极其困难的事情，因为它涉及了许多的知识，例如符号化执行，z3求解。此外，更加令人头疼的是，限于篇幅，许多概念和细节，作者并没有真正阐释清楚，例如5.2 TESTGEN章节关于assignments同构这一概念和对于conflict coverage代码的实现。以下，我们将从Commuter的这三个部分分别展开详细论述他们的设计与实现，方便后人继续深入研究Commuter工具。

Analyzer

本小节根据[2]中的文献5. Analyzing interfaces using COMMUTER和5.1 ANALYZER章节整理得到。

总体来说，Analyzer的功能是输入对符号化的接口建立的模型（代码实现在spec.py中），然后针对该模型计算出准确的运行条件，使得系统调用之间可以并行。

注：为了简化问题，作者在考虑系统调用序列并行化问题时候，将序列中调用的个数规定为2个。

接下来，我们逐步讲解：

首先是对符号化的接口进行建模，这是Analyzer的输入。我们需要定义每个系统调用的python model（即行为描述或specification）。例如论文中的rename model，有两个参数，一个是原先的名字，一个是希望重新命名的名字。通过输入参数的不同，rename会有相应不同的操作，当两个参数相同时，不需要任何操作，若不然，则需要进行若干操作。

接着，我们会通过符号化执行的方法来分析系统调用序列是否满足SIM条件，即序列以任意先后顺序调用，最后的系统状态和返回结果都是一致的，而且满足单调特性（系统调用的子序列同样满足SIM条件）。判断是否满足条件的方式也比较暴力，即枚举全部调用序列，然后观察系统状态和返回结果是否一致。

符号化执行的方法，会让我们可以遍历系统调用内部的所有if分支路径（我们称之为path），其中有一些路径可以做到并行，而有一些路径则不行，于是通过Analyzer的分析后，我们会得到一组可以并行化的路径，而在此基础之上，便可以得到运行路径需要path condition，我们将这组满足SIM条件的path condition称之为Commutativity conditions。

再次以rename系统调用为例，当我们输入给Analyzer rename的行为描述后，Analyzer会说renane(a, b)和rename(c, d)可以并行化，如果参数满足以下任意一个条件：

1. 文件a和c存在，而且a，b，c，d两两不相同。
2. 文件a存在，b≠c，文件c不存在；或者文件c存在，d≠a，文件a不存在。
3. 文件a和c都不存在。
4. a=b，c=d。
5. 文件a存在，a=b，a≠c；或者文件c存在，c=d，a≠c。
6. a，c是两个相同的索引节点，而且a≠c，b=d。

接下来，我们讨论一下Analyzer的核心代码实现：

simtest.py的test函数，这个函数就是用来检测SIM条件的。它采用了对系统调用序列深度优先搜索方法（Depth-first search）来检测，对于第一次抵达树根时，记录下系统状态和返回值，此后每次抵达树根就会进行比较，如果不一样就会直接退出函数。值得注意的是，在simtest.py中调用test的方式是符号化执行的，所以说不会出现说特定参数恰好返回值和系统状态一致的情况，符号化变量的结果

一致，我们就可以安心地认为他们满足系统调用。

simtest.py的test_callset函数，这个函数利用符号化执行的方式调用了test函数，对于每个path都验证了sim条件，并会将所有满足sim条件的path condition取或打印出来（即Commutativity conditions），所有不满足sim条件的path condition也会取或打印出来。

TestGen

本小节根据[2]中的文献5. Analyzing interfaces using COMMUTER和5.2 TESTGEN章节整理得到。

作为Commuter的第二阶段，TestGen是其中概念最难理解，代码阅读最为困难的部分。主要难点在于：

1. 限于篇幅，许多概念和细节，作者并没有真正阐释清楚，例如关于assignments同构这一概念。为此，我们还阅读了作者的博士毕业论文，其关于assignment同构的描述和[2]中的论文是比较一致的，同样含糊不清。
2. 鉴于对论文的不理解，我们希望可以通过阅读代码的方式，解答困惑。但是结果令人失望，杂乱的代码注释，复杂的函数调用关系，以及python本身欠缺的类型声明的语法特性给我们的分析工作带来了空前的挑战。
3. 没有其他相关的文献参考，仅有的文献便是[2]和作者的博士毕业论文。
4. 我们是唯一一组对Commuter工具进行分析和理解的小组，没有其他小组可以给予帮助，而询问了杨兴杲学长，同样问题没有得到解答。

不过，在长期的不懈努力下，我们还是成功克服了上述困难，并且对这一部分的工作有了十分彻底的理解和分析，这样的结果让人十分振奋！

我们还是从头说起，TestGen的任务是记录下来Analyzer输出的Commutativity conditions，并将其中的符号化变量实例化，然后生成实际的测试代码，理论上这些测试代码都是可以被并行执行的。（因为这些代码都是由Analyzer分析认为满足SIM条件的）

也正因为如此，TestGen生成测试代码的目的就是寻找的潜在的系统调用实现的并行化问题。但是，我们仍然不会知道究竟什么样的输入参数会触发访存冲突。因此，我们需要构造一种合适的覆盖模式，使得我们的测试代码可以尽可能地找出可能的并行化问题。

论文中提到的覆盖模式有两种：

1. path coverage：针对每个path condition，通过z3求解器求得满足条件的输入参数，使得测试路径可以遍历每一条路径。
2. conflict coverage：上述的path coverage是一个path condition对应一组参数赋值，而conflict coverage是一个path condition对应多组参数赋值。为了解释这样做的目的，我们举一个例子：对于两个pwrite系统调用，他们对内存进行写操作的offset相同或者不相同，code path是一致的，但是他们的访问模式却是不同的，offset相同时，会造成访问冲突；offset不同时，则可以并行化。换一句话来说，相同的系统调用序列，相同的path路径，有可能出现一个没有访问冲突，一个出现访问冲突。因此，简单的路径覆盖并不能察觉上述例子的问题。

注：起初，我们将conflict coverage直接翻译为了冲突覆盖，但是此处的冲突和我们普遍意义上认为的存储访问冲突概念是不一样的，所以为了避免混淆，我们直接引用了英语原文，我个人的理解是这个名字取得不太妥当，更改为访问模式覆盖会更好。

关于assignment同构的概念，我们通过一个例子来阐述：

表达式	x=1, y=1	x=2, y=2	x=1, y=3
x - y	0	0	-2
x + y	2	4	4
2x - y	1	2	-1
2y - x	1	2	5

我们可以看出来，四个表达式根据三组赋值分别得到的结果进行值划分，前两组的划分结果是一致的，而第三组与前两组的划分结果不一样。**那么我们说：前两组assignment是同构的。**

更为严格的定义如下：

给定一组表达式集合：$S = \{E_1, E_2, \ldots, E_n\}$

其中每个表达式的输入参数并集为：$T = \{x_1, x_2, \ldots, x_m\}$

对于两组实际的参数取值$T_1 = \{a_1, a_2, \ldots, a_m\}$和$T_2 = \{b_1, b_2, \ldots, b_m\}$，我们称他们同构当且仅当：

$\forall i \neq j \in \{1, 2, \ldots, n\}$

$$E_i(a_1, a_2, \ldots, a_m) = E_j(a_1, a_2, \ldots, a_m) \iff E_i(b_1, b_2, \ldots, b_m) = E_j(b_1, b_2, \ldots, b_m)$$

我们先只叙述TestGen的实现方法，而方法的正确性在后续给出。为了实现conflict coverage，我们需要针对一条path取多组参数赋值，以遍历所有的访问模式，为此，我们需要这些参数赋值两两不同构。

具体而言，针对一组参数赋值和给定的表达式集合，我们可以求解与这组参数赋值同构的条件，然后将该条件取非再与path condition取并，更新为新的path condition，然后求解得到一组新的参数赋值，那么这组的赋值就一定与原先的参数赋值不同构。我们反复迭代这个过程，直到我们得到一定数量的参数赋值（一条code path对应的参数赋值组数有最大限制）或是说path condition已经无解的时候。

那么，我们来重新梳理一下TestGen的流程逻辑：

1. 简单的path coverage有时候无法发现潜在的并行化问题，所以我们需要conflict coverage遍历所有的访问模式。
2. 为了实现conflict，我们引入了assignment同构的概念，然后确定了一组表达式的集合（如何产生这些表达式将在后续代码讨论中讲解）。
3. 基于上述讨论，我们通过一组参数赋值可以求解与它同构的条件，然后将这个条件取非，再与path condition取并更新为新的path condition，以便得到一组与原参数赋值不同构的新参数赋值。
4. 我们反复更新path condition，这样可以得到一系列参数赋值，使得他们两两不同构。
5. 然后我们就可以通过这些实例化的参数赋值生成实际的c测试代码，而且是conflict coverage级别的，对于潜在并行化问题有更强的针对性。

接下来，我们通过讨论TestGen的代码实现（testgen.py和spec.py）来回答仅剩的两个问题：

1. 为什么同构概念的引入就可以实现conflict coverage?

2. 同构概念中的表达式集合从哪里来的?

之前也说过，我们阅读TestGen的代码有极大的困难：杂乱的代码注释，复杂的函数调用关系，以及python本身欠缺的类型声明的语法特性。因此，我们决定通过动态输出变量信息的方法快速理解TestGen的代码实现，而且顺便修复了其中的两个小bug。（显然作者并没有针对verbose输出模式的commuter代码进行检查）

我们先将Assignments的变量信息展示出来：

```
Assignments 1:
defaultdict(<type 'list'>,
  {
  <testgen.Interpreter object at 0x7f71fb8b4810>:
  [(Fs.proc0.fd_map._map.inum[a.close.fd], SInum!val!0),
  (Fs.proc0.fd_map._map.inum[b.close.fd], SInum!val!0)],

  None:
  [(a.close.pid, False), (b.close.pid, False),
  (Fs.proc0.fd_map._valid[a.close.fd], True),
  (Fs.proc0.fd_map._valid[b.close.fd], True),
  (Fs.proc0.fd_map._map.ispipe[a.close.fd], False),
  (Fs.proc0.fd_map._map.ispipe[b.close.fd], False)],

  <testgen.Interpreter object at 0x7f71fb8b4e50>:
  [(a.close.fd, 0), (b.close.fd, 1)],

  <testgen.Interpreter object at 0x7f71fb8b4890>:
  [(Fs.proc0.fd_map._map.off[a.close.fd], 2),
  (Fs.proc0.fd_map._map.off[b.close.fd], 2)]
  }
)

Assignments 2:
defaultdict(<type 'list'>,
  {
  <testgen.Interpreter object at 0x7f4e0c8e6ad0>:
  [(Fs.proc0.fd_map._map.pipeid[a.close.fd], SPipeId!val!0),
  (Fs.proc0.fd_map._map.pipeid[b.close.fd], SPipeId!val!0)],

  None:
  [(a.close.pid, False), (b.close.pid, False),
  (Fs.proc0.fd_map._valid[a.close.fd], True),
  (Fs.proc0.fd_map._valid[b.close.fd], True),
  (Fs.proc0.fd_map._map.ispipe[a.close.fd], True),
  (Fs.proc0.fd_map._map.pipewriter[a.close.fd], False),
  (Fs.proc0.fd_map._map.ispipe[b.close.fd], True),
  (Fs.proc0.fd_map._map.pipewriter[b.close.fd], False)],

  <testgen.Interpreter object at 0x7f4e0c8e6d50>:
  [(a.close.fd, 2), (b.close.fd, 3)],
```

```
  <testgen.Interpreter object at 0x7f4e0c8fc8d0>:
  [(Fs.proc0.fd_map._map.inum[a.close.fd], SInum!val!1),
  (Fs.proc0.fd_map._map.inum[b.close.fd], SInum!val!0)]
  }
)
```

我们可以看到每一个assignment都是一个字典，字典的key不是我们所关心的东西，而value是一个列表，列表的每一项就是我们的同构概念中的表达式和他们对应的取值。例如：

```
[(Fs.proc0.fd_map._map.inum[a.close.fd], SInum!val!1),
(Fs.proc0.fd_map._map.inum[b.close.fd], SInum!val!0)]
```

`Fs.proc0.fd_map._map.inum[a.close.fd]`和`Fs.proc0.fd_map._map.inum[b.close.fd]`是表达式，`SInum!val!1`和`SInum!val!0`是对应的表达式取值。

我们可以看到通过构造不同构的参数赋值，assignments 1中`Fs.proc0.fd_map._map.inum[a.close.fd]`和`Fs.proc0.fd_map._map.inum[b.close.fd]`的取值是相等的，而assignments 2中则是不同的。

具体而言，我们来解释一下表达式`Fs.proc0.fd_map._map.inum[a.close.fd]`的含义：`fd_map`是文件描述符号对文件描述符的映射，`inum`是文件描述符的inode号，整个表达式的含义是取文件描述符号对应的文件的inode号。我们容易知道，访问同一个inode会造成访问冲突，而不同的inode访问则可以并行化，所以在这里我们遍历了不同的访问模式，从而实现了conflict coverage，也就回答了我们的第一个问题。

然后是第二个问题，这个问题有些复杂，表达式产生的方式是多样的，我们只选其中一种进行论述，选取的表达式是`Fs.proc0.fd_map._map.inum[a.close.fd]`。而这个表达式的产生涉及到了Commuter中符号化执行引擎的实现，符号化执行引擎实现的字典中，key和value都是符号化变量表示的，所以判断一个给定的符号化变量是否在这个字典中，返回的也是一个符号化的表达式，而返回的对应某个key的value也是一个符号化的表达式。TestGen会自动记录下这些返回的表达式，例如`Fs.proc0.fd_map._map.inum[a.close.fd]`，作为后续遍历访问模式的重要的部分。

MTrace

本小节根据[2]中的文献5. Analyzing interfaces using COMMUTER和5.3 MTRACE章节整理得到。

这一部分的工作较为简单，MTrace的功能是运行TestGen的测试代码，MTrace会查看每个测试代码是否会有访问冲突。通过MTrace，开发者可以利用这些测试代码来寻找并修正代码中的并行性问题，亦或是作为测试，保证并行性的bug不会出现在系统调用的实现中。

测试代码分为两部分：

1. setup函数，负责初始化系统初始状态和系统调用的参数。
2. 若干个系统调用函数，将在不同的核上运行，这此处由于只观察系统调用两两组合的并行情况，所以有两个系统调用函数，MTrace会记录这两个函数所有对内存的读和写操作，然后根据这些记录分析并检测是否出现访问冲突，如果没有就说明这些系统调用可以并行，不然，则会记录下访问冲突的变量和对应调用他们的代码。

接下来，我们讨论一下MTrace的核心代码实现：

Mtrace的代码仓库在[16]中，核心代码是`mtrace.c`中的`mtrace_ld`、`mtrace_st`、`mtrace_cline_update_ld`和`mtrace_cline_update_st`四个函数。

其中`mtrace_cline_update_ld`和`mtrace_cline_update_st`两个函数会共同维护一组名为`block->cline_track`的位串，其中每一个缓冲行对应一个位串。如果`block->cline_track`对应的CPU位置的值为1，则说明该CPU目前拥有了对应缓冲行最新的数据。所以说，当我们进行读操作，发现对应CPU位置的位串值不为1的时候，则说明别的CPU对这个缓冲行可能进行了写操作（也可能是该缓冲行第一次被读入寄存器），可能出现访问冲突，需要记录这一次的访存信息；类似地，当我们进行写操作，我们会比较位串和`1 << CPU`的值，如果不相等，则说明别的CPU对这个缓冲行可能进行了读操作或是写操作（也可能是该缓冲行第一次被写入数据），可能出现访问冲突，需要记录这一次的访存信息。然后我们将位串更新为`1 << CPU`，以确认只有该CPU拥有缓冲行最新的数据。

通过上述分析，我们不难看出，MTrace追踪的访存冲突的粒度不是地址级别的，而是缓冲行级别的。所以说，可能出现虽然两个函数访问的内存地址不一样，但是因为他们在同一个缓冲行，而导致了访存冲突。

这里，我们发现了作者写论文的一个trick，在文中作者比较了sv6和Linux两个操作系统分别跑TestGen生成的测试代码结果，发现sv6明显好于Linux。但是，事实上，sv6中有的数据结构按照了缓存行大小进行了对齐，所以这样在多数情况下就会避免缓存行级别的缓存冲突了，所以说sv6潜在地就有利于这些测试。

Commuter小结

本小节根据[2]中的文献8. Conclusion章节和我们自己的理解体会整理得到。

在本篇论文中，作者首先提出并通过数学推导严格证明了一条规则：如果对于一组系统调用，他们之间的任意调用顺序不会影响最后的系统状态和返回结果（即SIM条件），那么这些系统调用便可以被并行执行。然后基于这一规则，使得我们仅仅通过系统调用的行为描述（specification）来判断系统调用与系统调用之间的并行性得到了可能。更进一步地，作者开发了一个名为Commuter的工具用于测试和分析实际OS中的并行性问题。Commuter分为三个部分：Analyzer，TestGen以及MTrace，Analyzer通过符号化执行分析行为描述级别系统调用的并行性，并给出可并行的Commutativity conditions；TestGen则通过Commutativity conditions生成实际的c测试代码；MTrace则在运行测试代码的时候记录访存冲突信息，检查测试代码是否可以并行。

我们使用Commuter工具有两个好处：

1. 利用Analyzer和系统调用的行为描述（specification）来判断系统调用与系统调用之间的并行性，在开发OS的前期阶段及时发现并修改系统调用潜在的并行性问题。
2. 利用TestGen生成测试代码，并利用MTrace记录测试代码的访存冲突信息，以此来检测实际操作系统中实现的系统调用是否出现了不必要的共享存储，而导致了并行性变差，并以此定位并行性的瓶颈，方便后续os的优化。

基于Commuter对POSIX socket API建模

代码请见GitHub仓库：https://github.com/twd2/commuter 。

我们只添加了一个文件：`/models/socket.py`

注：我们在这里实现的socket模型，比较简单，仅仅对socket在应用层和部分传输层的行为进行了建模。

考虑到tcp的协议状态机太过于复杂，我们选择了对无连接不可靠的udp进行建模，主要建模的系统调用有9个，分别是：socket，read，write，connect，bind，listen，accept，shutdown以及close。

我们首先介绍socket总体设计：

socket中需要维护两个缓冲队列，一个负责缓冲socket接收到的数据，另一个负责缓冲socket需要发送的数据，socket还保存了它感兴趣的远程地址和远程端口号。每个进程中。会维护一个整数到socket的映射`fd_map`，在创建socket的时候会返回其对应的整数，而对于其他系统调用，会有一个输入参数是socket对应的整数。此外，我们还需要维护一个全局的表，表里面存放的是哪些进程对哪些UDP端口接收到的数据感兴趣。

我们接下来对每个系统调用的建立进行论述：

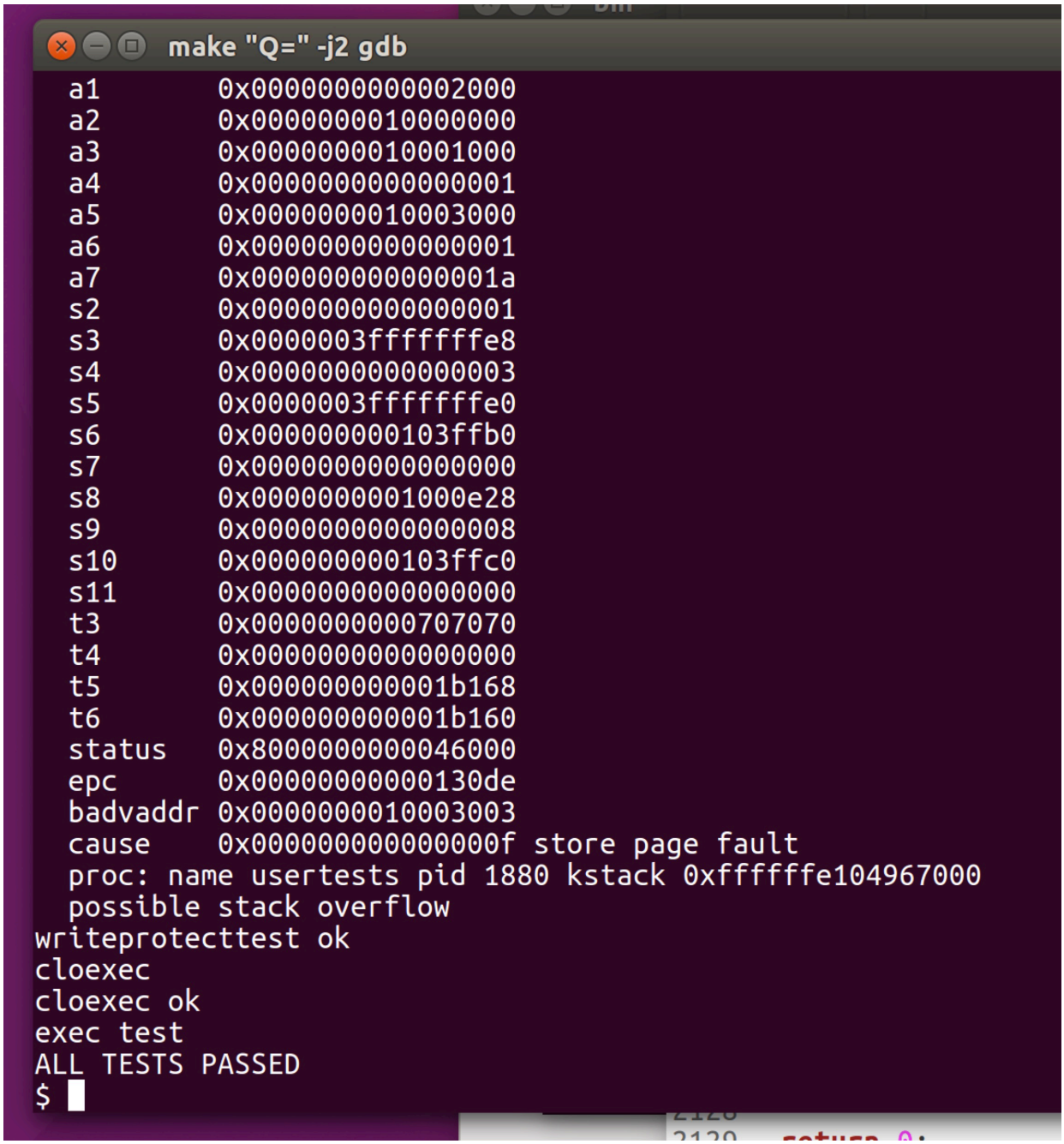
1. `socket`：首先检查参数合法性，然后创建socket即可。
2. `read`：首先进行相关检查，然后从接收缓冲队列中取出一个数据包。
3. `write`：首先进行相关检查，然后向发送缓冲队列中放入一个数据包。
4. `connect`：首先进行相关检查，并设定自己感兴趣的远程地址和远程端口号。
5. `bind`：首先进行相关检查，设定自己的地址和端口号并修改全局的表，告诉操作系统当前进程对该端口号感兴趣。
6. `listen`：首先进行相关检查，UDP不支持`listen`系统调用，直接返回错误码。
7. `accept`：首先进行相关检查，UDP不支持`accept`系统调用，直接返回错误码。
8. `shutdown`：首先进行相关检查，然后关闭socket。
9. `close`：首先进行相关检查，从全局的表中移除bind系统调用添加的信息，然后从`fd_map`中移除相应的整数到socket的映射。

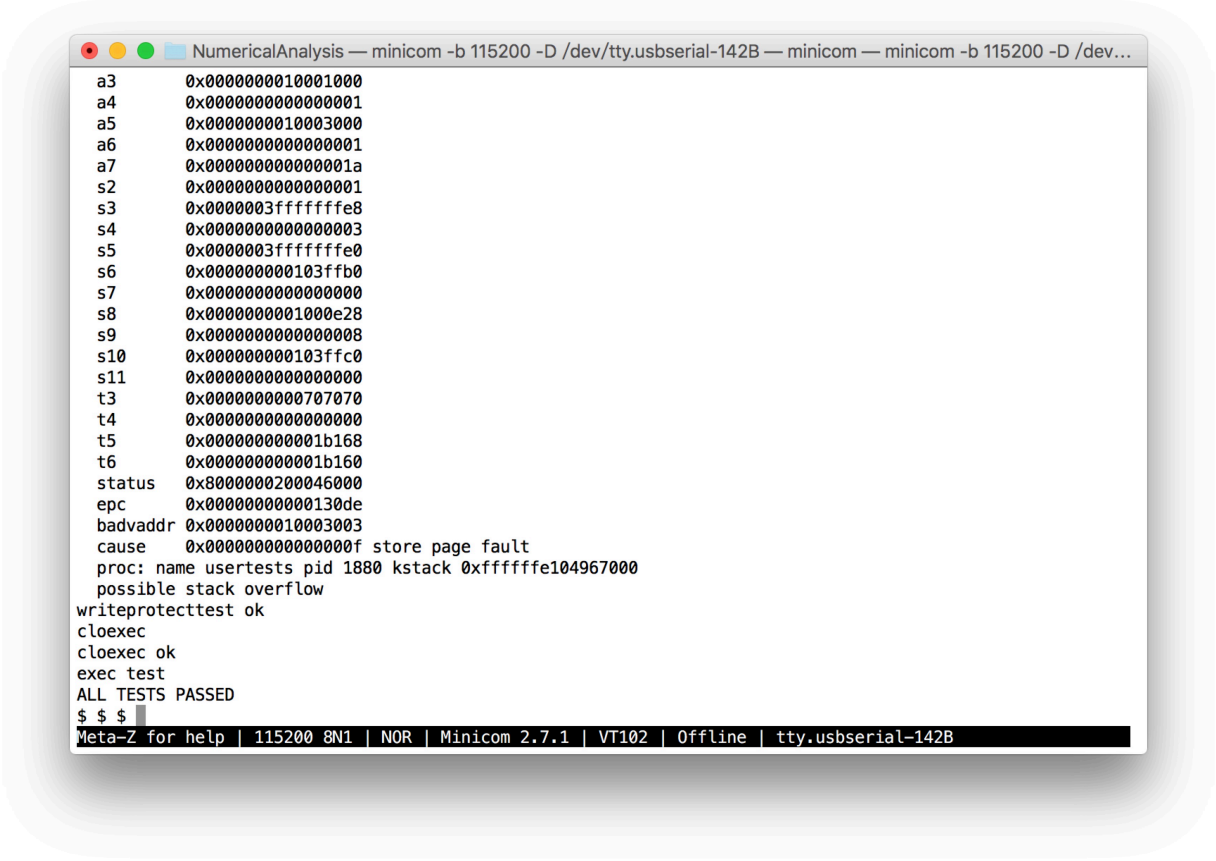
结束语

实验成果

sv6移植

如下图片为sv6移植版本分别在QEMU模拟器以及HiFive Unleashed真实硬件（通过串口连接）上运行的情况，从图中可以看出，功能测例除浮点运算、信号以及几个运行较慢的测例外，均分别能够在模拟器以及真实硬件上测试通过。


make "Q=" -j2 gdb
a1 0x0000000000002000
a2 0x0000000010000000
a3 0x0000000010001000
a4 0x0000000000000001
a5 0x0000000010003000
a6 0x0000000000000001
a7 0x000000000000001a
s2 0x0000000000000001
s3 0x0000003fffffffe8
s4 0x0000000000000003
s5 0x0000003fffffffe0
s6 0x000000000103ffb0
s7 0x0000000000000000
s8 0x0000000001000e28
s9 0x0000000000000008
s10 0x000000000103ffc0
s11 0x0000000000000000
t3 0x0000000000707070
t4 0x0000000000000000
t5 0x000000000001b168
t6 0x000000000001b160
status 0x8000000000046000
epc 0x00000000000130de
badvaddr 0x0000000010003003
cause 0x000000000000000f store page fault
proc: name usertests pid 1880 kstack 0xffffffe104967000
possible stack overflow
writeprotecttest ok
cloexec
cloexec ok
exec test
ALL TESTS PASSED
$



同时，我们的sv6移植能够解析boot loader传入的FDT，获知系统内存大小以及处理器核心数量，无需硬编码。

此外，由于sv6优越的并行性，就我们目前有限的调查分析，我们移植到RV64 SMP环境的sv6可以称得上是RV64 SMP环境下并行性能最好的操作系统。

Commuter分析与理解

此部分成果体现在上面几节对于Commuter的详细分析和阐述中。

基于Commuter对POSIX socket API建模

我们还基于Commuter对POSIX socket API的UDP（协议）部分进行了建模，目前的模型描述了从传输层到应用层的socket的行为，未来还可以用此模型对实际操作系统的网络子模块进行并行性测试与优化。

实验收获

在本次课程设计中，我们：

- 熟悉了RV64指令集及其工具链
- 进一步熟悉了x86-64指令集
- 了解了sv6的设计与实现
- 了解了C++运行时环境的设计
- 了解了RV64的代码模型
- 了解了Berkeley Boot Loader的设计与实现
- 熟悉并经历了一个多核操作系统的移植过程
- 学习了符号化执行方法

- 学习了Z3 SMT求解器的使用方法
- 深入理解了Commuter各个子模块的设计与实现
- 较为深入地了解了POSIX socket API
- 简单了解了QEMU模拟器的设计与实现

心得体会

谭闻德

在做本次课程设计之前，我还不敢十分确定我们能够成功移植sv6，不过最后我们确实成功完成了。通过本次课程设计，我学到了实验收获中描述的很多很多很多的知识。通过和自己小组成员合作，我们增进了感情和友谊；通过和不同小组联络、共同讨论问题，全系同学们的友谊和凝聚力进一步加深了。

总之，本次课程设计应当说是能够对我今后从事学习、科研工作有很大的帮助。

顺便，膜峰。

尹宇峰

本次课程设计对我来说一次非常好的锻炼，尤其是Commuter部分，十分有意思而且富有挑战性。通过分析和理解Commuter的设计和实现，让我阅读论文和大工程代码的能力得到了极大的提升。比较可惜的是，因为时间有限，我们最终并没有将基于Commuter的POSIX socket API模型投入TestGen生成测试代码，让其可以在真实os上跑起来。

另外值得一提的是，由于看Commuter太入迷了，最近用git上传代码的时候，我都把`git commit`给打成了`git commute`了。

未来的工作

我们希望可以将基于Commuter对POSIX socket API所建得的模型，结合Commuter中的TestGen模块生成相应的测试代码，以便可以对实际操作系统的网络子模块进行并行性测试与优化。

致谢

感谢陈渝老师、向勇老师对于我们课程设计的耐心指导；

感谢辉总给予我们硬件的大力支持；

感谢路橙、于志竟成组与我们小组合作与讨论。

参考文献

1. sv6 https://github.com/aclements/sv6
2. Commuter https://pdos.csail.mit.edu/archive/Commuter/ https://github.com/aclements/Commuter
3. The RISC-V Instruction Set Manual Volume I: User-Level ISA

   https://riscv.org/specifications/
4. The RISC-V Instruction Set Manual Volume II: Privileged Architecture

https://riscv.org/specifications/privileged-isa/

5. bbl-ucore https://ring00.github.io/bbl-ucore/
6. ucore labs RISC-V 64移植 https://gitee.com/shzhxh/ucore_os_lab/tree/riscv64-priv-1.10
7. Booting a RISC-V Linux Kernel https://www.sifive.com/blog/2017/10/09/all-aboard-part-6-booting-a-risc-v-linux-kernel/
8. Entering and Exiting the Linux Kernel on RISC-V https://www.sifive.com/blog/2017/10/23/all-aboard-part-7-entering-and-exiting-the-linux-kernel-on-risc-v/
9. Paging and the MMU in the RISC-V Linux Kernel https://www.sifive.com/blog/2017/12/11/all-aboard-part-9-paging-and-mmu-in-risc-v-linux-kernel/
10. 符号执行入门 https://zhuanlan.zhihu.com/p/26927127
11. mini-mc https://github.com/xiw/mini-mc
12. Z3 https://github.com/Z3Prover/z3
13. HiFive Unleashed https://www.sifive.com/products/hifive-unleashed/
14. Freedom U540-C000 Manual https://www.sifive.com/documentation/chips/freedom-u540-c000-manual/
15. 翼辉信息 http://www.acoinfo.com/index.html
16. MTrace https://github.com/aclements/mtrace