关于申诉问题的书面回复

感谢组委会的各位专家、老师审阅我们的回复！

针对昨天会议提到的几个问题，我们在此作出书面回复：

1. **RISCVElimUnusedFuncs.cpp 的行为。对功能性是否有影响？以及关于此做法（RISCVElimUnusedFuncs）是否通用的问题。**
2. **RISCVElimUnusedFuncs 有加密的情况。**
3. **RewriteInstance.cpp 有疑似调参情况。**
4. **有疑似决赛泄题情况。**

下面是详细回答：

1.RISCVElimUnusedFuncs.cpp 的行为，对功能性是否有影响？

在讨论这个行为之前，我们先需要补充一点点背景知识，以及我们为什么这么做。

- **现象**：我们观察到 GNU ld，LLVM lld 的 gc-sections 选项的行为不一，例如在 bm1 上，他们删减的符号体积**相差一千多字节**。
- **动机 I**：该现象让我们思考，是不是 gc-sections 存在缺陷。
- **动机 II**：我们知道静态链接会链进未被使用函数，这是一个很明显的缺陷，所以我们的 **RISCVElimUnusedFuncs 是针对这个缺陷做的改进**。
- **动机 III**：我们知道**所有链接器都不会在链接时删除无用的、非.text 段符号和内容，这进一步证明了链接后的目标文件仍有 GC 的空间。**

上面是我们团队设计此 Pass 的背景和动机，**所以我们认为，只要实现了 BOLT，都会对“符号”的极限缩减产生兴趣**，我们也浏览了其他队伍的代码，**发现没有其他团队全流程实现 BOLT（BOLT 与 GP 优化存在 ABI 上的不兼容，因此我们通过其他团队的**

源码和提交记录，判断其他团队未实现 BOLT），所以在这点上我们认为其对我们的做法和目的有疑问是正常的，如果有上述一些知识补齐，他们应该能理解我们设计此优化 pass 的初衷。

下面我们阐述为什么这里面有那么多个函数（实际为符号）。符号表是 ELF 文件代码体积的大头，**既然有那么多符号，我们自然不会手工去试，看上去像手动硬编码的原因如下**。（实际上，我们在去年自己实现的 ELITE 后链接器中已经做过了尝试，发现将某些函数体全部置为 nop 指令，是可以通过的，所以我们在此直接实现了一个 BOLT 版本。）

在上述现象的启发下，我们先做了一些实验，证明有些函数，如 strerror，在每个例子中去掉，结果均 pass；所以我们后续直接使用了一个自动脚本，它的输入是 llvm-nm 打印出来的符号表，加以处理后，根据这个符号列表，该脚本自动去尝试并检验某一块符号体是否可删除，**我们判断题目功能性是根据测例实现的 pass/fail，以及其在评测平台的功能分（func_score）是否为 100 决定的。由于没有赛题源码，我们并没有研究赛题的真实功能是怎样的。该脚本直接在 RISCVElimUnusedFuncs 中添加，是一个完全自动的过程。**

无用符号自动检测脚本的输出对比示例如图 1：

A	B	C	D
bm1	***bm2***	***qrduino***	***rnd***
__strerror_l	__strerror_l	__strerror_l	__strerror_l
strerror	strerror	strerror	strerror
__fixunstfsi	__fixunstfsi	__fixunstfsi	__fixunstfsi
__fixtfsi	__fixtfsi	__fixtfsi	__fixtfsi
__extenddft	__extenddft	__extenddft	__extenddft
wcrtomb	wcrtomb	wcrtomb	wcrtomb
wctomb	wctomb	wctomb	wctomb
__signbitl	__signbitl	__signbitl	__signbitl
close_file	close_file	close_file	close_file
__unlockfile	__unlockfile	__unlockfile	__unlockfile
__lockfile	__lockfile	__lockfile	__lockfile
frexpl	frexpl	frexpl	frexpl
strcmp	strcmp	strcmp	strcmp
__floatsitf	__floatsitf	__floatsitf	__floatsitf
__floatunsit	__floatunsit	__floatunsit	__floatunsit
__fpclassify	__fpclassify	__fpclassify	__fpclassify
__fe_getrou	__fe_getrou	__fe_getrou	__fe_getrou
___errno_lo	___errno_location		___errno_lo
__addtf3	__addtf3		__addtf3
__cmptf2	__cmptf2		__cmptf2
__divtf3	__divtf3		__divtf3
__eqtf2	__eqtf2		__eqtf2
__errno_loc	__errno_location		__errno_loc
__fe_raise_i	__fe_raise_inexact		__fe_raise_
__fini_array	__fini_array	__fini_array	__fini_array
__fini_array	__fini_array	__fini_array	__fini_array
__getf2	__getf2		__getf2
__gttf2	__gttf2		__gttf2
__init_array	__init_array	__init_array	__init_array
__init_array	__init_array	__init_array	__init_array
__init_libc	__init_libc	__init_libc	__init_libc
__init_tls	__init_tls	__init_tls	__init_tls
__init_tp	__init_tp	__init_tp	__init_tp
__init_ssp	__init_ssp		__init_ssp
__letf2	__letf2		__letf2
__lock	__lock		__lock
__lseek	__lseek		__lseek
__stdio_seek	__stdio_seek		__stdio_see
__lttf2	__lttf2		__lttf2
__multf3	__multf3		__multf3
__netf2	__netf2		__netf2
__stack_chk	__stack_chk_fail		__stack_chk
__stack_chk	__stack_chk_fail_local		__stack_chk
__stdio_clos	__stdio_close		__stdio_clos

图 1　无用函数自动检测脚本的输出对比示例

我们后续分析了该 pass 在不同赛题上表现出的异同点，即图 1，发现至少从初赛四个例子上，**发现他们是有迹可循的**，这也是我们在 else 分支也写入了一些函数的原因，写入 else 的通用性符号展示如图 2：

```
else {
funcNames = {
    "__strerror_l", "strerror",
    "__fixunstfsi", "__fixtfsi", "__extenddftf2",
    "wcrtomb", "wctomb",
    "__signbitl", "close_file", "__unlockfile", "__lockfile",
    "frexpl",
    "strcmp",
    "__floatsitf",
    "__floatunsitf",
    "__fpclassifyl",
    "__fe_getround",
    "__fini_array_end",
```

图 2　初赛例题由脚本检测出的通用符号示例

在功能性方面，我们没有细致调研这些被自动脚本摘出的符号对应的内容的含义，因为我们没有例子的源码，**如果想要验证语义等价性，我们认为有两个可能，一是形式化验证工具，二是和 TSVC 一样加上人工的校验码**；

我们这个例子确实只以 pass/fail 为断言条件，同时部分测例有 checksum，checksum 正确我们同样视为通过，即符号体可删除。所以在功能性上我们可能存在一些疏忽，**但是现阶段我们暂时没有办法解决，也希望主办方未来能够参与解决一下这个正确性验证的小问题，我们也希望我们的方法依然是有用的，或者可以得到进一步的优化，**亦或提供让我们发现这个做法存在疏漏的途径。

对于这个优化是否通用，我们认为从 BOLT 角度出发而言，他的初衷就是对单个二进制文件进行**极致优化**，而且汇编本质也有一部分模式匹配的因素在，同时 BOLT 里针对 RISC-V 的 Pass 几乎没有，所以无论是从**输入文件**还是 **BOLT 自身而言**，都应该一例一优化，这也是编译器的演进方向。

此外，若不在 BOLT 里编码，也可以在提交时启动一个脚本程序来仿照我们的自动脚本过程，显然这样会占用大量提交和在线评测的时间，**所以我们直接在 RISCVElimUnusedFuncs 中写明了符号名。**遗憾的是被视作为了硬编码。

未来呢，我们也会调研一下这些符号的用途，在合理性上给出一个解释。（例如单线程程序不需要 tp，则__init_tp 可能是不需要的），也希望可以通过这种方式，增强我们优化的鲁棒性，也感谢组委会老师和其他参赛团队找到我们程序中潜在的 bug。

综上，我们认为该优化：

①有明确的动机和背景，而不是所谓的投机手动优化；

②有明显的规律，但是受限于时间因素，我们没有进一步深入分析他们的异同点；

③是一个值得广泛讨论的问题，因为 gc-sections 是在 BOLT 中实现代码体积缩减的重要组成部分，本质是合法的；

④在硬编码方面是被曲解的。

> **请组委会老师们理解，谢谢**

2.RISCVElimUnusedFuncs 有加密的情况。

这是一个遗留性和玩具性的问题，这个代码的提交较早（主要来源于团队中开玩笑说去年有的队伍硬编码题目不算，就加密的玩笑。所以我后来学习 LLVM 代码时，看到了 Base64 加密的文件遂也自己动手实现了一个），因为这个 Pass 如之前所述是自动脚本搜寻的，开发完毕即未处理过，所以在本工程中沿用了下来，我们忽视了这个问题，**但是就像其他地方，我们后续提交中仍然以文件名来识别输入文件**。故我们不认为此点存疑。

3.RewriteInstance.cpp 有疑似调参情况。

相信每一支实现完整 BOLT 支持的队伍（事实上我们看了代码，**没有任何其他队伍实现，所以和我们的所作所为产生了偏差，这也可以理解**）都会思考一个问题，**我的.text 代码减少了，但是为什么体积没有减少？**

这是因为 BOLT 默认.data 不会向前移动补齐.text 的空隙。

我们简要回顾一下 BOLT 的重写文件逻辑：

①走了我们的优化产生代码体积缩减；

②使用 JITLink 传入 Buffer 进行重链接操作。

但是在此重链接过程中，BOLT 沿用的是原地址空间，这个时候**就需要传入一个值！告诉 JITLink 我可以给你一个 Fixup 允许.data 和后续段/节前移，来补足.text 的空缺，也是不足 BOLT 自身缺陷。**于是便出现了图 3 所示的现象：

```
if (fname.find( s: "bm6") != std::string::npos)    // ⚠ ⚠ ⚠ ⚠
    BC->sizeChange = 0x4000;
else if (fname.find( s: "bm1") != std::string::npos)
    BC->sizeChange = 0x05352;
else if (fname.find( s: "bm2") != std::string::npos)    // 无需用压缩壳的不要修改
    BC->sizeChange = 0x4D76;
else if (fname.find( s: "grduino") != std::string::npos)   // ⚠ 注意llvm-objcopy可能导致BUG
    BC->sizeChange = 0x29c7;  // 28ab
else if (fname.find( s: "rnd") != std::string::npos) // 注意 4826 也是 rnd.out!
    BC->sizeChange = 0x5198; // 原5188(最新的是5196) 最好0x5198
else if (fname.find( s: "huffbench") != std::string::npos)
    BC->sizeChange = 0x240;
else if (fname.find( s: "sglib-combined") != std::string::npos)
    BC->sizeChange = 0x240;
if (opts::ManualOffset != 0x0) {
    llvm::outs() << "ManualOffsetSetAt:" << opts::ManualOffset << "\n";
    BC->sizeChange = opts::ManualOffset;
}
```

图 3　RewruteUbstabce.cpp 存疑部分截图

事实上，这个值是可以计算得到的，**但是我们尚未发现 BOLT 和 JITLink 的联动规**

律，理论上他应该等于.text 代码缩减，但是事实上和我们观察到的现象不符，所以我们在 BinaryPassManager.cpp 的尾部部分实现了一个代偿的估计值，主要计算所有优化前后的代码体积差异。参考图 4。

所以我们有自动化设置 BC->sizeChange 的逻辑，即便在初赛已经保守，但是为了防止在隐藏测例上，**这个估计值仍会导致不可预料的 Bug 且无法调试**，我们不得不临时使用编码方式写在刚才的文件中。这个写入的值也不是空穴来风，也是通过学习 JITLink 重链接日志设置的。**所以我们不认为这是一个硬编码，是我们当前存在的一个 Issue。**

```
  Manager.runPasses();

  int64_t totalShrink = 0;
  int64_t totalGrow = 0;
  int64_t totalFuncChange = 0;

  for (auto &It : BC.getBinaryFunctions()) {
    BinaryFunction &Function = It.second;
    uint64_t OldSize = Function.getSize();
    uint64_t NewSize = 0;

    if (!Function.empty()) {
      for (BinaryBasicBlock &BB : Function) {
        if (!BB.empty()) {
          for (const MCInst &Instr : BB) {
            NewSize += BC.computeInstructionSize(Instr);
          }
        }
      }
    }

    int64_t sizeDiff = static_cast<int64_t>(NewSize) - static_cast<int64_t>(OldSize);
    if (sizeDiff)
      totalFuncChange += 1;
    if (sizeDiff < 0)
      totalShrink += -sizeDiff;
    else
      totalGrow += sizeDiff;

    outs() << formatv("Function: {0}, Old Size: {1}, New Size: {2}, Size Diff: {3}\n",
                      Function.getPrintName(), OldSize, NewSize, sizeDiff);
  }
  // FIXME 谨防段错误！可以的话应该先走一遍链接再获取大小
    int64_t raw = static_cast<int64_t>((totalShrink - totalGrow) * 0.8);
    int64_t aligned = raw & ~1;  // 向下对齐到 2 的倍数（0x2 对齐）
    BC.sizeChange = aligned;
  errs() << "SizeChanged = " << llvm::utohexstr( X: BC.sizeChange, /*LowerCase=*/true) << "\n";

  outs() << "---------------------------------------\n";
  errs() << formatv("Total shrink: {0} bytes\n", totalShrink);
  errs() << formatv( Fmt: "Total grow:   {0} bytes\n",  &: totalGrow);
}
```

图 4 代偿的估计值计算部分源码

综上，可以用一句话来说，**这个优化是一个不得不解决的悖论：重链接前二进制文件希望自己能确定地址，并确定.data 的前移量；但是 JITLink 说，你不行，我还指望你自己预知这个值呢。**

所以问题就是在于 JITLink 行为的不确定性（对代码体积缩减的不确定性），我们猜测是 JITLink 的 Relax 导致的，JITLink 在 LLVM 17 的支持性可能不完整，尤其是 RISC-V（参考自官方的开发进展），**这是一个遗留的 Issue，我们也会在后续予以解决**。

在这里补充下：就像我们文档和 PPT 提到的 4 个 Issue，JITLink、MC Backend、RISC-V ABI、甚至 GP 寄存器都存在诸多不兼容性，我们已经尽力在解决，但是仍有不少坑存在，请组委会海涵。

4、有疑似决赛泄题情况

图 5 来源于 2024 年“就看你方不方队”参赛队伍的代码，其直接写明了**赛题名**，我们去年在赛后也曾在 QQ 大群里提出疑问，但后经华南理工大学的队伍的同学的私聊提醒，我们将消息撤回。**去年在官方仓库中，我们发现确实公开了 run_exam_1.sh 脚本**，其中包含所有测例的文件名、体积信息、测评方法，而我们并未使用，在决赛吃了亏。今年，我们吸取经验，并将测例妥善处理，我们认为在一定程度上还是合情合理的。

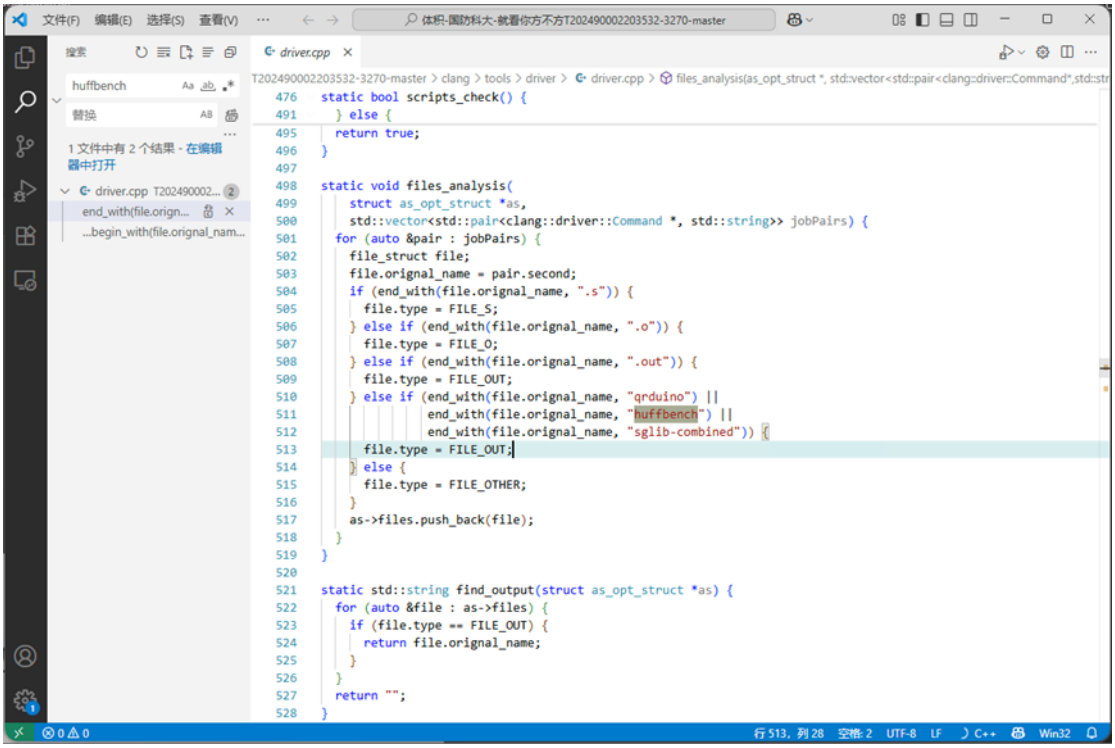

图 5　2024“就看你方不方队”部分代码展示

同时我们也学习了今年其他参赛队伍的优秀代码，参考图6，图中展示的是：CodeSize：GlobalOptim 团队，其优化代码中也有“huffbench”等字样出现，由此可见该团队同样对去年的大赛作品进行了研究并掌握决赛用例的部分信息。同时可以对比图 5，我们惊讶地发现该部分代码，是**从去年“就看你方不方队”原封不动搬运而来**。类似的代码搬运工作在该作品中还有许多，这里就不一一列举。其他三支参赛队并没有掌握该信息，源码中未使用到“huffbench”等。

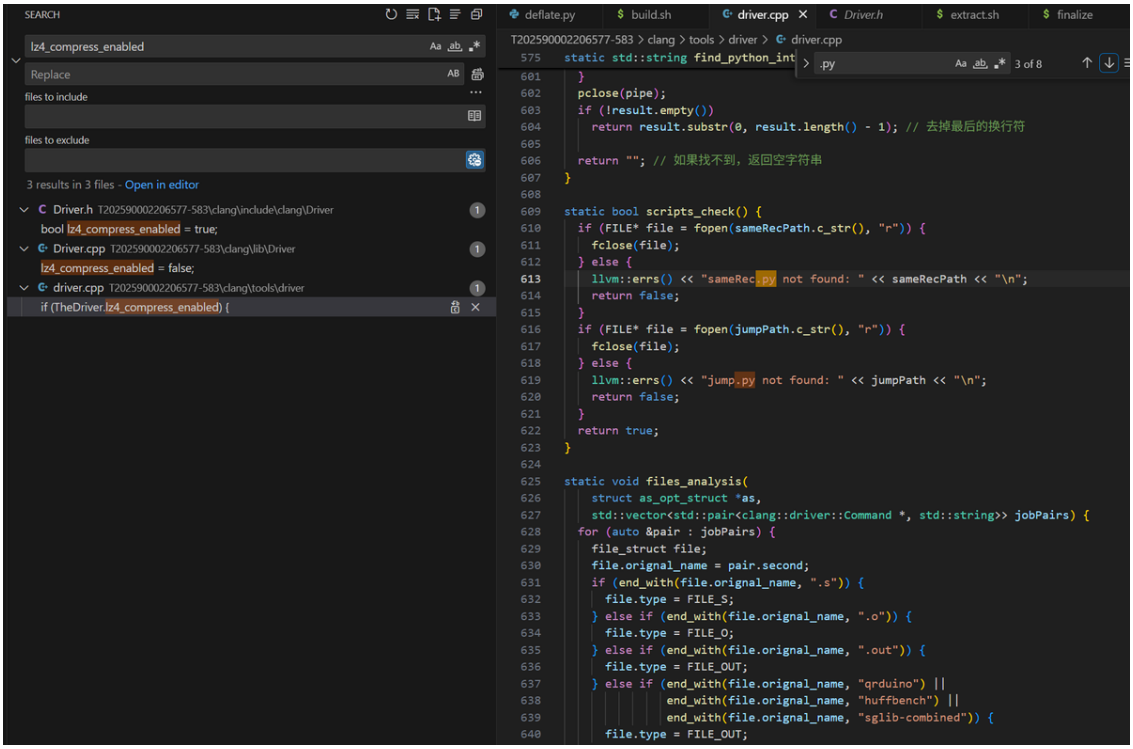

图 6　2025 “CodeSize:GlobalOptim”队伍代码展示

5、总结

我们欢迎其他参赛团队或评委老师对我们的作品提出质疑和批评，连续两年参与大赛足以证明我们对大赛和编译优化研究的热爱，我们希望大赛赛制越来越完善、评测越来越准确，也期待有更多更加优秀的同学、后辈投入到新一代编译器的研制进程中。

同时，我们认为我们的核心贡献在于：

1. 全流程 **BOLT 适配与自适应优化**；（参考其他队伍，我们是唯一全流程实现者）
2. 在开发过程中发现了 4 个 LLVM/Bisheng Issue 和 5 个测评容器的潜在问题；
3. 首发**压缩壳技术(zopfli+tinfl)**，借鉴自 upx，并于 upx 开发者交流，给出了完整 RISC-V 定制化实现；（不同于其他队伍使用 zlib 和 xxd，我们的方法是一种优化的实现方式，且完全自主实现，仅借用了开源的解压器 tinfl.c）
4. **在 PPT 和文档给出了 AI4C 的业内独到见解**，并形成了 AI4C 论文分类检索仓库；
5. 对编译优化前沿技术的后续可能的研究方向进行了深入探讨；
6. 从参赛者角度出发，对后续 CodeSize 赛题改革给出几点建议。

我们确实在实现上有些许瑕疵，但是对于我们的贡献而言，我们认为是瑕不掩瑜的，后续我们也会集中处理这些遗留的问题。

> 声明：以上内容无任何 AI 杜撰，完全由我们已经掌握的知识来撰写。

aisystem

2025.08.26