输入IR

对func进行merg

对多个func生成一个加速器

MIP求取最优的方案

Early dse

合并精简结构

**给定soc具有很多加速器，App，给定面积约束**

将应用映射到soc上面

那些函数可以固化成硬件

Early dse

**给定一些设计参数 对加速器进行c描述 新加app 进行c描述 把他们进行合并**

**Merge**

**设计空间有多大**

**谁和谁merge 谁merge？ 谁用硬件 ，谁用软件**

**阿拉丁 硬件怎么表示？**

**Merge**

**Function merger**

启发式的策略

怎么将硬甲用软件程序写出来

可重构硬件：：流片之后依然可以被灵活调整的硬件

数据流图

专用硬件电路（用硬件实现软件功能） 实现对特殊器件的重用

通过对软件和时钟的控制自动生成配置文件，实现了在几十ms内完成的芯片数据通路的重构

一些优化方法 ：

子图相似重构

感知缓存，轮摆式加载技术

并行化控制方法

可重构芯片与这种形式的区别

对整个soc系统进行的优化

手动和临时识别代码非常困难

对加速器进行合并可以改善soc的性能

计算机辅助设计工具自动化

这是第一个创建粗粒度、控制和数据流丰富的**合并加速器**的自动化方法。

早期的soc设计空间探索（提前设计出性能良好的/符合要求的soc）

为什么细粒度合并不行

合并和非合并加速器 已经用软件实现

合并的时机，或者说是合并的条件

P‘是PUM，merge函数后加入P

有了各个函数的评价指标 给个个函数进行分类？排分

选择不同的函数进行硬件化？进行合并，合并的前提是有更好的指标（面积或者延迟）

输入是soc，一组函数（有硬件有软件）

对硬件加速器进行建模 使其表示成函数的形式？？？

统一表示？

函数表示的下一级别可以是cdfg，各种图结构，有了图结构，可以进行分析各种的可能优化后的面积和延迟等性能信息。

有了这些信息后可以选择partion，

然后使用评估模型判断该设计选择是否合理

加速器建模？这里加速器建模指的是直接生成了性能参数 ，加速器有现成的RTL的代码吗？

合并候选加速。据我们所知，我们首次合并粗粒度加速器。我们描述了有效识别和合并加速器所需的机器学习模型、代码生成技术、指标、动态分析和优化技术。

• 大面积设计的自动化设计空间探索。我们提供从高级 C/C++ 到 RTL 的自动化工具流，当应用程序可综合时，它会生成合并的加速器。当应用程序仅以不可合成的格式提供时，AccelMerger 仍然能够提供关于最适合加速和合并的代码的见解。通过使用神经网络，可以使用 AccelMerger 在不到 10 秒的时间内分析 4.5×106 个合并加速器，而纯 Late-DSE 技术很难在不到 22 小时内分析 800 个合并加速器。

• 合并和非合并加速器选择。 AccelMerger 能够选择原始程序中功能的合并、非合并和软件版本的最佳组合。

我们提供了一个混合整数规划模型，该模型可以在嵌套 CDFG 上运行，同时根据区域预算决定合并什么。

### Neirong

任意力度合并

选择不同的函数进行合并后，应该生成合并后函数的代码RTL，和cdfg，后续会使用相关的模型进行性能分析。这里指的是具有面积减少的合并操作

加速器建模

软硬件划分

是否最终选择合并两个函数需要考虑到的不仅仅只有面积方面的属性

前端：

输入源代码，输出为加速器面积预测模型结果（可以确认那些函数可以被合并？）

合并：

1. 候选人排名
2. 合并驱动
3. 面积，延迟模型
4. 属性有点

获得分区可能结果

评价最终的结果（有许多的指标）

函数很多，首先根据函数是否相似选择出待定的func进行**合并的尝试（具体怎样合并函数也是一个需要注意的问题比如参数合并，函数体合并）**

然后，利用延迟，面积等模型判断这样的模型是否有利可图，

如果有利益，则应该保留该合并操作，并在最后的规划期间选择合适的合并函数进行hw，sw之间的划分。

我们通过根据指示两个函数之间有多少共享指令的简单指纹对函数对进行排序来开始函数合并过程。此功能允许合并以扩展到大型代码库。然后将最相似的候选者线性化，这意味着我们将图结构转换为顺序的指令串。在 AccelMerger 中，我们对不同的 post order 进行采样（每个基本块在其所有后代之后被访问）线性化以寻找高相似度匹配。在每条路线上，我们？？？

这些工作中比较耗时的操作是？

1. 通过神经网络模型对面积进行预测？
2. 使用nW算法计算合并函数之间具有多少重复的指令
3. 对指定的函数生成合并后的函数表达？

NW 指示哪些指令可以重用，哪些不能。

为了使用NW算法，我们需要对f进行线性化，因为NW算法是最长子序列对比算法。首先要将f表示成指令序列？

而在llvm 的ir中，其表示为cdfg，并不是指令序列

首先过滤掉面积没有缩减的合并函数

其次过滤点延迟比较严重的函数

其次过滤掉EP值为负的合并函数

Function merging

三个步骤

1. 将函数线性化，将cfg转化为一系列标签和指令
2. 应用序列对比方法，识别序列之间的相似区域
3. 执行代码生成

函数的调用可能会出问题？因为参数的数量不一致

包留原函数？？或者设置默认参数？

Vi指的是原函数 属于元素在 新函数中的位置

函数的重写？？

格式不一致？？？？

考虑的是一个问题吗？？

使用nw算法减少代码数量

使用fingerprint选择合适的函数进行merge

怎么生成rename表呢？

在merge的过程中生成rename table

参数相同则直接重命名为

参数不同使用select

利用nw算法可以选择对齐方式，但不是所有的函数都需要对齐。

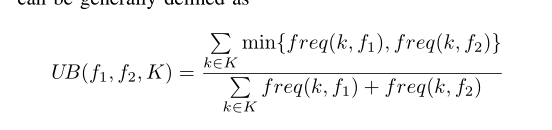
到底什么是价值高的？Rank值高的？

最后一个语句不一致怎么办？

一般情况下是一样的ret

只能一对一合并 如果是多个要怎么解决

Bool类型 bin类型



各个操作符的频次除以总的操作个数，相似的操作数个数占全部百分比

每个函数都有类型，ir指令频次

根据频次相似度，设置分数

根据类型相似度设置分数

设置阈值，探索几个相似的函数执行merge，观察时机收益

执行轻量级的探索

我们需要调用图

我们的函数合并是应用在硬件上面的。函数的合并本来就是可以在不同的时机进行合并

合并后的代价，或者收益进行量化

每条ir代码的成本模型llvm编译器中对应模块中有相应的模型 ，实际评估模型

### Youtube：

<https://www.youtube.com/watch?v=sOCFYfF3iwE>

Fast focus on function merging

：

讨论的是如何加速识别不同的函数之间的相似性，进而选择不同的函数进行合并

我们的目标是？

不同的函数可以用软硬件实现，进而达到解决面积，减少latency的目的

Merge一般是从减少代码数量的角度出发的

我们需要的仅仅是对函数进行merge，找出使函数占用面积最小的merge方案？

我们需要的是软硬件指标？

什么东西可以表示成软件，什么东西可以表示成硬件？

Q1 函数合并似乎只能两两合并

Q2 合并的目的是为了减少代码数量，而没有考虑实际的硬件开销

以往的合并操作，操作对象是IR，控制流图

在accelmerge中

，操作对象是dcfg，包括了控制流和数据流

每个func配备的有cdfg，dyn，syn等信息

然而，Early-DSE 没有准备好利用可变 CDFG 粒度，更不用说对重用 CDFG 逻辑的架构进行推理。

直接把soc转换成相应的func，挑选合适的function进行merge？设计空间探索，减少设计空间的大小

对函数进行merge会

他是怎么考虑带宽的，从带宽的角度考虑加速器的设计

粗粒度加速器？ 加速任务？

细粒度加速器 加速计算

生成许多设计点 有不同的性能 和 面积开销

约束函数

Merge之后一定需要用硬件实现吗？

### AutoDSE

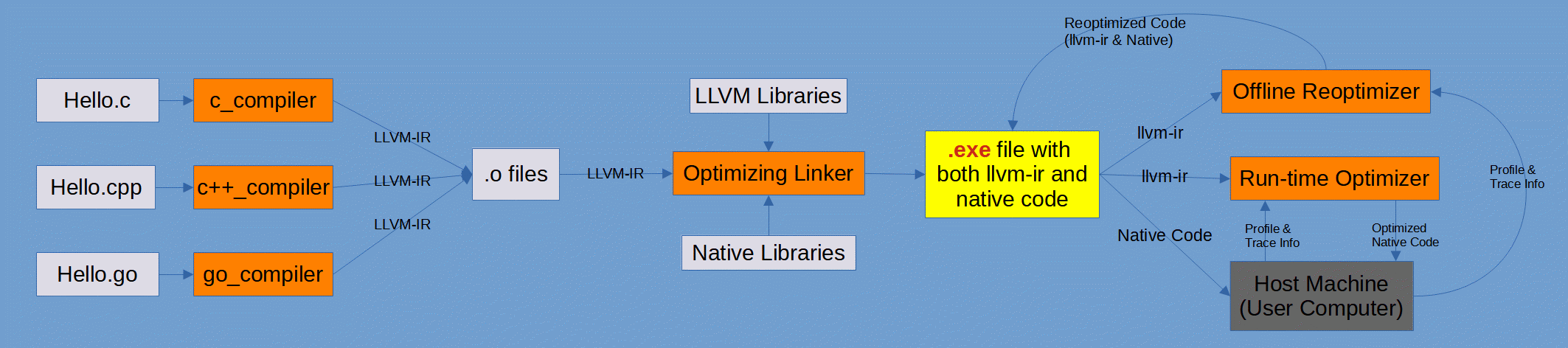
个人空间

<https://web.cs.ucla.edu/~atefehsz/>

Merge的一个显而易见的好处是减少了面积开销

牺牲了延迟，换来了面积

### LLVM



llvm/examples

这个目录主要是一些简单例子，演示如何使用LLVM IR 和 JIT。还有建立一个简单的编译器的例子的代码。

建立简单编译器的文档地址：http://llvm.org/docs/tutorial/

llvm/include

这个目录主要包含LLVM library的公共头文件。

llvm/lib

这个目录包含了大部分的LLVM的源码。在LLVM中大部分的源码都是以库的形式存在的，这样不同的工具之前就很容易共用代码。

llvm/projects

这个目录包含着一些依赖LLVM的工程，这些工程严格来说又不算LLVM一部分。

llvm/runtimes

这个目录包含了一些库，这些库会编译成LLVM的bitcode，然后当clang linking 程序的时候使用。

llvm/test

这个目录是LLVM的测试套件，包含了很多测试用例，这些测试用例是测试LLVM的所有基本功能的。

llvm/tools

这个目录理是各个工具的源码，这些工具都是建立在刚才上面的那些库的基础之上的。也是主要的用户接口。

llvm/utils

这个目录包含了一些和LLVM源码一起工作的应用。有些应用在LLVM的编译过程中是不可或缺的。

修改硬件结构 使其具有可拓展性，，比如功能添加。给出建议？

这个是不是需要硬件的verilog表示？

给定的输入 是加速器的什么形式呢？

Aladdin好像是给定一组输入程序 ，然后对程序进行并行性分析，循环展开等手段，模拟程序的硬件area，latency等？这实际上就是加速器的硬件模型？？？

高层此硬件建模代码？

利用 sa sequence alignment ，来对两个函数进行对比 （执行函数合并的第一阶段）

发现merge时机

进行merge

对于每个函数 f1，我们使用优先级队列来根据所有其他函数 f2 的相似性（由 s(f1, f2) 定义）对最相似的候选者进行排名。我们使用探索阈值来限制我们将为任何给定函数评估的最佳候选者的数量。然后我们以贪婪的方式执行这个候选探索，在找到第一个导致有利可图的合并的候选并提交该合并操作后终止。

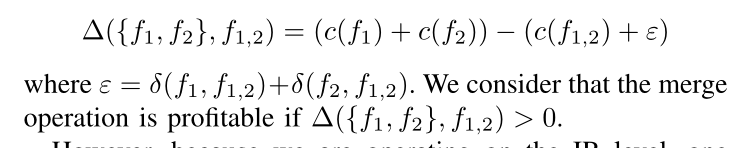
### Func merg工作流程

1. 对all输入函数进行**fingerprint**进行提取（）
2. 对每个函数f1，比较所有的函数的**相似程度**，及合并指标，进行排名ranking
3. 根据设定的排名**阈值**，对阈值内的函数进行merge**尝试**，并利用**开销模型**评估该操作是否有利可图
4. 提交有利可图的merge，并更新函数调用图等。将修改提交至优化列表。

### 函数合并流程

选择要合并的函数、生成合并的函数和估计合并的性能。

Tip：



如果从硬件面积减少的角度考虑开销模型，对于硬件加速器来说可能有更大的收益

FM:

时间开销主要集中在

1. Rank ：F3M
2. Alignment ：HyFM

Hyfm旨在减少代码大小的同时，减少过程所需要的内存数量（10x）和编译时间（5x）

我们以每个基本块的方式对齐输入函数。首先，我们通过最小化指纹之间的距离来配对相似的基本块。然后，我们只对齐每对基本块内的指令。即使使用二次对齐算法，基本块通常也比函数短得多，从而转化为更快的对齐方式。

• 我们提出线性成对对齐作为二次对齐的替代方案。对于高度相似的基本块，它实现了相似的结果，但时间和空间开销可以忽略不计。

• 我们在实际生成它们的合并代码之前估计对齐的基本块的性能。如果不适合，我们将忽略它们，从而提高整个合并函数的整体盈利能力并简化代码生成。如果一对函数中的所有配对块都是不可配置的，我们完全跳过合并函数对，与 SalSSA 相比加快了优化过程。

LLVM

Phi节点

什么是 PHI [node](https://so.csdn.net/so/search?q=node&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_30846599/article/details/_blank)？

所有 [LLVM](https://so.csdn.net/so/search?q=LLVM&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_30846599/article/details/_blank) 指令都使用 SSA (Static Single Assignment，静态一次性赋值) 方式表示。意思是所有变量都只能被赋值一次，这样做主要是便于后期的代码优化。

1 a = 1;2 if (v < 10)3 a = 2;4 b = a;

假设 v 的值小于 10，变量 a 就要被赋值为 2，但 a 已经被赋值了一次，由于 SSA 性质的约束，只能赋值另外一个“a”。最后在给 b 赋值时，通过添加一个 PHI node，由其来决定选择哪个版本的 a 来给 b 赋值。

1 a1 = 1;2 if (v < 10)3 a2 = 2;4 b = PHI(a1, a2);

PHI node 根据控制流是从哪一个 block (“a1 = 1” or “a2 = 2”) 到达，来决定使用 a1 还是 a2 来给 b 赋值。

这些 PHI node 必须在 IR 中显示创建，llvm 指令集中有对应的 phi 指令。

Hyfm：

1. 对基本快提取 fingerprint ，在基本快之间进行rank
2. 如果拒绝两个函数间的一定程度基本块，则整个函数都会被拒绝（阈值，做实验得到吗）
3. 基本快的对齐 选择使用 相同长度的基本块进行对齐
4. 实际上放弃了很多的合并机会

Pa线形成对对齐 ：

进一步限制了可以执行merge操作的基本块

具体来说，我们使用代码的 MinHash [10] 表示作为指纹和局部敏感哈希 (LSH) [11] 来快速发现指纹空间中的近邻。

可变参数

python的可变参数是指传入到同一函数中的参数数量可以不同。可变参数会将传入的参数组成一个tuple.

形式

在参数前加\*，如：

def sum(\*num):

S=0

for n in num:

S+=n

return S

调用时可以这样

sum(1,2,3,4)

sum(1,2)

sum()#一个参数都不传入

如果有a=[1,2,3],要调用该函数计算元素之和，写成 sum(a[0],a[1],a[2])就太繁琐，可以在a前面加个\*，然后传入函数：

sum(\*a)

关键字参数

关键字参数会将传入的0个或多个含参数名的参数组装成一个dict.

形式

def student(name,class,\*\*kw):

print('name:',name)

print('class:',class)

print('other:',kw)

调用时

当然，如果已经有一个dict，要想把其当成关键字参数传入函数，与上面的可变参数类似，在dict前加上\*\*

注意：函数内对传入的dict的修改不会影响函数外的info。

————————————————

版权声明：本文为CSDN博主「lccrun」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/lccrun/article/details/88421475>

使用J指数捕捉函数相似性与可合并之间的关系

j指数是线性的 中性应用程序使用问题规模也很大

使用minhash，其可能的结果与j相差不大 ，误差仅为O【根号k】 （k为哈希函数的个数，也是fingerprint的长度）

通过调整k的参数 ，可以选择更为准确的模型

将指令转换为32整数，32位整数对操作数不敏感，忽略操作数的相关特征

输入索引是benchmark，根据索引的不同选择不同的应用程序

对应用程序进行编译 得到 中间表示 IR

然后对IR进行merge的分析

生成最终的iR表现形式

统计开销 和代码大小

global\_flags

里面表示了 程序的输入是什么 ，要做到和里面的文件一样的文件结构

有些时候需要修改其他的 config文件

3.43 -mergefunc：合并函数

此pass查找可合并的等价函数并折叠它们。

在函数集中引入了total-ordering：我们定义了对应于每两个函数中哪个更大的比较。它允许将函数排列到二叉树中。

对于每一个新函数，我们都要在树中检查其等价性。

如果等价存在，我们折叠这些函数。如果这两个函数都是可覆盖的，那么我们将该函数移动到一个新的内部函数中，并给它留下两个可覆盖的重击。

如果没有相等的，那么我们将这个函数添加到树中。

查找例程复杂度为O(log(n))，而整个合并过程复杂度为O(n\*log(n))。

阅读本文了解更多细节。

3.44 -mergereturn：统一函数退出节点

确保函数中最多有一个ret指令。此外，它还跟踪哪个节点是CFG的新退出节点。

FastFM 好像并没有将函数划分为 一个一个的块 。

二hyfm技术是将函数划分为块，减少搜索的空间

划分为块的依据是啥来着 是一个基本控制流块

模板函数的使用情况

：本质上是代替了不同的相似功能的函数（函数只有参数的类型不一致）。此时起到了相似功能函数合并的作用

通过模板，可以根据输入参数的类型确定所使用到的函数的类型



加上取地址符号后，可以直接修改对应地址的目标元素，起到了直接修改外部变量的作用

\*值得是传入的是一个指针，即地址

Private，**只能通过类的成员函数访问，确保了开发的程序不受破坏**

**使用方法操作对象和 使用对象的方法之间的区别**

**有了fingerprint之后要做的事情是？**

**挑选function进行similiar**

**有一个简单的思路**

**手动生成M，M中仅包含两个函数**

**相似性阈值会因为程序的规模而发生改变**

**函数数量少时，可以适当进行更大范围的相似性合并，增加合并结果的正确性，合理性。**

**函数数量多时，可以较小相似性阈值，加快函数合并rank的速度，进而加速整个函数的编译过程**

**根据程序的大小 选择不同的T阈值**

线性化的过程在哪里？

好像是要对 alignment的时候才进行的

工作流程：

函数输入 module M ，后端平台信息**TTI**？

1. 生成技术选项option

根据类型 指令操作码的的次数等信息生成了指定长度的minhash向量

MatcherEntry

存储了很多有用的信息

Add Conniedine 设置hash值，并将其加入bucket中 ，供以后查询他调用

生成hash值以后，会对

Python调用顺序

通过生成shell命令 调用execute命令，执行命令行输入，运行相应的脚本文件，并且传入对应的参数（编译对象是什么？技术是什么？），打开F3M等技术

输出在哪里呢？M直接传入M，传出的也是M

用LLVM创建模块之前，需要首先初始化LLVMContext，然后才能调用LLVM API，代码如下（示例）：

自动识别为M？

编译后将文件生成ll文件

Aladdin的

赶紧把东西弄出来

通过状态机的拆分实现通用性和高效性（paogrammabe）之间的权衡



ADD\_EXECUTABLE(hello ${SRC\_LIST}) ⽣成的可执⾏⽂件名是hello，源⽂件读取变量SRC\_LIST中的内容

输入 输出

工作流程

预处理：处理头文件等

编译：将高级语言编译为汇编语言

汇编：将汇编语言翻译为机器语言（2进制）

链接：将不同的二进制文件连接成一个文件

Lib uer/lib usr/local/lib

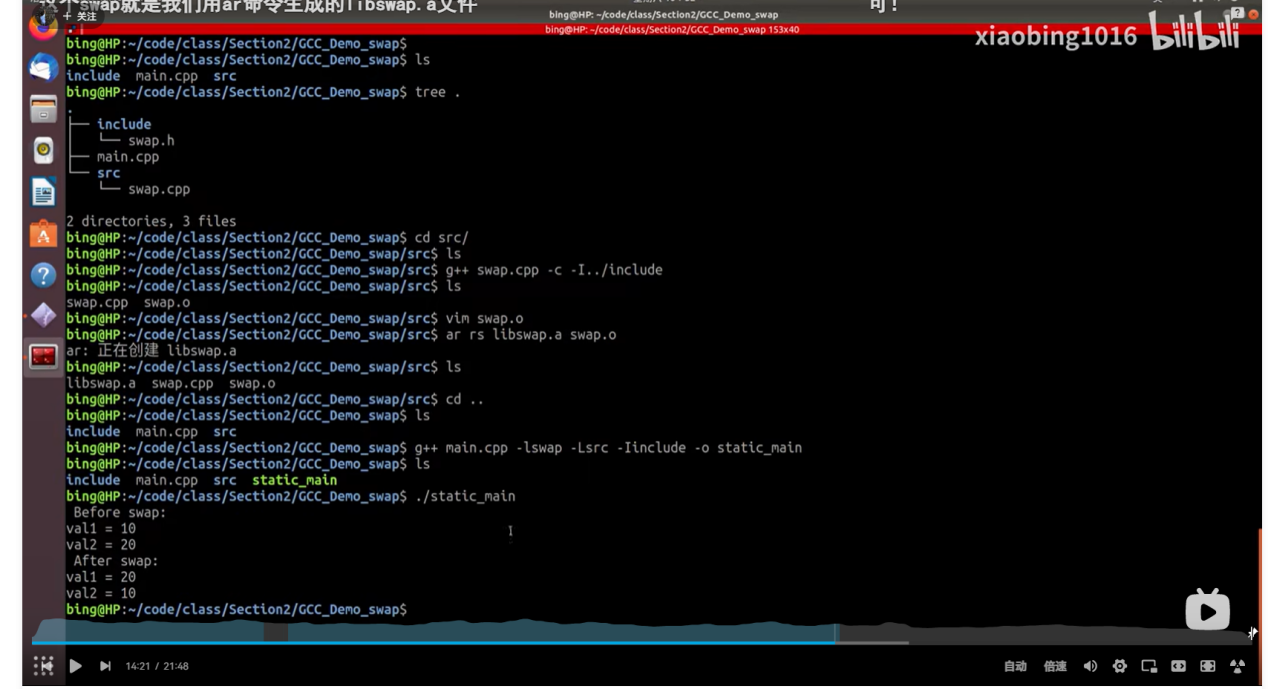
链接库（二进制文件）

Gcc -D 定义宏 咋debug

需要制定头文件和源文件的位置

很明显pass文件里面有许多链接的库？

1. 将llvm里的那个pass转化成一个动态库



1. 使用opt等方式调用该动态库？同时对目标程序进行编译执行
2. 将整个过程封装？完成任务
   1. 修改benmarks里面的内容，在执行的过程中添加只包含两个函数的benmark，然后运行整个过程
3. 修改对应pass实现任意函数的合并
   1. 通过生成包含两个函数的文件直接传入调用动态库实现任意函数合并
   2. 提取其中的merge函数
   3. 自己写一个merge函数

**调试gem5-aladdin环境，模拟器**

**对所有硬件加速器表示的函数进行重命名标识出这是一个硬件加速器函数**



Os

Make lib

hello.Cpp

Cmake

Makefile

Make

Make install

Autotools(autoconf,automake,...)>>>>>Configure（makefile.in）>>>>makefile

Confugure根据系统信息将makefile.in转换成makefile

将merge后的结果传递给gem5-aladdin项目集成

自动解压文件的一个脚本

#!/bin/sh

ftype=`file "$1"`

case "$ftype" in

"$1: Zip archive"\*)

　　unzip "$1" ;;

"$1: gzip compressed"\*)

　　gunzip "$1" ;;

"$1: bzip2 compressed"\*)

　　bunzip2 "$1" ;;

\*) echo "File $1 can not be uncompressed with smartzip";;

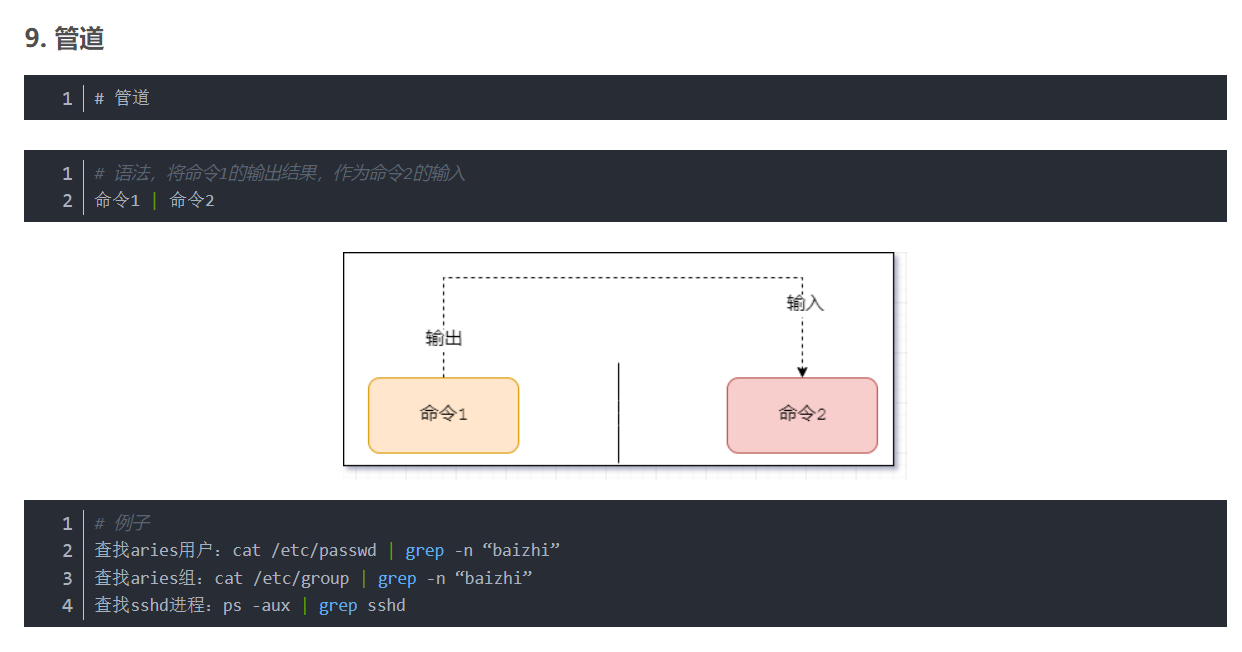
esac

————————————————

版权声明：本文为CSDN博主「tealex」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/tealex/article/details/69397776>

单引号更严格一些。它可以防止任何变量扩展。双引号可以防止通配符扩展但允许变量扩展。



Shell是一种弱类型语言 并不会强调变量的类型

对于function来说 输入就是一个程序 输出呢？ 就是合并了什么函数？或者说是统计了一些函数的大小信息 ，顺便统计了执行函数的时间

？利用插桩的方式

开发这个项目的过程是什么？

Llvm添加benchmark怎么做？

Loop extractor与F3M是否有一点点的作用重合？

F3m使用到了

Loop extractor提取的时机

Linux环境变量值之间是通过冒号进行隔开的( : )

Path=/address/:$path

Export path暂时的添加环境变量

三、为当前用户永久添加**环境**变量：

编辑 .bashrc 文件 vim ~/.bashrc

文件末尾添加：export PATH=“/usr/local/nginx/sbin/:$PATH”

source ~/.bashrc

四、为所有用户永久添加某一环境变量：

编辑 /etc/profile 文件 vim /etc/profile

文件末尾添加：export PATH=“/usr/local/nginx/sbin/:$PATH”

source /etc/profile

五、/etc/environment 下面添加，

打开 /etc/environment ，在最后添加，运行下面2条命令。

sudo vi /etc/environment

source /etc/environment

————————————————

版权声明：本文为CSDN博主「hooking\_parse」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/qq_35490522/article/details/125405469>

C++继承方式 public继承 protect继承 private继承

一些时候using起到了typedef的作用

****#include<iostream>****

****Using cout ;****

****Using endl;****

****#include<iostream.h>****

****Using namespace std;****

****. 将命名空间释放出来****

****注：using namespace std 告诉编辑器我们将要使用名字空间std中的函数或者对象，所以cout和endl前面不用注明他们是std这个名字空间中的cout和endl****

****标准库里大概都有一些什么样的函数？和数据定义****



）子类对父类成员的访问权限跟如何继承没有任何关系，“子类可以访问父类的**public和protected成员，不可以访问父类的private成员”——这句话对任何一种继承都是成立的。**  
（2）继承修饰符影响着谁可以知道“继承”这件事。public继承大家都知道，有点像“**法定继承人**”，因此，任何代码都可以把子类的引用（或指针）直接转换为父类。也因为这个原因，public继承常用来表达设计中所谓的“is-a”关系。private继承则有点像“**私生子**”，除了子类自己，没有人知道这层关系，也因此，除了子类自己的代码之外，没有其它人知道自己还有个父亲，于是也就没有其它人可以做相应的类型转换。为此，私有继承常用于表达非“is-a”的关系，这种情况下子类只是借用父类的某些实现细节。protected继承则有点特殊，外界同样不知道这层关系，但家族内部的子孙们可以知道，有点像“自家知道就行了，不许外扬”的意思，于是子孙们是可以做这种向上转型，其它代码则不可以。因为这种特殊性，protected继承在实际中用得很少。

Using的三种用法：

1. 引入命名空间 using namespace
2. 类型别名 typedef
3. 在子类中引入基类的成员 private继承时，基类的成员对派生类对象并不可见，可以将其通过using引入，转换为派生类的成员

构造函数和析构函数？

构造函数负责在类实例化时警示数据的初始化一般与类名保持一致。调用时自动执行

析构函数相反，在对象结束其生命周期时，自动执行析构函数

this指针

一般来说 this指针指向用来调用成员函数的对象（this作为隐藏参数传递给方法），在例子中， item.topPrice(item1)，将this设置为item对象的地址，注意每个成员函数（包括构造函数和析构函数）都有一个this指针，该指针指向调用对象。如果方法需要调用整个调用对象，则可以使用表达式\*this,this是对象的地址，\*this才是对象本身，另外，在函数的括号后面使用const限定符将this限定位const,使得不能使用this来修改对象的值

————————————————

版权声明：本文为CSDN博主「xiaoliyu00」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/cmdxly/article/details/123993842>

Const的用法读法以及一些特殊含义

基类private成员在派生类中不能被访问，如果基类成员不想在派生类外直接被访问，但需要在派生类中访问

父类和之类的相互赋值，三种赋值方式

1. = 只能打的子类给父类赋值
2. 引用 human& hm = st

Print（“）

变量和声明

引用变量可以理解为后续变量的别名。具有同等的性质



预处理

编译：

词法分析

语法分析

语义分析

Pass优化





Clang

-i 制定include搜索路径

-l 指定库的名字？

-L制定库的搜搜地址