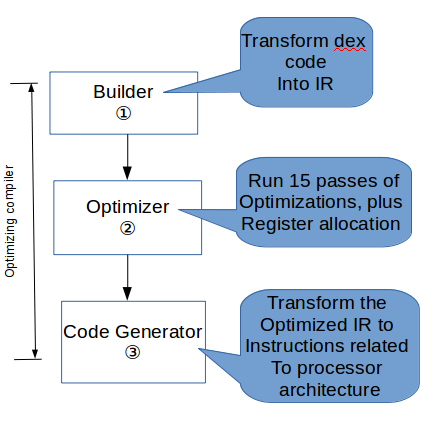
技术文档



上图为optimizing compiler的概要流程图，编译器主要进行以上三步工作。

其中：

①builder负责将dex字节码转换为中间表达形式HIR(high level intermediate representation)；

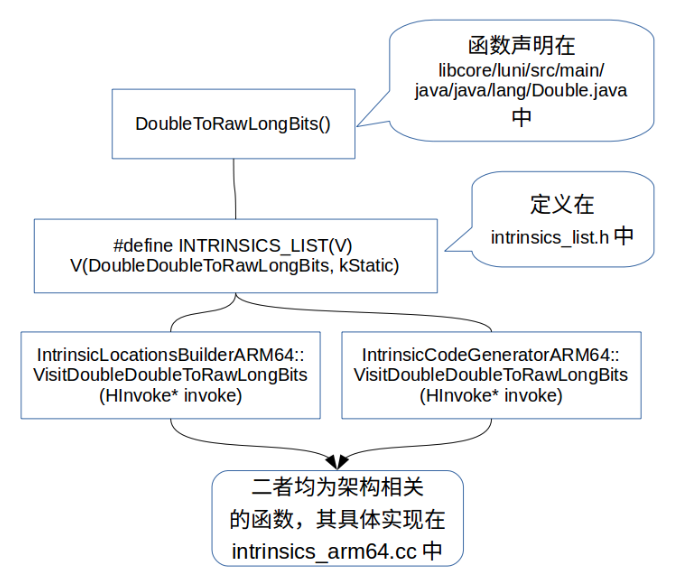
②optimizer负责对IR进行优化，共有15轮优化，包括intrinsic functions recognization, constant folding, boolean simplifier, dead code elimination, inliner, LICM(移除循环无关代码), GVN(值编号，去除冗余), Bounds Check Elimination, Instruction simplifier, Reference type propagation, side effects analysis等。在code generator之前还会进行寄存器分配。

③code generator 负责将优化后的IR转化为架构相关的指令。这一部分的实现包括了对vixl（安卓系统中集成的针对AArch64架构的动态代码生成工具）中assembler的调用，负责将指令转化为二进制形式。

1. 添加/提取污点

利用②optimizer 15轮优化中的intrinsic function来进行添加/提取污点的实现（原因①可以将污点的添加/提取过程最大限度的控制在optimizing compiler中，且在污点传播之前; ② android6.0中system image的编译工作默认交给了kQuick编译器，然而在Android7.0中已经取消了quick编译器的部分，仅保留了kOptimizing编译器，因此将污点添加/提取的实现放在optimizing compiler中可以更好的兼容Android7.0及以后版本）

以Java.lang.Double.doubleToRawLongBits()方法为例：



参照上例，在libcore/libart/src/main/java/java/lang中增加Taint类（Taint.java），在该类中仅声明污点增加/提取函数，具体的实现放在intrinsic\_arm64.cc中，相应还需要修改的文件包括dex\_file\_method\_inliner.cc/.h, intrinsics.cc, inline\_method\_analyser.h, intrinsics\_list.h, intrinsics\_arm64.h。

1. 污点传播

选择在③code\_generator\_arm64处进行污点传播，由于此时IR已经经过了多重优化，而其IR中包含的诸多类名又与dalvik字节码指令名或者架构相关的指令名相似，例如HParallelMove/HAdd等，可以了解这些IR类的目的。

将寄存器的污点定为2bits，可以代表4种状态（其中00表示污点清空，剩下的01/10/11数值越大，敏感度越高）。由于AArch64共63个寄存器，所以需要两个64位的通用寄存器或者一个128位的浮点寄存器来作为污点存储寄存器。最终，选取x11作为integral type data的污点存储寄存器，x12为floating point type data的污点存储寄存器（选取原因：①二者在AArch64架构中原本便是temporary register的用途；②在安卓系统原有的汇编程序中，二者的用途仅局限在art\_quick\_invoke\_static\_stub和art\_quick\_invoke\_stub这两个函数中，通过分析调用链可知并不影响运行时对x11/x12的使用；③选用128位的浮点寄存器会增多arm64的指令条数，仅污点分析相关的指令就比选用通用寄存器多出40%，且还涉及到两次FMOV指令会增大系统开销）

污点传播分为register -> register，register -> stack, stack -> register,stack -> stack四种，其中后三种涉及的主要是处理HParallelMove时所需要用到的。由于用类Location来表示IR的src和dst的存储位置，且Location中有多个public函数可以用来确定存储位置的种类，如Register或者FPRegister或者Stack slot等，所以比较容易确定如何处理具体IR的污点传播。

以下taint\_str表示污点存储寄存器，dst表示IR的目的寄存器，src表示源寄存器，temp表示临时寄存器，temp会在函数结束后恢复未使用时的状态。Stack\_index表示stack slot在栈中的索引值。

①对于register -> register情况：

1. 清除dst寄存器编号对应的taint\_str中的污点。采用指令：Bfm
2. 将src寄存器编号对应的taint\_str中的污点存入temp中，且格式为temp的第0:1的bitfield中。采用指令：Ubfm
3. 如果src有两个，那么选取两个寄存器编号对应的taint\_str中taint值大的那一个作为存入dst的taint；否则，直接将src寄存器对应的taint存入taint\_str中dst寄存器编号对应的位置中。采用指令：Orr

②对于register -> stack

1. 将src对应的污点存入temp中。采用指令：Ubfm
2. 将src对应的污点清空。采用指令：Bfm
3. 将temp中的值存入stack\_index + 1的地方。采用指令：Str

③对于stack -> register

1. 将stack\_index + 1处的值存入temp中。采用指令：Ldr
2. 清空taint\_str中dst对应的污点。采用指令：Bfm
3. 将temp中所存的值存入taint\_str对应的位置。采用指令：Orr

④对于stack1 -> stack2

1. 将stack1\_index + 1处的值存入temp中。采用指令：Ldr
2. 将temp中的值存入stack2\_index + 1处。采用指令：Str

涉及到修改的函数包括：

3.AArch32 vs AArch64（未完）

|  |  |  |
| --- | --- | --- |
|  | AArch32 | AArch64 |
| the number of core registers | 16（R0...R15） | 31(W/X0...W/X31) |
| the number of fpu registers | 32(S0...S31) | 32(S/D0...S/D31) |
| the use of the Program Counter(PC) | PC(R15) can be used in some T32 data processing instructions explicitly,or in branch instructions implicitly. And during the execution,the PC does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically PC–8 for A32, or PC–4 for T32. | The programmer cannot refer to PC explicitly by name or by number. PC contains the address of the currently executing instruction. |
| the register number of Stack Pointer(SP) | 13 | 63(in vixl)  31(in compiler) |
| register number of which parameters use(core register) | r1 - r3 | w/x1 - w/x7 |
| register number of which parameters use(fpu register) | none | s/d0 - s/d7 |
| The instructions that are used to load/store multiple registers from/into memory | POP/PUSH  VPOP/VPUSH(floating-point) | LDR/STR(can only load/store two registers once).  VLDR/VSTR(floating-point,the limitation is the same) |