乐乐(00:09:37): 然后你也是头皮囊？他都没在后头。他都没在后头把声音关掉。

刘晓义(00:18:00): 今天我们再次重新定义七点钟。正式大概7十开始，大家可以拿一点饮料，零食。Also 今天的 youtube 直播修好。所以。

刘晓义(00:19:56): 是的，我们的活动会有录制会上，通常来说网站上会 eventually 更新录播。youtube 因为我们过去一年时间里面活动都有 youtube，所以直接看 youtube 的回放就可以。

刘晓义(00:21:12): 可以听到我说话吗?

zenkexi(00:21:18): 腾讯会议这边是可以的。

...(00:29:26): 喂我肯定对的，这个是挂胸口的那种，用这个，然后我帮你把摄像头打开。还要开摄像头吗？没有是他这个摄像头可牛逼了，他这个摄像头是不是摄像头不好意思，这个让大家看到了我的脸。吓到了没晚饭吃。因为因为好像不太对劲。可以调互动视角，不是远程互动，全景近景反出全景。挺好的就出来了。Ok.

...(00:30:27): 现在可以开始了吗？我可以先介绍一下你或者你自我介绍一下。待会 PPT 里面会有个自我介绍没有那就。可以开始了，可以开始！大家好，我叫龙英，然后今天跟大家介绍这个话题是在 LV M 中支持一个。新的向量指令集。在所有事情开始之前要感谢一下 tuna，感谢喵喵给了大家这个机会来讲这个 talk?

...(00:31:06): 介绍一些背景。我在一年前为某个信创 R 的 S IM D 扩展编写了自动向量化编译器。它其实是分为 LV M 跟 GCC 两个不同的项目，我们就接到了 LV M 的项目，因为老师比较嫌弃 GCC，反正就没有接。然后指标有性能指标跟功能指标，我主要负责就是所有的代码编写工作。比如说所有的 CPP 所有的 TD 包括怎么改之类的，然后本次 talk 的地址？可以在这个地方看见 ok. 背影介绍完之后，我们就直接进入正体。大概要讲这么一些东西，首先第一个就是什么程序，它搞向量化是有收益的？第二个就是我们要做向量化，需要做什么工作。第三个是介绍一个基本概念。用五分钟的时间入门 LV M。就可以开始一些练习题了。第四个是从标量 AR 到向量 AR 我们说去如何去指导一些已有的一些 pass 去完成这个工作。第五个就是去把向量 AR 变成真正的机器指令，主要是介绍这个指令选择算法。那第五点会是我们今天晚上讲的比较多的一个部分，然后会涉及到一些骚操作跟踩过坑。第六个就是一些吹牛 future work，大家都是说着玩的。

...(00:32:48): 然后第七个是小杰，首先来第一个就是什么样的程序做向量化有收益，大家可能脑子里最开始想到的是 M copy 之类的。看起来就是很大 loop 的程序。但是这个地方给出了毕生编译器必声这个词在上面很难打出来，所以我们一般就用拼音代替在 o3下的 spec2017 这个整体提升情况，大家可以看到。必胜编译器，绝大多数都是富有化。只有叉264有75.12%的提升，最后在75.12%的情况下，虽然别的东西都是负优化或者没有优化，但几何平均也能优化个6%真是奇迹。

...(00:33:37): 下面这个是 o3的浮点提升情况，浮点就不太一样，浮点基本上每一个课题都会有一点点提升，这个提升明显是比 int 多很多的。这个例子可以看出来像叉264就是一些涉及到是什么编码之类的操作，它其实看起来不是一个很大的 loop，它就是一些很短的，比如说长度为16的长度为八的那一些小 vector 拼在一起。大家可能本来会觉得我写很多的循环或者写很多的 m copy 会对这个 SM D 会有所提升，但提升比较多的，反而是这种涉及到视频编码的一些一些。写过 close。

...(00:34:31): 总的来说，做自动向量化大概要做很多系统工具上的工作在编译器方面就是去实现，然后需要不同编译器之间需要约定 calling convention，比如说怎么传参，比如说返回值要放在哪谁会走站。谁会走计算器。除此之外，在 JIT，比如说 v8跟 JVM 是需要专门去做对应的修改的，当然他们不修改也行，就慢了一点，这个 GC?

...(00:35:07): 它的这个 DLTS 这个地方熟悉 GDG 的朋友们都知道，这是一个解析动态符号的地方，这个地方它涉及到一件事，就是它需要把所有寄存器都去保存一遍，就是保存到站上，然后再给它返回回来。如果你拿到一个不含向量计算器扩展的指令集的话，它的 GDP C 很有可能是没有做这件事情的，然后你的自动现代化程序就会在动态链接时候坏掉，因为向量计算器只要一解析动态符。符号，然后他就整个都记了下来计算器就会出现一些随机性的东西。就是因为在这个 GDP C 里面动态解析符号的时候，我们需要去存一下。计算器到站上所有寄存器，然后再给 reload 回来就是 GDP C 涉及到的操作。

...(00:36:19): 然后还有一个很重要的事情，就是在 kernel 里面需要做什么，比较重要的事情就是 alignment 有的处理器是并不能接受非对齐缓存的，包括前几天有我们另一个组的同学跟我们吐槽香山处理器的 RVV 好像是不支持非对齐缓存的。要从我们这里抄解决方案，还有一件事就是这个浮点，它会存在非规格话术，然后这个玩意会在运算的时候有可能会 trap 到 kernel 上去，然后它在 SM D 里面可能是不一

样的，那这就需要一些约定和一些 work 当然这些东西在我们做一个横向的时候。都是不知道的。需要去自己去，然后去猜哪里出了问题？

...(00:37:12): 当然，今天的工作主要是集中在第一个编译器的方面。编译器内部，主要就是做如何从向量标量的 AR 到向量 AR。做一个转换，就我们本来写的程序，它是标量的，但是我们现在需要把它表译成向量的，那这个转换怎么做到，那第二件事是我们假设已经生成了带向量的 IR。我们要怎么去生成 machine code 生成的方法是什么？用什么算法，我们今天会介绍这些细节。

...(00:37:48): 第一点，这个从从标量到向量 AR，它主要是 LV M 中会有已有的 pass 来解决这个问题。但是这个 pass 它不是乱来，它是由后端去给他一些信息的。那么第二点，这个向量 AR 怎么断 motion code 就是用这个 selection dag 这个算法去做的之后我们也会介绍这个算法特点是它是 BBY 的，它不能跨 BB，它是 BB 内部的。

...(00:38:19): 第二个是特点是它是有一堆 pattern 我们去匹配到这些 pattern 之后，再把它重写为另一些东西，然后一直不断重写，直到这个 dag 和我们的 target 的非常像那么。就可以生成 code。这个 overview 就展示了一下刚刚那个过程，那首先，我们从标量的 LV MIR 它过两个 pass。过两个 pass 的时候，会给一些类似于 cos model 给他某个指令是什么代价向量寄存器有多宽之类的信息，方便去他做这种转换。转换完事之后，生成的是 vector 的 LV MI。然后它会过一个指定选择，这指定选择之后也会讲，然后再过一系列的复杂的操作就生成了 as sembly object。当然，这些复杂的操作不是特别重要，因为主要的。主要的工作量都在 ICL 部分后面的东西基本上是你写这个 table 之后，它就会自己帮你生成出来。

...(00:39:32): 这个是一些基本概念。那刚刚提到 AR 有没有同学不知道 AR 是什么？举手我感受一下，没有听说过 AR 的。这个 module 就是从一个点 C 点 CPP 还有我画画删除掉，那大家能看清楚。可以从一个 create 生成。然后还有就是 function 这个 function，它是对应每个程序里面的函数。BB 是函数里面不带控制流的一个指令序列。整个一个 module，它就是由很多 function 组成的那一个 function，它就是由很多 BB 组成的。大家大概明白这个逻辑关系吧！

...(00:40:30): IR 就是中间表示。就是我们在源代码和生成汇编代码中间的一种一种东西。不是红外光谱，那么这个向量数据要怎么表示就是 IR 到 IR 中的一个一个转换过程，我们要怎么表示它就是向量?

...(00:40:56): 一个直观的想法就是假设基本数据类型是 int 跟 float，那我们怎么去表示向量的数据类型，直观想法就是给他们前面加一个数。比如说八成 float 跟四成。那么问题来了，八成和四成可以支持，那我要是程序是 N 成怎么办？这个 N 可能是一个变量，可能是程序中传进来的，有可能是每个 CPU 特有的常量。大家有大家可以猜一猜 N 城是怎么样的？是支持吗。拆成若干个八成？但是答案是这样的，答案是 LV M 只能部分的支持这个 N 乘某个玩意。就它可以支持一个 CPU，它这个 N 是一个常量的情况，就我运行时可以知道这个 N 是一个 CPU 的常量，但是它不能去处理每个循环，它这个 N 都会变的这件事情。不是很支持 RVV。

...(00:42:17): 这个模型其实对叉86或者你用这种指令集是非常有效的。因为叉86这个模型就是我们知道它的指令集扩展大概有 ssesser avx 这些东西它的特点都是定长的。大家懂什么是定场了吧？fx512是有一点点变长，但变的不多。new 是一个128位的纯定长的 SM D 扩展。

...(00:42:48): 它那个 SM D 扩展就是120，128位，你想放三想放整数或者想放浮点都行，但它一定是定长的，对于这种八层 float 或者四层是比较有效的。然后对 RVVSVE，这种比较新颖的指令集，支持就相对比较有限。以至于 LM 在生成 RVV 自动向量化的时候，它会先退化到这个进程的模型，然后再去生成进程的代码，然后其实不是很能去建模出 RVV 的这个特性。从从这个向量标量 IR 到向量 IR，我们可以看一个例子。这个例子就是我放这都能看清楚吧？那左边，这个是源代码，它的这个逻辑就是我这有 abcd4个四个数组。然后 a0等于这个 b0加 c0乘 d0减10，然后 a1等于 b1加 c1乘 d1减一。大家可以看到每一个下标上的这个东西都是完全一模一样的运算？我们其实可以很简单的想到我们把 B 的每一个元素 C 的某个元素 D 的都放到某一个向量寄存器上，然后这做一次减法，这做一次加法，然后再做一次乘法就完事了？完全没有必要这么一条一条去 load store 去搞这边这个东西，就是比如说我开一个 OE，然后杠 SE LV M 生成的玩意儿，它的这个逻辑就是 load load sub 然后 load。

...(00:44:50): Load load load 就是每一个指令都是运算的，按照源代码给的这个逻辑去算每一个操作数算出来之后再去加。当我们开了一个这玩意挡住了。开了一个 flslp vectorize 之后，它就能生成右边。

...(00:45:15): 带向量代码，我们今天不会去介绍 SLP 的细节是什么，我们现在只感兴趣为什么是四乘 i32，大家为什么不想它为什么是四成是它可以是八成吗？开始提问是例子可以是八乘 s32吗？对这个 CI Y 打的挺好的，就是多少乘多少，其实是跟向量指令集的宽度是有关的

。除此之外？有些指令可以的。什么副作用？因为因为我们的 source code 里面就会漏的八个东西，那这八个东西我们也可以一次漏的八成。这是没问题的。就源代码里我们就要缓存这个东西，那么是否可以假定缓存一定不会越界。那么是否可以假定漏的四成也可以？就是漏的四个，那么我就可以漏的四层。但不一定能漏了八成就是这个地方，因为它只漏掉了四个，那我去漏了八个就可能会缓存到外面的东西？

...(00:46:52): 除了这个东西是四乘多少有几结构的限制以外，还有一个限制，就是做加法，做减法，做这些东西可能是没有收益的。就是在有些比较拉的体系结构上它 SM D 的这个算法。它不一定比标量快。那么这就是我们待会要讲的，对每个指令需要一定的代价模型去衡量它。Ok.

...(00:47:28): 回到这个 PPT，我们现在已经知道 IR 到 IR 的变换是可以通过已有基础设施去完成的。所有的 target，比如说 RVV，比如说 on，比如说叉86，它都是共用的这个东西。那么有个问题就是这些 SM D 指令集之间差别很大。比如说 SSE 跟 RVV 差别就挺大的，它如何共用？LV M 给出的方案是提供一些信息来指导这个公共的 pass。当然，这个信息就不会是每个体系结构特别多细节，而是给出一些大概有多少信息，然后去完成这个任务。具体来说就是它会有一些虚函数。后端在实现的时候就涉及到去重写这些虚函数，比如说有的 cost 在这个体系结构，你可能是一，然后在那个体系结构，它的卖点就是个四？终端可能会感兴趣，什么问题，就是这些公共的 pass，他可能会感兴趣什么信息，大家可以猜一方面是向量有多长，就是他会问。你的体育结构向量有多长，是256位长还是128位长，这是一个很重要的信息，不然这个终端生成 IL 的时候它可能就超了。另外就是运算的这个代价。后端的实现，就是如果硬件支持的话，我就直接输出这个代价。如果后端不支持的话，后端就要诚实的告诉终端我做了什么，然后输出一个根据后端做转换，然后输出这个代价。

...(00:49:21): 当然，我刚刚说到诚实的，那么有没有一些后端是不诚实的，当然是有的，就是在实现一些草案的时候。这个东西并不是根据后端真正做了什么，而返回他的 cost，而是程序员认为后端做了什么，返回这个 cost。待会我们会讲一些认为的例子。

...(00:49:45): 那么既然提到 cost 那么什么是这个 cost，后端就是大家约定了有这么多种代价模型，一个是这个存储量倒数，一个是指令延迟，一个是指令产生的二进制的大小，还有一个是二进制的大小与延迟的加权和。有这么多种代价的类型，那么相当于我就要问后端你的加法指令的存储量导数是多少？你的加法指令的延迟是多少？你加法指令是多少？这个时候就有一个很有趣的事情。大家可以猜的是什么？什么唯一。它会根据不同的 CPU 的模型去返回这一值，信息是有的。他就只能估摸着返回一个书，因为这个代价是 LMIR 里的，就他就只有一个 A。那比如 CPU 有十个 a，它就只能根据这个 a 的某些操作数的信息去返回它的代价。这个事真是存在的，因为体系结构可能有不同的指令，但是 LBM 中只有一种 sha vector。那么有些 pattern 交错的奶体系结构可能是已经有的，但是随便一个 pattern，它可能就没有那么没有的那种 pattern 的代价肯定就会高一些，因为涉及到去拼一个 pattern。

...(00:51:25): 一件有趣的事情就是。虽然它提供了如此多种代价的类型，但是绝大多数后端对所有类型都会输出同一个数。虽然类型很多，但是其实没有很多人 care 这个类型到底是如何 work 那么就是我们首先设计上支持如此多的类型，但是实现上支持这么多太复杂的加法，就设一个一页。减法也是一乘法就多点二。基本上就是这样，比较随意的完成这件事情。众所周知，浮点是比较慢的，那就来个四或者八一般就这样写了。他是取决于一些实现的，他不会那不一定，那有人家有可能是没有 FPU？他可能要调库，这都是有可能的。我们需要编译器需要处理各种各样乱七八糟的 target 这是正常的。但是这件事情真的至少在 LBM 中。我写到的就哪怕是别人的 target 基本上都是如此处理的，因为这个类型实在是太多了，你要去算的话，其实根本是没有这么多 switch case 的，那没有办法我们就给他。都说是同一个数，简单一点。因为用户没有报你这个性能有问题，那就用？

...(00:53:11): 我们总结一下，刚刚提到 victor，首先这个提供信息，主要是通过 override 虚函数实现大家都知道虚函数是什么吧？Ok 然后这个信息主要是包括每个指令代价，然后计算相应计算器的长度跟数量。数量也是有一定作用，它会衡量这个寄存器的压力大不大。如果它太大的话，它有可能会放弃这个向量化。还有就是这个代价做的粗糙一点也没有问题，实际上大家都这么干的。就这个大模型没有很细。下面这个问题就是我们已经有了向量的 IR 我们要怎么生成？

...(00:54:00): 汇编或者这个 object 生成二进制 ok 这就是我们介绍的一个算法，这个算法就是 LM 中的指定选择算法。它的数据结构叫做 selection。同学们都知道什么是带。有像无环图，可能会在什么离散数学之类的课学到我现在就假定大家都知道什么是 D。哈哈。它的特点是在 BB 内，它是不包含控制流的。就是当代码走到这的时候，我们已经看不到函数，看不到控制流的转换，比如说 if branch，比如说这个 for loop 都是看不见的，所以说想在这个阶段做类似于循环向量化不可能的。

...(00:54:57): 拿到它的时候，它就已经只有 BB 内的一些数据流的过程。那么它的特点就是用每一个节点代表一个运算。这个节点，它会有

一些前驱，这个前驱就是操作数。待会给大家看一些图片来感性认识一下，因为这个挺重要的。然后算法就是很简单，就是去图上做一个 pattern match。Pass 然后给它 rewrite 成一些。我们比较后端会比较喜欢的结构，然后逐渐的把一些非法的操作变得合法。然后这样的话就能生成最后的指令。这个代表结构，大概就长这个样子，大家能看懂这是个什么程序吗？首先这是一个一，然后这是一个二。常量，然后常量做一个加法之后会得到一个新的数值，然后新的数值这里再做一个乘法。是不是很简单?

...(00:55:58): 感受一下。这种设计，在现在的各种别的编译器中挺常见，比如说可以自己给大家看一下 v8的这个 T。它也是，比如说是一个 function 是 fx。它要返回 X 加一，那么这个地方的节点就是啥，是一个 add 的结果的操作数是啥，是参数 X 和常数一。Ok. 除了表意这种简单的算术运算之外，它还会有一些别的需要表意的东西，比如说。我们知道是会有一些副作用的。前一个 store 和后一个 store 之间在这种结构上是看不出来区别的。还有一个问题比如说 VI 和 F 都是。我们怎么区分它到底是什么样的？这里只有个号的话，你就可能不知道它到底是什么。还有一个问题就是控制流。控制流刚刚已经提到它基本上是不能支持的。

...(00:57:20): 首先讲这个 load store 要怎么办？它的解决方案是我们返程的时候多接触一个参，大家听懂为什么 load store 是不能表意的吗？我可以画一下。假设现在程序是这个 store 这个 PTR 答案是 P，然后我们往 P 这个地方存一个一在。迟到 P 这个位置再放一个二它画成图可能就长这个样子。是 store 节点？它有一个参数是 P。然后它另一个参数是一下面是第二个 store 节点？还有一个参数是二还有另一个参数是 P。那么现在问题来了，假设你只能看到右边的结构，哪个 store 会先发生哪个 store 会后发生？就看不出来了。我们把这个 store 放到前面 store 放到后面，这个时候放或者调换它们的顺序，在这个图上面都是可以的。因为因为这个 store 依赖是 P 这个 store 依赖是二，这个时候也只依赖 P 和一，它们之间是没有一个顺序关系的。那么持续这个顺序关系？我们如果把这两个东西的顺序给调换一下。那么同一个内存的 P，它就可能会导致调换之后，本来我们期望它留下是二。但是调完之后，这个地方就被写了个一进去，那程序就错了。成绩错，可是天大的事情。你做再多的性能程序，错的话都记。

...(00:59:21): 这个问题要怎么解决？大家有没有什么想法？怎么加。我们对所有的 store 都要加我们对所有的 load store 都要加一个边吗？如果我确认他之间是没有依赖他就可以不加对吗？要怎么加边？就是这是第二个 store，那我们就给它连一个 B 连一个虚线这样。那然后这样就表示它是依赖。对那真实的答案是这样的，就是我们在缓存的时候，会多接受一个参数。这个参数代表上进来的 memory 状态。然后每一个节点，它都会吐出去一个 memory 的这个状态是一个抽象，你别管我是什么状态，反正我就需要一个这样的参数。对于所有的我们都要让这个参数是存在的。目前解决方案是这样!

...(01:00:48): 解决 VI 的跟 FI 的，这会让我们想到一个方法，就是我们对所有的这些数据类型都带一个 T。这样的话，比如说 VI 的跟 FI 的，我们就能通过它是 V 什么什么什么还是 F 什么什么来区分。

...(01:01:05): 然后控制流，它是不能支持一些各种复杂控制流的，前面也讲过了。比如说这个爱的这个东西，他就是在真正的 LV MR 中就长这个样子，我们可以看到这加了一个 typei32。这个 type 是 i32，然后这个地方最后做个加法，第一个操作是这个玩意儿。第二个操作是这个玩意儿。那刚刚讲这个 load store 的 store1和 store2最后要怎么解决，这就要引入一个概念叫做 chain。他的这样，我对所有的。所有的内需要缓存的东西，我都需要接受一个东西就是特殊的内存状态，每个程序开头都会有一个。然后这个 store，它因为在前面，所以说它要先把内存状态作为依赖，它会吐一个状态。我们给它叫 s1！然后这个 store 它就会依赖于这个状态。所有的 store 都必须多接受一个参数。再多吐一个参数熟悉的人大概就会瞬间想到这和某些 mona 的之类的概念其实挺像，当我们需要。需要解决一些副作用的时候，我们就把这个副作用当成参数，传给这个函数，这样就行了。但是这意味着你你多接受一个函数，然后再吐吐一多接受一个参数，再吐一个返回值回来的话，就好像你吃了饭总得给它拉出来一样。

...(01:02:46): 所有的 load store 都需要去多增加一个这样的参数。就比如说刚刚这个二？刚刚在真正的 selection 战略中就长这个样子。我们注意到这个 store 就放到了这个地方，这个时候放到这个地方，它有一条蓝色虚线连到这个 entry。

...(01:03:38): 就让这个拖过来，对一个摄像头看好的，那没事，我们先讲这个吧，就是第2个 storestore1然后这是第1个 store 第二 store store，这个是第一个 store store 一。

...(01:03:56): 我们可以看到蓝色这条线就是我们感兴趣的东西，它会多接受一个参数，从这个地方连一条蓝色虚线，然后这个地方再连一个蓝色虚线连到第二个 store 这样的话我们就知道它的相对顺序是什么。当然，不是所有的节点都需要接受一个蓝色的，这样特殊的线，只需要对那些有副作用的进行处理就可以了。这是因为我们需要让它之间有一定的，我们可以预测的顺序。不然的话处理这种带副作用的节

点就会出问题。这个设计我们可以看一下谷歌的这个 v8是怎么搞的，他们相当于说是有一个。非常明确的 start 跟 end 所有的节点，他可能就会接受。除此之外，还有因为 javascript 是有异常。所以为了去异常这个东西，它还需要一个另外蓝色的线去连关系。但是基本上所有的编译器 IR 都会有类似这样的设计。就是我们去多加一个代表副作用的节点去建模程序的设计副作用的，比如内存访问之间的指令的一个依赖关系。

...(01:05:26): 那么下面就介绍一下指定选择的过程。这个指令选择的过程，主要是合法化需要介绍一下 deck to deck is 基本上是。可以自动生成的。那么现在大家就比较关心如何合法化这个 T。然后另一个是合法化这个 operation。type 就是说可能有人来的是一个长度为256位的整数，你要怎么处理？还有一个 operation 就是说。我们的体系结构，它只支持加减法，不支持乘法，你是不是需要一些别的玩意来模拟一下？

...(01:06:08): 先看一下如何合法化这个 type。这个依据是后端会给定一系列的，它支持什么 type，比如说支持 I 32，支持 I 64就是32位的整数跟64位的整数。可以支持。就类似这样的东西。或者说比如说 v8i32长度为八个的 SSR 每个另一项都是一个 SSR。他给出了这些支持和 type 之后，它就会自动完成类似于 promote expand 这类的操作。promote 把。把 i8变成 i32，就是把它的这个数据类型变大。expand 就是说假设我们是16层 i8，但放不下了，我们可能会拆成两个8层 I 八之类的操作。这些东西都是后端会有一个框架，所有 target 都统一完成的。那么现在开始提问这个 type 跟 OP 是否相关，就我们合法化 type 类型这个合法化过程中，跟它的涉及到操作是相关的。无关的。我们如果要假定他无关的话，我们就必须假定这个体系结构要支持 i32的话，它就支持 I 三的乘法，除法等等等所有的操作。我们要认为它是有关的话，我们就需要对每一个 OP 跟 type 都做一些代码操作可能会让程序变复杂。

...(01:07:49): 如果你是来设计编译器的人，你会让合法化 type 的过程跟 OP 相关吗？我们刚刚介绍有合法 type 跟合法 OP。就只有这两个？先不说我的 S。那么我们就说正常一下行吧！这个 OP 是否是跟这个 type 的合法化是否跟 OP 相关的，比如说我们要认为 i32合法的还是认为成逗号 i32合法？我们 i32就永远的合法了？

...(01:08:54): C YY 你讲一下！你觉得这个跟 OP 是否是相关无奖竞猜。是相关的。答案是这样的，答案是在现在这个状态的确不相关了，然后他的确会在下一个 stage 处理，那么这样的话就会带来一些问题，就是某些 as，它的这个知识是相关的。就会很恶心，这个具体的方法是在 LM 代码中编写叫 add register class 这样的东西，然后你就能通知他到底有什么样的类型是合法的。你只能做这件事情就是告诉他什么是合法的。然后具体跟某个 type 是合法，它是这个过程中是不能救你。那么合法 type 的一些例子，就比如说 i31有的人编译程序，他比较逆天，他就喜欢用31位的整数，那么就只需要。有什么问题吗？

...(01:10:02): 他就喜欢用31位的整数，这个可能也是可能会生成 i7这样的玩意的。那么对于这种例子，我们就可能会给它放到 s2里面。比如说四乘 i32，要是这个体结构是 BPF 这样的比较拉的东西的话。他怎么办？终端已经生成四乘32，但是你你他给的是个 BPF，那我们就把四乘32拆成四个 i32。这个就像这样，这是一个节点，它是 add。然后四乘 i32，那么我们在如果这个 target 是不支持四乘三的，我们就把这个拆成。四个 A。大家可以想象一下。听懂这个过程了吗？就是匹配到了四乘 s32这样的非法节点的时候，我们就给它拆成四个不一样的节点。然后每个节点都是标量，这样的话它就能处理的，比如 BP。Ok. 然后比如说16乘八，它可能在某一体系结构上，它只能支持八乘32。我们就需要先把16乘 a8给它变成16乘32，就我们可以用更大的数据类型去表示一个更小的。那但是16乘 s2这玩意瞬间就放不下了，那怎么办，那就只能再把向量又拆成两个。就是我们可以组合拆的过程跟 promote 的过程。

...(01:11:39): 同时的去 promote1下再拆一下，这也是可以的。下面来看合法化 OP。合法的 OP，它的对象是 OP 跟 type 元组。为什么它对象是合法化 OP 跟 type 元组，每个 operation 它都有一个 type。Operation 可能对特定的 type 是有效，但对某些 type 就是没有效果的。我们去做这件事情的时候，永远都是去操作 OP 跟 type 这个。原组的。那么对于每个 OP 跟 type 后端需要决定他要做 expand 和 customer legal。那大家可以猜一下 expand custom lego 都是什么意思？首先，expand 就是四乘 s2拆成四个 s32，就类似于刚刚的过程。就是请求 LM 的某个流程，把这个向量的操作给它变成一些标量操作，然后我们假定标量操作是支持的，就是说。要不你别帮我写，我要自己来自力更生写 C 加加这个 lego 这个体系结构有非常好的对应，比如说 a 的这个操作在很多体系结构是有这个指令的，那么就它就很完美的对应了这件事情，我们就可以说明它是 lego 的。Ok.

...(01:13:22): 下面来做一些小练习，想必大家已经听过了，听懂了李范跟 lego 现在假设某二支持向量指令有这么一些。整数的加法和减法是可以的，然后向量移位，但移位量必须常数不能是一位一位一个不同的长度。还有你可以做被运算。还有就是布置非对齐放。那么我们如何去合法和艾特和不讲竞猜。没错，现在我们如何实现这个 SRA，大家知道 SR 是什么东西向向右算数一位。对所以说应该选哪个？你

只能三选一？你编写一个 C 加加代码，你检查一下它的操作是不是常数的话，我们就不动了，就让它这样继续下去就行了。如果是常数的话，你赶快把它 expand 了一下？下面再看一个乘法怎么办？不对 expand。因为你看这个限量指令是没有乘法的。我只说了加法和减法，他没有说乘法。是不能模拟的，因为这个乘法的操作数可能是一个变的。然后你你去搞一个加法去模拟的话，是会比较困难的。是可以做的，没有必要，很可能不如标量。因为 explain 对那标量是支持乘法的一般体结构都是支持标量乘法？Ok.

...(01:15:27): 再下面来看下一个 load store 怎么办。就必须因为因为它不支持非对齐缓存，那我们就需要看一下它是否是对齐的话，就让它继续下去。如果不对齐的话，可能就需要做一些操作，比如生成 copy。B swap 字节交换把大端序变成小程序怎么办。就变成标量了。也可以。就可以凑出来，所以我们就可以用给它凑一个出来，因为绝大多数现在能凑出来的东西都是慢于都是快于半成标量的。

...(01:16:23): 还有一个是 ABS 求绝对值。大家感受一下应该用什么，就是要么就变成标量，要么就凑一下，要么就你肯定自杀不行。是能凑出来的待会我们会讲怎么凑挺有意思的。有什么问题，行，这个 load store 刚刚已经讲过了，我们会根据它的这个 point 它的操作数是怎么个对齐的？这个对齐信息是包含在它的操作数里面的，我们就要决定他要不要拆成两个，比如说 mips 里面就会有 L DLL DR 这种指令给它拆成两个。如果当然 LDLDR 也是有对其要求的。如果还不支持的话，就需要来个 M cop。能理解吧，就是我们先把 M copy 把站上某个不对齐的地方 copy 到一个对齐的地方，然后再从对齐的地方开始漏的，这就非常慢。

...(01:17:30): 这个地址对齐是从前端也是从 GCC 这种玩意一路的分析出来，然后从终端传到后端传到这个位置的。它必须要分析正确传递正确。之所以这样说信息之后会给大家看一个传错了。哈哈。就单靠谱。不是调库是不是 intrinsic 就是 LV M 中去生成 load store 去做 mem copy 这件事情，但不是生成一个 call me copy。这肯定是不行，比如说我们要漏的一个不对齐的 i32，那你可能就需要四个 i8的。漏的就是你要漏了一个32位的整数，但它不对齐，我们能够漏的一个字节，然后把每一个字节的 load 出来，再把每个字节都 store 的对齐的位置，就可以 load。这样的所有体积都是这样的。如果能做到的话，当然是可以的，但是默认的框架是 load。就是它有一个通用框架，如果你的体系结构没有写自己的 C 的话，那么就会走这个框架就是 road 加 store，它非常通用。对一些短平快的开发，一开始都是这样的。现在来看这是 C。不是它是生成 bag 上的 loader 跟 store 就是我们是在这个数据结构上操作，它已经不是 LV MIR 了。明白吗。所以我们会在这生成节点。它不是 LMI。

...(01:19:33): 绝大多数优化都在终端后面还会有一些 machine ssa 之类的骚操作是可以优化。但是大家一般都不在那做。Ok. 我们继续讲就是这是 chrome 的一个需求。那么它其实是可以通过1个移位和 or 拼出来的，比如说我们需要把 a0a 一 a2a3它是四个 B，那我们可以移位成这个样子。有成这个样子之后再给它握起来，这是能做到的，就是在向量里面拼一下。下面来看求绝对值。它可以先把。这个 X 给它算术又移它的 size 减一这么多位。然后假设这个是不是 T，然后求 X 的绝对值，就等于把这个 T 和 X 加起来再求 X。是不是很神奇，真人求绝对是。大家可以想一下为什么？T 是一个 T 把符号位移到一个整个上，这个爱的 T 就是假设它是个负数，我们就给它弄个负一。它在一下这个玩意就会变成它的正的版本就是补码小小骚操作。这个例子就是比如说八位的整数负五，它等于这个玩意，那 T 就是负一。那它把它加起来就等于这个玩意在以下这个负一就写错了，这最后会抑或出来这个零。抑或衰就会变成正误?

...(01:21:30): Ok. 然后现在是一个怎样选择的小结，它的整个算法结构就是刚刚画过的 selection bag，它是一个有向无环图。对于它的整个的指定选择的过程就是带上面的这个 rewrite。涉及到 type 上的 rewrite 跟 OP 上的 rewrite。OP 的时候，是 OP 的时候，是一个元组，就是我们在 rewrite op 的时候上也会关注它 type。type 是非法的话也会做一些事情。

...(01:22:07): 下面来介绍指令选择中的一些趣事。放松一下，大家听听一些比较严肃的数据结构与算法，我们现在来讲一些有趣的事情。就是前端的对齐信息，我怎么感觉是他瞎编的，怎么感觉错了。第二个是这个 SM D 居然不支持整数乘法。那怎么办？然后是这个 new 是太天顶星了，这谁想出来的，然后还有一个就是别人全 legal 我全得 expand。然后这 i32跟 a8都是 I，我们可以倒腾一下，下面来介绍一些趣事，到底怎么个去法就是首先是前端这个对接信息，为什么说它是瞎编？就是我们先介绍一下背景，就是很多体系结构会支持单元素对齐的这个漏的就是我们可以给它拆成两个，那拆成两个，前提是你至少要对齐到 SM D 的一个元素，那么多对齐。这是合理的要求。为了优化，我在编程的时候就假定了 double 至少对齐到八。这是合理的假定，因为。

...(01:23:23): 怎么会有人把 double 放到一个不对齐的地方？我真的拿到了前端给我的东西，他说 double 是一。你不觉得很神奇吗？就是这个 double 怎么他怎么可能是 online1的玩意，后端就给我生成了 M cop。因为这个。

...(01:23:50): 到底是啥问题，我先给大家看一下这个症状如何，我现在还留着证据。fl nvidia 的编译器。他的这个代码都能看清楚。上面是 f

ortune 什么。上面这个玩意是 fortune 的源代码。能看见。这个源代码就是 complex complex 就是复数是两个 double，我们看1下它吐出 LV MI 它在这个 entry 这个地方 a local 一个对齐到16的两个 double**太厉害了，竟然能对齐到16。它下面就开始 load 了 load 中括号的尖括号 d ouble double ptra。这明显是个 bug，就是你看这个程序。你你你看常数不是这个程序 a local 明明是个 a16的东西，然后就紧接着它下面几行就开始 online1了。这个 F 浪多少是有点问题了，我觉得在下面说。这个时候发生的事情，我就说。could you please double check why 来这个东西是一？他说这个 allocated 是16，然后为什么就对齐到一了，他说什么你能不能给我搞个 reproducer？我说你这 reproduce 就在上面评论里面。下面开始解解释是为什么就是这个 LMR 的 verify，它会添加一个一如果这个前端是没有生成这个 alignment 的。然后这个问题就是这个 fl2，这个 fr2在计算这个玩意的时候给漏了，然后他说他将要提一个 PR 来 fix。我们来看一下这个 PR。

...(01:26:02): 这个 PR 大概就是说这个原先的写法是某个特殊的曾经的某个需求给加上去的，它只会在操作数为 complex 的时候是会有问题的。你要是不处理复数的话，它就是对的。但凡处理到复数，它就会进入一个奇怪的三目表示，这个 flag 就错了，然后必须要调它的某个 U 来，才能把它给搞对。

...(01:26:37): 这里也要吐槽一下。很多编译器前端包括 clone 跟 RC，它生成 LI 都不是生成文本。它是生成那个 a st1在代码中就是数据结构是一个树一样的玩意。但是 CF R 就是我们现在给大家看的这个玩意，它生成是文本。文本 A 就是他先吐一个文本到你的 temp 下面，然后再由 LV LLC 来读那个。这就会出一些问题，比如说。他会 LINK LM 中的一些库。它为了某种神奇的原因就会生成 text。不知道，反正 f2已经被那个 nvidia 给弃用了，大家现在说的都是什么，lv N project 里面那个 NEW F 就是 FL 为什么会有两个原因，可能就是因为。

...(01:27:46): 对 nvidia 那个 F这个 FL 是类似于 C 语言的 C 加加写的。这肯定就不太符合 LV M 的标准，并且它的这个技术路线比较奇怪，生成了一个。文本 IR 然后再让 LM 去 pass。比较逆天。当我们发现这个问题之后，我们肯定是不会报给甲方的，因为如果我们不报给甲方的话，我们把这个玩意修了，这就可以算是我们 SM B 的加速比。这个是一个重大的 bug，可以直接导致浮点慢非常多，因为里面有非常多的 complex 这个问题就是这个 CF 会先生成 IR 字符串吐到一个文件里，然后在 MP。这个对齐的信息就丢了，他就没打出来，然后 L M 在 parse 的时候就自动补了个一。这个是我们看到的东西。对就是这就是因为他这个的时候压根就没有生成是多少。我们重新 purse 的时候，这个 all 就变成了一。它是有的数据结构里面是有的，但是它是因为一些历史原因，然后对这个 complex 的处理是漏了，你看它这个修复就是这样。它的别的类型是能生成正确的 alignment，但是 complex 好像有某种奇妙的处理，然后就漏了。也不知道他写的是什么玩意。

...(01:29:38): Ok. 解决前端对齐这个问题，现在来吐槽，第二个就是这个 SM D 它居然不支持整数乘法，那怎么办？其实这个在别的这别的体系结构里面也是存在的，比如说叉86的 CMD，它在一开始的时候就没有乘法。它虽然有这个 PM 什么 U DQ，但是它跟 LV M 中想要的那个 SM B 乘法是不太一样的。他会把两个64乘出来就是一个。I128我们希望两个64相乘也等于两个64就跟普通的整数乘法是一样的。他这样搞的话，就我们每次弄完就需要一下，那相当于高的那个64就没有什么用。他如果不支持整数乘法的话，有些的这个 workload 它的确会涉及到 SM D 乘法，这就会导致 ISEL 或者 cos model 写的很复杂，因为。如果支持乘法的话，那你前面那个 cos 就会比较慢，当然我们前面已经讲过 cos 是比较不太精确的。那么为了省事，cos model 甚至会他说 SM D 乘法代价是99824353。

...(01:30:57): 我随便搞一个比较大的数，终端很可能就不会向量化了，那我就不用支持了。哪个书。它只是一个我随便写的大数字而已。就实际编程的时候，可能是连滚键盘出来的目的就是老师说你这个乘法不行，你别让它生成。那有什么办法就先搞个这种事情，在别的体系结构中，时而也会遇见一般因为开源大家也就休了，但是在一些闭源的神秘架构中，就可能会发生。因为前面讲的 cos model 并不是你做了什么，然后按照你做了什么来生成的，而是你告诉终端你做了什么，当然我就可以不诚实的说，我觉得代价就很大。

...(01:31:50): 我说的不是叉86。下面是这个趋势是我觉得你用真是个非常厉害的指令集，就是它这个指令集，虽然只有128位，但是它加速比是比叉86还高的。这是为什么，因为它的这个指令。它的支持特别完备，基本上你想要什么就有什么，比如说乘法它是有的，并且它对特定 pattern shuffle 是有特殊支持的。我说的不是叉86。还有一个特点就是 OP 跟 type 是正交的。就你只要支持了这个 type，那他就支持所有可以支持的 OP。而不是说部分的 type 跟部分的 OP 就搞得编辑器很难受，比如说他要支持 i32加法，他就一定会支持 a 八加法。

...(01:32:48): 比如说这个叉64中的 SATD 是一个比较热的函数。我们看一下类型，它的 pixel 是 U。然后它的别的数据类型是16或32。它的整个比较就计算量比较密集，就是算 a0a1a2a3，然后它的逻辑看起来就能被向量化。这看起来这东西就很规整。下面就是做了一个哈德玛的四，这个哈德玛四大概是说做了1个类似于位置交换，就是 s0加 s1s0又减 S 一 s2加 s3s2又减 s3。这样的东西。最后 D 又等于 t0加 t2t0减 t2t1加 t3t1又减 t3，那大家可以发现。看代码可能比较复杂，我这里有一个图来展示整个过程。就是 pixel1假设有这么几个元素，那么它刚

刚的 temp0需要这样的移位才能达到。在这样的移位之后需要求个 sum，并且还要求绝对值，当然刚才已经讲过求绝对值其实并不复杂，可以拼出来，那么我们现在就比较关心如何 SM D 上面这一块这个过程。

...(01:34:08): 这个 function 大概占18%的运行时间，只要优化这个 function，它就会跑的比较快。我们来看一下 new 大致是什么？这个 arm 的文档，他有专门的一个目录里面就有一个东西叫做 permutation，然后这个上面是大概就是讲的是我们要如何去。

...(01:34:34): 去这个 vector 它的支持什么，它支持这样的 C 把 a 换成 AB 换这 C 换成这 D 换这，然后给他下一下。他也支持这样的 C。就 a 换到这个地方，然后 B 换这 C 换这 D 换在这，然后 E 又放在这，然后 F 又放到这。他又支持这样的 S。a 放这 B 放这 C 放这。它还支持这样的 S 就是 a 放到很远这个地方 a，然后 H 放到这个地方 H 能看懂吧？他还支持这样的下。这样的家伙能看懂吗？就是把它的偶数位放到这儿，然后偶数位放到这个地方。然后偶数位一下放在这儿，然后偶数位放在。他是为了模拟这个矩阵的转制这个过程。然后大家可以看到它的这个各种各样的虾头是非常全的。

...(01:35:56): 他的有人评论看一下。转一下摄像头。大家可以看到我们面对这个需求就跟他的这个 shuffle 能够完全对应上。不得不怀疑他设计这个指定企业的时候是有考虑到这种需求的。那么。所以说他就会。为什么叉64能够在 arm 上有特别高的加速比，但是可能别的体系结构就没有这么好。

...(01:36:37): 现在来讲一些 future work 就是一些不太存在的东西。第一个是 global il，这个 global il 是说现在 ICL 是 PER BB 的，它会丢失一些信息。另外一个问题是，in combine combine 会重复代码 in combine 就是说 LV M 终端做的这个 combine combine 变成 selection 之后，它要做一个 combine。它会重复很多一模一样的代码。那么 global icl，它就是说我们全局的含 BB 的进行指定选择这个玩意的。未来可期现在写的还不错。A R 和64跟 R V 的后端还不错。信创后端其实是不太行的。我记得我去年大概。不算特别信创英国信创去年大概这个时候我给 BPF 交了一个 group iel，然后到现在还没开始写。就是交了一个 INIT 代码，但是其实还没写。然后第二个事是这个 RVVSVE，这框架表意是比较困难的。什么。这个地方就变成 future work。因为看起来就比刚刚那个要遥远。然后这个玩意在 MIR 中是有一些实践的，但我觉得做的也一般吧。感觉其实不如手写，因为像 M copy 这种玩意，它其实可以通过手写汇编，就不需要编辑器这种像话像叉64这样的东西。

...(01:38:24): 像 RV 那种变长，其实也不太适合，刚刚大家看到叉264？就讲完了，下面今天讲的一个总结。这个向量化，这个收益是很看 workload，这是第一点。比如说这个 arm 就在叉表现特别好。但是，像别的体系结构或者别的 workload 就表现不行。然后第二个是标量 A R 到向量 AR 我们只需要指导已有的框架就行，主要指导方式就是给他提供一个 cos。这 cos 可以瞎编。第三个是向量 IR 到具体的指令集。然后靠的主要是 select 上的花式操作。还有一个就是增强向量寄存器的长度。不如增加点真正的指令，增加点操作。变长没有什么用。多操作才有用。未来的工作就是这个 RVVSVE。可能还得加油干。大概就没有了。下面提问时间。

...(01:39:45): 哪个地方就是那个地方？我刚才有讲到 cos model 是很粗糙的，那个 loop vectorizer 是会生成特别长的代码，那么我们很多时候能看出这个 loop vectorizer 在这个循环的长度很短的时候，它就有负优化的。主要还是因为 arm 非常多，它不一定适合鲲鹏920。

...(01:40:43): 你说在正经的开源开发中怎么定还是在?

...(01:40:58): 其实是很难衡量的，因为在座也有做处理器的，大家知道这个处理器的每一条指令的 cost 其实你是不太好衡量的，所以一般来说指令集，做 CPU 的会给你一个手册，这个手册会涉及到那个指令它的 latency 是多少，这个数是有的。然后很多 cos model 是参考 latency 来做的。我看一下这有这个地方就是其实很多 cos model 是参考指令延迟做的。它会根据这个指令，从从开始算到有结果会有多少 latency，然后就把这个值来当做一个 C。

...(01:42:01): 一般不会做 normalization。你做的我要怎么写9982435？这个东西，比如说在后端，在实现的时候，它会说我的乘法指令是多少，那么四乘这个乘法是多少，它就会递归的掉自己。那其实这个过程你是比较难帮得住一个范围的。他就是把这个递归掉自己，然后加起来。他挺 work 的。

...(01:43:00): 有很多不正交。

...(01:43:06): 因为你在做合法和 OP 的时候，你就会涉及到这个 T。比如说你对乘法的 i32是 lego，你可以设乘法的 v8s32就是。是可以的。对它主要是为了方便程序员编程。因为我们可以假定绝大多数都是正交的，但是有部分比较奇怪的东西不正交。比如说在前面那个过程，你就可以把 i7i31这种玩意给处理掉，然后把 type 变成某种比较好看的样子，不然的话这个圆组就会非常大。就比如说 OP 你处理了 i32，

那你要不要处理一下 i31？在前面不做这个事情，那这个地方就会写非常多的 case。现在那个 global iel 基本上就这样写的，他会说 legal for 巴拉 type，然后巴拉 type 它会比现在这个框架要优雅一些。

...(01:44:29): 还有什么问题吗？展开讲一下怎么做，就是我们没有做。但是大概思路是这样的，就是我们知道内向的假设它只能处理 i32加法，那我其实可以只用一半来放一个 s6。然后我们现在来做加 s6的这个 SM D 加法，它是不会溢出到下一个 line 的。大概就是这样的思路。当然大家发现这个加速比其实不做这件事情就够了。做这件事情在 LM 中会比较抽象是不得已而为之的做法。就是就实际上我们做 SM D 的时候就怕的是这个另加法加到下一个令上去。这样他就错了，但是如果我们支持 i32加法，我只给他的第16位填数字，然后他做加法是不会错的。但这就很危险，听起来就不太适合刚刚提到这个框架，所以说一般人也不会这样做，这只是前期一直没有加速比的时候想着骚操作。最后也没有这样做。

...(01:45:44): 还有什么问题吗？基本上是他会一开始就求出这个我翻一下。我看看在哪？在这个地方提到一个 V scale，这个 V scale 是对每个 CPU 的产量。那么它在做 RVV 的时候，就会用一个指令去求它的 VL。然后求到这个长度之后，它就会一直用这个长度。他不会真的用 RVV 那个 VVL 来动态的决定长度。他直接去读 VB，就根据这个 VB 去做，剩下的 code 他不会去每个循环用不同的 VVVL。目前的情况就是这样。就是 RVV 你要手写汇编那种很优化的话，它就是会变的，但 LV M 它就生成 code 就比较笨，它不会发生这种事情。比如它 VC 的 VL 时候我们手写汇编会传入这个数组的长度。但是 LM 在 VC 和 VL 的时候，它会把那个数组长度传一个，比如说零号寄存器，然后这样的话，它就会选择尽量长是这样写的。如果你不知道这个长度，你就传一个零进去，我就会选一个尽量长的长度，那就很符合它现在这个模型。长度的确也不会变。

...(01:47:49): 这个事情其实 RVV 也想做类似于我们把这个东西给它做上去，但是 LBM 社区在吵架，你只有 rvv1家这样的。我要做的终端的话，别人都不适配了，就一直合不上去。

...(01:48:19): 对。它对 code size 有优化，对程序的性能不好说有没有优化，RV 现在有规吗？他感觉是有优化的，当然我没有测过。因为他支持不了。类似于当 CD 用的它不会用变长的。也没有白写。

...(01:49:24): 有请香山开发者解答。既然现在的威尔，所以就不做它的性能了。没有一个。L 里面有加了一个什么玩意儿，但是反正 in progress 你也不知道他就会写多久。

...(01:50:07): 对还有比如说它生成这个 V 版本，它会在后面吐一个标量的循环对，但是我们用变成的话，那标准选项就可以省掉。那还有什么问题吗?

...(01:50:38): 对怎么。

...(01:50:55): 对是他不会在 BB？是的。对。对。

...(01:52:20): 对的确是可以的，这个问题在 LV M 中是这样的！这就这给大家都讲一下不就是 LM 的这个 BB，它的 BB 内部其实是有一个顺序的，就是这 BB 内部它的指令是123456。我们其实是假定这个指令是一条一条执行的，它这个顺序是严格的。而不是说。

...(01:53:29): 我觉这个舒服，我就用这个。行。Ok，我们在 LV MIR 中，它虽然就是某些指令是没有依赖的，但是我们在 code 的时候，我们就生成一个链表，然后这个链表在生成的时候，它就决定了这个顺序。

...(01:53:56): 这个顺序是在我们生成那个 code selection dag 的时候是不能去动它的。就是所有的 store 和 load 之间，它就天然的有一个顺序，而不是说根据指针分析来决定它到底需不需要顺序。而是在 IR 中，它就已经有一个顺序，这个顺序就是 list 是全序的。然后这个序，你就必须去遵守它。就是 IR 中断的时候就不太好。所以后面的就他就会尝试解决这个问题，就是你刚刚提到 LV M 的这个 BB 在中断的时候就必须是完全有序的。

...(01:54:36): 还有什么问题吗？也需要的。你说但是你问题你不要快的死的。如果可以去他最需要的就是它其实有足够的。我会做绝对！然后还有一个问题就是可能漏的会有 exception。然后如果交换顺序，它 except exception 这地方可能是有问题的。就会有一些妙妙影响。就是我们在编 chrome 的时候上也遇到一些类似这样的问题，就我们生成 store 的时候，恰好生成到某个页边界上，然后它就出它的大。对。

...(01:55:39): 还有什么问题吗？就其实这样就是假设这是一个配置，然后这是下一个配置。然后下一个配置是不可读的。那么它现在要非对齐缓存这个位置？把飞起放在这个位置。我可能就会拆出两条漏的，因为编辑器不知道它到底那个漏的有没有对齐，我可能就会拆漏的跟

这然后还漏的这。明白吗。就是它实际上漏的是这个位置，我不知道它对齐了。那我拆成两条路的时候就会拆成一个路的，在这一个路的在这那第二个漏的实际上正确的体系结构会把它解释成一个 L。就 LDR 会是一个 not，我们的体系结构？它是恰好对齐的话，按照语义来说，它是一个 L。他不有问题。因为我们 LDR 的话相当于你这个东西就漏的东西就不缓存。没有什么方法。我觉得你说的可能是对的，我没有试过。因为它在 IR 设计上的时候就会有那条边，所以你不可能交换所有的位置。是快到了。还有什么问题吗?

...(01:57:42): 要不要开一下投影?

...(01:57:46): 线上我刚看了好像没有人提问对。我把评论给关了就不想。

...(01:58:14): 我们可以看到前面那个有个系统软件的地方跑哪去有一个系统软件有个 DLS 这个有没有谁能给我解释一下？他为什么会出问题反问观众，就这个 GBC 它会在？什么 GDP 当然是标量的。没开。它用了它是?

...(01:58:51): 但是 GDC 的这个 DLL 为什么会就是它会涉及到保存所有寄存器，就在别的体系结构上也看到是它会把所有寄存器都 save1遍再漏出来。就为什么他有没有可能是遵守什么？你说它涉及到函数的时候，它可能就坏了。我为什么不能在外面就是我什么的，那他也不会坏？我要遵守什么 calling 的话，感觉也不会有问题。为什么。

...(01:59:51): 这个 DL 是在 resolve 的时候涉及到的操作就是什么 PLT 一刚开始进刚刚进去的时候。他会 resolve 符号到底是啥？就感觉不会有的体系结构是会的就是它的浮点跟向量是共用的话，那你去改浮点可能会动到向量。对，就是你动浮点低位，它可能给你把向量高位给搞坏。如它的浮顶跟向量是共用的话。反正这的确是我们曾经遇到过一个问题。他就会在他要是不去故意保存相应寄存器的话。他后面向量计算器就坏了。这个事情只能去通知 G。你编译器也改不了?

...(02:01:17): Ok 大家还有什么问题吗？ec1下大美。这聊天有人说话。sm da bi. 不是很懂。哈哈。

...(02:01:53): 他说，你必须的。这个 API 反正等我拿到这个东西的时候，他已经存在了，我已经动不了，就我有一个手册，他告诉我这些东西需要存那些东西，不需要存传开走哪个。因为这种东西是一个约定。就是因为比如说人家开发 GCC 的，也需要去遵守这个他可能不会跟你去同步，这些约定是什么？还有什么问题吗?

...(02:02:35): 接近吃饭。

...(02:02:50): Ok 那我们今天就结束吧！感觉不会有人问。