计算机研究与发展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(5): 1136−1150, 2025

本期推荐

面向 LoongArch 边界检查访存指令的 GCC 优化

舒燕君[1] 郑翔宇[1] 徐成华[2,3] 黄 沛[2] 王永琪[1] 周 凡[1] 张 展[1] 左德承[1]
[1]（哈尔滨工业大学计算学部 哈尔滨 150001）
[2]（龙芯中科技术股份有限公司 北京 100095）
[3]（中国科学技术大学计算机科学与技术学院 合肥 230026）
（yjshu@hit.edu.cn）

GCC Optimization for LoongArch Memory Accessing Instructions with Bound-Checking

Shu Yanjun[1], Zheng Xiangyu[1], Xu Chenghua[2,3], Huang Pei[2], Wang Yongqi[1], Zhou Fan[1], Zhang Zhan[1], and Zuo Decheng[1]
[1]（*Faculty of Computing*, *Harbin Institute of Technology*, *Harbin* 150001）
[2]（*Loongson Technology Corporation Limited*, *Beijing* 100095）
[3]（*School of Computer Science and Technology*, *University of Science and Technology of China*, *Hefei* 230026）

Abstract LoongArch ISA (instruction set architecture) introduces new memory accessing instructions with bound-checking to decrease the overhead of memory security check. However, as a new type of memory accessing instruction, the existing GCC (GNU compiler collection) compiler tools cannot support it and thus LoongArch based hardware remains underutilized. Therefore, in this paper, we revise the GCC compiler with the LoongArch memory accessing instructions to optimize the memory security check. Specifically, our work is divided into three parts: 1) designing built-in functions for the memory accessing instructions; 2) improving the RTL (register transfer language) optimizer of GCC to recognize two kinds of semantic patterns of memory accessing instructions with bound-checking, which are non-exception handling and exception handling; 3) implementing a new exception signal SIGBCE for the bound check exception BCE that is raised by CPU in Linux kernel, and implementing the corresponding signal handling function in glibc (GNU C library) to deal with the bound check exception. The experiments on GCC 12.2.0 and Loongson 3C5000L server show that the revised compiler is able to correctly employ the new memory accessing instructions and bring an acceleration of approximately 20% in some security routines. Our work improves the ecosystem of LoongArch and boosts the development of LoongArch ISA. It will also be referential to GCC optimization for the specialized instructions.

Key words compiler optimization; LoongArch; GCC; memory accessing with bound-checking; Loongson CPU; exception handling; memory security

摘 要 为了减少内存安全检查的开销，LoongArch 指令集架构引入了边界检查访存类指令. 然而，作为一种新的内存访问指令，目前 GCC（GNU compiler collection）编译器不支持该类指令，LoongArch 硬件能力不能得到充分利用. 针对此 LoongArch 边界检查访存指令改进了 GCC 编译器，实现利用该类指令优化程序的内存安全检查. 具体而言，完成了 3 个方面的工作：1）设计实现了针对边界检查访存指令的内建函数；2）改进 GCC RTL（register transfer language）阶段的优化器，使其能够识别无异常处理和带异常处理 2 种情况的边界检查访存语义，并自动优化；3）面向 LoongArch 边界检查访存指令触发的边界检查异常

收稿日期：2024-02-21；**修回日期**：2024-08-12
基金项目：国家自然科学基金项目（61202091, 62171155）
This work was supported by the National Natural Science Foundation of China (61202091, 62171155).

（bound check exception, BCE），设计了新的 Linux 内核异常信号 SIGBCE 和相应的运行时库 glibc（GNU C library）的信号处理函数，实现了 BCE 处理. 通过在 GCC 12.2.0 和龙芯 3C5000L 服务器进行实验，验证了改进后的编译器不仅能正确使用新引入的边界检查访存指令，而且在某些安全函数中带来接近 20% 的性能提升. 完善了 LoongArch 生态，推进了 LoongArch 指令集发展，对此类特定指令编译器优化工作有一定的借鉴意义.

关键词 编译器优化；LoongArch；GCC；边界检查访存；龙芯 CPU；异常处理；内存安全

中图法分类号 TP391

DOI: 10.7544/issn1000-1239.202440100 **CSTR:** 32373.14.issn1000-1239.202440100

内存安全在软件开发和程序设计中有着重要地位，与系统安全密切相关. 内存安全问题的根源在于对内存的非法读取与篡改，例如缓冲区溢出攻击[1]等攻击手段. 现代高级语言中，提供了一系列安全机制，对访存操作进行安全检查，以保障内存安全. 然而安全检查会引入较大的时间开销，特别是一些基于软件的安全机制[2]会使程序性能严重下降，无法应用到实际系统中.

对此，研究者们提出在指令集架构层面引入访存安全的相关指令，基于硬件来实现内存操作的安全性检查，减小内存安全检查所带来的开销. 比较著名的设计有 Intel MPX（memory protection extension）[3]，ARM MTE（memory tagging extension）[4]等. LoongArch[5]是一种新型的精简指令集计算机（reduced instruction set computer, RISC）架构[6]. 在 LoongArch 架构中，针对访存操作的安全性检查设计了一类边界检查访存指令. 这类指令在访存之前检查访存地址与边界地址之间的关系，当边界检查条件满足时正常访存，否则触发访存异常. 通过将安全检查和访存操作合并在 1 条指令中完成，LoongArch 边界检查访存指令可以减少程序的指令数量，降低安全检查的开销.

然而，在指令集中引入新指令，会对编译器优化提出新的要求，需要对编译器进行改进以支持新指令. 例如针对 Intel MPX，GCC（GNU compiler collection）编译器实现了一套边界信息管理机制，在指针解引用的代码中插入检查指令[7]；针对 ARM MTE，Clang 编译器也增加了相应的指针检查器，优化了安全机制[8]. 由于 LoongArch 指令集架构较新，而且 LoongArch 边界检查访存指令采取了一些明显不同于已有指令集的设计，目前 GCC 等编译器并不支持该类指令，从而导致基于 LoongArch 指令集架构的龙芯处理器能力不能充分发挥.

编译器优化是利用计算机硬件能力的根本手段，也是对指令集架构设计的最终检验. 为此，本文围绕 LoongArch 边界检查访存指令的编译优化展开一系列工作：首先分析了 LoongArch 边界检查访存指令的语义特征，结合该指令的实际应用场景，提出了区分无异常处理和带异常处理 2 种情况来设计边界检查访存指令编译优化方案. 然后基于 GCC 编译器，设计实现了内建函数用于提示编译器进行优化，并分别实现了无异常处理和带异常处理 2 种情况下的语义识别算法，用于自动优化含有边界检查访存语义的 C 语言程序. 此外，针对带异常处理的情况，本文设计了一种嵌入异常处理所需信息的方法，形式上扩展了指令操作数. 通过以上改进 GCC 的工作，无论是在无异常处理还是带异常处理的情况，对用户程序而言，边界检查访存指令的行为都是简单而明确的，从而可以比较方便地在 GCC 实现 LoongArch 边界检查访存指令匹配.

由于 LoongArch 边界检查访存指令在边界检查条件不满足时会触发边界检查异常（bound check exception, BCE），而 Linux 内核和程序运行环境不支持该异常，本文还设计实现了 Linux 内核和运行环境的 BCE 处理机制. 在 Linux 内核中设计了新的边界检查异常信号（signal of BCE, SIGBCE），并基于 glibc（GNU C library）实现了 SIGBCE 的信号处理函数. 通过以上异常处理流程，确保程序执行边界检查访存指令的实际语义和编译器优化实现的语义相一致.

为了检验本文改进的 GCC 编译器的优化效果，在 3 个使用边界检查访存的安全函数上进行测试. 实验结果表明，改进后的 GCC 编译器能够正确识别和利用 LoongArch 边界检查访存指令以实现安全函数的优化，减小安全检查带来的开销，其性能改善显著，最大接近 20%. 此外，本文提出的编译优化方案不需要重新编写程序，只需要使用改进后的编译器重新编译程序，即可得到 LoongArch 边界检查访存指令带来的性能优化效果.

本文的主要贡献有 3 点：

1)研究分析了 LoongArch 边界检查访存指令的语义特征,结合该指令的实际应用场景提出了区分无异常处理和带异常处理 2 种情况,确定了边界检查异常的编译处理方法,便于编译器实现用户程序语义与边界检查访存指令语义匹配,简化编译优化过程.

2)设计实现了一套基于 GCC 编译器的 LoongArch 边界检查访存指令的编译优化方法,包括内建函数和语义自动识别优化,其实现思路可作为类似指令优化的参考.

3)设计实现了一种在程序中嵌入异常处理信息的方法,并在 Linux 内核和 glibc 库中实现了对 BCE 的处理机制,为在 GCC 编译器、Linux 内核以及 glibc 库实现新的异常处理提供借鉴.

1 背景知识

1.1 LoongArch 边界检查访存指令

LoongArch 指令集针对访存操作的安全性检查设计了一类边界检查访存指令[9],其具体行为定义如下:检查访存地址与边界地址的关系是否符合检查条件,如果不符合条件则触发边界检查异常并终止访存操作;如果符合条件则使用访存地址进行访存操作,并进行访存操作中的其他地址合法性检查.

LoongArch 边界检查访存指令支持多种访存数据类型、访存操作类型和边界检查条件,其指令汇编记号格式写作{*op*}{*cond*}.{*width*},指令中各部分记号的具体内容如表 1 所示.

Table 1 Assembly Mnemonics of Memory Accessing Instructions with Bound-Checking

表 1 边界检查访存指令汇编记号

op	*cond*	*width*
ld, st, fld, fst	le, gt	b, h, w, d, s

注: *width* 类型记号"d"表示 8 B 整数或双精度浮点数.

记号{*op*}为访存操作的操作符,包括整数读内存 ld、整数写内存 st、浮点读内存 fld、浮点写内存 fst. 记号{*cond*}为边界检查条件,可以是访存地址小于等于边界地址 le 或是访存地址大于边界地址 gt. 记号{*width*}表示数据宽度类型,包括 1 B 整数(b)、2 B 整数(h)、4 B 整数(w)、8 B 整数(d)、单精度浮点(s)、双精度浮点(d).

LoongArch 边界检查访存指令的编码格式如图 1 所示. 指令的 15~31 b 是指令的操作码,对应了汇编记号. 其中 20~31 b 指明该条指令为边界检查访存指令,值为 001110000111; 15~19 b 是字段 *F*,编码了边界检查访存指令的数据类型、访存操作以及检查条件,例如 19 b 编码指令是浮点指令还是整数指令. 指令的 0~14 b 表示 3 个通用寄存器字段 *rd*, *rj*, *rk*. 其中,字段 *rd* 是访存数据寄存器,读操作将读取到的内容写入此寄存器,写操作则将此寄存器的内容写到内存;字段 *rj* 是访存地址寄存器,指定了访存地址,所有指令访存地址均要求自然对齐,否则将触发对齐异常;字段 *rk* 是边界地址寄存器,指定参与安全检查的边界地址,指令执行时,比较字段 *rj* 和字段 *rk* 索引的寄存器的值是否满足条件要求.

1.2 GCC 编译器优化工作流程

GCC 编译器的整体优化流程如图 2 所示,可以分为前端和后端 2 个部分[10]. 图 2 中,前端主要在虚线框之外的剩余部分,前端包括对 C 语言程序的词法分析、语法分析、语义分析,以及中间表示的转换和优化过程. GCC 支持多种高级语言,因此首先由 Generic 转换过程将高级语言的语法分析树翻译为通用的中间表示 Generic,然后通过 Gimple 转换过程将 Generic 中间表示翻译为 Gimple 中间表示. 前端会在 Generic 中间表示和 Gimple 中间表示上做许多优化.

图 2 中,GCC 优化的后端位于虚线框,首先将优化后的 Gimple 中间表示展开为 RTL(register transfer language)序列. 后端实现 RTL 序列的优化并解决 RTL 中未解决的问题. 所有环节结束后,将 RTL 序列翻译为汇编文本,然后编译命令还会自动调用汇编器、链接器等生成最终的二进制文件.

1.3 Linux 内核与 glibc 库

LoongArch 边界检查访存指令在条件不满足时会触发 BCE 异常. 在现代计算机系统中,处理器硬件

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
操作码																	边界地址					访存地址					访存数据				
0	0	1	1	1	0	0	0	0	1	1	1	*F*					*rk*					*rj*					*rd*				

Fig. 1 Format of LoongArch memory accessing instructions with bound-checking

图 1 LoongArch 边界检查访存指令格式

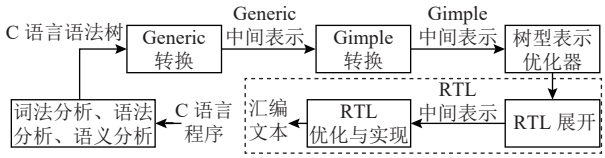


Fig. 2 The workflow of GCC optimization

图 2 GCC 优化的工作过程

触发的异常通常由操作系统内核直接处理，用户程序需要通过内核提供的机制接收和处理异常. Linux 信号机制[11]就是这样一种软件层面对于中断异常的处理机制.

为了确保程序执行边界检查访存指令的实际语义和编译器优化的语义相一致，必须在运行时库中处理异常信号. glibc 库是 GNU 组织开发的、适用于一类 POSIX(portable operating system interface)环境的 C 语言运行时库. 在 GNU/Linux 操作系统上，一般的 C 语言程序都会链接到 glibc. glibc 提供各种封装好的库函数如函数 *printf*、函数 *strncpy* 等供用户程序使用，并在执行用户程序主函数之前进行各种初始化工作. glibc 库是操作系统中必不可少的组件.

2 相关工作

2.1 边界检查访存的硬件加速机制

鉴于系统安全的重要性和纯软件实现安全检查对性能影响较大的现实问题[2]，使用硬件加速安全检查十分必要.

针对边界检查访存，Intel 曾向 x86 指令集引入 Intel MPX 扩展[3]，包括边界寄存器和边界检查指令. GCC 编译器于 5.0 版本开始支持 Intel MPX. 然而由于 Intel MPX 的设计受到 x86 指令集兼容性包袱的约束，通用寄存器数量很少导致需要引入额外的专用寄存器保存边界信息，许多指令都能访存导致不适合直接在访存指令上增加检查功能而需要额外的检查指令. 因此，实际 Intel MPX 扩展加速效果不理想，优化效果也不佳[12]. Intel 处理器、GCC 编译器、Linux 内核以及 glibc 库等目前已经删除 MPX 支持.

在 AArch64 指令集中，通过 MTE 扩展[4]引入了边界检查访存的安全机制. MTE 包括内存标签技术和标签操作指令，使用带标签的指针(也称为染色指针)访问内存，硬件自动检查指针标签与内存块标签是否一致. 这种技术可阻止堆栈溢出攻击以及释放后使用等攻击和误用.

LoongArch 指令集中，边界检查访存指令是基础指令集的一部分. LoongArch 不使用专用的边界寄存器，而是将边界信息保存在通用寄存器中. 边界检查访存指令有效利用 RISC 架构的 Load-Store 特点，将安全检查和访问内存的功能合并在 1 条指令中完成，在访存之前检查访存地址与边界地址是否满足要求，避免边界检查对程序引入额外开销.

此外，LoongArch 边界检查访存指令充分利用 RISC 指令集通用寄存器数量多的特点，使用通用寄存器保存边界信息. 通用寄存器数量多、灵活性好、效率较高，此类指令执行简单，有助于减少安全检查对程序执行的影响. 与已经被删除的 Intel MPX 相比，作为新型指令集，LoongArch 边界检查访存指令发挥 RISC 指令集的优点，避免了不必要的兼容性包袱，更简明高效.

2.2 基于特殊指令的程序优化

利用特殊指令优化程序的直接方法是编写对应的汇编程序[13]，常见于系统库、密码算法加速库、多媒体应用程序等. 例如：赵翔等人[14]深入分析图像处理、数据压缩等领域中的实数快速傅里叶变换(fast Fourier transform, FFT)算法的并行性，使用 ARM Neon 汇编开发高性能的实数 FFT 算法库；Cai 等人[15]面向 LoongArch 架构，为最常用的 glibc 运行库的库函数编写汇编语言版本；杨昊等人[16]采用 AVX2 指令设计实现现代加解密标准算法，减少加解密算法的运算时间和开销；赵龙等人[17]采用 SIMD(single instruction multiple data)指令集设计实现高强度 ECC(elliptic curve cryptography)攻击算法；沈洁等人[18]基于飞腾处理器，利用 SIMD 向量指令和部件，优化了向量三角函数. 鉴于汇编程序需要针对特定平台、特定场景，编写不方便，研究人员通常在设计实现运算库时将底层资源加以抽象，并将各种常用加速算法实现为模块.

相比于编写汇编程序，利用特殊指令优化程序更好的方法是改进编译器. 一般来说，改进编译器以利用特殊指令，主要有 2 种手段. 一种手段是引入内建函数作为 C 语言的语义扩展[19]，内建函数直接提示执行某类优化. 例如对于 SIMD 指令，单纯的 C 语言难以描述向量类型和相应的并行操作，因之引入了大量的内建函数，被称为 intrinsics 函数[20]. 内建函数具有一定的通用性，相比直接使用汇编语言可移植性更好. 另一种手段是语义识别和优化，通过一类匹配算法，分析程序结构和变量之间的关系，明确某一段 C 语言程序隐含的语义，并实现特殊指令的优化. 例如对于位操作指令(bit manipulation instruction, BMI)的优化. Koppelmann 等人[21]扩展 RISC-V 指令集，通过改进 GCC 等编译器使其能够识别适合使用

BMI 优化的代码模式. 研究表明，通过引入 BMI 指令并实现面向 BMI 的编译器优化，可以有效提高 RISC-V 的性能并减小代码体积[22]. 针对更为复杂的 SIMD 指令，研究者提出了大量自动向量化方法进行相关的语义识别和优化[23-25]. ARM 等指令集还引入了一类特殊的指令形式，即条件执行(又称为谓词执行). 针对条件执行，编译器可采用 IF 转化方法进行优化[26]，将一部分分支指令消除，从而提高性能. 所消除的分支指令可包括安全检查带来的分支指令. 并且，条件执行优化还可以带来更多优化机会，例如寄存器分配方面的优化[27].

2.3 边界检查访存的编译优化

针对 Intel MPX, GCC-5.0 编译器引入了一套称作指针边界检查器(pointer bounds checker)的安全机制. 编译器首先利用扩展的数据流分析方法分析程序中各个指针的边界. 对于能够确定边界的指针，增加额外的、程序中不可见的数据结构记录边界信息，在解引用代码之前插入边界检查代码阻止对这些指针越界访问. Intel MPX 使用硬件机制加速上述实现，即使用边界寄存器 bnd0-bnd3 保存边界信息和引入边界检查指令如 bndcu 等用于实现解引用之前的检查. 此外 Intel MPX 修改 Linux 内核以支持处理新的检查异常，修改 glibc 以维护边界寄存器的信息.

对于 ARM MTE，目前已在 Clang/LLVM 编译器中为既有的地址检查机制 ASan(address sanitize)增加了基于 MTE 的实现. 编译器在分配内存的代码中插入分配标签的代码，将所需要的边界信息等通过 ARM MTE 引入的专用指令写入标签对应的影子内存中，并确保指针所带的标签正确传播. 访存时，若使用带标签的指针，则安全检查由硬件实现，无需额外的检查指令.

无论是 GCC 编译器对 Intel MPX 的支持还是 LLVM 编译器对 ARM MTE 的支持，都是利用硬件来加速既有的检查机制. 与这些工作不同的是，本文聚焦于优化程序中显式存在的安全检查. 基于对 LoongArch 边界检查访存语义的分析，本文引入内建函数提示编译器针对边界检查访存语义进行优化. 本文还采用 RTL 中间表示上的优化算法来识别程序中的边界检查访存语义并自动优化. 总的来说，本文工作针对 LoongArch 边界检查访存指令的特点，直接识别程序中显式的边界检查访存语义和降低程序中此类显式检查的开销，方法简明有效.

3 总体设计

图 3 是本文基于 GCC 编译器实现 LoongArch 边界检查访存指令优化的总体设计，主要包括 3 个部分：GCC 编译器优化用户程序、LoongArch 处理器执行边界检查访存指令、Linux 内核与 glibc 运行时库在发生 BCE 时进行异常处理.

图 3 的左起第 2 个方框是 GCC 编译器对用户程序的优化. 首先对 C 语言编写的用户程序进行分析，识别出使用 if 语句、三目运算符或内建函数等描述的对访存地址进行的安全检查. 这里的内建函数是本文针对边界检查访存指令设计实现的. 由于内建函数需要程序员手动调用，因此本文进一步改进 RTL 优化器，使之能进行语义识别并自动开展优化. 改进后的 RTL 优化器会匹配安全检查对应的内建函数或特殊代码模式，在生成汇编代码时生成 LoongArch 边界检查访存指令优化代码，并根据情况嵌入异常处理所需信息，最终得到使用边界检查访存指令优化的可执行用户程序.

包含 LoongArch 边界检查访存指令的用户程序在运行环境中执行. 在检查访存地址时，若满足访存

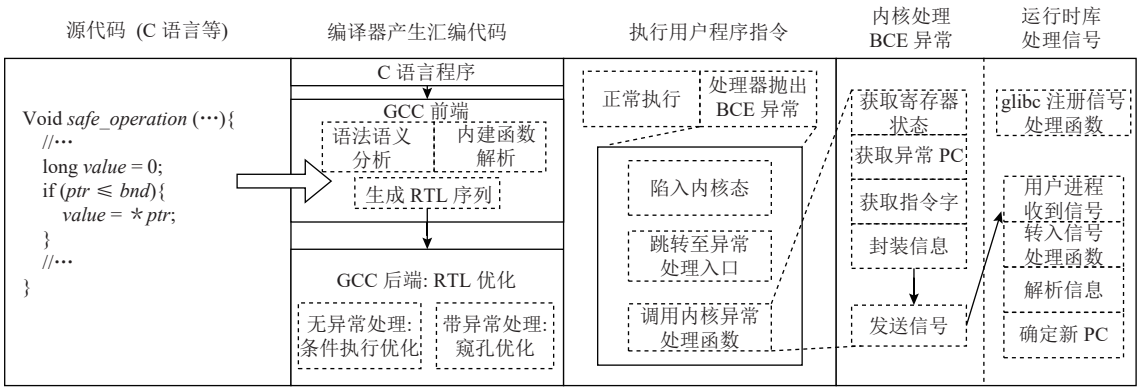


Fig. 3 Overall design of GCC optimization for LoongArch memory accessing instructions with bound-checking

图 3 LoongArch 边界检查访存指令 GCC 优化的总体设计

条件要求，处理器正常执行用户态程序，若不满足访存条件要求，则触发 BCE 并陷入内核态执行，如图 3 的左起第 3 个方框所示.

由于目前 Linux 内核并不支持 LoongArch 边界检查访存指令触发的 BCE，本文修改了 Linux 内核和运行时库 glibc，如图 3 的最右侧方框所示. 为了能够捕获和处理硬件抛出的 BCE 异常，本文扩展了 Linux 内核，设计实现了函数 *do_bce* 以解析 BCE 异常发生时的处理器状态信息，包括寄存器状态、异常 PC（program counter）、指令字等，并将这些信息封装到一个新的信号 SIGBCE 发送给程序运行时 glibc 以完成异常处理. 为了处理内核发送的 SIGBCE 信号，本文还进一步改进了运行环境的运行时库 glibc，设计实现了 SIGBCE 信号处理函数，使程序对边界检查异常信号 SIGBCE 的处理无感知，并避免程序错误地修改边界检查异常的信号处理函数. 通过这样一系列异常处理流程，确保对程序而言，边界检查访存指令的实际语义和编译器优化过程中所实现的语义一致.

4 编译优化

本节详细介绍 GCC 编译器优化相关内容. 4.1 节首先针对边界检查不满足条件时是否影响程序的执行流程，设计 2 种边界检查访存指令语义，以便于编译器利用该指令进行程序优化. 4.2 节针对边界检查访存指令设计实现内建函数，直接提示编译器进行优化. 由于内建函数还需要程序员手动调用，4.3 节进一步改进 RTL 优化器，使 GCC 编译器能够自动识别语义并进行优化.

4.1 语义设计

LoongArch 指令集的边界检查访存指令语义如图 4 所示. 访存之前首先检查访存地址与边界地址之间是否满足边界检查条件，如果满足则正常进行访存；如果边界检查条件不满足，则不访存，并触发 BCE. 图 4 中的操作 *MemoryLoad* 表示以 *rj* 为访存地址读取 8 B 的数据，if-else 语句说明了边界检查操作，#BCE 操作表示条件不满足则触发一个 BCE.

```
// ldle. d rd, rj, rk
if (rj≤ rk){
  rd = MemoryLoad (rj, 8);
}
else {
  #BCE;
}
```

Fig. 4 Semantic of memory accessing instruction with bound-checking

图 4 边界检查访存指令语义

然而在 C 语言中，并不支持类似 Java 语言的异常及捕获语法，LoongArch 边界检查访存指令的 BCE 异常语义很难直接应用在 C 语言程序中，编译器也很难利用边界检查访存指令进行程序优化. 因此，需要针对边界检查访存指令设计更为简单而明确的语义，便于编译器实现用户程序与该指令的匹配.

LoongArch 边界检查访存指令在检查条件不满足时触发 BCE 异常，通过采用不同的异常处理方式可以灵活实现用户程序的任意语义. 然而，过于复杂的设计难以进行编译优化，而且也可能并不会被用户程序所使用，通过分析一些典型的带有安全检查的程序代码可以发现，在边界检查不满足条件时，依据程序的执行流程是否发生改变，主要分为 2 种场景，如图 5（a）（b）所示.

1）场景 1.如图 5（a）所示，在解引用指针 *ptr* 之前

```
extern noreturn void excep (const char* err, …);
long test (long defval, long* ptr, long* bnd){
  if (ptr ≤ bnd) {
    defval= *ptr;
  }
  return defval;
}
```

(a) 无异常处理的 C 程序示例

```
extern noreturn void excep (const char* err, …);
long test (long defval, long* ptr, long* bnd){
  if (ptr ≤ bnd) {
    return *ptr;
  }
  else {
    excep ("Bad pointer", ptr, bnd);
  }
}
```

(b) 带异常处理的 C 程序示例

```
// ldle.d rd, rj, rk
if (rj≤ rk){
  rd = MemoryLoad (rj, 8);
}
```

(c) 无异常处理的边界检查访存

```
// ldle.d rd, rj, rk, offset
if (rj≤ rk){
  rd= MemoryLoad (rj, 8);
}
else {
  pc = pc + offset;
}
```

(d) 带异常处理的边界检查访存

Fig. 5 Examples in C language and semantic designs of memory accessing with bound-checking

图 5 C 语言用户程序示例与边界检查访存语义设计

检查访存地址是否小于等于边界地址, 当不满足条件则不进行访存, 接着顺序执行程序, 不因边界检查异常对用户程序执行流程产生影响.

2)场景 2.如图 5(b)所示, 在解引用指针 *ptr* 之前检查访存地址是否小于等于边界地址, 若不满足条件也不进行访存, 但是会因边界检查异常而改变用户程序的执行流程, 跳转执行特定的程序逻辑, 如打印错误信息等.

基于以上 2 种场景, 本文分别设计了无异常处理的边界检查访存和带异常处理的边界检查访存 2 种语义, 如图 5(c) (d)所示. 图 5(c)是无异常处理的边界检查访存语义说明, 在这种情况下边界检查异常对程序执行流程没有影响, 可以解释为一个条件传送形式的指令, *rd* 是访存数据寄存器, *rj* 是访存地址寄存器, *rk* 是边界地址寄存器. 从用户程序来看, 如果边界检查条件不满足, 则不访存且不改变访存数据寄存器的内容, 并继续执行后续指令.

图 5(d)是带异常处理的边界检查访存语义说明, 当边界检查异常发生时, 跳转到目标地址处执行特定的用户程序逻辑. 其形式上是一个带有分支跳转操作的指令, 不满足访存条件时将进行分支跳转, *rd* 是访存数据寄存器, *rj* 是访存地址寄存器, *rk* 是边界地址寄存器, *offset* 是分支跳转的 PC 相对偏移量. 从用户程序来看, 如果边界检查访存条件不满足, 则不访存并跳转到一个指定地址处继续执行, 相当于访存指令和分支指令合二为一.

通过将 LoongArch 边界检查访存指令实际应用场景分为无异常处理和带异常处理 2 种情况, 确定了 BCE 在编译过程的处理方法, 能够极大地方便编译器实现用户程序与该指令的匹配以及利用该指令进行程序优化.

4.2 内建函数

为特殊指令设计内建函数是一种常见的优化手段, 程序员调用内建函数可直接提示编译器进行优化. 内建函数具有一定的通用性, 相比直接使用汇编语言可移植性更好.

GCC 的内建函数原型框架非常完善, 所有内建函数统一以 *_builtin_* 开头作为特殊记号. 内建函数本质上借助函数的形式去描述一种运算或者操作, 通常标记为相应的内部函数, 即一种通用表达式, 其具体实现由 RTL 序列描述. 一般来说, 编译器识别了内建函数后, 最终都能将代码在相应位置翻译成优化的机器指令片段, 免除了函数调用开销, 与内联函数类似.

对于无异常处理的边界检查访存, 在边界检查条件不满足时不做任何操作. 对于带异常处理的边界检查访存, 需要建立一种方式描述“边界检查不满足时跳转到指定位置继续执行”. 本文使用 if-else 语句描述这种语义.

为了实现适合于无异常和带异常 2 种边界检查访存情况的设计, 本文分析 GCC 编译器的内建函数, 发现形式上访存边界检查与有符号数加法溢出检查类似. 参考“带溢出检查的算术操作”相关的内建函数, 本文设计了适用于各种情况的边界检查访存内建函数原型, 下面以 4 种数据类型的边界检查读内存为例进行说明, 如图 6 所示.

```
bool_builtin_ldle_bndchk (long* ptr, long* bnd, long* res);
bool_builtin_ldle_bndchk (int* ptr, int* bnd, int* res);
bool_builtin_ldle_bndchk (short* ptr, short* bnd, short* res);
bool_builtin_ldle_bndchk (char* ptr, char* bnd, char* res);
```

Fig. 6 Prototypes of built-in functions

图 6 内建函数原型

在图 6 的边界检查访存内建函数原型中, 指针 *ptr* 是欲访问数据的地址, 指针 *bnd* 是参与安全检查的地址, 指针 *res* 指向一个保存访存结果的变量. 内建函数的语义为: 判断指针 *ptr* 是否小于等于指针 *bnd*, 如果边界检查条件满足, 则正常访存, 从指针 *ptr* 指向处读取一个数据然后将结果保存到 **res*, 接着返回 false 表示顺利访存; 如果边界检查条件不满足, 则返回 true, 表示边界检查有错误. 边界检查访存内建函数可直接与 if 语句配合表示异常处理语义. 若不使用 if 语句判断内建函数返回值, 则意味着采用无异常处理.

4.3 RTL 优化

由于内建函数需要程序员手动调用, 因此有必要进一步改进编译器, 使之能进行语义识别并自动开展优化. GCC 编译器的许多优化都是在 RTL 阶段实现的, 例如条件执行(条件传送)、分支指令和基本块的重排、延迟槽调度、窥孔优化等, 优化算法的基础是匹配 RTL 指令模式. 本文也选择在 RTL 阶段实现边界检查访存指令的优化, 主要由于 3 个方面的原因:

1)边界检查访存指令是 LoongArch 指令集特有的指令类型.

2)其他指令集架构不能就优化改动获得收益, 那么优化改动不应该影响其他架构的程序执行.

3)GCC 编译器的匹配-简化框架不能很好地支持指针变量.

LoongArch 边界检查访存指令的 BCE 异常语义使该指令难以直接在程序中使用. 根据 4.1 节的设计，边界检查访存的语义分为无异常处理和带异常处理 2 种情况，由于这 2 种情况的语义差异较大，本文采用不同的方式进行优化.

1)无异常处理语义

如图 5(c)所示，LoongArch 边界检查访存指令在无异常处理场景下可以直接看作条件传送指令，边界检查异常发生对控制流无影响. 一般认为，条件传送指令可以减少分支指令的使用，进而减少分支指令控制相关带来的开销，适合优化一类短分支. 短分支的特点是其分割产生的基本块只包含很少的几条指令.

GCC 编译器中条件执行优化阶段名称为 ce3，在机器描述中引入了语句 *cond_exec*，用于描述条件执行语义. 在 ce3 优化阶段，首先寻找短分支，然后尝试匹配所有的 *cond_exec* 描述模板，检查基本块内的指令是否满足模板的约束. 如果成功匹配，就将相应的 RTL 序列替换为条件执行的 RTL 指令.

在 ce3 阶段之前，边界检查访存的安全检查基于分支指令实现. 此类分支是典型的短分支，由其分割产生的基本块只会包含 1 条访存指令. 在 ce3 优化阶段首先发现这类短分支，然后根据 *cond_exec* 的描述，检查基本块中的指令是否为访存指令、条件是否是访存地址和边界地址构成的关系等.

2)带异常处理语义

如图 5(d)所示，LoongArch 边界检查访存指令在带异常处理场景下是条件分支指令和访存指令的复合，BCE 发生将会改变控制流. 由于带异常处理的边界检查访存指令比较复杂，本文选择在窥孔优化阶段实现优化，因为窥孔优化允许灵活地匹配指令序列中由多条指令表示的各种各样的复杂语义，能够有效地发现带异常处理的边界检查访存.

GCC 编译器的窥孔优化阶段名称为 peephole. 对于窥孔优化，GCC 在机器描述中引入了 define_peephole 语句，用来描述能够执行窥孔优化的 RTL 指令序列. 在 peephole 优化阶段，依次匹配当前位置之后的几条指令是否按照顺序满足 peephole 模板中描述的指令要求来确定是否可实施窥孔优化. 如果匹配成功，将相应的 RTL 序列替换为另一段更优化的 RTL 序列.

在 peephole 阶段之前，边界检查访存的安全检查基于分支指令，这条分支指令之后支配着 1 条访存指令. 因此，在 peephole 优化阶段，根据窥孔优化模板，首先尝试匹配分支指令，然后匹配第 2 条指令是否为访存指令，如果匹配，则进一步检查分支条件是否为访存地址与边界地址构成的边界检查条件.

值得注意的是，带异常处理的边界检查访存在形式上需要 4 个操作数，分别是访存数据、访存地址、边界地址、异常跳转目标. 这里的异常跳转目标是指边界检查条件不满足时的分支跳转目标，一般来说分支跳转目标采用 PC 相对偏移量来记录. 但是 LoongArch 定义的边界检查访存指令只有 3 个操作数字段，需要一种机制编码第 4 个操作数，存放跳转偏移量.

LoongArch 指令集架构有恒零寄存器，利用恒零寄存器可以构造出各种 nop 指令，并嵌入各种数据. 本文选择以 LoongArch 指令集的 pcaddi 指令来实现 nop 指令的构造，并做出相应约定，即在边界检查访存指令之后增加 pcaddi 指令，带异常处理的第 4 操作数(异常跳转目标)将被编码在该 pcaddi 指令之中. 这种嵌入异常跳转目标信息的方法，相比于额外增设 ELF(executable linking format)节或数据指令混排，不仅方便编译器生成指令，避免了对编译器以外的链接器等进行修改，而且可简化异常处理的实现.

图 7 中是带异常处理的第 4 操作数的嵌入方法，当边界检查条件不满足时，指令触发 BCE，后续将由内核、运行时库接管处理逻辑，即从 pcaddi 指令中取出第 4 操作数，计算出指定的异常跳转目标，并使程序跳转到目标地址继续执行. 当边界检查条件满足时，正常访存不发生异常，此时第 4 操作数无效. 由于嵌入在 nop 指令中，除了需要额外发射 1 条指令之外不对程序执行有任何影响.

```
ldle. d $a0，$a1，$a2
pcaddi $zero, offset # nop encoding the 4th operand
00000000000002e0 <test>:
2e0:   387b98a4    ldle. d      $a0，$a1，$a2
2e4:   18000040    pcaddi       +0$zero, 2 (0×2)
2e8:   4c000020    jirl         +1$zero, $ra, 0
2ec:   02ffc063    addi. d      +2$sp, $sp, −16 (0×ff0)
2f0:   1c000004    pcaddui      $a0, 0
2f4:   02c04084    addi. d      $a0, $a0, 16 (0×10)
2f8:   29c02061    st. d        $ra, $sp, 8 (0×8)
2fc:   57ffd7ff    b1           −44 (0×fffffd4)   #  2d0 <excep@plt>
```

Fig. 7 Embedding the fourth operand

图 7 第 4 操作数嵌入方法

图 7 的汇编代码展示了一个使用带异常处理边界检查访存的函数. 0x2e0 处使用 ldle.d 指令检查访存地址以及进行访存操作. 为了编码边界检查条件不满足时的分支跳转地址，0x2e4 处使用了 pcaddi 形式的 nop 指令，第 4 操作数编码在字段 *offset* 中为 2，在图 7 中用矩形框出，表示指向相对于 pcaddi 指令的 PC 加 2 处的指令. 根据 LoongArch 指令字 4 B 对

齐的特点, 可计算出实际所指向的地址是 0x2ec, 即图 7 中箭头指向的 +2 处的指令. 因此, 调用此函数后, 首先执行 0x2e0 处的边界检查访存指令, 若访存地址满足条件, 不触发 BCE 异常, 则继续顺序执行 0x2e4 处的 pcaddi 指令, 该指令此处作为一个 nop 指令无实际影响; 若访存地址不满足条件, 则触发 BCE 异常, 经过一系列异常处理后, 返回程序时将跳转到箭头指向的 0x2ec 处继续执行.

5 异常处理

5.1 内核处理 BCE 异常

LoongArch 架构下不满足访存边界要求时会触发 BCE, 其编号为 0xA. 然而, 目前 Linux 内核源码中没有对应的边界检查异常的处理函数, 将使用默认的处理函数 *do_reserved* 直接终止进程并返回 SIGUNUSED 信号(等同于 SIGSYS 信号), 如此无法实现在带异常处理场景下(图 5(b)所示)异常发生时用户程序分支跳转执行特定程序逻辑.

为此, 本文设计并实现了 BCE 的 Linux 内核处理机制, 主要包括 2 部分工作:

1)设计实现异常处理函数 *do_bce* 以解析 BCE 异常发生时的寄存器状态、异常 PC、指令字等信息;

2)设计实现 BCE 异常信号 SIGBCE 携带程序运行时库 glibc 进行信号处理所需的状态信息.

由于异常处理函数 *do_bce* 发送的信号需要携带寄存器状态、异常 PC、指令字等信息, 那么有必要设计实现一种信号, 与 Linux 通用的段错误信号 SIGSEGV 相区分. SIGSEGV 信号通常会导致进程被终止, 因为有关情况通常是程序编写存在错误导致的. 而当内核捕获到 BCE 异常后, 希望用户进程接收有关信息并跳转至特定的用户程序逻辑而非终止, 所以内核需要为边界检查访存指令触发的异常分配一个新的信号.

为了减少对已有 Linux 信号系统的影响, 本文通过扩展已有的自定义 SIGUSR1 信号来实现新信号的分配, 并将其命名为 SIGBCE 信号. 这样的设计并不会导致使用 SIGUSR1 信号的程序出现异常, 系统会根据接收到的信号信息结构体 *kernel_siginfo* 中的信号代码 *si_code* 来识别具体是哪一种信号. 类似这种共用信号的设计如 Linux 中 SIGSYS 和 SIGUNUSED 信号, 内核发送的 SIGSYS 信号可能携带值为 SYS_SECCOMP, SYS_USER_DISPATCH 的 *si_code* 信息, 而 SIGUNUSED 信号不会携带 *si_code* 信息, 从而区分.

Linux 中异常处理函数一般通过调用函数 *force_sig_fault* 来对出错进程发送信号, 而函数 *force_sig_fault* 不具有发送访存边界值的功能. 为此, 本文还实现了一种新的对进程发送指定信号的函数 *force_sig_bcebnderr*, 此函数利用 Linux 提供的信号信息结构体 *kernel_siginfo*, 将其成员变量 *si_code* 赋值为 SEGV_BNDERR. 同时, 扩展结构体 *kernel_siginfo* 新增成员变量 *lower* 和 *upper*, 分别用于存储访存时边界检查访存指令可能存在的下界与上界, 之后将信号发送给出错进程, glibc 中实现的信号处理函数会根据约定处理.

当用户程序中的边界检查访存指令触发边界检查异常时, 内核向该进程发送 SIGBCE 信号, 而用户程序的运行时库 gilbc 信号处理函数将捕获该信号, 若其携带的结构体 *kernel_siginfo* 中的成员变量 *si_code* 的值为 SEGV_BNDERR, 则认为此信号为 SIGBCE, glibc 将接管其处理. 否则按 SIGUSR1 信号进行处理, 保证了其他正常使用 SIGUSR1 信号的程序不会受到影响.

5.2 运行时库 glibc 信号处理函数

为了识别和处理内核发出的边界检查异常信号 SIGBCE, 本文在运行时库 glibc 中设计并实现了 SIGBCE 信号处理函数 *sigbce_sigaction*, 其功能主要是在用户进程捕获到 SIGBCE 信号之后, 解析 GCC 嵌入的 pcaddi 指令获得跳转的目标地址, 引导用户程序跳转到该目标地址去执行相关逻辑.

信号处理函数 *sigbce_sigaction* 的伪代码如算法 1 所示. 无论是无异常处理还是带异常处理的边界检查访存, 在条件不满足时都不进行访存操作且不修改数据寄存器的内容, 因此信号处理函数 *sigbce_sigaction* 首先需要将 PC 寄存器加 4 跳过边界检查访存指令. 然后, 信号处理函数获取紧随在边界检查访存指令之后的那条指令并进行解析, 如果是嵌入了信息的 pcaddi 形式的 nop 指令(如图 7 所示), 则计算其编码的相对跳转目标地址, 并将 PC 寄存器的值设为此地址. 信号处理函数结束后, 更新上下文, 之后回到用户程序从新的 PC 开始执行.

算法 1. *sigbce_sigaction*.

输入: 用户程序上下文 *ctx*;

输出: 更新的用户程序上下文 *ctx*.

① *ctx* → *pc* += 4;

② *inst* = *get_inst_word*(*ctx* → *pc*);

③ if (*is_inst_pcaddi*(*inst*)) then /*判断是否为 nop*/

④ if (*is_inst_rd_zero*(*inst*)) then /*计算 PC 相对寻址*/

⑤ *target* = *get_pcaddi_pcrel*(*ctx*→*pc*, *inst*);

⑥ *ctx* → *pc* = *target*;
⑦ end if
⑧ end if
/*结束信号处理，更新上下文*/
⑨ *signal_return*(*ctx*).

信号处理函数 *sigbce_sigaction* 为了获得带异常处理情况下指令的相对跳转目标地址，通过函数 *get_pcaddi_pcrel* 解析 GCC 编译嵌入的 pcaddi 指令. 如图 8 所示，嵌入的 pcaddi 指令最高 25~31 b 的操作码字段为 0001100; 5~24 b 为异常处理程序的相对地址，即立即数 si20; 最低 0~4 b 寄存器 *rd* 为 00000. 因此只需将该条指令字左移 7 b，再逻辑右移 12 b，即可得到相对跳转地址.

执行完信号处理函数后，将通过系统调用 *sigreturn* 返回到内核，从内核再返回到用户程序中断处继续运行. 系统调用 *sigreturn* 的程序上下文实际保存在当前信号处理函数的栈中，包含了系统调用 *sigreturn* 执行完成返回用户态的返回地址. 因此设置新的 PC 需要在当前栈中找到系统调用 *sigreturn* 使用的返回地址并将其修改为新的 PC.

为了使用户程序对边界检查访存指令所触发异常的处理无感知，SIGBCE 信号处理函数的实现与注册应当对用户透明. 为此，函数 *sigbce_sigaction* 注册使用标准的函数 *sigaction*，并添加到 glibc 对用户程序运行环境的初始化步骤当中. 此外，为了避免用户程序修改信号处理函数而导致错误，在 glibc 导出给用户程序的函数 *sigaction* 等中，检查用户程序是否试图修改处理 SIGBCE 信号的信号处理函数，如果是，则阻止相应的操作. 对于希望自己灵活处理 SIGBCE 信号的用户程序，可直接使用 Linux 内核系统调用而不使用 glibc 导出的函数 *sigaction*.

6 实验分析

本节对改进后的 GCC 编译器优化效果进行实验分析，首先检查编译器是否正确使用 LoongArch 边界检查访存指令，然后分析改进后的编译器编译用户程序是否获得了性能提升.

6.1 实验环境

本文采用 GCC-12.2.0 编译器作为基础，针对 LoongArch 边界检查访存指令进行改进. GCC-12.2.0 支持 LoongArch 架构基本指令集，已经并入主线并发布，但是不支持 LoongArch 边界检查访存指令. 本文采用的实验平台为龙芯 3C5000L 服务器，所修改 Linux 内核版本为 6.3.0. 为了分析实验结果，采用 Linux perf 工具读取性能监测单元（performance monitor unit, PMU）记录的细节信息.

6.2 正确性检查

本文通过设计内建函数和改进 RTL 优化器实现 LoongArch 边界检查访存指令的编译优化. 对于内建函数的正确性检查，只需使用改进后的 GCC 编译器编译涉及边界检查访存指令内建函数的代码并检查编译器是否正确识别内建函数即可. 图 9(a) 是使用本文设计的内建函数编写的 C 语言程序片段，其含义是读取数组元素和检查下标是否存在超出上界的情况. 图 9(b) 是改进后的 GCC 编译器编译该程序片段得到的汇编代码. 从图 9(b) 可以看出改进后的 GCC 编译器能够正确识别内建函数，使用 LoongArch 边界检查访存指令 ldle.d 实现访存地址小于等于边界地址的检查和访存，并通过 pcaddi 指令正确嵌入了跳转地址信息.

对于 RTL 优化器的改进，则需要检查改进后的编译器是否能正确识别出代码中隐含的无异常处理边界检查访存和带异常处理边界检查访存. 此外，对于带异常处理的情况，还需要检查所嵌入的信息是否正确. 图 10(a) 与图 10(b) 是改进后的 GCC 编译器分别编译图 5(a) 与图 5(b) 所示 2 种场景的 C 语言程序片段所得的汇编代码. 图 10(a) 表明改进后的编译器能够基于 LoongArch 边界检查访存指令编译优化无异常处理的边界检查访存程序；图 10(b) 表明改进后的编译器能够基于 LoongArch 边界检查访存指令编译优化带异常处理的边界检查访存程序，且通过 pcaddi 指令正确嵌入了跳转地址信息. 本文改进的 GCC 编译器成功识别了用户程序隐含的边界检查访



Fig. 8 Encoding format of the pcaddi instruction
图 8 pcaddi 指令编码格式

```
extern_attribute_((noreturn)) void print_error(const char* err, …);

unsigned long test_builtin(unsigned long* array, unsigned long n, unsigned long i){
  unsigned long res;
  if(_builtin_ldle_bndchk(&array[i], &array[n−1], &res)){
    print_error("index out of bound!", i);
  }
  else {
    return res;
  }
}
```

(a) 内建函数C语言片段

```
0000000000000340 <test_builtion>:
340:  002d10ac  alsl. d     $t0，$a1，$a0，0×3
344:  02ffe18c  addi. d     $t0，$t0，−8(0×ff8)
348:  002d10c4  alsl. d     $a0，$a2，$a0，0×3
34c:  387bb08d  ldle. d     $t1，$a0，$t0
350:  18000060  pcaddi      $zero，3(0×3)
354:  001501a4  move        $a0，$t1
358:  4c000020  jirl        $zero，$ra，0
35c:  02ffc063  addi. d     $sp，$sp，−16(0×ff0)
360:  001500c5  move        $a1，$a2
364:  1c000004  pcaddu12i   $a0，0
368:  02c0d084  addi. d     $a0，$a0，52(0×34)
36c:  29c02061  st. d       $ra，$sp，8(0×8)
370:  57ffc3ff  bl          −64(0×fffffc0)  # 330 <print_error@plt>
```

(b) 内建函数的编译结果

Fig. 9 GCC compilation result for built-in function based on LoongArch memory accessing instruction with bound-checking

图 9 基于 LoongArch 边界检查访存指令的内建函数 GCC 编译结果

```
00000000000002e0 <test>:
 2e0:   387b98a4   ldle. d    $a0，$a1，$a2
 2e4:   4c000020   jirl       $zero，$ra，0
```

(a) 无异常处理的汇编实现

```
00000000000002e0 <test>:
 2e0:   387b98a4   ldle. d    $a0，$a1，$a2
 2e4:   18000040   pcaddi     $zero，2(0×2)
 2e8:   4c000020   jirl       $zero，$ra，0
 2ec:   02ffc063   addi. d    $sp，$sp，−16
// …
// call excep()
```

(b) 带异常处理的汇编实现

Fig. 10 GCC compilation result of RTL optimization based on LoongArch memory accessing instruction with bound-checking

图 10 基于 LoongArch 边界检查访存指令的 RTL 优化的 GCC 编译结果

存语义，并基于边界检查访存指令正确实现了程序的编译.

6.3 性能优化测试

LoongArch 边界检查访存指令主要是针对需要检查访存地址的一类安全函数的优化而设计的，这类安全函数通过检查访存地址来阻止程序访问不应当访问的内存区域，例如阻止缓冲区溢出攻击. 为了检验改进后的编译器程序优化的效果，本文选取了3个典型的安全函数：

1）函数 *strncat*. glibc 的一种安全的 C 字符串操作函数，用于将 2 个字符串连接为 1 个字符串. 相比于函数 *strcat*，函数 *strncat* 指定了用于保存连接结果的字符串缓冲区的大小，在复制字符时检查指针是否超过缓冲区上界. 此安全函数可阻止缓冲区溢出攻击.

2）函数 *strncpy*. glibc 的一种安全的 C 字符串操作函数，用于将 1 个字符串复制到另一个缓冲区. 相比于函数 *strcpy*，函数 *strncpy* 指定了目标字符串缓冲区的大小，在复制字符时检查指针是否超过缓冲区上界. 此安全函数可阻止缓冲区溢出攻击.

3）函数 *vector_get*. 此安全函数安全地获取数组元素，检查所访问的下标是否在合法范围内.

针对以上 3 个函数，对比相应的前后实现版本，即直接使用改进前未优化的 GCC-12.2.0 编译生成的版本、使用改进后利用边界检查访存指令优化的 GCC-12.2.0 编译生成的版本. 由于单独的安全函数不能直接执行，本文为上述安全函数分别编写了相应的测试程序，每一个安全函数是测试程序的一部分. 如图 11 所示，每个测试程序按各自的程序逻辑调用安全函数实现其功能.

为了消除其他因素可能对实验结果带来的影响，需要控制其他代码不变、仅改变安全函数的编译实现，对比改进前后编译器对安全函数的优化效果. 通过采用动态链接技术，将安全函数单独在动态库中实现. 位于主可执行文件的用户程序逻辑调用位于动态库中的安全函数，执行安全检查并完成相应的功能. 通过替换动态库文件，选择在运行时使用或不

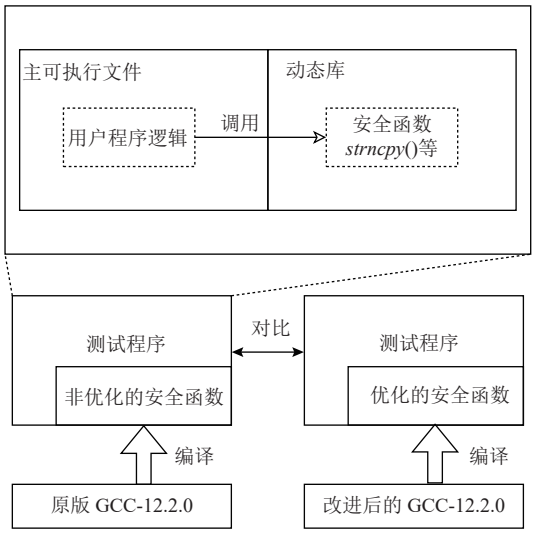


Fig. 11 Framework of testing programs

图 11 测试程序框架

使用优化版本的安全函数. 如此, 所对比运行的程序中, 其他程序逻辑的二进制代码实现完全一致, 只有安全函数因为编译器优化而有不同的实现.

6.4 性能优化分析

1)执行时间对比

表 2 是测试程序 *strncat*、测试程序 *strncpy* 以及测试程序 *vector_get* 执行多次的平均程序执行时间的比较. 每个测试程序包含使用未改进的 GCC 编译版本, 以及使用基于 LoongArch 边界检查访存指令改进后的 GCC 编译优化版本, 性能提升比例是指优化后的 GCC 版本的测试程序执行时间减少的比例.

Table 2 Comparison of Optimized Execution Time

表 2 执行时间优化对比

测试程序	程序执行时间/周期数	性能提升比例/%
strncat	14 528 340 679	
*strncat**	11 710 354 480	19.4
strncpy	7 555 284 838	
*strncpy**	6 265 294 025	17.1
vector_get	13 360 290 911	
*vector_get**	12 080 285 716	9.58

注：无“*”的测试程序表示使用未改进的 GCC 编译版本；有“*”表示使用基于 LoongArch 边界检查访存指令改进后的 GCC 编译优化版本；性能提升比例是指优化后的 GCC 编译版本的测试程序执行时间减少的比例.

从表 2 看出, 测试程序 *strncat* 和测试程序 *strncpy*, 程序执行时间分别从 1.45×10^{10} 周期数改进到 1.17×10^{10} 周期数, 以及从 7.56×10^{9} 周期数改进到 6.27×10^{9} 周期数, 改进效果非常显著, 性能提升比例均超过 15%, 最高接近 20%. 测试程序 *vector_get* 的程序执行时间从 1.37×10^{10} 周期数改进到 1.21×10^{10} 周期数, 性能提升比例也接近 10%.

通过表 2 可以得出, 3 个测试程序改进后的 GCC 编译器所编译版本的执行时间均有所减小, 性能得到了提升. 因此, 本文改进的 GCC 编译器能够有效地支持 LoongArch 边界检查访存指令, 并且可以显著提升安全函数的性能.

2)程序指令数对比

使用 LoongArch 边界检查访存指令直接在访存指令中实现安全检查, 省去了分支指令并减小了执行指令总数. 为了考察改进后的 GCC 编译器对于测试程序的指令数的影响, 本文进一步分析对比测试程序的执行指令数.

图 12 是使用 Linux perf 记录的测试程序 *strncat*, *strncpy*, *vector_get* 使用改进前后 GCC 编译器编译所得版本的执行指令数, 包括执行指令总数以及执行分支指令数, 图中的执行非分支指令数由执行指令总数减去执行分支指令数得到.

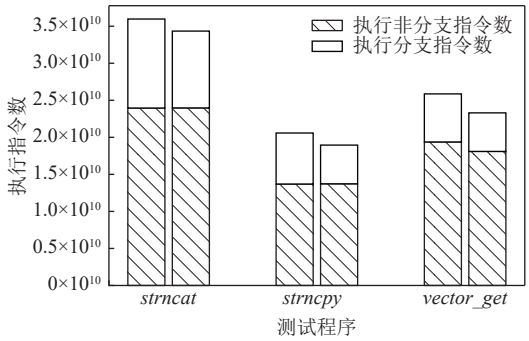


Fig. 12 Executed instructions counts of testing programs

图 12 测试程序的执行指令数

依据图 12 统计的指令数, 对于测试程序 *strncpy* 和 *strncat*, 执行指令总数的减少主要是由执行分支指令数减少贡献的. 可以推测, 由于使用 LoongArch 边界检查访存指令消除了循环体中安全检查的分支指令, 减小了总体分支不命中次数, 允许处理器更好地动态循环展开, 显著提高了性能.

此外, 从图 12 还可以看出, 在改进后的 GCC 编译器优化下, 测试程序 *vector_get* 不仅所执行的分支指令减少了, 而且所执行的其他指令也有所减少. 这说明改进后的 GCC 编译器发现了新的优化机会, 实现了更充分的优化. 可以推测, 在消除了安全检查的分支指令后, 不必要的基本块也被消除, 从而使部分被基本块隔断而不能进行的优化能够进行.

3)起始地址的影响

根据对程序指令数的分析, 使用 LoongArch 边界检查访存指令的编译优化会减少用户程序的分支指

令以及消除基本块, 使测试程序的静态指令数发生变化. 这使得优化前后安全函数的指令内存布局发生变化, 可能会影响指令缓存的利用, 进而影响程序执行的性能. 因此, 本文以测试程序 *strncpy* 为例, 进一步分析在改进前后 GCC 编译器优化下, 安全函数 *strncpy* 放置在不同的起始地址处对程序执行的影响.

图 13 是安全函数 *strncpy* 放置在不同的起始地址处、使用或不使用边界检查访存指令的 GCC 编译器优化的测试程序 *strncpy* 的执行时间, 仍以 CPU 时钟周期数为单位. 所记录的起始地址从某个随机选择的地址 *A* 开始, 按照 LoongArch 架构指令 4 B 对齐的要求, 之后每个地址依次加 4, 共记录 32 个地址. 图 13 中实线表示优化前的 GCC 编译器所编译测试程序的执行时间, 虚线表示使用边界检查访存指令优化的 GCC 编译器所编译测试程序的执行时间.

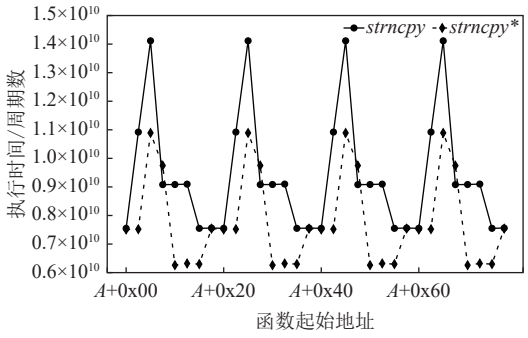


Fig. 13 Execution time of *strncpy* and *strncpy** with different start addresses

图 13 *strncpy* 和 *strncpy**在不同起始地址的执行时间

从图 13 可以看出, 测试程序执行时间在不同的安全函数 *strncpy* 放置的起始地址呈现出周期性变化, 每 8 个地址为 1 个周期, 虚线整体在实线之下. 由于实验采用动态链接技术, 测试程序 *strncpy* 和测试程序 *strncpy**中只有安全函数 *strncpy* 因为编译器优化而有不同的实现, 其他程序逻辑的二进制代码实现完全一致. 故由图 13 可得, 即使在指令内存布局发生变化的情况下, 使用 LoongArch 边界检查访存指令编译优化后得到的安全函数 *strncpy* 相比于优化前, 总体上也可认为具有更好的性能.

4)实验结果分析

综合以上实验结果可以看出, 改进后的 GCC 编译器不仅能够成功识别出用户程序代码中隐含的无异常处理边界检查访存语义和带异常处理边界检查访存语义, 并且在这 2 种异常处理的语义场景下都正确使用了 LoongArch 边界检查访存指令实现相应的编译优化.

LoongArch 边界检查访存指令可以消除安全检查的分支指令, 减少安全函数的静态指令数, 进而减少程序动态执行指令数. 从实验结果可以看出, 改进后的 GCC 编译器有效利用了 LoongArch 边界检查访存指令, 通过硬件实现访存的安全检查操作, 显著优化了安全函数 *strncpy*, *strncat*, *vector_get*, 且不会影响指令缓存的利用和程序执行的性能.

理论上来说, 使用本文改进的 GCC 编译器编译程序不会导致性能下降, 因为在不触发边界检查异常的一般情况下边界检查访存指令没有额外的性能开销且不会增加程序执行的指令数.

7 结 语

本文基于 LoongArch 指令集, 针对其特有的边界检查访存指令改进了 GCC 编译器、Linux 内核以及 glibc 运行库, 所做优化设计特点总结如下: 1)对 LoongArch 边界检查访存指令的各种配置组合(读写操作、整数浮点、数据大小、上界下界)均有效可行; 2)区分了无异常处理与带异常处理 2 种边界检查访存指令应用场景, 确定边界检查异常的处理, 便于语义识别; 3)设计实现了边界检查访存内建函数, 提示程序优化; 4)改进了 RTL 优化器, 在 RTL 优化阶段识别边界检查访存语义, 实现自动编译优化; 5)改进了边界检查异常处理机制和运行环境, 确保程序实际执行语义与编译优化语义相一致, 并使用户对编译优化改进无感知; 6)对一些典型的安全函数, 减小了安全检查带来的开销, 性能改善可接近 20%. 总的来说, 本文工作完善 LoongArch 生态, 有助于推进 LoongArch 指令集发展, 所提出的特定指令编译优化方法对于此类工作有一定的参考价值.

作者贡献声明: 舒燕君提出了论文的主要思路和行文框架; 郑翔宇调研相关内容、设计实验方案并进行了编译优化方面的实验开发; 徐成华和黄沛指导了编译器和异常处理协同的总体方案的设计; 王永琪和周凡参与设计实验方案并进行了异常处理方面的实验开发; 张展对实验方案和实验结果分析提出了指导意见; 左德承总体把控方向并提出了指导意见. 所有作者均参与了论文的撰写、讨论和修改。

参 考 文 献

[1] Jiang Weihua, Li Weihua, Du Jun. Buffer overflow attack: Theory,

recovery and detection[J]. Computer Engineering, 2003, 29(10): 5−7 (in Chinese)
(蒋卫华, 李伟华, 杜君. 缓冲区溢出攻击: 原理, 防御及检测[J]. 计算机工程, 2003, 29(10): 5−7)

[2] Li Yawei, Zhang Longbing, Zhang Fuxin, et al. A security protection mechanism on program runtime based on software and hardware cooperation[J]. Chinese Journal of Computers, 2023, 46(1): 180−201 (in Chinese)
(李亚伟, 章隆兵, 张福新, 等. 基于软硬协同的程序运行时安全保护机制[J]. 计算机学报, 2023, 46(1): 180−201)

[3] Otterstad C W. A brief evaluation of Intel®MPX[C]//Proc of the 9th Annual IEEE Systems Conf. Piscataway, NJ: IEEE, 2015: 1−7

[4] Serebryany K. ARM memory tagging extension and how it improves C/C++ memory safety[J]. Login The USENIX Magazine, 2019, 44(2): 12−16

[5] Hu Weiwu, Wang Wenxiang, Wu Ruiyang, et al. Loongson instruction set architecture technology[J]. Journal of Computer Research and Development, 2023, 60(1): 2−16 (in Chinese)
(胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术[J]. 计算机研究与发展, 2023, 60(1): 2−16)

[6] Patterson D A, Ditzel D R. The case for the reduced instruction set computer[J]. SIGARCH Computer Architecture News, 1980, 8(6): 25−33

[7] GCC Team. GCC wiki [EB/OL]. [2024-02-05]. https://gcc.gnu.org/wiki/Intel MPX support in the GCC compiler

[8] Clang Team. Clang documentation [EB/OL]. [2024-02-05]. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html

[9] Loongson Technology Corporation. LoongArch architecture manual [EB/OL]. [2024-02-05]. https://www.loongson.cn/system/loongarch

[10] Diego N. Tree SSA — A new high-level optimization framework for the GNU compiler collection[C/OL]//Proc of the 5th NordU/USENIX Users Conf. Berkeley, CA: USENIX Association, 2003[2024-06-15]. https://www.airs.com/dnovillo/Papers/nordu2003.pdf

[11] Wang Wenyi, Wu Huabei. Analysis and application of the signal communication mechanism of Linux[J]. Computer Engineering and Applications, 2005, 41(3): 108−110, 115 (in Chinese)
(王文义, 武华北. Linux 中进程间信号通信机制的分析及其应用[J]. 计算机工程与应用, 2005, 41(3): 108−110, 115)

[12] Oleksenko O, Kuvaiskii D, Bhatotia P, et al. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2018, 2(2): 28: 1−28: 30

[13] Rigger M, Marr S, Kell S, et al. An analysis of x86−64 inline assembly in C programs[C]//Proc of the 14th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2018: 84−99

[14] Zhao Xiang, Jia Haipeng, Zhang Yunquan, et al. Real FFT implementation and performance optimization based on ARMv8 CPUs[J]. Chinese Journal of Computers, 2023, 46(5): 1003−1018 (in Chinese)
(赵翔, 贾海鹏, 张云泉, 等. 基于 ARMv8 处理器的实数 FFT 实现与性能优化研究[J]. 计算机学报, 2023, 46(5): 1003−1018)

[15] Cai Lulu, Wang Yagang, Chen Xiaolong. Glibc hot spot function assembly optimization for LoongArch[C]//Proc of the 41st Chinese Control Conf. Piscataway, NJ: IEEE, 2022: 2053−2058

[16] Yang Hao, Liu Zhe, Huang Junhao, et al. High-speed AVX2 implementation of AKCN-MLWE[J]. Chinese Journal of Computers, 2021, 44(12): 2560−2572 (in Chinese)
(杨昊, 刘哲, 黄军浩, 等. AKCN-MLWE 算法 AVX2 高效实现[J]. 计算机学报, 2021, 44(12): 2560−2572)

[17] Zhao Long, Han Wenbao, Yang Hongzhi. Research on ECC attacking algorithm based on SIMD instructions[J]. Journal of Computer Research and Development, 2012, 49(7): 1553−1559 (in Chinese)
(赵龙, 韩文报, 杨宏志. 基于 SIMD 指令的 ECC 攻击算法研究[J]. 计算机研究与发展, 2012, 49(7): 1553−1559)

[18] Shen Jie, Long Biao, Jiang Hao, et al. Implementation and optimization of vector trigonometric functions on Phytium processors[J]. Journal of Computer Research and Development, 2020, 57(12): 2610−2620 (in Chinese)
(沈洁, 龙标, 姜浩, 等. 飞腾处理器上向量三角函数的设计实现与优化[J]. 计算机研究与发展, 2020, 57(12): 2610−2620)

[19] Rigger M, Marr S, Adams B, et al. Understanding GCC builtins to develop better tools[C]//Proc of the ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering, 18th ESEC/27th SIGSOFT FSE 2019. New York: ACM, 2019: 74−85

[20] Chen Xiaolong, Wang Yagang, Cai Lulu. GCC built-in function mechanism analysis and LoongArch-based implementation[C]//Proc of the 41st Chinese Control Conf. Piscataway, NJ: IEEE, 2022: 2046−2052

[21] Koppelmann B, Adelt P, Mueller W, et al. RISC-V extensions for bit manipulation instructions[C]//Proc of the 29th Int Symp on Power and Timing Modeling, Optimization and Simulation. Piscataway, NJ: IEEE, 2019: 41−48

[22] Babu P S, Sivaraman S, Sarma D N, et al. Evaluation of bit manipulation instructions in optimization of size and speed in RISC-V[C]//Proc of the 34th Int Conf on VLSI Design and 20th Int Conf on Embedded Systems. Piscataway, NJ: IEEE, 2021: 54−59

[23] Levy M, Olson R. Autovectorization for GCC compiler[J]. Electrical Design News: The Magazine of the Electronics Industry, 2007, 52(15): 69−70, 72, 74

[24] Jiang Weihua, Mei Chao, Guo Yi, et al. Vectorization for real-life multimedia applications on processors' multimedia extensions[J]. Chinese Journal of Computers, 2005, 28(8): 1255−1266 (in Chinese)
(姜伟华, 梅超, 郭一, 等. 一种针对多媒体扩展指令集和实际多媒体程序的自动向量化方法[J]. 计算机学报, 2005, 28(8): 1255−1266)

[25] Feng Jingge, He Yeping, Tao Qiuming, et al. SLP vectorization method based on multiple isomorphic transformations[J]. Journal of Computer Research and Development, 2023, 60(12): 2907−2927 (in Chinese)
(冯竞舸, 贺也平, 陶秋铭, 等. 基于多种同构化变换的 SLP 向量化

方法[J]. 计算机研究与发展, 2023, 60(12): 2907−2927)

[26] Tian Zuwei, Sun Guang. Research of compiler optimization technology based on predicated code[J]. Computer Science, 2010, 37(5): 130−133,138 (in Chinese)
(田祖伟, 孙光. 基于谓词代码的编译优化技术研究[J]. 计算机科学, 2010, 37(5): 130−133, 138)

[27] Wang Fengqin, Hu Dinglei, Liu Chunlin. A register allocation algorithm for predicated code[J]. Journal of Computer Research and Development, 2006, 43(8): 1471−1476 (in Chinese)
(王凤芹, 胡定磊, 刘春林. 一种基于谓词执行优化技术的寄存器分配算法[J]. 计算机研究与发展, 2006, 43(8): 1471−1476)



Shu Yanjun, born in 1981. PhD, associate professor. Member of CCF. Her main research interests include computer architecture, fault-tolerant computing, and high-performance computing.

舒燕君, 1981 年生. 博士, 副教授. CCF 会员. 主要研究方向为计算机体系结构、容错计算、高性能计算.



Zheng Xiangyu, born in 2001. Master candidate. His main research interests include computer architecture, and complier design and optimization.

郑翔宇, 2001 年生. 硕士研究生. 主要研究方向为计算机体系结构、编译器设计与优化.



Xu Chenghua, born in 1989. PhD candidate. His main research interests include performance analysis of processor, and complier design and optimization.

徐成华, 1989 年生. 博士研究生. 主要研究方向为处理器性能分析、编译器设计与优化.



Huang Pei, born in 1983. Engineer. His main research interests include computer architecture and operating system.

黄 沛, 1983 年生. 工程师. 主要研究方向为计算机体系结构、操作系统.



Wang Yongqi, born in 2001. Master candidate. His main research interests include performance analysis, operating system, and virtualization.

王永琪, 2001 年生. 硕士研究生. 主要研究方向为性能分析、操作系统、虚拟化.



Zhou Fan, born in 2001. Undergraduate. His main research interests include computer architecture, operating system, and virtualization.

周 凡, 2001 年生. 本科生. 主要研究方向为计算机体系结构、操作系统、虚拟化.



Zhang Zhan, born in 1978. PhD, professor. Senior member of CCF. His main research interests include computer architecture, edge computing, and mobile computing.

张 展, 1978 年生. 博士, 教授. CCF 高级会员. 主要研究方向为计算机体系结构、边缘计算、移动计算.



Zuo Decheng, born in 1972. PhD, professor. Senior member of CCF. His main research interests include computer architecture, fault-tolerant computing, and high-performance computing.

左德承, 1972 年生. 博士, 教授. CCF 高级会员. 主要研究方向为计算机体系结构、容错计算、高性能计算.