关于组委会评定结果的回复

尊敬的各位组委会专家、老师：

感谢您对我们的作品进行进一步审查。我们很遗憾我们某个模块的方法并没有得到规则的认可。作出此回复是想声明，我们并没有为了提高评测成绩而恶意使用违规操作，该方法是对 bolt 进行代码体积优化工作的必要补丁，并且在 X86、ARM 或 RV64 平台是可以通过采样、插桩等方式进行。我们认为这块研究工作还是有价值的，我们在大赛期间的确没找到在 rv32 平台更好的替代方案。

下面是我们“违规操作”的方法论，为什么其对 bolt 优化代码体积是必要的、为什么采用这样的实现方法、以及为什么我们在知晓 bolt 对代码体积优化贡献有限的前提下仍尝试去研究这块内容。

---

1. RISCVElimUnusedFuncs.cpp 的必要性

我们认为主要问题在于：“背景是对的、动机是对的、理论是对的、结果是对的，但过程碰巧是错的。”

延续之前的背景知识和相关论文：

- **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，这两种优化删减的符号体积**相差一千多字节**。
- **背景 I：有如下的参考论文均做了类似目的的优化：**
  - Nibbler: Debloating Binary Shared Libraries
  - Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries
  - LibFilter: Debloating Dynamically-Linked Libraries through Binary Recompilation
  - BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation
  - One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared
- **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。**

所以，我们在此做出进一步澄清。

首先，我们使用插桩工具在 X86 上观察到了存在冗余函数的问题，但是就如我们的 PPT 和文档所述，在 RISC-V 尤其是 RV32 中：第一，perf 不支持或有限支持，第二，我们缺乏满足需求的硬件设备，第三，测评环境容器本身不支持 perf。以上三点决定了我们注定不可能像在 X86 上一样快速合理地实现这个 gc-sections 优化。

其次，这个优化的必要性在于**“如果没有这个优化，BOLT 只会徒劳增大代码体积”**，这是因为 BOLT 自身段管理的失误，窥孔优化不会导致任何函数发生偏移以填充函数空洞，即题目所要求的后链接优化——BOLT 失去有优化的意义，这也是我们答辩时候遇到的一个比较难的问题，即——BOLT 到底能带来多少体积缩减。

最后，我们为什么认为背景、动机、理论、结果正确，但是过程错误，这是因为如果是别的架构一样具有完整采样/插桩功能性支持的架构，我们完全可以以一个合理，全面的方式来证明这点。**我们使用的方法和汇编层面一样，即扫描一个反汇编后的文件是否出现超过一次，或者被调用过。这是一个自然而然的过程，即 objdump 协同脚本观测即可**，这是我们去年采用的发现办法，我认为这一点毫无问题。

但是很遗憾，我们的工具仍存在缺陷，我们验证了一部分简单函数的插桩，但是对于较为复杂的情况，我们的自研插桩工具仍失效。

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_clo

图 1　无用函数自动检测脚本的输出对比示例

举一些例子，对于图1，多线程函数有__errno_location, __lockfile, __unlockfile, __init_tls, __init_tp，而嵌入式设备中采用的是单线程程序，因此可以安全删除；此外close, lseek, stdio_close, __stdio_seek 也是被链接进来的库，如果文件没有 IO 操作，这些都可以直接处理掉，就和我们的objdump法一样；strerror, __strerror_l因为我们使用了rodata裁剪优化，所以这一块在极限代码压缩上也可以不需要；__fixunstfsi, __fixtfsi, __extenddftf2, __addtf3, __multf3等是静态链接进来的可能无用函数，**使用我们的objdump 法可以直接删除（去年采用）**。

结合上面所述，即发生了一个悖论，BOLT 需要这个优化充当适配的“测试者”，但是 RISC-V32 因为种种技术原因（插桩、采样）不支持这个优化，所以除了上述objdump 看函数调用法外，我们只能采用试错法来搜寻和找到二进制文件的规律。注意，

这兴许是对特例进行优化，但是实际基本都是通用函数，我们也希望有充足的时间能研制出一个可在 RISC-V32 上对 ELF 插桩的工具来进行详细甄别。

我们承认我们的方法在某种意义上是一种次优策略，但是它的行为是和插桩是一样等效的，我们没有足够的时间来完成之前所述工具的研制，我们已经花了大量时间去修复 BOLT 的其它 Issue（PPT 第 24 页），当然也愿意在之后予以补齐这个 Issue。

同时，我们已经论证了部分函数在满足特定要求是可以删除的，所以我们不回避这种办法，**这个 Pass 无论是理论还是动机都是没有问题的**，尤其是参考动态链接那么多论文都在处理共享库的删除。我们也愿意进一步交流我们的观点，在此问题上已经讲述得较为清楚。

2. 为什么采用这种实现方式

在与各位专家老师在线进行申诉问题交流前，我们未察觉到这种方式是违规的，因为去年和前年某些参赛队伍的代码，都直接进行了类似的匹配操作，我们在学习过程中无意参考了这种实现方式。可以参考我们申诉回复信中的截图和 24 年优秀作品源码。

此外，bolt的优化理念也是对具体程序的操作。经过采样，该程序/目标文件对于 bolt 是完全透明的，优化任务也是一个白盒问题，我们的优化方法针对可以拿到程序汇编文件或目标文件的场景，可以起到不错的代码体积缩减效果。**同时，BOLT 原生支持以下选项：**

- **--skip-funcs：设定不采用优化的函数**
- **--skip-funcs-regex：设定不采用优化函数（正则匹配）**
- **... ...**

这些选项印证着 BOLT 设计者们认为针对特定文件进行某些函数的优化是合理的。

3. 为什么坚持基于 bolt 做 CodeSize 优化

去年参赛时，我们团队大赛经验不足，自认为作品已经做到比较优秀，但是在决赛答辩时，评委老师仍然指出我们的不足之处，让我们受益匪浅。

其中一条是我们没有较为**细化的团队分工和规范的代码仓库管理**，这一条我们在今

年的大赛中已经有了明显的进步。

另一条就是我们的后链接器实现存在很多设计上的不足，评委老师问到我们是否已经实现**了后链接器 bolt 对 rv32 的支持**。这一条意见激励着我们，今年一定要实现 bolt，bolt 的作用主要在于其针对程序的性能提升。而针对 CodeSize 的优化我们确实没有调研到业界有其他团队做了相关研究，但是开弓没有回头箭，我们仍然探索出通过 bolt 与插桩、采样工具以及我们自主实现的脚本配合，来实现对程序代码体积的压缩，只是由于 rv32 平台工具不足以及我们缺少 rv32 开发环境，导致该问题成为我们待解决的 issue。

4. 结语

我们认为我们的作品和 PPT、文档都有非常重要的价值，通过今年大赛，我们的表达能力、协作能力、写作能力和专业技术都得到了提升。也感谢大赛提供了这样的平台让我们得到提升和锻炼。我们同样会对今年比赛中，我们的问题和不足进行总结和反思。

我们始终认可大赛的评定，也认可大赛对规则进一步细化，同时我们也坚定自己的研究方向。未来我们会继续以这个研究工作为基础继续发表相关专利和论文，同时继续深入二进制相关的研究。

aisystem

2025.08.27